Graz University of Technology

Doctor of Philosophy Dissertation

Using Machine Learning for Automatic Software Fault Prediction and Multi-label Classification of Software Change Requests

Syed Nadeem Ahsan

Institute For Software Technology
Faculty of Computer Science
Graz University of Technology, Austria

First Examiner:
Univ.-Prof. Dr. Franz Wotawa
Graz University of Technology, Austria

Second Examiner:
Univ.-Prof. Dr. A Min Tjoa
Vienna University of Technology, Austria

To my father Syed Nizam Uddin and my late mother Khurshida Begum

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommene Stellen als solche kenntlich gemacht habe.

Graz, am ....

(Unterschrift)

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, on ....

(Signature)

Foreword

This dissertation was written as a partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science at the Technical University, Graz, Austria. The research work presented in this thesis was carried out at the Institute for Software Technology, Inffeldgasse 16b/II, 8010 Graz, Austria. This work was started in January 2007 and involved both research and development.

The subject of this thesis is the automation of software maintenance tasks such as fault prediction, impact analysis, and bug triaging. In order to achieve the goals different methods were used to extract data from software repositories, and used machine learning techniques to build prediction and triaging models. A significant part of this dissertation has been published at various international conferences and journal of high repute.

This work was suggested and supervised by Prof. Franz Wotawa and was partially funded by Higher Education Commission, Government of Pakistan and SoftNet Austria.

Abstract

During the evolution of a software system, a huge amount of data is generated. These data are maintained in software repositories. This thesis aims to utilize the available software's repositories to provide novel techniques that facilitate the automation of various software maintenance tasks. This thesis presents two contributions: (1) Models for effort estimation, risk estimation, and fault prediction. (2) Multi-label classification of SCRs for impact analysis and bug triaging.

We propose an effort estimation model for open source software systems (OSS), which can be used to estimate how much effort is required to fix a fault in source codes. Since, in case of OSS, effort data related to the fixing of faults is unavailable, therefore, to build an effort estimation model, we introduce a novel approach to mine the effort data from a software repository. Once we obtain the effort data, we use it in our model as a dependent variable. For model's independent variables, we use a set of code and process metrics. Finally, we use the metrics and effort data to build an effort estimation model using parametric and non-parametric regression techniques. To obtain an effort estimation model we performed an experiment by using the data of Mozilla project. In our experiment, the maximum obtained values of model's correlation and MMRE (relative error) are 0.51 and 63.8% respectively, when support vector machine used to build the effort model. We also propose a model for risk estimation, the purpose of a risk model is to estimate a risk of faults associated with new changes in a source file. To build a risk model, we use an approach to transform the previous history of software changes into four types, i.e., bug-fix, bug fix-introducing, buggy changes and clean changes. Each source file has different values related to these changes. We use this information to build a risk model.

In order to obtain a high performance fault prediction model, we performed two different experiments using the project evolution data of Firefox, Apache, and GNOME. The purpose of the first experiment was to find the best statistical or machine learning techniques for the construction of fault prediction models. The purpose of the second experiment was to improve the fault prediction capability of a model by using a set of predictors (metrics), which are highly correlated with the number of faults. In the first experiment, we used two different approaches, i.e., regression and classification to build the fault prediction models. We found multiple linear regression and neural network as the best regression based fault prediction models. Whereas, bagging, boosting and decision tree (J48) are found as the best classification based fault prediction models. In the second experiment, we introduced eight new metrics, which are related to the logical couplings among the source files. We found that these metrics are highly correlated with the number of faults, and therefore, can be used to build a fault prediction model. In case of regression the obtained value of R^2 of the model (when logical coupling metrics used as predictors) is 0.8. Similarly, in case of classification, the obtained value of accuracy is 97%.

In the second part of this thesis, we propose two different approaches to obtain a classification based prediction model for the impact analysis of SCRs: i) We obtain different groups of source files which

change together. To obtain the group of source files, we use the technique of frequent pattern mining and association rules. Once we obtain the group of source files, then each resolved SCR is labeled with one of the appropriate groups. Afterwards we use labeled data of SCRs to build three different models for the classification of SCRs. These models are based on support vector machine, decision tree and naive bayes. ii) We directly label each resolved SCR with one or more impacted source files, and use this data to build models, which are based on multi-label classifiers, i.e., Label Power set method. In order to evaluate both the approaches, we performed experiments by using the data from three OSS projects, i.e., Mozilla, GNOME and Eclipse. Our experimental results are promising, the obtained maximum precision values for single and multi label classifications are 58.2% and 47.1% respectively. Furthermore, in case of single and multi-label classification, the maximum attained precision values for any individual label are 86.5% and 92% respectively. Finally, to obtain a *Bug Triage System* (BTS), we propose two different classification models, i.e., single and multi label classification of bug reports (SCRs). In order to build a single-label classification model for BTS, we label each resolved bug report with the name of developers who actually resolved that bug report. After labeling, we use labeled data to build single-label classifiers for BTS. In case of multi-label classification model for BTS, we label each resolved bug report not only with the name of the developer but also labeled with names of impacted source files and an estimate for the time required to resolve a given bug report. After labeling we use the multi-labeled data to build multi-label classifiers for a BTS. In order to evaluate our approaches, we performed experiment using project evolution data of Mozilla. In case of single label classification, the best obtained BTS is based on support vector machine having 44.4% classification accuracy with 30% precision and 28% recall. Whereas, in case of multi-label, we performed experiments by using Problem Transformation and Algorithm Adaptation methods of multi-label machine learning. In our experiment, we have obtained precision levels up to 71.3% with 40.1% recall.

Kurzfassung

Während der Entwicklung eines Software-Systems wird eine riesige Menge an Daten generiert. Diese Daten werden in Software-Repositories gespeichert. Das Ziel der vorliegenden Dissertationsschrift ist die verfügbaren Software-Repositories zu nutzen, um neuartige Techniken zu entwickeln, die die Automatisierung der verschiedenen Software-Wartungsarbeiten erleichtern. Diese Arbeit behandelt zwei Beiträge: (1) Modelle für Aufwandsschätzung, Risikoabschätzung und Fehlervorhersage. (2) Multi-Label-Klassifikation der SCR für Impact-Analyse und Fehlersichtung.

Wir präsentieren ein Modell zur Aufwandsschätzung für Open-Source-Software-Systeme (OSS), welches dazu dient abzuschätzen wie viel Aufwand erforderlich ist, um einen Fehler in Quellcode zu beheben. Da im Fall von OSS der Aufwand in Bezug auf die Festsetzung von Fehlern nicht verfügbar ist, stellen wir einen neuartigen Ansatz zur Abschätzung des Aufwands aus einem Software-Repository vor. Nach Berechnung des Aufwands, verwenden wir diesen in unserem Modell als abhängige Variable. Für das Modell der unabhängigen Variablen verwenden wir eine Reihe von Code und Prozesskennzahlen. Schließlich verwenden wir die Metrik und Aufwandsdaten inAufwandsschätzungsmodellen mit parametrischen und nicht parametrischen Regressionsverfahren. Um die erhaltenen Aufwandsschätzungsmodell zu testen, machen wir ein Experiment mit Hilfe der Daten des Mozilla-Projekts. In unserem Experiment erhalten die Maximalwerte des Modells Korrelation und MMRE (relative Fehler) 0,51% bzw. 63,8% bei Support-Vektor-Maschine um das Aufwandsmodell zu bauen. Wir präsentieren auch ein Modell für die Abschätzung des Risikos, die mit einem Risiko von Fehlern mit neuen Änderungen in einer Quelldatei verbunden sind. Um ein Risikomodell abzuschätzen, benutzen wir eine Annäherung an die bisherige Historie der Software-Änderungen durch vier Änderungstypen: Bug-Fix, Bug-Fix-Einführung, fehlerhafte Änderungen und saubere Änderungen. Jede Quelldatei hat andere Werte in Bezug auf diese Änderungen. Wir verwenden diese Informationen, um ein Risiko-Modell zu bauen.

Um ein hoch leistungsfähiges Fehler-Vorhersagemodell zu erhalten, führten wir zwei verschiedene Experimente mit den Projektfortschrittsdaten von Firefox, Apache und GNOME durch. Der Zweck des ersten Experiments war es, die besten Techniken der Statistik oder des maschinellen Lernens für den Bau von Fehlervorhersagemodellen zu finden. Der Zweck des zweiten Experiments war es, die Fehlervorhersagefähigkeit eines Modells mithilfe einer Reihe von Vorhersagefunktionen (Metriken) zu verbessern, die stark mit der Anzahl den Fehlern korrelieren. Im ersten Experiment verwendeten wir die zwei Ansätze Regression und Klassifikation, um das Fehler-Vorhersage-Modell zu bauen. Wir fanden multiple lineare Regressionen und neuronale Netze als beste Regression Fehlervorhersagemodelle. Im zweiten Experiment verwendeten wir acht neue Metriken, die logische Verbindungen zwischen den Quelldateien befreffen. Wir fanden heraus, dass diese Metriken hoch korrelierend mit der Anzahl der Fehler sind und deshalb verwendet werden hönnen, um ein Störungsprognosemodell zu bauen. Im Falle der Regression ist der erhaltene Wert von R2 des Modells (wenn logische Kopplung Metriken als Prädiktoren verwendet werden) 0,8. Im Fall der Klasssifikation ist der erhaltene Genauigkeitswert 97%.

Im zweiten Teil dieser Arbeit zeigen wir zwei verschiedene Ansätze, um ein Klassifizierungs-Vorhersage-Modell für die Analyse der Auswirkungen SCRs zu erhalten: i) erhalten wir verschiedene Gruppen von Quelldateien, die gemeinsam zu ändern sind. Um Quelldateien zu gruppieren, verwenden wir die Technik der Mustererkennung und Assoziationsregeln. Sobald wir die Gruppe von Quelldateien erhalten, wird jeder gelöste SCR mit einer der entsprechenden Gruppen gekennzeichnet. Danach verwenden wir von SCR gekennzeichnete Daten um drei verschiedene Modelle für die Klassifizierung von SCRs zu erstellen. Diese Modelle basieren auf Support-Vektor-Maschine-, Entscheidungs-Baum- und Naive Bayes-basierten Modellen. ii) Wir ordnen jeder aufgelösten SCR ein oder mehrere beeinflusste Quelldateien zu, und nutzen diese Daten, um Modelle zu bauen, die auf Multi-Label-Klassifikatoren basieren, unter anderem die Label Power-Methode. Um beide Ansätze zu evaluieren, führten wir Experimente mit Hilfe der Daten aus den drei OSS-Projekten Mozilla, GNOME und Eclipse durch. Unsere experimentellen Ergebnisse sind vielversprechend. Die erhaltenen Werte für maximale Präzision Single- und Multi-Label Klassifikationen sind 58,2% und 47,1%. Single-und Multi-Label-Klassifikationen, erlauben ein Höchstmaß an Präzision bis zu 86,5% und 92% für jedes einzelne Etikett. Schließlich, um ein Bug Triage-System (BTS) erhalten, schlagen wir zwei unterschiedliche Modelleinstufungen, d.h. Einzel-und Multi-Label-Klassifikation von Bug-Reports (SCR). Um den Bau eines einzelnen Klassifikationsmodells für BTS zu ermöglichen, markieren wir jeden aufgelösten Fehlerbericht mit den Namen der Entwickler, die den tatsächlichen gemeldeten Fehler behoben haben. Nach der Markierung verwenden wir die markierten Daten zur Klassifikationen, um ein BTS zu bauen. Im Falle von Multi-Label-Klassifikation Modell für BTS kennzeichnen wir jeden geschlossenen Fehlerbericht nicht nur mit den Namen der Entwickler sondern auch mit den Namen von betroffenen Quelldateien und einer Schätzung für die erforderliche Zeit um einen bestimmten gemeldeten Fehler zu beheben. Nach der Markierung verwenden wir die Multi-markierten Daten als Multi-Label-Klassifikatoren um eine BTS aufzubauen. Um unsere Ansätze zu bewerten, führten wir Experimente mit Projektfortschrittsdaten von Mozilla durch. Im Fall, einer Single-lable Classification, basiert die beste erhaltene BTS auf Support-Vektor-Maschine mit 44,4% Klassifikationsgenauigkeit mit 30% Precision und 28% Recall. Im Falle der Multi-Label-Klassifikation hingegen führten wir Experimente mit Hilfe von Problemtransformationen und Methoden zur Anpassung von Algorithmen des Multi-Label machine-learning durch. In unserem Experiment erhielten wir Genauigkeiten bis auf 71,3% bei 40,1% Recall.

Acknowledgments

Many people accompanied me during the endeavor of my doctoral studies and of writing my thesis. My thesis would have not been possible without their kind support. I am grateful to all of them.

The first acknowledgment is for all the co-workers, staff members and technicians at the Institute for Software Technology (IST), who ensured in providing a relaxed and very friendly working environment. Many thanks are due to my colleagues who have helped me by discussions and guidance.

My advisor for the PhD work, Prof. Dr. Franz Wotawa has always provided me the necessary guidance and help. His encouragement in different aspects of the work contributed to my personal and professional development. I am thankful to Prof. Dr. A Min Tjoa for taking time out of his busy schedule to act as a second reviewer of my thesis.

I am thankful to my colleague Mr. Javed Ferzund for his valuable comments and suggestions on my work. We worked together and shared our knowledge to solve the research problems. We spent a very good time with each other under the supervision of Prof. Dr. Franz Wotawa. I am also thankful to my office colleague/friends Mrs. Zamurrud Baig, Mrs. Farah Naz Musharaf, Mrs. Tayyaba Naem, Mr. Imran Siddique, Mr. M. Salman Khan, Mr. M. Shafiq Siraj, Mr. Safdar Zaman and Mr. M. Tanvir Afzal for their kind supports during my research work. Last, but not least, the acknowledgment is due to course teachers Prof. Dr. Dines Bjørner's, Dr. Hausler Stefan and Dr. Bernhard Peischl, who have helped me in numerous ways during this work, and provided me all required resources to accomplish the task.

I would also like to acknowledge the financial support of the Higher Education Commission of Pakistan (HEC), KNPC and PAEC, which made this work possible. The Softnet Austria played a vital role in the dissemination of the research results.

Thanks to my father and siblings, my wife Saima Beenish, my daughter Syeda Wania Ahsan and my son S.M. Daniyal Ahsan who have remained the binding force in my life all through this work. Thanks to my Lord, Allah who gifted me with this life and all its surprises.

Syed Nadeem Ahsan
Graz, Austria, December 2010

Contents

1. Introduction **1**
1.1. Mining Software Repositories . . . 1
1.2. Thesis Overview . . . 2
1.2.1. Motivation . . . 2
1.2.2. Problem Statement . . . 3
1.2.3. Research Hypothesis . . . 4
1.2.4. Goals/Objectives . . . 4
1.2.5. Thesis Contribution . . . 5
1.3. Thesis Layout . . . 6

2. Machine Learning Techniques **7**
2.1. Application of Machine Learning In Software Engineering . . . 7
2.2. Supervised and Unsupervised Machine Learning Techniques . . . 8
2.2.1. Artificial Neural Network (ANN) . . . 9
2.2.2. RBF Network . . . 12
2.2.3. Support Vector Machine (SVM) . . . 13
2.2.4. Decision Trees . . . 14
2.2.5. Naive Bayes . . . 15
2.2.6. Boosting and Bagging Algorithm . . . 15
2.3. Statistical Regression Analysis . . . 15
2.3.1. Multiple Linear Regression . . . 16
2.3.2. Logistic Regression . . . 17
2.4. Model's Evaluation Criteria . . . 18
2.4.1. Evaluation Criteria for Regression Model . . . 18
2.4.2. Evaluation Criteria for Classification Model . . . 19

3. Mining Bug Fix Effort Data to Construct Effort Estimation Models for OSS **23**
3.1. Our Approach . . . 24
3.1.1. Obtaining Data From Software Repositories . . . 25
3.1.2. Obtaining Metrics Data . . . 26
3.1.3. Role of Developers and Contributors in OSS . . . 26
3.1.4. Mining Bug Fix Effort Data . . . 27
3.2. Application of Developer's Bug Fix Activity Data in OSS . . . 32
3.2.1. Developer Activity Log Book . . . 32
3.2.2. Developer's Contributions Measures to Identify Expertise . . . 35
3.2.3. Visualization of Developer Bug Fix Activity . . . 36
3.3. Effort Estimation Model . . . 38
3.3.1. Regression Based Machine Learning Algorithm . . . 39
3.3.2. Multiple Linear Regression Model . . . 40
3.3.3. Machine Learning Models . . . 40
3.4. Threats To Validity . . . 41
3.5. Summary . . . 42

4. Prediction of Risky Program Files 43
4.1. Risk of Faults Associated with the New Changes in a Program Source File 45
4.1.1. Data Collection 45
4.1.2. Types of Program File Changes 47
4.1.3. Model Building 50
4.2. Experimental Setup 54
4.2.1. Results and Discussion 54
4.2.2. Threats To Validity 55
4.3. Application of Hunk Data as Hunk Metrics 57
4.4. Summary 57

5. Software Fault Prediction Models 59
5.1. Approach to Build Fault Prediction Models 62
5.1.1. Collecting Data 63
5.1.2. Computing Code Metrics 63
5.2. Fault Prediction Models Using Regression Techniques 64
5.2.1. Metrics Correlation 64
5.2.2. Metrics Scatter Plot 65
5.2.3. Principle Component Analysis (PCA) 65
5.2.4. Fault Prediction Models Using Multiple Linear Regression 67
5.2.5. Fault Predictor Model Using Machine Learning Algorithms 71
5.2.6. Comparison of Regression based Fault Prediction Models 75
5.3. Fault Prediction Models Using Classification Techniques 78
5.3.1. Point Biserial Correlation 78
5.3.2. Experimental Setup to Build The Models 80
5.3.3. Classification Results and Discussion 81
5.4. Case Study: Analyzing Bug Prediction Capabilities of Static Code Metrics 92
5.4.1. Study Approach 92
5.4.2. Results and Discussion 98
5.5. Summary 100

6. Fault Prediction Capability of Program Files Logical Coupling Metrics 103
6.1. Our Approach 105
6.1.1. Data Extraction and Filtering 105
6.1.2. Program Files Coupling Patterns and Coupling Measures 106
6.2. Experimental Setup and Results 110
6.2.1. Analysis of Metrics Correlation With Number of Bugs 110
6.2.2. Bug Prediction and Classification Using Logical-Coupling Metrics 114
6.3. Threats to Validity 116
6.4. Summary 117

7. Impact Analysis of Software Change Request Using Single and Multi-label Classification 121
7.1. Our Approach 124
7.2. Data Extraction and Filtering 125

7.3. Indexing and Dimension Reduction 126
7.3.1. Latent Semantic Indexing 127
7.4. Labeling and Classification 127
7.4.1. Single-Label Classification 128
7.4.2. Bug-Report's Mapping with Source Files' Group-Id 129
7.4.3. Multi-Label Classification 131
7.5. Classification Evaluation Measures 132
7.5.1. Evaluation Measures for Single Label Classification 132
7.5.2. Evaluation Measures for Multi Label Classification 132
7.5.3. Labels Statistics 132
7.6. Experimental Setup 133
7.6.1. Results And Discussion 134
7.6.2. Threat to Validity 136
7.7. Summary 138

8. Automatic Bug Triaging System Based on Multi-Label Classification of SCRs 141
8.1. Different Approaches For an Automatic Bug Triage System 143
8.2. Bug Triage System Based on Single Label Classification of SCR 143
8.2.1. Data Preparation 144
8.2.2. Experimental Setup and Results 145
8.3. Bug Triage System Based on the Multi-label Machine Learning Classification 149
8.3.1. Data Collection 151
8.3.2. Experimental Setup and Results 153
8.4. Threats to Validity 160
8.5. Summary 161

9. Related Work 163
9.1. Extracting Data From Software Repositories 163
9.2. Bug Fix Effort Estimation Model for OSS 164
9.3. Analysis of Source Code Changes to Identify Risky Source Files 165
9.4. Fault Prediction Model 166
9.5. Machine Learning Classification of Software Change Request 168

10. Conclusion and Future Work 171
10.1. Conclusion 171
10.1.1. Estimation and Prediction Models 171
10.1.2. Multi-Label Classification of SCRs 174
10.2. Future Work 175

A. Appendix 177
A.1. List of Publications 177

B. Appendix 179
B.1. Terminology 179

Bibliography 181

List of Figures

2.1. Artificial neural network . . . 9
2.2. Example of how neural networks trained by adjusting output for some specific input. leads to a specific target output . . . 10
2.3. Artificial Neural Network (Feed-forward and Feed-Backward) . . . 11
2.4. An example of two dimensional linearly separable data, and the process of SVM to divide the data using a hyperplane.) . . . 13
2.5. An interpretation of an ROC curve. . . . 20

3.1. Classification Approach . . . 25
3.2. (a) A GUI interface of Bugzilla bug tracking system. (b) A complete bug life cycle of Bugzilla, each rectangle represents a unique status of a bug. . . . 28
3.3. The frequency distribution of different priority levels i.e., P1, P2, P3, P4 and P5, which are assigned to different bug severity level. . . . 30
3.4. A plot between the bugs of different severity level and the number of days spent to fix them. Each bug is represented by a dot where the color of the dot represents its severity level. . . . 31
3.5. Classification Approach . . . 32
3.6. An example of developer bug fix activity log-book . . . 33
3.7. Mozilla project average bug fix effort per bug related to the number of bugs fixed per year. . . . 34
3.8. Number of bug counts as function of the number of bug fix days. . . . 34
3.9. A list of top 10 Mozilla developers on the basis of the contribution metrics. The pop up window shows the details of the metric DCM-WNBFixed. . . . 35
3.10. Distribution of five different contribution metrics of the top 20 Mozilla developers. . . 36
3.11. Plot between the percentage of fixed bugs and the percentage of developers who fixed them. . . . 37
3.12. A visualization of the developer's bug fix activity data. Interaction among developers. 37
3.13. Interaction between source files and developers. . . . 38

4.1. An ERD of a database for program file change analysis . . . 49
4.2. Example of bug fixing and bug introducing transaction . . . 51
4.3. Distribution of source file changes per year (Mozilla 1000 C++ files). . . . 55
4.4. Change history of 10 C++ source file of Mozilla project. . . . 56

5.1. Classification Approach . . . 63
5.2. Scatter plot between code metrics and number of faults. . . . 66
5.3. Training of fault prediction model using single hidden layer neural network. The training data set is labeled with isbuggy (isbuggy=yes or isbuggy=no) criteria. . . . 73
5.4. Training of fault prediction model using double hidden layers of neural network. The training data set is labeled with isbuggy (isbuggy=yes or isbuggy=no) criteria. . . . 74
5.5. Training of fault prediction model using REPTree with two folds. The tree is built with metrics data, where nof=NOF, numberOfIncludedfiles=NOIF, totalrpoints=RPC, maximumcdensity=MCD, totallocvar=LVC and loc=LOC (line of codes) . . . 77

5.6. Training of fault prediction models using Neural Network. The training data set is labeled with buglevel (low, medium and high) criteria. 81
5.7. Precision-Recall values of individual labels (yes, no), obtained when classifiers trained with different machine learning algorithms 85
5.8. Precision-Recall Curve of different classifiers at different threshold value of the classifier. 87
5.9. Receiver Operating Characteristic Curve (ROC) of different classifiers. 87
5.10. Precision-Recall curve of different models. When models are trained using *bugLevel* criteria. 91
5.11. Percentage of Buggy Files 93
5.12. Average Function Metrics 96
5.13. Average File Sizes 96
5.14. Average File Metrics 97
5.15. Predictions for individual files 98

6.1. An Example to obtain a set of logical-coupling patterns for source file f_1, which were fixed 3-times in the past, and logically coupled with source files f_2,f_3,f_4,f_5,f_6. The obtained set of logical-coupling patterns of source file f_1 with maximum frequency is shown in the right-most column. 107
6.2. Percentage of the selected source files, which were fixed 1-to-15 times. Like the percentage of those source files that were fixed 2-times is 17%. These source files are selected from GNOME project repository to perform the experiment. 110
6.3. Scatter Plots of the logical coupling metrics versus number of bugs. 112

7.1. Frequency of submitted bug reports during the evolution of Mozilla project 122
7.2. Process to convert the textual data of SCRs into LSI based TDM matrix. We took the transpose of actual TDM matrix and then labeled with the names of source files. For labeling and classification two different approaches are used i.e., single and multi label classifications 125
7.3. An example of bug report's summary and description. 126
7.4. An example of a groups of source files which is obtained by applying *frequent Patterns Mining* and *Association Rules* on the bug-fix transaction data of Mozilla CVS repository. 129
7.5. Labeling of Bug-Reports with group-id. Each group-id represent a group of source files which are changed together. It is an indirect approach to label each bug-fix report with multiple source files, and use single label classifiers for classification. 130
7.6. Precision values of single and multi-label classification when classifiers trained with three different machine learning algorithms i.e., support vector machine, J48 and Naive Bayes (NB) 137
7.7. Recall values of single and multi-label classification when classifiers trained with three different machine learning algorithms i.e., support vector machine, J48 and Naive Bayes (NB) 137
7.8. Precision-Recall plot of individual labels. 138

8.1. Process flow diagram of the proposed *Automatic Bug Triage System (BTS)* based on single label machine learning classification. 143

8.2. Classification Accuracy of 7 different classifiers, train and test with 4 different dimension size data sets (dimension reduced by LSI). . . . 146
8.3. Classification Precision of 7 different classifiers, train and test with 4 different dimension size data sets (dimensions are reduced by LSI). . . . 148
8.4. Classification Recall of 7 different classifiers, train and test with 4 different dimension size data sets (dimensions are reduced by LSI). . . . 148
8.5. F-Measure of 7 different classifiers, train and test with 4 different dimension size data sets (dimensions are reduced by LSI). . . . 149
8.6. Classification accuracy of 7 different classifiers, train and test with a data set whose dimensions are reduced by feature selection method. . . . 150
8.7. Precision, Recall & F-measure of 7 diff. classifiers, train & test with a data set whose dimensions are reduced by feature selection method. . . . 150
8.8. Multi-Label classification of software CR 155
8.9. The average Precision values of Multi-label Classifiers i.e., Label Power Set and ML-kNN methods (with data sets of 3-different Label size) 160
8.10. The average Recall values of Multi-label Classifiers i.e., Label Power Set and ML-kNN methods (with data sets of 3-different Label size) 161

List of Tables

3.1. List of Metrics 27
3.2. Classification of bug reports on the basis of bug severity level and developer defined bug priority. 29
3.3. Person and Spearman's Correlation of bug severity level with a set of source file metrics. 31
3.4. Correlation of Metrics With Effort 41
3.5. Evaluation Measures of Different Machine Learning Based Effort Estimation Model . . 42

4.1. Description of database schema 48
4.2. Hunk pair format description 52
4.3. Mozilla project program file change history (only 8251 C++ source files revisions). . . 54
4.4. Source files change distribution (randomly selected 10 C++ files from Mozilla project) and calculation for Risky Program File for new changes. 56

5.1. Source code metrics correlation with the number of faults 66
5.2. Metrics correlation with each other 68
5.3. The top ten Principal components obtained from the transformation of raw metrics data of Firefox (Rel. 0.8) project. 69
5.4. Fault prediction models using multiple linear regression 70
5.5. Evaluation of fault predictor models obtained using raw metrics data i.e., without PCA 76
5.6. Point Biserial Correlation between Code Metrics and bug level 79
5.7. Class distribution for the classification schema *isBuggy* (only C++ source files) 81
5.8. Class distribution for the classification schema *bugLevel* (only C++ source files) 82
5.9. Evaluation measures (Precision, Recall, F-measure, and ROC Area) of different classifiers. These classifiers are trained and tested using the source files metrics and fault data of Firefox Release 0.8 84
5.10. Evaluation measures of different fault prediction models, trained with the source files (cpp) of Firefox Release 0.8 88
5.11. Classification of cpp files of Firefox Release 0.8 using *bugLevel* criteria. 89
5.12. Function Metrics of AHS files 94
5.13. Function Metrics of Firefox files 95
5.14. File Metrics 95
5.15. Classification of Firefox files using the predictor obtained from AHS 99
5.16. Classification of AHS files using the predictor obtained from Firefox 99
5.17. Using AHS 1.3.x as predictor 100
5.18. Using AHS 2.0.x as predictor 100
5.19. Using Firefox 0.8 as predictor 100
5.20. Using Firefox 1.0 as predictor 101
5.21. Using Firefox 1.5 as predictor 101
5.22. Using Firefox 2.0 as predictor 101

6.1. Spearman's Correlation Between the Program Files Coupling Measures and the Number of Bugs 113
6.2. Stepwise Linear Regression (Model Summary) 116

6.3. Total Variance Explained (PCA) . . . 116
6.4. Extracted Four Principle Component Using Principle Component Analysis Method . . 117
6.5. Regression After Applying PCA (Model Summary) . . . 117
6.6. Classification of source files on the basis of complexity level i.e., low, medium and high 118
6.7. Classification evaluation measures of each class with PCA (training instances = 1537) 119

7.1. Percentage of bug fix transactions in which single or multiple source files have changed 123
7.2. Training data sets for single and multi label Machine Learning classification . . . 133
7.3. Single-label Classification Evaluation Using 3 Different ML-Algorithms . . . 135
7.4. Multi-label Classification Evaluation Using *Label Power set (LP)* With 3 Different ML-Algorithms as Base Classifiers . . . 136

8.1. Data related to the bug report's index terms . . . 145
8.2. Different data sets on the basis of features size (dimensions). . . . 145
8.3. Evaluation results of bug report's classification (When train & test with LSI based low dimensional data) . . . 147
8.4. Evaluation results of bug report's classification (When train & test with data set obtained without using LSI) . . . 149
8.5. SCR assigned with multiple labels . . . 152
8.6. Frequency distribution of effort data (bug fix time) to obtain distinct classes of effort . 154
8.7. Size of labels and the label selection criteria for three different data sets . . . 154
8.8. Multi-Label Classification of Bug Reports Using *Label Power Set* Classifiers (Problem Transformation Methods Based on Support Vector Machine, J48 and Naive Bayes) . . 157
8.9. Multi-Label Classification of Bug Reports Using *ML-kNN* Classifiers (Algorithm Adaptation Methods) . . . 158
8.10. Average Classification Measures of Multi-Label Classifiers i.e., *LP-method* and *ML-kNN* 159

1 Introduction

Failure is not fatal, but failure to change might be [John Wooden]

The purpose of this chapter is to give the reader an overview of this thesis. This chapter is divided into three parts. In the first part, we present a brief introduction to the field of MSR, and discuss its contributions to the field of software engineering. In the second part, we discuss the motivations, goals and contributions of this thesis. Finally, in the third part, we present the layout of this thesis.

1.1 Mining Software Repositories

During the evolution of a large software system, a huge amount of data is generated. These data are maintained by the repositories of a bug tracking system and a version control system. Software repositories contain a wealth of valuable information for empirical studies in software engineering. The availability of software repository data leads to a new research area called *Mining Software Repositories* (MSR) [Zimmermann, 2009]. The field of MSR provides a platform for researchers to analyze the rich data available in software repositories. The overall goal of MSR is to extract the hidden information from the evolutionary data of software projects, and used them to build new theories, models and define new standards for software engineering.

In case of open source software systems, the repositories of software's evolution data are freely available. However, software repositories were used mostly for their intended activities such as maintaining versions of the program file and tracking the status of defect reports. Today, researchers can leverage softwares repositories data to understand the evolution of a software system. Software practitioners and researchers are beginning to recognize the potential benefit of mining software repositories for software maintenance and other purposes. In the recent years, research is now proceeding to find appropriate ways in which MSR can be used to support the maintenance of software systems, improve software design quality, and empirically validate novel approaches and methods [Hassan, 2008].

Although, MSR has several important applications in software engineering, but still, MSR facing several challenges and difficulties. One of the main reasons of these difficulties might be software repositories, which are not designed to support empirical understanding of a software project. Software version control system and defect tracking systems have various irregularities and issues in their

recorded information. Similarly, the other challenges are, building relevant theories and models of software development that can be empirically validated using the available information in software repositories [Hassan et al., 2005]. Moreover, there are challenges in developing new data management and mining algorithms, or customizing existing mining algorithms for large-scale software repositories.

Besides, these difficulties and challenges, MSR has been contributed a huge amount of valuable research works in the field of software engineering. The analysis of MSR publications shows that 80% (approx) of the published papers (from 2004 to 2008) focus on source code and bug-related repositories [Hassan and Xie, 2010]. These repositories are mainly evolved during the maintenance phase. Similarly, the results of a detailed survey on MSR show that the published papers on MSR are mostly related to the software maintenance tasks like, program comprehension, origion analysis and refactoring, change comprehension, fault prediction, impact analysis, etc. [Kagdi et al., 2007].

Furthermore, MSR focus on developing tools to support software developers and empowering software maintainers. Examples of some of the developed tools are: *EROSE* plugin for Eclipse [Zimmermann et al., 2005a], *EROSE* is used to guide a developer, when a developer wants to change a key then *EROSE* on the basis of previous change history, recommend to change some other function also. *MAPO*, it is used for mining API usage from open source repositories [Xie and Pei, 2006]. *BugTriage*, it is used to assign a new bug report to an appropriate developer [Anvik et al., 2006]. *DynaMine*, it is used for mining error or usage patterns from code revision histories [Livshits and Zimmermann, 2005]. *HIPIKAT*, it can be viewed as a recommender system for software developers. *HIPIKAT* draws its recommendations from a project's evolution data [Čubranić and Murphy, 2003].

1.2 Thesis Overview

The analysis given in the previous section, reveals that MSR research is more focused on software maintenance activities. The research presented in this thesis is also focused on development of models for software maintenance tasks. In the following sections, we present an overview of this thesis.

1.2.1 Motivation

In case of a software project life cycle, software maintenance is the last phase. Software maintenance phase consists of several activities. Most of the maintenance activities are directly linked with the modifications in source codes. Modifications in source codes are required to accomplish four different types of modification tasks. If modifications are required to fix the fault, then it is called *Corrective* maintenance. If modifications are performed to keep a software product alive in a changed or changing environment, then it is called *Adaptive* maintenance. If modifications are performed to improve performance or maintainability, then it is called *Perfective* maintenance. Finally, if modifications are performed to detect and correct latent faults before they become effective faults, then it is called *Preventive* maintenance [Lientz and Swanson, 1980]. Most of the software development cost is spent to accomplish the above mentioned maintenance tasks. The results of different surveys indicate that software maintenance consumes 60% to 80% of the total life cycle costs. The most challenging problems related to the software maintenance task are: identification of fault prones modules, impact

analysis of software changes, and program comprehension [Canfora and Cimitile, 2000].

In order to reduce the cost and improve the quality of software maintenance tasks, several techniques of MSR have been developed. These techniques use the software development history, and apply machine learning techniques on it to build automation models or tools. Thomas Ball et al. [Ball et al., 1997] gives details how to utilize version control system data to study the system evolution. Michael Fischer and Harald Gall developed approaches to extract data from software repositories and populate a database for logical coupling analysis [Gall et al., 2003], [Fischer et al., 2003].

The previous history of source code changes can be used to find the risk of fault associated with the new changes in the source code. Jack and Zimmermann [Śliwerski et al., 2005b] analyzed the CVS and the Bugzilla (a bug tracking system) repositories of Mozilla and Eclipse projects. They used *annotation* and *diff* (file difference) commands of CVS to identify fix-inducing changes. Kim et al. [Kim et al., 2008] introduced a technique for classifying a software change as clean or buggy. Similarly, Mockus and Weiss [Mockus and Weiss, 2000] presented a model to predict the risk of new changes, based on historical information. Most of the previous work used the history of software changes to classify a source files as faulty or clean source files. Whereas, the available data of software changes can be used to define a new set of metrics, which measures the number of different types of changes. These, metrics can be used to build a risk model.

Bug fixing is the main task of corrective software maintenance that takes up a large amount of software testing and maintenance effort. Particularly, software testing requires a lot of effort to verify that the new changes are not error prone. Several works have been done to find the fault-prone locations in software systems. The majority of this work builds prediction models using metrics that predicts where future defects are likely to occur. Most of the developed fault prediction models used code metrics as predictors [Nagappan et al., 2006]. The performance of the fault prediction models can be improved if we used metrics, which are highly correlated with the number of faults. It is found that logical coupling measures are highly correlated with the number of defects [D'Ambros et al., 2009b, Eaddy et al., 2008a]. Therefore, logical coupling metrics can be used to build better fault prediction models.

Bug tracking system maintained the complete history of bug fixing activities of each reported bugs. This data can be used to mine the actual time that has been spent to fix a big. This information can be used to build an effort estimation model. Similarly, classification of bug reports (software change request-SCR) can be used for impact analysis and bug triaging [Canfora and Cerulo, 2005] [Anvik et al., 2006]. In case of single label classification SCRs cannot be assigned multiple labels (i.e., source files name) simultaneously. Whereas, in case of impact analysis of SCRs, some SCRs impacted more than one source files. Therefore, it is required to label each SCR with multiple source files. This issue can be resolved by using multi-label classification techniques of machine learning.

1.2.2 Problem Statement

In case of open source software system, effort data related to the fixing of faults are not available. Therefore, there is no state of the art effort estimation model for OSS is available. This thesis

investigates that how developer's bug fix activity data can be used to mine the effort data. Moreover, thesis investigates that which source code and process metrics can be used to build the effort estimation model. Furthermore, this thesis investigates the previous history of source file changes to answer the question how much risk of fault is associated with the new changes in a source file? The thesis also investigates the issues related to the building of fault prediction models. Fault prediction models are built using software metrics and machine learning techniques. The main issues addressed by this thesis related to the fault prediction models:

- In case of software fault prediction model it is found that in most of the cases a set of code metrics are used as predictors. These predictors are not highly correlated with the number of post release defects. Consequently, the obtained models are not reliable. Therefor, in order to construct the state of the art fault prediction model, one must have a new set of predictors, which should have high correlation with the number of faults.
- To build the fault prediction models, binary classifiers are mostly used. Whereas, the class distribution of source files, i.e., faulty and clean source files are skewed. In case of skewed class distribution the binary classifiers results are biased. Therefore, one has to find a set of classifiers, which perform well with skewed class distribution.

Finally, this thesis investigates the limitation of single label classification of SCRs, which are frequently used for bug-triaging and impact analysis. The main issues addressed by this thesis related to the single-label classification of SCRs are: how one can assign multiple labels against each SCR and how one can improve the indexing technique of the textual data of SCR.

1.2.3 Research Hypothesis

The research work presented in this PhD thesis is for proving the following research hypothesis:

- **H1:** Activities of the developers related to the fixing of bugs are stored into a bug tracking system. This data can be mined to build an effort estimation model.
- **H2:** The history of software changes can be used to build a risk model.
- **H3:** Logical coupling metrics are highly correlated with the number of faults. The prediction power of software fault prediction models can be improved by using a set of logical coupling metrics as predictors.
- **H4:** Multi-label classification technique of machine learning can be used in software engineering for the impact analysis of software change requests, and automatic bug triaging.

1.2.4 Goals/Objectives

The main objective of this thesis is to build models for effort estimation, risk analysis, and fault predictions. Another objective of this thesis, is to apply multi-label classification techniques to classify SCRs for impact analysis and bug triaging.

1.2.5 Thesis Contribution

In this thesis, we made two contributions to the body of MSR research. First, we extract metrics data from software's repository, and build models for effort estimation, risk estimation, and faults prediction. Second, we classify software changes requests (SCRs) using multi-label classification for the impact analysis of SCRs and bug triaging. Parts of the work presented in this thesis have been published in international peer reviewed journal, conferences and workshops:

I) Estimation and Prediction Models:

- The effort estimation model for open source software system (OSS), published in [Ahsan et al., 2009b] and [Ahsan et al., 2010]. Effort estimation model built in two steps. In the first step, we mined the effort data from the history of developer's bug-fix activity. Then, in the second step, we used the obtained effort data and the set metrics to build the effort estimation model. In this experiment, we found that in case of OSS, bug fixing activities of developers can be used to mine the effort data.
- The risk model, which estimates the risk of fault associated with the new changes in source files, published in [Ahsan et al., 2008]. We published the results of a case study of Mozilla project. In this case study, we found that source files, which have high probability of buggy changes are more risky for new changes.
- The fault prediction models built using code metrics, published in [Ahsan and Wotawa, 2010c], [Ferzund et al., 2008a] and [Ferzund et al., 2008b]. The published models are obtained using regression and classification techniques of machine learning. In order to analyze the impact of skewed class distribution on the classification results, we performed an experiment with two different classification schema. Furthermore, to analyze the performance of different machine learning classifiers, we performed another experiment with eight different classifiers. Classifiers trained and tested using the metrics data of Firefox and Apache projects.
- The fault prediction models built using logical coupling metrics, published in [Ahsan and Wotawa, 2010a]. We used our own heuristic approach to mine the bug fixing patterns of logically coupled source files. We used the bug-fixing coupling patterns to define a new set of logical coupling metrics. We found that logical coupling metrics highly correlated with the number of faults and can be used to build a fault prediction model.

II) Multi-label Classification of SCRs:

- Our approach to perform the impact analysis of software change requests, published in [Ahsan and Wotawa, 2010b]. We, used two different approaches to perform the impact analysis of SCRs. These approaches are based on single and multi-label classification of SCRs. First, we directly labeled each SCRs with multiple impacted source files and applied multi-label machine learning for the classification of SCRs. Second, we labeled each SCRs with a single label, i.e., a group id of source files. We obtained different groups of source files, which frequently changed together. For grouping of source files, we used pattern mining and association rules. Finally, In order to overcome the issue of high dimensionality, synonymy and polysemy of *vector space model* (VSM), we used *latent semantic indexing* (LSI).

- Our approach to build an automatic bug triage system, published in [Ahsan et al., 2009a] and [Ahsan et al., 2009c]. We used both single and multi-label classification techniques to build a bug triage system. We, found that a bug triage system based on multi-label classification performed well. The bug triage system based on multi-label classification, is not only capable, to assign SCRs to the appropriate developers. But also identify the name of the suspected source files, which are required to be modified, and estimate time, which is required to fix the SCR.

1.3 Thesis Layout

The remaining chapter of this thesis organized as follows: Chapter 2 gives an overview of different types of machine learning techniques. These techniques have been used to accomplish various task related to this thesis. Chapter 3 describes an approach to mine the effort data (related to the fixing of a fault) from the developer's bug-fixing activity. Developer's activity data are obtained from the repository of a bug tracking system. This chapter further shows that how effort estimation model can be built using a set of metrics and different machine learning algorithms.

Chapter 4 introduces a light weight approach to construct a risk model using the data of source file changes. The risk model is capable, to identify risky source files on the basis of their previous change history. Furthermore, in this chapter it is described in detail about those approaches, which are used to extract the change history of source files and stored into a relational database. Chapter 5 present two different approaches for the construction of fault prediction models, i.e., regression and classification. Furthermore, this chapter describes the parametric and non-parametric approaches to construct the fault prediction models. Different parameters, which influence the model evaluation criteria are described in details. Chapter 6 defines eight new metrics for logical couplings, and shows that logical coupling metrics highly correlated with the number of faults (post release defects).

Chapter 7 shows that how information search and retrieval techniques can be applied on the textual data of software change requests (SCRs) to convert SCRs into term-to-document matrix (TDM). This chapter also describes that how TDM data of SCRs can be classified for impact analysis using single and multi-label classification techniques. Chapter 8 describe that how single and multi-label classification of SCRs can be used to develop a bug triaging system.

Chapter 9 describe the previous research work, which are more related with this thesis work. Finally, a conclusion of this thesis work and future work are given in Chapter 10.

2

Machine Learning Techniques

In this chapter, a brief description of theories and techniques of machine learning is presented. These theories and techniques have been used repeatedly to accomplish the experimental tasks related to this thesis work, and discuss in Chapter 3 to Chapter 8. Since most of the experimental work is based on machine learning techniques. Therefor, the importance of the application of machine learning in software engineering is described in Section 2.1. Then, in Section 2.2 several supervised and unsupervised machine learning techniques are described.

2.1 Application of Machine Learning In Software Engineering

Software engineering is concerned with the development and evolution of large and complex software-intensive systems. It covers theories, methods and tools for the specification, architecture, design, testing, and maintenance of software systems. Today's software systems are significantly large, complex and critical, that only through the use of automated approaches can such systems be developed and evolve in an economic and timely manner.

The application of artificial intelligence techniques to software engineering has produced some encouraging results [Fenton and Neil, 1999], [Gyimothy et al., 2005]. Artificial intelligence (AI) techniques like knowledge-based approach, automated reasoning, expert systems, heuristic search strategies, temporal logic, planning, and pattern recognition can play an important role in different areas of software engineering. Examples are automatic fault prediction model [Khoshgoftaar and Seliya, 2003] and automatic software debugging [Wotawa, 2002]. As a subfield of AI, Machine Learning (ML) deals with the issue of how to build computer programs that improve their performance at some task through experience [Zhang and Tsai, February 28, 2005].

Machine learning usually refers to the changes in systems that perform tasks associated with artificial intelligence. Such tasks involve recognition, diagnosis, planning, robot control, prediction, etc. The *changes* might be either enhancements to already performing systems or ab initio synthesis of new systems [Nilsson, November 03, 1998]. In general, Machine Learning is about learning to do better in the future based on what was experienced in the past.

In the field of software engineering, one can use ML to solve complex and time consuming problems. This could be possible, if a software engineer or software practitioner knows how to use ML tools.

One has to be a clear understanding of both the problems, and the tools and methodologies utilized. It is imperative that we know,

1. The available machine learning methods at our disposal
2. Main features of those methods
3. Possible condition under which the methods can be most effectively applied, and
4. Their theoretical understanding.

In the next section, we describe the basic functionality of some useful machine learning tools

2.2 Supervised and Unsupervised Machine Learning Techniques

Machine learning algorithms can be categorized into two main categories, i.e., *supervised* and *unsupervised*. The distinction is drawn from how the learner classifies data.

Supervised Machine Learning: In case of supervised machine learning algorithms, the classes are predetermined. Classes can be considered as a finite set (which may be discrete or continuous), which are previously known. The goal of supervised machine learning is to build a concise model of the distribution of class labels in terms of predictor features. Finally, the resulting classifier is then used to assign class labels to the testing instances or examples where the values of the predictor features are known, but the value of the class label is unknown. There are many examples of supervised machine learning classification like decision tree induction, naive Bayes, neural network, support vector machine etc.

Supervised machine learning techniques are further divided into two different categories, i.e., *Regression* and *Classification*. In *Regression* type supervised machine learning technique. Machine learning algorithms predict continuous or numerical values. Whereas, *Classification* type machine learning algorithms are used to classify each instance into a predefined discrete value or label like, *true* or *false*. Similarly, *Regression* further divided into two different types of regression methods, i.e., *Parametric Regression* and *Non-parametric Regression*. Parametric regression is used when we have a data set with known distribution. Whereas, non parametric regression is used when the distribution of data is unknown [Weisberg, 2005].

Unsupervised Machine Learning: Unsupervised learners are not provided with classifications. In fact, the basic task of unsupervised learning is to develop classification labels automatically. Unsupervised algorithms seek out a similarity between pieces of data in order to determine whether they can be characterized as forming a group. These groups are termed clusters, and there is a whole family of clustering machine learning techniques. In unsupervised classification, often known as *cluster analysis* the machine is not informed that how the texts are grouped. Its task is to find some grouping of the data. In case of cluster analysis (K-means), the machine is informed in advance how many clusters or groups it should form.

In our research work, we mostly used supervised machine learning algorithm to build a fault prediction model, effort estimation model, and for the bug triage system. In the following subsections, we

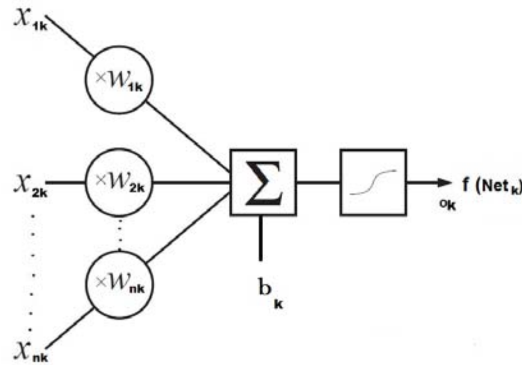


Figure 2.1. Artificial neural network

briefly describe only those supervised machine learning algorithms, which have been used to perform the thesis work.

2.2.1 Artificial Neural Network (ANN)

Non-parametric regression is used when we have a data set with unknown distribution. Artificial neural networks are a useful statistical tool for non-parametric regression [Stern, 1996]. ANN has the advantage of learning from data that exhibits either highly nonlinear relationship since the inherent transfer functions is nonlinear in nature.

ANNs have been widely applied to solve many difficult problems in different areas, including pattern recognition, signal processing, language learning, software engineering, etc. In 1962, Rosenblatt [Rosenblatt, 1962] first introduced single layer perceptrons. The initial model was so simple and have several limitation. Therefore, did not receive much attention. Rumelhart et al. [Rumelhart and Williams, 1986] presented the new developments of ANN, including more complex and flexible ANN structures and a new network learning method. Since then, ANN has become a rapidly growing research area.

Artificial neural networks are considered as models of biological neural structures. The main unit of a neural network is a neuron. This neuron consists of multiple inputs and a single output, as shown in Figure 2.1. Each input is modified by a weight, which multiplies with the input value. The neuron will combine these weighted inputs and, with reference to a threshold value and activation function, use these to determine its output. This behavior follows closely our understanding of how real neurons work in human brain.

How Neural Network Works:

Neural networks are composed of simple neurons operating in parallel. The connections between neurons largely determine the network function. One can train a neural network to perform a particular function by adjusting the values of the connections (weights) between neurons. Typically, neural networks are adjusted, or trained, so that a particular input leads to a specific target output. Figure

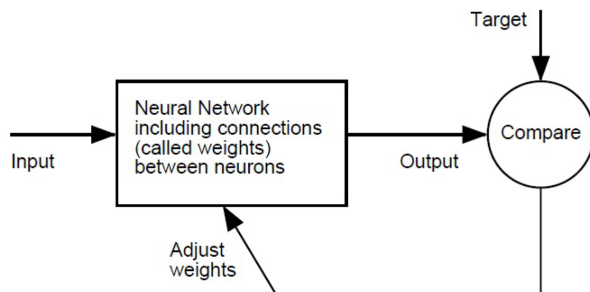


Figure 2.2. Example of how neural networks trained by adjusting output for some specific input. leads to a specific target output

2.2 illustrates such a situation. There, the network is adjusted, based on a comparison of the output and the target, until the network output matches the target. Typically, many such input/target pairs are needed to train a network.

The Learning Process of ANN

According to learning rules, ANN can be classified into two categories, supervised-learning networks and unsupervised-learning networks (Lin and Lee, 1996). In case of supervised learning networks, at each instant of time when input is applied to an ANN, the corresponding desired response of the system is given. The network is thus told precisely what it should be emitting as output.

Neural networks consist of neurons. Figure 2.1 shows the structure of a neuron. In this model, the k_{th} processing element computes a weighted sum of its inputs x_j (independent variable) and basis b_k as the input to the activation function, the output of the activation function is the output of the neuron o_k (dependent variable). Suppose there are m inputs, $x_1; x_2; \ldots ; x_n$, to the neurons and the weights associated with these inputs are $w_1; w_2; \ldots ; w_n$. So the operation of the neuron can be described as following.

$$Net_k = b_k + w_{1k}x_1 + w_{2k}x_2 + w_{3k}x_3 + ...w_{nk}x_n \tag{2.1}$$

$$o_k = f(Net_k) \tag{2.2}$$

Where, f(.) is the activation function neuron

Architecture of Neural Networks

Feed-forward networks: In case of feed-forward ANNs, signals are allowed to travel one way only i.e., from input to output. There is no feedback (loops) i.e. the output of any layer does not affect that same layer (see Figure 2.3). Feed-forward ANNs tend to be straight forward networks that associate inputs with outputs. They are extensively used in pattern recognition. This type of organization is

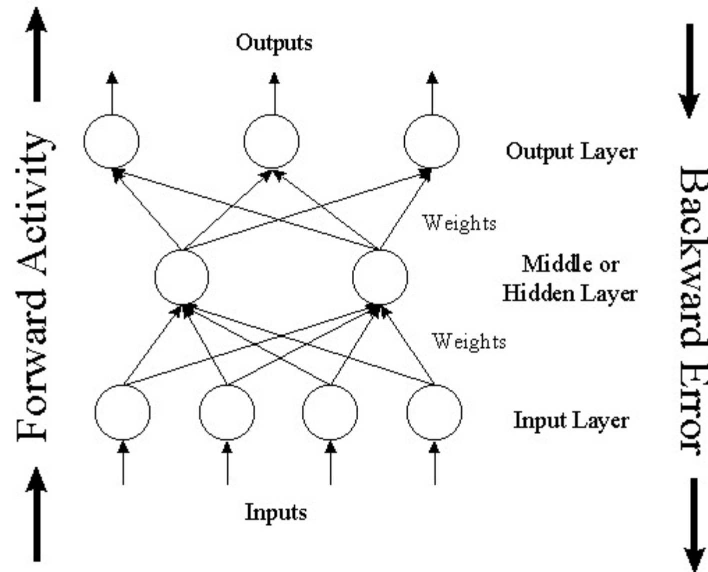


Figure 2.3. Artificial Neural Network (Feed-forward and Feed-Backward)

also referred to as bottom-up or top-down.

Feedback networks: In case of feedback networks, signals traveling in both directions by introducing loops in the network. Feedback networks are very powerful and can get extremely complicated. Feedback networks are dynamic; their *state* is changing continuously until they reach an equilibrium point (see Figure 2.3). They remain at the equilibrium point until the input changes and a new equilibrium needs to be found. Feedback architectures are also referred to as interactive or recurrent, although the latter term is often used to denote feedback connections in single-layer organizations.

Backpropagation Training Algorithm

The most popular training algorithm for multilayer neural networks is *Backpropagation* [PDP Research Group, 1986]. The training steps of *Backpropagation* are,

1. First, initializes the network with a random set of weights and basis, and the network trains from a set of input-output pairs.
2. Each pair requires a two stage learning algorithm: forward pass and backward pass. The forward pass propagates the input vector through the network until it reaches the output layer. First the input vector propagates to the hidden units. Each hidden unit calculates the weighted sum of the input vector and its interconnection weights.
3. Each hidden unit uses the weighted sum to calculate its activation.
4. Next, hidden unit activations propagate to the output layer. Each node in the output layer calculates its weighted sum and activation.

5. The output of the network is compared to the expected output of the input-output pair, and their difference (error vector) is used to train the network to minimize the error, this is called backward pass.

6. First the error passes from the output layer to the hidden layer updating output weights.

7. Next each hidden unit calculates an error based on the error from each output unit, the error from the hidden units updates input weight. The training stops only when the sum of squared error satisfies the requirement or the number of epochs passes the set point, where an epoch means that all the training data go through the forward pass and backward pass once.

8. The least mean square algorithm computes the weight updates for each input sample and the weights are modified after each sample. This procedure is called sample-by-sample learning. An alternative solution is to compute the weight update for each input sample and store these values (without changing the weights) during one pass through the training set (epoch).

9. At the end of the epoch, all the weight updates are added together, and only then the weights will be updated with the composite value.

In our work, we used ANN for both classification and regression. We used WEKA[1](weka.classifiers. functions.MultilayerPerceptron) that uses backpropagation to classify instances. The network can also be monitored and modified during training time. The nodes in this network are all sigmoid (except for when the class is numeric in which case the output nodes become unthresholded linear units).

2.2.2 RBF Network

RBF (Radial Basis Function) network is a three-layer feedback network, in which each hidden unit implements a radial activation function, and each output unit implements a weighted sum of hidden units outputs. RBF divides its training procedure into two stages. In the first stage, the centers and widths of the hidden layer are determined by clustering algorithms. Whereas, in the second stage, the weights connecting the hidden layer with the output layer are determined by Least Mean Squared (LMS) algorithms or Singular Value Decomposition (SVD) [Kotsiantis, 2007].

In case of RBF Network, selecting the appropriate number of basis functions is an issue. Finding optimum number of basis function plays an important role. The number of basis functions controls the complexity and the generalization ability of RBF networks. RBF networks with very few basis functions cannot fit the training data adequately due to limited flexibility. Whereas, RBF Netwok with too many basis functions yield poor generalization abilities, since they are too flexible and erroneously fit the noise in the training data.

In our work, we used RBF Network for both classification and regression. To train the model with RBF Network, we used Weka (weka.classifiers.functions.RBFNetwork)that implements a normalized Gaussian radial basis function network. It uses the k-means clustering algorithm to provide the basis functions and learns either a logistic regression (discrete class problems) or linear regression (numeric

[1] http://www.cs.waikato.ac.nz/ ml/weka/

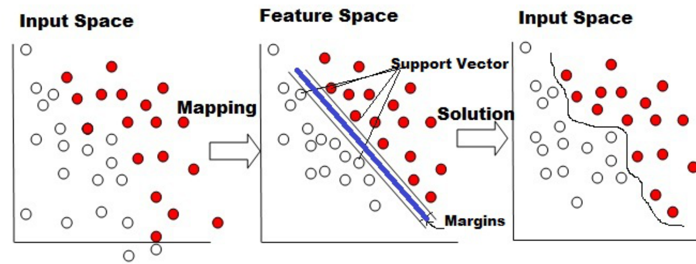


Figure 2.4. An example of two dimensional linearly separable data, and the process of SVM to divide the data using a hyperplane.)

class problems) on top of that.

2.2.3 Support Vector Machine (SVM)

Support Vector Machine supports both regression and classification tasks and can handle multiple continuous and categorical variables. SVM performs classification tasks by constructing hyperplanes in a multidimensional space that separates cases of different class labels. An excellent property of SVM is that their ability to learn can be independent of the dimensionality of the feature space.

SVM's measure the complexity of hypotheses based on the boundary with which they separate the data, not the number of features. This means that we can generalize, even if we have a very large number of features provided that our data is separable with a wide boundary using function from the hypothesis space [Smola and Schölkopf, 2004].

How Support Vector Machine Work

In case of SVM, a predictor variable is called an attribute, and a transformed attribute that is used to define the hyperplane is called a feature. The task of choosing the most suitable representation is known as feature selection. A set of features that describes one case (i.e., a row of predictor values) is called a vector. The task of SVM modeling is to find the optimal hyperplane that separates clusters of vector in such a way that cases with one category of the target variable are on one side of the plane and cases with the other category are on the other size of the plane. The vectors near the hyperplane are the support vectors.

A simple example a vector space model is shown in Figure 2.4 that contains data belongs to two separate target categories. One category of the target variable is represented by white circles while the other category is represented by red circles. In this idealized example, the instances with one category are in the lower left corner and the instances with the other category are in the upper right corner. The instances belong to two different categories are completely separated. The SVM analysis attempts to find a 1-dimensional hyperplane (i.e. a thick blue line) that separates the instances based on their target categories. There is an infinite number of such possible lines. The question is which line is better, and how do we define the optimal line. Thin black lines drawn parallel to

the separating thick blue line mark the distance between the dividing line and the closest vectors to the line. The distance between the thin black lines is called the margin. The vectors (points) that constrain the width of the margin are the support vectors. An SVM analysis finds the line (or, in general, hyperplane) that is oriented so that the margin between the support vectors is maximized. Figure 2.4 is an example of linearly separable data belongs to two different categories. SVM can also work with non-linearly separable and multi category data (for details read [Cristianini and Taylor, March 28, 2000]).

In our research work, we used support vector machine for both classification and regression. We used WEKA[2] tool to build regression and classification models. For regression, we used SMOreg (weka.classifiers.functions.SMOreg). Whereas, for classification we used SMO (weka.classifiers.functions.SMO) [S.S. Keerthi, 2001]. SMO stand for *Sequential Minimal Optimization*. It is a SVM training algorithm developed by John C. Platt [Platt, 1998], who claims that SMO is a simple and fast technique to solve the SVM's quadratic problem (QP). The main advantage of SMO compared to other SVM training algorithms is that it always chose the smallest QP problem to solve at each iteration. Another advantage is that SMO does not use storage matrix, which greatly reduces the memory usage.

2.2.4 Decision Trees

Decision Trees are commonly used in classification problems with categorical data [Ian H. Witten and Kaufmann, 2005]. Decision trees construct a tree of questions to be asked of a given example (instance) in order to determine the class membership by way of class labels associated with leaf nodes of the decision tree. This approach is simple and has the advantage that it produces decision rules that can be interpreted by a human as well as a machine. In our experiments, we used Weka class J48, which belongs to decision trees classifiers. J48 is an implementation of C4.5 release 8(3) that produces decision trees. This is a standard algorithm that is used for machine learning.

RepTree

The Reduced Error Pruning Trees(REPTrees) was introduced by Matti et al. in 2004 [Kääriäinen et al., 2004]. REPTree is a decision tree based machine learning algorithm. It is a fast tree learner who uses reduced error pruning. The REPTrees use a fast pruning algorithm to produce an optimal pruning of a given tree using information gain/variance and prunes [Ian H. Witten and Kaufmann, 2005].

Pruning of the decision tree is done by replacing a whole subtree by a leaf node. The replacement takes place if a decision rule establishes that the expected error rate in the subtree is greater than in the single leaf. The REP algorithm works in two phases: First the set of pruning examples S is classified using the given tree T to be pruned. Counters who keep track of the number of examples of each class passing through each node are updated simultaneously. In the second phase, which is a bottom up pruning phase, these parts of the tree that can be removed without increasing the error of

[2] http://www.cs.waikato.ac.nz/ ml/weka/

the remaining hypothesis are pruned away. The pruning decisions are based on the node statistics calculated in the top-down classification phase. Weka provides REPTree (weka.classifiers.trees.REPTree) algorithm to perform the task of regression through classification.

2.2.5 Naive Bayes

Naive Bayesian networks (NB) are very simple Bayesian networks which are composed of directed acyclic graphs with only one parent (representing the unobserved node) and several children (corresponding to be observed nodes) with a strong assumption of independence among child nodes in the context of their parent (Good, 1950). Thus, the independence model (Naive Bayes) is based on estimating (Nilsson, 1965). We used Weka to perform an experiment with Naive Bayes (weka.classifiers.bayes.NaiveBayes). For more information on Naive Bayes classifiers, see [John and Langley, 1995].

2.2.6 Boosting and Bagging Algorithm

Classical supervised learning algorithms generate a single model such as a *Decision Tree* classifier or *Multilayer Perceptron* (MLP) and use it to classify examples. Ensemble learning algorithms combine the predictions of multiple base models, each of which is learned using a traditional algorithm. *Bagging* and *Boosting* are well-known ensemble learning algorithms that have been shown to improve generalization performance compared to the individual base models.

Bagging method was formulated by Leo Breiman. *Bagging* stands for **b**ootstrap **agg**regat**ing** [Breiman, 1996]. Bagging predictors is a method for generating multiple versions of a predictor and using these to get an aggregated predictor.

Boosting It provides a general way to improve the accuracy of any given learning algorithm. The main idea behind boosting refers to a general method of producing very accurate predictions by combining moderately inaccurate or weak classifiers. *AdaBoost* is an algorithm that calls a given weak learning algorithm repeatedly, where at each step the weights of incorrectly classified examples are increased in order to force the weak learner to focus on the hard examples. For a detailed description of *AdaBoost* reader can refer to [Freund and Schapire, 1999].

2.3 Statistical Regression Analysis

In order to construct a regression base prediction models, we used two different statistical technique, i.e., *Multiple Linear Regression* (MLR) and *Logistic Regression* (LR). MLR is used when the correlation between dependent and independent variables is linear, and the value of each variable is a continuous number. Whereas, LR is used when the dependent variable has discrete or categorical value and independent variables have continuous values.

2.3.1 Multiple Linear Regression

Regression analysis is used to explain the dependence of a response variable on one or more predictors. Furthermore, it is used for the prediction of future values of a response, discovering which predictors are important and estimating the impact of changing a predictor on the value of the response. Multiple linear regression belongs to the parametric class of regression, which are used when we have a data set with known distribution [Weisberg, 2005].

In statistics, multiple linear regression (MLR) is used to model the linear relationship between a dependent variable and one or more independent variables. The model is fit such that the sum-of-squares of differences of observed and predicted values is minimized. The model expresses the value of a dependent variable as a linear function of one or more predictor variables and an error term.

$$y_i = b_0 + b_1x_1 + b_2x_2 + b_3x_3 + ... + b_kx_k + e_i \tag{2.3}$$

Where, b_0=regression constant, x_k=predictor (e.g., code metrics), b_k=regression coefficient, k=total number of predictor, y_i=predictand or dependent variable (e.g., number of defects) and e_i=residual errors. The model 2.3 is estimated by least squares, which yields parameter estimates such that the sum of squares of errors is minimized. The resulting prediction equation is,

$$y_i^{'} = b_0^{'} + b_1^{'}x_1 + b_2^{'}x_2 + b_3^{'}x_3 + ... + b_k^{'}x_k \tag{2.4}$$

Where, $y^{'}$ is the estimated value. The error term in 2.3 is unknown because the true model is unknown. Once the model has been estimated, the regression residuals are defined as,

$$e_i^{'} = y_i - y_i^{'} \tag{2.5}$$

Where, y_i=Observed value and $y_i^{'}$=Predicted value. Several regression statistics are computed as functions of the sums of square terms.

$$SSE = \sum_{i=1}^{n} e_i^{'2} \tag{2.6}$$

$$SST = \sum_{i=1}^{n} (y_i - \bar{y_i})^2 \tag{2.7}$$

$$SSR = \sum_{i=1}^{n} (y_i^{'} - \bar{y_i})^2 \tag{2.8}$$

Where, SSE=sum of squares error, SST=sum of squares total, SSR=sum of squares regression, and n=sample size. The regression equation is estimated such that the total sum-of squares can be

partitioned into components due to regression and residuals.

$$SST = SSR + SSE \tag{2.9}$$

Coefficient of determination ($\mathbf{R^2}$): The explanatory power of the regression is summarized by its R^2 value, computed from the sums-of-squares terms as,

$$R^2 = \frac{SSR}{SST} = 1 - \frac{SSE}{SST} \tag{2.10}$$

The multiple coefficient of determination, R^2, provides the quantification of how much variability in the fault prediction model can be explained by the regression model. R^2, is often described as the proportion of variance *accounted for*, *explained*, or *described* by regression. R^2 also called the coefficient of determination. The relative sizes of the sums-of-squares terms indicate how *good* the regression is in terms of fitting the calibration data. If the regression is *perfect*, all residuals are zero, SSE is zero, and R^2 is 1. If the regression is a total failure, the sum-of-squares of residuals equals the total sum-of-squares, no variance is accounted for by regression, and R^2 is zero.

2.3.2 Logistic Regression

Logistic regression is a variation of ordinary linear regression, which is used when the dependent variable is a dichotomous variable (i. e. it takes only two values, which usually represent the occurrence or non-occurrence of some outcome event, usually coded as 0 or 1) and the independent variables are continuous, categorical, or both.

In logistic regression, instead of predicting the value of a dependent variable Y from a predictor variable X or several predictor variables Xs, we predict the probability of Y occurring given known values of X or Xs. The logistic regression equation is similar to regression equations. If we have multiple predictor variables, i.e., Xs, then the logistic regression equation from which the probability of dependent variable, i.e., Y is predicted is given by the following equation.

$$P(Y) = \frac{1}{1 + e^{-(C_0 + C_1.X_{i_1} + ... + C_n.X_{i_n})}} \tag{2.11}$$

In equation 2.11 P(Y)is the probability of Y occurring, e is the base of natural algorithm, Cs are the coefficients which form a linear combination of independent variables Xs. In our experiment, the X_is are the code metrics and P is the probability of a source file being buggy.

2.4 Model's Evaluation Criteria

There are several measures, which are used to evaluate the performance of a model. In this section, these measures are categories into two categories on the basis of their applications. Those measures which are commonly used to evaluate a regression model are described in subsection 2.4.1. Similarly, those measures which are frequently used to evaluate a classification model are described in subsection 2.4.2. However, the evaluation measures, like absolute (MAE, RMSE) and relative error (MRE/RAE, RRSE), which are commonly used for the evaluation of regression models, are also used to evaluate classification models.

2.4.1 Evaluation Criteria for Regression Model

In order evaluate the performance of regression models following error measures are used. The mean absolute error (MAE), the root mean square error (RMSE), the mean relative absolute error (MRE) and the mean magnitude of relative error (MMRE), the root relative square error (RRSE), and the percentage of prediction PRED(x).

MAE is a quantity used to measure how close predictions are to the eventual outcomes. The mean absolute error is an average of the absolute errors. The absolute error is the difference between the actual and predicted value. MSE measures the average of the square of the error, and RMSE is the square root of MSE. It is used to measures the average magnitude of the error. Whereas, MRE, MMRE and RRSE are relative errors. Relative error is a ratio that compared the error to the size of the actual value. According to Conte et al [Conte and Shen, 1986], the MMRE value for effort prediction model should be $\leq$0.25. PRED(x) is obtained from relative error. Mostly, PRED(0.25) is used but PRED(0.30) is also acceptable. To be self-contained In the following definitions E_{ACT} stands for the actual effort, E_{PRED} for the predicted effort, n for the total number of observed values, and m for the number of correctly predicted values.

$$\text{Mean Absolute Error (MAE)} = \frac{\sum |E_{ACT} - E_{PRED}|}{n} \tag{2.12}$$

$$\text{Root Mean Square Error (RMSE)} = \sqrt{\frac{\sum (E_{ACT} - E_{PRED})^2}{n}} \tag{2.13}$$

$$\text{Mean Relative Error (MRE/RAE)} = \frac{|E_{ACT} - E_{PRED}|}{E_{ACT}} \tag{2.14}$$

$$\text{Mean Magnitude of Relative Error (MMRE)} = \sum_j MRE_j / n \tag{2.15}$$

$$\text{Root Relative Square Error (RRSE)} = \sqrt{\frac{\sum (E_{ACT} - E_{PRED})^2}{E_{ACT}}} \tag{2.16}$$

$$PRED(x) = \frac{1}{n} \sum_{i=1}^{n} \begin{cases} 1 & \text{if } MRE_i \leq x \\ 0 & \text{otherwise} \end{cases} \tag{2.17}$$

2.4.2 Evaluation Criteria for Classification Model

To measures the performance of classifiers five different measures are commonly used. These measures are, accuracy, precision, recall, F-Measure and ROC area. We explain these measures with the use of the following confusion matrix. We consider a binary classifier, which classify a source file as a buggy or bug-free file. We represent buggy files with Yes and bug-free files with No.

		Predicted	
		No	Yes
Observed	No	n_{11} (TN: True Negative)	n_{12} (FP: False Positive)
	Yes	n_{21} (FN: False Negative)	n_{22} (TP: True Positive)

Accuracy: Accuracy is the percentage of correctly classified instances. It is the ratio of the correct classifications to the total number of instances. Correct classifications is the sum of actual buggy files classified as buggy and the actual bug-free files classified as bug-free. Accuracy can be calculated by the following formula:

$$Accuracy = \frac{(n_{11} + n_{22})}{n_{11} + n_{12} + n_{21} + n_{22}} * 100$$

Buggy file Precision: It is the ratio of actual buggy files predicted as buggy to the total number of files predicted as buggy.

$$Buggy\ File\ Precision = \frac{n_{22}}{n_{22} + n_{12}}$$

Buggy files Recall: It is the ratio of actual buggy files predicted as buggy to the total number of actual buggy files. Recall represents the probability of detection of faulty/buggy modules, sometimes denoted as *PD*.

$$Buggy\ File\ Recall = \frac{n_{22}}{n_{22} + n_{21}}$$

Probability of False Alarm (PF) for Buggy Files : The probability of false alarm (PF) or False Positive for Buggy files is the ratio of false buggy files predicted as buggy to the total number of actual false buggy files.

$$Buggy\ File\ PF = \frac{n_{12}}{n_{12} + n_{11}}$$

Bug-free File Precision: It is the ratio of actual bug-free files predicted as bug-free to the total number of files predicted as bug-free.

$$Bug - Free\ File\ Precision = \frac{n_{11}}{n_{11} + n_{21}}$$

Bug-free File Recall: is the ratio of actual bug-free files predicted as bug-free to the total number of actual bug-free files.

$$Bug - Free\ File\ Recall = \frac{n_{11}}{n_{11} + n_{12}}$$

Probability of False Alarm (PF) for Bug-free Files: The probability of false alarm (PF) or False Positive for Bug free files is the ratio of false bug free files predicted as bug free to the total number of actual false bug free files.

$$Bug - Free\ File\ PF = \frac{n_{21}}{n_{21} + n_{22}}$$

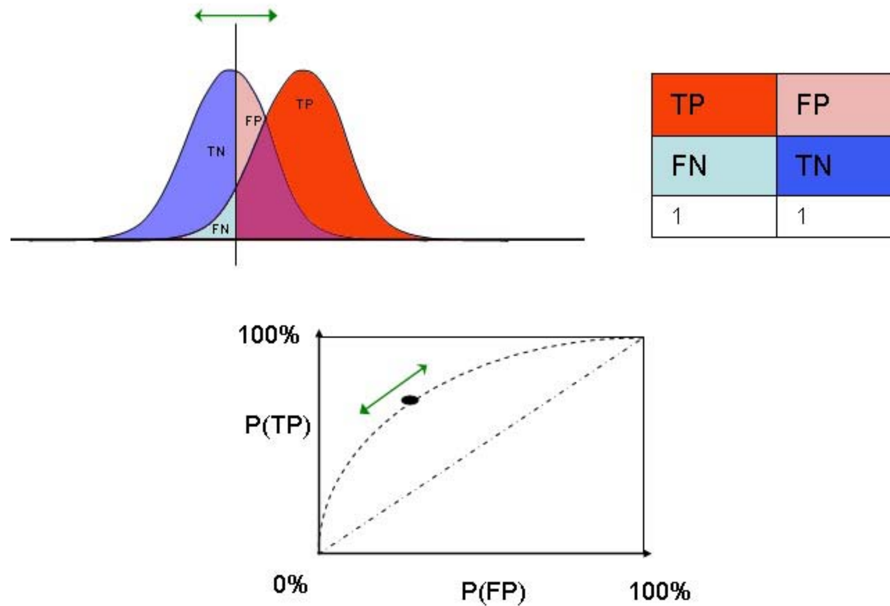


Figure 2.5. An interpretation of an ROC curve.

F-measure: F-measure is a harmonic mean of precision and recall with the weight of α. Normally, the weight α is set to 1, which puts the same weight on precision and recall.

$$F - Measure = \frac{(1+\alpha) * Precision * Recall}{\alpha * Precision + Recall}$$

$$F - Measure = \frac{2 * Precision * Recall}{Precision + Recall}$$

Receiver Operating Characteristic Curve (ROC): ROC is a plot of the probability of detection, i.e., recall (or PD) as a function of the probability of false alarm (PF) across all threshold settings. In simple words, an ROC space is defined by *False Positive Rate* (FPR) and *True Positive Rate* (TPR) as x and y axes respectively, which shows relative trade-offs between *True Positive* (benefits) and *False Positive* (costs). In classification, *Sensitivity* measures the proportion of actual positives, which are correctly identified and *Specificity* measures the proportion of negatives, which are correctly identified. Therefore, TPR is equivalent with sensitivity and FPR is equal to 1 - specificity. The ROC graph is sometimes called the sensitivity vs (1 - specificity) plot. Each prediction result or one instance of a confusion matrix represents one point in the ROC space.

Figure 2.5 gives an interpretation of ROC curve[3]. The best classifier would generate a point in the upper left corner or coordinate (0,1) of Figure 2.5, representing 100% sensitivity, i.e., no false negatives, and 100% specificity, i.e., no false positives. The (0,1) point is also called a perfect classification. Whereas, a completely random guess would give a point along a diagonal line from the left bottom

[3]This figure is obtained from the following website http://en.wikipedia.org/wiki/Receiver-operating-characteristic

to the top right corners. In case of good classifier, points are above the diagonal line. Whereas, if the points are below the diagonal line, then the classifier is considered as poor classifier. The Area Under the ROC Curve (AUC) is a numeric performance evaluation measure directly associated with an ROC curve. Larger the value of AUC (Area Under the curve)the better the model. The value of the AUC ranges from 0 to 1. [Ian H. Witten and Kaufmann, 2005].

A classifier who provides a large area under the curve is preferable over a classifier with a smaller area under the curve. A perfect classifier provides an AUC that equals 1. The advantages of the ROC analysis are its robustness toward imbalanced class distributions and to vary and asymmetric misclassification costs [Provost and Fawcett, 2001]. Since, in case software fault prediction model, the class distribution is skewed. Therefore, ROC is particularly well suited for software fault prediction tasks.

ROC-Area (Area Under the ROC Curve or AUC): All the metrics (like precision, recall, etc.) are, point measures, i.e. they are based on the TP, FP, TN, FN that corresponds to a single threshold on the probability of the positive class (i.e. the default threshold of 0.5). However, ROC area is not a point measure. ROC area is a ranking metric, that is computed by considering all the test instances ranked according to the probability assigned to the positive class by the classifier. WEKA computes AUC for each class by considering each in turn to be the *positive* class and all the remaining classes are the negative class.

Kappa Statistics (KS): Kappa is a chance-corrected measure of agreement between the classifications and the true classes. It is calculated by taking the agreement expected by chance away from the observed agreement and dividing by the maximum possible agreement. A value greater than 0 means that classifier is doing better than chance. KS is a means of classifying agreement in categorical data [Sands and Murphy, 1996].

$$KS = \frac{P(A)-P(E)}{1-P(E)}$$

Where P(A) is the proportion of times the model values are equal to the actual values and P(E) is the proportion of times the model values are expected to agree with actual values by chance. A KS value of 1 means a statistically perfect modeling, whereas a 0 means every model value was different from the actual value.

3

Mining Bug Fix Effort Data to Construct Effort Estimation Models for OSS

The contents of this chapter has been published in the papers [Ahsan et al., 2009b] and [Ahsan et al., 2010].

The main function of the software maintenance is to keep the software alive by performing three major tasks, i.e. customer support, update documents and perform changes in the source code. The changes in the source code are required to remove faults, or to enhance existing features or to add new features. Efforts are made to remove faults or to add new features in any software. Whereas, the timely completed maintenance task makes it possible to deliver the product in time, which is the key requirement of the today's software industry and has to be based on an accurate estimation model.

There are several advantages of having an effort estimator. First, it provides initial knowledge about the complexity of the product. Second, it may be used to obtain the cost of the product. Moreover, an effort estimator allows for task assignment and resource management. Different approaches are used to establish estimation models, like algorithmic, analogy, hybrid and machine learning. Initially algorithmic estimation methods were used for software estimation, like Boehm's constructive cost model COCOMO [Boehm, 1981] and COCOMO II [Boehm et al., 1995], Albrecht's function point method [Albrecht and Gaffney, 1983] and Putnam's software life cycle management (SLIM) [Putnam, 1978]. These are based on historical data of effort. These approaches involve the construction of mathematical models from empirical data.

Recent Trends: Jai Asundi [Asundi, 2005] discussed the importance of effort estimation models for OSS. The author identified that the current effort models are inadequate. Consequently, there is a need for new effort models in the context of OSS development. In this regards, the author outlined some issues that should be taken into consideration when developing an effort estimation model for OSS. In particular, Asundi suggested an Activity Based Cost type model for effort estimation.

Motivation: It may be observed that most of the research work on effort estimation models is related to the closed-software system [Albrecht and Gaffney, 1983, Boehm et al., 1995], while very few attempts have been made to develop an effort estimation model for OSS [Koch, 2008b, Yu, 2006a, Weiss et al., 2007]. Reasons may be the inherent complexity of OSS, the large number of software developers or contributors and the absence of effort data. The role of previous history of

effort data is indispensable to build an automatic effort estimation model. But unfortunately most of the Open Source Software does not maintain effort data. Whereas, the recent trend in software engineering shows that even commercial organization like IBM and SUN have developed OSS products, or have made the code of their proprietary product publically available. The consequences of this trend may impact various software engineering methodologies and project management activities. Especially planning and delivery for OSS will be one of the greatest challenges [Asundi, 2005].

Our Goals: In this chapter we employ the idea of mining effort data to answer three research questions:

1. How can we mine effort data from the history of developer's bug-fix-activity and construct the developer's activity log-book?
2. How can we use the bug-fix-activity data to estimate developer's contribution measures?
3. How can we build an effort estimation model using mined effort data?

Our Approach: In order to tackle these three research questions, we present an approach to mine effort data from software repositories and use them to build an effort estimation model for OSS. For this purpose, we propose to construct a developer activity log-book to manage the development activities of developers and contributors. We assume that the actual time spent to fix a bug is an effort. Furthermore, we describe and elaborate different applications of bug fix effort data.

Our Contributions: The work present in this chapter comprises the following contributions:

1. We propose a method for building a developer's log-book that maintains the developer's bug fix activity records. We use the log-book data to calculate the actual time spent to fix a bug and assume it as an effort.
2. We also show how to discover the expertise level of developers from their bug fix activity data and use these information to define a set of developer contribution measures.
3. We developed statistical and ML based regression models for effort estimation. We, used different evaluation techniques to measure the performance of the developed estimation models.
4. We present empirical results obtained when applying our method for effort estimation on the software evolution data of Mozilla project.

Chapter Layout: The rest of the chapter is organized as follows: In Section 3.1 we describe our approach and explain that how we obtained the data from the software repositories. Furthermore, we explain the used program file change metrics. In Section 3.2 we describe the different application of effort data in OSS. In Section 3.3 we discuss the estimation models and analyze the obtained results. In Section 3.4, threats to validity is described. Finally, in Section 3.5 we summarize this chapter.

3.1 Our Approach

To develop an effort estimation model, we use a set of code and process metrics as estimators (or independent variables) and the bug fix effort data as a dependent variable. Whereas, we obtain the

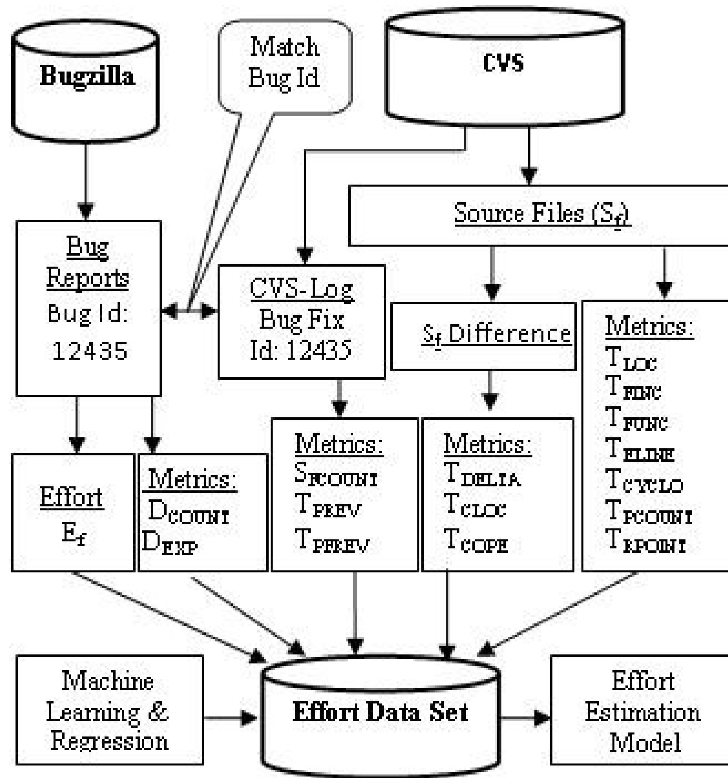


Figure 3.1. Classification Approach

bug fix effort data from the developer activity log book. To obtain the log book, we process the developer bug fix activity data. To perform experiments, we selected the Mozilla open source project and downloaded bug fix reports along with bug fix activity data from the corresponding bugzilla server. We also downloaded source files revisions and CVS log data from CVS repository. Figure 3.1 depicts the overall process involved in extracting the data from software repository.

3.1.1 Obtaining Data From Software Repositories

To perform the experiment, we selected the Mozilla open source project as our source for the version control system and the bug database. For this purpose, we downloaded developers' bug fix activity data where the bug activities have been reported and fixed between Nov-1999 and Dec-2007. We also obtained the list of bug ids from the MSR 2007 data repository [Weiss et al., 2007]. We used the wget tool to extract the data from the Bugzilla web site. We preprocessed the downloaded html files and extracted the developers' bug fix activity data. Furthermore, we connected to the Mozilla CVS server and downloaded CVS log data and source file revisions. Unfortunately, in case of OSS bug databases are not directly linked with a version control system. Therefore, there is no direct mapping between the fixed bug reports and the name of the fixed/updated source files. However, in recent years, a number of techniques have been developed to relate bug reports to fixes in source files. To establish a link between a bug report and the list of fixed source files, we used an approach similar to M. Fischer's approach [Fischer et al., 2003]. Once, we obtained all the required data. We used this data to build an effort estimation model as explained in the next subsection in detail.

3.1.2 Obtaining Metrics Data

For obtaining the metrics, we downloaded the selected revisions of the C++ program files and bug reports from the Mozilla CVS and bugzilla repositories and stored them on a local disk. In the following paragraph, we describe the process which we used to extract the metrics data. The complete list of metrics with a description is shown in Table 3.1.

First we identify those revisions of program files where bugs were fixed. To accomplish this task, we parsed the CVS log comments of each source file revision. If the comment contains a word like Bug, Fix or Fixed followed by some integer value, which is similar to any of the existing bug reports id. Then it means that revision is a bug fixed revision. After identifying all the bug fixed revisions, we extracted all those lines of code from bug fixed revisions, which have been changed to fix the bug. To perform this task, we take the differences between two consecutive revisions, i.e., the revision where the bug has been fixed and its immediate predecessor. Program files difference data are further processed to obtain a set of metrics related to a program file changes, i.e. T_{CLOC}, T_{DELTA} and T_{COPE}. We also processed all those program file revisions, which are the immediate predecessor of the bug fix revisions and extracted the set metrics related to whole program file, i.e. T_{LOC}, T_{FUNC}, T_{ELINE}, T_{FINC}, T_{CYCLO}, T_{PCOUNT}, and T_{RPOINT}. We also obtained the total number of previous revisions T_{PREV}, the total number of program files, which are changed to fix a bug, i.e. S_{FCOUNT} and the total number of previous bug fix revisions T_{PFREV} of each bug fixed revision of a source file.

Finally, we processed bug reports and developer's activity data which are related to the bug fixing activities, and obtained the two measures i.e. developer counts (D_{COUNT}) and developer's expertise (D_{EXP}). The D_{COUNT} is the count of developers or contributors who were involved in fixing a bug. The D_{EXP} is obtained by adding the rank values of all those developers who were involved in fixing a bug. We rank the developers according to their number of bugs fix count. To perform this task, we downloaded 93,607 bug reports together with the bug fix history from the bugzilla repository.

3.1.3 Role of Developers and Contributors in OSS

The main development difficulties in OSS systems are the managing and monitoring of development activities. In OSS development environments thousands of software contributors provide manpower for developing the OSS system. Usually, the contributors are spread throughout the world, spend tremendous amounts of time and effort in writing and debugging software, and most often with no direct monetary rewards [Joseph, Dec 2001]. In OSS development developers are those who have the right to commit the changes in the version control system, whereas the software contributors have no such rights. Instead the contributors are allowed to submit the solution in the form of patches to the developers. Usually, there are a small number of developers compared to the number of contributors [Alonso et al., 2008]. When a bug is reported, normally developers (who are working as software quality assurance personals) validate the reported bug, and assigned it to any appropriate contributor. Alternatively, sometimes contributors voluntarily take a challenge to fix new reported bugs. After resolving the bug, contributors submit patches providing a solution. These patches are then validated and committed by developers into a version control system. These bug-fix-activities are stored into a version control system and a bug tracking system. Because of this process of bug

Table 3.1. List of Metrics

Metrics	Description
S_{FCOUNT}	Number of source files which are changed to fix the bug.
D_{COUNT}	Number of developers who are involved in fixing a bug.
D_{EXP}	Developer's Expertise
T_{PREV}	Total Source File Age: Adding all the previous revisions of the source files which are involved in fixing the bug.
T_{PFREV}	Total previous fix revisions of source files.
T_{LOC}	Total line of source code.
T_{CLOC}	Total changed line of code.
T_{DELTA}	Total number of change location in source files. Delta is change hunk pair, we have at least one delta whenever a file is changed.
T_{FINC}	Total number of included source files/packages.
T_{FUNC}	Total number of function.
T_{ELINE}	Total number of executable lines.
T_{CYCLO}	Total cyclomatic complexity metrics.
T_{PCOUNT}	Total number of parameter count.
T_{RPOINT}	Total number of return points.
T_{COPE}	Total number of changed operators.

fixing one can argue that a person committing a change is not necessarily an expert. It reveals that in case of OSS measuring effort and expertise is not trivial. In the following section, we describe the data extraction process and describe how we used the extracted bug-fix-activity data to construct an effort estimation model.

3.1.4 Mining Bug Fix Effort Data

The conventional maintenance effort estimation process involved three steps. In the first step, it is required to extract maintenance effort data along with other related measures from the previous maintenance records. Then in the second step, it is required to build the model using the data obtained in the first step. While, in the third and final step the model may be used to predict the future maintenance effort [Yu, 2006a]. Unfortunately, most of the OSS system does not maintain actual effort data and consequently, it becomes more difficult to develop an accurate effort estimation model. However, software repositories contain a lot of maintenance related data. In the previous section, we have described in detail how we have extracted all the maintenance related measures from these repositories. Now in this section we describe our approach and method which we have used to extract the actual bug fix effort data from a bug repository.

The most frequently used maintenance effort measure is the total number of man-hours required to accomplish the maintenance task. Therefore, we focus to extract the actual time spent by developer to fix the bugs, and we considered that time as an estimated actual bug fix effort.

Bug reporting systems are used in open source software to store the software maintenance records.

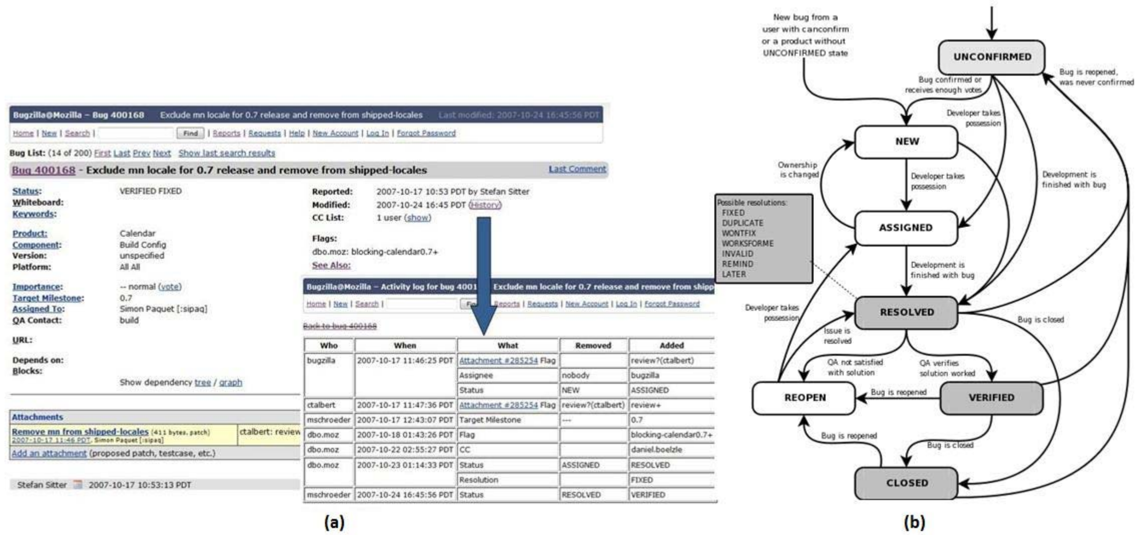


Figure 3.2. (a) A GUI interface of Bugzilla bug tracking system. (b) A complete bug life cycle of Bugzilla, each rectangle represents a unique status of a bug.

Mozilla (http://www.mozilla.org) and lots of others OSS uses Bugzilla (http://bugzilla.mozilla.org), as a bug tracking system. Bugzilla maintained the complete history of a bug life cycle of all the reported bugs. Each reported bug in bugzilla has a complete life cycle. Figure 3.2 shows the bug life cycle. According to this life cycle, a bug starts as UNCONFIRMED. It immediately moves to the status NEW, after that the bug is validated by the quality assurance (QA) person. Then it is moved to ASSIGNED status, which is then followed by the RESOLVED status. Finally, a bug may reach a status of VERIFIED, which is then followed by CLOSED. In some cases after the VERIFIED status, a bug may go to REOPEN status, and the cycle is repeated.

Let us now go into more detail of the bug life cycle. If a bug is said to be NEW, the QA person assigns the bug to any relevant developer/contributor in order to find a solution. Hence, the real work for fixing a bug starts when a bug is moved to the ASSIGNED status. The bug is solved when the developer or contributor provides a solution and moves the status to RESOLVED. The duration between ASSIGNED and RESOLVED is the actual period where effort spent on source code changes to fix the bug. Hence, we assume that this time period is the actual bug fix time and thus the only time to be considered as an effort. Note that in some cases a QA person reassigns the same bug to another developer or in some cases the previously assigned developer assigns the bug to some other developer, which makes the computation of the overall effort even more difficult. Since no information regarding the distribution of effort among the different developers is available, we assume that all developers contribute. Thus we calculate the sum of all time periods for each developer or contributor to come up with a single total bug fix effort.

For a single developer, we compute the effort required to provide a solution to a bug report as follows: We start with the bug reports assigned to the developer. Subsequently, we compute the effort assigned to a bug report for each month of the year using the given dates for ASSIGNED and RESOLVED. The time span between ASSIGNED and RESOLVED cannot be directly used to

Table 3.2. Classification of bug reports on the basis of bug severity level and developer defined bug priority.

Bug Severity Level	**Severity Weight (SW)**	**Number of Bugs**	**Avg. Bug-Fix Time (days)**	**Std. Dev Bug-Fix Time**	**Priority (Assigned by Developer)**	**Severity Factor**
blocker	7	412	11	0.4434	P1,P3	1
critical	6	1413	20	0.2660	P1,P2,P3	1
major	5	1071	103	25.362	P1	1
normal	4	7005	45	0.6571	P3,P4	2
trivial	3	152	58	28.388	P4	3
minor	2	314	50	25.959	P4	3
enhancement	1	326	119	31.015	P4,P5	3

compute the effort. The reason is that a developer works on several bug reports in parallel, but it is impossible to spend more than all days of a month in working. Hence, the time spent has to be multiplied by a factor. This factor takes into account the limited number of days available for working within a particular month. Therefore, we have to multiply the duration of bug fix with a common multiplication factor (M_k). We obtain the multiplication factor by dividing the total working days of a month (T_w) with the sum of all the assigned working days for all the assigned bugs, i.e.,

$$M_k = T_w \;/\; \sum \text{Days} \tag{3.1}$$

$$E_i = \sum \;(\text{Bug fix duration for bug}_i)_k \;\times\; M_k. \tag{3.2}$$

Where, E_i is the estimated bug fix effort of bug_i, which is the sum of the bug fix duration that is spent in k months. If a bug is fixed in more than one month, then the bug fix duration for the first month is obtained by subtracting the bug assigned day from the last day of the month, and for the last month, the bug fix duration is obtained by subtracting the first day of the month from the bug resolved day. We consider the whole days of a month as bug fix duration for all of the intermediate months. We multiply each month's bug fix duration with the respective multiplication factor. Finally, we add all the obtained values to get an estimated time spent to fix a bug, which we assume as the estimated effort value. But, during this experiment we found that the bug severity level affects a bug fix time. Therefore, we further improve our effort estimation model by taking into account the bug severity level. The bug severity level is defined by a user at the time of submitting a new bug to the bug database. On the basis of severity level, a developer assigns a priority level to each assigned bug report. In our experiment, we found that high severity and high priority bugs are fixed in less time compared to low severity and low priority bugs. The average bug fix time related to different severity level is listed in Table 3.2. From the data of Table 3.2, one may conclude that the bug fix time is dependent on the severity level. In the following paragraph, we elaborate the relation between a bug severity level, and bug priority levels.

In case of OSS, developers or contributors do not treat bugs equally. Rather bugs are treated on the basis of their severity level. In case of the Mozilla project, bug reports have seven different

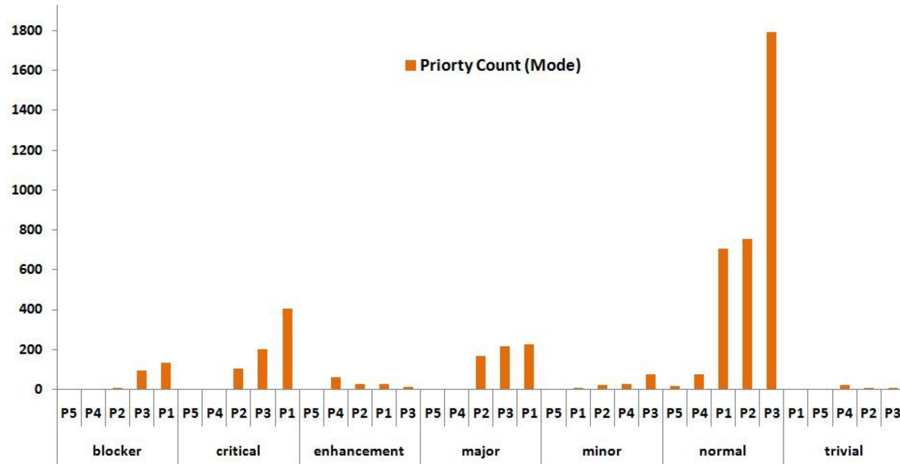


Figure 3.3. The frequency distribution of different priority levels i.e., P1, P2, P3, P4 and P5, which are assigned to different bug severity level.

severity levels, namely critical, blocker, major, normal, trivial, minor and enhancement. At the time when a user reports a new bug, an appropriate severity level is associated with the bug report. Before fixing any bug report, developers, first assign a priority level to the bug report. In case of Mozilla, there are five different type of priority level, i.e., P1, P2, P3, P4 and P5 [Herraiz et al., 2008]. During the experiment, we found that it is not necessary for a developer to assign any fix priority to any severity level. However, it should be mentioned that a developer not necessarily treats a bug of critical or blocker severity level at a high priority i.e., P1 (see Table 3.2). Table 1 shows that developer assigned different priority levels to the same severity level. Hence, there is no one-to-one mapping between severity and priority levels. Figure 3.3 shows the frequency distribution of different priority levels associated to each severity level. A plot comparing the bugs of different severity level and their bug fix time is shown in Figure 3.4, which depicts that less time is spent to fix high severity level bugs.

In order to find the impact of bug severity level on a bug fix effort, we determine the correlation between the bug severity levels and a set of source file metrics. We use those set of source file's metrics, which measure the degree of complexity in source files. The results are shown in Table 3.3, which shows that there is no significant correlation between the bug severity levels and the source file metrics. Therefore, we may conclude that bug severity levels are not related to the complexity of source files. However, developers spend less time to fix high severity level bugs, not because they are less complex and therefore, need less time or effort to fix. But in fact high severity level bugs should be fixed in short time. Similarly, less severity bugs, which take a longer time to be fixed, are not necessarily less complex. In order to treat all of the bugs equally, and to nullify the impact of a biased attitude of developers from the bug fixing time, we divided the estimated effort, obtained in Equation 1 by the time scaling factor Severity Factor (SF_j). Where, SF_j has three values, i.e., SF_1 =1 for critical, blocker and major severity level, SF_2 =2 for normal severity level, and SF3 =3 for trivial, minor and enhancement severity level (see Table 3.2).

$$E_i = [\sum \text{(Bug fix duration for bug}_i)_k \times M_k]/SF_j. \tag{3.3}$$

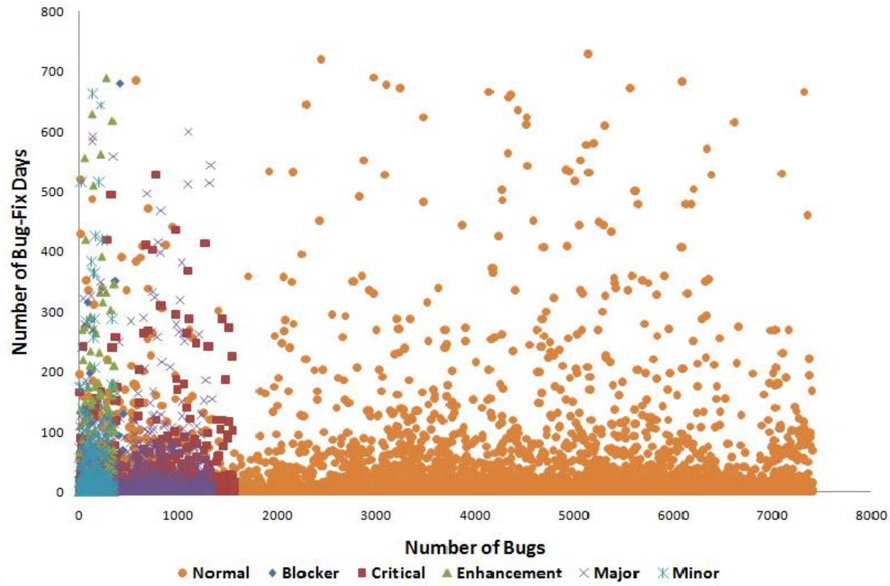


Figure 3.4. A plot between the bugs of different severity level and the number of days spent to fix them. Each bug is represented by a dot where the color of the dot represents its severity level.

Table 3.3. Person and Spearman's Correlation of bug severity level with a set of source file metrics.

Corr. of Bug-Severity	**Total Pointers**	**Total Logical Operator**	**Total Relational Operator**	**Total Function Call**	**Total Assert**	**Total Arrays**	**Total Function Declaration**
Pearson	0.026	0.031	0.026	0.014	0.031	0.004	0.018
Sig. level	0.068	0.038	0.065	0.205	0.036	0.419	0.154
Spearman's	-0.033	0.002	-0.036	0.009	0.035	0.001	0.014
Sig. level	0.029	0.450	0.018	0.293	0.022	0.490	0.205

Developer Name: X Month: February

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28

Bug Id= 1, Bug fix duration 06 days

Bug Id= 2, Bug fix duration 10 days

Bug Id= 3, Bug fix duration 21 days

Bug Id= 4, Bug fix duration 10 days

Total working days T_w = 24 days

Figure 3.5. Classification Approach

To understand, how we estimate the bug fix effort from bug reports, consider an example, in which a developer fixed four bugs in February, 2008. All of the bugs were critical, i.e., $SF_{ij} = 1$. The example data is shown in Figure. 3.5. The gray bar represents the durations in which the developer fixed those bugs. We obtain these durations from bug reports by subtracting the bug assigned date from the bug resolved date. In this example, we assume that during some days of the month, the developer worked in parallel on multiple bugs, now we apply our model to estimate the effort value.

$$M_k = 24/47 = 0.51$$

Now, the bug fix effort (E_i) for bug i, and k is the number of months spent to fix a bug. In this example k=1.

$$E_1 = (6 \times 0.51)/1 = 3.06$$

Similarly, we estimate the effort for other bugs, E2=5.1, E3=10.7, and E4=5.1. We built a program that automatically performs all of the mentioned steps to obtain the effort value and store it into a database.

3.2 Application of Developer's Bug Fix Activity Data in OSS

In the previous section, we described our approach to mine effort data from the history of the developers' bug-fix-activities. In this section we describe some applications of developer bug-fix-activity data.

3.2.1 Developer Activity Log Book

In commercial software organization, different tools and techniques are available to monitor and control the developer's monthly development activities. One of the techniques is to store and maintain each developer's activity logs in the developer's log-book. The developer activity log provides past and present development history related to the developer. In OSS there is no such system, which logs the developer or contributors development activities. However, this information can be mined from the developer bug fix activity records stored in bug tracking systems. The developer's log-book contains useful data, which can further be processed intelligently to obtain the following information.

1. Development work done by each developer in any month of a year. This may be used to analyze the developer's performance and the development load on each developer in any month of a year.

Figure 3.6. An example of developer bug fix activity log-book

2. Time spent by each developer to resolve the assigned development task. This may be used to analyze the developer efficiency and expertise.

3. Total time spent by one or more developers to fix a bug. This may be considered as bug fix efforts in days.

We use the log-book data to calculate the bug fix effort data. The Log-book contains the complete history of each bug and the name of developers whoever have been involved in bug fixing. An example of the automatically generated log-book is shown in Figure 3.6. Besides its main advantage of providing the effort data, there may be several other advantages of log-books. For example, a log-book may be used for the analysis of developer's activity patterns. It is shown in the pop up window of Figure 3.6 that the developer Neil is more active during the last months of the year 2001 as compared to the initial months of the same year. Similarly, we can use this log-book to analyze the month wise or year wise effort distribution of developers.

For the Mozilla project, we used the developer's activity log-book data to obtain the average bug fix efforts in days. We added all the bug fix days during any year and divided it by the number of bugs fixed in that year. Figure 3.7) shows a comparison between the total number of bugs fixed in a year and the corresponding average bug fix efforts. It is observed from that figure that during the initial years of the project, small effort was required to fix a large number of bugs. However, as the time advances the count of bug fixed per year decrease, while the average effort to fix bug increases. Figure 3.8) shows the relationship between the bug fix days with the frequency of bug's occurrence in that period. It is observed that for Mozilla project, the frequency of the bugs that needs small effort (actual bug fix days) is much larger as compare to those which need more efforts.

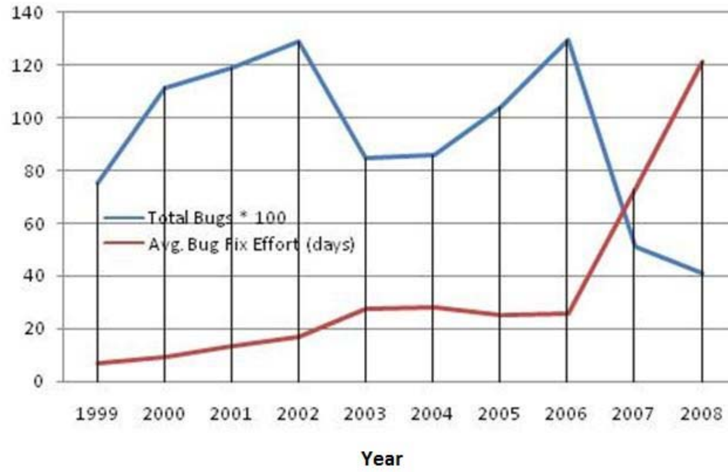


Figure 3.7. Mozilla project average bug fix effort per bug related to the number of bugs fixed per year.

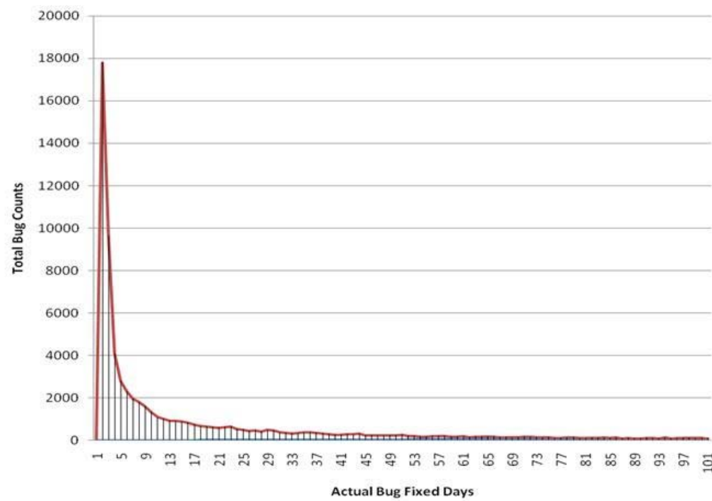


Figure 3.8. Number of bug counts as function of the number of bug fix days.

Developers	DCM-NBFixed	DCM-WNBFixed	DCM-NFFixed
bzbarsky%mit.edu;	1730	7072	490
roc+%cs.cmu.edu;	883	3650	192
jst%netscape.com;	717	3179	282
karnaze%netscape.com;	716	3312	125
cbiesinger%web.de;			
bryner%brianryner.com;			
waterson%netscape.com;			
aaronleventhal%moonset.net;			
timeless%mozdev.org;			

Developer	BugSeverity	SeverityFactor	BugFixed	DCM-WNBFixed
bzbarsky%mit.edu;	blocker	7	16	112
	critical	6	161	966
	major	5	132	660
	normal	4	1256	5024
	minor	3	42	126
	trivial	2	61	122
	enhancement	1	62	62

Figure 3.9. A list of top 10 Mozilla developers on the basis of the contribution metrics. The pop up window shows the details of the metric DCM-WNBFixed.

3.2.2 Developer's Contributions Measures to Identify Expertise

During the evolution of large software systems, it is essential to identify those individuals who are expert in some specific part of the product. The current approaches for expertise recommender systems are mostly based on variations of the Line 10 Rules. According to the Line 10 Rule, developers who changed a file most often do have the most expertise for implementation. The name Line 10 Rule is come from a version control system that stores the author who did the source file change commit, in line 10 of the commits' log message [Schuler and Zimmermann, 2008]. In short, developers who commit changes in source files are considered to have expertise in changing those source files. In order to identify the expertise level, we need to measure the developer's contributions during the evolution of software. One can discuss the contribution in different contexts. However, a contribution is commonly related to the developer's productivity or developer's activities [Gousios et al., 2008]. In our experiment, we used the developers' bug fix activity data to measure a set of developers' contribution metrics. These contributions metrics are used to list the name of expert developers.

We propose three different metrics for measuring a developer's contributions. Let DCM stand for Developer's Contribution Measure. The first metric DCM-NBFixed, is the sum of bugs fixed by a developer. The second metric, i.e., DCM-WNBFixed is the weighted sum of bugs fixed by a developer. The third metrics DCM-NFFixed is the sum of source files fixed by a developer. The weighted bug fixed is obtained by multiplying each fixed bug with a weighting factor. The weighting factor (SW) represents the severity level of the fixed bug. The severity level was defined in Section 3.3 (see severity weight column of Table 3.2)). The formal representations of developer's contribution metrics are given below,

$$\text{DCM-NBFixed}_i = \sum\nolimits_j \text{Bug-Fixed-Id}_i(\text{j}) \tag{3.4}$$

$$\text{DCM-WNBFixed}_i = \sum\nolimits_j \text{Bug-Fixed-Id}_i(\text{j}) \times \text{SW(j)} \tag{3.5}$$

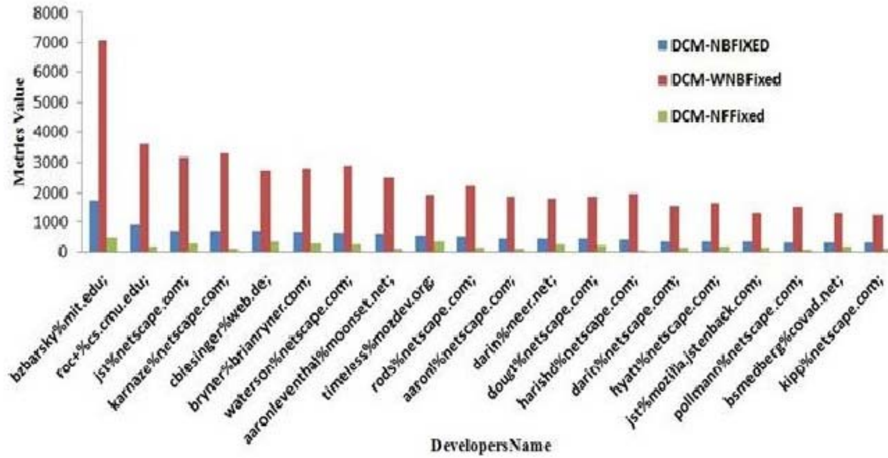


Figure 3.10. Distribution of five different contribution metrics of the top 20 Mozilla developers.

$$\text{DCM-NFFixed}_i = \sum{}_j \text{ Fixed-Source-File-Id}_i(\text{j}) \tag{3.6}$$

Where, i represent developer's id, and j represents bug id. In order to obtain the defined set of contribution measures. We further processed the developer activity log-book, and obtained a metrics value for each developer. We performed this experiment on a sample data set comprising 10,693 bug reports. 410 different developers fixed these bug reports. Figure 3.9, shows the contribution measures of the top 10 developers. Figure 3.10 depicts the metrics distribution of the top 20 developers, and Figure 3.11 shows the relation between the percentages of bug fixed and the developer who fixed the bugs. It is clear from the figure that only 4.9% developers fixed 71.5% bugs. In our experiment, we found that the most expert developer in Mozilla project is 'bzbarsky@mit.edu'.

3.2.3 Visualization of Developer Bug Fix Activity

Software evolution needs to handle a huge amount of data, which is the major problem and a strong reason for doing research in this field. The huge amount of data comes from analyzing several versions of the same software in parallel. A technique, which may be used to reduce this complexity, is software visualization that allows researchers to study multiple aspects of complex problems in parallel [Lanza and Ducasse]. In our experiment, we figured out that the developer's bug fix activity data could be used to visualize the developer's interactions. Therefore, we developed a visual tool using Java API. This tool may be used to explore interesting patterns among the developers and the corresponding source files. We use the bug fix activity data to visualize the interaction between two or more developers, and the interaction of developers with their assigned source files. The main features of our visualization tool are shown in Figure 3.12 and Figure 3.13 . The upper part of the figure shows the connection between developers. Each circular node represents a developer.

The size of the circle represents the developers total bug fix activity. The number inside the circle is the developer's id. The lower part of the figure represents the relationship between source files and developers. Squares represent source files, and circles represent developers. If developers have

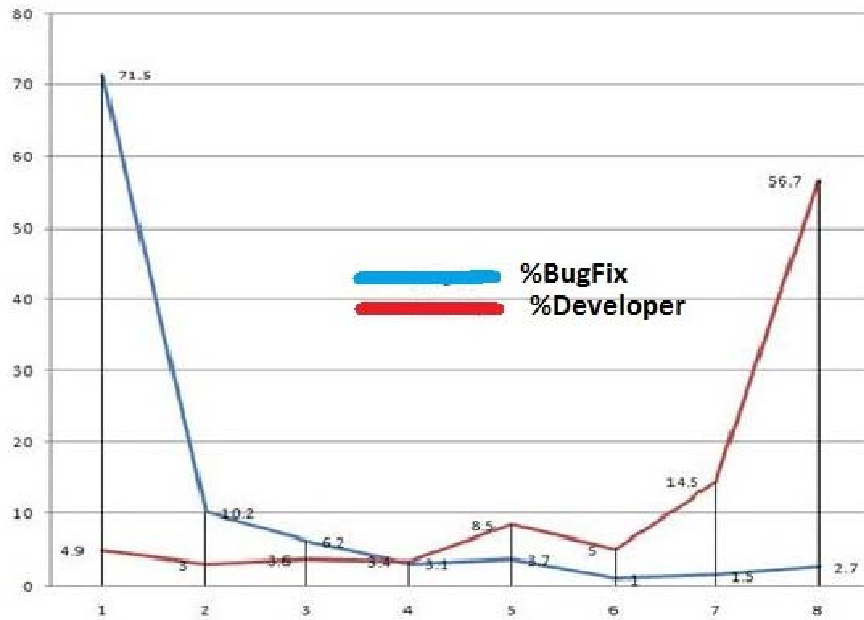


Figure 3.11. Plot between the percentage of fixed bugs and the percentage of developers who fixed them.

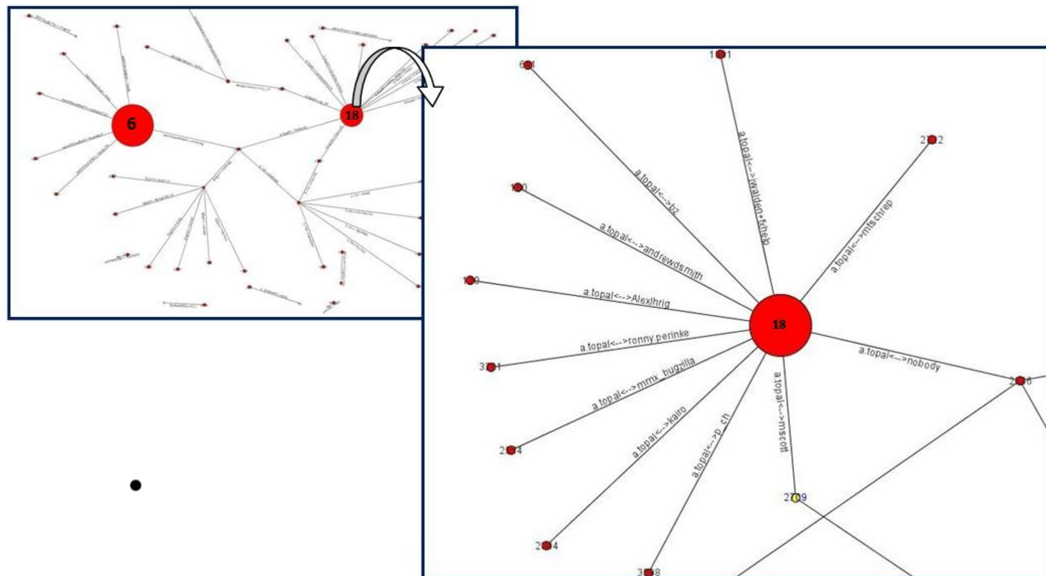


Figure 3.12. A visualization of the developer's bug fix activity data. Interaction among developers.

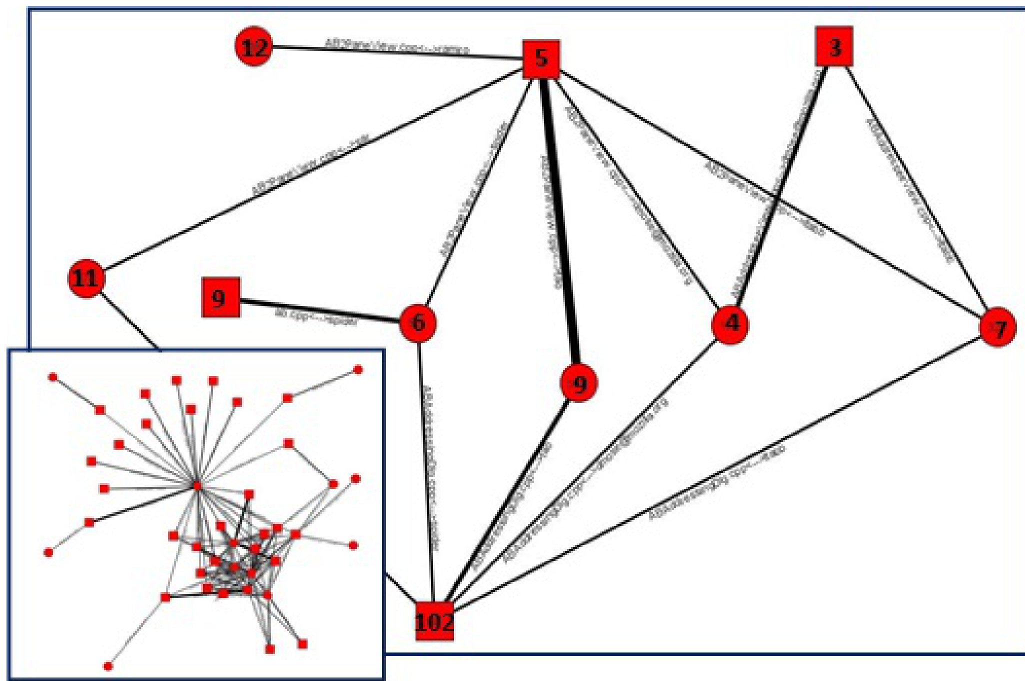


Figure 3.13. Interaction between source files and developers.

performed one or more changes in any source file, then there is an edge between the developer and the source file in the graph. The thickness of the edge represents the number of times a developer has changed a source file. The label attached with each link depicts the names of connected developer and source file. The number inside the square is the source file's id. Developers are represented by red circles, and the source files are represented by red squares. The different size of the circle represents the developer bug fix activity count. From the analysis of Figure 3.13 and Figure 3.13, we can conclude that in case of Mozilla very few developers have fixed large number of bugs, whereas a majority of the developers has fixed very few bugs. Another interesting finding is that some source files are connected with a large number of developers. This means that multiple developers have changed these source files. Therefore, these types of source files are risky to modify.

3.3 Effort Estimation Model

One of most important contribution of our research work is the development and comparison of different models for extracting an effort estimator from the obtained metrics and effort data. To build the model five different regression based machine learning algorithm have used. These models are trained and tested using metrics and effort data which we have obtained in section 3.1.4. Finally models are evaluated using five different evaluation criteria of machine learning.

3.3.1 Regression Based Machine Learning Algorithm

Supervised Machine Learning considers the problem of approximating a function that gives the value of a dependent or targets variable y (in our case effort), based on the values of a number of independent or input variables (in our case metrics) x_1, x_2,...., x_n. If y takes real values, then the learning task is called regression, while if y takes discrete values then it is called classification. Since, in case of a prediction or estimation model one has to predict or compute a real value. Therefore, a regression based models are used to construct estimation models. In the following, we list the name of regression based statistical and machine learning algorithm used in our experiments:

1. Multiple Linear Regression
2. Support Vector Machine (SVMreg),
3. Neural Network (Multilayer Perceptron),
4. Classification Rule (M5Rules)
5. Decision Tree (REPTree) and
6. Decision Tree (M5P).

MLR is a statistical method commonly used for numerical prediction. We used MLR because it is a simple tool and most commonly used for effort estimation purpose [10]. Support vector machines (SVM) belong to the class of supervised learning methods, which are used for classification and regression. SVM algorithms are also used for regression or numerical prediction. One example is SVMreg. They generate a model that can be expressed in terms of a few support vectors and may be used in non-linear problem solving using the kernel function. Neural networks are commonly used for numerical prediction and effort estimation. We used *multilayer perceptrons*, which belongs to the *feedforward* class of networks. In *feedforward* type of network the output only depends on the current input instances and there are no cycles. M5Rules generates a decision list for regression problems. It builds a model tree using M5 and makes the best leaf into a rule. Decision trees classify input instances by sorting them according to feature values. Each node in a decision tree represents a feature, and each branch represents a feature value. REPTree is a decision tree based machine learning algorithm. It is a fast tree learner who uses reduced error pruning. M5P is the model tree learner. Model tree combines decision tree with the possibility of linear regression function at the leaves. This algorithm is used twice, one for the production of a model tree and one for the production of a regression tree [Ian H. Witten and Kaufmann, 2005].

For the comparison, we use the data obtained from the Mozilla project repositories. Moreover, the comparison is based on the following error measures. The mean absolute error (MAE), root mean square error (RMSE), the mean relative absolute error (MRE) and the mean magnitude of relative error (MMRE), the root relative square error (RRSE), the Pearson correlation coefficient R, and the percentage of prediction PRED(x). The formal representation and further details related to all the above mentioned model evaluation measures are given in Chapter 2 (see 2.4).

To perform the experiment we used freely available Java based machine learning tool known as WEKA [Ian H. Witten and Kaufmann, 2005]. WEKA contains a large number of Java libraries of

machine learning algorithms.

3.3.2 Multiple Linear Regression Model

To develop an effort estimation model using statistical multiple linear regression (MLR), we used a set of metrics as set of estimators for the model, and effort as a dependent variable. The set of metrics is shown in Table 3.4. Beside the used metrics, Table 3.4 also comprises the Pearson and Spearman correlation of the metrics with the effort. The obtained MLR model is given below,

$$\begin{aligned} E \;=\; & -13.8 \times D_{COUNT} - 1.09 \times D_{EXP} + 5.8 \times S_{FCOUNT} - 0.01 \times T_{PREV} + \\ & 0.02 \times T_{PFREV} + 0.001 \times T_{LOC} + 0.02 \times T_{CLOC} - 0.035 \times T_{DELTA} - \\ & 0.03 \times T_{FUNC} + 0.004 \times T_{ELINE} - 0.025 \times T_{CYCLO} + 0.02 \times T_{PCOUNT} - 12.9 \end{aligned} \tag{3.7}$$

The correlation data in Table 3.4 shows that S_{FCOUNT}, D_{COUNT}, and D_{EXP} are positively correlated with the effort data. The metrics D_{COUNT}, and D_{EXP} are related to developers while SF_{COUNT} is the total numbers of source files, which are changed to fix a bug. Whereas the metrics which are directly related to the source code like line of code T_{LOC}, cycloramic complexity T_{CYCLO}, etc. are positively correlated with the effort value, but their correlation with the effort is not high. It shows that in case of OSS, metrics related to developers are more correlated with the effort as compared to the source code metrics. The obtained Pearson's correlation coefficient value of the model is 0.53. This indicates that the predicted effort value using MLR model is highly correlated with the actual effort values. While R^2 and adjusted R^2 values are 0.289 and 0.291 respectively. The further results of model accuracy estimation are shown in Table 3.5.

3.3.3 Machine Learning Models

In order to develop an accurate effort estimation model, we have analyzed several machine-learning (ML) algorithms using WEKA tool [Ian H. Witten and Kaufmann, 2005]. All the selected ML algorithms belong to the class of supervised learning methods, which are commonly used for classification and regression. To obtain a better model, we used 10 fold cross-validation. It is an important technique to avoid over-fitting models on training data, as over-fitting will give low accuracy on validation. It actually divides the data set into 10 equal parts and randomly selects 9 parts for training and 1 part for testing and repeats it for 10 times. In the following paragraph, we discuss each model and its performance.

To obtain the support vector machine (SVM) based model. We used SVMreg algorithm, and we used the filter data type, i.e. normalized training data. While to obtain a neural network based model, we used multilayer perceptron, which belongs to the feed forward class of networks. We designed multiple neural networks using 1, 2, and 3 hidden layers with 2, 4-2 and 6-4-2 perceptron per hidden layer, and set the number of learning steps between 500-1000. For M5Rules method, we used two classification rules on the basis of the D_{COUNT} metrics value. We also used the decision tree M5P and fast decision tree learner method REPTree with the pruning option. Table 3.5

Table 3.4. Correlation of Metrics With Effort

Sr.	Metrics	Correlation With Effort		Mean	Std. Deviation
		Pearson	Spearman		
1	S_{FCOUNT}	0.32	0.51	01.8	01.6
2	D_{COUNT}	0.41	0.43	02.2	01.5
3	D_{EXP}	0.36	0.36	15.2	10.6
4	T_{PREV}	0.19	0.24	224.4	289.4
5	T_{PFREV}	0.18	0.28	144.1	199.5
6	T_{LOC}	0.20	0.27	2472.0	2799.3
7	T_{CLOC}	0.18	0.30	39.4	85.5
8	T_{DELTA}	0.22	0.23	10.0	20.9
9	T_{FINC}	0.21	0.32	52.8	59.7
10	T_{FUNC}	0.21	0.28	103.3	115.3
11	T_{ELINE}	0.19	0.27	2200.8	2425.9
12	T_{CYCLO}	0.20	0.26	424.1	482.9
13	T_{PCOUNT}	0.21	0.27	179.4	200.1
14	T_{RPOINT}	0.18	0.24	171.3	215.4
15	T_{COPE}	0.13	0.24	29.M,,r	67.4
For all metrics p-value=0.00 at significance level 0.001					

depicts the obtained results. We see that each model has a good correlation value. The highest correlation value is for the classification rule M5Rules i.e., 0.56, while its MMRE value is 74%. In case of SVMreg the correlation value is 0.51, and the MMRE value is 63 %, which is the lowest. Therefore, in our experiment the best model is the support vector machine SVMreg, although the value of MMRE is acceptable, while PRED(0.25) = 0.20 and PRED(0.5) = 0.45, which is not very close to the ideal value. There are several reasons of MMRE value being so high. One may be the presence of noise or outlier in the data set. The other may be the quality of the effort data set because we don't get it from Mozilla project rather we extract it by our own heuristic method. Another big issue with the MMRE is its value strongly influenced by a few very large MRE values [10].

3.4 Threats To Validity

The computation of effort spent to correct a bug described in a bug report assumes that the real effort is distributed evenly. This might not be the case, but since there is no other information available, it is the best we can do. This assumption as well as the others introduce some errors in the resulting data. But given the huge amount of data available in the repositories we expect that there is no error inherently built in this system of computing the effort. Hence, there might be a decrease of reliability in the data but there should always be an upper bound.

We also assumed that developers spend a whole assigned period in fixing the bug, but this might not be the case, because in OSS only some experienced developers are doing as a full paid job, while most of the contributors are volunteers, and they may be involved in some other jobs during bug

Table 3.5. Evaluation Measures of Different Machine Learning Based Effort Estimation Model

Sr.	Method Name	R	MAE	RMSE	MMRE (%)	RRSE (%)	*PRED(x)*	
							0.25	**0.50**
1	Multiple Linear Regression	0.54	11.88	23.5	81.0	84.7	0.09	0.17
2	Support Vector Regression (SVMreg)	0.51	9.36	26.7	63.8	95.8	0.20	0.45
3	Neural Network (Multilayer Perceptron)	0.54	29.2	58.2	93.6	86.3	0.10	0.22
4	Classification Rule (M5Rules)	0.56	10.8	23.0	73.6	82.5	0.18	0.28
5	Decision Tree (REPTree)	0.51	10.8	23.9	74.23	86.0	0.10	0.23
6	Decision Tree (M5P)	0.55	10.6	22.9	77.7	82.5	0.13	0.26

assigned period. Also one cannot completely rule out the existence of any outliers. However, we have almost removed most of them from our dataset.

3.5 Summary

In this chapter, we presented the result obtained from different effort estimation models for OSS system. These models are based on statistical methods as well as machine-learning methods. All the models are based on the same underlying metrics and effort data and trained with the same number of instances, i.e. 7027. Since most of the OSS system does not maintain the bug fix effort data, therefore we developed a novel method for deriving this information from bug repositories. We processed the developer's activity log data and obtained the bug fix effort values in terms of bug fix days. Furthermore, we presented several applications of developer bug fix activity data. The experimental results obtained shows that the regression based statistical and machine learning algorithm can be used to construct effort estimation models. Our obtained models have correlation values between 0.51 and 0.56. Similarly, the MMRE values lie between 63% and 93%. This shows that the performance of our models is satisfactory.

4
Prediction of Risky Program Files

The experimental results discuss in this chapter have been published in [Ahsan et al., 2008].

The term *Software Evolution* is commonly used in the field of software engineering to refer to the process of developing software initially and then subsequently changing it to incorporate the change requirements. According to the Lehman laws of software evolution, software must continually evolve or grow according to user needs or other external factors otherwise it might become useless [Lehman and Ramil, 2001]. This shows that software that is really in use must go for change to become and stay alive.

Mostly changes in the source code are made to add new features or to enhance the existing features of software or sometimes to fix the bugs. The change in source code is an essential and routine activity of development and maintenance. When a programmer makes changes to accomplish a specific change requirement, it is observed that this activity might result in faults that might harm the use of the software. Therefore, it is always useful for software managers and programmers to identify those program files which are risky or at least sensitive to changes. Software developers also need to know the bug introducing change patterns, so that they can avoid program changes from introducing faults. In order to find such patterns or to classify risk, previous knowledge like contained in a software repository is might of help. To obtain the the previous change history data of a program source file, one has to take the difference between the two consecutive revisions of a source file. The difference may give one or more set of source code lines, which are added or modified or deleted in the new revision of a source file. Each set of these source code lines is called *hunk* [Thomas Zimmermann, May 23-28, 2004].

Research Trends: In the recent years several research works have been performed to analyze the different types of changes in the program source files and build models for the classification of program file change data. Porter and Selby [Porter and Selby, 1990] used classification trees based on metrics from previous releases to identify components having high-risk properties. Mockus and Weiss [Mockus and Weiss, 2000] presented a model to predict the risk of new changes, based on historical information. The authors modeled the probability of causing failure of a change made to software, using properties of a change as model parameters. Jack and Zimmermann [Śliwerski et al., 2005b] analyzed the CVS repository of Mozilla and Eclipse together with the information stored in the corresponding Bugzilla bug reporting system to identify fix inducing changes. Livshits and Zimmermann [Livshits and Zimmermann, 2005] combined revision history mining and dynamic analysis techniques for discovering application specific patterns and for finding errors. They discovered 56 application

specific-patterns and found 263 pattern violations by mining history data of Eclipse and jEdit. Kim et al. [Kim et al., 2006b] built bug fix memories by extracting change data from bug fixing changes. They developed a tool named *BugMem*, which learned from previous bug fix change data. They only considered bug fixing changes and left other changes like bug introducing and bug fix-introducing changes. Kim et al. [Kim et al., 2008] introduced a technique for classifying a software change as clean or buggy. The authors trained a machine learning classifier using features extracted from revision history of a software project. The features used, include all terms in the complete source code the lines modified in each change (delta), change metadata such as author, change time, and complexity metrics. The proposed model could classify changes as clean or buggy with 70 percent accuracy and 60 percent buggy change recall on average.

Motivations: The change history (i.e., hunk data sets) of source files are useful to analyze the previous change patterns of source files. The previous change patterns might be used for the analysis of risky program files for new changes.

Our Goals: One of our thesis objective is to extract the software change data from the open source software repository of Mozilla project and build a model for the source code change analysis. The overall goal of our research work present in this chapter is to perform the following tasks,

- Device a method to extract the data of software changes from software repository.
- Construct a model to measure the risk value, which is associated with the new changes in a program file.

Our Approach: We extract the change history data of program files from the software repository of Mozilla Project. We apply an approach on the extracted data that allows us to identify the different types of change transactions like bug fixing, clean, bug introducing, and bug fix-introducing transactions (A change transaction is a process in which developers made changes in the program files and commit those changes into a version control system). Furthermore, we use the probability of bug introducing and bug fix-introducing changes to identify the source file as being risky or not for further changes. We also estimate the impacts of a faulty change in terms of their introducing number of new faults. Finally, to obtain an estimated risk value, we multiply the probability of faulty change with its impact value. In order to validate our approach, we performed an experiment, we downloaded 9552 program source files (C++), extracted the CVS log data, and extracted the Mozilla bugs information from the Bugzilla database.

Our Contribution: Our main contribution is we used the change history of program files, and constructed a risk model. The defined risk model can be used to find the risk associated to the new changes in source files. Such information is not only useful for developers but also for software managers in order to assign resources, e.g., for testing. Furthermore, we performed an experiment by extracting data from the freely available Firefox and Apache project repository.

Chapter Layout: The rest of the chapter is organized as follows. Section 4.1 presents our approach to measure the risk value associated with the new changes in a source file. In Section 4.2 we describe the experimental setup and discuss results. In Section 4.3 we discuss some application of

hunk metrics, and finally in Section 4.4 we present the summary of this chapter.

4.1 Risk of Faults Associated with the New Changes in a Program Source File

The change in source code is an essential and routine activity of development and maintenance. When a programmer makes changes to accomplish a specific change requirement, it is observed that this activity might result in faults that might harm the use of the software. Therefore, it is always useful for software managers and programmers to identify those program files which are risky or at least sensitive to changes.

We used an approach to divide all the previous changes of a program source file into four different types of changes, i.e., clean, bug introducing, bug fix and bug fix-introducing changes [Sliwerski et al., 2005]. Then, count the different types of changes and obtain the probability of source file change, which are related to bug and bug fix-introducing change. Finally, we use this data as input to our risk model and measure the risk value of a source file.

4.1.1 Data Collection

In software systems program files are created to perform some required functionality. During the whole life of any software system, program files are frequently changed by developers to accomplish the required maintenance or development task. A change in a program file is occurred, when developers made a change transaction into a CVS system (version control system). In a change transaction, developer made changes in a single or in multiple source files and commit (save) those changes in a version control system like CVS or Subversion. Therefore, the repository of version control system contains a huge amount of program file change data; the challenge is to properly identify all those changes according to their types.

In the following sub sections, we first describe our approach to extract the data from the software repository. Then we describe our database schema that store all the extracted data into different database entities. Finally, we describe our approach to use the stored data and identify the different types of software changes.

A Database to Store the Extracted Data

One focus of this section is to introduce a database that stores program file change data, program file change meta data together with the existing program change patterns and bug reports data. To accomplish this task first we extracted the data from source code repositories and bug tracking systems and stored into a relational database. Therefore, in order to store the extracted data, we developed an appropriate database schema. The database schema description is given in Table 4.1 and database entity relationship diagram (ERD) is shown in Figure 4.1. In Figure 4.1, the set of tables in light green color are base tables for CVS log and bug reports data. The set of tables in dark

green color are base tables for revision difference and annotation data. While, the set of tables with gray color are used to store the processed data of different base tables. The set of tables with yellow color contains the metrics data related to each revision of source file.

In order to perform the experiment and to populate the database tables, we downloaded all the revisions of initial 1000 Mozilla C++ source files. We selected these files according to the alphabetic order of their files names. The total number of downloaded source files was 9552. Furthermore, we used *wget* tool to download the bug reports data in html format. We used *CVS difference* and *CVS Annotate* command in a script to automatically download source file revisions, difference between two consecutive revisions of source files, and the annotation of each revision of source file. The coding is shown in Script 1.

Script 1 A script to extract the source file difference and annotation data from a CVS server of Mozilla project. Same script may be used to extract data from the CVS repository of other projects.

```
#!/bin/bash
# First login to CVS server
export  CVSROOT=:pserver:anonymous@cvs-mirror.mozilla.org:/cvsroot
cvs login
IFS=','
# Read a text file FileName.txt line by line in which F_COLUMN1 contains file name,
# F_COLUMN2 contains file paths and F_COLUMN3 contains filecode values
FileName="/path/FileName"
while read F_COLUMN1 F_COLUMN2 F_COLUMN3
do
    FLAG=0
    for I in $F_COLUMN2  # for each file path, I contains file path
    do
    FileRevisions="/path/FileRevision"
    # Read another file FileRevision.txt line by line in which R_COLUMN1 contains file
    # paths, R_COLUMN2 contains file revisions and R_COLUMN3 contains filecode values
    while read R_COLUMN1 R_COLUMN2 R_COLUMN3
    do
    for J in $R_COLUMN2 # for each file revisions, J contains file revisions
    do
    if ["$I" = "$R_COLUMN1" ]  #Check file path are same
      then
    DNAME=$R_COLUMN3':'$F_COLUMN1':'$J # FileCode:FileName:FileRevision
       cvs checkout -r $J -d $DNAME $I
       # It will checkout revision J of source file I in a local directory DNAME
       if [ $FLAG == "1" ]
       then
       diff $TEMP  $DNAME'/'$F_COLUMN1  >$F_COLUMN3':'$F_COLUMN1':'$L'-'$J'.txt'
       cvs annotate -r $J $DNAME'/'$F_COLUMN1 > $F_COLUMN3':'$F_COLUMN1':'$J'-anot.txt'
       else
       cvs annotate -r $J $DNAME'/'$F_COLUMN1 > $F_COLUMN3':'$F_COLUMN1':'$J'-anot.txt'
       FLAG=1
       fi
       TEMP=$DNAME'/'$F_COLUMN1
       L=$J
       fi
      done
     done <"$FileRevisions"
   done
done < "$FileName"
```

The downloaded data are source code files, text files or html files. Therefore, to extract the textual information from the downloaded files and to extract the metrics data from the source code, we used

different parsers which extract all the relevant information and metrics data, and inserted into the database entities. The following list of steps describes the key activities, which have been followed during the data extraction process:

1. We logged in to the Mozilla CVS server and obtained CVS log data, which we stored in a text file. We processed the log file to extract CVS log data. The CVS log contains all information regarding the revision of each source file. The bug report data of Mozilla project are available on the Bugzilla website. The bug report is in xml format. We used an approach [Fischer et al., 2003] to parse and extract the bug report data of Mozilla project. We have collected all the bugs reports which have been reported till 2007.

2. We downloaded all the required program files into our local disk from CVS repository, by using a shell script (see Script1).

3. We used a program which parse each revision of downloaded program file, and obtain a set of source code metrics.

4. We used the GNU diff tool to find the difference between two consecutive revisions. To obtain the difference between two consecutive source file we executed a script, which uses GNU diff for all downloaded source file and stores the output in text files. We scanned these text files to extract file difference data.

5. We used the CVS annotation command to find the annotation for each downloaded program file. The output of annotation command is again stored in a text file. We processed this text file to extract all annotation information and insert it into corresponding database table.

4.1.2 Types of Program File Changes

In the previous sub section we have extracted and stored all the relevant data related to the program file changes. Now, in this section we describe that how we use these data to obtain the different types of program file changes.

1. **Bug Fixing Change:** A bug fixing change is a type of program file change in which programmers change a program source file to fix the reported bugs. To identify bug fixing changes there are two popular approaches. In the first approach a link is established between a change log messages of CVS to the bug report data. The link is established when the bug id, which is found in the bug report data, matches with log message of CVS that contains a word *fix* or *bug* followed by the similar integer number, which has found in bug report, see [Zimmermann, 2006]. This type of CVS log message is used in identifying a bug fix. The second approach does not only establish a link but also uses two independent levels of confidence, i.e., syntactic and semantic level of confidence. The syntactic level is used to infer the link between change log messages and bug reports whereas the semantic level is used to validate the link by matching the bug report data, see [Śliwerski et al., 2005b]. We used the second approach and defined each link with syntactic and semantic level of confidence.

Table 4.1. Description of database schema

Project	Contains the project record like Mozilla or Eclipse and other collected data from the CVS log
FileDirectory	Contains file directory information
FileDetails	Contains source file metadata
FileRevision	Contains file revision metadata, like revision LOC, revision id etc
BugReport	Contains bug report data.
FileRevisionDelta	Contains metadata of revisions differences. The difference is between two consecutive revisions of each source file.
HunkData	Contains revision difference change data of source file.
LineAnnotation	Contains the annotation data of each source file.
Transaction	Contains the summary of each change transaction. To obtain the data we processed log comments and identified those changed source files, which are committed by the same developer with the same log message on the same date with small difference of time (less than 200 sec). We used a sliding time window approach Fischer et al. [2003]. We identified all those changes as single transaction and assigned a unique transaction id to each change transaction.
BuggyFiles	To find the bug introducing revision, we process FileRevisionDelta table, and select those line numbers of a revisions, which are changed to fix the bugs. Then we used the LineAnnotation table to find the last revision in which the selected sets of lines were changed. The last revision in which the set of selected lines were changed is the actual bug introducing revision [Sliwerski et al., 2005].
FunctionMetrics	Contains the function level metrics data. To obtain metric data we scanned each revision of a source file
Author	Contains the author name, with author id and author address
ClassMetrics	Contains the class level metrics data. To obtain metric data we scanned each revision of a source file.
BugFixed	Contains bug fix transaction id with bug Id.
Boundary Table	It contains data about how many times a function or method has been changed. If the change line number lies between the boundaries of any function or class, then we consider this function or class as a change function. We scan source file to find the starting and end line number of each function and class.

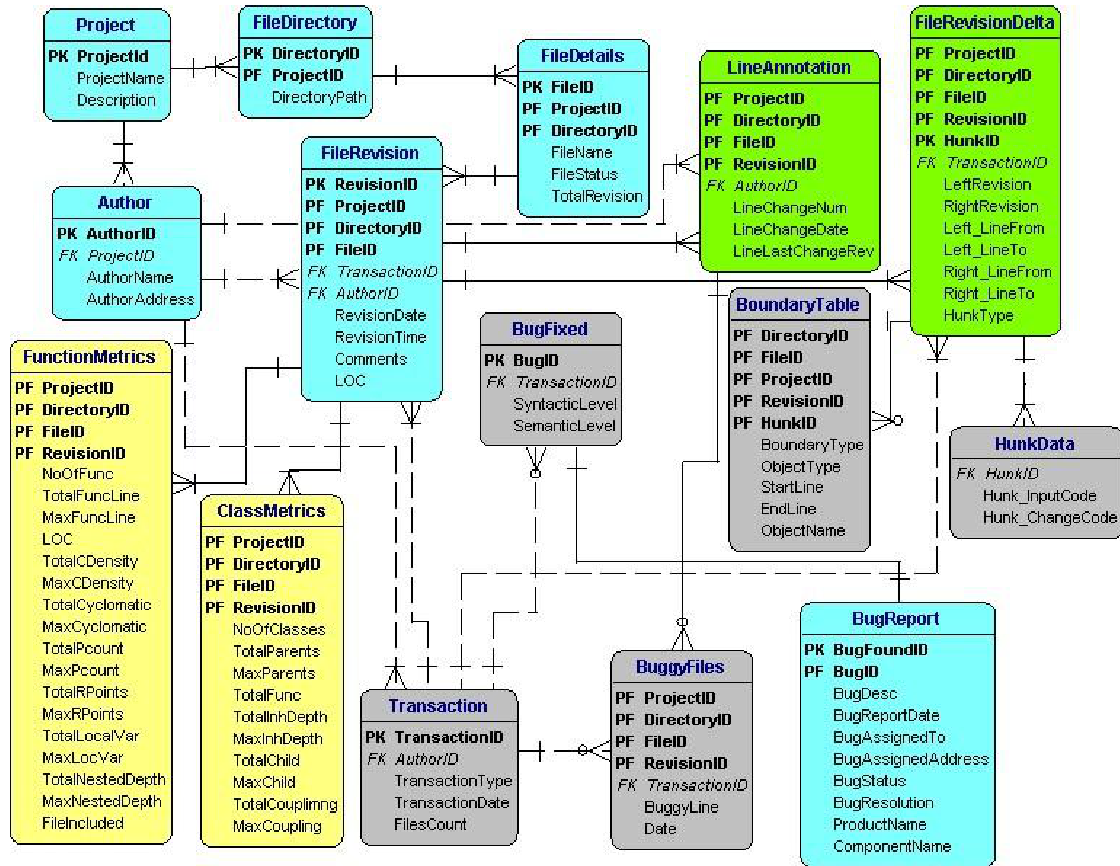


Figure 4.1. An ERD of a database for program file change analysis

2. **Bug Introducing Change** After identifying that the change is a bug fix it is required to identify those changes which actually introduced the bug. We used the approach proposed by Sliwerski, Zimmermann, and Zeller known as SZZ algorithm [Sliwerski et al., 2005]. According to this approach we first compute the difference between the fix revisions and its previous revision using the GNU diff tool. The diff tool gives a set of lines from previous revisions which are changed in the new revision to fix the bug. This is in the form of hunk pairs set. Hunk pair format details are given in Table 4.2. The hunk pair set provides information about the set of buggy lines in the previous revision. To find out the revision in which these buggy lines were introduced, CVS annotation is used. CVS annotation provides information about the last change revision of each code line. CVS annotation also gives the information about the author name that changed the line and the date of change. The last revision in which the set of buggy lines were last changed are identified as bug introducing change.

   Example: Consider an example as shown in Figure 4.2. It describes two different types of change transactions, which are committed or checked in at different times in the CVS repository. Transaction 1762 is a bug fixed transaction, in which two files are changed. File-A changed from revision 1.11 to 1.12 and File-C changed from revision 1.7 to 1.8. The GNU diff of the two successive revisions of File-A and File-C gives 3 hunk pair set (add, delete, modify). The annotation of these two files shows that the buggy code line of File-A revision 1.11 was last modified in revision 1.4 having transaction id 762 and similarly the buggy line in File-C were last changed in revision 1.3 of transaction 567. Therefore change transaction id 1762 is a bug fixed transaction while transaction id 567 and 762 are bug introducing transaction. Similarly the associated hunk pair set with each transaction may be identified as bug fixed or bug introducing hunk pairs.

   We also process each hunk pair set to identify the set of actual hunk pairs which introduced the bug. Figure 4.2 shows that not all the hunk pairs of bug introducing transactions are involved in introducing the bugs. The hunk pairs which are marked with a rectangle in a bug introducing transaction are responsible to introduce the bugs.

3. **Bug Fix-Introducing Change:** If program file changes are identified as a bug fix change and also identified as a bug introducing change, then this type of change is recognized as Bug Fix-Introducing Change. It happens when a developer introduces a new bug while fixing a previous bug.

4. **Clean Program Change:** For sets of change, which are not identified as bug fix or bug introducing or bug fix-introducing, are identified as clean change.

At this point, we have identified the different types of software changes, now in the next section we describe our risk model.

4.1.3 Model Building

In order to build the risk model we use the following standard definition of risk, *An expression of the impact and possibility of a mishap in terms of potential mishap severity and probability of occurrence* (MIL-STD-882-D, IEEE 1483). In engineering risk corresponds to the costs in case of an accident

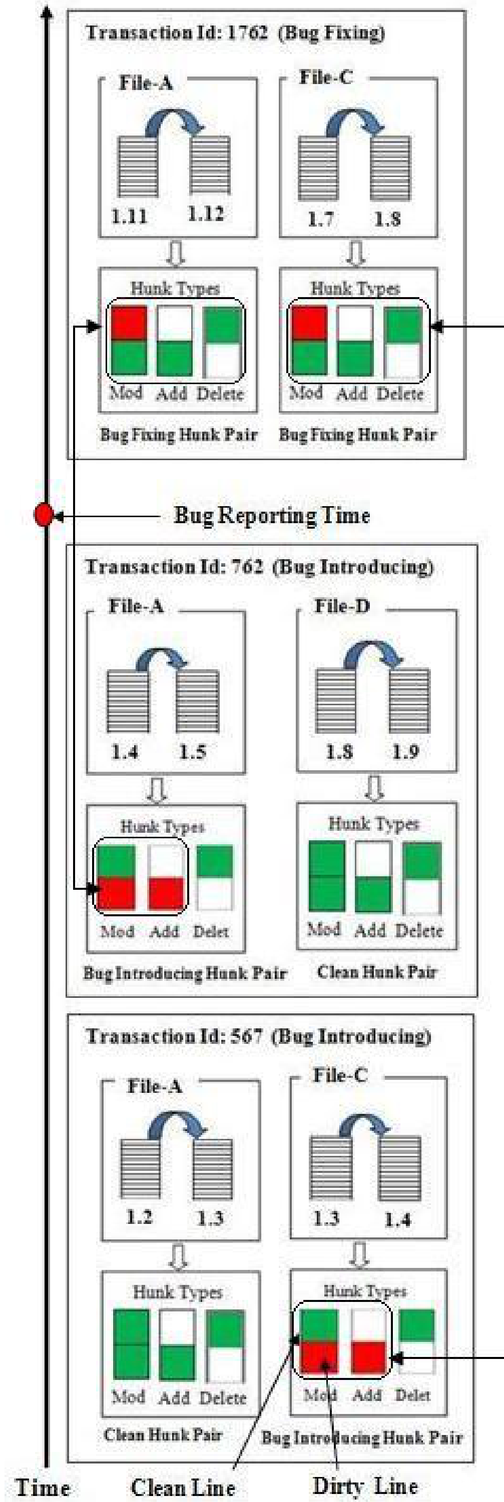


Figure 4.2. Example of bug fixing and bug introducing transaction

Table 4.2. Hunk pair format description

Example	Summary
9a10	Line or comma separated range of lines of previous file revision
> #include "XYZ.h"	concatenated with a single character (a/c/d), again concatenated
52,53c53,54	with line or comma separated range of line of current release.
<* if	**a** used for line addition
<foo==1*/	**c** used for modification and
> foo==1	**d** used for line delete
128d145	
<*/	

[Pressman, 2001] [Hall, February 28, 2005]. Hence, formally risk can be defined as

$$\text{Risk} = (Probability\ \ of\ \ an\ \ accident) \times (Expected\ \ costs) \tag{4.1}$$

$$\text{Risk}_j = \text{p}_{FAULT} \times \text{C}_j \tag{4.2}$$

Where, Risk_j is the risk that a change in a source file could be a faulty change, and it could introduce j number of faults. p_{FAULT} is the probability that a new change could be a faulty change, and C_j represent the expected impact of that faulty change. C_j represent the impact in term of an estimated cost that needs to be spent in fixing the introducing number of faults. A faulty change in a program file may introduce j number of faults into a software system. C_j is the cost of fixing j faults. Therefor, the risk that a faulty change may introduce j faults is given by equation 4.2.

The probability of a change to lead to a fault (or accident) i.e., p_{FAULT} can be obtained from the previous history of *bug introducing* and *bug fix-introducing* changes of a source file. For this purpose, the number of bug introducing (n_I) and bug fix-introducing changes (n_{FI}) has to be divided by the number of all changes (n_C), e.g.,

$$\text{p}_{FAULT} = \frac{\text{n}_I + \text{n}_{FI}}{\text{n}_C} \tag{4.3}$$

Now, to find the risk value for a program file we have to know the expected costs of the fault/faults that could be generated. Let, p_j be the probability of introducing j fault/faults by a faulty change, and c_j is the cost of fixing j fault/faults.

$$\text{C}_j = \text{p}_j \times \text{c}_j = \frac{\text{n}_j}{\text{n}_C} \times \text{c}_j \tag{4.4}$$

$$\text{Risk}_j = \frac{\text{n}_I + \text{n}_{FI}}{\text{n}_C} \times \frac{\text{n}_j}{\text{n}_C} \times \text{c}_j \tag{4.5}$$

Where, n_j is the number of changes that introduced j fault/faults per change. Its value can be obtained from the previous history of a source file. If we consider the cost of each fault as a constant value i.e., c, then 4.5 can be written as,

$$\text{Risk}_j = \frac{\text{n}_I + \text{n}_{FI}}{\text{n}_C} \times \frac{\text{n}_j}{\text{n}_C} \times \text{j} \times \text{c} \tag{4.6}$$

In case of OSS, there is no information regarding the costs for correcting a fault in terms of amount of work necessary, and moreover there is no information available about costs related to the fault when the program is executed. Hence, we might assume unit costs for each fault i.e., c=1.

$$\text{Risk}_j = \frac{\text{n}_I + \text{n}_{FI}}{\text{n}_C} \times \frac{\text{n}_j}{\text{n}_C} \times \text{j} \tag{4.7}$$

Equation 4.7 can be used to find the risk associated with source files, which could generate j faults. Every change might lead to several faults. Hence, the expected costs can be estimated by averaging the costs of correcting k faults multiplied with the probability that a change causes k faults, i.e.:

$$\text{Risk}_{avg(j)} = \frac{\text{n}_I + \text{n}_{FI}}{\text{n}_C} \times \frac{1}{\text{k}} \sum_{j=1}^{k} \frac{\text{n}_j}{\text{n}_C} \times j \tag{4.8}$$

$$\text{Risk}_{avg(j)} = \frac{\text{n}_I + \text{n}_{FI}}{\text{n}_C} \times \frac{1}{\text{k}} \times \frac{1}{\text{n}_C} \times \sum_{j=1}^{k} (\text{n}_j \times j) \tag{4.9}$$

Where, $\sum_{j=1}^{k}(n_j \times j) = n_B$, and n_B is the number of bugs introduced. Its value is obtained by adding the value of bug counts n_{BI} which is generated by bug introducing changes and bug count value n_{BFI} which is generated by bug fix-introducing changes (n_B=n_{BI}+n_{BFI}).

$$\text{Risk}_{avg(j)} = \frac{\text{n}_I + \text{n}_{FI}}{\text{n}_C} \times \frac{1}{\text{k}} \times \frac{\text{n}_B}{\text{n}_C} \tag{4.10}$$

Where, k is the maximum number of faults introduced by a single change in a source file. Furthermore, equation 4.10 can be used to obtain the total or maximum risk instead of an average risk.

$$\text{Risk}_{max(j)} = \frac{\text{n}_I + \text{n}_{FI}}{\text{n}_C} \times \frac{\text{n}_B}{\text{n}_C} \tag{4.11}$$

Equation 4.11 may be used to measure the maximum risk associated with the new changes in a source files. In equation 4.11, $\frac{n_B}{n_C}$ is the average number of faults per change. It may be written as $C_{max(j)}$. In our experiment we used equation 4.11 to measure the risk value. In equation 4.11, we considered unit cost i.e., c=1, if we don't consider unit cost then equation 4.11 may be written as,

Table 4.3. Mozilla project program file change history (only 8251 C++ source files revisions).

Change Type	Transaction Count	Change or Revision Count	Revision/ Transaction	Hunk Count	Hunk/ Transaction	Hunk/ Revision	Change LOC	Change LOC/ Transaction	Change LOC/ Revision
Bug Fixing	589 (15.80%)	1034 (12.50%)	1.75	2874 (7.30%)	4.88	2.78	5904 (9.40%)	10.0	5.7
Bug Introducing	676 (18.2%)	1812 (22.00%)	2.68	11456 (29.00%)	16.9	6.32	15752 (25.20%)	23.3	8.7
Bug Fix Introducing	1405 (37.8%)	3154 (38.20%)	2.24	16945 (42.80%)	12	5.37	29174 (46.60%)	20.76	9.2
Clean	1046 (28.10%)	2251 (27.30%)	2.15	8289 (21.00%)	7.9	3.68	11754 (18.80%)	11.23	5.2
Total	3716	8251	2.2	39564	13.6	4.8	62584	15.84	7.58

$$\text{Risk}_{max(j)} = \frac{\text{n}_I + \text{n}_{FI}}{\text{n}_C} \times \frac{\text{n}_B}{\text{n}_C} \times \text{c} \tag{4.12}$$

4.2 Experimental Setup

We performed an experiment by using the data which we have obtained in Section 4.1.1. We used these data sets to extract the total number of changes and their types. Furthermore, we assumed that changes with more than 100 lines changed as an outliers (this may be the point where branches were merged). When a change has occurred in a file, then its new revision is introduced. After removing all the outliers' revision, we have 8251 source revisions out of 9552 revisions.

Our experiment result is shown in Table 4.3. The average changed LOC (line of code) per transaction is more in case of bug introducing transactions. It means that whenever large numbers of code lines are changed then it may introduce new bugs. A hunk represents a change location in the source code, therefore large number of hunk mean larger number of different change locations in the source code. Table 4.3 shows that the hunk value is high in case of bug fix- introducing change, it means that whenever a source file is changed from multiple locations, then the probability of bug creation will be increased. Table 4.3 also shows that the change transaction for bug fix-introducing change is high as compared to other transaction.

Figure 4.3 shows that, in 1998, the number of clear change transactions was high, the reason may be the earlier development phase of the project. While from 1999 to 2007 the number of bug fix-introducing change transactions is high as compared to other changes transactions.

4.2.1 Results and Discussion

To obtain the results we used the relations which we have described in section 4.1.3. Table 4.4 depicts the obtained results of risk associated with 10 different C++ source files of Mozilla project. For example consider file BrowserView.cpp from Table 4.4. The number of fix-introducing changes is 31.

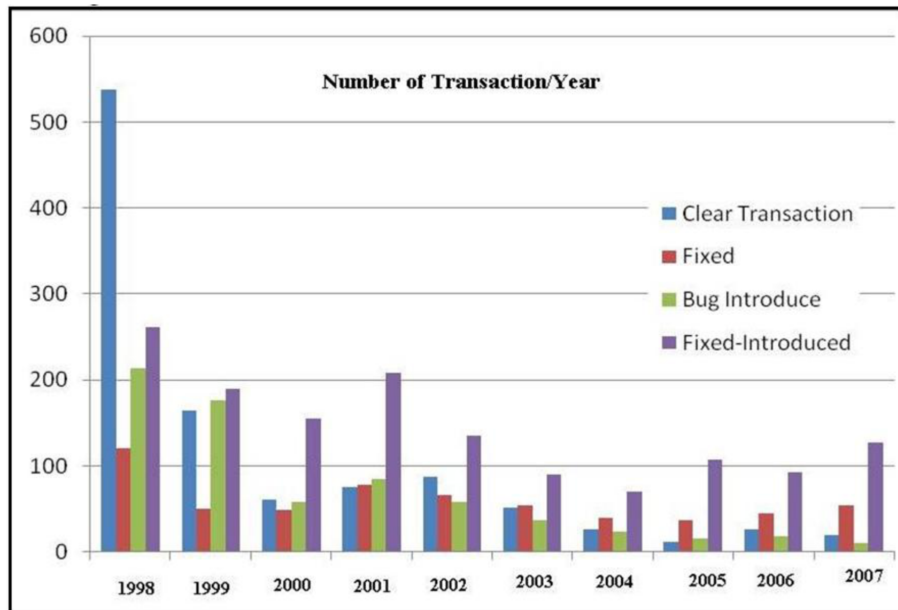


Figure 4.3. Distribution of source file changes per year (Mozilla 1000 C++ files).

Hence, the probability of a change to introduce at least one bug in this example is,

$$p_{BUG} = (n_I + n_{FI}) \ / \ n_C = (9+31)/59 = 0.67.$$

From the data of Table 4.4, we can compute the expected cost of fixing faults in source file BrowserView.cpp , which is $C_{max(j)} = \frac{n_B}{n_C} = \frac{151}{59} = 2.56$. From this the risk of the file is given as follows:

$$R_{max(j)} = p_{BUG} \times C = 0.67 \times 2.56 = 1.715$$

The risk can be computed for all files in the same way. The results of this computation are given in Table 4.4. Figure 4.4 shows the distribution of different types of program file changes. To perform this experiment we randomly selected all the changes/revisions of 10 C++ source files. Figure 4.4 shows that the bug fix-introduce change is high in all the files as compare to other changes

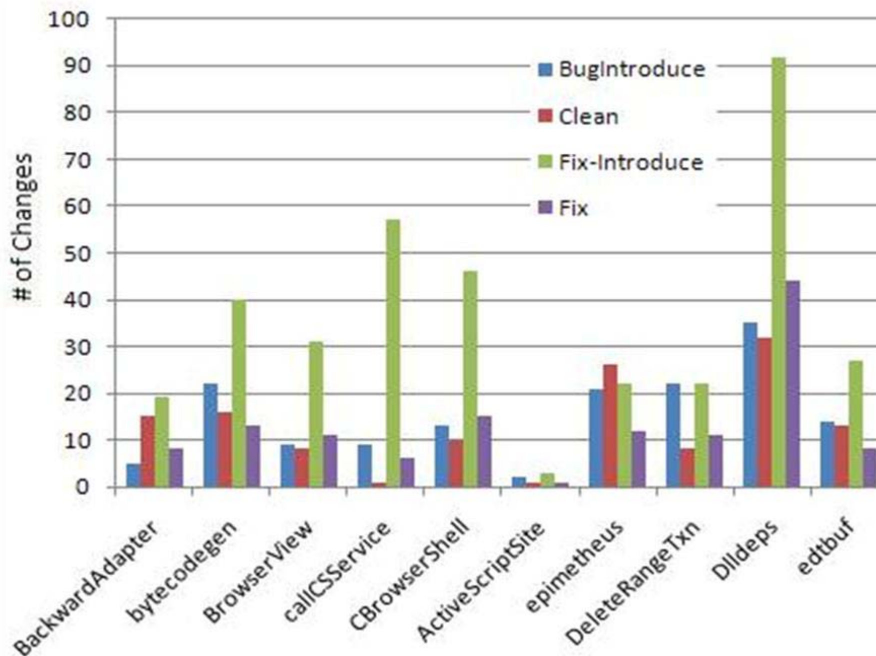
4.2.2 Threats To Validity

In this section we presented our approach to analyze the hunk data set, and presented the results obtained from Mozilla projects. We have processed the log comments to identify bug fix revisions, some bugs may not be captured by log comments. Some comments are marked bug fix but actually they may be enhancements. The quality of log comments may affect the results.

Table 4.4. Source files change distribution (randomly selected 10 C++ files from Mozilla project) and calculation for Risky Program File for new changes.

File Name (.cpp)	Change Type				Total Change	Bugs Count		Total Bug	Total Buggy Change	p_{BUG}	$C_{max(j)}$	$Risk_{max(j)} = p_{BUG} \times C_{max(j)}$
	n_I	Clean	n_{FI}	Fix	n_C	n_{BI}	n_{BFI}	n_B	n_I+n_{FI}			
BackwardAdapter	5	15	19	8	47	8	36	44	24	0.51	0.94	0.48
bytecodegen	22	16	40	13	91	77	103	180	62	0.68	1.98	1.35
BrowserView	9	8	31	11	59	25	136	161	40	0.67	2.73	1.83
calICSService	9	1	57	6	73	44	187	231	66	0.9	3.16	2.84
CBrowserShell	13	10	46	15	84	32	169	201	59	0.7	2.39	1.67
ActiveScriptSite	2	1	3	1	7	5	14	19	5	0.71	2.71	1.92
epimetheus	21	26	22	12	81	31	36	67	43	0.53	0.83	0.44
DeleteRangeTxn	22	8	22	11	63	80	72	152	44	0.7	2.41	1.69
Dlldeps	35	32	92	44	203	63	176	239	127	0.63	1.18	0.74
edtbuf	14	13	27	8	62	35	142	177	41	0.66	2.85	1.88

Figure 4.4. Change history of 10 C++ source file of Mozilla project.



4.3 Application of Hunk Data as Hunk Metrics

The *hunk* data of source files may be used to measure a set of metrics. These metrics are known as *hunk metrics*. The hunk metrics may be used for the classification of new changes as clean or buggy. Furthermore, the *hunk* data may be used to identify the bug inducing language constructs. Therefore, in order to explore the applications of hunk data. Two different experiments have been performed. In the following paragraph we briefly describe the experimental results of both the experiments (for details see [Ferzund, 2009])

In an experiment we classify a program source file change (hunk) as buggy or clean. We classify individual hunks as buggy or bug-free, thus we provide an approach for bug prediction at the smallest level of granularity. We introduce a set of hunk metrics and build classification models based on these metrics. Classification models are built using logistic regression and random forests. We evaluated the performance of our approach on seven different open source software projects. Our classification approach can classify hunks as buggy or bug free with 81 percent accuracy, 77 percent buggy hunk precision and 67 percent buggy hunk recall on average. Most of the hunk metrics are significant predictors of bugs but the set of significant metrics varies among different projects [Ferzund et al., 2009a].

Furthermore, we empirically investigate the language constructs which frequently contribute to bugs. Revision histories of eight open source projects developed in multiple languages are processed to extract bug-inducing language constructs. Twenty six different language constructs and syntax elements are identified. We find that most frequent bug-inducing language constructs are function calls, assignments, conditions, pointers, use of NULL, variable declaration, function declaration and return statement. These language constructs account for more than 70 percent of bug-inducing hunks. Different projects are statistically correlated in terms of frequencies of bug-inducing language constructs. Quality assurance developers can focus code reviews on these frequent bug-inducing language constructs before committing changes [Ferzund et al., 2009b].

4.4 Summary

In this chapter, we presented our approach to extract the program file change data from software's repository. From the extracted data, we defined a *hunk* as a set of changed source code lines, and classified hunks or software changes into four different type of changes. Like bug free, bug introducing, bug fixing and bug fix-introducing. Sometimes bug introducing are also called bug inducing hunk. After extracting and identifying hunk data sets, we performed an experiment. In our experiment, we used Mozilla project data. We used this data to find four different types of program file changes and showed how these changes are distributed during the evolution of a source file. Moreover, we presented a way for calculating the risk value for a file. The files which have higher risk values are more risky when applying the next change and thus should be tested more thoroughly.

These findings can be helpful for the analysis of program file changes, and maybe used to construct a tool for the automation of prediction risk associated with the new changes in source codes.

5

Software Fault Prediction Models

Parts of the contents of this chapter have been published [Ahsan and Wotawa, 2010c], [Ferzund et al., 2008a] and [Ferzund et al., 2008b].

In software engineering, faults and failures in software is accounts for a significant amount of any project budget as activities related to faults detection and faults correction often correspond to 30-50% of the budget [Sommerville, 2006]. Moreover, a major part of today's software industries involved in the development of mission and safety critical complex software systems. These systems are heavily dependent on reliability of their underlying software applications. An early software fault prediction is a proven technique in achieving high software reliability. To evaluate the quality of software, source code metrics provides ways to construct fault prediction models. Prediction models based on software metrics can predict numbers of faults in software modules or classify software modules as faulty or non fault modules. Timely predictions of such models can be used to direct cost-effective quality enhancement efforts to modules that are likely to have large numbers of faults [Khoshgoftaar and Seliya, 2003].

Research Trends: A software fault is a defect that causes software failure in an executable product. Fixing software defects is one area that takes up a large amount of software testing and maintenance effort. Particularly, software testing requires a lot of effort and resources. To efficiently utilize the available resources for testing, it is better to schedule these resources to those parts of the source code where the likelihood of bugs is greater. An extensive body of work has focused on finding the fault-prone locations in software systems. They identify software subsystems (modules, components, classes, or files) which are likely to contain faults. These subsystems, in turn, receive additional resources for verification and validation activities. The majority of this work builds prediction models using metrics that predicts where future defects are likely to occur. Most of the studies indicate a relationship between the obtained metrics and the number of bugs [Gyimothy et al., 2005] [Ostrand et al., 2005]. The research relies on software configuration management systems and bug databases hold important information related to bugs and changes made to files.

In order to predict the number of bugs or to provide a predictor with regard to a classification schema, there are two possible approaches. The first approach uses statistical methods like multiple linear regression, logistic regression, and principal components analysis [Nagappan et al., 2006]. Linear regression can be successfully used if the dependent variables change linearly with the independent variables. As most of the metrics normally correlates with each other, there is a strong need to overcome the multicollinearity problem. Principal component analysis is used in this

respect to reduce the multicollinearity effect. Logistic regression can be used for binary classifications. The second approach relies on machine learning techniques like decision tree induction, support vector machine, artificial neural networks, k-nearest neighbors to mention some of them. Machine learning techniques have the ability to learn from past data and these techniques can be employed in a variety of complex situations (see [Ian H. Witten and Kaufmann, 2005]). Zhang [Zhang, 2000] showed the application of machine learning techniques in tackling software engineering problems. He suggested that machine learning algorithms can be used to build tools for software development and maintenance tasks as well as can be incorporated into software products to make them adaptive and self-configuring. Koru et al. [Koru and Liu, 2005b] combined static software measure with defect data at class level and applied different machine learning techniques to develop fault predictor models.

Motivation: It is observed that as the time goes by fault prediction models are becoming more and more complex. New studies are investigating more aspects that may help improve prediction accuracy, which leads to more metrics (i.e., independent variables) being input to the prediction models. Although adding more metrics (i.e., independent variables) to the prediction models may increase the overall prediction accuracy, it also introduces some negative side effects. First, it makes the models more complex and therefore, makes them less desirable to adopt in practical settings. Second, adding more independent variables to the prediction model makes determining which independent variables actually produce the effect (i.e., impact) on post-release defects more complicated (due to multicollinearity). The aforementioned problems turn the complex prediction models into black box solutions that can be used to know where the defects are, but do not provide insight into the underlying reasons for what impacts the defects. The black-box nature of such models, bug prediction is a major hurdle in the adoption of these models in practice [Ball and Siy, 1997]. Another challenge in the construction of reliable fault prediction models is, labeling of program files with the exact number of bugs. In case of open source software (OSS), the labeling of program files with the exact number of fixed bugs depends on the link between the revision control system (like CVS) and the bug tracking system (like Bugzilla). Unfortunately, there are no direct ways to establish such links. However, parsing of CVS check in comments are used to identify the number of bugs fixed in a revision of a source files [Śliwerski et al., 2005b]. Since, parsing of CVS comments is not a reliable way. Therefore, there are chances that one can label source files with the wrong number of fixed bugs, which ultimately lead to unreliable fault prediction model.

Our Goals: To overcome the above mentioned issues, we are motivated to build a prediction model with a small number of predictors (code metrics), which are highly correlated with the number of post release defects. Our main objective is to apply two different techniques i.e., regression and classification on the code metrics data, and construct a fault prediction model for software modules. Furthermore, our aim is to investigate the fault prediction capability of the constructed model using several statistical and machine learning models' evaluation criteria.

Our Research Approach: We used two different approaches to construct a fault prediction model. In the first approach, we work with the different techniques, commonly used for the construction of regression based fault prediction models. The best examples of such techniques are, *Multiple Linear Regression* and *Neural Network*. The first step, is to map each source file with the number of its post release defects. Then, source files are processed to extract the code metrics data. Code

metrics data are also known as raw metrics data [Khoshgoftaar and Seliya, 2003]. The next step, is to transform the raw metrics data into principal components. Finally, raw metrics and principal components both are used to construct regression based fault prediction model. Whereas, in the second approach, instead of labeling of source files with the exact number of its post release defects, we classify source files according to a predefined classification schema. We compare two classification schemes. The first schema allows classifying files as being buggy or not. The second schema is called bug level classification where files are classified as low buggy, medium buggy, or highly buggy respectively. Once the source files are classified according to the classification schema, then files are processed to extract code metrics and principal components. The code metrics data (both raw metrics and principal components) and different classification techniques of machine learning are used to construct fault predictor models, which classify source files according to a predefined classification schema. Finally, different model evaluation criteria are used to evaluate the fault prediction capability of the obtained models. Our approach makes use of code metrics, which are highly correlated with the number of bugs and have small correlation among each other.

Furthermore, we performed an experiment to answer the question whether bug predictors obtained from one project can be applied to a different project with reasonable accuracy. Two open source projects Firefox and Apache HTTP Server (AHS) are used for this study. Static code metrics calculated for both projects using in house software and the bug information is obtained from bug databases of these projects. The source code files are classified as clean or buggy using a Decision Tree classifier. The classifier is trained on metrics and bug data of Firefox and tested on Apache HTTP Server and vice versa. The results obtained vary with different releases of these projects and can be as good as 92% of the files correctly classified and as poor as 68% of the files correctly classified by the trained classifier.

Our Contributions: The main contributions are,

1. Construction and evaluation of five different regression based fault prediction models using data set of four releases of Firefox project.
2. Construction and evaluation of five different classification based machine learning models. We used different techniques of classification to classify source files according to a predefined classification schema.
3. We evaluated each model by training and testing on different project data, e.g., we trained our fault prediction model on Firefox and tested on Apache HTTP Server.

Chapter Layout: The rest of the chapter is organized as follows. Section 5.1 presents our approach for the construction of fault prediction models. In Section 5.2, we describe regression approach to construct a prediction model. Whereas, in Section 5.3, we describe our approach to construct fault prediction models using machine learning classification techniques. In Section 5.4, we discuss the results of a case study in which we evaluate the performance of our bug predictor models by applying them on different project's data. Finally, in Section 5.5 we summarize the chapter.

5.1 Approach to Build Fault Prediction Models

This study presents a comparative evaluation of predictive accuracy of different fault prediction models. These models are based on two different approaches, which are commonly used for the construction of fault prediction models. These approaches are:

1. **Regression based Fault Prediction Models:** Predictors (metrics) are used to predict the expected number of faults in a module/source file,
2. **Classification based Fault Prediction Models:** Classification are used to predict whether a module/source file will have at least one fault reported.

In order to obtain a state of the art fault prediction model, we used both the approaches and constructed different fault prediction models. In case of regression, we used two different types of regression techniques i.e., parametric regression (Multiple Linear Regression) and non-parametric regression (Artificial Neural Network). Similarly, in case of classification, we used logistic regression and different machine learning techniques. An overview of all these techniques briefly described in Chapter 2. We used all these techniques and build different models. With the help of these models, we performed a comprehensive analysis, which led us to select an appropriate fault prediction model. Our approach, in a nutshell:

1. **Data Preprocessing:** We, first extracted the code metrics data from the source files and also extracted the defect or fault information from the bug tracking system. Finally, we mapped the two data i.e., source file's code metrics and number of fault by using an approach give by [Thomas Zimmermann, May 23-28, 2004].
2. **Correlations:** The first step to build the model is to find the correlation between the predictor (code metric) and the number of faults. In case of regression based model we, used both pearson and spearman's correlation techniques. Whereas, in case of classification based models we used point-biserial correlation.
3. **Remove Multicollinearity:** We, used principal component analysis to remove the multicollinearity from the data.
4. **Model Building:** Regression and Classification techniques are used to build the model
5. **Selecting Models:** Different evaluation criteria of statistics and machine learning are used to select an appropriate model.
6. **Model's Validation:** Firefox Releases 0.8, 1.0, 1.5, and 2.0 are used as test data sets to evaluate the prediction accuracy of the selected models. Furthermore, we trained the model on Firefox project data and tested on Apache project data. We, used 10-fold cross validation techniques to evaluate each model.

In the next sections, we describe the tasks that had been carried out in order to obtain the empirical results related to the fault prediction models. The motivation is to provide background information. We start with a discussion of how to obtain the data collection. Afterwards, we introduce the concepts behind the metrics calculation. Finally, we briefly describe the used machine learning techniques. Figure 5.1 represents a schematic diagram of our approach.

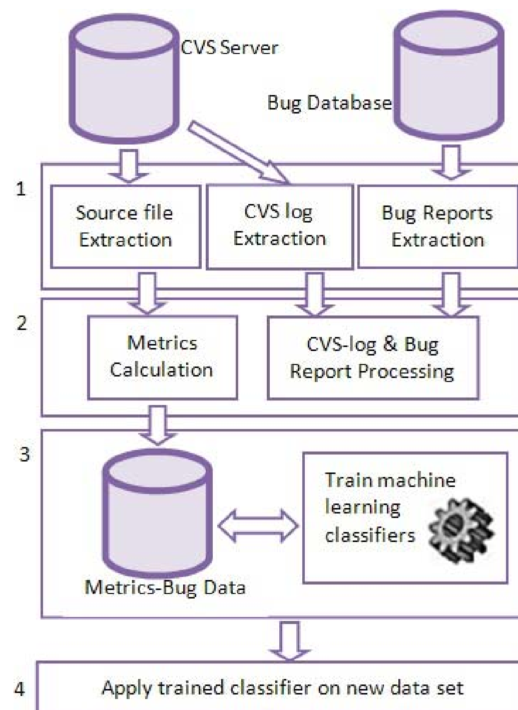


Figure 5.1. Classification Approach

5.1.1 Collecting Data

We downloaded Firefox source code (Release 0.8, 1.0, 1.5 and 2.0) on a local drive using the ftp server. We obtained the bug information related to Firefox from Bugzilla[1]. Wget[2] tool can be used to retrieve bug reports from Bugzilla server. We also processed the CVS log output to find the revisions of files in which bugs were fixed. Whenever a developer makes a change to a file and commits it to the CVS repository, he records a message explaining the cause for change. These comments by developers hold important information regarding bugs. The bugs reported in Bugzilla as well as the CVS comments were used to map bugs to individual revisions of files. We calculated for each file the number of bugs that occurred between two consecutive releases of Firefox. A database was designed to hold the information regarding bugs and metrics. We processed the following number of cpp, header, and c files in each release of Firefox:

File Type	Firefox 0.8	Firefox 1.0	Firefox 1.5	Firefox 2.0
cpp	3913	4017	4216	4276

5.1.2 Computing Code Metrics

We used our own software to calculate the source code metrics. We performed static analysis of downloaded source code files and calculated individual metrics for functions and classes. Metrics then aggregated on files getting total and maximum values of each metrics for each file. We used the following function metrics for our study:

[1] https://bugzilla.mozilla.org/
[2] http://www.gnu.org/software/wget/

Metrics	Description
CC	Total Cyclomatic Complexity
MCC	Maximum Cyclomatic Complexity
CD	Total Control Density
MCD	Maximum Control Density
PC	Total Parameter Count
MPC	Maximum Parameter Count
RPC	Total Return Point Count
MRPC	Maximum Return Point Count
LVC	Total Local Variable Count
MLVC	Maximum Local Variable Count
ND	Total Nesting Depth
MND	Maximum Nesting Depth
FLOC	Function Lines Of Code
MLOC	Maximum Function Lines Of Code

File metrics include the following along with the aggregated metrics for functions and classes:

Metrics	Description
NOF	Total Number Of Functions
NOIF	Number Of Included Files
LOC	Lines Of Code
NOGIF	Number of global if

5.2 Fault Prediction Models Using Regression Techniques

Regression based fault prediction models are built using statistical regression and machine learning techniques. The most commonly used methods for the regression based fault prediction models are multiple linear regression (MLR) and neural network (NN). We used both MLR and NN. Moreover, to perform a comparative study, we used three other techniques of machine learnings, which are commonly used for numerical prediction. The names of these techniques are, support vector machine (SVM), naive bayes (NB), and radial basis function (RBF). We built these models from the computed metrics data of Firefox. The independent variables in our prediction models are the set of metrics under study for each source/class file, while the dependent variable is the number of post-release faults. In the following subsections, we describe in details about our methodology, which we used to construct the regression based fault prediction model. Our methodology is based on an approach proposed by Nagappan et al. [Nagappan et al., 2006].

5.2.1 Metrics Correlation

In Statistics, correlation is a measure of association between two variables. Like, in our case these variables are the number of bugs (post release defects) and the code metrics. The value of the correlation coefficient varies between +1 and -1. When the value of the correlation coefficient lies around $\pm$ 1, then it is said to be a perfect degree of association between the two variables. As the

value goes towards 0, the relationship between the two variables will be weaker.

Usually, in statistics, we measure two types of correlations, i.e., Pearson's correlation and Spearman's rank correlation. In case of Pearson's correlation, we compute the degree of the relationship between the linear related variables, and both variables should be normally distributed. Therefore, Person's correlation belongs to parametric statistics, where sample distribution is known. Whereas, Spearman's rank correlation coefficient is nonparametric that does not assume any assumptions related to the distributions of correlated variables. Therefore, for Spearman's rank correlation it is not necessary that both the variable should have linear relationships. Spearman's rank correlation can be used if the relation between the two variable is non-linear [Breu, 2006]. Furthermore, Spearman's rank correlation is the correlation between the ranks of the measures of attributes, rather than on the magnitude of their measures, and is not sensitive to outliers.

In our experiment, we used both technique of correlation i.e., pearson and spearman rank correlation. We, find the correlation between the code metrics and the number of post release bugs. Table 5.1 list the correlation results of four different releases of Firefox project. Correlation significant at the 0.01 level.

Table 5.1, shows that FLOC (function line of code) is highly correlated with the number of post release bugs. The correlation value of FLOC is 0.653 (Pearson), similarly we found that NOF (number of function), control density, and cyclomatic complexity are also highly correlated with the number of post release defects. Theoretically, correlation above 0.5 is considered as strong correlation. However, in case of fault prediction models, correlations over 0.4 are considered as strong correlation [Ohlsson and Alberg, 1996, Zimmermann and Nagappan, 2008]. Furthermore, Table 5.1, depicts that the correlation patterns are similar, among all projects and code metrics are positively correlated with each other.

5.2.2 Metrics Scatter Plot

Scatter graph is a graph, which help to plot the information on two related variables. It consists of plotted points, which are scattered all around the X-Y grid. The main use of the scatter graph is to visualize the strength of the co-relation between the two metrics. Therefore, we decided to draw a scatter plot of each metric with the number of post release defects. The results are shown in Figure 5.2

Figure 5.2 depicts that NOF, FLOC, MFLOC, CC, PC and LOC are highly correlated with the number of post release defects.

5.2.3 Principle Component Analysis (PCA)

It is shown in Table 5.2 that some code metrics correlated with each other, which means that our metric data has multicollinearity issues. Multicollinearity occurs when predictors (dependent variables) are so highly correlated with each other that it is difficult to come up with reliable estimates of their individual regression coefficients. Multicollinearity does not reduce the predictive power or reliability of the model as a whole; it only affects calculations regarding individual predictors. A multiple

Table 5.1. Source code metrics correlation with the number of faults

Metrics	Firefox 0.8		Firefox 1.0		Firefox 1.5		Firefox2	
	Pearson	Spareman	Pearson	Spareman	Pearson	Spareman	Pearson	Spareman
NOF	**0.520***	0.449*	**0.518***	**0.531***	**0.585***	0.407*	0.491*	0.280*
FLOC	**0.536***	0.477*	**0.525***	**0.551***	**0.653**	0.429*	**0.542***	0.290*
MFLOC	0.309*	0.411*	0.341*	0.460*	0.369*	0.372*	0.308*	0.260*
CD	**0.504***	0.444*	**0.503***	**0.517***	**0.606***	0.411*	0.495*	0.264*
MCD	0.255*	0.393*	0.283*	0.464*	0.234*	0.336*	0.182*	0.224*
CC	**0.503***	0.450*	0.487*	**0.513***	**0.608***	0.413*	0.499*	0.265*
MCC	0.293*	0.397*	0.307*	0.439*	0.322*	0.358*	0.259*	0.229*
PC	0.471*	0.451*	0.476*	**0.550***	0.595*	0.407*	0.495*	0.264*
MPC	0.271*	0.365*	0.309*	0.465*	0.294*	0.329*	0.247*	0.226*
RPC	0.462*	0.457*	0.481*	0.503*	0.551*	0.424*	0.427*	0.242*
MRPC	0.202*	0.343*	0.223*	0.312*	0.214*	0.313*	0.153*	0.146*
LVC	**0.515***	0.434*	0.465*	0.476*	**0.563***	0.397*	0.455*	0.260*
MLVC	0.274*	0.376*	0.267*	0.407*	0.292*	0.348*	0.234*	0.238*
NDh	**0.512***	0.458*	0.495*	**0.532***	**0.630***	0.412*	**0.532***	0.286*
MND	0.294*	0.358*	0.260*	0.368*	0.308*	0.351*	0.243*	0.216*
NOIF	**0.526***	0.475*	**0.562***	**0.546***	**0.612***	0.433*	**0.517***	0.288*
NOGIF	0.073*	0.125*	0.117*	0.166*	0.159*	0.189*	0.147*	0.110*
LOC	0.496*	0.467*	**0.520***	**0.536***	**0.632***	0.398*	**0.503***	0.271*

Correlation is significant at the 0.01 (2 tailed)

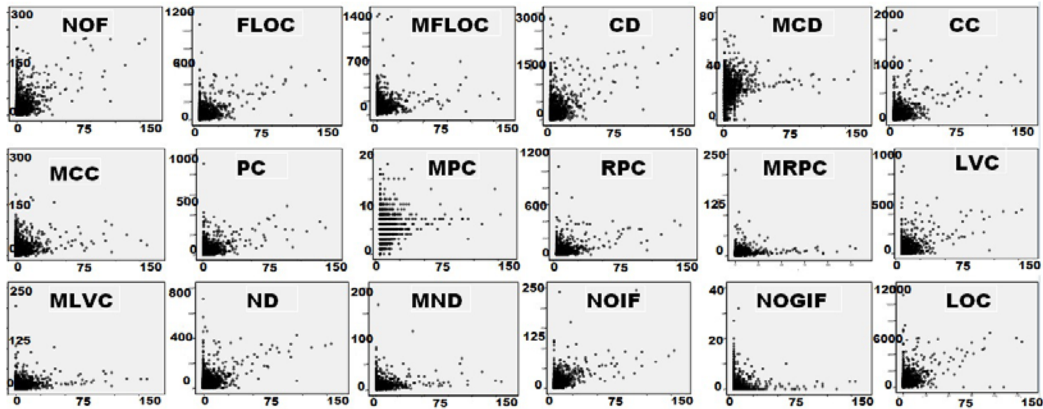


Figure 5.2. Scatter plot between code metrics and number of faults.

regression model with correlated predictors can indicate how well the entire bundle of predictors predicts the outcome variable, but it may not give valid results about any individual predictor, or about which predictors are redundant with others.

The presence of multicollinearity is not a favorable condition to construct a predictor model. To overcome the multicollinearity problem, we used a standard statistical approach, namely principal component analysis (PCA) [Jackson, 2003]. Principal component analysis is appropriate when we have obtained measures on a number of observed variables and wish to develop a smaller number of new variables (called principal components) that will account for most of the variance in the observed variables. With PCA, a smaller number of uncorrelated linear combinations of metrics that account for as much sample variance as possible are selected for use in regression. These principal components are independent and free from multicollinearity. The principal components may then be used as predictors in fault prediction models.

To transform the metrics (raw) data into principal components, statistical tool, SPSS is used. Using SPSS, we obtained all the principal components for each of the four releases of Firefox project that account for a cumulative sample variance greater than 96%. Table 5.3 shows, top 10 principal components, which accounts 97% of the total variance in Firefox (Release 0.8) project. After transforming the raw metrics data into principal components, the next step is to use them and build models.

5.2.4 Fault Prediction Models Using Multiple Linear Regression

To perform our experiments, we built four MLR models. These models are trained using metrics data of four different releases of Firefox, i.e., Rel. 0.8, 1.0, 1.5, and 2.0. We used both raw metrics and principal components to construct the models. The built MLR models have the following form,

$$y = b_0+b_1x_1+b_2x_2+.....b_nx_n$$

Where, y is the dependent variable and x_n are the set of n predictor (code metrics) variables and b_n are n different coefficients. For each model, y represents the number of faults in a subsystem or module. MLR is formally described in chapter 2. We used the techniques of MLR and performed two different experiments to build the fault prediction models. In the first experiment, we used the principal components as the predictors (independent) variable and the post-release defects as the dependent variable. In the second experiment, we used raw metrics data as the dependent variable. We evaluated our models with the help of metrics, which are most commonly used for the evaluation of regression models.

For each model, we present the R, R^2, Adjusted R^2 and F-ration value. Where R is the correlation between the actual and the predicted value. R^2 is the ratio of the regression sum of squares to the total sum of squares. As a ratio, it takes values between 0 and 1, with larger values indicating more variability explained by the model and less unexplained variation. In simple words, the higher the R^2 value, the better the predictive power. The adjusted R^2 measure also can be used to evaluate how well a model will fit a given data set. Adjusted R^2 are commonly used to explain for any bias in the R^2 measure by taking into account the degrees of freedom of the independent variables

Table 5.2. Metrics correlation with each other

Metrics	FLOC	MFLOC	CD	MCD	CC	MCC	PC	MPC	RPC	MRPC	LVC	MLVC	ND	MND	NOIF	NOGIF	LOC
NOF	0.856*	0.441*	0.894*	0.481*	0.810*	0.447*	0.910*	0.515*	0.824*	0.319*	0.764*	0.416*	0.907*	0.415*	0.587*	0.195*	0.837*
FLOC		0.676*	0.913*	0.465*	0.979*	0.657*	0.881*	0.528*	0.868*	0.466*	0.934*	0.564*	0.957*	0.602*	0.653*	0.232*	0.960*
CD			0.497*	0.410*	0.651*	0.871*	0.469*	0.460*	0.504*	0.572*	0.580*	0.631*	0.580*	0.686*	0.529*	0.208*	0.664*
MCD				0.546*	0.927*	0.524*	0.845*	0.499*	0.875*	0.387*	0.862*	0.471*	0.919*	0.492*	0.606*	0.225*	0.889*
CC					0.482*	0.484*	0.445*	0.515*	0.426*	0.354*	0.426*	0.411*	0.471*	0.400*	0.384*	0.147*	0.474*
MCC						0.693*	0.830*	0.502*	0.875*	0.486*	0.935*	0.567*	0.938*	0.618*	0.607*	0.230*	0.937*
PC							0.460*	0.449*	0.522*	0.605*	0.585*	0.622*	0.594*	0.732*	0.429*	0.196*	0.647*
MPC								0.596*	0.806*	0.331*	0.813*	0.441*	0.898*	0.455*	0.594*	0.202*	0.856*
RPC									0.445*	0.271*	0.480*	0.439*	0.520*	0.405*	0.417*	0.206*	0.529*
MRPC										0.597*	0.839*	0.436*	0.828*	0.422*	0.568*	0.158*	0.841*
LVC											0.417*	0.359*	0.374*	0.327*	0.312*	0.101*	0.473
MLVC												0.663*	0.887*	0.565*	0.590*	0.200*	0.889*
ND													0.518*	0.534*	0.398*	0.148*	0.553*
MND														0.646*	0.617*	0.227*	0.922*
NOIF															0.397*	0.154*	0.581*
NOGIF																0.270*	0.657*
LOC																	0.252*

Correlation is significant at the 0.01 (2 tailed)

Table 5.3. The top ten Principal components obtained from the transformation of raw metrics data of Firefox (Rel. 0.8) project.

Metrics	Principal Component									
	1	2	3	4	5	6	7	8	9	10
NOF	**0.90**	0.13	0.09	0.08	0.25	0.08	0.08	0.12	0.08	-0.06
FLOC	**0.89**	0.22	0.04	0.11	0.11	0.12	0.07	0.12	0.25	0.07
MFLOC	**0.89**	0.09	0.04	0.06	0.18	0.14	0.05	0.13	0.02	**0.31**
CD	**0.88**	0.32	0.12	0.14	0.08	0.12	0.07	0.16	0.13	-0.07
MCD	**0.87**	0.10	0.05	0.09	0.08	0.28	0.05	0.13	0.07	0.23
CC	**0.86**	0.32	0.15	0.15	0.12	0.09	0.08	0.10	0.16	-0.18
MCC	**0.85**	0.09	0.40	0.08	0.09	0.08	0.02	0.12	0.04	-0.05
PC	**0.84**	0.33	0.13	0.13	0.10	0.13	0.09	0.18	0.10	-0.05
MPC	**0.84**	0.18	0.12	0.35	0.06	0.09	0.05	0.12	0.15	-0.22
RPC	0.31	**0.82**	0.22	0.22	0.07	0.14	0.07	0.20	0.16	0.02
MRPC	0.33	**0.77**	0.27	0.19	0.19	0.11	0.07	0.03	0.26	-0.02
LVC	0.24	0.31	**0.90**	0.08	0.11	0.05	0.02	0.06	0.04	0.00
MLVC	0.31	0.32	0.10	**0.85**	0.13	0.13	0.04	0.08	0.13	0.01
ND	0.29	0.16	0.12	0.11	**0.90**	0.19	0.04	0.09	0.09	0.01
MND	0.34	0.17	0.06	0.13	0.21	**0.88**	0.08	0.10	0.08	0.01
NOIF	0.12	0.07	0.02	0.03	0.03	0.06	**0.98**	0.08	0.03	0.00
NOGIF	0.46	0.18	0.08	0.09	0.11	0.11	0.12	**0.83**	0.07	0.01
LOC	0.33	0.44	0.05	0.17	0.12	0.11	0.04	0.08	**0.79**	0.00

Table 5.4. Fault prediction models using multiple linear regression

Firefox	Metrics Data	R	R^2	Adj. R^2	F-test*	MAE	RMSE	RAE%	RRSE%
0.8	Without PCA	0.65	0.43	0.42	161.23	2.59	5.87	78.99	79.71
	With PCA	0.60	0.36	0.36	223.69	2.59	6.41	78.70	86.89
1.0	Without PCA	0.63	0.40	0.40	148.12	2.85	6.62	73.67	80.69
	With PCA	0.61	0.37	0.37	236.32	3.00	7.03	77.73	85.64
1.5	Without PCA	0.74	0.54	0.54	276.29	3.11	6.95	73.79	71.57
	With PCA	0.71	0.51	0.50	431.01	3.33	7.57	79.07	77.98
2.0	Without PCA	0.64	0.41	0.41	165.31	2.35	6.02	89.83	81.66
	With PCA	0.61	0.37	0.37	251.32	2.41	6.34	92.25	86.06

** significant at $p<0.001$*

and the sample population. The adjusted R^2 tends to remain constant as the R^2 measure for large population samples. Whereas, *F-ratio* are commonly used to assessing the models. It is a measure of how much the model has improved the prediction of the outcome compared to the level of inaccuracy of the model. In short a good model must have a large *F-ratio* (at least *F-ratio* should be greater than 1) [Field, 2004]. Furthermore, to analyze the presence of errors in the prediction results of models, we used the metrics MAE, RMSE, RAE and RRSE. We formally described all these metrics in chapter 2. In the following paragraphs we discuss the obtained results of model's evaluation metrics.

Table 5.4 shows the evaluation results of MLR models. These results show that the models built with raw metrics have little larger values of evaluation metrics (i.e., R, R^2, and Adjusted R^2) as compared to the models built with principal components. This difference in the results of two different types of models are not prominent. However, the results of Table 5.4 also show that the value of *F-ratio* is higher for those models which are built using principal components. Therefor, we can conclude that, although the value of R, R^2, and Adjusted R^2 of PCA based models is smaller as compared to those models which are built with raw metrics, but still we can consider PCA based model as better models because its *F-ratio* value is higher.

Table 5.4, shows that the value R range between 0.63 to 0.74 (without PCA) and 0.60 to 0.71 (with PCA). The corresponding values of R^2 range between 0.40 to 0.54 and 0.36 to 0.51 respectively. The R^2 values of two different types of models indicate that our raw metrics explained 54.0% (maximum) of the variance. Principal components explained 51.0% (maximum) of the variance. The R^2 indicates the efficacy of the built regression models. The adjusted R^2 values indicate the lack of bias in our R^2 values. We used the techniques of 10-fold cross validation to evaluate the predictive power of our models. Ten fold cross validation is a technique which is used to divide the whole data set into ten parts, and train the model using only one part of the data set, while the rest of the nine parts of the data used for testing. This process repeats 10 times. Each time model is trained with differnt part of the data and tested on the rest of the data. Since, the value of $R > 0.7$ and $R^2 > 0.5$ is considered as a good value for any model, therefore, we can conclude that our regression models are robust. Till now we discuss the regression power of our MLR models. Now in the next paragraph we discuss the

accuracy of our prediction models.

In order to measure the prediction accuracy of our models, we used four different metrics, which are commonly used to measure the accuracy of a prediction model. The metrics belongs to the two different classes of error, i.e., *Absolute Error* and *Relative Error*. In case of *Absolute Error*, error in prediction value may be represented by the actual amount of error. Therefor, the absolute error of the prediction model shows how large the error actually is. The *Relative Error* is a ratio that compared the error to the size of the actual value. In short, *Relative Error* shows how large the error is in relation to the correct value.

To measure the *Absolute Error* we used two different metrics i.e., the mean absolute error (MAE) and the root mean square error (RMSE). Whereas, to measure the relative error of the models, we used the metrics, relative absolute error (RAE) and the root relative square error (RRSE). The formal description of error metrics described in Chapter 2.

The absolute and relative errors of our models are characterized in the last four columns of Table 5.4. It is shown in the table that the prediction accuracy of models trained with raw data (without PCA) and principal components are almost equal. Table 5.4 shows that the models trained with the raw metrics have different values for each error metrics. Like, MAE lies in the range 2.35 to 3.11. RMSE lies in the range 5.87 to 6.95. RAE lies in the range 73.67% to 89.83% and RRSE lies in the range 71.57% to 81.66%. Similar pattern are observed when models are trained with principal components, i.e., MAE lies in the range 2.41 to 3.33. RMSE lies in the range 6.34 to 7.57. RAE lies in the range 77.3% to 92.25% and RRSE lies in the range 77.98% to 86.89%. The obtained values of MAE and RMSE are small, but the value of relative error, i.e., RAE and RRSE are large. The big issue with the MMRE is its value is strongly influenced by a few very large MRE values [Conte and Shen, 1986].

5.2.5 Fault Predictor Model Using Machine Learning Algorithms

The scatter plots of code metrics (see Figure 5.2) show that most of the code metrics not linearly correlated with the post release defects. Moreover, the distribution of metrics data is also unknown. When the distribution of data is not known, then non-parametric regression commonly used to construct the regression models [Stern, 1996]. Therefor, in this section we described our regression models, which are built using different non-parametric regression methods. The name of these methods: neural networks (NN) or multi layer perceptron, radial basis function (RBF), support vector machine (SMOreg) and REPTree. A brief description about these methods given in Chapter 2.

Each model trained with the source file's metrics data of four different releases of Firefox project. Before training, we transformed the source file's metrics data (also called raw metrics data) into principal components and trained the model with both raw metrics and principal components. Therefore, each model actually trained with eight different data sets. Four of them are raw metrics data and the rest four of them are principal components. We, used Weka tool to perform the experiment. Table 5.5 shows that five different evaluation measures used to evaluate the non parametric regression models. The names of these metrics are, R, MAE, RMSE, RAE and RRSE. In the following paragraph, we

describe these models.

Neural Network (NN): First, we used neural network *back-propagation* method to classify instances. The first step is to train the neural network. The main goal of the training process is to find the set of weight values that will cause the output from the neural network to match the actual target values as closely as possible. There are several issues involved in designing and training a multilayer perceptron network. Some of the important issues are given below,

1. **Selecting how many hidden layers to use in the network:** In most of problems, one hidden layer is sufficient. Using two hidden layers rarely improves the model, and it may introduce a greater risk of converging to a local minima (a point on error space, for details read [Bishop, First Edition, October 2007]).

2. **Deciding how many neurons to use in each hidden layer:** One of the most important characteristics of a perceptron network is the number of neurons in the hidden layer(s). If an inadequate number of neurons used, the network will be unable to model complex data, and the resulting fit will be poor. If too many neurons are used, the training time may become excessively long, and worse, the network may over fit the data. When over fitting occurs, the network will begin to model random noise in the data. The result is that the model fits the training data extremely well, but it generalizes poorly to new, unseen data. Validation must be used to test for this.

3. **Avoids Local Minima:** To find a globally optimal solution one should avoid the presence of local minima. Several methods have been tried to avoid local minima. The simplest is just to try a number of random starting points on the error surface and use the one with the best value. A more sophisticated technique called *simulated annealing* improves on this by trying widely separated random values and then gradually reducing (*cooling*) the random jumps in the hope that the location is getting closer to the global minimum. For details read the following book [Bishop, First Edition, October 2007].

4. **Converging Time:** Another design issue is to design a neural network, which converge to an optimal solution in a reasonable period of time. Mostly *conjugate gradient algorithm* is used for this purpose, which optimize the weight values in numbers of cycles or iterations. Each cycle or iteration is called an epoch.

5. **Learning Rate:** The training of neural network using an algorithm such as in our case error back-propagation usually requires a lot of time on large and complex problems. Such algorithms typically have a learning rate parameter that determines how much weights can change in response to an observed error on the training set. The choice of this learning rate can have a dramatic effect on generalization accuracy as well as the speed of training [Wilson and Martinez, 2001].

In order to design an optimal neural network for our experiment, we considered all the above mentioned issues. We used *back-propagation* method of neural network and designed multiple neural networks using one and two hidden layers. In each NN, we used different combinations of perceptrons per hidden layer. In case of single hidden layer, we designed two different neural networks. In the first neural network, we used 4 perceptron in the single hidden layer, whereas, in the second designed

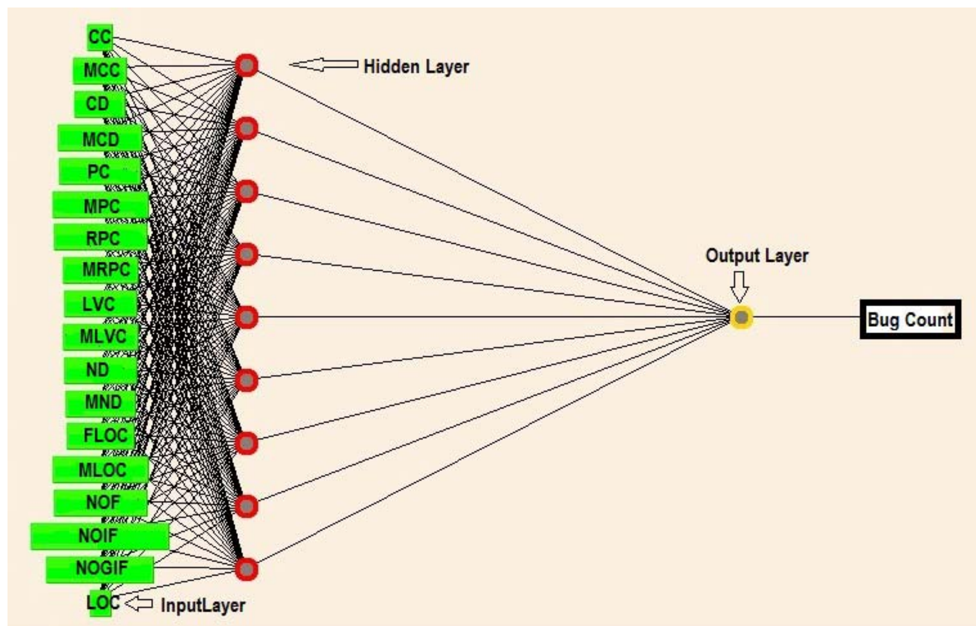


Figure 5.3. Training of fault prediction model using single hidden layer neural network. The training data set is labeled with isbuggy (isbuggy=yes or isbuggy=no) criteria.

we used eight perceptrons in the single hidden layer. Similarly, we designed two more neural networks with two hidden layers. In the first, two hidden layers neural network, we used four perceptrons in the first hidden layer and two perceptrons in the second hidden layer. Whereas, in case second, two hidden layers neural networks, we used eight perceptrons in first hidden layer, and four perceptrons in second hidden layer. Furthermore, for each neural network, we set number of epoch 500 and used different values for learning rate i.e., 0.3, 0.2, 0.1 and 0.01. Figure 5.3 shows the designed of single hidden layer neural network, while Figure 5.4 is an example of two hidden layer perseptron.

Once we designed the four different neural networks, then we trained each of them using eight data sets. We found the best performance of a neural network when we used a single hidden layer with eight perceptron (see Figure 5.3). The evaluation measures of neural network (single layer with eight perceptrons) is shown in Table 5.5, which shows that the maximum value of R (prediction correlation with the actual value) is 0.68 or 68%, whereas the associated error values are 3.29 MAE, 7.11 RMSE, 78.17% RAE and 73.26% RRSE. These metrics values are obtained when NN trained with the raw metrics data. When the same NN trained with principal component the maximum obtained value of R is 0.6166 or 61.66%, and the associated value error metrics are 3.28 MAE, 7.65 RMSE, 78.07% RAE and 78.81% RRSE. The minimum value of R is 0.564 or 56.4% (when trained with raw metrics data), and 0.483 or 48.3% (when trained with principal components). It is also shown in Table 5.5, that the variation in the evaluation measures is not very large, when models are trained with raw metrics or principal components. Its means that most of the predictors used in our models have small correlation with other predictors, and therefore, pca is less significant in our experiment. This is the main reason, that the value of evaluation measures of pca based model is almost equal to the evaluation measures of the models which are trained with raw metrics data. Another possibel reason is, the estimated evaluation measures of our model may be over estimated by those models

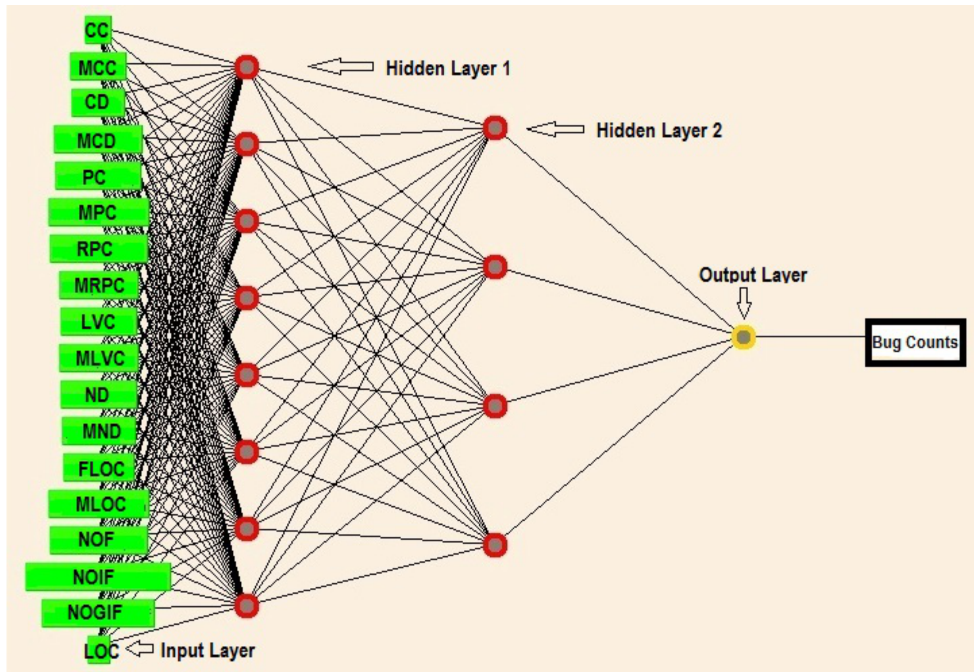


Figure 5.4. Training of fault prediction model using double hidden layers of neural network. The training data set is labeled with isbuggy (isbuggy=yes or isbuggy=no) criteria.

which are trained with the raw metrics data. The estimated measure relaibel, when models are trained with principal components. Finally, on the basis of the high value of R and the small absolute errors, we can conclude that neural network can be used as a fault prediction model for software.

Radial Basis Function (RBF) Network: A Radial Basis Function (RBF) Network is a *feedforward* network. It has an input layer, a hidden layer and an output layer. The neurons in the hidden layer contain Gaussian transfer functions whose outputs are inversely proportional to the distance from the center of the neuron. To build the model using RBF, we used WEKA that implements a normalized Gaussian radial basis function network. It uses the k-means clustering algorithm to provide the basis functions and learns a linear regression on top of that. To design an optimal RBF network, we set the input parameter i.e., number of clusters = 12 for each model.

The evaluation measures of RBF network are shown in Table 5.5. First we trained the RBF Network models using raw metrics data. The obtained value of R range from 0.464 to 0.59, MAE range from 2.15 to 3.09, RMSE range from 6.25 to 7.85, RAE range from 73.38% to 82.41% and RRSE range from 80.94% to 88.68%. The evaluation measures when models are trained with principal components are, R range from 0.477 to 0.61, MAE range from 2.11 to 3.02, RMSE range from 6.47 to 7.71, RAE range from 71.55% to 80.83% and RRSE range from 79.42% to 87.90%. From these results, it is observed that models performed well when trained with principal components.

Support Vector Machine (Sequential Minimal Optimization SMOreg): To build the regression model using support vector machine we used WEKA tool, which provides SMOreg algorithm to implement the support vector machine for regression. Furthermore, WEKA provides various

algorithms to learn the parameters. To design the SMOreg model for our experiment, we selected *RegSMOImproved* algorithm for *RegOptimizer* (For more details, read [Ian H. Witten and Kaufmann, 2005]).

The evaluation measures of SMOreg are shown in the fifth column of Table 5.5. First we trained the models using raw metrics data. The obtained value of R range from 0.55 to 0.67, MAE range from 1.65 to 2.56, RMSE range from 6.6 to 8.06, RAE range from 59.74% to 63.31% and RRSE range from 83.03% to 97.53%. The obtained evaluation measures when models are trained with principal components are, R range from 0.51 to 0.63, MAE range from 1.64 to 2.65, RMSE range from 6.81 to 8.63, RAE range from 60.01% to 68.72% and RRSE range from 88.47% to 99.0%. From these results, we can conclude that the support vector machine (i.e., SMOreg) can be used to build the fault prediction model with 67% correlation.

Reduced Error Pruning Trees(REPTrees): In order to analyze the prediction capability of decision tree learner algorithm, we used REPTrees to build the fault prediction models. It is a fast tree learner that uses reduced error pruning. REPTree is actually a mixture of decision tree and linear regression, where each leaf node corresponds to a linear regression algorithm. REPTree algorithm is available in WEKA. In order to design an optimal model using REPTree, we used different combinations of the input parameters of the models. Finally, we selected a set of input parameters, which built an optimal REPTree model for fault prediction. An example of two folds REPTree is shown in Figure 5.5. Following is the list of important parameters, which were changed to train the model.

1. We used 2, 4,and 6 numbers of instances per leaf.
2. The number of folds for reduced error pruning set to 2,3,4 and 5.
3. The number maximum tree depth set to 2, 4, and 6.

The last column of Table 5.5 depicts the value of the model evaluation measure. First, we trained the models using raw metrics data. The obtained value of R range from 0.46 to 0.67, MAE range from 2.03 to 2.86, RMSE range from 6.31 to 7.35 RAE range from 67.81% to 79.35% and RRSE range from 83.86% to 97.53%. The obtained evaluation measures when models are trained with principal components are, R range from 0.3716 to 0.63, MAE range from 1.64 to 2.65. RMSE range from 2.75 to 8.04, RAE range from 72.43% to 86.68% and RRSE range from 82.89% to 86.68%. Since, the maximum obtained value of R is 0.67, its mean that the correlation between the actual and predicted values is large. Therefore, we can conclude that REPTree can be used to build the fault prediction models.

5.2.6 Comparison of Regression based Fault Prediction Models

In order to evaluate the performance of regression models, we used both parametric and non-parametric regression techniques. We, built five different models using the raw metrics data and the principal components. The evaluation metrics of each model has been computed. The comprehensive analysis of these computed metrics leads to the following conclusions:

Table 5.5. Evaluation of fault predictor models obtained using raw metrics data i.e., without PCA

Firefox	Eval. Measures	Metrics Data	Fault Prediction Models			
			NN	RBF	SMOreg	REPTree
Rel. 0.8	R	Without PCA	0.6423	0.5308	0.5503	0.4681
		With PCA	0.523	0.5247	0.5127	0.4157
	MAE	Without PCA	2.4712	2.6131	2.1016	2.609
		With PCA	2.6383	2.5649	2.0877	2.7544
	RMSE	Without PCA	5.6989	6.2579	6.6906	6.5555
		with PCA	6.2907	6.2952	6.8117	6.7698
	RAE	Without PCA	75.16%	79.47%	63.92%	79.35%
		With PCA	80.24%	78.01%	63.49%	83.77%
	RRSE	Without PCA	77.28%	84.86%	90.72%	88.89%
		With PCA	85.30%	85.36%	92.37%	91.80%
Rel. 1.0	R	Without PCA	0.5681	0.494	0.5938	0.5465
		with PCA	0.5148	0.5278	0.5446	0.4517
	MAE	Without PCA	2.9926	3.0747	2.5658	2.8404
		With PC	3.0729	3.0239	2.6566	3.0922
	RMS	Without PCAE	6.8024	7.1399	6.9534	6.8846
		With PCA	7.0407	6.9764	7.2631	7.5162
	RAE	Without PCA	77.42%	79.54%	66.38%	73.48%
		With PCA	79.49%	78.23%	68.72%	79.99%
	RRSE	Without PCA	82.86%	86.97%	84.69%	83.86%
		With PCA	85.76%	84.97%	88.47%	91.55%
Rel. 1.5	R	Without PCA)	0.6887	0.5905	0.6784	0.6748
		With PCA	0.6166	0.6083	0.6346	0.5615
	MAE	Without PCA	3.2938	3.0918	2.5172	2.8573
		With PCA	3.2895	3.0145	2.5285	3.0517
	RMSE	Without PCA	7.114	7.8594	8.0623	7.3013
		With PCA	7.653	7.7114	8.6353	8.0484
	RAE	without PCA	78.17%	73.38%	59.74%	67.81%
		With PCA	78.07%	71.55%	60.01%	72.43%
	RRSE	Without PCA	73.26%	80.94%	83.03%	75.19%
		With PCA	78.81%	79.42%	88.93%	82.89%
Rel. 2.0	R	Without PCA	0.564	0.4643	0.5552	0.5277
		with PCA	0.4837	0.477	0.5234	0.3716
	MAE	Without PCA	2.1763	2.1581	1.658	2.0278
		With PCA	2.2087	2.1166	1.6403	2.27
	RMSE	Without PCA	6.1169	6.5376	7.1897	6.3103
		with PCA	6.4572	6.4795	7.2981	6.8761
	RAE	Without PCA	83.11%	82.41%	63.31%	77.43%
		With PCA	84.34%	80.83%	62.64%	86.68%
	RRSE	Without PCA	82.98%	88.68%	97.53%	85.60%
		With PCA	87.59%	87.90%	99.00%	93.27%

```
nof < 198
|   numberofincludedfiles < 28.5
|   |   numberofincludedfiles < 10.5
|   |   |   loc < 103.5 : 0.18 (733/1.7) [752/1.24]
|   |   |   loc >= 103.5
|   |   |   |   totallocvar < 101.5
|   |   |   |   |   numberofincludedfiles < 5.5 : 0.72 (314/4.66) [320/4.53]
|   |   |   |   |   numberofincludedfiles >= 5.5 : 1.74 (344/11.38) [295/11.05]
|   |   |   |   totallocvar >= 101.5 : 4.43 (15/39.4) [12/19.19]
|   |   numberofincludedfiles >= 10.5
|   |   |   maxcdensity < 33.5
|   |   |   |   totalrpoints < 30.5
|   |   |   |   |   totalnestdepth < 10.5 : 0.97 (87/2.78) [86/5.24]
|   |   |   |   |   totalnestdepth >= 10.5 : 2.26 (146/15.81) [175/12.17]
|   |   |   |   totalrpoints >= 30.5
|   |   |   |   |   nof < 22.5 : 4.42 (39/42.95) [30/31.92]
|   |   |   |   |   nof >= 22.5
|   |   |   |   |   |   totalcdensity < 703.5 : 4.26 (119/27.54) [129/36.86]
|   |   |   |   |   |   totalcdensity >= 703.5 : 7.83 (4/30.69) [13/318.49]
|   |   |   maxcdensity >= 33.5 : 6.41 (37/70.41) [28/52.58]
|   numberofincludedfiles >= 28.5
|   |   loc < 64
|   |   |   numberofincludedfiles < 63.5 : 18.57 (4/219) [4/859]
|   |   |   numberofincludedfiles >= 63.5 : 102.5 (2/56.25) [0/0]
|   |   loc >= 64
|   |   |   totalrpoints < 218 : 9.98 (98/103.31) [95/269.34]
|   |   |   totalrpoints >= 218 : 33.08 (11/422.88) [13/1005.07]
nof >= 198 : 70 (3/116.67) [5/2284.2]
```

Figure 5.5. Training of fault prediction model using REPTree with two folds. The tree is built with metrics data, where nof=NOF, numberOfIncludedfiles=NOIF, totalrpoints=RPC, maximumcdensity=MCD, totallocvar=LVC and loc=LOC (line of codes)

1. In our experiment, the prediction results of PCA based models are almost same as compared to the results of non-PCA based models. The possible reasons are: principal components normally perform well when the large number of predictors used to build the model, and most of the predictors are correlated with each other. Whereas, in our experiment, we used small numbers of predictors, i.e., 18, and only few of them are found to be correlated with other predictors (see Table 5.2 and Table 5.3). Therefore, we can conclude that, if the numbers of predictors are in small numbers, and have a weak correlation with other predictors, then one can build the fault prediction models without using principal components. Moreover, in our experiment, we also observed that the difference in the evaluation measure is almost same, when the models are trained with raw metrics and principal components, therefor, it would be better to use principal components to build the prediction models. Because, principal components are used to avoid over estimation from the prediction result, which is normally present when the prediction models have correlated predictors.

2. From the results of Table 5.5 and Table 5.4, it may be observed that the maximum value of correlation coefficient (i.e., R) is 0.74, which is obtained when the model is trained using MLR (multiple linear regression) technique. Whereas, the other maximum values of R are 0.68 (NN), 0.678 (SMOreg), 0.674 (REPTree), and 0.59 (RBF). The mean absolute error values i.e MAE in ascending order are 1.65 (SMOreg), 2.02 (REPTree), 2.11 (RBF), 2.17 (NN), and 2.35 (MLR). The RMSE values in ascending order are, 5.69 (NN), 5.87 (RMSE), 6.31 (REPTree), 6.47 (RBF), and 6.69 (SMOreg). The relative absolute error values in ascending order are, 59.74% (SMOreg), 67.81% (REPTree), 71.55% (RBF), 73.67% (RAE), and 75.16% (NN). The RRSE values in ascending order are, 71.52 (MLR), 73.26% (NN), 75.19% (REPTree), 79.42%

SMOreg and 83.03%. From these observations the minimum MAE value is 1.65 (SMOreg). The mimimum value of RMSE is 5.69 (NN). The mimimum relative error, i.e., RAE is 59.74% (SMOreg). The minimum RRMSE value is 71.57 (MLR). On the basis of these results, we can conclude that the best regression models in desending order are, MLR, NN, SMOreg, REPTree and RBF.

5.3 Fault Prediction Models Using Classification Techniques

In this section, we describe our approach to construct a fault prediction model using classification techniques of machine learning. In case of classification based fault prediction model, a classification schema is used to build the model. Therefore, in order to provide a predictor to classify source files according to a pre-defined classification schema, we compare two classification schemes. The first schema allows classifying files as being buggy or not. The second schema is called bug level classification where files are classified as low buggy, medium buggy, or highly buggy respectively.

- *isBuggy*: If a file contains no bugs, its *isBuggy* value is set to “no”, otherwise the value is set to “yes”.
- *bugLevel*: We define three bug levels based on the number of bugs in each file as follows:

Number of Bugs	Bug Level
0-10	low
10-30	medium
30 and above	high

In our work, we used different machine learning techniques. The name these techniques are logistic regression (LR), decision tree J48 (DT), support vector machine (SMO), radial basis function (RBF), naive bayes (NB), boosting and bagging. The approach is to use these techniques for automatically obtaining a classifier from one version of a program. In order to judge the quality of the obtained classifier, we used other versions of the same program in order to test the classifier. In particular, we used the code metrics data of Firefox 0.8 for extracting the classifier, and versions 1.0, 1.5, and 2.0 to test them. We used principal components to train the classifiers, built using classification schema *isBuggy* and *bugLevel*. Furthermore, to perform a comparative study, we also used raw metrics data (without PCA) to train the classifiers, built using classification schema *isBuggy*. In the following subsection, we describe our methodology which we used to build the fault prediction model.

5.3.1 Point Biserial Correlation

The first step to build the software fault prediction model is to find the correlation between the predictors (like code metrics) and the dependent variable (number of faults). In Section 5.2.1, we obtained the sparemann and pearson correlation between the predictors and the number of faults. The correlation results shows (see Table 5.1) that most of the predictors are highly correlated with the number of faults. Now, in this section we find the point biserial correlation, because in case of classification the number of faults, i.e., dependent variable is discrete variable. If the dependent

Table 5.6. Point Biserial Correlation between Code Metrics and bug level

Code Metrics	Firefox Rel 0.8
NOF	**0.437***
FLOC	**0.466***
MFLOC	**0.401***
CD	**0.431***
MCD	0.385*
CC	**0.438***
MCC	0.386*
PC	**0.439***
MPC	0.357*
RPC	**0.447***
MRPC	0.335*
LVC	**0.421***
MLVC	0.365*
ND	**0.444***
MND	0.341*
NOIF	**0.462***
NOGIF	0.115*
LOC	**0.459***

Correlation is significant at the 0.01 (2 tailed)

variable is discrete (i.e., *true* or *false*) then we cannot use traditional correlation techniques, i.e., sparemann and pearson correlation. Whereas, point biserial correlation coefficient is a correlation technique, used when one variable is discrete or dichotomous. It measures the association between a continuous variable and a discrete variable [Field, 2004]. It can take values between -1 and +1. In the following paragraph, we describe the formal description of point biserial correlation.

Assuming X as a continuous variable and Y as categorical with values 0 and 1, point biserial correlation can be calculated using the following formula,

$$r = \frac{(\bar{X_1} - \bar{X_0})\sqrt{p(1-p)}}{S_x} \tag{5.1}$$

Where $\bar{X_1}$ is the mean of X when Y=1 , $\bar{X_0}$ is the mean of X when Y=0, S_x is the standard deviation of X, and p is the proportion of values where Y=1. Positive point biserial correlation indicates that large values of X are associated with Y=1 and small values of X are associated with Y=0. In our experiment, the results of point biserial correlation is shown in Table 5.6. The results of Table 5.6 show that, correlation value range from 0.115 to 0.466. In case of software fault prediction model, correlations over 0.4 are considered as strong correlation [Ohlsson and Alberg, 1996].

5.3.2 Experimental Setup to Build The Models

We used WEKA (a Machine Learning tool developed in JAVA)[3] and SPSS tool (statistical tool) [Field, 2004] for obtaining a classifier for source files based on the number of bugs and the obtained complexity metrics. Eight different machine learning approaches are used to obtain different classifiers. The selection was based on the research literature available about the performance of these approaches. We selected eight approaches for our study in order to find the most suitable classifier. The brief description of all these models is given in Chapter 2. All the selected classifiers were trained on metrics and bug data of Firefox Release 0.8 according to both the *isBuggy* and the *bugLevel* criteria. After training of classifiers WEKA can store models in the form of Java serializable objects holding different options. These models were afterwards tested in order to classify the source files of later Firefox releases.

In order to perform the experiment using machine learning classifiers, the first step is to design the classifiers model using input parameters. The optimum value of all these parameters, are not known. Therefore, one has to use different combinations of these parameters to find the optimum design of the model. In the following paragraph, we describe some of the designed parameters of our classification models.

In case of *Decision Tree (J48)* WEKA provides a class i.e., weka.classifiers.trees.J48 for generating a pruned or unpruned C4.5 decision tree. Some of the important input parameters of J48 are *confidenceFactor*, *numFolds*, *minNumObj* and *unpruned*. By setting the different value of *confidenceFactor*, we can control the degree of pruning i.e., minimal to aggressive. *numFolds* specifies the number of subsets into which the training data is divided. One fold is reserved as a validation set and the remaining folds are used for training/building a tree. By changing the number of folds, we can control the portion of the data used for training. A small number of folds make the validation set larger, whereas, a large number of folds make the validation set smaller. The parameter *minNumObj* is used to set the minimum number of instances per leaf. *unpruned* is used to set the pruning option true or false.

For logistic regression (LR) WEKA provides a class, i.e., weka.classifiers.functions.Logistic. To train the LR classifiers, we used different values of parameter *maxIts*, which is used to set the value of maximum number of iteration. To train the classifier with boosting algorithm, we used WEKA class *AdaBoostM1* (weka.classifiers.meta.AdaBoostM1). In case of *AdaBoostM1* the parameter *classifier* is used to provide the name of base classifier. We used the following base classifiers DecisionStump, RandomForest and REPTree. For bagging algorithem, we used WEKA class *Bagging* (weka.classifiers.meta.Bagging). *Bagging* need a base classifier, we used DecisionStump, RandomForest and REPTree as base classifier. Similarly, in case of neural network, we used WEKA class *MultilayerPerceptron* (weka.classifiers.functions.MultilayerPerceptron). In case of *MultilayerPerceptron*, we used different combinations of numbers of the hidden layers and the number of perceptrons per hidden layers. Furthermore, we set different values of learning rate, i.e., 0.3, 0.2, 0.1 and 0.01. Figure 5.6 illustrate an example of neural network, which is designed to classify a source file on the basis of *bugLevel*. After obtaining optimal parameters of each classifier, we trained and tested all the classifiers using optimal parameters. The obtained results of each classifiers are discussed in the

[3]http://www.cs.waikato.ac.nz/ml/weka/

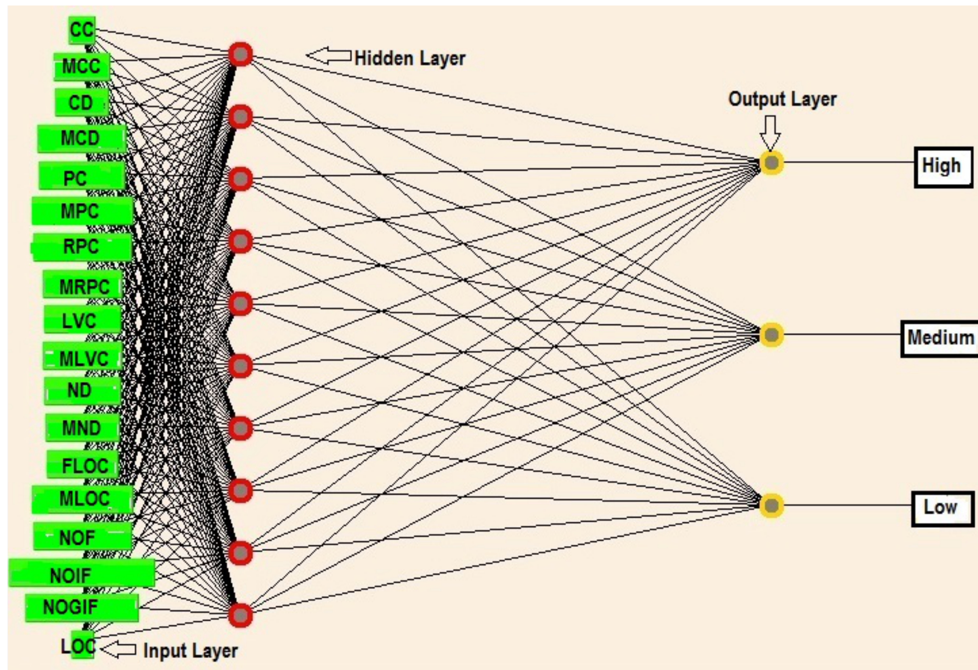


Figure 5.6. Training of fault prediction models using Neural Network. The training data set is labeled with buglevel (low, medium and high) criteria.

Table 5.7. Class distribution for the classification schema *isBuggy* (only C++ source files)

isBuggy	Firefox 0.8	Firefox 1.0	Firefox 1.5	Firefox 2.0	Total	Avg.
YES	1053 (26.91%)	2296 (57.15%)	842 (19.97%)	1036 (24.22%)	5227	31.83%
NO	2860 (73.08%)	1721 (42.8%)	3374 (80.02%)	3240 (75.77%)	11195	68.17%
Total	3913	4017	4216	4276	16422	

following section.

5.3.3 Classification Results and Discussion

The class distribution of each release of Firefox is shown in the Table 5.7. This distribution is obtained using classification schema *isBuggy*. It is shown in the table that in case of Firefox release 0.8, the percentage of the faulty source files is 26.91% and the percentage of the clean source files is 73.08%. Moreover, the average class distribution of four releases shows that 31.83% of the source files are faulty and 68.17% of the source files are clean or bug free. Similarly, Table 5.8 shows the class distribution according to the classification schema *bugLevel*. It is shown in the Table 5.8 that in case of Firefox 0.8, the percentage of the class distribution i.e., *LOW*, *MEDIUM*, and *HIGH* are 93.43%, 04.06% and 02.50% respectively. Whereas, the average class distributions of four different releases are 88.51% *LOW*, 04.32 *MEDIUM*, and 07.16% *HIGH*.

These results show that in our data set the class distribution is skewed in both the cases, i.e.,

Table 5.8. Class distribution for the classification schema *bugLevel* (only C++ source files)

bugLevel	Firefox 0.8	Firefox 1.0	Firefox 1.5	Firefox 2.0	Total	Avg.
LOW	3656 (93.43%)	3437 (85.56%)	3885 (92.15%)	3558 (83.12%)	14536	88.51%
MEDIUM	159 (04.06%)	240 (05.97%)	217 (05.15%)	94 (02.19%)	710	04.32%
HIGH	98 (02.50%)	340 (08.46)	114 (02.70%)	624 (14.59%)	1176	07.16%
Total	3913	4017	4216	4276	16422	

when we used classification schema *isBuggy* or *bugLevel*. In case of machine learning classification problems, the metrics *Accuracy* can be used as a reliable evaluation metrics, if the class distribution among examples (data instances) is not skewed. However, in the real world this is rarely the case that class distribution equal or uniform. It is observed that the unusual and interesting class is rare among the general population [Provost and Fawcett, 1997]. Since, in our experiment the class distribution is skewed. Therefore, we cannot evaluate machine learning classifier only on the basis of evaluation measure *Accuracy*. In order to evaluate each class, following evaluation metrics have been used: *Precision*, *Recall*, *Fmeasure* and *ROC Area*. Moreover, to evaluate the overall prediction accuracy of each classifier, following metrics have been used: absolute error (MAE and RMSE), relative error (RAE and RRSE), Kappa statistics (KS) and Accuracy. In the following, we first discuss the obtained results of the classifiers, trained with *isBuggy* classification schema. Then, we discuss the obtained results of the classifiers, which are trained with *bugLevel* classification schema.

Classification of Source Files Using Classification Schema isBuggy

Table 5.9 shows the results of different classifiers, when classifiers trained and tested using the data set of Firefox 0.8, whose instances are distributed using *isBuggy* criteria. This is shown in the table that in case of raw metrics data (non PCA), the evaluation measures for the class *isBuggy*=yes have the following values: precision range from 57.4%(DT) to 72.3%(SMO), recall range from 19.4%(DT) to 56.1%(AdaBoost), Fmeasure range from 30.6%(DT) to 57.0%(AdaBoost), and ROC area range from 0.583(SMO) to 0.831(Bagging). Whereas, in case of principal components (with PCA), the evaluation measures for the class *isBuggy*=yes have the following values: precision range from 55.8%(DT) to 70.2%(SMO), recall range from 15.0%(DT) to 60.6%(AdaBoost), Fmeasure range from 24.7%(DT) to 56.9%(AdaBoost), and ROC area range from 0.563(SMO) to 0.803(NN). The evaluation measures i.e., precision, recall and Fmeasure for the class *isBuggy*=no are ranged from 76.6% to 97.7%. In our experiment our interest is to find those classifiers which can classify faulty source files with high precision and recall. Therefore, in this experiment, classifiers are compared on the basis of their ability to classify the class *isBuggy*=yes (faulty source files). Furthermore, In our experiment the evaluation measures are almost same when classifiers trained with raw metrics or principal components. But, the results which are obtained using principal components are considered as more reliable, because PCA based classification results are not overestomated. Therefore, in order to compare the performance of the classifiers, we considered only PCA based evaluation measures.

From the obtained results of Table 5.9, the list of classifiers according to the high precision, high

recall, and high precision and high recall values are given below,

Classifier's Name	**High Precision**	Recall
Support Vector Machine (SMO)	70.2%	15.0%
Logistic Regression (LR)	65.2%	35.0%
Naive Bayes (NB)	62.7%	27.4%
Neural Network (NN)	61.2%	36.5%
Radial Basis Function (RBF)	59.6%	36.7%
Bagging	56.0%	45.9%
Decision Tree (J48)	55.8%	46.0%
Boosting (AdaBoost)	53.7%	60.6%

Classifier's Name	Precision	**High Recall**
Boosting (AdaBoost)	53.7%	60.6%
Decision Tree (J48)	55.8%	46.0%
Bagging	56.0%	45.9%
Radial Basis Function (RBF)	59.6%	36.7%
Neural Network (NN)	61.2%	36.5%
Logistic Regression (LR)	65.2%	35.0%
Naive Bayes (NB)	62.7%	27.4%
Support Vector Machine (SMO)	70.2%	15.0%

Classifier's Name	**High Precision**	**High Recall**
Boosting(AdaBoost)	53.7%	60.7%
Bagging	56.0%	45.9%
Decision Tree (J48)	55.8%	46%

The above distribution of the classifier's evaluation results, shows that the highest value of precision is 70.2% when support vector machine (SMO) is used as classifier. But, this precision is achieved at the cost of very low recall and Fmeasure i.e., 15% and 24.7% respectively. Furthermore, the value of ROC area is 0.563, which is close to 0.5. When, ROC area equal to 0.5, then classifier is considered as a random classifier, because its predictions are random. Therefore, in our experiment SMO cannot be considered as a good classifier for fault predictions. The highest recall value is 60.6% when AdaBoost is used as classifier. At this recall value, the precision value of AdaBoost is equal to 53.7. Whereas, the value of Fmeasure and ROC area are 56.9% and 0.79 respectively. Although the precision of AdaBoost classifier is not very high but its recall value is very high. Furthermore, its ROC area and Fmeasure are also acceptable. Therefore, we can consider AdaBoost, as good classifier for fault prediction. Finally, on the basis of high-precision and high recall values, the top three classifiers are Boosting, Bagging and Decision Tree (J48). The Fmeasure values of these top three classifiers are ranged from 50.4 to 56.9, and the values of ROC area are ranged from 0.742 to 0.796.

Table 5.9. Evaluation measures (Precision, Recall, F-measure, and ROC Area) of different classifiers. These classifiers are trained and tested using the source files metrics and fault data of Firefox Release 0.8

Classifier	Class	Precision		Recall		Fmeasure		ROC Area	
		Without PCA	With PCA	Without PCA	With PCA	Without PCA	With PCA	Without PCA	With PCA
LR	no	80	79.5	92.6	93.1	85.9	85.8	0.819	0.818
	yes	65	65.2	37.1	34.9	47.3	45.5	0.819	0.818
	Avg	76	75.7	77.7	75.5	75.5	75	0.819	0.818
DT	no	83.6	81.3	85	86.6	84.3	83.9	0.723	0.742
	yes	57.4	55.8	54.8	46	56	50.4	0.723	0.742
	Avg	76.6	74.4	76.9	75.6	76.7	74.9	0.723	0.742
SMO	no	76.6	75.7	97.3	97.7	85.7	85.3	0.583	0.563
	yes	72.3	70.2	19.4	15	30.6	24.7	0.583	0.563
	Avg	75.5	74.2	76.3	75.4	70.09	69	0.583	0.563
RBF	no	79.9	79.6	90.1	90.8	84.6	84.8	0.796	0.8
	yes	58.5	59.6	34.9	36.7	43.7	45.4	0.796	0.8
	Avg	73.6	74.2	75.8	76.3	73.6	74.2	0.796	0.8
NN	no	81.1	79.6	91.4	91.5	86	85.2	0.824	0.803
	yes	64	61.2	42.2	36.5	51	45.7	0.824	0.803
	Avg	76.6	74.7	78.2	76.7	76.5	74.5	0.824	0.803
Naive Bayes	no	79.2	77.9	91.6	94	85	85.2	0.795	0.764
	yes	60.4	62.7	34.9	27.4	42.2	38.1	0.795	0.764
	Avg	74.2	73.8	76.3	76.1	74	72.5	0.795	0.764
AdaBoost	no	84	84.8	85	80.8	84.5	82.7	0.809	0.79
	yes	57.9	53.7	56.1	60.6	57	56.9	0.809	0.79
	Avg	77	76.4	77.2	75.3	77.1	75.8	0.809	0.79
Bagging	no	82.9	81.3	89.6	86.7	86.1	83.9	0.831	0.796
	yes	63.8	56	49.8	45.9	55.9	50.4	0.831	0.796
	Avg	77.8	74.5	78.9	75.7	78	74.9	0.831	0.796

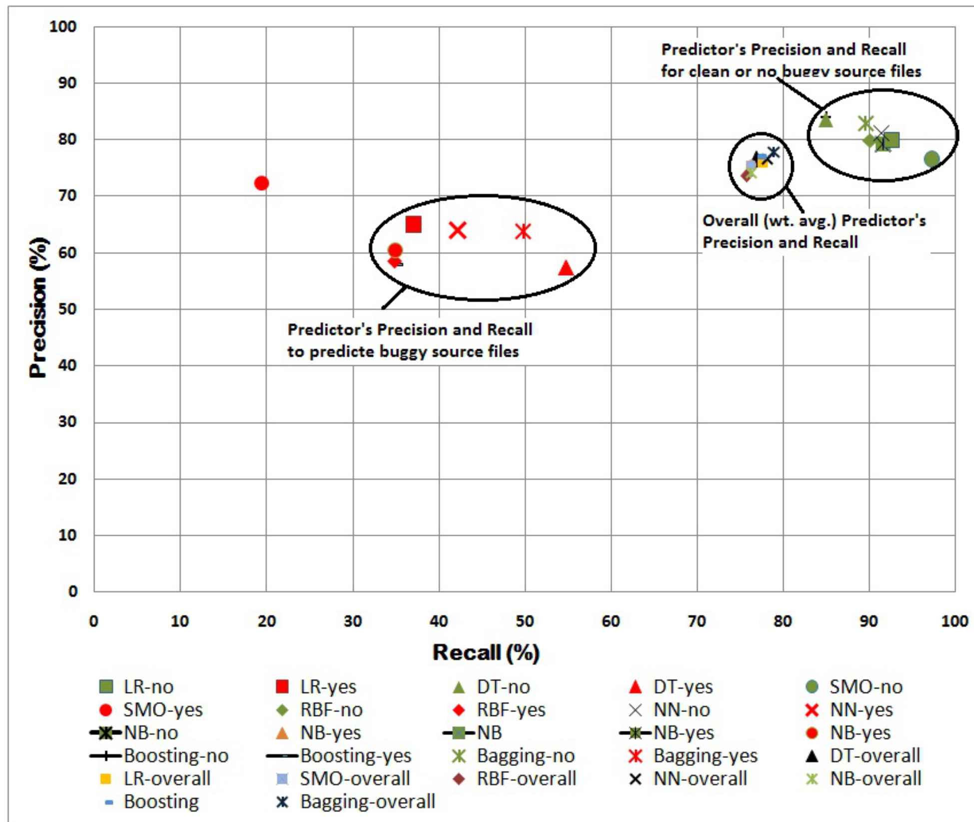


Figure 5.7. Precision-Recall values of individual labels (yes, no), obtained when classifiers trained with different machine learning algorithms

Figure 5.7, depicts the precision and recall values of each class, when the class distribution is based on *isBuggy*. Furthermore, Figure 5.7 depicts the overall, i.e., weighted average of precision and recall. The red color markers at the center of the figure represent the precision and recall values of class *isBuggy*=yes (faulty source file). The green color markers at the top right corner represent the class *isBuggy*=no. Whereas, the markers lies in between green and red markers are the aggregated average of both the class. From Figure 5.7, one can conclude that the *isBuggy*=yes, has small precision and recall value as compared to class *isBuggy*=no. This difference is because our class distribution is skewed. The ideal condition for any classifier is, both classes (i.e., *isBuggy*=no and *isBuggy*=yes) should have their precision and recall values at the top right corner. It is shown in the Figure 5.7 that the classifiers bagging, boosting, neural network and decision tree more closed to the top right corner as compared to the other classifiers.

Furthermore, the performances of the classifiers can be compared directly by visualizing *Precision-Recall* curve and *Receiver Operating Characteristics* (ROC) curve. Moreover, the metrics accuracy, kappa statistics and error metrics, i.e., MAE, RMSE, RAE and RRSE can be used to evaluate the prediction accuracy of each classifier. Now, in the following paragraph we first discuss the *Precision-Recall* curve and ROC curve of each classifier, and then discuss the prediction accuracy of each classifiers.

A Precision-Recall (PR) curve captures the trade off between accuracy and noise as the detector threshold varies. The curve is hyperbolic in shape and the desired operating point is near the upper right. In our experiment, the obtained Precision-Recall curves of all the classifiers are shown in Figure 5.8. Each curve is obtained at different threshold value. Figure 5.8 depicts that all the classifiers follow the same patterns. Initially, classifiers have high precision values at low recall values, and then precision value start decreasing as the recall value start increasing at different threshold. The ideal situation is that, precision and recall should have direct relation, but empirical studies of retrieval performance have shown a tendency for precision to decline as recall increases[Buckland and Gey, 1994]. It is shown in the figure that at different recall values the precision values of boosting, decision tree (J48), RBF and NB are smaller than the precision values of LR, SMO, NN and Bagging.

The ROC curve is used to characterize the trade-off between true positive rate and false positive rate. A classifier, which has a large area under the curve is preferable over a classifier with a smaller area under the curve. A perfect classifier provides an AUC or ROC area that equals 1. The advantages of the ROC analysis are its robustness toward imbalanced or skewed class distributions. Furthermore, ROC provides a way of looking at the performance of a classifier, which is independent of any particular decision threshold. Since, in our experiment the class distribution is skewed (see Table 5.7 and Table 5.8), therefore we obtained the value of AUC or ROC area to evaluate the performance of different classifiers. Figure 5.9 depicts the ROC curves of different classifiers. It is shown in the figure that all classifiers have almost similar ROC curve. In our experiment, the classifiers LR, NN and Bagging, have a larger ROC area, i.e., ≥ 0.8, as compared to other classifiers.

The classification evaluation measures, i.e., *Accuracy* and *Kappa Statistics* are given in Table 5.10. The value *Accuracy* lies in the range 75.4%(SMO) to 77.5%(LR), which shows that all the classifiers have high prediction accuracy. In case binary classifiers, if the class distribution is skewed, then it is

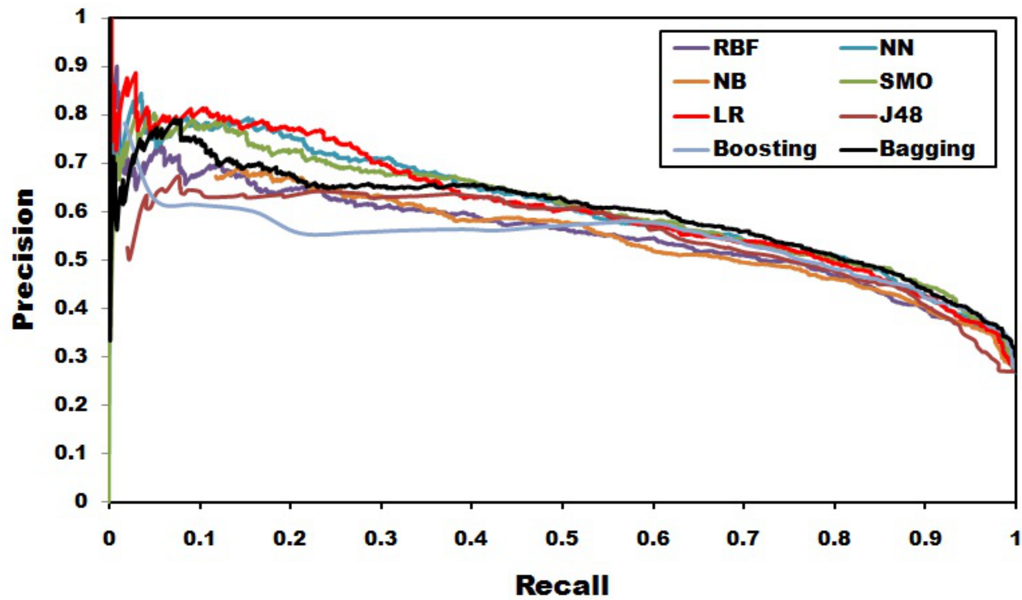


Figure 5.8. Precision-Recall Curve of different classifiers at different threshold value of the classifier.

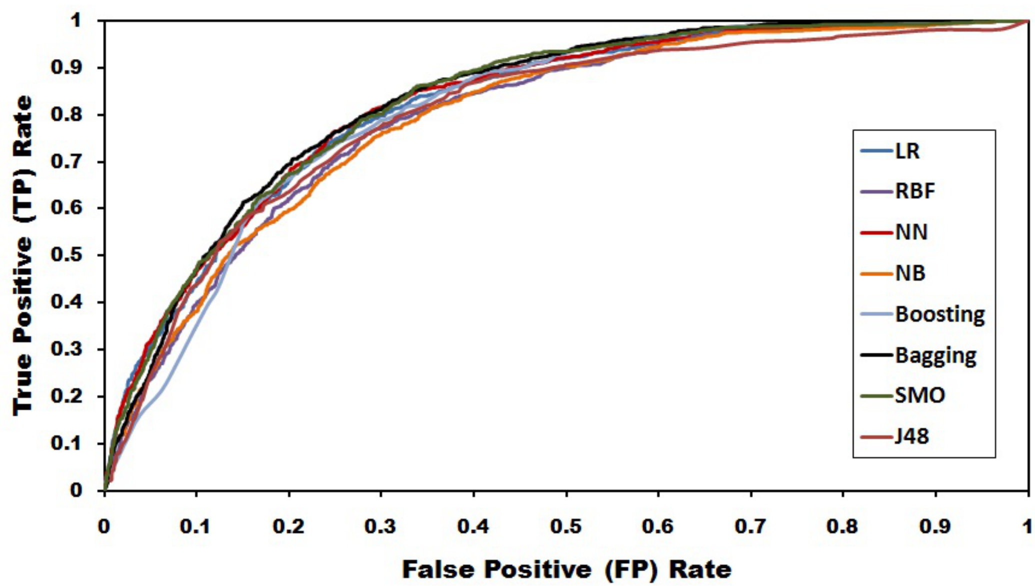


Figure 5.9. Receiver Operating Characteristic Curve (ROC) of different classifiers.

Table 5.10. Evaluation measures of different fault prediction models, trained with the source files (cpp) of Firefox Release 0.8

Classifier	Accuracy		MAE		RMSE		RAE(%)		RRSE (%)		KS	
	No PCA	With PCA	No PCA	With PCA	No PCA	With PCA	No PCA	With PCA	No PCA	With PCA	No PCA	With PCA
LR	77.7	77.5	0.2927	0.2991	0.3846	0.3881	74.40	76.02	86.73	87.50	0.344	0.33
DT	76.9	75.64	0.2806	0.3257	0.4266	0.4076	71.32	82.79	96.20	91.91	0.404	0.345
SMO	76.3	75.4	0.2369	0.2458	0.4867	0.4958	60.21	62.49	109.75	111.80	0.213	0.167
RBF	75.8	76.3	0.3042	0.3041	0.3937	0.3921	77.31	77.30	88.78	88.41	0.295	0.313
NN	78.2	76.7	0.2911	0.3199	0.3813	0.3957	73.98	81.30	85.99	89.23	0.377	0.321
NB	76.3	76.1	0.2365	0.2386	0.4771	0.4797	60.10	60.65	107.59	108.16	0.305	0.260
Boosting	77.2	75.3	0.2964	0.3019	0.3877	0.3957	75.35	76.72	87.41	89.23	0.356	0.397
Bagging	78.9	75.7	0.2767	0.2963	0.3758	0.3997	70.33	75.32	84.74	90.12	0.423	0.346

found that accuracy is always biased toward that class, which has large numbers of instances. Since, our data is skewed therefore, in our experiment, we gave least weightage to accuracy. Furthermore, we computed the value of *Kappa Statistics*, the value of KS lies in the range 0.167(SMO) to 0.346(Bagging). KS value $\geq$ 0.2 is considered as good for any classifier. Therefore, on the basis of kappa statistics the top three classifiers are boosting(KS=0.397), bagging(KS=0.346) and DT(0.345). Furthermore, the data of Table 5.10 shows that MAE value lies in the range 0.2386(NB) to 9.3257(J48). RMSE value lies in the range 0.3881(LR) to 0.4958(SMO). RAE value lies in the range 62.49% to 82.79(J48), and RRSE value lies in the range 87.50(LR) to 111.8(SMO). From the obtained results of different absolute and relative metrics, it can be concluded that in our experiment SMO has not performed well because its RRSE value is highest, i.e., 111.8%. Whereas, NN, LR, bagging, boosting, DT and NB have almost similar values for absolute and relative error metrics. Furthermore, NB has the smallest value of RAE, i.e., 60.65%, but have high value of RRSE, i.e., 108.16%. In case of prediction models, relative error is considered as more important as compared to the absolute error. Therefor, we can conclude that on the basis of accuracy, kappa statistics and error measures, the top five classifiers are LR, DT, NN, Boosting and Bagging.

Until now we have described the classification of source files using classification schema *isBuggy*. Now, in the following section it will be described that how classification schema *bugLevel* can be used for the classification of source files.

Classification of Source Files Using Classification Schema bugLevel

The results of the classification are shown in the Table 5.11. These results are obtained when classifiers trained and tested using classification schema *bugLevel* and principal components. It is shown in the Table 5.11 that for the class *bugLevel*=Medium, the evaluation measures have the following values: precision values range from 0.0%(SMO) to 27.8%(Bagging), recall values range from 0.0%(SMO) to 35.2%(NB), Fmeasure values range from 0.0%(SMO) to 24.4%(NB), and ROC values range from 0.566(SMO) to 0.879(NN). Whereas, the class *bugLevel*=High has the following values: precision values range from 0%(SMO) to 44.4%(Bagging), recall values range from 0%(SMO) to 23.5%(NB), Fmeasure values range from 0%(SMO) to 24.3%(NB), and ROC area range from 0.497(SMO) to 0.734(bagging). The evaluation measures i.e., precision, recall and Fmeasure for the class *isBuggy*=low are ranged from 94% to 100%. In our experiment, our interest is to find those

Table 5.11. Classification of cpp files of Firefox Release 0.8 using *bugLevel* criteria.

Classifier	Class	Pre.	Recall	F-m.	ROC	Acc.	MAE	RMSE	RAE %	RRSE %	KS
LR	Low	94.3	99.2	96.7	0.815						
	Medium	19	5	8	0.866						
	High	5	11.2	18.3	0.666						
	Wt. Avg	90.1	93.2	91.1	0.813	93.2	0.0714	0.1934	85.50	94.8295	0.154
DT	Low	94.7	98.5	96.5	0.606						
	Medium	27.6	13.2	17.9	0.564						
	High	33.3	11.2	16.8	0.497						
	Wt. Avg	90.4	92.8	91.3	0.602	92.8	0.0716	0.2102	85.74	103.06	0.211
SMO	Low	93.4	1	96.6	0.5						
	Medium	0	0	0	0.566						
	High	0	0	0	0.497						
	Wt. Avg	87.3	93.4	90.3	0.5	93.4	0.2425	0.3072	290.47	150.62	0.0
RBF	Low	94.2	99	96.5	0.766						
	Medium	26.3	6.3	10.2	0.811						
	High	37.5	12.2	18.5	0.634						
	Wt. Avg	90	93	91.1	0.765	93.1	0.0711	0.1991	85.11	97.64	0.15
NN	Low	94	99.6	96.7	0.826						
	Medium	11.1	1.3	2.3	0.879						
	High	40	8.2	13.6	0.712						
	Wt. Avg	89.3	93.4	90.8	0.825	95.36	0.0787	0.1908	94.31	93.54	0.107
NB	Low	95.5	92.4	94.1	0.804						
	Medium	18.7	35.2	24.4	0.803						
	High	24.4	23.5	24.3	0.661						
	Wt. Avg	91	88.4	89.6	0.801	88.37	0.0778	0.2699	93.16	132.36	0.2515
Boosting	Low	94.1	99.2	96.6	701						
	Medium	25.7	5.7	9.3	717						
	High	40.9	9.2	15	0.663						
	Wt. Avg	90	93.2	91	0.701	93.15	0.0457	0.2113	54.71	103.61	0.1307
Bagging	Low	94	99.5	96.7	0.815						
	Medium	27.8	3.1	5.6	0.858						
	High	44.4	12.2	19.2	0.734						
	Wt. Avg	90.1	93.4	91	0.815	93.4	0.0736	0.1916	88.16	93.95	0.123

classifiers which can classify faulty source files with high precision and recall. Therefore, in this experiment, classifiers are compared on the basis of their ability to classify the class *bugLevel*=Medium and *bugLevel*=High. Now, from the obtained results of Table 5.11, the list of classifiers according to the high precision and high recall values are given below,

Classifier's Name	Precision (*bugLevel*=Medium)	Precision (*bugLevel*=High)
Bagging	27.8%	44.4%
Decision Tree (J48)	27.6%	33.3%
Radial Basis Function (RBF)	26.3%	37.5%
Naive Bayes (NB)	18.7%	24.4%

Classifier's Name	Recall (*bugLevel*=Medium)	Recall (*bugLevel*=High)
Naive Bayes (NB)	35.2%	23.5%
Decision Tree (J48)	13.2%	11.2%
Radial Basis Function (RBF)	6.2%	12.2%
Bagging	3.1%	12.2%

From the above distribution, it can be concluded that in case of *bugLevel* class distribution, Naive Bayes (NB), Decision Tree (DT) and Bagging have high precision and recall values as compared to other classifiers. However, the attained value of classification evaluations are not very high. One reason to have small values of precision and recall is that the data set related to the *bugLevel* is very skewed. Table 5.11 also depicts that the classifiers have small absolute errors. Whereas, the value of RAE range from 54.7%(Boosting) to 290.47%(SMO) and RRSE range from 94.82%(LR)to 150.62%(SMO). The value of Kappa statistics (KS) range from 0.0(SMO) to 0.2515(NB). KS measures the agreement between two observers. KS values near 1 indicate good agreement and negative values indicate that the two observers agreed less than would be expected just by chance. Finally, the value of ROC area range from 0.497(SMO) to 0.866(LR). ROC value close to 1 is acceptable, whereas, ROC area close to 0.5 is not acceptable.

Figure 5.10, depicts the precision and recall values of each class, when the class distribution is based on *bugLevel* (i.e., Low, Medium and High). Figure 5.10 also depicts the weighted average of precision and recall. The red markers near left corner of the figure represent the precision and recall values of class *bugLevel*=HIGH. The yellow markers at the bottom left corner represent the class *bugLevel*=MEDIUM. Whereas, the markers that lies at the top right corner are the precision and recall values of the class *bugLevel*=HIGH.

From Figure 5.10, one can conclude that the *bugLevel*=High and *bugLevel*=Medium have small precision and recall as compared to class *bugLevel*=Low. This difference is because our class distribution is skewed. The ideal condition for any classifier is, classes (i.e., *bugLevel*=High and *bugLevel*=Medium) should have their precision and recall values at the top right corner.

Comparison of Classification Models

In any release of a software, the number of faulty source files always in small numbers as compared to the clean source files. Therefore, the distribution of clean and faulty source files is highly skewed in any release of a software. Whereas, to construct a fault prediction model to classify a faulty and non faulty source files, classifier has to train with the source files data, which are related to a specific release of a software. Since, the aggregated classification evaluation measures (like accuracy, average precision and average recall) is highly influenced by skewed class distribution. Therefore, to avoid the biased impact of skewed class distribution from the evaluation measures, one has to evaluate each classifier on the basis of their individual class level performance. Therefore, in this experiment, classifiers are evaluated on the basis of individual class level performance. Furthermore, to see the performance of a classifiers at different class distribution, two different experiments have been performed using two different classification schema, i.e., *isBuggy* and *bugLevel*. In each experiment the class distribution is skewed. The class distribution is more skewed when *bugLevel* scheme has used.

Multiple classifiers have been used to train and test with each type of classification schema. The classifiers that performed well when trained and tested with *isBuggy* class distribution are, bagging, boosting and decision tree (J48). Whereas, in case of classification schema *bugLevel* only naive bayes performed well, while the rest of the classifiers attained very small precision and recall values. These results guide us to conclude that in case of binary classification the best classifiers are bagging, boosting and J48. Whereases, if the number classes are more than two and the distribution are highly

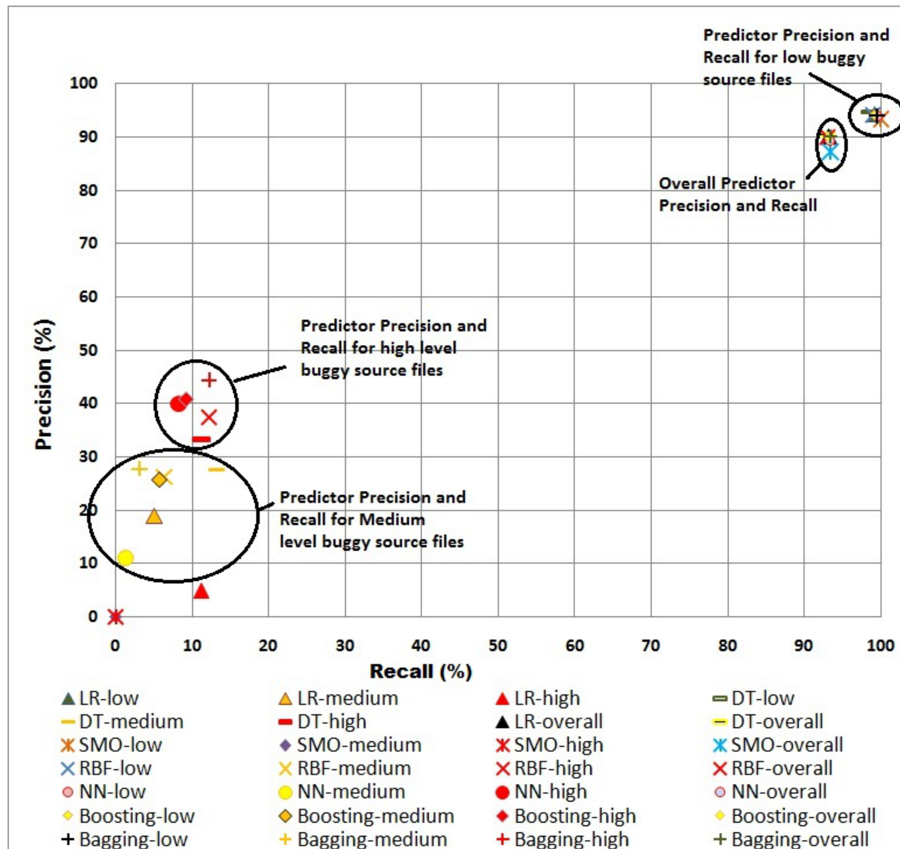


Figure 5.10. Precision-Recall curve of different models. When models are trained using *bugLevel* criteria.

skewed then, naive bayes perform well.

5.4 Case Study: Analyzing Bug Prediction Capabilities of Static Code Metrics

Open Source Software provides a rich resource of empirical research in software engineering. Static code metrics are a good indicator of software quality and maintainability. In this work, we have tried to answer the question whether bug predictors obtained from one project can be applied to a different project with reasonable accuracy. Two open source projects Firefox and Apache HTTP Server (AHS) are used for this study. Static code metrics are calculated for both projects using in-house software and the bug information is obtained from bug databases of these projects. The source code files are classified as clean or buggy using a Decision Tree classifier. The classifier is trained on metrics and bug data of Firefox and tested on Apache HTTP Server and vice versa.

5.4.1 Study Approach

In this section, we describe the steps followed to obtain the results. We first describe the data extraction process and the related tasks. Then, we describe the static code metrics used in this study and the calculation of metrics. Finally, we discuss how to obtain the bug prediction model and related concepts.

Data Extraction

We checked out source code of Apache HTTP Server (AHS) version 1.3.x and 2.0.x using Subversion (SVN) client. Subversion is a software versioning system used to maintain versions of files including source code and documentation. A collection of bug reports related to AHS was obtained from Bugzilla. To find the files in which these bugs were fixed, we also obtained log information from SVN repositories. The SVN log records revision number, author, date and time, lines modified and the comment by the developer. If a problem was fixed, the developer also mentions the Problem Report number. We filtered the comments for the occurrence of bugs using the regular expression, e.g. if a comment contained the keyword fix or Fix or PR: or Fixed, we marked that revision as buggy. To assign numbers of bugs to individual files, we selected two dates for each version and counted the numbers of bugs that were fixed between these two dates.

- For AHS 1.3.x 15-08-2002 to 15-08-2005
- For AHS 2.0.x 01-12-2003 to 31-12-2005

We downloaded source code of Firefox Releases 0.8, 1.0, 1.5 and 2.0 using the ftp server. Bug information related to Firefox was obtained from Bugzilla[4]. To find the files in which these bugs were fixed, we also obtained log information from CVS repositories. We processed the comments in CVS

[4]http://www.bugzilla.org/

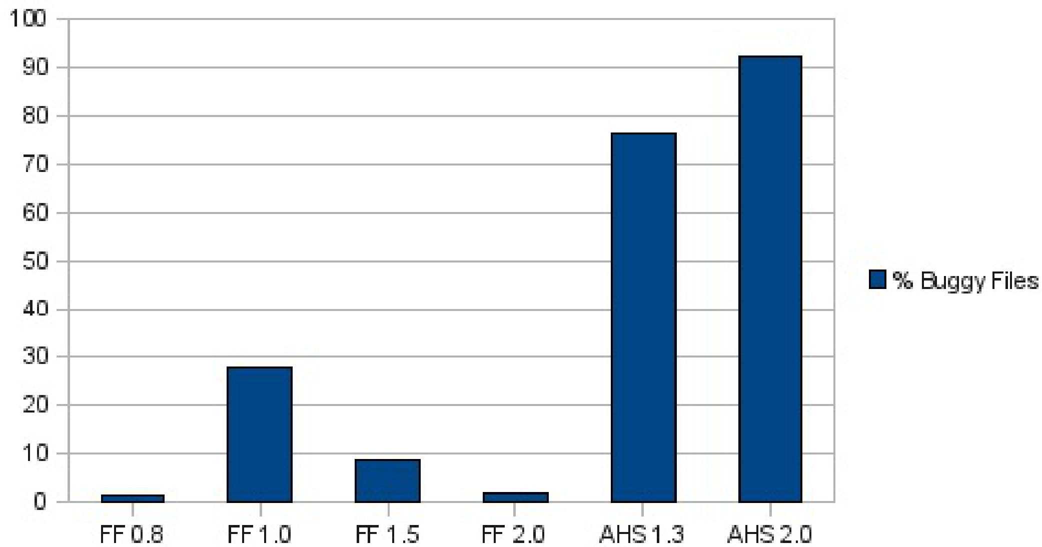


Figure 5.11. Percentage of Buggy Files

log output to find the revisions of files in which bugs were fixed. Each file was assigned a bug count indicating the number of bugs that occurred between two consecutive releases of Firefox. As AHS source code mainly consists of C files and Firefox source code contains C, C++ and JAVA files as well, we selected C files from both projects. We processed the following number of C files in different releases of both projects:

Project	Release	No. of C Files
AHS	1.3.x	155
AHS	2.0.x	191
Firefox	0.8	1389
Firefox	1.0	1387
Firefox	1.5	1522
Firefox	2.0	1527

AHS releases contain more buggy files than Firefox releases. AHS 2.0.x has a highest %age of buggy files with 92% buggy files followed by AHS 1.3.x having 76% buggy files. Firefox 0.8 and 2.0 have the lowest %age of buggy files with less than 2% buggy files. Figure 5.11 compares the buggy files in all releases of both projects.

Metrics Calculation

We used our own software to calculate the static code metrics. As C is a procedural language, we calculated individual metrics for functions and aggregated these metrics on files getting total and maximum values of each metrics for each file. We used the following function metrics for our study:

Table 5.12. Function Metrics of AHS files

Metrics	AHS 1.3.x			AHS 2.0.x		
	Max	Min	Avg	Max	Min	Avg
NEL	642	2	29	744	2	31
CD	43	0	12	60	0	11
CC	143	1	6	154	1	6
PC	21	0	2	21	0	2
RP	92	0	2	56	0	2
LVC	74	0	2	79	0	3
ND	86	1	2	52	1	3

- *NEL(Number of Executable Lines).* The NEL is number of lines of code in a function excluding the blank and comment lines.
- *CD(Control Density).* The CD is the ratio of control lines and executable lines in a function.
- *CC(Cyclomatic Complexity).* The CC is the number of linearly independent paths through a function.
- *PC(Parameter Count).* The PC is the number of arguments a function receives.
- *RP(Return Points).* The RP is the number of return statements in a function.
- *LVC(Local Variable Count).* The LVC is the number of variables locally defined in a function.
- *ND(Nesting Depth).* The ND is the maximum depth of the nested scope in a function.

We also calculated following file metrics in addition to the aggregated metrics:

- *LOC(Lines Of Code).* The LOC is the total number of lines in a file.
- *NOF(Number Of Functions).* The NOF is the total number of functions in a file.
- *NOIF(Number Of Included Files).* The NOIF is the number of files included in a file using the include statement.
- *NOGC(Number Of Global Conditions).* The NOGC is the number of conditional statements globally defined in a file.

Table 5.12 and 5.13 show the maximum, minimum and average values for the function metrics discussed above. Table 5.14 shows the maximum, minimum and average values for the file metrics discussed above.

If we compare both projects, the average and minimum values for the metrics are almost similar. However, there are differences in maximum values. In Firefox 14% of the functions have more than 50 executable lines, whereas in AHS 16% of the functions contain more than 50 executable lines. Functions having cyclomatic complexity greater than 20 are 4% in Firefox and 5% in AHS respectively, whereas functions having parameter counts greater than 5 are 4.5% in Firefox and 3%

Table 5.13. Function Metrics of Firefox files

Metrics	Firefox 0.8			Firefox 1.0			Firefox 1.5			Firefox 2.0		
	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg
NEL	7616	2	29	1931	2	29	12026	2	29	1759	2	30
CD	58	0	11	58	0	11	85	0	11	100	0	11
CC	401	1	5	398	1	5	508	1	5	583	1	6
PC	22	0	2	22	0	2	22	0	2	22	0	2
RP	206	0	2	90	0	2	249	0	2	276	0	2
LVC	191	0	3	191	0	3	191	0	3	282	0	3
ND	302	1	2	302	1	2	302	1	2	302	1	2

Table 5.14. File Metrics

Project	LOC			NOF			NOIF			NOGC		
	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max	Min	Avg
AHS 1.3.x	4981	7	589	138	1	13	38	0	6	87	0	3
AHS 2.0.x	4801	29	734	111	1	15	34	0	9	43	0	3
Firefox 0.8	9301	13	596	284	1	13	39	0	5	372	0	3
Firefox 1.0	9353	13	592	291	1	13	39	0	5	372	0	3
Firefox 1.5	12677	13	639	312	1	14	40	0	5	372	0	3
Firefox 2.0	9338	13	641	220	1	14	39	0	5	372	0	3

in AHS respectively. Firefox and AHS contain 4.2% and 8.5% of functions having more than 5 return points respectively. 1.8% of the functions in Firefox and 4% of the functions in AHS have more than 10 nesting depth. Figure 5.12 shows the average values of function metrics for both projects.

Average file sizes are comparable in both projects with slight differences. AHS 2.0 has the highest average file size and AHS 1.3 has the smallest average file size. 15% of files in Firefox 0.8 and Firefox 1.0, 17% of files in Firefox 1.5 and Firefox 2.0, 18% of files in AHS 1.3 and 27% of files in AHS 2.0 have more than 1000 LOC. Figure 5.13 compares the average file sizes in both projects.

Average number of functions/file is similar in both projects. However, 18% of files in Firefox 0.8 and Firefox 1.0, 19% of files in Firefox 1.5 and Firefox 2.0, 16% of files in AHS 1.3 and 48% of files in AHS 2.0 contain more than 20 functions. In Firefox 5% files contain more than 10 globally defined conditions, whereas in AHS 6% files contain more than 10 globally defined conditions. Firefox contains 14 % files having more than 10 files included using the #include statements, whereas in AHS 1.3 and AHS 2.0 this amount is 12% and 34% respectively. This indicates AHS 2.0 has highly correlated files. Figure 5.14 compares the file metrics of both projects.

Obtaining the Bug Predictor

We stored the information obtained from previous two steps into relations. Each relation consisted of files as entities. The relations consisted of file attributes including the static code metrics and the

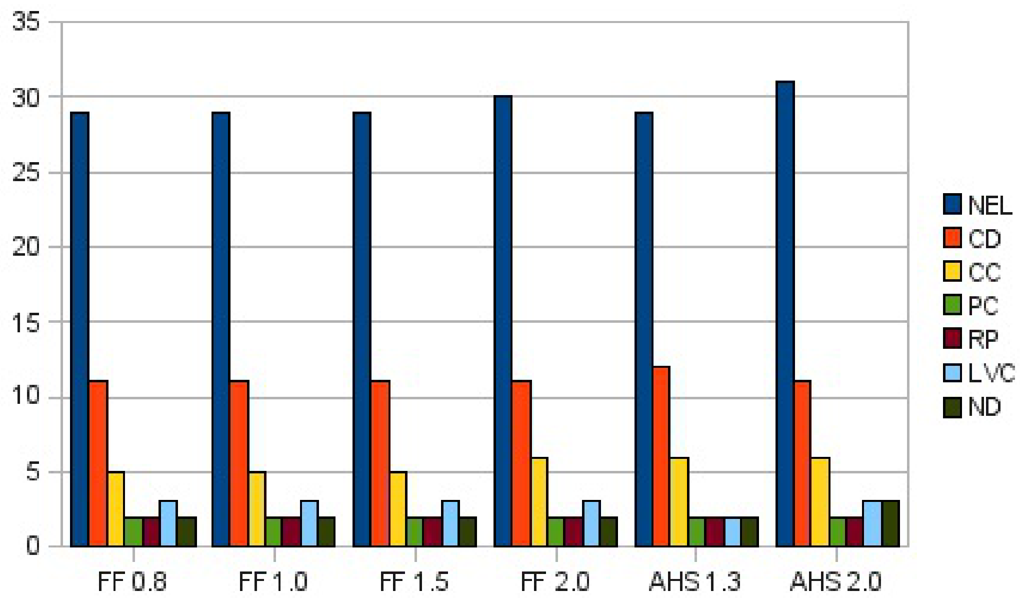


Figure 5.12. Average Function Metrics

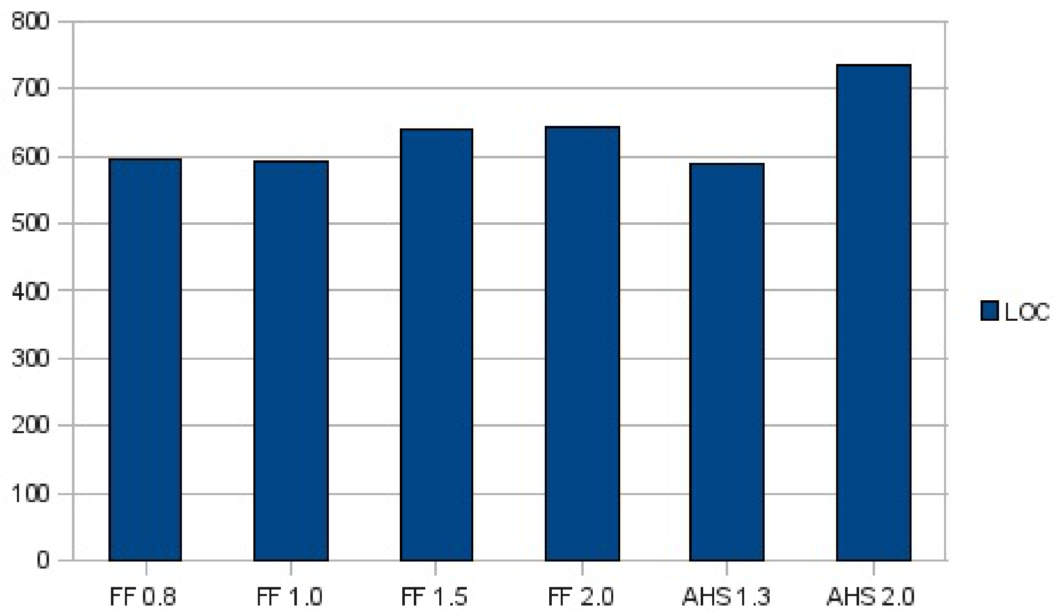


Figure 5.13. Average File Sizes

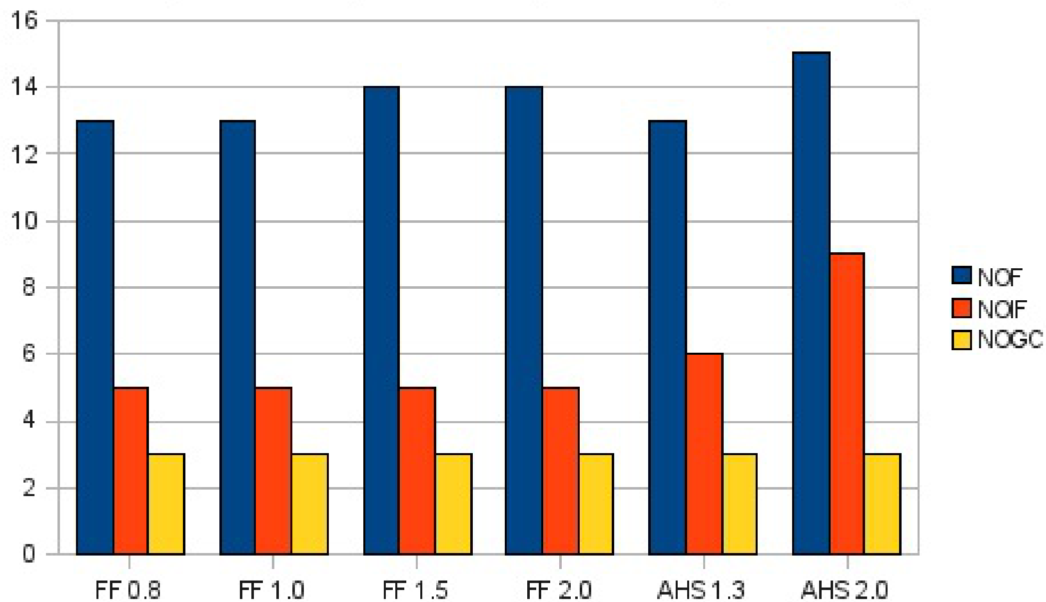


Figure 5.14. Average File Metrics

bugs related to each file. We trained a decision tree classifier on this data using WEKA[5] (a Machine Learning tool developed in JAVA) to classify files as clean or buggy. We selected decision tree for our study due to its strong classification capabilities. We trained the classifier on four releases of Firefox and tested each model obtained on two versions of AHS. Alternatively, we also trained the classifier on two releases of AHS and tested each model obtained on four releases of Firefox. After training of classifier WEKA stored models in the form of Java serializable objects holding different options, which later on can be applied to the test data.

A major proportion of time required to obtain a bug predictor is involved in data preparation. Metrics calculation and extraction of bug information requires much time which depends on the project size, and it may take hours even using a high speed processor. Once the data is prepared, weka classifiers take few seconds to learn and produce outputs. Weka classifiers hold multiple options to be used during training and testing. Following is a brief summary of these options.

- *-t* <*name of training file* >. Sets training file.
- *-T* <*name of test file* >. Sets test file. If missing, a cross-validation will be performed on the training data.
- *-c* <*class index* >. Sets index of class attribute (default: last).
- *-x* <*number of folds* >. Sets number of folds for cross-validation (default: 10).
- *-s* <*random number seed* >. Sets random number seed for cross-validation (default: 1).
- *-l* <*name of input file* >. Sets model input file.

[5] http://www.cs.waikato.ac.nz/ml/weka/

```
0 yes 0.982973149967256 yes (8,182,46,146,35,43,13,25,9,18,5,13,4,14,5,11,1,293)
1 yes 0.982973149967256 yes (8,205,78,79,23,38,16,18,4,22,8,26,9,24,11,9,0,304)
2 yes 0.982973149967256 yes (6,122,55,52,20,17,9,13,5,13,5,6,3,8,3,8,0,227)
3 yes 0.982973149967256 yes (9,185,52,79,19,32,11,23,4,22,7,21,8,13,5,10,0,282)
4 yes 0.982973149967256 yes (43,1455,212,544,28,230,32,111,5,108,18,124,12,97,14,19,2,2096)
5 yes 0.982973149967256 yes (3,35,27,0,0,0,0,9,7,2,1,1,1,3,1,2,0,71)
6 yes 0.982973149967256 yes (4,107,61,29,18,17,9,12,5,4,2,10,7,7,3,15,1,181)
7 yes 0.982973149967256 yes (35,871,177,307,35,157,61,82,7,62,7,55,14,55,11,10,1,1139)
8 yes 0.982973149967256 yes (17,1429,488,176,23,192,79,56,5,87,33,88,23,48,8,17,0,1752)
9 yes 0.982973149967256 yes (11,440,138,105,23,73,26,32,5,15,4,34,9,46,22,15,1,540)
10 yes 0.982973149967256 yes (11,241,73,115,23,36,11,27,5,24,6,19,6,13,2,13,4,405)
11 yes 0.982973149967256 yes (28,488,79,216,25,55,7,70,5,36,5,43,9,38,3,6,0,735)
12 yes 0.982973149967256 yes (31,1045,124,282,24,132,17,86,5,79,8,110,14,69,13,9,0,1495)
13 yes 0.982973149967256 yes (6,46,13,7,7,2,2,15,5,5,2,6,4,6,1,5,0,97)
14 yes 0.982973149967256 yes (39,1556,227,435,23,197,25,118,7,115,9,128,16,79,11,11,5,2130)
15 yes 0.982973149967256 yes (8,92,24,72,25,16,6,7,2,9,3,9,3,9,2,6,0,140)
16 yes 0.982973149967256 yes (71,3833,356,877,25,565,51,135,5,343,28,294,20,164,13,12,0,4801)
17 yes 0.982973149967256 yes (18,743,155,254,25,106,22,34,6,26,4,63,20,63,18,6,0,1116)
18 yes 0.982973149967256 yes (2,9,5,0,0,0,0,1,1,1,1,0,0,2,1,4,0,33)
19 yes 0.982973149967256 yes (6,111,75,12,12,16,16,12,5,6,3,6,5,6,1,4,0,194)
20 yes 0.982973149967256 yes (34,1611,489,367,25,207,40,49,7,96,18,95,17,94,15,9,1,2020)
21 yes 0.982973149967256 yes (12,585,100,166,21,76,12,22,5,44,6,46,10,37,7,8,1,791)
22 yes 0.982973149967256 yes (5,47,26,12,12,4,4,12,5,4,2,3,3,6,2,7,0,92)
23 yes 0.982973149967256 yes (7,100,28,45,16,10,3,4,1,8,3,8,2,7,1,7,2,171)
24 yes 0.982973149967256 yes (10,188,83,133,33,26,8,12,4,15,3,11,5,14,3,5,2,290)
25 yes 0.982973149967256 yes (16,195,21,169,37,34,7,33,4,20,5,14,3,17,2,5,3,290)
```

Figure 5.15. Predictions for individual files

- *-d* <*name of output file* >. Sets model output file.
- *-i*. Outputs detailed information-retrieval statistics for each class.
- *-k*. Outputs information-theoretic statistics.
- *-p* <*attribute range* >. Only outputs predictions for test instances, along with attributes (0 for none).

During training -t and -d options can be used to train on an input file and store the model obtained into a model file which later can be used. For predictions or testing -T and -l options can be used to mention test and model files. For getting output statistics -i and -k options are used. In order to get output predictions for individual instances -p option is used. It displays the actual and predicted values for each instance. Figure 5.15 displays a screen shot of the predictions, the values in each line are separated by a single space. The fields are the zero-based test instance id, followed by the predicted class value, the confidence for the prediction (estimated probability of predicted class), the true class and the values of selected attributes.

5.4.2 Results and Discussion

In this section, we discuss the results obtained by applying the Decision Tree (J48) classifier. We obtained the number of correctly classified instances (CCI), the Kappa statistics (KS), the mean absolute error (MAE), and the root mean squared errors (RMSE). KS is a means of classifying agreement in categorical data.

Table 5.15. Classification of Firefox files using the predictor obtained from AHS

Predictor	Release	CCI	KS	MAE	RMSE
AHS 1.3.x	Firefox 0.8	92%	0.0586	0.18	0.28
	Firefox 1.0	68%	-0.036	0.36	0.51
	Firefox 1.5	84%	-0.003	0.24	0.37
	Firefox 2.0	91%	0.097	0.19	0.30
AHS 2.0.x	Firefox 0.8	92%	0.01	0.12	0.29
	Firefox 1.0	70%	0.04	0.31	0.52
	Firefox 1.5	86%	0.02	0.16	0.36
	Firefox 2.0	92%	-0.02	0.12	0.29

Table 5.16. Classification of AHS files using the predictor obtained from Firefox

Predictor	Release	CCI	KS	MAE	RMSE
Firefox 0.8	AHS 1.3.x	78%	0.0	0.22	0.45
	AHS 2.0.x	92%	0.0	0.09	0.28
Firefox 1.0	AHS 1.3.x	68%	0.0	0.34	0.52
	AHS 2.0.x	78%	0.09	0.29	0.45
Firefox 1.5	AHS 1.3.x	77%	0.04	0.23	0.46
	AHS 2.0.x	92%	0.09	0.12	0.29
Firefox 2.0	AHS 1.3.x	79%	0.0	0.22	0.45
	AHS 2.0.x	92%	0.09	0.09	0.27

Using the predictor obtained from AHS 1.3.x, we correctly classified 92% of the files in Firefox 0.8 and Firefox 2.0. However, for Firefox 1.0 the predictor could correctly classify 68% of the files. The results for Firefox 1.5 were inbetween with a value of 84% correctly classified files. The results were almost similar using the predictor obtained from AHS 2.0.x. Table 5.15 shows that MAE values are below 0.4, indicating fair accuracy of results. In most cases RMSE values are near 0.4, which further validates the results.

Using the predictors obtained from Firefox 0.8 , Firefox 1.5 and Firefox 2.0, we correctly classified 92% of the files in AHS 2.0.x. However for AHS 1.3.x the predictors could classify 78% of the files. The predictor obtained from Firefox 1.0 showed poor results with 78% and 68% of the files of AHS 2.0.x and AHS 1.3.x correctly classified respectively. Table 5.16 shows that MAE values in most cases are below 0.3 indicating good accuracy of results. The RMSE values are slightly higher in some cases, which indicate poor accuracy.

We have also calculated the True Positive rate, False Positive rate, Precision, Recall and F-Measure for the predictions obtained by applying each model. Tables 5.17-5.22 show the values of these measures for each predictor. Class "yes"indicates the buggy files whereas class "no"indicates the clean files. For Firefox predictions, precision and recall are high for clean file predictions but its poor for buggy file predictions. However for AHS predictions, precision and recall are high for buggy file predictions and its poor for clean file predictions. The reason may be the high percentage of buggy

Table 5.17. Using AHS 1.3.x as predictor

Release	TP Rate	FP Rate	Precision	Recall	F-Measure	Class
Firefox 0.8	0.85	0.95	0.98	0.85	0.95	no
	0.05	0.15	0.01	0.05	0.01	yes
Firefox 1.0	0.83	0.91	0.70	0.83	0.76	no
	0.09	0.17	0.17	0.09	0.12	yes
Firefox 1.5	0.88	0.64	0.94	0.88	0.91	no
	0.36	0.12	0.22	0.36	0.27	yes
Firefox 2.0	0.86	0.96	0.98	0.86	0.92	no
	0.04	0.14	0.01	0.04	0.01	yes

Table 5.18. Using AHS 2.0.x as predictor

Release	TP Rate	FP Rate	Precision	Recall	F-Measure	Class
Firefox 0.8	0.93	0.90	0.98	0.93	0.96	no
	0.1	0.07	0.02	0.1	0.03	yes
Firefox 1.0	0.94	0.91	0.73	0.94	0.82	no
	0.1	0.06	0.37	0.09	0.15	yes
Firefox 1.5	0.94	0.91	0.92	0.94	0.93	no
	0.08	0.06	0.11	0.08	0.09	yes
Firefox 2.0	0.93	1.0	0.98	0.93	0.96	no
	0	0.01	0	0	0	yes

files in AHS and high percentage of clean files in Firefox. In AHS more than 70% files are buggy whereas in Firefox less than 10% files are buggy except Firefox 1.0 in which 28% files are buggy, as depicted in Figure 5.11.

5.5 Summary

In this chapter, we presented different fault prediction models. These models are obtained using regression and classification techniques. To build the model, we used multiple code metrics as predictors, and to avoid the collinearity among metrics, we used the technique of principal component analysis. The results of the obtained models have shown that software complexity metrics can be

Table 5.19. Using Firefox 0.8 as predictor

Release	TP Rate	FP Rate	Precision	Recall	F-Measure	Class
AHS 1.3.x	1.0	1.0	0.76	1.0	0.86	yes
	0	0	0	0	0	no
AHS 2.0.x	1	1	0.92	1	0.96	yes
	0	0	0	0	0	no

Table 5.20. Using Firefox 1.0 as predictor

Release	TP Rate	FP Rate	Precision	Recall	F-Measure	Class
AHS 1.3.x	0.82	0.81	0.76	0.82	0.79	yes
	0.19	0.18	0.25	0.19	0.21	no
AHS 2.0.x	0.82	0.67	0.93	0.82	0.87	yes
	0.33	0.18	0.13	0.33	0.19	no

Table 5.21. Using Firefox 1.5 as predictor

Release	TP Rate	FP Rate	Precision	Recall	F-Measure	Class
AHS 1.3.x	0.97	1	0.76	0.97	0.85	yes
	0	0.02	0	0	0	no
AHS 2.0.x	0.99	0.93	0.93	0.99	0.96	yes
	0.07	0.01	0.33	0.07	0.11	no

efficiently used for prediction of faults.

This Chapter is divided into three parts. In the first part, regression based fault prediction model are described in details. Both, parametric and non-parametric techniques are described to build the models. It is found that multiple linear regression and neural network can be used to build a high performance regression based fault prediction model.

In the second part of this chapter, machine learning classification techniques described in details to construct different fault prediction models. It is discussed that skewed class distribution generate biased results. Furthermore, it is described in details that in case of skewed class distribution, the best way to evaluate a classifier is to measure precision and recall of each class. Two different classification schema have used to construct the fault prediction models. Multiple classifiers have been used to train and test with each type of classification schema. The classifiers that performed well, when trained and tested with *isBuggy* class distribution: bagging, boosting and decision tree (J48). In case of classification schema *bugLevel*, naive bayes performed well.

In the third part of this chapter, results of a case study are discussed. These results have shown that predictions vary with different releases, however bug predictor obtained from one project can be applied to a different project with a reasonable accuracy. The results obtained vary with different releases of these projects, and accuracy can be as good as 92 % of the files correctly classified and as

Table 5.22. Using Firefox 2.0 as predictor

Release	TP Rate	FP Rate	Precision	Recall	F-Measure	Class
AHS 1.3.x	1	1	0.76	1	0.86	yes
	0	0	0	0	0	no
AHS 2.0.x	1	1	0.92	1	0.96	yes
	0	0	0	0	0	no

poor as 68 % of the files correctly classified by the trained classifier. The performance of classifier on the basis of individual class is poor, which shows that although we have high accuracy but the predicted accuracy is biased to that class, which have large numbers of instance. Factors other than static code metrics can be considered for more accurate predictions.

6 Fault Prediction Capability of Program Files Logical Coupling Metrics

Parts of the contents of this chapter have been published in paper [Ahsan and Wotawa, 2010a].

In the recent years several research studies reveal that the presence of logical couplings makes the structure of the software system unstable, and any new changes in the coupled source files become more error prone. Hence, logically-coupled source files are strong candidates for restructuring. Software metrics and visualization techniques are used to identify source files, which are logically coupled with other source files.

Research Trends: Recently, many research works have been done on logical coupling, which extracted coupling information of source files from software's repository and used them to obtain a set of coupling metrics and to visualize the structural complexity of the software systems [Breu, 2006, D'Ambros et al., 2009b, Lanza and Ducasse, 2003]. Most of the research work has been performed, which discussed the significance of logical coupling [Gall et al., 2003, D'Ambros et al., 2009b, Eaddy et al., 2008a]. The recent research trends in software engineering, reveals that logical coupling measures are highly correlated with the number of defects [D'Ambros et al., 2009b, Eaddy et al., 2008a]. M. Eaddy et al. and D'Ambros et al. have addressed this issue in separate attempts, M. Eaddy et al. correlated cross-cutting concerns with defects, and D'Ambros et al. link logical coupling measures with the number of bugs. D'Ambros results, reveals that logical coupling metrics highly correlated with the number of bugs as compared to others classical set of metrics, i.e., LOC, Chindamber and Kemerer metrics suite etc.

Motivations: Software corrective maintenance, restructuring and reengineering are costly endeavor, because it is nontrivial and time consuming to identify the causes which induce defects into software systems. Therefore, before releasing a new version of a software product huge effort (e.g., code inspections, program analysis, pre-release testing) goes to ensure that there are no bugs remain in the release products of a software. These efforts might be better and well directed if we had a better understanding of what causes, make the system unstable and generate defects [Breu, 2006].

One major cause of frequently generating faults in software is modifying a program source file without prior analysis of its previous change-coupling patterns. The development scenario became

worst, when developers made changes in program source files, which is logically coupled with other source files, and programmer unintentionally forget to made changes in all the related coupled source files. Consequently, system became unstable. The possible remedy to reduce the frequent occurrence of such scenarios, is to equip software developers with tools and techniques that provide information related to the logical coupling nature of source files. Formally, logical coupling is defined as, the implicit dependency relationship between two or more software artifacts that have been observed to be frequently changed together during the evolution of the system [Bieman et al., 2003a]. Logical coupling information helps to understand how the system has evolved and provides support in system restructuring.

Our Goals and Approach: The goal of our research work is to find a set of logical coupling metrics using bug fix patterns of coupled source files. Furthermore, find the metrics correlation with the number of bugs. Therefore, in this chapter we propose an approach to obtain a set of logical coupling patterns, which are related to the fixing of bugs. We obtain these patterns by mining bug-fix transaction's data of source files, and use them to define a set of metrics.

Research Hypothesis: Our research hypothesis are,

1. The history of coupled source file's bug-fix patterns can be used to obtain a set of coupling metrics, which are more correlated with the number of bugs. Consequently, metrics can be used to identify those source files which are candidates for restructuring.
2. Logical coupling metrics can be used to construct a regression based fault prediction models. Furthermore, logical coupling metrics can be used for the classification of source files as low, medium or high level of complexity.

Our Contributions: The main contributions of our research work are, we defined eight different metrics to measure the logical coupling tendency of a source file. We also performed an experiment by extracting the bug-fix transaction data from the software repository of GNOME OSS project, and obtained a set logical coupling metrics. We used statistical technique, i.e., spearman's rank correlation to find the correlation between logical coupling metrics and number of bugs. We found that logical coupling metrics set is highly correlated with numbers of bugs. The maximum obtained correlation value is 97%. We also obtained the correlation between source files level metrics (i.e., LOC, cyclomatic complexity, etc.,) and the number of bugs, and compared its value with the correlation values of logical coupling metrics. We, found that logical coupling metrics out-class the classical source file level metrics. In order to analyze the bug prediction and classification capabilities of the logical coupling metrics, we used statistical regression and classification techniques of statistics and machine learning. In case of statistical regression, we used *Stepwise Linear Regression Model* and *Regression Model based on PCA* (Principal Component Analysis). The obtained R-Square values for the two models are 95.5% and 80% respectively. Similarly, in case of machine learning based classification, we used J48 classifier and obtained 97% accuracy.

Chapter Layout: The remaining part of the chapter is organized as follows: In Section 6.1 we describe our approach to obtain the data from software's repository, and also describe our approach to extract the coupling patterns and define a new set of logical coupling metrics. In the next section

i.e., 6.2, we describe in details the experimental setup and discuss results. In Section 6.3 we discuss some threats to validity. Finally, in section 6.4 we summarize the chapter.

6.1 Our Approach

We obtain a set of metrics for logical-couplings. These metrics are based on the history of coupled source file's bug-fix patterns. To achieve the goal, we first extract the data from software's repository. Then, we process the extracted data of source files to obtain the different sets of logical-coupling patterns, which are related to the fixing of bugs. We use these patterns to define a set of metrics.

6.1.1 Data Extraction and Filtering

The logical coupling's data are obtained from software change transactions. These transactions are accumulated during the evolution of any large software system, and stored into a CVS/Subversion repository. In each software change transaction, developer's made changes in one or more source files and commit these changes into CVS/Subversion. From these set of software change transactions data, one can easily extract the set of source files, which are committed together, and call them *Coupled Source Files*. Developers made software change transactions, whenever a software change request is reported into a bug tracking system like *Bugzilla*. Therefore, the first step is to extract the raw data from the software repository, i.e., CVS/Subversion and bug tracking system, i.e., Bugzilla. We, therefore, downloaded Subversion log files and Bugzilla bug reports from the software repositories of Gnome open source project. The downloaded data are in html, xml or text format. Therefore, we parsed each data file and stored them into a relational database, by using an approach gave by Fischer [Fischer et al., 2003].

After obtaining the data, filtering and mapping are required. In the first step, we established a link between the bug reports and the associated check-ins (commit) in the version control system e.g. Subversion. In each check-in, developers make changes in the source files and save those changes into Subversion. We obtained these links by processing the log data of Subversion, and received the number of check-in transactions that are related to bug fix changes. To identify the bug fix transaction, we parsed the developer's check-in comments. If comments contain any of the following words *bug fix*, *bug fixed*, or *fixed*, or any *bug-id*, and followed by 6-8 integer numbers, then we considered this transaction as a bug fix transaction [Fischer et al., 2003]. We also used two independent levels of confidence, i.e., syntactic and semantic level of confidence. The syntactic level is used to infer the link between change log messages and bug reports, whereas the semantic level is used to validate the link by matching the bug report data, see [Thomas Zimmermann, May 23-28, 2004]. We used this approach and defined each link with syntactic and semantic level of confidence. After identifying all the bug fix transactions and linked them with bug reports, the second step is to find the names of the source files, which have been modified in each transaction. In case of Subversion all those files, which committed together have the same revision number. Therefore, in case of Subversion one can easily obtain the list of source files, which are changed in a single transaction.

6.1.2 Program Files Coupling Patterns and Coupling Measures

In this section, we first describe the method which we used to obtain a set of bug-fix patterns of source files. Then, we formally describe a set of metrics, which we used to measure the logical-coupling tendency of source files.

Logical-Coupling Patterns

In the field of software engineering pattern mining are very common. Patterns are obtained by analyzing the history data and found that how the things have evolved. To obtain the bug-fixing patterns of coupled source files, we use our own heuristic approach, which process the bug-fix transaction data related to each source file, and obtain a set of coupling patterns. These patterns are the set of source files, which have been fixed in one or more transactions. Each unique set of source files is considered as a unique pattern. Whereas, the number of transactions in which a same unique pattern has observed, is considered as the frequency of that pattern. In the following paragraph, we formally describe our method which we used to obtain logical coupling patterns.

Let, F be the set of source files and T be the set of bug-fix transactions. A transaction t_i is a set of transactions $\{t_1,t_2,..,t_k\}$ such that $f_j \in F$ i.e., each f_j is a source file changed in transaction t_i. Let, file f_p is changed in the transactions $t_1,t_2,..,t_q \in T_c$, where T_c be the set of coupling transactions such that $T_c \subseteq T$. In order to find the set of source files, which have been changed together with f_p in one or more transactions, we have to take the intersection of each possible combination of transactions T_c. The possible number of combinations are 2^q-1, which is used to find the patterns. Consider $PT=\{pt_1,pt_2,pt_3,..pt_i..pt_r\}$ be the set of patterns, where $pt_i \subseteq F$ and r is the number of distinct pattern. Now, in order to compute patterns from transactions T of source files F, let $|T_c| = |2^T\text{-}1|$ be the possible combinations of transactions. A pattern from T_c is given as follows:

$$\{ f_i \mid f_i \in F \wedge \forall\, t \in T \rightarrow f_i \in t\}.$$

Figure 6.1 depicts an example, which shows that how the patterns are obtained from bug-fix transaction data of a source file f_1.

Logical Coupling Metrics

Metrics are commonly used in software engineering for basic understanding and obtain higher level views of the software. Furthermore, metrics are used for finding of design violations. In case of software evolution, metrics can be used to identify those parts of the software, which are unstable and candidate for refactoring [Mens et al., 2002]. In our experiment, we defined 8 different metrics, which measures the coupling tendency of source files, and may be used to identify those set of source files, which are candidates for refactoring or restructuring.

Before formally describe the set coupling measures, once again consider the set of source files F such that $f_p \in F$, represent a source file whose logical-coupling patterns are needs to be determined. Whereas, $f_c \subseteq F$, represents the set of logically-coupled source files with f_p. Now, in our experiment an

Combination of Transactions (2^n-1) 2^3-1=7			List of Common Source Files Changed in One or More Transactions	Number of Common Transactions	Obtained Patterns With Frequency
t_1	t_2	t_3	$\{f_1,f_2,f_3,f_4,f_5,f_6\} \cap \{f_1,f_2,f_3\} \cap \{f_1,f_3\} = \{f_1,f_3\}$	3	$\{f_1,f_3\}$=3 $\{f_1,f_2,f_3\}$=2 $\{f_1,f_2,f_3,f_4,f_5,f_6\}$ =1
t_1	t_2	0	$\{f_1,f_2,f_3,f_4,f_5,f_6\} \cap \{f_1,f_2,f_3\} = \{f_1,f_2,f_3\}$	2	
t_1	0	t_3	$\{f_1,f_2,f_3,f_4,f_5,f_6\} \cap \{f_1,f_3\} = \{f_1,f_3\}$	2	
t_1	0	0	$\{f_1,f_2,f_3,f_4,f_5,f_6\}$	1	
0	t_2	t_3	$\{f_1,f_2,f_3\} \cap \{f_1,f_3\} = \{f_1,f_3\}$	2	
0	t_2	0	$\{f_1,f_2,f_3\}$	1	
0	0	t_3	$\{f_1,f_3\}$	1	
Source Files: F=$\{f_1,f_2,f_3,f_4,f_5,f_6\}$; Change Transactions: T=$\{t_1,t_2,t_3\}$; Number of Transaction: n=3 $t_1=\{f_1,f_2,f_3,f_4,f_5,f_6\}$; $t_2=\{f_1,f_2,f_3\}$; $t_3=\{f_1,f_3\}$;					

Figure 6.1. An Example to obtain a set of logical-coupling patterns for source file f_1, which were fixed 3-times in the past, and logically coupled with source files f_2,f_3,f_4,f_5,f_6. The obtained set of logical-coupling patterns of source file f_1 with maximum frequency is shown in the right-most column.

event, i.e., software change transactions t, is defined as,

$$F_c(f_p, f_c) = \{ t \mid t \in T \wedge f_p \in t \wedge f_c \in t \}.$$

Count of Coupled Source Files (COCSF): In case of COCSF metrics, we count the element of f_c, where each element of f_c is a source file which has been fixed with a source file f_p. In simple, COCSF counts the total number of source files coupled with a source file f_p i.e.,

$$\mid f_p \mid = k, \tag{6.1}$$

where k is the cardinality of f_c.

Coupling Patterns Count (CPC): CPC count the distinct number of coupling patterns. In the previous section, we described how to obtain the number of distinct coupling patterns. A pattern is obtained when we take the intersection of two or more transactions. If PT=$\{pt_1,pt_2,pt_3,..pt_j....,pt_r\}$ be the set of obtained patterns for source file f_p, where r is the number of distinct pattern or cardinality of PT. Therefore,

$$\mid CPC(f_p) \mid = r. \tag{6.2}$$

Sum of Coupling Pattern Frequency (SCPF): Each pattern associated to a source file have some frequency of occurrence. Frequency is the count of the same pattern, that occurred in one or more transactions. SCPF simply adds all those frequencies. If $pt_j \in$PT, then SCPF is given as,

$$SCPF(f_p) = \sum_{pt_j \in PT \wedge f_p \in pt_j} Frequency(pt_j) \tag{6.3}$$

Average of Coupling Pattern Frequency (ACPF): ACPF adds all the pattern frequency of source file f_p, and divide it by the total number of patterns r. Where $r > 0$.

$$\text{ACPF}(\text{f}_p) = \sum_{pt_j \in PT \wedge f_p \in pt_j} \text{Frequency}(\text{pt}_j) \ / \ \text{r} \tag{6.4}$$

Maximum of Coupling Pattern Frequency (MCPF): It represents the frequency value of a pattern, which is the most frequently occurred pattern of a source file f_p,

$$\text{MCPF}(\text{f}_p) = \text{Maximum}(\text{Frequency}(\text{pt}_j)) \tag{6.5}$$

Sum of Coupling Conditional Probabilities (SCCP) SCCP measure the sum of the conditional probabilities of each pattern associated to a source file f_p,

$$\text{SCCP}(\text{f}_p) = \sum_{pt_j \in PT \wedge f_p \in pt_j} \text{p}(\text{PT}{=}\text{pt}_j \ , \text{F}{=}\text{f}_p) \ /\text{p}(\text{F}{=}\text{f}_p \ ) \tag{6.6}$$

Weighted Sum of Coupled Source Files having Same Parent Folder (WCSFSPF) : WCSFSPF add all the coupled source files (f_c) frequency (number of time f_c coupled with f_p), such that both f_c and f_p belongs to the same directory or same parent folder.

$$\text{WCSFSPF}(\text{f}_p) = \sum_k \text{Frequency}(\text{f}_k \in \text{f}_c) \tag{6.7}$$

where, parent folder of f_p = parent folder of f_k

Weighted Sum of Coupled Source Files having Different Parent Folder (WCSFDPF) : WCSFDPF add all the coupled source files (f_c) frequency (number of time f_c coupled with f_p), such that both f_c and f_p belongs to the different directory or different parent folder.

$$\text{WCSFDPF}(\text{f}_p) = \sum_k \text{Frequency}(\text{f}_k \in \text{f}_c) \tag{6.8}$$

where, parent folder of $\text{f}_p \neq$ parent folder of f_k

Interpretation of Logical Coupling Metrics :

In the previous section, we formally defined a set of logical coupling metrics. In this section, we discuss some of the important interpretation of the defined set of metrics.

The set of logical coupling metrics can be categorized into three main categories, like the following metrics set i.e., COCSF, CPC, SCPF, ACPF and MCPF, belongs to the category that measures coupling patterns of a source file. Whereas, metric SCCP belongs to the category, which measures the probabilistic value of coupling patterns. Finally, the metrics set WCSFSPF and WCSFDPF belongs to the category of metrics, which exploits the cross cutting concerned nature of a source file. To understand the semantic of logical coupling's measure, consider the following examples,

Example-1: Let we have two source files, A and B. Both the source files have been changed 10 and 20 times respectively. Suppose that file A is coupled with file C in 4 transactions, whereas file B is coupled with file D in four transactions. Now, we compute their logical coupling metrics values using the relations which we have discussed in previous section, i.e., COCSF(A)=COCSF(A)=1,

CPC(A)=CPC(B)=1, SCPF(A)=SCPF(B)=4, ACPF(A) = ACPF(B)=4 and MCPF(A) = MCPF(B)=4. These metrics values show that both source files have same logical coupling measures, but the values of SCCP(A)=4/10=0.4 is different from SCCP(B)=4/20=0.2.

Example-2: Now, consider the example of Figure 2. Let FP={f_1} has been changed 10 times, out of which 3 times f_1 were changed to fix the bug with five other source files i.e., FC={f_2,f_3,f_4,f_5,f_6}. The number of all the possible combination of bug fix transaction is 2^3-1=7. Let source files f_2,f_3 and f_4 have different parent folder as compared to the parent folder of f_1. Similarly, f_5, f_6 and f_1 have the same parent folder. Now, the calculated set of metrics according to the relation given in Section 4.2 are, COCSF(f1)=5(total number of coupled files is 5), CPC(f1) = 3 (From Figure 2. the obtained number of frequent pattern), SCPF(f1) = 3+2+1=6, ACPF(f1) =6/5, MCPF(f1) =3, SCCP(f1) =3/10+2/10+1/10=0.6, WCSFSPE(f1) =2 and WCSFDPE(f1) =3

Source File Metrics Set

In order to perform a comparison, we also computed a group of metrics, which may be used to measure the coupling tendency of C files. We selected only those metrics, which are valid for source files written in C-language. Because, we performed this experiment on GNOME project data. Whereas, GNOME is mainly developed in C-language. Following, are the set of classical metrics, which we used in our experiment for comparison purpose.

1. LOC : Line of Code, which count all the executable lines of source code present in a source files. FINC : File included, which simply count the syntax *include* present in a source file.
2. TFUNC : Total function count, which count all the functions belongs to an individual source file.
3. TFPC : Total function parameter count, which count all the function's parameters presents in a source file.
4. TRP . Total return point, which count all the returns point of each function present in a source file.
5. TCYCLO : Total cyclomatic complexity, it add all the cyclomatic complexity related to each function present in a source file.
6. FINC : File included, which simply count the syntax *include* present in a source file.
7. TFUNC : Total function count, which count all the functions belongs to an individual source file.
8. TFPC : Total function parameter count, which count all the function's parameters present in a source file.
9. TRP . Total return point, which count all the returns point of each function present in a source file.
10. TCYCLO : Total cyclomatic complexity, it add all the cyclomatic complexity related to each function present in a source file.

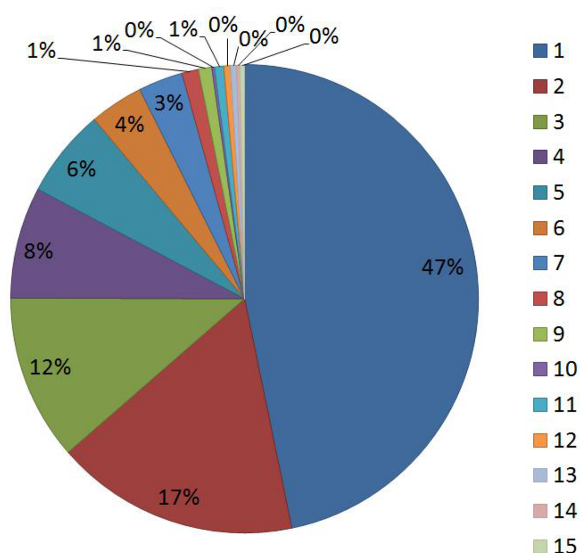


Figure 6.2. Percentage of the selected source files, which were fixed 1-to-15 times. Like the percentage of those source files that were fixed 2-times is 17%. These source files are selected from GNOME project repository to perform the experiment.

6.2 Experimental Setup and Results

In order validate the propose set of metrics and to find its correlation with the number of bugs, we performed an experiment by extracting data from the OSS repository of GNOME project. We downloaded 1,537 source files (c-files) from the GNOME project repository. These source files have been fixed in 2,093 transactions during 2000-2008. After downloaded the source files and their associated transaction's data from subversion, we established a linked between the subversion transaction data and the bug-tracking system. We found that in our data set, the number of bug-fixed transactions associated to each of the source file is lies between 1 to 15, whereas, the distribution of source files on the basis of their number of bug-fix transactions is shown in a pie-chart of Figure 6.2. It is shown in the Figure 6.2 that in our obtained data set, 47% of the source files have only one bug-fix transaction, whereas 53% of the source files have more than one bug-fix transactions. In order to perform the experiment we selected only those source files, which belongs to maximum 15 bug-fix transactions. However, our method is also valid for any large number of transactions, but during experiment, we found that most of the source files were fixed in very few transactions.

6.2.1 Analysis of Metrics Correlation With Number of Bugs

Our main objective is to find the correlation between the set of logical coupling measures and the number of bugs. Furthermore, for comparison purpose we want to obtain the correlation between the classical set of metrics (LOC, Cyclomatic-Complexity, etc.) with the number of bugs. We, therefore, obtained the program source files coupling data and defined a new set of coupling measures, which we have discussed in Section 6.1.2 and Section 6.1.2. Now, in this section, first we discuss the statistical technique which we used to compute the correlation values, and then discuss the obtained results of

our correlation values.

Correlation is a measure of association between two variables. In our experiment, we found that most of the metrics are non-linear related with the bug count. Figure 6.3, depicts the scatter plots of the concern metrics versus bug count, which shows that most of the metrics are non-linearly related with the number of bugs. However, there are few metrics, which are linearly related with the number of bugs. We, therefore, used Spearman's rank correlation. Following, are the main reasons,

1. Spearman's rank correlation is valid for both linear and non-linear relationships between the correlated variables, and in our experiment most of the metrics are non linearly correlated with the number of bugs
2. Since we don't know exactly which distribution is followed by the obtained metrics data set. Therefore, performing parametric statistics, i.e., using Pearson correlation is not a valid solution. Hence, the best solution for our experiment is Spearman's rank correlation, because it is belongs to non-parametric statistics.

Before computing the correlation values, we obtained the values of eight different logical-coupling metrics by using the relations which are given in Section 6.1.2. Similarly, we obtained the six different classical metrics, which are given in Section 6.1.2. Once we obtained the metrics data and the number of bugs related to each source file, then we used these data to compute the Spearman's rank correlation values between metrics and bug counts. Furthermore, in order to find the collinearity among the metrics values, we also computed the correlation values between each pair of metrics.

The obtained results of correlation are shown in Table 6.1. Most of the logical coupling metrics highly correlated with the number of bugs. While, in case of classical metrics set, unfortunately non of the metrics are found to be highly correlated with the number of bugs. We obtained correlation values at 0.01 level of significance. In case logical coupling metrics, the maximum obtained Spearman's rank correlation is for SCPF and CPC metrics i.e., 0.982 and 0.938. While, lowest in case of WCSFDPF, i.e., 0.343. The other metrics like COCSF, WCSFDPF, ACPF, MCPF, and SCCP have a positive and high correlation with the number of bugs, most of them have a correlation value greater than 0.65.

SCPF and CPC are directly related to the number of distinct patterns. Whereas, each pattern of a source file is a distinct group of one or more associated coupled source files. It means that SCPF and CPC are directly related with the degree of scattering of a source file, and it is found in one of the experiment performed by Eaddy M. that if the degree of scattering is high then the source files are more error prone [Eaddy et al., 2008a]. WCSFDPF, count the number of coupled file in different parent folders, therefore, we can assume that this metric is measuring the cross cutting concern. Because, in case of software development, source files are normally grouped into one separate folder to perform some specific task or used for some specific concern. Therefore, if the coupled source files are belongs to different parent folder folders, then it means that these source files are candidate for cross cutting concerns. Therefore, one might thinks that WCSFDPF should have a high correlation with the number of bugs, but in our experiment, we found small correlation value for WCSFDPF, the possible reasons may be the size of data which we used to find correlations, and second possible reason is, GNOME project development is more modular and therefore small number of coupled source files which are

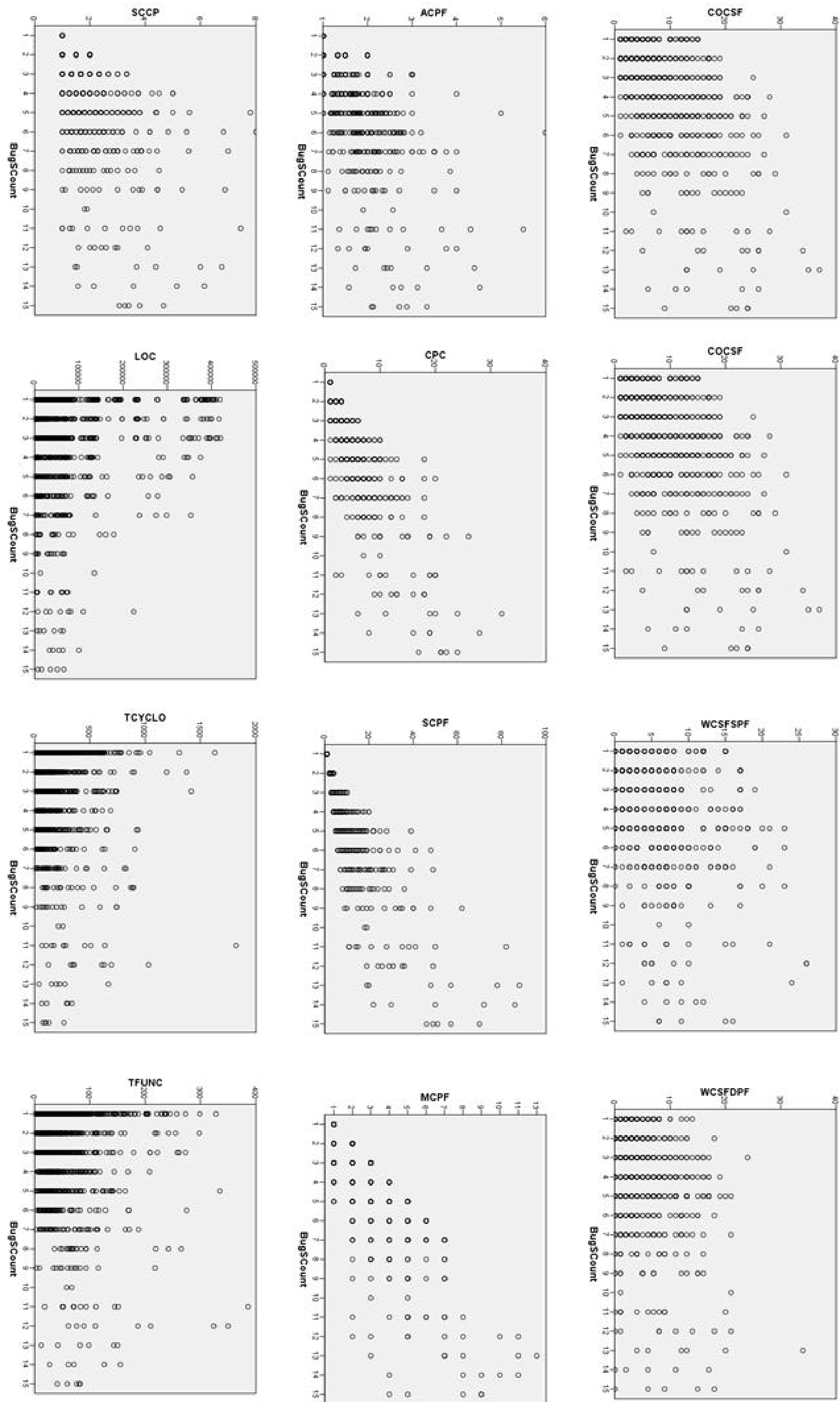


Figure 6.3. Scatter Plots of the logical coupling metrics versus number of bugs.

Table 6.1. Spearman's Correlation Between the Program Files Coupling Measures and the Number of Bugs

	COCSF	WCSFSPF	WCSFDPF	CPC	SCPF	ACPF	MCPF	SCCP	LOC	FINC	TFUNC	TFPC	TRP	TCYCLO
BugsCount	0.663**	0.538**	0.343**	0.938**	0.982**	0.832**	0.876**	0.726**	0.032	0.052*	0.107**	0.100**	0.060*	0.060*
COCSF		0.750**	0.517**	0.760**	0.700**	0.468**	0.558**	0.649**	0.092**	0.126**	0.03	0.023	0.007	-0.029
WCSFSPF			-0.046	0.604**	0.568**	0.411**	0.473**	0.535**	0.118**	0.141**	0.019	0.004	-0.018	-0.048
WCSFDPF				0.392**	0.345**	0.187**	0.252**	0.280**	0.01	0.025	0.036	0.053*	0.031	0.023
CPC					0.963**	0.705**	0.792**	0.810**	0.029	0.064*	0.082**	0.074**	0.038	0.028
SCPF						0.860**	0.908**	0.830**	0.019	0.041	0.083**	0.075**	0.042	0.034
ACPF							0.978**	0.787**	-0.013	-0.019	0.053*	0.046	0.026	0.02
MCPF								0.852**	0	0.003	0.060*	0.053*	0.022	0.017
SCCP									-0.017	0.007	0.02	0.009	-0.012	-0.039
LOC										0.934**	0.225**	0.212**	0.202**	0.200**
FINC											0.192**	0.182**	0.142**	0.134**
TFUNC												0.953**	0.805**	0.827**
TFPC													0.789**	0.834**
TRP														0.883**

**. Correlation is significant at the 0.01 level (1-tailed).

*. Correlation is significant at the 0.05 level (1-tailed).

belongs to different parent folder. Whereas, in case of classical metrics set, the possible reason to have a very small correlation values is their low capability to measure the coupling tendency of a source file.

Finally, we may conclude on the basis of correlation analysis that the high correlation of logical-coupling metrics with the number of bugs reveals that they can be used to construct a bug predictor or classification model. Therefore, in the next section we discuss the prediction and classification capabilities of logical coupling metrics.

6.2.2 Bug Prediction and Classification Using Logical-Coupling Metrics

In software engineering metrics are commonly used for the prediction and the classification of bug count. In the recent years several works have been done to provide empirical evidence that metrics can predict post-release defects [Nagappan et al., 2006]. Therefor, the main objective of this section is to analyze bug prediction and classification capabilities of logical coupling metrics.

In order to perform the experiment, we used *Stepwise Regression Analysis*, which first build regression model using a metrics that have largest correlation with the number of bugs. We then add further metrics to the model based on their partial correlation with the metrics that already in the model. After adding a new metric into the model, the model is evaluated and metrics that do not contribute significantly removed, so that, in the end, we got a model which consists of the metrics that explains the maximum variance is left. The amount of variance explained by model is give by R^2 [Eaddy et al., 2008a]. In statistics, R^2 is the proportion of variability in a data set that is accounted for by a statistical model. We, also obtained the value of Adjusted R^2 and the standard error of estimate. Adjusted R^2 is used to measure any bias in the R^2 measure. Whereas, standard error of estimate measure the deviation of the actual bug value from the bug value predicted by the model. The obtained results of *Stepwise Regression Analysis* is shown in Table 6.2.

Table 6.2, represent the stepwise regression results. It is given in the table that maximum variability, i.e., R=97.2% and R^2=94.6% explained by the three metrics, CPC, MCPF and SCCP. Whereas, after including the metrics ACPF, SCPF, WCSFSPF and TFUNC into the model. The increase in the variability is small. The same pattern is found in cas of R^2 and Adjusted R^2. The small difference between the values of R and R^2 reveals that bias is not present in our model.

Table 6.1, results show that the correlation between some metrics is very high, which indicate that our data set contains collinearity. Therefor, the obtained results of correlation and stepwise regression maybe over fitted. In order to overcome this collinearity from the data set we used *Principal Component Analysis* (PCA). In case of PCA, we transform the large number of metrics into the small number of uncorrelated weighted combination of metrics. The weighted combination of metrics are called principal components.

Table 6.3, shows the value of total variance explained by each principal component. Whereas, Table 6.4, shows the obtained four principle components. The results of Table 6.3 show that PCA 1 cover 36.5% and PCA 2 define 25.8% variability. Table 6.3 and Table 6.4 are divided into two parts left part of the table shows the value of principal component without rotation sum of square loading.

While right part of the table shows the PCA values with rotation sum of square loading. In case of PCA the best practise is to use PCA with rotation sum of square loading. Therfore, to obtain a new set of component we used PCA with rotation sum of square loading, which are shown in the right part Table 6.4. Once we obtained the principal components, then we used them for regression analysis. The results of PCA based regression model for bug predictor is shown in Table 6.5. The obtained results of Table 6.5, shows that the value of R and R^2 are 89% and 80%. If we compare the R and R^2 results of PCA based regression model with Stepwise Regression model, which is not PCA based, then we find that the PCA based regression value is less as compared to non PCA based model. Although, the regression values of stepwise regression model is high as compared to PCA based regression model, but they are overfitted, because their results are obtained from the data set that contained collinerity. However, PCA based regression model is more reliable, because their regression values are obtained from data set, which do not contain any collinearity.

After performingthe bug predictionexperiment, we performedanother experimentto checkthe classification performanceof logicalcouplingmetrics. For classification we used machine learning algorithmJ48. To evaluatetheclassification performance we, used three different classification evolution measures i.e., Precision, Recall and Accuracy. Precisionisdefinedas the numbers of relevant documents retrieved by a search divided by the total number of documents retrievedby thatsearch, and Recallis defined as thenumber of relevant documents retrieved by a search divided by the total number of existing relevant documents [Thomas Zimmermann, May 23-28, 2004]. of Table 6.5, shows that the value of R and R^2 are 89% and 80%. If we compare the R and R^2 results of PCA based regression model with Stepwise Regression model, which is not PCA based, then we find that the PCA based regression value is less as compared to non PCA based model. Although, the regression values of stepwise regression model is high as compared to PCA based regression model, but they are overfitted, because their results are obtained from the data set that contained collinerity. However, PCA based regression model is more reliable, because their regression values are obtained from data set, which do not contain any collinearity.

Before classification, we categorized the number of training instances (obtained data set) into three different class categories i.e., LOW, MEDIUM and HIGH. The defined criteria for each class category is the number of bugs. We performed the experiment three times with the same training instances but with different criteria for each class category. All the data set contained the same total number of instances, i.e., 1537. However, the number of instances per category is different in each data set. The obtained results of classification is given in Table 6.6 and Table 6.7

In case of data set A (shown in Table 6.6), class LOW is assigned to those files that were fixed only one time, while MEDIUM is assigned to those files that were fixed 2 to 5 times. Finally, we, assigned class HIGH to those source files that were fixed more than 5 times. Similarly, we changed the criteria of each class in case of data set B and data set C. The main reason to use different criteria for each class is to analyze the impact of class selection criteria on the classification results. The results of Table 6.6, depicts that there is no impact of class selection criteria on the classification results. The obtained accuracy, precision and recall values are approximately same, i.e., around 97% for all the three classification experiments, which indicate that we can use logical coupling metrics for the classification of source files as LOW, MEDIUM and HIGH level of complexity. High complex source

Table 6.2. Stepwise Linear Regression (Model Summary)

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	0.868_a	0.754	0.754	10.199
2	0.932_b	0.869	0.869	0.876
3	0.972_c	0.946	0.946	0.564
4	0.976_d	0.953	0.952	0.527
5	0.977_e	0.954	0.954	0.518
6	0.977_f	0.955	0.955	0.513
7	0.977_g	0.955	0.955	0.511

a. Predictors: (Constant), CPC
b. Predictors: (Constant), CPC, MCPF
c. Predictors: (Constant), CPC, MCPF, SCCP
d. Predictors: (Constant), CPC, MCPF, SCCP, ACPF
e. Predictors: (Constant), CPC, MCPF, SCCP, ACPF, SCPF
f. Predictors: (Constant), CPC, MCPF, SCCP, ACPF, SCPF, WCSFSPF
g. Predictors: (Constant), CPC, MCPF, SCCP, ACPF, SCPF, WCSFSPF, TFUNC

Table 6.3. Total Variance Explained (PCA)

Comp	Initial Eigenvalues			Rotation Sums of Squared Loadings		
	Total	% of Var.	Cumulative	Total	%of Var.	Cumulative
1	5.119	36.565	36.565	3.766	26.903	26.903
2	3.622	25.870	62.436	3.512	25.083	51.986
3	1.759	12.568	75.003	2.506	17.902	69.888
4	1.132	8.085	83.088	1.848	13.201	83.088

files, means it has fixed large number of bugs and therefore, need to be restructured. The classifiers performance at individual class level is shown in Table 6.7.

6.3 Threats to Validity

Size of the data: We performed the experiment by selecting only one project data, i.e., GNOME. We selected GNOME, because of the following two reasons. The first one is, GNOME is using Subversion, and in case of Subversion it is easy to identify a unique transaction as compared to CVS. While, the second reasons is GNOME is using C-language for the development purpose. While, C-language is not pure object oriented programming language, and initially we want to study logical coupling analysis for those source files which are not developed using object oriented programming languages. Therefore, selecting a single project data not only reduce the size of our data, but also decrease the validity of the results. Furthermore, we used only bug-fix transaction data for our experiment, due to which the size of the data became small. While, the statistical correlation analysis and machine learning classification are highly dependent on the size of the data.

Bug reports mapped to Bug-Fix Transactions: In our experiment, we need only those transaction's data, which have been used in fixing of bugs. Unfortunately, there is no direct way to identify a transaction as a bug-fix transaction. We used a heuristic approach to identify a transaction as bug-fix transaction, which is not necessarily correct for all the cases. The second big issue is related

Table 6.4. Extracted Four Principle Component Using Principle Component Analysis Method

Component Matrix					Rotated Component Matrix$_a$				
Metrics	Component				Metrics	Component			
	1	2	3	4		1	2	3	4
CPC	0.93	-0.07	0.02	-0.09	MCPF	0.92	0.04	0.17	-0.05
SCPF	0.92	-0.09	-.06	0.10	ACPF	0.90	0.05	-.07	-0.05
SCCP	0.88	-0.17	-0.03	0.06	SCPF	0.82	0.04	0.44	-0.04
MCPF	0.84	-0.08	-0.16	0.36	SCCP	0.76	-0.04	0.47	-0.05
COCSF	0.80	-0.08	0.27	-0.44	CPC	0.71	0.06	0.69	-0.01
ACPF	0.69	-0.06	-0.21	0.55	TCYCLO	0.02	0.95	-0.02	0.09
WCSFSPF	0.62	-0.05	0.29	-0.04	TFUNC	0.04	0.94	0.04	0.12
TFUNC	0.18	0.92	-0.13	-0.04	TFPC	0.04	0.94	0.02	0.11
TFPC	0.17	0.92	-0.14	-0.04	TRP	0.01	0.89	-0.01	0.02
TCYCLO	0.13	0.92	-0.20	-0.03	COCSF	0.37	-0.01	0.88	0.12
TRP	0.12	0.86	-0.21	-0.06	WCSFDPF	0.08	0.03	0.82	-0.09
FINC	0.01	0.32	0.85	0.18	WCSFSPF	0.44	-0.03	0.47	0.26
LOC	-0.01	0.42	0.82	0.22	LOC	-0.04	0.17	-0.01	0.93
WCSFDPF	0.54	-0.07	0.09	-0.62	FINC	-0.05	0.08	0.04	0.93

a: Rotation converged in six iterations

Table 6.5. Regression After Applying PCA (Model Summary)

Model	R	R Square	Adjusted R Square	Std. Error of the Estimate
1	0.895	0.80	0.80	1.081

to the mapping of a bug-report with a relevant bug-fix transaction. Again there is no direct link between bug tracking systems, i.e., Bugzilla and Subversion. Therefore, we used heuristic approaches to obtain a link between a bug report and a bug fix transaction, which is not necessarily true for all cases.

Threats to external validity: Making a general conclusion on the basis of our results is considered as a threat to external validity. Although the experiment was designed and performed with care. But still one cannot confirm that the outcome of the experiment has no error. Similarly, one cannot confirm the generalization of the findings. In our case, we defined a set metrics for logical coupling measures, and we performed the experiment only on C-files, and assumed that it may perform well in other type of source files. Second threat is we performed the experiment on OSS project development data, while the development strategy of closed software system is different as compared to the OSS. These are some threats, which we will handle in our future works, specially we are currently focusing to repeat the experiment using those project data, which have been developed using C++ and Java languages.

6.4 Summary

In this chapter, we presented our approach to extract the bug fixing coupling patterns and used them to define a set of logical coupling metrics. First, we described in details the method of pattern mining and defined our own customize method to extract the different set of bug fix patterns from the evolution data of GNOME project. Once we obtained the coupling patterns, we used these patterns and defined eight different logical coupling metrics. We have shown that these metrics are highly correlated with the number of faults and can be used to construct a regression based bug prediction model. We used the same metrics and built a machine learning based fault prediction

Table 6.6. Classification of source files on the basis of complexity level i.e., low, medium and high

Data Set	Total Instance Size	Class Distribution of Source Files			J48 (Without PCA)			J48 (With PCA)		
		LOW	MEDIUM	HIGH	Accuracy	Precision	Recall	Accuracy	Precision	Recall
		Bugs : Instance	Bugs : Instance	Bugs : Instance						
A	1537	1 : 676	2-5 : 689	$>$ 6 : 172	98.8	98.8	98.8	96.8	96.7	96.68
B	1537	1-2 : 945	3-5 : 420	$>$ 5 : 172	97.65	97.7	97.7	97.2	97.2	97.2
C	1537	1-5 : 1365	6-10 : 137	$>$ 10 : 35	97.59	97.6	97.6	95.57	95.3	95.6

Table 6.7. Classification evaluation measures of each class with PCA (training instances = 1537)

Data Set	Class	Instances	TP Rate	FP Rate	Precision	Recall	F-Measure	ROC Area
A	HIGH	172	0.936	0.005	0.958	0.936	0.947	0.985
	MEDIUM	689	0.99	0.013	0.984	0.99	0.987	0.995
	LOW	676	1	0	1	1	1	1
B	HIGH	172	0.907	0.015	0.886	0.907	0.897	0.955
	MEDIUM	420	0.952	0.014	0.962	0.952	0.957	0.977
	LOW	945	1	0	1	1	1	1
C	HIGH	35	0.686	0.004	0.8	0.686	0.738	0.938
	MEDIUM	137	0.905	0.017	0.838	0.905	0.87	0.952
	LOW	1365	0.99	0.041	0.995	0.99	0.993	0.985

model. Furthermore, in this chapter we discussed the results of an experiment which was performed with data of GNOME project. In case of bug predictor model, the obtained value of R^2 is 80%. Whereas, in case machine learning based classification, the obtained value of accuracy is 97%. The results presented in this chapter have some threats, like size of the data, identification of bug fix patterns, etc. These threats are described in details in the last part of the chapter.

7

Impact Analysis of Software Change Request Using Single and Multi-label Classification

Parts of the contents of this chapter have been published in [Ahsan and Wotawa, 2010b].

To add a new change or to fix a fault in a software, a software change request (SCR) is generated. Most of the software change requests are related to the software corrective maintenance task, and are called bug reports. The accurate and timely resolution of these bug reports not only improves the quality of software maintenance task but also provides the basis to keep particular software alive. The handling of software change requests (SCRs) is an integral part of software maintenance and a task running daily for larger applications.

Current Research Trends: In the recent years information retrieval and machine learning techniques have been applied on SCRs data to perform single label classification of SCRs. The research trend is to use the textual data of SCRs and apply vector space model to transfer the textual data of SCRs into key terms and then apply single label machine learning classification technique to classify SCR either for bug triaging [Cubranic, 2004] [Anvik et al., 2006], effort prediction or impact analysis. Zimmermann et al. [Thomas Zimmermann, May 23-28, 2004] applied data mining techniques on software change data. They obtained information regarding which entities changed together in the past and used this information to guide the developers in case of new changes. They used association rules to perform the experiments. Canfora et al in 2008 [Canfora and Cerulo, 2005] presented their approach for the impact analysis of SCRs. They used resolved SCR data along with developer CVS comments and the name of the impacted source files. After obtaining the data, they used information retrieval technique for the impact analysis of SCRs. Although the current approaches have performed well, but we found that there are some issues, when we used single label classification directly for the impact analysis of SCRs. These issues are discuss in the following paragraph.

Motivations and Findings: SCRs are frequently generated during the evolution of any large software system. To establish this claim, we have analyzed the reported SCRs of the Mozilla and Eclipse project. We found that in case of Mozilla a maximum of 393 SCRs had been reported in a single day. The average reported number of SCRs per day is 97.5. But even more there are about 1,534 days out of total 3,185 days (during the years 1999-2007) when more than 100 SCRs have been reported. In Figure 7.1 we show the frequency of the SCR count reported per day. Within the Eclipse

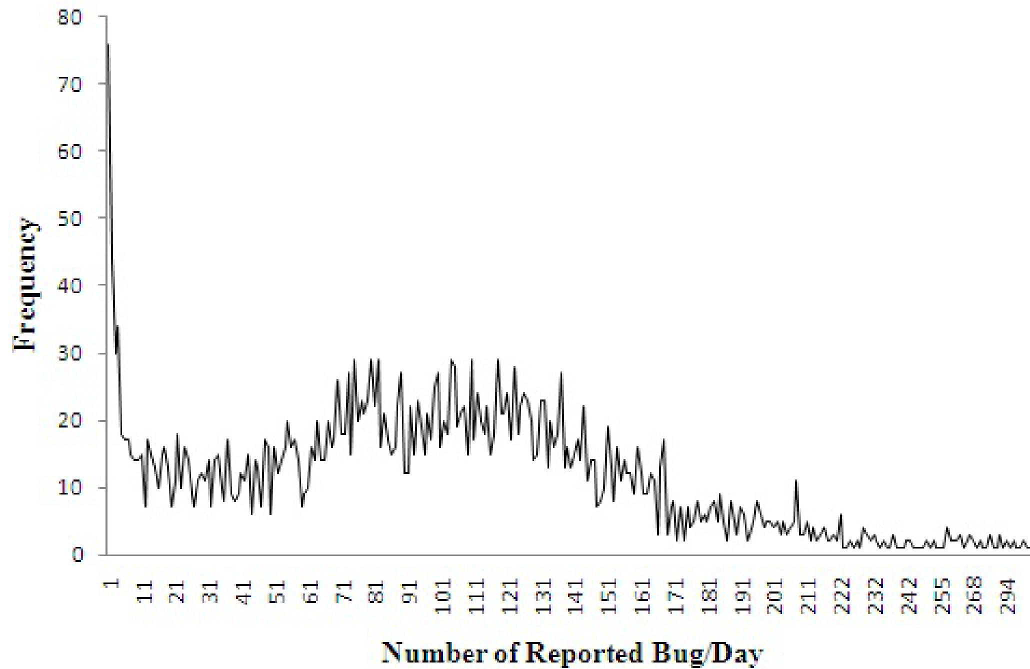


Figure 7.1. Frequency of submitted bug reports during the evolution of Mozilla project

project the figures are similar. The average number of SCRs reported between Revision 3.0 and 3.1 is 37 reports per day. Whereas a maximum of 220 SCRs had been reported in a single day [Anvik et al., 2006]. Given the fact that the number of reported SCRs per day is high, the task of handling SCRs becomes a very demanding one.

The process of handling SCRs comprises several actions including the assignment of SCRs to the most appropriate developer, the identification of those program files and parts of the programs to be changed, and estimating the expected time required for resolving the SCR. All these actions are closely interconnected and have to be performed based on available information mostly provided in textual form. An automatic tool capable of performing all the above mentioned tasks potentially leads to an improvement of the overall maintenance task. In particular the maintenance time and effort can be decreased. Therefore, an automated tool for handling SCRs would not only support the person in charge for SCR handling but also reduce maintenance time and effort.

The timely and error free fixing of SCRs plays an important role to ensure a reliable software maintenance task. Ensuring reliability and thus program quality can be obtained by performing a comprehensive change impact analysis prior to the implementation of the SCRs. The change impact analysis avoids to create faults in programs due to SCRs. A well known and major issue in software engineering and in software maintenance in particular is that apparently small changes might ripple through the system and cause major unintended impacts somewhere else. This effect is called ripple effect. Hence, changing programs without understanding the corresponding effects definitely lead to unreliable software products [Lee, 1998]. In order to avoid ripple and other effects leading to unreliable

Table 7.1. Percentage of bug fix transactions in which single or multiple source files have changed

Project Name	**Mozilla**	**Eclipse**	**Gnome**
Duration in which Bug-Fix Transactions Occurred	1999-2009	2001-2009	1997-2008
Bug-Fix Transactions (sample-size)	33617	3599	68058
Bug-Fix Transactions Impacted Single Source File	24602 (73.2%)	1960 (54.7%)	51018 (74.9%)
Bug-Fix Transactions Impacted Multiple Source File	9015 (26.8%)	1630 (45.3%)	17040 (25.1%)
Average Bug-Fix Transactions Impacted Single Source File	67.6%		
Average Bug-Fix Transactions Impacted Multiple Source File	32.4%		

software, different change impact analysis approaches for predicting effects of SCRs have been introduced, described, and used. Some of the approaches are based on static impact analysis and others on dynamic impact analysis. The latter are based on the analysis of the program's behavior [Arnold and Bohner, 1993] and are usually more expensive in terms of computational time and space requirements when compared with static impact analysis methods [Huang and Song, 2006][Chen et al., 2007].

Problem Statement: In case of SCR's impact analysis, there are some cases in which SCR impacted more than one source files. Therefore, to perform the supervised machine learning classification, some SCRs should be labeled with multiple source file names. Hence, single label classification cannot be used directly for the impact analysis of SCRs. To get an idea of what percentage of SCRs impacted more than one source files (i.e., java, cpp, etc.), we performed an experiment by randomly selecting three different sample data sets of SCRs bug fix transactions from the project repositories of Mozilla, Eclipse, and Gnome. The experiment results are shown in Table 7.1. An average 33.4% bug-fix transactions impacted more than one source files. Therefore, we may conclude that in case of impact analysis of SCRs the traditional single label classification techniques cannot be able to directly classify at least 33.4% SCRs in the giving setting. Furthermore, in case of textual classifcation, we mostly come across high dimensional feature space which reduces the performace of machine learning classifiers, and need to be handeld.

Our Goal and Approach: The main objective of the study presented in this chapter is to resolve the problems stated above, and to explore the possible solutions for the automated impact analysis of SCR. Therefore, in order to resolve labeling issue, we present two different approaches which are based on automated classification of SCR. The first approach is two-fold and semi-automated, in which we use an apriori algorithm to obtain different groups of source files. We use these groups to label resolved SCRs, and perform *Single-label* classification. Whereas, the second approach is one-fold and fully automated, in which we directly label the resolved SCRs with one or more impacted source files and perform *Multi-label* classification. In order to handle the high dimensional feature space we use the LSI technique from information retrieval, which is commonly used to transfer the high

dimensional feature space into a semantic space with less dimensions.

Hypothesis and Contributions: The hypothesis behind our approach is that the history of resolved software change request together with the list of its impacted source files can be used effectively for impact analysis of new SCRs. The main contributions of the paper are:

1. We used 3 different approaches of information retrieval and machine learning to improve the existing textual classification of bug reports These approaches are: (a) LSI to overcome the existing issues of VSM, such as high dimensionality, synonymy and polysemy, (b) labeling the bug-fix reports with multiple impacted source files and using multi-label machine learning, and (c) labeling the bug-fix reports with a single label obtained from file clustering that is based on pattern mining and association rules.
2. We present the results of an empirical study based on three OSS projects for comparing the different techniques.

Chapter Layout: The rest of the chapter is organized as follows: In Section 7.1 we describe our approach. In Section 7.2 we describe the data extraction process. In Section 7.3 we discuss the indexing and dimension reduction techniques of information retrieval. In the next Section 7.4 we describe labeling and classification techniques. In Section 7.5 we briefly discuss the different evaluation measures for the single and multi label classifications. In Section 7.6 we introduce the experimental setup for collecting data and performing machine learning classification and present the obtained empirical results, and also discuss the results. Finally in Section 7.7, we summarize the paper.

7.1 Our Approach

We propose two different approaches for impact analysis of SCRs. Our proposed approach belongs to static impact analysis, and is based on the information gained when mining software repositories. The pictorial representation of our approach is shown in Figure 7.2. We use a text similarity technique from information retrieval and machine learning to extract similarities of textual SCRs descriptions together with information regarding the corresponding impact on one or more source files. In the first approach (see Figure 7.2 Steps: 1, 2, 3, 5 and 7), we first extract the data of bug fix transaction (a CVS commit in which bugs are fixed) and then apply the pattern mining technique on the data, and obtain different sets of source files, which are frequently changed together in order to fix a bug. With the help of these patterns, we define association rules and use them to cluster the source files into distinct groups. We are able to use a single group identifier to label each bug-fix report uniquely. As a consequent a single-label machine learning algorithm for classification can be used again. Currently, this clustering-based approach is only semi-automated. In the second approach (see Figure 7.2 Step: 1, 2, 3, 4 and 5), where no clustering is performed, we directly label each bug report with the set comprising all changed source files and use multi-label machine learning classification directly. The multi-label machine learning based approach is fully automated.

In order to validate our approaches, we performed experiments. In our experimental work, we used bug reports, the corresponding program files, and log data of the three different OSS projects

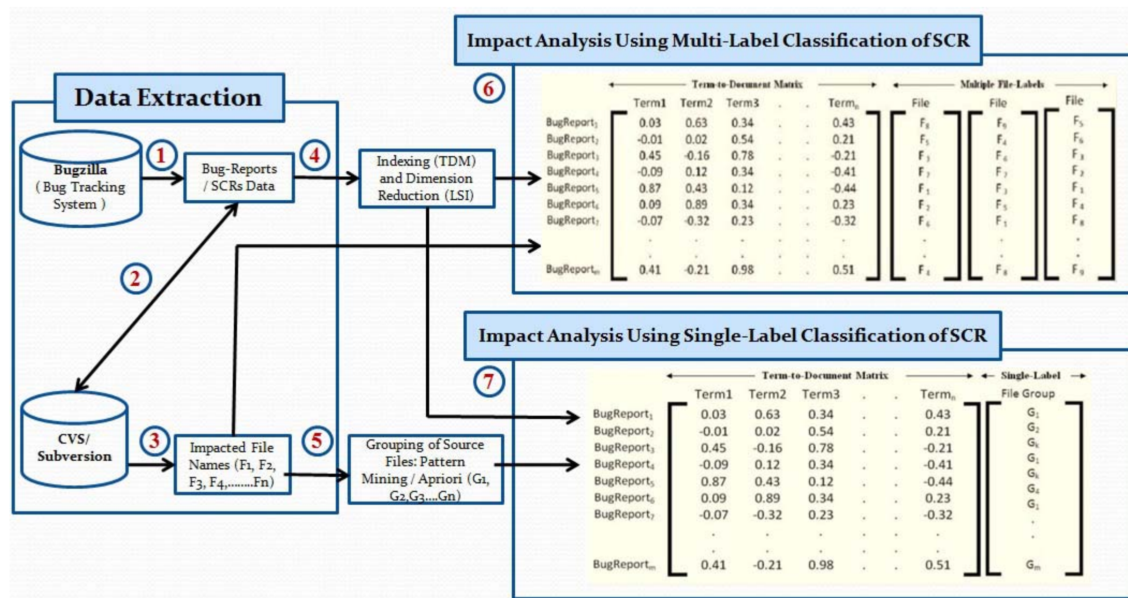


Figure 7.2. Process to convert the textual data of SCRs into LSI based TDM matrix. We took the transpose of actual TDM matrix and then labeled with the names of source files. For labeling and classification two different approaches are used i.e., single and multi label classifications

Mozilla, Eclipse and Gnome. For building the models for our impact analysis technique, we converted the textual part of the available bug reports, i.e., the bug summary and the bug description, into a low dimensional indexed term to document matrix (TDM). This technique usually leads to TDMs of larger dimensions, which reduces the classification performance. Therefore, we made use of *Latent Semantic Indexing* (LSI). Using LSI we also overcome the problems of synonymy and polysemy that exists in text analysis based on classical *Vector Space Model* (VSM). After indexing, we labeled each bug report with one or more impacted source files and performed classification using our two approaches for impact analysis of SCR.

In the the following subsections we discuss the main steps of our approach i.e., *data extraction and filtering*, *indexing and dimension reduction*, and *labeling* in detail.

7.2 Data Extraction and Filtering

We extracted the used textual data of the SCRs from the bug tracking system Bugzilla. In our current work we focus only those SCRs, which are related to corrective software maintenance, i.e., *Bug Reports*. Therefore, we downloaded the bug reports in HTML format from the Bugzilla server of the OSS projects Mozilla, Eclipse and Gnome. Figure 2 shows an example of a Bugzilla bug report. We further processed these reports afterwards to extract the relevant textual information, i.e., the bug report summary and the bug report description. We further filter the summary and description texts. For this purpose we parse the textual data of each bug report and remove all words which, are used very frequently. Such words usually are pronouns, prepositions, and articles, and are called stop-words. Furthermore, we performed stemming. Stemming is a process that removes the suffix

Figure 7.3. An example of bug report's summary and description.

from the words. For example, after stemming the word *walked* becomes *walk* (see [Baeza-Yates and Ribeiro-Neto, 1999]). We implemented the extraction and filtering step and applied it to all bug reports. Hence, we obtained a set of key-words or key-terms for each bug report, which is processed in the next step of our impact analysis approach.

7.3 Indexing and Dimension Reduction

The most commonly used document representation technique is the *Vector Space Model* (VSM). In VSM documents are represented as a vector of terms. In VSM each column represents a unique document, whereas each row contains the frequency or weight of a term. Therefore, sometime this representation is also called term-by-document or term-to-document matrix (TDM). The terms may be weighted according to a given weighting model, which may include local or global weighting, or both. If local weights are used, then the term weights usually represent the frequency of term occurrences in that document, i.e., the term frequencies (TF). If global weights are used, then the weight of a term is given by its inverse document frequency (IDF) values. The most common weighting scheme is one in which both local and global weights are used. This is commonly referred to as TF×IDF weighting [Sebastiani, 2002]. TF×IDF determine the relative frequency of words in a specific document compared to the inverse proportion of that word over the entire document corpus. Intuitively, this calculation determines how relevant a given word is in a particular document. Words that are common in a single or a small group of documents tend to have higher TF×IDF numbers than common words. Therefore to perform a classification of SCR, we use TF×IDF for indexing. The obtain TDM is high dimensional. When using the TDM in this form for classification the computational costs are high and the generated results are very likely to be poor. Therefore, we use the *Latent Semantic Indexing* (LSI) technique for reducing the dimension.

7.3.1 Latent Semantic Indexing

In VSM, similarity between the two documents e.g., d_1 and d_2 is tested lexically by taking the cosine of angle between the indexed terms of the corresponding vectors $\mathbf{d}_1$ and $\mathbf{d}_2$.

There are two main drawbacks of VSM. The first one is that VSM documents are presented in a high dimensional space. The dimensions depend on the number of indexing terms. The second drawback is the used lexical matching technique, which may retrieve irrelevant or inaccurate results because of synonyms and polysemy words. Polysemy (words having multiple meaning) and synonymy (multiple words having the same meaning) are fundamental problems in classical VSM. To overcome these issues Scott Deerwester in 1990 described a new technique i.e., *Latent Semantic Indexing* (LSI), which identifies some underlying latent semantic structure in the data [Deerwester et al., 1990]. LSI transforms the high dimensional TDM data into a low dimensional semantic space. In the semantic space terms and document that are closely associated are placed close to each other. E.g., documents, which contain synonyms, are closer together in LSI space than in the original space. Documents, which contain polysemy in a different context are more far away from each other in LSI space than in the original space. Because of the mentioned advantages of LSI we use this method instead of VSM in our approach. In the next paragraph we formally describe LSI using SVD.

Let us assume an $n \times m$ term-to-document matrix D, where n is the number of key-terms and m is number of documents. The singular value decomposition of matrix D is given by $D = USV^T$, where U and V are orthonormal vectors and S is a diagonal matrix with S=diag(σ_1, σ_2, σ_3,.. σ_r) and r is the rank of D. σ_1, σ_2, σ_3,.. σ_r are singular values of D. $U = [u_1, u_2, u_3, ...u_r]$ called the left singular vector. $V = [v_1, v_2, ...v_r]$ is called the right singular vector (U represents term vectors, and V represents document vectors). We truncate the singular values, i.e., keep only the first k largest terms. LSI used the first k vectors in D as the transformation matrix to convert the original document into a k-dimensional space. Therefore the resulting matrix X_k is given as: $X_k = U_r\ D_r\ V_r^T \approx U_k\ D_k\ V_k^T$, where U_k is a $m \times k$ matrix, D_k is a $k \times k$ matrix and V_k^T is a $k \times n$ matrix. The new matrix X_k is approximately equal to X and is of rank k. Therefore, SVD can be considered as a technique for deriving a set of uncorrelated indexing variables or dimensions. The reduction in the dimension is based on the value of k. In LSI there is no direct way of finding an optimum value for k. This raises the important and open problem of choosing the dimensionality [Deerwester et al., 1990]. In our case we repeat the experiment with different values of k and found the best classification results at k=100 for all data sets. After indexing and dimension reduction the next step is labellings and classification.

7.4 Labeling and Classification

In case of text classification problems, two different approaches are used to label a document i.e., single-label or multi-label. We use both approaches. Before start labeling we extract the set of impacted source files related to each bug report. This task is accomplish in two steps. In the first step, we establish a link between the bug reports and the associated check-ins in the version control system e.g. CVS (in case of Mozilla and Eclipse) and Subversion (in case of Gnome). In check-in developers make changes in the source files and save those changes into CVS/Subversion. We obtain these links by processing the log data of CVS/Subversion, and obtain the number of check-in transactions that

are related to bug fix changes. To identify the bug fix transaction we parse the developer's check-in comments. If comments contain any of the following words *bug fix*, *bug fixed*, or *fixed*, or any *bug-id*, then we consider this transaction as a bug fix transaction [Fischer et al., 2003]. After identifying the bug fix transaction, the second step is to list the number of impacted source files in that transaction. In case of Subversion all those files, which are committed together have the same revision number. Therefore, in case of Subversion one can easily obtain the list of source files that are changed in a single transaction. While, in case of CVS the source file change together do not have the same revision number. Therefore, in case of CVS we use the *Sliding Time Window* approach introduced by Zimmermann [Thomas Zimmermann, May 23-28, 2004]. According to Zimmermann's approach, files are considered to be changed together, if the log of each of that files have the same log comments, the same author name, and the same date. The check-in time may be different, but this difference should not be more than 20 seconds.

Once we extract the list of impacted source files the next step is to use these information to label bug-reports data with set of impacted source files. In the following we describe the two different approaches for labeling of bug reports with the set of changed source files.

7.4.1 Single-Label Classification

In case of single-label classification the predefined data categories are mutually exclusive and each data or document may belong to exactly one class. A binary classification is the simplest case of the single-label classification, where each data point label is assigned to one of two predefined classes. Whereas in case of single-label multi-class classification, where each data point label is assigned to one of multiple predefined classes. In our experiment we used the single-label multi-class approach, because in different bug-fix transactions large number of source files are changed. There are some cases in which a bug-fix transaction impacted more than one source files. Whereas, in case of single-label multi-class classification we have to label each bug report with a single entity or class at a time. To overcome this issue we label fixed bug reports with a group-id. Where a group-id represent a distinct group of source files, which are frequently changed together. To cluster the source files that are changed together, we apply pattern mining technique on the bug-fix transaction's data of CVS/Subversion, and obtain frequent patterns of source files which are changed together. From source files change patterns data we obtain the sets of association rules. These association rules are than used to finally cluster the files into groups.

For mining the association rules from the patterns we use the *Apriori Algorithm*. The problem is to find file change patterns with a user-specified minimum support, where the support of a pattern is the percentage of data that contain the pattern [Agrawal and Srikant, 1994]. The obtain patterns are used to define association rules. Let $F_i = \{A, B, C\}$ be a set of source file items and let $T = \{t_1, t_2, t_3....t_n\}$ be a set of transactions. Let t_1 be a particular transaction in which source files A and B are changed. An association rule is of the form $A \Rightarrow B$ where A, B included in F_i and ($A \cap B = \phi$). The confidence of the rule is the conditional probability of A given B, Pr(A|B), and the support of the rule is the prior probability of A and B, Pr(A and B).

To obtain a group of source files we applied frequent pattern mining on the bug-fix transactional

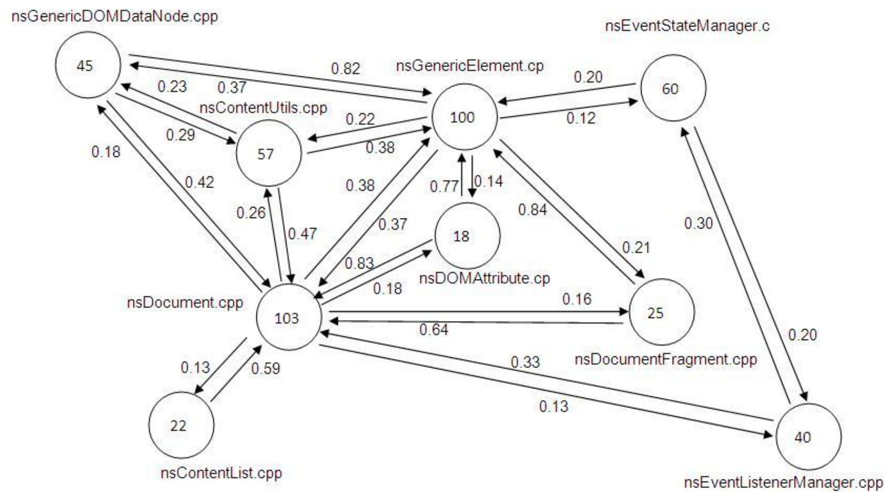


Figure 7.4. An example of a groups of source files which is obtained by applying *frequent Patterns Mining* and *Association Rules* on the bug-fix transaction data of Mozilla CVS repository.

data of three different OSS projects, and obtained patterns related to program file changes. We used these frequent patterns data along with the frequency and confidence value to define rules. Figure 7.4 depicts an example of such a group of source files, which we extracted from the data of the Mozilla project. Each circle in Figure 7.4 represents a source file and the integer value inside the circle represent the change frequency of that file. Whereas, the directed arcs represent the confidence values of the association rules stating that the two connected source files are changed together. In particular if we have a source file f_1 and another one f_2, then the graph comprises the directed arc (f_1, f_2) in case a change of f_1 may causes a change of f_2 with a given confidence. We obtain these relationships and corresponding confidence values from the obtained association rules.

We, used the directed graph, the minimum support value and the confidence value to build the group of source files. First, we extracted a sub-graph only comprising arcs where the provided confidence value is greater than a predefine value. We further removed all nodes that are not connected anymore. The remaining graph might comprise a set of connected sub-graphs. Each of these sub-graphs form a group of source files changed together. We assigned an unique group id to each of these groups.

7.4.2 Bug-Report's Mapping with Source Files' Group-Id

After creating the groups of source files, we labeled each bug report (which are in the form of reduced dimension TDM) with one of the appropriate group id, as shown in Figure 7.5. To assign such a group id to each bug-fix report, we used the procedure described in Progrm1. Its main function is to compare each element (i.e., name of impacted source files) of a bug report with each element (i.e., name of source files in a group) of all the groups one by one. Finally, we used the result of this comparison of file names, and assigned a group-id to a bug-report. We assigned only that group-id to a bug report, whose member's file names either perfectly matched with all the impacted file names of

← Term-to-Document Matrix →							← Single-Label →
	Term1	Term2	Term3	.	.	Term$_n$	File Group
BugReport$_1$	0.03	0.63	0.34	.	.	0.43	G$_1$
BugReport$_2$	-0.01	0.02	0.54	.	.	0.21	G$_2$
BugReport$_3$	0.45	-0.16	0.78	.	.	-0.21	G$_k$
BugReport$_4$	-0.09	0.12	0.34	.	.	-0.41	G$_1$
BugReport$_5$	0.87	0.43	0.12	.	.	-0.44	G$_k$
BugReport$_6$	0.09	0.89	0.34	.	.	0.23	G$_4$
BugReport$_7$	-0.07	-0.32	0.23	.	.	-0.32	G$_1$
	.	.	.	.	.	.	.
	.	.	.	.	.	.	.
BugReport$_m$	0.41	-0.21	0.98	.	.	0.51	G$_m$

Figure 7.5. Labeling of Bug-Reports with group-id. Each group-id represent a group of source files which are changed together. It is an indirect approach to label each bug-fix report with multiple source files, and use single label classifiers for classification.

the bug report, or more closely matched as compared to other groups. In this way, we labeled each row of TDM transpose matrix (which represent the indexed term each bug report) with a group-id. There are chances that the assigned group-id may contains more source files as compared to the actual set of files, which are impacted by that bug report. To overcome this issue, after single-label classification one has to manually select the most appropriate set of impacted source files from the group of predicted source files.

Program 1 A program to assign a group id to each bug-fix (SCR) report.

```
FileName1[][];//contains the groups of source files which
              //are changed together, row=FileGroups and Column=FileNames.
BugReportFileName2[];//Files Name Impacted by a BugReport.
BugReportGroupId //Required Group-Id of FileGroups.
int counter1=0; int counter2=0;
for(int j=0;j<NumOfGroups;j++)
 {for(int k=0;k<NumOfFilesImpactedByABugReport;k++)
  {for(int l=0;l<NumOfFilesAssociatedToFileGroup;l++)
  {if(BugReportFileName2[k]==FileName1[j][l])
    {counter1++;}
  }
  }
 }if(counter1>counter2)
    {counter2=counter1;
  indexOfFileGroup=j; // Obtain File Group-Id
    }counter1=0;
} BugReportGroupId=indexOfFileGroup;
```

Finally, we used these single labeled reduced dimension TDM as a training and test data sets for the single label machine learning text classifiers. For single-label classification we used the machine

learning classifiers *Support Vector Machine (SMO)*, *J48*, and *Naive Bayes*.

7.4.3 Multi-Label Classification

In case of multi-label classification, we directly assign multiple impacted source files to each bug fixed report, and use a multi-label classification algorithm. In multi-label classification the data categories or classes may not be either mutually exclusive or conditionally independent. In this case each data may belong to multiple classes simultaneously [Sebastiani, 2002]. Let us assume a set of documents with a single label π. Then π belongs to a set of disjoint labels K and $|K| >1$. If $|K|=2$, then the learning is called binary classification, and if $|K| >2$ then the learning is called multi-class classification. For multi-label classification the data set are associated with a set of labels $T \subseteq K$ [Tsoumakas et al., 2009].

There are different methods to solve multi-label classification problem. These methods are divided into two different groups, i.e., *Problem Transformation* and *Algorithm adaptation* methods.

Problem Transformation Methods

The problem transformation methods are not dependent on any algorithm. These methods transform the multi-label classification task into one or more single-label classification, or regression, or label ranking task, and then apply any existing single label multi-class machine learning algorithm like SVM [Tsoumakas et al., 2009].

The most commonly problem transformation method is the *Label Power Set* (LP) method, a set of labels corresponds to all combinations of any number of the classes. If a document is relevant to several classes, the document must be assigned the label corresponding to the subset of all relevant classes. Once the new labels are defined, the multi-label multi-class problem has been reduced to a single-label multi-class problem, and the categorization task is reduced down to choose the single best label set for a document.

Formally, we express LP type of classification as H: X$\rightarrow$P(L), where P(L) is the power set of label L. To understand this method consider the case of multi-label classification of software change request (SCR), which may have multiple classes i.e., impacted source files. Let R represent a set of SCR and C represent the set of multiple classes i.e., R=$\{r_1, r_2, r_3, r_4, ... r_i....r_n\}$ and C=$\{c_1, c_2, c_3, c_4,....c_i.....c_m\}$. Where n is the number of SCR reports and m is the number of classes. In our case we have one or more impacted source files (F) related to each SCR. Therefore, we may have multiple classes like, F=$\{f_1, f_2, f_3, f_4,...f_i...f_k\}$, where k is the total number of distinct source files. Therefore the total number of distinct possible set of classes is k. If c $\subseteq$ k, then each SCR report is labeled with c as multi label multi-class. While LP consider each unique set of labels that exist in a multi-label data set as one of the classes of a new single-label classification task. Therefore in case of LP the total number of distinct possible set of classes is 2^k-1.

Algorithm Adaption method

In the Algorithm Adaption method the already available machine learning methods for single label classification are modified in order to make them available for multi-label classification. There are different algorithm adaptation methods like *ML-kNN*. *ML-kNN* is a method that uses *kNN lazy* learning algorithms for classification [Zhang and Zhou, 2005].

In our approach we use *LP-Method* of multi-label classification, which is belong to the *Problem Transformation Method* of multi-label classification. We choose *LP-Method* because it is simple and commonly used for multi-label classification. While, the computational complexity of LP depends on the complexity of the base classifier with respect to the number of classes, which is equal to the number of distinct label sets in the training data set [Tsoumakas et al., 2009].

7.5 Classification Evaluation Measures

7.5.1 Evaluation Measures for Single Label Classification

The commonly used evaluation measures for the classification problems are *Precision*, *Recall* and *F-measure*. In case of *single label classification Precision* $= a/b$ and *Recall* $= a/c$, where a is the number of documents retrieved that are relevant, b is the number of documents that are retrieved, c is the number of documents that are relevant. Precision and recall are not discussed separately. Both of them are discussed with reference to each another. Therefore, these measures are combined into a new measures called F-measure, which is actually the weighted harmonic mean of precision and recall. and given as, *F-Measure=* $(2\times$ *Precision* $\times$ *Recall*$)/($*Precision* $+$ *Recall*$)$.

7.5.2 Evaluation Measures for Multi Label Classification

The evaluation measures for *multi-label classification* is different from the evaluation measures for single-label classification. In order to define these measures consider a set of multi-labels document $|D|$. The document set $|D|$ is given as (x_i, Y_i), i=1...$|D|$, with $Y_i \subseteq$ L, where L is the set of multi-label classes. Now assume that H be a multi-label classifier and Z=H(x) be the set of labels predicted by H, then the measured Hamming loss is given by *HammingLoss (H,D)*$=\frac{1}{|D|} \sum_{i=1}^{|D|} |Y_i \Delta Z_i|/|Z|$. Δ stand for symmetric difference of two label sets corresponding to the XOR operation. The accuracy is *Accuracy(H,D)* $= \frac{1}{|D|} \sum_{i=1}^{|D|} |Y_i \bigcap Z_i|/|Y_i \bigcup Z_i|$. The measured precision is given by *Precision (H,D)* $= \frac{1}{|D|} \sum_{i=1}^{|D|} |Y_i \bigcap Z_i|/|Z_i|$, the recall is given by *Recall(H,D)* $= \frac{1}{|D|} \sum_{i=1}^{|D|} |Y_i \bigcap Z_i|/|Y_i|$. The measure F is given by *F-Measure(H, D)* $= \frac{1}{|D|} \sum_{i=1}^{|D|} |Y_i \bigcap Z_i|/(|Y_i| + |Z_i|)$ [Godbole and Sarawagi, 2004].

7.5.3 Labels Statistics

In case of multi-label data there are two parameters, which are commonly used to distinguish the different sets of multi-label data. These two parameters are label cardinality and label density. To

Table 7.2. Training data sets for single and multi label Machine Learning classification

Project-Name	Mozilla		Eclipse		GNOME	
Bug-Report Fixed Period	1999-2009		2001-2006		1997-2008	
Label-Type	Single-Label	Multi-Label	Single-Label	Multi-Label	Single-Label	Multi-Label
No. of Bug Reports	1168	1168	534	534	802	802
No. of key-terms	1939	1939	1208	1208	1440	1440
No. of reduced dim (LSI)	100	100	100	100	100	100
Distinct Impacted Files	27	27	38	38	32	32
Number of Labels	14-Groups	27	14-Groups	38	22-Groups	32
Labels Cardinality	-	1.68	-	1.21	-	1.18
Labels Density	-	0.0622	-	0.032	-	0.036

define these parameters consider a multi-label data instances i.e. (x_i,Y_i), i=1...$|D|$ where Y_i is the set of label associated with document i, and $|D|$ be the multi-label data set. Label cardinality of a data set D is the average number of labels of all instances in D. Formally, it is written as LC(D)=$\frac{1}{|D|} \sum_{i=1}^{|D|} |Y|$. In contrast label density is the average number of labels instances in document set D divide by total number of labels $|L|$, LD(D)=$\frac{1}{|D|} \sum_{i=1}^{|D|} |Y_i|/|L|$ [Tsoumakas et al., 2009].

7.6 Experimental Setup

In order to validate our approach we performed an experiment. The experimental setup follows the approach discussed in Section 7.1. For this purpose we first obtained the bug report data which have been fixed and resolved during 1999-2009 (Mozilla), 2001-2006 (Eclipse) and 1997-2008 (Gnome). The details of the obtained data sets are given in Table 7.2. The data selection criteria that has been used to obtain the data sets is given in the next paragraph.

Initially, we selected only those bug-reports whose link with CVS or Subversion have been found using the approach described in Section 7.4. This is the only way to map the fixed bug reports with its impacted source file names, and in order to train the classifier we should know the name of the impacted source files. Then, we selected only those bug reports, which have been involved in fixing at least one or at the most five source files at a time. However, there is a large number of bug reports, which are involved in fixing more than five source files at a given time. If we choose bug reports, which are involved in fixing more than five source files, then we will increase the number of distinct impacted source file names, which ultimately increases the label size. For machine learning classification, if the label size is large to obtain better classification results, the size of data instances should also be large. In our experiment it is not easy to increase the data size, because we are bound to select only those bug reports whose links found in CVS/Subversion. Furthermore, we selected only those bug-reports that have been involved in fixing source files, which are frequently changed. We used frequent pattern mining technique to obtain the list of frequently changed source files. We imposed this constraint condition to avoid the class imbalance issue of machine learning classification [Japkowicz, 2000].

After obtaining the bug reports data we transformed the textual data of the bug reports into term-to-document matrix (TDM) by using MATLAB tool TMG [Zeimpekis and Gallopoulos, 2005]. We store the result of TMG in Matlab as sparse matrix. The stored matrix is in the form of high

dimensional TDM. The size of the TDM dimension (i.e., number of bug reports and number of key-terms) related to different projects are depicted in Table 7.2. Afterwards we used the LSI technique to reduce the number of dimensions (see also Table 7.2).

After obtaining the TDM data, the next step is labeling and classification. First we discuss the experimental set up for single-label classification and then discuss the multi-label classification.

In case of single label classification, first we have to obtain different groups of source files which are changed together. For this purpose we used the techniques *Pattern Mining* and *Association Rules of Mining*. In our implementation we used the *Weka*[1] tool, and applied the *Apriori* algorithm on the obtained bug-fix transaction data of Mozilla, Eclipse and Gnome projects. The obtained patterns are used to cluster the source files into groups. To perform the grouping of source files we used the techniques, discussed in Section 7.4.3 and 7.4.3.

After extracting the data and performing labeling, we finally come to the single-label classification step. For classification, we used three different machine learning classifiers i.e., *Support Vector Machine (SMO)*, *J48*, and *Naive Bayes*. In our implementation we again used the *Weka* tools. The results for single-label classification are shown in Table 7.3.

In case of multi-label, we first labeled each bug report directly with one or more impacted source files, and then used the *Label Power set (LP)* method together with 3 different machine learning approaches *Support Vector Machine (SMO)*, *J48* and *Naive Bayes* as different base classifiers. The obtained classification results of *LP* methods are shown in Table 7.4. To perform the classification task we used *Mulan*[2] with *Weka*. It is an open source Java API for the multi-label classification.

To obtain more reliable values for the classification evaluation measures, we used the 10 fold cross validation technique. We now discuss the obtained results in more detail.

7.6.1 Results And Discussion

Table 7.2, data show that the number of source file per group is large in case of Eclipse project as compared to Gnome and Mozilla. The large number of source file per group indicate that in case of Eclipse project more files are changed together to fix a bug as compared to Gnome and Mozilla.

Table 7.3 depicts the results of single label classification. In case of Mozilla, the maximum precision and recall values are 49.3% and 46.4% respectively. We obtained these values using SMO for learning the classifier. We trained the classifiers with 1168 bug reports and used 14 distinct group of source files as labels. These 14 groups of source files are shared by 27 distinct source files, where each source file belongs to only one group. Similarly in case of Eclipse, the maximum precision and recall values are 45.5% and 44.8%. We obtained these values using support vector machine (SMO). We trained the classifiers with 534 bug report and used 14 distinct group of source files as labels. These 14 groups of source files are shared by 38 distinct source files. One reason to obtain the low classification

[1] http://www.cs.waikato.ac.nz/ml/weka/
[2] http://mlkd.csd.auth.gr/multilabel.html

Table 7.3. Single-label Classification Evaluation Using 3 Different ML-Algorithms

Project	**Metrics**	**SMO** (%)	**J48** (%)	**NB** (%)
Mozilla	Precision	49.3	30.4	36.7
	Recall	46.4	31.0	26.8
	Fmeasure	46.4	30.6	27.7
Eclipse	Precision	45.5	33.8	33.7
	Recall	44.8	34.1	27.0
	Fmeasure	42.7	33.8	27.3
Gnome	Precision	58.2	39.5	42.7
	Recall	51.1	39.4	36.4
	Fmeasure	50.7	39.2	36.4

evaluation compared to Mozilla is the small number of training data and large number of files group. Finally, in case of Gnome, the precision and recall values are 58.2% and 51.1%. We obtained these values again using SMO. We trained the classifier with 802 bug reports and used 22 distinct group of source files as labels. These 22 groups of source files are build with 32 distinct source files. Each source file belongs to only one group. In case of Gnome we obtained the maximum F-measure value i.e., 50.7%.

Table 7.4 presents the results obtained using multi-label classification. In case of Mozilla the maximum precision and recall values are 41.3% and 40.5% respectively. We obtained these values when LP classifiers used support vector machine (SMO). The evaluation results of other classifiers are below 33%. We found the similar evaluation results in case of Eclipse, i.e., the precision and recall values are 32.0% and 27.9%. We obtained these values when LP used support vector machine (SMO) as base classifier. In case of Gnome the evaluation results are better than for Mozilla and Eclipse. The maximum precision and recall values are 47.1% and 46.5% respectively. We obtained these values when the LP classifiers used SMO. In case of Gnome, we trained multi-label classifiers with 802 bug reports with 32 distinct source files as multi-label. Because we used a large data set, the classification results are better as compared to the classification results of Mozilla and Eclipse. The obtained value of F-measure for Mozilla, Eclipse and Gnome are 40.9%, 30.0% and 46.8% respectively.

The experimental results for multi-label classification show that the label cardinality, label density and the number of labels affected the multi-label classification results. Classification result is improved by increasing the label density and, decreasing the label cardinality and number of labels. In case of Gnome we obtained the best classification result as compared to other, because in case of Gnome, the label cardinality is the smallest i.e., 1.18 (see Table 7.2).

Figure 7.6 and 7.7 depict the trend of precision and recall for single and multi label machine learning classifiers trained with different classification algorithms. We observe from both figures that the precision and recall values are better when classifiers are trained using single label classifiers. The main reasons for this is the number of labels used for classification. From Figure 7.6 and 7.7, it may also observed that in case of single-label and multi-label classification, Support Vector Machine

Table 7.4. Multi-label Classification Evaluation Using *Label Power set (LP)* With 3 Different ML-Algorithms as Base Classifiers

Project	**Metrics**	**SMO** (%)	**J48** (%)	**NB** (%)
Mozilla	Precision	41.3	25.0	28.6
	Recall	40.5	24.5	27.7
	Fmeasure	40.9	24.7	28.1
Eclipse	Precision	32.0	20.2	28.1
	Recall	27.9	20.4	26.0
	Fmeasure	30.0	20.2	26.8
Gnome	Precision	47.1	31.8	34.9
	Recall	46.5	31.9	34.0
	Fmeasure	46.8	31.8	34.5

performed well. There may be two main reasons for the better performance of SMO as compared to Naive Bayes and J48. The first one is, SMO perform well in case of text classification with large feature space. Whereas, the second reason is, SMO can be believed to be less prone to the class imbalance problem as compared to other supervised machine learning classifiers. Because, SMO classification is based on small number of support vectors and thus SMO may not be sensitive to the number of data representing each class.

Figure 7.8 depicts the precision-recall plot of each label or class. The pattern of the graph shows that in case of Gnome precision and recall are linearly increasing or decreasing, whereas in case Mozilla and Eclipse most of the labels have high recall values at low precision and high precision values at low recall. It is also shown in the figure that in case of multi-label classification most of the precision and recall values are near the origin of the plot as compared to single label-classification. One possible reason is, multi-label training data sets contain more class imbalance as compare to single label training data sets [Japkowicz, 2000].

7.6.2 Threat to Validity

There are some threats to validity of the proposed approach that are related to the classification and the size of the training data set. In this section we discuss them briefly.

In case of single label classification, we label each bug report with a source file's group-id. Where each group-id represents the possible cluster of impacted source files that are frequently changed together. These groups of source files are obtained using association rule mining. Each group comprises multiple distinct source files. However, there are some cases in which source files belong to multiple groups at a time. These are very few cases in our data set. Therefore, we assigned overlapping files to any one of the groups. Similarly, there may be some cases in which the assigned group-id to a bug report may contains more source files names as compared to the name of

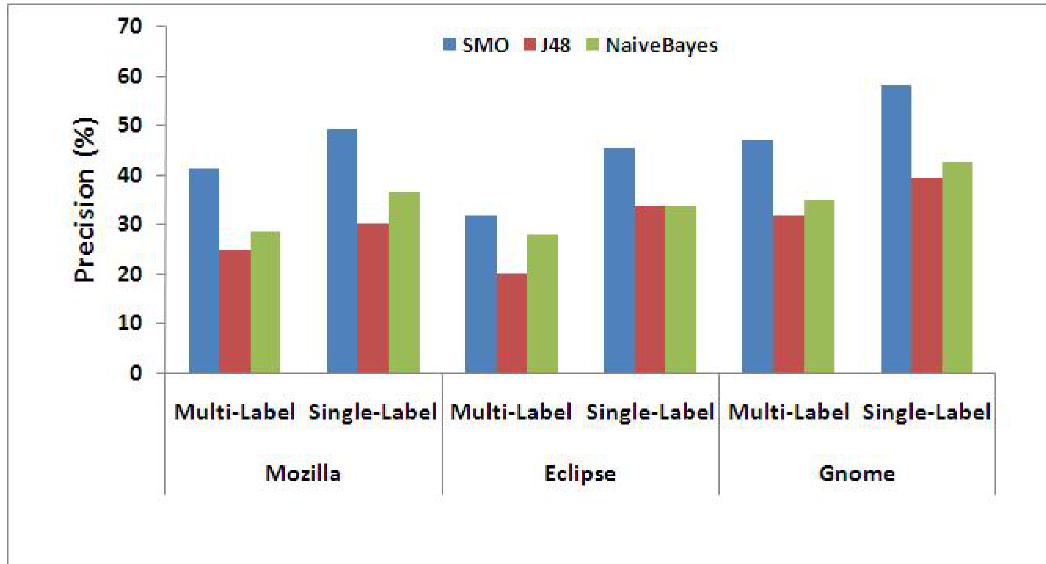


Figure 7.6. Precision values of single and multi-label classification when classifiers trained with three different machine learning algorithms i.e., support vector machine, J48 and Naive Bayes (NB)

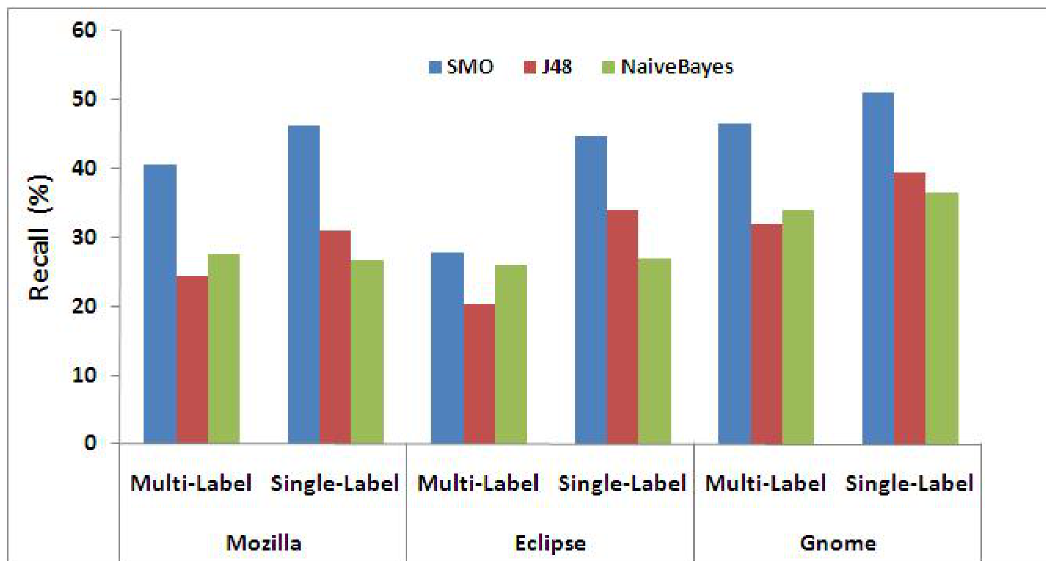


Figure 7.7. Recall values of single and multi-label classification when classifiers trained with three different machine learning algorithms i.e., support vector machine, J48 and Naive Bayes (NB)

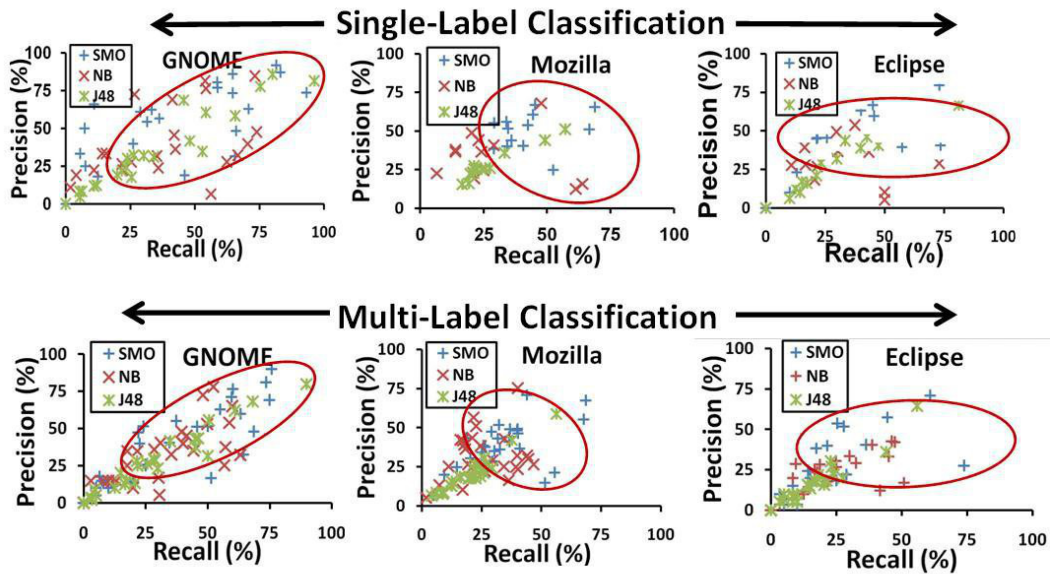


Figure 7.8. Precision-Recall plot of individual labels.

actual source files that have been impacted by that bug report. Therefore, in those cases after classification manual selection of source files from the predicted group of source files plays an important role.

In order to obtain the change-coupling data we performed frequent pattern mining and obtained groups of files which are changed together. These group of files are time dependent and there might be some cases in which source files are removed or deleted from the software repository. In these cases our system may include deleted source files in any of the file groups. Similarly, our system is not able to include newly added project's source files into any existing group of files. The newly added project's source files have small change-coupling history but we considered only those source files to be included in a group when their change-coupling frequency value is larger than a predefined value. The described cases are very few and have, therefore, very little impact on the performance of the classifiers.

In case of multi-label classification the size of the labels plays an important role. It is difficult to identify an optimum size of labels. Large label size reduce the performance of multi-label classification. In our experiment source files are labels, and in case of large OSS the number of source files are in thousands. This value varies from project to project. However, during labeling we consider only those source files, which are frequently changed. Therefore, we obtained a lower number of labels.

7.7 Summary

In this chapter we presented and compared two different approaches for the automation of SCR impact analysis. We evaluated these methods using three data sets from the three open source software projects repositories i.e., Mozilla, Eclipse and Gnome. Our experimental results indicate that the textual data of past SCR, which are labeled with the set of impacted source files, can be

used to build the supervised machine learning based models for the automation of SCR impact analysis.

In summary our approach comprises the following steps: Before building the models, we first transform the textual data of SCRs or bug reports into a low dimension indexed term-to-document matrix (TDM). In order to transform the SCRs data into TDM, we use indexing and dimension reduction techniques from information retrieval i.e., TF×IDF and Latent Semantic Indexing (LSI). After the transformation of SCR data into TDM, we label them and use single and multi-label classifiers to build the models.

From the empirical analysis we found out that the single label classifiers have large values for the classification measures compared with the multi-label classifiers. But the multi-label classifiers are more reliable compared to the single label classifiers. This is due to the fact that in case of single label, we label each SCR with a single group of source file, which are frequently changed together. We obtain these groups of source file using an indirect method, i.e., frequent pattern mining and association rules. Whereas in the case of multi-label we labeled each SCR directly with the set of actual impacted source files.

In case of Gnome project, the obtained values of precision and recall for the single-label classification are 58.2% and 51.1%. Whereas the precision and recall values of multi-label classification are 47.1% and 46.5%. We observed similar trends in case of Mozilla and Eclipse, but their evaluation measures values are less. Another result obtained is that in case of single and multi labe classification, learning based on Support Vector Machin (SMO) performed well.

8 Automatic Bug Triaging System Based on Multi-Label Classification of SCRs

Parts of the contents of this chapter have been published in [Ahsan et al., 2009a] and [Ahsan et al., 2009c]

In software engineering a *Bug Triage System* (BTS) is commonly used at the time when new SCRs are reported. BTS is used to assign a new SCR or bug report to an appropriate developer, who is capable to resolve the issue.

In chapter 7, we described our approach to extract the data of software change requests (SCRs) from the project's repository of GNOME, Mozilla and Eclipse. We used the extracted data of SCRs to perform an experiment for the impact analysis of SCRs. Now, in this chapter we use the SCR data of Mozilla project and describe our approach to construct a bug triage system, which is based on multi label classification of SCRs. The content of this chapter has been published in the paper [Ahsan et al., 2009a] and [Ahsan et al., 2009c].

Recent Research Trends: The recent trends in the software engineering are, researchers use the resolved bug reports data of OSS and labels them with the name of a developer who actually resolved them. After labeling supervised machine learning classifiers are used to train. Finally, at the time when a new bug report is submitted, the trained classifiers are used to predict the name of a developer who is appropriate to resolve the new bug report. This process is commonly known as *Automatic Bug Triaging* [Cubranic, 2004, Anvik et al., 2006]. In another approach, resolved bug reports are labeled with the time which have been spent to fix them and used this data to train machine learning classifiers. Finally, at the time, when new bug is reported the trained classifier is used to predict the approximate time required to fix the new bug report [Weiss et al., 2007]. Similarly, Canfora et al [Canfora and Cerulo, 2005] in 2008, presented their approach for the impact analysis of SCRs. They used the textual similarity technique of informational retrieval to retrieve past SCRs similar to a new SCR and to compute the rank list of impacted files.

Motivation and Findings: A major part of software costs is spent for software maintenance. The software maintenance cost may be reduced when completing maintenance tasks in time using fewer resources. To improve the corrective maintenance task most large and OSS software systems use a bug tracking system and perform the assignment of bug reports to the maintenance personal

manually. When a new bug is reported, the first measure is to validate it as a unique bug report and to assign it to the most appropriate developer. However, a manual validation and allocation of hundreds of bug reports per day is a challenging task. It needs lots of skills and consumes lots of time. Moreover, manual bug triage is error-prone, tedious and time consuming, especially in cases where hundreds of bugs are reported daily. Within our study we found that, in case of Mozilla, the average number of reported bugs per day is 97.5 (see Figure 7.1). From these observations we can conclude that in case of large and open source software system large number of bugs are reported per day, which ultimately increases the software maintenance time and cost. Therefore, an automatic and accurate bug triage system may play an important role to reduce the time and cost of software corrective maintenance task.

Our Goals and Approach: The goal of our research work is to utilize the advance techniques of information retrieval and machine learning classification to construct an automatic bug triaging system, which is not only capable of assign SCRs to the appropriate developers, but also identify the name of the suspected source files, which are required to be modified, and estimate time, which is required to fix the SCR. The underlying assumption is that the available SCRs together with the information regarding the developer resolving the SCR, set of modified source files and the time spent to fix the SCR would let us extract the mapping, which is close to the optimum. This assumption is to a great extent reasonable because, when manually assigning SCRs to developers someone would always like to assign the right task to the right people. In this case study, we make use of information retrieval and multi-label machine learning methods for indexing and classification of the textual information given in a SCR into a set of categories. In our experiments, the text files are the SCRs (or bug reports) and the categories into which bug reports are classified are the name of the appropriate developers who can fix the bug reports.

Research Contribution: Following are the main contributions,

- We performed two different experiments to contribute the objective of providing an automated bug triage system. The first experiment is based on the single label classification of SCRs which is capable to assign SCRs to the appropriate developers. Whereas, the second experiment is based on multi label classification of SCRs, which is not only capable to assign SCRs to the appropriate developers, but also identify the name of the suspected source files, which are required to be modified, and estimate time, which is required to fix the SCR.
- We compared the two different approaches and found that the multi label classification of SCRs is more comprehensive and performed well and can be used for the bug triage system.

Chapter Layout: The rest of the chapter is organized as. In Section 8.1 we describe two different approaches for a bug triage system. In Section 8.2 we describe in details the single label classification approach for a BTS system and discuss the experimental results. In Section 8.3 we describe the multi-label classification approach to construct a bug triaging system. Threats to validity are described in Section 8.4. Finally, in Section we summarize the chapter.

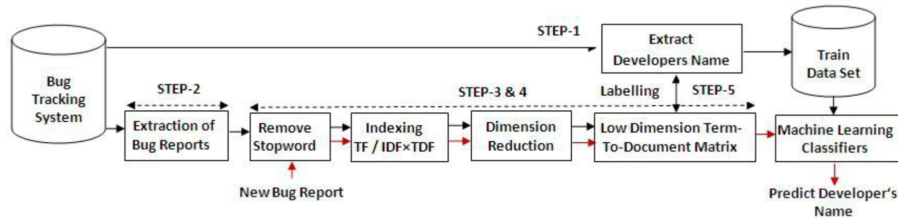


Figure 8.1. Process flow diagram of the proposed *Automatic Bug Triage System (BTS)* based on single label machine learning classification.

8.1 Different Approaches For an Automatic Bug Triage System

In this chapter, we present two different approaches to obtain an automatic bug triage system and discuss the results of two different experiments, which are based on our approaches. In the first experiment, we downloaded resolved bug reports along with the developer's activity data from the Mozilla open source project. We extracted the relevant features like the report title, report summary, etc., from each bug report, and extracted developer's name who resolved the bug reports from the developers activity data. We processed the extracted textual data, and obtained the term-to-document matrix (TDM) using an approach, which we have described in Chapter 7. For term weighting methods, we used simple term frequency and TF $\times$ IDF (term frequency inverse document frequency) methods. Furthermore, we reduced the dimensionality of the obtained TDM by applying feature selection and latent semantic indexing (LSI) methods. Finally, we used seven different machine learning methods for the classification of bug reports. The best obtained BTS is based on LSI and support vector machine having 44.4% classification accuracy. The average precision and recall values are 30% and 28%, respectively.

In the second experiment, we used the same TDM data of Mozilla project, which we have obtained in the first experiment. Now, in this experiment we labeled each TDM (SCR data transformed into key terms) with multiple labels, i.e., the developer name, the list of source files, and the time spent to resolve the SCR. To obtain effort value and the list of impacted source files, we used our own approaches, which we have described in Chapter 3 and 7. After obtaining the labeled data of SCRs, we perform classification. For classification, we used two different approaches of multi-label machine learning methods, i.e., *Problem Transformation Methods* and *Algorithm Adaption Methods*. Our experimental result shows that the best multi-label classifier for SCR is ML-kNN (*Algorithm Adaptation Methods*). The obtained precision and recall values are 71.3% and 40.1% respectively.

8.2 Bug Triage System Based on Single Label Classification of SCR

Figure 8.1 shows the process flow diagram of our proposed automatic bug triage system based on single label classification approach. In the following sub sections, we describe our approach in detail to develop an automatic bug triage system.

8.2.1 Data Preparation

To perform the experiment, we downloaded 1,983 bug reports along with the developer's activity data from the available repositories of Mozilla project. These bug reports have been fixed between 2005-06-01 and 2006-05-31. We considered only those bug reports which have been fixed or resolved. Whereas we left the other type of bug reports such as duplicate bug reports. Followings are the steps, which have used to prepare different data sets. We used these data sets to train and test the ML classifiers.

Step-1: After downloading the data, we extracted developer names from developer's activity data who have resolved the bug reports. To obtain developer names, we processed the developer's activity data, and extracted the name of those developers who changed the status of bug reports to "RESOLVED". We considered these developers as those who resolved bug reports [Ahsan et al., 2009b].

Step-2: Initially, we had 1,983 bug reports. We found that these bug reports had been fixed by 186 distinct developers. This number of developers is very large, while most of them fixed very few bugs. In our experiment, each developer represents a class. The better performance of the machine learning classifiers can be achieved, if the number of classes is small and each class should have a sufficient number of instances. Therefore, we further processed the bug reports data and finally selected 792 bug reports. These are the bug reports which have been fixed by 18 distinct developers. While each of these selected developers have resolved at least 30 bug reports.

Step-3: In this step, we processed each bug report by extracting the data from it. We extracted report title, description, product, component and the operating system names, and merged them into a text file. This text file was further processed to obtain the term-to-document matrix (TDM) by using MATLAB TMG tool [Zeimpekis and Gallopoulos, 2005]. To execute TMG. Some inputs are required, like a list of stop words, stemming option, name of term indexing method, and global frequency of term occurrence. For stop words we used a SMART stop word list. Stop words are the set of common words such as *the, is, an* etc. These words are not important for indexing and should be removed. We left the option of stemming because the previous research [5] shows that it has less impact. For terms weighting, we used inverse document frequency (TF$\times$IDF), and for dimension reduction, we set the global term frequency value $\geq$3 (global term frequency is a feature selection method). It means that each term in the TDM has appeared in at least three bug reports. After executing the TMG tool, we obtained a term-to-document matrix (TDM). This TDM matrix data is stored as a sparse matrix in MATLAB. The results obtained using TMG tool is shown in Table 8.1.

Step-4: We repeat step-3 with different options, i.e., for term indexing, we used simple term frequency (TF) and for dimension reduction, we set global term frequency value $\geq$2. The obtained TDM contains 2,769 keywords or terms, where each term is a dimension. To further reduce the dimensionality, we used LSI. To perform this task, we applied the techniques of singular value decomposition on the obtained TDM matrix data, and obtained 792 singular values, which are also called factors. It means that the 2,769 terms or dimensions now transformed into only 792 significant dimensions. The higher singular values are more significant. Therefore, to obtain the multiple low dimensional TDM matrices, we transformed the original TDM matrix into 4 different low dimensional

Table 8.1. Data related to the bug report's index terms

Number of bug reports	792
Total number of distinct terms found in 792 bug reports	1836
Average number of indexing terms per documents	57.19
Sparsity	1.57%
Removed stop words	566
Removed terms using the term-length thresholds	226
Removed terms using the global thresholds	6261

Table 8.2. Different data sets on the basis of features size (dimensions).

Data Set	Dimension Reduction Technique	Dimen. /Factor	Global words	Weight Func.
A	Feature Selec.(terms frequency)	1836	≥3	TF×IDF
B	Laten semantic indexing (LSI)	50	≥2	TF
C	Laten semantic indexing (LSI)	100	≥2	TF
D	Laten semantic indexing (LSI)	300	≥2	TF
E	Laten semantic indexing (LSI)	500	≥2	TF

TDM matrices by considering the top 50, 100, 300 and 500 singular values.

Step-5: After converting the bug report's data into the reduced dimensional TDM, we took the transpose of each TDM matrix because each column of TDM matrix represents a unique bug report. After converting column into rows, we labeled rows of each TDM matrix with the developer name. These labeled data sets have used to train the ML classifiers.

8.2.2 Experimental Setup and Results

To perform the bug classification experiment, we used five different data sets, which we have obtained in the previous section. The description of these data sets are shown in Table 8.2. To perform the machine learning based classification, we used WEKA tool. It is a java based open source tool [Ian H. Witten and Kaufmann, 2005], and to obtain the better classification results, we used 10-fold cross validation techniques. To perform the comparative study for the classification of bug reports, we used seven different ML algorithms, i.e., Decision Table (DT), NaiveBayes (NB), RBF Network (RB), Random Forest (RF), REPTree (RT), Support Vector Machine (SVM) and Tree-J48 (J48). In the following section, we discuss the obtained results.

The classification results of our proposed bug triage system are shown in Table 8.3 and 8.4. Table

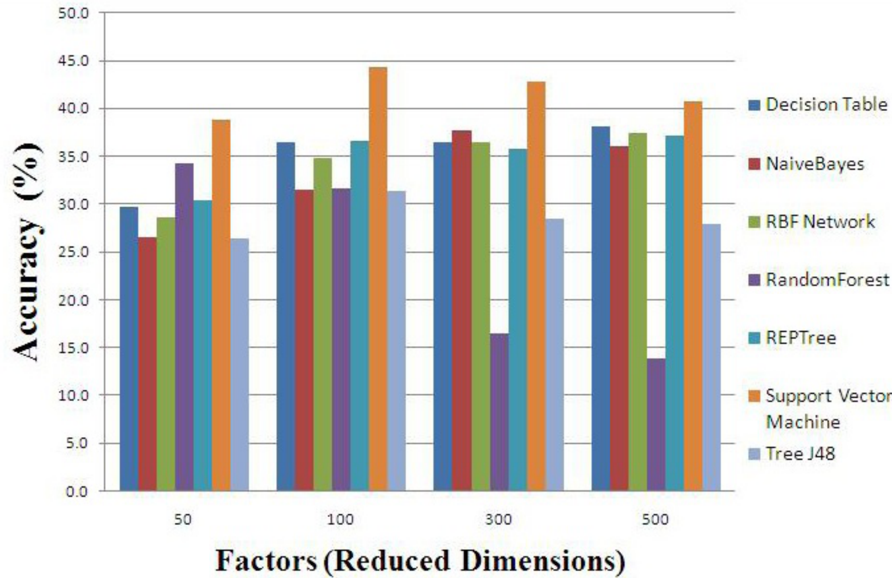


Figure 8.2. Classification Accuracy of 7 different classifiers, train and test with 4 different dimension size data sets (dimension reduced by LSI).

8.3 shows the classification results of seven different ML classifiers, when they are trained and tested with four different data sets, i.e., B, C, D and E. According to Table 8.3, the maximum value of classification accuracy is 44.4%, which is obtained when data set C is trained with SVM. Similarly, the maximum precision, recall and F-measure values are, 0.37, 0.348 and 0.349 respectively, which are obtained when SVM is used to classify low dimension data sets. The maximum obtained kappa-statistic value is 0.387, which is obtained when SVM is used to classify the data set C (A perfect classification would produce a Kappa value of one). Figure 8.2, 8.3, 8.4 and 8.5 shows patterns of different evaluation measures. According to these figures, the value of different evaluation measures decreases when dimension size is either very large such as 500 or very small such as 50. While the evaluation measures are maximum, when trained and tested with a data set of 100 dimensions. Therefore, we can say that in order to obtain the best classification results using LSI the dimension size of data set should be optimum.

Table 8.4 shows the classification results when seven different classifiers trained and tested with a data set A. According to Table 8.4, the RepTree classifier has the highest value for the accuracy, i.e., 42.98%. While, in case of SVM the obtained classification accuracy is 41.34%, but SVM has the highest values for the average precision and recall, i.e., 0.38 and 0.34 respectively. The maximum obtained kapa statistic values are, 0.363 and 0.351, which are obtained when data set A classify with RT and SVM respectively. Figure 8.6 shows the classification accuracy of seven different classifiers. According to this figure RBF-Network and Tree J48 has the lowest value of classification accuracy, i.e., 23.7% and 33.7% respectively. Whereas, the accuracy values lies between 38.5% to 42.9%, when DT, NB, RF, RT and SVM are used for classification. Figure 8.7 shows the evaluation measures, i.e., precision, recall and F-measure of seven different classifiers, when they trained with data set A. According to these figures Random Forest, RepTree, Decision Tree and J48 have lowest values for

Table 8.3. Evaluation results of bug report's classification (When train & test with LSI based low dimensional data)

ML	Data Set	Factors / Dimensions	Accuracy %	kappa	Avg. Precision	Avg. Recall	Avg. F-Measure
DT	B	50	29.7	0.213	0.193	0.228	0.197
DT	C	100	36.4	0.283	0.202	0.196	0.179
DT	D	300	36.4	0.280	0.226	0.229	0.206
DT	E	500	38.2	0.301	0.361	0.303	0.294
NB	B	50	26.5	0.218	0.312	0.272	0.277
NB	C	100	31.5	0.273	0.244	0.245	0.227
NB	D	300	37.7	0.330	0.319	0.325	0.289
NB	E	500	36.0	0.319	0.312	0.328	0.296
RB	B	50	28.6	0.223	0.235	0.303	0.239
RB	C	100	34.8	0.291	0.255	0.276	0.229
RB	D	300	36.4	0.280	0.312	0.272	0.265
RB	E	500	37.4	0.315	0.316	0.236	0.216
RF	B	50	34.3	0.276	0.112	0.100	0.096
RF	C	100	31.6	0.242	0.303	0.268	0.271
RF	D	300	16.4	0.074	0.170	0.166	0.157
RF	E	500	13.8	0.044	0.215	0.166	0.165
RT	B	50	30.3	0.231	0.244	0.268	0.245
RT	C	100	36.5	0.290	0.180	0.233	0.191
RT	D	300	35.8	0.298	0.285	0.243	0.221
RT	E	500	37.2	0.294	0.266	0.259	0.243
SVM	B	50	38.8	0.319	0.372	0.321	0.331
SVM	C	100	44.4	0.387	0.30	0.282	0.267
SVM	D	300	42.9	0.373	0.361	0.342	0.343
SVM	E	500	40.7	0.350	0.361	0.348	0.349
J48	B	50	26.4	0.202	0.222	0.218	0.219
J48	C	100	31.4	0.256	0.236	0.226	0.229
J48	D	300	28.4	0.223	0.282	0.280	0.278
J48	E	500	27.9	0.217	0.244	0.244	0.243

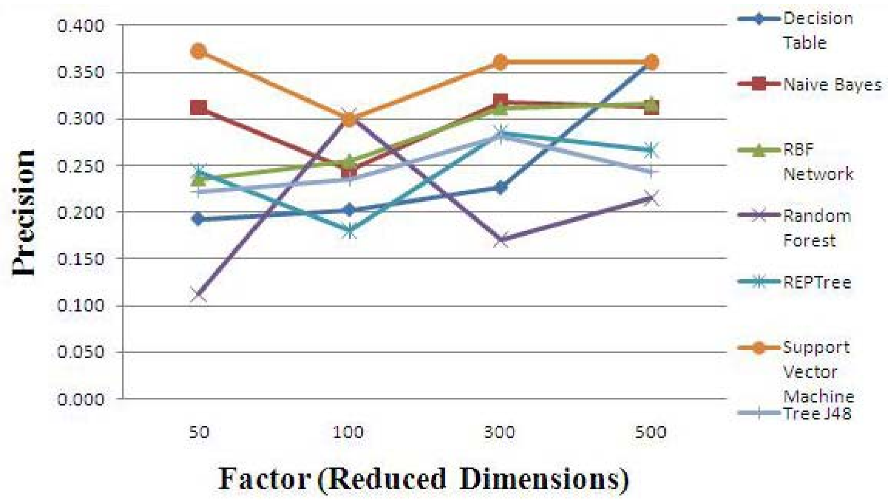


Figure 8.3. Classification Precision of 7 different classifiers, train and test with 4 different dimension size data sets (dimensions are reduced by LSI).

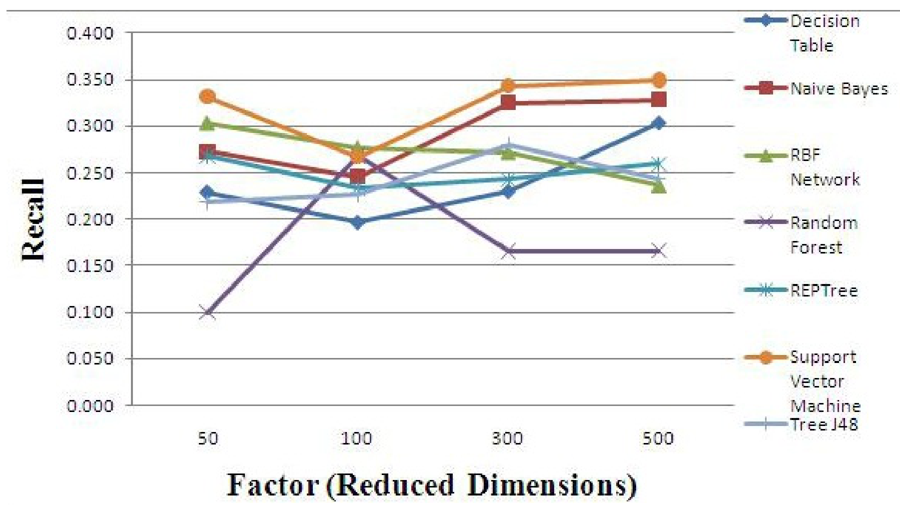


Figure 8.4. Classification Recall of 7 different classifiers, train and test with 4 different dimension size data sets (dimensions are reduced by LSI).

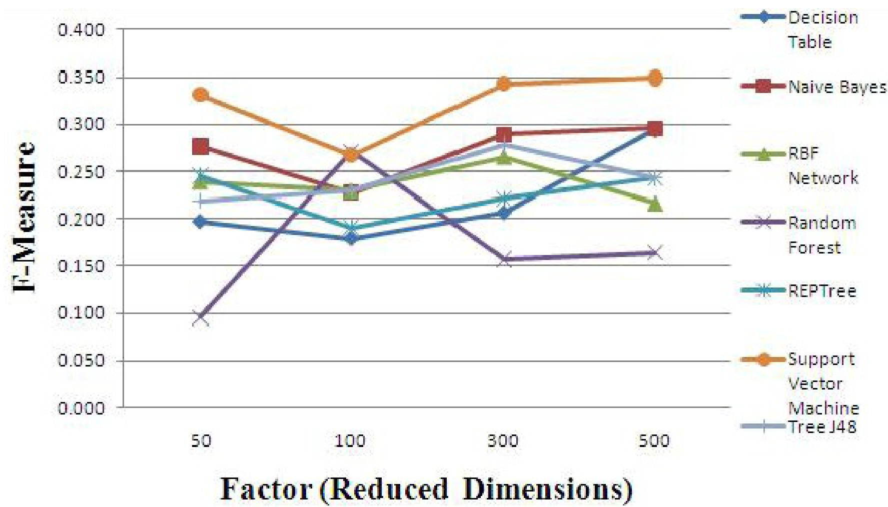


Figure 8.5. F-Measure of 7 different classifiers, train and test with 4 different dimension size data sets (dimensions are reduced by LSI).

Table 8.4. Evaluation results of bug report's classification (When train & test with data set obtained without using LSI)

ML	Data Set	Accuracy %	kappa	Avg. Precision	Avg. Recall	Avg. F-Measure
DT	A	40.2023	0.324	0.260	0.24	0.217
NB	A	39.1909	0.338	0.316	0.36	0.313
RBF	A	23.7674	0.168	0.371	0.33	0.325
RF	A	38.5588	0.319	0.126	0.09	0.092
RT	A	42.9836	0.363	0.288	0.25	0.232
SVM	A	41.3401	0.351	0.380	0.34	0.354
J48	A	33.7547	0.272	0.228	0.23	0.229

precision, recall and F-measures, i.e., between the range of 0.09-0.26. While SVM, Naive Bayes and RBF-Network have maximum values, i.e., between the range of 0.30 and 0.38.

8.3 Bug Triage System Based on the Multi-label Machine Learning Classification

Multi-label classification is a rapidly developing field of machine learning. There are large number of research areas such as protein function classification, semantic scene classification, among others all of them requiring multi-label classification [Tsoumakas et al., 2009]. We used the two different methods of multi-label classification, i.e., *Problem Transformation* and *Algorithm adaptation* methods. The detail of multi labels classification and its different methods are given in Chapter 7. In the following paragraph we describe that how we used *Label Power Set (LP)* to label SCRs data for the multi label

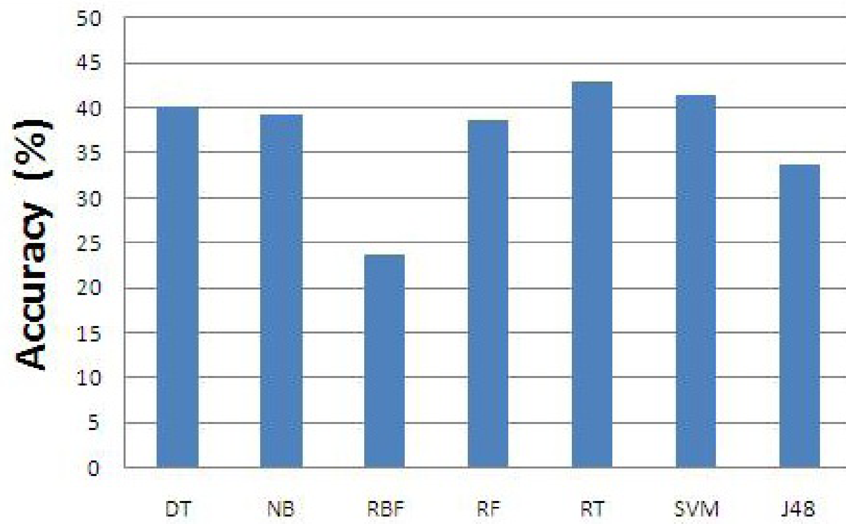


Figure 8.6. Classification accuracy of 7 different classifiers, train and test with a data set whose dimensions are reduced by feature selection method.

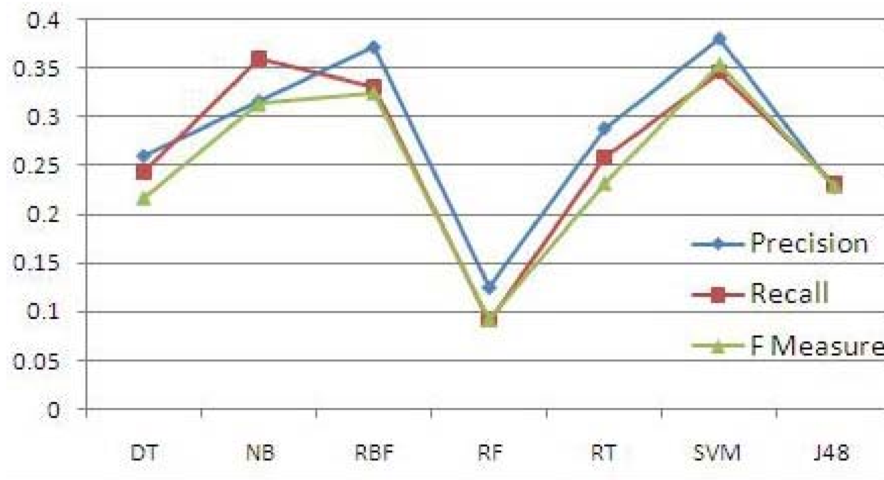


Figure 8.7. Precision, Recall & F-measure of 7 diff. classifiers, train & test with a data set whose dimensions are reduced by feature selection method.

classification.

The commonly used problem transformation method is *Label Power Set (LP)*. It considers each unique set of labels that exists in multi-label training set as one of the classes of a new single label classification task [Tsoumakas et al., 2009]. To understand LP method work in our experiment, consider an example of multi-label classification of CR, which have three different label categories to assign each software change request. Let R represent a set of SCR, and C represent the set of multiple classes of multi-label.

$$R=\{r_1,\ r_2,\ r_3,\ r_4,....\ r_i....r_n\}$$
$$C=\{c_1,\ c_2,\ c_3,\ c_4,....c_i.....c_m\}$$

Where n is the number of SCR reports and m is the number of classes. In our case we have three different label categories i.e., developers (D), files name (F) and effort (E). Each label category may have multiple classes like.

$$D=\{d_1,\ d_2,\ d_3,\ d_4,....d_i...d_j\}$$
$$F=\{f_1,\ f_2,\ f_3,\ f_4,...f_i...f_k\}$$
$$E=\{e_1,\ e_2,\ e_3,\ e_4,...e_i...e_n\}$$

Where j is the total number of distinct developers, k is the total number of distinct source files and n is the total number of distinct effort values. Therefore the total number of distinct multi-label class sets,

$$C_m = D_j \times F_k \times E_n$$

If c $\subseteq$ C, then each SCR is labeled with c. In our case we assigned a single developer, single effort value and one or more source files to each SCR. Therefore the possible examples of multi-label multi-class sets are given by:

$$C=\{\{d_1,f_1,e_1\},\{d_1,f_1,f_2,\ e_2\},\ ...,\{d_j,f_{1...k},\ e_l\}\}$$

Table 8.5 shows that, how multi-label multi-class is converted into single-label multi-class classes, and associated with different SCR.

We, also used the Algorithm Adaption method of multi-label classification, i.e., *ML-kNN*. *ML-kNN* is a method that uses *kNN lazy* learning algorithms for classification [Zhang and Zhou, 2005]. In order to evaluate the classifiers, we used the multi-label classification evaluation measures. For the formal description of the multi label classification evaluation measures, see Chapter 7.

8.3.1 Data Collection

We obtained data from the Mozilla project repository. These data are in the XML, html and text format. Therefore, we parsed all the downloaded data files to extract the relevant information. After

Table 8.5. SCR assigned with multiple labels

SCR(R)	Deve.(D) $\{d_1,..d_j\}$	Files(F) $\{f_1,..f_k\}$	Effort(E) $\{e_1,..e_l\}$	Multi-Label D×F×E
r1	$\{d_1\}$	$\{f_1,f_2\}$	$\{e_2\}$	$\{d_1,f_1,f_2,e_2\}$
r2	$\{d_3\}$	$\{f_2\}$	$\{e_1\}$	$\{d_3,f_2,e_1\}$
.	.	.	.	.
.	.	.	.	.
r_n	$\{d_j\}$	$\{f_k\}$	$\{e_l\}$	$\{d_j,f_k,e_l\}$

obtaining the bug reports data, we performed two main tasks, i.e., converting the bug report's data into the term-to-document matrix and performed labeling.

Conversion of Bug Reports into TDM

To convert the bug reports into TDM, we used the approach which we have described in Chapter 7. To perform the comparative experiment, we used different combination of the term's weighting function and dimension reduction techniques, and obtained two different groups of data sets. In the first group, we obtained data sets using TF×IDF as term weighting function, and different values for the *Document Frequency Thresholding* as dimension reduction, i.e., we selected those terms that have occurred at least 3, 6 and 9 times. In the second group, we used TF×IDF as weighting function and used *Latent Semantic Indexing (LSI)* for dimension reduction. We used different factor i.e., 100, 150, and 200 as dimension.

Data Labeling

In case of multi-label classification, if the number of training instances per label is significantly small compared to the total number of instances, then it may create a problem similar to class imbalance [Godbole and Sarawagi, 2004]. Therefore, to avoid the class imbalance issue, we selected only those classes for labels, which have some predefined minimum number of instances. Furthermore, to study the impact of label size, label density and label cardinality on multi-label classification. We used three different data set of different label size. These data sets are obtained by different combinations of the classes that belong to each label's category. We used three categories of labels, i.e., developer, source file and effort. While each category has multiple classes.

There are two approaches to label the bug reports with the developers name. One of the approaches is to assign a group of developers who have resolved the similar bug reports. This is basically a semi-automated approach. In which, after classification one may obtain a set of appropriate developers name, and one has to select a single most appropriate developer from the obtained set of developers (see [Anvik et al., 2006]). In the second approach, each bug report is labeled with the single developer name who actually resolved it [Cubranic, 2004]. We used the later approach and labeled each bug

report with single developer.

Now the issue is to identify the name of a developer who actually resolved the bug. To accomplish this task we considered an approach given by D. Cubranic [Cubranic, 2004]. According to this approach the developer who handles a bug report when the bug status is FIXED, is the actual developer who resolved that bug. However, there are some other scenarios, which are possible. For example, when bugs are resolved as DUPLICATE or as INVALID, then the developers who resolved them as DUPLICATE or INVALID are the actual developers who fixed those bugs. Since we considered only those bug reports data, which have been resolved as FIXED, therefore we have not considered other scenarios, i.e., DUPLICATE or INVALID bug reports.

To label the bug reports with the name of impacted source files, we processed the CVS-log data. As a result we obtained the name of the source files that have been used to fix a bug. We first established a link between the bug reports, and the CVS log comments of the corresponding source file revisions. To obtain the link between bug reports (stored into Bugzilla) and the CVS-log comments (stored into CVS repository), we used an approach which we have described in Chapter 7.

Finally, to label the resolved bug reports with their effort value, we have to extract the effort data. We consider the time that was spent between the ASSIGNED and the RESOLVED periods of a bug life as the estimated bug fix effort (in days). To obtain the effort data, we used an approach which we have described in chapter 3.

Because of the above mentioned assumptions, we found that in our data set the total number of labels are 2,567 i.e., 503 distinct developers, 907 distinct source files, and 1,157 distinct effort values. These numbers of labels are very large, while most of the developers have fixed very few bugs, similarly, most of the files have changed one or two times. Therefore, we further processed the data set and because of label, we obtained the three different groups of data. These groups of data are obtained by selecting only those developers that have fixed some specific number of bugs, and selected those source files which have changed at least for some specific number of times. For an effort, we used the statistical frequency distribution. We divided the effort data into 3 class intervals, and considered the mean value of each class interval as a class value of all those effort values that lie within that class interval. In our experiment, the standard deviation of each class was found small. Therefore, this reduction has less impact on the prediction. The predicted effort class values are almost equal to the predicted-effort ($\pm$ Std.Dev). The frequency distribution of the effort data is shown in Table 8.6. The details of the three different groups of data on the basis of labels are shown in Table 8.7.

8.3.2 Experimental Setup and Results

In the last section we performed indexing and labeling of SCRs. Now, after indexing and labeling, we obtained multiple data set for classification. The complete experiment setup is shown in Figure 8.8.

As discussed in Chapter 7 there are two methods for multi-label classification. First, we applied problem transformation methods, i.e., *Label Power set (LP)* methods and used *Support Vector Machine (SMO)*, *J48* and *Naive Bayes* as base classes. Then, we used the algorithm adaptation method,

Table 8.6. Frequency distribution of effort data (bug fix time) to obtain distinct classes of effort

Label based data set	Class Interval (Effort data in days)	Frequency	Mean	Std. Dev
44	0-1	369	1.0	0.0
	2-7	144	3.4	2.5
	8-30	35	17.9	5.5
59	0-1	449	1.0	0.0
	2-7	222	3.6	2.6
	8-30	48	16.9	5.3
91	0-1	522	1.0	0.0
	2-7	289	3.5	2.5
	8-30	72	17.8	5.7

Table 8.7. Size of labels and the label selection criteria for three different data sets

Data Set (label based)	44	59	91
Size of Data Instance	548	719	808
Count of Distinct Developer Labels	16	26	34
Developer Label Selection Criteria	Fix at least 60 bugs	Fix at least 13 bugs	Fix at least 10 bugs
Count of Distinct File Labels	25	30	54
File Label Selection Criteria	Changed at least 13 times	Changed at least 30 times	Changed at least 10 times
Count of Distinct Effort Labels	3	3	3
Total Number of Labels	44	59	91
Label Cardinality	3	3	3
Label Density	0.068	0.051	0.032

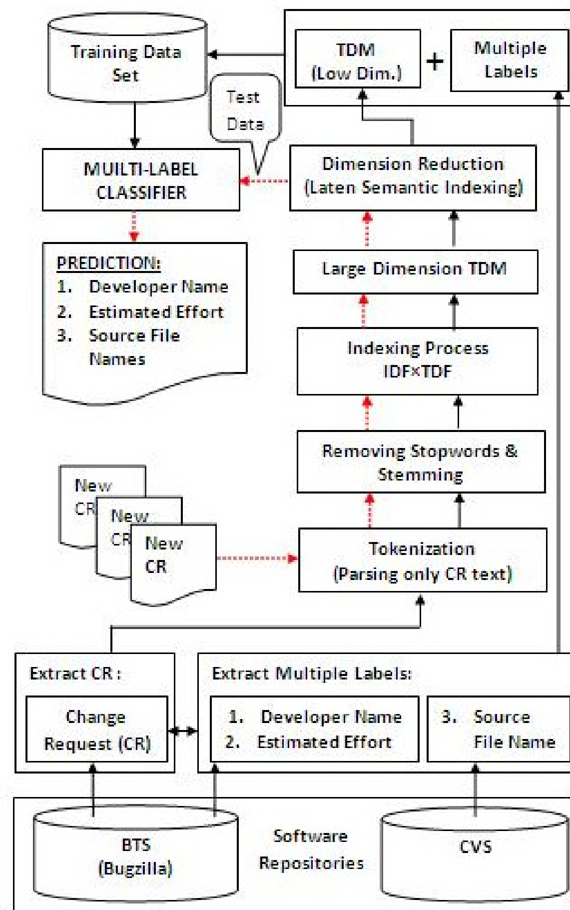


Figure 8.8. Multi-Label classification of software CR

i.e., *ML-kNN*. The obtained classification results of *LP* methods are shown in Table 8.8, while the classification results of *ML-kNN* is shown in Table 8.9. To perform the classification task we used *Mulan*. It is an open source java API for multi-label classification (http://mlkd.csd.auth.gr/multilabel.html). To obtain better classification results, we used 10 fold cross validation, and to evaluate the classification results, we used five different evaluation measures.

According to the obtained results of *Label Power Set* (LP) method as shown in Table 8.8, the classification accuracy is 45.2% with 54% precision, 54% recall and 3.0% HammingLoss. These results are obtained when LP classifier is trained and tested using the data set that has labeled size 91 and 200 dimensions, while a support vector machine is used as a base classifier. According to the Table 8.8 data, the obtained results of classification measures are better when LSI based low dimension data sets are used to train and test the LP classifier. However, the difference in the classifications results are not significant as compared to the classification results which are obtained using the data set whose dimensions are reduced by using term-global-frequency. Furthermore Table 8.8 depicts the significant impact of label size on the classification results. We obtained the best classification results when classifiers trained with the data set that has 91 labels. For LP based multi-label classification, we used SMO, J48 and Naive Bayes as base classifiers. One can use any other multi-class machine learning classifier as base classifier for LP method. However, we selected SMO (support vector machine), J48 and Naive Bayes because these machine learning classifiers are supposed to be the best for multi-class text classification purpose. In case LP based classification the best results are obtained when LP classifiers used SMO as base classifier.

The classification results of ML-kNN are given in Table 8.9. In case of ML-kNN classifiers, we obtained high precision and recall values as compared to LP classifier, while accuracy value is low. In case of ML-kNN classifiers, we obtained 71% precision, 40% recall, 37% accuracy and 2.5% HamminLoss. Table 8.9 shows that the evaluation measures are slightly changed when trained with different cluster size, i.e., 6, 8 and 10. However, classification results of ML-kNN are different when trained with the data sets of different label size. The best classification results are obtained when classifiers are trained with the data set that has 91 labels.

Table 8.10 depicts the average classification measures of LP and ML-kNN classifiers. The data of Table 8.10 is derived from Table 8.8 and 8.9. We took the average of each classification measure which is obtained when classifiers are trained with different dimensional data sets, i.e., LSI and term-global-frequency (TGF). The trend of the average classification measures is similar to the trend, which are found in Table 8.8 and 8.9 data. The maximum obtained average precision value is 70.93%, which is obtained when ML-kNN (cluster 8) classifier is used to classify the SCR data.

Figure 8.9 and 8.10 shows a plot between the obtained precision and recall values against different classifiers. The trend of the graph shows that the precision and recall values are maximum when a ML-kNN is used for the classification. While the minimum values of precision and recall are found when LP classifiers are used to classify CR using J48 as base classifier. From Figure 8.9 and 8.9, it may be observed that there is a small variation in the precision and recall values, when classifiers trained with data sets whose dimensions are reduced by LSI or TGF (term-global frequency). It shows low performance of LSI, one possible reason not to attain an optimum performance of LSI is

Table 8.8. Multi-Label Classification of Bug Reports Using *Label Power Set* Classifiers (Problem Transformation Methods Based on Support Vector Machine, J48 and Naive Bayes)

Labels	Metrics	Support Vector Machine (SMO)						J48						Naive Bayes					
		Term Global Freq.			LSI			Term Global Freq.			LSI			Term Global Freq.			LSI		
		3	6	9	100	150	200	3	6	9	100	150	200	3	6	9	100	150	200
44	HammingLoss	0.075	0.078	0.073	0.075	0.073	0.073	0.078	0.078	0.078	0.097	0.095	0.098	0.075	0.077	0.077	0.076	0.078	0.078
	Accuracy	0.356	0.342	0.367	0.353	0.368	0.371	0.336	0.342	0.334	0.208	0.211	0.193	0.356	0.340	0.333	0.350	0.330	0.331
	Precision	0.453	0.430	0.464	0.450	0.464	0.467	0.430	0.430	0.429	0.292	0.300	0.279	0.452	0.437	0.433	0.442	0.425	0.427
	Recall	0.453	0.430	0.464	0.450	0.464	0.467	0.430	0.430	0.429	0.292	0.300	0.279	0.452	0.437	0.433	0.442	0.425	0.427
	Fmeasure	0.453	0.430	0.464	0.450	0.464	0.467	0.430	0.430	0.429	0.292	0.300	0.279	0.452	0.437	0.433	0.442	0.425	0.427
59	HammingLoss	0.059	0.055	0.055	0.056	0.055	0.055	0.059	0.058	0.058	0.072	0.076	0.075	0.060	0.062	0.062	0.058	0.060	0.062
	Accuracy	0.331	0.366	0.371	0.363	0.366	0.365	0.330	0.344	0.336	0.214	0.181	0.185	0.318	0.301	0.304	0.336	0.326	0.309
	Precision	0.421	0.458	0.463	0.452	0.456	0.456	0.417	0.433	0.426	0.293	0.257	0.262	0.407	0.392	0.393	0.427	0.414	0.395
	Recall	0.421	0.458	0.463	0.452	0.456	0.456	0.417	0.433	0.426	0.293	0.257	0.262	0.407	0.392	0.393	0.427	0.414	0.395
	Fmeasure	0.421	0.458	0.463	0.452	0.456	0.456	0.417	0.433	0.426	0.293	0.257	0.262	0.407	0.392	0.393	0.427	0.414	0.395
91	HammingLoss	0.032	0.031	0.031	0.031	0.031	**0.030**	0.033	0.033	0.032	0.042	0.044	0.045	0.035	0.035	0.036	0.034	0.035	0.026
	Accuracy	0.422	0.446	0.445	0.442	0.443	**0.452**	0.416	0.414	0.415	0.278	0.237	0.235	0.379	0.379	0.360	0.395	0.384	0.357
	Precision	0.515	0.536	0.535	0.532	0.533	**0.540**	0.503	0.503	0.507	0.368	0.326	0.322	0.470	0.470	0.450	0.484	0.472	0.687
	Recall	0.515	0.536	0.535	0.532	0.533	**0.540**	0.503	0.503	0.507	0.368	0.326	0.322	0.470	0.470	0.450	0.484	0.472	0.385
	Fmeasure	0.515	0.536	0.535	0.532	0.533	**0.540**	0.503	0.503	0.507	0.368	0.326	0.322	0.470	0.470	0.450	0.484	0.472	0.493

Table 8.9. Multi-Label Classification of Bug Reports Using *ML-kNN* Classifiers (Algorithm Adaptation Methods)

Label	Metrics	Cluster 6						Cluster 8						Cluster 10					
		Term Global Freq.			LSI			Term Global Freq.			LSI			Term Global Freq.			LSI		
		3	6	9	100	150	200	3	6	9	100	150	200	3	6	9	100	150	200
44	HammingLoss	0.058	0.059	0.059	0.057	0.058	0.059	0.058	0.058	0.058	0.057	0.058	0.059	0.057	0.058	0.059	0.058	0.057	0.058
	Accuracy	0.306	0.293	0.282	0.310	0.297	0.283	0.305	0.298	0.294	0.308	0.292	0.283	0.304	0.297	0.290	0.300	0.304	0.304
	Precision	0.652	0.641	0.639	0.664	0.637	0.631	0.656	0.654	0.658	0.665	0.657	0.643	0.680	0.654	0.647	0.656	0.664	0.662
	Recall	0.324	0.311	0.299	0.328	0.314	0.298	0.324	0.315	0.309	0.325	0.306	0.298	0.320	0.313	0.307	0.314	0.321	0.321
	Fmeasure	0.432	0.418	0.407	0.438	0.420	0.404	0.433	0.424	0.420	0.436	0.417	0.407	0.435	0.423	0.416	0.424	0.432	0.431
59	HammingLoss	0.043	0.043	0.044	0.042	0.043	0.043	0.043	0.043	0.043	0.042	0.042	0.042	0.043	0.042	0.043	0.043	0.043	0.043
	Accuracy	0.317	0.310	0.294	0.317	0.304	0.286	0.311	0.305	0.287	0.319	0.322	0.324	0.315	0.318	0.321	0.295	0.316	0.307
	Precision	0.641	0.647	0.610	0.631	0.633	0.603	0.629	0.634	0.591	0.638	0.644	0.643	0.626	0.641	0.637	0.615	0.633	0.632
	Recall	0.340	0.329	0.313	0.339	0.326	0.305	0.332	0.326	0.306	0.341	0.347	0.346	0.336	0.338	0.342	0.312	0.336	0.325
	Fmeasure	0.444	0.436	0.414	0.440	0.430	0.405	0.434	0.430	0.403	0.444	0.451	0.449	0.437	0.442	0.444	0.413	0.438	0.429
91	HammingLoss	0.025	0.026	0.026	0.025	0.026	0.026	0.025	**0.025**	0.026	0.025	0.026	0.026	0.025	0.026	0.026	0.026	0.026	0.026
	Accuracy	0.382	0.363	0.344	0.362	0.360	0.357	0.371	**0.387**	0.354	0.361	0.350	0.349	0.368	0.349	0.337	0.346	0.341	0.349
	Precision	0.710	0.701	0.689	0.697	0.694	0.687	0.707	**0.713**	0.708	0.700	0.702	0.699	0.706	0.699	0.695	0.690	0.693	0.701
	Recall	0.415	0.390	0.372	0.387	0.385	0.385	0.399	**0.401**	0.380	0.387	0.373	0.371	0.392	0.371	0.357	0.366	0.364	0.375
	Fmeasure	0.523	0.501	0.483	0.497	0.495	0.493	0.510	**0.510**	0.495	0.498	0.487	0.484	0.504	0.484	0.471	0.478	0.477	0.488

Table 8.10. Average Classification Measures of Multi-Label Classifiers i.e., *LP-method* and *ML-kNN*

Label	Metrics	Dimension Reduction Using LSI						Dimension Reduction Using Term Global Frequency (TGF)					
		LP SMO	LP J48	LP NB	ML-kNN (Clst6)	ML-kNN (Clst8)	ML-kNN (Clst10)	LP SMO	LP J48	LP NB	ML-kNN (Clst6)	ML-kNN (Clst8)	ML-kNN (Clst10)
44	HammingLoss	07.37	09.67	07.73	05.80	05.80	05.77	07.53	07.80	7.63	05.87	05.80	05.80
	Accuracy	36.40	20.40	33.70	29.67	29.43	30.27	35.50	33.73	34.30	29.37	29.90	29.70
	Precision	46.03	29.03	43.13	64.40	**65.50**	**66.07**	44.90	42.97	44.07	64.40	65.60	66.03
	Recall	46.03	29.03	43.13	31.33	30.97	31.87	44.90	42.97	44.07	31.13	31.60	31.33
	Fmeasure	46.03	29.03	43.13	42.07	42.00	42.90	44.90	42.97	44.07	41.90	42.57	42.47
59	HammingLoss	05.53	07.43	06.00	04.27	04.20	04.30	05.63	05.83	06.13	04.33	04.30	04.27
	Accuracy	36.47	19.33	32.37	30.23	32.17	30.60	35.60	33.67	30.77	30.70	30.10	31.80
	Precision	**45.47**	27.07	41.20	62.23	64.17	62.67	44.73	42.53	39.73	63.27	61.80	63.47
	Recall	45.47	27.07	41.20	32.33	34.47	32.43	44.73	42.53	39.73	32.73	32.13	33.87
	Fmeasure	45.47	27.07	41.20	42.50	44.80	42.67	44.73	42.53	39.73	43.13	42.23	44.10
91	HammingLoss	03.07	04.37	03.17	02.57	02.57	02.60	03.13	03.27	03.53	02.57	02.53	02.57
	Accuracy	**44.57**	25.00	37.87	35.97	35.33	34.53	43.77	41.50	37.27	36.30	37.07	35.13
	Precision	**53.50**	33.87	54.77	69.27	**70.03**	**69.47**	52.87	50.43	46.33	**70.00**	**70.93**	**70.00**
	Recall	**53.50**	33.87	44.70	38.57	37.70	36.83	52.87	**50.43**	46.33	39.23	**39.33**	37.33
	Fmeasure	**53.50**	33.87	48.30	49.50	48.97	48.10	52.87	50.43	46.33	50.23	50.50	48.63

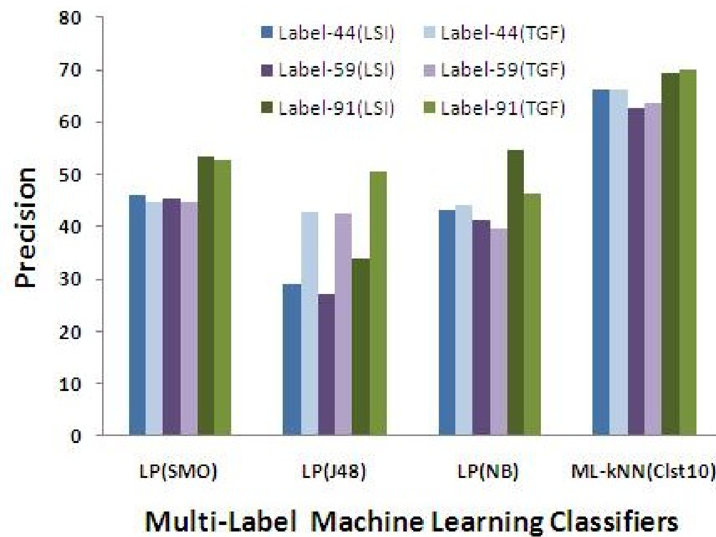


Figure 8.9. The average Precision values of Multi-label Classifiers i.e., Label Power Set and ML-kNN methods (with data sets of 3-different Label size)

the unstructured format of bug reports, which is not favorable for latent semantic analysis. However, the precision and recall values increase when the classifiers are trained with the data set of large number of labels. The possible reasons to obtain the high precision and recall values is the large number of instances. If we increase the label size then it automatically increase the number of data instances.

8.4 Threats to Validity

There are some threats that are related to the labeling of bug reports. In this section we describe them briefly.

The size of the labels plays an important role in prediction accuracy. It is difficult to identify an optimum size of labels. In fact, its value varies from project to project, and is larger for those projects that have a large development team and high number of source files. However, during labeling, we consider only those developers, who are very active, and only consider those source files, which are frequently changed. Therefore, we obtained a lower number of labels.

We assigned a single developer to each bug report. However, sometimes multiple developers are involved in fixing a bug or different developers are available to fix specific types of bugs. One approach is to assign a group of developers to a bug report. Then in that case, after classification one has to select the single most appropriate developer from the predicted group of developers. This makes the prediction process semi-automated and needs an expert, who should be able to choose an appropriate developer from the obtained group of developers. In this way, we might improve the prediction

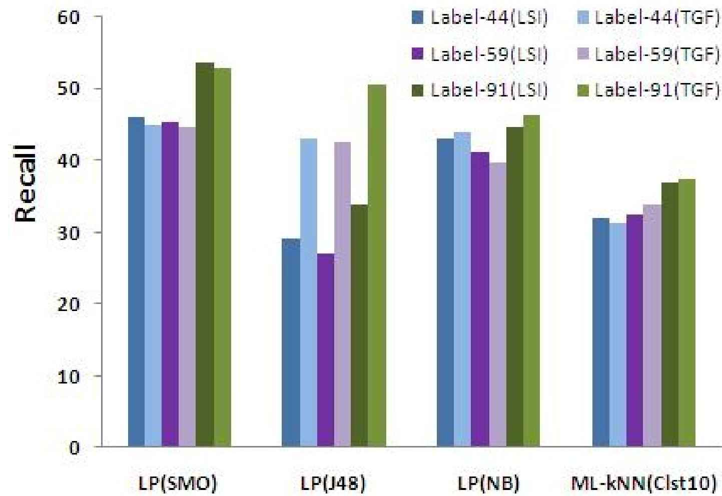


Figure 8.10. The average Recall values of Multi-label Classifiers i.e., Label Power Set and ML-kNN methods (with data sets of 3-different Label size)

accuracy but at the cost of full automation.

We assigned multiple source files to each bug report, due to which the size of the labels increased. The other solution is to first make the group of source files on the basis of source file co-change analysis, and then label each bug report with those groups. In this way, we reduce the size of labels and increase the predictive power, but the final size of the predicted source files becomes large, which may create a burden on developers to spend more time to choose the proper files for modification.

We assigned, mean estimated effort value to each bug report, by dividing the large class size of effort data (i.e., The time spent to fix a bug) into different class intervals. We used the mean value of each class interval as the effort value for those bug reports whose effort value lie in that class interval. Since in our data set the standard deviation for each class interval is small therefore it has low impact on predicted effort value, however if the number of classes interval is large with large deviation from mean class values then it will decrease the prediction accuracy for effort value. Another threat is the validity of effort data itself, since most of the OSS systems do not maintain effort data. Therefore, we used our own heuristic approach to obtain the effort data.

8.5 Summary

In this chapter, we have presented two different approaches for the construction of an automatic bug triage system. Our approaches are based on the single and multi label classification of SCRs. Furthermore, we used *Latent Semantic Indexing* technique of information retrieval to resolve the indexing and high dimensionality issue of the classical vector space model. To validate both the approaches, we have performed two different experiments and discussed their results.

In the first experiment, we presented a comparative analysis of different available information retrieval and machine learning methods in order to automate the assignment of bug reports to developers in an optimal fashion. Our target is to find a combination of methods, which is most suitable to finally develop a high performance bug triage system. We performed experiments using 792 bug reports. These bug reports are transformed into five different data sets on the basis of different indexing and dimension reduction methods of information retrieval. Finally, we used these five data sets and apply multiple machine learning algorithms. All of these seven ML algorithms belong to the class of supervised machine learning methods, and are used for single-label multi-class classification. Our experimental results indicate that classification based on LSI and support vector machine has the best accuracy, precision, and recall values.

In the second experiment, for classifying software change requests, we used multi-label machine learning classifiers. We used information retrieval techniques for the indexing and labeling of software change requests with multiple labels. These labels belong to three different categories, i.e., file name, developer name and expected time spent to resolve the request. We have evaluated our technique on different groups of data of a large open source project, Mozilla. The obtained results show that the multi-label classification techniques of machine learning classify software change requests using multiple labels. The main advantage of multi-label classification is that the software change requests can be automatically assigned to the appropriate developers. Furthermore, it provides the list of source files, which are required to be modified, and provides the expected time required to resolve the change request. In this experiment, we used two different methods of multi-label classification, i.e., problem transformation and algorithm adaptation method. We further analyzed the impact of label size, label cardinality and label density on the classification results, by using three different data sets of different label size. Our experimental results show that the size of labels affected the multi-label classification results, and the classification result is improved by increasing number of instances per labels. We got the best classification result at label size 91 while the number of instances was maximum, i.e., 808. Our technique could classify software change requests with 71.3% precision and 40.1% recall.

9

Related Work

Software repositories contain a huge amount of useful information for empirical studies in software engineering. These repositories mainly contain data related to the changes in the source code, bug reports and developer's communications. Research is now proceeding to mine these repositories and acquire some useful knowledge or patterns related to the fixing of bugs, or inducing of bugs, or impact analysis of software change requests, etc. The field of MSR provides a platform for those researchers who are interested to mine the hidden knowledge from the software repositories. Particularly, the focus of MSR research is to transform repositories from a static record keeping into active repositories. MSR's research is now proceeding to uncover the ways in which mining software repositories can help support the maintenance of software systems, improve software design/reuse, and empirically validate novel ideas and techniques.

The research trend in MSR reveals that MSR is very inter-disciplinary. The techniques and approaches used in MSR are commonly come from machine learning, information retrieval, applied statistics, artificial intelligence, social science and software engineering [Zimmermann, 2009]. In this chapter our objective is to discuss related research work. Therefore, in the reset of the chapter, first different approaches and techniques are discussed for extracting valuable information from software repositories. Next, we describe those research works in detail, which are related to our thesis work.

9.1 Extracting Data From Software Repositories

Fischer et al. [2003] extracted data from a version control system (CVS) and a bug tracking system (Bugzilla) to populate a release history database. They identified that although a version control system and a bug tracking system contains an enormous amount of useful data but these systems provide insufficient support for a detailed analysis of software evolution aspects. They addressed this problem and introduced an approach for populating a release history database that combines version data with bug tracking data and adds missing data, which are not covered by version control systems. They applied their technique on the large Open Source project Mozilla. We used a similar approach to extract the data and populated our own database for our research work. However, in contrast to this work we store further information about different types of change transaction in our database. Furthermore, we populated our database with code metrics data.

Čubranić and Murphy [2003] presented a tool Hipikat that forms implicit group memory for a project by predicting links between stored artifacts and then recommends relevant part of the group memory to a developer working on the task. It groups four types of artifacts: bug and feature descriptions, source file revisions, messages posted on developer forums, and other project documents. It provides great helps to new developers in an open source project by providing an efficient and effective access to the group memory for a software development project. Hipikat can be considered as a recommender system for software developers that draws its recommendation from the development history of OSS projects.

Zimmermann [2006] developed a java plug-in named APFEL that collects change data from version archives and stores them into a database. APFEL collects the change information between two revisions at the token level like method calls but does not identify changes accordingly to their types.

9.2 Bug Fix Effort Estimation Model for OSS

In case of an effort estimation model various approaches have been used to establish estimation models, like an algorithmic, analogy, hybrid and machine learning. Initially, for software estimation models algorithmic estimation methods were used, like Boehm's constructive cost model COCOMO [Boehm, 1981] and COCOMO II [Boehm et al., 1995], and Putnam's software life cycle management (SLIM) [Putnam, 1978].

Jørgensen [1995] developed eleven different effort prediction models based on regression, neural network and pattern recognition and reported that the model based on regression, and pattern recognition are best compared to other models. Eick et al. [2001] worked on the software evolution data of fifteen years. They showed that the code decays, means if the code life is large then it needs more effort to add new changes. They extracted a large number of features from the evolution data and constructed multiple models. Their regression based effort estimation model shows that more effort is required to make changes in the older source code. Their model also shows that the number of added or deleted lines has less impact on an effort.

De Lucia et al. [2002] obtained a data set from five different projects. They used multiple linear regression to build the estimation models. They found that the performance of their models was enhanced if they included the different types of a maintenance task into their models. They used cross validation to assess their models. Song et al. [2006] used the NASA's SEL defect data set and applied association rule mining on the data set, to classify the effort using intervals. They found that association rule mining technique is better to predict the effort compared with other machine learning techniques like PART, C4.5., and Naïve Bayes.

Yu [2006b] analysed an evolution data set of 121 revisions of Linux to develop an effort estimation model for OSS. Since Linux does not maintain effort data. Therefore, he first performed an experiment on NASA SEL database, which is a closed software system and maintained actual effort data. From this experiment, he identified those measures which can be used indirectly to represent a maintenance effort. In the next step, he used those indirect measures as an effort and developed two regressions

based estimation model for Linux project. Our work is similar to his work, but we used different approach to obtain the effort data. Furthermore, we used both multiple linear regression and machine learning methods to develop an effort estimation model.

Weiss et al. [2007] used the available effort data of the JBoss project, which is maintained by the JIRA bug reporting system. They used the text similarity and the nearest neighbor approach, and obtained the average effort data of all the resolved bugs whose summary and title are similar to the new bug report summary and title. They used the obtained average effort value as the predicted effort value of the new bug report. Koch [2008a] discussed the issue of programmer participation and effort modeling for OSS. He worked on the GNOME project data and estimated the effort on the basis of programmer participation and the product metrics. He showed that the impact of programmer participation on effort estimation is less compared to the product metrics.

9.3 Analysis of Source Code Changes to Identify Risky Source Files

Mockus and Weiss [2000] presented a model for the risk analysis of changes in the source code. They presented a model to predict the risk associated with the new changes. Their risk model based on historical information. The HATARI tool developed by Śliwerski et al. [2005a] provides an eclipse plug-in tool to the developer that indicates a risky code region using a complete code history. HATARI relates version archives to a bug database to locate the risky code locations. HATARI makes this risk visible to developers by annotating source code with color bars. Moreover, HATARI provides views to browse through the most risky locations and to analyze the risk history of a particular location in a source code.

Sliwerski et al. [2005] introduced *kenyon*. It is used to analyze the CVS repository of Mozilla and Eclipse together with the information stored in the corresponding Bugzilla bug reporting system to identify a fix inducing changes. They introduced a technique, which used CVS commands *annotation* and *diff* to identify the exact location of source code lines, which introduced the actual faults into the systems. Later their technique is known as *AZZ* algorithm.

Bevan et al. [2005] provides an automatic system for the extraction of source code changes data along with change metadata from software repositories like CVS and Subversion. Their system is called *Kenyon*. It may be used to automatically check out user defined ranges of revisions from source code repositories. Kenyon reduces the time of research, automates configuration retrieval and allows user control on configuration times. Different version control systems and multiple data input sources are supported. Kenyon provides efficient, accessible, and optional storage of extracted facts or information.

Kim et al. [2006a] built bug fix memories by extracting change data from bug fixing changes. They developed a tool named *BugMem*, which learned from previous bug fix change data. They used the algorithm *AZZ* to locate a bug introducing changes in the source code. They only considered bug fixing changes and left other changes like a bug introducing and bug fix-introducing changes.

Kim et al. [2008] in their work, *Classifying Software Changes Clean or Buggy ?* introduced a technique, which is used to predict software bugs. They called it change classification. They used machine learning techniques to classify software changes. They classified program file changes of 12 open source projects into the two classes clean or buggy. They used machine learning classifier to identify whether a current change in a program file is more comparable to any existing buggy or clean change. They trained the machine learning classifier using some features from the source code. These features were extracted from change metadata, complexity metrics, change source code, change to log message and file names. In our experiment, we also used the history of different types of changes. However, instead to use the change information and perform classification, we preferred to build our own model, which numerically predicts the risk value for a source file.

9.4 Fault Prediction Model

A lot of research has been carried out regarding complexity metrics and prediction of faults. Chidamber and Kemerer [1994] initially proposed the object-oriented metrics. Gyimothy et al. [2005] validated the object-oriented metrics for fault prediction in open source software. In this paper, the authors used logistic regression and machine learning techniques to identify faulty classes in mozilla. Zhang and Tsai [February 28, 2005] showed the application of machine learning techniques in tackling software engineering problems. He suggested that machine learning algorithms can be used to build tools for software development and maintenance tasks as well as can be incorporated into software products to make them adaptive and self-configuring.

Porter and Selby [1990] used classification trees based on metrics from previous releases to identify components having high-risk properties. The authors developed a method of automatically generating measurement-based models of high-risk components. Brun and Ernst [2004] used machine learning techniques to model program properties that result in errors. The authors applied these models to classify and rank properties of user written programs that may result in errors. Song et al. [2006] simulated data sets for comparing prediction techniques, including regression, rule induction, the nearest neighbor (a form of case-based reasoning), and neural networks. They concluded that the results of the prediction technique vary with the characteristics of the data set being used. Aljahdali et al. [2001] used feed-forward neural network to predict the faults in a program during the initial test/debug process. They also compared the results between regression models and neural networks. Fenton and Neil [1999] provided a critical review of the defect prediction models based on software metrics. They discussed the weaknesses of the models proposed previously and identified causes for these shortcomings. Kim et al. Boetticher [2005] performed an experiment to show how biased data distribution impacts the results and presented two nearest neighbor sampling approaches. Koru and Liu [2005a] combined static software measure with defect data at the class level and applied different machine learning techniques to develop a bug predictor model. Now, in the following we describe some research work related to logical couplings.

In the recent years, few attempts have been made by researchers to correlate the logical-coupling metrics with the number of faults [Eaddy et al., 2008b, D'Ambros et al., 2009a]. However, most of the research works have been performed, which discussed the significance of logical coupling [Mens and

Lanza, 2002, Zhou et al., 2008, Zimmermann et al., 2005b]. D'Ambros et al. [2009a] has addressed this issue, and correlated logical coupling measures with the number of bugs. Researchers explored the different aspect of logical or change coupling analysis like coupling visualization, coupling measures, software complexity, etc. [Mens and Lanza, 2002, Zhou et al., 2008, Zimmermann et al., 2005b]. Zhou et al. [2008] developed a technique called CAESAR to identify the change patterns by identifying the logical coupling among modules across several releases of a software. Their technique finds the co-change patterns among the source files of different modules. For this purpose, they processed the evolution data of a software system. Their technique found dependencies among modules. They validated these dependencies by examining the change reports.

Breu [2006] processed the co-change data for aspect mining, which identifies cross-cutting concerns in a program. They identify and rank cross-cutting concerns. Zimmermann et al. Zimmermann et al. [2004] applied data mining techniques on software change data sets. The authors obtained information regarding which entities changed together in the past and used this information to guide the developers in case of new changes. They used pattern mining and association rules to perform the experiment. Bieman et al. [2003b] processed those cluster of classes, which are most frequently changed together. They described a method to identify and visualize classes and their interactions that are most change-prone. They measured the degree of change-proneness with the help three metrics, i.e., local change proneness (related to single class change), pair change coupling (when two classes changed together) and sum of pair coupling (a class may be involved in multiple pair couplings).

Zhou et al. [2008] proposed a change propagation model based on Bayesian Network, which may be used to predict future change coupling. To build the model, they extracted a set of features from the co-changed data i.e., static source code dependency (such as message passing between classes, inheritance, or interface implementations), past co-change frequency, change a significance level (assign a significance level to source code changes in terms of the change impact on other source code entities e.g., changing the signature of a method will impact all of its callers. They classify each change on the basis of significance level as low, middle, high or crucial), age of a change, author information, change request, change candidate and co-change entity. They conducted an experiment on two medium open source project, i.e., Azureus and ArgoUML. They showed that the Bayesian network that was built by using the selected set of features provides useful predictions about a change coupling candidate. Their obtained model performed well with 80% precision and 60% recall.

Fluri et al. [2005] developed a method which considered only the structural changes in the source code to perform change coupling analysis. They extracted structural change data from source codes, because a version control system maintain only textual data. The textual data of a version control system are normally used for change coupling analysis, while there are chances that the change coupling is caused by the textual modification, such as the changes in the license text. Therefore, before performing the change coupling analysis, they filtered change coupling data and considered only those change couplings which are caused by structural changes in the source codes. For this purpose, they presented an approach that used the structural compare services shipped with the Eclipse IDE to obtain the fine-grained changes between subsequent versions of Java's classes. This information enables to identify those change couplings which result due to structural changes. Hence they

distilled the causes for change couplings along releases and filter out those that are structurally irrelevant. They found that reasonable amount of change couplings are not caused by source code changes.

Eaddy et al. [2008b] correlated cross-cutting concerns with defects. They performed an empirical study on three open source projects. They used the set of metrics related to the cross-cutting concerns and correlate them with defects. Their experimental results reveal that more scattered and tangled code for the implementation of a concern is highly correlated with the defect. D'Ambros et al. D'Ambros et al. [2009a] pointed out that change coupling (or logical coupling) measures are highly correlated with bugs. They defined a set of change coupling metrics and used them with Chidamber and Kemerer metric suite to correlate them with the number of bugs. Their experimental results show that change coupling metrics are more correlated with bugs as compared to other metrics such as complexity metrics. They performed their experiment on three large software system, i.e., Eclipse, ArgoUML and Mylyn. Their experimental results proved that the performance of a bug prediction model based on complexity metrics can be improved if using change coupling measures. To keep in mind the high correlation of change coupling measures with the number of bugs, and to further validate D'Ambros's approach, we designed an experiment on different data set and introduce a new set of change coupling metrics. Finally, we used these set of logical coupling metrics to build a high performance fault prediction model.

9.5 Machine Learning Classification of Software Change Request

To the best of our knowledge there is no work has been done using multi-label techniques for the labeling and classification of software change requests. However, there are few papers, which used single label classification technique for the classification of software change requests. In the following paragraphs, we discuss some of the related works, which are related to automatic bug triaging and impact analysis using classification of SCRs.

Cubranic [2004] worked on bug triage. They used Eclipse bug report data and labeled each bug report with the developer name who actually fixed them. They obtained 30% classification accuracy using Naive Bayes classifier. They used vector space model for indexing. They performed classification with and without truncation of vocabulary. The truncation of vocabulary is actually the feature selection technique. They trained the classifier with different data set. Each data set was obtained by selecting only those vocabularies that have some specific frequency of occurrence between the documents, i.e. more than 1, 2, 5, 10 and 20.

Anvik et al. [2006] used Eclipse and Firefox project data sets to present a semi automated solution for the assignment of bug reports to the set of appropriate developers. They developed a technique which parsed the bug report summary and description text, and obtained a set of index terms. They labeled each bug report with the group of developer names suitable to resolve them. Their work is based on vector space model and machine learning. For classification they used SVM, Naive Bayes and C4.5. They got the precision levels 57% for Eclipse and 64% for Firefox project. Another related work that used the information retrieval (IR) and the ML methods for the classification of bug reports is presented by Weiss et al. [2007], they used the average bug fix time of similar bug fix reports for

the effort prediction.

Canfora and Cerulo [2005] used bug reports and source file revision data of four different OSS projects. They used a probabilistic information retrieval to link the change request description to the set of historical source file revisions impacted by similar past change requests (CR). For this purpose Gerardo and Luigi constructed file descriptors for each source file by merging the textual data of the revision check in comments, the textual data of the summary, and the description of a CR. Moreover, they used textual similarity measure to retrieve past CRs, which are similar to the new CR. From these CRs the Gerardo and Luigi approach computes a ranked list of possible impacted source files. Our approach is more related to Canfora because we linked SCRs/bug-reports with impacted source file names to perform impact analysis. However, our approach is different. We have not used probabilistic model of information retrieval rather we first used the indexing technique of information retrieval, i.e., LSI to index the key term of bug report's textual data. Then, labeled each SCR/bug-reports with one or more impacted source files, and applied multi-label classification to perform the impact analysis.

All the mentioned previously published papers are similar to our research work. The main difference lies in the classification. All the past approaches classify bug fix reports with single labels, whereas we use multi-label techniques for the same purpose. Therefore, our proposed technique is more comprehensive regarding the classification of bug fix change requests. Furthermore, we use latent semantic indexing methods for indexing, whereas in most of the above mentioned approaches vector space models are used. LSI is a more advanced technique compared to classical vector space models. Other works that is related to the text classification using information retrieval and multi-label techniques include [Chen et al., 2007], and Joachims T. [Tsoumakas et al., 2009].

10 Conclusion and Future Work

10.1 Conclusion

The work presented in this dissertation is based on the application of machine learning techniques on software engineering data. This thesis contributed research in the field of mining software repositories. MSR is a very inter-disciplinary field. Researchers belong to MSR are normally motivated to apply machine learning techniques on the software engineering data and come up with some new models for the automation of software development and software maintenance.

The overall thesis of our research work is that change history of source files and bug reports have enough information to construct the state of the art estimation and prediction models, and develop methods to support the development and the maintenance task of software engineering projects. This thesis makes the following contributions in this area.

10.1.1 Estimation and Prediction Models

The presented work on the effort estimation model for OSS, based on a novel approach, which mine effort data from a software repository. The purpose of an effort estimation model is to estimate that how much effort is required to fix a fault. To build an effort estimation model one needs effort data and a set of metric as estimators. However, the dilemma is, most of the OSS does not maintain effort data. Therefore, we built an effort estimation model in two steps. First, we extracted the effort data in bug-fixing days from the logs of developer's bug-fixing activity. However, finding the exact number of bug-fixing days is non trivial. Because, we found that in case of OSS, multiple parameters are responsible to vary the bug-fixing time. The two important parameters, which impact the bug-fixing duration are developer's priority levels and bug' severity levels. We found that in case of OSS, developers do not treat bugs equally. Rather bugs are treated on the basis of their severity levels. We considered these parameters in computing the numbers of days, which spent in fixing a bug. Once we obtained the exact number of bug-fixing days (i.e., effort data), then we used a set of metrics and regression methods to build effort estimation models. All the models are based on the same underlying metrics and effort data and trained with the data set of Mozilla project. The obtained results show that the machine learning and multiple linear regression based effort estimation models have correlation values between 0.51 and 0.56. Similarly, the MMRE (mean magnitude of

relative error) values lie between 63% and 93%. This shows that the performance of our models is satisfactory.

In this thesis, we presented a novel approach to build a risk model for the analysis of software changes. In case of risk estimation models, the purpose is to estimate that the risk of faults, which is associated with a source file for new changes. To construct a risk estimation model, we used the data of different types of software's changes, particularly we used the historical data of faulty changes. We showed that how these changes are distributed during the evolution of a source file. Moreover, we used an approach to transform the previous change history of source files into the four types of changes, i.e., bug-fix, bug fix-introducing, buggy changes and clean changes. Each source file has different values for these changes. We used this information and built a risk model. In order to build the model, we performed a case study on Mozilla project data. We found that if a source file has a high probability of faulty changes and also fixed large number of bugs, then such source files are risky for new changes. We can conclude that the files, which have higher risk values are riskier when applying the next change and thus such source files demands more attention and should be tested more thoroughly.

The main contribution of this part of the thesis is to present a high performance fault prediction model. In order to obtain a high performance fault prediction model, we performed two different experiments. In the first experiment, we used two different techniques, i.e., regression and classification to build the fault prediction models. In this experiment, we trained and tested each model using raw metric (without PCA) and principal components. Although the results of classification evaluation using raw metrics and PCA are almost similar, but we preferred to choose the classification evaluation measures, which are obtained using principal components. The reason is, the evaluation results obtained from principal components are not over estimated. Whereas, the evaluation results obtained using raw metrics may be over estimated, because in case raw metrics data collinearity exist among the metrics.

In case of regression based fault prediction model, we used parametric regression techniques, i.e., multiple linear regression and non-parametric regression, i.e., neural network, support vector machine (SMOreg), radial basis function, and REPTree. The evaluation results show that in case of regression based fault prediction model, the best prediction models are multiple linear regression, and neural network. In case of multiple linear regression, the obtained value of R (correlation) and RRSE (root relative square error) are 0.71 and 77.98% respectively. Similarly, in case of neural network, the obtained value R and RRSE are 0.62 and 78.81% respectively.

In case of classification based fault prediction model, we used eight different classifiers, i.e., logistic regression, decision tree (J48), support vector machine (SMO), radial basis function, neural network, naive bayes, AdaBoost and Bagging. In case of binary classification, the performance of the classifiers becomes poor if the class distribution is skewed. Whereas, in case of software fault prediction model, the class distribution is mostly skewed. Because, in any release of a software product, the number of clean source files is greater than the number of faulty source files. Therefore, an objective to perform this experiment is to analyze the performance of each classifiers, when the class distribution is skewed. The skewed class distribution generates biased classification results, because larger classes usually

dominate smaller classes. The consequence is that the classifiers give more weight to the larger classes to maintain higher prediction accuracy. Therefore, in case of skewed class distribution we cannot use the classification evaluation measure *Accuracy* to evaluate the model. We used ROC to analyze the performance of individual class. Because, if the class distribution is skewed then ROC-Area (Receiving Operating Characteristic curve) is normally used to evaluate the performance of the classifiers.

To perform a detail analysis about the impact of skewed class distribution on the performance of a classification based fault prediction models. We built models using two different classification schema, i.e., *isBuggy* and *bugLevel*. In case of classification scheme *isBuggy*, instances classified based on two classes, i.e., *isBuggy*=yes (means buggy file) and *isBuggy*=false (means clean file). Whereas, in case of *bugLevel* classification scheme, instances classified based on three classes, i.e., low, medium, and high. Our experimental results show that, in case of a classification scheme *isBuggy*, the best performed classifier is *Boosting (AdaBoost)*, whose obtained value of precision for the class *isBuggy*=yes is 53.7% at 60.0% recall and 0.79 ROC area. Similarly, in case of a classification scheme *bugLevel*, the best performed classifier is *Naive Bayes*. Whose value of precision for the class *bugLevel*=Medium is 18.7% at 35.2% recall and 0.80 ROC area, and for the class *bugLevel*=High is 23.5% at 24.4% recall and 0.66 ROC area. The results show that classifiers performed well when class distribution follows *isBuggy* classification schem. The reason is, the class distribution *isBuggy* is less skewed as compared to *bugLevel*. Furthermore, results show that in our experiment *bugLevel* has highly skewed distribution. Therefore, most of the classifiers failed to perform well except *Naive Bayes*. Therefore, we can conclude that in case classification based fault prediction models, if the class distribution is highly skewed than *Naive Bayes*, performed well. However, if the class distribution is less skewed then the best classifiers are: bagging, boosting, decision tree (J48), neural network and logistic regression.

The experimental results of fault prediction models reveal that, some prediction models performed well, and their performance can be further improved if we build the model by using those estimators, who have high correlation with the number of faults. Therefore, we performed another experiment in which, we introduced eight new metrics, which are related to the logical couplings among the source files. We extracted these metrics from the logical coupling patterns, which are related to the fixing of faults. We found that these metrics are highly correlated with the number of faults, and therefore, can be used to construct a fault prediction model. The obtained value of R^2 of a regression based fault prediction model (when using logical coupling metrics as predictors) is 80%. Whereas, in case classification based fault prediction model, the obtained value of accuracy is 97%. The models trained and tested using GNOME project data.

The thesis of our research work is that the performance of the classification based fault prediction model is affected if the class distribution of the training data set is skewed. However, there are some classifier who perform well even in case of skewed class distribution, i.e., bagging, boosting and decision tree (J48) and Naive Bayes. In case of regression based fault prediction model, MLR and neural network perform performed well. Furthermore, logical couplings have a significant potential to create bugs in source files, which could be measured with the help some metrics. These metrics should be defined in a way that it should exploit the coupling tendency of a source file as maximum as possible. We have used bug fix patterns to define a set of eight metrics and found that these sets

of metrics are highly correlated with the number of bugs in source files.

10.1.2 Multi-Label Classification of SCRs

In order to perform an impact analysis of SCRs using classification techniques of machine learning. We found that there are some cases in which SCRs impacted more than one source files. Therefore, to perform the supervised machine learning classification, some of the SCR should be labeled with multiple source files names. Hence, single label classification cannot be used directly for the classification of SCRs. In order overcome this issue, in this thesis, we presented two novel techniques for the impact analysis of SCRs. The first technique is semi-automated and use frequent pattern mining and association rules to make different groups of source files, which are changed together. After creating the groups of source files, the group identifier can be used to label each SCR uniquely. As a consequence a single-label machine learning classifier can be used again. Whereas, the other technique is fully automated and use multi-label classification techniques. In case of multi-label classification, we directly label each SCR with the set comprising all changed source files and use multi-label machine learning classification directly.

We evaluated these methods by performing an experiment by using three data sets. These data sets were obtained from three OSS project's repositories, i.e., Mozilla, Eclipse and Gnome. Our experimental results are promising, which indicates that the textual data of past SCRs, which are labeled with the set of impacted source files, can be used to build the supervised machine learning based models for the automation of SCRs impact analysis. In case of single-label, the obtained maximum precision and recall values are 58.2% and 51.1%. Similarly, in case of multi-label. The obtained maximum precision and recall values are 47.1% and 46.5%. These values were obtained when SMO used as single-label, and as base classifier of *Label Power Set* (LP) method of multi-label classification.

In order to develop an automatic bug triage system. In this thesis, we presented a comparative analysis of different available information retrieval and machine learning techniques. Our target is to find a combination of methods, which is most suitable to finally develop a high performance bug triage system. We performed experiments using bug reports (SCRs) of Mozilla project. These bug reports were transformed into five different data sets using different indexing and dimension reduction methods of information retrieval. Finally, we used these five data sets and apply supervised machine learning algorithms. Our experimental results indicate that classification based on *Latent Semantic Indexing* and *Support Vector Machine* has the best accuracy, precision, and recall values.

Furthermore, in this thesis, we presented a novel approach to perform a bug triaging using multi-label classification of SCRs. The main advantage of multi-label classification is that software change requests can be labeled simultaneously with developer name, file name, and effort value (expected time spent to fix a bug report). We found that the classification of multi-labeled SCRs successfully assigned SCR to the appropriate developer. Moreover, it provides a list of source files, which is required to be modified, and provides an expected time required to resolve a change request. In this experiment, we used two methods of multi-label classification, i.e., *Problem Transformation* and *Algorithm Adaptation*. We have evaluated our technique on different groups of data of a large open source project, Mozilla. We further analyzed the impact of label size, label cardinality and label

density on the classification results, by using three different data set of different label size. The results show that the size of labels affected the multi-label classification results. We got the best classification result at label size 91. Our technique could classify software change requests with 71.3% precision and 40.1% recall. The obtained results show that the multi-label classification techniques of machine learning can be used for bug triaging.

10.2 Future Work

During the research work on this thesis, we encountered promising further research directions. In this section, we discuss some of the possible future research work, which will further enhance the presented work of this thesis.

In this thesis, we presented an effort estimation model. We built an effort estimation model by using code and process metrics. We found that the correlation between the process metrics and the effort value is high as compared to the correlation between code metrics and effort value. Since, in our experiment we used only three process metrics, and 12 code metrics. Therefore, further research can be done to find more process metrics, which are highly correlated with the bug-fixing effort value. This will further improve estimation power of our presented effort estimation model. During this experiment, we found that bug-fixing time is depended on two factors, i.e., bug severity levels and bug priority levels. We found that high severity and high priority bugs are fixed in short time. However, we also found that there are several examples in which high severity and high priority bugs are fixed in more than 100 days. Sometimes low priority bugs are fixed in small time. An interesting research work can be done to find that how developers allocate time to fix a bug. Furthermore, research work can be done to find the exact relation of these two factors with the bug-fixing time.

In case of a risk model, we used the data of four different types of source code changes. The quality of this data is based on the link between the developer's comments into a version control system and the corresponding bug reports of a bug tracking system. Since, currently there is no direct way to establish a link between a version control system and a bug tracking system. Therefore, some research work can be done to establish a new bug tracking system, which automatically linked with the version control system. Furthermore, in order to build a risk model we considered the expected cost of the severity as a unit cost, and then estimated the risk value. Therefore, research can be done to find the actual cost related to the severity of bug.

In case of classification based fault prediction model, we found that the class distribution is skewed. Therefore, we used several classification evaluation measures, which are normally used when class distribution is skewed. However, one can use some other machine learning techniques to overcome class imbalance issue or skewed class distribution. Furthermore, we found that logical coupling metrics are highly correlated with the numbers of bugs. Therefore, further research can be done to obtain more coupling metrics for fault prediction models.

In this thesis work, we found that textual classification of SCRs can be used for impact analysis and bug triaging. However, the result of textual classification is not very good. One possible reason is the

textual content of bug reports are unstructured. Therefore, new research work can be done to improve the quality of bug reports. Furthermore, in this thesis we used multi-label classification technique for the classification of SCRs. The obtained results show that the performance of multi-label classification is depending on the label size. Large number of labels reduces the performance of the multi-label classification. Whereas, for the impact analysis or bug triage the numbers of labels are always in a large number. Therefore, further research can be done to modify some of the existing multi-label classification algorithm, so that new modified algorithm ca be used efficiently for the classification of SCRs.

Appendix

A.1 List of Publications

1. Syed Nadeem Ahsan. "Software Bug Prediction Using Program Files Logical-Coupling Metrics", 6th Central and Estern European Software Engineering Conference in Russia (CEE-SECR-2010). Moscow, Russia, 11-15 October, 2010.

2. Syed Nadeem Ahsan and F. Wotawa. "Impact Analysis of SCRs Using Single and Multi-Label Machine Learning Classification", ACM 4th International Symposium on Empirical Software Engineering and Measurement. Bolzano Bozen, Italy, 15-16 September, 2010.

3. Syed Nadeem Ahsan, M.T. Afzal, S. Zaman, C. Gutel and F. Wotawa. "Mining Effort Data from the OSS Repository of Developer's Bug Fix Activity". Journal for Information Technology in Asia (JITA), 2010 (accepted).

4. Syed Nadeem Ahsan, J. Ferzund, F. Wotawa. "Automatic Classification of Software Change Request Using Multi-Label Machine Learning Methods", 33rd Annual IEEE Software Engineering Workshop 2009 (SEW-33), Skövde, Sweden, 13-15 October 2009

5. Javed Ferzund, Syed Nadeem Ahsan and Franz Wotawa. "Empirical Evaluation of Hunk Metrics as Bug Predictors," Software Process and Product Measurement, International Conferences IWSM 2009 and Mensura 2009, Amsterdam, The Netherlands, November 4-6, 2009. Proceedings 2009

6. Javed Ferzund, Syed Nadeem Ahsan and Franz Wotawa. "Bug-Inducing Language Constructs," 16th Working Conference on Reverse Engineering (WCRE 2009) October 13th16th, Lille, France, 2009

7. Syed Nadeem Ahsan, Javed Ferzund and Franz Wotawa, "Automatic Software Bug Triage System (BTS) Based on Latent Semantic Indexing and Support Vector Machine," Proc. Fourth International Conference on Software Engineering Advances (ICSEA 2009), Porto, Portugal, 2009.

8. Syed Nadeem Ahsan, Javed Ferzund, and Franz Wotawa. "Are There Language Specific Bug Patterns? Results Obtained from a Case Study Using Mozilla," Proc. Fourth International Conference on Software Engineering Advances (ICSEA 2009), Porto, Portugal, 2009.

9. Javed Ferzund, Syed Nadeem Ahsan, Franz Wotawa. "Software Change Classification using Hunk Metrics," 25th IEEE International Conference on Software Maintenance (ICSM), Edmonton, Canada, 2009

10. Syed Nadeem Ahsan, Javed Ferzund, Franz Wotawa. "Program File Bug Fix Effort Estimation Using Machine Learning Methods for OSS," 21st International Conference on Software Engineering and Knowledge Engineering (SEKE), July 1-3, 2009, Boston, USA.

11. Syed Nadeem Ahsan, Javed Ferzund, Franz Wotawa, "A Database for the Analysis of Program Change Patterns," ncm, pp. 32-39, 2008 Fourth International Conference on Networked Computing and Advanced Information Management, 2008, S. Korea.

12. Javed Ferzund, Syed Nadeem Ahsan, Franz Wotawa, "Automated Classification of Faults in Programs using Machine Learning Techniques," AISEW 2008, Gress.

13. Javed Ferzund, Syed Nadeem Ahsan, Franz Wotawa, "Analysing Bug Prediction Capabilities of Static Code Metrics in Open Source Software," Mensura 2008, Germany.

B Appendix

B.1 Terminology

List of terms used in this thesis:

Bug Tracking System: A bug tracking system is used to store and manage information about bugs such as when a bug is reported, who reported a bug, short description of a bug, severity of a bug, platform on which a bug is reported, module in which a bug is reported and status of a bug.

Version or Revision: These two terms are used interchangeably. A version or revision represents instance of a file at a particular time. As a software system evolves, changes are made to the files. Revisions are used to identify different instances of a changed file.

Version Control System: It is an important feature of a software configuration management system (SCM), used to manage different revisions of files in a software project. Whereas, SCM is the process of handling changes made to the software during its development. It is used to control the evolution of software projects. SCM comprises four operations: Identification, control, status accounting and audit. (IEEE Guide to Software Configuration Management. 1987. IEEE/ANSI Standard 1042-1987.)

Commit: It is the process of submitting changes to an SCM system. Initially new files of a project are committed to the SCM system. Then each change to a file is committed. A commit may involve a single file or multiple files together.

Change: Software evolution is characterized by making changes to the files. A change represents a single modification stored in the software configuration management system (SCM) repository.

Hunk: Changes are made to files in chunks of source code that are dispersed in a file. These chunks of contiguous source code lines are called hunks. There can be multiple hunks in a change delta.

Change log or CVS log: When a developer commits a change to the SCM system, she/he records a message describing the purpose of the change. This message is called change log. Change logs can be processed to identify different kinds of changes.

Change Annotation or CVS Annotation: It is a basic feature of configuration management systems. An SCM system annotates each source code line with the date of modification, author of the line and the revision in which that line was changed.

Bug: A bug is characterized by a programming mistake or error in source code that results in malfunctioning of software.

Fix: A fix is characterized by replacing erroneous source code with the correct code. A fix is used to remove a bug from software.

Bug Fix Change: A change applied to software, to fix a bug is called a bug fix change.

Bug-Inducing or Bug-Introducing Change: A change which resulted in malfunctioning of software later on is called a bug-inducing change or buggy change.

Bug Fix Hunk: A hunk which is part of a fix is called a bug fix hunk.

Bug-Inducing Hunk: A hunk which resulted in malfunctioning of software later on is called a bug-inducing hunk.

Bug-Fix Developer: A developer who makes changes to fix a bug is called a bug-fix developer.

Bug-Inducing Developer: A developer, modifications made by whom resulted in malfunctioning of software, is called a bug-inducing developer.

Bug-fixing Time: It is the time spent to fix a bug

Bug-fixing Activity: Activities or tasks performed by one or more developers to fix a bug.

Bug fix Effort: An effort, which is required to fix a bug.

Bibliography

Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. pages 487–499, 1994.

Syed Nadeem Ahsan and Franz Wotawa. Software bug prediction using program files logical-coupling metrics. In *6th Central and Eastern European Software Engineering Conference in Russia (CEE-SECR 2010)*, 2010a.

Syed Nadeem Ahsan and Franz Wotawa. Impact analysis of scrs using single and multi-label machine learning classification. In *ESEM*, 2010b.

Syed Nadeem Ahsan and Franz Wotawa. *Fault Prediction Models for Software*. IST Technical Report, TU-Graz, Austria, 2010c.

Syed Nadeem Ahsan, Javed Ferzund, and Franz Wotawa. A database for the analysis of program change patterns. In *NCM (2)*, pages 32–39, 2008.

Syed Nadeem Ahsan, Javed Ferzund, and Franz Wotawa. Automatic software bug triage system (bts) based on latent semantic indexing and support vector machine. In *ICSEA*, pages 216–221, 2009a.

Syed Nadeem Ahsan, Javed Ferzund, and Franz Wotawa. Program file bug fix effort estimation using machine learning methods for oss. In *SEKE*, pages 129–134, 2009b.

Syed Nadeem Ahsan, Javed Ferzund, and Franz Wotawa. Automatic classification of software change request using multi-label machine learning methods. In *33rd Annual IEEE Software Engineering Workshop*, Skövde, Sweden, 2009c.

Syed Nadeem Ahsan, Muhammad Tanvir Afzal, Safdar Zaman, Christian Gütel, and Franz Wotawa. Mining effort data from the oss repository of developer's bug fix activity [accepted]. *Journal for Information Technology in Asia (JITA)*, 2010.

A. J. Albrecht and J. E. Gaffney. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Trans. Softw. Eng.*, 9(6):639–648, 1983. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.1983.235271.

Sultan H. Aljahdali, David Rine, and Alaa Sheta. Prediction of software reliability: A comparison between regression and neural network non-parametric models. *Computer Systems and Applications, ACS/IEEE International Conference on*, 0:0470, 2001. doi: http://doi.ieeecomputersociety.org/10.1109/AICCSA.2001.934046.

Omar Alonso, Premkumar T. Devanbu, and Michael Gertz. Expertise identification and visualization from cvs. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 125–128, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-024-1. doi: http://doi.acm.org/10.1145/1370750.1370780.

John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 361–370, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: http://doi.acm.org/10.1145/1134285.1134336.

Robert S. Arnold and Shawn A. Bohner. Impact analysis - towards a framework for comparison. In *ICSM '93: Proceedings of the Conference on Software Maintenance*, pages 292–301, Washington, DC, USA, 1993. IEEE Computer Society. ISBN 0-8186-4600-4.

Jai Asundi. The need for effort estimation models for open source software projects. In *5-WOSSE: Proceedings of the fifth workshop on Open source software engineering*, pages 1–3, New York, NY, USA, 2005. ACM. ISBN 1-59593-127-9. doi: http://doi.acm.org/10.1145/1083258.1083260.

R. A. Baeza-Yates and B. A. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press Book, Addison Wesley, 1999.

Kim-J. Porter A. A. Ball, T. and H. P. Siy. If your version control system could talk... In *In ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering*. ACM, 1997.

Thomas Ball, Jung min Kim, Adam A. Porter, and Harvey P. Siy. If your version control system could talk... In *In ICSE Workshop on Process Modeling and Empirical Studies of Software Engineering*, 1997.

Jennifer Bevan, E. James Whitehead, Jr., Sunghun Kim, and Michael Godfrey. Facilitating software evolution research with kenyon. *SIGSOFT Softw. Eng. Notes*, 30:177–186, September 2005. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1095430.1081736. URL http://doi.acm.org/10.1145/1095430.1081736.

James M. Bieman, Anneliese A. Andrews, and Helen J. Yang. Understanding change-proneness in oo software through visualization. volume 0, page 44, Los Alamitos, CA, USA, 2003a. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/WPC.2003.1199188.

James M. Bieman, Anneliese A. Andrews, and Helen J. Yang. Understanding change-proneness in oo software through visualization. volume 0, page 44, Los Alamitos, CA, USA, 2003b. IEEE Computer Society. doi: http://doi.ieeecomputersociety.org/10.1109/WPC.2003.1199188.

C. M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, First Edition, October 2007.

B. W. Boehm. *Software Engineering Economics*. Prentice Hall PTR, Englewood Cliffs, 1981.

Barry Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby. Cost models for future software life cycle processes: Cocomo 2.0. 1995.

Gary D. Boetticher. Nearest neighbor sampling for better defect prediction. *SIGSOFT Softw. Eng. Notes*, 30:1–6, May 2005. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1082983.1083173. URL http://doi.acm.org/10.1145/1082983.1083173.

Leo Breiman. Bagging predictors. *Machine Learning*, 24(2):123–140, 1996.

Silvia Breu. Mining aspects from version history. In *In 21st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 221–230. IEEE Computer Society, 2006.

Yuriy Brun and Michael D. Ernst. Finding latent code errors via machine learning over program executions. In *ICSE'04, Proceedings of the 26th International Conference on Software Engineering*, pages 480–490, Edinburgh, Scotland, May 26–28, 2004.

M. Buckland and F. Gey. The relationship between recall and precision. *Journal of the American Society for Information Science*, 45:12–19, 1994.

Gerardo Canfora and Luigi Cerulo. Impact analysis by mining software and change request repositories. *Software Metrics, IEEE International Symposium on*, 0:29, 2005. ISSN 1530-1435. doi: http://doi.ieeecomputersociety.org/10.1109/METRICS.2005.28.

Gerardo Canfora and Aniello Cimitile. *Software Maintenance*. Technical Report, University of Sannio, Faculty of Engineering at Benevento 82100, Benevento Italy, 2000.

Weizhu Chen, Jun Yan, Benyu Zhang, Zheng Chen, and Qiang Yang. Document transformation for multi-label feature selection in text categorization. *Data Mining, IEEE International Conference on*, 0:451–456, 2007. ISSN 1550-4786. doi: http://doi.ieeecomputersociety.org/10.1109/ICDM.2007.18.

S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20(9):47H93, 1994.

Dunsmore H. E. Conte, S. D. and Y. E. Shen. *Metrics and Models*. Software Engg., Benjamin-Cummings Publishing Co., Englewood Cliffs, 1986.

Nello Cristianini and John Shawe Taylor. *An Introduction to Support Vector Machine and other Kernel based Learning Methods*. Cambridge University Press; 1 edition,, March 28, 2000. ISBN 978-0521780193.

Davor Cubranic. Automatic bug triage using text categorization. In *In SEKE 2004: Proceedings of the Sixteenth International Conference on Software Engineering and Knowledge Engineering*, pages 92–97. KSI Press, 2004.

Marco D'Ambros, Michele Lanza, and Romain Robbes. On the relationship between change coupling and software defects. In *WCRE '09: Proceedings of the 2009 16th Working Conference on Reverse Engineering*, pages 135–144, Washington, DC, USA, 2009a. IEEE Computer Society. ISBN 978-0-7695-3867-9. doi: http://dx.doi.org/10.1109/WCRE.2009.19.

Marco D'Ambros, Michele Lanza, and Romain Robbes. On the relationship between change coupling and software defects. *Reverse Engineering, Working Conference on*, 0:135–144, 2009b. ISSN 1095-1350. doi: http://doi.ieeecomputersociety.org/10.1109/WCRE.2009.19.

Andrea De Lucia, Eugenio Pompella, and Silvio Stefanucci. Effort estimation for corrective software maintenance. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, SEKE '02, pages 409–416, New York, NY, USA, 2002. ACM. ISBN 1-58113-556-4. doi: http://doi.acm.org/10.1145/568760.568831. URL http://doi.acm.org/10.1145/568760.568831.

Scott Deerwester, Susan T. Dumais, George W. Furnas, Thomas K. Landauer, and Richard Harshman. Indexing by latent semantic analysis. *JOURNAL OF THE AMERICAN SOCIETY FOR INFORMATION SCIENCE*, 41(6):391–407, 1990.

Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, 34:497–515, 2008a. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2008.36.

Marc Eaddy, Thomas Zimmermann, Kaitlin D. Sherwood, Vibhav Garg, Gail C. Murphy, Nachiappan Nagappan, and Alfred V. Aho. Do crosscutting concerns cause defects? *IEEE Trans. Softw. Eng.*, 34(4):497–515, 2008b. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2008.36.

Stephen G. Eick, Todd L. Graves, Alan F. Karr, J.s. Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27:1–12, 2001. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/32.895984.

Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Trans. Softw. Eng.*, 25(5):675–689, 1999. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.815326.

Javed Ferzund. *An Empirical Investigation into Changes and Bugs by Mining Software Development Histories*. IST, TU-Graz, Austria, 2009.

Javed Ferzund, Syed Nadeem Ahsan, and Franz Wotawa. Automated classification of faults in programms using machine learning techniques, patras, greece. In *In: AISEW, European Conference on Artificial Intelligence*, 2008a.

Javed Ferzund, Syed Nadeem Ahsan, and Franz Wotawa. Analysing bug prediction capabilities of static code metrics in open source software. In *IWSM/Metrikon/Mensura*, pages 331–343, 2008b.

Javed Ferzund, Syed Nadeem Ahsan, and Franz Wotawa. Software change classification using hunk metrics. In *ICSM*, pages 471–474, 2009a.

Javed Ferzund, Syed Nadeem Ahsan, and Franz Wotawa. Bug-inducing language constructs. In *WCRE*, pages 155–159, 2009b.

Andy Field. *Discovering Statistics Using SPSS for Windows*. SAGE Publications, London, 2004. ISBN 0-7619-5754-5 (hbk).

Michael Fischer, Martin Pinzger, and Harald Gall. Populating a release history database from version control and bug tracking systems. In *ICSM '03: Proceedings of the International Conference on Software Maintenance*, page 23, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1905-9.

Beat Fluri, Harald C. Gall, and Martin Pinzger. Fine-Grained Analysis of Change Couplings. In *Proceedings of the 5th International Workshop on Source Code Analysis and Manipulation*, pages 66–74. IEEE Computer Society, October 2005.

Yoav Freund and Robert E. Schapire. A short introduction to boosting. *Japanese Society for Artificial Intelligence*, 14(5), 1999.

Harald Gall, Mehdi Jazayeri, and Jacek Krajewski. Cvs release history data for detecting logical couplings. In *IWPSE '03: Proceedings of the 6th International Workshop on Principles of Software Evolution*, page 13, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1903-2.

Shantanu Godbole and Sunita Sarawagi. Discriminative methods for multi-labeled classification. In *In Proceedings of the 8th Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 22–30. Springer, 2004.

Georgios Gousios, Eirini Kalliamvakou, and Diomidis Spinellis. Measuring developer contribution from software repository data. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 129–132, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-024-1. doi: http://doi.acm.org/10.1145/1370750.1370781.

Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Trans. Softw. Eng.*, 31(10):897–910, 2005. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2005.112.

Elaine M. Hall. *Managing Risk: Methods for Software Systems Development(Sei Series in Software Engineering)*. Addison Wesley, February 28, 2005. ISBN 978-0201255928.

Ahmed E. Hassan. The road ahead for mining software repositories. In *In Proceedings of the Future of Software Maintenance (FoSM) at the 24th IEEE International Conference on Software Maintenance (ICSM), Beijing, China,*, 2008.

Ahmed E. Hassan and Tao Xie. Software intelligence: Future of mining software engineering data. In *Proc. FSE/SDP Workshop on the Future of Software Engineering Research (FoSER 2010)*, November 2010. URL http://www.csc.ncsu.edu/faculty/xie/publications/foser10-si.pdf.

Ahmed E. Hassan, Audris Mockus, Richard C. Holt, and Philip M. Johnson. Guest editor's introduction: Special issue on mining software repositories. *IEEE Transactions on Software Engineering*, 31:426–428, 2005. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2005.70.

Israel Herraiz, Daniel M. German, Jesus M. Gonzalez-Barahona, and Gregorio Robles. Towards a simplification of the bug report form in eclipse. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 145–148, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-024-1. doi: http://doi.acm.org/10.1145/1370750.1370786.

Lulu Huang and Yeong-Tae Song. Dynamic impact analysis using execution profile tracing. *Software Engineering Research, Management and Applications, ACIS International Conference on*, 0:237–244, 2006. doi: http://doi.ieeecomputersociety.org/10.1109/SERA.2006.30.

Eibe Frank Ian H. Witten and Morgan Kaufmann. *Data Mining: Practical Machine Learning Tools and Techniques*. John Wiley and Sons, 2005.

E. J. Jackson. *A User's Guide to Principal Components*. Hoboken, NJ: John Wiley and Sons Inc.,, 2003.

Nathalie Japkowicz. The class imbalance problem: Significance and strategies. In *In Proceedings of the 2000 International Conference on Artificial Intelligence (ICAI*, pages 111–117, 2000.

George H. John and Pat Langley. Estimating continuous distributions in bayesian classifiers. In *Eleventh Conference on Uncertainty in Artificial Intelligence*, pages 338–345, San Mateo, 1995.

Magne Jørgensen. Experience with the accuracy of software maintenance task effort prediction models. *IEEE Trans. Softw. Eng.*, 21:674–681, August 1995. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/32.403791. URL http://dx.doi.org/10.1109/32.403791.

Brian F. Joseph, F. *Understanding Open Source Software Development*. Addison Wesley, Pearson Education Book, Englewood Cliffs, Dec 2001. ISBN ISBN13: 9780201734966.

Huzefa H. Kagdi, Michael L. Collard, and Jonathan I. Maletic. A survey and taxonomy of approaches for mining software repositories in the context of software evolution. *Journal of Software Maintenance*, 19(2):77–131, 2007.

Taghi M. Khoshgoftaar and Naeem Seliya. Fault prediction modeling for software quality estimation: Comparing commonly used techniques. *Empirical Softw. Engg.*, 8(3):255–283, 2003. ISSN 1382-3256. doi: http://dx.doi.org/10.1023/A:1024424811345.

Sunghun Kim, Kai Pan, and E. E. James Whitehead, Jr. Memories of bug fixes. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, SIGSOFT '06/FSE-14, pages 35–45, New York, NY, USA, 2006a. ACM. ISBN 1-59593-468-5. doi: http://doi.acm.org/10.1145/1181775.1181781. URL http://doi.acm.org/10.1145/1181775.1181781.

Sunghun Kim, Kai Pan, and E. E. James Whitehead, Jr. Memories of bug fixes. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 35–45, New York, NY, USA, 2006b. ACM. ISBN 1-59593-468-5. doi: http://doi.acm.org/10.1145/1181775.1181781.

Sunghun Kim, E.J. Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *Software Engineering, IEEE Transactions on*, 34(2):181 –196, mar. 2008. ISSN 0098-5589. doi: 10.1109/TSE.2007.70773.

Stefan Koch. Effort modeling and programmer participation in open source software projects. *Information Economics and Policy*, 20(4):345–355, December 2008a. URL http://ideas.repec.org/a/eee/iepoli/v20y2008i4p345-355.html.

Stefan Koch. Effort modeling and programmer participation in open source software projects. *Information Economics and Policy*, 20(4):345–355, December 2008b.

A. Gunes Koru and Hongfang Liu. Building defect prediction models in practice. *IEEE Softw.*, 22:23–29, November 2005a. ISSN 0740-7459. doi: 10.1109/MS.2005.149. URL http://portal.acm.org/citation.cfm?id=1098520.1098572.

A. Gunes Koru and Hongfang Liu. Building defect prediction models in practice. *IEEE Softw.*, 22(6): 23–29, 2005b. ISSN 0740-7459. doi: http://dx.doi.org/10.1109/MS.2005.149.

S. B. Kotsiantis. Supervised machine learning: A review of classification techniques. *Informatica*, 31: 249–268, 2007.

Matti Kääriäinen, Tuomo Malinen, Tapio Elomaa, and L. Bartlett. Selective rademacher penalization and reduced error pruning of decision trees ? *Journal of Machine Learning Research*, 5:1107–1126, 2004.

Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *IEEE Trans. Softw. Eng.*, 29(9):782–795, 2003. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2003.1232284.

Michele Lanza and Stéphane Ducasse. Understanding software evolution using a combination of software visualization and software metrics.

M.L. Lee. *Change Impact Analysis of Object-Oriented Software*. PhD Thesis, George Mason University, 1998.

M. M. Lehman and J. F. Ramil. An approach to a theory of software evolution. In *IWPSE 2001: Proceedings of the 4th International Workshop on Principles of Software Evolution*, pages 70–74, New York, NY, USA, 2001. ACM. ISBN 1-58113-508-4. doi: http://doi.acm.org/10.1145/602461.602473.

B. P. Lientz and B. E. Swanson. *Software Maintenance Management*. Addison-Wesley, MA, USA, 1980.

V. Benjamin Livshits and Thomas Zimmermann. Dynamine: finding common error patterns by mining software revision histories. In *ESEC/SIGSOFT FSE*, pages 296–305, 2005.

Tom Mens and Michele Lanza. A graph-based metamodel for object-oriented software metrics. *Electronic Notes in Theoretical Computer Science*, 72(2), 2002.

Tom Mens, A Graph based Metamodel For, and Michele Lanza. A graph-based metamodel for object-oriented software metrics. In *Electronic Notes in Theoretical Computer Science*, page 72, 2002.

Audris Mockus and David M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5:169–180, 2000.

Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *ICSE '06: Proceedings of the 28th international conference on Software engineering*, pages 452–461, New York, NY, USA, 2006. ACM. ISBN 1-59593-375-1. doi: http://doi.acm.org/10.1145/1134285.1134349.

Nils J. Nilsson. *Introduction to Machine Learning.* Robotics Laboratory, Department of Computer Science, Stanford University, Stanford, CA 94305, November 03, 1998.

Niclas Ohlsson and Hans Alberg. Predicting fault-prone software modules in telephone switches. *IEEE Transactions on Software Engineering*, 22:886–894, 1996. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/32.553637.

Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Predicting the location and number of faults in large software systems. *IEEE Trans. Softw. Eng.*, 31(4):340–355, 2005. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.2005.49.

CORPORATE PDP Research Group. *Parallel distributed processing: explorations in the microstructure of cognition, vol. 1: foundations*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-68053-X.

John C. Platt. *Sequential minimal optimization: A fast algorithm for training support vector machines*. Technical Report MSR-TR-98-14, Microsoft, USA, 1998.

Adam A. Porter and Richard W. Selby. Empirically guided software development using metric-based classification trees. *IEEE Software*, 7(2):46–54, 1990.

Roger S. Pressman. *Software Engineering: A Practicioner's Approach*. 5th Edition, McGraw Hill, 2001. ISBN 0-07-365578-3.

Foster Provost and Tom Fawcett. Analysis and visualization of classifier performance: Comparison under imprecise class and cost distributions. In *In Proceedings of the Third International Conference on Knowledge Discovery and Data Mining*, pages 43–48. AAAI Press, 1997.

Foster Provost and Tom Fawcett. Robust classification for imprecise environments. *Machine Learning*, 42(3):203–231, 2001. doi: 10.1023/A:1007601015854.

L. H. Putnam. A general empirical solution to the macro software sizing and estimating problem. *IEEE Trans. Softw. Eng.*, 4(4):345–361, 1978. ISSN 0098-5589. doi: http://dx.doi.org/10.1109/TSE.1978.231521.

F. Rosenblatt. *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms.* Addison-Wesley,, York, NY, USA: Spartan Books, 1962.

Hinton G. E. Rumelhart, D. E. and R. J. Williams. Learning internal representations by error propagation. *In Rumelhart, D. E., McClelland, J. L., and the PDP Research Group, editors, Paralled Distributed Processing. Explorations in the Microstructure of Cognition.*, 1:318–362, 1986.

M. L. Sands and James R. Murphy. Use of kappa statistic in determining validity of quality filtering for meta-analysis: A case study of the health effects of electromagnetic radiation. *Journal of Clinical Epidemiology*, 49(9):1045–1051, 1996. ISSN 0895-4356. doi: DOI:10.1016/0895-4356(96)00058-3.

David Schuler and Thomas Zimmermann. Mining usage expertise from version archives. In *MSR '08: Proceedings of the 2008 international working conference on Mining software repositories*, pages 121–124, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-024-1. doi: http://doi.acm.org/10.1145/1370750.1370779.

Fabrizio Sebastiani. Machine learning in automated text categorization. *ACM Comput. Surv.*, 34(1):1–47, 2002. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/505282.505283.

Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *MSR*, 2005.

Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. Hatari: raising risk awareness. *SIGSOFT Softw. Eng. Notes*, 30:107–110, September 2005a. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/1095430.1081725. URL http://doi.acm.org/10.1145/1095430.1081725.

Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? In *MSR '05: Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, New York, NY, USA, 2005b. ACM. ISBN 1-59593-123-6. doi: http://doi.acm.org/10.1145/1083142.1083147.

Alex J. Smola and Bernhard Schölkopf. A tutorial on support vector regression. *Statistics and Computing*, 14(3):199–222, 2004. ISSN 0960-3174. doi: http://dx.doi.org/10.1023/B:STCO.0000035301.49549.88.

Ian Sommerville. *Software Engineering*. Addison-Wesley,, 2006. ISBN 9780321313799.

Qinbao Song, M. Shepperd, M. Cartwright, and C. Mair. Software defect association mining and defect correction effort prediction. *Software Engineering, IEEE Transactions on*, 32(2):69 – 82, 2006.

C. Bhattacharyya K.R.K. Murthy S.S. Keerthi, S.K. Shevade. Improvements to platt's smo algorithm for svm classifier design. *Neural Computation*, 13(3):637–649, 2001.

Hal S. Stern. Neural networks in applied statistics. *Technometrics*, 38(3):205–214, 1996. ISSN 0040-1706. doi: http://dx.doi.org/10.2307/1270601.

Stephan Diehl Andreas Zeller Thomas Zimmermann, Peter Weisgerber. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572, May 23-28, 2004. ISBN 0-8186-8779-7.

G. Tsoumakas, I. Katakis, and I. Vlahavas. Mining multi-label data, 2009. unpublished book chapter.

Davor Čubranić and Gail C. Murphy. Hipikat: recommending pertinent software development artifacts. In *Proceedings of the 25th International Conference on Software Engineering*, ICSE '03, pages 408–418, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1877-X. URL http://portal.acm.org/citation.cfm?id=776816.776866.

Sanford Weisberg. *Applied Linear Regression.* Wiley Series in Probability and Statistics,, 2005. ISBN 978-0471663799.

Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *MSR '07: Proceedings of the Fourth International Workshop on Mining Software Repositories*, page 1, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2950-X. doi: http://dx.doi.org/10.1109/MSR.2007.13.

D.R. Wilson and T.R. Martinez. The need for small learning rates on large problems. In *Neural Networks, 2001. Proceedings. IJCNN '01. International Joint Conference on*, volume 1, pages 115 –119 vol.1, 2001. doi: 10.1109/IJCNN.2001.939002.

Franz Wotawa. On the relationship between model-based debugging and program slicing. *Artificial Intelligence*, 135(1-2):125–143, 2002. doi: http://dx.doi.org/10.1016/S0004-3702(01)00161-8.

Tao Xie and Jian Pei. Mapo: mining api usages from open source repositories. In *Proceedings of the 2006 international workshop on Mining software repositories*, MSR '06, pages 54–57, 2006. ISBN 1-59593-397-2.

Liguo Yu. Indirectly predicting the maintenance effort of open-source software: Research articles. *J. Softw. Maint. Evol.*, 18(5):311–332, 2006a. ISSN 1532-060X. doi: http://dx.doi.org/10.1002/smr.v18:5.

Liguo Yu. Indirectly predicting the maintenance effort of open-source software: Research articles. *J. Softw. Maint. Evol.*, 18:311–332, September 2006b. ISSN 1532-060X. doi: 10.1002/smr.v18:5. URL http://portal.acm.org/citation.cfm?id=1165036.1165037.

Dimitrios Zeimpekis and Efstratios Gallopoulos. Tmg: A matlab toolbox for generating term-document matrices from text collections, 2005.

Du Zhang. Applying machine learning algorithms in software development. In *Proceedings of the 2000 Monterey Workshop on Modeling Software System Structures in a Fastly Moving Scenario*, pages 275–291, 2000.

Du Zhang and Jeffrey J. P. Tsai. *Machine Learning Applications In Software Engineering (Series on Software Engineering and Knowledge Engineering)*. World Scientific Publishing Company, February 28, 2005. ISBN 978-9812560940.

Min-Ling Zhang and Zhi-Hua Zhou. A k-nearest neighbor based algorithm for multi-label classification. In *GrC*, pages 718–721, 2005.

Yu Zhou, Michael Würsch, Emanuel Giger, Harald C. Gall, and Jian Lü. A bayesian network based approach for change coupling prediction. In *WCRE '08: Proceedings of the 2008 15th Working Conference on Reverse Engineering*, pages 27–36, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3429-9. doi: http://dx.doi.org/10.1109/WCRE.2008.39.

Thomas Zimmermann. Fine-grained processing of cvs archives with apfel. In *eclipse '06: Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 16–20, New York, NY, USA, 2006. ACM. ISBN 1-59593-621-1. doi: http://doi.acm.org/10.1145/1188835.1188839.

Thomas Zimmermann. Changes and bugs – mining and predicting development activities (doctoral symposium). In *In Proceedings of the 25th IEEE International Conference on Software Maintenance (ICSM 2009),*, 2009.

Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 531–540, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-079-1. doi: http://doi.acm.org/10.1145/1368088.1368161.

Thomas Zimmermann, Peter Weißgerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. In *Proceedings of the 26th International Conference on Software Engineering*, pages 563–572. IEEE Computer Society, May 2004. ISBN 0769521630.

Thomas Zimmermann, Valentin Dallmeier, Konstantin Halachev, and Andreas Zeller. erose: guiding programmers in eclipse. In *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 186–187, 2005a. ISBN 1-59593-193-7.

Thomas Zimmermann, Peter Weingerber, Stephan Diehl, and Andreas Zeller. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31:429–445, 2005b. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/TSE.2005.72.