Machine Learning Based Prediction of Complex Bugs in Source Code

Ishrat-Un-Nisa Uqaili and Syed Nadeem Ahsan
Department of Computer Science, Iqra University, Karachi

Abstract: *During software development and maintenance phases, the fixing of severe bugs are mostly very challenging and needs more efforts to fix them on a priority basis. Several research works have been performed using software metrics and predict fault-prone software module. In this paper, we propose an approach to categorize different types of bugs according to their severity and priority basis and then use them to label software metrics' data. Finally, we used labeled data to train the supervised machine learning models for the prediction of fault prone software modules. Moreover, to build an effective prediction model, we used genetic algorithm to search those sets of metrics which are highly correlated with severe bugs.*

Keywords: *Software bugs, software metrics, machine learning, fault prediction model.*

Received March 28, 2017; accepted June 8, 2017
https://doi.org/10.34028/iajit/17/1/4

1. Introduction

Software engineering literature reveals extensive interest of researchers to predict faults in software. Availability of limited resources as compared to bugs' quantity needs appropriate allocation of these resources [4, 10]. One of the important requirements of quality assurance is not only to perform code testing, but also to identify fault-prone modules as early as possible. Therefore, in recent years researchers have put more efforts to minimize the maintenance cost by developing fault prediction models.

Software repositories i.e., the databases of version controlling and bug tracking systems are being used to develop Machine Learning (ML) based fault prediction models. Bug repository and version controlling data are accumulated during the evolution of any software. Researchers use these evolutionary data to extract software metrics and apply ML techniques to predict fault prone software modules. Different types of metrics such as code, design and requirement are effectively used to predict the faulty modules. Jiang *et al*. [11] used code and design metrics data of 14 different software projects and applied different modeling techniques on the data to build fault prediction models. They discovered that code and design metrics were useful, but code metrics were more reliable than design metrics. Similarly, several researches have been conducted using software metrics to predict software bugs [8, 9, 12]. Whereas, few research works have been performed for the classification and prediction of severe bugs [15].

Bugs are expected in the software; some of them are severe in nature and should be fixed immediately. However, non-severe and low priority bugs may be delayed for resource allocation [23]. Prediction of fault-prone source code modules which can generate severe or complex bugs will help software quality assurance personnel to perform thorough testing on such fault-prone modules. Xuan *et al*. [25] concentrated the issues of prioritization and focused on developing a model to predict high priority bugs. Hall *et al*. [7] analyzed 208 studies published in eleven years of fault prediction models on the basis of source code; they identified that performance of the model depends on the selection of data, independent variables or metrics, and modeling techniques. Zimmermann *et al*. [28] revealed that the cross-project fault prediction is very important for the software having insufficient or little evolutionary data of the project. Therefore, for such projects, they proposed to build a model by using the evolutionary data of other similar projects.

In this research study, we addressed the major challenges of the software fault prediction model and proposed an approach to build ML based fault prediction model in order to predict those source code modules which can generate complex or severe bugs. Furthermore, to enhance the model's prediction capability we also addressed the major issues of ML based prediction models like feature selection, multicollinearity and class imbalance.

Our research hypothesis (H_0) is: code's metrics data labeled with already occurred bugs (like complex and ordinary bugs) are correlated, and can be used to build ML based bug prediction model. In order to validate our research hypothesis, we used a simple approach: first, we selected the metrics data of each version of software modules and labeled them with the associated bugs' types (if any). Then, we used the labeled data to train ML models for the prediction of fault prone modules which can induce severe/complex bugs.

The major contribution of our research work is the identification of those set of software metrics which can be used to build software fault prediction model, and also produce classification of software bugs into different classes including ordinary, complex, severe, and priority bugs. To validate our research objectives, we performed an experiment by using the publicly available software metrics and bug repository[1] data of these four projects: Eclipse, Pde, Mylyn, and Equinox. Our experimental data comprised of 37 metric values of each version of software modules (classes) extracted from the project's version controlling system. The data also contained the bugs' information like severity and priority, which were linked with the metrics data of those versions of software modules that actually induced those bugs. This bug information was extracted from the project's bug-tracking-system, Bugzilla/Jira. Out of 37 software metrics, 15 were change-metrics obtained from CVS Log data, 17 were source code metrics such as Chidamber and Kemerer (CK) and Object Oriented (OO), and 05 were Complexity Code Change metrics (ComCdChg). Then, we used different bug categories to label the metric data and proposed an approach to categorizing software bugs into complex (Comp) and Ordinary (Ord) bugs. In order to define bug complexity, we used different characteristics of bugs like Non-Trivial Bugs (NTB), Major Bugs (MJB), Critical Bugs (CRB), and High Priority Bugs (HPB) and Low Priority Bugs (LPB). We also pre-processed the downloaded data to handle multicollinearity and class imbalance issues using Principal Components Analysis (PCA), Genetic Search, Resample and Synthetic Minority Oversampling Technique (SMOTE). In order to design a better model we trained our data by multiple ML algorithms (Alg) and found that Complexity Code metrics were more crucial for High Priority bugs, while Chidamber and Kemerer and Object Oriented (CKOO) and Change metrics were more important for severe bugs.

In section 3 of the paper, we discussed Related Work. Section 3 describes the Data Extraction and Experimental Setup, while section 4 comprises of Results and Discussions. Finally, section 5 is the Conclusion and discusses future work.

2. Related Work

It is difficult to predict software defects reliably. Researchers have developed different prediction approaches depending on precision, complexity, and requirement of input data [5]. Software metrics are imperative for fault prediction and resource allocation in quality assurance, hence, identification of proper metrics plays important role in software projects [14]. Since, our research approach is also to identify relevant set of metrics and addresses machine learning challenges to build fault prediction model, therefore, in the following paragraphs we highlighted those research work which are more relevant to our work:

D'Ambros *et al*. [5] introduced a benchmark to allow for common comparison which provides all the data needed to apply multiple prediction techniques on 5 publicly available datasets and compared with previously available defect prediction approaches. They showed that Weighted Churn and Linearly Decayed Entropy of source code metrics are best performing techniques and single metric cannot work reliably across all systems. Different approaches have been proposed for handling the problem with the variety of metrics such as line of code and complexity (code metrics) [6, 17, 21], and the number of changes, and recent activity (process metrics) [8, 16] or previous faults [9].

Cotroneoa *et al*. [3] examined the features of the whole process of bug manifestation by studying 666 bug reports of two applications Apache web server and open source relational database management system (MySQL). Their study showed that the appearance of bug and its relation with the environment is highly important for fault removing process and its effectiveness. Shatnawi and Li [20] compared efficiency of various prediction models and proposed a model for prediction of three fault quantities such as count, fix cost and fix effort. Catel *et al*. [2] compared different ML models with Statistical models and found that ML models were better than Statistical models.

If a developer changes one of the logically coupled program files instead of all files, it may produce severe scenarios and unstable software with a bulk of errors [1]. Zimmermann *et al*. [29] categorized bugs on the basis of their reopening and discovered various factors such as metrics, people involved and their relationship which impacts the reopening of bugs. It is desirable to expose more severe concurrent bugs before the release of the software [27]. Therefore, it is essential for a software engineer to identify the severity of each problem during testing, especially when designing critical systems. It is very important for test engineers to properly recognize the severity of each issue they identify during the testing process, hence there is a need for appropriate resource allocation, scheduling of fixing activities, and additional testing [15]. Lamkanfi *et al*. [13] compared different ML algorithms and found Naive Bayes Multinomial as most suitable for bug classification in terms of accuracy and speed. They also classified severe and non-severe bugs. MySQL In real scenarios, the classification data are normally class imbalanced; one class has more training instances as compared to other class/classes. The skewed class distribution can negatively impact the classifier's performance, since classifier may be biased towards classifying new, unseen instances as belonging to the majority class. Another challenge is high dimensionality which means datasets with a huge

[1]http://bug.inf.usi.ch/index.php

number of features [15]. Several data mining methods are used to enhance the performance of classifier for imbalanced data; one of them is data sampling, which is used to generate a sampled dataset having a more balanced distribution of both classes by eliminating biased class. Then, the classifier will train the new generated and unbiased dataset [24].

3. Data Extraction and Experimental Setup

To perform experiment, data was obtained from the publicly available and published data in bug prediction datasets which stored software metrics along with defect information of several projects. We designed our model after combining data of four projects: Eclipse, Pde, Mylyn and Equinox. Each dataset comprised of various classes and versions (CVS/Subversion) as shown in Table 1. The data also includes their bug categories according to severity and priority from defect tracking systems, Bugzilla/Jira repositories [5].

Table 1. Datasets used in this study.

S#	DataSets	Instances	Versions	Description
1	**Eclipse**	998	91	Eclipse JDT Core www.eclipse.org/jdt/core/
2	**Pde**	1498	97	Eclipse PDE UI www..eclipse.org/pde/pde-ui/
3	**Mylyn**	1863	98	Eclipse MyLyn www.eclipse.org/mylyn/
4	**Equinox**	325	91	Equinox Framework www.eclipse.org/equinox/

3.1. Bug and its Categories

A bug is an error, or fault in software which produces an incorrect or unexpected result, or an unplanned behavior. The importance of a bug is described as the combination of its Priority and Severity. Severity describes the impact of a bug, whereas priority describes the importance and order in which a bug should be fixed compared to other bugs and, how it should be utilized by the programmers and engineers to prioritize their work [19]. Thung *et al*. [22] states that bug reporter can assign 5 severity levels in bug repositories: Blocker, Critical, Major, Minor and Trivial. In this study, we used following 5 bug categories according to their complexity from available bug categories in datasets as shown in Table 2.

- *Bugs Found Until (BFU)*: These bugs include all types of bugs which do not lie in any specific category.
- *Non-Trivial Bugs (NTB)*: These bugs do not impact overall functionality; they cause some undesirable behavior, but the system remains functional. The severity of NTB is higher than trivial bugs [4].
- *Major Bugs (MJB)*: These bugs affect major functionality of software or data. The severity of major bugs is higher than non-trivial bugs [4].
- *Critical Bugs (CRB)*: These bugs affect further testing. The severity of critical bugs includes critical and blocker bugs and higher than major bugs [4].
- *High Priority Bugs (HPB)*: These bugs affect the application or product critically and must be resolved as soon as possible. The priority of these bugs is greater than default priority [4].

Table 2. List of already available bugs in datasets.

S#	Bugs (Count)	Abbr	Type
1	**NumberOf BugsFoundUntil**	BFU	General
2	**NumberOfNonTrivialBugs FoundUntil**	NTB	Severity
3	**NumberOfMajorBugs FoundUntil**	MJB	Severity
4	**NumberOfCriticalBugs FoundUntil**	CRB	Severity
5	**NumberOfHighPriorityBugsFoundUntil**	HPB	Priority

3.2. Bug Classification in Terms of Data

Available bugs were further classified as clean, ordinary, complex, NTB, MJB, CRB, HPB and LPB (Low Priority Bugs) bugs. For classification of all bugs, the data was divided in two sets, described as:

1. *Whole data*: It comprises of all instances which have any type of bug. Also, those which do not have any type of bug (buggy and non-buggy/clean data).
2. *Buggy data*: It includes only those instances which have any type of bug. The overall distribution of data on the basis of bug categories is shown in Figure 1.

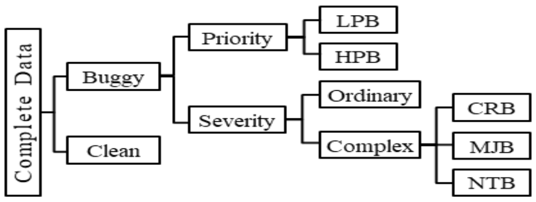


Figure 1. Distribution of data in terms of buggy and clean.

3.3. Bug Classification in Terms of Categories

The whole and buggy data sets were further classified into six groups on the basis of different class distribution of bugs, and are shown in Table 3.

3.3.1. General Bugs (All)

The whole data was classified & labeled as clean and buggy (with any type of bug) with 400 and 4280 instances respectively.

Table 3. Bugs Categories used and classified in this study.

S#	Bug Categories	Data used	Model Classification
1	**General Bugs**	Whole	Clean, Buggy
2	**Complex and Ordinary**	Whole	Clean, Complex, Ordinary
3	**Complex Categories**	Whole	Clean, Ordinary, NTB, MJB, CRB
4	**High Priority Bugs**	Buggy	LPB, HPB
5	**Complex and Ordinary**	Buggy	Complex, Ordinary
6	**Complex Categories**	Buggy	Ordinary, NTB, MJB, CRB

3.3.2. Complex and Ordinary Bugs (All)

The whole data was classified as clean, complex and ordinary. If any type of severe bug such as CRB, MJB, NTB was found, then it was labeled as Complex. If only BFU existed then it was labeled as ordinary, otherwise it was labeled as Clean. These values are defined as:

IF (NTB >0) OR (MJB >0) OR (CRB >0) then Value= "Complex"
Else IF (BFU >0) then Value="Ordinary"
Else value = "Clean"

3.3.3. Complex Categories (All)

The whole data was classified as Clean, Ordinary, NTB, MJB and CRB. If any type of severe bug such as CRB, MJB and NTB was found, then we labeled it as CRB, MJB, and NTB respectively. If only BFU found, then it was labeled as Ordinary, otherwise it was labeled as Clean. These class values are defined as:

```
IF (CRB >0) then Value= "CRB"
  Else IF (MJB >0) then Value= "MJB"
    Else IF (NTB >0) then Value= "NTB"
      Else IF (BFU > 0) then Value= "Ordinary"
Else value = "Clean"
```

3.3.4. High Priority Bugs (Buggy)

The buggy data was classified as HPB and LPB. If High Priority Bugs existed, then it was labeled as HPB, otherwise it was labeled as LPB and defined as:

IF (HPB >0) then Value= "HPB"
Else Value= "LPB"

3.3.5. Complex and Ordinary Bugs (Buggy)

The buggy data was labeled as Complex when any type of severe bug existed such as CRB, MJB, NTB, otherwise it was labeled as Ordinary and defined as:

IF (NTB >0) OR (MJB >0) OR (CRB >0) then Value= "Complex"
Else Value="Ordinary"

3.3.6. Complex Categories (Buggy)

The buggy data was classified as Ordinary, NTB, MJB, and CRB. If any type of severe bug such as CRB, MJB and NTB existed then it was labeled as CRB, MJB and NTB respectively. If only BFU existed then it was labeled as Ordinary. These class values are defined as:

```
  IF (CRB >0) then Value= "CRB"
   Else IF (MJB >0) then Value= "MJB"
     Else IF (NTB >0) then Value= "NTB"
Else Value= "Ordinary"
```

3.4. Metrics

The metrics used in this research with abbreviated names are given in the second and third columns of Table 4 respectively. The first 15 values, i.e., row 1 to 15 are change-metrics obtained from CVS log data labeled with ChgMet, the next 17 i.e., row 16 to 32 are source-code metrics CK and Object Oriented labeled with CkOO, and the last 5 i.e., row 33 to 37 are complexity-code-change labeled with ComCdChg.

Table 4. List of metrics and its correlation with bugs counts.

S#	Metrics Name	ABBR	Pearson Corr.	Significance (2-tailed)
1	**NumberOfVersionsUntil**	NVU	0.8696	0.0000000
2	**NumberOfFixesUntil**	NFU	0.5998	0.0000000
3	**NumberOfRefactoringsUntil**	NRU	0.3049	0.0000000
4	**NumberOfAuthorsUntil**	NAU	0.3622	0.0000000
5	**LinesAddedUntil**	LAU	0.6525	0.0000000
6	**MaxLinesAddedUntil**	MLAU	0.553	0.0000000
7	**AvgLinesAddedUntil**	ALAU	0.1548	0.0000000
8	**LinesRemovedUntil**	LRU	0.6087	0.0000000
9	**MaxLinesRemovedUntil**	MLRU	0.5418	0.0000000
10	**AvgLinesRemovedUntil**	ALRU	0.1638	0.0000000
11	**CodeChurnUntil**	CCU	0.6358	0.0000000
12	**MaxCodeChurnUntil**	MCCU	0.4743	0.0000000
13	**AvgCodeChurnUntil**	ACCU	0.0616	0.0000245
14	**AgeWithRespectTo**	AWR	0.1896	0.0000000
15	**WeightdAgeWithRespectTo**	WAWR	0.2012	0.0000000
16	**CouplingBetwObjectClasses**	CBO	0.4827	0.0000000
17	**DepthOfInheritanceTree**	DIT	-0.0172	**0.2385547**
18	**FanIn**	FIN	0.2895	0.0000000
19	**FanOut**	FOUT	0.5614	0.0000000
20	**LackOfCohesionInMethods**	LCOM	0.3201	0.0000000
21	**NumberOfChildren**	NOC	0.0407	0.0053559
22	**NumberOfAttributes**	NOA	0.3581	0.0000000
23	**NumberOfAttributeInherited**	NOAI	0.1099	0.0000000
24	**NumberOfLinesOfCode**	NLOC	0.5907	0.0000000
25	**NumberOfMethods**	NOM	0.5077	0.0000000
26	**NumberOfMethodsInherited**	NOMI	0.0161	**0.2706124**
27	**NumberOfPrivateAttributes**	NPRA	0.2601	0.0000000
28	**NumberOfPrivateMethods**	NPRM	0.3802	0.0000000
29	**NumberOfPublicAttributes**	NPBA	0.2665	0.0000000
30	**NumberOfPublicMethods**	NPBM	0.3784	0.0000000
31	**ResponseForaClass**	RFC	0.6051	0.0000000
32	**WeightedMethodPerClass**	WMC	0.615	0.0000000
33	**CvsEntropy**	CE	0.7244	0.0000000
34	**CvsWEntropy**	CWE	0.5962	0.0000000
35	**CvsLinEntropy**	CLINE	0.5164	0.0000000
36	**CvsLogEntropy**	CLOE	0.3335	0.0000000
37	**CvsExpEntropy**	CEXE	0.5582	0.0000000

3.5. Hypothesis Testing

To test the null hypothesis (H_0), metrics are correlated with the number of bugs and therefore can be used to build fault prediction models. We labeled the metrics data of each source file (total number of source files = 4680) with the number of bugs found. The labeled data was then used for correlation analysis. The metrics' correlation (Pearson) with bugs' count is shown in the fourth column of Table 4, which shows metrics are highly correlated with bugs' count. Moreover, the hypothesis testing (two tailed) with a significance value less than 0.01 i.e., $\alpha \leq 0.01$ is shown in the fifth column of Table 4. Since most of the metrics, i.e., 35 out of 37 have $\alpha \leq 0.01$, therefore, H_0 was accepted with 99.09% confidence. Finally, to build fault prediction model we considered only those metrics whose correlation values with bug's count were high.

3.6. Machine Learning Issues Related to Dataset

High dimensionality and class imbalance are the two main issues related to the data used to train the ML models and can degrade the performance of the prediction models. Some ML techniques can resolve these issues, i.e., high dimensionality reduced by feature selection; class imbalance issue can be addressed by data sampling and ensemble learners (Data Transformation Techniques). Both problems can be addressed by combining these techniques [18].

3.7. Data Transformation Techniques

Principal Component Analysis (PCA) and Genetic Search (GenSch) are modeling techniques used to minimize correlation problem, while SMOTE (Synthetic Minority Oversampling Technique) and Resample Filter (Res) are modeling techniques used to minimize class imbalance issue.

We pre-processed our datasets and applied different transformation techniques in combination to address both ML issues. In this way, we got 4 groups of transformed data:

a) Genetic Search-SMOTE.
b) Genetic Search-Resample.
c) PCA-SMOTE.
d) PCA-Resample.

These 4 groups of data were further tested with 6 Bug categories classified in section 3.3.

3.8. Machine Learning (ML) Algorithms

We built fault prediction models using ML Algorithms. The following 03 classifiers were used: Random Forest (RF), MultiLayer Perceptron (MLP) and Naive Bayes (NB), with 4 groups of transformed dataset as discussed in section 3.7. and 6 bug categories as discussed in section 3.3. and shown in Table 3.

3.9. Building Fault Prediction Models

For building a better fault prediction model:

1. Initially all 37 metrics were used along with 6 derived bug categories.
2. Set of highly correlated metrics with each bug category was chosen for prediction of severe bugs
3. Prediction models were improved by handling feature selection and class imbalance issues.
4. Three ML classifiers were applied to all groups of datasets after applying transformation techniques with 6 bug categories.

These four techniques were applied after combining data of all projects to adopt general model building and validation approach in fault prediction modeling with PCA, GenSch, SMOTE and Resample data transformation techniques according to their bug category, as shown in Figure 2 and summarized in the following steps.

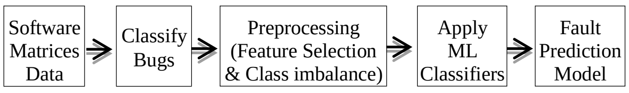


Figure 2. Fault prediction model.

3.10. Evaluation of Prediction Models

Models along with the modeling techniques were compared according to their highest values of Precision (Prec) and Recall (Rec) for all classes according to their 6 bug categories defined in section 3.3.

1. *Precision (Prec) and Recall (Rec)*: Precision and Recall rate are used for the assessment of binary prediction models. Precision is used for the assessment of positive signal predictions; and Recall rate is used for the assessment of prediction power for positive signals. In software engineering field, identifying fault is considered most crucial, therefore, Recall rate has been considered to be the most important metric [26].
2. *Bug Category*: Bug data is categorized in terms of their severity and priority as shown in Table 3.

4. Results and Discussions

4.1. Discussion based on Metrics Selection

Metrics were analyzed for different type of bugs, using Statistical Correlation and GenSch methods.

1. *Statistical Correlation Method*: It was observed that overall 21 metrics were important according to their bug types, most of them were crucial for severe bugs (10 for CRB, 14 for MJB, and 17 for NTB), 4 for priority and 18 for BFU (Gen) bugs. It was also seen that ComCdChg metrics were more crucial for HPB, while metrics CKOO, ChgMet and ComCdChg were found more important for severe bugs. On the other hand, some metrics were found only important for one category of bug as shown in Table 5.

Table 5 shows that three ComCdChg metrics (CLNE, CLGE and CEXE) and one ChgMet (NRU) were more crucial for HPB. All types of metrics (CKOO, ChgMet and ComCdChg) were more important for severe bugs. The following 14 metrics found crucial for MJB: NVU, NFU, LAU, MLAU, LRU, MLRU, CCU, FOUT, NLOC, NOM, RFC, WMC, CE, and CWE; 10 metrics such as NVU, LAU, MLAU, LRU, MLRU, NLOC, RFC, WMC, CE, and CWE were crucial for CRB; 17 metrics such as NVU, NFU, NAU, LAU, MLAU, LRU, MLRU, CCU, MCCU, CBO, FOUT, NLOC, NOM, RFC, WMC, CE, and CWE were crucial for NTB. Whereas all three types of 18 metrics were important for general bugs such as NVU, NFU, LAU, MLAU, LRU, MLRU, CCU, MCCU, CBO, FOUT, NLOC, NOM, RFC, WMC, CE, CWE, CLNE, and CEXE Only 3 ComCdChg metrics such as CLNE, CLGE and CEXE were crucial for all types of severe and high priority bugs.

Table 5. Metrics correlation by Bug Category.

S#	Metrics	Type	Severity					Priority	Severity/Priority		Gen
			NTB	MJB	CRB	AVG	Comp	HPB	Comp-v-HPB	Comp-ʌ-HPB	BFU
1	**NVU**	ChgMet	.908	.765	.649	.774	.193	.253	.166	.180	.870
2	**NFU**	ChgMet	.689	.525	.443	.552	.111	-.008	.091	.059	.600
3	**NRU**	ChgMet	.183	.115	.096	.131	.056	.456	.036	.138	.305
4	**NAU**	ChgMet	.450	.369	.315	.378	.193	-.098	.137	-.065	.362
5	**LAU**	ChgMet	.723	.612	.542	.626	.090	.078	.078	.095	.652
6	**MLAU**	ChgMet	.623	.530	.501	.551	.127	.056	.108	.078	.553
7	**ALAU**	ChgMet	.181	.155	.158	.165	.086	-.008	.067	-.014	.155
8	**LRU**	ChgMet	.676	.581	.533	.597	.082	.079	.072	.099	.609
9	**MLRU**	ChgMet	.604	.515	.501	.540	.121	.073	.104	.102	.542
10	**ALRU**	ChgMet	.184	.162	.177	.175	.080	.022	.064	.033	.164
11	**CCU**	ChgMet	.700	.554	.413	.556	.094	.047	.081	.051	.636
12	**MCCU**	ChgMet	.518	.416	.344	.426	.132	.066	.109	.028	.474
13	**ACCU**	ChgMet	.085	.066	.044	.065	.056	-.061	.040	-.096	.062
14	**AWR**	ChgMet	.274	.230	.217	.240	.082	-.134	.008	-.153	.190
15	**WAWR**	ChgMet	.243	.202	.176	.207	.086	-.028	.035	-.049	.201
16	**CBO**	CkOO	.488	.426	.365	.426	.166	.236	.146	.146	.483
17	**DIT**	CkOO	.023	.011	.028	.020	-.018	-.136	-.057	-.155	-.017
18	**FIN**	CkOO	.291	.269	.246	.269	.086	.169	.076	.104	.289
19	**FOUT**	CkOO	.575	.484	.389	.482	.206	.217	.179	.137	.561
20	**LCOM**	CkOO	.347	.278	.256	.294	.030	.032	.026	.049	.320
21	**NOC**	CkOO	.049	.050	.053	.051	.047	-.003	.040	.027	.041
22	**NOA**	CkOO	.310	.156	.079	.182	.043	.037	.039	.029	.358
23	**NOAI**	CkOO	.167	.135	.181	.161	.054	-.090	.032	-.058	.110
24	**NLOC**	CkOO	.638	.543	.465	.549	.122	.121	.111	.128	.591
25	**NOM**	CkOO	.543	.456	.390	.463	.139	.123	.125	.098	.508
26	**NOMI**	CkOO	.060	.055	.062	.059	.015	-.100	-.015	-.080	.016
27	**NPRA**	CkOO	.214	.176	.138	.176	.113	.256	.116	.118	.260
28	**NPRM**	CkOO	.392	.323	.214	.310	.096	.087	.086	.047	.380
29	**NPBA**	CkOO	.221	.086	.018	.109	.016	-.012	.013	.002	.266
30	**NPBM**	CkOO	.397	.345	.328	.357	.093	.132	.083	.115	.378
31	**RFC**	CkOO	.648	.555	.468	.557	.140	.145	.125	.122	.605
32	**WMC**	CkOO	.684	.603	.512	.600	.126	.079	.112	.116	.615
33	**CE**	ComCdChg	.719	.598	.504	.607	.299	.323	.252	.200	.724
34	**CWE**	ComCdChg	.608	.527	.450	.528	.150	.228	.132	.081	.596
35	**CLNE**	ComCdChg	.400	.305	.264	.323	.302	.612	.354	.652	.516
36	**CLGE**	ComCdChg	.226	.165	.149	.180	.241	.512	.320	.652	.333
37	**CEXE**	ComCdChg	.426	.324	.276	.342	.299	.673	.330	.629	.558

2. *Genetic Search method*: It was observed that overall 29 metrics were important, whereas most of them were crucial for severe bugs (21 for CRB, 19 for MJB, and 12 for NTB), 5 for priority and 9 for general bugs (Gen), as shown in Table 6.

Table 6 shows that 21 metrics were important for CRB such as NVU, NFU, NRU, LRU, MLRU, MCCU, WAWR, CBO, FOUT, LCOM, NOC, NOAI, NLOC, NPRM, NPBA, WMC, CE, CWE, CLNE, CLGE, and CEXE; 19 metrics were important for MJB such as NVU, NFU, LAU, MLRU, CCU, CBO, FOUT, NOA, NOAI, NLOC, NPRM, NPBA, NPBM, WMC, CE, CWE, CLNE, CLGE, and CEXE; 12 metrics were important for NTB such as NVU, NRU, LAU, AWR, WAWR, CBO, NOM, CE, CWE, CLNE, CLGE, and CEXE; 9 metrics were important for general bugs such as AWR, WAWR, FIN, NOM, CE, CWE, CLNE, CLGE, and CEXE. 5 metrics such as MCCU, CBO, CLNE, CLGE, and CEXE were important for HPB. 5 metrics were important for both Complex and HPB such as WAWR, NOMI, CLNE, CLGE, and CEXE. 12 metrics were important for any Complex bug such as NVU, NRU, LAU, AWR, WAWR, CBO, NOM, CE, CWE, CLNE, CLGE, and CEXE. 5 metrics were important for any Complex or HPB such as CBO, CE, CLNE, CLGE, and CEXE.

Table 6. Metrics selection by genetic search.

S#	Metrics	Type	Severity				Priority	Severity/Priority		Gen
			NTB	MJB	CRB	Comp	HPB	Comp-v-HPB	Comp-ʌ-HPB	BFU
1	**NVU**	ChgMet	√	√	√	√				
2	**NFU**	ChgMet		√	√					
3	**NRU**	ChgMet	√		√	√				
4	**NAU**	ChgMet								
5	**LAU**	ChgMet	√	√		√				
6	**MLAU**	ChgMet								
7	**ALAU**	ChgMet								
8	**LRU**	ChgMet			√					
9	**MLRU**	ChgMet		√	√					
10	**ALRU**	ChgMet								
11	**CCU**	ChgMet		√						
12	**MCCU**	ChgMet			√		√			
13	**ACCU**	ChgMet								
14	**AWR**	ChgMet	√			√				√
15	**WAWR**	ChgMet	√		√	√			√	√
16	**CBO**	CkOO	√	√	√	√	√	√		
17	**DIT**	CkOO								
18	**FIN**	CkOO								√
19	**FOUT**	CkOO		√	√					
20	**LCOM**	CkOO			√					
21	**NOC**	CkOO			√					
22	**NOA**	CkOO		√						
23	**NOAI**	CkOO		√	√					
24	**NLOC**	CkOO		√	√					
25	**NOM**	CkOO	√			√				√
26	**NOMI**	CkOO							√	
27	**NPRA**	CkOO								
28	**NPRM**	CkOO		√	√					
29	**NPBA**	CkOO		√	√					
30	**NPBM**	CkOO		√						
31	**RFC**	CkOO								
32	**WMC**	CkOO		√	√					
33	**CE**	ComCdChg	√	√	√	√		√		√
34	**CWE**	ComCdChg	√	√	√	√				√
35	**CLNE**	ComCdChg	√	√	√	√	√	√	√	√
36	**CLGE**	ComCdChg	√	√	√	√	√	√	√	√
37	**CEXE**	ComCdChg	√	√	√	√	√	√	√	√

4.2. Discussion according to ML Techniques and Classifiers

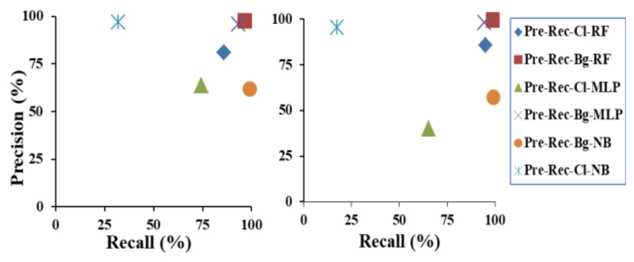
Results of all datasets were improved when using class imbalance techniques, i.e., Resample and SMOTE in combination with feature selection techniques, i.e. GenSch and PCA. Best results were achieved with Complex bugs. The results were also improved with NTB, MJB and HPB and with those bugs which were both Complex as well as HPB. Detail of results for all categories of bugs is discussed as under:

4.2.1. Experimental Results Obtained Using Dataset for General Bugs

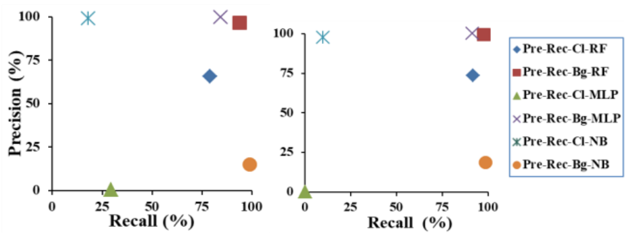
There were two classes: clean and buggy. The results of the buggy class were best classified for all ML classifiers except NB, whereas the results of dataset for both classes were improved by using feature selection techniques, i.e., GenSch and PCA with class imbalance techniques, i.e., SMOTE and Resample as displayed in Table 7 and Figure 3.

Table 7. Precision/Recall values (in %) of all datasets according to the general bugs.

ML Alg	Class	GenSch-SMOTE		GenSch-Res		PCA-SMOTE		PCA-Res	
		Prec	Rec	Prec	Rec	Prec	Rec	Prec	Rec
RF	CLEAN	85.9	81	95.1	85.6	79	65.8	91.7	73.5
	BUGGY	96.5	97.5	98.7	99.6	93.8	96.7	97.6	99.4
MLP	CLEAN	74.3	63.6	65.4	39.9	29.4	0.6	0	0
	BUGGY	93.4	95.9	94.5	98	84.3	99.7	91.6	100
NB	CLEAN	32.1	96.9	17.4	95.3	17.9	99.1	9.9	97.5
	BUGGY	99.1	61.7	99.2	57.2	98.9	15.1	98.8	18.6



a) Genetic search-SMOTE. b) Genetic search-resample.



c) PCA-SMOTE. d) PCA-resampl.

Figure 3. Precision/recall according to the general bugs.

Approximately 90% precision/recall values were produced when RF was applied after using GenSch with Resample. 80% precision/recall values were produced when RF was applied on dataset after using GenSch with SMOTE. 70% results were produced when RF was applied on dataset after using PCA with Resample and PCA with SMOTE. 60% precision/recall values were produced when MLP was applied on dataset after using GenSch with SMOTE.

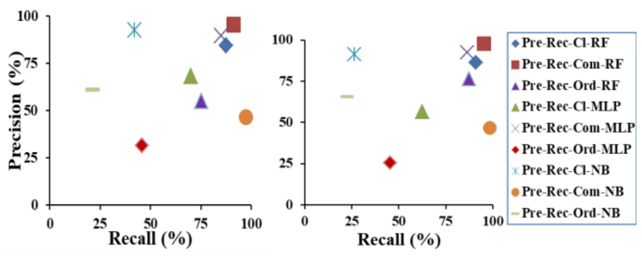
4.2.2. Experimental Results Obtained Using Dataset for Complex and Ordinary Bugs (All)

There were three classes: Clean, Ordinary and Complex. The Complex class was best classified with all ML classifiers after using modeling techniques, whereas clean class was better classified than ordinary. Results of dataset for all classes were improved by using feature selection technique, i.e. GenSch in combination with class imbalance technique, Resample as displayed in Table 8 and Figure 4.

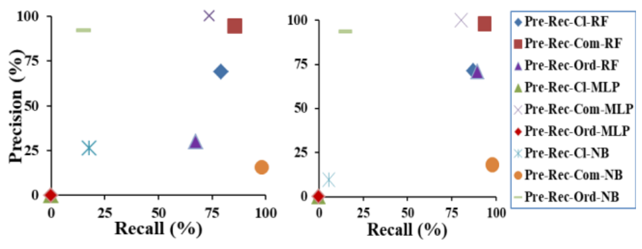
Approximately 80% precision/recall values were obtained when RF was applied on dataset after using GenSch with Resample. 70% precision/recall values were produced when RF was applied on dataset after using PCA with Resample. 60% precision/recall values were produced when RF was applied on dataset after using GenSch with SMOTE.

Table 8. Precision/recall values (in %) of all datasets according to complex and ordinary bugs.

ML Alg	Class	GenSch-SMOTE		GenSch-Res		PCA-SMOTE		PCA-Res	
		Prec	Rec	Prec	Rec	Prec	Rec	Prec	Rec
RF	CLEAN	87.5	84.5	90.9	86.3	79.3	69.1	87.2	71.2
	COMPLEX	91.2	95.3	95.4	97.6	85.5	94.6	93.4	97.9
	ORDINARY	75.3	55.2	87.3	76.5	67.4	29.9	89.5	71
MLP	CLEAN	70.1	68.4	62.4	56.5	0	0	0	0
	COMPLEX	85.1	89.4	86.3	92.5	73.8	100	80.3	100
	ORDINARY	45.9	31.5	45.6	25.6	0	0	0	0
NB	CLEAN	42	92.5	26.5	91.3	17.6	26.3	5.5	9.7
	COMPLEX	97.5	46.5	98.3	46.6	98	15.9	97.8	18
	ORDINARY	21.2	60.8	22.5	65.7	14.9	91.9	15	93.6



a) Genetic search-SMOTE. b) Genetic search-resample.



c) PCA-SMOTE. d) PCA-resample.

Figure 4. Precision/recall according to complex and ordinary bugs.

- *Clean Class*: Clean class was best classified with all ML classifiers except NB classifier after using all modeling techniques.

Approximately 90% precision/recall values were produced when RF was applied on dataset after using GenSch with Resample. 80% precision/recall values were produced when RF was applied on dataset after using GenSch with SMOTE. 70% precision/recall values were produced when RF was applied after using PCA with SMOTE and Resample, and MLP was applied on dataset after using GenSch with Resample.

- *Complex Class*: Complex class was best classified with all ML classifiers except NB after using all modeling techniques.

Approximately 90% precision/recall values were produced when RF was applied on dataset after using all modeling techniques, and MLP was applied on dataset after using GenSch with SMOTE and GenSch with Resample. 80% precision/recall values were produced when MLP was applied after using PCA with Resample.

- *Ordinary Class*: Approx. 80% precision/recall values were produced when RF was applied on dataset after using GenSch with Resample technique. 70% precision/recall values were produced when RF was applied on dataset after using PCA with Resample.

4.2.3. Experimental Results Obtained Using Dataset for Complex Categories (All)

There were five classes clean, ordinary, NTB, MJB and CRB. Results of dataset were improved by using feature selection techniques, i.e., GenSch and PCA, in combination with class imbalance technique, i.e., Resample as displayed in Table 9 and Figure 5.

Approximately 70% precision/recall values were produced when RF was applied on dataset after using GenSch and PCA with Resample.

- *Clean Class*: Approximately 90% precision/recall values were produced when RF was applied on dataset after using GenSch with SMOTE. 70% precision/recall values were produced when RF was applied on dataset after using PCA with SMOTE and Resample.
- *Ordinary Class*: Approximately 80% precision/recall values were produced when RF was applied on dataset after using GenSch with Resample. 70% precision/recall values were produced when RF was applied on dataset after using PCA with Resample. 60% precision/recall values were produced when RF was applied on dataset after using GenSch with SMOTE.
- *NTB Class*: Approximately 90% precision/recall values were produced when RF was applied on dataset after using GenSch with Resample. 80% precision/recall values were produced when RF was applied on dataset after using PCA with Resample. 70% precision/recall values were produced when RF was applied on dataset after using GenSch and PCA with SMOTE, and MLP was applied after applying GenSch with Resample. 60% precision/recall values were produced when MLP was applied after using GenSch and PCA with SMOTE, and PCA with Resample.
- *MJB Class*: Approximately 70% precision/recall values were produced when RF was applied on dataset after using GenSch and PCA with Resample.
- *CRB Class*: Approximately 80% precision/recall values were produced when RF was applied on dataset after using GenSch and PCA with Resample.

Table 9. Precision/recall values (in %) of all datasets according to complex categories and ordinary bugs.

ML Alg	Class	GenSch-SMOTE		GenSch-Res		PCA-SMOTE		PCA-Res	
		Prec	Rec	Prec	Rec	Prec	Rec	Prec	Rec
RF	CLEAN	86.5	86.6	90.1	86	75.2	73.4	83.9	72.8
	ORDINARY	73.8	57.8	90.9	81	61.8	34.1	89.8	73.2
	NTB	74.5	90	87.5	96	66.8	86.3	84.1	94.7
	MJB	51.1	22.1	88.3	69.4	32.1	9.5	86.1	65.9
	CRB	69.3	48.9	89.9	78.8	59	40.5	86	77.3
MLP	CLEAN	4.7	71.7	60.1	54.7	40.7	3	0	0
	ORDINARY	3.8	39	35	29	0	0	0	0
	NTB	55.8	64	66.1	85	55.4	97.7	59.9	97.9
	MJB	1.6	38.3	43.8	14.8	50	0.6	29.1	2.8
	CRB	2.7	59.9	63.2	31	65.7	30.3	62.3	31.8
NB	CLEAN	43.6	93	28.2	92.4	42.6	10.1	21	67.7
	ORDINARY	23.6	59.9	22.6	58.2	15.3	91.7	22.1	88.5
	NTB	67.5	34	65.8	34.5	56.7	27.9	58.2	19.4
	MJB	26.1	22.5	28.3	21.7	22.1	10.4	30.6	12.3
	CRB	64.7	25.8	62.9	26.3	71.2	16.4	65.3	22.5

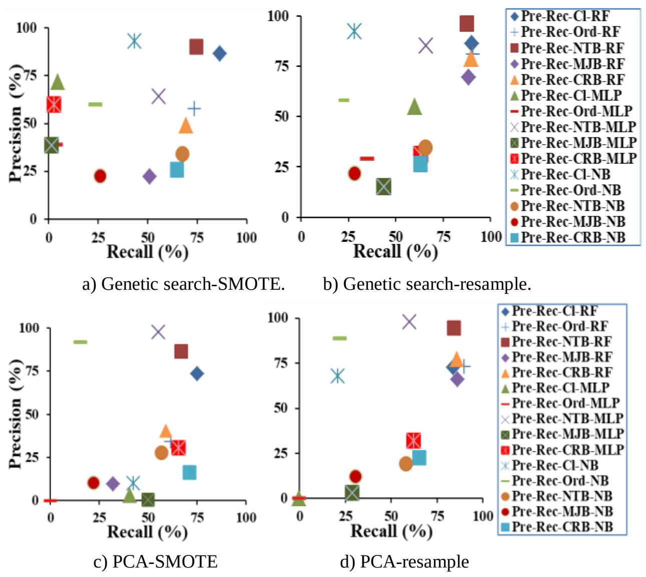


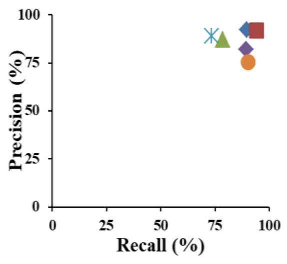
Figure 5. Precision/Recall according to complex categories and ordinary bug.

4.2.4. Experimental Results obtained using Dataset for High Priority Bugs (Buggy)

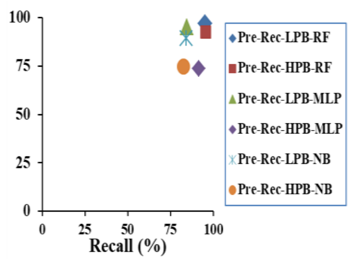
Only buggy data was selected and classified as LPB and HPB. Results of dataset were improved by using feature selection techniques, i.e., GenSch and PCA in combination with class imbalance techniques, i.e. SMOTE and Resample as displayed in Table 10 and Figure 6.

Table 10. Precision/Recall values (in %) of All Datasets according to HPB Bugs in Buggy Data.

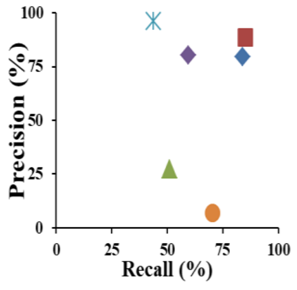
ML Alg	Class	GenSch-SMOTE		GenSch-Res		PCA-SMOTE		PCA-Res	
		Prec	Rec	Prec	Rec	Prec	Rec	Prec	Rec
RF	LPB	89.5	92.4	95.1	97	84	79.5	91.7	94.5
	HPB	94.1	91.8	95.3	92.5	85.2	88.6	91.3	87.3
MLP	LPB	78.4	87.1	84.3	95.3	51	27.4	60.4	97.7
	HPB	89.4	81.9	91.3	73.6	59.5	80.2	56.1	4.3
NB	LPB	73.3	88.9	84.1	89.2	43.7	96.2	60.6	96.6
	HPB	90.1	75.6	82.3	74.8	70.3	6.8	54.9	6.2



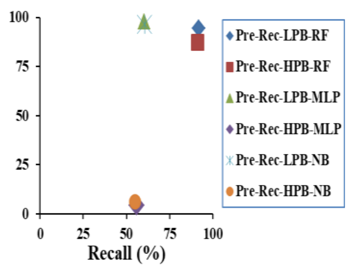
a) Genetic search-SMOTE.



b) Genetic search-resample.



c) PCA-SMOTE.



d) PCA-resample.

Figure 6. Precision/Recall according to HPB in buggy data.

Approximately 90% precision/recall values were produced when RF was applied on dataset after using GenSch with SMOTE and Res, and PCA with Resample. 80% precision/recall values were produced when RF was applied on dataset after using PCA with Resample, and MLP was applied after using GenSch with SMOTE. 70% precision/recall values were produced when MLP was applied after using GenSch with Resample and NB was applied after using GenSch with SMOTE and Resample.

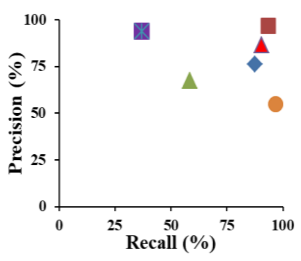
4.2.5. Experimental Results Obtained using Complex and Ordinary Bugs Dataset (Buggy)

Only buggy data was selected and classified as Complex and Ordinary classes. Results of all datasets were improved by using feature selection technique, i.e., GenSch with class imbalance technique, i.e., Resample as displayed in Table 11 and Figure 7.

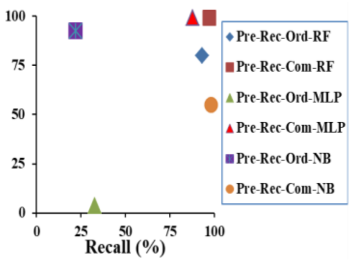
Approximately 80% precision/recall values were produced when RF was applied on dataset after using GenSch with Resample and SMOTE. 70% precision/recall values were produced when RF was applied on dataset after using PCA with Resample. 60% precision/recall values were produced when RF was applied on dataset after using PCA with SMOTE, and MLP was applied after using GenSch with SMOTE.

Table 11. Precision/recall values (in %) according to complex and ordinary bugs in buggy data.

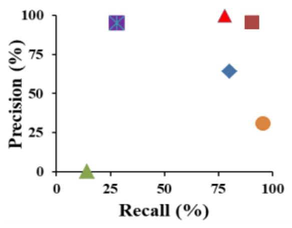
ML Alg	Class	GenSch-SMOTE		GenSch-Res		PCA-SMOTE		PCA-Res	
		Prec	Rec	Prec	Rec	Prec	Rec	Prec	Rec
RF	ORDINARY	87.4	76.3	93.1	80	80.3	64	92.4	71.9
	COMPLEX	93.5	96.9	97.3	99.2	90.4	95.5	96.2	99.2
MLP	ORDINARY	58.4	67.4	32.8	3.6	14.3	0.1	0	0
	COMPLEX	90.4	86.4	88	99	77.9	99.8	100	87.7
NB	ORDINARY	37	93.6	22.3	92.4	28	95	16.4	94.3
	COMPLEX	96.8	54.9	98.1	55	95.6	30.9	97.6	32.8



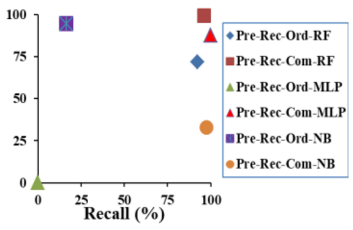
a) Genetic search-SMOTE.



b) Genetic search-resample.



c) PCA-SMOTE.



d) PCA-resampl.

Figure 7. Precision/Recall according to complex and ordinary bugs in buggy data.

- *Ordinary Class*: Approx. 80% precision/recall values were produced when RF was applied on dataset after using GenSch with Resample and SMOTE. 70% precision/recall values were produced when RF was applied on dataset after using PCA with Resample. 60% precision/recall values were produced when RF was applied on dataset after using PCA with SMOTE, and MLP applied on dataset after using GenSch with SMOTE.
- *Complex Class*: Approximately 90% precision/recall values were produced when RF was applied on dataset after using all modeling techniques. MLP was applied after using GenSch with SMOTE and Resample and, PCA with Resample. 80% precision/recall values were produced when MLP was applied after using PCA with SMOTE. 60% precision/recall values were produced when NB was applied on dataset after using GenSch with Resample.

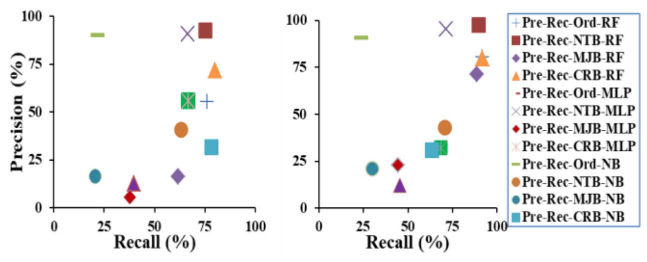
4.2.6. Experimental Results Obtained using Dataset for Complex Categories (Buggy)

Only buggy data was selected and classified in four classes Ordinary, NTB, MJB and CRB. Results of all datasets were improved by using feature selection techniques, i.e., GenSch and PCA with class imbalance technique, i.e., Resample as displayed in Table 12 and Figure 8.

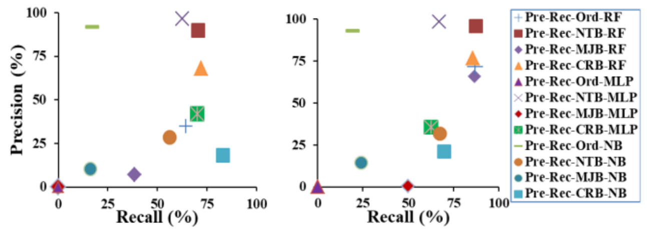
Approximately 70% precision/recall values were produced when RF was applied on dataset after using GenSch with Resample, and PCA with Resample.

Table 12. Precision/Recall values (in %) according to Complex Categories and Ordinary Bugs in Buggy Data.

ML Alg	Class	GenSch-SMOTE		GenSch-Res		PCA-SMOTE		PCA-Res	
		Prec	Rec	Prec	Rec	Prec	Rec	Prec	Rec
RF	ORDINARY	76.2	55.4	91.7	80.2	64.6	35	86.9	71.7
	NTB	75.1	92.8	89.6	97.5	70.7	89.8	87.3	96
	MJB	61.7	16.4	88.8	71.2	38.4	7.2	86.5	66
	CRB	80	71.7	92	79.5	72.1	68	85.7	76.7
MLP	ORDINARY	39.6	12.6	45.4	12.2	0	0	0	0
	NTB	66.5	90.7	71.3	95.3	62.5	96.6	67	98.5
	MJB	37.7	5.5	44.4	22.9	0	0	50	0.4
	CRB	66.8	55.8	68.5	32.1	70.3	41.7	62.7	35.5
NB	ORDINARY	21.8	90.2	23.6	90.5	17.1	91.9	19.4	93
	NTB	63.1	40.9	70.8	42.8	56.1	28.6	67.4	32
	MJB	20.5	16.3	29.9	21.1	16.1	10.4	24.1	14.5
	CRB	78.2	31.5	63.6	31	83	18.2	69.7	21.3



a) Genetic search-SMOTE. b) Genetic search-resample.



c) PCA-SMOTE. d) PCA-resample.

Figure 8. Precision/Recall according to complex categories and ordinary bugs in buggy data.

- *Ordinary Class*: Approx. 80% precision/recall values were produced when RF was applied on dataset after using GenSch with Resample. 70% precision/ recall values were produced when RF was applied on dataset after using PCA with Resample. 60% precision/recall values were produced when RF was applied on dataset after using GenSch with SMOTE.
- *NTB Class*: Approximately 90% precision/recall values were produced when RF was applied on dataset after using GenSch and PCA with Resample. 80% precision/recall values were produced when RF was applied on dataset after using GenSch with SMOTE. 70% precision/recall values were produced when RF was applied on dataset after using PCA with SMOTE, and MLP was applied on dataset after using GenSch with Resample and SMOTE, and PCA with SMOTE. 60% precision/recall values were produced when MLP was applied on dataset after using PCA with SMOTE.
- *MJB Class*: Approximately 70% precision/recall values were produced when RF was applied on dataset after using GenSch and PCA with Resample.
- *CRB Class*: Approx. 80% precision/recall values were produced when RF was applied on dataset after using GenSch and PCA with Resample. 70% precision/recall values were produced when RF was applied on dataset after using GenSch and PCA with SMOTE. 60% precision/recall values were produced when MLP was applied on dataset after using GenSch with SMOTE modeling techniques.

5. Conclusions

Prediction of software defect provides a list of defect-prone code modules which need thorough testing. Software metrics (e.g., process metrics and product metrics) and machine learning techniques are commonly used to develop automatic software fault (bug) prediction models. Since fixing of complex faults consumes more resources, therefore, in this paper, we used a specific set of metrics to construct machine learning based fault prediction models to predict different types of bugs. Our proposed models can be used to predict more precisely those source codes' modules which could generate complex bugs.
Our research revealed that complexity code Change Metrics (ComCdChg), Change Metrics (ChgMet), and Chidamber and Kemerer and object-oriented (CKOO) metrics are crucial to predicting those source codes which may induce high priority and severe bugs. Overall five metrics such as MCCU, CBU, CLNE, CLGE, and CEXE were found to be more crucial for severe and high priority bugs.

It was also observed that after dealing with the well-known class imbalance and feature selection issues of machine learning, the overall rate of model's precision and recall were improved (to above 90%). Our experimental results revealed that the supervised ML technique, i.e., Random-Forest outclassed the other techniques of machine learning. In order to further enhance the prediction capability of our proposed fault prediction models in future, the metrics data of other projects can be used to build the fault prediction models, and additional metrics related to software project, product and process could also be considered.

References

[1] Ahsan S. and Wotawa F., "Fault Prediction Capability of Program File's Logical-Coupling Metrics," *in Proceedings of Software Measurement, Joint Conference of the 21st Int'l Workshop on and 6th Int'l Conference on Software Process and Product Measurement*, Nara, pp. 257-262, 2011.

[2] Catala C. and Diri B., "A Systematic Review of Software Fault Prediction Studies," *Expert Systems with Applications*, vol. 36, no. 4, pp. 7346-7354, 2009.

[3] Cotroneoa D., Pietrantuono R., Russo S., and Trivedi K., "How Do Bugs Surface? A Comprehensive Study on The Characteristics of," *Journal of Systems and Software*, vol. 113, pp. 27-43, 2016.

[4] D'Ambros M., Lanza M., and Robbe R., "Evaluating Defect Prediction Approaches," *Empirical Software Engineering, An International Journal*, vol. 17, no. 4-5, pp. 531-577, 2012.

[5] D'Ambros M., Lanza M., and Robbe R., "An Extensive Comparison of Bug Prediction Approaches," *in Proceedings of 7th IEEE Working Conference on Mining Software Repositories*, Cape Town, pp. 31-41, 2010.

[6] Gyimothy T., Ferenc R., and Siket I., "Empirical Validation Of Object-Oriented Metrics on Open Source Software for Fault Prediction," *IEEE Transactions on Software Engineering (IEEE Computer Society)*, vol. 31, no. 10, pp. 897-910, 2005.

[7] Hall T., Beecham S., Bowes D., Gray D., and Counsell S., "A Systematic Literature Review on Fault Prediction Performance in Software Engineering," *Software Engineering, IEEE Transactions (IEEE Computer Society)*, vol. 38, no. 6, pp. 1276-1304, 2012.

[8] Hassan A., "Predicting Faults Using the Complexity of Code Changes," *in Proceedings of the 31st International Conference on Software Engineering*, Vancouver, pp. 78-88, 2009.

[9] Hassan A. and Holt R., "The Top Ten List: Dynamic Fault Prediction, " *in Proceedings of the 21st IEEE International Conference on Software Maintenance*, Budapest, pp. 263-272, 2005.

[10] Jeon C., Kim N., and In H., "A Probabilistic Approach to Building Defect Prediction Model for Platform-based Product Lines," *The International Arab Journal of Information Technology*, vol. 14, no. pp. 413-422, 2017.

[11] Jiang Y., Cukic B., Menzies T., and Lin J., "Incremental Development of Fault Prediction Models," *International Journal of Software Engineering and Knowledge Engineering (World Scientic Publishing Company)*, vol. 23, no. 10, pp. 1399-1425, 2013.

[12] Kamei Y. and Shihab E., "Defect Prediction: Accomplishments and Future Challenges," *in Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering*, Suita, pp. 33-45, 2016.

[13] Lamkanfi A., Demeyer S., Soetens Q., and Verdonck T., "Comparing Mining Algorithms for Predicting the Severity of a Reported Bug," *in Proceedings of the 15th European Conference on Software Maintenance and Reengineering*, Oldenburg, pp. 249-258, 2011.

[14] Madeyski L. and Jureczko M., "Which Process Metrics Can Significantly Improve Defect Prediction Models? An Empirical Study," *Software Quality Journal*, vol. 23, no. 3, pp. 393-422, 2015.

[15] Menzies T. and Marcus A., "Automated Severity Assessment of Software Defect Reports," *in Proceedings of IEEE International Conference on Software Maintenance, ICSM. Software Maintenance*, Beijing, pp. 346-355, 2008.

[16] Nagappan N. and Ball T., "Use of Relative Code Churn Measures to Predict System Defect Density," *in Proceedings of the 27th International Conference on Software Engineering*, St. Louis, pp. 284-292, 2005.

[17] Nagappan N., Ball T., and Zeller A., "Mining Metrics to Predict Component Failures," *in Proceedings of the 28th International Conference on Software Engineering*, Shanghai, pp. 452-461, 2006.

[18] Prusa J., Khoshgoftaar T., and Seliya N., "Enhancing Ensemble Learners with Data Sampling on High-Dimensional Imbalanced Tweet Sentiment Data," *in Proceedings of the 29th International Flairs Conference*, Key Largo, pp. 322-327, 2016.

[19] Sharma M., Kumari M., and Singh V., "Understanding the Meaning of Bug Attributes and Prediction Models," *in Proceedings of the 5th IBM Collaborative Academia Research Exchange Workshop*, New Delhi, 2013.

[20] Shatnawi R. and Li W., "An Empirical Investigation of Predicting Fault Count, Fix Cost and Effort Using Software Metrics," *International Journal of Advanced Computer Science and Applications*, vol. 7, no. 2, pp. 484-491, 2016.

[21] Subramanyam R. and Krishnan M., "Empirical Analysis of Ck Metrics for Object-Oriented Design Complexity: Implications for Software Defects," *IEEE Transactions on Software Engineering (IEEE Computer Society)*, vol. 29, no. 4, pp. 297-310, 2003.

[22] Thung F., Wang S., Lo D., and Jiang L., "An Empirical Study of Bugs in Machine Learning Systems," *in Proceedings of 23rd International Symposium on Software Reliability Engineering*, Dallas, pp. 271-280, 2012.

[23] Tian Y., Lo D., and Sun C., "Information Retrieval Based Nearest Neighbor Classification for Fine-Grained Bug Severity Prediction," *in Proceedings of 19th Working Conference on Reverse Engineering*, Kingston, pp. 215-224, 2012.

[24] Van Hulse J., Khoshgoftaar T., and Napolitano A., "Experimental Perspectives on Learning From Imbalanced Data. " *in Proceedings of the 24th International Conference on Machine*

Learning, Corvalis, pp. 935-942, 2007.

[25] Xuan J., Jiang H., Ren Z., and Zou W., “Developer Prioritization in Bug Repositories,” *in Proceedings of the 34th International Conference on Software Engineering*, Zurich, pp. 25-35, 2012.

[26] Yu L. and Mishra A., “Experience in Predicting Fault-Prone Software Modules Using Complexity Metrics,” *Quality Technology and Quantitative Management*, vol. 9, no. 4, pp. 421-433, 2012.

[27] Zhang W., Sun C., and Lu S., “ConMem: Detecting Severe Concurrency Bugs through an Effect-Oriented Approach,” *in Proceedings of the 15th Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, Pittsburgh, pp. 179-192, 2010.

[28] Zimmermann T., Nagappan N., Gall H., Giger E., and Murphy B., “Cross-Project Defect Prediction: A Large Scale Experiment on Data Vs. Domain Vs. Process,” *in Proceedings 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Amsterdam, pp. 91-100, 2009.

[29] Zimmermann T., Nagappan N., Guo P., and Murphy B., “Characterizing and Predicting Which Bugs Get Reopened, ” *in Proceedings 34th International Conference on Software Engineering*, Zurich, pp. 1074-1083, 2012.

Ishrat-Un-Nisa Uqaili is a Final year student of M.S (Computer Science) at Faculty of Engineering Science and Technology (FEST), Iqra University (IU), Defence View (Main Campus), Shaheed-e-Millat Road (Ext.) Karachi-75500, Pakistan. She did B.E in Computer Systems from Mehran University of Engineering & Technology, Jamshoro, Pakistan. Her research interests include machine learning application in Software Engineering, and build models for automatic software maintenance. She has recently completed her MS final year thesis on Fault Prediction Model for Software using Soft Computing Techniques.

Syed Nadeem Ahsan did his Ph.D. in Computer Science from GRAZ University of Technology, Austria. Currently, he is doing R&D work in software engineering and machine learning applications, and also associated with FEST, IU, Main Campus, Karachi. His Research interest includes software maintenance & evolution, software testing, formal methods in software engineering, modeling and simulation of complex system, and computational intelligence.