4. CLASSIFIER MINING OF SOFTWARE DEFECT PREDICTION

4.1 INTRODUCTION

Though there has been a rapid growth in software development, owing to various reasons, software comes with many defects. In Software development process, testing of software is the main phase which reduces the defects of the software. If a developer or a tester can predict the software defects properly, it will reduce the cost, time and effort. We proposed to undertake a comparative analysis of software defect prediction based on classification rule mining. We developed a scheme for this purpose and chose different classification algorithms for comparison. This evaluation analyzes the prediction performance of competing learning schemes for given historical data sets (NASA MDP Data Set). The result of this scheme evaluation shows that we have to choose different classifier rule for different data set.

The content of this chapter is published in Journal of Computer Science and Engineering (3785 – 0855), P32-45, Volume 2, Issue 3, April 2015, The reprint is appended towards the end of this thesis.

To improve the software productivity and quality, software engineers are applying data mining algorithms to various SE tasks. Many algorithms can help engineers figure out how to invoke API methods provided by a complex library or framework with insufficient documentation. In terms of maintenance, such type of data mining algorithms can assist in determining what code locations must be changed when another code location is changed. Software engineers can also use data mining algorithms to hunt for potential bugs that can cause future in-field failures as well as identify buggy lines of code (LOC) responsible for already-known failures. The second and third columns of Table 4.1 list several example data mining algorithms and the SE tasks to which engineers apply by Tao Xie (2009).
4.2 PROPOSED SCHEME

Overview of the framework before building defect prediction model and using them for prediction purposes, we first need to decide which learning scheme or learning algorithm should be used to construct the model. Thus, the predictive performance of the learning scheme should be determined, especially for future data. However, this step is often neglected and so the resultant prediction model may not be Reliable. As a consequence, we use a software defect prediction framework that provides guidance to address these potential shortcomings.

The framework consists of two components:

i) Scheme evaluation, and

ii) Defect prediction.

Figure 4.1 contains the details. At the scheme evaluation stage, the performances of the different learning schemes are evaluated with historical data to determine whether a certain learning scheme performs sufficiently well for prediction purposes or to select the best from a set of competing schemes.
From Figure 4.1, we can see that the historical data are divided into two parts: a training set for building learners with the given learning schemes, and a test set for evaluating the performances of the learners, it is very important that the test data are not used in any way to build the learners. This is a necessary condition to assess the generalization ability of a learner that is built according to a learning scheme and to further determine whether or not to apply the learning scheme or select one best scheme from the given schemes.

At the defect prediction stage, according to the performance report of the first stage, a learning scheme is selected and used to build a prediction model and predict software defect. From Figure 4.1, we observe that all of the historical data are used to build the predictor here. This is very different from the first stage; it is very useful for improving the generalization ability of the predictor. After the predictor is built, it can be used to predict the defect-proneness of new software components.

MGF proposed by Tim Menzies (2007) a baseline experiment and reported the performance of the Naive Bayes data miner with log- filtering as well as attribute selection, which performed the scheme evaluation but with in appropriate data. This is because they used both the training (which can be viewed as historical data) and test (which can be viewed as new data) data to rank attributes, while the labels of the new data are unavailable when choosing attributes in practice.

4.3 SCHEME EVALUATION

The scheme evaluation is a fundamental part of the software defect prediction framework. At this stage, different learning schemes are evaluated by building and evaluating learners with them. The first problem of scheme evaluation is how to divide historical data into training and test data. As mentioned above, the test data should be independent of the learner construction. This is a necessary precondition to evaluate the performance of a learner for new data. Cross-validations usually used to estimate how accurately a predictive model will perform in practice. One round of cross-validation involves partitioning a data set into complementary subsets, performing the analysis on one subset, and validating the analysis on the other subset. To reduce variability, multiple rounds of cross-validation are performed using different partitions, and the validation results are averaged over the rounds.

In our framework, a percentage split used for estimating the performance of each predictive model, that is, each data set is first divided into 2 parts, and after that a
predictor is learned on 60% instances, and then tested on the remaining 40%. To overcome any ordering effect and to achieve reliable statistics, each holdout experiment is also repeated M times and in each repetition the data sets are randomized. So overall, \( M \times N \) (N=Data sets) models are built in all during the period of evaluation; thus \( M \times N \) results are obtained on each data set about the performance of the each learning scheme.

After the training-test splitting is done each round, both the training data and learning scheme(s) are used to build a learner. A learning scheme consists of a data preprocessing method, an attribute selection method, and a learning algorithm.

Evaluation of the proposed framework is comprised of:

1. A data preprocessor
   - The training data are preprocessed, such as removing outliers, missing values, and discretizing or transforming numeric attributes using NASA preprocessing tool.

2. An attribute selector
   - User can select all or any of the attributes provided by the NASA MDP Data Set.

3. Learning Algorithms
   - NaiveBayse Simple from bayse classification
   - Logistic classification
   - From Rule based classification
   - Decision Table
   - OneR
   - JRip
   - PART
   - From Tree based classification
   - J48
   - J48Graft

4.4 SCHEME EVALUATION ALGORITHM

Data: Historical Data Set
Result: The mean performance values
1 M=12: No of Data Set
2 i=1;
3 while i<=M do
4 Read Historical Data Set D(i);
5 Split Data set Instances using % split;
6 Train(i)=60% of D; % Training Data;
7 Learning(Train(i),scheme);
8 Test Data=D(i)-Train(i);% Test Data;
9 Result=TestClassifier(Test(i),Learner);
10 end

Algorithm 4.1 Scheme Evaluation
4.5 DEFECT PREDICTION

The defect prediction part of our framework is straightforward; it consists of predictor construction and defect prediction. During the period of the predictor construction:

i) A learning scheme is chosen according to the performance report.

ii) A predictor is built with the selected learning scheme and the whole historical data. While evaluating a learning scheme, a learner is built with the data and tested on the test data. Its final performance is the mean over rounds. This reveals that the evaluation indeed covers all the data. Therefore as we use all of the historical data to build the predictor, it is expected that constructed predictor has stronger generalization ability.

iii) After the predictor is built, new data are preprocessed in the same way as historical data, then the constructed predictor can be used to predict software defect with preprocessed new data.

4.6 DIFFERENCE BETWEEN PROPOSED FRAMEWORK AND OTHERS

Thus, to summarize, the main difference between our framework and that of others in the following:

i) We choose the entire learning scheme, not just one out of the learning algorithm, attribute selector, or data preprocessor

ii) We use the appropriate data to evaluate the performance of a scheme NASA MDP Data Set by Martin Shepperd (2013)

iii) We choose percentage split for training data set (60%) and test dataset (40%)

Data Set

We used the data taken from the public NASA MDP repository, which was also used by MGF and many others, e.g., Stefan Lessmann (2008). Thus, there are 12 data sets in total from NASA MDP repository. Table 4.1 and Table 4.2 provides some basic summary information. Each data set is comprised of a number of software modules (cases), each containing the corresponding number of defects and various software static code attributes. After preprocessing, modules that contain one or more defects were labeled as defective. A very detailed description of code attributes or the origin of the MDP data sets can be obtained from Tim Menzies (2007).
### Table 4.1 NASA MDP Data Sets

<table>
<thead>
<tr>
<th>Data Set</th>
<th>System</th>
<th>Language</th>
<th>Total Loc</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMI - 5</td>
<td>Spacecraft Instrument</td>
<td>C</td>
<td>17K</td>
</tr>
<tr>
<td>KC3-4</td>
<td>Storage management for ground data</td>
<td>Java</td>
<td>8K and 25K</td>
</tr>
<tr>
<td>KC1-2</td>
<td>Storage management for ground data</td>
<td>C++ *</td>
<td></td>
</tr>
<tr>
<td>MW1</td>
<td>Database</td>
<td>C</td>
<td>8K</td>
</tr>
<tr>
<td>PC1,2,5</td>
<td>Flight Software for Earth orbiting Software</td>
<td>C</td>
<td>26K</td>
</tr>
<tr>
<td>PC3,4</td>
<td>Flight Software for Earth orbiting Software</td>
<td>C</td>
<td>30-36K</td>
</tr>
</tbody>
</table>

### Table 4.2 Data Sets

<table>
<thead>
<tr>
<th>Data Set</th>
<th>Attribute</th>
<th>Module</th>
<th>Defect</th>
<th>Defect(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM1</td>
<td>38</td>
<td>344</td>
<td>42</td>
<td>1.22</td>
</tr>
<tr>
<td>JM1</td>
<td>22</td>
<td>9593</td>
<td>1759</td>
<td>18.34</td>
</tr>
<tr>
<td>KC1</td>
<td>22</td>
<td>2096</td>
<td>325</td>
<td>15.5</td>
</tr>
<tr>
<td>KC3</td>
<td>40</td>
<td>200</td>
<td>36</td>
<td>18</td>
</tr>
<tr>
<td>MC1</td>
<td>39</td>
<td>9277</td>
<td>68</td>
<td>0.73</td>
</tr>
<tr>
<td>MC2</td>
<td>40</td>
<td>127</td>
<td>44</td>
<td>34.65</td>
</tr>
<tr>
<td>MW1</td>
<td>38</td>
<td>264</td>
<td>27</td>
<td>10.23</td>
</tr>
<tr>
<td>PC1</td>
<td>38</td>
<td>759</td>
<td>61</td>
<td>8.04</td>
</tr>
<tr>
<td>PC2</td>
<td>37</td>
<td>1585</td>
<td>16</td>
<td>1.0</td>
</tr>
<tr>
<td>PC3</td>
<td>38</td>
<td>1125</td>
<td>140</td>
<td>12.4</td>
</tr>
<tr>
<td>PC4</td>
<td>38</td>
<td>1399</td>
<td>178</td>
<td>12.72</td>
</tr>
<tr>
<td>PC5</td>
<td>39</td>
<td>17001</td>
<td>503</td>
<td>2.96</td>
</tr>
</tbody>
</table>

### 4.7 Performance Measurement

The Performance measured according to the Confusion matrix given in Table 4.3, which is used by many researchers. Table 4.3 illustrates a confusion matrix for a two class problem having positive and negative class values.

#### Table 4.3 Confusion Matrix

<table>
<thead>
<tr>
<th>Actual Class</th>
<th>Predicted Class</th>
<th>Positive</th>
<th>True Positive</th>
<th>False Negative</th>
<th>Negative</th>
<th>False Positive</th>
<th>True Negative</th>
</tr>
</thead>
</table>

Software defect predictor performance of the proposed scheme based on Accuracy, Sensitivity, Specificity, Balance, and ROC Area defined as

\[
\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}
\]

\[
= \frac{\text{TRUE positive} + \text{TRUE negative}}{\text{TRUE positive} + \text{FALSE positive} + \text{TRUE negative} + \text{FALSE negative}}
\]

\[
= \text{The percentage of prediction that is correct}
\]

\[
pd = \text{True positive}
\]
\begin{align*}
    \text{Rate (tpr)} &= \text{Sensitivity} = \frac{TP}{TP + FN} \\
    &= \text{The Percentage Of positive labeled instance that predicted as POSITIVE} \\
    \text{Specify} &= \frac{TN}{FP + TN} \\
    &= \text{The Percentage Of positive labeled instance that predicted as negative}
\end{align*}

\begin{align*}
    Pf &= \text{False Positive Rate (fpr)} = 1 - \text{specificity} \\
    &= \text{The percentage Of negative labeled instances that predicted as negative}
\end{align*}

Formal definitions for \( p_d \) and \( p_f \) are given in the formula. Obviously, higher \( p_d \)s and lower \( p_f \)s are desired. The point (\( p_d=1 \), \( p_f=0 \)) is the ideal position where we recognize all defective modules and never make mistakes.

MGF introduced a performance measure called balance, which is used to choose the optimal (\( p_d, p_f \)) pairs. The definition is shown below from which we can see that it is equivalent to the normalized Euclidean distance from the desired Point (0, 1) to (\( p_f, p_d \)) in a ROC curve.

\[
    \text{BALANCE} = 1 - \sqrt{\frac{(1 - p_d)^2 + (0 - p_f)^2}{2}}
\]

The receiver operating characteristic (ROC) Charles E Metz (1998), curve is often used to evaluate the performance of binary predictors. A typical ROC curve is shown in Figure 4.2. The y-axis shows probability of detection (pd) and the x-axis shows probability of false alarms (pf).

Formal definitions for \( p_d \) and \( p_f \) are given above. Obviously, higher \( p_d \)s and lower \( p_f \)s are desired. The point (\( p_f=0 \), \( p_d=1 \)) is the ideal position where we recognize all defective modules and never make mistakes.

The Area under ROC Curve (AUC) is often calculated to compare different ROC curves. Higher AUC values indicate the classifier is an average, more to the upper left region of the graph. AUC represents the most informative and commonly used, thus it is used as another performance measure for our work.
4.8. RESULTS

This section provides simulation results of some of the Classification algorithm techniques collected by simulation on Software tool named Weka (version 3.6.9) with java embedded program. In the thesis, however, proposed schemes are more comprehensively compared with competent schemes. According to best accuracy value we choose 8 classification algorithms among many classification algorithms. All the evaluated values are collected and compare with different performance measurement parameter.

Accuracy

From the accuracy Table 4.4 we can see different algorithms giving different accuracy on different data set. But the average performance nearly same. For Storage management software (KC1-3) LOG, J48G giving better Accuracy value. For database software written in JAVA programming language (MW1) only PART giving better accuracy value. The performance graph is given in the Figure 4.4.

Sensitivity from the Table 4.5 we see that NB algorithm gives better performance in maximum data set. In case of Decision Table gives the sensitivity zero (sometimes) that means it considering all the class as a true negative. It cannot be considered for defect prediction. LOG, OneR, PART, J48, J48G algorithms giving average performance.
Table 4.4 Accuracy

<table>
<thead>
<tr>
<th>Methods</th>
<th>NB</th>
<th>LOG</th>
<th>DT</th>
<th>JRip</th>
<th>OneR</th>
<th>PART</th>
<th>J48</th>
<th>J48G</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM1</td>
<td>83.94</td>
<td>87.68</td>
<td>89.13</td>
<td>86.23</td>
<td>89.13</td>
<td>73.91</td>
<td>86.23</td>
<td>86.96</td>
</tr>
<tr>
<td>JM1</td>
<td>81.28</td>
<td>82.02</td>
<td>81.57</td>
<td>81.42</td>
<td>79.67</td>
<td>81.13</td>
<td>79.8</td>
<td>79.83</td>
</tr>
<tr>
<td>KC1</td>
<td>83.05</td>
<td>86.87</td>
<td>84.84</td>
<td>84.84</td>
<td>83.29</td>
<td>83.89</td>
<td>85.56</td>
<td>85.56</td>
</tr>
<tr>
<td>KC3</td>
<td>77.5</td>
<td>71.25</td>
<td>75</td>
<td>76.25</td>
<td>71.25</td>
<td>81.25</td>
<td>80</td>
<td>82.5</td>
</tr>
<tr>
<td>MC1</td>
<td>94.34</td>
<td>99.27</td>
<td>99.25</td>
<td>99.22</td>
<td>99.3</td>
<td>99.19</td>
<td>99.3</td>
<td>99.3</td>
</tr>
<tr>
<td>MC2</td>
<td>66</td>
<td>66.67</td>
<td>56.86</td>
<td>56.86</td>
<td>56.86</td>
<td>70.59</td>
<td>52.94</td>
<td>54.9</td>
</tr>
<tr>
<td>MW1</td>
<td>79.25</td>
<td>77.36</td>
<td>85.85</td>
<td>86.79</td>
<td>85.85</td>
<td>88.68</td>
<td>85.85</td>
<td>85.85</td>
</tr>
<tr>
<td>PC1</td>
<td>88.82</td>
<td>92.11</td>
<td>92.43</td>
<td>89.14</td>
<td>91.45</td>
<td>89.8</td>
<td>87.83</td>
<td>88.49</td>
</tr>
<tr>
<td>PC2</td>
<td>94.29</td>
<td>99.05</td>
<td>99.37</td>
<td>99.21</td>
<td>99.37</td>
<td>99.37</td>
<td>98.9</td>
<td>98.9</td>
</tr>
<tr>
<td>PC3</td>
<td>34.38</td>
<td>84.67</td>
<td>80.22</td>
<td>82.89</td>
<td>82.89</td>
<td>82.67</td>
<td>82.22</td>
<td>83.56</td>
</tr>
<tr>
<td>PC4</td>
<td>87.14</td>
<td>91.79</td>
<td>90.18</td>
<td>90.36</td>
<td>90.18</td>
<td>88.21</td>
<td>88.21</td>
<td>88.93</td>
</tr>
<tr>
<td>PC5</td>
<td>96.56</td>
<td>96.93</td>
<td>97.01</td>
<td>97.28</td>
<td>96.9</td>
<td>96.93</td>
<td>97.13</td>
<td>97.16</td>
</tr>
</tbody>
</table>

Table 4.5 Sensitivity

<table>
<thead>
<tr>
<th>Methods</th>
<th>NB</th>
<th>LOG</th>
<th>DT</th>
<th>JRip</th>
<th>OneR</th>
<th>PART</th>
<th>J48</th>
<th>J48G</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM1</td>
<td>0.44</td>
<td>0.267</td>
<td>0</td>
<td>0</td>
<td>0.133</td>
<td>0.333</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>JM1</td>
<td>0.198</td>
<td>0.102</td>
<td>0.07</td>
<td>0.157</td>
<td>0.109</td>
<td>0.03</td>
<td>0.131</td>
<td>0.123</td>
</tr>
<tr>
<td>KC1</td>
<td>0.434</td>
<td>0.238</td>
<td>0.197</td>
<td>0.328</td>
<td>0.254</td>
<td>0.32</td>
<td>0.32</td>
<td>0.32</td>
</tr>
<tr>
<td>KC3</td>
<td>0.412</td>
<td>0.412</td>
<td>0.118</td>
<td>0.118</td>
<td>0.176</td>
<td>0.353</td>
<td>0.353</td>
<td>0.353</td>
</tr>
<tr>
<td>MC1</td>
<td>0.548</td>
<td>0.161</td>
<td>0.194</td>
<td>0.161</td>
<td>0.161</td>
<td>0.194</td>
<td>0.161</td>
<td>0.161</td>
</tr>
<tr>
<td>MC2</td>
<td>0.571</td>
<td>0.545</td>
<td>0</td>
<td>0.091</td>
<td>0.091</td>
<td>0.5</td>
<td>0.045</td>
<td>0.045</td>
</tr>
<tr>
<td>MW1</td>
<td>0.429</td>
<td>0.286</td>
<td>0.429</td>
<td>0.143</td>
<td>0.071</td>
<td>0.286</td>
<td>0.214</td>
<td>0.214</td>
</tr>
<tr>
<td>PC1</td>
<td>0.28</td>
<td>0.24</td>
<td>0.16</td>
<td>0.16</td>
<td>0.08</td>
<td>0.36</td>
<td>0.24</td>
<td>0.24</td>
</tr>
<tr>
<td>PC2</td>
<td>0.333</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>PC3</td>
<td>0.886</td>
<td>0.178</td>
<td>0.233</td>
<td>0.014</td>
<td>0.137</td>
<td>0.288</td>
<td>0.288</td>
<td>0.288</td>
</tr>
<tr>
<td>PC4</td>
<td>0.431</td>
<td>0.538</td>
<td>0.231</td>
<td>0.508</td>
<td>0.323</td>
<td>0.677</td>
<td>0.692</td>
<td>0.677</td>
</tr>
<tr>
<td>PC5</td>
<td>0.427</td>
<td>0.308</td>
<td>0.332</td>
<td>0.521</td>
<td>0.303</td>
<td>0.474</td>
<td>0.498</td>
<td>0.479</td>
</tr>
</tbody>
</table>

Table 4.6 Specificity

<table>
<thead>
<tr>
<th>Methods</th>
<th>NB</th>
<th>LOG</th>
<th>DT</th>
<th>JRip</th>
<th>OneR</th>
<th>PART</th>
<th>J48</th>
<th>J48G</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM1</td>
<td>0.893</td>
<td>0.951</td>
<td>1</td>
<td>0.943</td>
<td>0.984</td>
<td>0.789</td>
<td>0.943</td>
<td>0.951</td>
</tr>
<tr>
<td>JM1</td>
<td>0.956</td>
<td>0.988</td>
<td>0.99</td>
<td>0.968</td>
<td>0.957</td>
<td>0.994</td>
<td>0.954</td>
<td>0.956</td>
</tr>
<tr>
<td>KC1</td>
<td>0.898</td>
<td>0.976</td>
<td>0.959</td>
<td>0.937</td>
<td>0.932</td>
<td>0.927</td>
<td>0.947</td>
<td>0.947</td>
</tr>
<tr>
<td>KC3</td>
<td>0.873</td>
<td>0.794</td>
<td>0.921</td>
<td>0.937</td>
<td>0.857</td>
<td>0.937</td>
<td>0.921</td>
<td>0.952</td>
</tr>
<tr>
<td>MC1</td>
<td>0.947</td>
<td>1</td>
<td>0.999</td>
<td>0.999</td>
<td>1</td>
<td>0.999</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>MC2</td>
<td>0.724</td>
<td>0.759</td>
<td>1</td>
<td>1.031</td>
<td>0.862</td>
<td>0.897</td>
<td>0.931</td>
<td>0.931</td>
</tr>
<tr>
<td>MW1</td>
<td>0.848</td>
<td>0.848</td>
<td>0.924</td>
<td>0.978</td>
<td>0.978</td>
<td>0.978</td>
<td>0.957</td>
<td>0.957</td>
</tr>
<tr>
<td>PC1</td>
<td>0.943</td>
<td>0.982</td>
<td>0.993</td>
<td>0.957</td>
<td>0.989</td>
<td>0.946</td>
<td>0.935</td>
<td>0.943</td>
</tr>
<tr>
<td>PC2</td>
<td>0.946</td>
<td>0.997</td>
<td>1</td>
<td>0.998</td>
<td>1</td>
<td>1</td>
<td>0.995</td>
<td>0.995</td>
</tr>
<tr>
<td>PC3</td>
<td>0.219</td>
<td>0.976</td>
<td>0.958</td>
<td>0.944</td>
<td>0.987</td>
<td>0.96</td>
<td>0.926</td>
<td>0.942</td>
</tr>
<tr>
<td>PC4</td>
<td>0.929</td>
<td>0.968</td>
<td>0.99</td>
<td>0.956</td>
<td>0.978</td>
<td>0.909</td>
<td>0.907</td>
<td>0.917</td>
</tr>
<tr>
<td>PC5</td>
<td>0.983</td>
<td>0.99</td>
<td>0.991</td>
<td>0.987</td>
<td>0.99</td>
<td>0.985</td>
<td>0.986</td>
<td>0.987</td>
</tr>
</tbody>
</table>

70
Specificity

From Table 4.6 we can see some of the algorithms are giving 100 percent specificity that cannot be considered as their respective sensitivity zero. These algorithms can give wrong prediction. So According to the sensitivity and specificity Decision Table algorithm should not be considered for software defect prediction as they are giving high (100%) specificity but 0% sensitivity.

Balance

Looking to the accuracy, sensitivity and specificity performance table we consider the NB, LOG, JRip, OneR, PART, J48, J48G, as their performances are average. From the Figure 4.1, we see that in maximum cases the OneR algorithm giving lower balance value than others. So, no need to use for defect prediction. See values from Table 4.7.

<table>
<thead>
<tr>
<th>Methods</th>
<th>NB</th>
<th>LOG</th>
<th>DT</th>
<th>JRip</th>
<th>OneR</th>
<th>PART</th>
<th>J48</th>
<th>J48G</th>
</tr>
</thead>
<tbody>
<tr>
<td>CM1</td>
<td>0.569</td>
<td>0.481</td>
<td>0.293</td>
<td>0.433</td>
<td>0.387</td>
<td>0.505</td>
<td>0.433</td>
<td>0.433</td>
</tr>
<tr>
<td>JM1</td>
<td>0.432</td>
<td>0.365</td>
<td>0.342</td>
<td>0.403</td>
<td>0.369</td>
<td>0.314</td>
<td>0.385</td>
<td>0.379</td>
</tr>
<tr>
<td>KC1</td>
<td>0.593</td>
<td>0.461</td>
<td>0.431</td>
<td>0.523</td>
<td>0.47</td>
<td>0.516</td>
<td>0.518</td>
<td>0.518</td>
</tr>
<tr>
<td>KC3</td>
<td>0.575</td>
<td>0.559</td>
<td>0.374</td>
<td>0.375</td>
<td>0.409</td>
<td>0.54</td>
<td>0.539</td>
<td>0.541</td>
</tr>
<tr>
<td>MC1</td>
<td>0.678</td>
<td>0.407</td>
<td>0.43</td>
<td>0.407</td>
<td>0.407</td>
<td>0.43</td>
<td>0.407</td>
<td>0.407</td>
</tr>
<tr>
<td>MC2</td>
<td>0.639</td>
<td>0.636</td>
<td>0.293</td>
<td>0.293</td>
<td>0.355</td>
<td>0.633</td>
<td>0.321</td>
<td>0.323</td>
</tr>
<tr>
<td>MW1</td>
<td>0.582</td>
<td>0.484</td>
<td>0.593</td>
<td>0.394</td>
<td>0.343</td>
<td>0.495</td>
<td>0.443</td>
<td>0.443</td>
</tr>
<tr>
<td>PC1</td>
<td>0.489</td>
<td>0.462</td>
<td>0.406</td>
<td>0.405</td>
<td>0.349</td>
<td>0.546</td>
<td>0.461</td>
<td>0.461</td>
</tr>
<tr>
<td>PC2</td>
<td>0.527</td>
<td>0.293</td>
<td>0.293</td>
<td>0.293</td>
<td>0.293</td>
<td>0.293</td>
<td>0.293</td>
<td>0.293</td>
</tr>
<tr>
<td>PC3</td>
<td>0.448</td>
<td>0.419</td>
<td>0.292</td>
<td>0.456</td>
<td>0.303</td>
<td>0.389</td>
<td>0.494</td>
<td>0.495</td>
</tr>
<tr>
<td>PC4</td>
<td>0.595</td>
<td>0.673</td>
<td>0.456</td>
<td>0.651</td>
<td>0.521</td>
<td>0.763</td>
<td>0.772</td>
<td>0.764</td>
</tr>
<tr>
<td>PC5</td>
<td>0.595</td>
<td>0.511</td>
<td>0.528</td>
<td>0.661</td>
<td>0.507</td>
<td>0.628</td>
<td>0.645</td>
<td>0.631</td>
</tr>
</tbody>
</table>

Depending on Accuracy, Sensitivity, Specificity, Balance performance we chose 6 Algorithms from 8 algorithms:

- Naïve BayesSimple
- Logistic
- JRip
- PART
- J48 and
- J48Graft
ROC AREA

Software defect prediction performance based on ROC Area simulated by our scheme given in the Table 4.8. According to ROC Area Logistic and Nave based algorithm gives the better performance for software defect prediction.

<table>
<thead>
<tr>
<th>Methods</th>
<th>CM1</th>
<th>JM1</th>
<th>KC1</th>
<th>KC3</th>
<th>MC1</th>
<th>MC2</th>
<th>MW1</th>
<th>PC1</th>
<th>PC2</th>
<th>PC3</th>
<th>PC4</th>
<th>PC5</th>
</tr>
</thead>
<tbody>
<tr>
<td>NB</td>
<td>0.685</td>
<td>0.681</td>
<td>0.801</td>
<td>0.745</td>
<td>0.861</td>
<td>0.745</td>
<td>0.666</td>
<td>0.736</td>
<td>0.846</td>
<td>0.793</td>
<td>0.84</td>
<td>0.804</td>
</tr>
<tr>
<td>Log</td>
<td>0.668</td>
<td>0.709</td>
<td>0.808</td>
<td>0.604</td>
<td>0.893</td>
<td>0.686</td>
<td>0.592</td>
<td>0.821</td>
<td>0.7</td>
<td>0.802</td>
<td>0.911</td>
<td>0.958</td>
</tr>
<tr>
<td>JRip</td>
<td>0.572</td>
<td>0.562</td>
<td>0.633</td>
<td>0.527</td>
<td>0.58</td>
<td>0.5</td>
<td>0.561</td>
<td>0.561</td>
<td>0.561</td>
<td>0.499</td>
<td>0.589</td>
<td>0.735</td>
</tr>
<tr>
<td>PART</td>
<td>0.492</td>
<td>0.713</td>
<td>0.709</td>
<td>0.612</td>
<td>0.773</td>
<td>0.639</td>
<td>0.611</td>
<td>0.566</td>
<td>0.481</td>
<td>0.728</td>
<td>0.821</td>
<td>0.942</td>
</tr>
<tr>
<td>J48</td>
<td>0.537</td>
<td>0.67</td>
<td>0.698</td>
<td>0.572</td>
<td>0.819</td>
<td>0.259</td>
<td>0.5</td>
<td>0.646</td>
<td>0.39</td>
<td>0.727</td>
<td>0.784</td>
<td>0.775</td>
</tr>
<tr>
<td>J48G</td>
<td>0.543</td>
<td>0.666</td>
<td>0.698</td>
<td>0.587</td>
<td>0.819</td>
<td>0.274</td>
<td>0.5</td>
<td>0.651</td>
<td>0.39</td>
<td>0.738</td>
<td>0.778</td>
<td>0.775</td>
</tr>
</tbody>
</table>

Figure 4.3 Balance

Figure 4.4 ROC Area
Comparison with Other's Results

Song Ying and Liu (2011) proposed a framework in which they used OneR algorithms for defect prediction. But, this can't be useful in defect prediction as it gives "0" sensitivity sometimes and balance values are too lower than others. They used only 10 data set, whereas we used 12 data set with more modules in each data set. Also in our results, the balance values are better to their results.

Figure 4.5 Accuracy

In the comparative results, measurement values are high. Mainly accuracy increased as we used percentage split.

Figure 4.6 Sensitivity
4.9. CONCLUSION

We have attempted to solve the software defect prediction problem through different Data mining (Classification) algorithms. NB and Logistic algorithm gave the overall better performance for defect prediction. PART and J48 gives better performance than OneR and JRip. We understood from this analysis that the data preprocessor/attribute selector can play different roles with different learning algorithms for different data sets and that no learning scheme dominates, i.e., always outperforms the others for all data sets. This means we should choose different learning schemes for different data sets, and consequently, the evaluation and decision process is important.
In order to improve the efficiency and quality of software development, we can make use of the advantage of data mining to analyze and predict large number of defect data collected in the software development. The work reviewed the current state of software defect management, software defect prediction models and data mining technology briefly. Then proposed an ideal software defect management and prediction system, researched and analyzed several software defect prediction methods based on data mining techniques and specific models (NB, Logistic, PART, J48G).
import java.io.BufferedReader;
import java.io.FileNotFoundException;
import java.io.FileReader;
import weka.classifiers.Classifier;
import weka.classifiers.Evaluation;
import weka.classifiers.evaluation.NominalPrediction;
import weka.classifiers.rules.DecisionTable;
import weka.classifiers.rules.PART;
import weka.classifiers.trees.DecisionStump;
import weka.classifiers.trees.J48;
import weka.core.FastVector;
import weka.core.Instances;

public class WekaTest {
    public static BufferedReader readDataFile(String filename) {
        BufferedReader inputReader = null;
        try {
            inputReader = new BufferedReader(new FileReader(filename));
        } catch (FileNotFoundException ex) {
            System.err.println("File not found: " + filename);
        }
        return inputReader;
    }

    public static Evaluation classify(Classifier model, Instances trainingSet, Instances testingSet) throws Exception {
        Evaluation evaluation = new Evaluation(trainingSet);
        model.buildClassifier(trainingSet);
        evaluation.evaluateModel(model, testingSet);
        return evaluation;
    }

    public static double calculateAccuracy(FastVector predictions) {
        double correct = 0;
        for (int i = 0; i < predictions.size(); i++) {
            NominalPrediction np = (NominalPrediction) predictions.elementAt(i);
            if (np.predicted() == np.actual()) {
                correct++;
            }
        }
        return 100 * correct / predictions.size();
    }

    public static Instances[][] crossValidationSplit(Instances data, int numberOfFolds) {
        Instances[][] split = new Instances[2][numberOfFolds];
        for (int i = 0; i < numberOfFolds; i++) {
            split[0][i] = data.trainCV(numberOfFolds, i);
            split[1][i] = data.testCV(numberOfFolds, i);
        }
        return split;
    }

    public static void main(String[] args) throws Exception {
        BufferedReader datafile = readDataFile("dataset.txt");
        Instances data = new Instances(datafile);
        data.setClassIndex(data.numAttributes() - 1);
        // Do 10-split cross validation
        Instances[][] split = crossValidationSplit(data, 10);
        // Separate split into training and testing arrays
        Instances[] trainingSplits = split[0];
        Instances[] testingSplits = split[1];
        // Use a set of classifiers
        Classifier[] models = {
            new J48(), // a decision tree
            new PART(),
            new DecisionTable(), // decision table majority classifier
            new DecisionStump() // one-level decision tree
        };
        // Run for each model
        for (int j = 0; j < models.length; j++) {
            // Collect every group of predictions for current model in a FastVector
            FastVector predictions = new FastVector();
            // For each training-testing split pair, train and test the classifier
            for (int i = 0; i < trainingSplits.length; i++) {
            
        
    
}
Evaluation validation = classify(models[j], trainingSplits[i], testingSplits[i]);
predictions.appendElements(validation.predictions());

// Uncomment to see the summary for each training-testing pair.
// System.out.println(models[j].toString());
}

// Calculate overall accuracy of current classifier on all splits
double accuracy = calculateAccuracy(predictions);
double Specivity = calculateSpecivity(predictions);
double Sensitivity = calculateSensitivity(predictions);
double Balance = calculateBalance(predictions);

// Print current classifier's name and accuracy in a complicated, but nice-looking way.
System.out.println("Accuracy of "+ models[j].getClass().getSimpleName() + ":
  + String.format("%.2f\%", accuracy)
System.out.println("Specivity of "+ models[j].getClass().getSimpleName() + ":
  + String.format("%.2f\%", Specivity)
System.out.println("Sensitivity of "+ models[j].getClass().getSimpleName() + ":
  + String.format("%.2f\%", Sensitivity)
System.out.println("Balance of " + models[j].getClass().getSimpleName() + ":
  + String.format("%.2f\%", Balance) + 
  + String.format("\n---------------------")

}