Software Defect Prediction with Supervised Learning

Author: Christian Heinrich | cah2792@rit.edu
Advisor: Leon Reznik | lr@cs.rit.edu
## Contents

List of Tables .................................................................................................................. 3  
List of Figures .................................................................................................................. 4  
Introduction ................................................................................................................... 5  
Background .................................................................................................................... 6  
Solution ........................................................................................................................... 7  
Data & Metrics .................................................................................................................. 8  
Implementation ............................................................................................................... 12  
Testing & Results ............................................................................................................ 14  
Conclusion & Future Work ............................................................................................... 18  
References ...................................................................................................................... 19
List of Tables

Table 1. Characteristics of NASA Data Sets ................................................................. 9
Table 2. Characteristics of the AR data sets ................................................................. 10
Table 3. Percent of classifications correct against NASA data set ............................... 11
Table 4. AUC and # of points for the ROC curve for each main data set ....................... 15
Table 5. AUC and # of points for the ROC curves for the AR subsets ............................. 15
Table 6. The AUC observed when training using one data set and testing against the others ....................................................... 17
List of Figures

Figure 1. The ROC curves generated by the NASA and AR data sets ................................................................. 14
Figure 2. The ROC curves generated by the AR subsets .................................................................................. 16
Introduction

In less than a century, the presence of software in all walks of life has exploded dramatically, and today forms the foundation upon which business, academia, and life in general is conducted. However, with software becoming so ubiquitous, the potential impact of even a single software failure has grown exponentially. And if that software failure occurs within a critical piece of technology, the results can be catastrophic. Big-box retailers increasingly rely on online storefronts for the majority of their sales, meaning that a software failure bringing down their storefront for even an hour could mean millions of dollars in lost sales. And this is doubly true for online retailers like Amazon. Beyond just financial consequences, software failures represent a real danger, as a failure within the software of critical medical or safety equipment has the potential for fatal results. Even outside of the catastrophic scenarios, software failures can result in everything from a minor annoyance, like a game crashing, to a major problem. While the classic example of Word crashing resulting in the loss of hours of unsaved work may not be fatal, it could still have dire consequences. As the potential consequences resulting from software failures grow, so too does the need for systems to predict the cause of a failure before it occurs, in order to rectify the problem before it results in costly failures.

However, this presents a major problem, as there is no one single cause for every software failure. Failures can result from incompatibilities between different software, the presence of malware, or defects within its source code. Predicting the cause of a failure while taking into account all these possibilities is a difficult problem to tackle. But predicting the presence of any one of the common causes for software failures is a far more manageable one, while still serving to mitigate potential failures. And by far one of the most common sources of failure is the presence of defects within the source code of a program, meaning that determining if a segment of source code is likely to contain a defect represents an attractive problem to solve, as it is more reasonable to handle while still addressing the larger threat of failures. In addition, predicting the presence of defects within source code segments results in cleaner and more reliable code overall. This is likely why this problem has become a popular area for research. Thus, the goal of my project was to develop a system to predict whether a given sample contained a defect, with a supervised learning classifier at its core, which classified samples as either defective or not defective using metrics from the segment of code as input.
Background

This problem of software defect prediction has seen extensive research, most likely due to its importance, as well as the potential applications for a solution. However, there is no single path to solving this problem, and there have been a wide range of methods which have been utilized in addressing it. Of these, the most common method is the same that I developed for this project; specifically a supervised learning based classifier trained which uses static code measurements taken from a code segment as input. Multiple different classifiers have been tested against this problem, with examples including SVM, Random Forest, and Neural Network based solutions. However, in spite of the wide variety in different classifiers employed by different papers, most of them use the same data sets to train and test the classifiers. These datasets were originally made publicly available for use as part of the NASA Metrics Data program, or MDP, in 2004. Since then they have become the most common data sets to use for this research, as demonstrated in Hall et al. [7] which found that over a quarter of the papers they examined utilized those data sets. For example, in Lessmann et al. [4] twenty two different classifiers were benchmarked against one another across the dozen NASA MDP data sets. The best performing classifiers from their comparison were Random Forest, Least Squares SVM and both forms of Multi-layer Perceptron. But they found that between the top 17 classifiers there was a relatively minor difference in accuracy, to the point that they suggested more emphasis should be put on other factors, like complexity, over accuracy.

However, this finding was later shown to be a result of the quality of the data within the NASA data sets themselves. The original NASA data sets were shown to be both noisy and dirty in Sheppard et al. [6], containing samples which had invalid or missing values, and multiple duplicate samples. They laid out a framework for how to clean the NASA data sets, and made available their own cleaned versions. The impact of this cleaned version of the data in Ghotra et al. [5], which revisited the findings from Lessmann et al. [4], but comparing the results seen when using the known to be dirty data sets to the cleaned data sets, and found that while the dirty data sets replicated the previous findings, with the classifiers separated into only two performance brackets, indicating that most classifiers performed the same, when the classifiers were trained using the cleaned data sets, there was a far wider variation in performance, with the best performing classifiers able to achieve higher accuracy than was seen previously.

Unfortunately, the difference between the clean versus dirty NASA data sets only exacerbates their major flaw. When they were originally released in 2004, they represented one of the largest set of data sets dedicated to defect prediction at that time. But over the past decade, the NASA MDP has ended, and the original source for the data sets is no longer active. They are still available from several sources, including the PROMISE repository [1] which I will be using, but slight variations in the data sets exist between the different sources, meaning that there is no guarantee for a valid comparison between the results seen between two papers if they used data sets from different sources. Hence it is critical to specify which source is being used.
The focus of the project was gaining a firsthand understanding of software defect prediction, as this will likely be relevant to me in the immediate future. As such, the overall goal of the project was to develop a solution for software defect prediction based around a supervised learning classifier which was able to achieve reasonable performance. The target for this system was to achieve greater than 0.7 area under curve (AUC) average across all test cases. The curve in this case refers to a Receiver Operating Characteristics (ROC) curve, as seen in Fawcett et al. [10], which serves to visualize the performance of the classifier. In layman’s terms, the target was for samples classified as defective to be correct 70% of the time. This goal line was established in multiple places during my research as reasonably solid performance for this problem.

In order to build the system, I made use of 5 data sets from the Promise Repository of Empirical Software Engineering Data [1]. 4 of these were cleaned data sets, specifically JM1, PC1, KC3 and CM1 from Sheppard et al. [6], which were originally from the NASA MDP. Rather than rely exclusively on the NASA data sets, as have the majority of papers I reviewed, I opted to also utilize the AR data set, again from PROMISE [1], which originally came from Softlab in Turkey, which is broken down into 5 smaller sets, ar1, ar3, ar4, ar5 and ar6, though I also tested a merged version of the data set which contained all the data from the smaller sets, resulting in 10 test cases in total. I opted to use the cleaned version of the NASA data sets in order to avoid the problems seen in Lessmann et al. [4], though I still reviewed the data sets to verify that no further cleaning was needed. By utilizing data sets from two different sources, rather than just the NASA MDP, I hoped to get a more general understanding of the classifiers performance given different development environments.

In order to choose which classification model would be at the core of my solution, I ran an initial round of testing to compare three possible candidates, Radial Basis Function (RBF) network, Multi-layer Perceptron (MLP) network, or Random Forest. This testing was performed using the software WEKA [8], using only the NASA data sets. This also served to gain a better understanding of the data and its characteristics, which led to some alterations in the planned implementation. Based on this initial testing, I was able to determine that an MLP classifier had certain characteristics which made it more attractive than the other two, so I opted to use an MLP as the core of my solution.
Data & Metrics

Despite being derived from two entirely different sources, both the NASA and AR data sets use the same categories of metrics, meaning that many of the attributes are shared between them. There are three major metric categories which the attributes can be divided into, lines of code (LOC), McCabe metrics, and Halstead attributes. While LOC metrics are just what the name implies, McCabe and Halstead metrics represent measurements computed from the source code, and both measure the complexity of the code segment, though in different ways. McCabe metrics originate in McCabe et al [2], and measure the complexity of a code segment by calculating the possible paths of the program. The main metric in this category is CYCLOMATIC_COMPLEXITY. As the number of possible branching paths increases, so too does the CYCLOMATIC_COMPLEXITY. McCabe’s intention was to provide a measurement allowing for developers to tell if their code is becoming overcomplicated, and they should break it into smaller code segments/methods. His original guideline was to break up methods if their complexity exceeded 10. These measurements were later applied to predicting defects in software, with the logic (supported by research) that higher complexity methods are more likely to contain a defect. Halstead attributes, on the other hand are focused less on the complexity of the methods operation, and more on its readability. They originate in Halstead’s Book Elements of Software Science [3], and are based on measurements of the operators and operands within a method. This still measures the complexity of the program, but also metrics like the effort or time needed to program the method. These are used to predict defects on the idea that hard to read code is more likely to contain defects. Finally, LOC metrics, including the number of comments, is used as longer methods are more likely to contain errors.

However, the fact that most of the attributes for each of the data sets can be placed into one of these categories doesn’t mean that they all use the same attributes. In fact, while all of the AR data sets have the same number of attributes at 30, the number of attributes in each NASA data set varies widely. For example, JM1 has the fewest, as 22, while KC3 has almost doubled that at 40. But there is still a core group of attributes which are shared between data sets, even if from different sources, so I opted to build my classifier to use only attributes which are shared across all data sets, with one exception. The intention was to allow for samples from any of the data sets to be plugged in without altering the model in order to fit it. Ultimately this resulted in the classifier using the 20 attributes listed in Tables 1 and 2. All of these bar HALSTEAD_CONTENT appeared in the original data sets. In the case of HALSTEAD_CONTENT, it was present in all the NASA data sets, but none of the AR data sets, but it could be derived fairly easily, as it was equivalent to HALSTEAD_VOLUME/HALSTEAD_DIFFICULTY, so I added it to the AR data sets. Initially in the implementation phase had I pared this down further to only about 12, but I found this negatively impacted the performance. Further investigation revealed that there was no one dominate attribute that was heavily influencing the classification across all or most of the data sets. Instead the most influential attributes for each data set varied widely, so narrowing the attributes to these 20 may have reduced the performance for certain data sets.
Tables 1 and 2 also illustrate one of the major difficulties when tackling this problem, in that not only do several of the data sets contain fairly few samples, but the samples are heavily skewed toward non-defective, with only two data sets breaking 20% of the samples within part of the defective class. These two issues combine to cause a major problem, as there are too few samples to employ common solutions to class skew like removing samples from one to balance the distribution. But the impact of the skewed classes couldn’t be ignored, either. I had originally planned on judging classifiers based on percentage of samples correctly classified, however this skew made that impossible, as seen in table 3, as classifiers could achieve relatively high performance just by classifying every sample as non-defective, which would be completely useless for solving the problem.
Since percentage correct wouldn’t work as a performance measurement, an alternative measure which would account for the skewed class distribution was needed. This came in the form of ROC analysis, which isn’t impacted by the class skew at all, meaning that the results would be accurate. An introduction to utilizing this form of analysis to evaluate classifiers can be seen in Fawcett et al. [10]. Essentially the problem is reframed slightly, as the goal is no longer to correctly classify samples as either defective or not defective. Instead, the goal is to maximize the true positive rate while minimizing the false positive rate. The true positive rate is the number of true positives divided by the number of positive samples, while the false positive rate is the number of false positives divided by the number of negative samples, as seen in Figure 1.

<table>
<thead>
<tr>
<th>Attributes</th>
<th>ar1</th>
<th>ar3</th>
<th>ar4</th>
<th>ar5</th>
<th>ar6</th>
<th>AR</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOC_TOTAL</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>LOC_BLANK</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>LOC_COMMENTS</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>LOC_CODE_AND_COMMENT</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>LOC_EXECUTABLE</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>NUM_UNIQUE_OPERANDS</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>NUM_UNIQUE_OPERATORS</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>NUM_OPERANDS</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>NUM_OPERATORS</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>HALSTEAD_CONTENT</td>
<td>Added</td>
<td>Added</td>
<td>Added</td>
<td>Added</td>
<td>Added</td>
<td>Added</td>
</tr>
<tr>
<td>HALSTEAD_LENGTH</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>HALSTEAD_VOLUME</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>HALSTEAD_LEVEL</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>HALSTEAD_DIFFICULTY</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>HALSTEAD_EFFORT</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>HALSTEAD_ERROR_EST</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>HALSTEAD_PROG_TIME</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>BRANCH_COUNT</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>CYCLOMATIC_COMPLEXITY</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>DESIGN_COMPLEXITY</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>Total Attributes</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>Samples</td>
<td>121</td>
<td>63</td>
<td>107</td>
<td>36</td>
<td>101</td>
<td>428</td>
</tr>
<tr>
<td>Defective</td>
<td>9</td>
<td>8</td>
<td>20</td>
<td>8</td>
<td>15</td>
<td>60</td>
</tr>
<tr>
<td>%Defective</td>
<td>7.44%</td>
<td>12.70%</td>
<td>18.69%</td>
<td>22.22%</td>
<td>14.85%</td>
<td>14.02%</td>
</tr>
</tbody>
</table>

Table 2. Characteristics of the AR data sets
Maximizing the true positive rate can be seen as increasing the confidence that the classifier is correct when it determines that a given sample has a defect, while minimizing the false positive rate decreases the chance that the classifier will incorrectly classify a negative sample as positive. These rates can be used to graph the ROC curve, with the true positive rate as the y-axis and false positive rate as the x-axis, with each going from 0 to 1, resulting in a visual representation of how the false positive rate increases in relation to the true positive rate. Furthermore, the AUC of the ROC curve can be used to evaluate the quality of a given classifier, as it corresponds to the confidence in that classifier’s positive classifications, so more accurate classifiers will have a higher AUC than less accurate one. More importantly, because neither of these rates considers the number of true negatives at all, the skewed class distribution won’t end up impacting the performance. Because of this I opted to use these rates to judge my classifiers performance rather than straight accuracy, which was the same method employed by both Lessmann et al. [4] and Ghotra et al. [5] for their evaluations.

\[
\text{True Positive rate} = \frac{\text{True Positives}}{\text{Total Positives}}
\]

\[
\text{False Positive rate} = \frac{\text{False Positives}}{\text{Total Negatives}}
\]

Equation 1. True positive and False positive rates
Implementation

I opted to implement my MLP based classifier in C#, as I had originally planned to utilize the Encog Machine Learning library [9] in my implementation, as I have used it previously, with good results. However, when I changed how I was doing my analysis I could no longer do so, as it doesn’t support ROC analysis. I instead opted to utilize the ALGLIB library [11], which contained the tools necessary to complete my implementation without issue, as well as tune it to achieve the desired performance.

The highest level class is DefectPredictor.cs, which contains the main method and the RunClassifier method. The RunClassifier method is designed to train and test the classifier against the given data set several times sequentially (in my testing I had it at 10 runs through the classifier). The ROC curve resulting from each of these runs is saved in a new directory created at the start of the method. More importantly, the ROCCurve object output from each run of the classifier is saved in a list of curves, and once the final run of the classifier is completed, the curves are averaged together in order to find the average curve and AUC of the entire run overall. This is intended both to generate a smoother curve overall, as well as allow for evaluation of the classifiers average performance, rather than the best or worst case performance seen from a single run.

The actual classifier is implemented in the MLPAL.cs class, which includes most of the functionality required for the classifier. This includes importing the .csv file which contains the data set, the data from which will be immediately split into training and testing sets using a Random object with a time based seed, with some level of guidance to provide consistency and prevent running over the size of the array. The classifier is then trained by calling TrainNetwork(), which utilizes the functionality in ALGLIB for the actual training, which is accomplished using L-BFGS with 3 restarts, and which ends after 10000 iterations or the change in weights has decreased to below 0.005. The network itself employs an input layer of twenty nodes, a hidden layer of thirty nodes, and 2 output nodes, with a weight decay of 0.2.

The testing is then performed by going through the list of test samples, inputting their values into the network, and getting back a double array prediction, which contains two values, the first is the confidence that the sample contains a defect, and the second is the confidence that it doesn’t. These results are saved in a new TestResult object, which contains the predicted class, the actual class, and the confidence that the sample contains a defect. Once the entire test set has been gone through, a new ROCCurve object is created using the list of results and the number of positive and negative samples, which employs the algorithms described in Fawcett et al. [10] to generate the points of the ROC curve while also calculating the AUC. The results of this are then output and the ROCCurve object is returned.

Within the ROCCurve class there is the calcROC method, which calculates the points within the ROC curve and the AUC, by applying the methods laid out in Fawcett et al. [10]. This method begins with the list of TestResults, which have previously been sorted by the confidence that the sample contained a defect in descending order. The method then iterates through each result object in results, and if the confidence value for the current TestResult isn’t
equal to the previous, then a new point is added to the curve, the variable storing the confidence from the previous TestResults object is set to the current one instead. Finally, if the sample corresponding to the TestResult object was positive, the True Positive number is incremented; otherwise the false positive number is incremented. This generates the ROC curve and the AUC found with only a single iteration over the test results.

Once all ten runs of the classifier have been completed, the curves must be merged together, and the average AUC found. The procedure for this is fairly simple, and is implemented in the static method MergeCurves, and takes a list of curves, and combines both the number of positive/negative samples, as well as combining each ROCCurve object’s results list into a single larger list, which is then sorted in descending order of confidence and the same calcROC method run. This iterates through all the results from each of the ten test runs, finding the correct points for the curve based on the average between the various runs, as well as the area under that curve.
Testing & Results

33% of the data set was set aside for the test set for each run of the classifier. And as stated in the previous section, the final curve to be used for the analysis was created by averaging the results from 10 runs of the classifier (i.e. 1 run of RunClassifier() in DefectPredictor.cs) with each run starting with randomized weights and a randomly selected training vs test split. Figure 2 displays the ROC curves created by the classifier run against each of the main datasets. AR in this case refers to the entire AR data set, i.e. all of the subsets merged together. The line black line represents the performance from a purely random 50/50 guess. This represents an AUC of 0.5, and an ROC curve below this line is indicative of a classifier which is performing worse than a purely random guess. However, none of the classifier runs performed that poorly. There are cases where a ROC curve goes over this line near (1, 1), most apparently with the JM1 data set, but this doesn’t present a major issue.

Figure 1. The ROC curves generated by the NASA and AR data sets
Overall the classifier performed extremely well on the CM1 and PC1 datasets, and fairly poorly on the JM1 data set. However, this seems to line up with the trends I observed in my research, for example in Lessmann et al. [4], the AUC observed for the JM1 data set was consistently up to 0.1 worse than the performance on most of the other data sets. This holds true here as well, as illustrated by Table 4. There are other interesting aspects to the ROC curves, including how closely aligned the KC3 and AR curves are, as they both follow the same path, though the AR curve does separate a bit at the end. An additional point to note is that the reason for the JM1 curve appearing to be much smoother than the other curves is due to the number of points in the curve, as JM1 has over 10x the number of points, due to the size of the data set relative to the others.

Overall, the classifier performance observed meets the stated goal of an average of >0.7 AUC across the tested data sets. However, The classifier performed worse than expected on certain data sets, with JM1 the most prominent among these, only seeing an AUC of 0.63, which is almost 0.1 AUC below the performance observed in Lessmann et al. [4], despite using the cleaned data and that JM1 changed the least when going down to 20 attributes for the input, as it only had 22 attributes to begin with. In fact, outside of CM1, the performance for all of the data sets is lower than observed in Lessmann et al. [4], though this may be the result of removing important features from the data sets. This seems to indicate that a single set of common attributes probably doesn’t work across all data sets, which would indicate that if each data set represents a different section of an organization/company as is the case with the NASA data sets, then addressing this problem would require tuning the system to align with that specific section of the company.

Examining the performance of the classifier when run using each subset of AR results in a much higher performance in comparison to the when the classifier is run against the whole AR data set, which seems to further suggest that the different sets of code from which the metrics are drawn may have certain trends, which are lost when working with the entire set of samples. The ROC curve generated by the classifier when run using these subsets can be seen in
Figure 3. The curves are far less smooth due to the small number of samples available, which meant that far fewer points were generated for the curves. Despite the high performance seen across the AR subsets, better than when the classifier was run using the AR data set as a whole, I’ve opted to treat the average from Table 4, between the 4 NASA data sets and the entire AR data set as a whole, as the final average performance for my classifier, rather than the average between all of the NASA data sets and all of the AR subsets, as I feel that will be more representative of the classifiers performance in comparison to other systems.

Figure 2. The ROC curves generated by the AR subsets

The final round of testing was intended to evaluate how well the classifier could predict defects in samples from new sources when trained on existing data sets. To test this, I ran the classifier, but instead of splitting a single data set into training and testing sets, I used one data set as the training set, and another as the test set. This was repeated for each of the NASA data sets plus the full AR data set, using each in turn, until the classifier had been run on each possible combination of training and testing sets, excluding using the same set for both training and test sets. The results of this testing can be seen in Table 6. Overall, there was a significant
drop in performance observed with this testing, with the classifier actually performing worse than random in two cases, when it was trained using KC3 and tested against CM1, and vice versa. Across the board, the classifier performed at least .1 AUC worse than when it was run normally with the just the training set. This seems to support the earlier observation that the different data sets vary widely in which metrics best indicate a defective sample. The reason behind this is most likely that since the data sets represent different code bases, either from different segments of the organization, in the case of the NASA data sets, or a different one altogether, as with the AR data set, the differences between these code bases, either in language, packages, or practices utilized, tend to result in different flaws. For example, one section might enforce strict standards for code commenting, while another might not, resulting in the latter having more defects due to poor code readability than the first.

<table>
<thead>
<tr>
<th>Training set</th>
<th>AR</th>
<th>KC3</th>
<th>PC1</th>
<th>CM1</th>
<th>JM1</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR</td>
<td>N/A</td>
<td>0.51</td>
<td>0.66</td>
<td>0.6</td>
<td>0.57</td>
</tr>
<tr>
<td>KC3</td>
<td>0.58</td>
<td>N/A</td>
<td>0.69</td>
<td>0.4</td>
<td>0.55</td>
</tr>
<tr>
<td>PC1</td>
<td>0.56</td>
<td>0.63</td>
<td>N/A</td>
<td>0.66</td>
<td>0.58</td>
</tr>
<tr>
<td>CM1</td>
<td>0.54</td>
<td>0.48</td>
<td>0.63</td>
<td>N/A</td>
<td>0.53</td>
</tr>
<tr>
<td>JM1</td>
<td>0.51</td>
<td>0.64</td>
<td>0.72</td>
<td>0.62</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 6. The AUC observed when training using one data set and testing against the others
Conclusion & Future Work

In conclusion, I believe I have successfully achieved the goals I set out at the beginning of this project, not only because I managed to build a solution which performs better than my target, but because I achieved the focus of my project to gain a better understanding of how this problem works, and what can be implemented to solve it. That being said, I am still somewhat dissatisfied with the performance of the classifier against certain data sets, specifically KC3 and JM1. Having run some tests with the classifier against KC3 when using the full data set, rather than the 20 attributes I chose, I firmly believe that the source of the problems with KC3 comes down to removing key attributes it needed to achieve the best performance possible, as with all attributes it seemed to perform significantly better. The same cannot be said for JM1, as very little was removed from it. Instead I believe the performance seen on JM1 is a result of it being a significantly harder data set to tackle, as the same drop is observed in other papers like Lessmann et al. But outside of those two data sets, the classifiers performance seems to have been excellent, as both PC1 and CM1 were well above the 0.7 target, as were the majority of the AR subsets.

Most importantly I’ve learned several key lessons about tackling this problem which I would have likely had to learn down the road anyway. The biggest of these was during my initial testing when I saw classifiers reading as having achieved high performance, but never having classified anything as defective, which defeats the purpose of the solution in the first place. This observation is what led me to reconsider using accuracy to compare the performance between classifiers, and go back to my research to find another method to use, i.e. ROC analysis.

Learning this lesson sooner rather than later will most likely prove to be useful for my work in the future. Come January I will be starting work with Amazon Video, and I already know that the problem I will most likely be confronting there will be predicting software failures, so gaining direct experience implementing a solution to this problem prior to starting work will serve me well, as I already have ideas for how to potentially improve the performance. Specific examples include some form of AUC measurement as part of the model evaluation during training, which could result in improved performance overall. Another possibility is improving the feature selection by implementing a system which selects the most significant features prior to training, and uses only those for the training. Using that type of system could achieve what I was intending when I opted to narrow the features being used to only those shared between all the data sets, by creating a system which can run on data sets with different numbers of features while still achieving solid performance.
References


11. "ALGLIB (www.alglib.net), Sergey Bochkanov".