Comparing Accuracy of Software Defect Prediction and its Dimensionality

John Harakas

November 30, 2015

Abstract

Large scale software requires tremendous resources in assessing and identifying potential defects prior to mass deployment. Although most software bugs are properly patched in the development phase, the reliance on large software packages by key infrastructures necessitates greater scrutiny of code design prior to release. If predictive models can be constructed to identify defect-prone modules with a reasonable degree of accuracy, a degree of software quality can be assured, minimizing post-release software patches that may have to be applied on-site. In assessing potential defects, a number of software metrics can be selected as attributes for predictive model building. The quality of the model is largely influenced by the quality its attributes and to make any reasonable gains, a reduction in dimensionality must be achieved. The following paper analyzes performance of defect prediction models based on 42 and 28 software metrics.

1 Introduction

Data dimensionality pose tremendous difficulties in data sciences and machine learning and conventional statistics in many cases do not apply. [1] Software on an enterprise scale, particularly with code lines numbering in the millions, defects are inevitable and rather prevalent. It is unlikely that one single or even a very small number of instances determinatively predict faults in modules, but on the other hand, attempting to build a model on every possible software feature is computationally expensive and most of all does not work. [2] For multivariate independent variable cases, if the dimensionality can be reduced to a level where expensive ranking techniques can be used, the most relevant features may come to surface.

2 Related Work

A large number of studies have examined the determining factors in software defects using machine learning approaches. [3] Gao et al. (2011) applied the same data set, and in determining the best ranking methods for feature reduction. [4] Baesens et al (2011) used association filter methods in determining software defect prediction.

3 Methodology

Five classification models were tested and analyzed. The Receiver Operating Characteristic (ROC) curve was the metric of each algorithm's performance. The training phase used five-fold cross validation using the following classifiers: Naive-Bayes (NB), C4.5, Multilayer Perceptron (MLP), Logistic Regression, and Random Forrest (RF). Models were constructed for both datasets across all four releases to evaluate their performance. In addition to assessing performance based on the given data, the same procedure was done with a modified resampled data set to compensate for the large class imbalance. Note: 10 runs of 5-fold cross-validation were not used in assessing all data because the computation time was prohibitively long.

4 Case Study

4.1 Data Description

The data for this study was collected from four releases from a very Large Legacy Telecommunications software System (LLTS) denoted R1, R2, R3, and R4. The data was split into two sets containing 28 and 42 metrics referred to as Data28 and Data42. Data28 contains 24 product and 4 execution metrics. Data42

contains 24 product, 14 process, and 4 execution metrics. The last variable in each data set is the dependent variable denoted as CLASSID with corresponding class identifiers of 1 for *nfp* and 2 for *fp*

The *fp* and *nfp* module counts for each successive software release are outlined below:

	nfp	fp	fp %	total
R1	3420	229	6.3%	3649
R2	3792	189	4.7%	3981
R3	3484	47	1.3%	3541
R4	3886	92	2.3%	3978

In all instances, the two classes are imbalanced with the *nfp* class overwhelmingly larger. Classifiers, particularly binary, by nature are going to exhibit bias towards the majority class resulting in a much higher rate of misclassification of the minority class.[5] For that reason it is important to assess performance in terms of the *fp* class. The confusion matrix for each classifier in addition provides useful metrics of performance:

$$\begin{Bmatrix} TN & FP \\ FN & TP \end{Bmatrix}$$

Where TP, FP, FN make up the confusion matrix of True Positive, (True Negative), False Positive, and False Negative:

As for classifiers themselves, a number of sampling techniques have been proposed to equalize class distributions, and the resulting performance has been studied. The *fp* class ranges from 1.3-6.3% across the four releases and other experimental data has shown that certain sample manipulation techniques work fairly consistently; in this case Synthetic Minority Oversampling Technique (SMOTE) was used. SMOTE typically extrapolates new points using k-nearest-Neighbors (typically $k = 5$) in order to upsample the minority class.[6]. Although their performance can potentially hinge on countless factors, a modest adjustment in sampling may produce noticeable gains.

4.2 Results

In many instances performance was a trade off whereas the classifiers had a reasonable degree of accuracy in predicting the correct class. However, the *fp* prediction was very poor. [6]. The dataset by nature is overwhelmingly imbalanced; leading to a tremendous bias towards the majority class; in this case instances being labeled as *nfp*. [7]

4.2.1 No Resampling

Disregarding the imbalanced class distribution, the results were as predicted; by measure of correct classifications, the classifiers all performed extraordinarily well. However the incorrectly classified percentage was at least as high as the size of the *fp* class. For both Data42 and Data28, the NB classifier performed the best of all learners where it properly classified between 39.7% and 46.8% of the *fp* instances. The higher rate can largely be attributed to a higher false positive rate for the *fp* class which is evident in the ROC performance. However in terms of detecting defects, a more zealous approach may be more suitable.

	AUC (Weighted Average)			
	R1 (42)	R2 (42)	R3 (42)	R4 (42)
NB	0.781	0.804	0.805	0.752
LR	0.815	0.844	0.755	0.780
MLP	0.791	0.803	0.754	0.798
C4.5	0.564	0.634	0.487	0.565
RF	0.803	0.829	0.768	0.807

The confusion matrix for each classifier gave a better overall performance measure. All but NB, classified less than 10% of all *fp* instances.

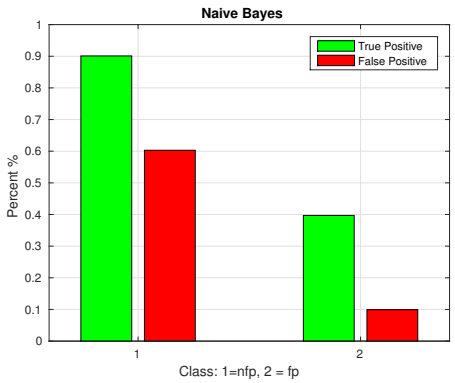

Figure 1: Naive Bayes (R1-42)

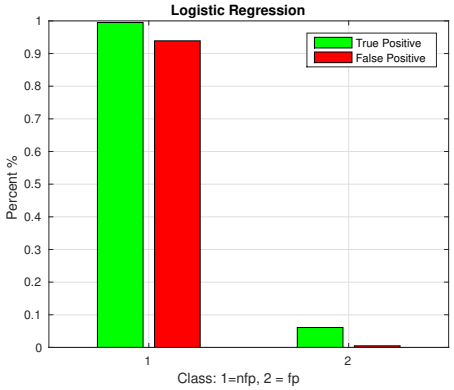

Figure 2: Logistic Regression (R1-42)

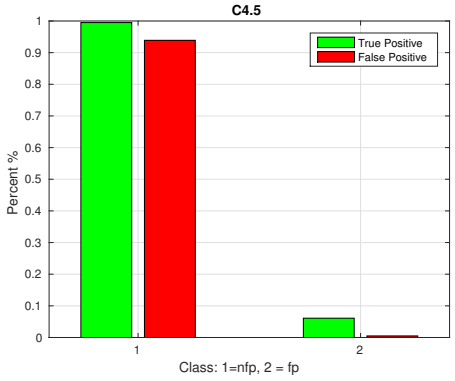

Figure 3: C4.5 (R1-42)

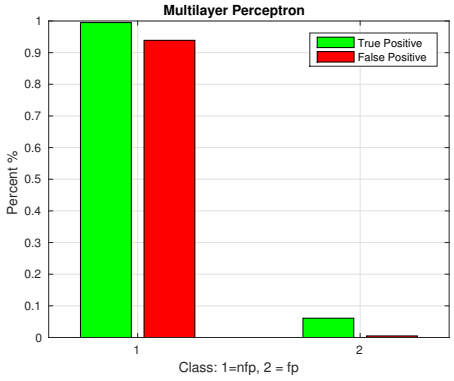

Figure 4: Multilayer Perceptron (R1-42)

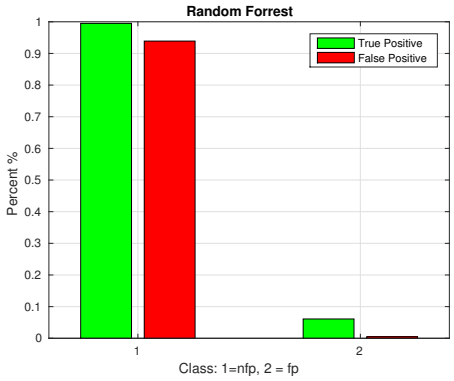

Figure 5: Random Forest (R1-42)

The Data28 results did not display significantly better performance. Their AUC values were roughly comparable to Data42.

	AUC (Weighted Average)			
	R1 (28)	R2 (28)	R3 (28)	R4 (28)
NB	0.761	0.794	0.796	0.717
LR	0.806	0.823	0.809	0.793
MLP	0.779	0.772	0.809	0.738
C4.5	0.615	0.631	0.487	0.545
RF	0.775	0.817	0.725	0.748

Overall, several factors can be attributed to the performance of the classifiers for both Data42 and Data28; the most significant of which was class imbalance and the software metric size. Class imbalance is known to bias classifiers towards the majority class. [7] The classifier confusion matrices reflected the bias tremendously where the *nfp* class false positive rate was exceedingly high. Additionally, estimating the covariance among multivariate independent variables requires an exceedingly large number of instances. [2]. The low number of *fp* instances results in a very poor estimation of covariance. Another apparent issue was in how Weka implements cross-validation by returning the average performance across all models. In this case where five-fold cross-validation was used, the corresponding model likely had been disproportionately influenced by poor performance in certain folds. A higher fold can reduce the likelihood of such influence. [5] The poor implementation of five-fold cross-validation can be checked when reevaluating the model on the same data that it was trained on. In every case the model was tested on the same training set, the performance dropped greatly; indicating an insufficient number of folds during training.

Models generated from both cases performed generally poorly other than Naive-Bayes and consistently misclassified nearly all defects, catching less than 1% for other releases.

	fp	TP		
	R1 (42)	R2 (42)	R3 (42)	R4 (42)
NB	0.397	0.423	0.468	0.446
LR	0.135	0.095	0.043	0.043
MLP	0.031	0.053	0.000	0.033
C4.5	0.114	0.122	0.000	0.065
RF	0.061	0.032	0.000	0.000

	fp	TP		
	R1 (28)	R2 (28)	R3 (28)	R4 (28)
NB	0.362	0.386	0.404	0.457
LR	0.087	0.053	0.043	0.011
MLP	0.017	0.026	0.000	0.000
C4.5	0.039	0.063	0.000	0.022
RF	0.044	0.021	0.000	0.000

In assessing subsequent releases, the models performed just as poorly, showing little predictive capability. Models from the best performing releases were tested against the other releases and the performance was in line with the training model. For example, NB and LR models were chosen from R3 and R1 respectively. The models were tested against the other releases and tabulate below:

	fp	TP		
	R1 (42)	R2 (42)	R3 (42)	R4 (42)
NB	0.332	0.402	0.447	0.370
LR	0.166	0.127	0.170	0.185

4.2.2 With Resampling

Resampling the minority class using SMOTE displayed much better performance. Based on the minority upsample rate of 300%, the best classifiers were C4.5 and Random Forest. Other studies have been done to show that Random Forest and C4.5 respond best to SMOTE. [5]

	AUC (Average)			
	R1 (42)	R2 (42)	R3 (42)	R4 (42)
NB	0.792	0.817	0.819	0.804
LR	0.849	0.867	0.836	0.866
MLP	0.834	0.854	0.815	0.827
C4.5	0.856	0.847	0.757	0.821
RF	0.965	0.973	0.975	0.977

	AUC (Average)			
	R1 (28)	R2 (28)	R3 (28)	R4 (28)
NB	0.770	0.813	0.820	0.768
LR	0.812	0.841	0.829	0.805
MLP	0.813	0.839	0.812	0.771
C4.5	0.794	0.789	0.749	0.766
RF	0.944	0.952	0.965	0.960

	fp TP			
	R1 (42)	R2 (42)	R3 (42)	R4 (42)
NB	0.417	0.460	0.521	0.560
LR	0.389	0.352	0.229	0.261
MLP	0.564	0.475	0.186	0.345
C4.5	0.733	0.690	0.543	0.644
RF	0.753	0.770	0.489	0.603

	fp TP			
	R1 (28)	R2 (28)	R3 (28)	R4 (28)
NB	0.388	0.410	0.484	0.516
LR	0.319	0.294	0.133	0.179
MLP	0.295	0.430	0.074	0.253
C4.5	0.567	0.604	0.410	0.557
RF	0.600	0.569	0.250	0.457

In general it can be observed that *fp* detection rates increased in nearly all instances with C4.5 and RF making the greatest gains. Contrary to what was expected, Data42 produced better models than Data28. However, this may be an indicator of the underlying bias in the data's feature selection.

The figures below are the ROC curves for R1 with Data28, Date28 SMOTE, Data42, and Data42 SMOTE. Although all learners performed better, NB stayed very close to the same, LR and MLP made slight gains, and C4.5 and RF made tremendous gains.

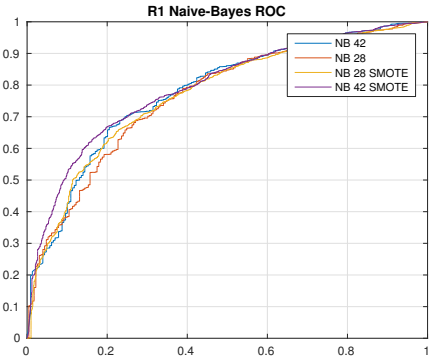

Figure 6: Naive Bayes R1 ROC

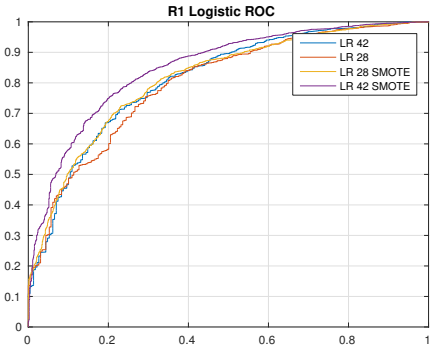

Figure 7: Logistic Regression R1 ROC

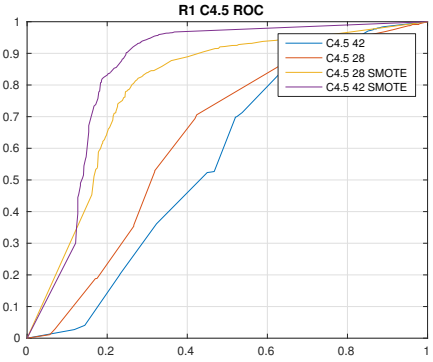

Figure 8: C4.5 (R1-42)

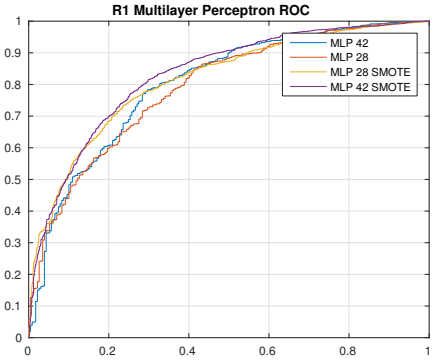

Figure 9: Multilayer Perceptron R1 ROC

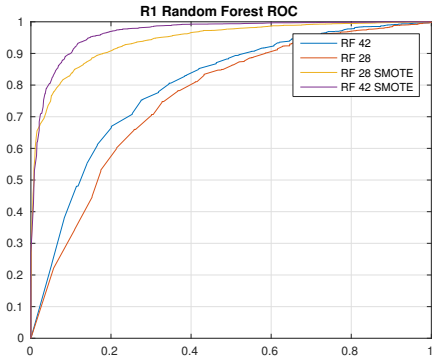

Figure 10: Random Forest R1 ROC

The results tabulated at the end of this paper are for the ten runs of five-fold cross validation; the models which had not been resampled performed with similarly poor results. The output is consistent with the inferences made previously that a resampled data set has significantly better performance. The tabulated results below were models trained from R1 for C4.5 and R2 for RF, then tested against the other three original unmodified data sets.

	fp	TP		
	R1 (42)	R2 (42)	R3 (42)	R4 (42)
C4.5	0.907	0.212	0.277	0.250
RF	0.157	1.000	0.255	0.174

The resampled data sets did not perform well when tested against the original sets. However they did perform well against the other upsampled sets.

	fp	TP		
	R1 (42)	R2 (42)	R3 (42)	R4 (42)
C4.5	0.907	0.676	0.691	0.660
RF	0.400	1.000	0.574	0.530

As for Data28 that was resampled, the performance was actually slightly better when tested against the original data. Below is the TP for *fp* class using R1 to model C4.5 and R2 to model RF.

	fp	TP		
	R1 (28)	R2 (28)	R3 (28)	R4 (28)
C4.5	0.245	0.852	0.277	0.217
RF	0.965	0.259	0.298	0.326

Compared against the other resampled data sets, their performance although much better, was not as good as Data42. There are a couple of possible explanations; the first of which is that Data28 was resampled after its feature set had been reduced. A worthwhile experiment would be to assess the feature rankings of the resampled data sets prior to reducing the feature set. Another point to be made is that there are other upsampling or undersampling techniques in addition to the magnitude of change.

	fp	TP		
	R1 (28)	R2 (28)	R3 (28)	R4 (28)
C4.5	0.371	0.938	0.452	0.410
RF	0.991	0.397	0.388	0.448

5 Conclusion

An inherent problem not only in fault prediction but in data analysis in general is the problem of accurately modeling a high dimensional data set. In all instances Naive-Bayes achieved the greatest number of *fp* classifications but resampling the *fp* class gave a performance boost to most notably C4.5 and Random Forest. Much better results can likely be achieved by optimizing the resampling type and rate. [5] Other classifiers have performed better using other methods and by downsampling the majority class; Random Oversampling (RO) and Random Undersampling (RU). RO randomly duplicates instances from the minority class while RU randomly discards instances from the majority class. Both of which were studied in their effectiveness based on class imbalance comparable to the above data sets and had varying levels of success across each algorithm. [5]

In assessing the performance, the ROC curve did not fully describe how well the model performed in either data sets. Other measures such as F-measure, recall, and precision have better descriptive values for highly imbalanced classification.[8] Outside the scope of this research, a gain in performance from resampling the classes is reflected in recall *without* detriment to the models precision, the sampling rate may be maximized through a constrained optimization solution. The false positive rate for the *fp* class gave a better description although it is important to note that it does not take into consideration the number of false positives for the *nfp* class. However in the case where the goal is defect prediction, the misclassification of non defective modules might be a reasonable cost. An additional consideration to be made is reassessing the selection of software metrics for a reduced search space. Resampling prior to feature ranking will likely yield better results as opposed to what was done in this work.

References

[1] David L Donoho et al. High-dimensional data analysis: The curses and blessings of dimensionality. *AMS Math Challenges Lecture*, pages 1–32, 2000.

[2] Pushpa L Gupta and RD Gupta. Sample size determination in estimating a covariance matrix. *Computational Statistics & Data Analysis*, 5(3):185–192, 1987.

[3] Wang H Seliya N Gao K, Khoshgoftaar T. Choosing software metrics for defect prediction: An investigation on feature selection techniques. 41:579–606, April 2011.

[4] Baojun Ma, Karel Dejaeger, Jan Vanthienen, and Bart Baesens. Software defect prediction based on association rule classification. *Available at SSRN 1785381*, 2011.

[5] Jason Van Hulse, Taghi M Khoshgoftaar, and Amri Napolitano. Experimental perspectives on learning from imbalanced data. In *Proceedings of the 24th international conference on Machine learning*, pages 935–942. ACM, 2007.

[6] Hall L Kegelmeyer W Chawla N, Bowyer K. Smote: Synthetic minority over-sampling technique. 16:321–357, June 2002.

[7] Nitesh V Chawla. Data mining for imbalanced datasets: An overview. In *Data mining and knowledge discovery handbook*, pages 853–867. Springer, 2005.

[8] Victor H Guo H. Learning from imbalanced data sets with boosting and data generation: The databoost-im approach. 6:1263–1284, June 2009.

6 Java Source Code

```
import java.io.*;
import weka.core.Instances;
import weka.classifiers.Classifier;
import weka.classifiers.Evaluation;
import weka.core.converters.CSVSaver;
import java.util.Random;
import weka.classifiers.evaluation.ThresholdCurve;
import weka.gui.visualize.*;
import weka.classifiers.evaluation.NominalPrediction;
import weka.classifiers.trees.J48;
import weka.classifiers.bayes.NaiveBayes;
import weka.classifiers.functions.Logistic;
import weka.classifiers.functions.MultilayerPerceptron;
import weka.classifiers.trees.RandomForest;
import weka.core.FastVector;

public class Reading
{
    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
    {
        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 CurveToCSV(Instances data, int i, String modName)
throws IOException {
    CSVSaver csv_saver=new CSVSaver();
    csv_saver.setInstances(data);
    String cs = "csv/"+ modName + (i+1) + ".csv";
    csv_saver.setFile(new File(cs));
    csv_saver.writeBatch();
    System.out.println("CSV Written");
}

public static void MeanStats(double[] m, String type, String mod) throws IOException
{
    double sum = 0;
    for (int i = 0; i < m.length; i++)
    {
        sum += m[i];
    }
    System.out.println(type + " mean: \t" + sum / m.length);
    String s = type + " mean: \t" + sum / m.length + "\n";
    WriteStats(s,mod);
}

public static void WriteStats(String s, String mod) throws IOException
{
    BufferedWriter writer = null;
    String logFile = "mean/" + mod + ".dat";
    File f = new File(logFile);
    writer = new BufferedWriter(new FileWriter(f, true));
    writer.write(s);
    writer.close();
}


public static void doClassify(Classifier mod, Instances data, String modName, String[] args, String opt)
{
    int runs = 10;
    int folds = 5;
    int seed = 1;
    double [] TP1;
    double [] TP0;
    double [] auc;
    TP0 = new double[runs];
    TP1 = new double[runs];
    auc = new double[runs];
    String[] options;
    System.out.println("ok");
    data.setClassIndex(data.numAttributes() - 1);
    for (int i = 0; i < runs; i++)
    {
    seed = i+1;                        // New seed on every run
    Random rand = new Random(seed);
    data.randomize(rand);          // randomize data
    Evaluation eval = new Evaluation(data);
    for (int n = 0; n < folds; n++)
    {
        Instances train = data.trainCV(folds, n);
        Instances test = data.testCV(folds,n);
        Classifier clsCopy = Classifier.makeCopy(mod);
        if (opt != null)
        {
            options = weka.core.Utils.splitOptions(opt);
            clsCopy.setOptions(options);
        }
        clsCopy.buildClassifier(train);
        eval.evaluateModel(clsCopy, test);
    }
    System.out.println();
    System.out.println("=== Setup run " + (i+1) + " ===");
    System.out.println("Dataset: " + data.relationName());
```

```
        System.out.println("Folds: " + folds);
        System.out.println("Seed: " + seed);
        System.out.println();
        System.out.println(eval.toSummaryString("=== " + folds + "-fold Cross-validation run " + (i+1) + "==
        System.out.println(eval.truePositiveRate(0));
        System.out.println(eval.truePositiveRate(1));
        System.out.println(eval.areaUnderROC(1));
        TP0[i] =eval.truePositiveRate(0);
        TP1[i] = eval.truePositiveRate(1);
        auc[i] = eval.areaUnderROC(1);
        ThresholdCurve tc = new ThresholdCurve();
        Instances result = tc.getCurve(eval.predictions(), 0);
        CurveToCSV(data,i,modName);
        }
        MeanStats(TP0,"TP C1",modName);
        MeanStats(TP1,"TP C2",modName);
        MeanStats(auc,"AUC",modName);
    }

    public static void main(String[] args) throws Exception
    {
        int num = 3;
        Classifier[] models = {
        new NaiveBayes(),
        new Logistic(),
        new MultilayerPerceptron(),
        new J48(),
        new RandomForest()
        };
        String[] mod;
        mod = new String[5];
        mod[0] = "NB";
        mod[1] = "LR";
        mod[2] = "MLP";
        mod[3] = "C45";
        mod[4] = "RF";

        String[] d28;
        d28 = new String[4];
        d28[0] = "data/28metrics/R1.arff";
        d28[1] = "data/28metrics/R2.arff";
        d28[2] = "data/28metrics/R3.arff";
        d28[3] = "data/28metrics/R4.arff";

        String[] d28Smote;
        d28Smote = new String[4];
        d28Smote[0] = "data/28smote/R1.arff";
        d28Smote[1] = "data/28smote/R2.arff";
        d28Smote[2] = "data/28smote/R3.arff";
        d28Smote[3] = "data/28smote/R4.arff";

        String[] d42;
        d42 = new String[4];
        d42[0] = "data/42metrics/R1.arff";
        d42[1] = "data/42metrics/R2.arff";
        d42[2] = "data/42metrics/R3.arff";
        d42[3] = "data/42metrics/R4.arff";

        String[] d42Smote;
        d42Smote = new String[4];
        d42Smote[0] = "data/42smote/R1.arff";
        d42Smote[1] = "data/42smote/R2.arff";
        d42Smote[2] = "data/42smote/R3.arff";
        d42Smote[3] = "data/42smote/R4.arff";

        BufferedReader datafile = readDataFile(d28Smote[0]);
        Instances data = new Instances(datafile);   // create copy of original data
        WriteStats("======== R1_28Smote ======\n",mod[num]);
        doClassify(models[num],data,mod[num],args,null);
        for (int i = 1; i < 4; i++)
        {
```

```
        datafile = new BufferedReader(readDataFile(d28Smote[i]));
        data = new Instances(datafile);   // create copy of original data
        WriteStats("======== R"+(i+1)+"_28Smote ======\n",mod[num]);
        doClassify(models[num],data,mod[num],args,null);
        }

        for (int i = 0; i < 4; i++)
        {
        datafile = new BufferedReader(readDataFile(d28[i]));
        data = new Instances(datafile);   // create copy of original data
        WriteStats("======== R"+(i+1)+"_28 ======\n",mod[num]);
        doClassify(models[num],data,mod[num],args,null);
        }

        for (int i = 0; i < 4; i++)
        {
        datafile = new BufferedReader(readDataFile(d42Smote[i]));
        data = new Instances(datafile);   // create copy of original data
        WriteStats("======== R"+(i+1)+"_42Smote ======\n",mod[num]);
        doClassify(models[num],data,mod[num],args,null);
        }

        for (int i = 0; i < 4; i++)
        {
        datafile = new BufferedReader(readDataFile(d42[i]));
        data = new Instances(datafile);   // create copy of original data
        WriteStats("======== R"+(i+1)+"_42S ======\n",mod[num]);
        doClassify(models[num],data,mod[num],args,null);
        }

        System.out.println();
        System.out.println("All done");
    }

}
```

7 10 Runs, 5-Fold Cross Validation Results

```
[Release Number]_[Metric Numer] [SMOTE]

------------------------------- Naive-Bayes -------------------------------
======== R1_28Smote ======
TP C1 mean:     0.8917251461988304
TP C2 mean:     0.39115720524017467
AUC mean:   0.7730428828621774
======== R2_28Smote ======
TP C1 mean:     0.9016350210970465
TP C2 mean:     0.4128306878306878
AUC mean:   0.813431821099279
======== R3_28Smote ======
TP C1 mean:     0.9073840870062965
TP C2 mean:     0.48457446808510635
AUC mean:   0.8161209489824502
======== R4_28Smote ======
TP C1 mean:     0.8525218733916624
TP C2 mean:     0.5209239130434782
AUC mean:   0.778915113338853
======== R1_28 ======
TP C1 mean:     0.9021052631578946
TP C2 mean:     0.36506550218340605
AUC mean:   0.7703825429658571
======== R2_28 ======
TP C1 mean:     0.9109968354430379
TP C2 mean:     0.40582010582010575
AUC mean:   0.8004476145826358
======== R3_28 ======
TP C1 mean:     0.9159988551803091
TP C2 mean:     0.3893617021276595
AUC mean:   0.779196555797781
```

```
======== R4_28 ======
TP C1 mean:     0.8778950077200204
TP C2 mean:     0.4815217391304348
AUC mean:   0.7495726017588221
======== R1_42Smote ======
TP C1 mean:     0.8897368421052633
TP C2 mean:     0.41943231441048034
AUC mean:   0.7930431063101714
======== R2_42Smote ======
TP C1 mean:     0.8963607594936709
TP C2 mean:     0.4619047619047619
AUC mean:   0.8142743948552228
======== R3_42Smote ======
TP C1 mean:     0.9042644533485976
TP C2 mean:     0.5255319148936171
AUC mean:   0.8190849967725828
======== R4_42Smote ======
TP C1 mean:     0.8638960370560989
TP C2 mean:     0.5654891304347825
AUC mean:   0.8071070691333437
-------------------------------- Logistic Regression -------------------------------
======== R1_28Smote ======
TP C1 mean:     0.947046783625731
TP C2 mean:     0.321288209606987
AUC mean:   0.8141324312418602
======== R2_28Smote ======
TP C1 mean:     0.9592827004219409
TP C2 mean:     0.29828042328042337
AUC mean:   0.8410146046815351
======== R3_28Smote ======
TP C1 mean:     0.9927017744705209
TP C2 mean:     0.11648936170212768
AUC mean:   0.8353633584625314
======== R4_28Smote ======
TP C1 mean:     0.9887545033453422
TP C2 mean:     0.17554347826086955
AUC mean:   0.8023516693145964
======== R1_28 ======
TP C1 mean:     0.9930994152046783
TP C2 mean:     0.08733624454148471
AUC mean:   0.8001448581424448
======== R2_28 ======
TP C1 mean:     0.9940928270042194
TP C2 mean:     0.06349206349206349
AUC mean:   0.8184781941374769
======== R3_28 ======
TP C1 mean:     0.9970807097882084
TP C2 mean:     0.010638297872340425
AUC mean:   0.7632555505486609
======== R4_28 ======
TP C1 mean:     0.9965517241379309
TP C2 mean:     0.02934782608695652
AUC mean:   0.7850094542281099
======== R1_42Smote ======
TP C1 mean:     0.9528070175438597
TP C2 mean:     0.3957423580786027
AUC mean:   0.850726094256748
======== R2_42Smote ======
TP C1 mean:     0.9620516877637131
TP C2 mean:     0.3648148148148148
AUC mean:   0.8665926107315876
======== R3_42Smote ======
TP C1 mean:     0.9912707498568976
TP C2 mean:     0.2367021276595745
AUC mean:   0.8395017598558014
======== R4_42Smote ======
TP C1 mean:     0.9840967575913536
TP C2 mean:     0.2529891304347826
AUC mean:   0.8587255113115084
======== R1_42 ======
TP C1 mean:     0.9888304093567252
```

```
TP C2 mean:     0.13449781659388646
AUC mean:   0.810464835669961
======== R2_42 ======
TP C1 mean:     0.9908755274261603
TP C2 mean:     0.10317460317460318
AUC mean:   0.8360220905052127
======== R3_42 ======
TP C1 mean:     0.9958214081282198
TP C2 mean:     0.03404255319148936
AUC mean:   0.7325007002886407
======== R4_42 ======
TP C1 mean:     0.9946217189912506
TP C2 mean:     0.0923913043478261
AUC mean:   0.7860516570073172
------------------------------ Multilayer Perceptron ------------------------------
======== R1_28Smote ======
TP C1 mean:     0.916374269005848
TP C2 mean:     0.40611353711790393
AUC mean:   0.8140518463188539
======== R2_28Smote ======
TP C1 mean:     0.9506856540084389
TP C2 mean:     0.32407407407407407
AUC mean:   0.8343881856540084
======== R3_28Smote ======
TP C1 mean:     0.994848311390956
TP C2 mean:     0.05851063829787234
AUC mean:   0.8393979344529833
======== R4_28Smote ======
TP C1 mean:     0.9858466289243438
TP C2 mean:     0.23097826086956522
AUC mean:   0.7716342388507239
======== R1_28 ======
TP C1 mean:     0.9953216374269006
TP C2 mean:     0.06550218340611354
AUC mean:   0.7914502413238336
======== R2_28 ======
TP C1 mean:     0.9997362869198312
TP C2 mean:     0.0
AUC mean:   0.811361429241176
======== R3_28 ======
TP C1 mean:     1.0
TP C2 mean:     0.0
AUC mean:   0.7353456990098528
======== R4_28 ======
TP C1 mean:     0.9971693257848687
TP C2 mean:     0.021739130434782608
AUC mean:   0.7272035623979055
======== R1_42Smote ======
TP C1 mean:     0.9230994152046783
TP C2 mean:     0.4497816593886463
AUC mean:   0.8294273027911847
======== R2_42Smote ======
TP C1 mean:     0.943301687763713
TP C2 mean:     0.421957671957672
AUC mean:   0.8608892572500167
======== R3_42Smote ======
TP C1 mean:     0.9894104178591872
TP C2 mean:     0.28191489361702127
AUC mean:   0.8237784530319453
======== R4_42Smote ======
TP C1 mean:     0.9825012866700978
TP C2 mean:     0.2826086956521739
AUC mean:   0.8276931963122917
======== R1_42 ======
TP C1 mean:     0.997953216374269
TP C2 mean:     0.034934497816593885
AUC mean:   0.801389208100309
======== R2_42 ======
TP C1 mean:     0.9978902953586498
TP C2 mean:     0.05291005291005291
AUC mean:   0.8000273480231286
```

```
======== R3_42 ======
TP C1 mean:     0.9994275901545506
TP C2 mean:     0.0
AUC mean:   0.7737215165207224
======== R4_42 ======
TP C1 mean:     0.9971693257848687
TP C2 mean:     0.03260869565217391
AUC mean:   0.7697364004564882
------------------------------- C4.5 -------------------------------
======== R1_28Smote ======
TP C1 mean:     0.9090350877192982
TP C2 mean:     0.5729257641921397
AUC mean:   0.8000370604458744
======== R2_28Smote ======
TP C1 mean:     0.932331223628692
TP C2 mean:     0.5981481481481482
AUC mean:   0.8004480331748264
======== R3_28Smote ======
TP C1 mean:     0.977819118488838
TP C2 mean:     0.4207446808510638
AUC mean:   0.739235954645654
======== R4_28Smote ======
TP C1 mean:     0.9737519300051465
TP C2 mean:     0.5353260869565217
AUC mean:   0.7776599806440063
======== R1_28 ======
TP C1 mean:     0.9906432748538012
TP C2 mean:     0.05458515283842794
AUC mean:   0.6364167879670063
======== R2_28 ======
TP C1 mean:     0.9917194092827005
TP C2 mean:     0.06507936507936507
AUC mean:   0.6736077623735852
======== R3_28 ======
TP C1 mean:     1.0
TP C2 mean:     0.0
AUC mean:   0.42966392234712386
======== R4_28 ======
TP C1 mean:     0.9974523932063819
TP C2 mean:     0.019565217391304346
AUC mean:   0.5404456353912597
======== R1_42Smote ======
TP C1 mean:     0.9425438596491229
TP C2 mean:     0.7209606986899564
AUC mean:   0.852568630455323
======== R2_42Smote ======
TP C1 mean:     0.9610495780590715
TP C2 mean:     0.7175925925925926
AUC mean:   0.8609308374076307
======== R3_42Smote ======
TP C1 mean:     0.9862335432169435
TP C2 mean:     0.5494680851063831
AUC mean:   0.7985078523669756
======== R4_42Smote ======
TP C1 mean:     0.9786155429747814
TP C2 mean:     0.6293478260869565
AUC mean:   0.8249255619391797
======== R1_42 ======
TP C1 mean:     0.9768713450292397
TP C2 mean:     0.14104803493449783
AUC mean:   0.600381968385301
======== R2_42 ======
TP C1 mean:     0.9832014767932489
TP C2 mean:     0.15343915343915343
AUC mean:   0.611711930435559
======== R3_42 ======
TP C1 mean:     0.9988838008013738
TP C2 mean:     0.002127659574468085
AUC mean:   0.4773748919119707
======== R4_42 ======
TP C1 mean:     0.9936953165208442
```

```
TP C2 mean:     0.05652173913043479
AUC mean:   0.546265579896619
------------------------------ Random Forest -------------------------------
======== R1_28Smote ======
TP C1 mean:     0.9669590643274854
TP C2 mean:     0.5938864628820961
AUC mean:   0.9415632421665517
======== R2_28Smote ======
TP C1 mean:     0.9754746835443038
TP C2 mean:     0.5648148148148148
AUC mean:   0.9541845614828209
======== R3_28Smote ======
TP C1 mean:     0.9991413852318259
TP C2 mean:     0.2712765957446808
AUC mean:   0.9680417189345869
======== R4_28Smote ======
TP C1 mean:     0.9956253216675245
TP C2 mean:     0.4375
AUC mean:   0.9572668190158652
======== R1_28 ======
TP C1 mean:     0.9941520467836257
TP C2 mean:     0.03056768558951965
AUC mean:   0.7616518552567737
======== R2_28 ======
TP C1 mean:     0.9984177215189873
TP C2 mean:     0.026455026455026454
AUC mean:   0.8083866340722881
======== R3_28 ======
TP C1 mean:     1.0
TP C2 mean:     0.0
AUC mean:   0.6928716705842234
======== R4_28 ======
TP C1 mean:     1.0
TP C2 mean:     0.0
AUC mean:   0.7549872451833785
======== R1_42Smote ======
TP C1 mean:     0.97953216374269
TP C2 mean:     0.732532751091703
AUC mean:   0.9656590758190965
======== R2_42Smote ======
TP C1 mean:     0.9825949367088608
TP C2 mean:     0.7711640211640212
AUC mean:   0.9741498392605987
======== R3_42Smote ======
TP C1 mean:     0.9994275901545506
TP C2 mean:     0.4946808510638298
AUC mean:   0.9765304351532719
======== R4_42Smote ======
TP C1 mean:     0.9948533196088523
TP C2 mean:     0.6005434782608695
AUC mean:   0.9765007887847121
======== R1_42 ======
TP C1 mean:     0.9947368421052631
TP C2 mean:     0.048034934497816595
AUC mean:   0.7968046936847213
======== R2_42 ======
TP C1 mean:     0.9981540084388185
TP C2 mean:     0.07936507936507936
AUC mean:   0.8393478054606747
======== R3_42 ======
TP C1 mean:     1.0
TP C2 mean:     0.0
AUC mean:   0.7474393793615803
======== R4_42 ======
TP C1 mean:     0.9997426659804426
TP C2 mean:     0.0
AUC mean:   0.8029310904249368
```