A List of WEKA Parameters & Functions used to Calculate Metrics

By: Noureddin Sadawi

19 Feb 2014

1 Params:

This is a list of parameters (variables) that WEKA uses for evaluation. They are computed/required for functions which are used to generate measurements/metrics for evaluation.

Notice that a definition has been provided for each of them (definitions were extracted from WEKA's source files). Also, notice that if there is one or more numbers, those numbers refer to other parameters in the list and that means this parameter requires those parameters to be computed first, hence it is dependenet on them and therefore it is a secondary parameter.

1. int m_NumClasses:
   The number of classes
2. double m_WithClass:
   The weight of all instances that had a class assigned to them
3. double m_Incorrect:
   The weight of all incorrectly classified instances
4. double m_Correct:
   The weight of all incorrectly classified instances.
5. double m_TotalCost:
   The weight of all incorrectly classified instances. 8
6. double m_Unclassified:
   The weight of all unclassified instances
7. double[][] m_ConfusionMatrix:
   Array for storing the confusion matrix 1, 31
8. CostMatrix m_CostMatrix:
   The cost matrix (if given)
9. int classIndex:
   classIndex the index of the class
10. FastVector m_Predictions:
    The list of predictions that have been generated (for computing AUC)

11. double m_SumPriorEntropy:
    Total entropy of prior predictions 27

12. double m_SumSchemeEntropy:
    Total entropy of scheme predictions 29

13. double m_SumAbsErr:
    Sum of absolute errors 30,1

14. double m_SumPriorAbsErr:
    Sum of absolute errors of the prior 30,1

15. double m_SumSqrErr:
    Sum of squared errors 30,1

16. double m_SumPriorSqrErr:
    Sum of absolute errors of the prior 30,1

17. double m_SumKBInfo:
    Total Kononenko & Bratko Information 27, 28, 31

18. double[] m_ClassPriors:
    The prior probabilities of the classes 33, 34

19. double m_ClassPriorsSum:
    The sum of counts for priors 34

20. double m_SumSqrClass:
    Sum of squared class values 31,35

21. double m_SumClass:
    Sum of class values 31,35

22. double m_SumSqrPredicted:
    Sum of squared predicted values 31,36

23. double m_SumPredicted:
    Sum of predicted values 31,36

24. double m_SumClassPredicted:
    Sum of predicted class values 31,35, 36

25. double m_TotalCoverage:
    Total coverage of test cases at the given confidence level 31

26. double m_TotalSizeOfRegions:
    Total size of predicted regions at the given confidence level 37, 38, 39

27. priorProb :
    18, 19

28. d[] predictedDistribution:
    the probabilities assigned to each class

29. predictedProb:
    28

30. weight:
    the weight associated with this prediction (For updating the numeric accuracy measures. For numeric classes, the accuracy is between the actual and predicted class values. For nominal classes, the accuracy is between the actual and predicted class probabilities)

31. tst_InstanceWeight:
    the weight associated with the test instance to be classified

32. double m_SumErr:
    Sum of errors 30, 1

33. tr_InstanceClassValue:
    the class value of Training Instance

34. tr_InstanceWeight:
    the weight associated with the training instance

35. tst_InstanceClassValue:
    the class value of Test Instance

36. predictedValue:
    the numeric value the classifier predicts

37. double sizeOfRegions:
    28, 40

38. double m_MaxTarget:
    Maximum target Class value (numeric training class). Could be # of Classes

39. double m_MinTarget:
    Minimum target Class value (numeric training class)

40. double m_ConfLevel = 0.95:
    The confidence level used for coverage statistics

2 Functions:

This is a list of functions that WEKA uses to generate measurements/metrics for evaluation. These functions use the parameters explained in the previous section.

Notice that a description has been provided for each of these functions and they have been categorised (description and categorisation were extracted from WEKA's source files).

Also, notice that if there is one or more numbers, those numbers refer to the parameters that these functions use. If there is an "=" sign, this means the function just returns that parameter rather than using it to compute something. Whenever the letter "F" is used, it means that particular function uses/requires another function (with the number of the required/used function following the "F"). This indicates that the function is dependenet on other function(s) and therefore it is a secondary function.

1. **Basic performance stats - right vs wrong**
   (a) **correct()** =4
   Gets the number of instances correctly classified (that is, for which a correct prediction was made).
   (b) **pctCorrect()** 2,4
   Gets the percentage of instances correctly classified (that is, for which a correct prediction was made).
   (c) **pctIncorrect()**2,3
   Gets the percentage of instances incorrectly classified (that is, for which an incorrect prediction was made).
   (d) **pctUnclassified()**2,6
   Gets the percentage of instances not classified (that is, for which no prediction was made by the classifier).
   (e) **incorrect()** =3
   Gets the number of instances incorrectly classified (that is, for which an incorrect prediction was made).
   (f) **kappa()** 7
   Returns value of kappa statistic if class is nominal.
   (g) **unclassified()** =6
   Gets the number of instances not classified (that is, for which no prediction was made by the classifier).
2. **IR stats**
   (a) **areaUnderROC(int classIndex)** 1,7,9,10
   Returns the area under ROC for those predictions that have been collected in the evaluateClassifier(Classifier, Instances) method.
   (b) **falseNegativeRate(int classIndex)** 1,7,9
   Calculate the false negative rate with respect to a particular class.
   (c) **falsePositiveRate(int classIndex)** 1,7,9
   Calculate the false positive rate with respect to a particular class.
   (d) **fMeasure(int classIndex)** F2i, F2j, 9
   Calculate the F-Measure with respect to a particular class.

(e) **numTrueNegatives(int classIndex)** 1,7,9
Calculate the number of true negatives with respect to a particular class.

(f) **numTruePositives(int classIndex)** 1,7,9
Calculate the number of true positives with respect to a particular class.

(g) **numFalseNegatives(int classIndex)** 1,7,9
Calculate number of false negatives with respect to a particular class.

(h) **numFalsePositives(int classIndex)** 1,7,9
Calculate number of false positives with respect to a particular class.

(i) **precision(int classIndex)** 1,7,9
Calculate the precision with respect to a particular class.

(j) **recall(int classIndex)** =F2l
Calculate the recall with respect to a particular class.

(k) **trueNegativeRate(int classIndex)** 1,7,9
Calculate the true negative rate with respect to a particular class.

(l) **truePositiveRate(int classIndex)** 1,7,9
Calculate the true positive rate with respect to a particular class.

3. **Weighted IR stats**

(a) **weightedAreaUnderROC()** F2a,1,7
Calculates the weighted (by class size) AUC.

(b) **weightedFalseNegativeRate()** F2b, 1,7
Calculates the weighted (by class size) false negative rate.

(c) **weightedFalsePositiveRate()** F2c, 1,7
Calculates the weighted (by class size) false positive rate.

(d) **weightedFMeasure()**F2d, 1,7
Calculates the macro weighted (by class size) average F-Measure.

(e) **weightedPrecision()** F2i, 1,7
Calculates the weighted (by class size) precision.

(f) **weightedRecall()** = F3h
Calculates the weighted (by class size) recall.

(g) **weightedTrueNegativeRate()** F2k, 1,7
Calculates the weighted (by class size) true negative rate.

(h) **weightedTruePositiveRate()** F2l, 1,7
Calculates the weighted (by class size) true positive rate.

4. **SF stats**

(a) **SFEntropyGain()** 11, 12
Returns the total SF, which is the null model entropy minus the scheme entropy.

(b) **SFMeanEntropyGain()** 2, 6, 11, 12
Returns the SF per instance, which is the null model entropy minus the scheme entropy, per instance.

(c) **SFMeanPriorEntropy()** 2, 11
Returns the entropy per instance for the null model.

(d) **SFMeanSchemeEntropy()** 2, 6, 12
Returns the entropy per instance for the scheme

(e) **SFPriorEntropy()** = 11
Returns the total entropy for the null model.

(f) **SFSchemeEntropy()** = 12
Returns the total entropy for the scheme.

5. **Sensitive stats - certainty of predictions**

(a) **relativeAbsoluteError()** F5d, F15
Returns the relative absolute error.

(b) **rootMeanSquaredError()** 2, 6, 15
Returns the root mean squared error.

(c) **rootRelativeSquaredError()** F5b, F18
Returns the root relative squared error if the class is numeric.

(d) **meanAbsoluteError()** 2, 6, 13
Returns the mean absolute error.

6. **K&B stats**

(a) **KBInformation()** = 17
Return the total Kononenko & Bratko Information score in bits.

(b) **KBMeanInformation()** 2, 6, 17
Return the Kononenko & Bratko Information score in bits per instance.

(c) **KBRelativeInformation()** F6a, F17
Return the Kononenko & Bratko Relative Information score.

7. **areaUnderPRC(int classIndex)** 9, 10
Returns the area under precision-recall curve (AUPRC) for those predictions that have been collected in the evaluateClassifier(Classifier, Instances) method.

8. **avgCost()** 5, 2
Gets the average cost, that is, total cost of misclassifications (incorrect plus unclassified) over the total number of instances.

9. **confusionMatrix()**
Returns a copy of the confusion matrix.

10. **correlationCoefficient()** 2, 6, 20, 21, 22, 23, 24
Returns the correlation coefficient if the class is numeric.

11. **coverageOfTestCasesByPredictedRegions()** 2, 25
Gets the coverage of the test cases by the predicted regions at the confidence level specified when evaluation was performed.

12. **errorRate()** 2,3,6,15 OR = F8
Returns the estimated error rate or the root mean squared error (if the class is numeric).

13. **getClassPriors()** = 18
Get the current weighted class counts.

14. **matthewsCorrelationCoefficient(int classIndex)** 9, F2e, F2f, F2g, F2h
Calculates the matthews correlation coefficient (sometimes called phi coefficient) for the supplied class

15. **meanPriorAbsoluteError()** 2, 14
    Returns the mean absolute error of the prior.

16. **numInstances()** = 2
    Gets the number of test instances that had a known class value (actually the sum of the weights of test instances with known class values

17. **priorEntropy()** 1, 18, 19
    Calculate the entropy of the prior distribution.

18. **rootMeanPriorSquaredError()** 2, 16
    Returns the root mean prior squared error.

19. **setMetricsToDisplay(java.util.List¡java.lang.String¿ display)**
    Set a list of the names of metrics to have appear in the output.

20. **sizeOfPredictedRegions()** 2, 26
    Gets the average size of the predicted regions, relative to the range of the target in the training data, at the confidence level specified when evaluation was performed

21. **totalCost()** = 5
    Gets the total cost, that is, the cost of each prediction times the weight of the instance, summed over all instances.

22. **unweightedMacroFmeasure()** 1, F2f, F2g
    Unweighted macro-averaged F-measure.

23. **unweightedMicroFmeasure()** 1, F2f, F2g, F2h
    Unweighted micro-averaged F-measure.

24. **weightedAreaUnderPRC()** 1, 7, F7
    Calculates the weighted (by class size) AUPRC.

25. **weightedMatthewsCorrelation()** 1, 7, F14
    Calculates the weighted (by class size) matthews correlation coefficient.

26. Number_of_training_instances

27. Number_of_testing_instances

28. Elapsed_Time_training

29. Elapsed_Time_testing

30. UserCPU_Time_training

31. UserCPU_Time_testing

32. Serialized_Model_Size

33. Serialized_Train_Set_Size

34. Serialized_Test_Set_Size