OPTION 1: SUPERVISED DATA MINING

TERM PROJECT REPORT

Course: CS634

Professor: Dr. Jason Wang

Presented By: Kavita Sharma

UCID: ks689

Email Id: ks689@njit.edu

Table of Contents

INTRODUCTION

In Supervised Data Mining , the data we used comes with additional attributes that we want to predict.

In this project the data used is taken from one of the most popular Kaggle Challenges 'Titanic: Machine Learning from Disaster'. The RMS titanic is one of the most infamous sinking in the history which killed most of the passengers. An evaluation and comparison between different parameters of Logistic Regression Algorithm and the Decision Tree Algorithm will be done in this report.

LOGISTIC REGRESSION

Logistic Regression is a type of regression analysis used for predicting the outcome of a dependent variable based on the other variables of the dataset. Logistic regression not only predicts the outcome but provides the probability of that outcome being correct i.e. in Logistic Regression the output is interpreted as a probability.

DECISION TREE

Decision Tree is a method to predict the outcome of a target variable by inferring certain decision rules from data features. Decision Trees are very flexible and able to describe the complicated relationships. We split the data according to one attribute and end up with subsets until we have a single outcome.

DATA SET USED

OVERVIEW

The data has been split into two groups:

- training set (train.csv)
- test set (test.csv)

The training set should be used to build your machine learning models. For the training set, we provide the outcome (also known as the "ground truth") for each passenger. Your model will be based on "features" like passengers' gender and class. You can also use feature engineering to create new features.

The test set should be used to see how well your model performs on unseen data. For the test set, we do not provide the ground truth for each passenger. It is your job to predict these outcomes. For each passenger in the test set, use the model you trained to predict whether or not they survived the sinking of the Titanic.

We also include gender_submission.csv, a set of predictions that assume all and only female passengers survive, as an example of what a submission file should look like.

DATA DICTIONARY

Variable	Definition	Key
survival	Survival	0 = No, 1 = Yes

pclass	Ticket class	1 = 1st, 2 = 2nd, 3 = 3rd
sex	Sex	
Age	Age in years	
sibsp	# of siblings / spouses aboard the Titanic	
parch	# of parents / children aboard the Titanic	
ticket	Ticket number	
fare	Passenger fare	
cabin	Cabin number	
embarked	Port of Embarkation	C = Cherbourg, Q = Queenstown, S = Southampton

Variable Notes

pclass: A proxy for socio-economic status (SES)
1st = Upper
2nd = Middle
3rd = Lower

age: Age is fractional if less than 1. If the age is estimated, is it in the form of xx.5

sibsp: The dataset defines family relations in this way...
Sibling = brother, sister, stepbrother, stepsister
Spouse = husband, wife (mistresses and fiancés were ignored)

parch: The dataset defines family relations in this way...
Parent = mother, father
Child = daughter, son, stepdaughter, stepson
Some children travelled only with a nanny, therefore parch=0 for them.

Reference link - https://www.kaggle.com/c/titanic/data

DESCRIPTION

1. **Project Option 1 :** Supervised Data Mining – Classification
2. **Algorithms Chosen:** Category 3 – Decision Trees (J48)
   Category 12 – Regression Analysis (Logistic)
3. **Software Used:** WEKA Tool version 3.8.1
4. **Data taken from:** http://www.hakank.org/weka/titanic.arff
5. **Data Source:** https://www.kaggle.com/c/titanic/data

PURPOSE

The main purpose of this project is to predict the survival of the Titanic passengers based on different attributes such as Age, Sex , Class etc. using Logistic Regression Model and Decision Tree Classifier and compare the two algorithms for which gave the better prediction results.

For each algorithm, I will be analyzing the difference in prediction values for each parameter and for different test options in the WEKA tool Explorer and then compare both algorithms in the WEKA Experimenter.

ALGORTITHM DESCRIPTIONS

LOGISTIC REGRESSION

Logistic regression is a statistical method for analyzing a dataset in which there are one or more independent variables that determine an outcome. The outcome is measured with a dichotomous variable (in which there are only two possible outcomes).

> **In logistic regression, the dependent variable is binary or dichotomous, i.e. it only contains data coded as 1 (TRUE, success, pregnant, etc.) or 0 (FALSE, failure, non-pregnant, etc.).**

The goal of logistic regression is to find the best fitting (yet biologically reasonable) model to describe the relationship between the dichotomous characteristic of interest (dependent variable = response or outcome variable) and a set of independent (predictor or explanatory) variables. Logistic regression generates the coefficients (and its standard errors and significance levels) of a formula to predict a *logit transformation* of the probability of presence of the characteristic of interest:

$$logit(p) = b_0 + b_1X_1 + b_2X_2 + b_3X_3 + \ldots + b_kX_k$$

where p is the probability of presence of the characteristic of interest. The logit transformation is defined as the logged odds:

$$odds = \frac{p}{1-p} = \frac{probability\ of\ presence\ of\ characteristic}{probability\ of\ absence\ of\ characteristic}$$

and

$$logit(p) = \ln\left(\frac{p}{1-p}\right)$$

Rather than choosing parameters that minimize the sum of squared errors (like in ordinary regression), estimation in logistic regression chooses parameters that maximize the likelihood of observing the sample values.

Reference link - https://www.medcalc.org/manual/logistic_regression.php

C4.5 DECISION TREE

The decision trees generated by C4.5 can be used for classification, and for this reason, C4.5 is often referred to as a statistical classifier. At each node of the tree, C4.5 chooses the attribute of the data that most effectively splits its set of samples into subsets enriched in one class or the other. The splitting criterion is the normalized information gain (difference in entropy). The attribute with the highest normalized information gain is chosen to make the decision. The C4.5 algorithm then recurs on the smaller sublists.

This algorithm has a few base cases.

- All the samples in the list belong to the same class. When this happens, it simply creates a leaf node for the decision tree saying to choose that class.
- None of the features provide any information gain. In this case, C4.5 creates a decision node higher up the tree using the expected value of the class.
- Instance of previously-unseen class encountered. Again, C4.5 creates a decision node higher up the tree using the expected value.

In pseudocode, the general algorithm for building decision trees is:[4]

1. Check for the above base cases.
2. For each attribute a, find the normalized information gain ratio from splitting on a.
3. Let a_best be the attribute with the highest normalized information gain.
4. Create a decision node that splits on a_best.
5. Recur on the sublists obtained by splitting on a_best, and add those nodes as children of node.

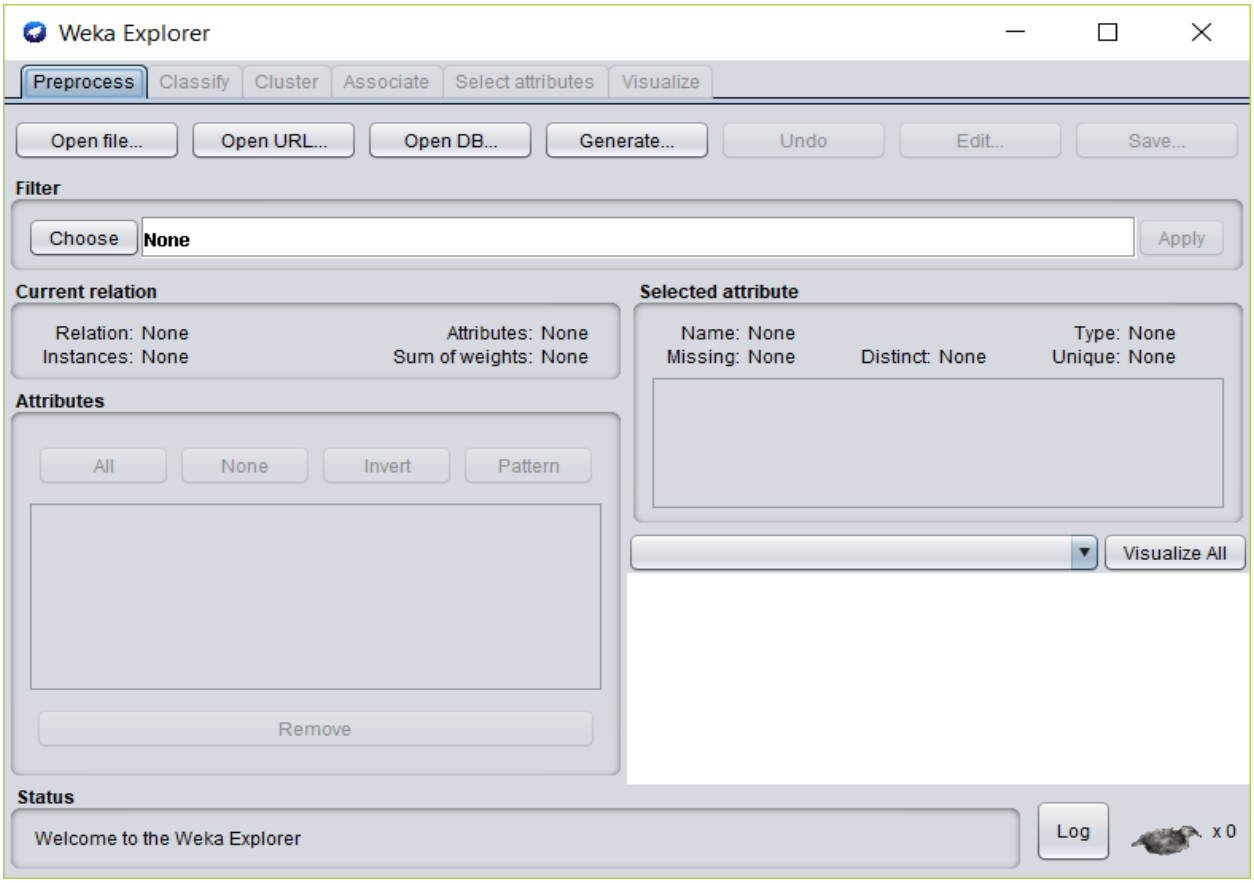
Reference link - https://en.wikipedia.org/wiki/C4.5_algorithm

STEP BY STEP PROCESS FOR USING WEKA:

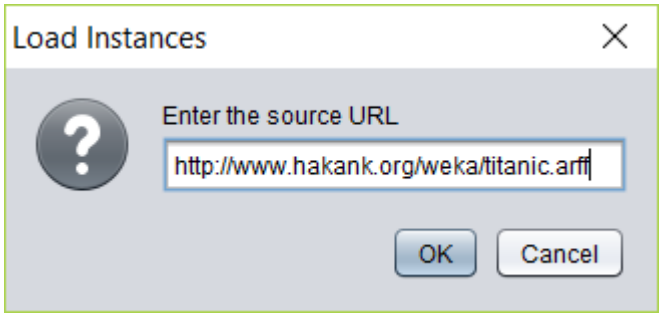
Following are the common steps to use WEKA for our analysis for both the Algorithms:

1. Install the WEKA tool from https://www.cs.waikato.ac.nz/ml/weka/downloading.html
2. Open the WEKA Desktop App

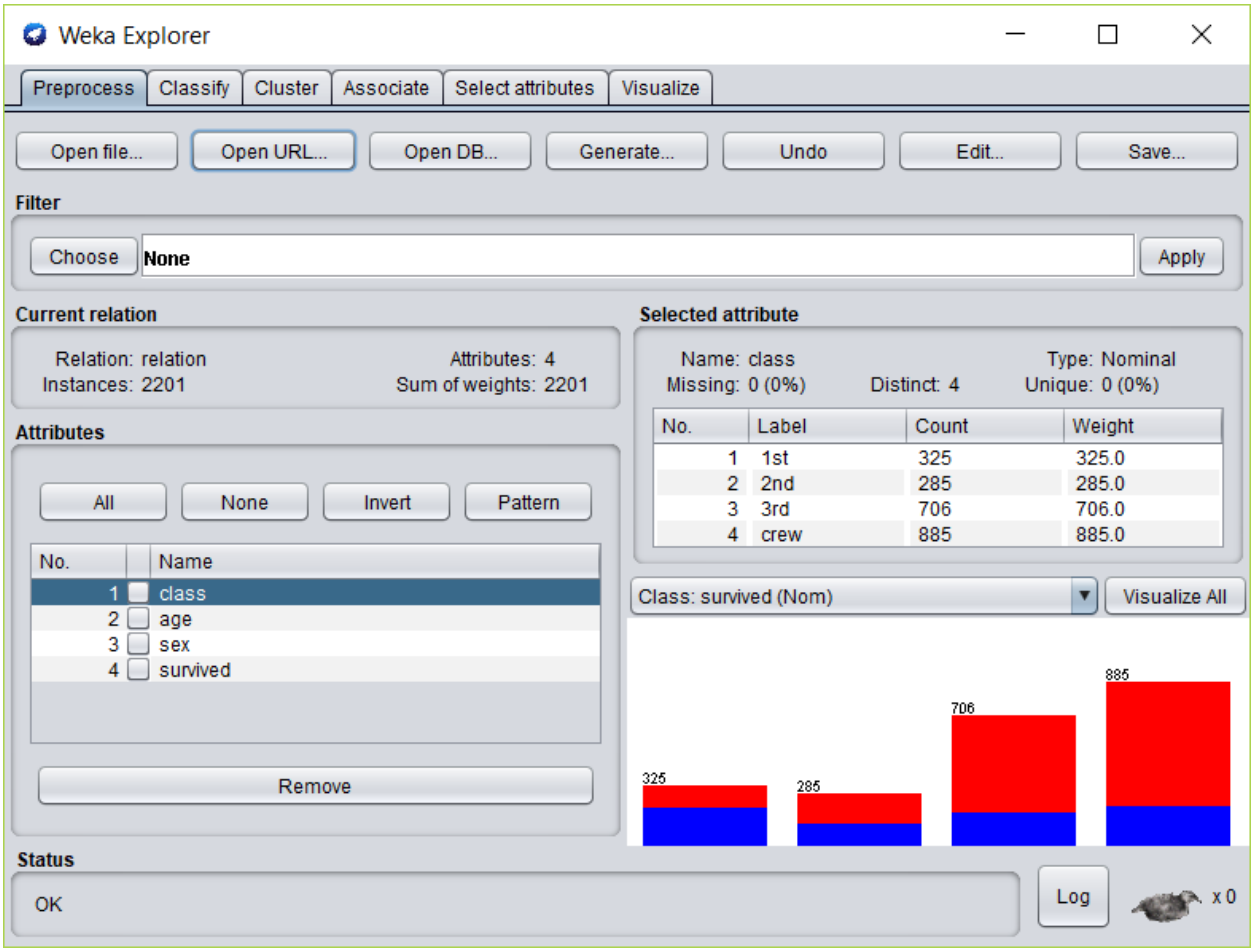
3. Click on the Explorer Option. A Weka Explorer Window will open.



4. Click on option Open URL to load the required dataset and enter the URL http://www.hakank.org/weka/titanic.arff as shown below.

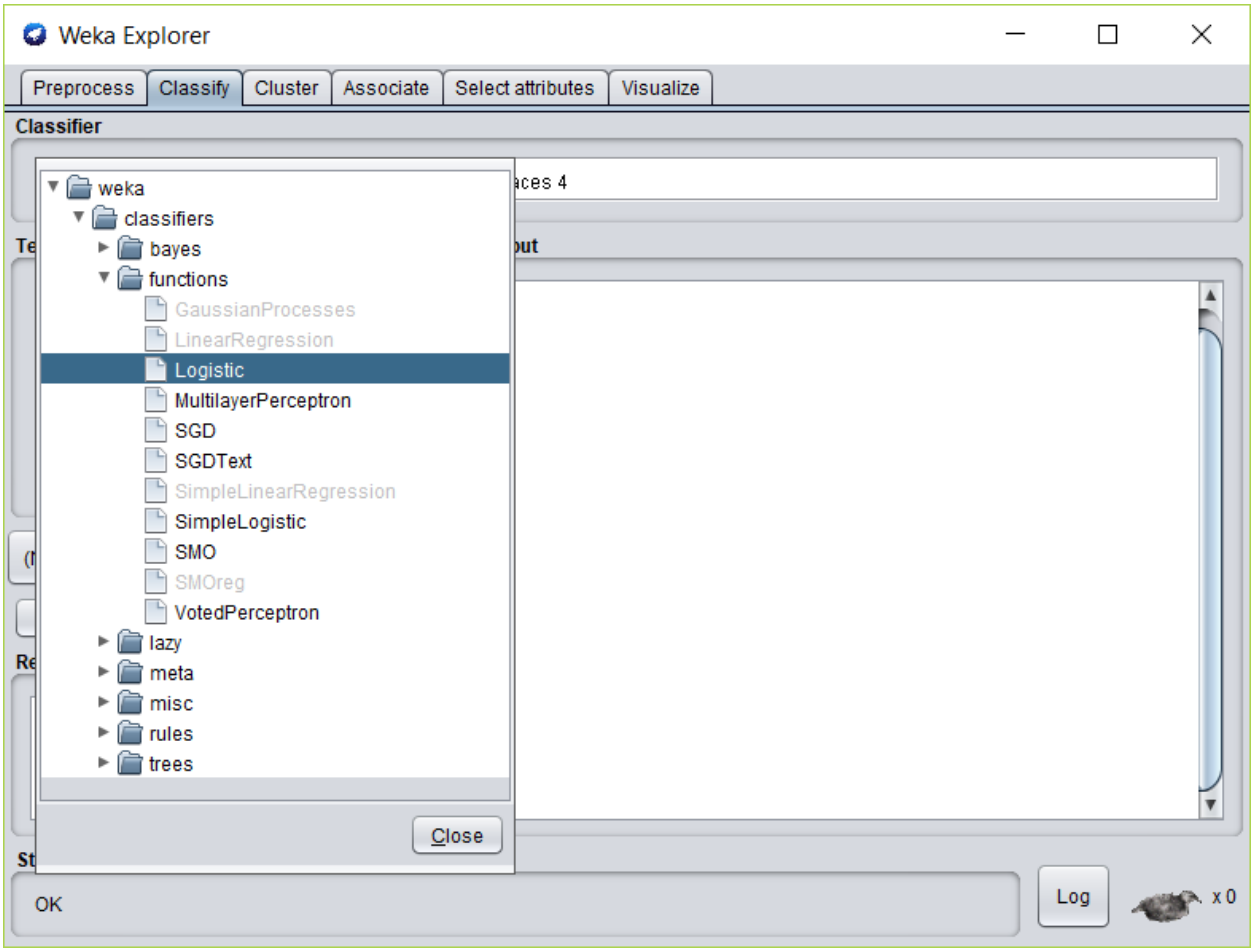


5. Click on OK. The Data will be loaded.


Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Open file...
Open URL...
Open DB...
Generate...
Undo
Edit...
Save...
Filter
Choose
None
Apply
Current relation
Relation: relation
Instances: 2201
Attributes: 4
Sum of weights: 2201
Attributes
All
None
Invert
Pattern
No.
Name
1 class
2 age
3 sex
4 survived
Remove
Selected attribute
Name: class
Missing: 0 (0%)
Distinct: 4
Type: Nominal
Unique: 0 (0%)
No.
Label
Count
Weight
1 1st 325 325.0
2 2nd 285 285.0
3 3rd 706 706.0
4 crew 885 885.0
Class: survived (Nom)
Visualize All
325
285
706
885
Status
OK
Log
x 0

LOGISTIC REGRESSION STEPS

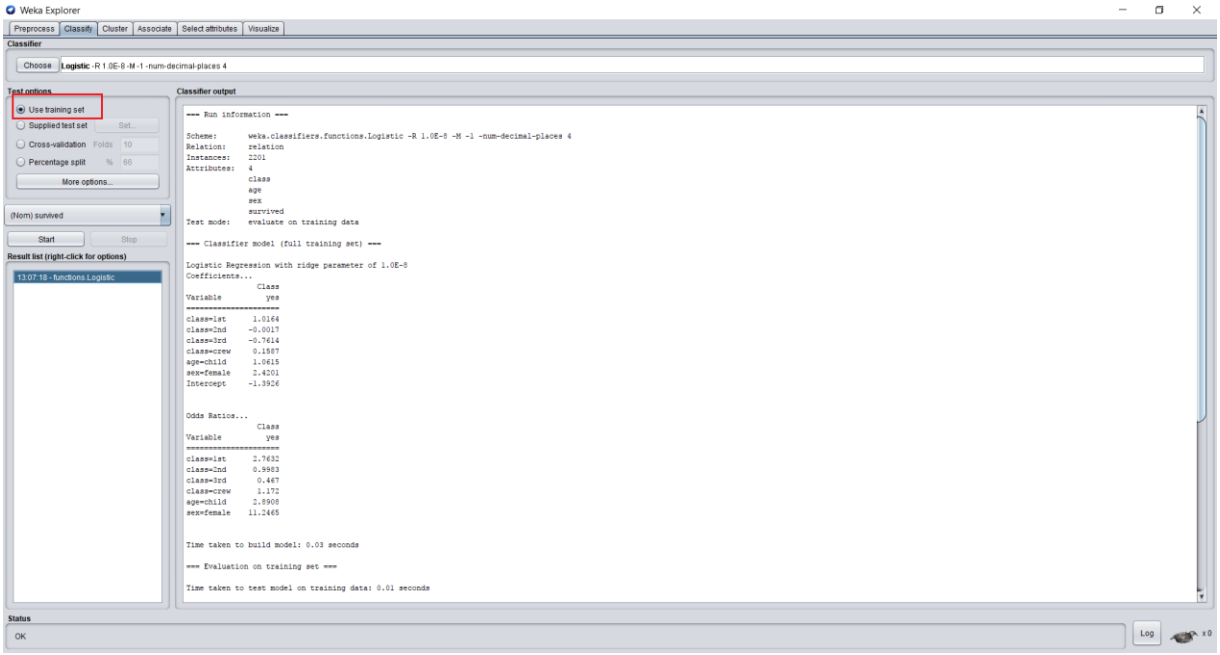
1. Go to ‘Classify’ tab and select Logistic under the Function Menu.

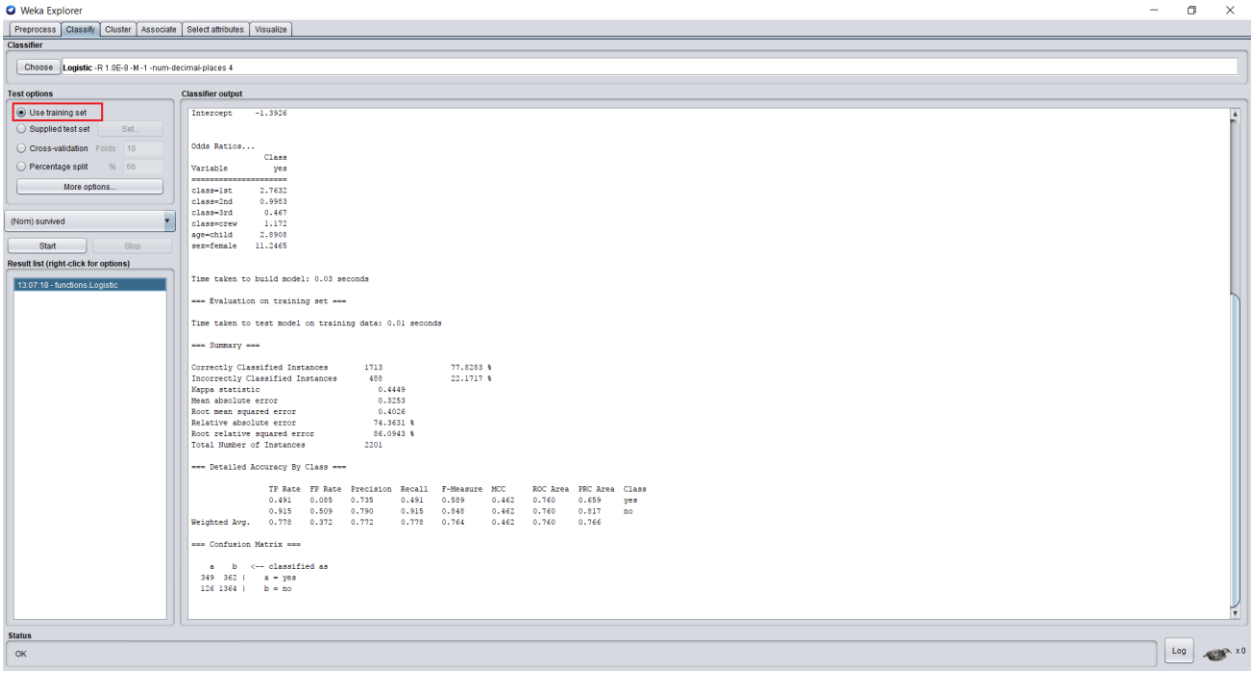


2. Select the ‘Use Training set’ option and click Start. The run would be successfully completed and a Classifier output would be shown as below.
3. Repeat the above step for the Test options ‘cross-validation’ and ‘percentage-splits’. The results would be as shown.

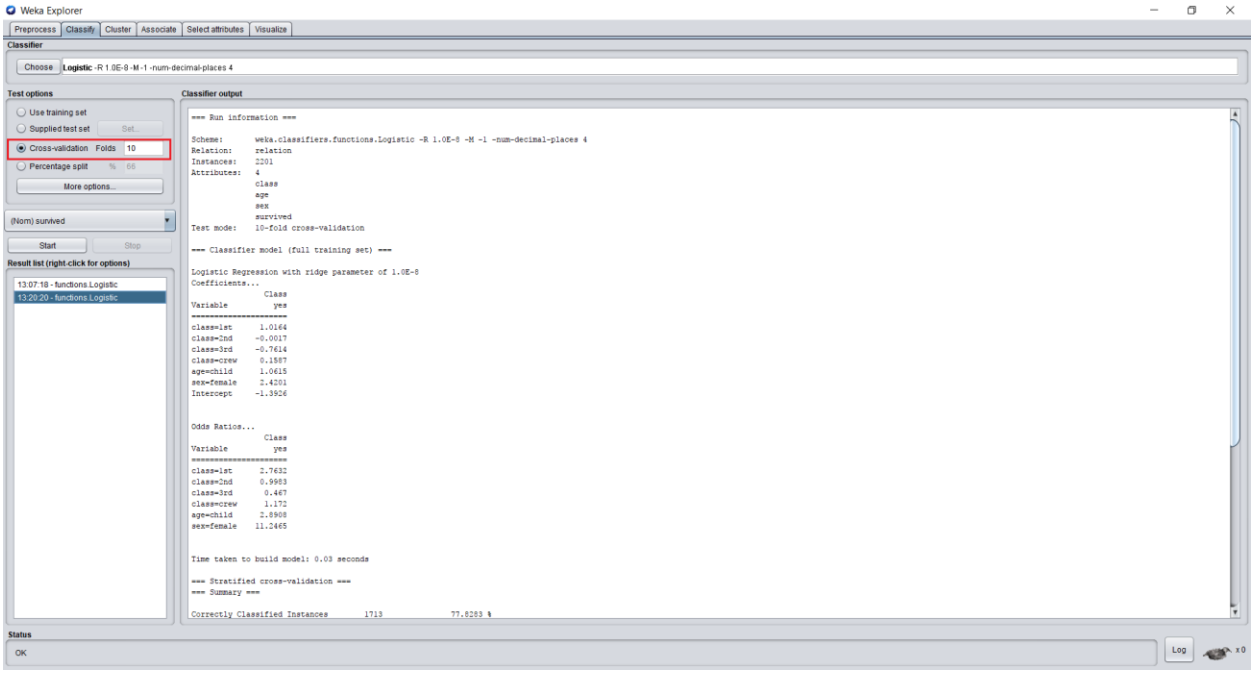
COMPLETE CLASSIFIER OUPUT FOR DIFFERENT TEST OPTIONS

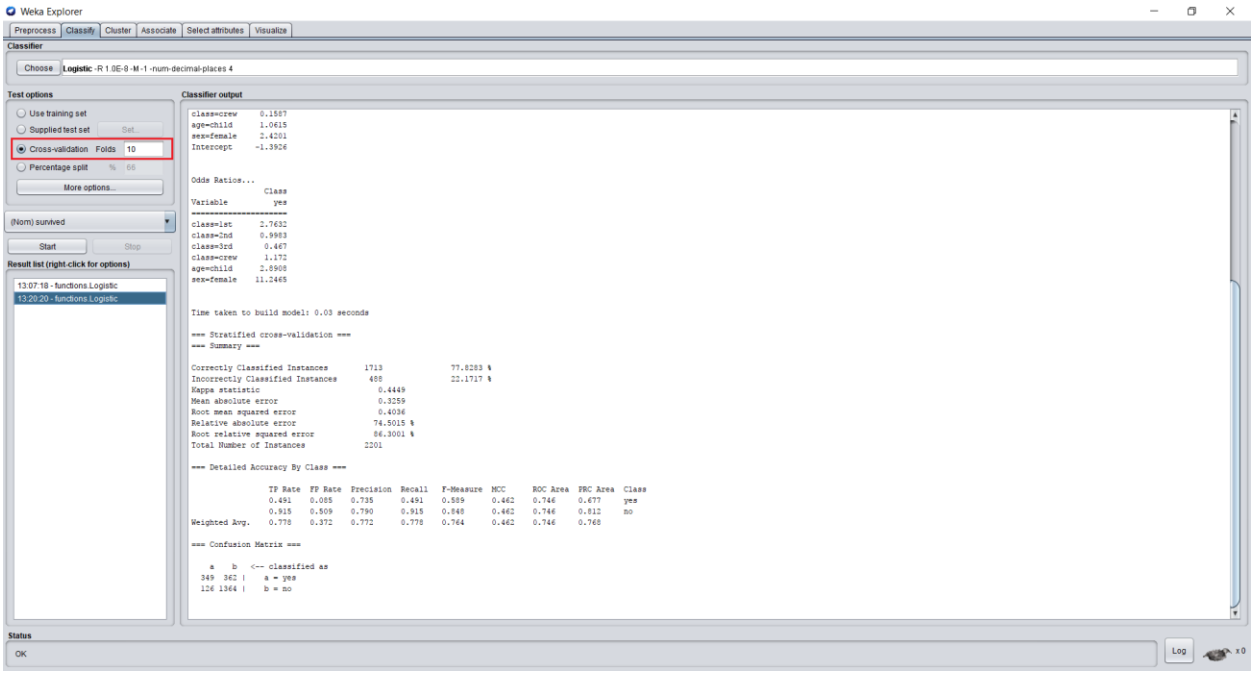
Test Option : Use training set



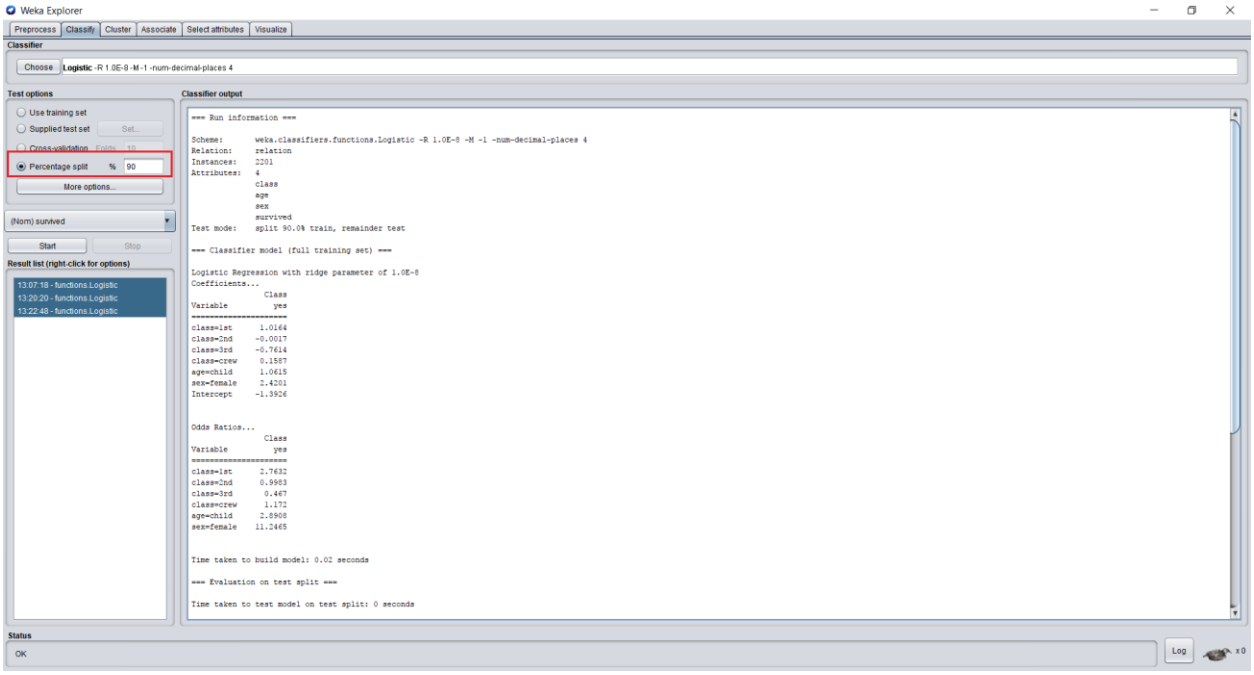


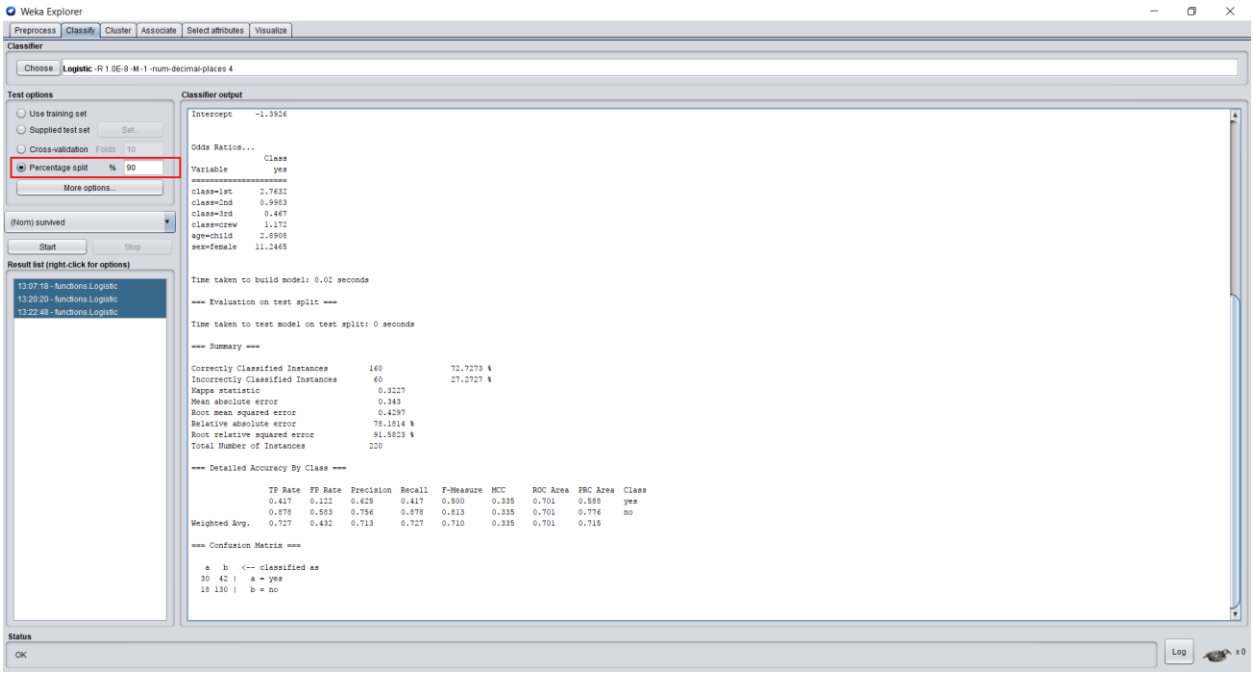
Test Option : Cross-validation (10-fold)



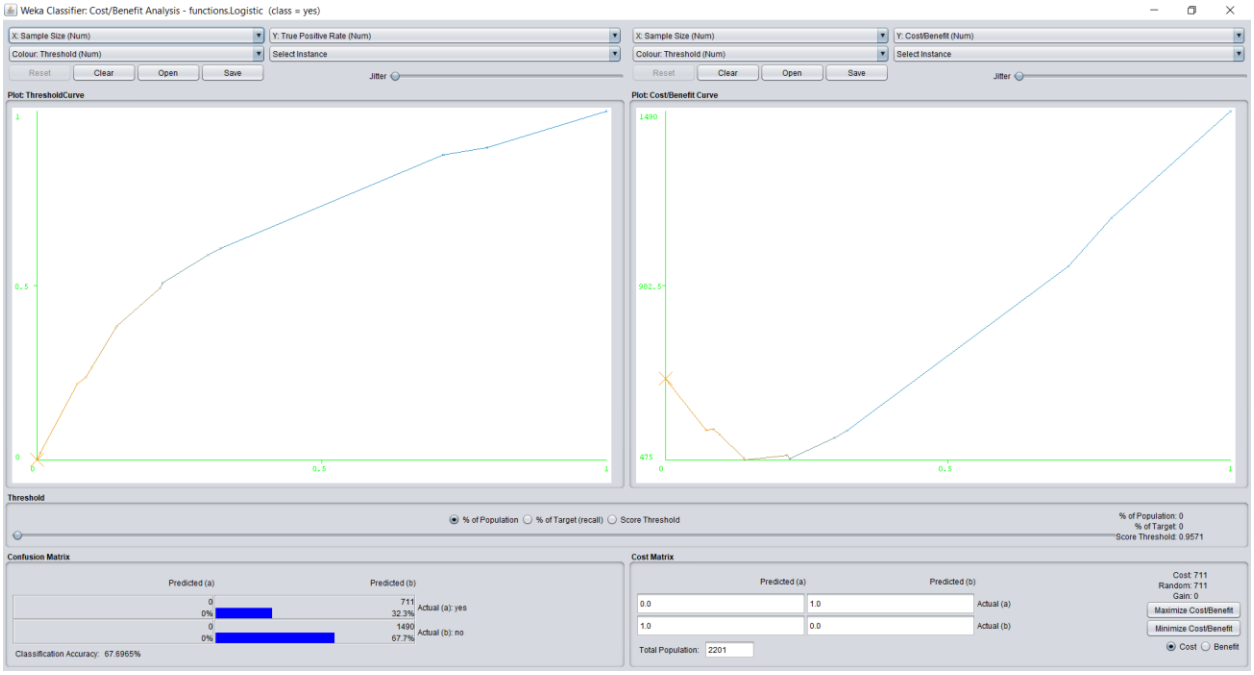


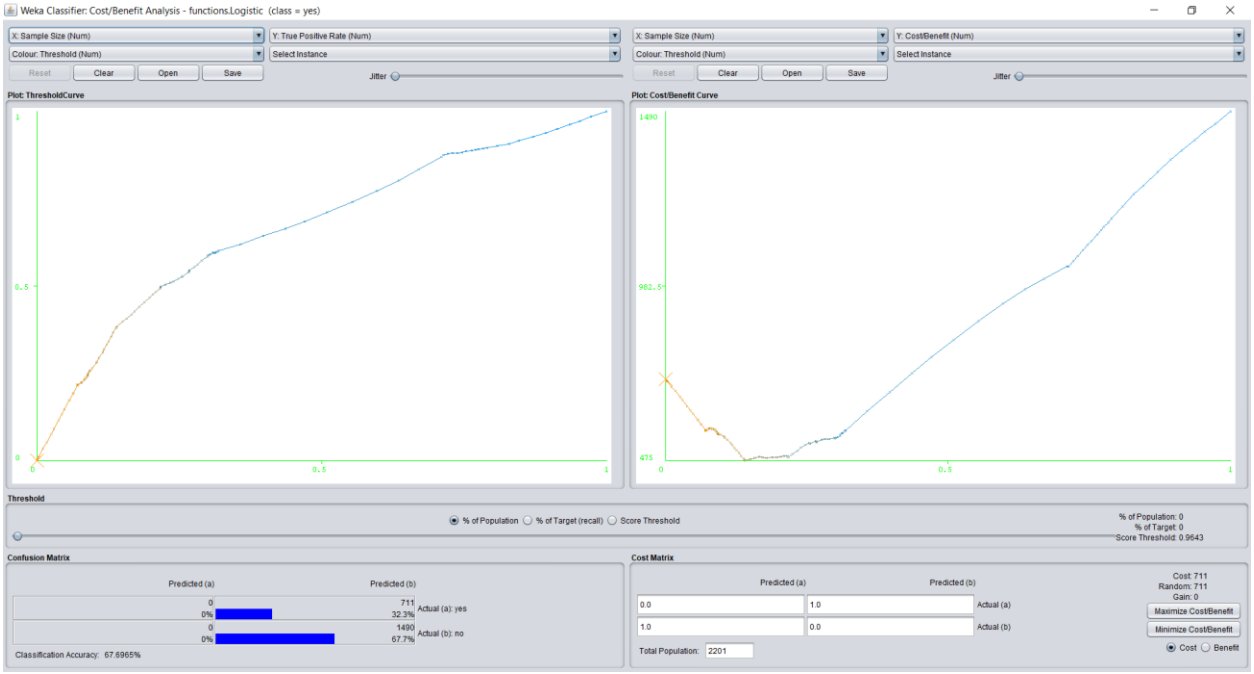
Test Option : Percenetage-Split (90%)

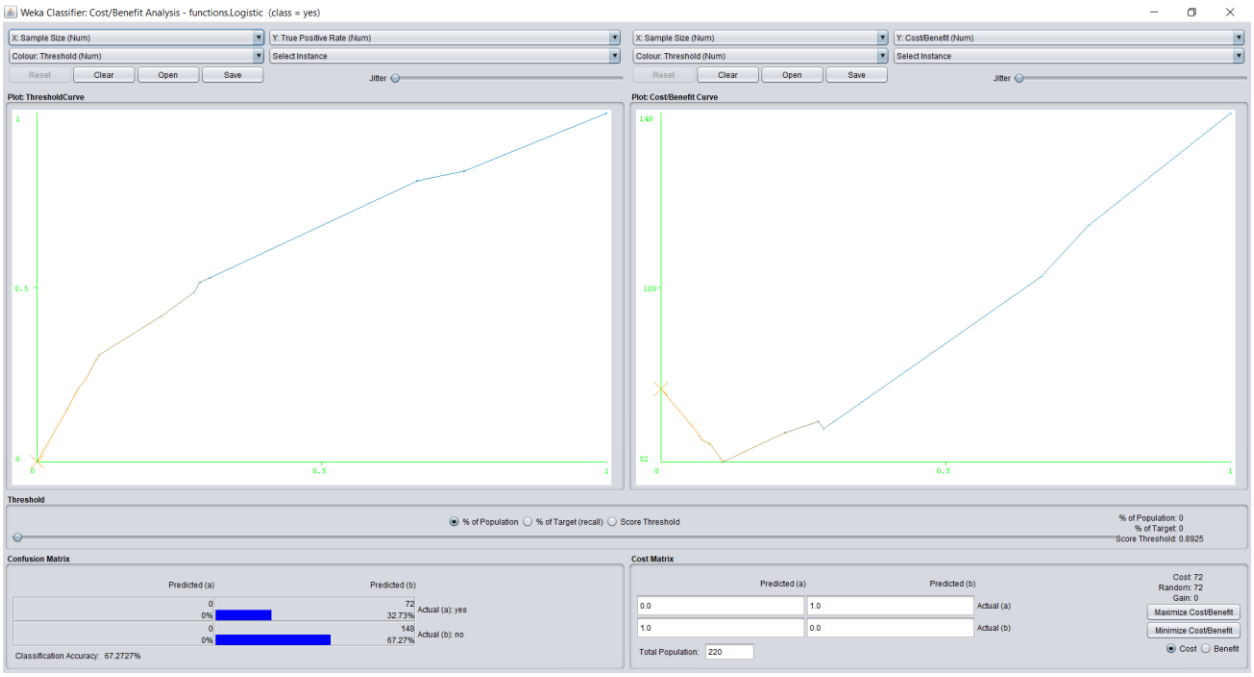




COST/BENEFIT ANALYSIS

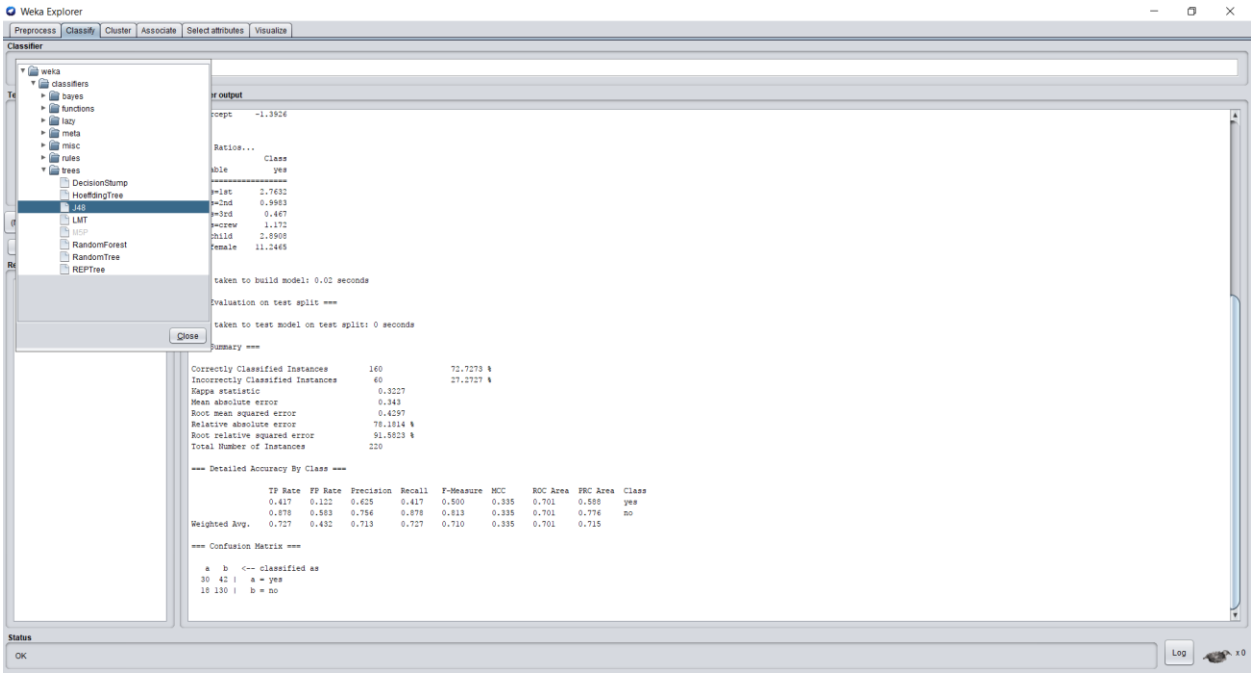






DECISION TREE(J48) STEPS

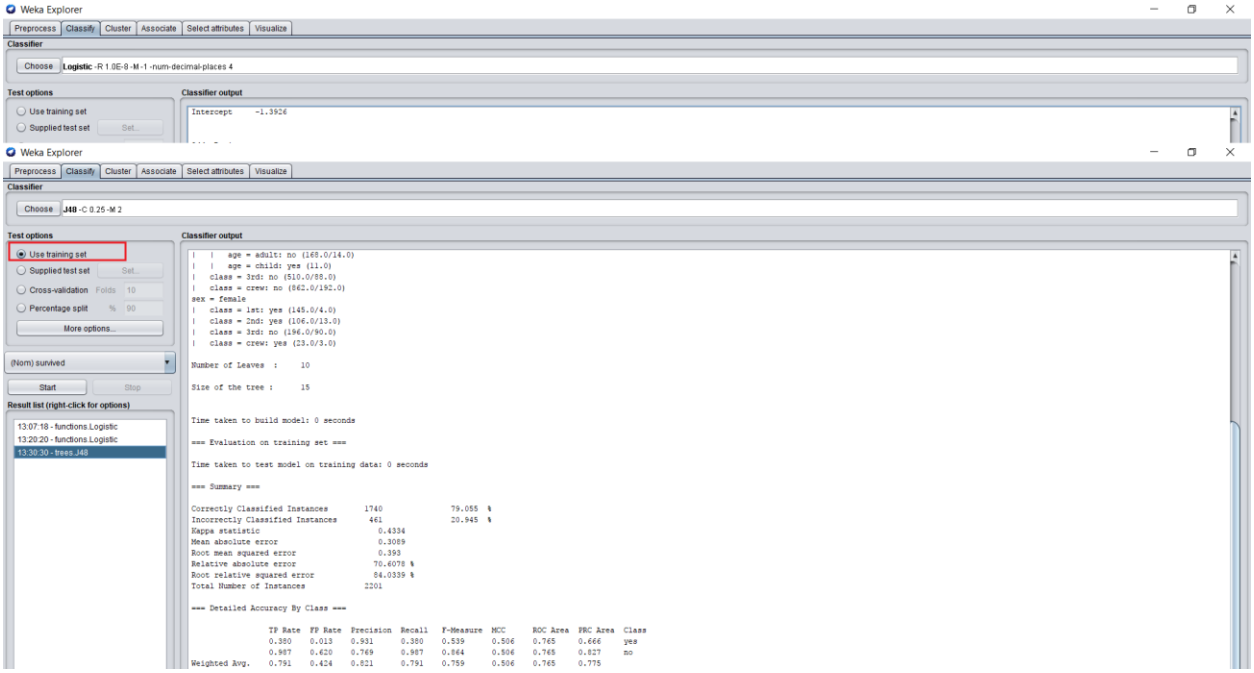
1. Go to ‘Classify’ tab and select J48 under the Tree



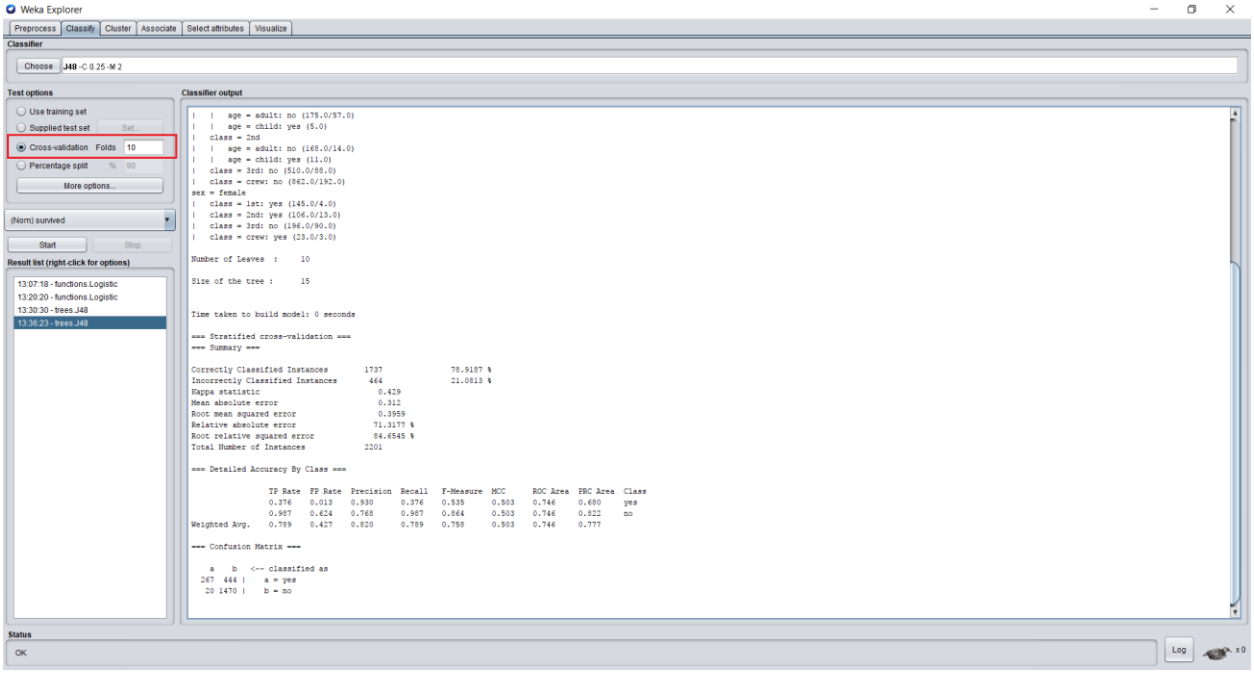
2. Select the ‘Use Training set’ option and click Start. The run would be successfully completed and a Classifier output would be shown as below.
3. Repeat the above step for the Test options ‘cross-validation’ and ‘percentage-splits’. The results would be as shown.

COMPLETE CLASSIFIER OUPUT FOR DIFFERENT TEST OPTIONS

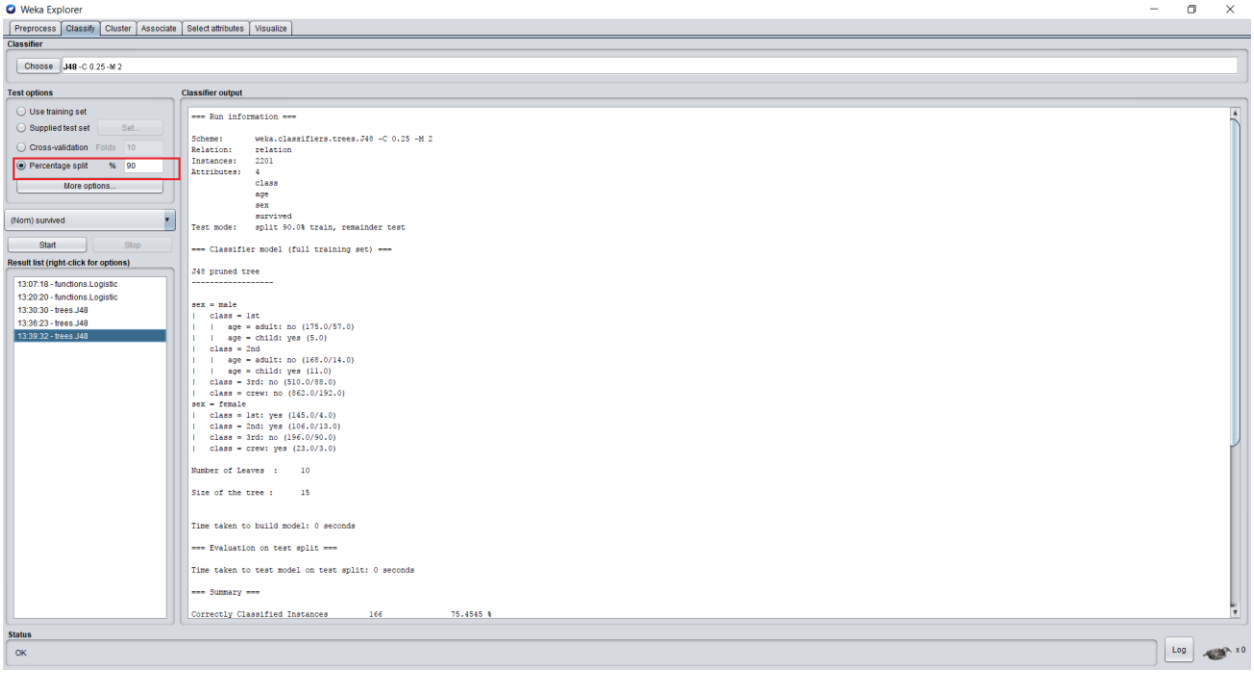
Test Option : Use training set



Test Option: Cross-validatio(10 Fold)

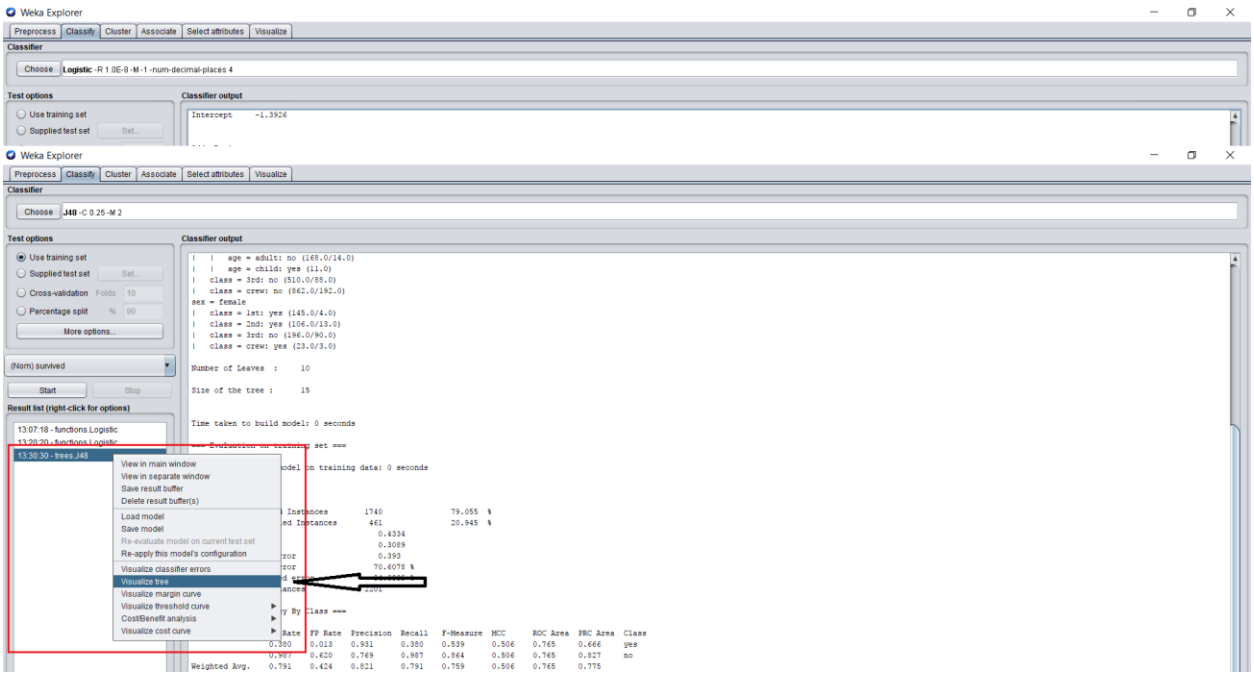


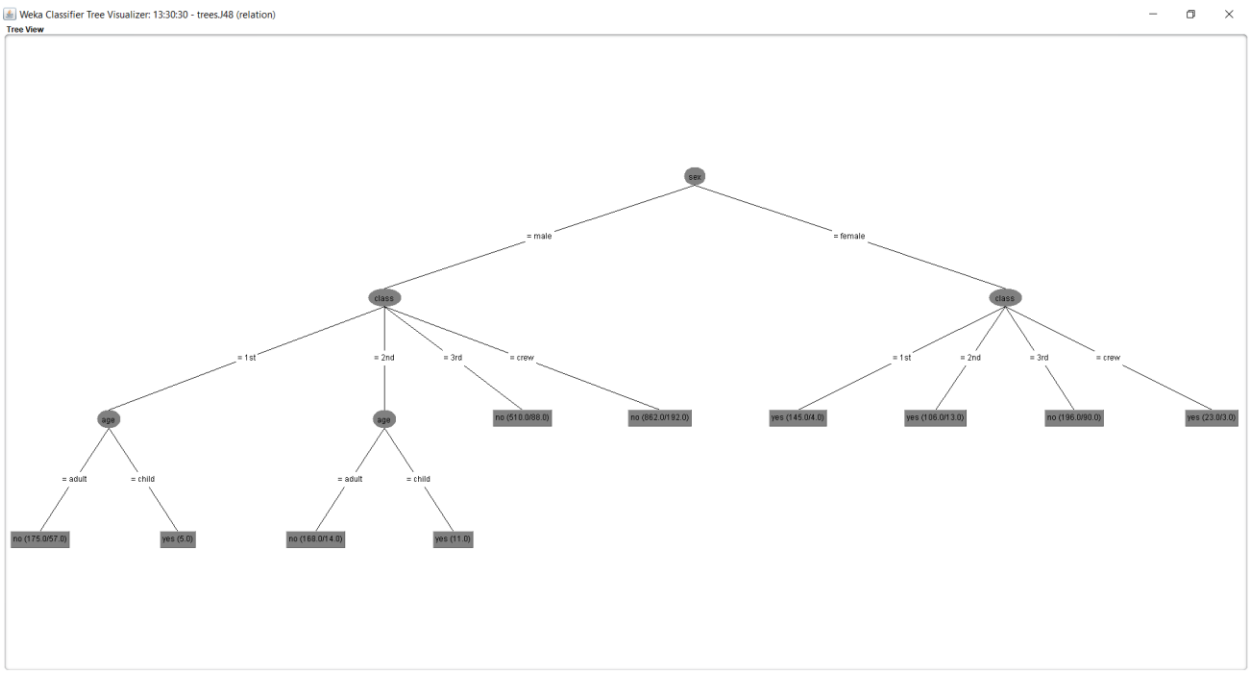
Test Option: Percentage-split (90%)



DECISION TREE VISUALIZATION

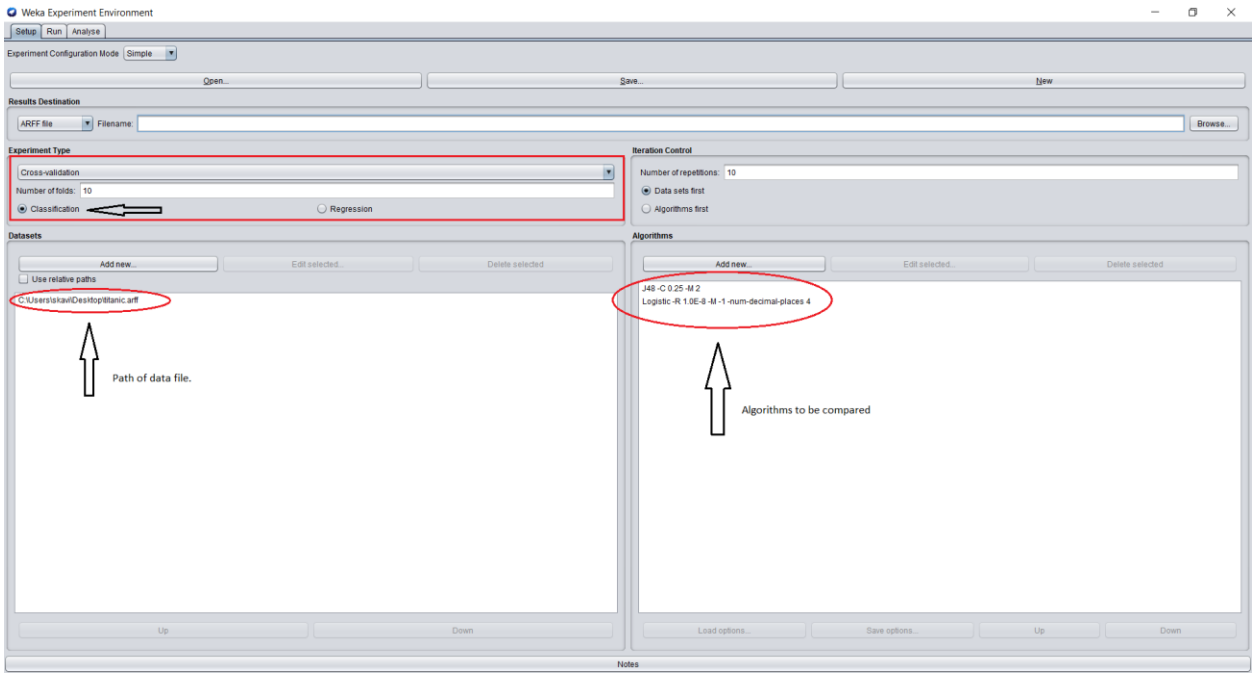
Right click the result buffer option to view the decision tree.



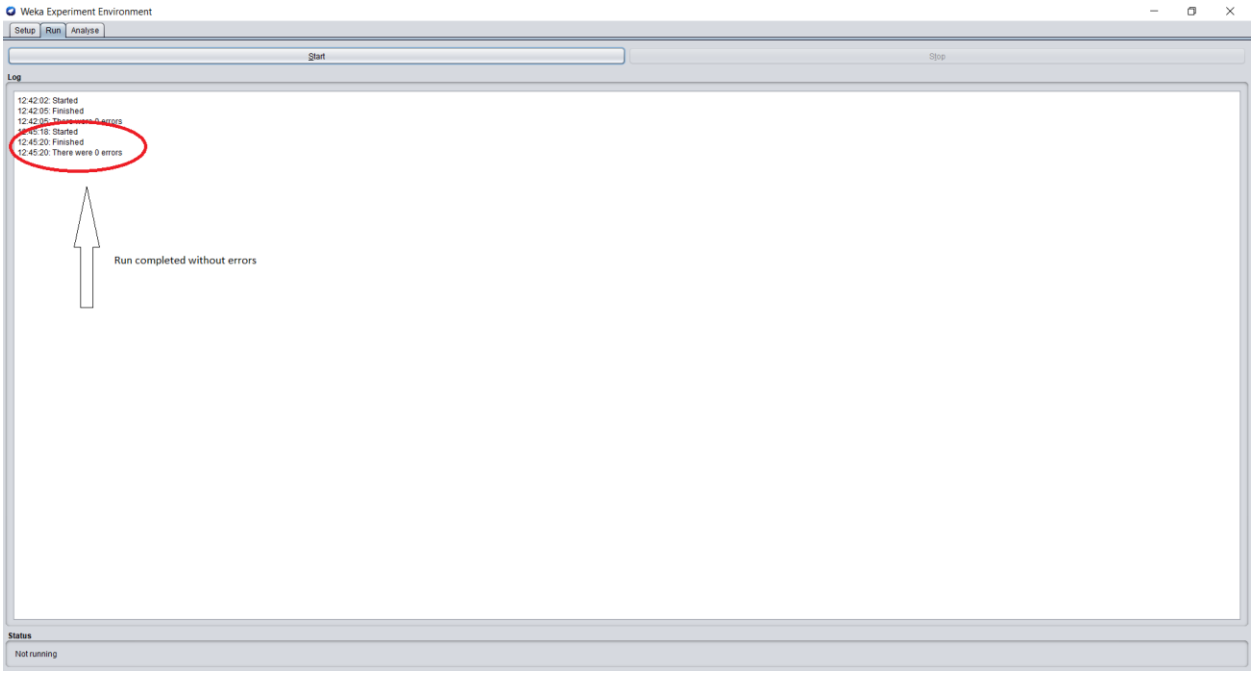


COMPARISON OF THE TWO ALGORITHMS USING WEKA

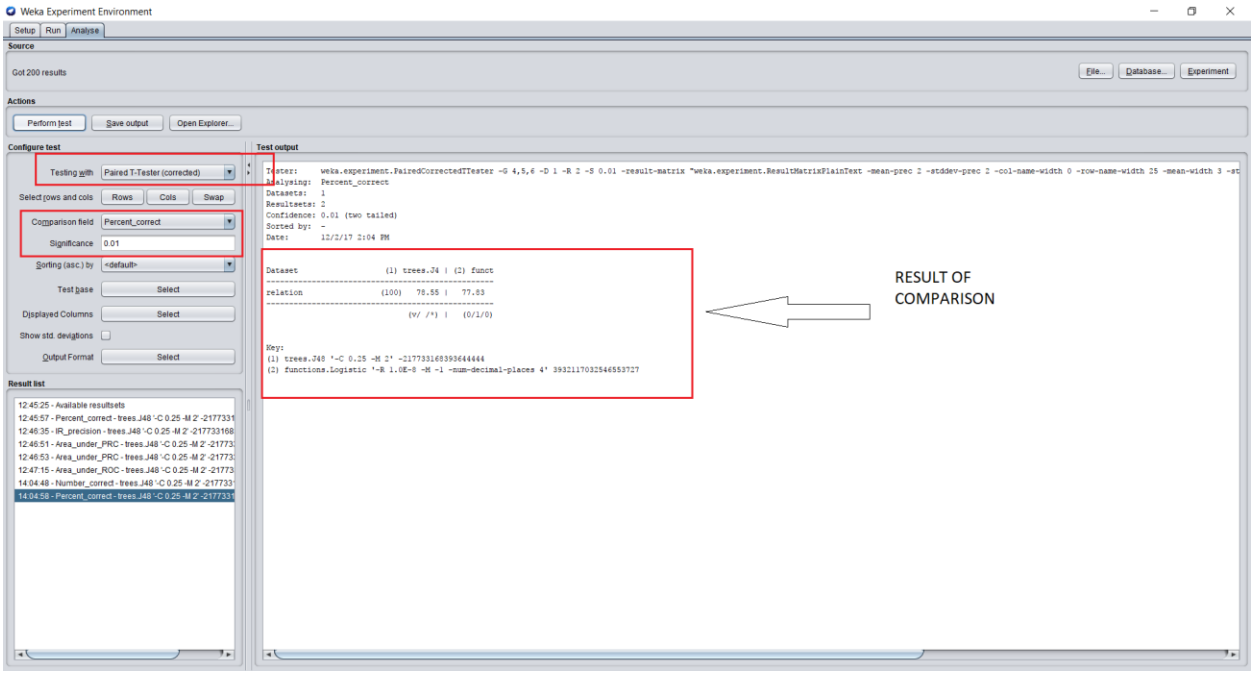
1. To compare the two algorithms select the Experimenter option from the weka tool.
2. In the Experimenter window, go to the **Setup** tab and select the appropriate options for the comparison of the algorithms as shown in below figure.



3. Now go to the **Run** tab and click on start. The process should be completed without any errors.



4. Now in th **analyse** tab under the option 'Configure test', select **Paired T-tester(Corrected)** for 'Testin with' and change the Significance to 0.01 and Click on Perform Test. A comparison of the two algorithms would be shown in the result window.



RESULT OF COMPARISON OF TWO ALGORITHMS

We observe the result which shows that Logistic Regression Analysis shows a 77.83% while the J48 shows 78.55%. Although there isn't a major difference between the results of the two algorithms, J48 still proves to be a better algorithm for our data set with a higher percentage-correct.

WEKA SOURCE CODE

LOGISTIC REGRESSION:

The source code for the Logistic Regression Algorithm in WEKA is obtained from the following website:

http://grepcode.com/file/repo1.maven.org/maven2/nz.ac.waikato.cms.weka/weka-dev/3.7.12/weka/classifiers/functions/Logistic.java?av=f

```
/*
 *   This program is free software: you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation, either version 3 of the License, or
 *   (at your option) any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *   along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */


/*
 *    Logistic.java
 *    Copyright (C) 2003-2012 University of Waikato, Hamilton, New Zealand
 *
```

```
 */


package weka.classifiers.functions;


import java.util.Collections;
import java.util.Enumeration;
import java.util.Vector;


import weka.classifiers.AbstractClassifier;
import weka.classifiers.pmml.producer.LogisticProducerHelper;
import weka.core.Aggregateable;
import weka.core.Capabilities;
import weka.core.Capabilities.Capability;
import weka.core.ConjugateGradientOptimization;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.Optimization;
import weka.core.Option;
import weka.core.OptionHandler;
import weka.core.RevisionUtils;
import weka.core.TechnicalInformation;
import weka.core.TechnicalInformation.Field;
import weka.core.TechnicalInformation.Type;
import weka.core.TechnicalInformationHandler;
import weka.core.Utils;
import weka.core.WeightedInstancesHandler;
import weka.core.pmml.PMMLProducer;
import weka.filters.Filter;
import weka.filters.unsupervised.attribute.NominalToBinary;
```

```
import weka.filters.unsupervised.attribute.RemoveUseless;

import weka.filters.unsupervised.attribute.ReplaceMissingValues;


/**

 * <!-- globalinfo-start --> Class for building and using a multinomial logistic

 * regression model with a ridge estimator.<br/>

 * <br/>

 * There are some modifications, however, compared to the paper of leCessie and

 * van Houwelingen(1992): <br/>

 * <br/>

 * If there are k classes for n instances with m attributes, the parameter

 * matrix B to be calculated will be an m*(k-1) matrix.<br/>

 * <br/>

 * The probability for class j with the exception of the last class is<br/>

 * <br/>

 * Pj(Xi) = exp(XiBj)/((sum[j=1..(k-1)]exp(Xi*Bj))+1) <br/>

 * <br/>

 * The last class has probability<br/>

 * <br/>

 * 1-(sum[j=1..(k-1)]Pj(Xi)) <br/>

 * = 1/((sum[j=1..(k-1)]exp(Xi*Bj))+1)<br/>

 * <br/>

 * The (negative) multinomial log-likelihood is thus: <br/>

 * <br/>

 * L = -sum[i=1..n]{<br/>

 * sum[j=1..(k-1)](Yij * ln(Pj(Xi)))<br/>

 * +(1 - (sum[j=1..(k-1)]Yij)) <br/>

 * * ln(1 - sum[j=1..(k-1)]Pj(Xi))<br/>

 * } + ridge * (B^2)<br/>
```

*

* In order to find the matrix B for which L is minimised, a Quasi-Newton Method

* is used to search for the optimized values of the m*(k-1) variables. Note

* that before we use the optimization procedure, we 'squeeze' the matrix B into

* a m*(k-1) vector. For details of the optimization procedure, please check

* weka.core.Optimization class.

*

* Although original Logistic Regression does not deal with instance weights, we

* modify the algorithm a little bit to handle the instance weights.

*

* For more information see:

*

* le Cessie, S., van Houwelingen, J.C. (1992). Ridge Estimators in Logistic

* Regression. Applied Statistics. 41(1):191-201.

*

* Note: Missing values are replaced using a ReplaceMissingValuesFilter, and

* nominal attributes are transformed into numeric attributes using a

* NominalToBinaryFilter.

* <p/>

* <!-- globalinfo-end -->

*

* <!-- technical-bibtex-start --> BibTeX:

*

* <pre>

* @article{leCessie1992,

* author = {le Cessie, S. and van Houwelingen, J.C.},

* journal = {Applied Statistics},

* number = {1},

* pages = {191-201},

```
*    title = {Ridge Estimators in Logistic Regression},
*    volume = {41},
*    year = {1992}
* }
* </pre>
* <p/>
* <!-- technical-bibtex-end -->
*
* <!-- options-start --> Valid options are:
* <p/>
*
* <pre>
* -D
*  Turn on debugging output.
* </pre>
*
* <pre>
* -R &lt;ridge&gt;
*  Set the ridge in the log-likelihood.
* </pre>
*
* <pre>
* -M &lt;number&gt;
*  Set the maximum number of iterations (default -1, until convergence).
* </pre>
*
* <!-- options-end -->
*
* @author Xin Xu (xx5@cs.waikato.ac.nz)
```

```
 * @version $Revision: 11247 $
 */
public class Logistic extends AbstractClassifier implements OptionHandler,
  WeightedInstancesHandler, TechnicalInformationHandler, PMMLProducer,
  Aggregateable<Logistic> {

  /** for serialization */
  static final long serialVersionUID = 3932117032546553727L;

  /** The coefficients (optimized parameters) of the model */
  protected double[][] m_Par;

  /** The data saved as a matrix */
  protected double[][] m_Data;

  /** The number of attributes in the model */
  protected int m_NumPredictors;

  /** The index of the class attribute */
  protected int m_ClassIndex;

  /** The number of the class labels */
  protected int m_NumClasses;

  /** The ridge parameter. */
  protected double m_Ridge = 1e-8;

  /** An attribute filter */
  private RemoveUseless m_AttFilter;
```

```
/** The filter used to make attributes numeric. */
private NominalToBinary m_NominalToBinary;

/** The filter used to get rid of missing values. */
private ReplaceMissingValues m_ReplaceMissingValues;

/** Log-likelihood of the searched model */
protected double m_LL;

/** The maximum number of iterations. */
private int m_MaxIts = -1;

/** Wether to use conjugate gradient descent rather than BFGS updates. */
private boolean m_useConjugateGradientDescent = false;

private Instances m_structure;

/**
 * Returns a string describing this classifier
 *
 * @return a description of the classifier suitable for displaying in the
 *         explorer/experimenter gui
 */
public String globalInfo() {
  return "Class for building and using a multinomial logistic "
    + "regression model with a ridge estimator.\n\n"
    + "There are some modifications, however, compared to the paper of "
    + "leCessie and van Houwelingen(1992): \n\n"
```

```
    + "If there are k classes for n instances with m attributes, the "
    + "parameter matrix B to be calculated will be an m*(k-1) matrix.\n\n"
    + "The probability for class j with the exception of the last class is\n\n"
    + "Pj(Xi) = exp(XiBj)/((sum[j=1..(k-1)]exp(Xi*Bj))+1) \n\n"
    + "The last class has probability\n\n"
    + "1-(sum[j=1..(k-1)]Pj(Xi)) \n\t= 1/((sum[j=1..(k-1)]exp(Xi*Bj))+1)\n\n"
    + "The (negative) multinomial log-likelihood is thus: \n\n"
    + "L = -sum[i=1..n]{\n\tsum[j=1..(k-1)](Yij * ln(Pj(Xi)))"
    + "\n\t+(1 - (sum[j=1..(k-1)]Yij)) \n\t* ln(1 - sum[j=1..(k-1)]Pj(Xi))"
    + "\n\t} + ridge * (B^2)\n\n"
    + "In order to find the matrix B for which L is minimised, a "
    + "Quasi-Newton Method is used to search for the optimized values of "
    + "the m*(k-1) variables.  Note that before we use the optimization "
    + "procedure, we 'squeeze' the matrix B into a m*(k-1) vector.  For "
    + "details of the optimization procedure, please check "
    + "weka.core.Optimization class.\n\n"
    + "Although original Logistic Regression does not deal with instance "
    + "weights, we modify the algorithm a little bit to handle the "
    + "instance weights.\n\n"
    + "For more information see:\n\n"
    + getTechnicalInformation().toString()
    + "\n\n"
    + "Note: Missing values are replaced using a ReplaceMissingValuesFilter, and "
    + "nominal attributes are transformed into numeric attributes using a "
    + "NominalToBinaryFilter.";
}

/**
 * Returns an instance of a TechnicalInformation object, containing detailed
```

```
 * information about the technical background of this class, e.g., paper
 * reference or book this class is based on.
 *
 * @return the technical information about this class
 */
@Override
public TechnicalInformation getTechnicalInformation() {
  TechnicalInformation result;

  result = new TechnicalInformation(Type.ARTICLE);
  result.setValue(Field.AUTHOR, "le Cessie, S. and van Houwelingen, J.C.");
  result.setValue(Field.YEAR, "1992");
  result.setValue(Field.TITLE, "Ridge Estimators in Logistic Regression");
  result.setValue(Field.JOURNAL, "Applied Statistics");
  result.setValue(Field.VOLUME, "41");
  result.setValue(Field.NUMBER, "1");
  result.setValue(Field.PAGES, "191-201");

  return result;
}

/**
 * Returns an enumeration describing the available options
 *
 * @return an enumeration of all the available options
 */
@Override
public Enumeration<Option> listOptions() {
  Vector<Option> newVector = new Vector<Option>(4);
```

```
  newVector.addElement(new Option(
    "\tUse conjugate gradient descent rather than BFGS updates.", "C", 0,
    "-C"));
  newVector.addElement(new Option("\tSet the ridge in the log-likelihood.",
    "R", 1, "-R <ridge>"));
  newVector.addElement(new Option("\tSet the maximum number of iterations"
    + " (default -1, until convergence).", "M", 1, "-M <number>"));

  newVector.addAll(Collections.list(super.listOptions()));

  return newVector.elements();
}

/**
 * Parses a given list of options.
 * <p/>
 *
 * <!-- options-start --> Valid options are:
 * <p/>
 *
 * <pre>
 * -D
 *  Turn on debugging output.
 * </pre>
 *
 * <pre>
 * -R &lt;ridge&gt;
 *  Set the ridge in the log-likelihood.
```

```
 * </pre>
 *
 * <pre>
 * -M &lt;number&gt;
 *  Set the maximum number of iterations (default -1, until convergence).
 * </pre>
 *
 * <!-- options-end -->
 *
 * @param options the list of options as an array of strings
 * @throws Exception if an option is not supported
 */
@Override
public void setOptions(String[] options) throws Exception {

  setUseConjugateGradientDescent(Utils.getFlag('C', options));

  String ridgeString = Utils.getOption('R', options);
  if (ridgeString.length() != 0) {
    m_Ridge = Double.parseDouble(ridgeString);
  } else {
    m_Ridge = 1.0e-8;
  }

  String maxItsString = Utils.getOption('M', options);
  if (maxItsString.length() != 0) {
    m_MaxIts = Integer.parseInt(maxItsString);
  } else {
    m_MaxIts = -1;
```

```
  }

  super.setOptions(options);

  Utils.checkForRemainingOptions(options);
}

/**
 * Gets the current settings of the classifier.
 *
 * @return an array of strings suitable for passing to setOptions
 */
@Override
public String[] getOptions() {

  Vector<String> options = new Vector<String>();

  if (getUseConjugateGradientDescent()) {
    options.add("-C");
  }
  options.add("-R");
  options.add("" + m_Ridge);
  options.add("-M");
  options.add("" + m_MaxIts);

  Collections.addAll(options, super.getOptions());

  return options.toArray(new String[0]);
}
```

```
/**
 * Returns the tip text for this property
 *
 * @return tip text for this property suitable for displaying in the
 *         explorer/experimenter gui
 */
@Override
public String debugTipText() {
  return "Output debug information to the console.";
}

/**
 * Sets whether debugging output will be printed.
 *
 * @param debug true if debugging output should be printed
 */
@Override
public void setDebug(boolean debug) {
  m_Debug = debug;
}

/**
 * Gets whether debugging output will be printed.
 *
 * @return true if debugging output will be printed
 */
@Override
public boolean getDebug() {
```

```
  return m_Debug;
}

/**
 * Returns the tip text for this property
 *
 * @return tip text for this property suitable for displaying in the
 *         explorer/experimenter gui
 */
public String useConjugateGradientDescentTipText() {
  return "Use conjugate gradient descent rather than BFGS updates; faster for problems with many parameters.";
}

/**
 * Sets whether conjugate gradient descent is used.
 *
 * @param useConjugateGradientDescent true if CGD is to be used.
 */
public void setUseConjugateGradientDescent(boolean useConjugateGradientDescent) {
  m_useConjugateGradientDescent = useConjugateGradientDescent;
}

/**
 * Gets whether to use conjugate gradient descent rather than BFGS updates.
 *
 * @return true if CGD is used
 */
public boolean getUseConjugateGradientDescent() {
  return m_useConjugateGradientDescent;
```

```
}

/**
 * Returns the tip text for this property
 *
 * @return tip text for this property suitable for displaying in the
 *         explorer/experimenter gui
 */
public String ridgeTipText() {
  return "Set the Ridge value in the log-likelihood.";
}

/**
 * Sets the ridge in the log-likelihood.
 *
 * @param ridge the ridge
 */
public void setRidge(double ridge) {
  m_Ridge = ridge;
}

/**
 * Gets the ridge in the log-likelihood.
 *
 * @return the ridge
 */
public double getRidge() {
  return m_Ridge;
}
```

```
/**
 * Returns the tip text for this property
 *
 * @return tip text for this property suitable for displaying in the
 *         explorer/experimenter gui
 */
public String maxItsTipText() {
  return "Maximum number of iterations to perform.";
}

/**
 * Get the value of MaxIts.
 *
 * @return Value of MaxIts.
 */
public int getMaxIts() {

  return m_MaxIts;
}

/**
 * Set the value of MaxIts.
 *
 * @param newMaxIts Value to assign to MaxIts.
 */
public void setMaxIts(int newMaxIts) {

  m_MaxIts = newMaxIts;
```

```
}

private class OptEng extends Optimization {

  OptObject m_oO = null;

  private OptEng(OptObject oO) {
    m_oO = oO;
  }

  @Override
  protected double objectiveFunction(double[] x) {
    return m_oO.objectiveFunction(x);
  }

  @Override
  protected double[] evaluateGradient(double[] x) {
    return m_oO.evaluateGradient(x);
  }

  @Override
  public String getRevision() {
    return RevisionUtils.extract("$Revision: 11247 $");
  }
}

private class OptEngCG extends ConjugateGradientOptimization {

  OptObject m_oO = null;
```

```
  private OptEngCG(OptObject oO) {
    m_oO = oO;
  }

  @Override
  protected double objectiveFunction(double[] x) {
    return m_oO.objectiveFunction(x);
  }

  @Override
  protected double[] evaluateGradient(double[] x) {
    return m_oO.evaluateGradient(x);
  }

  @Override
  public String getRevision() {
    return RevisionUtils.extract("$Revision: 11247 $");
  }
}

private class OptObject {

  /** Weights of instances in the data */
  private double[] weights;

  /** Class labels of instances */
  private int[] cls;
```

```
/**
 * Set the weights of instances
 *
 * @param w the weights to be set
 */
public void setWeights(double[] w) {
  weights = w;
}

/**
 * Set the class labels of instances
 *
 * @param c the class labels to be set
 */
public void setClassLabels(int[] c) {
  cls = c;
}

/**
 * Evaluate objective function
 *
 * @param x the current values of variables
 * @return the value of the objective function
 */
protected double objectiveFunction(double[] x) {
  double nll = 0; // -LogLikelihood
  int dim = m_NumPredictors + 1; // Number of variables per class

  for (int i = 0; i < cls.length; i++) { // ith instance
```

```
  double[] exp = new double[m_NumClasses - 1];
  int index;
  for (int offset = 0; offset < m_NumClasses - 1; offset++) {
    index = offset * dim;
    for (int j = 0; j < dim; j++) {
      exp[offset] += m_Data[i][j] * x[index + j];
    }
  }
  double max = exp[Utils.maxIndex(exp)];
  double denom = Math.exp(-max);
  double num;
  if (cls[i] == m_NumClasses - 1) { // Class of this instance
    num = -max;
  } else {
    num = exp[cls[i]] - max;
  }
  for (int offset = 0; offset < m_NumClasses - 1; offset++) {
    denom += Math.exp(exp[offset] - max);
  }

  nll -= weights[i] * (num - Math.log(denom)); // Weighted NLL
}

// Ridge: note that intercepts NOT included
for (int offset = 0; offset < m_NumClasses - 1; offset++) {
  for (int r = 1; r < dim; r++) {
    nll += m_Ridge * x[offset * dim + r] * x[offset * dim + r];
  }
```

```
  }

  return nll;
}

/**
 * Evaluate Jacobian vector
 *
 * @param x the current values of variables
 * @return the gradient vector
 */
protected double[] evaluateGradient(double[] x) {
  double[] grad = new double[x.length];
  int dim = m_NumPredictors + 1; // Number of variables per class

  for (int i = 0; i < cls.length; i++) { // ith instance
    double[] num = new double[m_NumClasses - 1]; // numerator of
                                        // [-log(1+sum(exp))]'
    int index;
    for (int offset = 0; offset < m_NumClasses - 1; offset++) { // Which
                                                // part of x
      double exp = 0.0;
      index = offset * dim;
      for (int j = 0; j < dim; j++) {
        exp += m_Data[i][j] * x[index + j];
      }
      num[offset] = exp;
    }
```

```
  double max = num[Utils.maxIndex(num)];
  double denom = Math.exp(-max); // Denominator of [-log(1+sum(exp))]'
  for (int offset = 0; offset < m_NumClasses - 1; offset++) {
    num[offset] = Math.exp(num[offset] - max);
    denom += num[offset];
  }
  Utils.normalize(num, denom);

  // Update denominator of the gradient of -log(Posterior)
  double firstTerm;
  for (int offset = 0; offset < m_NumClasses - 1; offset++) { // Which
                                                              // part of x
    index = offset * dim;
    firstTerm = weights[i] * num[offset];
    for (int q = 0; q < dim; q++) {
      grad[index + q] += firstTerm * m_Data[i][q];
    }
  }

  if (cls[i] != m_NumClasses - 1) { // Not the last class
    for (int p = 0; p < dim; p++) {
      grad[cls[i] * dim + p] -= weights[i] * m_Data[i][p];
    }
  }
}

// Ridge: note that intercepts NOT included
for (int offset = 0; offset < m_NumClasses - 1; offset++) {
  for (int r = 1; r < dim; r++) {
```

```
        grad[offset * dim + r] += 2 * m_Ridge * x[offset * dim + r];
      }
    }

    return grad;
  }
}

/**
 * Returns default capabilities of the classifier.
 *
 * @return the capabilities of this classifier
 */
@Override
public Capabilities getCapabilities() {
  Capabilities result = super.getCapabilities();
  result.disableAll();

  // attributes
  result.enable(Capability.NOMINAL_ATTRIBUTES);
  result.enable(Capability.NUMERIC_ATTRIBUTES);
  result.enable(Capability.DATE_ATTRIBUTES);
  result.enable(Capability.MISSING_VALUES);

  // class
  result.enable(Capability.NOMINAL_CLASS);
  result.enable(Capability.MISSING_CLASS_VALUES);

  return result;
```

```
}

/**
 * Builds the classifier
 *
 * @param train the training data to be used for generating the boosted
 *          classifier.
 * @throws Exception if the classifier could not be built successfully
 */
@Override
public void buildClassifier(Instances train) throws Exception {
  // can classifier handle the data?
  getCapabilities().testWithFail(train);

  // remove instances with missing class
  train = new Instances(train);
  train.deleteWithMissingClass();

  // Replace missing values
  m_ReplaceMissingValues = new ReplaceMissingValues();
  m_ReplaceMissingValues.setInputFormat(train);
  train = Filter.useFilter(train, m_ReplaceMissingValues);

  // Remove useless attributes
  m_AttFilter = new RemoveUseless();
  m_AttFilter.setInputFormat(train);
  train = Filter.useFilter(train, m_AttFilter);

  // Transform attributes
```

```
m_NominalToBinary = new NominalToBinary();
m_NominalToBinary.setInputFormat(train);
train = Filter.useFilter(train, m_NominalToBinary);

// Save the structure for printing the model
m_structure = new Instances(train, 0);

// Extract data
m_ClassIndex = train.classIndex();
m_NumClasses = train.numClasses();

int nK = m_NumClasses - 1; // Only K-1 class labels needed
int nR = m_NumPredictors = train.numAttributes() - 1;
int nC = train.numInstances();

m_Data = new double[nC][nR + 1]; // Data values
int[] Y = new int[nC]; // Class labels
double[] xMean = new double[nR + 1]; // Attribute means
double[] xSD = new double[nR + 1]; // Attribute stddev's
double[] sY = new double[nK + 1]; // Number of classes
double[] weights = new double[nC]; // Weights of instances
double totWeights = 0; // Total weights of the instances
m_Par = new double[nR + 1][nK]; // Optimized parameter values

if (m_Debug) {
  System.out.println("Extracting data...");
}

for (int i = 0; i < nC; i++) {
```

```
  // initialize X[][]
  Instance current = train.instance(i);
  Y[i] = (int) current.classValue(); // Class value starts from 0
  weights[i] = current.weight(); // Dealing with weights
  totWeights += weights[i];

  m_Data[i][0] = 1;
  int j = 1;
  for (int k = 0; k <= nR; k++) {
    if (k != m_ClassIndex) {
      double x = current.value(k);
      m_Data[i][j] = x;
      xMean[j] += weights[i] * x;
      xSD[j] += weights[i] * x * x;
      j++;
    }
  }

  // Class count
  sY[Y[i]]++;
}

if ((totWeights <= 1) && (nC > 1)) {
  throw new Exception(
    "Sum of weights of instances less than 1, please reweight!");
}

xMean[0] = 0;
xSD[0] = 1;
```

```
for (int j = 1; j <= nR; j++) {
  xMean[j] = xMean[j] / totWeights;
  if (totWeights > 1) {
    xSD[j] = Math.sqrt(Math.abs(xSD[j] - totWeights * xMean[j] * xMean[j])
      / (totWeights - 1));
  } else {
    xSD[j] = 0;
  }
}

if (m_Debug) {
  // Output stats about input data
  System.out.println("Descriptives...");
  for (int m = 0; m <= nK; m++) {
    System.out.println(sY[m] + " cases have class " + m);
  }
  System.out.println("\n Variable     Avg       SD    ");
  for (int j = 1; j <= nR; j++) {
    System.out.println(Utils.doubleToString(j, 8, 4)
      + Utils.doubleToString(xMean[j], 10, 4)
      + Utils.doubleToString(xSD[j], 10, 4));
  }
}

// Normalise input data
for (int i = 0; i < nC; i++) {
  for (int j = 0; j <= nR; j++) {
    if (xSD[j] != 0) {
      m_Data[i][j] = (m_Data[i][j] - xMean[j]) / xSD[j];
```

```
    }
  }
}

if (m_Debug) {
  System.out.println("\nIteration History...");
}

double x[] = new double[(nR + 1) * nK];
double[][] b = new double[2][x.length]; // Boundary constraints, N/A here

// Initialize
for (int p = 0; p < nK; p++) {
  int offset = p * (nR + 1);
  x[offset] = Math.log(sY[p] + 1.0) - Math.log(sY[nK] + 1.0); // Null model
  b[0][offset] = Double.NaN;
  b[1][offset] = Double.NaN;
  for (int q = 1; q <= nR; q++) {
    x[offset + q] = 0.0;
    b[0][offset + q] = Double.NaN;
    b[1][offset + q] = Double.NaN;
  }
}

OptObject oO = new OptObject();
oO.setWeights(weights);
oO.setClassLabels(Y);

Optimization opt = null;
```

```
if (m_useConjugateGradientDescent) {
  opt = new OptEngCG(oO);
} else {
  opt = new OptEng(oO);
}
opt.setDebug(m_Debug);

if (m_MaxIts == -1) { // Search until convergence
  x = opt.findArgmin(x, b);
  while (x == null) {
    x = opt.getVarbValues();
    if (m_Debug) {
      System.out.println("First set of iterations finished, not enough!");
    }
    x = opt.findArgmin(x, b);
  }
  if (m_Debug) {
    System.out.println(" ----------<Converged>----------");
  }
} else {
  opt.setMaxIteration(m_MaxIts);
  x = opt.findArgmin(x, b);
  if (x == null) {
    x = opt.getVarbValues();
  }
}

m_LL = -opt.getMinFunction(); // Log-likelihood
```

```
  // Don't need data matrix anymore
  m_Data = null;

  // Convert coefficients back to non-normalized attribute units
  for (int i = 0; i < nK; i++) {
    m_Par[0][i] = x[i * (nR + 1)];
    for (int j = 1; j <= nR; j++) {
      m_Par[j][i] = x[i * (nR + 1) + j];
      if (xSD[j] != 0) {
        m_Par[j][i] /= xSD[j];
        m_Par[0][i] -= m_Par[j][i] * xMean[j];
      }
    }
  }
}

/**
 * Computes the distribution for a given instance
 *
 * @param instance the instance for which distribution is computed
 * @return the distribution
 * @throws Exception if the distribution can't be computed successfully
 */
@Override
public double[] distributionForInstance(Instance instance) throws Exception {

  m_ReplaceMissingValues.input(instance);
  instance = m_ReplaceMissingValues.output();
  m_AttFilter.input(instance);
```

```
  instance = m_AttFilter.output();
  m_NominalToBinary.input(instance);
  instance = m_NominalToBinary.output();

  // Extract the predictor columns into an array
  double[] instDat = new double[m_NumPredictors + 1];
  int j = 1;
  instDat[0] = 1;
  for (int k = 0; k <= m_NumPredictors; k++) {
    if (k != m_ClassIndex) {
      instDat[j++] = instance.value(k);
    }
  }

  double[] distribution = evaluateProbability(instDat);
  return distribution;
}

/**
 * Compute the posterior distribution using optimized parameter values and the
 * testing instance.
 *
 * @param data the testing instance
 * @return the posterior probability distribution
 */
private double[] evaluateProbability(double[] data) {
  double[] prob = new double[m_NumClasses], v = new double[m_NumClasses];

  // Log-posterior before normalizing
```

```
  for (int j = 0; j < m_NumClasses - 1; j++) {
    for (int k = 0; k <= m_NumPredictors; k++) {
      v[j] += m_Par[k][j] * data[k];
    }
  }
  v[m_NumClasses - 1] = 0;

  // Do so to avoid scaling problems
  for (int m = 0; m < m_NumClasses; m++) {
    double sum = 0;
    for (int n = 0; n < m_NumClasses - 1; n++) {
      sum += Math.exp(v[n] - v[m]);
    }
    prob[m] = 1 / (sum + Math.exp(-v[m]));
  }

  return prob;
}

/**
 * Returns the coefficients for this logistic model. The first dimension
 * indexes the attributes, and the second the classes.
 *
 * @return the coefficients for this logistic model
 */
public double[][] coefficients() {
  return m_Par;
}
```

```
/**
 * Gets a string describing the classifier.
 *
 * @return a string describing the classifer built.
 */
@Override
public String toString() {
  StringBuffer temp = new StringBuffer();

  String result = "";
  temp.append("Logistic Regression with ridge parameter of " + m_Ridge);
  if (m_Par == null) {
    return result + ": No model built yet.";
  }

  // find longest attribute name
  int attLength = 0;
  for (int i = 0; i < m_structure.numAttributes(); i++) {
    if (i != m_structure.classIndex()
      && m_structure.attribute(i).name().length() > attLength) {
      attLength = m_structure.attribute(i).name().length();
    }
  }

  if ("Intercept".length() > attLength) {
    attLength = "Intercept".length();
  }

  if ("Variable".length() > attLength) {
```

```
  attLength = "Variable".length();
}
attLength += 2;

int colWidth = 0;
// check length of class names
for (int i = 0; i < m_structure.classAttribute().numValues() - 1; i++) {
  if (m_structure.classAttribute().value(i).length() > colWidth) {
    colWidth = m_structure.classAttribute().value(i).length();
  }
}

// check against coefficients and odds ratios
for (int j = 1; j <= m_NumPredictors; j++) {
  for (int k = 0; k < m_NumClasses - 1; k++) {
    if (Utils.doubleToString(m_Par[j][k], 12, 4).trim().length() > colWidth) {
      colWidth = Utils.doubleToString(m_Par[j][k], 12, 4).trim().length();
    }
    double ORc = Math.exp(m_Par[j][k]);
    String t = " "
      + ((ORc > 1e10) ? "" + ORc : Utils.doubleToString(ORc, 12, 4));
    if (t.trim().length() > colWidth) {
      colWidth = t.trim().length();
    }
  }
}

if ("Class".length() > colWidth) {
  colWidth = "Class".length();
```

```
}
colWidth += 2;


temp.append("\nCoefficients...\n");
temp.append(Utils.padLeft(" ", attLength)
  + Utils.padLeft("Class", colWidth) + "\n");
temp.append(Utils.padRight("Variable", attLength));


for (int i = 0; i < m_NumClasses - 1; i++) {
  String className = m_structure.classAttribute().value(i);
  temp.append(Utils.padLeft(className, colWidth));
}
temp.append("\n");
int separatorL = attLength + ((m_NumClasses - 1) * colWidth);
for (int i = 0; i < separatorL; i++) {
  temp.append("=");
}
temp.append("\n");


int j = 1;
for (int i = 0; i < m_structure.numAttributes(); i++) {
  if (i != m_structure.classIndex()) {
    temp.append(Utils.padRight(m_structure.attribute(i).name(), attLength));
    for (int k = 0; k < m_NumClasses - 1; k++) {
      temp.append(Utils.padLeft(Utils.doubleToString(m_Par[j][k], 12, 4)
        .trim(), colWidth));
    }
    temp.append("\n");
    j++;
```

```
  }
}

temp.append(Utils.padRight("Intercept", attLength));
for (int k = 0; k < m_NumClasses - 1; k++) {
  temp.append(Utils.padLeft(
    Utils.doubleToString(m_Par[0][k], 10, 4).trim(), colWidth));
}
temp.append("\n");

temp.append("\n\nOdds Ratios...\n");
temp.append(Utils.padLeft(" ", attLength)
  + Utils.padLeft("Class", colWidth) + "\n");
temp.append(Utils.padRight("Variable", attLength));

for (int i = 0; i < m_NumClasses - 1; i++) {
  String className = m_structure.classAttribute().value(i);
  temp.append(Utils.padLeft(className, colWidth));
}
temp.append("\n");
for (int i = 0; i < separatorL; i++) {
  temp.append("=");
}
temp.append("\n");

j = 1;
for (int i = 0; i < m_structure.numAttributes(); i++) {
  if (i != m_structure.classIndex()) {
    temp.append(Utils.padRight(m_structure.attribute(i).name(), attLength));
```

```
      for (int k = 0; k < m_NumClasses - 1; k++) {
        double ORc = Math.exp(m_Par[j][k]);
        String ORs = " "
          + ((ORc > 1e10) ? "" + ORc : Utils.doubleToString(ORc, 12, 4));
        temp.append(Utils.padLeft(ORs.trim(), colWidth));
      }
      temp.append("\n");
      j++;
    }
  }

  return temp.toString();
}

/**
 * Returns the revision string.
 *
 * @return the revision
 */
@Override
public String getRevision() {
  return RevisionUtils.extract("$Revision: 11247 $");
}

protected int m_numModels = 0;

/**
 * Aggregate an object with this one
 *
```

```
 * @param toAggregate the object to aggregate
 * @return the result of aggregation
 * @throws Exception if the supplied object can't be aggregated for some
 *          reason
 */
@Override
public Logistic aggregate(Logistic toAggregate) throws Exception {
  if (m_numModels == Integer.MIN_VALUE) {
    throw new Exception("Can't aggregate further - model has already been "
      + "aggregated and finalized");
  }

  if (m_Par == null) {
    throw new Exception("No model built yet, can't aggregate");
  }

  if (!m_structure.equalHeaders(toAggregate.m_structure)) {
    throw new Exception("Can't aggregate - data headers dont match: "
      + m_structure.equalHeadersMsg(toAggregate.m_structure));
  }

  for (int i = 0; i < m_Par.length; i++) {
    for (int j = 0; j < m_Par[i].length; j++) {
      m_Par[i][j] += toAggregate.m_Par[i][j];
    }
  }

  m_numModels++;
```

```
  return this;
}

/**
 * Call to complete the aggregation process. Allows implementers to do any
 * final processing based on how many objects were aggregated.
 *
 * @throws Exception if the aggregation can't be finalized for some reason
 */
@Override
public void finalizeAggregation() throws Exception {

  if (m_numModels == Integer.MIN_VALUE) {
    throw new Exception("Aggregation has already been finalized");
  }

  if (m_numModels == 0) {
    throw new Exception("Unable to finalize aggregation - "
      + "haven't seen any models to aggregate");
  }

  for (int i = 0; i < m_Par.length; i++) {
    for (int j = 0; j < m_Par[i].length; j++) {
      m_Par[i][j] /= (m_numModels + 1);
    }
  }

  // aggregation complete
  m_numModels = Integer.MIN_VALUE;
```

```
  }

  /**
   * Main method for testing this class.
   *
   * @param argv should contain the command line arguments to the scheme (see
   *          Evaluation)
   */
  public static void main(String[] argv) {
    runClassifier(new Logistic(), argv);
  }

  /**
   * Produce a PMML representation of this logistic model
   *
   * @param train the training data that was used to construct the model
   *
   * @return a string containing the PMML representation
   */
  @Override
  public String toPMML(Instances train) {
    return LogisticProducerHelper.toPMML(train, m_structure, m_Par,
      m_NumClasses);
  }
}
```

J48 DECISION TREE

The source code for the J48 Decision Tree Algorithm in WEKA is obtained from the following website:

http://grepcode.com/file/repo1.maven.org/maven2/nz.ac.waikato.cms.weka/weka-dev/3.7.12/weka/classifiers/trees/j48/ClassifierTree.java?av=f

```
/*
 *   This program is free software: you can redistribute it and/or modify
 *   it under the terms of the GNU General Public License as published by
 *   the Free Software Foundation, either version 3 of the License, or
 *   (at your option) any later version.
 *
 *   This program is distributed in the hope that it will be useful,
 *   but WITHOUT ANY WARRANTY; without even the implied warranty of
 *   MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *   GNU General Public License for more details.
 *
 *   You should have received a copy of the GNU General Public License
 *   along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */


/*
 *    ClassifierTree.java
 *    Copyright (C) 1999-2012 University of Waikato, Hamilton, New Zealand
 *
 */


package weka.classifiers.trees.j48;
```

```
import java.io.Serializable;
import java.util.LinkedList;
import java.util.Queue;

import weka.core.Capabilities;
import weka.core.CapabilitiesHandler;
import weka.core.Drawable;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.RevisionHandler;
import weka.core.RevisionUtils;
import weka.core.Utils;

/**
 * Class for handling a tree structure used for classification.
 *
 * @author Eibe Frank (eibe@cs.waikato.ac.nz)
 * @version $Revision: 11269 $
 */
public class ClassifierTree implements Drawable, Serializable,
  CapabilitiesHandler, RevisionHandler {

  /** for serialization */
  static final long serialVersionUID = -8722249377542734193L;

  /** The model selection method. */
  protected ModelSelection m_toSelectModel;
```

```
/** Local model at node. */
protected ClassifierSplitModel m_localModel;

/** References to sons. */
protected ClassifierTree[] m_sons;

/** True if node is leaf. */
protected boolean m_isLeaf;

/** True if node is empty. */
protected boolean m_isEmpty;

/** The training instances. */
protected Instances m_train;

/** The pruning instances. */
protected Distribution m_test;

/** The id for the node. */
protected int m_id;

/**
 * For getting a unique ID when outputting the tree (hashcode isn't guaranteed
 * unique)
 */
private static long PRINTED_NODES = 0;
```

```
/**
 * Gets the next unique node ID.
 *
 * @return the next unique node ID.
 */
protected static long nextID() {

  return PRINTED_NODES++;
}

/**
 * Resets the unique node ID counter (e.g. between repeated separate print
 * types)
 */
protected static void resetID() {

  PRINTED_NODES = 0;
}

/**
 * Constructor.
 */
public ClassifierTree(ModelSelection toSelectLocModel) {

  m_toSelectModel = toSelectLocModel;
}

/**
```

```
 * Returns default capabilities of the classifier tree.
 *
 * @return the capabilities of this classifier tree
 */
@Override
public Capabilities getCapabilities() {
  Capabilities result = new Capabilities(this);
  result.enableAll();

  return result;
}

/**
 * Method for building a classifier tree.
 *
 * @param data the data to build the tree from
 * @throws Exception if something goes wrong
 */
public void buildClassifier(Instances data) throws Exception {

  // can classifier tree handle the data?
  getCapabilities().testWithFail(data);

  // remove instances with missing class
  data = new Instances(data);
  data.deleteWithMissingClass();

  buildTree(data, false);
```

```
}

/**
 * Builds the tree structure.
 *
 * @param data the data for which the tree structure is to be generated.
 * @param keepData is training data to be kept?
 * @throws Exception if something goes wrong
 */
public void buildTree(Instances data, boolean keepData) throws Exception {

  Instances[] localInstances;

  if (keepData) {
    m_train = data;
  }
  m_test = null;
  m_isLeaf = false;
  m_isEmpty = false;
  m_sons = null;
  m_localModel = m_toSelectModel.selectModel(data);
  if (m_localModel.numSubsets() > 1) {
    localInstances = m_localModel.split(data);
    data = null;
    m_sons = new ClassifierTree[m_localModel.numSubsets()];
    for (int i = 0; i < m_sons.length; i++) {
      m_sons[i] = getNewTree(localInstances[i]);
      localInstances[i] = null;
```

```
    }
  } else {
    m_isLeaf = true;
    if (Utils.eq(data.sumOfWeights(), 0)) {
      m_isEmpty = true;
    }
    data = null;
  }
}

/**
 * Builds the tree structure with hold out set
 *
 * @param train the data for which the tree structure is to be generated.
 * @param test the test data for potential pruning
 * @param keepData is training Data to be kept?
 * @throws Exception if something goes wrong
 */
public void buildTree(Instances train, Instances test, boolean keepData)
  throws Exception {

  Instances[] localTrain, localTest;
  int i;

  if (keepData) {
    m_train = train;
  }
  m_isLeaf = false;
```

```
  m_isEmpty = false;
  m_sons = null;
  m_localModel = m_toSelectModel.selectModel(train, test);
  m_test = new Distribution(test, m_localModel);
  if (m_localModel.numSubsets() > 1) {
    localTrain = m_localModel.split(train);
    localTest = m_localModel.split(test);
    train = null;
    test = null;
    m_sons = new ClassifierTree[m_localModel.numSubsets()];
    for (i = 0; i < m_sons.length; i++) {
      m_sons[i] = getNewTree(localTrain[i], localTest[i]);
      localTrain[i] = null;
      localTest[i] = null;
    }
  } else {
    m_isLeaf = true;
    if (Utils.eq(train.sumOfWeights(), 0)) {
      m_isEmpty = true;
    }
    train = null;
    test = null;
  }
}

/**
 * Classifies an instance.
 *
```

```
 * @param instance the instance to classify
 * @return the classification
 * @throws Exception if something goes wrong
 */
public double classifyInstance(Instance instance) throws Exception {

  double maxProb = -1;
  double currentProb;
  int maxIndex = 0;
  int j;

  for (j = 0; j < instance.numClasses(); j++) {
    currentProb = getProbs(j, instance, 1);
    if (Utils.gr(currentProb, maxProb)) {
      maxIndex = j;
      maxProb = currentProb;
    }
  }

  return maxIndex;
}

/**
 * Cleanup in order to save memory.
 *
 * @param justHeaderInfo
 */
public final void cleanup(Instances justHeaderInfo) {
```

```
  m_train = justHeaderInfo;
  m_test = null;
  if (!m_isLeaf) {
    for (ClassifierTree m_son : m_sons) {
      m_son.cleanup(justHeaderInfo);
    }
  }
}

/**
 * Returns class probabilities for a weighted instance.
 *
 * @param instance the instance to get the distribution for
 * @param useLaplace whether to use laplace or not
 * @return the distribution
 * @throws Exception if something goes wrong
 */
public final double[] distributionForInstance(Instance instance,
  boolean useLaplace) throws Exception {

  double[] doubles = new double[instance.numClasses()];

  for (int i = 0; i < doubles.length; i++) {
    if (!useLaplace) {
      doubles[i] = getProbs(i, instance, 1);
    } else {
      doubles[i] = getProbsLaplace(i, instance, 1);
```

```
    }
  }

  return doubles;
}

/**
 * Assigns a uniqe id to every node in the tree.
 *
 * @param lastID the last ID that was assign
 * @return the new current ID
 */
public int assignIDs(int lastID) {

  int currLastID = lastID + 1;

  m_id = currLastID;
  if (m_sons != null) {
    for (ClassifierTree m_son : m_sons) {
      currLastID = m_son.assignIDs(currLastID);
    }
  }
  return currLastID;
}

/**
 * Returns the type of graph this classifier represents.
 *
```

```
 * @return Drawable.TREE
 */
@Override
public int graphType() {
  return Drawable.TREE;
}

/**
 * Returns graph describing the tree.
 *
 * @throws Exception if something goes wrong
 * @return the tree as graph
 */
@Override
public String graph() throws Exception {

  StringBuffer text = new StringBuffer();

  assignIDs(-1);
  text.append("digraph J48Tree {\n");
  if (m_isLeaf) {
    text.append("N" + m_id + " [label=\""
      + Utils.backQuoteChars(m_localModel.dumpLabel(0, m_train)) + "\" "
      + "shape=box style=filled ");
    if (m_train != null && m_train.numInstances() > 0) {
      text.append("data =\n" + m_train + "\n");
      text.append(",\n");
```

```
    }
    text.append("]\n");
  } else {
    text.append("N" + m_id + " [label=\""
      + Utils.backQuoteChars(m_localModel.leftSide(m_train)) + "\" ");
    if (m_train != null && m_train.numInstances() > 0) {
      text.append("data =\n" + m_train + "\n");
      text.append(",\n");
    }
    text.append("]\n");
    graphTree(text);
  }

  return text.toString() + "}\n";
}

/**
 * Returns tree in prefix order.
 *
 * @throws Exception if something goes wrong
 * @return the prefix order
 */
public String prefix() throws Exception {

  StringBuffer text;

  text = new StringBuffer();
  if (m_isLeaf) {
```

```
    text.append("[" + m_localModel.dumpLabel(0, m_train) + "]");
  } else {
    prefixTree(text);
  }

  return text.toString();
}

/**
 * Returns source code for the tree as an if-then statement. The class is
 * assigned to variable "p", and assumes the tested instance is named "i". The
 * results are returned as two stringbuffers: a section of code for assignment
 * of the class, and a section of code containing support code (eg: other
 * support methods).
 *
 * @param className the classname that this static classifier has
 * @return an array containing two stringbuffers, the first string containing
 *         assignment code, and the second containing source for support code.
 * @throws Exception if something goes wrong
 */
public StringBuffer[] toSource(String className) throws Exception {

  StringBuffer[] result = new StringBuffer[2];
  if (m_isLeaf) {
    result[0] = new StringBuffer("    p = "
      + m_localModel.distribution().maxClass(0) + ";\n");
    result[1] = new StringBuffer("");
  } else {
```

```
StringBuffer text = new StringBuffer();
StringBuffer atEnd = new StringBuffer();

long printID = ClassifierTree.nextID();

text.append("  static double N")
  .append(Integer.toHexString(m_localModel.hashCode()) + printID)
  .append("(Object []i) {\n").append("    double p = Double.NaN;\n");

text.append("    if (")
  .append(m_localModel.sourceExpression(-1, m_train)).append(") {\n");
text.append("      p = ").append(m_localModel.distribution().maxClass(0))
  .append(";\n");
text.append("    } ");
for (int i = 0; i < m_sons.length; i++) {
  text.append("else if (" + m_localModel.sourceExpression(i, m_train)
    + ") {\n");
  if (m_sons[i].m_isLeaf) {
    text.append("      p = " + m_localModel.distribution().maxClass(i)
      + ";\n");
  } else {
    StringBuffer[] sub = m_sons[i].toSource(className);
    text.append(sub[0]);
    atEnd.append(sub[1]);
  }
  text.append("    } ");
  if (i == m_sons.length - 1) {
    text.append('\n');
```

```
      }
    }

    text.append("    return p;\n  }\n");

    result[0] = new StringBuffer("    p = " + className + ".N");
    result[0].append(Integer.toHexString(m_localModel.hashCode()) + printID)
      .append("(i);\n");
    result[1] = text.append(atEnd);
  }
  return result;
}

/**
 * Returns number of leaves in tree structure.
 *
 * @return the number of leaves
 */
public int numLeaves() {

  int num = 0;
  int i;

  if (m_isLeaf) {
    return 1;
  } else {
    for (i = 0; i < m_sons.length; i++) {
      num = num + m_sons[i].numLeaves();
```

```
    }
  }

  return num;
}

/**
 * Returns number of nodes in tree structure.
 *
 * @return the number of nodes
 */
public int numNodes() {

  int no = 1;
  int i;

  if (!m_isLeaf) {
    for (i = 0; i < m_sons.length; i++) {
      no = no + m_sons[i].numNodes();
    }
  }

  return no;
}

/**
 * Prints tree structure.
 *
```

```
 * @return the tree structure
 */
@Override
public String toString() {

  try {
    StringBuffer text = new StringBuffer();

    if (m_isLeaf) {
      text.append(": ");
      text.append(m_localModel.dumpLabel(0, m_train));
    } else {
      dumpTree(0, text);
    }
    text.append("\n\nNumber of Leaves  : \t" + numLeaves() + "\n");
    text.append("\nSize of the tree : \t" + numNodes() + "\n");

    return text.toString();
  } catch (Exception e) {
    return "Can't print classification tree.";
  }
}

/**
 * Returns a newly created tree.
 *
 * @param data the training data
 * @return the generated tree
```

```
 * @throws Exception if something goes wrong
 */
protected ClassifierTree getNewTree(Instances data) throws Exception {

  ClassifierTree newTree = new ClassifierTree(m_toSelectModel);
  newTree.buildTree(data, false);

  return newTree;
}

/**
 * Returns a newly created tree.
 *
 * @param train the training data
 * @param test the pruning data.
 * @return the generated tree
 * @throws Exception if something goes wrong
 */
protected ClassifierTree getNewTree(Instances train, Instances test)
  throws Exception {

  ClassifierTree newTree = new ClassifierTree(m_toSelectModel);
  newTree.buildTree(train, test, false);

  return newTree;
}

/**
```

```
 * Help method for printing tree structure.
 *
 * @param depth the current depth
 * @param text for outputting the structure
 * @throws Exception if something goes wrong
 */
private void dumpTree(int depth, StringBuffer text) throws Exception {

  int i, j;

  for (i = 0; i < m_sons.length; i++) {
    text.append("\n");
    ;
    for (j = 0; j < depth; j++) {
      text.append("|   ");
    }
    text.append(m_localModel.leftSide(m_train));
    text.append(m_localModel.rightSide(i, m_train));
    if (m_sons[i].m_isLeaf) {
      text.append(": ");
      text.append(m_localModel.dumpLabel(i, m_train));
    } else {
      m_sons[i].dumpTree(depth + 1, text);
    }
  }
}

/**
```

```
 * Help method for printing tree structure as a graph.
 *
 * @param text for outputting the tree
 * @throws Exception if something goes wrong
 */
private void graphTree(StringBuffer text) throws Exception {


  for (int i = 0; i < m_sons.length; i++) {
    text.append("N" + m_id + "->" + "N" + m_sons[i].m_id + " [label=\""
      + Utils.backQuoteChars(m_localModel.rightSide(i, m_train).trim())
      + "\"]\n");
    if (m_sons[i].m_isLeaf) {
      text.append("N" + m_sons[i].m_id + " [label=\""
        + Utils.backQuoteChars(m_localModel.dumpLabel(i, m_train)) + "\" "
        + "shape=box style=filled ");
      if (m_train != null && m_train.numInstances() > 0) {
        text.append("data =\n" + m_sons[i].m_train + "\n");
        text.append(",\n");
      }
      text.append("]\n");
    } else {
      text.append("N" + m_sons[i].m_id + " [label=\""
        + Utils.backQuoteChars(m_sons[i].m_localModel.leftSide(m_train))
        + "\" ");
      if (m_train != null && m_train.numInstances() > 0) {
        text.append("data =\n" + m_sons[i].m_train + "\n");
        text.append(",\n");
      }
```

```
      text.append("]\n");
      m_sons[i].graphTree(text);
    }
  }
}

/**
 * Prints the tree in prefix form
 *
 * @param text the buffer to output the prefix form to
 * @throws Exception if something goes wrong
 */
private void prefixTree(StringBuffer text) throws Exception {

  text.append("[");
  text.append(m_localModel.leftSide(m_train) + ":");
  for (int i = 0; i < m_sons.length; i++) {
    if (i > 0) {
      text.append(",\n");
    }
    text.append(m_localModel.rightSide(i, m_train));
  }
  for (int i = 0; i < m_sons.length; i++) {
    if (m_sons[i].m_isLeaf) {
      text.append("[");
      text.append(m_localModel.dumpLabel(i, m_train));
      text.append("]");
    } else {
```

```
      m_sons[i].prefixTree(text);
    }
  }
  text.append("]");
}

/**
 * Help method for computing class probabilities of a given instance.
 *
 * @param classIndex the class index
 * @param instance the instance to compute the probabilities for
 * @param weight the weight to use
 * @return the laplace probs
 * @throws Exception if something goes wrong
 */
private double getProbsLaplace(int classIndex, Instance instance,
  double weight) throws Exception {

  double prob = 0;

  if (m_isLeaf) {
    return weight * localModel().classProbLaplace(classIndex, instance, -1);
  } else {
    int treeIndex = localModel().whichSubset(instance);
    if (treeIndex == -1) {
      double[] weights = localModel().weights(instance);
      for (int i = 0; i < m_sons.length; i++) {
        if (!son(i).m_isEmpty) {
```

```
          prob += son(i).getProbsLaplace(classIndex, instance,
            weights[i] * weight);
        }
      }
      return prob;
    } else {
      if (son(treeIndex).m_isEmpty) {
        return weight
          * localModel().classProbLaplace(classIndex, instance, treeIndex);
      } else {
        return son(treeIndex).getProbsLaplace(classIndex, instance, weight);
      }
    }
  }
}

/**
 * Help method for computing class probabilities of a given instance.
 *
 * @param classIndex the class index
 * @param instance the instance to compute the probabilities for
 * @param weight the weight to use
 * @return the probs
 * @throws Exception if something goes wrong
 */
private double getProbs(int classIndex, Instance instance, double weight)
  throws Exception {
```

```
  double prob = 0;


  if (m_isLeaf) {
    return weight * localModel().classProb(classIndex, instance, -1);
  } else {
    int treeIndex = localModel().whichSubset(instance);
    if (treeIndex == -1) {
      double[] weights = localModel().weights(instance);
      for (int i = 0; i < m_sons.length; i++) {
        if (!son(i).m_isEmpty) {
          prob += son(i).getProbs(classIndex, instance, weights[i] * weight);
        }
      }
      return prob;
    } else {
      if (son(treeIndex).m_isEmpty) {
        return weight
          * localModel().classProb(classIndex, instance, treeIndex);
      } else {
        return son(treeIndex).getProbs(classIndex, instance, weight);
      }
    }
  }
}


/**
 * Method just exists to make program easier to read.
 */
```

```
private ClassifierSplitModel localModel() {

  return m_localModel;
}

/**
 * Method just exists to make program easier to read.
 */
private ClassifierTree son(int index) {

  return m_sons[index];
}

/**
 * Computes a list that indicates node membership
 */
public double[] getMembershipValues(Instance instance) throws Exception {

  // Set up array for membership values
  double[] a = new double[numNodes()];

  // Initialize queues
  Queue<Double> queueOfWeights = new LinkedList<Double>();
  Queue<ClassifierTree> queueOfNodes = new LinkedList<ClassifierTree>();
  queueOfWeights.add(instance.weight());
  queueOfNodes.add(this);
  int index = 0;
```

```
// While the queue is not empty
while (!queueOfNodes.isEmpty()) {

  a[index++] = queueOfWeights.poll();
  ClassifierTree node = queueOfNodes.poll();

  // Is node a leaf?
  if (node.m_isLeaf) {
    continue;
  }

  // Which subset?
  int treeIndex = node.localModel().whichSubset(instance);

  // Space for weight distribution
  double[] weights = new double[node.m_sons.length];

  // Check for missing value
  if (treeIndex == -1) {
    weights = node.localModel().weights(instance);
  } else {
    weights[treeIndex] = 1.0;
  }
  for (int i = 0; i < node.m_sons.length; i++) {
    queueOfNodes.add(node.son(i));
    queueOfWeights.add(a[index - 1] * weights[i]);
  }
}
```

```
    return a;
  }


  /**
   * Returns the revision string.
   *
   * @return the revision
   */
  @Override
  public String getRevision() {
    return RevisionUtils.extract("$Revision: 11269 $");
  }
}
```