Data Mining - Final Project

Name - **Shreerang Paradkar**
ucid - **scp37**
Email - **scp37@njit.edu**

Option - 1 (Supervised Data Mining - Classification)

Category 3 - **Decision Trees (J48) - Weka**

Category 5 - **Naive Bayes - NaiveBayes - Weka**

Test Mode – **Training Set, 10-Fold Cross Validation**

Data Set – Breast Cancer Wisconsin UCI Machine Learning Repository

CONTENTS

Data set - Breast Cancer Prediction

Link - http://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/

- Data Set has a total of **699 Instances**.
- Data Set consists of **10 plus the class attribute**. They are

Attribute	Value
Sample code number	Id number
Clump Thickness	1 – 10
Uniformity of Cell Size	1 – 10
Uniformity of Cell Shape	1 – 10
Marginal Adhesion	1 – 10
Single Epithelial Cell Size	1 – 10
Bare Nuclei	1 – 10
Bland Chromatin	1 – 10
Normal Nucleoli	1 – 10
Mitoses	1 – 10
Class	2 - Benign 4 - Malignant

- Class attribute has values 2 and 4 in the dataset where 2 is for benign and 4 for malignant, so values 2 and 4 are replaced with benign and malignant.
- Class Distribution
  a. Benign: 458 (65.5%)
  b. Malignant: 241 (34.5%)

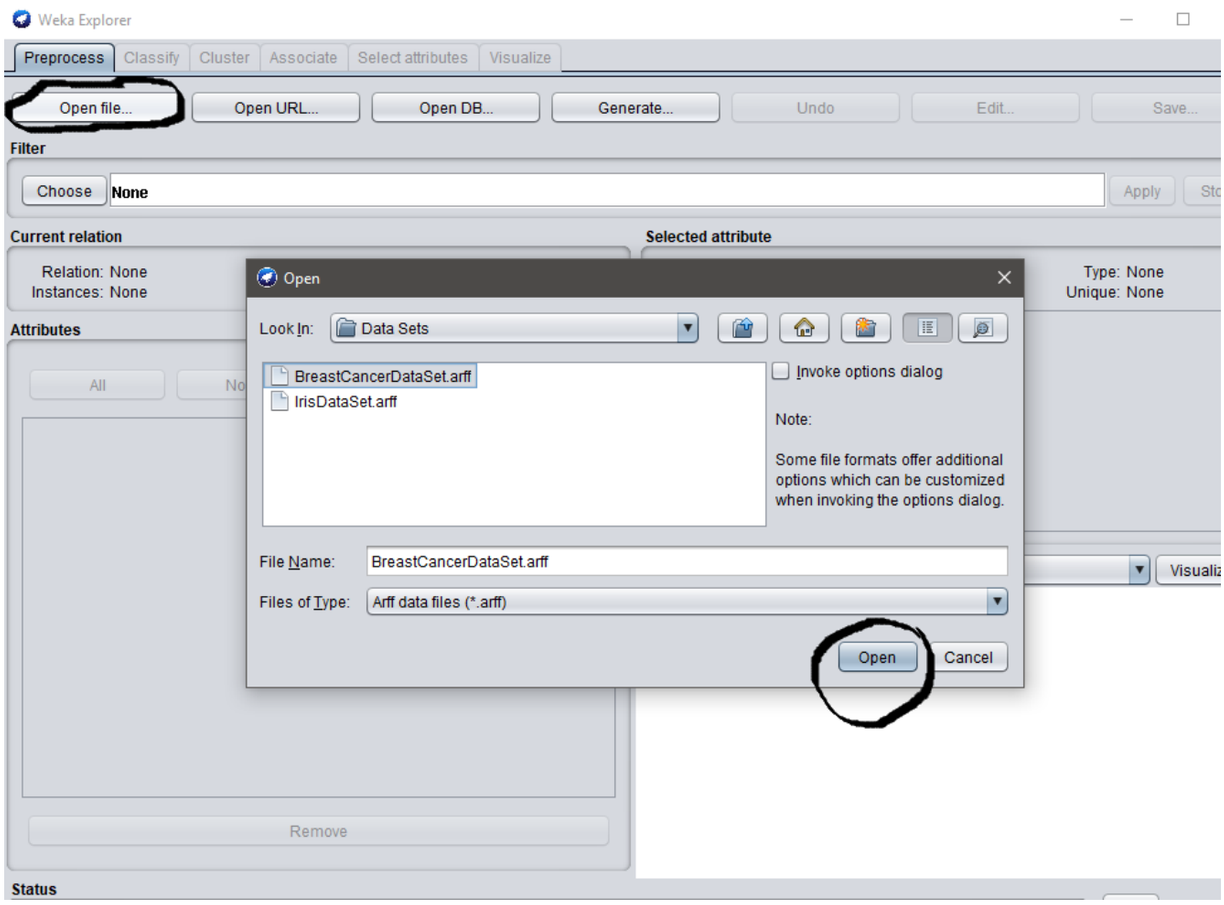
Manual

Weka contains a collection of visualization tools and algorithms for data analysis and predictive modelling, together with graphical user interfaces for easy access to these functions.
In this example, we are going to use 2 options under Explorer module of Weka for data analysis and prediction.

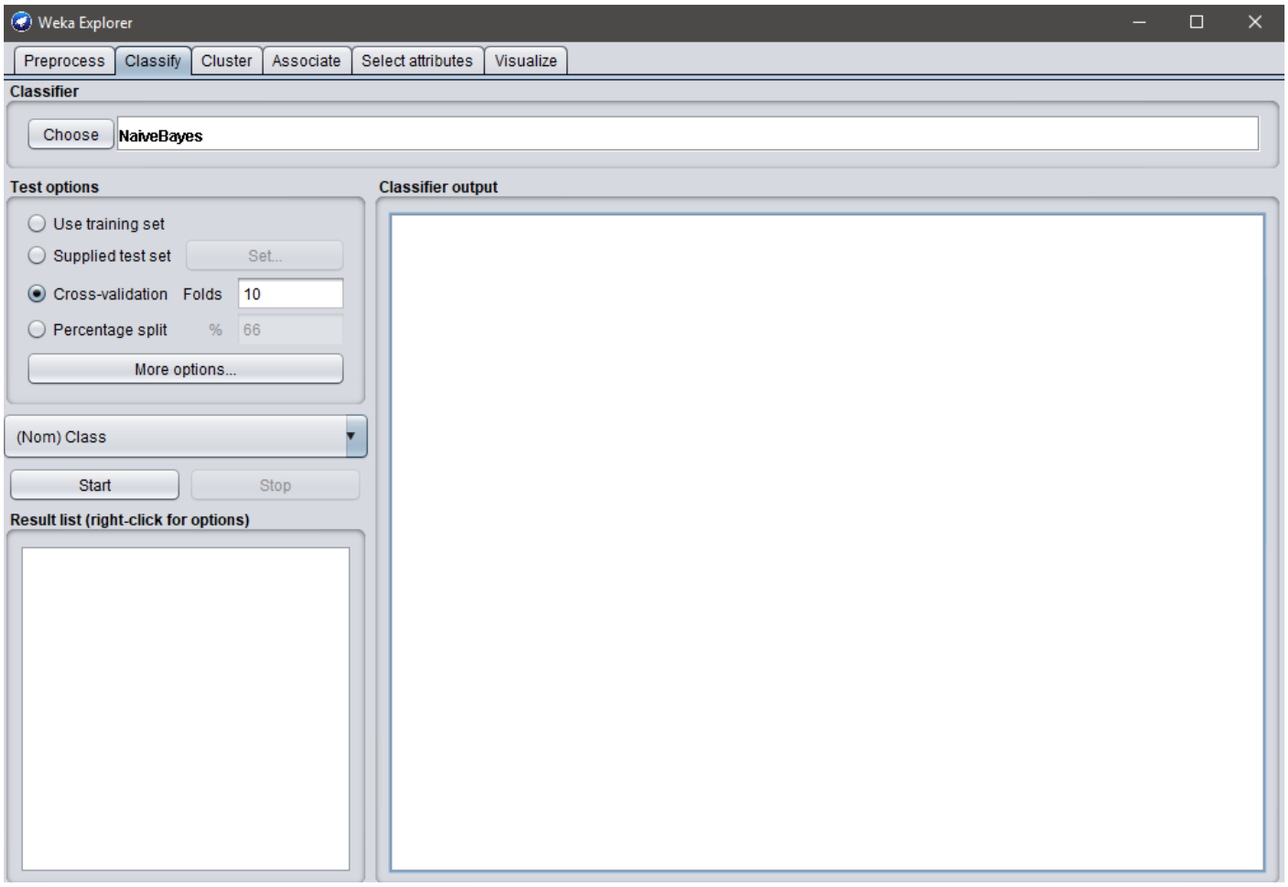
1. Preprocess.
2. Classify.

1. PREPROCESS

- Step 1 – Convert the text file into CSV file.
- Step 2 – Convert the CSV into ARFF file Format.
- Load the file in Weka Explorer.



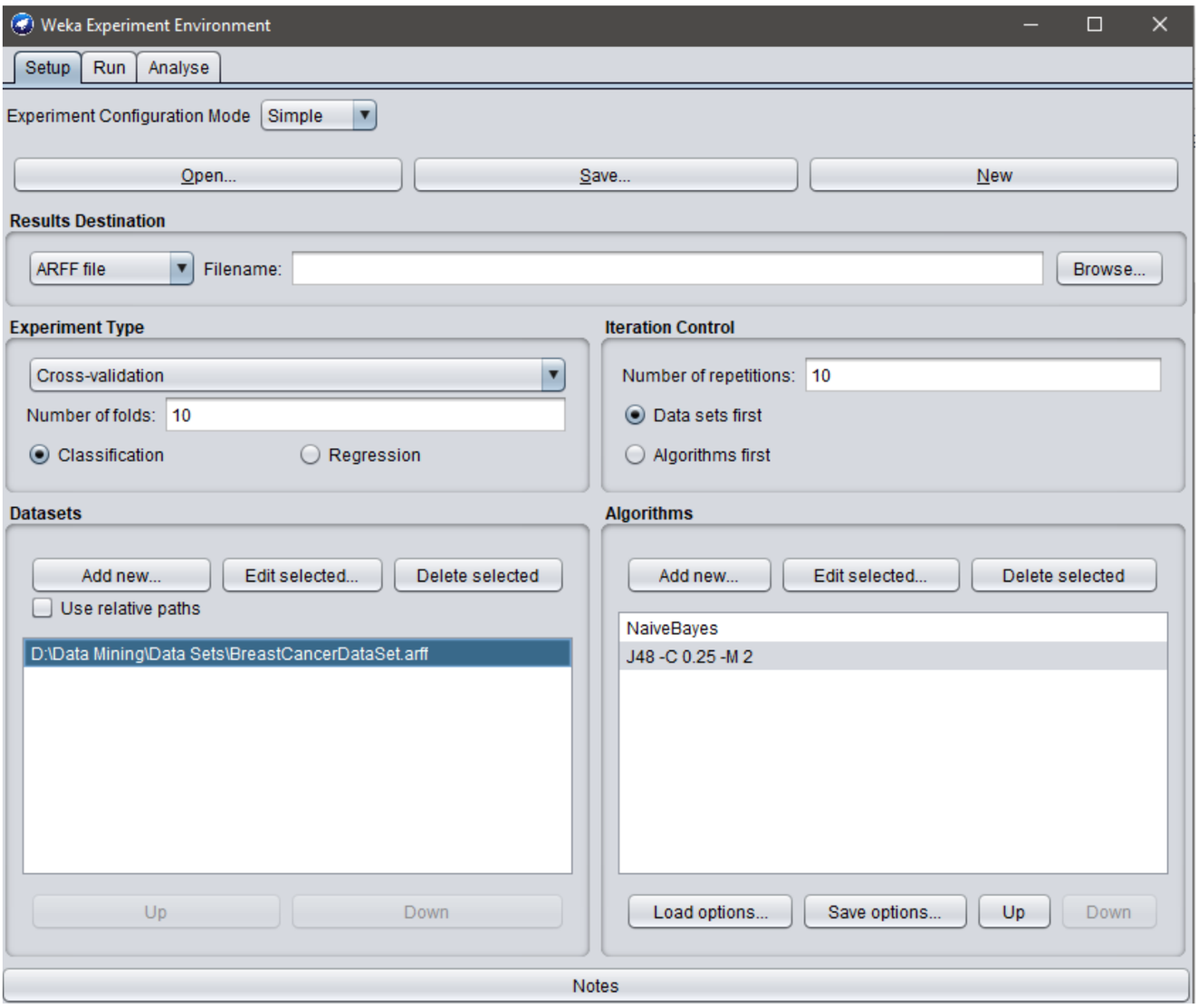
2. CLASSIFY



The above Figure shows the screens for Module Classify under Explorer Option.

1. Classifier – Select the Algorithm which we want to use for the selected Data Set.
2. Test Options
   a. Use training set. The classifier is evaluated on how well it predicts the class of the instances it was trained on.
   b. Supplied test set. The classifier is evaluated on how well it predicts the class of a set of instances loaded from a file.
   c. Cross-validation. The classifier is evaluated by cross-validation, using the number of folds that are entered in the Folds text field.
   d. Percentage split. The classifier is evaluated on how well it predicts a certain percentage of the data which is held out for testing. The amount of data held out depends on the value entered in the % field.
3. Classifier Output
   a. Summary – A list of statistics summarizing how accurately the classifier was able to predict the true class of the instances under the chosen test mode.
   b. Classifier model (full training set) – A textual representation of the classification model that was produced on the full training data.
   c. Detailed Accuracy by Class – It represents a more detailed per-class break down of the classifier's prediction accuracy.
   d. Confusion Matrix – It shows number of instances that have been assigned to each class. Elements show the number of test examples whose actual class is the row and whose predicted class is the column.
4. Result List
   a. Visualize classifier errors – Shows up a visualization window with correctly classified instances represented by crosses and incorrectly classified ones show up as squares.
   b. Visualize tree or Visualize graph. Brings up a graphical representation of the structure of the classifier model, if possible (i.e. for decision trees or Bayesian networks).
   c. Visualize margin curve – Generates a plot illustrating the prediction margin which is difference between the probability predicted for the actual class and the highest probability predicted for the other classes.
   d. Visualize threshold curve – Generates a plot illustrating the trade-offs in prediction that are obtained by varying the threshold value between classes.
   e. Visualize cost curve – Generates a plot that gives an explicit representation of the expected cost.

EXPERIMENTER



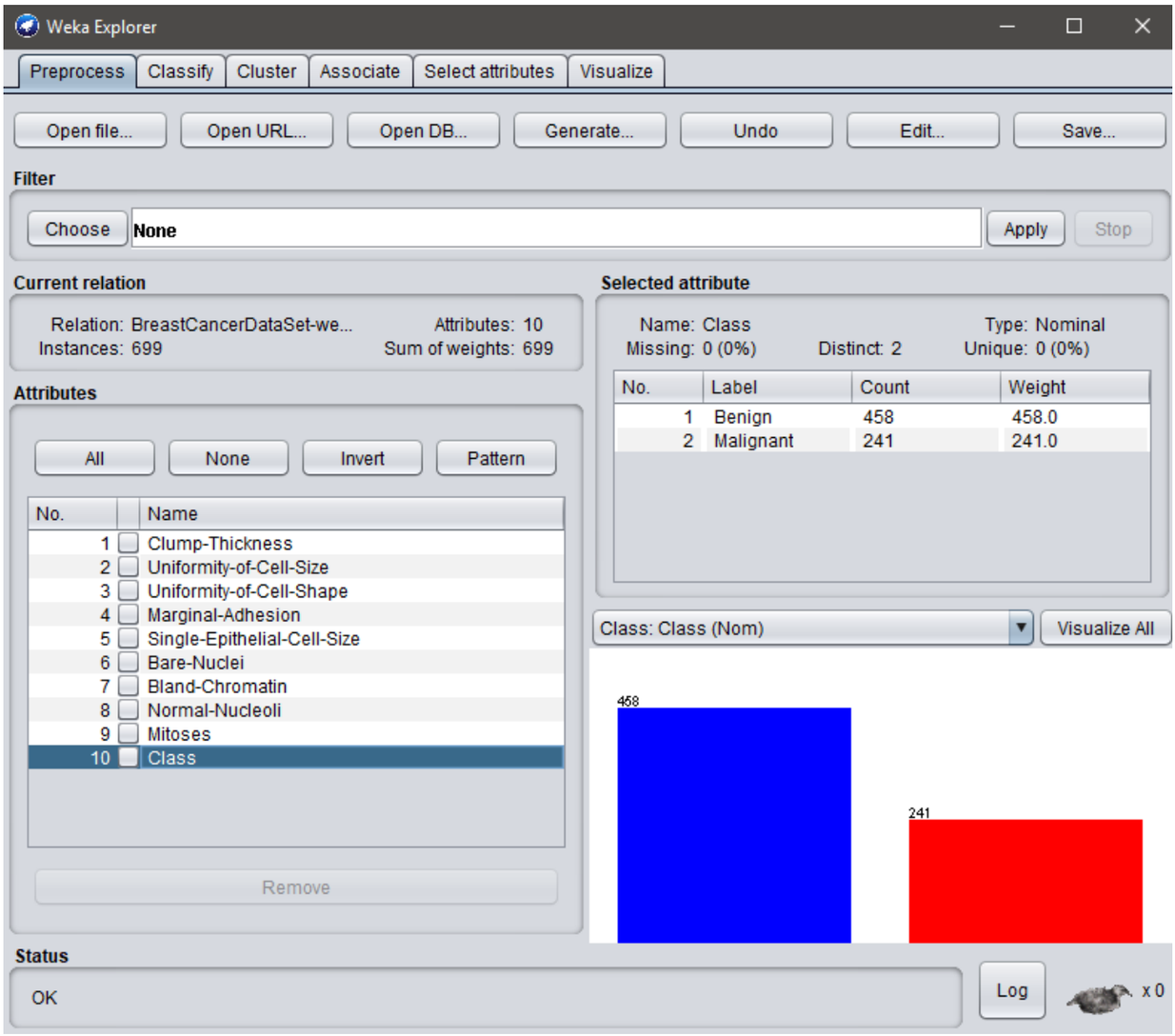
- Select the Experimenter button in Weka GUI to compare 2 algorithms. The above figure (window) appears.
- It has 3 options
  a. SETUP
     - ❖ Click the "New" button to start the experiment.
     - ❖ In "Datasets" pane, click on "Add new" button and select the dataset which you want.
     - ❖ In "Algorithms" pane, click on "Add new" button and select the algorithms which you want to compare.

  b. RUN
     - ❖ Click the "Start" button to start the experiment

  c. ANALYSE
     - ❖ Load the results from the experiment we just executed by clicking the "Experiment" button in the "Source" pane.
     - ❖ Results are collected for many different performance measures, such as classification accuracy.
     - ❖ The Experiment Environment allows us to perform statistical tests on the different performance measures to allow us to draw conclusions from the experiment.

PREPROCESS

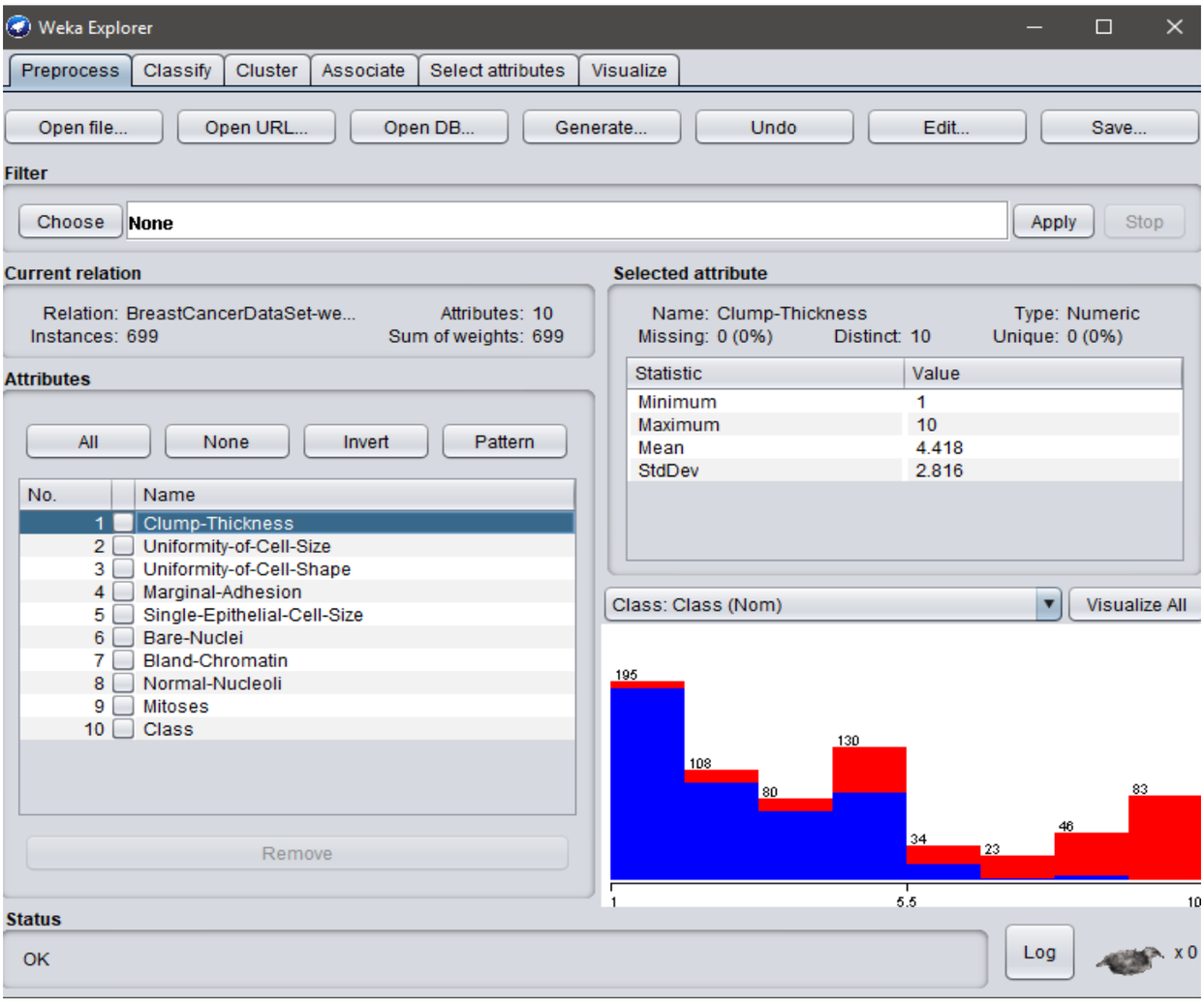
a. Attribute – CLASS



Conclusion

- The above Histogram consists of 2 bar Graphs, one with Blue and other with Red
- Blue indicates the number of patients tested with Benign and Red indicates number of patients tested with Malignant.
- Class Distribution
  1. Benign – 458 (65.5%)
  2. Malignant – 241 (34.5%)

b. Attribute – CLUMP THICKNESS

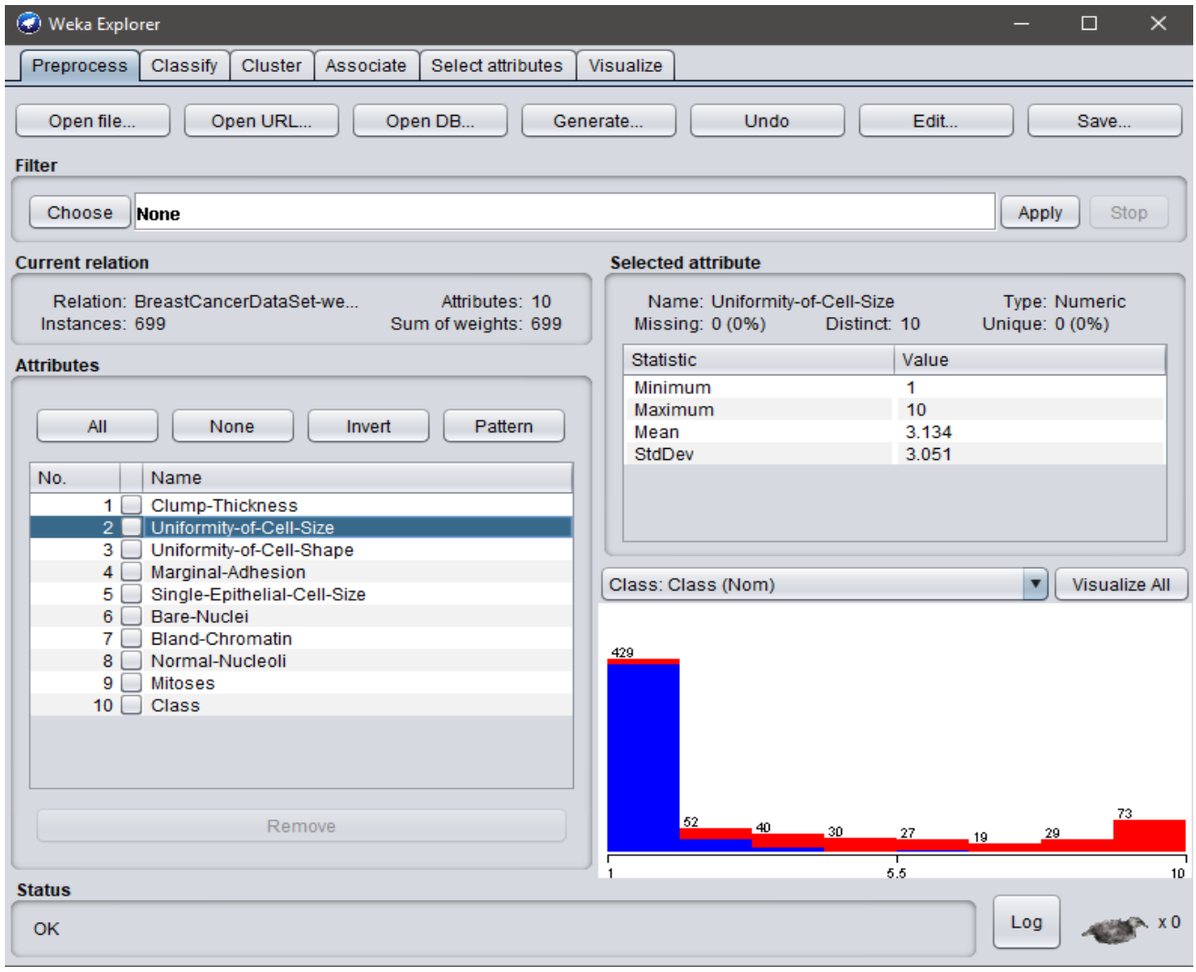


- The above Figure shows Histogram for Attribute Clump-Thickness.
- Blue is for Benign and Red is for Malignant.

➢ Conclusions

1. For Clump Thickness below 5.5, it is unlikely that the patient has Breast Cancer as Histogram clearly shows Blue Bar for values less than 5.5

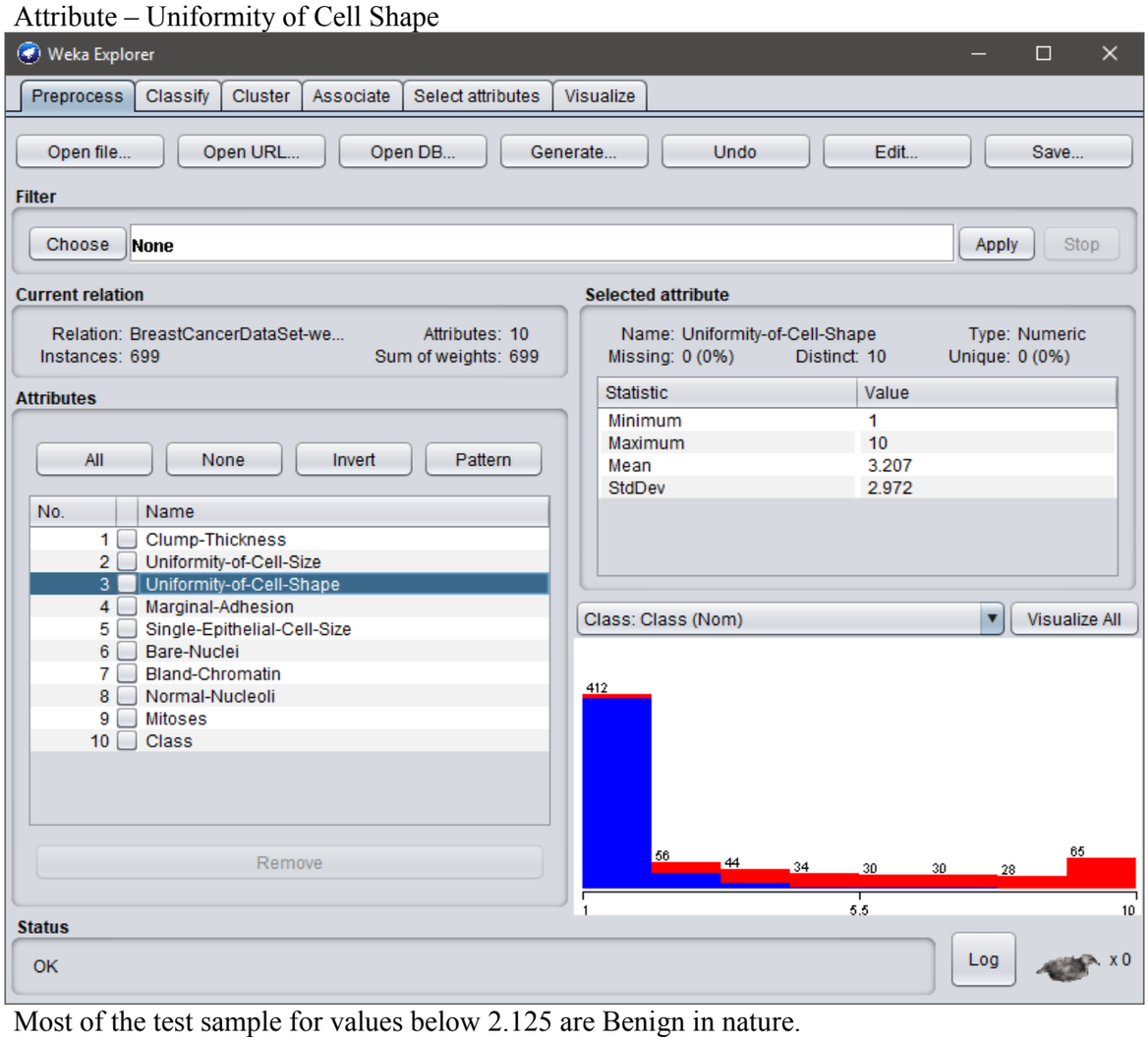
c. Attribute – Uniformity of Cell Shape



➢ Conclusions

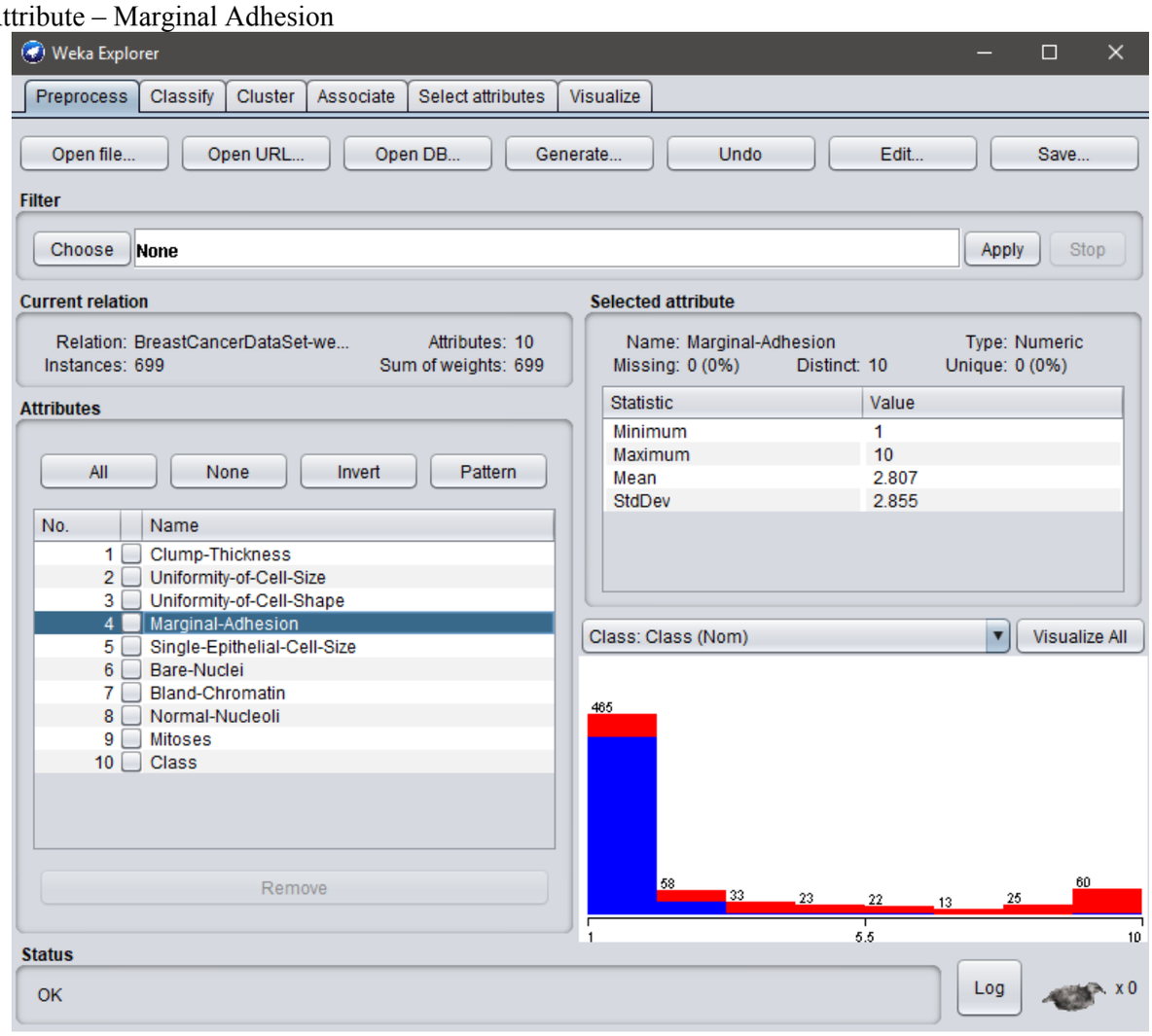
1. For uniformity of cell size, values above 3.25 are likely to lead to Malignant behavior of the disease.

d. Attribute – Uniformity of Cell Shape

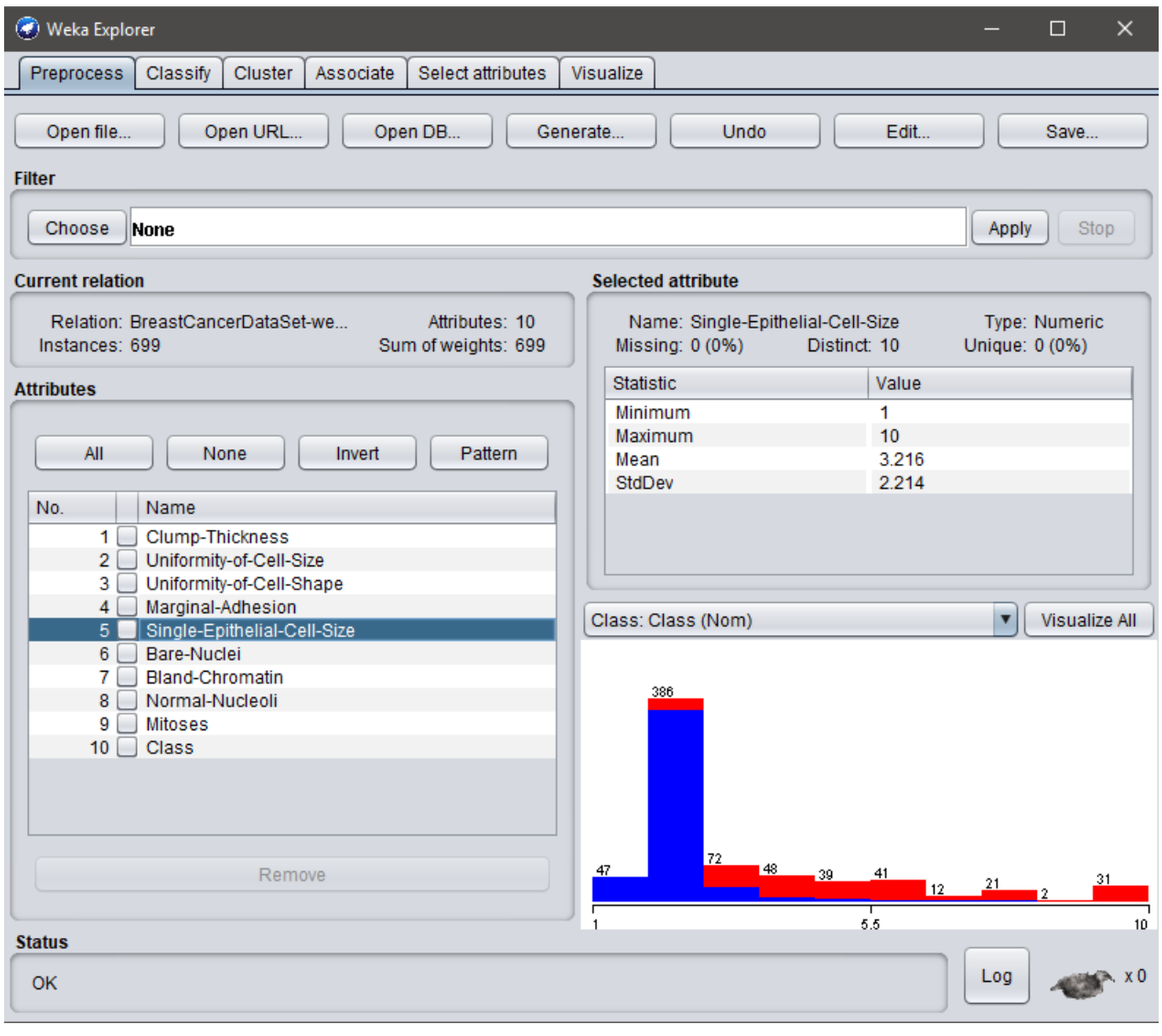


➢ Most of the test sample for values below 2.125 are Benign in nature.

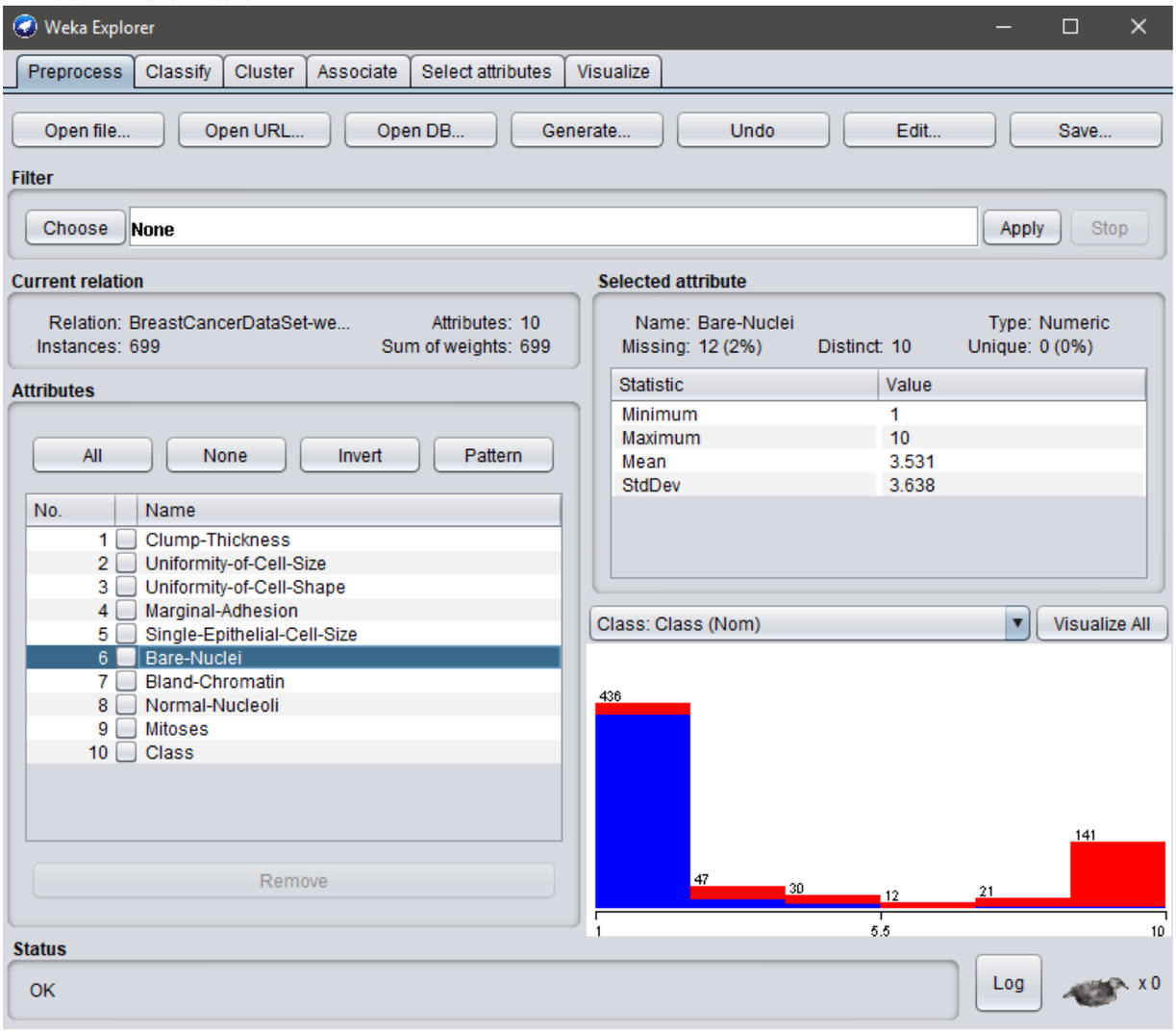
e. Attribute – Marginal Adhesion



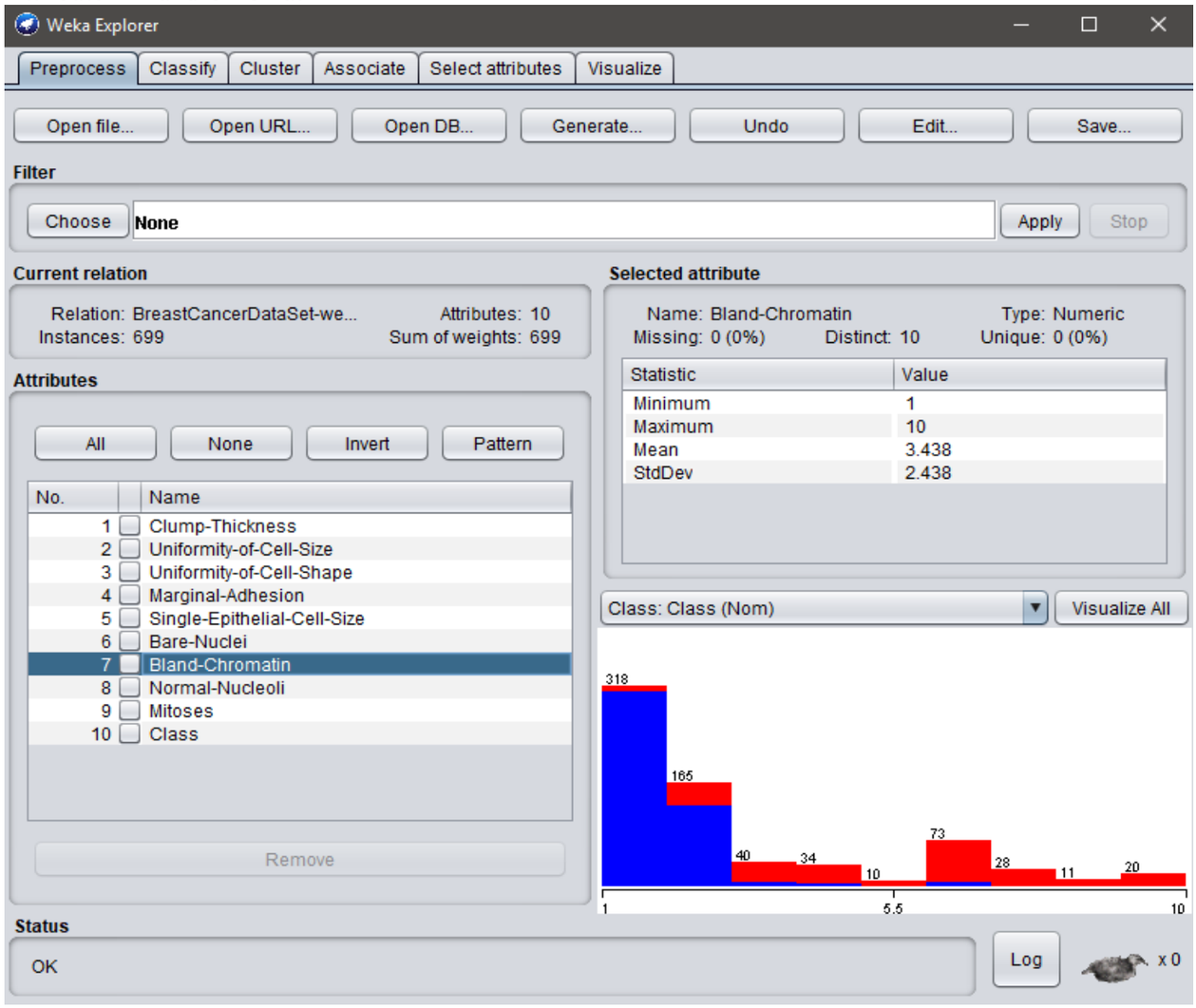
f. Attribute – Single Epithelial Cell Size



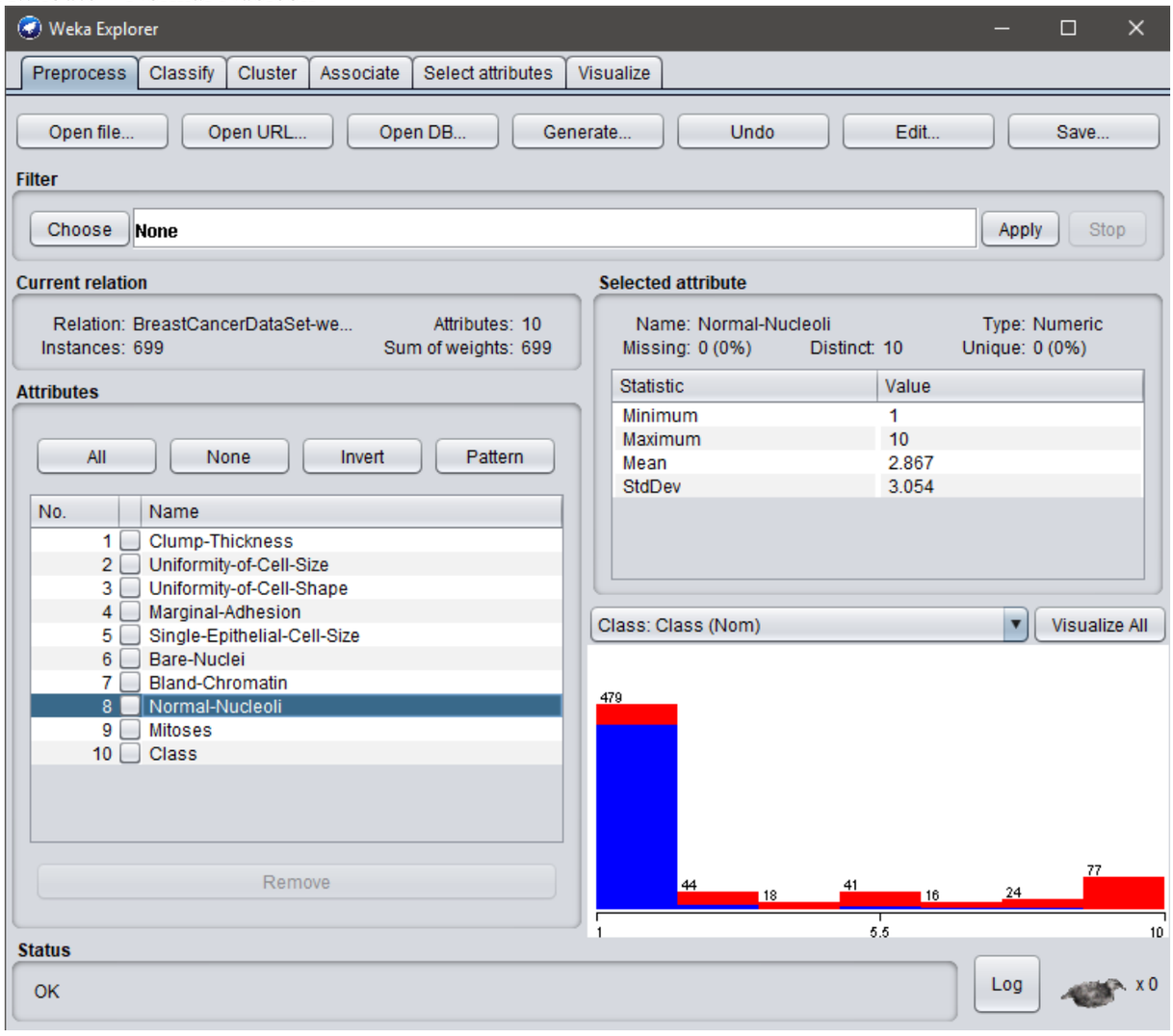
g. Attribute – Bare Nuclei



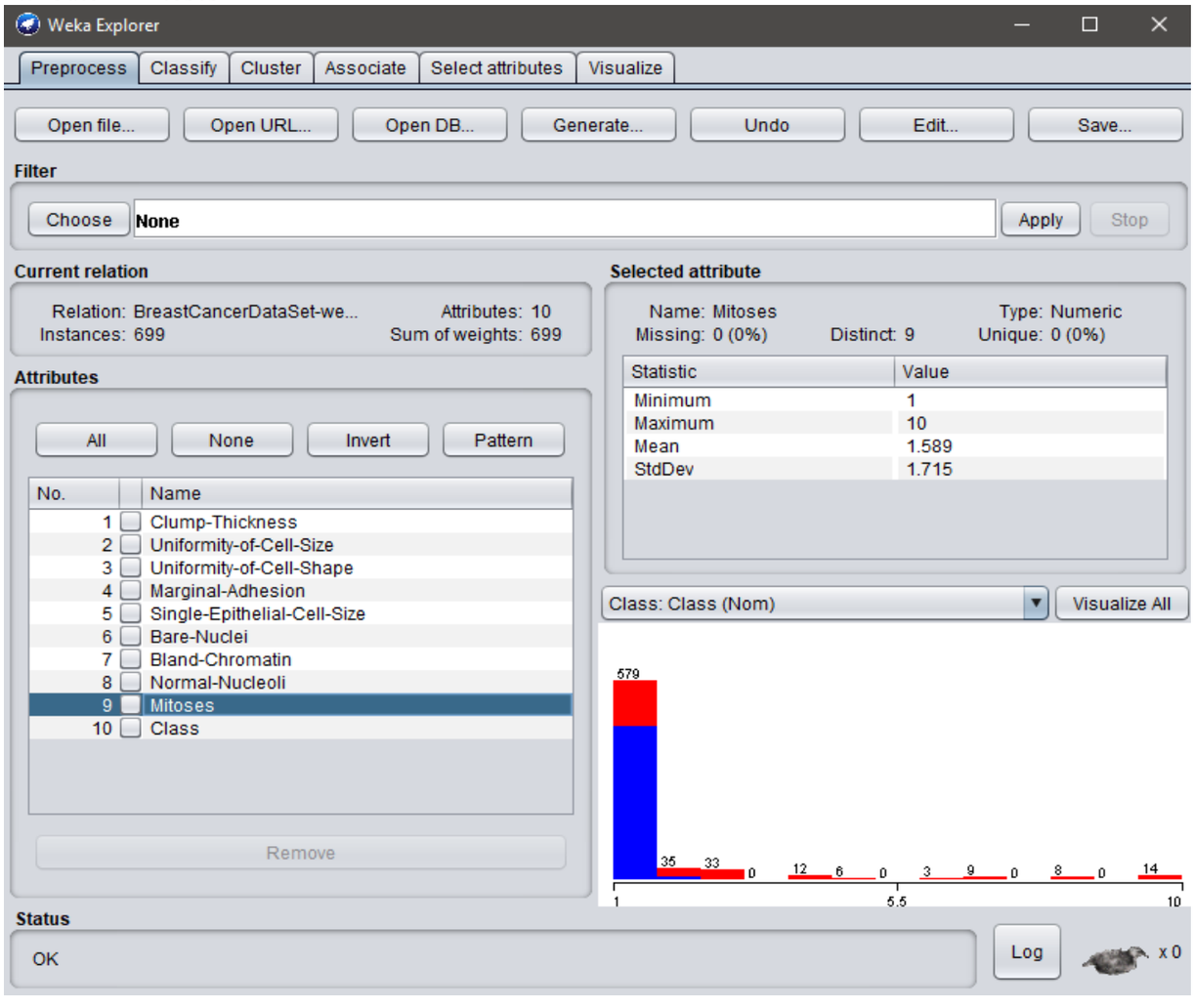
h. Attribute – Bland Chromatin



i. Attribute – Normal Nucleoli



j. Attribute – Mitoses



Category 3 - Decision Trees (J48) – Weka

- C4.5 is an algorithm used to generate a decision tree developed by Ross Quinlan. C4.5 is an extension of Quinlan's earlier ID3 algorithm. The decision trees generated by C4.5 can be used for classification. J48 is an open source Java implementation of the C4.5 algorithm in the Weka data mining tool.

1. Source Code - (Taken from Weka Tool)

- After Installing Weka, one jar file got generated "weka-src.jar". Imported the Jar File in Eclipse and Extracted it.
- Once this was done, source code got generated.

J48.java

```
/*
 *   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/>.
 */

/*
 *    J48.java
 *    Copyright (C) 1999-2012 University of Waikato, Hamilton, New Zealand
 *
 */

package weka.classifiers.trees;

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

import weka.classifiers.AbstractClassifier;
import weka.classifiers.Sourcable;
import weka.classifiers.trees.j48.BinC45ModelSelection;
import weka.classifiers.trees.j48.C45ModelSelection;
import weka.classifiers.trees.j48.C45PruneableClassifierTree;
import weka.classifiers.trees.j48.ClassifierTree;
import weka.classifiers.trees.j48.ModelSelection;
import weka.classifiers.trees.j48.PruneableClassifierTree;
import weka.core.AdditionalMeasureProducer;
import weka.core.Capabilities;
import weka.core.Drawable;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.Matchable;
import weka.core.Option;
import weka.core.OptionHandler;
import weka.core.PartitionGenerator;
import weka.core.RevisionUtils;
import weka.core.Summarizable;
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.Capabilities;
import weka.core.Capabilities.Capability;

public class J48 extends AbstractClassifier implements OptionHandler, Drawable,
  Matchable, Sourcable, WeightedInstancesHandler, Summarizable,
  AdditionalMeasureProducer, TechnicalInformationHandler, PartitionGenerator {
```

```
static final long serialVersionUID = -217733168393644444L;
protected ClassifierTree m_root;
protected boolean m_unpruned = false;
protected boolean m_collapseTree = true;
protected float m_CF = 0.25f;
protected int m_minNumObj = 2;
protected boolean m_useMDLcorrection = true;
protected boolean m_useLaplace = false;
protected boolean m_reducedErrorPruning = false;
protected int m_numFolds = 3;
protected boolean m_binarySplits = false;
protected boolean m_subtreeRaising = true;

protected boolean m_noCleanup = false;
protected int m_Seed = 1;
protected boolean m_doNotMakeSplitPointActualValue;

public String globalInfo() {
  return "Class for generating a pruned or unpruned C4.5 decision tree. For more "
    + "information, see\n\n" + getTechnicalInformation().toString();
}

@Override
public TechnicalInformation getTechnicalInformation() {
  TechnicalInformation result;

  result = new TechnicalInformation(Type.BOOK);
  result.setValue(Field.AUTHOR, "Ross Quinlan");
  result.setValue(Field.YEAR, "1993");
  result.setValue(Field.TITLE, "C4.5: Programs for Machine Learning");
  result.setValue(Field.PUBLISHER, "Morgan Kaufmann Publishers");
  result.setValue(Field.ADDRESS, "San Mateo, CA");

  return result;
}


@Override
public Capabilities getCapabilities() {
  Capabilities result;

  result = new Capabilities(this);
  result.disableAll();
  // attributes
  result.enable(Capability.NOMINAL_ATTRIBUTES);
  result.enable(Capability.NUMERIC_ATTRIBUTES);
  result.enable(Capability.DATE_ATTRIBUTES);
  result.enable(Capability.MISSING_VALUES);

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

  result.setMinimumNumberInstances(0);

  return result;
}

@Override
public void buildClassifier(Instances instances) throws Exception {

  getCapabilities().testWithFail(instances);

  ModelSelection modSelection;

  if (m_binarySplits) {
    modSelection = new BinC45ModelSelection(m_minNumObj, instances,
      m_useMDLcorrection, m_doNotMakeSplitPointActualValue);
  } else {
    modSelection = new C45ModelSelection(m_minNumObj, instances,
      m_useMDLcorrection, m_doNotMakeSplitPointActualValue);
  }
  if (!m_reducedErrorPruning) {
    m_root = new C45PruneableClassifierTree(modSelection, !m_unpruned, m_CF,
      m_subtreeRaising, !m_noCleanup, m_collapseTree);
  } else {
    m_root = new PruneableClassifierTree(modSelection, !m_unpruned,
```

```
      m_numFolds, !m_noCleanup, m_Seed);
  }
  m_root.buildClassifier(instances);
  if (m_binarySplits) {
    ((BinC45ModelSelection) modSelection).cleanup();
  } else {
    ((C45ModelSelection) modSelection).cleanup();
  }
}


@Override
public double classifyInstance(Instance instance) throws Exception {
  return m_root.classifyInstance(instance);
}


@Override
public final double[] distributionForInstance(Instance instance)
  throws Exception {

  return m_root.distributionForInstance(instance, m_useLaplace);
}

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

@Override
public String graph() throws Exception {

  return m_root.graph();
}


@Override
public String prefix() throws Exception {

  return m_root.prefix();
}


@Override
public String toSource(String className) throws Exception {

  StringBuffer[] source = m_root.toSource(className);
  return "class " + className + " {\n\n"
    + "  public static double classify(Object[] i)\n"
    + "    throws Exception {\n\n" + "    double p = Double.NaN;\n"
    + source[0] // Assignment code
    + "    return p;\n" + "  }\n" + source[1] // Support code
    + "}\n";
}


@Override
public Enumeration<Option> listOptions() {

  Vector<Option> newVector = new Vector<Option>(13);

  newVector.addElement(new Option("\tUse unpruned tree.", "U", 0, "-U"));
  newVector.addElement(new Option("\tDo not collapse tree.", "O", 0, "-O"));
  newVector.addElement(new Option("\tSet confidence threshold for pruning.\n"
    + "\t(default 0.25)", "C", 1, "-C <pruning confidence>"));
  newVector.addElement(new Option(
    "\tSet minimum number of instances per leaf.\n" + "\t(default 2)", "M",
    1, "-M <minimum number of instances>"));
  newVector.addElement(new Option("\tUse reduced error pruning.", "R", 0,
    "-R"));
  newVector.addElement(new Option("\tSet number of folds for reduced error\n"
    + "\tpruning. One fold is used as pruning set.\n" + "\t(default 3)", "N",
    1, "-N <number of folds>"));
  newVector.addElement(new Option("\tUse binary splits only.", "B", 0, "-B"));
  newVector.addElement(new Option("\tDo not perform subtree raising.", "S", 0,
    "-S"));
  newVector.addElement(new Option(
```

```
      "\tDo not clean up after the tree has been built.", "L", 0, "-L"));
    newVector.addElement(new Option(
      "\tLaplace smoothing for predicted probabilities.", "A", 0, "-A"));
    newVector.addElement(new Option(
      "\tDo not use MDL correction for info gain on numeric attributes.", "J",
      0, "-J"));
    newVector.addElement(new Option(
      "\tSeed for random data shuffling (default 1).", "Q", 1, "-Q <seed>"));
    newVector.addElement(new Option("\tDo not make split point actual value.",
      "-doNotMakeSplitPointActualValue", 0, "-doNotMakeSplitPointActualValue"));

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

    return newVector.elements();
  }

  @Override
  public void setOptions(String[] options) throws Exception {

    // Other options
    String minNumString = Utils.getOption('M', options);
    if (minNumString.length() != 0) {
      m_minNumObj = Integer.parseInt(minNumString);
    } else {
      m_minNumObj = 2;
    }
    m_binarySplits = Utils.getFlag('B', options);
    m_useLaplace = Utils.getFlag('A', options);
    m_useMDLcorrection = !Utils.getFlag('J', options);

    // Pruning options
    m_unpruned = Utils.getFlag('U', options);
    m_collapseTree = !Utils.getFlag('O', options);
    m_subtreeRaising = !Utils.getFlag('S', options);
    m_noCleanup = Utils.getFlag('L', options);
    m_doNotMakeSplitPointActualValue = Utils.getFlag(
      "doNotMakeSplitPointActualValue", options);
    if ((m_unpruned) && (!m_subtreeRaising)) {
      throw new Exception(
        "Subtree raising doesn't need to be unset for unpruned tree!");
    }
    m_reducedErrorPruning = Utils.getFlag('R', options);
    if ((m_unpruned) && (m_reducedErrorPruning)) {
      throw new Exception(
        "Unpruned tree and reduced error pruning can't be selected "
          + "simultaneously!");
    }
    String confidenceString = Utils.getOption('C', options);
    if (confidenceString.length() != 0) {
      if (m_reducedErrorPruning) {
        throw new Exception("Setting the confidence doesn't make sense "
          + "for reduced error pruning.");
      } else if (m_unpruned) {
        throw new Exception(
          "Doesn't make sense to change confidence for unpruned " + "tree!");
      } else {
        m_CF = (new Float(confidenceString)).floatValue();
        if ((m_CF <= 0) || (m_CF >= 1)) {
          throw new Exception(
            "Confidence has to be greater than zero and smaller " + "than one!");
        }
      }
    } else {
      m_CF = 0.25f;
    }
    String numFoldsString = Utils.getOption('N', options);
    if (numFoldsString.length() != 0) {
      if (!m_reducedErrorPruning) {
        throw new Exception("Setting the number of folds"
          + " doesn't make sense if"
          + " reduced error pruning is not selected.");
      } else {
        m_numFolds = Integer.parseInt(numFoldsString);
      }
    } else {
      m_numFolds = 3;
    }
```

```
    String seedString = Utils.getOption('Q', options);
    if (seedString.length() != 0) {
      m_Seed = Integer.parseInt(seedString);
    } else {
      m_Seed = 1;
    }

    super.setOptions(options);

    Utils.checkForRemainingOptions(options);
  }

  @Override
  public String[] getOptions() {

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

    if (m_noCleanup) {
        options.add("-L");
    }
    if (!m_collapseTree) {
        options.add("-O");
    }
    if (m_unpruned) {
        options.add("-U");
    } else {
      if (!m_subtreeRaising) {
          options.add("-S");
      }
      if (m_reducedErrorPruning) {
          options.add("-R");
          options.add("-N");
          options.add("" + m_numFolds);
          options.add("-Q");
          options.add("" + m_Seed);
      } else {
          options.add("-C");
          options.add("" + m_CF);
      }
    }
    if (m_binarySplits) {
        options.add("-B");
    }
    options.add("-M");
    options.add("" + m_minNumObj);
    if (m_useLaplace) {
        options.add("-A");
    }
    if (!m_useMDLcorrection) {
        options.add("-J");
    }
    if (m_doNotMakeSplitPointActualValue) {
        options.add("-doNotMakeSplitPointActualValue");
    }

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

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


  public String seedTipText() {
    return "The seed used for randomizing the data "
      + "when reduced-error pruning is used.";
  }


  public int getSeed() {
    return m_Seed;
  }

  public void setSeed(int newSeed) {
    m_Seed = newSeed;
  }

  public String useLaplaceTipText() {
    return "Whether counts at leaves are smoothed based on Laplace.";
```

```
}

public boolean getUseLaplace() {
  return m_useLaplace;
}

public void setUseLaplace(boolean newuseLaplace) {
  m_useLaplace = newuseLaplace;
}

public String useMDLcorrectionTipText() {
  return "Whether MDL correction is used when finding splits on numeric attributes.";
}

public boolean getUseMDLcorrection() {
  return m_useMDLcorrection;
}

public void setUseMDLcorrection(boolean newuseMDLcorrection) {
    m_useMDLcorrection = newuseMDLcorrection;
}

@Override
public String toString() {
  if (m_root == null) {
    return "No classifier built";
  }
  if (m_unpruned) {
    return "J48 unpruned tree\n------------------\n" + m_root.toString();
  } else {
    return "J48 pruned tree\n------------------\n" + m_root.toString();
  }
}

@Override
public String toSummaryString() {
    return "Number of leaves: " + m_root.numLeaves() + "\n"
    + "Size of the tree: " + m_root.numNodes() + "\n";
}

public double measureTreeSize() {
  return m_root.numNodes();
}

public double measureNumLeaves() {
  return m_root.numLeaves();
}

public double measureNumRules() {
  return m_root.numLeaves();
}

@Override
public Enumeration<String> enumerateMeasures() {
  Vector<String> newVector = new Vector<String>(3);
  newVector.addElement("measureTreeSize");
  newVector.addElement("measureNumLeaves");
  newVector.addElement("measureNumRules");
  return newVector.elements();
}

@Override
public double getMeasure(String additionalMeasureName) {
  if (additionalMeasureName.compareToIgnoreCase("measureNumRules") == 0) {
    return measureNumRules();
  } else if (additionalMeasureName.compareToIgnoreCase("measureTreeSize") == 0) {
    return measureTreeSize();
  } else if (additionalMeasureName.compareToIgnoreCase("measureNumLeaves") == 0) {
    return measureNumLeaves();
  } else {
    throw new IllegalArgumentException(additionalMeasureName
      + " not supported (j48)");
  }
}

public String unprunedTipText() {
  return "Whether pruning is performed.";
```

```
}

public boolean getUnpruned() {
  return m_unpruned;
}

public void setUnpruned(boolean v) {
  if (v) {
    m_reducedErrorPruning = false;
  }
  m_unpruned = v;
}

public String collapseTreeTipText() {
  return "Whether parts are removed that do not reduce training error.";
}

public boolean getCollapseTree() {
  return m_collapseTree;
}

public void setCollapseTree(boolean v) {
  m_collapseTree = v;
}

public String confidenceFactorTipText() {
  return "The confidence factor used for pruning (smaller values incur "
    + "more pruning).";
}

public float getConfidenceFactor() {
  return m_CF;
}

public void setConfidenceFactor(float v) {
  m_CF = v;
}

public String minNumObjTipText() {
  return "The minimum number of instances per leaf.";
}

public int getMinNumObj() {
 return m_minNumObj;
}

public void setMinNumObj(int v) {
  m_minNumObj = v;
}

public String reducedErrorPruningTipText() {
  return "Whether reduced-error pruning is used instead of C.4.5 pruning.";
}

public boolean getReducedErrorPruning() {
  return m_reducedErrorPruning;
}

public void setReducedErrorPruning(boolean v) {
  if (v) {
    m_unpruned = false;
  }
  m_reducedErrorPruning = v;
}

public String numFoldsTipText() {
  return "Determines the amount of data used for reduced-error pruning. "
    + " One fold is used for pruning, the rest for growing the tree.";
}

public int getNumFolds() {
  return m_numFolds;
}

public void setNumFolds(int v) {
  m_numFolds = v;
}
```

```
public String binarySplitsTipText() {
  return "Whether to use binary splits on nominal attributes when "
    + "building the trees.";
}

public boolean getBinarySplits() {
  return m_binarySplits;
}

public void setBinarySplits(boolean v) {
  m_binarySplits = v;
}

public String subtreeRaisingTipText() {
  return "Whether to consider the subtree raising operation when pruning.";
}

public boolean getSubtreeRaising() {
  return m_subtreeRaising;
}

public void setSubtreeRaising(boolean v) {
  m_subtreeRaising = v;
}

public String saveInstanceDataTipText() {
  return "Whether to save the training data for visualization.";
}

public boolean getSaveInstanceData() {

  return m_noCleanup;
}

public void setSaveInstanceData(boolean v) {

  m_noCleanup = v;
}

public String doNotMakeSplitPointActualValueTipText() {
  return "If true, the split point is not relocated to an actual data value."
    + " This can yield substantial speed-ups for large datasets with numeric attributes.";
}

public boolean getDoNotMakeSplitPointActualValue() {
  return m_doNotMakeSplitPointActualValue;
}

public void setDoNotMakeSplitPointActualValue(
  boolean m_doNotMakeSplitPointActualValue) {
  this.m_doNotMakeSplitPointActualValue = m_doNotMakeSplitPointActualValue;
}

@Override
public String getRevision() {
  return RevisionUtils.extract("$Revision: 14534 $");
}

@Override
public void generatePartition(Instances data) throws Exception {
  buildClassifier(data);
}

@Override
public double[] getMembershipValues(Instance inst) throws Exception {

  return m_root.getMembershipValues(inst);
}

@Override
public int numElements() throws Exception {
  return m_root.numNodes();
}

public static void main(String[] argv) {
  runClassifier(new J48(), argv);
```

```
  }
}
```

Sourcable.java

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

package weka.classifiers;

public interface Sourcable {
  String toSource(String className) throws Exception;
}
```

BinC45ModelSelection.java

```
/*
 *   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/>.
 *    BinC45ModelSelection.java
 *    Copyright (C) 1999-2012 University of Waikato, Hamilton, New Zealand
*/

package weka.classifiers.trees.j48;

import java.util.Enumeration;

import weka.core.Attribute;
import weka.core.Instances;
import weka.core.RevisionUtils;
import weka.core.Utils;

public class BinC45ModelSelection extends ModelSelection {

  private static final long serialVersionUID = 179170923545122001L;

  protected final int m_minNoObj;

  protected final boolean m_useMDLcorrection;

  protected Instances m_allData;

  protected final boolean m_doNotMakeSplitPointActualValue;

  public BinC45ModelSelection(int minNoObj, Instances allData,
    boolean useMDLcorrection, boolean doNotMakeSplitPointActualValue) {
    m_minNoObj = minNoObj;
    m_allData = allData;
    m_useMDLcorrection = useMDLcorrection;
```

```
    m_doNotMakeSplitPointActualValue = doNotMakeSplitPointActualValue;
  }

  public void cleanup() {
    m_allData = null;
  }

  @Override
  public final ClassifierSplitModel selectModel(Instances data) {

    double minResult;
    BinC45Split[] currentModel;
    BinC45Split bestModel = null;
    NoSplit noSplitModel = null;
    double averageInfoGain = 0;
    int validModels = 0;
    boolean multiVal = true;
    Distribution checkDistribution;
    double sumOfWeights;
    int i;

    try {

      checkDistribution = new Distribution(data);
      noSplitModel = new NoSplit(checkDistribution);
      if (Utils.sm(checkDistribution.total(), 2 * m_minNoObj)
        || Utils.eq(checkDistribution.total(),
          checkDistribution.perClass(checkDistribution.maxClass()))) {
        return noSplitModel;
      }

      Enumeration<Attribute> enu = data.enumerateAttributes();
      while (enu.hasMoreElements()) {
        Attribute attribute = enu.nextElement();
        if ((attribute.isNumeric())
          || (Utils.sm(attribute.numValues(), (0.3 * m_allData.numInstances())))) {
          multiVal = false;
          break;
        }
      }
      currentModel = new BinC45Split[data.numAttributes()];
      sumOfWeights = data.sumOfWeights();

      for (i = 0; i < data.numAttributes(); i++) {
        if (i != (data).classIndex()) {
          currentModel[i] = new BinC45Split(i, m_minNoObj, sumOfWeights,
            m_useMDLcorrection);
          currentModel[i].buildClassifier(data);

          if (currentModel[i].checkModel()) {
            if ((data.attribute(i).isNumeric())
              || (multiVal || Utils.sm(data.attribute(i).numValues(),
                (0.3 * m_allData.numInstances())))) {
              averageInfoGain = averageInfoGain + currentModel[i].infoGain();
              validModels++;
            }
          }
        } else {
          currentModel[i] = null;
        }
      }

      if (validModels == 0) {
        return noSplitModel;
      }
      averageInfoGain = averageInfoGain / validModels;

      minResult = 0;
      for (i = 0; i < data.numAttributes(); i++) {
        if ((i != (data).classIndex()) && (currentModel[i].checkModel())) {
          // Use 1E-3 here to get a closer approximation to the original
          if ((currentModel[i].infoGain() >= (averageInfoGain - 1E-3))
            && Utils.gr(currentModel[i].gainRatio(), minResult)) {
            bestModel = currentModel[i];
            minResult = currentModel[i].gainRatio();
          }
        }
```

```
      }

      if (Utils.eq(minResult, 0)) {
        return noSplitModel;
      }

      bestModel.distribution().addInstWithUnknown(data, bestModel.attIndex());

      if (!m_doNotMakeSplitPointActualValue) {
        bestModel.setSplitPoint(m_allData);
      }
      return bestModel;
    } catch (Exception e) {
      e.printStackTrace();
    }
    return null;
  }

  @Override
  public final ClassifierSplitModel selectModel(Instances train, Instances test) {

    return selectModel(train);
  }

  @Override
  public String getRevision() {
    return RevisionUtils.extract("$Revision: 10531 $");
  }
}
```

C45ModelSelection.java

```
/*
 *   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/>.
 *    C45ModelSelection.java
 *    Copyright (C) 1999-2012 University of Waikato, Hamilton, New Zealand
 */

package weka.classifiers.trees.j48;

import java.util.Enumeration;

import weka.core.Attribute;
import weka.core.Instances;
import weka.core.RevisionUtils;
import weka.core.Utils;

public class C45ModelSelection extends ModelSelection {

  private static final long serialVersionUID = 3372204862440821989L;

  protected final int m_minNoObj;

  protected final boolean m_useMDLcorrection;

  protected Instances m_allData; //

  protected final boolean m_doNotMakeSplitPointActualValue;

  public C45ModelSelection(int minNoObj, Instances allData,
    boolean useMDLcorrection, boolean doNotMakeSplitPointActualValue) {
    m_minNoObj = minNoObj;
    m_allData = allData;
    m_useMDLcorrection = useMDLcorrection;
    m_doNotMakeSplitPointActualValue = doNotMakeSplitPointActualValue;
```

```
  }

public void cleanup() {
    m_allData = null;
  }

  @Override
  public final ClassifierSplitModel selectModel(Instances data) {

    double minResult;
    C45Split[] currentModel;
    C45Split bestModel = null;
    NoSplit noSplitModel = null;
    double averageInfoGain = 0;
    int validModels = 0;
    boolean multiVal = true;
    Distribution checkDistribution;
    Attribute attribute;
    double sumOfWeights;
    int i;

    try {
      checkDistribution = new Distribution(data);
      noSplitModel = new NoSplit(checkDistribution);
      if (Utils.sm(checkDistribution.total(), 2 * m_minNoObj)
        || Utils.eq(checkDistribution.total(),
          checkDistribution.perClass(checkDistribution.maxClass()))) {
        return noSplitModel;
      }

      if (m_allData != null) {
        Enumeration<Attribute> enu = data.enumerateAttributes();
        while (enu.hasMoreElements()) {
          attribute = enu.nextElement();
          if ((attribute.isNumeric())
            || (Utils.sm(attribute.numValues(),
              (0.3 * m_allData.numInstances())))) {
            multiVal = false;
            break;
          }
        }
      }

      currentModel = new C45Split[data.numAttributes()];
      sumOfWeights = data.sumOfWeights();


       for (i = 0; i < data.numAttributes(); i++) {
             if (i != (data).classIndex()) {

          currentModel[i] = new C45Split(i, m_minNoObj, sumOfWeights,
            m_useMDLcorrection);
          currentModel[i].buildClassifier(data);
          if (currentModel[i].checkModel()) {
            if (m_allData != null) {
              if ((data.attribute(i).isNumeric())
                || (multiVal || Utils.sm(data.attribute(i).numValues(),
                  (0.3 * m_allData.numInstances())))) {
                averageInfoGain = averageInfoGain + currentModel[i].infoGain();
                validModels++;
              }
            } else {
              averageInfoGain = averageInfoGain + currentModel[i].infoGain();
              validModels++;
            }
          } else {
            currentModel[i] = null;
          }
      }

      if (validModels == 0) {
        return noSplitModel;
      }
      averageInfoGain = averageInfoGain / validModels;

      minResult = 0;
```

```
      for (i = 0; i < data.numAttributes(); i++) {
        if ((i != (data).classIndex()) && (currentModel[i].checkModel())) {
          if ((currentModel[i].infoGain() >= (averageInfoGain - 1E-3))
            && Utils.gr(currentModel[i].gainRatio(), minResult)) {
            bestModel = currentModel[i];
            minResult = currentModel[i].gainRatio();
          }
        }
      }

      if (Utils.eq(minResult, 0)) {
        return noSplitModel;
      }

      bestModel.distribution().addInstWithUnknown(data, bestModel.attIndex());

      if ((m_allData != null) && (!m_doNotMakeSplitPointActualValue)) {
        bestModel.setSplitPoint(m_allData);
      }
      return bestModel;
    } catch (Exception e) {
      e.printStackTrace();
    }
    return null;
  }

  @Override
  public final ClassifierSplitModel selectModel(Instances train, Instances test) {
    return selectModel(train);
  }

  @Override
  public String getRevision() {
    return RevisionUtils.extract("$Revision: 10531 $");
  }
}
```

C45PruneableClassifierTree.java

```
/*
 *   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/>.
 */

/*
 *    C45PruneableClassifierTree.java
 *    Copyright (C) 1999-2012 University of Waikato, Hamilton, New Zealand
 *
 */

package weka.classifiers.trees.j48;

import weka.core.Capabilities;
import weka.core.Capabilities.Capability;
import weka.core.Instances;
import weka.core.RevisionUtils;
import weka.core.Utils;

public class C45PruneableClassifierTree
  extends ClassifierTree {

  static final long serialVersionUID = -4813820170260388194L;

  protected boolean m_pruneTheTree = false;

  protected boolean m_collapseTheTree = false;
```

```
protected float m_CF = 0.25f;

protected boolean m_subtreeRaising = true;

protected boolean m_cleanup = true;

public C45PruneableClassifierTree(ModelSelection toSelectLocModel,
                              boolean pruneTree,float cf,
                              boolean raiseTree,
                              boolean cleanup,
                                    boolean collapseTree)
     throws Exception {

  super(toSelectLocModel);

  m_pruneTheTree = pruneTree;
  m_CF = cf;
  m_subtreeRaising = raiseTree;
  m_cleanup = cleanup;
  m_collapseTheTree = collapseTree;
}

public void buildClassifier(Instances data) throws Exception {

  data = new Instances(data);
  data.deleteWithMissingClass();

 buildTree(data, m_subtreeRaising || !m_cleanup);
 if (m_collapseTheTree) {
   collapse();
 }
 if (m_pruneTheTree) {
   prune();
 }
 if (m_cleanup) {
   cleanup(new Instances(data, 0));
 }
}

public final void collapse(){

  double errorsOfSubtree;
  double errorsOfTree;
  int i;

  if (!m_isLeaf){
    errorsOfSubtree = getTrainingErrors();
    errorsOfTree = localModel().distribution().numIncorrect();
    if (errorsOfSubtree >= errorsOfTree-1E-3){

     m_sons = null;
     m_isLeaf = true;

     m_localModel = new NoSplit(localModel().distribution());
    }else
     for (i=0;i<m_sons.length;i++)
       son(i).collapse();
  }
}

public void prune() throws Exception {

  double errorsLargestBranch;
  double errorsLeaf;
  double errorsTree;
  int indexOfLargestBranch;
  C45PruneableClassifierTree largestBranch;
  int i;

  if (!m_isLeaf){


  for (i=0;i<m_sons.length;i++)
     son(i).prune();
    indexOfLargestBranch = localModel().distribution().maxBag();
    if (m_subtreeRaising) {
```

```
      errorsLargestBranch = son(indexOfLargestBranch).
        getEstimatedErrorsForBranch((Instances)m_train);
      } else {
      errorsLargestBranch = Double.MAX_VALUE;
      }

      errorsLeaf =
      getEstimatedErrorsForDistribution(localModel().distribution());

      errorsTree = getEstimatedErrors();

      if (Utils.smOrEq(errorsLeaf,errorsTree+0.1) &&
        Utils.smOrEq(errorsLeaf,errorsLargestBranch+0.1)){

      m_sons = null;
      m_isLeaf = true;

      m_localModel = new NoSplit(localModel().distribution());
      return;
      }

      if (Utils.smOrEq(errorsLargestBranch,errorsTree+0.1)){
      largestBranch = son(indexOfLargestBranch);
      m_sons = largestBranch.m_sons;
      m_localModel = largestBranch.localModel();
      m_isLeaf = largestBranch.m_isLeaf;
      newDistribution(m_train);
      prune();
      }
    }
  }

  protected ClassifierTree getNewTree(Instances data) throws Exception {

    C45PruneableClassifierTree newTree =
      new C45PruneableClassifierTree(m_toSelectModel, m_pruneTheTree, m_CF,
                                     m_subtreeRaising, m_cleanup, m_collapseTheTree);
    newTree.buildTree((Instances)data, m_subtreeRaising || !m_cleanup);

    return newTree;
  }

  private double getEstimatedErrors(){

    double errors = 0;
    int i;

    if (m_isLeaf)
      return getEstimatedErrorsForDistribution(localModel().distribution());
    else{
      for (i=0;i<m_sons.length;i++)
      errors = errors+son(i).getEstimatedErrors();
      return errors;
    }
  }

  private double getEstimatedErrorsForBranch(Instances data)
       throws Exception {

    Instances [] localInstances;
    double errors = 0;
    int i;

    if (m_isLeaf)
      return getEstimatedErrorsForDistribution(new Distribution(data));
    else{
      Distribution savedDist = localModel().m_distribution;
      localModel().resetDistribution(data);
      localInstances = (Instances[])localModel().split(data);
      localModel().m_distribution = savedDist;
      for (i=0;i<m_sons.length;i++)
      errors = errors+
        son(i).getEstimatedErrorsForBranch(localInstances[i]);
      return errors;
    }
  }
```

```
  private double getEstimatedErrorsForDistribution(Distribution
                                          theDistribution){

    if (Utils.eq(theDistribution.total(),0))
      return 0;
    else
      return theDistribution.numIncorrect()+
       Stats.addErrs(theDistribution.total(),
                   theDistribution.numIncorrect(),m_CF);
  }

  private double getTrainingErrors(){

    double errors = 0;
    int i;

    if (m_isLeaf)
      return localModel().distribution().numIncorrect();
    else{
      for (i=0;i<m_sons.length;i++)
       errors = errors+son(i).getTrainingErrors();
      return errors;
    }
  }

  private ClassifierSplitModel localModel(){

    return (ClassifierSplitModel)m_localModel;
  }

  private void newDistribution(Instances data) throws Exception {

    Instances [] localInstances;

    localModel().resetDistribution(data);
    m_train = data;
    if (!m_isLeaf){
      localInstances =
       (Instances [])localModel().split(data);
      for (int i = 0; i < m_sons.length; i++)
       son(i).newDistribution(localInstances[i]);
    } else {

      if (!Utils.eq(data.sumOfWeights(), 0)) {
       m_isEmpty = false;
      }
    }
  }

  private C45PruneableClassifierTree son(int index){

    return (C45PruneableClassifierTree)m_sons[index];
  }

  public String getRevision() {
    return RevisionUtils.extract("$Revision: 14534 $");
  }
}
```

ClassifierTree.java

```
/*
 *   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;

public class ClassifierTree implements Drawable, Serializable, RevisionHandler,
CapabilitiesHandler {

  static final long serialVersionUID = -8722249377542734193L;

  protected ModelSelection m_toSelectModel;

  protected ClassifierSplitModel m_localModel;

  protected ClassifierTree[] m_sons;

  protected boolean m_isLeaf;

  protected boolean m_isEmpty;

  protected Instances m_train;

  protected Distribution m_test;

  protected int m_id;

  private static long PRINTED_NODES = 0;

  public ClassifierSplitModel getLocalModel() {
    return m_localModel;
  }

  public ClassifierTree[] getSons() {
    return m_sons;
  }

  public boolean isLeaf() {
    return m_isLeaf;
  }

  public Instances getTrainingData() {
    return m_train;
  }

  protected static long nextID() {

    return PRINTED_NODES++;
  }

  protected static void resetID() {

    PRINTED_NODES = 0;
  }

  @Override
  public Capabilities getCapabilities() {
    Capabilities result = new Capabilities(this);
    result.enableAll();

    return result;
  }

  public ClassifierTree(ModelSelection toSelectLocModel) {
```

```
    m_toSelectModel = toSelectLocModel;
  }

  public void buildClassifier(Instances data) throws Exception {

    data = new Instances(data);
    data.deleteWithMissingClass();

    buildTree(data, false);
  }

  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;
    }
  }

  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;
    }
  }

  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;
}

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);
    }
  }
}

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;
}

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;
}

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

@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";
}

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();
}

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;
}
```

```
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;
}

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;
}

@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.";
  }
}

protected ClassifierTree getNewTree(Instances data) throws Exception {

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

  return newTree;
}

protected ClassifierTree getNewTree(Instances train, Instances test)
  throws Exception {

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

  return newTree;
}

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);
    }
  }
}

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);
    }
  }
}

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("]");
}

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);
      }
    }
  }
}

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);
      }
    }
  }
}

private ClassifierSplitModel localModel() {
  return m_localModel;
}

private ClassifierTree son(int index) {

  return m_sons[index];
}

public double[] getMembershipValues(Instance instance) throws Exception {

    double[] a = new double[numNodes()];

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

  while (!queueOfNodes.isEmpty()) {

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

    if (node.m_isLeaf) {
      continue;
    }

    int treeIndex = node.localModel().whichSubset(instance);

    double[] weights = new double[node.m_sons.length];

    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;
  }

  @Override
  public String getRevision() {
    return RevisionUtils.extract("$Revision: 14534 $");
  }
}
```

ModelSelection.java

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

package weka.classifiers.trees.j48;

import java.io.Serializable;

import weka.core.Instances;
import weka.core.RevisionHandler;

public abstract class ModelSelection
  implements Serializable, RevisionHandler {

  private static final long serialVersionUID = -4850147125096133642L;

  public abstract ClassifierSplitModel selectModel(Instances data) throws Exception;

  public ClassifierSplitModel selectModel(Instances train, Instances test)
       throws Exception {

    throw new Exception("Model selection method not implemented");
  }
}
```

PruneableClassifierTree.java

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

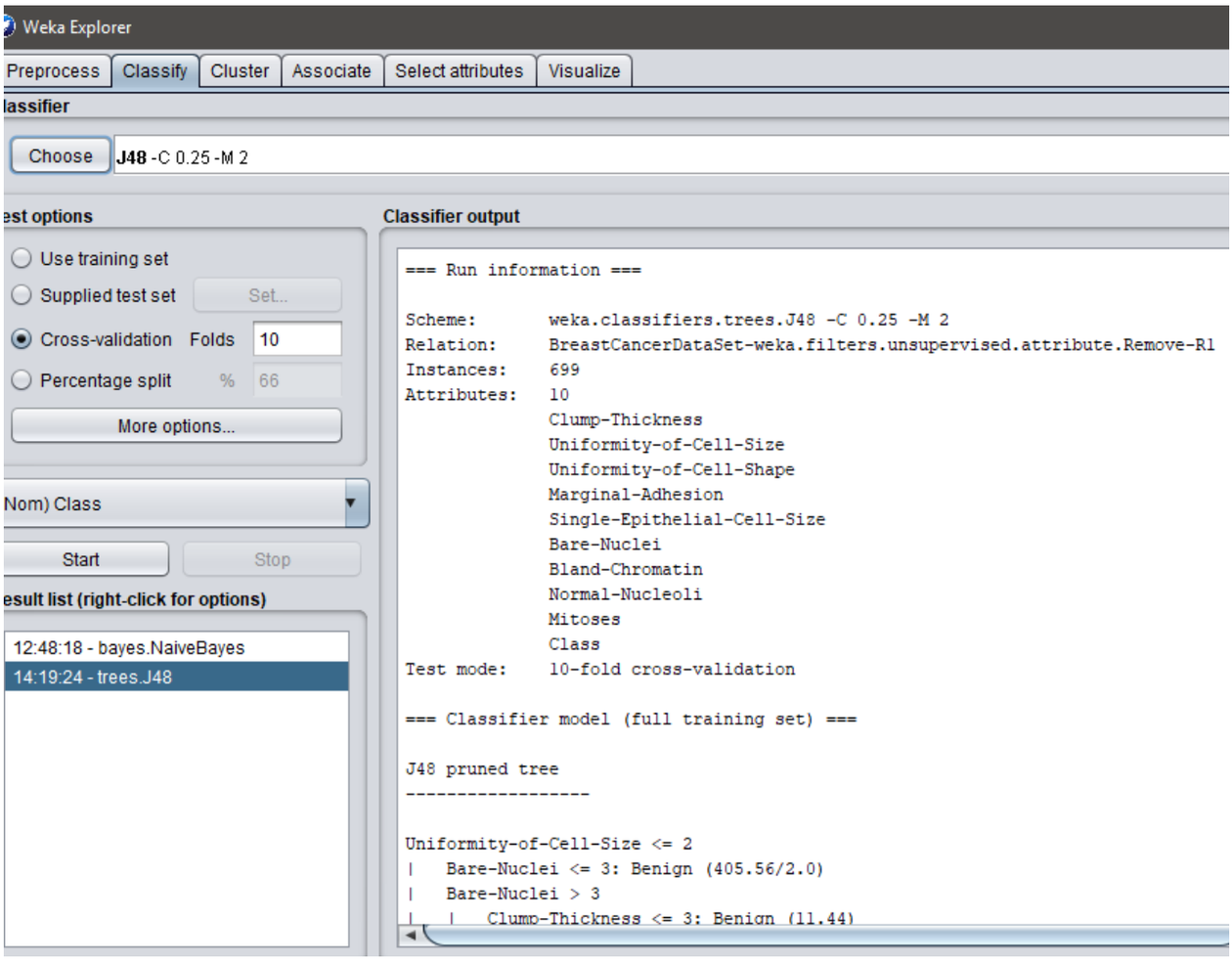
```
 *
 */

package weka.classifiers.trees.j48;

import java.util.Random;

import weka.core.Capabilities;
import weka.core.Capabilities.Capability;
import weka.core.Instances;
import weka.core.RevisionUtils;
import weka.core.Utils;

public class PruneableClassifierTree
  extends ClassifierTree {

  static final long serialVersionUID = -555775736857600201L;

  protected boolean pruneTheTree = false;

  protected int numSets = 3;

  protected boolean m_cleanup = true;

  protected int m_seed = 1;

  public PruneableClassifierTree(ModelSelection toSelectLocModel,
                           boolean pruneTree, int num, boolean cleanup,
                           int seed)
       throws Exception {

    super(toSelectLocModel);

    pruneTheTree = pruneTree;
    numSets = num;
    m_cleanup = cleanup;
    m_seed = seed;
  }

  public void buildClassifier(Instances data)
       throws Exception {

    data = new Instances(data);
    data.deleteWithMissingClass();

   Random random = new Random(m_seed);
   data.stratify(numSets);
   buildTree(data.trainCV(numSets, numSets - 1, random),
            data.testCV(numSets, numSets - 1), !m_cleanup);
   if (pruneTheTree) {
     prune();
   }
   if (m_cleanup) {
     cleanup(new Instances(data, 0));
   }
  }

  public void prune() throws Exception {

    if (!m_isLeaf) {


   for (int i = 0; i < m_sons.length; i++)
       son(i).prune();

    if (Utils.smOrEq(errorsForLeaf(),errorsForTree())) {

       m_sons = null;
       m_isLeaf = true;

       m_localModel = new NoSplit(localModel().distribution());
      }
    }
  }

  protected ClassifierTree getNewTree(Instances train, Instances test)
       throws Exception {
```

```
    PruneableClassifierTree newTree =
      new PruneableClassifierTree(m_toSelectModel, pruneTheTree, numSets, m_cleanup,m_seed);
    newTree.buildTree(train, test, !m_cleanup);
    return newTree;
  }

  private double errorsForTree() throws Exception {

    double errors = 0;

    if (m_isLeaf)
      return errorsForLeaf();
    else{
      for (int i = 0; i < m_sons.length; i++)
       if (Utils.eq(localModel().distribution().perBag(i), 0)) {
         errors += m_test.perBag(i)-
           m_test.perClassPerBag(i,localModel().distribution().
                         maxClass());
       } else
         errors += son(i).errorsForTree();

      return errors;
    }
  }

  private double errorsForLeaf() throws Exception {

    return m_test.total()-
      m_test.perClass(localModel().distribution().maxClass());
  }

  private ClassifierSplitModel localModel() {

    return (ClassifierSplitModel)m_localModel;
  }

  private PruneableClassifierTree son(int index) {

    return (PruneableClassifierTree)m_sons[index];
  }

  public String getRevision() {
    return RevisionUtils.extract("$Revision: 14534 $");
  }
}
```

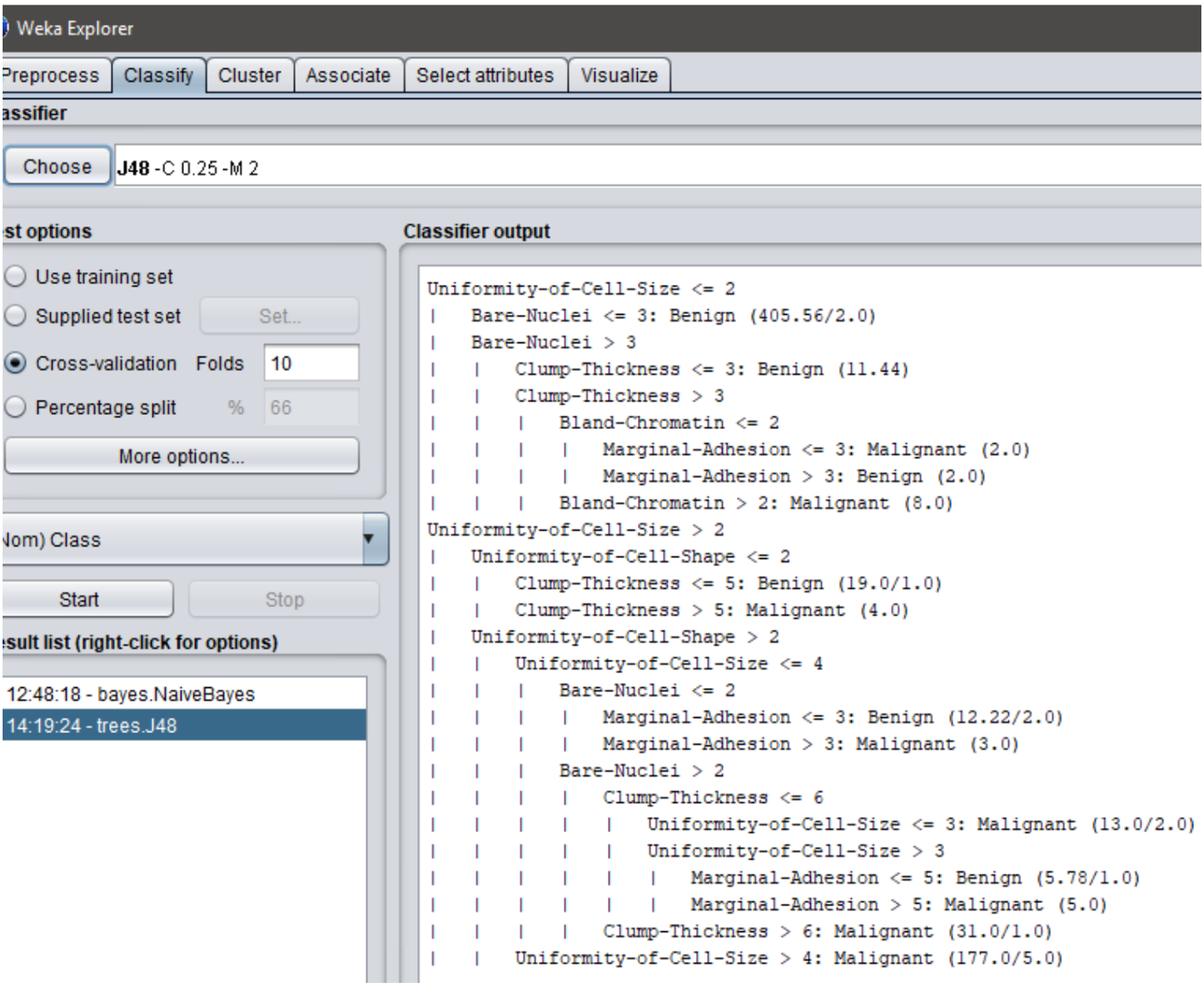
Data set - Breast Cancer Prediction
Link - http://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/

Tool Used - Weka
Link for Download - https://sourceforge.net/projects/weka/

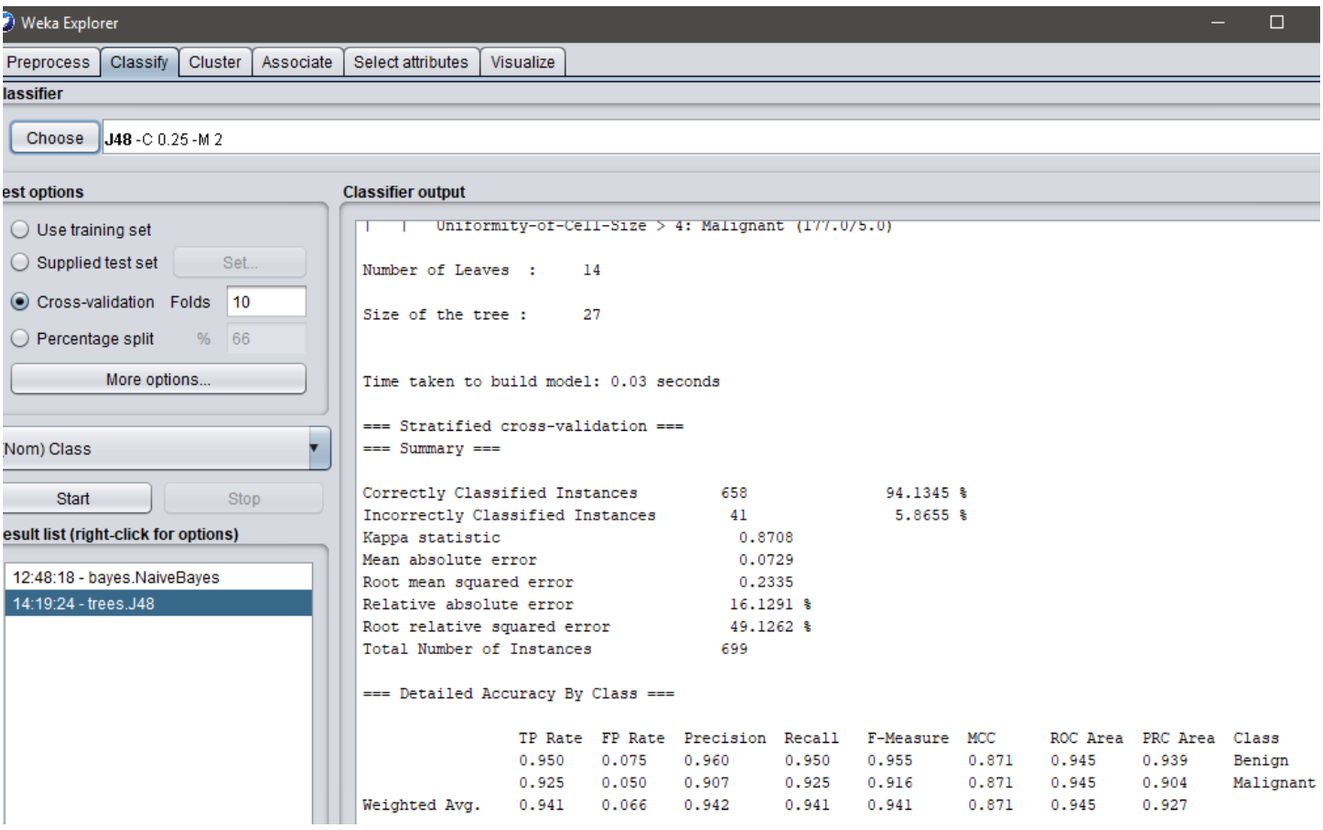
OUTPUT – J48 TEST MODE – 10-FOLD CROSS VALIDATION



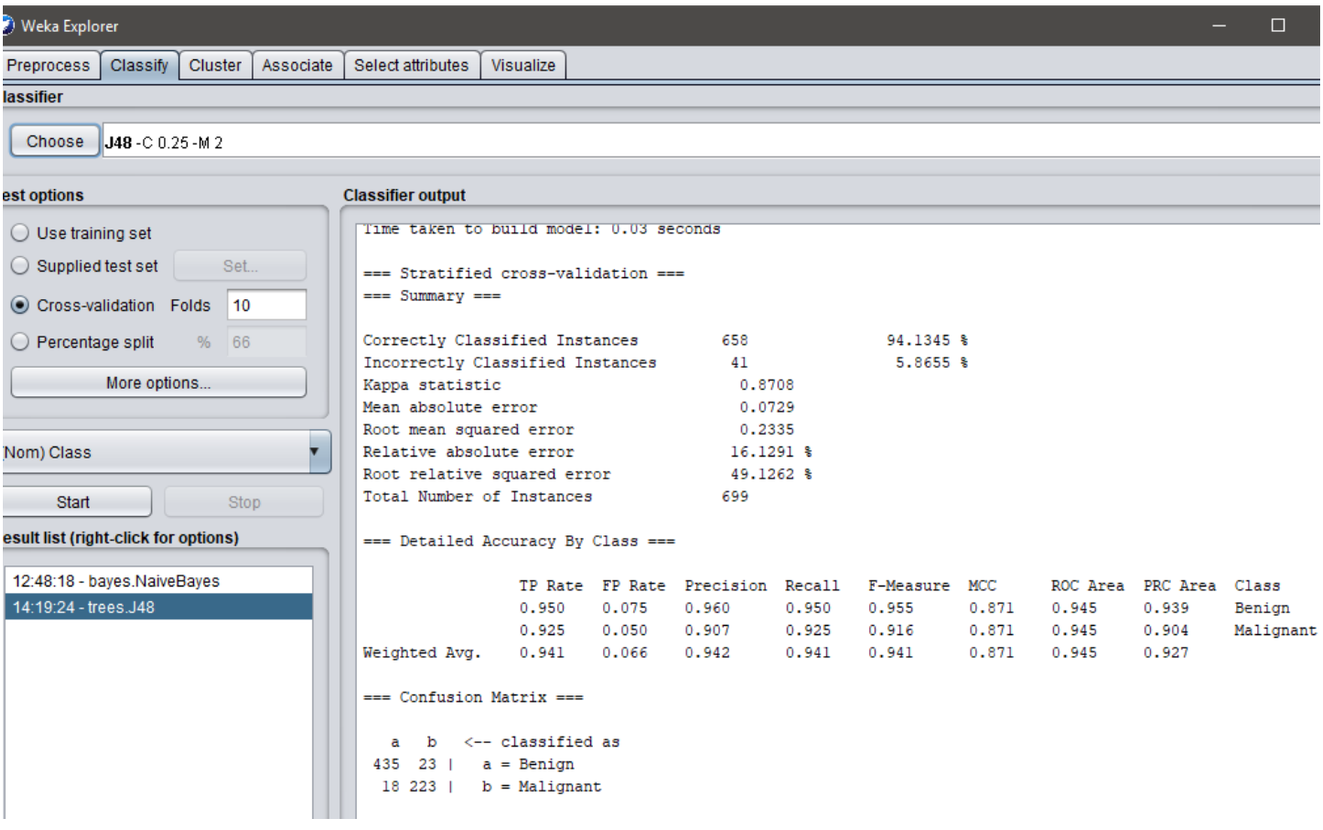
The above Figure displays all the attributes, no of Instances, Test mode which is used and Data Set.

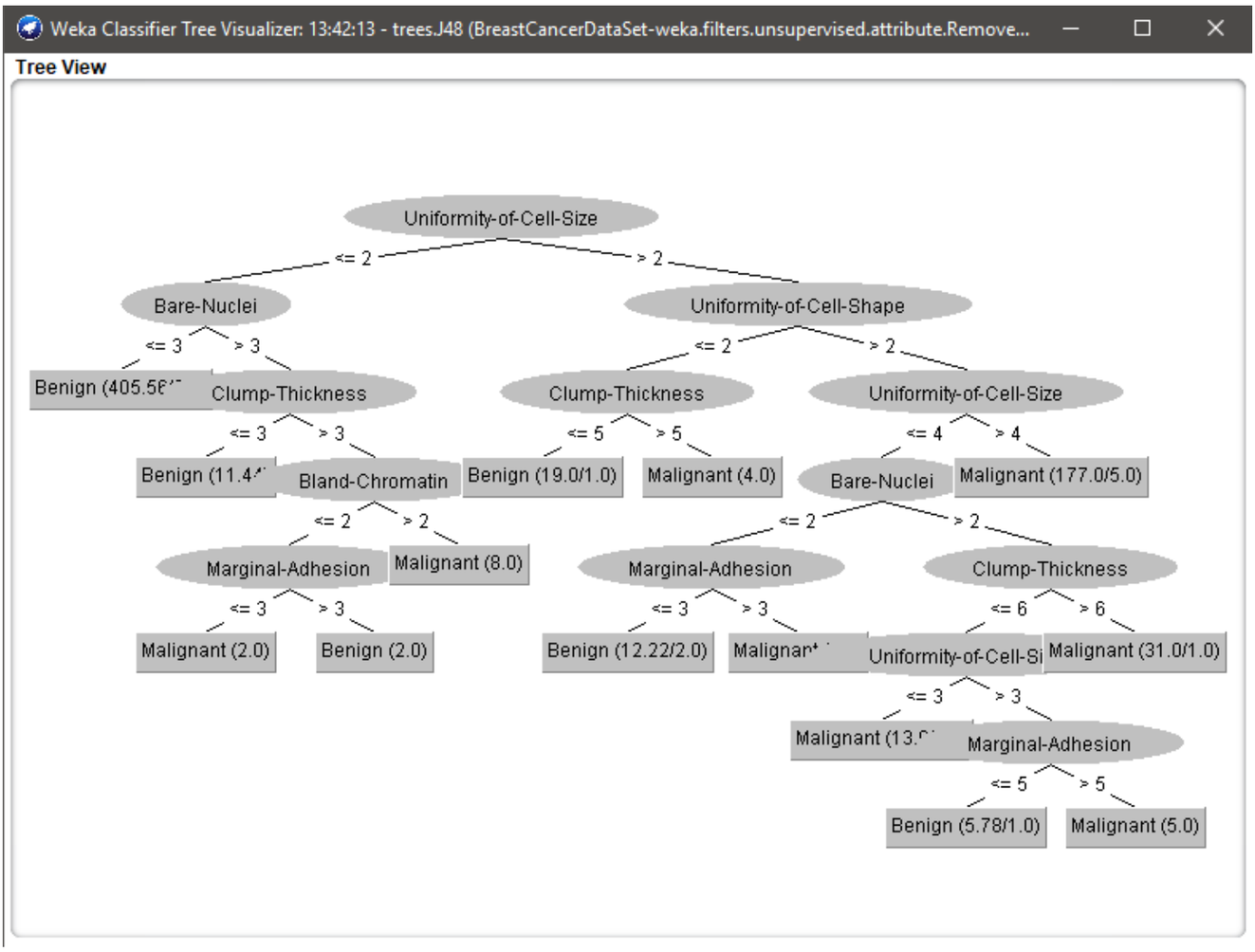


The Classifier Output shows the Tree Structure for the Dataset.



The above and below figures display Summary i.e. No of Correctly and incorrectly classified Instances, various statistical values like Mean absolute error, relative absolute error, corresponding percentage. Detailed accuracy by class i.e. for Benign and for Malignant. Also, the Confusion Matrix as per 2 classes present.



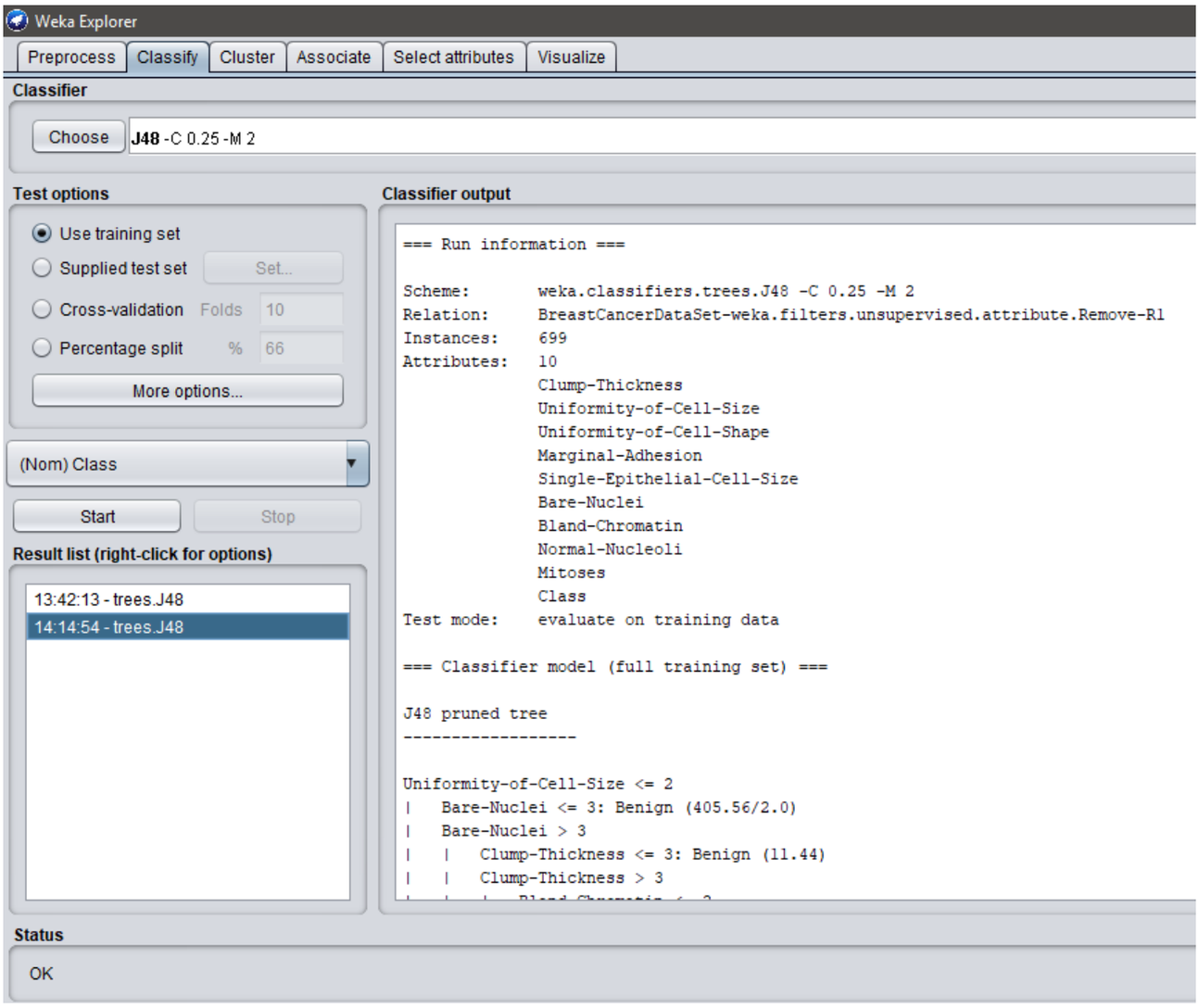


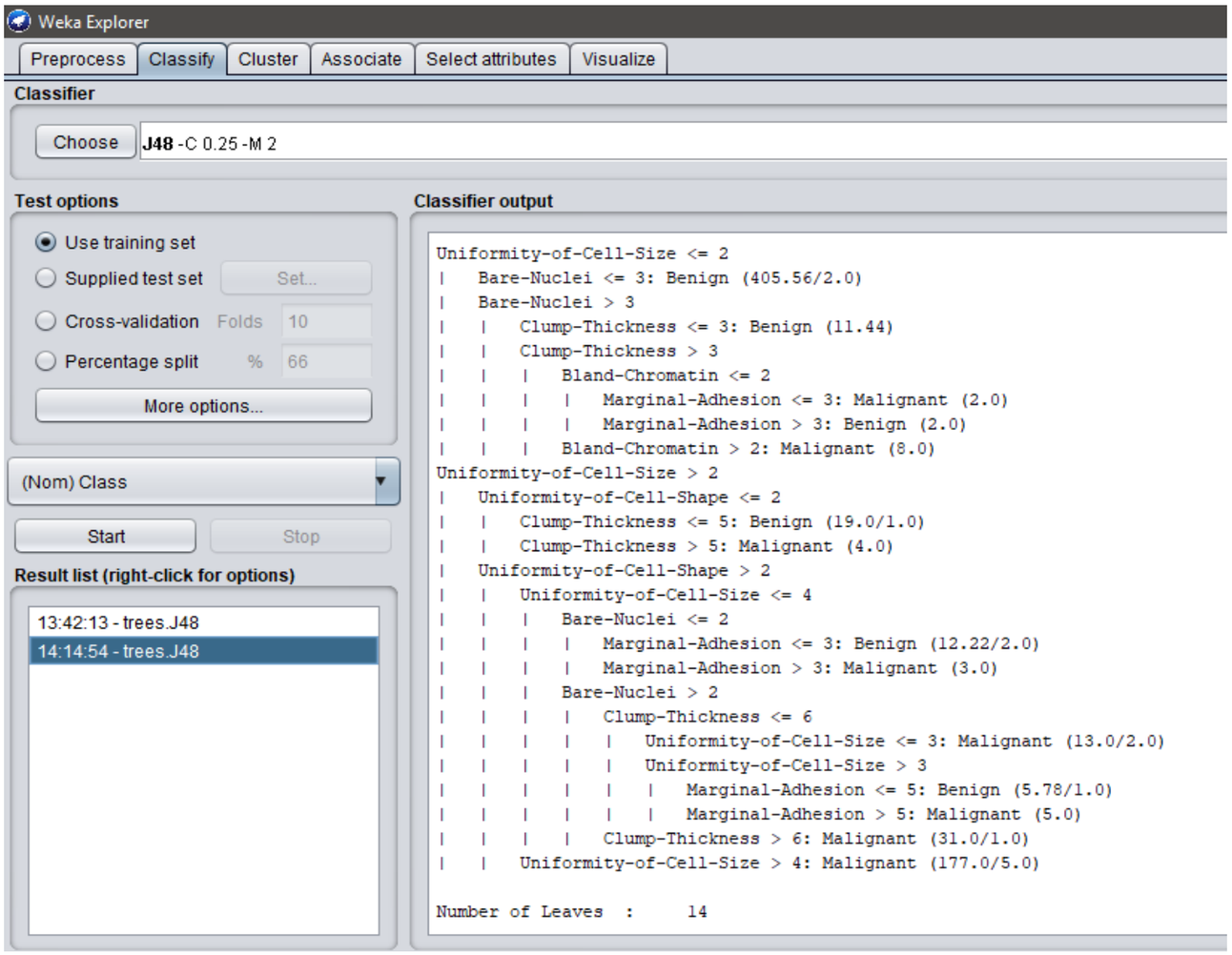
The above Figure shows Tree Representational for the Data Set.

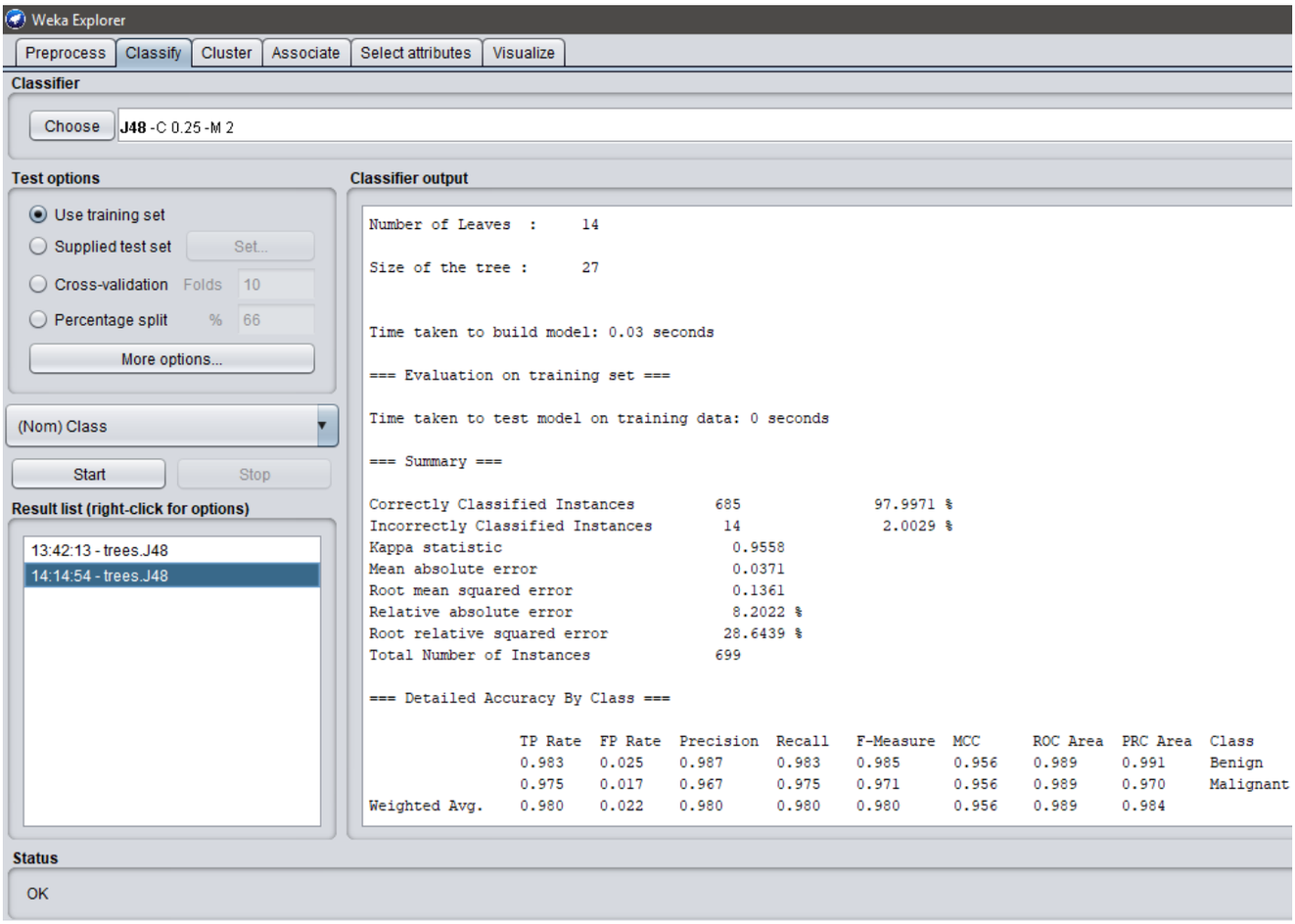
Classification Rules from Tree

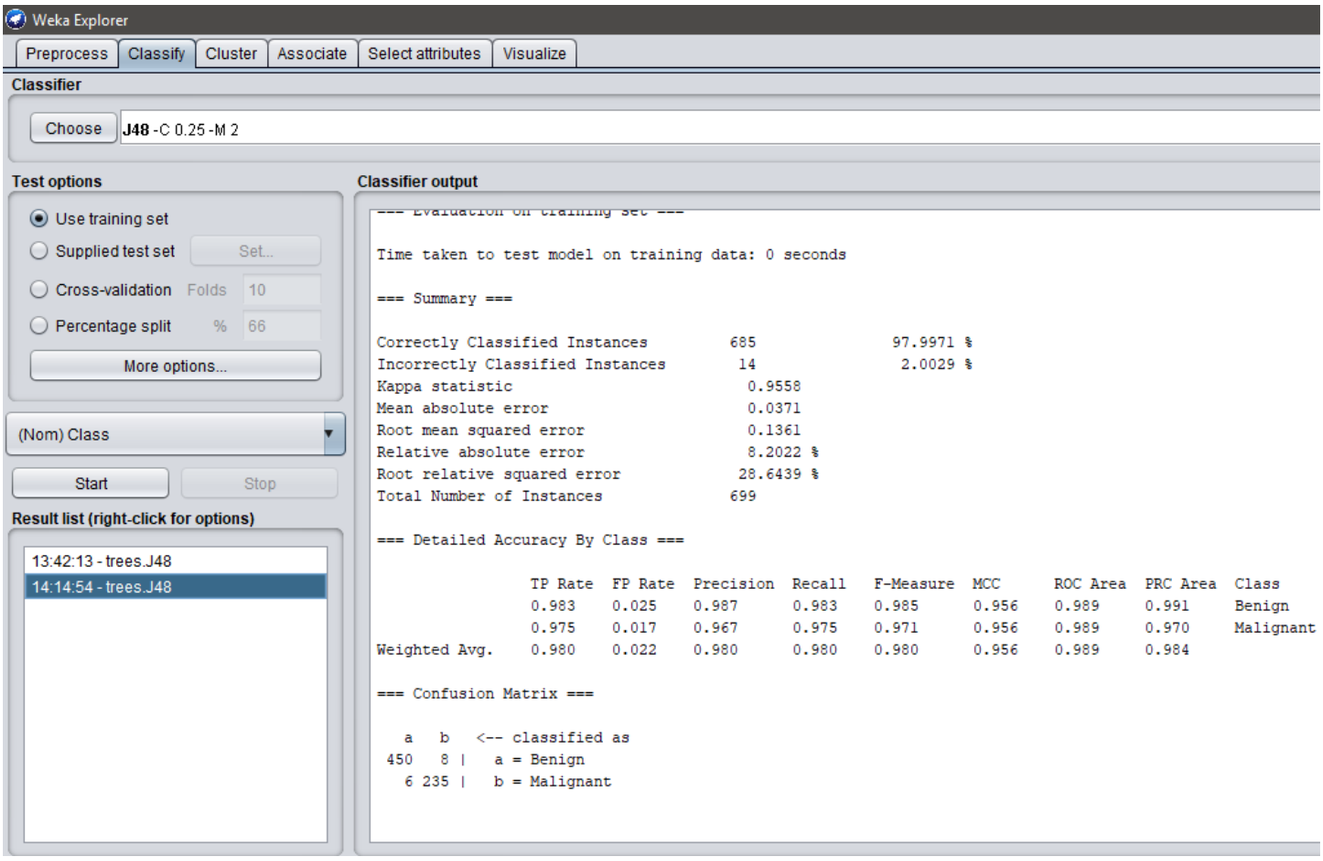
- **IF** Uniformity of Cell Size <=2 **AND** Bare Nuclei <=3 **THEN** Class = “Benign”
- **IF** Uniformity of Cell Size <=2 **AND** Bare Nuclei >3 **AND** Clump Thickness <= 3 **THEN** Class = “Benign”
- **IF** Uniformity of Cell Size <=2 **AND** Bare Nuclei >3 **AND** Clump Thickness > 3 **AND** Bland Chromation > 2 **THEN** Class = “Malignant”
- **IF** Uniformity of Cell Size <=2 **AND** Bare Nuclei >3 **AND** Clump Thickness >3 **AND** Bland Chromation <= 2 **AND** Marginal Adhesion <= 3 **THEN** Class = “Malignant”
- **IF** Uniformity of Cell Size <=2 **AND** Bare Nuclei >3 **AND** Clump Thickness >3 **AND** Bland Chromation <= 2 **AND** Marginal Adhesion > 3 **THEN** Class = “Benign”
- **IF** Uniformity of Cell Size > 2 **AND** Uniformity of Cell Shape <=2 **AND** Clump Thickness <= 5 **THEN** Class = “Benign”
- **IF** Uniformity of Cell Size > 2 **AND** Uniformity of Cell Shape <=2 **AND** Clump Thickness > 5 **THEN** Class = “Malignant”
- **IF** Uniformity of Cell Size > 2 **AND** Uniformity of Cell Shape >4 **THEN** Class = “Malignant”

OUTPUT – J48 TEST MODE – TRAINING SET






Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Classifier
Choose
J48 -C 0.25 -M 2
Test options
Use training set
Supplied test set
Set...
Cross-validation
Folds
10
Percentage split
%
66
More options...
(Nom) Class
Start
Stop
Result list (right-click for options)
13:42:13 - trees.J48
14:14:54 - trees.J48
Classifier output
Number of Leaves : 14
Size of the tree : 27
Time taken to build model: 0.03 seconds
=== Evaluation on training set ===
Time taken to test model on training data: 0 seconds
=== Summary ===
Correctly Classified Instances 685 97.9971 %
Incorrectly Classified Instances 14 2.0029 %
Kappa statistic 0.9558
Mean absolute error 0.0371
Root mean squared error 0.1361
Relative absolute error 8.2022 %
Root relative squared error 28.6439 %
Total Number of Instances 699
=== Detailed Accuracy By Class ===
TP Rate FP Rate Precision Recall F-Measure MCC ROC Area PRC Area Class
0.983 0.025 0.987 0.983 0.985 0.956 0.989 0.991 Benign
0.975 0.017 0.967 0.975 0.971 0.956 0.989 0.970 Malignant
Weighted Avg. 0.980 0.022 0.980 0.980 0.980 0.956 0.989 0.984
Status
OK


Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Classifier
Choose
J48 -C 0.25 -M 2
Test options
Use training set
Supplied test set
Set...
Cross-validation
Folds
10
Percentage split
%
66
More options...
(Nom) Class
Start
Stop
Result list (right-click for options)
13:42:13 - trees.J48
14:14:54 - trees.J48
Classifier output
Time taken to test model on training data: 0 seconds
=== Summary ===
Correctly Classified Instances 685 97.9971 %
Incorrectly Classified Instances 14 2.0029 %
Kappa statistic 0.9558
Mean absolute error 0.0371
Root mean squared error 0.1361
Relative absolute error 8.2022 %
Root relative squared error 28.6439 %
Total Number of Instances 699
=== Detailed Accuracy By Class ===
TP Rate FP Rate Precision Recall F-Measure MCC ROC Area PRC Area Class
0.983 0.025 0.987 0.983 0.985 0.956 0.989 0.991 Benign
0.975 0.017 0.967 0.975 0.971 0.956 0.989 0.970 Malignant
Weighted Avg. 0.980 0.022 0.980 0.980 0.980 0.956 0.989 0.984
=== Confusion Matrix ===
a b <-- classified as
450 8 | a = Benign
6 235 | b = Malignant

Category 5 - Naive Bayes - NaiveBayes – Weka

- Naive Bayes is a simple technique for constructing classifiers: models that assign class labels to problem instances, represented as vectors of feature values, where the class labels are drawn from some finite set.

1. Source Code - (Taken from Weka Tool)

NaiveBayes.java

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

package weka.classifiers.bayes;
import java.util.Collections;
import java.util.Enumeration;
import java.util.Vector;
import weka.classifiers.AbstractClassifier;
import weka.core.*;
import weka.core.Capabilities.Capability;
import weka.core.TechnicalInformation.Field;
import weka.core.TechnicalInformation.Type;
import weka.estimators.DiscreteEstimator;
import weka.estimators.Estimator;
import weka.estimators.KernelEstimator;
import weka.estimators.NormalEstimator;

public class NaiveBayes extends AbstractClassifier implements OptionHandler,
  WeightedInstancesHandler, WeightedAttributesHandler, TechnicalInformationHandler,
  Aggregateable<NaiveBayes> {

  static final long serialVersionUID = 5995231201785697655L;

  protected Estimator[][] m_Distributions;
  protected Estimator m_ClassDistribution;
  protected boolean m_UseKernelEstimator = false;
  protected boolean m_UseDiscretization = false;
  protected int m_NumClasses;
  protected Instances m_Instances;
  protected static final double DEFAULT_NUM_PRECISION = 0.01;
  protected weka.filters.supervised.attribute.Discretize m_Disc = null;
  protected boolean m_displayModelInOldFormat = false;

  public String globalInfo() {
    return "Class for a Naive Bayes classifier using estimator classes. Numeric"
      + " estimator precision values are chosen based on analysis of the "
      + " training data. For this reason, the classifier is not an"
      + " UpdateableClassifier (which in typical usage are initialized with zero"
      + " training instances) -- if you need the UpdateableClassifier functionality,"
      + " use the NaiveBayesUpdateable classifier. The NaiveBayesUpdateable"
      + " classifier will  use a default precision of 0.1 for numeric attributes"
      + " when buildClassifier is called with zero training instances.\n\n"
      + "For more information on Naive Bayes classifiers, see\n\n"
      + getTechnicalInformation().toString();
  }

  @Override
  public TechnicalInformation getTechnicalInformation() {
    TechnicalInformation result;

    result = new TechnicalInformation(Type.INPROCEEDINGS);
    result.setValue(Field.AUTHOR, "George H. John and Pat Langley");
```

```
    result.setValue(Field.TITLE,
      "Estimating Continuous Distributions in Bayesian Classifiers");
    result.setValue(Field.BOOKTITLE,
      "Eleventh Conference on Uncertainty in Artificial Intelligence");
    result.setValue(Field.YEAR, "1995");
    result.setValue(Field.PAGES, "338-345");
    result.setValue(Field.PUBLISHER, "Morgan Kaufmann");
    result.setValue(Field.ADDRESS, "San Mateo");

    return result;
  }

  @Override
  public Capabilities getCapabilities() {
    Capabilities result = super.getCapabilities();
    result.disableAll();

    result.enable(Capability.NOMINAL_ATTRIBUTES);
    result.enable(Capability.NUMERIC_ATTRIBUTES);
    result.enable( Capability.MISSING_VALUES );

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

    result.setMinimumNumberInstances(0);

    return result;
  }

  @Override
  public void buildClassifier(Instances instances) throws Exception {

    getCapabilities().testWithFail(instances);

    instances = new Instances(instances);
    instances.deleteWithMissingClass();

    m_NumClasses = instances.numClasses();

    m_Instances = new Instances(instances);

    if (m_UseDiscretization) {
      m_Disc = new weka.filters.supervised.attribute.Discretize();
      m_Disc.setInputFormat(m_Instances);
      m_Instances = weka.filters.Filter.useFilter(m_Instances, m_Disc);
    } else {
      m_Disc = null;
    }

    m_Distributions = new Estimator[m_Instances.numAttributes() - 1][m_Instances
      .numClasses()];
    m_ClassDistribution = new DiscreteEstimator(m_Instances.numClasses(), true);
    int attIndex = 0;
    Enumeration<Attribute> enu = m_Instances.enumerateAttributes();
    while (enu.hasMoreElements()) {
      Attribute attribute = enu.nextElement();

      double numPrecision = DEFAULT_NUM_PRECISION;
      if (attribute.type() == Attribute.NUMERIC) {
        m_Instances.sort(attribute);
        if ((m_Instances.numInstances() > 0)
          && !m_Instances.instance(0).isMissing(attribute)) {
          double lastVal = m_Instances.instance(0).value(attribute);
          double currentVal, deltaSum = 0;
          int distinct = 0;
          for (int i = 1; i < m_Instances.numInstances(); i++) {
            Instance currentInst = m_Instances.instance(i);
            if (currentInst.isMissing(attribute)) {
              break;
            }
            currentVal = currentInst.value(attribute);
            if (currentVal != lastVal) {
              deltaSum += currentVal - lastVal;
              lastVal = currentVal;
              distinct++;
            }
          }
```

```
        if (distinct > 0) {
          numPrecision = deltaSum / distinct;
        }
      }
    }

    for (int j = 0; j < m_Instances.numClasses(); j++) {
      switch (attribute.type()) {
      case Attribute.NUMERIC:
        if (m_UseKernelEstimator) {
          m_Distributions[attIndex][j] = new KernelEstimator(numPrecision);
        } else {
          m_Distributions[attIndex][j] = new NormalEstimator(numPrecision);
        }
        break;
      case Attribute.NOMINAL:
        m_Distributions[attIndex][j] = new DiscreteEstimator(
          attribute.numValues(), true);
        break;
      default:
        throw new Exception("Attribute type unknown to NaiveBayes");
      }
    }
    attIndex++;
  }

  Enumeration<Instance> enumInsts = m_Instances.enumerateInstances();
  while (enumInsts.hasMoreElements()) {
    Instance instance = enumInsts.nextElement();
    updateClassifier(instance);
  }

  m_Instances = new Instances(m_Instances, 0);
}

public void updateClassifier(Instance instance) throws Exception {

  if (!instance.classIsMissing()) {
    Enumeration<Attribute> enumAtts = m_Instances.enumerateAttributes();
    int attIndex = 0;
    while (enumAtts.hasMoreElements()) {
      Attribute attribute = enumAtts.nextElement();
      if (!instance.isMissing(attribute)) {
        m_Distributions[attIndex][(int) instance.classValue()].addValue(
          instance.value(attribute), instance.weight());
      }
      attIndex++;
    }
    m_ClassDistribution.addValue(instance.classValue(), instance.weight());
  }
}

@Override
public double[] distributionForInstance(Instance instance) throws Exception {

  if (m_UseDiscretization) {
    m_Disc.input(instance);
    instance = m_Disc.output();
  }
  double[] probs = new double[m_NumClasses];
  for (int j = 0; j < m_NumClasses; j++) {
    probs[j] = m_ClassDistribution.getProbability(j);
  }
  Enumeration<Attribute> enumAtts = instance.enumerateAttributes();
  int attIndex = 0;
  while (enumAtts.hasMoreElements()) {
    Attribute attribute = enumAtts.nextElement();
    if (!instance.isMissing(attribute)) {
      double temp, max = 0;
      for (int j = 0; j < m_NumClasses; j++) {
        temp = Math.max(1e-75, Math.pow(m_Distributions[attIndex][j]
          .getProbability(instance.value(attribute)),
          m_Instances.attribute(attIndex).weight()));
        probs[j] *= temp;
        if (probs[j] > max) {
          max = probs[j];
        }
```

```
        if (Double.isNaN(probs[j])) {
          throw new Exception("NaN returned from estimator for attribute "
            + attribute.name() + ":\n"
            + m_Distributions[attIndex][j].toString());
        }
      }
      if ((max > 0) && (max < 1e-75)) { // Danger of probability underflow
        for (int j = 0; j < m_NumClasses; j++) {
          probs[j] *= 1e75;
        }
      }
    }
    attIndex++;
  }

  Utils.normalize(probs);
  return probs;
}

@Override
public Enumeration<Option> listOptions() {

  Vector<Option> newVector = new Vector<Option>(3);

  newVector.addElement(new Option(
    "\tUse kernel density estimator rather than normal\n"
      + "\tdistribution for numeric attributes", "K", 0, "-K"));
  newVector.addElement(new Option(
    "\tUse supervised discretization to process numeric attributes\n", "D",
    0, "-D"));

  newVector
    .addElement(new Option(
      "\tDisplay model in old format (good when there are "
        + "many classes)\n", "O", 0, "-O"));

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

  return newVector.elements();
}

@Override
public void setOptions(String[] options) throws Exception {

  super.setOptions(options);
  boolean k = Utils.getFlag('K', options);
  boolean d = Utils.getFlag('D', options);
  if (k && d) {
    throw new IllegalArgumentException("Can't use both kernel density "
      + "estimation and discretization!");
  }
  setUseSupervisedDiscretization(d);
  setUseKernelEstimator(k);
  setDisplayModelInOldFormat(Utils.getFlag('O', options));
  Utils.checkForRemainingOptions(options);
}

@Override
public String[] getOptions() {

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

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

  if (m_UseKernelEstimator) {
    options.add("-K");
  }

  if (m_UseDiscretization) {
    options.add("-D");
  }

  if (m_displayModelInOldFormat) {
    options.add("-O");
  }

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

```
}

@Override
public String toString() {
  if (m_displayModelInOldFormat) {
    return toStringOriginal();
  }

  StringBuffer temp = new StringBuffer();
  temp.append("Naive Bayes Classifier");
  if (m_Instances == null) {
    temp.append(": No model built yet.");
  } else {

    int maxWidth = 0;
    int maxAttWidth = 0;
    boolean containsKernel = false;

    for (int i = 0; i < m_Instances.numClasses(); i++) {
      if (m_Instances.classAttribute().value(i).length() > maxWidth) {
        maxWidth = m_Instances.classAttribute().value(i).length();
      }
    }
    for (int i = 0; i < m_Instances.numAttributes(); i++) {
      if (i != m_Instances.classIndex()) {
        Attribute a = m_Instances.attribute(i);
        if (a.name().length() > maxAttWidth) {
          maxAttWidth = m_Instances.attribute(i).name().length();
        }
        if (a.isNominal()) {
          for (int j = 0; j < a.numValues(); j++) {
            String val = a.value(j) + "  ";
            if (val.length() > maxAttWidth) {
              maxAttWidth = val.length();
            }
          }
        }
      }
    }

    for (Estimator[] m_Distribution : m_Distributions) {
      for (int j = 0; j < m_Instances.numClasses(); j++) {
        if (m_Distribution[0] instanceof NormalEstimator) {
          NormalEstimator n = (NormalEstimator) m_Distribution[j];
          double mean = Math.log(Math.abs(n.getMean())) / Math.log(10.0);
          double precision = Math.log(Math.abs(n.getPrecision()))/ Math.log(10.0);
          double width = (mean > precision) ? mean : precision;
          if (width < 0) {
            width = 1;
          }
          width += 6.0;
          if ((int) width > maxWidth) {
            maxWidth = (int) width;
          }
        } else if (m_Distribution[0] instanceof KernelEstimator) {
          containsKernel = true;
          KernelEstimator ke = (KernelEstimator) m_Distribution[j];
          int numK = ke.getNumKernels();
          String temps = "K" + numK + ": mean (weight)";
          if (maxAttWidth < temps.length()) {
            maxAttWidth = temps.length();
          }
          if (ke.getNumKernels() > 0) {
            double[] means = ke.getMeans();
            double[] weights = ke.getWeights();
            for (int k = 0; k < ke.getNumKernels(); k++) {
              String m = Utils.doubleToString(means[k], maxWidth, 4).trim();
              m += " ("
                + Utils.doubleToString(weights[k], maxWidth, 1).trim() + ")";
              if (maxWidth < m.length()) {
                maxWidth = m.length();
              }
            }
          }
        } else if (m_Distribution[0] instanceof DiscreteEstimator) {
          DiscreteEstimator d = (DiscreteEstimator) m_Distribution[j];
          for (int k = 0; k < d.getNumSymbols(); k++) {
```

```
        String size = "" + d.getCount(k);
        if (size.length() > maxWidth) {
          maxWidth = size.length();
        }
      }
      int sum = ("" + d.getSumOfCounts()).length();
      if (sum > maxWidth) {
        maxWidth = sum;
      }
    }
  }
}

for (int i = 0; i < m_Instances.numClasses(); i++) {
  String cSize = m_Instances.classAttribute().value(i);
  if (cSize.length() > maxWidth) {
    maxWidth = cSize.length();
  }
}

for (int i = 0; i < m_Instances.numClasses(); i++) {
  String priorP = Utils.doubleToString(
    ((DiscreteEstimator) m_ClassDistribution).getProbability(i),
    maxWidth, 2).trim();
  priorP = "(" + priorP + ")";
  if (priorP.length() > maxWidth) {
    maxWidth = priorP.length();
  }
}

if (maxAttWidth < "Attribute".length()) {
  maxAttWidth = "Attribute".length();
}

if (maxAttWidth < "  weight sum".length()) {
  maxAttWidth = "  weight sum".length();
}

if (containsKernel) {
  if (maxAttWidth < "  [precision]".length()) {
    maxAttWidth = "  [precision]".length();
  }
}

maxAttWidth += 2;

temp.append("\n\n");
temp.append(pad("Class", " ",
  (maxAttWidth + maxWidth + 1) - "Class".length(), true));

temp.append("\n");
temp.append(pad("Attribute", " ", maxAttWidth - "Attribute".length(),
  false));
for (int i = 0; i < m_Instances.numClasses(); i++) {
  String classL = m_Instances.classAttribute().value(i);
  temp.append(pad(classL, " ", maxWidth + 1 - classL.length(), true));
}
temp.append("\n");
temp.append(pad("", " ", maxAttWidth, true));
for (int i = 0; i < m_Instances.numClasses(); i++) {
  String priorP = Utils.doubleToString(
    ((DiscreteEstimator) m_ClassDistribution).getProbability(i),
    maxWidth, 2).trim();
  priorP = "(" + priorP + ")";
  temp.append(pad(priorP, " ", maxWidth + 1 - priorP.length(), true));
}
temp.append("\n");
temp.append(pad(
  "",
  "=",
  maxAttWidth + (maxWidth * m_Instances.numClasses())
    + m_Instances.numClasses() + 1, true));
temp.append("\n");

int counter = 0;
for (int i = 0; i < m_Instances.numAttributes(); i++) {
  if (i == m_Instances.classIndex()) {
```

```
      continue;
    }
    String attName = m_Instances.attribute(i).name();
    temp.append(attName + "\n");

    if (m_Distributions[counter][0] instanceof NormalEstimator) {
      String meanL = "  mean";
      temp.append(pad(meanL, " ", maxAttWidth + 1 - meanL.length(), false));
      for (int j = 0; j < m_Instances.numClasses(); j++) {
        NormalEstimator n = (NormalEstimator) m_Distributions[counter][j];
        String mean = Utils.doubleToString(n.getMean(), maxWidth, 4).trim();
        temp.append(pad(mean, " ", maxWidth + 1 - mean.length(), true));
      }
      temp.append("\n");
      String stdDevL = "  std. dev.";
      temp.append(pad(stdDevL, " ", maxAttWidth + 1 - stdDevL.length(),
        false));
      for (int j = 0; j < m_Instances.numClasses(); j++) {
        NormalEstimator n = (NormalEstimator) m_Distributions[counter][j];
        String stdDev = Utils.doubleToString(n.getStdDev(), maxWidth, 4)
          .trim();
        temp.append(pad(stdDev, " ", maxWidth + 1 - stdDev.length(), true));
      }
      temp.append("\n");
      String weightL = "  weight sum";
      temp.append(pad(weightL, " ", maxAttWidth + 1 - weightL.length(),
        false));
      for (int j = 0; j < m_Instances.numClasses(); j++) {
        NormalEstimator n = (NormalEstimator) m_Distributions[counter][j];
        String weight = Utils.doubleToString(n.getSumOfWeights(), maxWidth,
          4).trim();
        temp.append(pad(weight, " ", maxWidth + 1 - weight.length(), true));
      }
      temp.append("\n");
      String precisionL = "  precision";
      temp.append(pad(precisionL, " ",
        maxAttWidth + 1 - precisionL.length(), false));
      for (int j = 0; j < m_Instances.numClasses(); j++) {
        NormalEstimator n = (NormalEstimator) m_Distributions[counter][j];
        String precision = Utils.doubleToString(n.getPrecision(), maxWidth,
          4).trim();
        temp.append(pad(precision, " ", maxWidth + 1 - precision.length(),
          true));
      }
      temp.append("\n\n");

    } else if (m_Distributions[counter][0] instanceof DiscreteEstimator) {
      Attribute a = m_Instances.attribute(i);
      for (int j = 0; j < a.numValues(); j++) {
        String val = "  " + a.value(j);
        temp.append(pad(val, " ", maxAttWidth + 1 - val.length(), false));
        for (int k = 0; k < m_Instances.numClasses(); k++) {
          DiscreteEstimator d = (DiscreteEstimator) m_Distributions[counter][k];
          String count = "" + d.getCount(j);
          temp.append(pad(count, " ", maxWidth + 1 - count.length(), true));
        }
        temp.append("\n");
      }
      String total = "  [total]";
      temp.append(pad(total, " ", maxAttWidth + 1 - total.length(), false));
      for (int k = 0; k < m_Instances.numClasses(); k++) {
        DiscreteEstimator d = (DiscreteEstimator) m_Distributions[counter][k];
        String count = "" + d.getSumOfCounts();
        temp.append(pad(count, " ", maxWidth + 1 - count.length(), true));
      }
      temp.append("\n\n");
    } else if (m_Distributions[counter][0] instanceof KernelEstimator) {
      String kL = "  [# kernels]";
      temp.append(pad(kL, " ", maxAttWidth + 1 - kL.length(), false));
      for (int k = 0; k < m_Instances.numClasses(); k++) {
        KernelEstimator ke = (KernelEstimator) m_Distributions[counter][k];
        String nk = "" + ke.getNumKernels();
        temp.append(pad(nk, " ", maxWidth + 1 - nk.length(), true));
      }
      temp.append("\n");
      String stdDevL = "  [std. dev]";
      temp.append(pad(stdDevL, " ", maxAttWidth + 1 - stdDevL.length(),
```

```
          false));
        for (int k = 0; k < m_Instances.numClasses(); k++) {
          KernelEstimator ke = (KernelEstimator) m_Distributions[counter][k];
          String stdD = Utils.doubleToString(ke.getStdDev(), maxWidth, 4)
            .trim();
          temp.append(pad(stdD, " ", maxWidth + 1 - stdD.length(), true));
        }
        temp.append("\n");
        String precL = "  [precision]";
        temp.append(pad(precL, " ", maxAttWidth + 1 - precL.length(), false));
        for (int k = 0; k < m_Instances.numClasses(); k++) {
          KernelEstimator ke = (KernelEstimator) m_Distributions[counter][k];
          String prec = Utils.doubleToString(ke.getPrecision(), maxWidth, 4)
            .trim();
          temp.append(pad(prec, " ", maxWidth + 1 - prec.length(), true));
        }
        temp.append("\n");
        int maxK = 0;
        for (int k = 0; k < m_Instances.numClasses(); k++) {
          KernelEstimator ke = (KernelEstimator) m_Distributions[counter][k];
          if (ke.getNumKernels() > maxK) {
            maxK = ke.getNumKernels();
          }
        }
        for (int j = 0; j < maxK; j++) {
          String meanL = "  K" + (j + 1) + ": mean (weight)";
          temp
            .append(pad(meanL, " ", maxAttWidth + 1 - meanL.length(), false));
          for (int k = 0; k < m_Instances.numClasses(); k++) {
            KernelEstimator ke = (KernelEstimator) m_Distributions[counter][k];
            double[] means = ke.getMeans();
            double[] weights = ke.getWeights();
            String m = "--";
            if (ke.getNumKernels() == 0) {
              m = "" + 0;
            } else if (j < ke.getNumKernels()) {
              m = Utils.doubleToString(means[j], maxWidth, 4).trim();
              m += " ("
                + Utils.doubleToString(weights[j], maxWidth, 1).trim() + ")";
            }
            temp.append(pad(m, " ", maxWidth + 1 - m.length(), true));
          }
          temp.append("\n");
        }
        temp.append("\n");
      }

      counter++;
    }
  }

  return temp.toString();
}

protected String toStringOriginal() {

  StringBuffer text = new StringBuffer();

  text.append("Naive Bayes Classifier");
  if (m_Instances == null) {
    text.append(": No model built yet.");
  } else {
    try {
      for (int i = 0; i < m_Distributions[0].length; i++) {
        text.append("\n\nClass " + m_Instances.classAttribute().value(i)
          + ": Prior probability = "
          + Utils.doubleToString(m_ClassDistribution.getProbability(i), 4, 2)
          + "\n\n");
        Enumeration<Attribute> enumAtts = m_Instances.enumerateAttributes();
        int attIndex = 0;
        while (enumAtts.hasMoreElements()) {
          Attribute attribute = enumAtts.nextElement();
          if (attribute.weight() > 0) {
            text.append(attribute.name() + ":  "
              + m_Distributions[attIndex][i]);
          }
          attIndex++;
```

```
        }
      }
    } catch (Exception ex) {
      text.append(ex.getMessage());
    }
  }

  return text.toString();
}

private String pad(String source, String padChar, int length, boolean leftPad) {
  StringBuffer temp = new StringBuffer();

  if (leftPad) {
    for (int i = 0; i < length; i++) {
      temp.append(padChar);
    }
    temp.append(source);
  } else {
    temp.append(source);
    for (int i = 0; i < length; i++) {
      temp.append(padChar);
    }
  }
  return temp.toString();
}

public String useKernelEstimatorTipText() {
  return "Use a kernel estimator for numeric attributes rather than a "
    + "normal distribution.";
}

public boolean getUseKernelEstimator() {

  return m_UseKernelEstimator;
}

public void setUseKernelEstimator(boolean v) {

  m_UseKernelEstimator = v;
  if (v) {
    setUseSupervisedDiscretization(false);
  }
}

public String useSupervisedDiscretizationTipText() {
  return "Use supervised discretization to convert numeric attributes to nominal "
    + "ones.";
}

public boolean getUseSupervisedDiscretization() {

  return m_UseDiscretization;
}

public void setUseSupervisedDiscretization(boolean newblah) {

  m_UseDiscretization = newblah;
  if (newblah) {
    setUseKernelEstimator(false);
  }
}

public String displayModelInOldFormatTipText() {
  return "Use old format for model output. The old format is "
    + "better when there are many class values. The new format "
    + "is better when there are fewer classes and many attributes.";
}

public void setDisplayModelInOldFormat(boolean d) {
  m_displayModelInOldFormat = d;
}

public boolean getDisplayModelInOldFormat() {
  return m_displayModelInOldFormat;
}
```

```
  public Instances getHeader() {
    return m_Instances;
  }

  public Estimator[][] getConditionalEstimators() {
    return m_Distributions;
  }

  public Estimator getClassEstimator() {
    return m_ClassDistribution;
  }

  @Override
  public String getRevision() {
    return RevisionUtils.extract("$Revision: 14534 $");
  }

  @SuppressWarnings({ "rawtypes", "unchecked" })
  @Override
  public NaiveBayes aggregate(NaiveBayes toAggregate) throws Exception {

    if (m_UseDiscretization || toAggregate.getUseSupervisedDiscretization()) {
      throw new Exception("Unable to aggregate when supervised discretization "
        + "has been turned on");
    }

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

    ((Aggregateable) m_ClassDistribution)
      .aggregate(toAggregate.m_ClassDistribution);

    for (int i = 0; i < m_Distributions.length; i++) {
      for (int j = 0; j < m_Distributions[i].length; j++) {
        ((Aggregateable) m_Distributions[i][j])
          .aggregate(toAggregate.m_Distributions[i][j]);
      }
    }
    return this;
  }

  @Override
  public void finalizeAggregation() throws Exception {
  }

  public static void main(String[] argv) {
    runClassifier(new NaiveBayes(), argv);
  }
}
```

AbstactClassifier.java

```
/*
 *   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/>.
 */

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

package weka.classifiers;
```

```
import weka.core.Attribute;
import weka.core.BatchPredictor;
import weka.core.Capabilities;
import weka.core.CapabilitiesHandler;
import weka.core.CapabilitiesIgnorer;
import weka.core.CommandlineRunnable;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.Option;
import weka.core.OptionHandler;
import weka.core.RevisionHandler;
import weka.core.RevisionUtils;
import weka.core.SerializedObject;
import weka.core.Utils;

import java.io.Serializable;
import java.util.Enumeration;
import java.util.Vector;

public abstract class AbstractClassifier implements Classifier, BatchPredictor,
  Cloneable, Serializable, OptionHandler, CapabilitiesHandler, RevisionHandler,
  CapabilitiesIgnorer, CommandlineRunnable {
  private static final long serialVersionUID = 6502780192411755341L;
  protected boolean m_Debug = false;
  protected boolean m_DoNotCheckCapabilities = false;
  public static int NUM_DECIMAL_PLACES_DEFAULT = 2;
  protected int m_numDecimalPlaces = NUM_DECIMAL_PLACES_DEFAULT;
  public static String BATCH_SIZE_DEFAULT = "100";
  protected String m_BatchSize = BATCH_SIZE_DEFAULT;

  public static Classifier forName(String classifierName, String[] options)
    throws Exception {

    return ((AbstractClassifier) Utils.forName(Classifier.class, classifierName,
      options));
  }

  public static Classifier makeCopy(Classifier model) throws Exception {
    return (Classifier) new SerializedObject(model).getObject();
  }

  public static Classifier[] makeCopies(Classifier model, int num)
    throws Exception {

    if (model == null) {
      throw new Exception("No model classifier set");
    }
    Classifier[] classifiers = new Classifier[num];
    SerializedObject so = new SerializedObject(model);
    for (int i = 0; i < classifiers.length; i++) {
      classifiers[i] = (Classifier) so.getObject();
    }
    return classifiers;
  }

  public static void runClassifier(Classifier classifier, String[] options) {
    try {
      if (classifier instanceof CommandlineRunnable) {
        ((CommandlineRunnable)classifier).preExecution();
      }
      System.out.println(Evaluation.evaluateModel(classifier, options));
    } catch (Exception e) {
      if (((e.getMessage() != null)
        && (e.getMessage().indexOf("General options") == -1))
        || (e.getMessage() == null)) {
        e.printStackTrace();
      } else {
        System.err.println(e.getMessage());
      }
    }
    if (classifier instanceof CommandlineRunnable) {
      try {
        ((CommandlineRunnable) classifier).postExecution();
      } catch (Exception ex) {
        ex.printStackTrace();
      }
```

```
  }
}

@Override
public double classifyInstance(Instance instance) throws Exception {

  double[] dist = distributionForInstance(instance);
  if (dist == null) {
    throw new Exception("Null distribution predicted");
  }
  switch (instance.classAttribute().type()) {
  case Attribute.NOMINAL:
    double max = 0;
    int maxIndex = 0;

    for (int i = 0; i < dist.length; i++) {
      if (dist[i] > max) {
        maxIndex = i;
        max = dist[i];
      }
    }
    if (max > 0) {
      return maxIndex;
    } else {
      return Utils.missingValue();
    }
  case Attribute.NUMERIC:
  case Attribute.DATE:
    return dist[0];
  default:
    return Utils.missingValue();
  }
}

@Override
public double[] distributionForInstance(Instance instance) throws Exception {

  double[] dist = new double[instance.numClasses()];
  switch (instance.classAttribute().type()) {
  case Attribute.NOMINAL:
    double classification = classifyInstance(instance);
    if (Utils.isMissingValue(classification)) {
      return dist;
    } else {
      dist[(int) classification] = 1.0;
    }
    return dist;
  case Attribute.NUMERIC:
  case Attribute.DATE:
    dist[0] = classifyInstance(instance);
    return dist;
  default:
    return dist;
  }
}

@Override
public Enumeration<Option> listOptions() {

  Vector<Option> newVector = Option
    .listOptionsForClassHierarchy(this.getClass(), AbstractClassifier.class);

  newVector.addElement(new Option(
    "\tIf set, classifier is run in debug mode and\n"
      + "\tmay output additional info to the console",
    "output-debug-info", 0, "-output-debug-info"));
  newVector.addElement(new Option(
    "\tIf set, classifier capabilities are not checked before classifier is built\n"
      + "\t(use with caution).",
    "-do-not-check-capabilities", 0, "-do-not-check-capabilities"));
  newVector.addElement(new Option(
    "\tThe number of decimal places for the output of numbers in the model"
      + " (default " + m_numDecimalPlaces + ").",
    "num-decimal-places", 1, "-num-decimal-places"));
  newVector.addElement(new Option(
          "\tThe desired batch size for batch prediction " + " (default " + m_BatchSize + ").",
          "batch-size", 1, "-batch-size"));
```

```
    return newVector.elements();
  }

  @Override
  public String[] getOptions() {

    Vector<String> options = new Vector<String>();
    for (String s : Option.getOptionsForHierarchy(this, AbstractClassifier.class)) {
      options.add(s);
    }

    if (getDebug()) {
      options.add("-output-debug-info");
    }
    if (getDoNotCheckCapabilities()) {
      options.add("-do-not-check-capabilities");
    }
    if (getNumDecimalPlaces() != NUM_DECIMAL_PLACES_DEFAULT) {
      options.add("-num-decimal-places");
      options.add("" + getNumDecimalPlaces());
    }
    if (!(getBatchSize().equals(BATCH_SIZE_DEFAULT))) {
      options.add("-batch-size");
      options.add("" + getBatchSize());
    }
    return options.toArray(new String[0]);
  }

  @Override
  public void setOptions(String[] options) throws Exception {

    Option.setOptionsForHierarchy(options, this, AbstractClassifier.class);
    setDebug(Utils.getFlag("output-debug-info", options));
    setDoNotCheckCapabilities(
      Utils.getFlag("do-not-check-capabilities", options));

    String optionString = Utils.getOption("num-decimal-places", options);
    if (optionString.length() != 0) {
      setNumDecimalPlaces((new Integer(optionString)).intValue());
    }
    optionString = Utils.getOption("batch-size", options);
    if (optionString.length() != 0) {
      setBatchSize(optionString);
    }
  }

  public boolean getDebug() {
  return m_Debug;
  }

  public void setDebug(boolean debug) {
    m_Debug = debug;
  }

  public String debugTipText() {
    return "If set to true, classifier may output additional info to the console.";
  }

  @Override
  public boolean getDoNotCheckCapabilities() {
    return m_DoNotCheckCapabilities;
  }

  @Override
  public void setDoNotCheckCapabilities(boolean doNotCheckCapabilities) {
    m_DoNotCheckCapabilities = doNotCheckCapabilities;
  }

  public String doNotCheckCapabilitiesTipText() {
    return "If set, classifier capabilities are not checked before classifier is built"
      + " (Use with caution to reduce runtime).";
  }

  public String numDecimalPlacesTipText() {
    return "The number of decimal places to be used for the output of numbers in "
```

```
      + "the model.";
  }

  public int getNumDecimalPlaces() {
    return m_numDecimalPlaces;
  }

  public void setNumDecimalPlaces(int num) {
    m_numDecimalPlaces = num;
  }

  public String batchSizeTipText() {
    return "The preferred number of instances to process if batch prediction is "
      + "being performed. More or fewer instances may be provided, but this gives "
      + "implementations a chance to specify a preferred batch size.";
  }

  @Override
  public void setBatchSize(String size) {
    m_BatchSize = size;
  }

  @Override
  public String getBatchSize() {
    return m_BatchSize;
  }

  @Override
  public boolean implementsMoreEfficientBatchPrediction() {
    return false;
  }

  @Override
  public double[][] distributionsForInstances(Instances batch)
    throws Exception {
    double[][] batchPreds = new double[batch.numInstances()][];
    for (int i = 0; i < batch.numInstances(); i++) {
      batchPreds[i] = distributionForInstance(batch.instance(i));
    }

    return batchPreds;
  }

  @Override
  public Capabilities getCapabilities() {
    Capabilities result = new Capabilities(this);
    result.enableAll();

    return result;
  }

  @Override
  public String getRevision() {
    return RevisionUtils.extract("$Revision: 14259 $");
  }

  @Override
  public void preExecution() throws Exception {
  }

  @Override
  public void run(Object toRun, String[] options) throws Exception {
    if (!(toRun instanceof Classifier)) {
      throw new IllegalArgumentException("Object to run is not a Classifier!");
    }
    runClassifier((Classifier) toRun, options);
  }

  @Override
  public void postExecution() throws Exception {
  }
}
```

DiscreteEstimator.java

```
/*
 *   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/>.
 */

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

package weka.estimators;

import weka.core.Capabilities;
import weka.core.Capabilities.Capability;
import weka.core.Aggregateable;
import weka.core.RevisionUtils;
import weka.core.Utils;

public class DiscreteEstimator extends Estimator implements
    IncrementalEstimator, Aggregateable<DiscreteEstimator> {

  private static final long serialVersionUID = -5526486742612434779L;
  private final double[] m_Counts;
  private double m_SumOfCounts;
  private double m_FPrior;
  public DiscreteEstimator(int numSymbols, boolean laplace) {
    m_Counts = new double[numSymbols];
    m_SumOfCounts = 0;
    if (laplace) {
      m_FPrior = 1;
      for (int i = 0; i < numSymbols; i++) {
        m_Counts[i] = 1;
      }
      m_SumOfCounts = numSymbols;
    }
  }

  public DiscreteEstimator(int nSymbols, double fPrior) {
    m_Counts = new double[nSymbols];
    m_FPrior = fPrior;
    for (int iSymbol = 0; iSymbol < nSymbols; iSymbol++) {
      m_Counts[iSymbol] = fPrior;
    }
    m_SumOfCounts = fPrior * nSymbols;
  }

  @Override
  public void addValue(double data, double weight) {

    m_Counts[(int) data] += weight;
    m_SumOfCounts += weight;
  }

  @Override
  public double getProbability(double data) {
    if (m_SumOfCounts == 0) {
      return 0;
    }
    return m_Counts[(int) data] / m_SumOfCounts;
  }

  public int getNumSymbols() {
    return (m_Counts == null) ? 0 : m_Counts.length;
  }

  public double getCount(double data) {
    if (m_SumOfCounts == 0) {
      return 0;
```

```
  }
  return m_Counts[(int) data];
}

public double getSumOfCounts() {
  return m_SumOfCounts;
}

@Override
public String toString() {
  StringBuffer result = new StringBuffer("Discrete Estimator. Counts = ");
  if (m_SumOfCounts > 1) {
    for (int i = 0; i < m_Counts.length; i++) {
      result.append(" ").append(Utils.doubleToString(m_Counts[i], 2));
    }
    result.append("  (Total = ").append(
        Utils.doubleToString(m_SumOfCounts, 2));
    result.append(")\n");
  } else {
    for (int i = 0; i < m_Counts.length; i++) {
      result.append(" ").append(m_Counts[i]);
    }
    result.append("  (Total = ").append(m_SumOfCounts).append(")\n");
  }
  return result.toString();
}

@Override
public Capabilities getCapabilities() {
  Capabilities result = super.getCapabilities();
  result.disableAll();

  if (!m_noClass) {
    result.enable(Capability.NOMINAL_CLASS);
    result.enable(Capability.MISSING_CLASS_VALUES);
  } else {
    result.enable(Capability.NO_CLASS);
  }

  result.enable(Capability.NUMERIC_ATTRIBUTES);
  return result;
}

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

@Override
public DiscreteEstimator aggregate(DiscreteEstimator toAggregate)
    throws Exception {

  if (toAggregate.m_Counts.length != m_Counts.length) {
    throw new Exception("DiscreteEstimator to aggregate has a different "
        + "number of symbols");
  }

  m_SumOfCounts += toAggregate.m_SumOfCounts;
  for (int i = 0; i < m_Counts.length; i++) {
    m_Counts[i] += (toAggregate.m_Counts[i] - toAggregate.m_FPrior);
  }

  m_SumOfCounts -= (toAggregate.m_FPrior * m_Counts.length);

  return this;
}

@Override
public void finalizeAggregation() throws Exception {
}

protected static void testAggregation() {
  DiscreteEstimator df = new DiscreteEstimator(5, true);
  DiscreteEstimator one = new DiscreteEstimator(5, true);
  DiscreteEstimator two = new DiscreteEstimator(5, true);

  java.util.Random r = new java.util.Random(1);
```

```
    for (int i = 0; i < 100; i++) {
      int z = r.nextInt(5);
      df.addValue(z, 1);

      if (i < 50) {
        one.addValue(z, 1);
      } else {
        two.addValue(z, 1);
      }
    }

    try {
      System.out.println("\n\nFull\n");
      System.out.println(df.toString());
      System.out.println("Prob (0): " + df.getProbability(0));

      System.out.println("\nOne\n" + one.toString());
      System.out.println("Prob (0): " + one.getProbability(0));

      System.out.println("\nTwo\n" + two.toString());
      System.out.println("Prob (0): " + two.getProbability(0));

      one = one.aggregate(two);

      System.out.println("\nAggregated\n");
      System.out.println(one.toString());
      System.out.println("Prob (0): " + one.getProbability(0));
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  public static void main(String[] argv) {

    try {
      if (argv.length == 0) {
        System.out.println("Please specify a set of instances.");
        return;
      }
      int current = Integer.parseInt(argv[0]);
      int max = current;
      for (int i = 1; i < argv.length; i++) {
        current = Integer.parseInt(argv[i]);
        if (current > max) {
          max = current;
        }
      }
      DiscreteEstimator newEst = new DiscreteEstimator(max + 1, true);
      for (int i = 0; i < argv.length; i++) {
        current = Integer.parseInt(argv[i]);
        System.out.println(newEst);
        System.out.println("Prediction for " + current + " = "
            + newEst.getProbability(current));
        newEst.addValue(current, 1);
      }

      DiscreteEstimator.testAggregation();
    } catch (Exception e) {
      System.out.println(e.getMessage());
    }
  }
}
```

KernelEstimator.java

```
/*
 *   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/>.
 */

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

package weka.estimators;

import weka.core.Capabilities;
import weka.core.Capabilities.Capability;
import weka.core.Aggregateable;
import weka.core.RevisionUtils;
import weka.core.Statistics;
import weka.core.Utils;

public class KernelEstimator extends Estimator implements IncrementalEstimator,
    Aggregateable<KernelEstimator> {

  private static final long serialVersionUID = 3646923563367683925L;
  private double[] m_Values;
  private double[] m_Weights;
  private int m_NumValues;
  private double m_SumOfWeights;
  private double m_StandardDev;
  private double m_Precision;
  private boolean m_AllWeightsOne;
  private static double MAX_ERROR = 0.01;
  private int findNearestValue(double key) {
    int low = 0;
    int high = m_NumValues;
    int middle = 0;
    while (low < high) {
      middle = (low + high) / 2;
      double current = m_Values[middle];
      if (current == key) {
        return middle;
      }
      if (current > key) {
        high = middle;
      } else if (current < key) {
        low = middle + 1;
      }
    }
    return low;
  }

  private double round(double data) {
    return Math.rint(data / m_Precision) * m_Precision;
  }
  public KernelEstimator(double precision) {
    m_Values = new double[50];
    m_Weights = new double[50];
    m_NumValues = 0;
    m_SumOfWeights = 0;
    m_AllWeightsOne = true;
    m_Precision = precision;
    if (m_Precision < Utils.SMALL)
      m_Precision = Utils.SMALL;
    m_StandardDev = m_Precision / (2 * 3);
  }

  @Override
  public void addValue(double data, double weight) {
    if (weight == 0) {
      return;
    }
    data = round(data);
    int insertIndex = findNearestValue(data);
    if ((m_NumValues <= insertIndex) || (m_Values[insertIndex] != data)) {
      if (m_NumValues < m_Values.length) {
        int left = m_NumValues - insertIndex;
```

```
        System
            .arraycopy(m_Values, insertIndex, m_Values, insertIndex + 1, left);
        System.arraycopy(m_Weights, insertIndex, m_Weights, insertIndex + 1,
            left);

        m_Values[insertIndex] = data;
        m_Weights[insertIndex] = weight;
        m_NumValues++;
      } else {
        double[] newValues = new double[m_Values.length * 2];
        double[] newWeights = new double[m_Values.length * 2];
        int left = m_NumValues - insertIndex;
        System.arraycopy(m_Values, 0, newValues, 0, insertIndex);
        System.arraycopy(m_Weights, 0, newWeights, 0, insertIndex);
        newValues[insertIndex] = data;
        newWeights[insertIndex] = weight;
        System.arraycopy(m_Values, insertIndex, newValues, insertIndex + 1,
            left);
        System.arraycopy(m_Weights, insertIndex, newWeights, insertIndex + 1,
            left);
        m_NumValues++;
        m_Values = newValues;
        m_Weights = newWeights;
      }
      if (weight != 1) {
        m_AllWeightsOne = false;
      }
    } else {
      m_Weights[insertIndex] += weight;
      m_AllWeightsOne = false;
    }
    m_SumOfWeights += weight;
    double range = m_Values[m_NumValues - 1] - m_Values[0];
    if (range > 0) {
      m_StandardDev = Math.max(range / Math.sqrt(m_SumOfWeights),
          m_Precision / (2 * 3));
    }
  }

  @Override
  public double getProbability(double data) {
    double delta = 0, sum = 0, currentProb = 0;
    double zLower = 0, zUpper = 0;
    if (m_NumValues == 0) {
      zLower = (data - (m_Precision / 2)) / m_StandardDev;
      zUpper = (data + (m_Precision / 2)) / m_StandardDev;
      return (Statistics.normalProbability(zUpper) - Statistics
          .normalProbability(zLower));
    }
    double weightSum = 0;
    int start = findNearestValue(data);
    for (int i = start; i < m_NumValues; i++) {
      delta = m_Values[i] - data;
      zLower = (delta - (m_Precision / 2)) / m_StandardDev;
      zUpper = (delta + (m_Precision / 2)) / m_StandardDev;
      currentProb = (Statistics.normalProbability(zUpper) - Statistics.normalProbability(zLower));
      sum += currentProb * m_Weights[i];
      weightSum += m_Weights[i];
      if (currentProb * (m_SumOfWeights - weightSum) < sum * MAX_ERROR) {
        break;
      }
    }
    for (int i = start - 1; i >= 0; i--) {
      delta = m_Values[i] - data;
      zLower = (delta - (m_Precision / 2)) / m_StandardDev;
      zUpper = (delta + (m_Precision / 2)) / m_StandardDev;
      currentProb = (Statistics.normalProbability(zUpper) - Statistics.normalProbability(zLower));
      sum += currentProb * m_Weights[i];
      weightSum += m_Weights[i];
      if (currentProb * (m_SumOfWeights - weightSum) < sum * MAX_ERROR) {
        break;
      }
    }
    return sum / m_SumOfWeights;
  }

  @Override
```

```
  public String toString() {

    String result = m_NumValues + " Normal Kernels. \nStandardDev = "
        + Utils.doubleToString(m_StandardDev, 6, 4) + " Precision = "
        + m_Precision;
    if (m_NumValues == 0) {
      result += "  \nMean = 0";
    } else {
      result += "  \nMeans =";
      for (int i = 0; i < m_NumValues; i++) {
        result += " " + m_Values[i];
      }
      if (!m_AllWeightsOne) {
        result += "\nWeights = ";
        for (int i = 0; i < m_NumValues; i++) {
          result += " " + m_Weights[i];
        }
      }
    }
    return result + "\n";
  }

  public int getNumKernels() {
    return m_NumValues;
  }

  public double[] getMeans() {
    return m_Values;
  }

  public double[] getWeights() {
    return m_Weights;
  }

 public double getPrecision() {
    return m_Precision;  }

  public double getStdDev() {
    return m_StandardDev;
  }
  @Override
  public Capabilities getCapabilities() {
    Capabilities result = super.getCapabilities();
    result.disableAll();
    if (!m_noClass) {
      result.enable(Capability.NOMINAL_CLASS);
      result.enable(Capability.MISSING_CLASS_VALUES);
    } else {
      result.enable(Capability.NO_CLASS);
    }

    result.enable(Capability.NUMERIC_ATTRIBUTES);
    return result;
  }
  @Override
  public String getRevision() {
    return RevisionUtils.extract("$Revision: 9785 $");
  }

  @Override
  public KernelEstimator aggregate(KernelEstimator toAggregate)
      throws Exception {

    for (int i = 0; i < toAggregate.m_NumValues; i++) {
      addValue(toAggregate.m_Values[i], toAggregate.m_Weights[i]);
    }

    return this;
  }

  @Override
  public void finalizeAggregation() throws Exception {
  }

  public static void testAggregation() {
    KernelEstimator ke = new KernelEstimator(0.01);
    KernelEstimator one = new KernelEstimator(0.01);
```

```
    KernelEstimator two = new KernelEstimator(0.01);

    java.util.Random r = new java.util.Random(1);

    for (int i = 0; i < 100; i++) {
      double z = r.nextDouble();

      ke.addValue(z, 1);
      if (i < 50) {
        one.addValue(z, 1);
      } else {
        two.addValue(z, 1);
      }
    }

    try {

      System.out.println("\n\nFull\n");
      System.out.println(ke.toString());
      System.out.println("Prob (0): " + ke.getProbability(0));

      System.out.println("\nOne\n" + one.toString());
      System.out.println("Prob (0): " + one.getProbability(0));

      System.out.println("\nTwo\n" + two.toString());
      System.out.println("Prob (0): " + two.getProbability(0));

      one = one.aggregate(two);

      System.out.println("Aggregated\n");
      System.out.println(one.toString());
      System.out.println("Prob (0): " + one.getProbability(0));
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  public static void main(String[] argv) {

    try {
      if (argv.length < 2) {
        System.out.println("Please specify a set of instances.");
        return;
      }
      KernelEstimator newEst = new KernelEstimator(0.01);
      for (int i = 0; i < argv.length - 3; i += 2) {
        newEst.addValue(Double.valueOf(argv[i]).doubleValue(),
            Double.valueOf(argv[i + 1]).doubleValue());
      }
      System.out.println(newEst);

      double start = Double.valueOf(argv[argv.length - 2]).doubleValue();
      double finish = Double.valueOf(argv[argv.length - 1]).doubleValue();
      for (double current = start; current < finish; current += (finish - start) / 50) {
        System.out.println("Data: " + current + " "
            + newEst.getProbability(current));
      }
      KernelEstimator.testAggregation();
    } catch (Exception e) {
      System.out.println(e.getMessage());
    }
  }
}
```

NormalEstimator.java

```
/*
 *   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.
 *
```

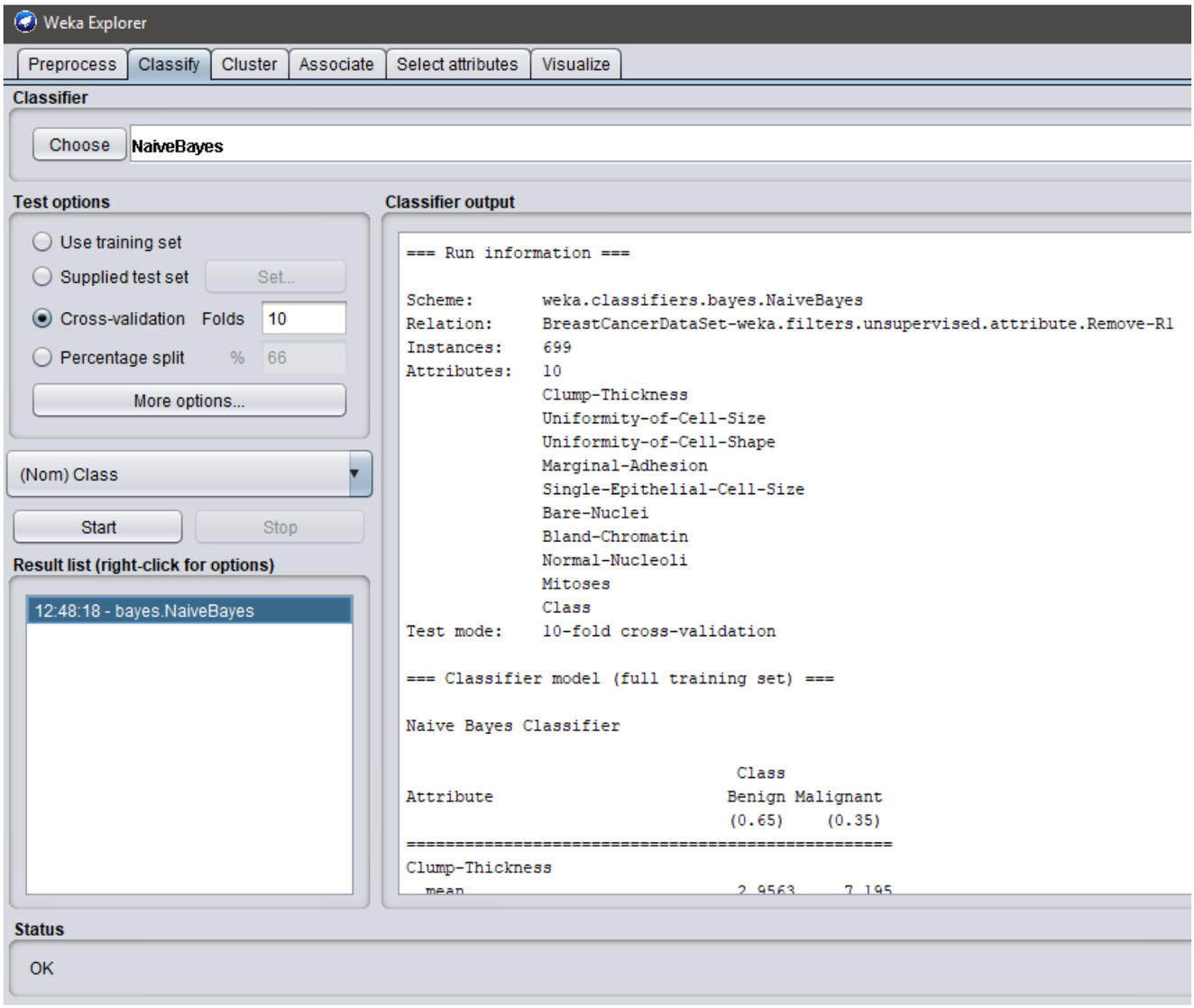
```
 *   You should have received a copy of the GNU General Public License
 *   along with this program.  If not, see <http://www.gnu.org/licenses/>.
 */

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

package weka.estimators;

import weka.core.Capabilities;
import weka.core.Capabilities.Capability;
import weka.core.Aggregateable;
import weka.core.RevisionUtils;
import weka.core.Statistics;
import weka.core.Utils;

public class NormalEstimator extends Estimator implements IncrementalEstimator,
    Aggregateable<NormalEstimator> {

  private static final long serialVersionUID = 93584379632315841L;
  private double m_SumOfWeights;
  private double m_SumOfValues;
  private double m_SumOfValuesSq;
  private double m_Mean;
  private double m_StandardDev;
  private double m_Precision;
  private double round(double data) {
    return Math.rint(data / m_Precision) * m_Precision;
  }
  public NormalEstimator(double precision) {
    m_Precision = precision;
    m_StandardDev = m_Precision / (2 * 3);
  }

  @Override
  public void addValue(double data, double weight) {

    if (weight == 0) {
      return;
    }
    data = round(data);
    m_SumOfWeights += weight;
    m_SumOfValues += data * weight;
    m_SumOfValuesSq += data * data * weight;

    computeParameters();
  }
  protected void computeParameters() {
    if (m_SumOfWeights > 0) {
      m_Mean = m_SumOfValues / m_SumOfWeights;
      double stdDev = Math.sqrt(Math.abs(m_SumOfValuesSq - m_Mean
          * m_SumOfValues)
          / m_SumOfWeights);
      if (stdDev > 1e-10) {
        m_StandardDev = Math.max(m_Precision / (2 * 3),stdDev);
      }
    }
  }
  @Override
  public double getProbability(double data) {

    data = round(data);
    double zLower = (data - m_Mean - (m_Precision / 2)) / m_StandardDev;
    double zUpper = (data - m_Mean + (m_Precision / 2)) / m_StandardDev;

    double pLower = Statistics.normalProbability(zLower);
    double pUpper = Statistics.normalProbability(zUpper);
    return pUpper - pLower;
  }

  @Override
  public String toString() {

    return "Normal Distribution. Mean = " + Utils.doubleToString(m_Mean, 4)
```

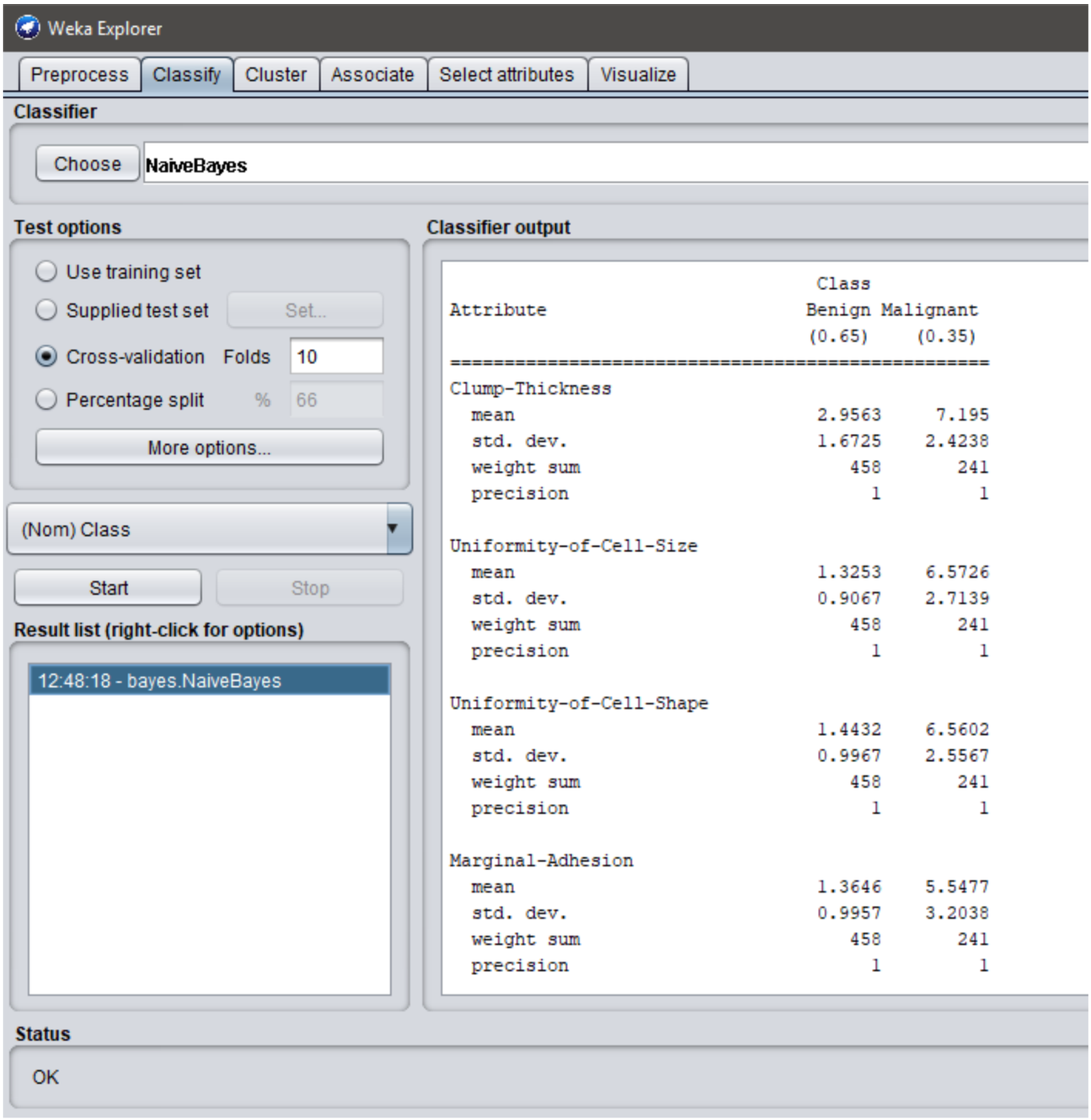
```
      + " StandardDev = " + Utils.doubleToString(m_StandardDev, 4)
      + " WeightSum = " + Utils.doubleToString(m_SumOfWeights, 4)
      + " Precision = " + m_Precision + "\n";
}

@Override
public Capabilities getCapabilities() {
  Capabilities result = super.getCapabilities();
  result.disableAll();

  if (!m_noClass) {
    result.enable(Capability.NOMINAL_CLASS);
    result.enable(Capability.MISSING_CLASS_VALUES);
  } else {
    result.enable(Capability.NO_CLASS);
  }

  result.enable(Capability.NUMERIC_ATTRIBUTES);
  return result;
}

public double getMean() {
  return m_Mean;
}
public double getStdDev() {
  return m_StandardDev;
}

public double getPrecision() {
  return m_Precision;
}

public double getSumOfWeights() {
  return m_SumOfWeights;
}

@Override
public String getRevision() {
  return RevisionUtils.extract("$Revision: 9785 $");
}

@Override
public NormalEstimator aggregate(NormalEstimator toAggregate)
    throws Exception {

  m_SumOfWeights += toAggregate.m_SumOfWeights;
  m_SumOfValues += toAggregate.m_SumOfValues;
  m_SumOfValuesSq += toAggregate.m_SumOfValuesSq;

  if (toAggregate.m_Precision < m_Precision) {
    m_Precision = toAggregate.m_Precision;
  }

  computeParameters();

  return this;
}

@Override
public void finalizeAggregation() throws Exception {
}

public static void testAggregation() {
  NormalEstimator ne = new NormalEstimator(0.01);
  NormalEstimator one = new NormalEstimator(0.01);
  NormalEstimator two = new NormalEstimator(0.01);

  java.util.Random r = new java.util.Random(1);

  for (int i = 0; i < 100; i++) {
    double z = r.nextDouble();

    ne.addValue(z, 1);
    if (i < 50) {
      one.addValue(z, 1);
    } else {
      two.addValue(z, 1);
```

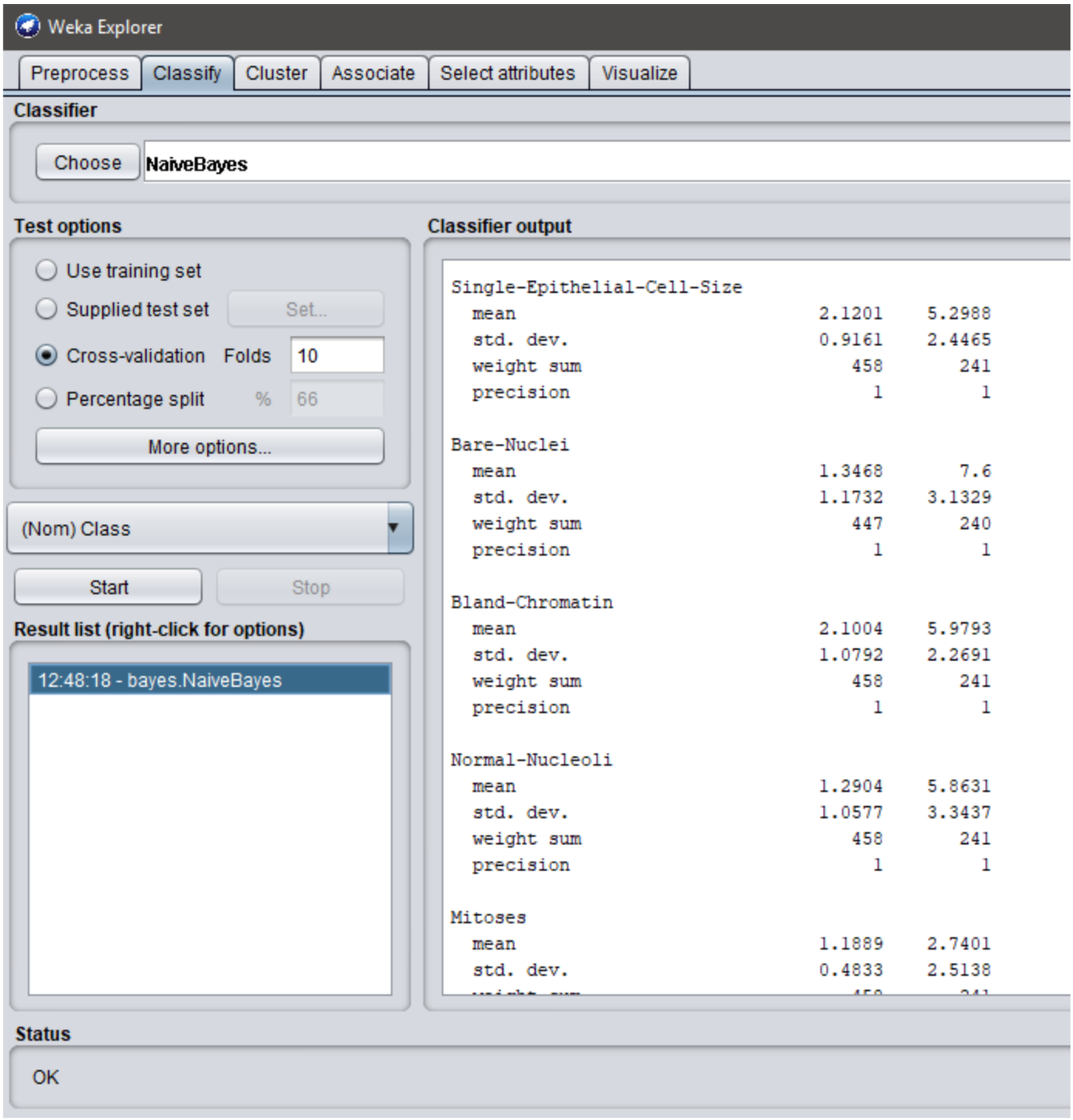
```
      }
    }
    try {
      System.out.println("\n\nFull\n");
      System.out.println(ne.toString());
      System.out.println("Prob (0): " + ne.getProbability(0));

      System.out.println("\nOne\n" + one.toString());
      System.out.println("Prob (0): " + one.getProbability(0));

      System.out.println("\nTwo\n" + two.toString());
      System.out.println("Prob (0): " + two.getProbability(0));
      one = one.aggregate(two);
      System.out.println("\nAggregated\n");
      System.out.println(one.toString());
      System.out.println("Prob (0): " + one.getProbability(0));
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  }
  public static void main(String[] argv) {
    try {
      if (argv.length == 0) {
        System.out.println("Please specify a set of instances.");
        return;
      }
      NormalEstimator newEst = new NormalEstimator(0.01);
      for (int i = 0; i < argv.length; i++) {
        double current = Double.valueOf(argv[i]).doubleValue();
        System.out.println(newEst);
        System.out.println("Prediction for " + current + " = "
            + newEst.getProbability(current));
        newEst.addValue(current, 1);
      }
      NormalEstimator.testAggregation();
    } catch (Exception e) {
      System.out.println(e.getMessage());
    }
  }
}
```

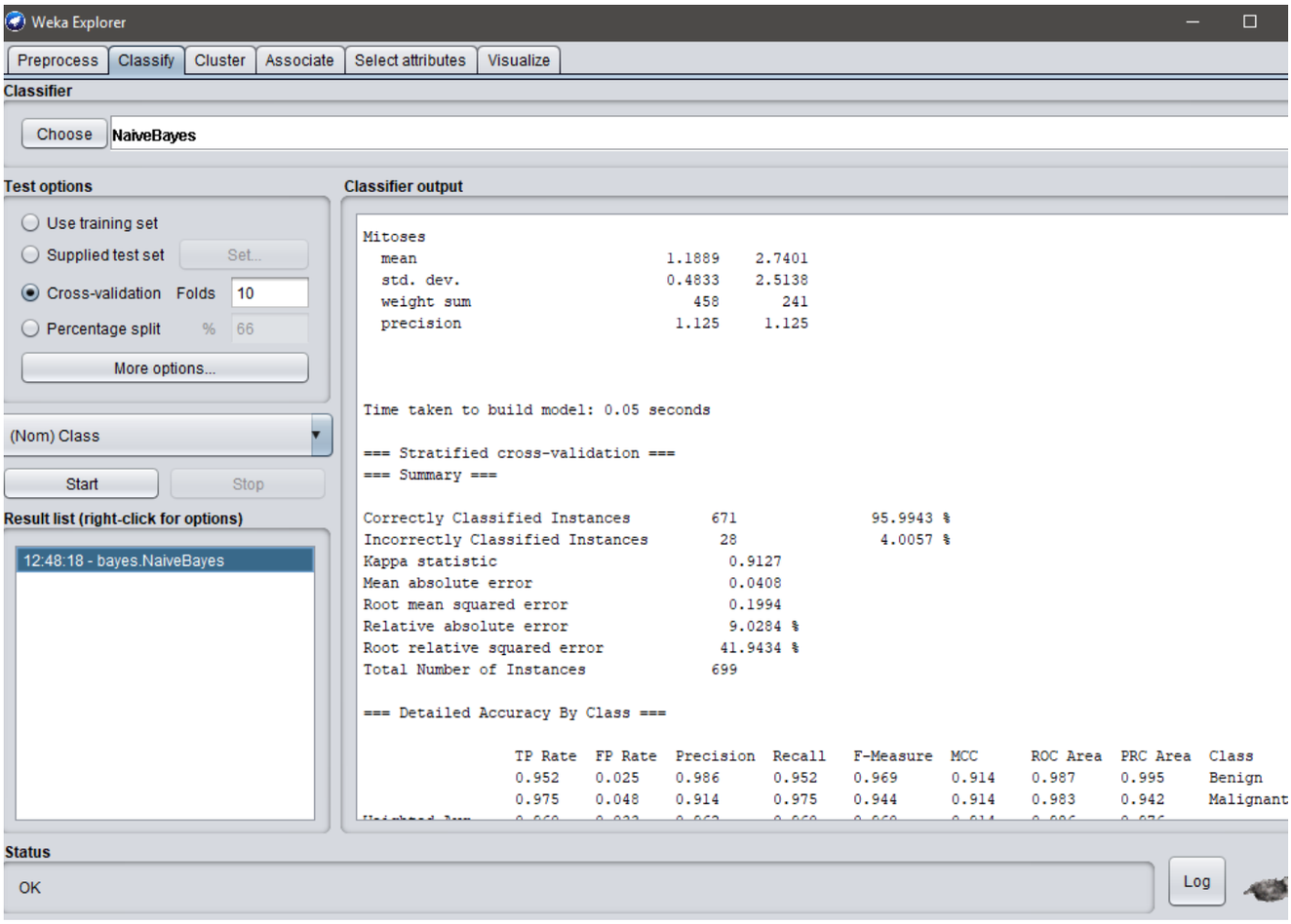
OUTPUT - NAIVEBAYES

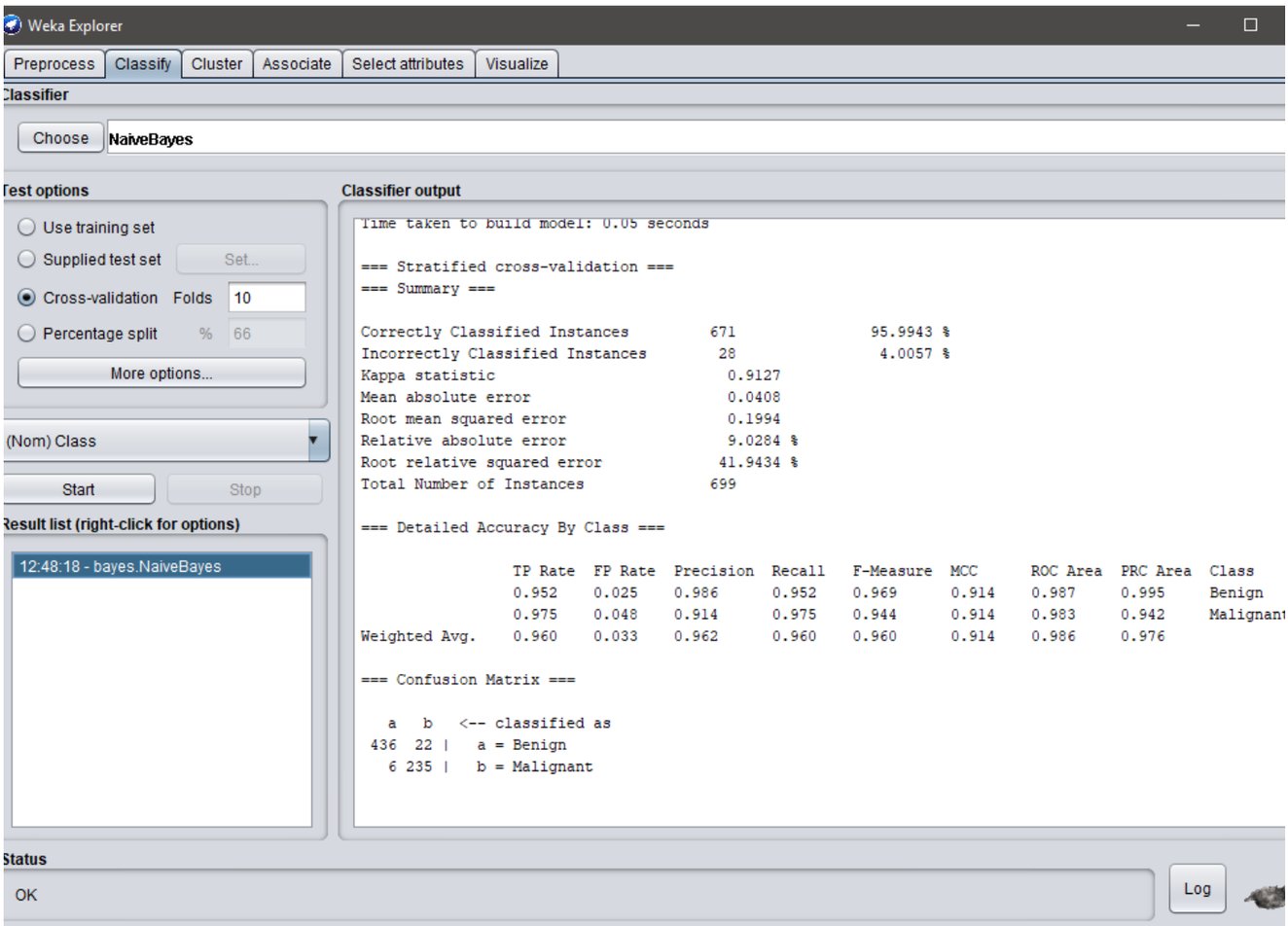
TEST MODE – 10-FOLD CROSS VALIDATION



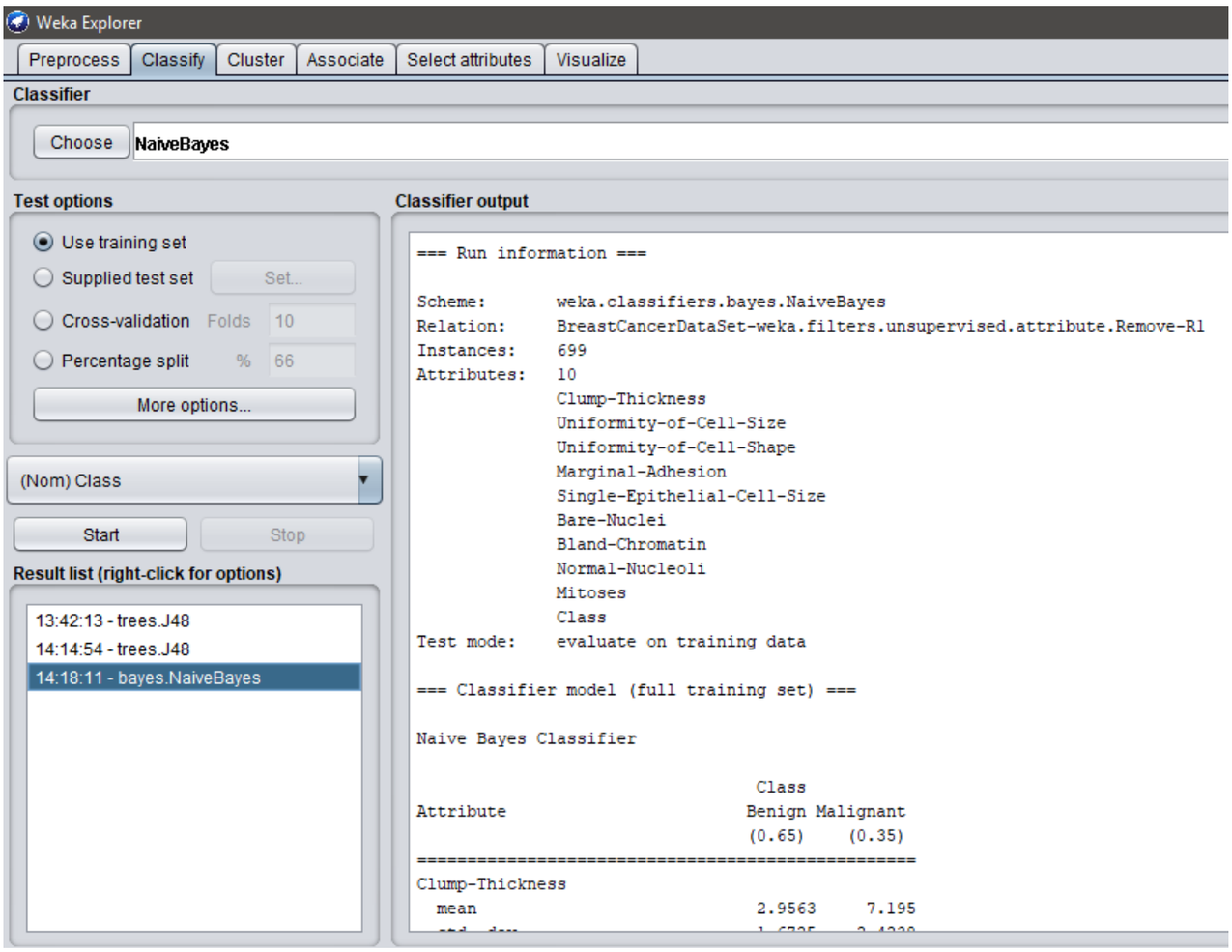

Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Classifier
Choose
NaiveBayes
Test options
Use training set
Supplied test set
Set...
Cross-validation
Folds
10
Percentage split
%
66
More options...
(Nom) Class
Start
Stop
Result list (right-click for options)
12:48:18 - bayes.NaiveBayes
Classifier output
Class
Attribute
Benign
Malignant
(0.65)
(0.35)
Clump-Thickness
mean 2.9563 7.195
std. dev. 1.6725 2.4238
weight sum 458 241
precision 1 1
Uniformity-of-Cell-Size
mean 1.3253 6.5726
std. dev. 0.9067 2.7139
weight sum 458 241
precision 1 1
Uniformity-of-Cell-Shape
mean 1.4432 6.5602
std. dev. 0.9967 2.5567
weight sum 458 241
precision 1 1
Marginal-Adhesion
mean 1.3646 5.5477
std. dev. 0.9957 3.2038
weight sum 458 241
precision 1 1
Status
OK

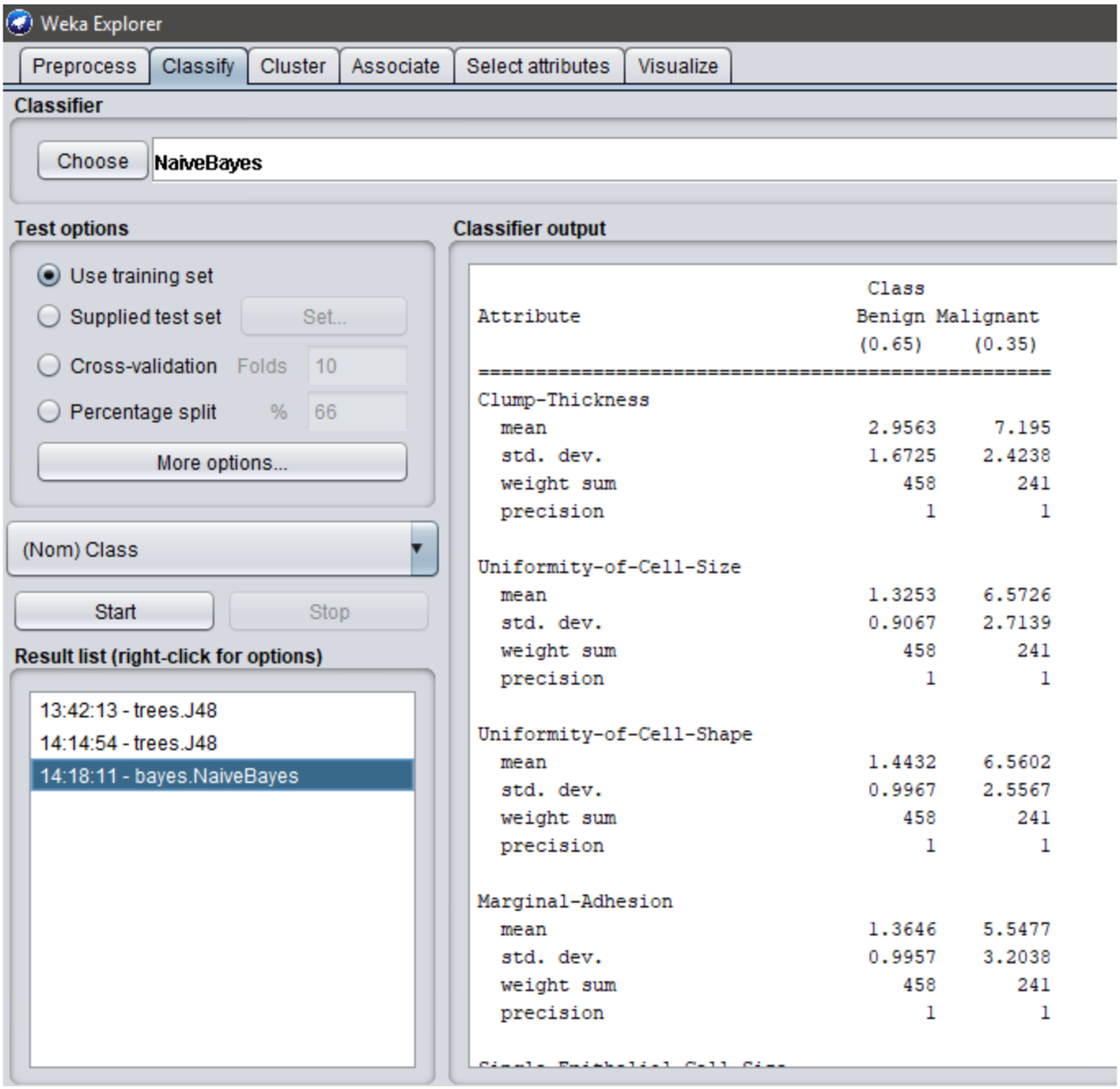

Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Classifier
Choose
NaiveBayes
Test options
Use training set
Supplied test set
Set...
Cross-validation
Folds
10
Percentage split
%
66
More options...
(Nom) Class
Start
Stop
Result list (right-click for options)
12:48:18 - bayes.NaiveBayes
Classifier output
Single-Epithelial-Cell-Size
mean 2.1201 5.2988
std. dev. 0.9161 2.4465
weight sum 458 241
precision 1 1
Bare-Nuclei
mean 1.3468 7.6
std. dev. 1.1732 3.1329
weight sum 447 240
precision 1 1
Bland-Chromatin
mean 2.1004 5.9793
std. dev. 1.0792 2.2691
weight sum 458 241
precision 1 1
Normal-Nucleoli
mean 1.2904 5.8631
std. dev. 1.0577 3.3437
weight sum 458 241
precision 1 1
Mitoses
mean 1.1889 2.7401
std. dev. 0.4833 2.5138
Status
OK

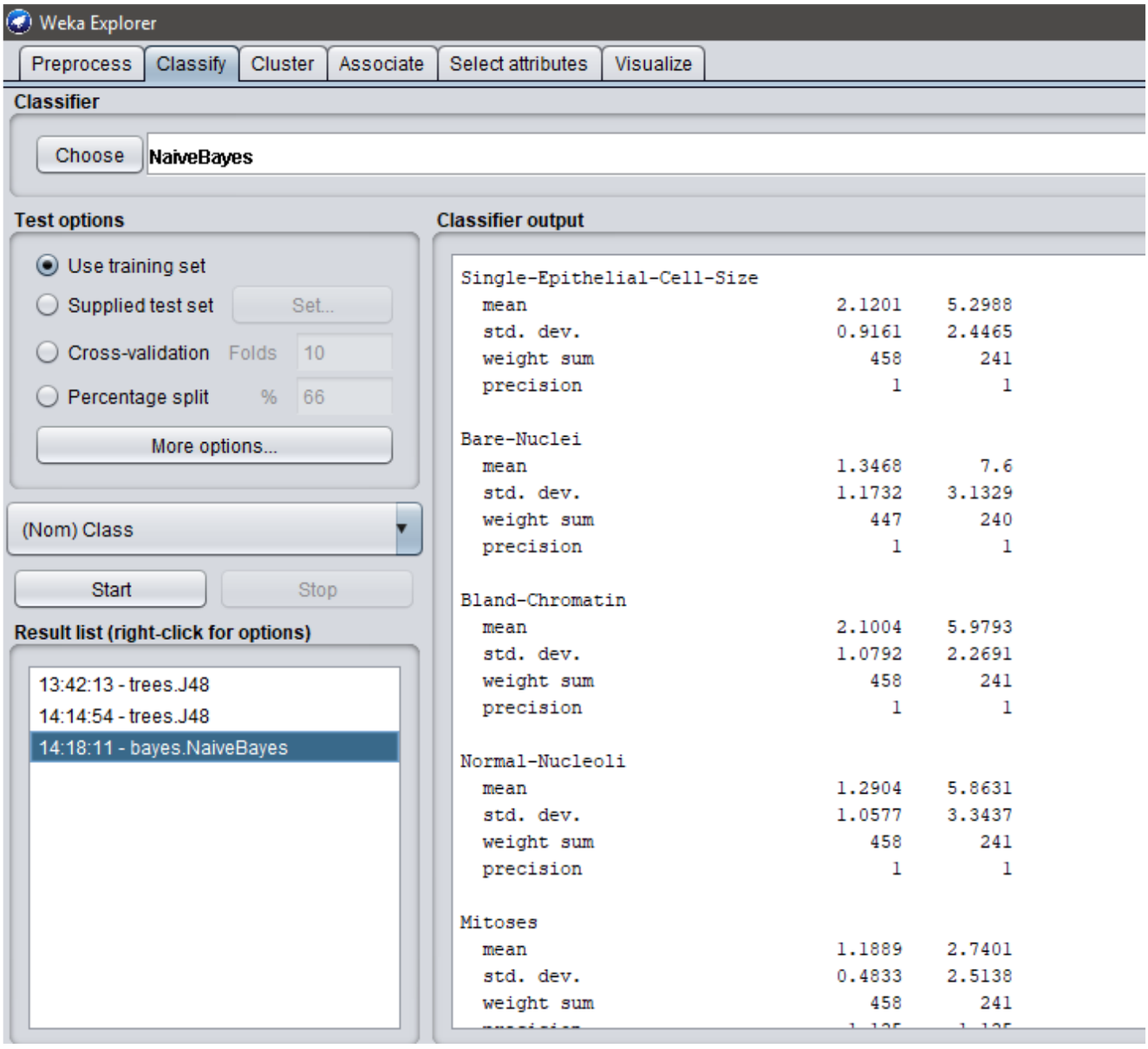

Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Classifier
Choose
NaiveBayes
Test options
Use training set
Supplied test set
Set...
Cross-validation
Folds
10
Percentage split
%
66
More options...
(Nom) Class
Start
Stop
Result list (right-click for options)
12:48:18 - bayes.NaiveBayes
Classifier output
Mitoses
mean 1.1889 2.7401
std. dev. 0.4833 2.5138
weight sum 458 241
precision 1.125 1.125
Time taken to build model: 0.05 seconds
=== Stratified cross-validation ===
=== Summary ===
Correctly Classified Instances 671 95.9943 %
Incorrectly Classified Instances 28 4.0057 %
Kappa statistic 0.9127
Mean absolute error 0.0408
Root mean squared error 0.1994
Relative absolute error 9.0284 %
Root relative squared error 41.9434 %
Total Number of Instances 699
=== Detailed Accuracy By Class ===
TP Rate FP Rate Precision Recall F-Measure MCC ROC Area PRC Area Class
0.952 0.025 0.986 0.952 0.969 0.914 0.987 0.995 Benign
0.975 0.048 0.914 0.975 0.944 0.914 0.983 0.942 Malignant
Status
OK
Log

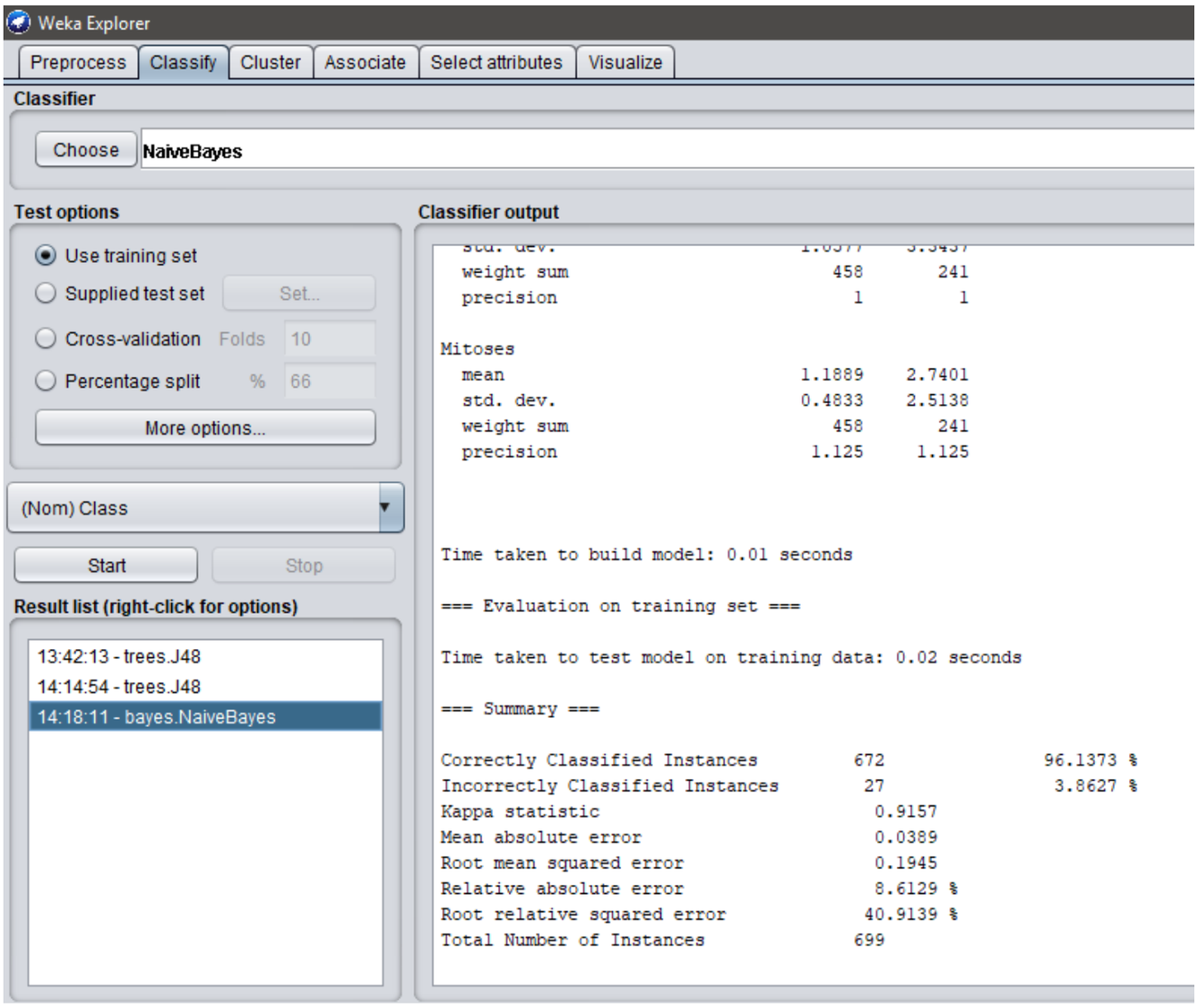

Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Classifier
Choose
NaiveBayes
Test options
Use training set
Supplied test set
Set...
Cross-validation
Folds
10
Percentage split
%
66
More options...
(Nom) Class
Start
Stop
Result list (right-click for options)
12:48:18 - bayes.NaiveBayes
Classifier output
Time taken to build model: 0.05 seconds
=== Stratified cross-validation ===
=== Summary ===
Correctly Classified Instances 671 95.9943 %
Incorrectly Classified Instances 28 4.0057 %
Kappa statistic 0.9127
Mean absolute error 0.0408
Root mean squared error 0.1994
Relative absolute error 9.0284 %
Root relative squared error 41.9434 %
Total Number of Instances 699
=== Detailed Accuracy By Class ===
TP Rate FP Rate Precision Recall F-Measure MCC ROC Area PRC Area Class
0.952 0.025 0.986 0.952 0.969 0.914 0.987 0.995 Benign
0.975 0.048 0.914 0.975 0.944 0.914 0.983 0.942 Malignant
Weighted Avg. 0.960 0.033 0.962 0.960 0.960 0.914 0.986 0.976
=== Confusion Matrix ===
a b <-- classified as
436 22 | a = Benign
6 235 | b = Malignant
Status
OK
Log

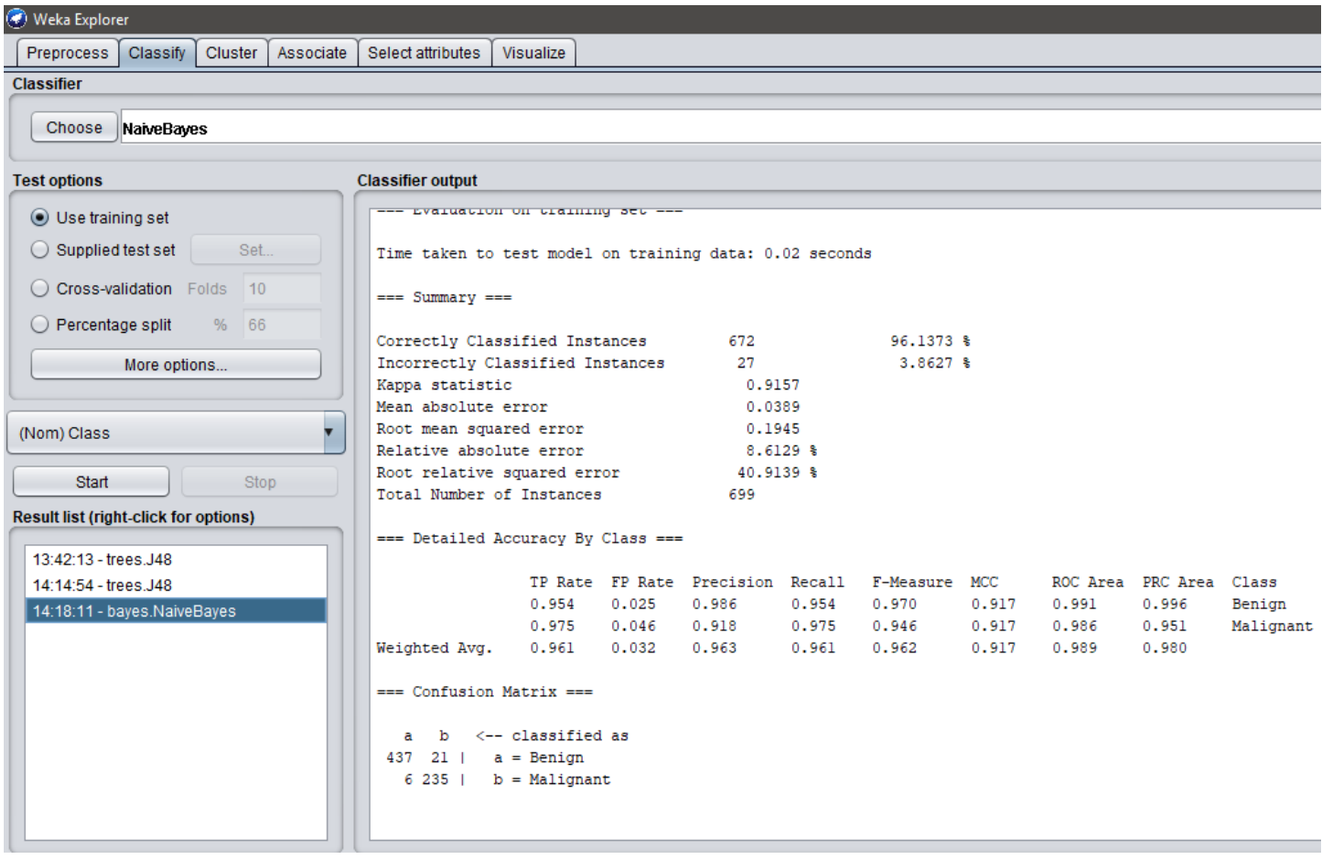
TEST MODE – TRAINING SET




Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Classifier
Choose
NaiveBayes
Test options
Use training set
Supplied test set
Set...
Cross-validation
Folds
10
Percentage split
%
66
More options...
(Nom) Class
Start
Stop
Result list (right-click for options)
13:42:13 - trees.J48
14:14:54 - trees.J48
14:18:11 - bayes.NaiveBayes
Classifier output
Class
Attribute
Benign
(0.65)
Malignant
(0.35)
Clump-Thickness
mean 2.9563 7.195
std. dev. 1.6725 2.4238
weight sum 458 241
precision 1 1
Uniformity-of-Cell-Size
mean 1.3253 6.5726
std. dev. 0.9067 2.7139
weight sum 458 241
precision 1 1
Uniformity-of-Cell-Shape
mean 1.4432 6.5602
std. dev. 0.9967 2.5567
weight sum 458 241
precision 1 1
Marginal-Adhesion
mean 1.3646 5.5477
std. dev. 0.9957 3.2038
weight sum 458 241
precision 1 1


Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Classifier
Choose
NaiveBayes
Test options
Use training set
Supplied test set
Set...
Cross-validation
Folds
10
Percentage split
%
66
More options...
(Nom) Class
Start
Stop
Result list (right-click for options)
13:42:13 - trees.J48
14:14:54 - trees.J48
14:18:11 - bayes.NaiveBayes
Classifier output
Single-Epithelial-Cell-Size
mean 2.1201 5.2988
std. dev. 0.9161 2.4465
weight sum 458 241
precision 1 1
Bare-Nuclei
mean 1.3468 7.6
std. dev. 1.1732 3.1329
weight sum 447 240
precision 1 1
Bland-Chromatin
mean 2.1004 5.9793
std. dev. 1.0792 2.2691
weight sum 458 241
precision 1 1
Normal-Nucleoli
mean 1.2904 5.8631
std. dev. 1.0577 3.3437
weight sum 458 241
precision 1 1
Mitoses
mean 1.1889 2.7401
std. dev. 0.4833 2.5138
weight sum 458 241


Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Classifier
Choose
NaiveBayes
Test options
Use training set
Supplied test set
Set...
Cross-validation
Folds
10
Percentage split
%
66
More options...
(Nom) Class
Start
Stop
Result list (right-click for options)
13:42:13 - trees.J48
14:14:54 - trees.J48
14:18:11 - bayes.NaiveBayes
Classifier output
std. dev. 1.0577 3.5457
weight sum 458 241
precision 1 1
Mitoses
mean 1.1889 2.7401
std. dev. 0.4833 2.5138
weight sum 458 241
precision 1.125 1.125
Time taken to build model: 0.01 seconds
=== Evaluation on training set ===
Time taken to test model on training data: 0.02 seconds
=== Summary ===
Correctly Classified Instances 672 96.1373 %
Incorrectly Classified Instances 27 3.8627 %
Kappa statistic 0.9157
Mean absolute error 0.0389
Root mean squared error 0.1945
Relative absolute error 8.6129 %
Root relative squared error 40.9139 %
Total Number of Instances 699


Weka Explorer
Preprocess
Classify
Cluster
Associate
Select attributes
Visualize
Classifier
Choose
NaiveBayes
Test options
Use training set
Supplied test set
Set...
Cross-validation
Folds
10
Percentage split
%
66
More options...
(Nom) Class
Start
Stop
Result list (right-click for options)
13:42:13 - trees.J48
14:14:54 - trees.J48
14:18:11 - bayes.NaiveBayes
Classifier output
Time taken to test model on training data: 0.02 seconds
=== Summary ===
Correctly Classified Instances 672 96.1373 %
Incorrectly Classified Instances 27 3.8627 %
Kappa statistic 0.9157
Mean absolute error 0.0389
Root mean squared error 0.1945
Relative absolute error 8.6129 %
Root relative squared error 40.9139 %
Total Number of Instances 699
=== Detailed Accuracy By Class ===
TP Rate FP Rate Precision Recall F-Measure MCC ROC Area PRC Area Class
0.954 0.025 0.986 0.954 0.970 0.917 0.991 0.996 Benign
0.975 0.046 0.918 0.975 0.946 0.917 0.986 0.951 Malignant
Weighted Avg. 0.961 0.032 0.963 0.961 0.962 0.917 0.989 0.980
=== Confusion Matrix ===
a b <-- classified as
437 21 | a = Benign
6 235 | b = Malignant

COMPARISON

TEST MODE **– 10-FOLD CROSS VALIDATION**

SUMMARY

Comparing Attribute	**Decision Tree – J48**	**NaïveBayes**
Correctly Classified Instances	658 (94.1345%)	671 (95.9943%)
Incorrectly Classified Instances	41 (5.8655%)	28 (4.0057%)
Relative Absolute Error	16.1291 %	9.0284 %
Root relative squared Error	49.1262 %	41.9434 %
Mean Absolute Error	0.0729	0.0408
Root Mean squared Error	0.2335	0.1994
Kappa Statistics	0.8708	0.9127

Detailed Accuracy for Class Benign

Comparing Attribute	**Decision Tree – J48**	**NaïveBayes**
TP Rate	0.950	0.952
FP Rate	0.075	0.025
Precision	0.960	0.986
Recall	0.950	0.952
F-Measure	0.955	0.969
MCC	0.871	0.914
ROC Area	0.945	0.987
PRC Area	0.939	0.995

Detailed Accuracy for Class Malignant

Comparing Attribute	**Decision Tree – J48**	**NaïveBayes**
TP Rate	0.925	0.975
FP Rate	0.050	0.048
Precision	0.907	0.914
Recall	0.925	0.975
F-Measure	0.916	0.944
MCC	0.871	0.914
ROC Area	0.945	0.983
PRC Area	0.904	0.942

Detailed Accuracy by Class

1. TP Rate – The rate of true positives (instances correctly classified as a given class).
2. FP Rate – The rate of false positives (instances falsely classified as a given class).
3. Precision – Proportion of instances that are truly of a class divided by the total instances classified as that class
4. Recall – Proportion of instances classified as a given class divided by the actual total in that class (equivalent to TP rate)
5. F-Measure – A combined measure for precision and recall calculated as 2 * Precision * Recall / (Precision + Recall)
6. ROC Area – ROC Area, or area under the ROC curve, is my preferred measure.

TEST MODE – **TRAINING SET**

SUMMARY

Comparing Attribute	**Decision Tree – J48**	**NaïveBayes**
Correctly Classified Instances	685 (97.9971 %)	672 (96.1373%)
Incorrectly Classified Instances	14 (2.0029 %)	27 (3.8627%)
Relative Absolute Error	8.2022 %	8.6129 %
Root relative squared Error	28.6439 %	40.9139 %
Mean Absolute Error	0.0371	0.0389
Root Mean squared Error	0.1361	0.1945
Kappa Statistics	0.9558	0.9157

Detailed Accuracy for Class Benign

Comparing Attribute	**Decision Tree – J48**	**NaïveBayes**
TP Rate	0.983	0.954
FP Rate	0.025	0.025
Precision	0.987	0.986
Recall	0.983	0.954
F-Measure	0.985	0.970
MCC	0.956	0.917
ROC Area	0.989	0.991
PRC Area	0.991	0.996

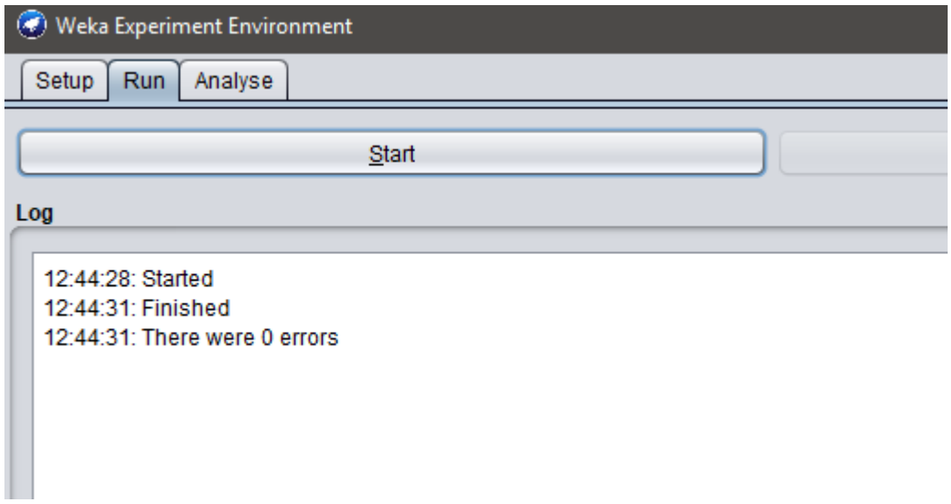
Detailed Accuracy for Class Malignant

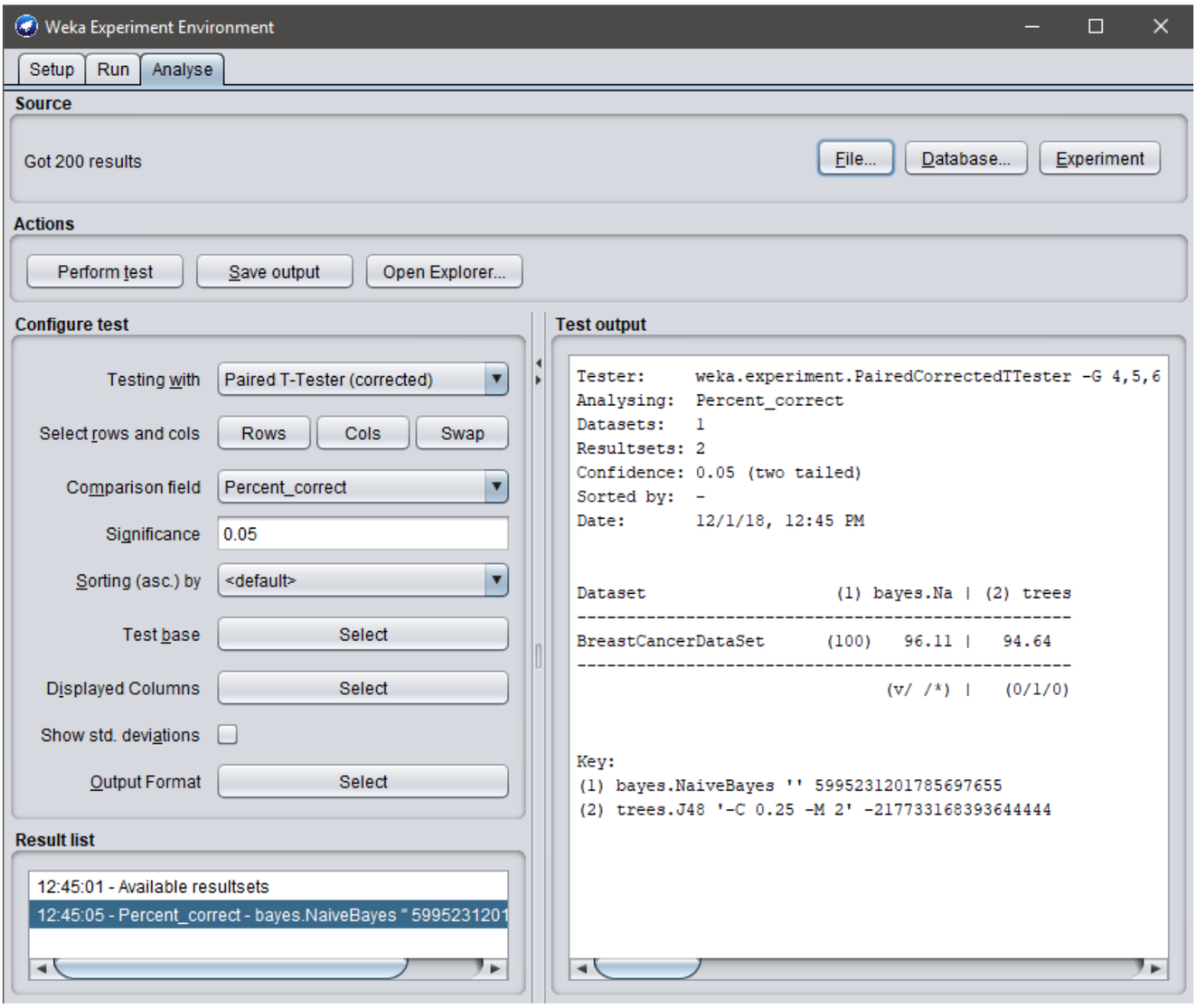
Comparing Attribute	**Decision Tree – J48**	**NaïveBayes**
TP Rate	0.975	0.975
FP Rate	0.017	0.046
Precision	0.967	0.918
Recall	0.975	0.975
F-Measure	0.971	0.946
MCC	0.956	0.917
ROC Area	0.989	0.986
PRC Area	0.970	0.951

Summary

- For Test Mode – 10-Fold Cross Validation, the accuracy for J48 is 94.1345 % where for NaïveBayes, it comes out to be 95.9943%. So, comparing the accuracy for two options, using NaïveBayes is a good option for the above Data Set.
- For Test Mode – Training Set, the accuracy for J48 comes out to be 97.9971 % as against 96.1373 % of NaïveBayes which indicates that for the given Data Set and Test mode as Training Set, using J48 would give better results.

Comparison using Experimenter





- There are 200 results loaded. This is because we had 2 algorithms that were each evaluated 100 times, 10-fold cross validation multiplied by 10 repeats.
- Configure Test Pane
  a. The type of statistical test can be selected in the "Testing with" option, by default this is set to "Paired T-Tester (corrected)".
  b. The significance level is set in the "Significance" parameter and is default to 0.05 (5%).
- Test output
  - By checking Test Output, we can clearly say that for test mode 10-Fold Cross Validation, NaïveBayes performs better than J48.

Hardware Specifications

- Operating System – Windows 10
- Hardware – Intel(R) Core (TM) i5-7200U @ 2.50GHz 8.00 GB ram 64-bit OS

References

1. Data Set - http://archive.ics.uci.edu/ml/machine-learning-databases/breast-cancer-wisconsin/

2. Tool Used – Weka (Link for Download - https://sourceforge.net/projects/weka/)

3. Source Code
   - After Installing Weka, search for the Jar File "**weka-src.jar**" in the installed directory.
   - Start Eclipse and make a new Project e.g. "Weka"
   - Import the Jar File in the newly created project in Eclipse.
   - Once the Import is successful, under the src folder, all the source code of Weka appears.
   - Select the Code as per requirements.

4. Other Links Referred
   - Weka Manual - https://www.cs.waikato.ac.nz/~ml/weka/documentation.html
   - Weka Output characteristics - https://en.wikipedia.org/wiki/Receiver_operating_characteristic#Basic_concept
   - More on Weka Output - https://stackoverflow.com/questions/2903933/how-to-interpret-weka-classification
   - Experimenter - https://machinelearningmastery.com/compare-performance-machine-learning-algorithms-weka/
   - J48 - https://en.wikipedia.org/wiki/C4.5_algorithm
   - NaïveBayes - https://en.wikipedia.org/wiki/Naive_Bayes_classifier