CHAPTER

16 Writing New Learning Schemes

Suppose you need to implement a special-purpose learning algorithm that is not included in Weka. Or suppose you are engaged in machine learning research and want to investigate a new learning scheme. Or suppose you just want to learn more about the inner workings of an induction algorithm by actually programming it yourself. This section uses a simple example to show how to make full use of Weka's class hierarchy when writing classifiers.

Weka includes the elementary learning schemes listed in Table 16.1, mainly for educational purposes. None take any scheme-specific command-line options. They are all useful for understanding the inner workings of a classifier. As an example, we describe the *weka.classifiers.trees.Id3* scheme, which implements the ID3 decision tree learner from Section 4.3 (page 99). Other schemes, such as clustering algorithms and association rule learners, are organized in a similar manner.

16.1 AN EXAMPLE CLASSIFIER

Figure 16.1 gives the source code of *weka.classifiers.trees.Id3*, which extends the *Classifier* class, as you can see from what is shown in the eight-page figure that follows the next page. Every classifier in Weka does so, whether it predicts a nominal class or a numeric one. It also implements two interfaces, *TechnicalInformationHandler* and *Sourcable*, which, respectively, allow the implementing class to provide bibliographical references for display in Weka's graphical user interface and a source code representation of its learned model.

The first method in *weka.classifiers.trees.Id3* is *globalInfo*(): We mention it here before moving on to the more interesting parts. It simply returns a string that is displayed in Weka's graphical user interface when this scheme is selected. Part of the string includes information generated by the second method, *getTechnicalInformation*(), which formats a bibliographic reference for the ID3 algorithm. The third method, *getCapabilities*(), returns information on the data characteristics that *Id3* can handle, namely nominal attributes and a nominal class—and the fact that it can deal with missing class values and data that contains no instances (although the latter does not produce a useful model!). Capabilities are described in Section 16.2.

Table 16.1 Simple Learning Schemes in Weka

Scheme	Description	Book Section
weka.classifiers.bayes.NaiveBayesSimple	Probabilistic learner	4.2
weka.classifiers.trees.Id3	Decision tree learner	4.3
weka.classifiers.rules.Prism	Rule learner	4.4
weka.classifiers.lazy.IB1	Instance-based learner	4.7

buildClassifier()

The *buildClassifier*() method constructs a classifier from a training dataset. In this case, it first checks the data's characteristics against *Id3*'s capabilities. Characteristics of the training data, such as numeric attributes or missing attribute values, will cause the *Capabilities* class to raise an exception, because the ID3 algorithm cannot handle these. It then makes a copy of the training set (to avoid changing the original data) and calls a method from *weka.core.Instances* to delete all instances with missing class values, because these instances are useless in the training process. Finally, it calls *makeTree*(), which actually builds the decision tree by recursively generating all subtrees attached to the root node.

makeTree()

The first step in *makeTree*() is to check whether the dataset is empty. If it is, a leaf is created by setting *m_Attribute* to null. The class value *m_ClassValue* assigned to this leaf is set to be missing, and the estimated probability for each of the dataset's classes in *m_Distribution* is initialized to 0. If training instances are present, *makeTree*() finds the attribute that yields the greatest information gain for them. It first creates a Java *enumeration* of the dataset's attributes. If the index of the class attribute is set—as it will be for this dataset—the class is automatically excluded from the enumeration.

Inside the enumeration, each attribute's information gain is computed by *computeInfoGain*() and stored in an array. We will return to this method later. The *index*() method from *weka.core.Attribute* returns the attribute's index in the dataset, which is used to index the array. Once the enumeration is complete, the attribute with the greatest information gain is stored in the instance variable *m_Attribute*. The *maxIndex*() method from *weka.core.Utils* returns the index of the greatest value in an array of integers or doubles. (If there is more than one element with maximum value, the first is returned.) The index of this attribute is passed to the *attribute*() method from *weka.core.Instances*, which returns the corresponding attribute.

You might wonder what happens to the array field corresponding to the class attribute. We need not worry about this because Java automatically initializes all elements in an array of numbers to 0, and the information gain is always greater

```
package weka.classifiers.trees;

import weka.classifiers.Classifier;
import weka.classifiers.Sourcable;
import weka.core.Attribute;
import weka.core.Capabilities;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.NoSupportForMissingValuesException;
import weka.core.RevisionUtils;
import weka.core.TechnicalInformation;
import weka.core.TechnicalInformationHandler;
import weka.core.Utils;
import weka.core.Capabilities.Capability;
import weka.core.TechnicalInformation.Field;
import weka.core.TechnicalInformation.Type;

import java.util.Enumeration;

/**
 * Class implementing an Id3 decision tree classifier.
 */
public class Id3 extends Classifier
  implements TechnicalInformationHandler, Sourcable {

  /** for serialization */
  static final long serialVersionUID = -2693678647096322561L;

  /** The node's successors. */
  private Id3[] m_Successors;

  /** Attribute used for splitting. */
  private Attribute m_Attribute;

  /** Class value if node is leaf. */
  private double m_ClassValue;

  /** Class distribution if node is leaf. */
  private double[] m_Distribution;

  /** Class attribute of dataset. */
  private Attribute m_ClassAttribute;

  /**
   * Returns a string describing the classifier.
   * @return a description suitable for the GUI.
   */
  public String globalInfo() {

    return "Class for constructing an unpruned decision tree "
      + "based on the ID3 algorithm. Can only deal with "
      + "nominal attributes. No missing values allowed. "
      + "Empty leaves may result in unclassified instances. "
      + "For more information see: \n\n"
      + getTechnicalInformation.toString();
    }
```

Continued

FIGURE 16.1

Source code for the ID3 decision tree learner.

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

  result = new TechnicalInformation(Type.ARTICLE);
  result.setValue(Field.AUTHOR, "R. Quinlan");
  result.setValue(Field.YEAR, "1986");
  result.setValue(Field.TITLE, "Induction of decision trees");
  result.setValue(Field.JOURNAL, "Machine Learning");
  result.setValue(Field.VOLUME, "1");
  result.setValue(Field.NUMBER, "1");
  result.setValue(Field.PAGES, "81-106");
  return result;
}

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

  // attributes
  result.enable(Capability.NOMINAL_ATTRIBUTES);

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

  // instances
  result.setMinimumNumberInstances(0);

  return result;
}

/**
 * Builds Id3 decision tree classifier.
 *
 * @param data the training data
 * @exception Exception if classifier can't be built successfully
 */
public void buildClassifier(Instances data) throws Exception {

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

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

FIGURE 16.1, cont'd

```
    makeTree(data);
  }

  /**
     * Method for building an Id3 tree.
     *
     * @param data the training data
     * @exception Exception if decision tree can't be built successfully
     */
    private void makeTree(Instances data) throws Exception {

      // Check if no instances have reached this node.
      if (data.numInstances() == 0) {
        m_Attribute = null;
        m_ClassValue = Instance.missingValue();
        m_Distribution = new double[data.numClasses()];
        return;
      }

      // Compute attribute with maximum information gain.
      double[] infoGains = new double[data.numAttributes()];
      Enumeration attEnum = data.enumerateAttributes();
      while (attEnum.hasMoreElements()) {
        Attribute att = (Attribute) attEnum.nextElement();
        infoGains[att.index()] = computeInfoGain(data, att);
      }
      m_Attribute = data.attribute(Utils.maxIndex(infoGains));

      // Make leaf if information gain is zero.
      // Otherwise create successors.
      if (Utils.eq(infoGains[m_Attribute.index()], 0)) {
        m_Attribute = null;
        m_Distribution = new double[data.numClasses()];
        Enumeration instEnum = data.enumerateInstances();
        while (instEnum.hasMoreElements()) {
          Instance inst = (Instance) instEnum.nextElement();
          m_Distribution[(int) inst.classValue()]++;
        }
        Utils.normalize(m_Distribution);
        m_ClassValue = Utils.maxIndex(m_Distribution);
        m_ClassAttribute = data.classAttribute();
      } else {
        Instances[] splitData = splitData(data, m_Attribute);
        m_Successors = new Id3[m_Attribute.numValues()];
        for (int j = 0; j < m_Attribute.numValues(); j++) {
          m_Successors[j] = new Id3();
          m_Successors[j].makeTree(splitData[j]);
        }
      }
    }

  /**
   * Classifies a given test instance using the decision tree.
   *
   * @param instance the instance to be classified
   * @return the classification
   * @throws NoSupportForMissingValuesException if instance has missing
     values
   */
  public double classifyInstance(Instance instance)
    throws NoSupportForMissingValuesException {
```

Continued

FIGURE 16.1, cont'd

```
    if (instance.hasMissingValue()) {
      throw new NoSupportForMissingValuesException("Id3: no missing values, "
                                                   + "please.");
    }
    if (m_Attribute == null) {
      return m_ClassValue;
    } else {
      return m_Successors[(int) instance.value(m_Attribute)].
        classifyInstance(instance);
    }
  }

  /**
   * Computes class distribution for instance using decision tree.
   *
   * @param instance the instance for which distribution is to be computed
   * @return the class distribution for the given instance
   * @throws NoSupportForMissingValuesException if instance
   * has missing values
   */
  public double[] distributionForInstance(Instance instance)
    throws NoSupportForMissingValuesException {

    if (instance.hasMissingValue()) {
      throw new NoSupportForMissingValuesException("Id3: no missing values, "
                                                   + "please.");
    }
    if (m_Attribute == null) {
      return m_Distribution;
    } else {
      return m_Successors[(int) instance.value(m_Attribute)].
        distributionForInstance(instance);
    }
  }

  /**
   * Prints the decision tree using the private toString method from below.
   *
   * @return a textual description of the classifier
   */
  public String toString() {

    if ((m_Distribution == null) && (m_Successors == null)) {
      return "Id3: No model built yet.";
    }
    return "Id3\n\n" + toString(0);
  }

  /**
   * Computes information gain for an attribute.
   *
   * @param data the data for which info gain is to be computed
   * @param att the attribute
   * @return the information gain for the given attribute and data
   * @throws Exception if computation fails
   */
  private double computeInfoGain(Instances data, Attribute att)
    throws Exception {
```

FIGURE 16.1, cont'd

```
  double infoGain = computeEntropy(data);
  Instances[] splitData = splitData(data, att);
  for (int j = 0; j < att.numValues(); j++) {
    if (splitData[j].numInstances() > 0) {
      infoGain -= ((double) splitData[j].numInstances() /
                   (double) data.numInstances()) *
        computeEntropy(splitData[j]);
    }
  }
  return infoGain;
}

/**
 * Computes the entropy of a dataset.
 *
 * @param data the data for which entropy is to be computed
 * @return the entropy of the data's class distribution
 * @throws Exception if computation fails
 */
private double computeEntropy(Instances data) throws Exception {

  double [] classCounts = new double[data.numClasses()];
  Enumeration instEnum = data.enumerateInstances();
  while (instEnum.hasMoreElements()) {
    Instance inst = (Instance) instEnum.nextElement();
    classCounts[(int) inst.classValue()]++;
  }
  double entropy = 0;
  for (int j = 0; j < data.numClasses(); j++) {
    if (classCounts[j] > 0) {
      entropy -= classCounts[j] * Utils.log2(classCounts[j]);
    }
  }
  entropy /= (double) data.numInstances();
  return entropy + Utils.log2(data.numInstances());
}

/**
 * Splits a dataset according to the values of a nominal attribute.
 *
 * @param data the data which is to be split
 * @param att the attribute to be used for splitting
 * @return the sets of instances produced by the split
 */
private Instances[] splitData(Instances data, Attribute att) {

  Instances[] splitData = new Instances[att.numValues()];
  for (int j = 0; j < att.numValues(); j++) {
    splitData[j] = new Instances(data, data.numInstances());
  }
  Enumeration instEnum = data.enumerateInstances();
  while (instEnum.hasMoreElements()) {
    Instance inst = (Instance) instEnum.nextElement();
    splitData[(int) inst.value(att)].add(inst);
  }
  for (int i = 0; i < splitData.length; i++) {
    splitData[i].compactify();
  }
  return splitData;
}
```

Continued

FIGURE 16.1, cont'd

```
/**
 * Outputs a tree at a certain level.
 *
 * @param level the level at which the tree is to be printed
 * @return the tree as string at the given level
 */
private String toString(int level) {

  StringBuffer text = new StringBuffer();

  if (m_Attribute == null) {
    if (Instance.isMissingValue(m_ClassValue)) {
      text.append(": null");
    } else {
      text.append(": " + m_ClassAttribute.value((int) m_ClassValue));
    }
  } else {
    for (int j = 0; j < m_Attribute.numValues(); j++) {
      text.append("\n");
      for (int i = 0; i < level; i++) {
        text.append("|  ");
      }
      text.append(m_Attribute.name() + " = " + m_Attribute.value(j));
      text.append(m_Successors[j].toString(level + 1));
    }
  }
  return text.toString();
}

/**
 * Adds this tree recursively to the buffer.
 *
 * @param id          the unqiue id for the method
 * @param buffer      the buffer to add the source code to
 * @return            the last ID being used
 * @throws Exception  if something goes wrong
 */
protected int toSource(int id, StringBuffer buffer) throws Exception {
  int                 result;
  int                 i;
  int                 newID;
  StringBuffer[]      subBuffers;

  buffer.append("\n");
  buffer.append("  protected static double node"
    + id + "(Object[] i) {\n");

  // leaf?
  if (m_Attribute == null) {
    result = id;
    if (Double.isNaN(m_ClassValue)) {
      buffer.append("    return Double.NaN;");
    } else {
      buffer.append("    return " + m_ClassValue + ";");
    }
    if (m_ClassAttribute != null) {
      buffer.append(" // " + m_ClassAttribute.value((int) m_ClassValue));
    }
    buffer.append("\n");
    buffer.append("  }\n");
```

FIGURE 16.1, cont'd

```
    } else {
      buffer.append("    checkMissing(i, "
        + m_Attribute.index() + ");\n\n");
      buffer.append("    // " + m_Attribute.name() + "\n");

      // subtree calls
      subBuffers = new StringBuffer[m_Attribute.numValues()];
      newID = id;
      for (i = 0; i < m_Attribute.numValues(); i++) {
        newID++;

        buffer.append("    ");
        if (i > 0) {
          buffer.append("else ");
        }
        buffer.append("if (((String) i[" + m_Attribute.index()
            + "]).equals(\"" + m_Attribute.value(i) + "\"))\n");
        buffer.append("      return node" + newID + "(i);\n");

        subBuffers[i] = new StringBuffer();
        newID = m_Successors[i].toSource(newID, subBuffers[i]);
      }
      buffer.append("    else\n");
      buffer.append("      throw new IllegalArgumentException
          (\"Value '\" + i[" + m_Attribute.index() + "]
          + \"' is not allowed!\");\n");
      buffer.append("  }\n");

      // output subtree code
      for (i = 0; i < m_Attribute.numValues(); i++) {
        buffer.append(subBuffers[i].toString());
      }
      subBuffers = null;
      result = newID;
    }

    return result;
  }

  /**
   * Returns a string that describes the classifier as source. The
   * classifier will be contained in a class with the given name (there
   * may be auxiliary classes),
   * and will contain a method with the signature:
   * <pre><code>
   * public static double classify(Object[] i);
   * </code></pre>
   * where the array <code>i</code> contains elements that are either
   * Double, String, with missing values represented as null. The
   * generated code is public domain and comes with no warranty. <br/>
   * Note: works only if class attribute is the last attribute in the
   * dataset.
   * @param className the name that should be given to the source class.
   * @return the object source described by a string
   * @throws Exception if the source can't be computed
   */
  public String toSource(String className) throws Exception {
    StringBuffer        result;
    int                 id;
```

Continued

FIGURE 16.1, cont'd

```
    result = new StringBuffer();

    result.append("class " + className + " {\n");
    result.append("  private static void checkMissing(Object[]
                  i, int index) {\n");
    result.append("    if (i[index] == null)\n");
    result.append("      throw new IllegalArgumentException (\"Null values "
      + "are not allowed!\");\n");
    result.append("  }\n\n");
    result.append("  public static double classify(Object[] i) {\n");
    id = 0;
    result.append("    return node" + id + "(i);\n");
    result.append("  }\n");
    toSource(id, result);
    result.append("}\n");

    return result.toString();
  }

  /**
   * Returns the revision string.
   *
   * @return  the revision
   */
  public String getRevision() {
    return RevisionUtils.extract("$Revision: 6404 $");
  }

  /**
   * Main method.
   *
   * @param args the options for the classifier
   */
  public static void main(String[] args) {
    runClassifier(new Id3(), args);
  }
}
```

FIGURE 16.1, cont'd

than or equal to 0. If the maximum information gain is 0, *makeTree*() creates a leaf. In that case, *m_Attribute* is set to null, and *makeTree*() computes both the distribution of class probabilities and the class with the greatest probability. (The *normalize*() method from *weka.core.Utils* normalizes an array of doubles to sum to 1.)

When it makes a leaf with a class value assigned to it, *makeTree*() stores the class attribute in *m_ClassAttribute*. This is because the method that outputs the decision tree needs to access this to print the class label.

If an attribute with nonzero information gain is found, *makeTree*() splits the dataset according to the attribute's values and recursively builds subtrees for each of the new datasets. To make the split it calls the method *splitData*(). This creates as many empty datasets as there are attribute values, stores them in an array (setting the initial capacity of each dataset to the number of instances in the original dataset), and then iterates through all instances in the original dataset and allocates them to

the new dataset that corresponds to the attribute's value. It then reduces memory requirements by compacting the *Instances* objects. Returning to *makeTree*(), the resulting array of datasets is used for building subtrees. The method creates an array of *Id3* objects, one for each attribute value, and calls *makeTree*() on each one by passing it the corresponding dataset.

computeInfoGain()

Returning to *computeInfoGain*(), the information gain associated with an attribute and a dataset is calculated using a straightforward implementation of the formula in Section 4.3 (page 104). First, the entropy of the dataset is computed. Then, *splitData*() is used to divide it into subsets, and *computeEntropy*() is called on each one. Finally, the difference between the former entropy and the weighted sum of the latter ones—the information gain—is returned. The method *computeEntropy*() uses the *log2*() method from *weka.core.Utils* to obtain the logarithm (to base 2) of a number.

classifyInstance()

Having seen how *Id3* constructs a decision tree, we now examine how it uses the tree structure to predict class values and probabilities. Every classifier must implement the *classifyInstance*() method or the *distributionForInstance*() method (or both). The *Classifier* superclass contains default implementations for both methods. The default implementation of *classifyInstance*() calls *distributionForInstance*(). If the class is nominal, it predicts the class with maximum probability, or a missing value if all probabilities returned by *distributionForInstance*() are 0. If the class is numeric, *distributionForInstance*() must return a single-element array that holds the numeric prediction, and this is what *classifyInstance*() extracts and returns. Conversely, the default implementation of *distributionForInstance*() wraps the prediction obtained from *classifyInstance*() into a single-element array. If the class is nominal, *distributionForInstance*() assigns a probability of 1 to the class predicted by *classifyInstance*() and a probability of 0 to the others. If *classifyInstance*() returns a missing value, all probabilities are set to 0. To give you a better feeling for just what these methods do, the *weka.classifiers.trees.Id3* class overrides them both.

Let's look first at *classifyInstance*(), which predicts a class value for a given instance. As mentioned in the previous section, nominal class values, like nominal attribute values, are coded and stored in *double* variables, representing the index of the value's name in the attribute declaration. This is used in favor of a more elegant object-oriented approach to increase the speed of execution. In the implementation of ID3, *classifyInstance*() first checks whether there are missing attribute values in the instance to be classified; if so, it throws an exception. The class attribute is skipped in this check, or the classifier would not be able to make predictions for new data, where the class is unknown. Otherwise, it descends the tree recursively, guided by the instance's attribute values, until a leaf is reached. Then it returns the class value *m_ClassValue* stored at the leaf. Note that this might be a missing value,

in which case the instance is left unclassified. The method *distributionForInstance*() works in exactly the same way; that is, it returns the probability distribution stored in *m_Distribution*.

Most machine learning models, and in particular decision trees, serve as a more or less comprehensible explanation of the structure found in the data. Accordingly, each of Weka's classifiers, like many other Java objects, implements a *toString*() method that produces a textual representation of itself in the form of a *String* variable. *Id3*'s *toString*() method outputs a decision tree in roughly the same format as *J48* (see Figure 11.5). It recursively prints the tree structure into a *String* variable by accessing the attribute information stored at the nodes. To obtain each attribute's name and values, it uses the *name*() and *value*() methods from *weka.core.Attribute*. Empty leaves without a class value are indicated by the string *null*.

toSource()

weka.classifiers.trees.Id3 implements the *Sourcable* interface. Classifiers that implement this interface can produce a source code representation of the learned model, which can be output on the command line by using the *–z* option (see Section 14.3, Table 14.1, page 527). Weka produces a Java class (the name of which is provided by the *–z* option) that can be used to make predictions independently of the Weka libraries. Also output is a class called *WekaWrapper* that extends *Classifier* and uses the named class to make predictions. This class can be used for testing the source code representation of the model within the Weka framework—that is, it can be run from the command line or used in the Explorer interface. Because the actual classifier is hard-coded, the *buildClassifier*() method of the *WekaWrapper* does nothing more than check the data against the capabilities.

Figure 16.2 shows the source code produced by *weka.classifiers.trees.Id3* when run on the nominal-attribute version of the weather data. The name *Id3Weather* was provided with the *–z* option, and a class of this name is shown in Figure 16.2(a). All its methods are static, meaning that they can be used without having to instantiate an *Id3Weather* object. The first method is called *classify* and takes as argument an array of Objects that give the attribute values for the instance to be classified. Each node in the ID3 tree that has been learned is represented in the source code by a static method. The *classify* method passes the instance to be classified to the method corresponding to the root of the tree—*node0*. The *node0* method corresponds to a test on the *outlook* attribute. Because it is not a leaf, it calls a node representing one of the subtrees—which one depends on the value of *outlook* in the instance being classified. Processing of a test instance continues in this fashion until a method corresponding to a leaf node is called, at which point a classification is returned.

Figure 16.2(b) shows the *WekaWrapper* class that uses *Id3Weather*. Its *classifyInstance* method constructs an array of objects to pass to the *classify* method in *Id3Weather*. The attribute values of the test instance are copied into the array and mapped to either String objects (for nominal values) or Double objects (for numeric

```
package weka.classifiers;

class Id3Weather {
  private static void checkMissing(Object[] i, int index) {
    if (i[index] == null)
      throw new IllegalArgumentException("Null values are not allowed!");
  }

  public static double classify(Object[] i) {
    return node0(i);
  }

  protected static double node0(Object[] i) {
    checkMissing(i, 0);

    // outlook
    if (((String) i[0]).equals("sunny"))
      return node1(i);
    else if (((String) i[0]).equals("overcast"))
      return node4(i);
    else if (((String) i[0]).equals("rainy"))
      return node5(i);
    else
      throw new IllegalArgumentException("Value '" + i[0]
                                         + "' is not allowed!");
  }

  protected static double node1(Object[] i) {
    checkMissing(i, 2);

    // humidity
    if (((String) i[2]).equals("high"))
      return node2(i);
    else if (((String) i[2]).equals("normal"))
      return node3(i);
    else
      throw new IllegalArgumentException("Value '" + i[2]
                                         + "' is not allowed!");
  }

  protected static double node2(Object[] i) {
    return 1.0; // no
  }

  protected static double node3(Object[] i) {
    return 0.0; // yes
  }

  protected static double node4(Object[] i) {
    return 0.0; // yes
  }

  protected static double node5(Object[] i) {
    checkMissing(i, 3);

    // windy
    if (((String) i[3]).equals("TRUE"))
      return node6(i);
    else if (((String) i[3]).equals("FALSE"))
      return node7(i);
    else
      throw new IllegalArgumentException("Value '" + i[3]
                                         + "' is not allowed!");
  }

  protected static double node6(Object[] i) {
    return 1.0; // no
  }

  protected static double node7(Object[] i) {
    return 0.0; // yes
  }
}
```

(a)

FIGURE 16.2

Source code produced by *weka.classifiers.trees.Id3* for the weather data: (a) *Id3Weather* class and (b) *WekaWrapper* class.

```
package weka.classifiers;

import weka.core.Attribute;
import weka.core.Capabilities;
import weka.core.Capabilities.Capability;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.RevisionUtils;
import weka.classifiers.Classifier;

public class WekaWrapper
  extends Classifier {

  /**
   * Returns only the toString() method.
   *
   * @return a string describing the classifier
   */
  public String globalInfo() {
    return toString();
  }

  /**
   * Returns the capabilities of this classifier.
   *
   * @return the capabilities
   */
  public Capabilities getCapabilities() {
    weka.core.Capabilities result = new weka.core.Capabilities(this);

    result.enable(weka.core.Capabilities.Capability.NOMINAL_ATTRIBUTES);
    result.enable(weka.core.Capabilities.Capability.NOMINAL_CLASS);
    result.enable(weka.core.Capabilities.Capability.MISSING_CLASS_VALUES);

    result.setMinimumNumberInstances(0);

    return result;
  }

  /**
   * only checks the data against its capabilities.
   *
   * @param i the training data
   */
  public void buildClassifier(Instances i) throws Exception {
    // can classifier handle the data?
    getCapabilities().testWithFail(i);
  }

  /**
   * Classifies the given instance.
   *
   * @param i the instance to classify
   * @return the classification result
   */
  public double classifyInstance(Instance i) throws Exception {
    Object[] s = new Object[i.numAttributes()];
```

(b)

FIGURE 16.2, cont'd

```
      for (int j = 0; j < s.length; j++) {
        if (!i.isMissing(j)) {
          if (i.attribute(j).isNominal())
            s[j] = new String(i.stringValue(j));
          else if (i.attribute(j).isNumeric())
            s[j] = new Double(i.value(j));
        }
      }

      // set class value to missing
      s[i.classIndex()] = null;

      return Id3Weather.classify(s);
    }

    /**
     * Returns the revision string.
     *
     * @return        the revision
     */
    public String getRevision() {
      return RevisionUtils.extract("1.0");
    }

    /**
     * Returns only the classnames and what classifier it is based on.
     *
     * @return a short description
     */
    public String toString() {
      return "Auto-generated classifier wrapper, based on
      weka.classifiers.trees.Id3 (generated with Weka 3.6.2).\n" +
        this.getClass().getName() + "/Id3Weather";
    }

    /**
     * Runs the classfier from commandline.
     *
     * @param args the commandline arguments
     */
    public static void main(String args[]) {
      runClassifier(new WekaWrapper(), args);
    }
  }
```

FIGURE 16.2, cont'd

values). If an attribute value is missing in the test instance, its corresponding entry in the array of Objects is set to *null*.

main()

Apart from *getRevision*(), which is optional and simply returns a version identifier, the only method in *Id3* that hasn't been described is *main*(), which is called whenever the class is executed from the command line. As you can see, it's simple: It calls the method *runClassifier*() from its superclass (*Classifier*) with a new instance

of *Id3* and the given command-line options. *runClassifier*() tells Weka's *Evaluation* class to evaluate the supplied classifier with the given command-line options and prints the resulting string. The one-line expression in *runClassifier*() that does this calls the *evaluation*() method in Weka's evaluation class and is enclosed in a *try-catch* statement, which catches the various exceptions that can be thrown by Weka's routines or other Java methods.

The *evaluation*() method in *weka.classifiers.Evaluation* interprets the generic scheme-independent command-line options described in Section 14.3 (page 526) and acts appropriately. For example, it takes the *–t* option, which gives the name of the training file, and loads the corresponding dataset. If there is no test file, it performs a cross-validation by creating a classifier object and by repeatedly calling *buildClassifier*() and *classifyInstance*() or *distributionForInstance*() on different subsets of the training data. Unless the user suppresses output of the model by setting the corresponding command-line option, it also calls the *toString*() method to output the model built from the full training dataset.

What happens if the scheme needs to interpret a specific option such as a pruning parameter? This is accomplished using the *OptionHandler* interface in *weka.core*. A classifier that implements this interface contains three methods—*listOptions*(), *setOptions*(), and *getOptions*()—which can be used to list all the classifier's scheme-specific options, to set some of them, and to get the options that are currently set. The *evaluation*() method in *Evaluation* automatically calls these methods if the classifier implements the *OptionHandler* interface. Once the scheme-independent options have been processed, it calls *setOptions*() to process the remaining options before using *buildClassifier*() to generate a new classifier. When it outputs the classifier, it uses *getOptions*() to output a list of the options that are currently set. For a simple example of how to implement these methods, look at the source code for *weka.classifiers.rules.OneR*.

OptionHandler makes it possible to set options from the command line. To set them from within the graphical user interfaces, Weka uses the JavaBeans framework. All that is required are *set…*() and *get…*() methods for every parameter used by the class. For example, the methods *setPruningParameter*() and *getPruningParameter*() would be needed for a pruning parameter. There should also be a *pruningParameterTipText*() method that returns a description of the parameter for the graphical user interface. Again, see *weka.classifiers.rules.OneR* for an example.

Some classifiers can be incrementally updated as new training instances arrive; they don't have to process all the data in one batch. In Weka, incremental classifiers implement the *UpdateableClassifier* interface in *weka.classifiers*. This interface declares only one method, namely *updateClassifier*(), which takes a single training instance as its argument. For an example of how to use this interface, look at the source code for *weka.classifiers.lazy.IBk*.

If a classifier is able to make use of instance weights, it should implement the *WeightedInstancesHandler*() interface from *weka.core*. Then other algorithms, such as those for boosting, can make use of this property.

In *weka.core* there are many other useful interfaces for classifiers—for example, interfaces for classifiers that are *randomizable*, *summarizable*, *drawable*, and *graphable*. For more information on these and other interfaces, look at the Javadoc for the classes in *weka.core*.

16.2 CONVENTIONS FOR IMPLEMENTING CLASSIFIERS

There are some conventions that you must obey when implementing classifiers in Weka. If you do not, things will go awry. For example, Weka's evaluation module might not compute the classifier's statistics properly when evaluating it. The *CheckClassifier* class can be used to check the basic behavior of a classifier, although it cannot catch all problems.

The first convention has already been mentioned: Each time a classifier's *buildClassifier*() method is called, it must reset the model. The *CheckClassifier* class performs tests to ensure that this is the case. When *buildClassifier*() is called on a dataset, the same result must always be obtained regardless of how often the classifier has previously been applied to the same or other datasets. However, *buildClassifier*() must not reset instance variables that correspond to scheme-specific options because these settings must persist through multiple calls of *buildClassifier*(). Also, calling *buildClassifier*() must never change the input data.

Two other conventions have also been mentioned. One is that when a classifier cannot make a prediction, its *classifyInstance*() method must return *Instance.missingValue*() and its *distributionForInstance*() method must return probabilities of zero for all classes. The ID3 implementation shown earlier in Figure 16.1 does this. Another convention is that with classifiers for numeric prediction, *classifyInstance*() returns the numeric value that the classifier predicts. Some classifiers, however, are able to predict nominal classes and their class probabilities, as well as numeric class values—*weka.classifiers.lazy.IBk* is an example. These implement the *distributionForInstance*() method, and if the class is numeric it returns an array of size 1, whose only element contains the predicted numeric value.

Another convention—not absolutely essential but useful nonetheless—is that every classifier implements a *toString*() method that outputs a textual description of itself.

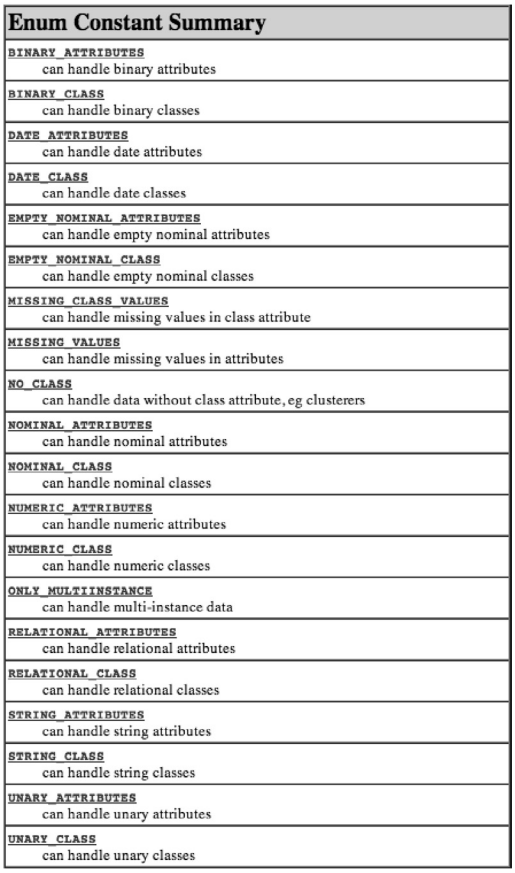
Capabilities

We close this chapter with a closer look at the idea of "capabilities." As mentioned earlier, these allow a learning scheme to indicate what data characteristics it can handle. This information is displayed in the object editor when the user presses the *Capabilities* button; it also serves to disable the application of a scheme in the Explorer interface when the current data does not match its stated capabilities.

To ease the programming burden for those new to developing with Weka, the superclasses of the major types of learning scheme—*Classifier*, *AbstractClusterer*,

and *AbstractAssociator*—disable all capabilities constraints by default. This allows programmers to focus on the main task of implementing the learning functionality without having to bother with capabilities. However, once satisfied that the scheme is working correctly, the programmer should specify capabilities that reflect the scheme's ability to handle various data characteristics by overriding the superclass's *getCapabilities*() method. This method returns a *weka.core.Capabilities* object that encapsulates the characteristics that the scheme can handle.

In Figure 16.1, *Id3*'s *getCapabilities*() method first obtains a *Capabilities* object by calling *super.getCapabilities*(). This returns a *Capabilities* object without any constraints. The best way to proceed is to call the *disableAll*() method on the *Capabilities* object and then enable the relevant characteristics—those that the scheme



Enum Constant Summary
BINARY_ATTRIBUTES can handle binary attributes
BINARY_CLASS can handle binary classes
DATE_ATTRIBUTES can handle date attributes
DATE_CLASS can handle date classes
EMPTY_NOMINAL_ATTRIBUTES can handle empty nominal attributes
EMPTY_NOMINAL_CLASS can handle empty nominal classes
MISSING_CLASS_VALUES can handle missing values in class attribute
MISSING_VALUES can handle missing values in attributes
NO_CLASS can handle data without class attribute, eg clusterers
NOMINAL_ATTRIBUTES can handle nominal attributes
NOMINAL_CLASS can handle nominal classes
NUMERIC_ATTRIBUTES can handle numeric attributes
NUMERIC_CLASS can handle numeric classes
ONLY_MULTIINSTANCE can handle multi-instance data
RELATIONAL_ATTRIBUTES can handle relational attributes
RELATIONAL_CLASS can handle relational classes
STRING_ATTRIBUTES can handle string attributes
STRING_CLASS can handle string classes
UNARY_ATTRIBUTES can handle unary attributes
UNARY_CLASS can handle unary classes

FIGURE 16.3

Javadoc for the *Capability* enumeration.

can handle. *Id3* does just this, enabling the ability to handle nominal attributes, a nominal class attribute, and missing class values. It also specifies that a minimum of zero training instances are required. For the most part, individual capabilities are turned on or off by calling the *enable*() or *disable*() method of the *Capabilities* object. These methods take constants that are defined in the enumeration shown in Figure 16.3, which is part of the *Capabilities* class.