CHAPTER

15 Embedded Machine Learning

When invoking learning schemes from the graphical user interfaces or the command-line interface, there is no need to know anything about programming in Java. In this section we show how to access these algorithms from your own code. In doing so, the advantages of using an object-oriented programming language will become clear. From now on, we assume that you have at least some rudimentary knowledge of Java. In most practical applications of data mining, the learning component is an integrated part of a far larger software environment. If the environment is written in Java, you can use Weka to solve the learning problem without writing any machine learning code yourself.

15.1 A SIMPLE DATA MINING APPLICATION

We present a simple data mining application for learning a model that classifies text files into two categories: *hit* and *miss*. The application works for arbitrary documents, which are referred to as *messages*. The implementation uses the *StringToWordVector* filter mentioned in Section 11.3 (page 439) to convert messages into attribute vectors in the manner described in Section 7.3 (page 328). We assume that the program is called every time a new file is to be processed. If the user provides a class label for the file, the system uses it for training; if not, it classifies it. The decision tree classifier *J48* is used to do the work.

Figure 15.1 shows the source code for the application program, implemented in a class called *MessageClassifier*. The command-line arguments that the *main*() method accepts are the name of a text file (given by *–m*), the name of a file holding an object of class *MessageClassifier* (*–t*), and, optionally, the classification of the message in the file (*–c*). If the user provides a classification, the message will be converted into an example for training; if not, the *MessageClassifier* object will be used to classify it as *hit* or *miss*.

The *main*() method reads the message into a Java *StringBuffer* and checks whether the user has provided a classification for it. Then it reads a *MessageClassifier* object from the file given by *–t*, and creates a new object of class *MessageClassifier* if this file does not exist. In either case the resulting object is called *messageCl*. After checking for illegal command-line options, the program calls the

```
/**
 * Java program for classifying text messages into two classes.
 */

import weka.classifiers.Classifier;
import weka.classifiers.trees.J48;
import weka.core.Attribute;
import weka.core.FastVector;
import weka.core.Instance;
import weka.core.Instances;
import weka.core.SerializationHelper;
import weka.core.Utils;
import weka.filters.Filter;
import weka.filters.unsupervised.attribute.StringToWordVector;

import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.Serializable;

public class MessageClassifier implements Serializable {

/** The training data gathered so far. */
  private Instances m_Data = null;

  /** The filter used to generate the word counts. */
  private StringToWordVector m_Filter = new StringToWordVector();

  /** The actual classifier. */
  private Classifier m_Classifier = new J48();

  /** Whether the model is up to date. */
  private boolean m_UpToDate;

  /** For serialization. */
  private static final long serialVersionUID = -123455813150452885L;

  /**
   * Constructs empty training dataset.
   */
  public MessageClassifier() {
    String nameOfDataset = "MessageClassificationProblem";

    // Create vector of attributes.
    FastVector attributes = new FastVector(2);

    // Add attribute for holding messages.
    attributes.addElement(new Attribute("Message", (FastVector) null));

    // Add class attribute.
    FastVector classValues = new FastVector(2);
    classValues.addElement("miss");
    classValues.addElement("hit");
    attributes.addElement(new Attribute("Class", classValues));
```

FIGURE 15.1

Source code for the message classifier *main*().

```
    // Create dataset with initial capacity of 100, and set index
       of class.
    m_Data = new Instances(nameOfDataset, attributes, 100);
    m_Data.setClassIndex(m_Data.numAttributes() - 1);
  }

  /**
   * Updates model using the given training message.
   *
   * @param message            the message content
   * @param classValue         the class label
   */
  public void updateData(String message, String classValue) {
    // Make message into instance.
    Instance instance = makeInstance(message, m_Data);

    // Set class value for instance.
    instance.setClassValue(classValue);

    // Add instance to training data.
    m_Data.add(instance);
    m_UpToDate = false;
  }

  /**
   * Classifies a given message.
   *
   * @param message            the message content
   * @throws Exception         if classification fails
   */
  public void classifyMessage(String message) throws Exception {

    // Check whether classifier has been built.
    if (m_Data.numInstances() == 0) {
      throw new Exception("No classifier available.");
    }

    // Check whether classifier and filter are up to date.
    if (!m_UpToDate) {
      // Initialize filter and tell it about the input format.
      m_Filter.setInputFormat(m_Data);

      // Generate word counts from the training data.
      Instances filteredData  = Filter.useFilter(m_Data, m_Filter);

      // Rebuild classifier.
      m_Classifier.buildClassifier(filteredData);

      m_UpToDate = true;
    }

    // Make separate little test set so that message
    // does not get added to string attribute in m_Data.
    Instances testset = m_Data.stringFreeStructure();

    // Make message into test instance.
    Instance instance = makeInstance(message, testset);
```

Continued

FIGURE 15.1, cont'd

```
  // Filter instance.
  m_Filter.input(instance);
  Instance filteredInstance = m_Filter.output();

  // Get index of predicted class value.
  double predicted = m_Classifier.classifyInstance(filteredInstance);

  // Output class value.
  System.err.println("Message classified as : " +
         m_Data.classAttribute().value((int) predicted));
}

/**
 * Method that converts a text message into an instance.
 *
 * @param text      the message content to convert
 * @param data      the header information
 * @return                  the generated Instance
 */
private Instance makeInstance(String text, Instances data) {

  // Create instance of length two.
  Instance instance = new Instance(2);

  // Set value for message attribute
  Attribute messageAtt = data.attribute("Message");
  instance.setValue(messageAtt, messageAtt.addStringValue(text));

  // Give instance access to attribute information from the dataset.
  instance.setDataset(data);

  return instance;
}

/**
 * Main method. The following parameters are recognized:
 *
 *   -m messagefile
 *      Points to the file containing the message to classify or use
 *      for updating the model.
 *   -c classlabel
 *      The class label of the message if model is to be updated.
 *      Omit for classification of a message.
 *   -t modelfile
 *      The file containing the model. If it doesn't exist, it will
 *      be created automatically.
 *
 * @param args      the commandline options
 */
public static void main(String[] args) {

  try {
```

FIGURE 15.1, cont'd

```
      // Read message file into string.
      String messageName = Utils.getOption('m', args);
      if (messageName.length() == 0) {
        throw new Exception("Must provide name of message
          + file ('-m <file>').");

      }

      FileReader m = new FileReader(messageName);
      StringBuffer message = new StringBuffer();
      int l;
      while ((l = m.read()) != -1) {
        message.append((char) l);
      }
      m.close();

      // Check if class value is given.
      String classValue = Utils.getOption('c', args);

      // If model file exists, read it, otherwise create new one.
      String modelName = Utils.getOption('t', args);
      if (modelName.length() == 0) {
        throw new Exception("Must provide name of model
          + file ('-t <file>').");
      }
      MessageClassifier messageCl;
      try {
        messageCl =
        (MessageClassifier) SerializationHelper. read(modelName);
      } catch (FileNotFoundException e) {
        messageCl = new MessageClassifier();
      }

      // Check if there are any options left
      Utils.checkForRemainingOptions(args);

      // Process message.
      if (classValue.length() != 0) {
        messageCl.updateData(message.toString(), classValue);
      } else {
        messageCl.classifyMessage(message.toString());
      }

      // Save message classifier object only if it was updated.
      if (classValue.length() != 0) {
        SerializationHelper.write(modelName, messageCl);
      }
    } catch (Exception e) {
      e.printStackTrace();
  }
}
```

Continued

FIGURE 15.1, cont'd

method *updateData()* message to update the training data stored in *messageCl* if a classification has been provided; otherwise, it calls *classifyMessage*() to classify it. Finally, the *messageCl* object is saved back into the file because it may have changed. In the following, we first describe how a new *MessageClassifier* object is created by the constructor *MessageClassifier*() and then explain how the two methods *updateData*() and *classifyMessage*() work.

MessageClassifier()

Each time a new *MessageClassifier* is created, objects for holding the filter and classifier are generated automatically. The only nontrivial part of the process is creating a dataset, which is done by the constructor *MessageClassifier*(). First, the dataset's name is stored as a string. Then an *Attribute* object is created for each attribute, one to hold the string corresponding to a text message and the other for its class. These objects are stored in a dynamic array of type *FastVector*. (*FastVector* is Weka's own implementation of the standard Java *Vector* class and is used throughout Weka for historical reasons.)

Attributes are created by invoking one of the constructors in the class *Attribute*. This class has a constructor that takes one parameter—the attribute's name—and creates a numeric attribute. However, the constructor we use here takes two parameters: the attribute's name and a reference to a *FastVector*. If this reference is null, as in the first application of this constructor in our program, Weka creates an attribute of type *string*. Otherwise, a nominal attribute is created. In that case, it is assumed that the *FastVector* holds the attribute values as strings. This is how we create a class attribute with two values, *hit* and *miss*—by passing the attribute's name (*class*) and its values, stored in a *FastVector*, to *Attribute*().

To create a dataset from this attribute information, *MessageClassifier*() must create an object of the class *Instances* from the *core* package. The constructor of *Instances* used by *MessageClassifier*() takes three arguments: the dataset's name, a *FastVector* containing the attributes, and an integer indicating the dataset's initial capacity. We set the initial capacity to 100; it is expanded automatically if more instances are added. After constructing the dataset, *MessageClassifier*() sets the index of the class attribute to be the index of the last attribute.

updateData()

Now that you know how to create an empty dataset, consider how the *MessageClassifier* object actually incorporates a new training message. The method *updateData*() does this job. It first converts the given message into a training instance by calling *makeInstance*(), which begins by creating an object of class *Instance* that corresponds to an instance with two attributes. The constructor of the *Instance* object sets all the instance's values to be *missing* and its weight to 1. The next step in *makeInstance*() is to set the value of the string attribute holding the text of the message. This is done by applying the *setValue*() method of the *Instance* object, providing it with the

attribute whose value needs to be changed, and a second parameter that corresponds to the new value's index in the definition of the string attribute. This index is returned by the *addStringValue*() method, which adds the message text as a new value to the string attribute and returns the position of this new value in the definition of the string attribute.

Internally, an *Instance* stores all attribute values as double-precision floating-point numbers regardless of the type of the corresponding attribute. In the case of nominal and string attributes this is done by storing the index of the corresponding attribute value in the definition of the attribute. For example, the first value of a nominal attribute is represented by 0.0, the second by 1.0, and so on. The same method is used for string attributes: *addStringValue*() returns the index corresponding to the value that is added to the definition of the attribute.

Once the value for the string attribute has been set, *makeInstance*() gives the newly created instance access to the data's attribute information by passing it a reference to the dataset. In Weka, an *Instance* object does not store the type of each attribute explicitly; instead, it stores a reference to a dataset with the corresponding attribute information.

Returning to *updateData*(), once the new instance has been returned from *makeInstance*(), its class value is set and it is added to the training data. We also initialize *m_UpToDate*, a flag indicating that the training data has changed and the predictive model is therefore not up to date.

classifyMessage()

Now let's examine how *MessageClassifier* processes a message of which the class label is unknown. The *classifyMessage*() method first checks whether a classifier has been built by determining whether any training instances are available. It then checks whether the classifier is up to date. If not (because the training data has changed), the classifier must be rebuilt. However, before doing so, the data must be converted into a format appropriate for learning using the *StringToWordVector* filter. First, we tell the filter the format of the input data by passing it a reference to the input dataset using *setInputFormat*(). Every time this method is called, the filter is initialized—that is, all its internal settings are reset. In the next step, the data is transformed by *useFilter*(). This generic method from the *Filter* class applies a filter to a dataset. In this case, because *StringToWordVector* has just been initialized, it computes a dictionary from the training dataset and then uses it to form a word vector. After returning from *useFilter*(), all the filter's internal settings are fixed until it is initialized by another call of *inputFormat*(). This makes it possible to filter a test instance without updating the filter's internal settings (in this case, the dictionary).

Once the data has been filtered, the program rebuilds the classifier—in our case a *J48* decision tree—by passing the training data to its *buildClassifier*() method. Then it sets *m_UpToDate* to *true*. It is an important convention in Weka that the *buildClassifier*() method completely initializes the model's internal settings before

generating a new classifier. Thus, we do not need to construct a new *J48* object before we call *buildClassifier*().

Having ensured that the model stored in *m_Classifier* is current, we proceed to classify the message. Before *makeInstance*() is called to create an *Instance* object from it, a new *Instances* object is created to hold the new instance and it is passed as an argument to *makeInstance*(). This is done so that *makeInstance*() does not add the text of the message to the definition of the string attribute in *m_Data*. Otherwise, the size of the *m_Data* object would grow every time a new message was classified, which is clearly not desirable—it should only grow when training instances are added. Thus, a temporary *Instances* object is created and discarded once the instance has been processed. This object is obtained using the method *stringFreeStructure*(), which returns a copy of *m_Data* with an empty string attribute. Only then is *makeInstance*() called to create the new instance.

The test instance must also be processed by the *StringToWordVector* filter before being classified. This is easy: The *input*() method enters the instance into the filter object, and the transformed instance is obtained by calling *output*(). Then a prediction is produced by passing the instance to the classifier's *classifyInstance*() method. As you can see, the prediction is coded as a *double* value. This allows Weka's evaluation module to treat models for categorical and numeric prediction similarly. In the case of categorical prediction, as in this example, the *double* variable holds the index of the predicted class value. To output the string corresponding to this class value, the program calls the *value*() method of the dataset's class attribute.

There is at least one way in which our implementation could be improved. The classifier and the *StringToWordVector* filter could be combined using the *FilteredClassifier* metalearner described in Section 11.5 (page 443). This classifier would then be able to deal with string attributes directly, without explicitly calling the filter to transform the data. We didn't do this because we wanted to demonstrate how filters can be used programmatically.