Intelligent Data Analysis Computer Laboratory
Practice on classification and clustering software

26th of January – 3rd of February 2012

Arnaud Quirin <arnaud.quirin@softcomputing.es>
http://aquirin.ovh.org/mastersoft/

Part II : How to add a new classifier to Weka?

Thanks to its framework, it is possible to extend Weka in a very easy way. This will allow you to add new capabilities inside Weka, and make it performing your own tasks, with the worthy benefit that you can use the already implemented functionalities: current classifiers, filters and data visualization capabilities will complement your own extensions.

With Weka, it is possible to add two kinds of new extensions: new classifiers or new filters. The goal of this tutorial is to let you write your own classifier. Of course programming in Java is a prerequisite, but the tutorial is clear enough to be followed even if you know another object-oriented language, such as C++.

This tutorial will guide you through all the steps you need to do to write and execute a new classifier that are:

- the preparation of the compilation environment,
- the installation of the Weka source code,
- the writing of a new classifier,
- and its execution on a basic dataset.

For the sake of simplicity, we will write a classifier that classify any instance of a dataset in the same class. This class will be the most frequent one found in this dataset, so we should expect at least some kind of accuracy.

Prerequisites:

- An environment to compile and execute Java code, for instance Eclipse: http://www.eclipse.org/. This tutorial is based on Eclipse, version 3.5.
- The Weka package: http://sourceforge.net/projects/weka/files/
  Do not download the EXE for Windows, but get the ZIP with the source code: for that, browse to weka-3-6 > 3.6.6 > download the weka-3-6-6.zip file.
- A tool to unzip files (for instance 7Zip)
- A tool to edit files (for instance Notepad++)

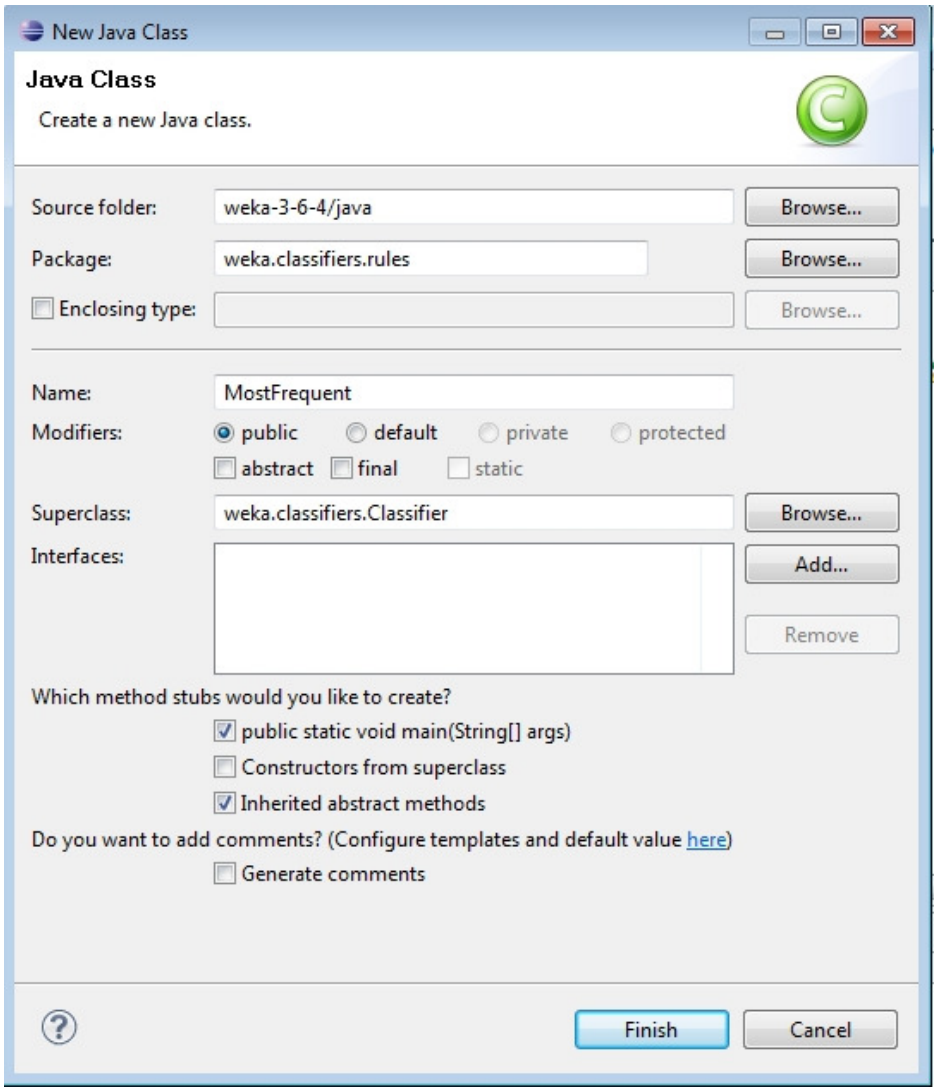
Preparation of the tools:

- Install and setup your Java programming environment
- Unzip the Weka zipped file inside a directory.
- Inside you will find a file named "weka-src.jar". This is the Weka source code. Unzip this file using your favorite unzip tool.
- Open Eclipse, and go to File > New > Project > Java Project from Existing Ant Buildfile. Give the Ant build file (it should be a file called "build.xml" inside the folder weka-3-6-6\weka-src\). Give the project a name (ex: weka-3-6-6). Select the first compilation task (compile [default]) and click on Finish.
- Right-click on your project name (weka-3-6-6) and then select Run As > Java Application. In the dialog box that should open, select weka.gui.GUIChooser (the first item). The Weka environment should be launched without error.

Adding a new classifier:

The goal of this project is to add a classifier that will classify any new instance as the most frequent class among the classes of the instances of this dataset. So basically the learning algorithm will be: get the list of the instances of the dataset, compute the most frequent class and store it.

The first step is to create a new Java class. Look inside your project for the package weka.classifiers.rules. Right click on this item and select New > Class. Name it "MostFrequent", set the superclass to weka.classifiers.Classifier, and select the creation of the method main().



Eclipse will generate a new Java class file, with two functions: buildClassifier() and main(). The main() function is the first one that will be called when we will launch the classifier using the command line, and the buildClassifier() function is the one that will be called when we will launch the learning. It is not possible to change the name of these functions, if not Weka won't find them.

This last function takes an argument: an object of the type "Instances". This object encodes the whole dataset. Its description can be viewed in the Weka API, on this website: http://weka.sourceforge.net/doc/weka/core/Instances.html

For instance, you have access to a function called enumerateInstances() to get the list of each instance, and a function called classAttribute() that return an Attribute object, giving information about the class of the instances of the dataset (for instance if the class is nominal or numeric, ...).

Before implementing the function that performs the learning, Weka needs to know which dataset your classifier can process, and which not. To simplify a lot the implementation of your classifier, we will consider that it can only process datasets that have numerical attributes and a nominal class (here, *nominal* means that the class is chosen in a list of predefined values, such as {blue, red, green}).

This check is done by a function called getCapabilities(). I will not enter in details about the Weka object needed for this. But the following piece of code should be self-explanatory. The following webpage gives more information:
http://weka.wikispaces.com/Writing+your+own+Classifier+%28post+3.5.2%29
So, in your class MostFrequent, insert the following function:

```
public class MostFrequent extends Classifier {

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

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

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

            // instances
            result.setMinimumNumberInstances(0);

            return result;
      }

      ... // remaining of the source code

}
```

Then, we need to call this function each time we want to perform a learning. So we will need to write the following code inside the buildClassifier function():

```
      /**
       * Generates the classifier.
       *
       * @param instances
       *    set of instances serving as training data
       * @throws Exception
       *    if the classifier has not been generated successfully
       */
      public void buildClassifier(Instances data) throws Exception {

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

This will disallow the classifier to work with any dataset that does not respect these requirements, and throw an error message if it is the case (in Weka your will not be able to select this classifier).

Now we need some Java class members to perform the learning of the classifier. First, we need a variable to store the (unique) class of all the instances. In Weka, this object has the type double. So add the following class member in the top of your Java class:

```
public class MostFrequent extends Classifier {
```

```
	/** The class value to predict. */
	private double m_ClassValue;

	... // remaining of the source code

}
```

Now we need an array to count how many instances we have for each possible value of the class attribute. So the value at the index *i* will tell how many instances has the class *i*. This will be a simple int[] array.

```
	...

	/** The number of instances in each class. */
	private int[] m_Counts;
```

Now, we can start to write the learning function. The first step is to initialize the array with the right length, and fill it with a lot of 1's. So let's continue the buildClassifier() function:

```
	public void buildClassifier(Instances data) throws Exception {

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

		// check how many classes we have
		int num_classes = data.numClasses();

		// select a default class for the prediction
		m_ClassValue = 0;

		// initialize m_Counts[]
		m_Counts = new int[num_classes];
		for (int i = 0; i < m_Counts.length; i++) {
			m_Counts[i] = 1;
		}
	}
```

We are ready for the count by itself. Weka provides the dataset as an enumeration of objects of type 'Instance'. Each object of this kind encodes the values of their attributes, the value of the class, ... In Java, an enumeration is an object that can be accessed using the member nextElement(), until we reach the end of the list, given by the function hasMoreElement(). More information in the following websites:

http://weka.sourceforge.net/doc/weka/core/Instance.html
http://download.oracle.com/javase/1.4.2/docs/api/java/util/Enumeration.html

So, let's continue the buildClassifier() function:

```
	public void buildClassifier(Instances data) throws Exception {

		...

		// count the instances
		Enumeration enu = data.enumerateInstances();
		while (enu.hasMoreElements()) {
```

```
                Instance instance = (Instance) enu.nextElement();
                int class_value = (int) instance.classValue();
                m_Counts[class_value]++;
            }
        }
```

The last thing we have to do is to get the most frequent class, and store it in our m_ClassValue member. For this, Weka has a class of useful functions that can be used for this purpose, in the class weka.core.Utils. The function maxIndex(int[] array) returns the index of the maximum element in a given array of integers. So we only need to write one line of code to complete our buildClassifier() function:

```
    public void buildClassifier(Instances data) throws Exception {

        ...

        // find the most frequent class
        m_ClassValue = Utils.maxIndex(m_Counts);
    }
```

Now we need to write the function that classifies a given instance into a class. This function will be very basic as the only task it has to do is to return the m_ClassValue member. The Weka graphical interface will guarantee the learning is performed *before* the prediction, so the member will be initialized. Here again, we will need to use a specific name for this function, in order Weka can find and call it. The reserved name for the function that classifies a new instance is classifyInstance(). So we insert inside our Java class the following piece of code:

```
public class MostFrequent extends Classifier {

    ...

    /**
     * Classifies a given instance.
     *
     * @param instance
     *    the instance to be classified
     * @return index of the predicted class
     */
    public double classifyInstance(Instance instance) {
        return m_ClassValue;
    }
}
```

Note that you can access the values of the attributes using the double[] instance.toDoubleArray() method.

The only last thing we will need to do is to call the classifier in the main() function, in order any user of your classifier can run it from the command line:

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

You do not need to do anything else: Weka will automatically recognize your class and add it to the graphical interface (GUI).

So you can directly test your classifier. In Eclipse, right click on the name of your project, and choose Run As > Java Application. Select the first item in the dialog box that should open (weka.gui.GUIChooser).

When Weka has compiled and is launched, select the Explorer. Load a dataset that has numerical values for the instance attributes, and a nominal class. For instance, take the Iris dataset inside the data folder of Weka (weka-3-6-6\data\iris.arff). You can also choose the Glass, Ionosphere, Segment or Diabetes datasets.

Go to the "Classify" tab, select the Choose button, and go to weka > classifiers > rules > MostFrequent. Click the Start button. Your classifier will launch and Weka will compute typical statistics for you. You can recognize that all the instances are classified in the same class, Iris-Setosa for the Iris dataset (the first class is chosen in case of equality).

Now you are ready to implement your own classifier. Improve your class or start from scratch to adapt to Weka any simple classifier seen during the lectures. You can also implement a classifier that launches pre-existing classifiers and filters.