Working with WEKA Java Code II

Prof. Pietro Ducange

The weka.classifier package

- This package contains the implementations of most of the algorithms for classification
- The key class is the abstract class ***abstractClassifier***, which denote the general structure of any scheme for classification and contains two main methods: *BuildClassifier(), classifyInstance(), distributionForInstance().*
- Each learning algorithm extends the class ***abstractClassifier*** and must implements the methods of the interface Classifier.
- Every scheme redefines the three methods according to how the classifier is built and how it classifies instances.

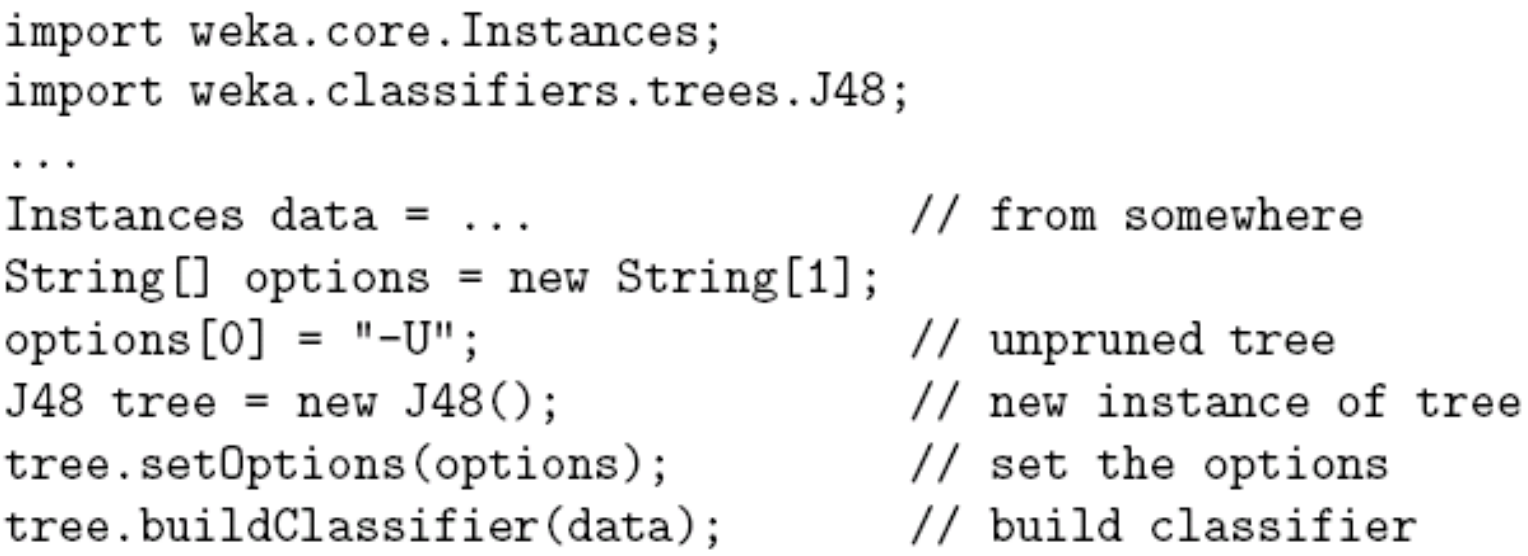
The weka.classifier subpackages

- bayes – contains bayesian classifiers, e.g., NaiveBayes
- evaluation – classes related to evaluation, e.g., confusion matrix, threshold curve (= ROC)
- functions – e.g., Support Vector Machines, regression algorithms, neural nets
- lazy – "learning" is performed at prediction time, e.g., k-nearest neighbor (k-NN)
- meta – meta-classifiers that use a base one or more classifiers as input, e.g., boosting, bagging or stacking
- mi – classifiers that handle multi-instance data
- misc – various classifiers that don't fit in any another category
- rules – rule-based classifiers, e.g., ZeroR
- trees – tree classifiers, like decision trees with J48 a very common one

Building a classifier

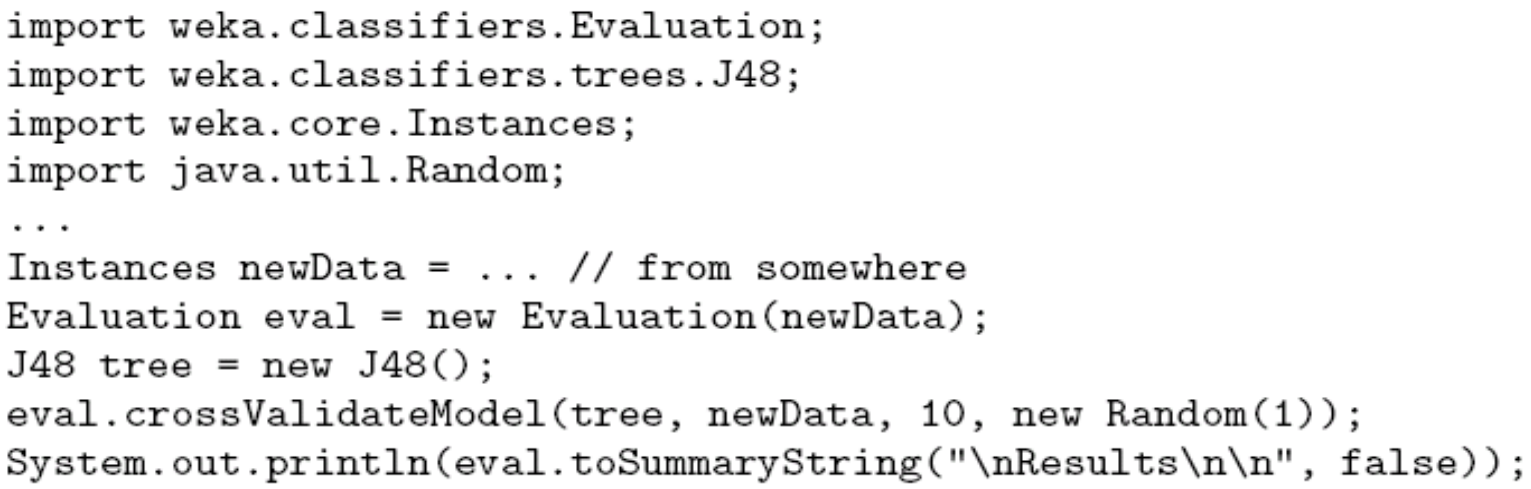
A batch classifier is really simple to build:

- set options – either using the setOptions(String[]) method or the actual methods
- train it – calling the buildClassifier(Instances) method with the training set.
- By definition, the buildClassifier(Instances) method resets the internal mode completely, in order to ensure that subsequent calls of this method with the same data result in the same model



```
import weka.core.Instances;
import weka.classifiers.trees.J48;
...
Instances data = ...                 // from somewhere
String[] options = new String[1];
options[0] = "-U";                   // unpruned tree
J48 tree = new J48();                // new instance of tree
tree.setOptions(options);            // set the options
tree.buildClassifier(data);          // build classifier
```

Evaluating a Classifier (CV)



```
import weka.classifiers.Evaluation;
import weka.classifiers.trees.J48;
import weka.core.Instances;
import java.util.Random;
...
Instances newData = ... // from somewhere
Evaluation eval = new Evaluation(newData);
J48 tree = new J48();
eval.crossValidateModel(tree, newData, 10, new Random(1));
System.out.println(eval.toSummaryString("\nResults\n\n", false));
```

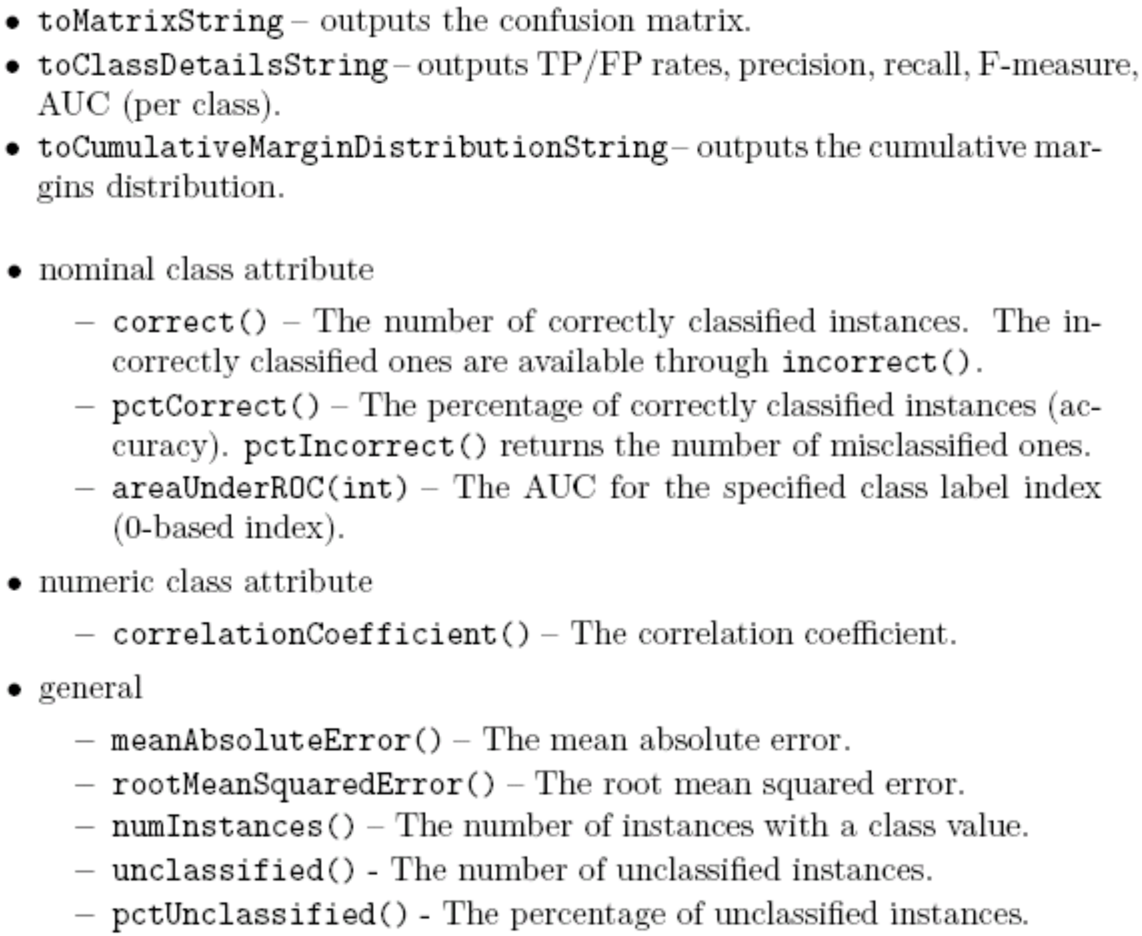
•The Evaluation object in this example is initialized with the dataset used in the evaluation process

•This is done in order to inform the evaluation **about the type of data** that is being evaluated, ensuring that all internal data structures are setup correctly

Evaluating a Classifier (train/test)

```
import weka.core.Instances;
import weka.classifiers.Evaluation;
import weka.classifiers.trees.J48;
...
Instances train = ...   // from somewhere
Instances test = ...    // from somewhere
// train classifier
Classifier cls = new J48();
cls.buildClassifier(train);
// evaluate classifier and print some statistics
Evaluation eval = new Evaluation(train);
eval.evaluateModel(cls, test);
System.out.println(eval.toSummaryString("\nResults\n\n", false));
```

Methods of the Evaluation Class

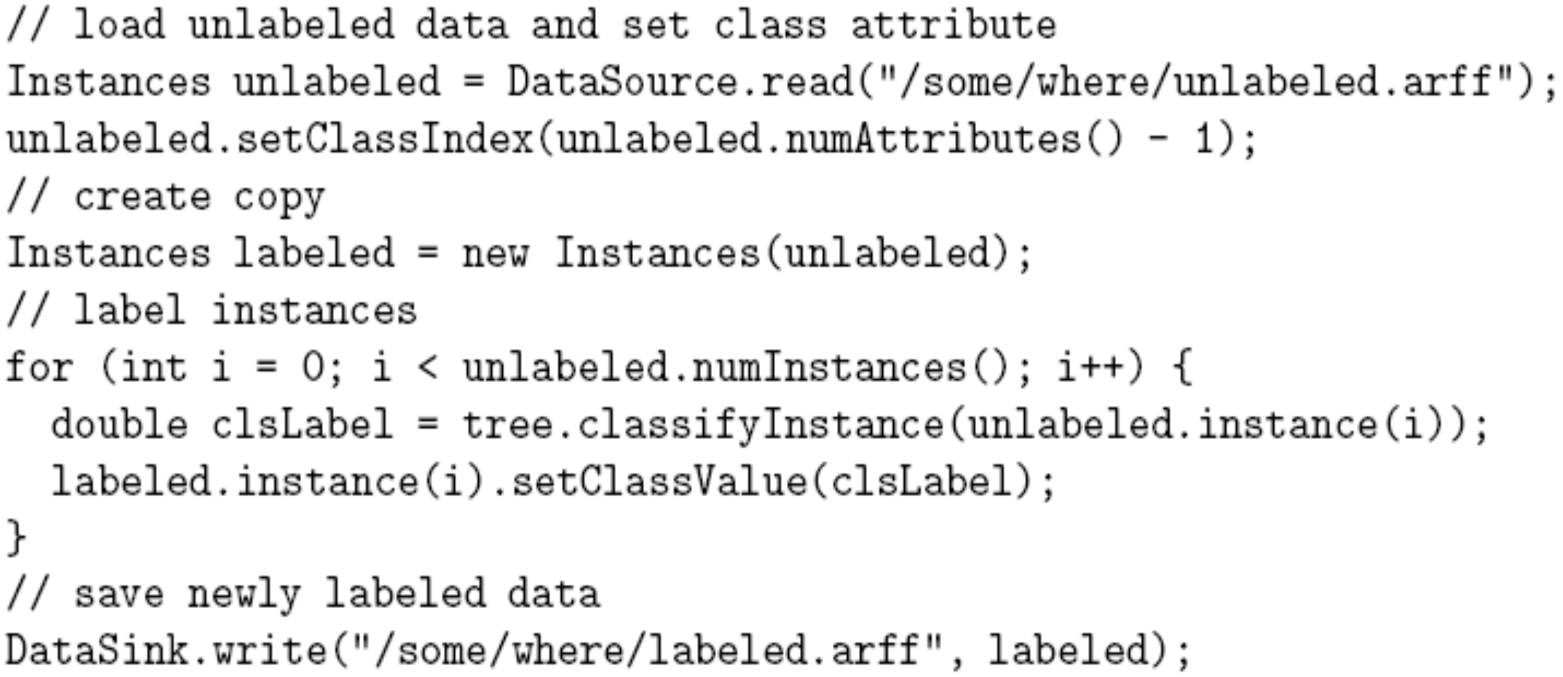


- `toMatrixString` – outputs the confusion matrix.
- `toClassDetailsString` – outputs TP/FP rates, precision, recall, F-measure, AUC (per class).
- `toCumulativeMarginDistributionString` – outputs the cumulative margins distribution.

- nominal class attribute
  - `correct()` – The number of correctly classified instances. The incorrectly classified ones are available through `incorrect()`.
  - `pctCorrect()` – The percentage of correctly classified instances (accuracy). `pctIncorrect()` returns the number of misclassified ones.
  - `areaUnderROC(int)` – The AUC for the specified class label index (0-based index).
- numeric class attribute
  - `correlationCoefficient()` – The correlation coefficient.
- general
  - `meanAbsoluteError()` – The mean absolute error.
  - `rootMeanSquaredError()` – The root mean squared error.
  - `numInstances()` – The number of instances with a class value.
  - `unclassified()` - The number of unclassified instances.
  - `pctUnclassified()` - The percentage of unclassified instances.

Classifying Instances

- After a classifier setup has been evaluated and proven to be useful, a built classifier can be used to make predictions and label previously unlabeled data



```
// load unlabeled data and set class attribute
Instances unlabeled = DataSource.read("/some/where/unlabeled.arff");
unlabeled.setClassIndex(unlabeled.numAttributes() - 1);
// create copy
Instances labeled = new Instances(unlabeled);
// label instances
for (int i = 0; i < unlabeled.numInstances(); i++) {
  double clsLabel = tree.classifyInstance(unlabeled.instance(i));
  labeled.instance(i).setClassValue(clsLabel);
}
// save newly labeled data
DataSink.write("/some/where/labeled.arff", labeled);
```

Exercise I

Write a Java program, which performs the following steps:

1. Reads a training and a test set from two specified file paths
2. Defines the Instances objects for the datasets and set the class index (use the setClassIndex method of the class Instances)
3. Defines and train a J48 decision tree without performing the pruning
4. Evaluates the accuracy of the decision tree both on the training and test set
5. Prints the results of the accuracy evaluations

A solution can be found in the file Classificazione.java

Exercise II

Write a Java program, which performs the following steps:

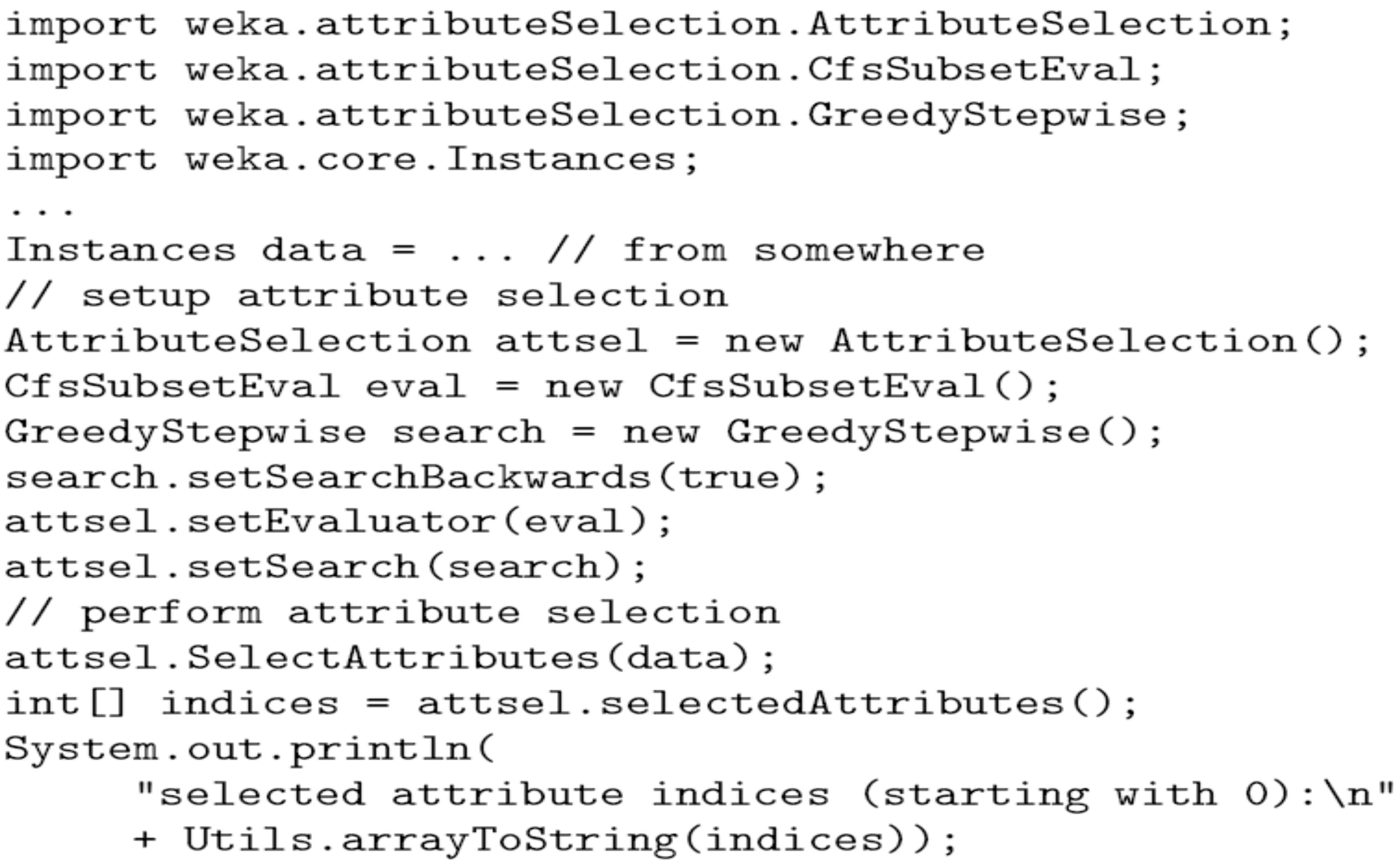
1. By using the test set previously loaded, generates an unlabeled dataset (use the *setClassMissing* method of the class *Instance*)

2. Classifies each instance of the unlabeled dataset by using the classifier defined in Exercise I

3. For each instance, prints the actual and the estimated class

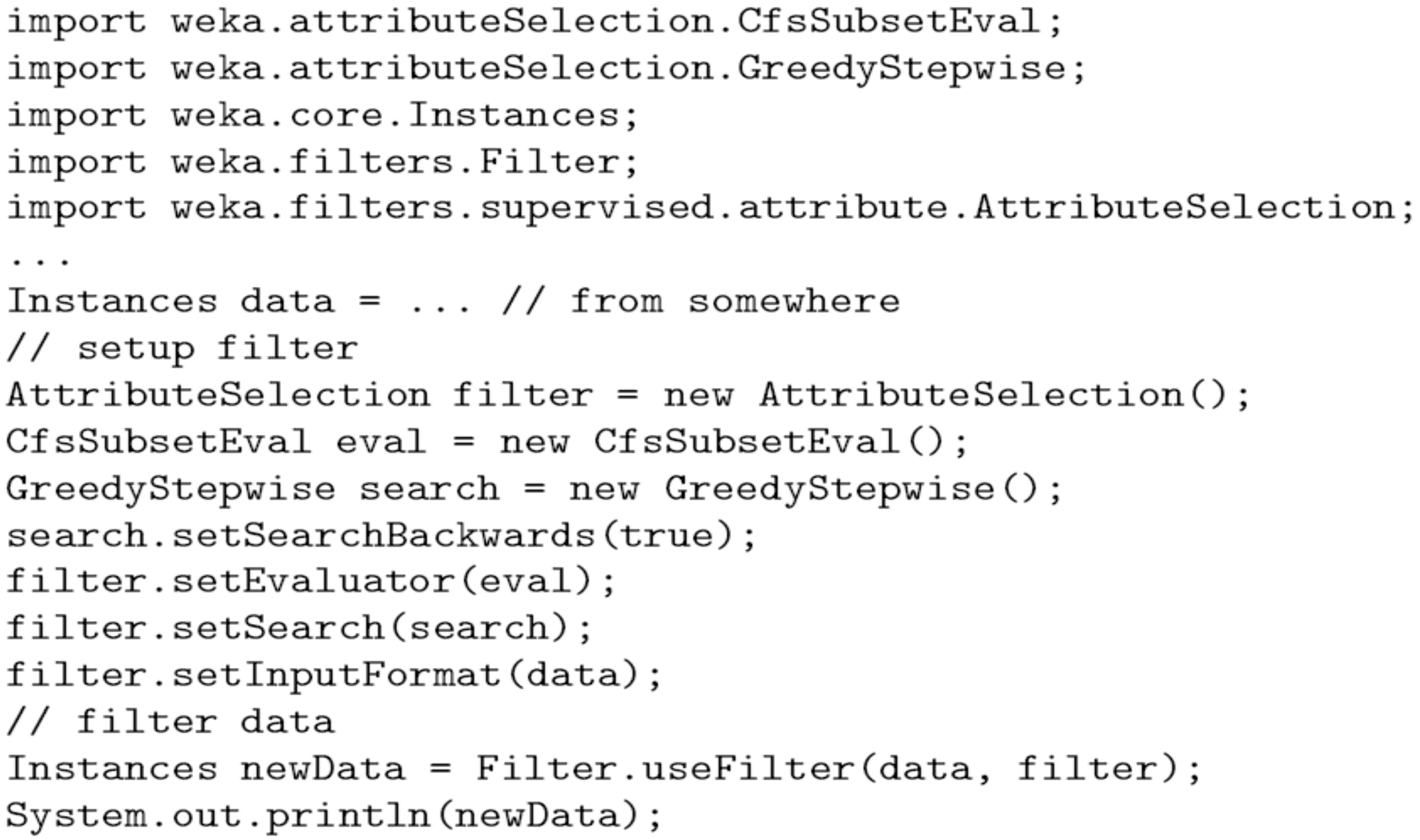
A solution can be found in the file ClassificazioneIstanze.java

Attribute Selection



```
import weka.attributeSelection.AttributeSelection;
import weka.attributeSelection.CfsSubsetEval;
import weka.attributeSelection.GreedyStepwise;
import weka.core.Instances;
...
Instances data = ... // from somewhere
// setup attribute selection
AttributeSelection attsel = new AttributeSelection();
CfsSubsetEval eval = new CfsSubsetEval();
GreedyStepwise search = new GreedyStepwise();
search.setSearchBackwards(true);
attsel.setEvaluator(eval);
attsel.setSearch(search);
// perform attribute selection
attsel.SelectAttributes(data);
int[] indices = attsel.selectedAttributes();
System.out.println(
     "selected attribute indices (starting with 0):\n"
     + Utils.arrayToString(indices));
```

Attribute Selection (Filter)



```
import weka.attributeSelection.CfsSubsetEval;
import weka.attributeSelection.GreedyStepwise;
import weka.core.Instances;
import weka.filters.Filter;
import weka.filters.supervised.attribute.AttributeSelection;
...
Instances data = ... // from somewhere
// setup filter
AttributeSelection filter = new AttributeSelection();
CfsSubsetEval eval = new CfsSubsetEval();
GreedyStepwise search = new GreedyStepwise();
search.setSearchBackwards(true);
filter.setEvaluator(eval);
filter.setSearch(search);
filter.setInputFormat(data);
// filter data
Instances newData = Filter.useFilter(data, filter);
System.out.println(newData);
```

Exercise III

Write a Java program, which performs the following steps:

1. Reads a training and a test set from two specified file paths
2. Defines the *Instances* objects for the datasets and set the class index
3. Performs a feature selection by using three appropriate approaches
4. Defines and trains four Classifiers (J48 with pruning) by using the original training set and the three training sets obtained after the feature selection
5. Selects the most performing classifiers (on the training set)
6. Verifies if the selected classifier is the one characterized with the best generalization capability

A solution can be found in the file FeatureSelection.java

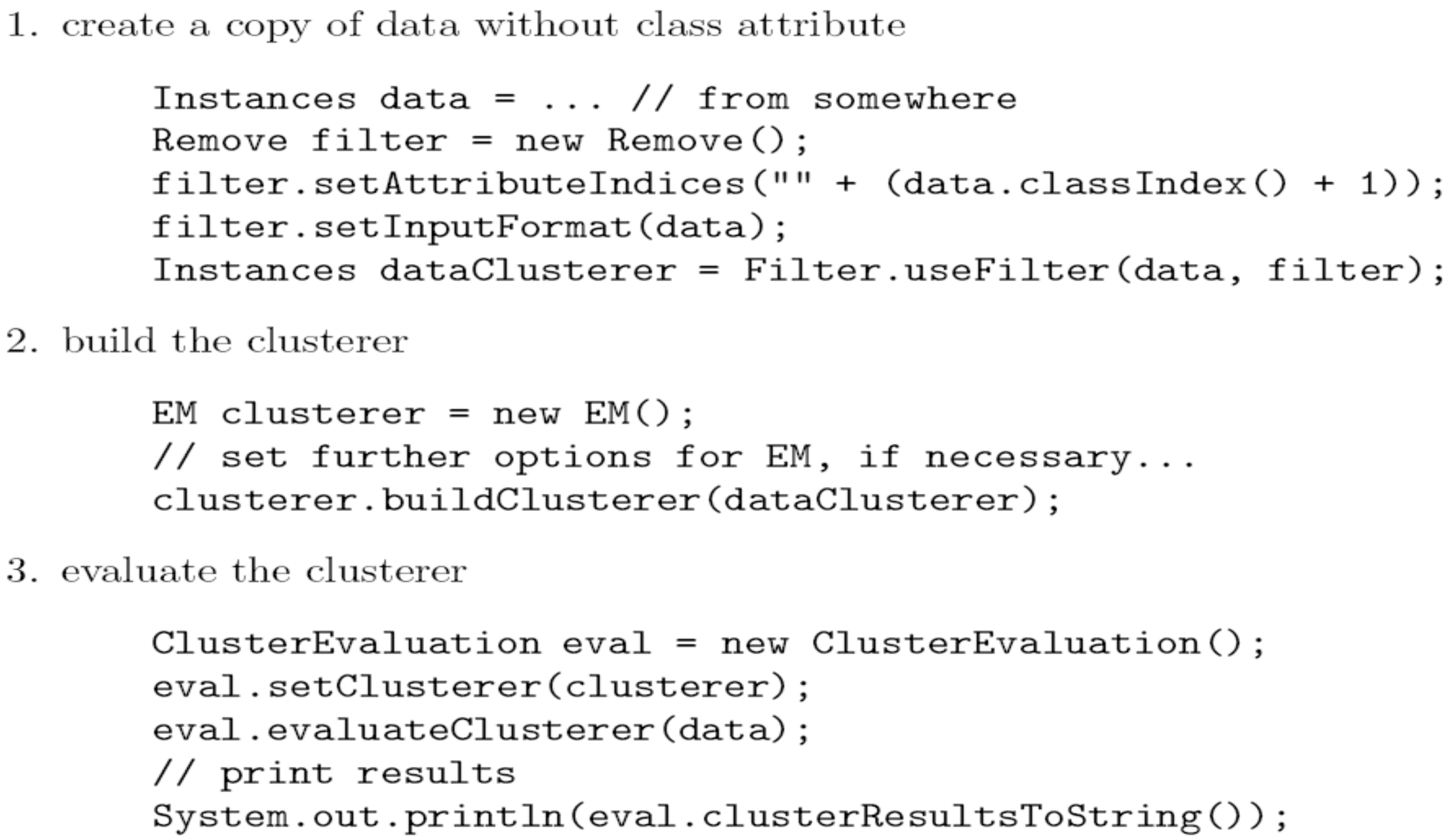
Building Cluster Models with WEKA API

```
import weka.clusterers.EM;
import weka.core.Instances;
...
Instances data = ... // from somewhere
String[] options = new String[2];
options[0] = "-I";                 // max. iterations
options[1] = "100";
EM clusterer = new EM();   // new instance of clusterer
clusterer.setOptions(options);     // set the options
clusterer.buildClusterer(data);    // build the clusterer
```

Evaluating Cluster Model

```
import weka.clusterers.ClusterEvaluation;
import weka.clusterers.EM;
import weka.core.Instances;
...
Instances data = ... // from somewhere
EM cl = new EM();
cl.buildClusterer(data);
ClusterEvaluation eval = new ClusterEvaluation();
eval.setClusterer(cl);
eval.evaluateClusterer(new Instances(data));
System.out.println(eval.clusterResultsToString());
```

Classes to clusters evaluation



1. create a copy of data without class attribute

```
Instances data = ... // from somewhere
Remove filter = new Remove();
filter.setAttributeIndices("" + (data.classIndex() + 1));
filter.setInputFormat(data);
Instances dataClusterer = Filter.useFilter(data, filter);
```

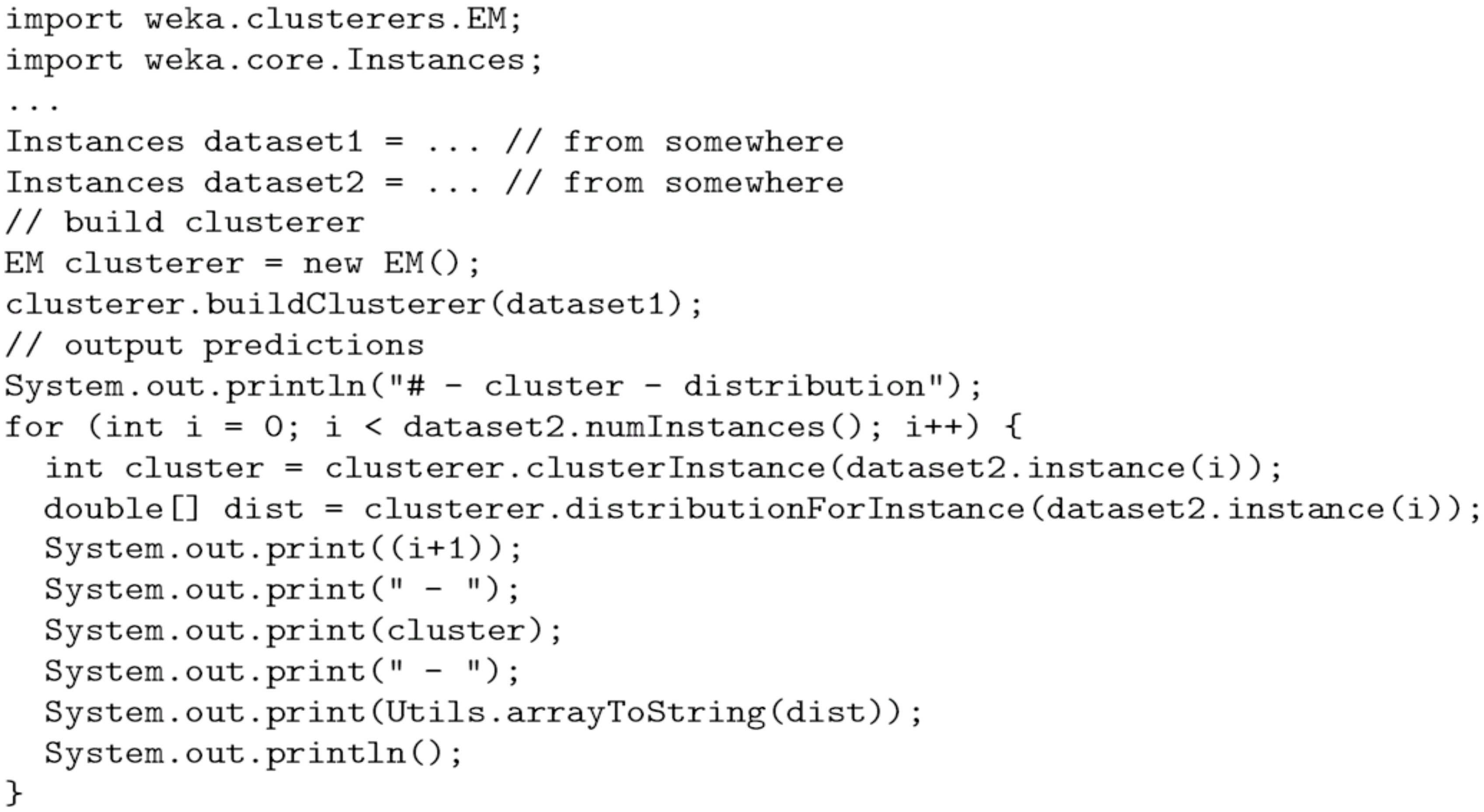
2. build the clusterer

```
EM clusterer = new EM();
// set further options for EM, if necessary...
clusterer.buildClusterer(dataClusterer);
```

3. evaluate the clusterer

```
ClusterEvaluation eval = new ClusterEvaluation();
eval.setClusterer(clusterer);
eval.evaluateClusterer(data);
// print results
System.out.println(eval.clusterResultsToString());
```

Assigning Instances to Clusters



```
import weka.clusterers.EM;
import weka.core.Instances;
...
Instances dataset1 = ... // from somewhere
Instances dataset2 = ... // from somewhere
// build clusterer
EM clusterer = new EM();
clusterer.buildClusterer(dataset1);
// output predictions
System.out.println("# - cluster - distribution");
for (int i = 0; i < dataset2.numInstances(); i++) {
  int cluster = clusterer.clusterInstance(dataset2.instance(i));
  double[] dist = clusterer.distributionForInstance(dataset2.instance(i));
  System.out.print((i+1));
  System.out.print(" - ");
  System.out.print(cluster);
  System.out.print(" - ");
  System.out.print(Utils.arrayToString(dist));
  System.out.println();
}
```

Exercise IV

Write a Java program, which performs the following steps:

- Reads a clustering dataset (cluss.arff, 16 clusters) for a specified file path
- Builds a clustering model by using the simple k-means algorithm
- Builds a clustering model by using Hierarchical Agglomerative
- Prints the results achieved by the two clustering models

A solution can be found in the file ClusteringCompare.java

Exercise V

Write a Java program, which performs the following steps:

- Reads a training and a test set from two specified file paths
- Builds a clustering model by using DBscan algorithms using the training set (do not consider the class attribute)
- Evaluates the model on the training set (classes to cluster evaluation mode)
- Assigns each instance of the test set to a cluster (do not consider the class attribute)
- For each instance, prints the actual class and the assigned cluster

A solution can be found in the file Clustering.java