Weka[-1] Use Weka in your Java code

译者：Koala++/屈伟

无意间在网上看到了：http://weka.wiki.sourceforge.net/Use+Weka+in+your+Java+code，相对我写的代码，它的当然更有权威性。翻译完了，第一次翻译，术语的汉语很多不清楚。还没有校对，有什么错误请告诉我。

你可能要用的最常用的组件(components)是：

- Instances 你的数据
- Filter 对数据的预处理
- Classifiers/Clusterer 被建立在预处理的数据上，分类/聚类
- Evaluating 评价 classifier/clusterer
- Attribute selection 去除数据中不相关的属性

下面将介绍如果在你自己的代码中使用 WEKA，其中的代码可以在上面网址的尾部找到。

Instances

ARFF 文件

3.5.5 和 3.4.X 版本

从 ARFF 文件中读取是一个很直接的

```
import weka.core.Instances;
import java.io.BufferedReader;
import java.io.FileReader;
...
Instances data = new Instances(
                    new BufferedReader(
                       new FileReader("/some/where/data.arff")));
// setting class attribute
data.setClassIndex(data.numAttributes() - 1);
```

Class Index 是指示用于分类的目标属性的下标。在 ARFF 文件中，它被默认为是最后一个属性，这也就是为什么它被设置成 numAttributes-1.

你必需在使用一个 Weka 函数(ex: weka.classifiers.Classifier.buildClassifier(data))之前设置 Class Index。

3.5.5 和更新的版本

DataSource 类不仅限于读取 ARFF 文件，它同样可以读取 CSV 文件和其它格式的文件(基本上 Weka 可以通过它的转换器(converters)导入所有的文件格式)。

```
import weka.core.converters.ConverterUtils.DataSource;
...
DataSource source = new DataSource("/some/where/data.arff");
Instances data = source.getDataSet();
```

```
// setting class attribute if the data format does not provide this
//information
// E.g., the XRFF format saves the class attribute information as well
if (data.classIndex() == -1)
    data.setClassIndex(data.numAttributes() - 1);
```

数据库

从数据库中读取数据稍微难一点，但是仍然是很简单的，首先，你需要修改你的DatabaseUtils.props（自己看一下原文，基本上都有链接）重组(resemble)你的数据库连接。比如，你想要连接一个 MySQL 服务器，这个服务器运行于 3306 端口(默认)，MySQL JDBC 驱动被称为 Connector/J(驱动类是 org.gjt.mm.mysql.Driver)。假设存放你数据的数据库是 some_database。因为你只是读取数据，你可以用默认用户 nobody，不设密码。你需要添加下面两行在你的 props 文件中:

```
jdbcDriver=org.gjt.mm.mysql.Driver
jdbcURL=jdbc:mysql://localhost:3306/some_database
```

其次，你的读取数据的 Java 代码，应该写成下面这样：

```
import weka.core.Instances;
import weka.experiment.InstanceQuery;
...
InstanceQuery query = new InstanceQuery();
query.setUsername("nobody");
query.setPassword("");
query.setQuery("select * from whatsoever");
// if your data is sparse, then you can say so too
// query.setSparseData(true);
Instances data = query.retrieveInstances();
```

注意：

- 别忘了把 JDBC 驱动加入你的 CLASSPATH 中
- 如果你要用 MS Access，你需要用 JDBC-ODBC-bridge，它是 JDK 的一部分。

参数设置(Option handling)

Weka 中实现了 weka.core.OptionHandler 接口，这个接口为比如 classifiers，clusterers，filers 等提供了设置，获取参数的功能，函数如下：

- `void setOptions(String[] Options)`
- `String[] getOptions()`

下面依次介绍几种参数设置的方法：

- 手工建立一个 String 数组

```
String[] options = new String[2];
options[0] = "-R";
options[1] = "1";
```

- 用 weka.core.Utils 类中的函数 splitOptions 将一个命令行字符串转换成一下数组

```
String[] options = weka.core.Utils.splitOptions("-R 1");
```

- 用 OptionsToCode.java 类自动将一个命令行转换成代码，对于命令行中包含 nested

classes，这些类又有它们自己的参数，如果 SMO 的核参数这种情况很有帮助。

```
java OptionsToCode weka.classifiers.functions.SMO
```

将产生以下输出：

```
//create new instance of scheme
weka.classifiers.functions.SMO scheme = new
    weka.classifiers.functions.SMO();
// set options
scheme.setOptions(weka.core.Utils.splitOptions("-C 1.0 -L 0.0010 -P
1.0E-12 -N 0 -V -1 -W 1 -K \"
weka.classifiers.functions.supportVector.PolyKernel -C 250007 -E
1.0\""));
```

并且，OptionTree.java 工具可以使你观察一个 nested 参数字符串。

Filter

一个 filter 有两种不同的属性

- 监督的或是监督的(supervised or unsupervised)
  是否受用户控制
- 基于属性的或是基于样本的(attribute- or instance-based)
  比如：删除满足一定条件的属性或是样本

多数 filters 实现了 OptionHandler 接口，这意味着你可以通过 String 数组设置参数，而不用手工地用 set-方法去依次设置。比如你想删除数据集中的第一个属性，你可用这个 filter。

```
weka.filters.unsupervised.attribute.Remove
```

通过设置参数

```
-R 1
```

如果你有一个 Instances 对象，比如叫 data，你可以用以下方法产生并使用 filter：

```
import weka.core.Instances;
import weka.filters.Filter;
import weka.filters.unsupervised.attribute.Remove;
...
String[] options = new String[2];
options[0] = "-R";                          // "range"
options[1] = "1";                           // first attribute
Remove remove = new Remove();               // new instance of filter
remove.setOptions(options);                 // set options
// inform filter about dataset //**AFTER** setting options
remove.setInputFormat(data);
Instances newData = Filter.useFilter(data, remove);   // apply filter
```

运行中过滤(Filtering on-the-fly)

FilteredClassifier meta-classifier 是一种运行中过滤的方式。它不需要在分类器训练之前先对数据集过滤。并且，在预测的时候，你也不需要将测试数据集再次过滤。下面的例子中使用 meta-classifier with Remove filter 和 J48，删除一个 attribute ID 为 1 的属性。

```
import weka.core.Instances;
```

```
import weka.filters.Filter;
import weka.filters.unsupervised.attribute.Remove;
...
String[] options = new String[2];
options[0] = "-R";                          // "range"
options[1] = "1";                           // first attribute
Remove remove = new Remove();               // new instance of filter
remove.setOptions(options);                 // set options
// inform filter about dataset **AFTER** setting options
remove.setInputFormat(data);
Instances newData = Filter.useFilter(data, remove);   // apply filter
import weka.classifiers.meta.FilteredClassifier;
import weka.classifiers.trees.J48;
import weka.filters.unsupervised.attribute.Remove;
...
Instances train = ...        // from somewhere
Instances test = ...         // from somewhere
// filter
Remove rm = new Remove();
rm.setAttributeIndices("1");  // remove 1st attribute
// classifier
J48 j48 = new J48();
j48.setUnpruned(true);       // using an unpruned J48
// meta-classifier
FilteredClassifier fc = new FilteredClassifier();
fc.setFilter(rm);
fc.setClassifier(j48);
// train and make predictions
fc.buildClassifier(train);
for (int i = 0; i < test.numInstances(); i++) {
   double pred = fc.classifyInstance(test.instance(i));
   System.out.printn("ID: " + test.instance(i).value(0));
   System.out.print(", actual: " + test.classAttribute().value((int)
        test.instance(i).classValue()));
   System.out.println(", predicted: " +
        test.classAttribute().value((int) pred));
 }
```

其它 Weka 中便利的 meta-schemes:

```
weka.clusterers.FilteredClusterer (since 3.5.4)
weka.associations.FilteredAssociator (since 3.5.6)
```

批过滤(Batch filtering)

在命令行中，你可以用-b 选项 enable 第二个 input/ouput 对，用对第一个数据集过滤的设置来过滤第二个数据集。如果你正使用特征选择(attribute selection)或是正规化

(standardization)，这是必要的，否则你会得到两个不兼容的数据集。其实这做起来很容易，只需要用 setInputFormat(Instances)去初始化一个过滤器，即用 training set，然后将这个过滤器依次用于 training set 和 test set。下面的例子将展示如何用 Standardize 过滤器过滤一个训练集和测试集的。

```
Instances train = ...   // from somewhere
Instances test = ...    // from somewhere
// initializing the filter once with training set
Standardize filter = new Standardize();
filter.setInputFormat(train);
// configures the Filter based on train instances and returns filtered
//instances
Instances newTrain = Filter.useFilter(train, filter);
// create new test set
Instances newTest = Filter.useFilter(test, filter);
```

调用转换(Calling conventions)

setInputFormat(Instances)方法总是必需是应用过滤器时最后一个调用，比如用 Filter.useFilter(Instances,Filter)。为什么？首先，它是使用过滤器的转换，其实，很多过滤器在 setInputFormat(Instances)方法中用当前的设置参数产生输出格式(output format)（在这个调用后设置参数不再有任何作用）。

分类(classification)

一些必要的类可以在下面的包中找到：

```
weka.classifiers
```

建立一个分类器(Build a classifier)

批(Batch)

在一个给定的数据集上训练一个 Weka 分类器是非常简单的事。例如，我们可以训练一个 C4.5 树在一个给定的数据集 data 上。训练是通过 buildClassifier(Instances)来完成的。

```
import weka.classifiers.trees.J48;
...
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
```

增量式(Incremental)

实现了 weka.classifiers.UpdateabeClassifier 接口的分类器可以增量式的训练，它可以节约内存，因为你不需要把数据一次全部读入内存。你可以查一下文档，看哪些分类器实现了这个接口。

真正学习一个增量式的分类器是很简单的：

- 调用 buildClassifier(Instances)，其中 Instances 包话这种数据集的结构，其中 Instances 可以有数据，也可以没有。
- 顺序调用 updateClassifier(Instances)方法，通过一个新的 weka.core.Instances，更新分类器。

这里有一个用 weka.core.converters.ArffLoader 读取数据，并用 weka.classifiers.bayes.NaiveBayesUpdateable 训练分类器的例子。

```
//   load data
ArffLoader loader = new ArffLoader();
loader.setFile(new File("/some/where/data.arff"));
Instances structure = loader.getStructure();
structure.setClassIndex(structure.numAttributes() - 1);
// train NaiveBayes
NaiveBayesUpdateable nb = new NaiveBayesUpdateable();
nb.buildClassifier(structure);
Instance current;
while ((current = loader.getNextInstance(structure)) != null)
    nb.updateClassifier(current);
```

Evaluating

交叉检验

如果你一个训练集并且没有测试集，你也话想用十次交叉检验的方法来评价分类器。这可以很容易地通过 Evaluation 类来实现。这里，我们用 1 作为随机种子进行随机选择，查看 Evaluation 类，可以看到更多它输出的统计结果。

```
import weka.classifiers.Evaluation;
import java.util.Random;
...
Evaluation eval = new Evaluation(newData);
eval.crossValidateModel(tree, newData, 10, new Random(1));
```

注意：分类器（在这个例子中是 tree）不应该在作为 crossValidateModel 参数之前训练，为什么？因为每当 buildClassifier 方法被调用时，一个分类器必需被重新初始化（换句话说：接下来调用 buildClassifier 方法总是返回相同的结果），你将得到不一致，没有任何意义的结果。crossValidateModel 方法处理分类器的 training 和 evaluation（每一次 cross-validation，它产生一个你作为参数的原分类器的复本(copy)）。

Train/Set set

如果你有一个专用的测试集，你可以在训练集上训练一个分类器，再在测试集上测试。在下面的例子中，一个 J48 被实例化，训练，然后评价。在控制台输出一些统计值。

```
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));
```

统计(statistics)

下面是一些获取评价结果的方法

- 数值型类别
  - Correct() 分类正确的样本数 （还有 incorrect() ）
  - pctCorrect() 分类正确的百分比 （还有 pctIncorrect()）
  - kappa() Kappa statistics
- 离散型类别
  - correlationCoefficient() 相关系数
- 通用
  - meanAbsoluteError() 平均绝对误差
  - rootMeanSquaredError() 均方根误差
  - unclassified() 未被分类的样本数
  - pctUnclassified() 未被分类的样本百分比

如果你想通过命令行获得相同的结果，使用以下方法：

```
import weka.classifiers.trees.J48;
import weka.classifiers.Evaluation;
...
String[] options = new String[2];
options[0] = "-t";
options[1] = "/some/where/somefile.arff";
System.out.println(Evaluation.evaluateModel(new J48(), options));
```

ROC 曲线/AUC（ROC curves/AUC）

从 Weka3.5.1 开始，你可以在测试中产生 ROC 曲线/AUC。你可以调用 Evaluation 类中的 predictions()方法去做。你可从 Generating Roc curve 这篇文章中找到许多产生 ROC 曲线的例子。

分类样本(classifying instances)

如果你想用你新训练的分类器去分类一个未标记数据集(unlabeled dataset)，你可以使用下面的代码段，它从/some/where/unlabeled.arff 中读取数据，并用先前训练的分类器 tree 去标记样本，并保存标记样本在/some/where/labeled.arff 中

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
```

```
import java.io.FileReader;
import java.io.FileWriter;
import weka.core.Instances;
...
// load unlabeled data
Instances unlabeled = new Instances(
                new BufferedReader(
                   new FileReader("/some/where/unlabeled.arff")));

// set class attribute
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 labeled data
BufferedWriter writer = new BufferedWriter(
                   new FileWriter("/some/where/labeled.arff"));
writer.write(labeled.toString());
writer.newLine();
writer.flush();
writer.close();
```

数值型类别注意事项

- 如果你对所有类别在分布感兴趣，那么使用 distributionForInstance(Instance)。这个方法返回一个针对每个类别概率的 double 数组。
- classifyInstance 返回的是一个 double 值(或者是 distributionForInstance 返回的数组中的下标)，它仅仅是属性的下标，例如，如果你想用字符串形式来表现返回的类别结果 clsLabel，你可以这样输出：

```
System.out.println(clsLabel + " -> " +
    unlabeled.classAttribute().value((int) clsLabel));
```

聚类(Clustering)

聚类与分类相似，必要的类可以在下面的包中找到

weka.clusterers

建立一个 Clusterer

批（Batch）

一个 clusterer 建立与建立一个分类器的方式相似，只是不是使用 buildClassifier(Instances)方法，它使用 buildClusterer(Instances)，下面的代码段展示了如何用 EM clusterer 使用最多 100 次迭代的方法。

```
import weka.clusterers.EM;
...
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
```

增量式

实现了 weka.clusterers.UpdateableClusterer 接口的 Clusterers 可以增量式的被训练(从 3.5.4 版开始)。它可以节省内存，因为它不需要一次性将数据全部读入内存。查看文档，看哪些 clusterers 实现了这个接口。

真正训练一个增量式的 clusterer 是很简单的：

- 调用 buildClusterer(Instances) 其中 Instances 包话这种数据集的结构，其中 Instances 可以有数据，也可以没有。
- 顺序调用 updateClusterer(Instances)方法，通过一个新的 weka.core.Instances，更新 clusterer。
- 当全部样本被处理完之后，调用 updateFinished()，因为 clusterer 还要进行额外的计算。

下面是一个用 weka.core.converters.ArffLoader 读取数据，并训练 weka.clusterers.Cobweb 的代码：

```
//load data
ArffLoader loader = new ArffLoader();
loader.setFile(new File("/some/where/data.arff"));
Instances structure = loader.getStructure();

// train Cobweb
Cobweb cw = new Cobweb();
cw.buildClusterer(structure);
Instance current;
while ((current = loader.getNextInstance(structure)) != null)
    cw.updateClusterer(current);
cw.updateFinished();
```

评价(Evaluating)

评价一个 clusterer，你可用 ClusterEvaluation 类，例如，输出聚了几个类:

```
import weka.clusterers.ClusterEvaluation;
```

```
import weka.clusterers.Clusterer;
...
ClusterEvaluation eval = new ClusterEvaluation();
// new clusterer instance, default options
Clusterer clusterer = new EM();
clusterer.buildClusterer(data);       // build clusterer
eval.setClusterer(clusterer);            // the cluster to evaluate
// data to evaluate the clusterer on
eval.evaluateClusterer(newData);
// output # of clusters
System.out.println("# of clusters: " + eval.getNumClusters());
```

在 density based clusters 这种情况下，你可用交叉检验的方法去做(注意:用 MakeDensityBasedClusterer 你可将任何 clusterer 转换成一下基于密度(density based)的 clusterer)。

```
import weka.clusterers.ClusterEvaluation;
import weka.clusterers.DensityBasedClusterer;
import java.util.Random;
...
ClusterEvaluation eval = new ClusterEvaluation();
eval.setClusterer(clusterer);      // the clusterer to evaluate
eval.crossValidateModel(               // cross-validate
    clusterer, newData, 10,              // with 10 folds
   new Random(1));       // and random number generator with seed 1
```

如果你想用命令行方式得到相同的结果，用以下方法：

```
import weka.clusterers.EM;
import weka.clusterers.ClusterEvaluation;
...
String[] options = new String[2];
options[0] = "-t";
options[1] = "/some/where/somefile.arff";
System.out.println(ClusterEvaluation.evaluateClusterer(new EM(),
    options));
```

聚类数据集(Clustering instances)

与分类唯一不同是名字不同。它不是用 classifyInstances(Instance)，而是用 clusterInstance(Instance)。获得分布的方法仍然是 distributionForInstance(Instance)。

Classes to cluster evaluation

如果你的数据包含一个类别属性，并且你想检查一下产生的 clusters 与类别吻合程度，你可进行所谓的 classes to clusters evaluation。Weka Exporer 提供了这个功能，并用它也很容易实现，下面是一些必要的步骤。

- 读取数据，设置类别属性下标

```
Instances data = new Instances(new BufferedReader(new
   FileReader("/some/where/file.arff")));
```

```
data.setClassIndex(data.numAttributes() - 1);
```

- 产生无类别的数据，并用下面代码训练

```
weka.filters.unsupervised.attribute.Remove filter = new
    eka.filters.unsupervised.attribute.Remove();
filter.setAttributeIndices("" + (data.classIndex() + 1));
filter.setInputFormat(data);
Instances dataClusterer = Filter.useFilter(data, filter);
```

- 学习一个 clusterer，比如 EM

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

- 用仍然包含类别属性的数据集评价这个 clusterer

```
ClusterEvaluation eval = new ClusterEvaluation();
eval.setClusterer(clusterer);
eval.evaluateClusterer(data)
```

- 输出评价结果

```
System.out.println(eval.clusterResultsToString());
```

属性选择(Attribute selection)

其实没有必要在你的代码中直接使用属性选择类，因为已经有 meta-classifier 和 filter 可以进行属性选择，但是为了完整性，底层的方法仍然被列出来了。下面就是用 CfsSubsetEVal 和 GreedStepwise 方法的例子。

Meta-Classifier

下面的 meta-classifier 在数据在传给 classifier 之前，进行了一个预外理的步骤:

```
Instances data = ...  // from somewhere
AttributeSelectedClassifier classifier = new
AttributeSelectedClassifier();
CfsSubsetEval eval = new CfsSubsetEval();
GreedyStepwise search = new GreedyStepwise();
search.setSearchBackwards(true);
J48 base = new J48();
classifier.setClassifier(base);
classifier.setEvaluator(eval);
classifier.setSearch(search);
// 10-fold cross-validation
Evaluation evaluation = new Evaluation(data);
evaluation.crossValidateModel(classifier, data, 10, new Random(1));
System.out.println(evaluation.toSummaryString());
```

Filter

过滤器方法是很直接的，在设置过滤器之后，你就可以通过过滤器过滤并得到过滤后的数据集。

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

Low-Level

如果 meta-classifier 和 filter 都不适合你的要求，你可以直接用 attribute selection 类。

```
Instances data = ...  // from somewhere
// package weka.attributeSelection!
AttributeSelection attsel = new AttributeSelection();
CfsSubsetEval eval = new CfsSubsetEval();
GreedyStepwise search = new GreedyStepwise();
search.setSearchBackwards(true);
attsel.setEvaluator(eval);
attsel.setSearch(search);
attsel.SelectAttributes(data);
// obtain the attribute indices that were selected
int[] indices = attsel.selectedAttributes();
System.out.println(Utils.arrayToString(indices));
```

Note on Randomization

大多数机器学习方法，比较分类器和 clusterer，都会受据的顺序影响。用不同的随机数种子随机化数据集很可能得到不同的结果，比如 Explorer 或是一个分类器/clusterer 在只使用一个 seeded java.util.Random number generator。而 weka.core.Instances.getgetRandomNumberGenerator(int)，同样考虑了对样本的随机，如果不是用 10-fold cross-validation 10 次，并求平均结果，很有可能得到的是不同的结果。