/site/signin?goto=https%3A%2F%2Fweka.wikispaces.com%2FUse%2520WEKA%2520in%2520your%2520Java%2520code)

Wiki Home
Recent Changes
Pages and Files
Members
Search
Home
All pages
All tags
All files
Packages
FAQ
Not So FAQ
Troubleshooting
Learning Resources
Mailing List

Use WEKA in your Java code (/Use+WEKA+in+your+Java+code)

Edit
0 (/Use+WEKA+in+your+Java+code#discussion)
11 (/page/history/Use+WEKA+in+your+Java+code)
… (/page/menu/Use+WEKA+in+your+Java+code)

Table of Contents

The most common components you might want to use are

- *Instances* - your data
- *Filter* - for preprocessing the data
- *Classifier/Clusterer* - built on the processed data
- *Evaluating* - how good is the classifier/clusterer?
- *Attribute selection* - removing irrelevant attributes from your data

The following sections explain how to use them in your own code. A link to an **example class** can be found at the end of this page, under the Links section. The classifiers and filters always list their options in the Javadoc API

(book stable , developer version) specification.

You might also want to check out the Weka Examples collection, containing examples for the different versions of Weka. Another, more comprehensive, source of information is the chapter *Using the API* of the Weka manual for the stable-3.6

and
deveioper
version
(snapshots
and
releases
later
than
09/08/2009).

Instances

ARFF File

Pre 3.5.5 and 3.4.x

Reading from an ARFF file is straightforward:

```
ir
ir
ir
..
B

Ir
r
//
d
```

The class index indicates the target attribute used for classification. By default, in an ARFF file, it is the last attribute, which explains why it's set to numAttributes-1. You **must** set it if your instances are used as a parameter of a weka

function (e.g.,: weka.classifiers.Classifier.buildClassifier(data));

3.5.5 and newer

The DataSource class is not limited to ARFF files. It can also read CSV files and other formats (basically all file formats that Weka can import via its converters).

```
ir
..
D
Ir
//
```

Database

Reading from Databases is slightly more complicated, but still very easy. First, you'll have to modify your DatabaseUtils.props file to reflect your database connection. Suppose you want to connect to a MySQL server that is running

on the local machine on the default port 3306. The MySQL JDBC driver is called Connector/J . (The driver class is org.gjt.mm.mysql.Driver.) The database where your target data resides is called some_database. Since you're only reading, you can use the default user nobody without

a password. Your props file must contain the following lines:

```
jd
jd
```

Secondly, your Java code needs to look like this to load the data from the database:

```
im
im
..
In
q
q
q
//
//
In
```

Notes:

- Don't forget to add the JDBC driver to your CLASSPATH.
- For MS Access, you must use the JDBC-ODBC-bridge that is part of a JDK. The Windows databases article explains how to do this.
- InstanceQuery automatically converts VARCHAR database columns to NOMINAL attributes,

and
iong
TEXT
database
columns
to
STRING
attributes.
So
if
you
use
InstanceQuery
to
do
text
mining
against
text
that
appears
in
a
VARCHAR
column,
Weka
will
regard
such
text
as
nominal
values.
Thus
it
will
fail
to
tokenize
and
mine
that
text.
Use
the

NominalToString or StringToNominal filter (package weka.filters.unsupervised.attribute) to convert the attributes into the correct type.

Option handling

Weka schemes that implement the weka.core.OptionHandler interface, such as classifiers, clusterers, and filters, offer the following methods for setting and retrieving options:

- void setOptions(String[] options)

- String[] getOptions()

There are several ways of setting the options:

- Manually creating a String array:

```
S
o
o
```

- Using a single command-line string and using the splitOptions method of the weka.core.Utils class to turn it into an array:

```
S
```

- Using the

OptionsToCode.java
Details Download 2 KB

class to automatically turn a command line into code. Especially handy if the command line contains nested classes that have their own options, such as kernels for SMO:

```
ja
```

will generate output like this:

Also, the

OptionTree.java
Details Download 8 KB

tool allows you to view a nested options string, e.g., used at the command line, as a tree. This can help you spot nesting errors.

Filter

A filter

has two different properties:

- *supervised* or *unsupervised* either takes the class attribute into account or not
- *attribute*- or *instance*-based e.g., removing a certain attribute or removing instances that meet a certain condition

Most filters implement the OptionHandler interface, which means you

can set the options via a String array, rather than setting them each manually via set-methods. For example, if you want to remove the *first* attribute of a dataset, you need this filter

```
w
```

with this option

```
-
```

If you have an Instances object, called data, you can create and apply the filter like this:

```
ir
ir
ir
..
S
o
o
R
r
r
I
```

Filtering on-the-fly

The FilteredClassifier meta-classifier is

an easy way of filtering data on the fly. It removes the necessity of filtering the data before the classifier can be trained. Also, the data need not be passed through the trained filter again at prediction time. The following is an example of

using this meta-classifier with the Remove filter and J48 for getting rid of a numeric ID attribute in the data:

```
in
in
in
..
In
In
//
R
ru
//
J4
j4
//
F
fo
fo
//
fo
fo
```

```
    }
```

Other handy meta-schemes in Weka:

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

Batch filtering

On the command line, you can enable a second input/output pair (via -r and -s) with the -b option, in order to

process the second file with the same filter setup as the first one. Necessary, if you're using attribute selection or standardization - otherwise you end up with incompatible datasets. This is done fairly easy, since one initializes the filter only once with the setInputFormat(Instances)

method nameiy with the training set, and then applies the filter subsequently to the training set *and* the test set. The following example shows how to apply the Standardize filter to a train and a test set.

Calling conventions

The setInputFormat(Instances) method **always** has to be the last call before the filter is applied, e.g., with Filter.useFilter(Instances,Filter).

Why?

First, it is the convention for using filters and, secondly, lots of filters generate the header of the output

format in the setInputFormat(Instances) method with the currently set options (setting otpions *after* this call doesn't have any effect any more).

Classification

The necessary classes can be found in this package:

w

Building a Classifier

Batch

A Weka classifier is rather simple to train on a given dataset. E.g., we can train an unpruned C4.5 tree algorithm on a given dataset *data*. The training is done via the buildClassifier(Instances) method.

```
ir
..
S
o
J4
tr
tr
```

incremental
Classifiers
implementing
the
weka.classifiers.UpdateableClassifier
interface
can
be
trained
incrementally.
This
conserves
memory,
since
the
data
doesn't
have
to
be
loaded
into
memory
all
at
once.
See
the
Javadoc
of
this
interface
to
see
what
classifiers
are
implementing
it.

The

actual process of training an incremental classifier is fairly simple:

- Call buildClassifier(Instances) with the structure of the dataset (may or may not contain any actual data rows).
- Subsequently call the updateClassifier(Instance) method to feed the classifier new weka.core.Instance objects, one by one.

Here is an example using data from a weka.core.converters.ArffLoader to train weka.classifiers.bayes.NaiveBayesUpdateable:

```
//
A
lo
In
st

//
N
n
In
w
```

A working example is

IncrementalClassifier.java
Details Download 1 KB

.

Evaluating

Cross-validation

If you only have a training set and no test you might want to evaluate the classifier by using 10 times 10-fold cross-validation. This can be easily done via the Evaluation class. Here we *seed* the random selection of our folds for

the CV with 1 Check out the Evaluation class for more information about the statistics it produces.

```
ir
ir
..
E
e
```

Note: The classifier (in our example *tree*) should not be trained when handed over to the crossValidateModel method.

Why? if the classifier does not abide to the Weka convention that a classifier must be re-initialized every time the buildClassifier method is called (in other words: subsequent calls to the buildClassifier method always return the same results), you will get inconsistent and

worthless results. The crossValidateModel takes care of training and evaluating the classifier. (It creates a copy of the original classifier that you hand over to the crossValidateModel for each run of the cross-validation.)

Train/test set

In case you have

a dedicated test set, you can train the classifier and then evaluate it on this test set. In the following example, a J48 is instantiated, trained and then evaluated. Some statistics are printed to stdout:

```
ir
ir
ir
..
Ir
Ir
//
C
```

Statistics

Some methods for retrieving the results from the evaluation:

- nominal class
  - correct() - number of correctly classified instances (see also incorrect())
  - pctCorrect() - percentage of correctly classified instances (see also pctIncorrect())
  - kappa() - Kappa

statistics
- numeric class
  - correlationCoefficient() - the correlation coefficient
- general
  - meanAbsoluteError() - the mean absolute error
  - rootMeanSquaredError() - the root mean squared error
  - unclassified() - number of unclassified instances
  - pctUnclassified() - percentage of unclassified instances

If you want to have the exact same behavior

as from the command line, use this call:

```
ir
ir
..
S
o
o
S
```

ROC curves/AUC

Since Weka 3.5.1, you can also generate ROC curves/AUC with the predictions Weka recorded during testing. You can access these predictions

via the predictions() method of the Evaluation class. See the Generating ROC curve article for a full example of how to generate ROC curves.

Classifying instances

In case you have an unlabeled dataset that you want to classify with your

/site/signin?goto=https%3A%2F%2Fweka.wikispaces.com%2FUse%2520WEKA%2520in%2520your%2520Java%2520code)

newly trained classifier. You can use the following code snippet. It loads the file /some/where /unlabeled.arff, uses the previously built classifier tree to label the instances, and saves the labeled data as /some/where /labeled.arff.

```
ir
ir
ir
ir
ir
..
//
Ir
```

```
//
In

//
fo


}
//
B

w
w
w
w
```

Note on nominal classes:

- If you're interested in the distribution over all the classes, use the method distributionForInstance(Instance). This method returns a

double array with the probability for each class.
- The returned double value from classifyInstance (or the index in the array returned by distributionForInstance) is just the index for the string values in the attribute. That is, if you want the string representation for the class

label returned above. clsLabel then you can print it like this:

```
Sy
```

Clustering

Clustering is similar to classification. The necessary classes can be found in this package:

```
w
```

Building a Clusterer

Batch

A clusterer is built in much the same way as a classifier, but the buildClusterer(Instances) method instead of buildClassifier(Instances). The following code snippet shows how to build an EM clusterer with a maximum of 100 iterations.

```
ir
..
S
o
o
E
c
```

Incremental
Clusterers
implementing
the
weka.clusterers.UpdateableClusterer
interface
can
be
trained
incrementally
(available
since
version
3.5.4).
This
conserves
memory,
since
the
data
doesn't
have
to
be
loaded
into
memory
all
at
once.
See
the
Javadoc
for
this
interface
to
see
which

clusterers impiement it The actual process of training an incremental clusterer is fairly simple:

- Call buildClusterer(Instances) with the structure of the dataset (may or may not contain any actual data rows).
- Subsequently call the updateClusterer(Instance) method to feed the clusterer new weka.core.Instance objects,

one by one.
- Call updateFinished() after all Instance objects have been processed, for the clusterer to perform additional computations.

Here is an example using data from a weka.core.converters.ArffLoader to train weka.clusterers.Cobweb:

```
//
A
lo
In

//
C
cl
In
w
```

```
C
A
working
example
is
```

IncrementalClusterer.java
Details Download 1 KB

.

Evaluating

For evaluating a clusterer, you can use the ClusterEvaluation class. In this example, the number of clusters found is written to output:

```
ir
ir
..
C
```

```
C
c
e
S
```

Or, in the case of density based clusters , you can cross-validate the clusterer (Note: with MakeDensityBasedClusterer you can turn any clusterer into a density-based one):

```
ir
ir
ir
ir
..
Ir
D
d
```

/site/signin?goto=https%3A%2F%2Fweka.wikispaces.com%2FUse%2520WEKA%2520in%2520your%2520Java%2520code)

Or, if you want the same behavior/print-out from command line, use this call:

```
ir
ir
..
S
o
o
S
```

Clustering instances

The only difference with regard to classification is the method name.

Instead of classifyInstance(Instance) it is now clusterInstance(Instance). The method for obtaining the distribution is still the same, i.e., distributionForInstance(Instance).

Classes to clusters evaluation

If your data contains a class attribute and you want to check how well the generated clusters

fit the classes. You can perform a so-called *classes to clusters* evaluation. The Weka Explorer offers this functionality, and it's quite easy to implement. These are the necessary steps (complete source code:

ClassesToClusters.java
Details Download 2 KB

):

- load the data and set the

class attribute

```
In
d
```

- generate the *class-less* data to train the clusterer with

```
w
fi
fi
Ir
```

- train the clusterer, e.g., EM

```
E
//
c
```

- evaluate the clusterer with the data still containing the class attribute

- print the results of the evaluation to *stdout*

Attribute selection

There is no real need to use the attribute selection classes directly in your own code, since there are already a meta-

classifier and a filter available for applying attribute selection, but the low-level approach is still listed for the sake of completeness. The following examples all use CfsSubsetEval and GreedyStepwise (backwards). The code listed below is taken from the

AttributeSelectionTest.java

Details Download 3 KB

Meta-Classifier

The following meta-classifier performs a preprocessing step of attribute selection before the data gets presented to the base classifier (in the example here, this is J48).

```
I
A
C
C
s
J
c
c
c
A
```

Filter

The filter approach is straightforward: after setting up the filter, one just filters the data through the filter and obtains the reduced dataset.

Low-level

If neither the meta-classifier nor filter approach is suitable for your purposes, you can use the attribute selection classes themselves.

Note on

randomization

Most machine learning schemes, like classifiers and clusterers, are susceptible to the ordering of the data. Using a different seed for randomizing the data will most likely produce a different result. For example, the Explorer, or a classifier/clusterer run from the command

line uses only a seeded java.util.Random number generator, whereas the weka.core.Instances.getgetRandomNumberGenerator(int) (which the

WekaDemo.java

Details Download 6 KB

uses) also takes the data into account for seeding. Unless one runs 10-fold cross-validation 10 times and averages the results, one will most likely

get different results.

See also

- Weka Examples - pointer to collection of example classes
- Databases - for more information about using databases in Weka (includes ODBC, e.g., for MS Access)
- weka/experiment/DatabaseUtils.props - the database setup file
- Generating cross-validation folds

(Java approach) in case you want to run 10-fold cross-validation manually

- Generating classifier evaluation output manually - if you want to generate some of the evaluation statistics output manually
- Creating Instances on-the-fly - explains how to generate a weka.core.Instances object

from scratch
- Save Instances to an ARFF File - shows how to output a dataset
- Using the Experiment API

Examples

The following are a few sample classes for using various parts of the Weka API:

- **WekaDemo.java**
  Details Download 6 KB

(book , stable-3.6 ,

developer) iittie demo class that loads data from a file, runs it through a filter and trains/evaluates a classifier

ClusteringDemo.java
Details Download 2 KB

- (book , stable-3.6 , developer) a basic example for using the clusterer API

ClassesToClusters.java
Details Download 2 KB

- (book , stable-3.6 , developer) performs a

classes.to clusters evaluation like in the Explorer

AttributeSelectionTest.java
Details Download 3 KB

- (book , stable-3.6 , developer) example code for using the attribute selection API

M5PExample.java
Details Download 4 KB

- (book , stable-3.6 , developer) example using M5P to obtain data from database, train model, serialize it to

a file, and use this serialized model to make predictions again.

OptionsToCode.java
Details Download 2 KB

- (book , stable-3.6 , developer) turns a Weka command line for a scheme with options into Java code, correctly escaping quotes and backslashes.

OptionTree.java
Details Download 8 KB

- (book , stable-3.6 , developer)

displays nested Weka options as tree.

IncrementalClassifier.java
Details Download 1 KB

- (book , stable-3.6 , developer) Example class for how to train an incremental classifier (in this case, weka.classifiers.bayes.NaiveBayesUpdateable).

IncrementalClusterer.java
Details Download 1 KB

- (stable-3.6 , developer) Example class for how to train an incremental clusterer (in this

case,
weka.clusterers.Cobweb).

Links

- Weka API
  - Book version
  - Stable 3.6 version
  - Developer version