ADAMS

Advanced **D**ata mining **A**nd **M**achine learning **S**ystem

Module: adams-weka

Peter Reutemann

June 10, 2013

DRAFT

©2009-2013

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Except where otherwise noted, this work is licensed under
http://creativecommons.org/licenses/by-sa/3.0/

Contents

List of Figures

Chapter 1

Introduction

The *adams-weka* module offers most of the functionality found in WEKA [2]: pre-processing, classification and regression, clustering, attribute selection, data visualization and visualization of results/models. But it does not stop there: the module also contains other features for optimization, experiment generation that are not available from WEKA, be it Explorer or KnowledgeFlow. It is assumed that you are familiar with WEKA[1] and machine learning in general, as common terms are not explained again.

If you have used WEKA's KnowledgeFlow before, then you will have to forget (mostly) everything that you know about setting up workflows. ADAMS does things quite differently in comparison to the WEKA. Additionally, ADAMS offers a range of general purpose actors that allow you to go further.

The manual is split into several sections, with: *classification and regression* and *clustering* comprising the most important sections.

---

[1] If you haven't used WEKA before, check out the Data Mining book [3], which gives you a good introduction to machine learning, data mining and WEKA.

Chapter 2

Flow

The *adams-weka* module has a comprehensive set of actors and conversions that allow you to build powerful flows using WEKA's functionality. The following sections give a quick overview of available functionality. If you are interested in flow examples, check out chapters 3 and 4.

2.1 Conversions

This module offers additional schemes for the *Convert* transformer:

- *AdamsInstanceToWekaInstance* – converts an ADAMS instance into a WEKA one.
- *MatchWekaInstanceAgainstFileHeader* – uses a dataset header stored in a file to convert the string attributes of the instance passing through into nominal ones (and vice versa).
- *MatchWekaInstanceAgainstStorageHeader* – uses a dataset header obtained from storage to convert the string attributes of the instance passing through into nominal ones (and vice versa).
- *ReportToWekaInstance* – turns a *Report* object into a WEKA instance.
- *SpreadSheetToWekaInstances* – turns a spreadsheet object into a WEKA dataset.
- *WekaInstancesToSpreadSheet* – turns a WEKA dataset into a spreadsheet object.
- *WekaInstanceToAdamsInstance* – turns a WEKA instance into an ADAMS one.
- *WekaPredictionContainerToSpreadSheet* – generates a spreadsheet object from a predicition container (useful for display).

2.2 Conditions

The following boolean conditions, e.g., used in the *IfThenElse* or *Switch* control actors, are available:

- *AdamsInstanceCapabilities* – checks an ADAMS intance against the specified capabilities that it must satisfy.

- *WekaCapabilities* – checks a WEKA instance against the specified capabilities.
- *WekaClassification* (used in conjunction with *Switch*) – uses the returned classification index to determine which branch of the switch statement should be used; for all other control actors, the condition evaluates to "true" if an index is returnedl; condition works only with nominal classes.

2.3 Actors

The following sources are available:

- *WekaClassifierGenerator* – generates parameter sweeps for
- *WekaClassifierSetup* – outputs a single classifier setup.
- *WekaClustererGenerator* – generates parameter sweeps for
- *WekaClustererSetup* – outputs a single clusterer setup.
- *WekaDatabaseReader* – reads data from a database into WEKA's internal format.
- *WekaDataGenerator* – generates artificial data using WEKA's data generators.
- *WekaFilterGenerator* – generates parameter sweeps for filters.
- *WekaNewInstances* – simple source for generating empty datasets.

These transformers:

- *WekaAccumulatedError* – extracts all the errors collected during an evaluation, sorted according to magnitude and creates plot output, for comparing classifier performances (most useful for numeric classes).
- *WekaAggregatedEvaluations* – aggregates incoming Evaluation objects and forwards the current, aggregate state.
- *WekaAttributeIterator* – iterates through the names of a dataset and outputs them.
- *WekaTrainClassifier* – used for generating a trained model using a dataset.
- *WekaChooseAttributes* – allows the user to interactively select attributes to keep in a dataset.
- *WekaClassifierOptimizer* – applies a classifier optimizer (e.g., GridSearch or MultiSearch) to a dataset and then forwards the best (untrained) setup.
- *WekaClassifierRanker* – evaluates an array of classifier setups on a dataset and outputs the top X performing setups.
- *WekaClassifying* – uses a serialized (or global) model to make predictions on incoming data.
- *WekaClassSelector* – sets the class attribute in a dataset.
- *WekaTrainClusterer* – trains a cluster algorithm setup on a dataset.
- *WekaClustering* – applies a serialized (or global) model to incoming data.
- *WekaCrossValidationEvaluator* – performs cross-validation on an incoming dataset using a referenced classifier setup.
- *WekaCrossValidationSplot* – generates train/test set splits like cross-validation would generate.
- *WekaEvaluationSummary* – generates a summary for an Evaluation.

- *WekaEvaluationValuePicker* – retrieves a single statistic from an Evaluation.
- *WekaEvaluationValues* – generates a spreadsheet with the selected statistics from an Evaluation.
- *WekaExperiment* – executes a WEKA experiment, like in the Experimenter.
- *WekaExperimentEvaluation* – evaluates a WEKA experiment, generating text output of various sorts.
- *WekaExtractArray* – extracts a row or column from a WEKA dataset (using the internal format).
- *WekaFileReader* – reads any dataset that WEKA can handle, either outputs the header, the complete dataset or row-by-row.
- *WekaFilter* – applies a WEKA filter to the data.
- *WekaGetInstanceValue* – retrieves an attribute's value from a dataset row.
- *WekaInstanceBuffer* – buffers either incoming instance objects and outputs datasets or outputs instance objects when getting datasets.
- *WekaInstanceDumper* – for dumping dataset rows into files, one row at a time (ARFF or CSV).
- *WekaInstanceEvaluator* – adds an attribute with the value returned by an instance evaluator.
- *WekaInstanceFileReader* – outputs ADAMS instance objects.
- *WekaInstancesAppend* – creates one large dataset from multiple ones, by appending them one after the other.
- *WekaInstancesInfo* – outputs information on a dataset.
- *WekaInstancesMerge* – allows the merging of several datasets (side-by-side).
- *WekaInstanceStreamPlotGenerator* – generates plot containers from a range of attributes of instance objects passing through (i.e., you can plot several attributes in one go).
- *WekaModelReader* – reads a serialized model.
- *WekaMultiLabelSplitter* – splits a datasets with multiple class attributes ("multi-label") into ones with only a single class attribute.
- *WekaNewInstance* – creates an instance object with only missing values using a dataset as template.
- *WekaPredictionsToInstances* – turns WEKA predictions into a WEKA dataset (actual, predicted, etc).
- *WekaRandomSplit* – generates a random split of a dataset.
- *WekaRegexToRange* – generates a range string using a regular expression applied to the names of a dataset.
- *WekaRelationName* – simply outputs the name of the dataset.
- *WekaRenameRelation* – renames a dataset.
- *WekaSetInstanceValue* – sets a specific attribute value in an instance object.
- *WekaStoreInstance* – appends the passing through instance
- *WekaStreamFilter* – works the same as *WekaFilter* but only allows stream filters to be selected.
- *WekaSubsets* – splits dataset into subsets using the unique values of an

attribute to identify subsets.
- *WekaTestSetEvaluator* – evaluates a trained classifier on a dataset obtained from a global actor.
- *WekaTextDiectoryReader* – reads in a directory with the documents in the sub-directories representing different classes.
- *WekaTrainTestSetEvaluator* – evaluates a referenced classifier using the incoming train/test split.

And these sinks:

- *WekaClassifierErrors* – displays the errors of a classifier.
- *WekaCostCurve* – generates a cost curve.
- *WekaDatabaseWriter* – writes a dataset to a database.
- *WekaExperimentGenerator* – generates a WEKA experiment by adding the incoming classifier setups and writing it to disk.
- *WekaFileWriter* – writes a dataset to any file format that WEKA can handle.
- *WekaInstancesDisplay* – displays datasets in table format.
- *WekaInstanceViewer* – visualizes incoming WEKA or ADAMS instance objects the same way as the *Instance Explorer* tool does.
- *WekaModelWriter* – writes a model container or classifier/clusterer to disk.
- *WekaThresholdCurve* – displays threshold curves like, receiver-operator curve (ROC) or precision/recall.

2.4 Templates

Here are some templates that make the flow development for WEKA easier:

- *InstanceDumperVariable* – generates a variable for the *WekaInstanceDumper* actor which contains an ARFF/CSV filename prefix aligned with the flow's filename, i.e., the ARFF/CSV file will always get placed in the same location as the flow.

Chapter 3

Classification and Regression

WEKA's main strength lies in its large number of classification and regression schemes. Most of the documentation will cover this functionality therefore.

We start out with some basic WEKA functionality, like loading and preprocessing data, building models and evaluating them. That includes visualization of the results and models as well. After that we will cover more advanced features like learning curves, experiment generation and evaluation, optimization of classifiers and also the current provenance support in ADAMS.

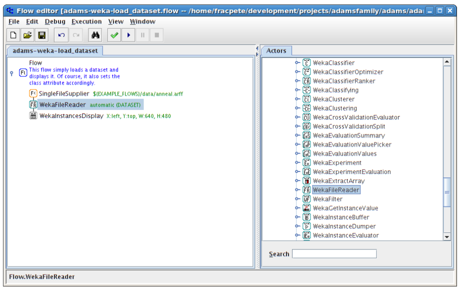
3.1 Basic

In this section we describe how to perform basic WEKA functionality that you are used to perform with the Explorer, but in the workflow context. Instead of having to repeat the same steps, like loading and preprocessing data, whenever you update your data, a flow allows you to define the steps apriori and then merely re-execute them time and time again. Also, flows make it very easy to *document* all the steps that you perform, not just merely recording what you are doing.

3.1.1 Loading data

Before we can build any models, we have to have data at hand, of course. So the first step will be to obtain data from somewhere, whether that is by loading a local dataset or by downloading a remote dataset.

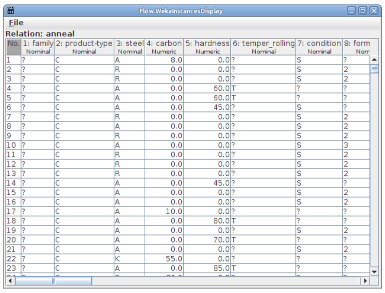
To start, we will be loading files that are stored locally. The actor used for loading datasets is the *WekaFileReader* transformer. This actor does not have an option for the file to load. Instead, it expects a file name, string or URL object to arrive at its input port. In order to supply a local file, we use the *SingleFileSupplier* source, which allows us to specify a single file that gets forwarded in the flow. If required, one can also use the *MultiFileSupplier* or *DirectoryLister* sources[1], which can forward multiple file names instead of just one. The latter one is especially handy, if the files are not known in advance, e.g., generated on the fly. In order to display the loaded data, we use the *WekaInstancesDisplay* sink actor, which displays the data in a nice tabular format. Figure 3.1 shows the flow for loading the dataset and Figure 3.2 the generated output.



Flow
This flow simply loads a dataset and displays it. Of course, it also sets the class attribute accordingly.
SingleFileSupplier ${EXAMPLE_FLOWS}/data/anneal.arff
WekaFileReader automatic (DATASET)
WekaInstancesDisplay X:left, Y:top, W:640, H:480

Flow.WekaFileReader

Figure 3.1: Flow for loading a local dataset.



Relation: anneal

No.	1: family Nominal	2: product-type Nominal	3: steel Nominal	4: carbon Numeric	5: hardness Numeric	6: temper_rolling Nominal	7: condition Nominal	8: form Nom
1	?	C	A	8.0	0.0	?	S	?
2	?	C	R	0.0	0.0	?	S	2
3	?	C	R	0.0	0.0	?	S	2
4	?	C	A	0.0	60.0	T	?	?
5	?	C	A	0.0	60.0	T	?	?
6	?	C	A	0.0	45.0	?	S	?
7	?	C	R	0.0	0.0	?	S	2
8	?	C	A	0.0	0.0	?	S	2
9	?	C	R	0.0	0.0	?	S	2
10	?	C	A	0.0	0.0	?	S	3
11	?	C	R	0.0	0.0	?	S	2
12	?	C	R	0.0	0.0	?	S	2
13	?	C	R	0.0	0.0	?	S	2
14	?	C	A	0.0	45.0	?	S	?
15	?	C	A	0.0	0.0	?	S	2
16	?	C	A	0.0	0.0	?	S	2
17	?	C	A	10.0	0.0	?	?	?
18	?	C	A	0.0	80.0	T	?	?
19	?	C	A	0.0	0.0	?	S	2
20	?	C	A	0.0	70.0	T	?	?
21	?	C	A	0.0	0.0	?	S	2
22	?	C	K	55.0	0.0	?	?	?
23	?	C	A	0.0	85.0	T	?	?

Figure 3.2: The dataset that got loaded from disk.

In this example[2] we let the *WekaFileReader* determine the correct file loader automatically, based on the file extension. If this automatic determination should fail, you can always check the "useCustomLoader" checkbox and then configure the appropriate loader yourself.

Another feature of this actor is the ability to output the dataset row by row (option "incremental"). This is very handy in case of very large files, where loading into memory could pose a problem. Even though the incremental feature

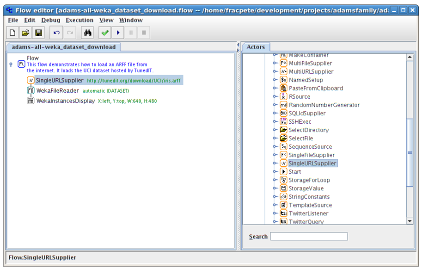
---

[1] adams-weka-crossvalidate_classifier_multiple_datasets.flow

[2] adams-weka-load_dataset.flow

works for any file type that WEKA can read, truly incremental, i.e., memory-efficient, loading is only possible if the underlying loader also supports incremental loading. In any other case, the dataset gets loaded fully into memory before being forwarded row by row.

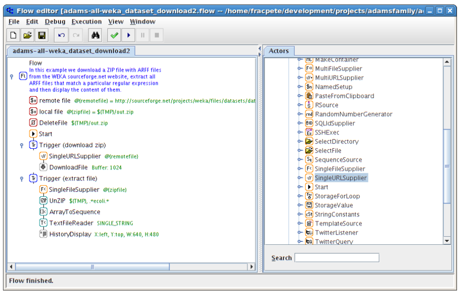
Nowadays, a lot of data is available online. Instead of relying on local files, one can use the flow also to download remote files. Some of the WEKA file loaders, like the *ArffLoader*, natively support the download via a URL. Figure 3.3 shows a flow[3] that downloads (and displays) an ARFF file available from a URL that was supplied by the *URLSupplier*. If the required dataset is encapsulated in an archive, e.g., a ZIP file and not just compressed with GZIP, then one has to download the archive first and extract the correct file before working with it. The flow[4] in Figure 3.4 downloads an archive from WEKA's sourceforge.net web site[5] using the *DownloadFile* sink and extracts all the datasets which filename fit a regular expression. The extracted files are then displayed in a *HistoryDisplay* sink.



Flow editor [adams-all-weka_dataset_download.flow -- /home/fracpete/development/projects/adamsfamily/ad

```
Flow
This flow demonstrates how to load an ARFF file from
the internet. It loads the UCI dataset hosted by TunedIT.
  SingleURLSupplier http://tunedit.org/download/UCI/iris.arff
  WekaFileReader automatic (DATASET)
  WekaInstancesDisplay X:left, Y:top, W:640, H:480
```

Figure 3.3: Flow for loading a local dataset.



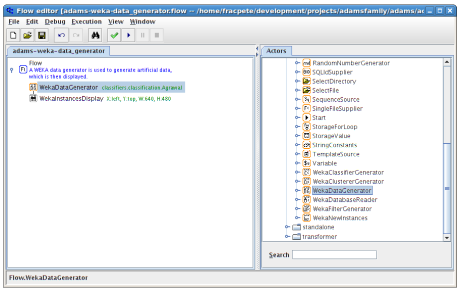
Flow editor [adams-all-weka_dataset_download2.flow -- /home/fracpete/development/projects/adamsfamily/a

```
Flow
In this example we download a ZIP file with ARFF files
from the WEKA sourceforge.net website, extract all
ARFF files that match a particular regular expression
and then display the content of them.
  remote file @{remotefile} = http://sourceforge.net/projects/weka/files/datasets/dat
  local file @{zipfile} = ${TMP}/out.zip
  DeleteFile ${TMP}/out.zip
  Start
  Trigger (download zip)
    SingleURLSupplier @{remotefile}
    DownloadFile Buffer: 1024
  Trigger (extract file)
    SingleFileSupplier @{zipfile}
    UnZIP ${TMP}, .*ecoli.*
    ArrayToSequence
    TextFileReader SINGLE_STRING
    HistoryDisplay X:left, Y:top, W:640, H:480
```

Flow finished.

Figure 3.4: The dataset that got loaded from disk.

Finally, artificial data can be generated within ADAMS as well. Using the *WekaDataGenerator* source, any WEKA data generator can be used to output data. The flow[6] depicted in Figure 3.5 generates a small dataset using the "Agrawal" data generator.



Flow editor [adams-weka-data_generator.flow -- /home/fracpete/development/projects/adamsfamily/adams/ad

```
Flow
A WEKA data generator is used to generate artificial data,
which is then displayed.
  WekaDataGenerator classifiers.classification.Agrawal
  WekaInstancesDisplay X:left, Y:top, W:640, H:480
```

Figure 3.5: Flow for generating and displaying an artificial dataset.

---

[3] adams-all-weka_dataset_download.flow

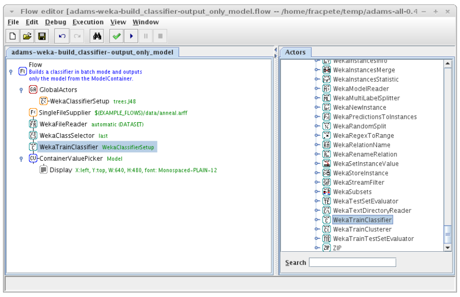
[4] adams-all-weka_dataset_download2.flow

[5] WEKA on sourceforge.net: `http://sourceforge.net/projects/weka/`

[6] adams-weka-data_generator.flow

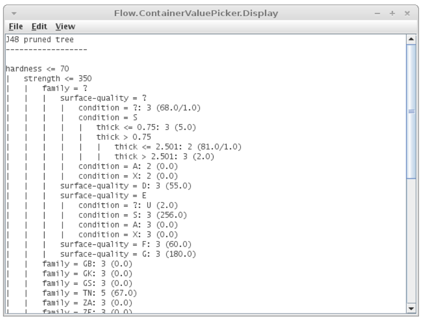
3.1.2 Building models

After having sorted out the loading of the data, it is time to check out how to build models. Since we are using supervised algorithms, we have to make sure that the datasets have a class attribute set. The *WekaClassSelector* actor allows the setting of the class attribute, in the default setting it simply uses the last attribute as the class attribute. With the *WekaTrainClassifier* actor you can choose a globally defined classifier to be built. You define a global classifier by adding a *WekaClassifierSetup* source to the *GlobalActors* standalone, which you then reference in your *WekaTrainClassifier* actor. By default, the *WekaTrainClassifier* actor outputs a container that comprises the built model and the header of the training set. In order to extract either of the container items, you need to use the *ContainerValuePicker* control actor. Figure 3.6 demonstrates how to train a J48 classifier on dataset and then displaying the built model (see Figure 3.7)[7].



```
Flow
Builds a classifier in batch mode and outputs
only the model from the ModelContainer.
  GlobalActors
    WekaClassifierSetup  trees.J48
  SingleFileSupplier  ${EXAMPLE_FLOWS}/data/anneal.arff
  WekaFileReader  automatic (DATASET)
  WekaClassSelector  last
  WekaTrainClassifier  WekaClassifierSetup
  ContainerValuePicker  Model
    Display  X:left, Y:top, W:640, H:480, font: Monospaced-PLAIN-12
```

Figure 3.6: Flow for building J48 model on a dataset and outputting the model.



```
J48 pruned tree
------------------

hardness <= 70
|   strength <= 350
|   |   family = ?
|   |   |   surface-quality = ?
|   |   |   |   condition = ?: 3 (68.0/1.0)
|   |   |   |   condition = S
|   |   |   |   |   thick <= 0.75: 3 (5.0)
|   |   |   |   |   thick > 0.75
|   |   |   |   |   |   thick <= 2.501: 2 (81.0/1.0)
|   |   |   |   |   |   thick > 2.501: 3 (2.0)
|   |   |   |   condition = A: 2 (0.0)
|   |   |   |   condition = X: 2 (0.0)
|   |   |   surface-quality = D: 3 (55.0)
|   |   |   surface-quality = E
|   |   |   |   condition = ?: U (2.0)
|   |   |   |   condition = S: 3 (256.0)
|   |   |   |   condition = A: 3 (0.0)
|   |   |   |   condition = X: 3 (0.0)
|   |   |   surface-quality = F: 3 (60.0)
|   |   |   surface-quality = G: 3 (180.0)
|   |   family = GB: 3 (0.0)
|   |   family = GK: 3 (0.0)
|   |   family = GS: 3 (0.0)
|   |   family = TN: 5 (67.0)
|   |   family = ZA: 3 (0.0)
```

Figure 3.7: J48 model output.

A built model can be saved to disk (and then re-used later) using the *WekaModelWriter*. The file generated can also be loaded in the WEKA Explorer again and applied to another test set there[8].

3.1.3 Preprocessing

A very important, but often underrated step is preprocessing. Unless your data is properly cleaned up and in the right format, your models will not be very meaningful. Preprocessing steps can be done within the flow using the *WekaFilter* transformer, which wraps around a single WEKA filter. One either chains multiple actors together or uses the *weka.filters.MultiFilter* meta-filter to executed several filter sequentially in a single actor.

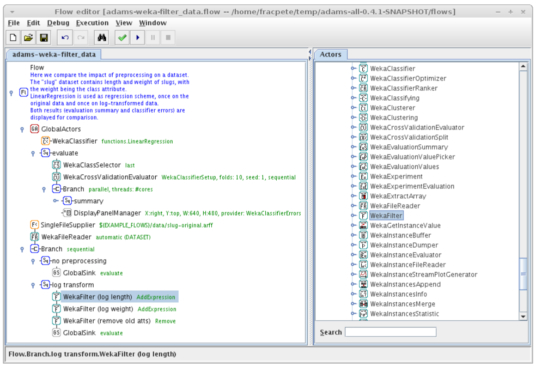
In Figure 3.8 we are investigating the impact of preprocessing on the “slug” dataset [4]. The flow[9] cross-validates *LinearRegression* on the original and log-transformed data. The log-transformed data is generated by applying the *AddExpression* filter on each of the two attributes of the dataset and then deleting the original ones. In each case, original or preprocessed, it displays the evaluation summary and classifier errors.

---

[7] adams-weka-build_classifier-output_only_model.flow

[8] adams-weka-build_classifier-save_model.flow

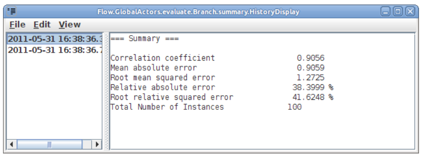
[9] adams-weka-filter_data.flow



```
Flow
Here we compare the impact of preprocessing on a dataset.
The "slug" dataset contains length and weight of slugs, with
the weight being the class attribute.
LinearRegression is used as regression scheme, once on the
original data and once on log-transformed data.
Both results (evaluation summary and classifier errors) are
displayed for comparison.
  GlobalActors
    WekaClassifier functions.LinearRegression
    evaluate
      WekaClassSelector last
      WekaCrossValidationEvaluator WekaClassifierSetup, folds: 10, seed: 1, sequential
      Branch parallel, threads: #cores
        summary
        DisplayPanelManager X:right, Y:top, W:640, H:480, provider: WekaClassifierErrors
  SingleFileSupplier ${EXAMPLE_FLOWS}/data/slug-original.arff
  WekaFileReader automatic (DATASET)
  Branch sequential
    no preprocessing
      GlobalSink evaluate
    log transform
      WekaFilter (log length) AddExpression
      WekaFilter (log weight) AddExpression
      WekaFilter (remove old atts) Remove
      GlobalSink evaluate
Flow.Branch.log transform.WekaFilter (log length)
```

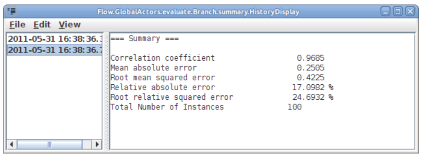
Figure 3.8: Flow for comparing results generated from original and preprocessed "slug" data [4].

Figures 3.9 and 3.10 show the evaluation summary, for the original and the log-transformed data. The log-transformed dataset gets not only a better correlation coefficient, but also smaller errors.



```
=== Summary ===

Correlation coefficient                  0.9056
Mean absolute error                      0.9059
Root mean squared error                  1.2725
Relative absolute error                 38.3999 %
Root relative squared error             41.6248 %
Total Number of Instances              100
```

Figure 3.9: Evaluation summary on "slug" dataset (original).



```
=== Summary ===

Correlation coefficient                  0.9685
Mean absolute error                      0.2505
Root mean squared error                  0.4225
Relative absolute error                 17.0982 %
Root relative squared error             24.6932 %
Total Number of Instances              100
```

Figure 3.10: Evaluation summary on "slug" dataset (log-transformed).

Figures 3.11 and 3.12 display the classifier errors. It is obvious from the funy log-shaped curve, that LinearRegression built on the original data is not a very good model. Something that is not so obvious by just looking at the correlation coefficient: 0.9056 is not bad.

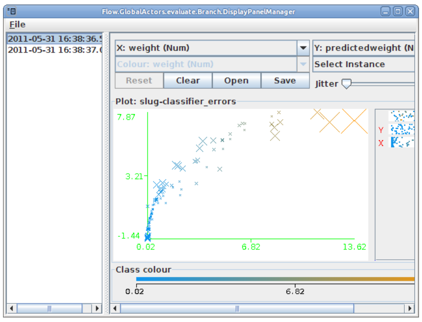


Figure 3.11: Classifier errors on "slug" dataset (original).

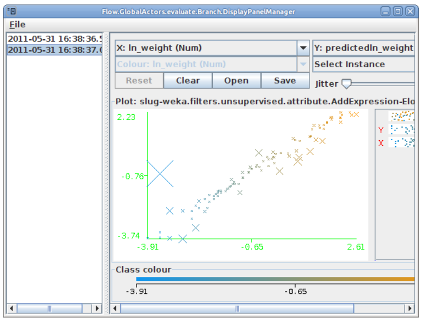


Figure 3.12: Classifier errors on "slug" dataset (log-transformed).

This flow can be quickly extended to accommodate other preprocessing techniques, all very easily comparable in the graphical output.

In this example the preprocessing was rather specific. On the other hand, if your are working mainly in a particular data domain, like spectral analysis of some kind, then certain preprocessing steps will always be same. In this case, it makes sense to store these externally in a *preprocessing library* which you then link to using external actors (see manual for the *adams-core* module for more details). This reduces duplication and you will only have to update the preprocessing step in a single location.

Instead of batch-filtering data, you can also filter streams of *weka.core.Instance* objects, using the *WekaStreamFilter* transformer. This filter offers a subset of WEKA's filters, which don't need a batch of data to be initialized with before being able to process data.

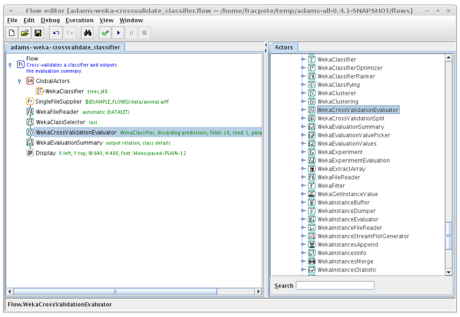
3.1.4 Evaluation

Knowing how to build a model is good, but how can you tell whether the model that you built is any good? Evaluation is the key to unlock this mystery. ADAMS offers several types of evaluations:

- *Cross-validation* – if you only have a single dataset.
- *Test set evaluation* – evaluating an already trained classifier with a separate dataset.
- *Train/test set evaluation* – training and evaluating a classifier with a training and test set. This can be either achieved using a *RandomSplit* actor or reading two separate files from disk.

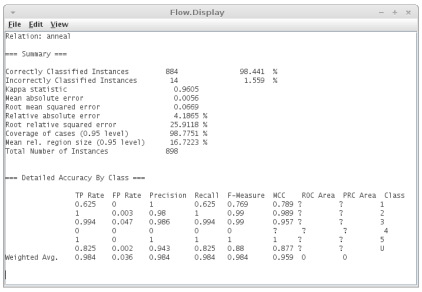
Cross-validation

We start with cross-validation, which is probably the most used type of evaluation. The *WekaCrossValidationEvaluator* transformer is used for cross-validation. In order to get around ADAMS' limitation of allowing only one input, the *WekaCrossValidationEvaluator* actor takes a dataset as input and obtains the classifier to evaluate from a *global actor*. This approach hides *how* the classifier is obtained, whether it is a simple *WekaClassifierSetup* definition or a more complex scheme for outputting a *Classifier* object (e.g., loading it from a serialized model file). Figure 3.13 shows a flow[10] with simple cross-validation using a global *WekaClassifierSetup* to obtain the classifier object from.



```
Flow
  Cross-validates a classifier and outputs
  the evaluation summary.
    GlobalActors
      WekaClassifier trees.J48
    SingleFileSupplier ${EXAMPLE_FLOWS}/data/anneal.arff
    WekaFileReader automatic (DATASET)
    WekaClassSelector last
    WekaCrossValidationEvaluator WekaClassifier, discarding predictions, folds: 10, seed: 1, para
    WekaEvaluationSummary output relation, class details
    Display X:left, Y:top, W:640, H:480, font: Monospaced-PLAIN-12
```

Figure 3.13: Cross-validating a classifier and outputting the summary.



```
Relation: anneal

=== Summary ===

Correctly Classified Instances         884               98.441  %
Incorrectly Classified Instances        14                1.559  %
Kappa statistic                          0.9605
Mean absolute error                      0.0056
Root mean squared error                  0.0669
Relative absolute error                  4.1865 %
Root relative squared error             25.9118 %
Coverage of cases (0.95 level)          98.7751 %
Mean rel. region size (0.95 level)      16.7223 %
Total Number of Instances              898


=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall  F-Measure  MCC    ROC Area  PRC Area  Class
                 0.625    0        1          0.625   0.769      0.789 ?         ?         1
                 1        0.003    0.98       1       0.99       0.989 ?         ?         2
                 0.994    0.047    0.986      0.994   0.99       0.957 ?         ?         3
                 0        0        0          0       0          ?      ?         ?         4
                 1        0        1          1       1          1     ?         ?         5
                 0.825    0.002    0.943      0.825   0.88       0.877 ?         ?         U
Weighted Avg.    0.984    0.036    0.984      0.984   0.984      0.959  0         0
```

Figure 3.14: Summary output of a cross-validated classifier.

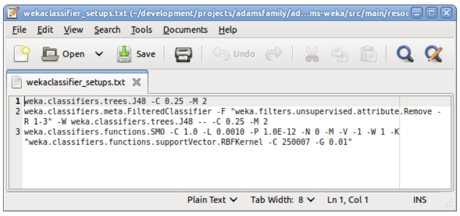
Most of the time, you don't just want to test a single classifier, but several ones. With ADAMS you can, for instance, load classifier command-lines from a text file and then evaluate them one after the after[11]. Reading the text file (see Figure 3.15) is fairly straight-forward, using the *TextFileReader* transformer.

For updating the global classifier's set up, we need to attach a variable to the global *WekaClassifierSetup* actor's "classifier" option and update this variable with each set up that we are reading from the text file using the *SetVariable* transformer. This update of the classifier set up has to happen before we are triggering the cross-validation. Figures 3.16 and 3.17 show the full flow and the generated output, when reading in three set ups from a text file (J48, filtered J48, SMO).

---

[10] adams-weka-crossvalidate_classifier.flow

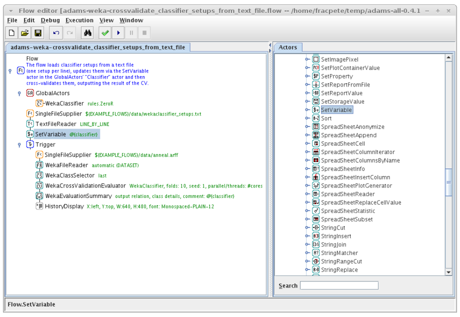
[11] adams-weka-crossvalidate_classifier_setups_from_text_file.flow



```
weka.classifiers.trees.J48 -C 0.25 -M 2
weka.classifiers.meta.FilteredClassifier -F "weka.filters.unsupervised.attribute.Remove -
R 1-3" -W weka.classifiers.trees.J48 -- -C 0.25 -M 2
weka.classifiers.functions.SMO -C 1.0 -L 0.0010 -P 1.0E-12 -N 0 -M -V -1 -W 1 -K
"weka.classifiers.functions.supportVector.RBFKernel -C 250007 -G 0.01"
```

wekaclassifier_setups.txt (~/development/projects/adamsfamily/ad...ms-weka/src/main/resou
File Edit View Search Tools Documents Help
Open
Save
Undo
wekaclassifier_setups.txt
Plain Text
Tab Width: 8
Ln 1, Col 1
INS

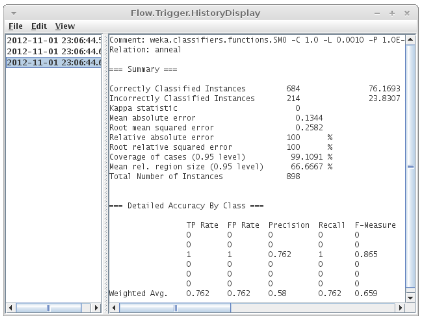
Figure 3.15: Text file with command-lines of various classifiers.



```
Flow
The flow loads classifier setups from a text file
(one setup per line), updates them via the SetVariable
actor in the GlobalActors' "Classifier" actor and then
cross-validates them, outputting the result of the CV.
  GlobalActors
    WekaClassifier rules.ZeroR
  SingleFileSupplier ${EXAMPLE_FLOWS}/data/wekaclassifier_setups.txt
  TextFileReader LINE_BY_LINE
  SetVariable @{classifier}
  Trigger
    SingleFileSupplier ${EXAMPLE_FLOWS}/data/anneal.arff
    WekaFileReader automatic (DATASET)
    WekaClassSelector last
    WekaCrossValidationEvaluator WekaClassifier, folds: 10, seed: 1, parallel/threads: #cores
    WekaEvaluationSummary output relation, class details, comment: @{classifier}
    HistoryDisplay X:left, Y:top, W:640, H:480, font: Monospaced-PLAIN-12
```

Flow editor [adams-weka-crossvalidate_classifier_setups_from_text_file.flow -- /home/fracpete/temp/adams-all-0.4.1
File Edit Debug Execution View Window
adams-weka-crossvalidate_classifier_setups_from_text_file
Actors
Search
Flow.SetVariable

Figure 3.16: Cross-validating classifier set ups read from a text file and displaying the evaluation summaries.

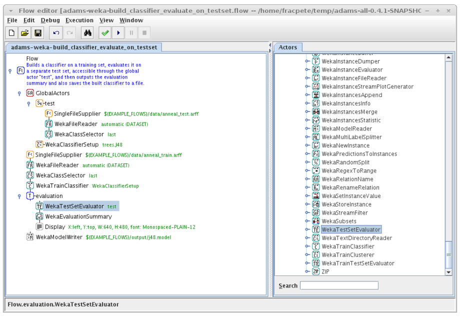


```
Comment: weka.classifiers.functions.SMO -C 1.0 -L 0.0010 -P 1.0E-
Relation: anneal

=== Summary ===

Correctly Classified Instances         684               76.1693
Incorrectly Classified Instances       214               23.8307
Kappa statistic                          0
Mean absolute error                      0.1344
Root mean squared error                  0.2582
Relative absolute error                100      %
Root relative squared error            100      %
Coverage of cases (0.95 level)          99.1091 %
Mean rel. region size (0.95 level)      66.6667 %
Total Number of Instances              898


=== Detailed Accuracy By Class ===

                 TP Rate   FP Rate   Precision   Recall   F-Measure
                 0         0         0           0        0
                 0         0         0           0        0
                 1         1         0.762       1        0.865
                 0         0         0           0        0
                 0         0         0           0        0
                 0         0         0           0        0
Weighted Avg.    0.762     0.762     0.58        0.762    0.659
```

Figure 3.17: Summary outputs of cross-validated classifiers.

Test set evaluation

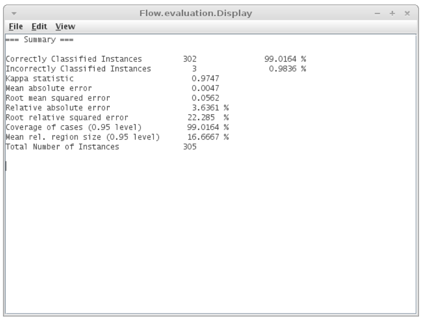
Simply testing a built classifier on a test set is useful when you are always intending to save the generated model to a file, but also want to keep an eye on the performance. In this case, you can very easily extend your current flow for building and saving the model. First, add a global actor that loads the separate training set from disk. Second, add a *Tee* control actor that performs the evaluation using the *WekaTestSetEvaluator* and *WekaEvaluationSummary* transformers and a *Display* sink for showing the results[12]. The full flow and the generated output are shown in Figures 3.18 and 3.19.



```
Flow
Builds a classifier on a training set, evaluates it on
a separate test set, accessible through the global
actor "test", and then outputs the evaluation
summary and also saves the built classifier to a file.
  GlobalActors
    test
      SingleFileSupplier ${EXAMPLE_FLOWS}/data/anneal_test.arff
      WekaFileReader automatic (DATASET)
      WekaClassSelector last
    WekaClassifierSetup trees.J48
  SingleFileSupplier ${EXAMPLE_FLOWS}/data/anneal_train.arff
  WekaFileReader automatic (DATASET)
  WekaClassSelector last
  WekaTrainClassifier WekaClassifierSetup
  evaluation
    WekaTestSetEvaluator test
    WekaEvaluationSummary
    Display X:left, Y:top, W:640, H:480, font: Monospaced-PLAIN-12
  WekaModelWriter ${EXAMPLE_FLOWS}/output/j48.model
```

Flow editor [adams-weka-build_classifier_evaluate_on_testset.flow -- /home/fracpete/temp/adams-all-0.4.1-SNAPSHC
File Edit Debug Execution View Window
adams-weka-build_classifier_evaluate_on_testset
Actors
Search
Flow.evaluation.WekaTestSetEvaluator

Figure 3.18: Flow for evaluating built classifier on a separate test set.



```
=== Summary ===

Correctly Classified Instances         302               99.0164 %
Incorrectly Classified Instances         3                0.9836 %
Kappa statistic                          0.9747
Mean absolute error                      0.0047
Root mean squared error                  0.0562
Relative absolute error                  3.6361 %
Root relative squared error             22.285  %
Coverage of cases (0.95 level)          99.0164 %
Mean rel. region size (0.95 level)      16.6667 %
Total Number of Instances              305
```

Figure 3.19: Summary output of classifier evaluated on separate test set.

---

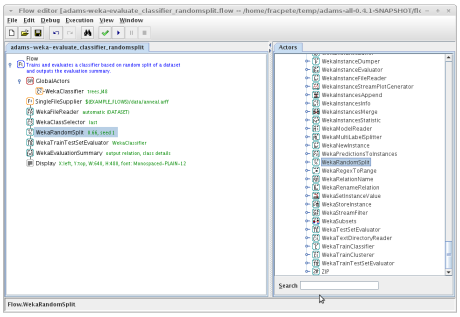
[12] adams-weka-build_classifier_evaluate_on_testset.flow

Train/test set evaluation

An evaluation using separate train and test set can be used, if you don't want to keep the evaluated model, but you are only interested in the evaluation output. The evaluation actor in this case is the *WekaTrainTestSetEvaluator* transformer. This actor accepts *WekaTrainTestSetContainer* data tokens. To generate this container you have several options:

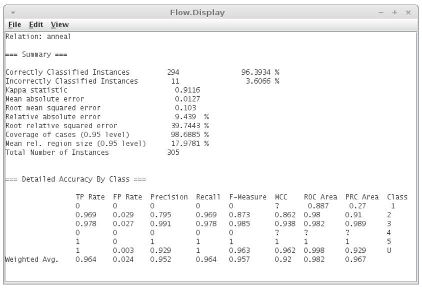
- *WekaRandomSplit* – splits a single dataset into a train and test set, based on the percentage supplied by the user.
- *WekaCrossValidationSplit* – Generates train/test splits like they occur in cross-validation. Useful, if you want to inspect the various models built during cross-validation, not just the summary.
- *MakeContainer* – manually generating a container from two individually loaded datasets.

Figures 3.20 and 3.21 show how to use the *RandomSplit* actor in the evaluation process[13]. For simulating cross-validation, simply exchange the *WekaRandomSplit* actor with a *WekaCrossValidationSplit* one (you might also want to change from *Display* to *HistoryDisplay*, to keep better track of the various evaluations).



```
Flow
Trains and evaluates a classifier based on random split of a dataset
and outputs the evaluation summary.
  GlobalActors
    WekaClassifier  trees.J48
  SingleFileSupplier  ${EXAMPLE_FLOWS}/data/anneal.arff
  WekaFileReader  automatic (DATASET)
  WekaClassSelector  last
  WekaRandomSplit  0.66, seed 1
  WekaTrainTestSetEvaluator  WekaClassifier
  WekaEvaluationSummary  output relation, class details
  Display  X:left, Y:top, W:640, H:480, font: Monospaced-PLAIN-12
```

Figure 3.20: Flow for building/evaluating classifier on a random split.



```
Relation: anneal

=== Summary ===

Correctly Classified Instances         294               96.3934 %
Incorrectly Classified Instances        11                3.6066 %
Kappa statistic                          0.9116
Mean absolute error                      0.0127
Root mean squared error                  0.103 
Relative absolute error                  9.439  %
Root relative squared error             39.7443 %
Coverage of cases (0.95 level)          98.6885 %
Mean rel. region size (0.95 level)      17.9781 %
Total Number of Instances              305     


=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall  F-Measure  MCC    ROC Area  PRC Area  Class
                 0        0        0          0       0          ?       0.887     0.27      1
                 0.969    0.029    0.795      0.969   0.873      0.862  0.98      0.91      2
                 0.978    0.027    0.991      0.978   0.985      0.938  0.982     0.989     3
                 0        0        0          0       0          ?      ?         ?         4
                 1        0        1          1       1          1      1         1         5
                 1        0.003    0.929      1       0.963      0.962  0.998     0.929     U
Weighted Avg.    0.964    0.024    0.952      0.964   0.957      0.92   0.982     0.967     
```

Figure 3.21: Summary output of classifier built/evaluated on random split.

Figures 3.22 and 3.23 display the flow[14] for manually creating a container using the general purpose *MakeContainer* source actor. In order to assemble a container, you need to know **what** type of container you want to create (the type is normally listed in the "Help" of an actor), **where** to obtain the data from (i.e., the global actors) and **how** to store the data (i.e., under which name in the container).

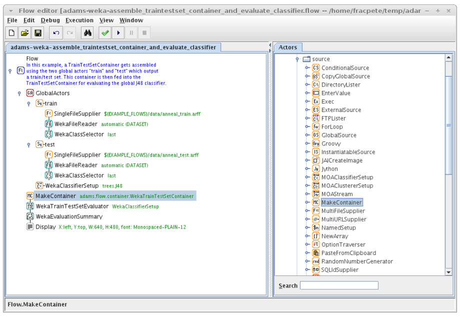
Visualization

You have already encountered the display of the classifier errors (in Figure 3.11). The sink for displaying these errors is *WekaClassifierErrors*, which takes an *Evaluation* object as input. If you want to evaluate and display multiple classifiers then you have to use the *DisplayPanelManager* with the *WekaClas-*

---

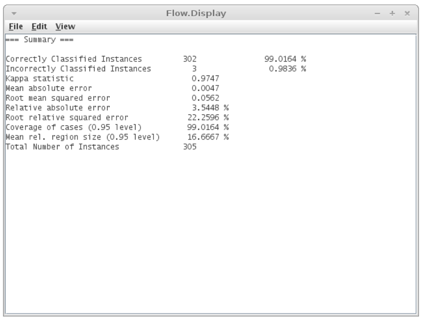
[13] adams-weka-evaluate_classifier_randomsplit.flow

[14] adams-weka-assemble_traintestset_container_and_evaluate_classifier.flow



```
Flow
In this example, a TrainTestSetContainer gets assembled
using the two global actors "train" and "test" which output
a train/test set. This container is then fed into the
TrainTestSetContainer for evaluating the global J48 classifier.
  GlobalActors
    -train
      SingleFileSupplier  ${EXAMPLE_FLOWS}/data/anneal_train.arff
      WekaFileReader  automatic (DATASET)
      WekaClassSelector  last
    -test
      SingleFileSupplier  ${EXAMPLE_FLOWS}/data/anneal_test.arff
      WekaFileReader  automatic (DATASET)
      WekaClassSelector  last
    -WekaClassifierSetup  trees.J48
  MakeContainer  adams.flow.container.WekaTrainTestSetContainer
  WekaTrainTestSetEvaluator  WekaClassifierSetup
  WekaEvaluationSummary
  Display  X:left, Y:top, W:640, H:480, font: Monospaced-PLAIN-12
```

Figure 3.22: Flow for evaluating classifier on separate train/test set.

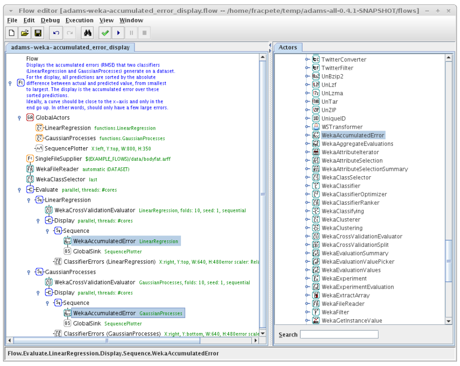


```
=== Summary ===

Correctly Classified Instances         302               99.0164 %
Incorrectly Classified Instances         3                0.9836 %
Kappa statistic                          0.9747
Mean absolute error                      0.0047
Root mean squared error                  0.0562
Relative absolute error                  3.5448 %
Root relative squared error             22.2596 %
Coverage of cases (0.95 level)          99.0164 %
Mean rel. region size (0.95 level)      16.6667 %
Total Number of Instances              305
```

Figure 3.23: Summary output of classifier evaluated on separate train/test set.

sifierErrors actor as "panelProvider". The *DisplayPanelManager* actor offers a history of generated panels, like the *HistoryDisplay* does for plain text.

Another interesting visualization is the *WekaAccumulatedError* transformer. This transformer takes also an *Evaluation* object and then turns it into a special sequence of plot containers: it creates a sequence of the prediction errors that were obtained during an evaluation and outputs them sorted, from smallest to largest[15]. The Figures 3.24 and 3.25 show the flow and the generated output respectively. As you can see from the graph, GaussianProcesses generates consistently larger errors than LinearRegression, which only seems to have a few big outliers (steep increase at the end).



```
Flow
Displays the accumulated errors (RMSE) that two classifiers
(LinearRegression and GaussianProcesses) generate on a dataset.
For the display, all predictions are sorted by the absolute
difference between actual and predicted value, from smallest
to largest. The display is the accumulated error over these
sorted predictions.
Ideally, a curve should be close to the x-axis and only in the
end go up. In other words, should only have a few large errors.
  GlobalActors
    -LinearRegression  functions.LinearRegression
    -GaussianProcesses  functions.GaussianProcesses
    -SequencePlotter  X:left, Y:top, W:800, H:350
  SingleFileSupplier  ${EXAMPLE_FLOWS}/data/bodyfat.arff
  WekaFileReader  automatic (DATASET)
  WekaClassSelector  last
  -Evaluate  parallel, threads: #cores
    -LinearRegression
      WekaCrossValidationEvaluator  LinearRegression, folds: 10, seed: 1, sequential
      -Display  parallel, threads: #cores
        -Sequence
          WekaAccumulatedError  LinearRegression
          GlobalSink  SequencePlotter
        -ClassifierErrors (LinearRegression)  X:right, Y:top, W:640, H:480error scaler: Rela
    -GaussianProcesses
      WekaCrossValidationEvaluator  GaussianProcesses, folds: 10, seed: 1, sequential
      -Display  parallel, threads: #cores
        -Sequence
          WekaAccumulatedError  GaussianProcesses
          GlobalSink  SequencePlotter
        -ClassifierErrors (GaussianProcesses)  X:right, Y:bottom, W:640, H:480error scale
```

Figure 3.24: Flow for displaying the "accumulated error" of a two classifiers.

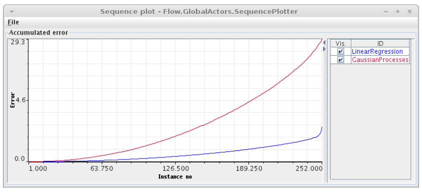


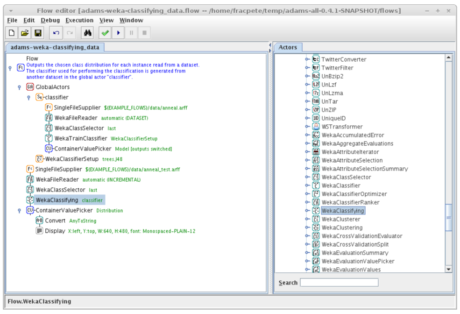
Figure 3.25: The "accumulated error" of LinearRegression and GaussianProcesses.

---

[15] adams-weka-accumulated_error_display.flow

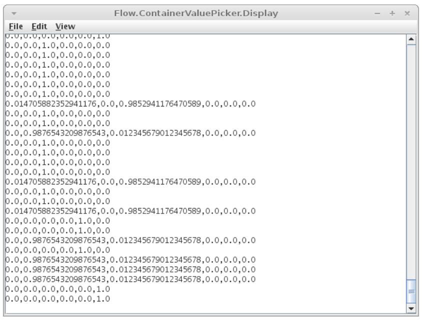
3.1.5 Making predictions

Of course, building models is only part of the picture. You will want to use this model as well and make predictions with it. The actor for making predictions on incoming data (i.e., single instance objects) is the *WekaClassifying* actor. This actor can either use a serialized model or a global actor that generates a trained classifier. The flow[16] in Figure 3.26 uses the global actor approach, training a classifier on a training set and then performing classifications on a test set, with the class distributions shown on screen (see Figure 3.27).



```
Flow
Outputs the chosen class distribution for each instance read from a dataset.
The classifier used for performing the classification is generated from
another dataset in the global actor "classifier".
  GlobalActors
    -classifier
      SingleFileSupplier ${EXAMPLE_FLOWS}/data/anneal.arff
      WekaFileReader automatic (DATASET)
      WekaClassSelector last
      WekaTrainClassifier WekaClassifierSetup
      -ContainerValuePicker Model [outputs switched]
    -WekaClassifierSetup trees.J48
  SingleFileSupplier ${EXAMPLE_FLOWS}/data/anneal_test.arff
  WekaFileReader automatic (INCREMENTAL)
  WekaClassSelector last
  WekaClassifying classifier
  -ContainerValuePicker Distribution
    Convert AnyToString
    Display X:left, Y:top, W:640, H:480, font: Monospaced-PLAIN-12
```

Figure 3.26: Flow for classifying new data and outputting the class distributions.



```
0.0,0.0,0.0,0.0,0.0,1.0
0.0,0.0,1.0,0.0,0.0,0.0
0.0,0.0,1.0,0.0,0.0,0.0
0.0,0.0,1.0,0.0,0.0,0.0
0.0,0.0,1.0,0.0,0.0,0.0
0.0,0.0,1.0,0.0,0.0,0.0
0.0,0.0,1.0,0.0,0.0,0.0
0.014705882352941176,0.0,0.9852941176470589,0.0,0.0,0.0
0.0,0.0,1.0,0.0,0.0,0.0
0.0,0.0,1.0,0.0,0.0,0.0
0.0,0.9876543209876543,0.012345679012345678,0.0,0.0,0.0
0.0,0.0,1.0,0.0,0.0,0.0
0.0,0.0,1.0,0.0,0.0,0.0
0.0,0.0,1.0,0.0,0.0,0.0
0.0,0.0,1.0,0.0,0.0,0.0
0.014705882352941176,0.0,0.9852941176470589,0.0,0.0,0.0
0.0,0.0,1.0,0.0,0.0,0.0
0.0,0.0,1.0,0.0,0.0,0.0
0.014705882352941176,0.0,0.9852941176470589,0.0,0.0,0.0
0.0,0.0,0.0,0.0,1.0,0.0
0.0,0.0,0.0,0.0,1.0,0.0
0.0,0.9876543209876543,0.012345679012345678,0.0,0.0,0.0
0.0,0.0,0.0,0.0,1.0,0.0
0.0,0.9876543209876543,0.012345679012345678,0.0,0.0,0.0
0.0,0.9876543209876543,0.012345679012345678,0.0,0.0,0.0
0.0,0.9876543209876543,0.012345679012345678,0.0,0.0,0.0
0.0,0.0,0.0,0.0,0.0,1.0
0.0,0.0,0.0,0.0,0.0,1.0
```

Figure 3.27: The generated class distributions for the new data.

---

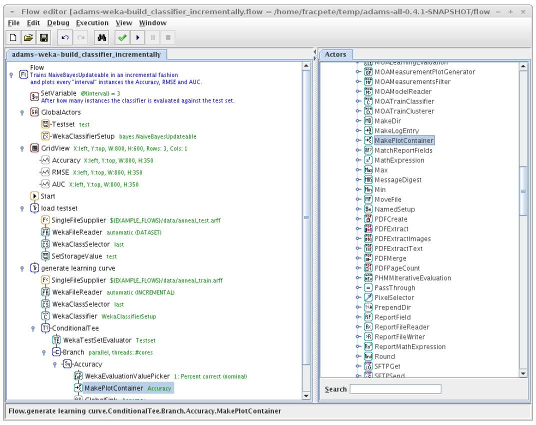
[16] adams-weka-classifying_data.flow

3.2 Advanced

3.2.1 Learning curves

Classifiers are susceptible to the order and amount of data that they are trained with. Using learning curves, one can investigate how the data influences the classifier performance.

Figures 3.28 and 3.29 show a flow and the generated output of an incremental NaiveBayes classifier, with the classifier being evaluated against a test set every 10 training instances (*ConditionalTee* in conjunction with the *Counting* condition).[17]



```
Flow
Trains NaiveBayesUpdateable in an incremental fashion
and plots every "interval" instances the Accuracy, RMSE and AUC.
    SetVariable  @{interval} = 3
    After how many instances the classifier is evaluated against the test set.
    GlobalActors
        Testset  test
        WekaClassifierSetup  bayes.NaiveBayesUpdateable
    GridView  X:left, Y:top, W:800, H:600, Rows: 3, Cols: 1
        Accuracy  X:left, Y:top, W:800, H:350
        RMSE  X:left, Y:top, W:800, H:350
        AUC  X:left, Y:top, W:800, H:350
    Start
    load testset
        SingleFileSupplier  ${EXAMPLE_FLOWS}/data/anneal_test.arff
        WekaFileReader  automatic (DATASET)
        WekaClassSelector  last
        SetStorageValue  test
    generate learning curve
        SingleFileSupplier  ${EXAMPLE_FLOWS}/data/anneal_train.arff
        WekaFileReader  automatic (INCREMENTAL)
        WekaClassSelector  last
        WekaClassifier  WekaClassifierSetup
        ConditionalTee
            WekaTestSetEvaluator  Testset
            Branch  parallel, threads: #cores
                Accuracy
                    WekaEvaluationValuePicker  1: Percent correct (nominal)
                    MakePlotContainer  Accuracy
```

Flow.generate learning curve.ConditionalTee.Branch.Accuracy.MakePlotContainer

Figure 3.28: Flow for generating learning curve for incremental classifier.

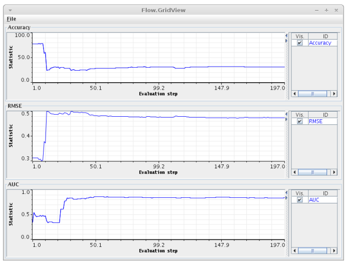


Figure 3.29: Generated learning curve (incremental).

Incremental classifiers are great for generating these kind of graphs. But even for batch classifiers you can generate learning curves. Using the *WekaInstanceBuffer* transformer, it is possible to buffer instances as they come through and output datasets with which the batch classifer can get trained (and evaluated). Figures 3.30 and 3.29 show how to generate a learning curve for the decision tree classifier J48, being evaluated every 10 instances.[18]

3.2.2 Experiments

experiment generation[19], execution and evaluation[20]

3.2.3 Optimization

setup generators[21], ranker[22], optimizer[23]

---

[17] adams-weka-build_classifier_incrementally.flow

[18] adams-weka-classifier_learning_curve.flow

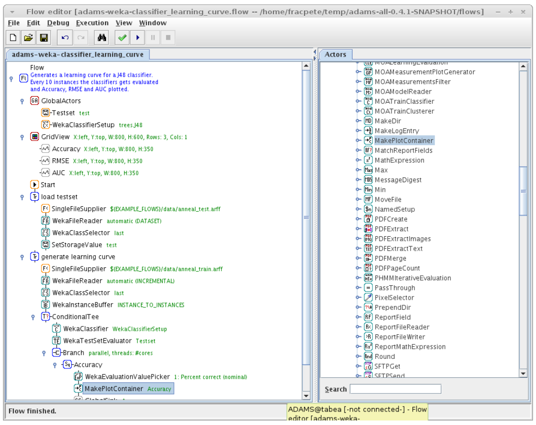
[19] adams-weka-experiment_generation.flow

[20] adams-weka-experiment.flow

[21] adams-weka-classifier_setup_generation.flow

[22] adams-weka-classifier_setup_ranking.flow

[23] adams-weka-classifier_optimizer.flow



```
Flow
Generates a learning curve for a J48 classifier.
Every 10 instances the classifiers gets evaluated
and Accuracy, RMSE and AUC plotted.
  GlobalActors
    -Testset  test
    -WekaClassifierSetup  trees.J48
  GridView  X:left, Y:top, W:800, H:600, Rows: 3, Cols: 1
    Accuracy  X:left, Y:top, W:800, H:350
    RMSE  X:left, Y:top, W:800, H:350
    AUC  X:left, Y:top, W:800, H:350
  Start
  load testset
    SingleFileSupplier  ${EXAMPLE_FLOWS}/data/anneal_test.arff
    WekaFileReader  automatic (DATASET)
    WekaClassSelector  last
    SetStorageValue  test
  generate learning curve
    SingleFileSupplier  ${EXAMPLE_FLOWS}/data/anneal_train.arff
    WekaFileReader  automatic (INCREMENTAL)
    WekaClassSelector  last
    WekaInstanceBuffer  INSTANCE_TO_INSTANCES
    ConditionalTee
      WekaClassifier  WekaClassifierSetup
      WekaTestSetEvaluator  Testset
      Branch  parallel, threads: #cores
        Accuracy
          WekaEvaluationValuePicker  1: Percent correct (nominal)
          MakePlotContainer  Accuracy
Flow finished.
ADAMS@tabea [-not connected-] - Flow editor [adams-weka-
```

Figure 3.30: Flow for generating learning curve for batch classifier.

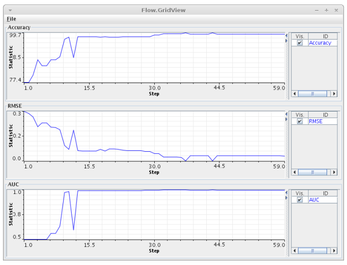


Figure 3.31: Generated learning curve (batch).

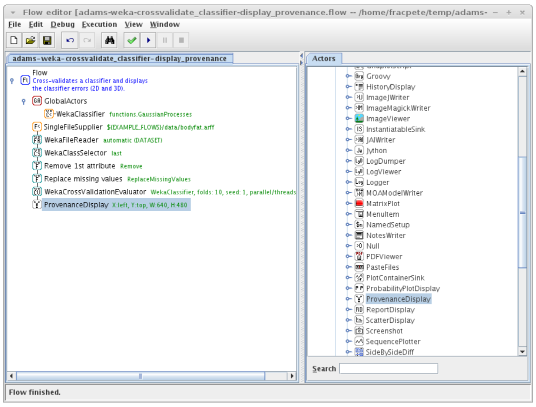
3.2.4 Provenance

Machine learning related actors can keep track of what operations happened along the way, or in other words, provenance.

Due to the additional load, provenance is turned off by default. You can turn it on by placing a properties file called `Provenance.props` home directory with the following content:

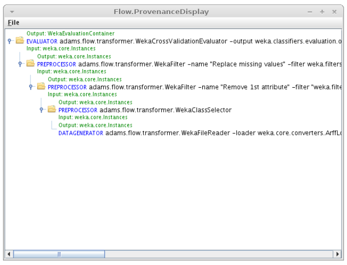
```
Enabled=true
```

Figure 3.32 shows the flow that cross-validates a classifier on a pre-processed dataset. The provenance trace of loading the data, pre-processing and evaluating it, can be seen in Figure 3.33.



```
Flow
Cross-validates a classifier and displays
the classifier errors (2D and 3D).
  GlobalActors
    -WekaClassifier  functions.GaussianProcesses
  SingleFileSupplier  ${EXAMPLE_FLOWS}/data/bodyfat.arff
  WekaFileReader  automatic (DATASET)
  WekaClassSelector  last
  Remove 1st attribute  Remove
  Replace missing values  ReplaceMissingValues
  WekaCrossValidationEvaluator  WekaClassifier, folds: 10, seed: 1, parallel/threads
  ProvenanceDisplay  X:left, Y:top, W:640, H:480
Flow finished.
```

Figure 3.32: Flow for cross-validating a classifier on a pre-processed dataset.



```
Output: WekaEvaluationContainer
EVALUATOR adams.flow.transformer.WekaCrossValidationEvaluator -output weka.classifiers.evaluation.o
Input: weka.core.Instances
  Output: weka.core.Instances
  PREPROCESSOR adams.flow.transformer.WekaFilter -name "Replace missing values" -filter weka.filters
  Input: weka.core.Instances
    Output: weka.core.Instances
    PREPROCESSOR adams.flow.transformer.WekaFilter -name "Remove 1st attribute" -filter "weka.filte
    Input: weka.core.Instances
      Output: weka.core.Instances
      PREPROCESSOR adams.flow.transformer.WekaClassSelector
      Input: weka.core.Instances
        Output: weka.core.Instances
        DATAGENERATOR adams.flow.transformer.WekaFileReader -loader weka.core.converters.ArffL
```

Figure 3.33: Provenance display.

provenance display[24]

---

[24] adams-weka-crossvalidate_classifier-display_provenance.flow

Chapter 4

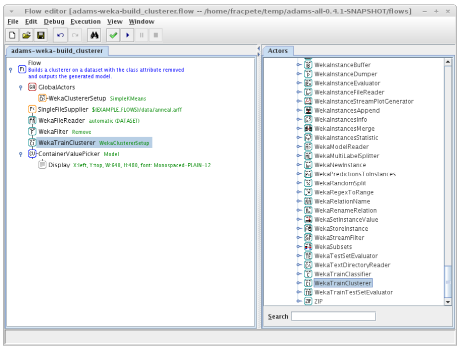
Clustering

Clustering behaves very much like Classification/Regression, the only difference being that it is an unsupervised learning process. This means that the flows won't contain a *WekaClassSelector* actor to set the class attribute in the loaded data. Due to the similarity, the section here will cover only the basics of clustering.

4.1 Building models

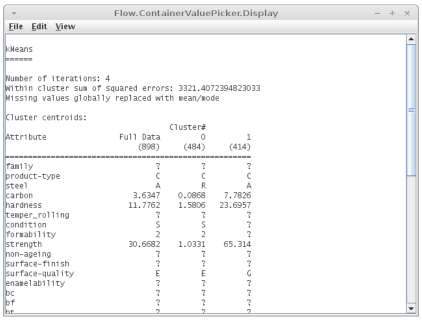
Building clustering models is as easy as building classification/regression models. Instead of the *WekaTrainClassifier* transformer, you use the *WekaTrainClusterer* one. Similar, you use a *WekaClustererSetup* source instead of the *WekaClassifierSetup* one to define (and output) a clusterer setup, placed inside a *GlobalActors* standalone.

Figures 4.1 and 4.2 show a flow [1] that builds a *SimpleKMeans* clusterer on a dataset (the class attribute gets removed using a *WekaFilter* actor) and the generated model gets displayed.



```
Flow
Builds a clusterer on a dataset with the class attribute removed
and outputs the generated model.
  GlobalActors
    WekaClustererSetup SimpleKMeans
  SingleFileSupplier ${EXAMPLE_FLOWS}/data/anneal.arff
  WekaFileReader automatic (DATASET)
  WekaFilter Remove
  WekaTrainClusterer WekaClustererSetup
  ContainerValuePicker Model
    Display X:left, Y:top, W:640, H:480, font: Monospaced-PLAIN-12
```

Figure 4.1: Building a clusterer and outputting the model.



```

kMeans
======

Number of iterations: 4
Within cluster sum of squared errors: 3321.4072394823033
Missing values globally replaced with mean/mode

Cluster centroids:
                                    Cluster#
Attribute                Full Data         0         1
                             (898)     (484)     (414)
======================================================
family                           ?         ?         ?
product-type                     C         C         C
steel                            A         R         A
carbon                      3.6347    0.0868    7.7826
hardness                   11.7762    1.5806   23.6957
temper_rolling                   ?         ?         ?
condition                        S         S         ?
formability                      2         2         ?
strength                   30.6682    1.0331    65.314
non-ageing                       ?         ?         ?
surface-finish                   ?         ?         ?
surface-quality                  E         E         G
enamelability                    ?         ?         ?
bc                               ?         ?         ?
bf                               ?         ?         ?
```

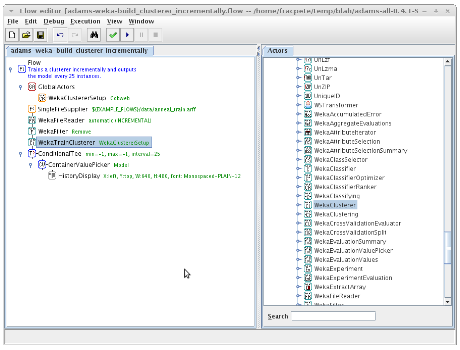
Figure 4.2: Cluster model output.

If the base cluster algorithm is an incremental one, i.e., one that implements

---

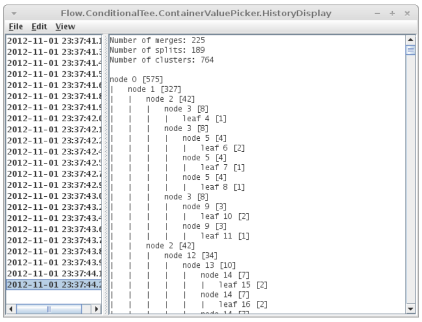
[1] adams-weka-build_clusterer.flow

the *weka.clusterers.UpdateableClusterer* interface, you can build your clustering model incrementally as well. The flow [2] in Figure 4.3 builds the CobWeb cluster algorithm incrementally and outputs the generated models every 25 instances (see Figure 4.4).



```
Flow
Trains a clusterer incrementally and outputs
the model every 25 instances.
  GlobalActors
    WekaClustererSetup Cobweb
  SingleFileSupplier ${EXAMPLE_FLOWS}/data/anneal_train.arff
  WekaFileReader automatic (INCREMENTAL)
  WekaFilter Remove
  WekaTrainClusterer WekaClustererSetup
  ConditionalTee min=-1, max=-1, interval=25
    ContainerValuePicker Model
      HistoryDisplay X:left, Y:top, W:640, H:480, font: Monospaced-PLAIN-12
```

Figure 4.3: Building a clusterer incrementally and outputting the model.



```
Number of merges: 225
Number of splits: 189
Number of clusters: 764

node 0 [575]
|   node 1 [327]
|   |   node 2 [42]
|   |   |   node 3 [8]
|   |   |   |   leaf 4 [1]
|   |   |   node 3 [8]
|   |   |   |   node 5 [4]
|   |   |   |   |   leaf 6 [2]
|   |   |   |   node 5 [4]
|   |   |   |   |   leaf 7 [1]
|   |   |   |   node 5 [4]
|   |   |   |   |   leaf 8 [1]
|   |   |   node 3 [8]
|   |   |   |   node 9 [3]
|   |   |   |   |   leaf 10 [2]
|   |   |   |   node 9 [3]
|   |   |   |   |   leaf 11 [1]
|   |   node 2 [42]
|   |   |   node 12 [34]
|   |   |   |   node 13 [10]
|   |   |   |   |   node 14 [7]
|   |   |   |   |   |   leaf 15 [2]
|   |   |   |   |   node 14 [7]
|   |   |   |   |   |   leaf 16 [2]
```

Figure 4.4: Cluster model outputs, generated every 25 instances.

4.2 Evaluating clusterers

ADAMS offers transformers for evaluation clusterers on data similar to the ones for classification:

- *WekaClusterEvaluationSummary* - generates a string representation of a cluster evaluation (or container)
- *WekaCrossValidationClustererEvaluator* - cross-validates a clusterer on a dataset, generates log-likelhood.
- *WekaTestSetClustererEvaluator* - evaluates a built clusterer on a test set.
- *WekaTrainTestSetClustererEvaluator* - builds and evaluates a clusterer on the training and test set from a train/test-set container.

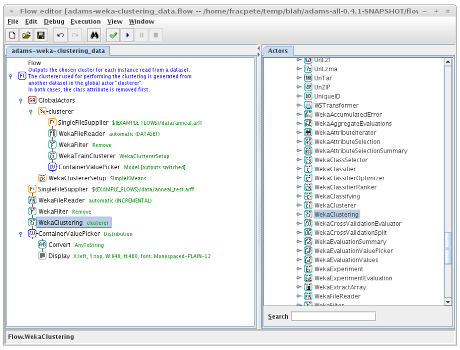
4.3 Clustering data

Clustering new data is done using the *WekaClustering* transformer, which takes a single instance as input and outputs the generated clustering information in form of a container (*WekaClusteringContainer*). You can either specify a serialized clusterer model to use or a global actor to obtain the clusterer from. The flow [3] in Figure 4.5 shows how to build a clusterer and use it to cluster new data, outputting the cluster distributions (see Figure 4.6 for the generated output).

---

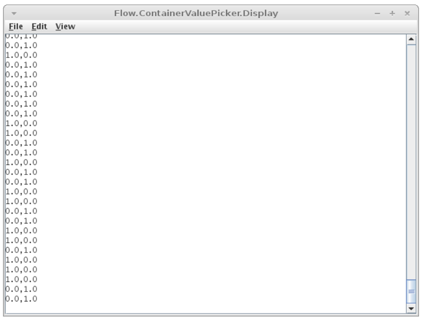
[2] adams-weka-build_clusterer_incrementally.flow

[3] adams-weka-clustering_data.flow



```
Flow
Outputs the chosen cluster for each instance read from a dataset.
The clusterer used for performing the clustering is generated from
another dataset in the global actor "clusterer".
In both cases, the class attribute is removed first.
  GlobalActors
    clusterer
      SingleFileSupplier ${EXAMPLE_FLOWS}/data/anneal.arff
      WekaFileReader automatic (DATASET)
      WekaFilter Remove
      WekaTrainClusterer WekaClustererSetup
      ContainerValuePicker Model [outputs switched]
    WekaClustererSetup SimpleKMeans
  SingleFileSupplier ${EXAMPLE_FLOWS}/data/anneal_test.arff
  WekaFileReader automatic (INCREMENTAL)
  WekaFilter Remove
  WekaClustering clusterer
  ContainerValuePicker Distribution
    Convert AnyToString
    Display X:left, Y:top, W:640, H:480, font: Monospaced-PLAIN-12
```

Figure 4.5: Flow for clustering new data.



```
0.0,1.0
0.0,1.0
1.0,0.0
0.0,1.0
0.0,1.0
0.0,1.0
0.0,1.0
0.0,1.0
0.0,1.0
1.0,0.0
1.0,0.0
0.0,1.0
0.0,1.0
1.0,0.0
0.0,1.0
0.0,1.0
1.0,0.0
1.0,0.0
0.0,1.0
0.0,1.0
1.0,0.0
1.0,0.0
0.0,1.0
1.0,0.0
1.0,0.0
1.0,0.0
0.0,1.0
0.0,1.0
```

Figure 4.6: Generated cluster.

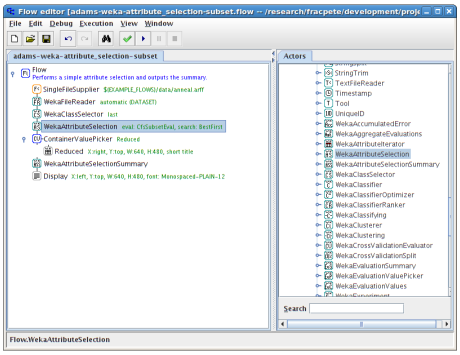
Chapter 5

Attribute selection

ADAMS also offers WEKA's functionality for attribute selection and ranking. The following transformers are available:

- *WekaAttributeSelection* – performs the attribute selection/ranking.
- *WekaAttributeSelectionSummary* – generates a summary from a attribute selection step.

In Figure 5.1 you can see a flow[1] that uses *CfsSubsetEval* as the attribute set evaluator and *BestFirst* as the search method. The generated output, summary and reduced dataset, are displayed in Figures 5.2 and 5.3.



```
Flow
Performs a simple attribute selection and outputs the summary.
  SingleFileSupplier ${EXAMPLE_FLOWS}/data/anneal.arff
  WekaFileReader automatic (DATASET)
  WekaClassSelector last
  WekaAttributeSelection eval: CfsSubsetEval, search: BestFirst
  ContainerValuePicker Reduced
    Reduced X:right, Y:top, W:640, H:480, short title
  WekaAttributeSelectionSummary
  Display X:left, Y:top, W:640, H:480, font: Monospaced-PLAIN-12
```

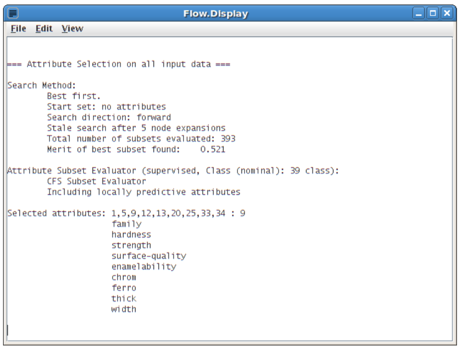
Figure 5.1: Flow for performing attribute selection (reduction).

The *WekaAttributeSelection* transformer outputs a container which can contain the following elements:

- *Train* – the training set.
- *Reduced* – the reduced dataset.
- *Transformed* – the transformed dataset, in case of evaluators that implement *AttributeTransformer*, like principal components.
- *Evaluation* – the generated attribute selection evaluation.

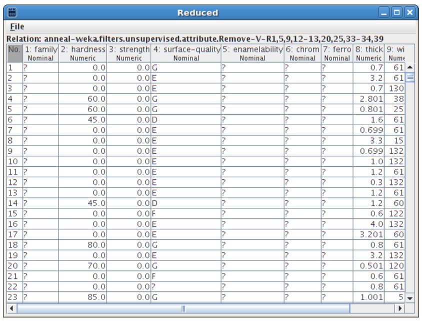
---

[1] adams-weka-attribute_selection-subset.flow



```
=== Attribute Selection on all input data ===

Search Method:
        Best first.
        Start set: no attributes
        Search direction: forward
        Stale search after 5 node expansions
        Total number of subsets evaluated: 393
        Merit of best subset found:    0.521

Attribute Subset Evaluator (supervised, Class (nominal): 39 class):
        CFS Subset Evaluator
        Including locally predictive attributes

Selected attributes: 1,5,9,12,13,20,25,33,34 : 9
                     family
                     hardness
                     strength
                     surface-quality
                     enamelability
                     chrom
                     ferro
                     thick
                     width
```

Figure 5.2: Summary of the reduction.



Relation: anneal-weka.filters.unsupervised.attribute.Remove-V-R1,5,9,12-13,20,25,33-34,39

No.	1: family Nominal	2: hardness Numeric	3: strength Numeric	4: surface-quality Nominal	5: enamelability Nominal	6: chrom Nominal	7: ferro Nominal	8: thick Numeric	9: wi Num
1	?	0.0	0.0	G	?	?	?	0.7	61
2	?	0.0	0.0	E	?	?	?	3.2	61
3	?	0.0	0.0	E	?	?	?	0.7	130
4	?	60.0	0.0	G	?	?	?	2.801	38
5	?	60.0	0.0	G	?	?	?	0.801	25
6	?	45.0	0.0	D	?	?	?	1.6	61
7	?	0.0	0.0	E	?	?	?	0.699	61
8	?	0.0	0.0	E	?	?	?	3.3	15
9	?	0.0	0.0	E	?	?	?	0.699	132
10	?	0.0	0.0	E	?	?	?	1.0	132
11	?	0.0	0.0	E	?	?	?	1.2	61
12	?	0.0	0.0	E	?	?	?	0.3	132
13	?	0.0	0.0	E	?	?	?	1.2	61
14	?	45.0	0.0	D	?	?	?	1.2	60
15	?	0.0	0.0	F	?	?	?	0.6	122
16	?	0.0	0.0	E	?	?	?	4.0	132
17	?	0.0	0.0	E	?	?	?	3.201	60
18	?	80.0	0.0	G	?	?	?	0.8	61
19	?	0.0	0.0	E	?	?	?	3.2	132
20	?	70.0	0.0	G	?	?	?	0.501	120
21	?	0.0	0.0	F	?	?	?	0.6	61
22	?	0.0	0.0	?	?	?	?	0.8	61
23	?	85.0	0.0	G	?	?	?	1.001	5

Figure 5.3: The reduced dataset.

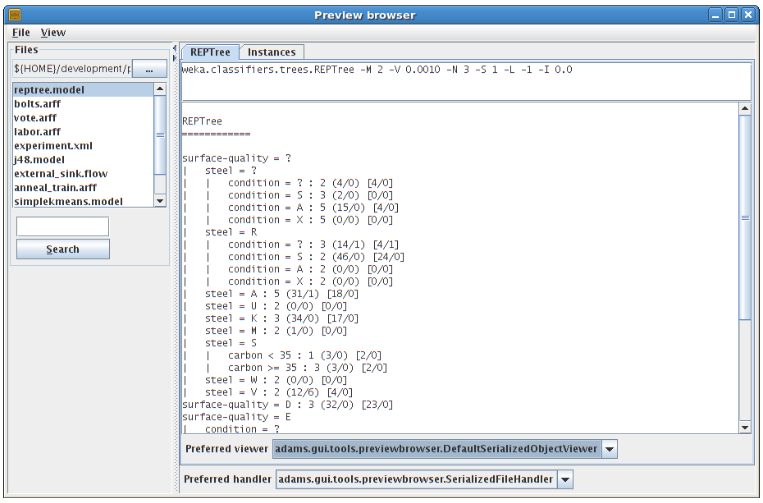
- *Statistics* – a spreadsheet with statistics, containing information whether an attribute was selected (0 or 1) or for ranking results the rank of the attribute.
- *Seed* – the seed value in case of cross-validation.
- *Folds* – the number of folds used in case of cross-validation.

Chapter 6

Visualization

6.1 Preview browser

The WEKA module comes with custom viewers for serialized files. Apart from the default view (Figure 6.1), you can also view the trees (Figure 6.2) and graphs (Figure 6.3) that some classifiers generate.



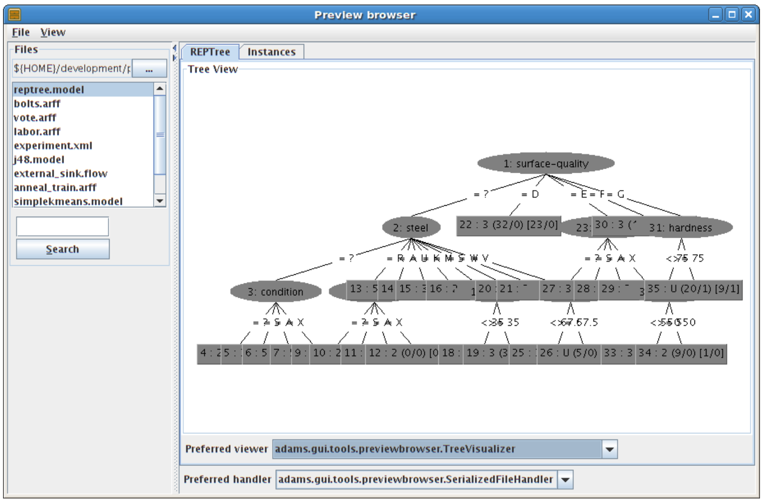
```
weka.classifiers.trees.REPTree -M 2 -V 0.0010 -N 3 -S 1 -L -1 -I 0.0
```

```
REPTree
============

surface-quality = ?
|   steel = ?
|   |   condition = ? : 2 (4/0) [4/0]
|   |   condition = S : 3 (2/0) [0/0]
|   |   condition = A : 5 (15/0) [4/0]
|   |   condition = X : 5 (0/0) [0/0]
|   steel = R
|   |   condition = ? : 3 (14/1) [4/1]
|   |   condition = S : 2 (46/0) [24/0]
|   |   condition = A : 2 (0/0) [0/0]
|   |   condition = X : 2 (0/0) [0/0]
|   steel = A : 5 (31/1) [18/0]
|   steel = U : 2 (0/0) [0/0]
|   steel = K : 3 (34/0) [17/0]
|   steel = M : 2 (1/0) [0/0]
|   steel = S
|   |   carbon < 35 : 1 (3/0) [2/0]
|   |   carbon >= 35 : 3 (3/0) [2/0]
|   steel = W : 2 (0/0) [0/0]
|   steel = V : 2 (12/6) [4/0]
surface-quality = D : 3 (32/0) [23/0]
surface-quality = E
|   condition = ?
```

Preferred viewer: adams.gui.tools.previewbrowser.DefaultSerializedObjectViewer

Preferred handler: adams.gui.tools.previewbrowser.SerializedFileHandler

Figure 6.1: Default preview for classifier.



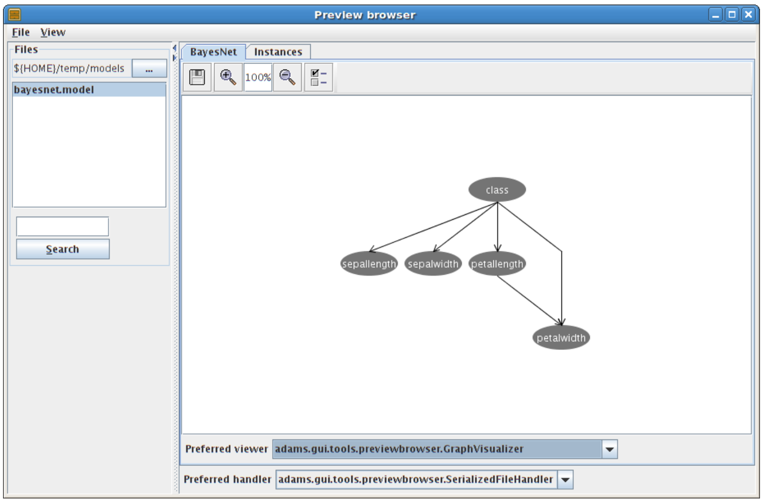
Files: ${HOME}/development/p

reptree.model
bolts.arff
vote.arff
labor.arff
experiment.xml
j48.model
external_sink.flow
anneal_train.arff
simplekmeans.model

Preferred viewer: adams.gui.tools.previewbrowser.TreeVisualizer

Preferred handler: adams.gui.tools.previewbrowser.SerializedFileHandler

Figure 6.2: Preview of tree-generating classifier.



Files: ${HOME}/temp/models

bayesnet.model

Preferred viewer: adams.gui.tools.previewbrowser.GraphVisualizer

Preferred handler: adams.gui.tools.previewbrowser.SerializedFileHandler

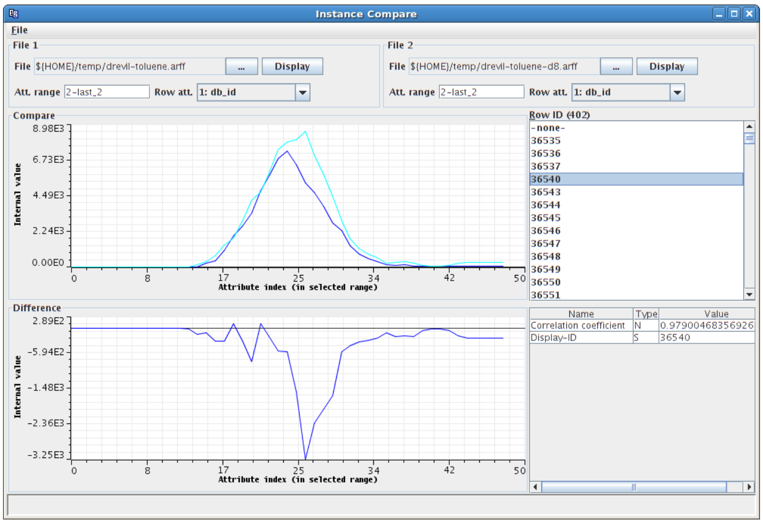
Figure 6.3: Preview for graph generated by BayesNet classifier.

6.2 Instance Compare

Quite often, you generate data with different or tweaked pre-processing techniques and you wonder how different the generated data looks like. The *Instance Compare* visualization allows you to graphically compare two datasets. You can either compare them row by row, or using a common attribute that can be used as unique row identifier.

Figure 6.4 shows a comparison of two datasets. Not only are the two rows overlayed, you also see the absolute difference plotter and a the correlation coefficient of the two being calculated.

If you don't want to compare all the attributes, you can restrict it to a subset, by using the *Att. range* text field. "first", "second", "third", "last", "last_1" (last minus 1) and "last_2" (last minus 2) are accepted indices. All other indices must be 1-based.



File 1: File ${HOME}/temp/drevil-toluene.arff; Att. range 2-last_2; Row att. 1: db_id

File 2: File ${HOME}/temp/drevil-toluene-d8.arff; Att. range 2-last_2; Row att. 1: db_id

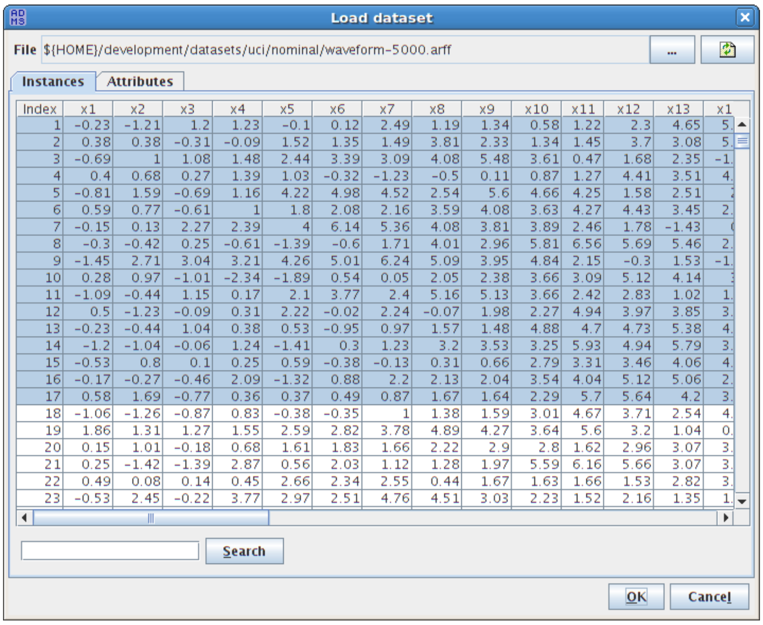
Name	Type	Value
Correlation coefficient	N	0.97900468356926
Display-ID	S	36540

Figure 6.4: Comparing two datasets.

6.3 Instance Explorer

When generating datasets, it pays to check the generated output in multiple ways. For instance, whether the data rows generated are actually aligning properly. The *Instance Explorer* allows you to select a range of rows and columns from a dataset (see Figures 6.5 and 6.6), which are then displayed in a single graph.

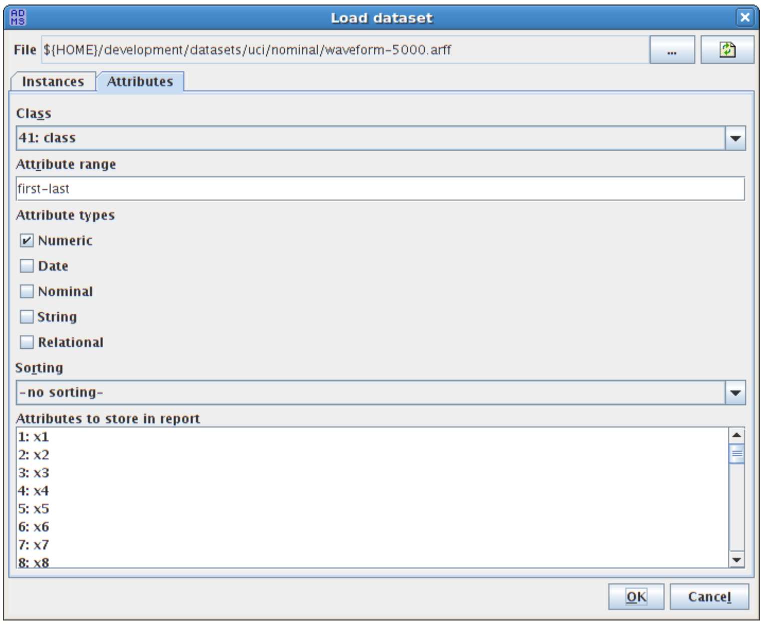
Figure 6.7 shows a subset of the UCI dataset *waveform-5000*. The top graph of the two is a zoom into the full graph, with the bottom graph showing the area that was zoomed into.



File ${HOME}/development/datasets/uci/nominal/waveform-5000.arff

Index	x1	x2	x3	x4	x5	x6	x7	x8	x9	x10	x11	x12	x13	x1
1	-0.23	-1.21	1.2	1.23	-0.1	0.12	2.49	1.19	1.34	0.58	1.22	2.3	4.65	5.
2	0.38	0.38	-0.31	-0.09	1.52	1.35	1.49	3.81	2.33	1.34	1.45	3.7	3.08	5.
3	-0.69	1	1.08	1.48	2.44	3.39	3.09	4.08	5.48	3.61	0.47	1.68	2.35	-1.
4	0.4	0.68	0.27	1.39	1.03	-0.32	-1.23	-0.5	0.11	0.87	1.27	4.41	3.51	4.
5	-0.81	1.59	-0.69	1.16	4.22	4.98	4.52	2.54	5.6	4.66	4.25	1.58	2.51	
6	0.59	0.77	-0.61	1	1.8	2.08	2.16	3.59	4.08	3.63	4.27	4.43	3.45	2.
7	-0.15	0.13	2.27	2.39	4	6.14	5.36	4.08	3.81	3.89	2.46	1.78	-1.43	
8	-0.3	-0.42	0.25	-0.61	-1.39	-0.6	1.71	4.01	2.96	5.81	6.56	5.69	5.46	2.
9	-1.45	2.71	3.04	3.21	4.26	5.01	6.24	5.09	3.95	4.84	2.15	-0.3	1.53	-1.
10	0.28	0.97	-1.01	-2.34	-1.89	0.54	0.05	2.05	2.38	3.66	3.09	5.12	4.14	
11	-1.09	-0.44	1.15	0.17	2.1	3.77	2.4	5.16	5.13	3.66	2.42	2.83	1.02	1.
12	0.5	-1.23	-0.09	0.31	2.22	-0.02	2.24	-0.07	1.98	2.27	4.94	3.97	3.85	3.
13	-0.23	-0.44	1.04	0.38	0.53	-0.95	0.97	1.57	1.48	4.88	4.7	4.73	5.38	4.
14	-1.2	-1.04	-0.06	1.24	-1.41	0.3	1.23	3.2	3.53	3.25	5.93	4.94	5.79	3.
15	-0.53	0.8	0.1	0.25	0.59	-0.38	-0.13	0.31	0.66	2.79	3.31	3.46	4.06	4.
16	-0.17	-0.27	-0.46	2.09	-1.32	0.88	2.2	2.13	2.04	3.54	4.04	5.12	5.06	2.
17	0.58	1.69	-0.77	0.36	0.37	0.49	0.87	1.67	1.64	2.29	5.7	5.64	4.2	3.
18	-1.06	-1.26	-0.87	0.83	-0.38	-0.35	1	1.38	1.59	3.01	4.67	3.71	2.54	4.
19	1.86	1.31	1.27	1.55	2.59	2.82	3.78	4.89	4.27	3.64	5.6	3.2	1.04	0.
20	0.15	1.01	-0.18	0.68	1.61	1.83	1.66	2.22	2.9	2.8	1.62	2.96	3.07	3.
21	0.25	-1.42	-1.39	2.87	0.56	2.03	1.12	1.28	1.97	5.59	6.16	5.66	3.07	3.
22	0.49	0.08	0.14	0.45	2.66	2.34	2.55	0.44	1.67	1.63	1.66	1.53	2.82	3.
23	-0.53	2.45	-0.22	3.77	2.97	2.51	4.76	4.51	3.03	2.23	1.52	2.16	1.35	1.

Figure 6.5: Viewing data in the Instance explorer.



File ${HOME}/development/datasets/uci/nominal/waveform-5000.arff

Class
41: class

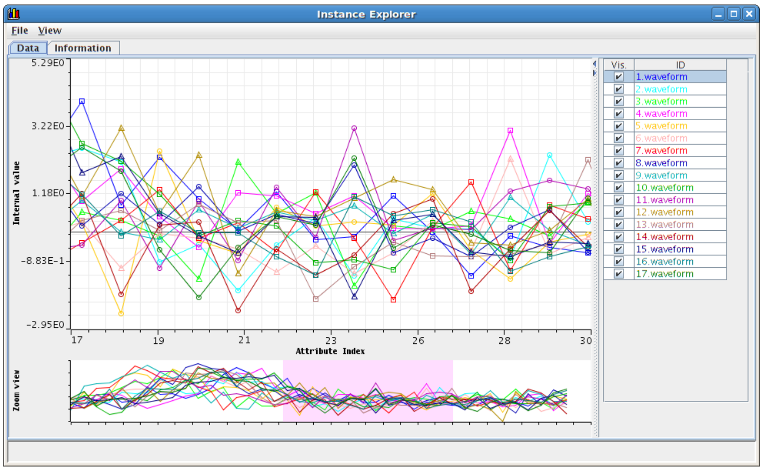
Attribute range
first-last

Attribute types
☑ Numeric
☐ Date
☐ Nominal
☐ String
☐ Relational

Sorting
-no sorting-

Attributes to store in report
1: x1
2: x2
3: x3
4: x4
5: x5
6: x6
7: x7
8: x8

Figure 6.6: Viewing data in the Instance explorer.



Vis	ID
☑	1.waveform
☑	2.waveform
☑	3.waveform
☑	4.waveform
☑	5.waveform
☑	6.waveform
☑	7.waveform
☑	8.waveform
☑	9.waveform
☑	10.waveform
☑	11.waveform
☑	12.waveform
☑	13.waveform
☑	14.waveform
☑	15.waveform
☑	16.waveform
☑	17.waveform

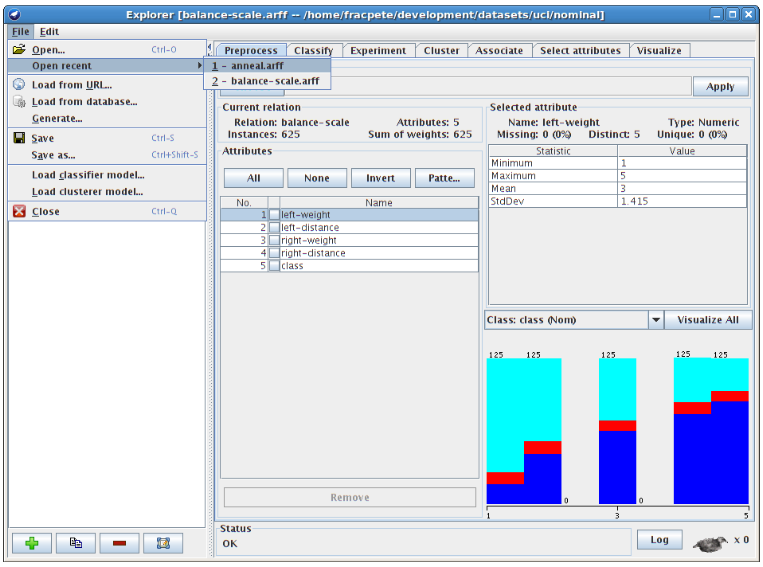
Figure 6.7: Viewing data in the Instance explorer.

Chapter 7

Tools

7.1 Explorer

ADAMS contains an extended version of the WEKA Explorer. The interface uses menus instead of buttons to declutter the pre-process tab. Also, it keeps track of the datasets that the user loads, to make re-loading recent files easier. This saves a lot of time when working with the same files on a frequent basis. Furthermore, the user can have an arbitrary number of Explorer sessions in the same window, distinguished by names. Figure 7.1 shows the new interface with the drop-down menu in action.



Current relation
Relation: balance-scale
Attributes: 5
Instances: 625
Sum of weights: 625

Selected attribute
Name: left-weight
Type: Numeric
Missing: 0 (0%)
Distinct: 5
Unique: 0 (0%)

Statistic	Value
Minimum	1
Maximum	5
Mean	3
StdDev	1.415

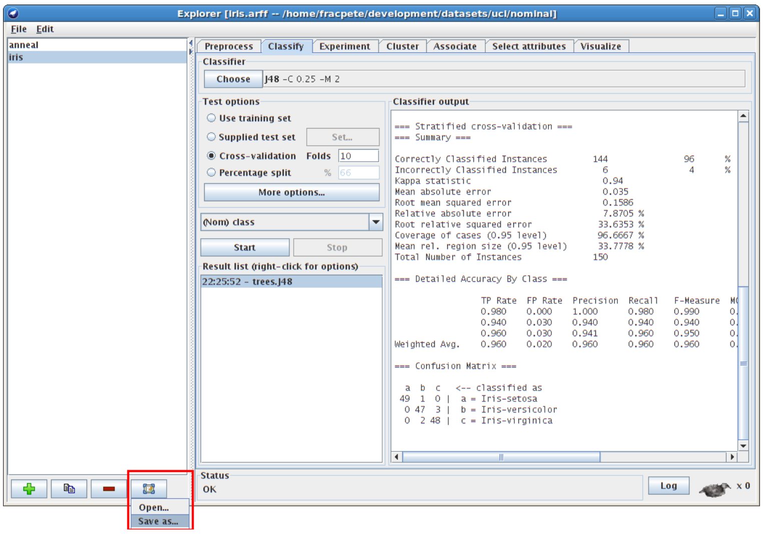
Attributes

No.		Name
1	☐	left-weight
2	☐	left-distance
3	☐	right-weight
4	☐	right-distance
5	☐	class

Status
OK

Figure 7.1: Explorer interface with menus.

One very useful feature is the notion of *workspaces* in this interface. You can save the current setup (current dataset, classifiers, clusterers, evaluation set up, results, etc.) to a file and restore all of it in one go again. Unfortunately, not all data can be stored, such as the log, the undo history and the built models or visualizations associated with a results. See Figure 7.2 for the button (highlighted in red) that allows you to load/save workspaces.



Explorer [iris.arff -- /home/fracpete/development/datasets/uci/nominal]

Classifier: J48 -C 0.25 -M 2

Test options:
○ Use training set
○ Supplied test set
● Cross-validation Folds 10
○ Percentage split % 66

(Nom) class

Result list (right-click for options): 22:25:52 - trees.J48

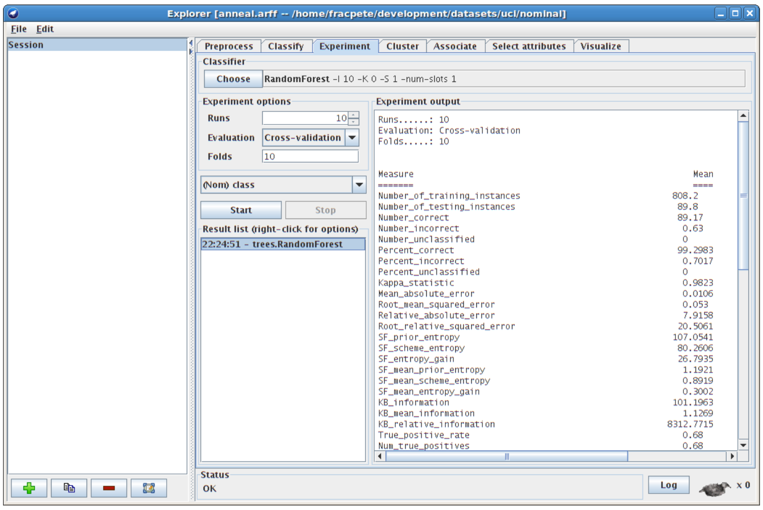
Classifier output

```
=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         144               96      %
Incorrectly Classified Instances         6                4      %
Kappa statistic                          0.94
Mean absolute error                      0.035
Root mean squared error                  0.1586
Relative absolute error                  7.8705 %
Root relative squared error             33.6353 %
Coverage of cases (0.95 level)          96.6667 %
Mean rel. region size (0.95 level)      33.7778 %
Total Number of Instances              150

=== Detailed Accuracy By Class ===

                 TP Rate  FP Rate  Precision  Recall   F-Measure  M(
                 0.980    0.000    1.000      0.980    0.990      0.
                 0.940    0.030    0.940      0.940    0.940      0.
                 0.960    0.030    0.941      0.960    0.950      0.
Weighted Avg.    0.960    0.020    0.960      0.960    0.960      0.

=== Confusion Matrix ===

  a  b  c   <-- classified as
 49  1  0 |  a = Iris-setosa
  0 47  3 |  b = Iris-versicolor
  0  2 48 |  c = Iris-virginica
```

Status: OK

Figure 7.2: Saving/restoring of workspaces.

The extended interface also has a dedicated tab for running experiments, using the currently loaded dataset and a single classifier. This allows you to perform 10 runs of cross-validation instead of the Explorer's default single run (see Figure 7.3).



Explorer [anneal.arff -- /home/fracpete/development/datasets/uci/nominal]

Classifier: RandomForest -I 10 -K 0 -S 1 -num-slots 1

Experiment options:
Runs 10
Evaluation Cross-validation
Folds 10

(Nom) class

Result list (right-click for options): 22:24:51 - trees.RandomForest

Experiment output

```
Runs......: 10
Evaluation: Cross-validation
Folds.....: 10


Measure                                                   Mean
=======                                                   ====
Number_of_training_instances                             808.2
Number_of_testing_instances                               89.8
Number_correct                                            89.17
Number_incorrect                                           0.63
Number_unclassified                                        0
Percent_correct                                           99.2983
Percent_incorrect                                          0.7017
Percent_unclassified                                       0
Kappa_statistic                                            0.9823
Mean_absolute_error                                        0.0106
Root_mean_squared_error                                    0.053
Relative_absolute_error                                    7.9158
Root_relative_squared_error                               20.5061
SF_prior_entropy                                         107.0541
SF_scheme_entropy                                         80.2606
SF_entropy_gain                                           26.7935
SF_mean_prior_entropy                                      1.1921
SF_mean_scheme_entropy                                     0.8919
SF_mean_entropy_gain                                       0.3002
KB_information                                           101.1963
KB_mean_information                                        1.1269
KB_relative_information                                 8312.7715
True_positive_rate                                         0.68
Num_true_positives                                         0.68
```

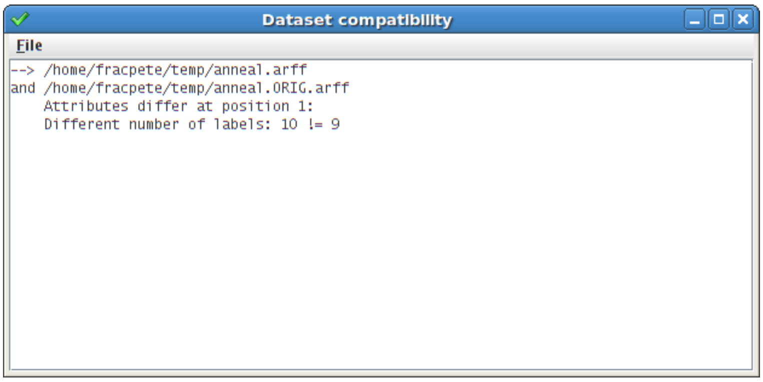
Status: OK

Figure 7.3: Experiment tab in the Explorer.

7.2 Dataset compatibility

WEKA requires training and test sets to have the same structure, down to the same name and order of nominal labels. Rather than relying on the error message in the Explorer, you can use the *Dataset compatibility* tool to quickly check whether two or more datasets are actually compatible.

The screenshot in Figure 7.4 shows the output when comparing two datases, one being the original *anneal* UCI dataset and the other one a transformed version.



```
--> /home/fracpete/temp/anneal.arff
and /home/fracpete/temp/anneal.ORIG.arff
    Attributes differ at position 1:
    Different number of labels: 10 != 9
```

Figure 7.4: Compatibility output for two datasets.

Bibliography

[1] *ADAMS* – Advanced Data mining and Machine learning System
`https://adams.cms.waikato.ac.nz/`

[2] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, Ian H. Witten (2009); *The WEKA Data Mining Software: An Update*; SIGKDD Explorations, Volume 11, Issue 1.
`http://www.cms.waikato.ac.nz/ml/weka/`

[3] Ian H. Witten, Eibe Frank, Mark A. Hall (2011); *Data Mining: Practical Machine Learning Tools and Techniques*; Third Edition; Morgan Kaufmann; ISBN 978-0-12-374856-0
`http://www.cs.waikato.ac.nz/ml/weka/book.html`

[4] Barker, G, and McGhie, R (1984) The Biology of Introduced Slugs (Pulmonata) in New Zealand: Introduction and Notes on Limax Maximus, NZ Entomologist 8, pp 106-111