ADAMS

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

Module: adams-weka

Peter Reutemann

January 27, 2012

DRAFT

©2009-2012

THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Contents

1 Introduction **7**

2 Classification and Regression **9**
2.1 Basic 10
2.1.1 Loading data 10
2.1.2 Building models 12
2.1.3 Preprocessing 12
2.1.4 Evaluation 15
2.1.5 Making predictions 19
2.2 Advanced 20
2.2.1 Learning curves 20
2.2.2 Experiments 20
2.2.3 Optimization 20
2.2.4 Provenance 20

3 Clustering **21**
3.1 Building models 21
3.2 Clustering data 22

Bibliography **23**

List of Figures

2.1 Flow for loading a local dataset. 10
2.2 The dataset that got loaded from disk. 10
2.3 Flow for loading a local dataset. 11
2.4 The dataset that got loaded from disk. 11
2.5 Flow for generating and displaying an artificial dataset. 11
2.6 Flow for building J48 model on a dataset and outputting the model. 12
2.7 J48 model output. 12
2.8 Flow for comparing results generated from original and preprocessed "slug" data [4]. 13
2.9 Evaluation summary on "slug" dataset (original). 13
2.10 Evaluation summary on "slug" dataset (log-transformed). 13
2.11 Classifier errors on "slug" dataset (original). 14
2.12 Classifier errors on "slug" dataset (log-transformed). 14
2.13 Cross-validating a classifier and outputting the summary. 15
2.14 Summary output of a cross-validated classifier. 15
2.15 Text file with command-lines of various classifiers. 16
2.16 Cross-validating classifier set ups read from a text file and displaying the evaluation summaries. 16
2.17 Summary outputs of cross-validated classifiers. 16
2.18 Flow for evaluating built classifier on a separate test set. 16
2.19 Summary output of classifier evaluated on separate test set. . . . 16
2.20 Flow for building/evaluating classifier on a random split. 17
2.21 Summary output of classifier built/evaluated on random split. . . 17
2.22 Flow for evaluating classifier on separate train/test set. 18
2.23 Summary output of classifier evaluated on separate train/test set. 18
2.24 Flow for displaying the "accumulated error" of a two classifiers. . 18
2.25 The "accumulated error" of LinearRegression and GaussianProcesses. 18
2.26 Flow for classifying new data and outputting the class distributions. 19
2.27 The generated class distributions for the new data. 19

3.1 Building a clusterer and outputting the model. 21
3.2 Cluster model output. 21
3.3 Building a clusterer incrementally and outputting the model. . . 22
3.4 Cluster model outputs, generated every 25 instances. 22
3.5 Flow for clustering new data. 22
3.6 Generated cluster. 22

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 two parts: *classification and regression* comprising the first one and *clustering* the second.

[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

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.

2.1 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.

2.1.1 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 2.1 shows the flow for loading the dataset and Figure 2.2 the generated output.

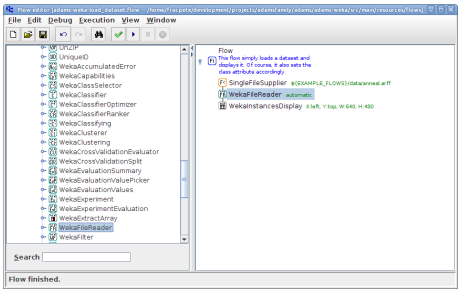

Figure 2.1: Flow for loading a local dataset.

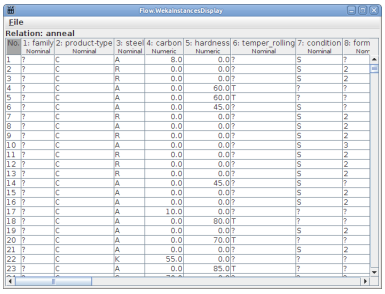

Figure 2.2: The dataset that got loaded from disk.

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 2.3 shows a flow [3] that downloads (and displays) an ARFF file available from a URL that was supplied by the *SingleURLSupplier*. 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 2.4 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.

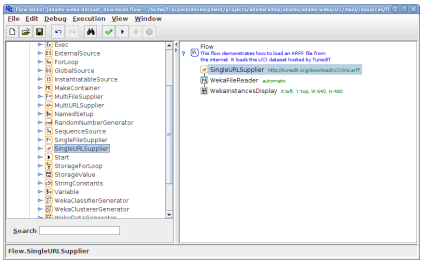

Figure 2.3: Flow for loading a local dataset.

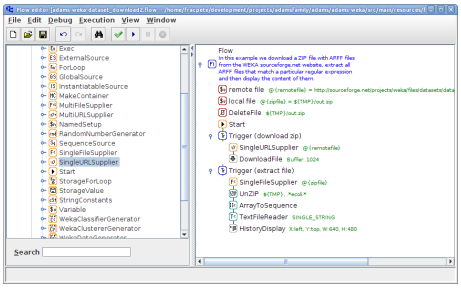

Figure 2.4: The dataset that got loaded from disk.

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 2.5 generates a small dataset using the "Agrawal" data generator.

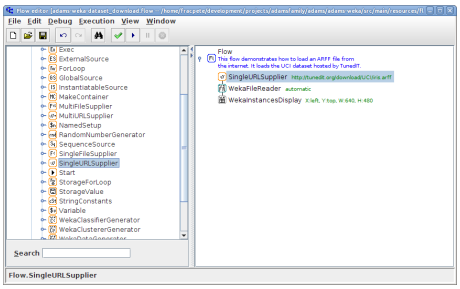

Figure 2.5: Flow for generating and displaying an artificial dataset.

[3] adams-weka-dataset_download.flow
[4] adams-weka-dataset_download2.flow
[5] WEKA on sourceforge.net: `http://sourceforge.net/projects/weka/`
[6] adams-weka-data_generator.flow

2.1.2 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 *WekaClassifier* actor you can choose a classifier to be built. By default, the *WekaClassifier* 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 2.6 demonstrates how to train a J48 classifier on dataset and then displaying the built model (see Figure 2.7) [7].

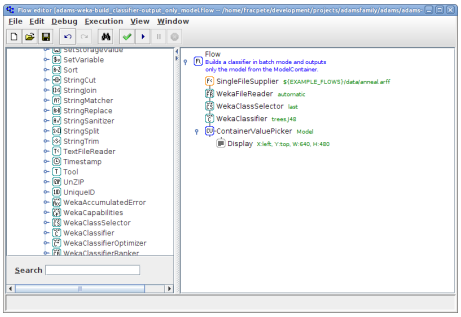

Figure 2.6: Flow for building J48 model on a dataset and outputting the model.

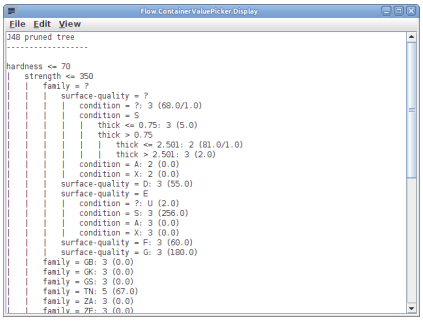

Figure 2.7: J48 model output.

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].

2.1.3 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 2.8 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.

Figures 2.9 and 2.10 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.

[7] adams-weka-build_classifier-output_only_model.flow
[8] adams-weka-build_classifier-save_model.flow
[9] adams-weka-filter_data.flow

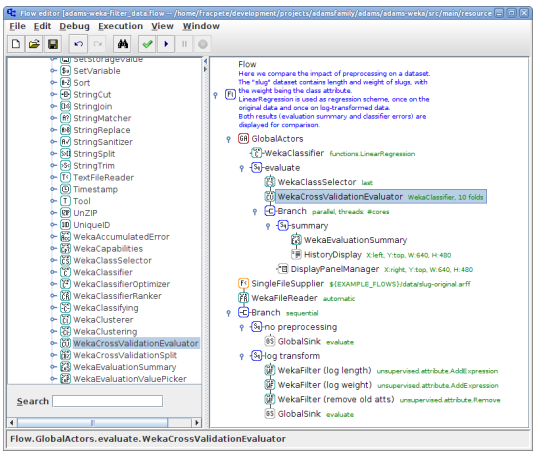

Figure 2.8: Flow for comparing results generated from original and preprocessed "slug" data [4].

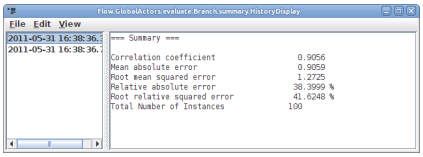

Figure 2.9: Evaluation summary on "slug" dataset (original).

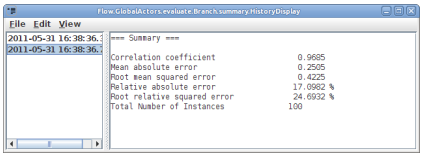

Figure 2.10: Evaluation summary on "slug" dataset (log-transformed).

Figures 2.11 and 2.12 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.

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.

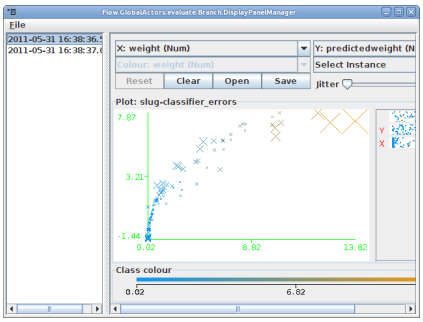

Figure 2.11: Classifier errors on "slug" dataset (original).

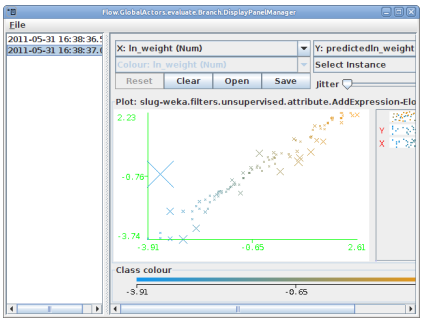

Figure 2.12: Classifier errors on "slug" dataset (log-transformed).

2.1.4 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 *WekaClassifier* definition or a more complex scheme for outputting a *Classifier* object (e.g., loading it from a serialized model file). Figure 2.13 shows a flow [10] with simple cross-validation using a global *WekaClassifier* to obtain the classifier object from.

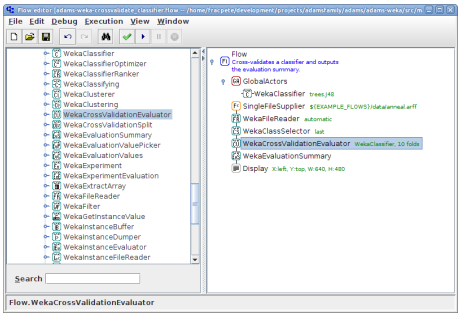

Figure 2.13: Cross-validating a classifier and outputting the summary.

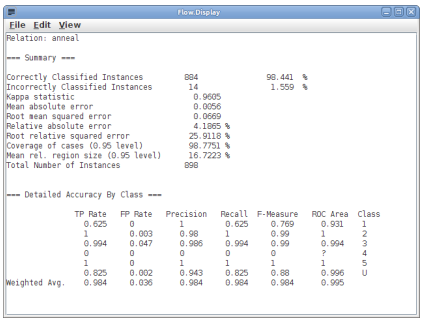

Figure 2.14: Summary output of a cross-validated classifier.

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 2.15) 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 *WekaClassifier* 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 2.16 and 2.17 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

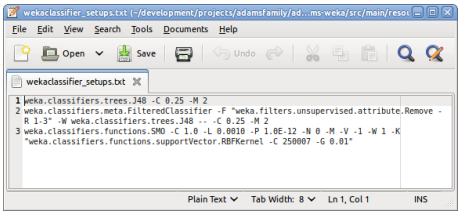

Figure 2.15: Text file with command-lines of various classifiers.

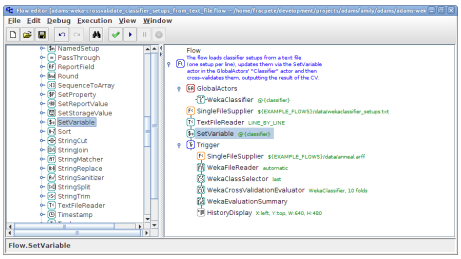

Figure 2.16: Cross-validating classifier set ups read from a text file and displaying the evaluation summaries.

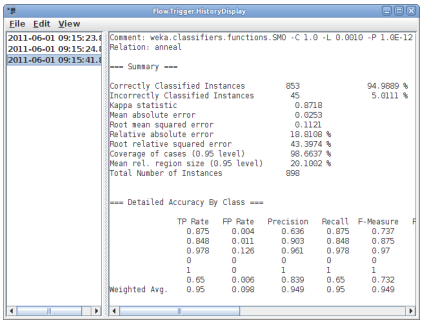

Figure 2.17: Summary outputs of cross-validated classifiers.

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 2.18 and 2.19.

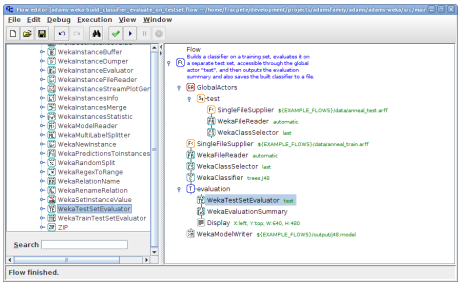

Figure 2.18: Flow for evaluating built classifier on a separate test set.

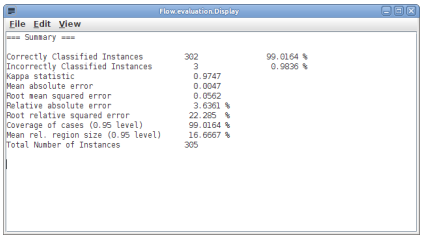

Figure 2.19: Summary output of classifier evaluated on separate test set.

[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 2.20 and 2.21 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).

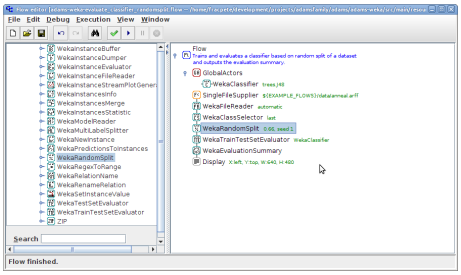

Figure 2.20: Flow for building/evaluating classifier on a random split.

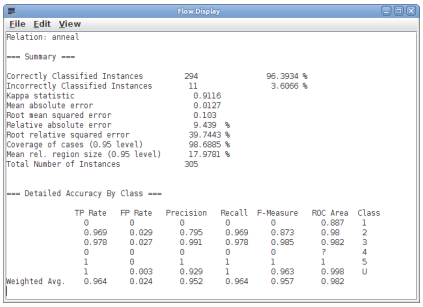

Figure 2.21: Summary output of classifier built/evaluated on random split.

Figures 2.22 and 2.23 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 2.11). 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 *WekaClassifierErrors* actor as "panelProvider". The *DisplayPanelManager* actor offers a history of generated panels, like the *HistoryDisplay* does for plain text.

[13] adams-weka-evaluate_classifier_randomsplit.flow
[14] adams-weka-assemble_traintestset_container_and_evaluate_classifier.flow

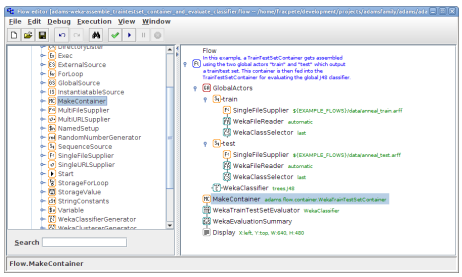

Figure 2.22: Flow for evaluating classifier on separate train/test set.

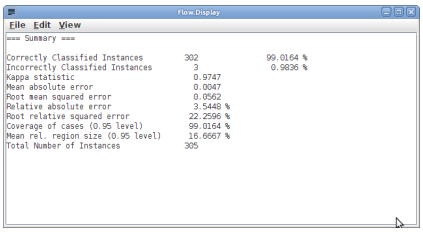

Figure 2.23: Summary output of classifier evaluated on separate train/test set.

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 2.24 and 2.25 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).

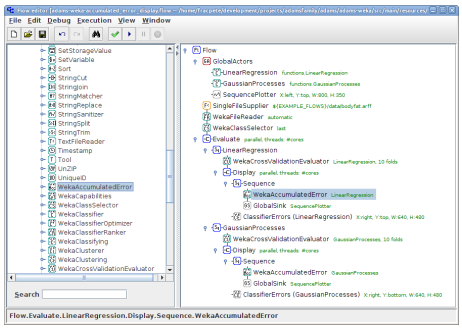

Figure 2.24: Flow for displaying the "accumulated error" of a two classifiers.

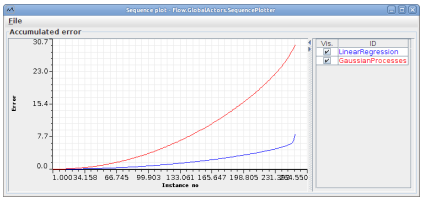

Figure 2.25: The "accumulated error" of LinearRegression and GaussianProcesses.

[15] adams-weka-accumulated_error_display.flow

2.1.5 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 2.26 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 2.27).

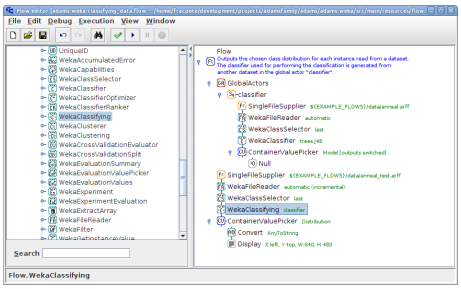

Figure 2.26: Flow for classifying new data and outputting the class distributions.

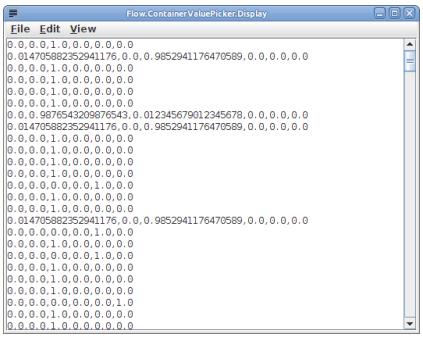

Figure 2.27: The generated class distributions for the new data.

[16] adams-weka-classifying_data.flow

2.2 Advanced

2.2.1 Learning curves

incremental [17] and non-incremental [18]

2.2.2 Experiments

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

2.2.3 Optimization

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

2.2.4 Provenance

provenance display [24]

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

Chapter 3

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.

3.1 Building models

Building clustering models is as easy as building classification/regression models. Instead of the *WekaClassifier* transformer, you use the *WekaClusterer* one.

Figures 3.1 and 3.2 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.

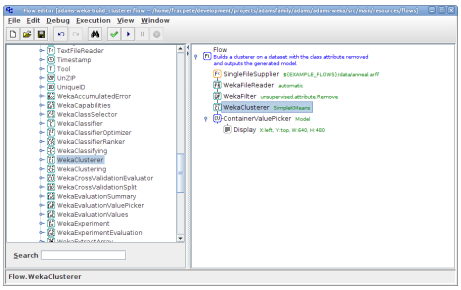

Figure 3.1: Building a clusterer and outputting the model.

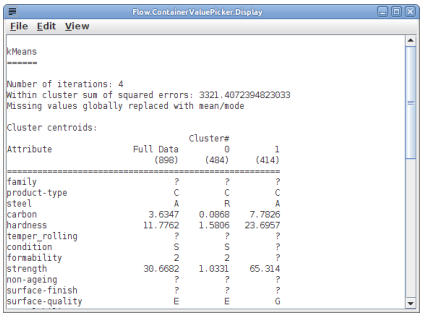

Figure 3.2: Cluster model output.

If the base cluster algorithm is an incremental one, i.e., one that implements the *weka.clusterers.UpdateableClusterer* interface, you can build your clustering model incrementally as well. The flow [2] in Figure 3.3 builds the CobWeb cluster algorithm incrementally and outputs the generated models every 25 instances (see Figure 3.4).

[1] adams-weka-build_clusterer.flow
[2] adams-weka-build_clusterer_incrementally.flow

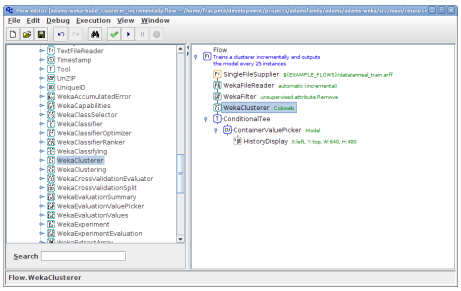

Figure 3.3: Building a clusterer incrementally and outputting the model.

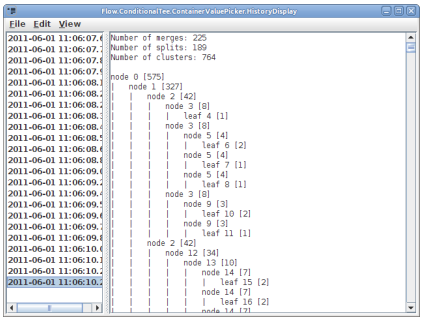

Figure 3.4: Cluster model outputs, generated every 25 instances.

3.2 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 3.5 shows how to build a clusterer and use it to cluster new data, outputting the cluster distributions (see Figure 3.6 for the generated output).

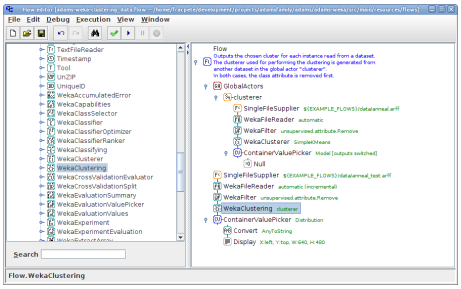

Figure 3.5: Flow for clustering new data.

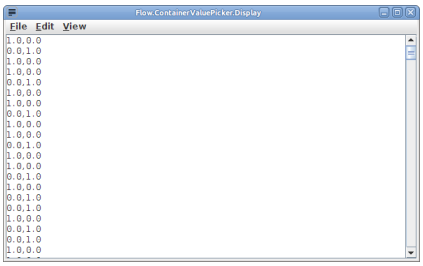

Figure 3.6: Generated cluster.

[3] adams-weka-clustering_data.flow

Bibliography

[1] *ADAMS* – Advanced Data mining and Machine learning System
`http://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.cs.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