THE UNIVERSITY OF
WAIKATO
Te Whare Wānanga o Waikato

Massive Online Analysis

Manual

Albert Bifet and Richard Kirkby

August 2009

Contents

1 Introduction 1
1.1 Data streams Evaluation 2

2 Installation 5

3 Using the GUI 7

4 Using the command line 9
4.1 Comparing two classifiers 11

5 Tasks in MOA 13
5.1 WriteStreamToARFFFile 13
5.2 MeasureStreamSpeed 13
5.3 LearnModel 14
5.4 EvaluateModel 14
5.5 EvaluatePeriodicHeldOutTest 15
5.6 EvaluateInterleavedTestThenTrain 15
5.7 EvaluatePrequential 16

6 Evolving data streams 19
6.1 Streams 19
6.1.1 ArffFileStream 19
6.1.2 ConceptDriftStream 20
6.1.3 ConceptDriftRealStream 21
6.1.4 FilteredStream 22
6.1.5 AddNoiseFilter 22
6.2 Streams Generators 22
6.2.1 generators.AgrawalGenerator 22
6.2.2 generators.HyperplaneGenerator 23
6.2.3 generators.LEDGenerator 24
6.2.4 generators.LEDGeneratorDrift 24
6.2.5 generators.RandomRBFGenerator 25
6.2.6 generators.RandomRBFGeneratorDrift 25
6.2.7 generators.RandomTreeGenerator 26
6.2.8 generators.SEAGenerator 27
6.2.9 generators.STAGGERGenerator 27
6.2.10 generators.WaveformGenerator 28
6.2.11 generators.WaveformGeneratorDrift 28

7 Classifiers **29**
7.1 Classifiers for static streams 29
7.1.1 MajorityClass 29
7.1.2 Naive Bayes 30
7.1.3 DecisionStump 30
7.1.4 HoeffdingTree 31
7.1.5 HoeffdingTreeNB 32
7.1.6 HoeffdingTreeNBAdaptive 32
7.1.7 HoeffdingOptionTree 33
7.1.8 HoeffdingOptionTreeNB 34
7.1.9 HoeffdingTreeOptionNBAdaptive 34
7.1.10 OzaBag 34
7.1.11 OzaBoost 35
7.1.12 OCBoost 35
7.2 Classifiers for evolving streams 36
7.2.1 OzaBagASHT 36
7.2.2 OzaBagADWIN 37
7.2.3 SingleClassifierDrift 39
7.2.4 AdaHoeffdingOptionTree 40

8 Writing a classifier **41**
8.1 Creating a new classifier 41
8.2 Compiling a classifier 46

9 Bi-directional interface with WEKA **47**
9.1 WEKA classifiers from MOA 47
9.1.1 WekaClassifier 48
9.1.2 SingleClassifierDrift 48
9.2 MOA classifiers from WEKA 50

1
Introduction

Massive **O**nline **A**nalysis (MOA) is a software environment for implementing algorithms and running experiments for online learning from evolving data streams.

MOA includes a collection of offline and online methods as well as tools for evaluation. In particular, it implements boosting, bagging, and Hoeffding Trees, all with and without Naïve Bayes classifiers at the leaves.

MOA is related to WEKA, the Waikato Environment for Knowledge Analysis, which is an award-winning open-source workbench containing implementations of a wide range of batch machine learning methods. WEKA is also written in Java. The main benefits of Java are portability, where applications can be run on any platform with an appropriate Java virtual machine, and the strong and well-developed support libraries. Use of the language is widespread, and features such as the automatic garbage collection help to reduce programmer burden and error.

One of the key data structures used in MOA is the description of an example from a data stream. This structure borrows from WEKA, where an example is represented by an array of double precision floating point values. This provides freedom to store all necessary types of value – numeric attribute values can be stored directly, and discrete attribute values and class labels are represented by integer index values that are stored as floating point values in the array. Double precision floating point values require storage space of 64 bits, or 8 bytes. This detail can have implications for memory usage.

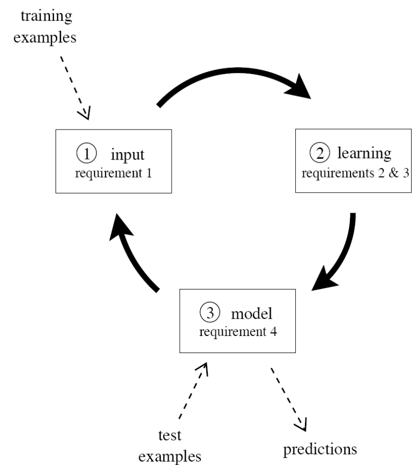


Figure 1.1: The data stream classification cycle

1.1 Data streams Evaluation

A data stream environment has different requirements from the traditional setting. The most significant are the following:

Requirement 1 Process an example at a time, and inspect it only once (at most)

Requirement 2 Use a limited amount of memory

Requirement 3 Work in a limited amount of time

Requirement 4 Be ready to predict at any time

We have to consider these requirements in order to design a new experimental framework for data streams. Figure 1.1 illustrates the typical use of a data stream classification algorithm, and how the requirements fit in a repeating cycle:

1. The algorithm is passed the next available example from the stream (requirement 1).

2. The algorithm processes the example, updating its data structures. It does so without exceeding the memory bounds set on it (requirement 2), and as quickly as possible (requirement 3).

3. The algorithm is ready to accept the next example. On request it is able to predict the class of unseen examples (requirement 4).

In traditional batch learning the problem of limited data is overcome by analyzing and averaging multiple models produced with different random arrangements of training and test data. In the stream setting the problem of (effectively) unlimited data poses different challenges. One solution involves taking snapshots at different times during the induction of a model to see how much the model improves.

The evaluation procedure of a learning algorithm determines which examples are used for training the algorithm, and which are used to test the model output by the algorithm. The procedure used historically in batch learning has partly depended on data size. As data sizes increase, practical time limitations prevent procedures that repeat training too many times. It is commonly accepted with considerably larger data sources that it is necessary to reduce the numbers of repetitions or folds to allow experiments to complete in reasonable time. When considering what procedure to use in the data stream setting, one of the unique concerns is how to build a picture of accuracy over time. Two main approaches arise:

- **Holdout**: When traditional batch learning reaches a scale where cross-validation is too time consuming, it is often accepted to instead measure performance on a single holdout set. This is most useful when the division between train and test sets have been pre-defined, so that results from different studies can be directly compared.

- **Interleaved Test-Then-Train or Prequential**: Each individual example can be used to test the model before it is used for training, and from this the accuracy can be incrementally updated. When intentionally performed in this order, the model is always being tested on examples it has not seen. This scheme has the advantage that no holdout set is needed for testing, making maximum use of the available data. It also ensures a smooth plot of accuracy over time, as each individual example will become increasingly less significant to the overall average.

As data stream classification is a relatively new field, such evaluation practices are not nearly as well researched and established as they are in the traditional batch setting. The majority of experimental evaluations use less

than one million training examples. In the context of data streams this is disappointing, because to be truly useful at data stream classification the algorithms need to be capable of handling very large (potentially infinite) streams of examples. Demonstrating systems only on small amounts of data does not build a convincing case for capacity to solve more demanding data stream applications.

MOA permits adequately evaluate data stream classification algorithms on large streams, in the order of tens of millions of examples where possible, and under explicit memory limits. Any less than this does not actually test algorithms in a realistically challenging setting.

2 Installation

The following manual is based on a Unix/Linux system with Java 5 SDK or greater installed. Other operating systems such as Microsoft Windows will be similar but may need adjustments to suit.

MOA needs the following files:

```
moa.jar
weka.jar
sizeofag.jar
```

They are available from

```
http://sourceforge.net/projects/moa-datastream/
http://sourceforge.net/projects/weka/
http://www.jroller.com/resources/m/maxim/sizeofag.jar
```

These files are needed to run the MOA software from the command line and the graphical interface:

```
java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar moa.DoTask \
  LearnModel -l DecisionStumpTutorial \
  -s generators.WaveformGenerator -m 1000000 -O model1.moa

java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar \
  moa.gui.TaskLauncher
```

or, using Microsoft Windows:

```
java -cp .;moa.jar;weka.jar -javaagent:sizeofag.jar moa.DoTask
  LearnModel -l DecisionStumpTutorial
  -s generators.WaveformGenerator -m 1000000 -O model1.moa

java -cp .;moa.jar;weka.jar -javaagent:sizeofag.jar
  moa.gui.TaskLauncher
```

3 Using the GUI

A graphical user interface for configuring and running tasks is available with the command:

```
java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar \
  moa.gui.TaskLauncher
```

Click 'Configure' to set up a task, when ready click to launch a task click 'Run'. Several tasks can be run concurrently. Click on different tasks in the list and control them using the buttons below. If textual output of a task is available it will be displayed in the bottom half of the GUI, and can be saved to disk.

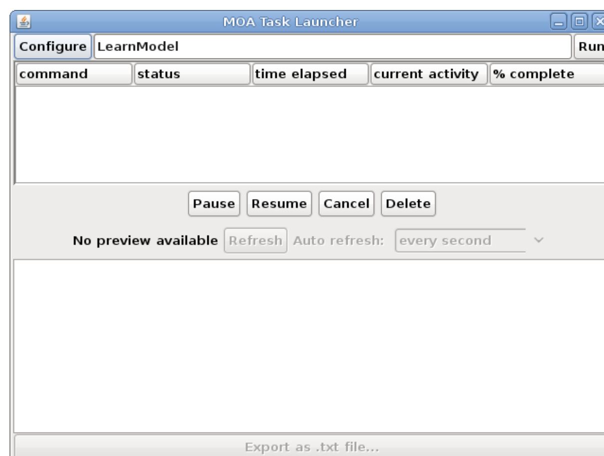


Figure 3.1: Graphical user interface of MOA

Note that the command line text box displayed at the top of the window represents textual commands that can be used to run tasks on the command line as described in the next chapter. The text can be selected then copied onto the clipboard.

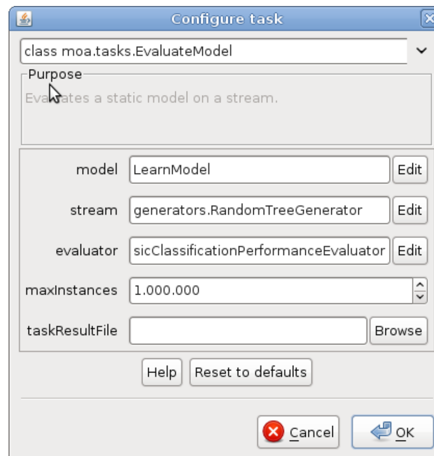


Figure 3.2: Options to set up a task in MOA

4

Using the command line

In this chapter we are going to show some examples of tasks performed using the command line.

The first example will command MOA to train the `HoeffdingTree` classifier and create a model. The `moa.DoTask` class is the main class for running tasks on the command line. It will accept the name of a task followed by any appropriate parameters. The first task used is the `LearnModel` task. The `-l` parameter specifies the learner, in this case the `HoeffdingTree` class. The `-s` parameter specifies the stream to learn from, in this case `generators.WaveformGenerator` is specified, which is a data stream generator that produces a three-class learning problem of identifying three types of waveform. The `-m` option specifies the maximum number of examples to train the learner with, in this case one million examples. The `-O` option specifies a file to output the resulting model to:

```
java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar moa.DoTask \
  LearnModel -l HoeffdingTree \
  -s generators.WaveformGenerator -m 1000000 -O model1.moa
```

This will create a file named `model1.moa` that contains the decision stump model that was induced during training.

The next example will evaluate the model to see how accurate it is on a set of examples that are generated using a different random seed. The `EvaluateModel` task is given the parameters needed to load the model produced in the previous step, generate a new waveform stream with a random seed of 2, and test on one million examples:

```
java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar moa.DoTask \
  "EvaluateModel -m file:model1.moa \
  -s (generators.WaveformGenerator -i 2) -i 1000000"
```

This is the first example of nesting parameters using brackets. Quotes have been added around the description of the task, otherwise the operating system may be confused about the meaning of the brackets.

After evaluation the following statistics are output:

```
classified instances = 1,000,000
classifications correct (percent) = 57.637
```

Note the the above two steps can be achieved by rolling them into one, avoiding the need to create an external file, as follows:

```
java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar moa.DoTask \
  "EvaluateModel -m (LearnModel -l HoeffdingTree \
  -s generators.WaveformGenerator -m 1000000) \
  -s (generators.WaveformGenerator -i 2) -i 1000000"
```

The task `EvaluatePeriodicHeldOutTest` will train a model while taking snapshots of performance using a held-out test set at periodic intervals. The following command creates a *comma separated values* file, training the `HoeffdingTree` classifier on the `WaveformGenerator` data, using the first 100 thousand examples for testing, training on a total of 100 million examples, and testing every one million examples:

```
java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar moa.DoTask \
  "EvaluatePeriodicHeldOutTest -l HoeffdingTree \
  -s generators.WaveformGenerator \
  -n 100000 -i 100000000 -f 1000000" > dsresult.csv
```

For the purposes of comparison, a bagging learner using ten decisions trees can be trained on the same problem:

```
java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar moa.DoTask \
  "EvaluatePeriodicHeldOutTest -l (OzaBag -l HoeffdingTree -s 10)\
  -s generators.WaveformGenerator \
  -n 100000 -i 100000000 -f 1000000" > htresult.csv
```

Another evaluation method implemented in MOA is *Interleaved Test-Then-Train* or *Prequential*: Each individual example is used to test the model before it is used for training, and from this the accuracy is incrementally updated. When intentionally performed in this order, the model is always being tested on examples it has not seen. This scheme has the advantage that no holdout set is needed for testing, making maximum use of the available data. It also ensures a smooth plot of accuracy over time, as each individual example will become increasingly less significant to the overall average.

An example of the EvaluateInterleavedTestThenTrain task creating a comma separated values file, training the HoeffdingTree classifier on the WaveformGenerator data, training and testing on a total of 100 million examples, and testing every one million examples, is the following:

```
java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar moa.DoTask \
  "EvaluateInterleavedTestThenTrain -l HoeffdingTree \
  -s generators.WaveformGenerator \
  -i 100000000 -f 1000000" > htresult.csv
```

4.1 Comparing two classifiers

Suppose we would like to compare the learning curves of a decision stump and a Hoeffding Tree. First, we will execute the task `EvaluatePeriodicHeldOutTest` to train a model while taking snapshots of performance using a held-out test set at periodic intervals. The following commands create *comma separated values* files, training the `DecisionStump` and the `HoeffdingTree` classifier on the `WaveformGenerator` data, using the first 100 thousand examples for testing, training on a total of 100 million examples, and testing every one million examples:

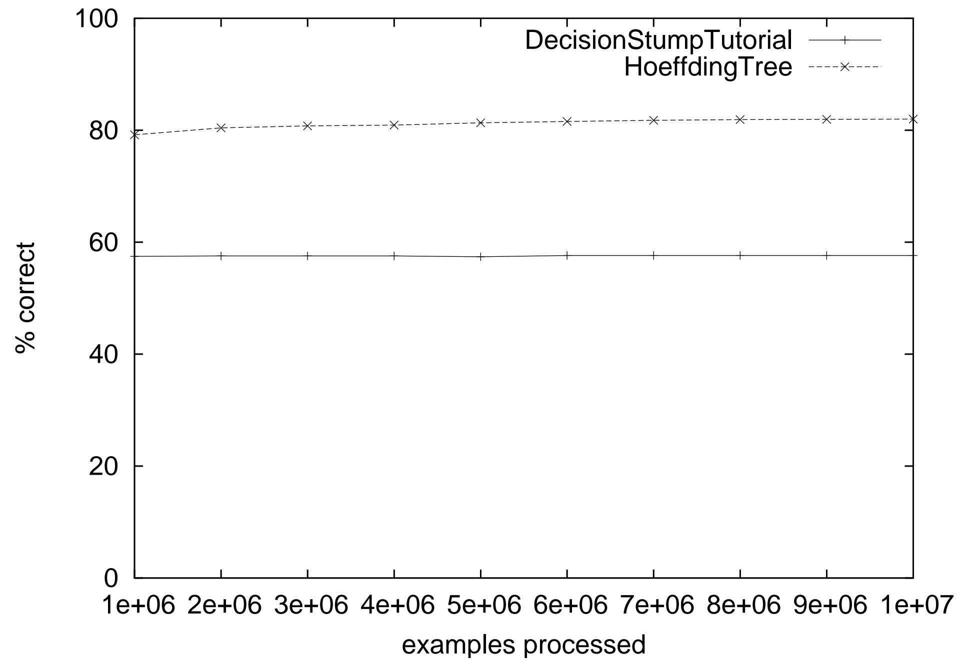
```
java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar moa.DoTask \
  "EvaluatePeriodicHeldOutTest -l DecisionStump \
  -s generators.WaveformGenerator \
  -n 100000 -i 100000000 -f 1000000" > dsresult.csv
```

```
java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar moa.DoTask \
  "EvaluatePeriodicHeldOutTest -l HoeffdingTree \
  -s generators.WaveformGenerator \
  -n 100000 -i 100000000 -f 1000000" > htresult.csv
```

Assuming that `gnuplot` is installed on the system, the learning curves can be plotted with the following commands:

```
gnuplot> set datafile separator ","
gnuplot> set ylabel "% correct"
gnuplot> set xlabel "examples processed"
gnuplot> plot [][0:100] \
  "dsresult.csv" using 1:9 with linespoints \
  title "DecisionStumpTutorial", \
  "htresult.csv" using 1:9 with linespoints \
  title "HoeffdingTree"
```

This results in the following graph:



For this problem it is obvious that a full tree can achieve higher accuracy than a single stump, and that a stump has very stable accuracy that does not improve with further training.

5 Tasks in MOA

The main Tasks in MOA are the following:

5.1 WriteStreamToARFFFile

Outputs a stream to an ARFF file. Example:

```
java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar moa.DoTask \
  "WriteStreamToARFFFile -s generators.WaveformGenerator \
  -f Wave.arff -m 100000"
```

Parameters:

- -s : Stream to write
- -f : Destination ARFF file
- -m : Maximum number of instances to write to file
- -h : Suppress header from output

5.2 MeasureStreamSpeed

Measures the speed of a stream. Example:

```
java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar moa.DoTask \
  "MeasureStreamSpeed -s generators.WaveformGenerator \
  -g 100000"
```

Parameters:

- -s : Stream to measure
- -g : Number of examples
- -O : File to save the final result of the task to

5.3 LearnModel

Learns a model from a stream. Example:

```
java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar moa.DoTask \
  LearnModel -l HoeffdingTree \
  -s generators.WaveformGenerator -m 1000000 -O model1.moa
```

Parameters:

- -l : Classifier to train
- -s : Stream to learn from
- -m : Maximum number of instances to train on per pass over the data
- -p : The number of passes to do over the data
- -b : Maximum size of model (in bytes). -1 = no limit
- -q : How many instances between memory bound checks
- -O : File to save the final result of the task to

5.4 EvaluateModel

Evaluates a static model on a stream. Example:

```
java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar moa.DoTask \
  "EvaluateModel -m (LearnModel -l HoeffdingTree \
  -s generators.WaveformGenerator -m 1000000) \
  -s (generators.WaveformGenerator -i 2) -i 1000000"
```

Parameters:

- -l : Classifier to evaluate
- -s : Stream to evaluate on
- -e : Classification performance evaluation method
- -i : Maximum number of instances to test
- -O : File to save the final result of the task to

5.5 EvaluatePeriodicHeldOutTest

Evaluates a classifier on a stream by periodically testing on a heldout set. Example:

```
java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar moa.DoTask \
  "EvaluatePeriodicHeldOutTest -l HoeffdingTree \
  -s generators.WaveformGenerator \
  -n 100000 -i 100000000 -f 1000000" > htresult.csv
```

Parameters:

- -l : Classifier to train
- -s : Stream to learn from
- -e : Classification performance evaluation method
- -n : Number of testing examples
- -i : Number of training examples, <1 = unlimited
- -t : Number of training seconds
- -f : Number of training examples between samples of learning performance
- -d : File to append intermediate csv results to
- -c : Cache test instances in memory
- -O : File to save the final result of the task to

5.6 EvaluateInterleavedTestThenTrain

Evaluates a classifier on a stream by testing then training with each example in sequence. Example:

```
java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar moa.DoTask \
  "EvaluateInterleavedTestThenTrain -l HoeffdingTree \
  -s generators.WaveformGenerator \
  -i 100000000 -f 1000000" > htresult.csv
```

Parameters:

- -l : Classifier to train

- -s : Stream to learn from
- -e : Classification performance evaluation method
- -n : Number of testing examples
- -i : Maximum number of instances to test/train on (-1 = no limit)
- -t : Maximum number of seconds to test/train for (-1 = no limit)
- -f : How many instances between samples of the learning performance
- -b : Maximum size of model (in bytes). -1 = no limit
- -q : How many instances between memory bound checks
- -d : File to append intermediate csv results to
- -O : File to save the final result of the task to

5.7 EvaluatePrequential

Evaluates a classifier on a stream by testing then training with each example in sequence. It may use a sliding window or a fading factor forgetting mechanism.

This evaluation method using sliding windows and a fading factor was presented in

[C] João Gama, Raquel Sebastião and Pedro Pereira Rodrigues. Issues in evaluation of stream learning algorithms. In *KDD'09*, pages 329–338.

The fading factor α is used as following:

$$E_i = \frac{S_i}{B_i}$$

with

$$S_i = L_i + \alpha \times S_{i-1}$$

$$B_i = n_i + \alpha \times B_{i-1}$$

where n_i is the number of examples used to compute the loss function L_i. $n_i = 1$ since the loss L_i is computed for every single example.

Examples:

```
java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar moa.DoTask \
  "EvaluatePrequential -l HoeffdingTree \
  -e WindowClassificationPerformanceEvaluator -w 10000 \
  -s generators.WaveformGenerator \
  -i 100000000 -f 1000000" > htresult.csv

java -cp .:moa.jar:weka.jar -javaagent:sizeofag.jar moa.DoTask \
  "EvaluatePrequential -l HoeffdingTree \
  -e FadingFactorClassificationPerformanceEvaluator -a .975 \
  -s generators.WaveformGenerator \
  -i 100000000 -f 1000000" > htresult.csv
```

Parameters:

- Same parameters as EvaluateInterleavedTestThenTrain
- -e : Classification performance evaluation method
  - WindowClassificationPerformanceEvaluator
  - FadingFactorClassificationPerformanceEvaluator
  - EWMAFactorClassificationPerformanceEvaluator
- -w : Size of sliding window to use with WindowClassificationPerformanceEvaluator
- -a : Fading factor to use with FadingFactorClassificationPerformanceEvaluator

6 Evolving data streams

MOA streams are build using generators, reading ARFF files, joining several streams, or filtering streams. MOA streams generators allow to simulate potentially infinite sequence of data. There are the following :

- Random Tree Generator
- SEA Concepts Generator
- STAGGER Concepts Generator
- Rotating Hyperplane
- Random RBF Generator
- LED Generator
- Waveform Generator
- Function Generator

.

6.1 Streams

Classes available in MOA to obtain input streams are the following:

6.1.1 ArffFileStream

A stream read from an ARFF file. Example:

```
ArffFileStream -f elec.arff
```

Parameters:

- -f : ARFF file to load
- -c : Class index of data. 0 for none or -1 for last attribute in file

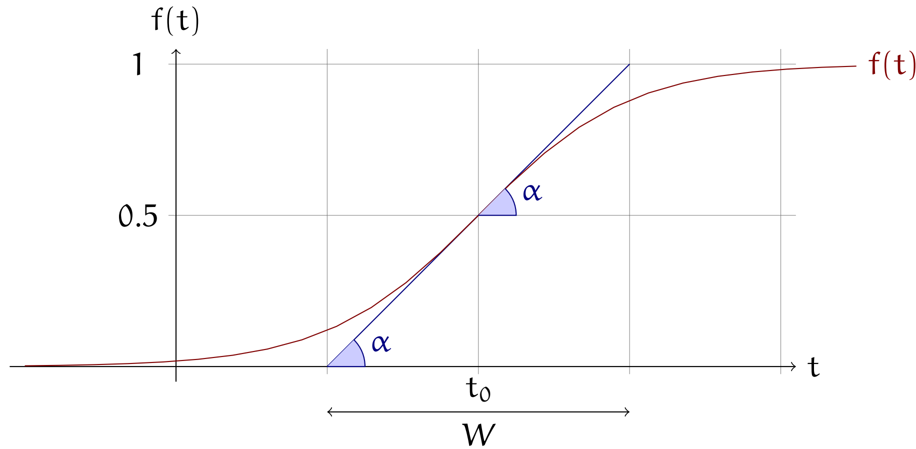


Figure 6.1: A sigmoid function $f(t) = 1/(1 + e^{-s(t-t_0)})$.

6.1.2 ConceptDriftStream

Generator that adds concept drift to examples in a stream.

Considering data streams as data generated from pure distributions, MOA models a concept drift event as a weighted combination of two pure distributions that characterizes the target concepts before and after the drift. MOA uses the sigmoid function, as an elegant and practical solution to define the probability that every new instance of the stream belongs to the new concept after the drift.

We see from Figure 6.1 that the sigmoid function

$$f(t) = 1/(1 + e^{-s(t-t_0)})$$

has a derivative at the point t_0 equal to $f'(t_0) = s/4$. The tangent of angle α is equal to this derivative, $\tan\alpha = s/4$. We observe that $\tan\alpha = 1/W$, and as $s = 4\tan\alpha$ then $s = 4/W$. So the parameter s in the sigmoid gives the length of W and the angle α. In this sigmoid model we only need to specify two parameters : t_0 the point of change, and W the length of change. Note that for any positive real number β

$$f(t_0 + \beta \cdot W) = 1 - f(t_0 - \beta \cdot W),$$

and that $f(t_0+\beta\cdot W)$ and $f(t_0-\beta\cdot W)$ are constant values that don't depend on t_0 and W:

$$f(t_0 + W/2) = 1 - f(t_0 - W/2) = 1/(1 + e^{-2}) \approx 88.08\%$$

$$f(t_0 + W) = 1 - f(t_0 - W) = 1/(1 + e^{-4}) \approx 98.20\%$$

$$f(t_0 + 2W) = 1 - f(t_0 - 2W) = 1/(1 + e^{-8}) \approx 99.97\%$$

Definition 1. *Given two data streams* a, b, *we define* $c = a \oplus_{t_0}^{W} b$ *as the data stream built joining the two data streams* a *and* b, *where* t_0 *is the point of change,* W *is the length of change and*

- $\Pr[c(t) = a(t)] = e^{-4(t-t_0)/W}/(1 + e^{-4(t-t_0)/W})$
- $\Pr[c(t) = b(t)] = 1/(1 + e^{-4(t-t_0)/W})$.

Example:

```
ConceptDriftStream -s (generators.AgrawalGenerator -f 7)
   -d (generators.AgrawalGenerator -f 2) -w 1000000 -p 900000
```

Parameters:

- -s : Stream
- -d : Concept drift Stream
- -p : Central position of concept drift change
- -w : Width of concept drift change

6.1.3 ConceptDriftRealStream

Generator that adds concept drift to examples in a stream with different classes and attributes. Example: real datasets
Example:

```
ConceptDriftRealStream -s (ArffFileStream -f covtype.arff) \
   -d (ConceptDriftRealStream -s (ArffFileStream -f PokerOrig.arff) \
   -d (ArffFileStream -f elec.arff) -w 5000 -p 1000000 ) -w 5000 -p 581012
```

Parameters:

- -s : Stream
- -d : Concept drift Stream
- -p : Central position of concept drift change
- -w : Width of concept drift change

6.1.4 FilteredStream

A stream that is filtered.

Parameters:

- -s : Stream to filter
- -f : Filters to apply : AddNoiseFilter

6.1.5 AddNoiseFilter

Adds random noise to examples in a stream. Only to use with FilteredStream.

Parameters:

- -r : Seed for random noise
- -a : The fraction of attribute values to disturb
- -c : The fraction of class labels to disturb

6.2 Streams Generators

The classes available to generate streams are the following:

6.2.1 generators.AgrawalGenerator

Generates one of ten different pre-defined loan functions

It was introduced by Agrawal et al. in

[A] R. Agrawal, T. Imielinski, and A. Swami. Database mining: A performance perspective. *IEEE Trans. on Knowl. and Data Eng.*, 5(6):914–925, 1993.

It was a common source of data for early work on scaling up decision tree learners. The generator produces a stream containing nine attributes, six numeric and three categorical. Although not explicitly stated by the authors, a sensible conclusion is that these attributes describe hypothetical loan applications. There are ten functions defined for generating binary class labels from the attributes. Presumably these determine whether the loan should be approved.

A public C source code is available. The built in functions are based on the paper (page 924), which turn out to be functions pred20 thru pred29

in the public C implementation Perturbation function works like C implementation rather than description in paper

Parameters:

- -f : Classification function used, as defined in the original paper.
- -i : Seed for random generation of instances.
- -p : The amount of peturbation (noise) introduced to numeric values
- -b : Balance the number of instances of each class.

6.2.2 generators.HyperplaneGenerator

Generates a problem of predicting class of a rotating hyperplane.

It was used as testbed for CVFDT versus VFDT in

[C] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *KDD'01*, pages 97–106, San Francisco, CA, 2001. ACM Press.

A hyperplane in d-dimensional space is the set of points x that satisfy

$$\sum_{i=1}^{d} w_i x_i = w_0 = \sum_{i=1}^{d} w_i$$

where x_i, is the ith coordinate of x. Examples for which $\sum_{i=1}^{d} w_i x_i \geq w_0$ are labeled positive, and examples for which $\sum_{i=1}^{d} w_i x_i < w_0$ are labeled negative. Hyperplanes are useful for simulating time-changing concepts, because we can change the orientation and position of the hyperplane in a smooth manner by changing the relative size of the weights. We introduce change to this dataset adding drift to each weight attribute $w_i = w_i + d\sigma$, where σ is the probability that the direction of change is reversed and d is the change applied to every example.

Parameters:

- -i : Seed for random generation of instances.
- -c : The number of classes to generate
- -a : The number of attributes to generate.
- -k : The number of attributes with drift.

- -t : Magnitude of the change for every example
- -n : Percentage of noise to add to the data.
- -s : Percentage of probability that the direction of change is reversed

6.2.3 generators.LEDGenerator

Generates a problem of predicting the digit displayed on a 7-segment LED display.

This data source originates from the CART book. An implementation in C was donated to the UCI machine learning repository by David Aha. The goal is to predict the digit displayed on a seven-segment LED display, where each attribute has a 10% chance of being inverted. It has an optimal Bayes classification rate of 74%. The particular configuration of the generator used for experiments (led) produces 24 binary attributes, 17 of which are irrelevant.

Parameters:

- -i : Seed for random generation of instances.
- -n : Percentage of noise to add to the data
- -s : Reduce the data to only contain 7 relevant binary attributes

6.2.4 generators.LEDGeneratorDrift

Generates a problem of predicting the digit displayed on a 7-segment LED display with drift.

Parameters:

- -i : Seed for random generation of instances.
- -n : Percentage of noise to add to the data
- -s : Reduce the data to only contain 7 relevant binary attributes
- -d : Number of attributes with drift

6.2.5 generators.RandomRBFGenerator

Generates a random radial basis function stream.

This generator was devised to offer an alternate complex concept type that is not straightforward to approximate with a decision tree model. The RBF (Radial Basis Function) generator works as follows: A fixed number of random centroids are generated. Each center has a random position, a single standard deviation, class label and weight. New examples are generated by selecting a center at random, taking weights into consideration so that centers with higher weight are more likely to be chosen. A random direction is chosen to offset the attribute values from the central point. The length of the displacement is randomly drawn from a Gaussian distribution with standard deviation determined by the chosen centroid. The chosen centroid also determines the class label of the example. This effectively creates a normally distributed hypersphere of examples surrounding each central point with varying densities. Only numeric attributes are generated.

Parameters:

- -r : Seed for random generation of model
- -i : Seed for random generation of instances
- -c : The number of classes to generate
- -a : The number of attributes to generate
- -n : The number of centroids in the model

6.2.6 generators.RandomRBFGeneratorDrift

Generates a random radial basis function stream with drift. Drift is introduced by moving the centroids with constant speed.

Parameters:

- -r : Seed for random generation of model
- -i : Seed for random generation of instances
- -c : The number of classes to generate
- -a : The number of attributes to generate
- -n : The number of centroids in the model

- -s : Speed of change of centroids in the model.
- -k : The number of centroids with drift

6.2.7 generators.RandomTreeGenerator

Generates a stream based on a randomly generated tree.

This generator is based on that proposed in

[D] P. Domingos and G. Hulten. Mining high-speed data streams. In *Knowledge Discovery and Data Mining*, pages 71–80, 2000.

It produces concepts that in theory should favour decision tree learners. It constructs a decision tree by choosing attributes at random to split, and assigning a random class label to each leaf. Once the tree is built, new examples are generated by assigning uniformly distributed random values to attributes which then determine the class label via the tree.

The generator has parameters to control the number of classes, attributes, nominal attribute labels, and the depth of the tree. For consistency between experiments, two random trees were generated and fixed as the base concepts for testingone simple and the other complex, where complexity refers to the number of attributes involved and the size of the tree.

A degree of noise can be introduced to the examples after generation. In the case of discrete attributes and the class label, a probability of noise parameter determines the chance that any particular value is switched to something other than the original value. For numeric attributes, a degree of random noise is added to all values, drawn from a random Gaussian distribution with standard deviation equal to the standard deviation of the original values multiplied by noise probability.

Parameters:

- -r: Seed for random generation of tree
- -i: Seed for random generation of instances
- -c: The number of classes to generate
- -o: The number of nominal attributes to generate
- -u: The number of numeric attributes to generate
- -v: The number of values to generate per nominal attribute
- -d: The maximum depth of the tree concept

- -l: The first level of the tree above maxTreeDepth that can have leaves
- -f: The fraction of leaves per level from firstLeafLevel onwards

6.2.8 generators.SEAGenerator

Generates SEA concepts functions. This dataset contains abrupt concept drift, first introduced in paper:

[S] W. N. Street and Y. Kim. A streaming ensemble algorithm (SEA) for large-scale classification. In *KDD '01*, pages 377–382, New York, NY, USA, 2001. ACM Press.

It is generated using three attributes, where only the two first attributes are relevant. All three attributes have values between 0 and 10. The points of the dataset are divided into 4 blocks with different concepts. In each block, the classification is done using $f_1 + f_2 \leq \theta$, where f_1 and f_2 represent the first two attributes and θ is a threshold value. The most frequent values are 9, 8, 7 and 9.5 for the data blocks.

Parameters:

- -f: Classification function used, as defined in the original paper
- -i: Seed for random generation of instances
- -b: Balance the number of instances of each class
- -n: Percentage of noise to add to the data

6.2.9 generators.STAGGERGenerator

Generates STAGGER Concept functions. They were introduced by Schlimmer and Granger in

[ST] J. C. Schlimmer and R. H. Granger. Incremental learning from noisy data. *Machine Learning*, 1(3):317–354, 1986.

The STAGGER Concepts are boolean functions of three attributes encoding objects: size (small, medium, and large), shape (circle, triangle, and rectangle), and colour (red,blue, and green). A concept description covering either green rectangles or red triangles is represented by (shape= rectangle and colour=green) or (shape=triangle and colour=red).

Parameters:

1. -i: Seed for random generation of instances
2. -f: Classification function used, as defined in the original paper
3. -b: Balance the number of instances of each class

6.2.10 generators.WaveformGenerator

Generates a problem of predicting one of three waveform types.

It shares its origins with LED, and was also donated by David Aha to the UCI repository. The goal of the task is to differentiate between three different classes of waveform, each of which is generated from a combination of two or three base waves. The optimal Bayes classification rate is known to be 86%. There are two versions of the problem, wave21 which has 21 numeric attributes, all of which include noise, and wave40 which introduces an additional 19 irrelevant attributes.

Parameters:

- -i: Seed for random generation of instances
- -n: Adds noise, for a total of 40 attributes

6.2.11 generators.WaveformGeneratorDrift

Generates a problem of predicting one of three waveform types with drift.

Parameters:

- -i: Seed for random generation of instances
- -n: Adds noise, for a total of 40 attributes
- -d: Number of attributes with drift

7 Classifiers

The classifiers implemented in MOA are the following:

- Naive Bayes
- Decision Stump
- Hoeffding Tree
- Hoeffding Option Tree
- Bagging
- Boosting
- Bagging using `ADWIN`
- Bagging using Adaptive-Size Hoeffding Trees.

7.1 Classifiers for static streams

7.1.1 MajorityClass

Always predicts the class that has been observed most frequently the in the training data.

Parameters:

- -r : Seed for random behaviour of the classifier

7.1.2 Naive Bayes

Performs classic bayesian prediction while making naive assumption that all inputs are independent.

Naïve Bayes is a classifier algorithm known for its simplicity and low computational cost. Given n_C different classes, the trained Naïve Bayes classifier predicts for every unlabelled instance I the class C to which it belongs with high accuracy.

The model works as follows: Let $x_1, \ldots, x_k$ be k discrete attributes, and assume that x_i can take n_i different values. Let C be the class attribute, which can take n_C different values. Upon receiving an unlabelled instance $I = (x_1 = v_1, \ldots, x_k = v_k)$, the Naïve Bayes classifier computes a "probability" of I being in class c as:

$$\begin{aligned} \Pr[C = c|I] &\cong \prod_{i=1}^{k} \Pr[x_i = v_i|C = c] \\ &= \Pr[C = c] \cdot \prod_{i=1}^{k} \frac{\Pr[x_i = v_i \wedge C = c]}{\Pr[C = c]} \end{aligned}$$

The values $\Pr[x_i = v_j \wedge C = c]$ and $\Pr[C = c]$ are estimated from the training data. Thus, the summary of the training data is simply a 3-dimensional table that stores for each triple (x_i, v_j, c) a count $N_{i,j,c}$ of training instances with $x_i = v_j$, together with a 1-dimensional table for the counts of $C = c$. This algorithm is naturally incremental: upon receiving a new example (or a batch of new examples), simply increment the relevant counts. Predictions can be made at any time from the current counts.

Parameters:

- -r : Seed for random behaviour of the classifier

7.1.3 DecisionStump

Decision trees of one level.

Parameters:

- -g : The number of instances to observe between model changes
- -b : Only allow binary splits
- -c : Split criterion to use. Example : InfoGainSplitCriterion
- -r : Seed for random behaviour of the classifier

7.1.4 HoeffdingTree

Decision tree for streaming data.

A *Hoeffding tree* is an incremental, anytime decision tree induction algorithm that is capable of learning from massive data streams, assuming that the distribution generating examples does not change over time. Hoeffding trees exploit the fact that a small sample can often be enough to choose an optimal splitting attribute. This idea is supported mathematically by the Hoeffding bound, which quantifies the number of observations (in our case, examples) needed to estimate some statistics within a prescribed precision (in our case, the goodness of an attribute). More precisely, the Hoeffding bound states that with probability $1-\delta$, the true mean of a random variable of range R will not differ from the estimated mean after n independent observations by more than:

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}.$$

A theoretically appealing feature of Hoeffding Trees not shared by other incremental decision tree learners is that it has sound guarantees of performance. Using the Hoeffding bound one can show that its output is asymptotically nearly identical to that of a non-incremental learner using infinitely many examples. See for details:

[C] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *KDD'01*, pages 97–106, San Francisco, CA, 2001. ACM Press.

Parameters:

- -m : Maximum memory consumed by the tree
- -n : Numeric estimator to use :
  - Gaussian approximation evaluating 10 splitpoints
  - Gaussian approximation evaluating 100 splitpoints
  - Greenwald-Khanna quantile summary with 10 tuples
  - Greenwald-Khanna quantile summary with 100 tuples
  - Greenwald-Khanna quantile summary with 1000 tuples
  - VFML method with 10 bins
  - VFML method with 100 bins

  - VFML method with 1000 bins
  - Exhaustive binary tree

- -e : How many instances between memory consumption checks
- -g : The number of instances a leaf should observe between split attempts
- -s : Split criterion to use. Example : InfoGainSplitCriterion
- -c : The allowable error in split decision, values closer to 0 will take longer to decide
- -t : Threshold below which a split will be forced to break ties
- -b : Only allow binary splits
- -z : Stop growing as soon as memory limit is hit
- -r : Disable poor attributes
- -p : Disable pre-pruning

7.1.5 HoeffdingTreeNB

Decision tree for streaming data with Naive Bayes classification at leaves.

- Same parameters as HoeffdingTree
- -q : The number of instances a leaf should observe before permitting Naive Bayes

7.1.6 HoeffdingTreeNBAdaptive

Decision tree for streaming data with adaptive Naive Bayes classification at leaves. This adaptive Naive Bayes prediction method monitors the error rate of majority class and Naive Bayes decisions in every leaf, and chooses to employ Naive Bayes decisions only where they have been more accurate in past cases.

- Same parameters as HoeffdingTreeNB

7.1.7 HoeffdingOptionTree

Decision option tree for streaming data

Hoeffding Option Trees are regular Hoeffding trees containing additional option nodes that allow several tests to be applied, leading to multiple Hoeffding trees as separate paths. They consist of a single structure that efficiently represents multiple trees. A particular example can travel down multiple paths of the tree, contributing, in different ways, to different options.

See for details:

[OP] B. Pfahringer, G. Holmes, and R. Kirkby. New options for hoeffding trees. In *AI*, pages 90–99, 2007.

Parameters:

- -o : Maximum number of option paths per node
- -m : Maximum memory consumed by the tree
- -n : Numeric estimator to use :
  - Gaussian approximation evaluating 10 splitpoints
  - Gaussian approximation evaluating 100 splitpoints
  - Greenwald-Khanna quantile summary with 10 tuples
  - Greenwald-Khanna quantile summary with 100 tuples
  - Greenwald-Khanna quantile summary with 1000 tuples
  - VFML method with 10 bins
  - VFML method with 100 bins
  - VFML method with 1000 bins
  - Exhaustive binary tree
- -e : How many instances between memory consumption checks
- -g : The number of instances a leaf should observe between split attempts
- -s : Split criterion to use. Example : InfoGainSplitCriterion
- -c : The allowable error in split decision, values closer to 0 will take longer to decide

- -w : The allowable error in secondary split decisions, values closer to 0 will take longer to decide
- -t : Threshold below which a split will be forced to break ties
- -b : Only allow binary splits
- -z : Memory strategy to use
- -r : Disable poor attributes
- -p : Disable pre-pruning
- -d : File to append option table to.

7.1.8 HoeffdingOptionTreeNB

Decision option tree for streaming data with Naive Bayes classification at leaves.

Parameters:

- Same parameters as HoeffdingOptionTree
- -q : The number of instances a leaf should observe before permitting Naive Bayes

7.1.9 HoeffdingTreeOptionNBAdaptive

Decision option tree for streaming data with adaptive Naive Bayes classification at leaves. This adaptive Naive Bayes prediction method monitors the error rate of majority class and Naive Bayes decisions in every leaf, and chooses to employ Naive Bayes decisions only where they have been more accurate in past cases.

Parameters:

- Same parameters as HoeffdingOptionTreeNB

7.1.10 OzaBag

Incremental on-line bagging of Oza and Russell.

Oza and Russell developed online versions of bagging and boosting for Data Streams. They show how the process of sampling bootstrap replicates from training data can be simulated in a data stream context. They observe that the probability that any individual example will be chosen for a replicate tends to a Poisson(1) distribution.

[OR] N. Oza and S. Russell. Online bagging and boosting. In *Artificial Intelligence and Statistics 2001*, pages 105–112. Morgan Kaufmann, 2001.

Parameters:

- -l : Classifier to train
- -s : The number of models in the bag

7.1.11 OzaBoost

Incremental on-line boosting of Oza and Russell.

See details in:

[OR] N. Oza and S. Russell. Online bagging and boosting. In *Artificial Intelligence and Statistics 2001*, pages 105–112. Morgan Kaufmann, 2001.

For the boosting method, Oza and Russell note that the weighting procedure of AdaBoost actually divides the total example weight into two halves – half of the weight is assigned to the correctly classified examples, and the other half goes to the misclassified examples. They use the Poisson distribution for deciding the random probability that an example is used for training, only this time the parameter changes according to the boosting weight of the example as it is passed through each model in sequence.

Parameters:

- -l : Classifier to train
- -s : The number of models to boost
- -p : Boost with weights only; no poisson

7.1.12 OCBoost

Online Coordinate Boosting.

Pelossof et al. presented Online Coordinate Boosting, a new online boosting algorithm for adapting the weights of a boosted classifier, which yields a closer approximation to Freund and Schapire's AdaBoost algorithm. The weight update procedure is derived by minimizing AdaBoost's loss when viewed in an incremental form. This boosting method may be reduced to a form similar to Oza and Russell's algorithm.

See details in:

[PJ] Raphael Pelossof, Michael Jones, Ilia Vovsha, and Cynthia Rudin. Online coordinate boosting. 2008.

Example:

```
OCBoost -l HoeffdingTreeNBAdaptive -e 0.5
```

Parameters:

- -l : Classifier to train
- -s : The number of models to boost
- -e : Smoothing parameter

7.2 Classifiers for evolving streams

7.2.1 OzaBagASHT

Bagging using trees of different size. The Adaptive-Size Hoeffding Tree (ASHT) is derived from the Hoeffding Tree algorithm with the following differences:

- it has a maximum number of split nodes, or *size*
- after one node splits, if the number of split nodes of the ASHT tree is higher than the maximum value, then it deletes some nodes to reduce its size

The intuition behind this method is as follows: smaller trees adapt more quickly to changes, and larger trees do better during periods with no or little change, simply because they were built on more data. Trees limited to size s will be reset about twice as often as trees with a size limit of 2s. This creates a set of different reset-speeds for an ensemble of such trees, and therefore a subset of trees that are a good approximation for the current rate of change. It is important to note that resets will happen all the time, even for stationary datasets, but this behaviour should not have a negative impact on the ensemble's predictive performance.

When the tree size exceeds the maximun size value, there are two different delete options:

- delete the oldest node, the root, and all of its children except the one where the split has been made. After that, the root of the child not deleted becomes the new root

- delete all the nodes of the tree, i.e., restart from a new root.

The maximum allowed size for the n-th ASHT tree is twice the maximum allowed size for the $(n-1)$-th tree. Moreover, each tree has a weight proportional to the inverse of the square of its error, and it monitors its error with an exponential weighted moving average (EWMA) with $\alpha = .01$. The size of the first tree is 2.

With this new method, it is attempted to improve bagging performance by increasing tree diversity. It has been observed that boosting tends to produce a more diverse set of classifiers than bagging, and this has been cited as a factor in increased performance.

See more details in:

[BHPKG] Albert Bifet, Geoff Holmes, Bernhard Pfahringer, Richard Kirkby, and Ricard Gavaldà. New ensemble methods for evolving data streams. In *15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009.

The learner must be ASHoeffdingTree, a Hoeffding Tree with a maximum size value.

Example:

```
OzaBagASHT -l ASHoeffdingTree -s 10 -u -r
```

Parameters:

- Same parameters as OzaBag
- -f : the size of first classifier in the bag.
- -u : Enable weight classifiers
- -r : Reset trees when size is higher than the max

7.2.2 OzaBagADWIN

Bagging using `ADWIN`. `ADWIN` is a change detector and estimator that solves in a well-specified way the problem of tracking the average of a stream of bits or real-valued numbers. `ADWIN` keeps a variable-length window of recently seen items, with the property that the window has the maximal length statistically consistent with the hypothesis “there has been no change in the average value inside the window”.

More precisely, an older fragment of the window is dropped if and only if there is enough evidence that its average value differs from that

of the rest of the window. This has two consequences: one, that change reliably declared whenever the window shrinks; and two, that at any time the average over the existing window can be reliably taken as an estimation of the current average in the stream (barring a very small or very recent change that is still not statistically visible). A formal and quantitative statement of these two points (a theorem) appears in

[BG07c] Albert Bifet and Ricard Gavaldà. Learning from time-changing data with adaptive windowing. In *SIAM International Conference on Data Mining*, 2007.

`ADWIN` is parameter- and assumption-free in the sense that it automatically detects and adapts to the current rate of change. Its only parameter is a confidence bound δ, indicating how confident we want to be in the algorithm's output, inherent to all algorithms dealing with random processes.

Also important, `ADWIN` does not maintain the window explicitly, but compresses it using a variant of the exponential histogram technique. This means that it keeps a window of length W using only $O(\log W)$ memory and $O(\log W)$ processing time per item.

`ADWIN` Bagging is the online bagging method of Oza and Rusell with the addition of the `ADWIN` algorithm as a change detector and as an estimator for the weights of the boosting method. When a change is detected, the worst classifier of the ensemble of classifiers is removed and a new classifier is added to the ensemble.

See details in:

[BHPKG] Albert Bifet, Geoff Holmes, Bernhard Pfahringer, Richard Kirkby, and Ricard Gavaldà. New ensemble methods for evolving data streams. In *15th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2009.

Example:

```
OzaBagAdwin -l HoeffdingTreeNBAdaptive -s 10
```

Parameters:

- -l : Classifier to train
- -s : The number of models in the bag

7.2.3 SingleClassifierDrift

Class for handling concept drift datasets with a wrapper on a classifier.

The drift detection method (DDM) proposed by Gama et al. controls the number of errors produced by the learning model during prediction. It compares the statistics of two windows: the first one contains all the data, and the second one contains only the data from the beginning until the number of errors increases. Their method doesn't store these windows in memory. It keeps only statistics and a window of recent errors.

They consider that the number of errors in a sample of examples is modeled by a binomial distribution. A significant increase in the error of the algorithm, suggests that the class distribution is changing and, hence, the actual decision model is supposed to be inappropriate. They check for a warning level and a drift level. Beyond these levels, change of context is considered.

The number of errors in a sample of n examples is modeled by a binomial distribution. For each point i in the sequence that is being sampled, the error rate is the probability of misclassifying (p_i), with standard deviation given by $s_i = \sqrt{p_i(1-p_i)/i}$. A significant increase in the error of the algorithm, suggests that the class distribution is changing and, hence, the actual decision model is supposed to be inappropriate. Thus, they store the values of p_i and s_i when $p_i + s_i$ reaches its minimum value during the process (obtaining p_{pmin} and s_{min}). And it checks when the following conditions trigger:

- $p_i + s_i \geq p_{min} + 2 \cdot s_{min}$ for the warning level. Beyond this level, the examples are stored in anticipation of a possible change of context.

- $p_i + s_i \geq p_{min} + 3 \cdot s_{min}$ for the drift level. Beyond this level, the model induced by the learning method is reset and a new model is learnt using the examples stored since the warning level triggered.

Baena-García et al. proposed a new method EDDM in order to improve DDM. It is based on the estimated distribution of the distances between classification errors. The window resize procedure is governed by the same heuristics.

See more details in:

[GMCR] J. Gama, P. Medas, G. Castillo, and P. Rodrigues. Learning with drift detection. In *SBIA Brazilian Symposium on Artificial Intelligence*, pages 286–295, 2004.

[BDF] Manuel Baena-García, José del Campo-Ávila, Raúl Fidalgo, Albert Bifet, Ricard Gavaldá, and Rafael Morales-Bueno. Early drift detection method. In *Fourth International Workshop on Knowledge Discovery from Data Streams*, 2006.

Example:

```
SingleClassifierDrift -d EDDM -l HoeffdingTreeNBAdaptive
```

Parameters:

- -l : Classifier to train
- -d : Drift detection method to use: DDM or EDDM

7.2.4 AdaHoeffdingOptionTree

Adaptive decision option tree for streaming data with adaptive Naive Bayes classification at leaves.

An *Adaptive Hoeffding Option Tree* is a Hoeffding Option Tree with the following improvement: each leaf stores an estimation of the current error. It uses an EWMA estimator with $\alpha = .2$. The weight of each node in the voting process is proportional to the square of the inverse of the error.

Example:

```
AdaHoeffdingOptionTree -o 50
```

Parameters:

- Same parameters as HoeffdingOptionTreeNB

8 Writing a classifier

8.1 Creating a new classifier

To demonstrate the implementation and operation of learning algorithms in the system, the Java code of a simple decision stump classifier is studied. The classifier monitors the result of splitting on each attribute and chooses the attribute the seems to best separate the classes, based on information gain. The decision is revisited many times, so the stump has potential to change over time as more examples are seen. In practice it is unlikely to change after sufficient training.

To describe the implementation, relevant code fragments are discussed in turn, with the entire code listed (Listing 8.7) at the end. The line numbers from the fragments match up with the final listing.

A simple approach to writing a classifier is to extend `moa.classifiers.AbstractClassifier` (line 10), which will take care of certain details to ease the task.

Listing 8.1: Option handling

```
14        public IntOption gracePeriodOption = new IntOption("gracePeriod", 'g',
15                "The number of instances to observe between model changes.",
16                1000, 0, Integer.MAX_VALUE);
17
18        public FlagOption binarySplitsOption = new FlagOption("binarySplits", 'b',
19                "Only allow binary splits.");
20
21        public ClassOption splitCriterionOption = new ClassOption("splitCriterion",
22                'c', "Split criterion to use.", SplitCriterion.class,
23                "InfoGainSplitCriterion");
```

To set up the public interface to the classifier, the options available to the user must be specified. For the system to automatically take care of option handling, the options need to be public members of the class, that extend the `moa.options.Option` type.

The decision stump classifier example has three options, each of a different type. The meaning of the first three parameters used to construct options are consistent between different option types. The first parameter

is a short name used to identify the option. The second is a character intended to be used on the command line. It should be unique—a command line character cannot be repeated for different options otherwise an exception will be thrown. The third standard parameter is a string describing the purpose of the option. Additional parameters to option constructors allow things such as default values and valid ranges to be specified.

The first option specified for the decision stump classifier is the "grace period". The option is expressed with an integer, so the option has the type `IntOption`. The parameter will control how frequently the best stump is reconsidered when learning from a stream of examples. This increases the efficiency of the classifier—evaluating after every single example is expensive, and it is unlikely that a single example will change the decision of the current best stump. The default value of 1000 means that the choice of stump will be re-evaluated only after 1000 examples have been observed since the last evaluation. The last two parameters specify the range of values that are allowed for the option—it makes no sense to have a negative grace period, so the range is restricted to integers 0 or greater.

The second option is a flag, or a binary switch, represented by a `FlagOption`. By default all flags are turned off, and will be turned on only when a user requests so. This flag controls whether the decision stumps should only be allowed to split two ways. By default the stumps are allowed have more than two branches.

The third option determines the split criterion that is used to decide which stumps are the best. This is a `ClassOption` that requires a particular Java class of the type `SplitCriterion`. If the required class happens to be an `OptionHandler` then those options will be used to configure the object that is passed in.

Listing 8.2: Miscellaneous fields

```
25      protected AttributeSplitSuggestion bestSplit;
26
27      protected DoubleVector observedClassDistribution;
28
29      protected AutoExpandVector<AttributeClassObserver> attributeObservers;
30
31      protected double weightSeenAtLastSplit;
32
33      public boolean isRandomizable() {
34              return false;
35      }
```

Four global variables are used to maintain the state of the classifier.

The `bestSplit` field maintains the current stump that has been chosen by the classifier. It is of type `AttributeSplitSuggestion`, a class used to split instances into different subsets.

The `observedClassDistribution` field remembers the overall distribution of class labels that have been observed by the classifier. It is of

type DoubleVector, which is a handy class for maintaining a vector of floating point values without having to manage its size.

The attributeObservers field stores a collection of AttributeClassObservers, one for each attribute. This is the information needed to decide which attribute is best to base the stump on.

The weightSeenAtLastSplit field records the last time an evaluation was performed, so that it can be determined when another evaluation is due, depending on the grace period parameter.

The isRandomizable() function needs to be implemented to specify whether the classifier has an element of randomness. If it does, it will automatically be set up to accept a random seed. This classifier is does not, so false is returned.

Listing 8.3: Preparing for learning

```
37      @Override
38      public void resetLearningImpl() {
39              this.bestSplit = null;
40              this.observedClassDistribution = new DoubleVector();
41              this.attributeObservers = new AutoExpandVector<AttributeClassObserver>();
42              this.weightSeenAtLastSplit = 0.0;
43      }
```

This function is called before any learning begins, so it should set the default state when no information has been supplied, and no training examples have been seen. In this case, the four global fields are set to sensible defaults.

Listing 8.4: Training on examples

```
45      @Override
46      public void trainOnInstanceImpl(Instance inst) {
47              this.observedClassDistribution.addToValue((int) inst.classValue(), inst
48                              .weight());
49              for (int i = 0; i < inst.numAttributes() - 1; i++) {
50                      int instAttIndex = modelAttIndexToInstanceAttIndex(i, inst);
51                      AttributeClassObserver obs = this.attributeObservers.get(i);
52                      if (obs == null) {
53                              obs = inst.attribute(instAttIndex).isNominal() ?
54                                      newNominalClassObserver() : newNumericClassObserver();
55                              this.attributeObservers.set(i, obs);
56                      }
57                      obs.observeAttributeClass(inst.value(instAttIndex), (int) inst
58                                      .classValue(), inst.weight());
59              }
60              if (this.trainingWeightSeenByModel - this.weightSeenAtLastSplit >=
61                              this.gracePeriodOption.getValue()) {
62                      this.bestSplit = findBestSplit((SplitCriterion)
63                              getPreparedClassOption(this.splitCriterionOption));
64                      this.weightSeenAtLastSplit = this.trainingWeightSeenByModel;
65              }
66      }
```

This is the main function of the learning algorithm, called for every training example in a stream. The first step, lines 47-48, updates the overall recorded distribution of classes. The loop from line 49 to line 59 repeats for every attribute in the data. If no observations for a particular attribute have been seen previously, then lines 53-55 create a new observing object. Lines 57-58 update the observations with the values from the new exam-

ple. Lines 60-61 check to see if the grace period has expired. If so, the best split is re-evaluated.

Listing 8.5: Functions used during training

```
79     protected AttributeClassObserver newNominalClassObserver() {
80         return new NominalAttributeClassObserver();
81     }
82
83     protected AttributeClassObserver newNumericClassObserver() {
84         return new GaussianNumericAttributeClassObserver();
85     }
86
87     protected AttributeSplitSuggestion findBestSplit(SplitCriterion criterion) {
88         AttributeSplitSuggestion bestFound = null;
89         double bestMerit = Double.NEGATIVE_INFINITY;
90         double[] preSplitDist = this.observedClassDistribution.getArrayCopy();
91         for (int i = 0; i < this.attributeObservers.size(); i++) {
92             AttributeClassObserver obs = this.attributeObservers.get(i);
93             if (obs != null) {
94                 AttributeSplitSuggestion suggestion =
95                     obs.getBestEvaluatedSplitSuggestion(
96                         criterion,
97                         preSplitDist,
98                         i,
99                         this.binarySplitsOption.isSet());
100                if (suggestion.merit > bestMerit) {
101                    bestMerit = suggestion.merit;
102                    bestFound = suggestion;
103                }
104            }
105        }
106        return bestFound;
107    }
```

These functions assist the training algorithm. `newNominalClassObserver` and `newNumericClassObserver` are responsible for creating new observer objects for nominal and numeric attributes, respectively. The `findBestSplit()` function will iterate through the possible stumps and return the one with the highest 'merit' score.

Listing 8.6: Predicting class of unknown examples

```
68     public double[] getVotesForInstance(Instance inst) {
69         if (this.bestSplit != null) {
70             int branch = this.bestSplit.splitTest.branchForInstance(inst);
71             if (branch >= 0) {
72                 return this.bestSplit
73                         .resultingClassDistributionFromSplit(branch);
74             }
75         }
76         return this.observedClassDistribution.getArrayCopy();
77     }
```

This is the other important function of the classifier besides training—using the model that has been induced to predict the class of examples. For the decision stump, this involves calling the functions `branchForInstance()` and `resultingClassDistributionFromSplit()` that are implemented by the `AttributeSplitSuggestion` class.

Putting all of the elements together, the full listing of the tutorial class is given below.

Listing 8.7: Full listing

```
1  package moa.classifiers;
2
3  import moa.core.AutoExpandVector;
```

```
4   import moa.core.DoubleVector;
5   import moa.options.ClassOption;
6   import moa.options.FlagOption;
7   import moa.options.IntOption;
8   import weka.core.Instance;
9
10  public class DecisionStumpTutorial extends AbstractClassifier {
11
12          private static final long serialVersionUID = 1L;
13
14          public IntOption gracePeriodOption = new IntOption("gracePeriod", 'g',
15                          "The number of instances to observe between model changes.",
16                          1000, 0, Integer.MAX_VALUE);
17
18          public FlagOption binarySplitsOption = new FlagOption("binarySplits", 'b',
19                          "Only allow binary splits.");
20
21          public ClassOption splitCriterionOption = new ClassOption("splitCriterion",
22                          'c', "Split criterion to use.", SplitCriterion.class,
23                          "InfoGainSplitCriterion");
24
25          protected AttributeSplitSuggestion bestSplit;
26
27          protected DoubleVector observedClassDistribution;
28
29          protected AutoExpandVector<AttributeClassObserver> attributeObservers;
30
31          protected double weightSeenAtLastSplit;
32
33          public boolean isRandomizable() {
34                  return false;
35          }
36
37          @Override
38          public void resetLearningImpl() {
39                  this.bestSplit = null;
40                  this.observedClassDistribution = new DoubleVector();
41                  this.attributeObservers = new AutoExpandVector<AttributeClassObserver>();
42                  this.weightSeenAtLastSplit = 0.0;
43          }
44
45          @Override
46          public void trainOnInstanceImpl(Instance inst) {
47                  this.observedClassDistribution.addToValue((int) inst.classValue(), inst
48                                  .weight());
49                  for (int i = 0; i < inst.numAttributes() - 1; i++) {
50                          int instAttIndex = modelAttIndexToInstanceAttIndex(i, inst);
51                          AttributeClassObserver obs = this.attributeObservers.get(i);
52                          if (obs == null) {
53                                  obs = inst.attribute(instAttIndex).isNominal() ?
54                                          newNominalClassObserver() : newNumericClassObserver();
55                                  this.attributeObservers.set(i, obs);
56                          }
57                          obs.observeAttributeClass(inst.value(instAttIndex), (int) inst
58                                          .classValue(), inst.weight());
59                  }
60                  if (this.trainingWeightSeenByModel - this.weightSeenAtLastSplit >=
61                                  this.gracePeriodOption.getValue()) {
62                          this.bestSplit = findBestSplit((SplitCriterion)
63                                  getPreparedClassOption(this.splitCriterionOption));
64                          this.weightSeenAtLastSplit = this.trainingWeightSeenByModel;
65                  }
66          }
67
68          public double[] getVotesForInstance(Instance inst) {
69                  if (this.bestSplit != null) {
70                          int branch = this.bestSplit.splitTest.branchForInstance(inst);
71                          if (branch >= 0) {
72                                  return this.bestSplit
73                                                  .resultingClassDistributionFromSplit(branch);
74                          }
75                  }
76                  return this.observedClassDistribution.getArrayCopy();
77          }
78
79          protected AttributeClassObserver newNominalClassObserver() {
80                  return new NominalAttributeClassObserver();
81          }
82
83          protected AttributeClassObserver newNumericClassObserver() {
84                  return new GaussianNumericAttributeClassObserver();
85          }
86
```

```
87         protected AttributeSplitSuggestion findBestSplit(SplitCriterion criterion) {
88                 AttributeSplitSuggestion bestFound = null;
89                 double bestMerit = Double.NEGATIVE_INFINITY;
90                 double[] preSplitDist = this.observedClassDistribution.getArrayCopy();
91                 for (int i = 0; i < this.attributeObservers.size(); i++) {
92                         AttributeClassObserver obs = this.attributeObservers.get(i);
93                         if (obs != null) {
94                                 AttributeSplitSuggestion suggestion =
95                                         obs.getBestEvaluatedSplitSuggestion(
96                                                 criterion,
97                                                 preSplitDist,
98                                                 i,
99                                                 this.binarySplitsOption.isSet());
100                                if (suggestion.merit > bestMerit) {
101                                        bestMerit = suggestion.merit;
102                                        bestFound = suggestion;
103                                }
104                        }
105                }
106                return bestFound;
107        }
108
109        public void getModelDescription(StringBuilder out, int indent) {
110        }
111
112        protected moa.core.Measurement[] getModelMeasurementsImpl() {
113                return null;
114        }
115
116 }
```

8.2 Compiling a classifier

The following five files are assumed to be in the current working directory:

```
DecisionStumpTutorial.java
moa.jar
weka.jar
sizeofag.jar
```

The example source code can be compiled with the following command:

```
javac -cp moa.jar:weka.jar DecisionStumpTutorial.java
```

This produces compiled java class file `DecisionStumpTutorial.class`.

Before continuing, the commands below set up directory structure to reflect the package structure:

```
mkdir moa
mkdir moa/classifiers
cp DecisionStumpTutorial.class moa/classifiers/
```

The class is now ready to use.

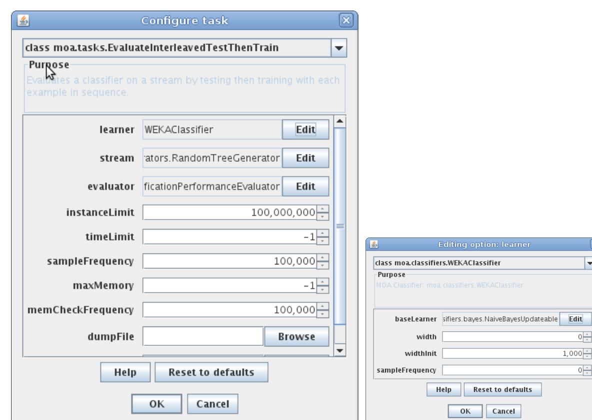
9

Bi-directional interface with WEKA

Now, it is easy to use MOA classifiers and streams from WEKA, and WEKA classifiers from MOA. The main difference between using incremental classifiers in WEKA and in MOA will be the evaluation method used.

9.1 WEKA classifiers from MOA

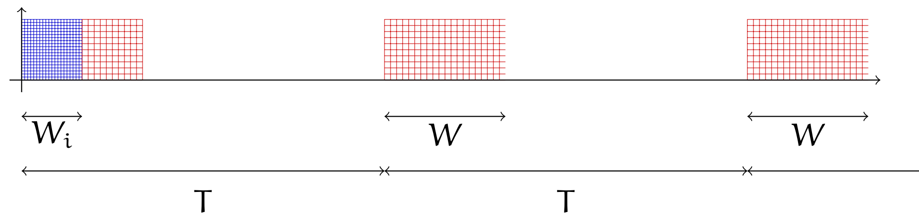
Weka classifiers may be incremental or non incremental methods.



To use the Weka classifiers from MOA it is necessary to use one of the following classes:

9.1.1 WekaClassifier

A classifier to use classifiers from WEKA.



WEKAClassifier builds a model of W instances every T instances only for non incremental methods. For WEKA incremental methods the WEKA classifier is trained for every instance.

Example:

```
WekaClassifier -l weka.classifiers.trees.J48
         -w 10000 -i 1000 -f 100000
```

Parameters:

- -l : Classifier to train
- -w : Size of Window for training learner
- -i : Size of first Window for training learner
- -f : How many instances between samples of the learning performance

9.1.2 SingleClassifierDrift

Class for handling concept drift datasets with a wrapper on a classifier.

The drift detection method (DDM) proposed by Gama et al. controls the number of errors produced by the learning model during prediction. It compares the statistics of two windows: the first one contains all the data, and the second one contains only the data from the beginning until the number of errors increases. Their method doesn't store these windows in memory. It keeps only statistics and a window of recent errors.

They consider that the number of errors in a sample of examples is modeled by a binomial distribution. A significant increase in the error of the algorithm, suggests that the class distribution is changing and, hence, the actual decision model is supposed to be inappropriate. They check for a warning level and a drift level. Beyond these levels, change of context is considered.

The number of errors in a sample of n examples is modeled by a binomial distribution. For each point i in the sequence that is being sampled, the error rate is the probability of misclassifying (p_i), with standard deviation given by $s_i = \sqrt{p_i(1 - p_i)/i}$. A significant increase in the error of the algorithm, suggests that the class distribution is changing and, hence, the actual decision model is supposed to be inappropriate. Thus, they store the values of p_i and s_i when $p_i + s_i$ reaches its minimum value during the process (obtaining p_{pmin} and s_{min}). And it checks when the following conditions trigger:

- $p_i + s_i \geq p_{min} + 2 \cdot s_{min}$ for the warning level. Beyond this level, the examples are stored in anticipation of a possible change of context.
- $p_i + s_i \geq p_{min} + 3 \cdot s_{min}$ for the drift level. Beyond this level, the model induced by the learning method is reset and a new model is learnt using the examples stored since the warning level triggered.

Baena-García et al. proposed a new method EDDM in order to improve DDM. It is based on the estimated distribution of the distances between classification errors. The window resize procedure is governed by the same heuristics.

See more details in:

[GMCR] J. Gama, P. Medas, G. Castillo, and P. Rodrigues. Learning with drift detection. In *SBIA Brazilian Symposium on Artificial Intelligence*, pages 286–295, 2004.

[BDF] Manuel Baena-García, José del Campo-Ávila, Raúl Fidalgo, Albert Bifet, Ricard Gavaldá, and Rafael Morales-Bueno. Early drift detection method. In *Fourth International Workshop on Knowledge Discovery from Data Streams*, 2006.

Example:

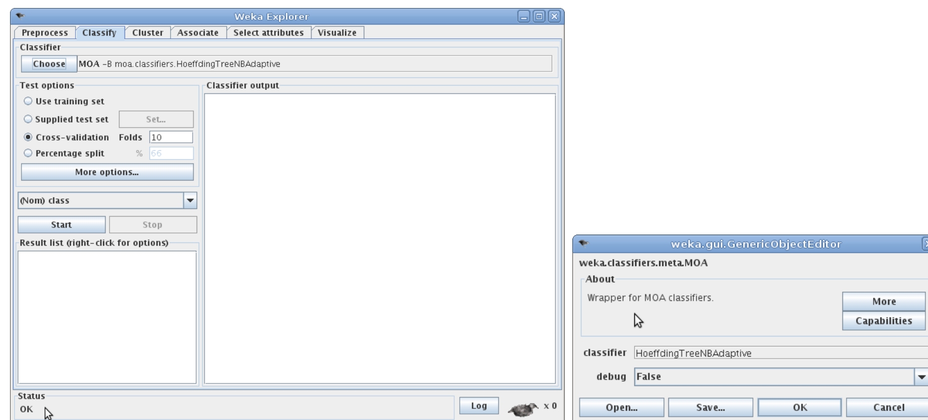
```
SingleClassifierDrift -d EDDM
    -l weka.classifiers.bayes.NaiveBayesUpdateable
```

Parameters:

- -l : Classifier to train
- -d : Drift detection method to use: DDM or EDDM

9.2 MOA classifiers from WEKA

You can use MOA classifiers quite easily as incremental classifiers within the WEKA Explorer, Knowledge Flow interface or command-line interface, using the `weka.classifiers.meta.MOA` meta-classifier. This meta-classifier is just a wrapper for MOA classifiers, translating the WEKA method calls into MOA ones.



You can use MOA streams within the WEKA framework using the `weka.datagenerators.classifiers.classification.MOA` data generator. For example:

```
weka.datagenerators.classifiers.classification.MOA
          -B moa.streams.generators.LEDGenerator
```

In order to manipulate the MOA classifiers in the GUI, like Explorer or Experimenter, you need to register a new editor for the GenericObjectEditor.

The following steps are necessary to integrate the MOA classifiers:

1. Determine the location of your home directory:
   - Windows in a command prompt, run the following command: echo %USERPROFILE%
   - Linux/Unix in a terminal (bash), run the following command: echo $HOME
2. Copy the ”GUIEdtitors.props.addon” file, contained in the MOA project in the source directory ”src/main/java/weka/gui” or in the moa.jar in ”weka/gui”, into your home directory. Rename the extension from ”.props.addon” to ”.props”. If the home directory already contains such a file, then just append the content of the version in MOA file to the already existing one.

3. Restart WEKA from the MOA project, e.g., using the ”run-explorer” target of the ANT build file.