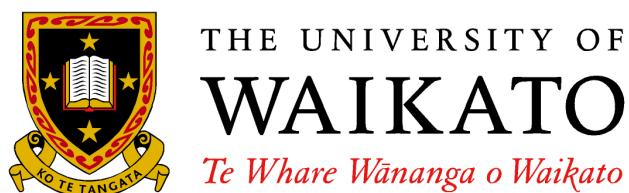# Tutorial 2. Introduction to the API of MOA
{M}assive {O}nline {A}nalysis

Albert Bifet and Richard Kirkby

March 2012

1 Using the API

It's easy to use the methods of MOA inside Java Code. For example, this is the Java code for a prequential evaluation:

Listing 1: Java Code Example

```
1   int numInstances=10000;
2
3   Classifier learner = new HoeffdingTree();
4   RandomRBFGenerator stream = new RandomRBFGenerator();
5   stream.prepareForUse();
6
7   learner.setModelContext(stream.getHeader());
8   learner.prepareForUse();
9
10  int numberSamplesCorrect=0;
11  int numberSamples=0;
12  boolean isTesting = true;
13  while(stream.hasMoreInstances() && numberSamples < numInstances){
14      Instance trainInst = stream.nextInstance();
15      if(isTesting){
16          if(learner.correctlyClassifies(trainInst)){
17              numberSamplesCorrect++;
18          }
19      }
20      numberSamples++;
21      learner.trainOnInstance(trainInst);
22  }
23  double accuracy = 100.0*(double) numberSamplesCorrect / (double)numberSamples;
24  System.out.println(numberSamples+" instances processed with "+accuracy+"% accuracy");
```

1.1 Exercises

Exercise 1 *Write a program in Java that learns a model using one stream and tests the model using another stream.*

2 Writing 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) 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.

<div align="center">Listing 2: Option handling</div>

```
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 3: 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 4: 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 5: 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 example. Lines 60-61 check to see if the grace period has expired. If so, the best split is re-evaluated.

Listing 6: 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 7: 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: 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++) {
```

5

```
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 }
```

2.1 Exercises

Exercise 2 *Write a classifier in Java that implements a* K *nearest-neighbor algorithm.*