15 Developing New Methods in MOA

In some cases, it is useful to develop new methods that extend MOA capabilities. In this chapter, we present the structure of some of the most popular objects in MOA, and show how to implement new ones. This chapter requires a good understanding of data structures in Java.

15.1 Main Classes in MOA

All objects in MOA should implement the `MOAObject` interface and extend the `AbstractMOAObject` abstract class. To create a new class in Java that needs to configure options, one needs to extend the `AbstractOptionHandler` abstract class. MOA options are very flexible and, since they can be recursive, it is possible to define, as options, objects that have options themselves. For example, this is the specification of a Hoeffding Tree classifier with options different from the default ones:

```
moa.classifiers.trees.HoeffdingTree
    -n (VFMLNumericAttributeClassObserver -n 20)
     -g 300 -s GiniSplitCriterion -t 0.04 -l MC
```

In this example, we are selecting as a numeric attribute class observer a `VFMLNumericAttributeClassObserver` object that has 20 bins. We pass the number 20 as an option to this class. Notice that the options are recursive, and can contain a large number of Java classes.

The main learner interfaces in MOA are:

- **Classifier:** This interface is used by classifiers and regressors. Classifiers usually extend the `AbstractClassifier` abstract class, and implement the following three methods:
  - `resetLearningImpl` to reset or initialize the classifier
  - `trainOnInstanceImpl` to update the learner with a new instance
  - `getVotesForInstance` to obtain a prediction from the learner

  The `AbstractClassifier` abstract class contains the following two methods, used in the API:
  - `prepareForUse`, which calls `resetLearningImpl` to prepare the classifier

  - `trainOnInstance`, which uses `trainOnInstanceImpl` to train the learner with a new instance

- **Regressor:** Regression methods implement this interface and extend the `AbstractClassifier` abstract class. The difference between classification and regression is that, for classification, `getVotesForInstance` returns an array with estimated prediction values for each class, while, for regression, `getVotesForInstance` returns an array of length 1 with the predicted outcome.
- **Clusterer:** This interface is used by clusterers and outlier detectors. Clusterers extend the `AbstractClusterer` abstract class, and implement the following methods:
  - `resetLearningImpl` to reset or initialize the clusterer
  - `trainOnInstanceImpl` to update the clusterer with a new instance
  - `getClustering` to obtain the clustering points computed by the clusterer
- **OutlierDetector:** Similar to clusterers, but specifically designed for outlier detection.

15.2 Creating a New Classifier

To demonstrate the implementation and use of learning algorithms in the system, we show and explain the Java code of a simple decision stump classifier. The classifier monitors the information gain on the class when splitting on each attribute and chooses the attribute that maximizes this value. 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, if there is no change in the stream distribution.

To describe the implementation, relevant code fragments are discussed in turn, with the entire code listed at the end (listing 15.7). The line numbers from the fragments refer to the final listing.

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

Listing 15.1: Option handling

```
public IntOption gracePeriodOption = new
    IntOption("gracePeriod", 'g',
              "The number of instances to observe between
                  model changes.",
              1000, 0, Integer.MAX_VALUE);

public FlagOption binarySplitsOption = new
    FlagOption("binarySplits", 'b',
              "Only allow binary splits.");

public ClassOption splitCriterionOption = new
    ClassOption("splitCriterion",
              'c', "Split criterion to use.",
                  SplitCriterion.class,
              "InfoGainSplitCriterion");
```

To set up the public interface to the classifier, you must specify the options available to the user. For the system to automatically take care of option handling, the options need to be public members of the class and extend the `moa.options.Option` type, as seen in listing 15.1.

The example classifier–the decision stump–has three options, each of a different type. The meanings of the first three parameters used to build 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 you tp specify additional information, such as default values, or valid ranges for values.

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 choice of the currently best stump. The default value of 1,000 means that the choice of stump will be re-evaluated only every 1,000 examples. 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 the user requests so. This flag controls whether the decision stumps are only allowed to split two ways. By default the stumps are allowed to 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 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 15.2: Miscellaneous fields.

```
protected AttributeSplitSuggestion bestSplit;

protected DoubleVector observedClassDistribution;

protected AutoExpandVector<AttributeClassObserver>
    attributeObservers;

protected double weightSeenAtLastSplit;

public boolean isRandomizable() {
        return false;
}
```

In listing 15.2 four global variables are used to maintain the state of the classifier:

- The `bestSplit` field maintains the stump that is currently 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`, 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 the classifier can determine 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 does not, so `false` is returned.

Listing 15.3: Preparing for learning.

```
@Override
public void resetLearningImpl() {
    this.bestSplit = null;
    this.observedClassDistribution = new DoubleVector();
    this.attributeObservers = new
        AutoExpandVector<AttributeClassObserver>();
    this.weightSeenAtLastSplit = 0.0;
}
```

The `resetLearningImpl` function in listing 15.3 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 15.4: Training on examples.

```
@Override
public void trainOnInstanceImpl(Instance inst) {
    this.observedClassDistribution.addToValue((int)
        inst.classValue(), inst
                .weight());
    for (int i = 0; i < inst.numAttributes() - 1; i++) {
            int instAttIndex =
                modelAttIndexToInstanceAttIndex(i, inst);
            AttributeClassObserver obs =
                this.attributeObservers.get(i);
            if (obs == null) {
                    obs =
                        inst.attribute(instAttIndex).isNominal()
                        ?
                            newNominalClassObserver() :
                                newNumericClassObserver();
                    this.attributeObservers.set(i, obs);
            }
```

```
13                  obs.observeAttributeClass(inst.value(instAttIndex),
                        (int) inst
                                    .classValue(), inst.weight());
15          }
            if (this.trainingWeightSeenByModel -
                this.weightSeenAtLastSplit >=
17                          this.gracePeriodOption.getValue()) {
                    this.bestSplit =
                        findBestSplit((SplitCriterion)
19                  getPreparedClassOption(this.splitCriterionOption));
                    this.weightSeenAtLastSplit =
                        this.trainingWeightSeenByModel;
21          }
    }
```

Function `trainOnInstanceImpl` in listing 15.4 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 on lines 49–59 repeats for every attribute in the data. If no observations for a particular attribute have been seen before, 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 whether the grace period has expired. If so, the best split is reevaluated.

Listing 15.5: Functions used during training.

```
    protected AttributeClassObserver newNominalClassObserver() {
2           return new NominalAttributeClassObserver();
    }
4
    protected AttributeClassObserver newNumericClassObserver() {
6           return new GaussianNumericAttributeClassObserver();
    }
8
    protected AttributeSplitSuggestion
        findBestSplit(SplitCriterion criterion) {
10          AttributeSplitSuggestion bestFound = null;
            double bestMerit = Double.NEGATIVE_INFINITY;
12          double[] preSplitDist =
                this.observedClassDistribution.getArrayCopy();
            for (int i = 0; i < this.attributeObservers.size();
                i++) {
14                  AttributeClassObserver obs =
                        this.attributeObservers.get(i);
                    if (obs != null) {
```

```
16                          AttributeSplitSuggestion suggestion
                                =
18                             obs.getBestEvaluatedSplitSuggestion(
                                    criterion,
20                                  preSplitDist,
                                    i,
22                                  this.binarySplitsOption.isSet());
                            if (suggestion.merit > bestMerit) {
                                    bestMerit =
24                                      suggestion.merit;
                                    bestFound = suggestion;
                            }
26                  }
            }
28          return bestFound;
}
```

Functions in listing 15.5 assist the training algorithm. Classes `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 15.6: Predicting the class of unknown examples.

```
1  public double[] getVotesForInstance(Instance inst) {
       if (this.bestSplit != null) {
3              int branch =
                   this.bestSplit.splitTest.branchForInstance(inst);
               if (branch >= 0) {
5                 return this.bestSplit
                    .resultingClassDistributionFromSplit(branch);
7              }
       }
9      return this.observedClassDistribution.getArrayCopy();
}
```

Function `getVotesForInstance` in listing 15.6 is the other important function of the classifier besides training—using the model that has been induced to predict the class of new examples. For the decision stump, this involves calling the functions `branchForInstance()` and `resultingClassDistributionFromSplit()`, which are implemented by the `AttributeSplitSuggestion` class.

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

Listing 15.7: Full listing.

```
package moa.classifiers;

import moa.core.AutoExpandVector;
import moa.core.DoubleVector;
import moa.options.ClassOption;
import moa.options.FlagOption;
import moa.options.IntOption;
import weka.core.Instance;

public class DecisionStumpTutorial extends
    AbstractClassifier {

  private static final long serialVersionUID = 1L;

  public IntOption gracePeriodOption = new
      IntOption("gracePeriod", 'g',
        "The number of instances to observe between model
            changes.",
        1000, 0, Integer.MAX_VALUE);

  public FlagOption binarySplitsOption = new
      FlagOption("binarySplits", 'b',
        "Only allow binary splits.");

  public ClassOption splitCriterionOption = new
      ClassOption("splitCriterion",
        'c', "Split criterion to use.",
            SplitCriterion.class,
        "InfoGainSplitCriterion");

  protected AttributeSplitSuggestion bestSplit;

  protected DoubleVector observedClassDistribution;

  protected AutoExpandVector<AttributeClassObserver>
      attributeObservers;

  protected double weightSeenAtLastSplit;

  public boolean isRandomizable() {
    return false;
  }

```

```
    @Override
38  public void resetLearningImpl() {
      this.bestSplit = null;
40    this.observedClassDistribution = new DoubleVector();
      this.attributeObservers = new
          AutoExpandVector<AttributeClassObserver>();
42    this.weightSeenAtLastSplit = 0.0;
    }
44
    @Override
46  public void trainOnInstanceImpl(Instance inst) {
      this.observedClassDistribution.addToValue((int)
          inst.classValue(), inst
48          .weight());
      for (int i = 0; i < inst.numAttributes() - 1; i++) {
50      int instAttIndex =
            modelAttIndexToInstanceAttIndex(i, inst);
        AttributeClassObserver obs =
            this.attributeObservers.get(i);
52      if (obs == null) {
          obs = inst.attribute(instAttIndex).isNominal() ?
54            newNominalClassObserver() :
                newNumericClassObserver();
          this.attributeObservers.set(i, obs);
56      }
        obs.observeAttributeClass(inst.value(instAttIndex),
            (int) inst
58            .classValue(), inst.weight());
      }
60    if (this.trainingWeightSeenByModel -
          this.weightSeenAtLastSplit >=
            this.gracePeriodOption.getValue()) {
62      this.bestSplit = findBestSplit((SplitCriterion)
          getPreparedClassOption(this.splitCriterionOption));
64      this.weightSeenAtLastSplit =
            this.trainingWeightSeenByModel;
      }
66  }

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

```
    }

    protected AttributeClassObserver
        newNominalClassObserver() {
        return new NominalAttributeClassObserver();
    }

    protected AttributeClassObserver
        newNumericClassObserver() {
        return new GaussianNumericAttributeClassObserver();
    }

    protected AttributeSplitSuggestion
        findBestSplit(SplitCriterion criterion) {
        AttributeSplitSuggestion bestFound = null;
        double bestMerit = Double.NEGATIVE_INFINITY;
        double[] preSplitDist =
            this.observedClassDistribution.getArrayCopy();
        for (int i = 0; i < this.attributeObservers.size();
            i++) {
            AttributeClassObserver obs =
                this.attributeObservers.get(i);
            if (obs != null) {
                AttributeSplitSuggestion suggestion =
                    obs.getBestEvaluatedSplitSuggestion(
                        criterion,
                        preSplitDist,
                        i,
                        this.binarySplitsOption.isSet());
                if (suggestion.merit > bestMerit) {
                    bestMerit = suggestion.merit;
                    bestFound = suggestion;
                }
            }
        }
        return bestFound;
    }

    public void getModelDescription(StringBuilder out, int
        indent) {
    }

    protected moa.core.Measurement[]
        getModelMeasurementsImpl() {
        return null;
    }

}
```

15.3 Compiling a Classifier

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

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

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

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

This command produces a compiled Java class file named `DecisionStumpTutorial.class`.

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

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

The class is now ready to use.

15.4 Good Programming Practices in MOA

We recommend the following good practices in Java programming when developing methods in MOA:

- When building new learners or tasks, to add new strategies or behavior, add them using class options. Follow these steps:
  1. Create a new interface for the strategy.
  2. Create different strategy classes that implement that new interface.
  3. Add a class option, so that users can choose what strategy to use from the GUI or command line.
- Minimize the scope of variables.
- Favor composition over inheritance.
- Favor static member classes over nonstatic ones.

- Minimize the accessibility of classes and members.
- Refer to objects by their interfaces.
- Use builders instead of constructors with many parameters.

The book *Effective Java* by Bloch [45] is a good reference for mastering the best practices in Java programming.