Implementing classifier in TSML

October 31, 2019

1 General

There are three 'levels' you could inherit at. Remember that in Java, we can have only single inheritance, but multiple interface implementations:

1. The classifier interface (WEKA). Declares the basic functionality to train and test classifiers.
2. The AbstractClassifier base class (WEKA). Implements Classifier, among other generally useful interfaces.
3. The EnhancedAbstractClassfier base class (TSML). Extends EnhancAbstractClassifier, and implement a couple more generally useful interfaces, some WEKA and some our own.

All WEKA classifiers extend AbstractClassifier. All TSML classifiers should AT LEAST extend this, but preferably extend EnhancedAbstractClassifier. Based on our past (and present to some extent) development practices which were far more personal research than public usability focused, our systems do not use a lot of the very large inheritance hierarchy provided by WEKA. Wanting to make use of this is a fair reason to not extend from EnhancedAbstractClassifier. In a similar vein, our expectations are often relaxed compared to other extremely software-engineered packages. The high-level ideas of writing WEKA classifiers [1] will generally apply to TSML as well, but most importantly our primary interests at all times are:

BASIC USABILITY The classifier should of course provide an implementation for `buildClassifier(Instances)` and at least one of `classifyInstance(Instance)` or `distributionForInstance(Instance)`. The latter two are default-implemented in terms of one another, and so only one is needed to be re-implemented. In general you should always implement `distributionForInstance(Instance)`, and then the individual classification is calculated simply is the index of the max value in the distribution. All classifiers should have a default constructor that takes no arguments, and as such be usable (perhaps sub-optimally, but still) out-of-the-box.

CORRECTNESS The classifier should be able to replicate the results (read: predictions on the same data) of the original implementation and/or results reported in the accompanying paper. There will very often be (within reason) unavoidable reasons for minor differences. Differences in random seeding functions between languages, differences in native precision between non C-like languages, etc. None of these should produce significant differences on average over a sizable number of datasets, however. A good measure of correctness would be to replicate the original results to within $p = 0.01$ over 50 or more UCR datasets, for example. Highly stochastic methods may need a wider margin.

REPRODUCIBILITY Correctness covered similarity in results with the original code/implementation. Reproducibility in this context means being able to **exactly** recreate results (again: predictions and probability distributions) of previous classifier evaluations, given the same train data, test data and seeds. EnhancedAbstractClassifier implements Randomizable by default (but does not *turn on* seeding by default), even if the extending classifier is not innately stochastic.

[1] https://waikato.github.io/weka-wiki/writing_classifier/

2 EnhancedAbstractClassifier

Most importantly for extending classes, EnhancedAbstractClassifier stores information about the training process in a ClassifierResults instance. All EnhancedAbstractClassifiers should store timing information internally, the time to build at minimum (potentially minus any computation done that was not needed strictly to build a model ready for testing, e.g. logging). See code listing 1 for a toy classifier implementation that does not estimate its own accuracy.

```
import timeseriesweka.classifiers.EnhancedAbstractClassifier;
import weka.core.Instance;
import weka.core.Instances;
import java.util.concurrent.TimeUnit;

public class BasicClassifierExample extends EnhancedAbstractClassifier {

    // hyper-parameters
    private int para1;

    // internal attributes
    private int numClasses;

    public BasicClassifierExample() {
        // all classifiers should have a default constructor with no args
        // and be usable out-the-box
        super(CANNOT_ESTIMATE_OWN_PERFORMANCE);
        para1 = 0;
    }

    public BasicClassifierExample(int somePara) {
        super(CANNOT_ESTIMATE_OWN_PERFORMANCE);
        this.para1 = somePara;
    }

    public int getPara1() {
        return para1;
    }

    public void setPara1(int para1) {
        this.para1 = para1;
    }

    @Override
    public void buildClassifier(Instances trainInsts) throws Exception {
        // test whether the classifier can handle the data
        // EnhancedAbstractClassifier provides a bare-bones implementation of
        // getCapabilities(), which you can build upon if your classifer has
        // extra restrictions/capabilities in terms of the data it can handle beyond
        // basic univariate time series data
        getCapabilities().testWithFail(trainInsts);

        long buildTime = System.nanoTime();

        // learn from trainInsts
        numClasses = trainInsts.numClasses();

        // at minimum, set the build time (and time unit if not millis, nanoseconds
        // generally preferred for precision) this may exclude time spent for
        // 'meta' or 'experimental' computation, e.g. the capabilities check at
        // the start of this function
        buildTime = System.nanoTime() - buildTime;
        this.trainResults.setBuildTime(buildTime);
        this.trainResults.setTimeUnit(TimeUnit.NANOSECONDS);
    }

    @Override
    public double[] distributionForInstance(Instance testInst) throws Exception {
```

```
        double[] dist = new double[numClasses];

        // form a probability distribution over the class space
        dist[para1] = 1.0;

        return dist;
    }
}
```

Listing 1: Toy non-self estimating classifier example

For e.g. weighted ensembles that need to estimate the performance of their base classifiers, or when we want to test generalisability and bias, we need to produce estimates of the classifier's performance from the train data. The standard non-biased method for this we would use is a nested cross validation outside of buildClassifier()

EnhancedAbstractClassifier has a flag that needs to be set in the constructor of extending classes. This defines whether the classifier is capable of estimating its own performance more efficiently (but potentially with some acceptable bias) than some nested process such as cross validation. A random forest computing out of bag error would be an example of this. The out of bag error can be computed naturally as part of the build process. For classifiers such as this, the training data stored in EnhancedClassifierResults can be a full set of predictions on the train data.

In this case, classifiers should flag that they CAN_ESTIMATE_OWN_PERFORMANCE in their constructors.

These classifiers should check whether they have been told to produce an estimates, and populate their trainResults object by the end of buildClassifier(). This results object should also be finalised via having finaliseResults() called upon it.

```
import evaluation.evaluators.SingleSampleEvaluator;
import evaluation.storage.ClassifierResults;
import timeseriesweka.classifiers.EnhancedAbstractClassifier;
import weka.core.Instance;
import weka.core.Instances;
import java.util.concurrent.TimeUnit;

public class SelfEstimatingClassifierExample extends EnhancedAbstractClassifier {

    // internal attributes
    private int numClasses;

    public SelfEstimatingClassifierExample() {
        // all classifiers should have a default constructor with no args and be
        // usable out-the-box
        super(CAN_ESTIMATE_OWN_PERFORMANCE);
    }

    @Override
    public void buildClassifier(Instances trainInsts) throws Exception {
        getCapabilities().testWithFail(trainInsts);

        long buildTime = System.nanoTime();

        // learn from trainInsts
        numClasses = trainInsts.numClasses();

        buildTime = System.nanoTime() - buildTime;
        this.trainResults.setBuildTime(buildTime);
        this.trainResults.setTimeUnit(TimeUnit.NANOSECONDS);

        if (getEstimateOwnPerformance()) {
            // If we've been told to estimate our own performance, find it
            // In this example, we'll pretend by splitting the train data into train'
            // and validation data to evaluate

            long estimateTime = System.nanoTime();
```

```
38
39              SingleSampleEvaluator eval = new SingleSampleEvaluator();
40              if (this.seedClassifier)
41                  eval.setSeed(this.seed);
42              this.setEstimateOwnPerformance(false);
43              ClassifierResults res = eval.evaluate(this, trainInsts);
44              this.setEstimateOwnPerformance(true);
45
46              estimateTime = System.nanoTime() - estimateTime;
47
48              //the evaluator will be using nanoseconds by default
49              res.setBuildTime(buildTime);
50              res.setErrorEstimateTime(estimateTime);
51
52              this.trainResults = res;
53
54              // In this case the evaluator will have done it for us, but in general
55              // always finalise the results of your estimate
56              this.trainResults.finaliseResults();
57          }
58      }
59
60      @Override
61      public double[] distributionForInstance(Instance testInst) throws Exception {
62          double[] dist = new double[numClasses];
63
64          // form a probability distribution over the class space
65          dist[0] = 1.0;
66
67          return dist;
68      }
69  }
```

Listing 2: Toy self-estimating classifier example