JAICore Experiment Package Documentation

Felix Mohr

May 2018

1 Motivation

The JAICore experiment package aims at supporting researchers to systematically conduct parallelized experiments with Java code.

Formally, the experiment package offers a small framework to reasonably compute, in a point-wise manner, a function

$$f : \prod_{i=1}^{n} I_i \rightarrow \prod_{i=1}^{m} D_i$$

where $I_1, .., I_n$ are finite sets of values for independent experiment variables, and $D_1, .., D_m$ are (possibly infinite) domains for dependent variables we want to observe. In other words, the function f is to be evaluated over the whole grid, which is why the domains of independent variables need to be finite; if they are actually infinite, some discretization must be used instead. For each point in $I_1 \times .. \times I_n$, a result vector $(d_1, .., d_m) \in D_1 \times .. \times D_m$ is computed.

Important features of the experiment package are:

- $I_1, .., I_n$ and $D_1, .., D_m$ can be easily configured in a configuration file. The configuration can be even changed during runtime, which is important if one does not want to interrupt a cluster-based computation of f
- the explicit computation of f is stored in an (My)SQL database, which natively allows to conduct many useful operations on it
- Partial evaluations possible. It may be that, for whatever reason, a dependent variable is not defined or cannot be computed at a specific point. The problem is that the reason for the problem also causes a crash on the entire computation of the point's dependent variables, which would then yield an empty result vector. To avoid this, the experiment package allows to deliver results whenever they become available, i.e., one does not deliver the entire result vector $(d_1, .., d_m)$ as a whole but any subset of it at different points in time. If the computation crashes, at least the results delivered until then have been saved.
- updatable results. It may be that one wants to update a particular value after some more information has become available.

- bookkeeping of time and memory required to compute each dependent variable.
- native support for massive parallelization on independent machines
- a database handle that allows to store additional, problem-specific information, is available

2 Usage

To use the JAICore experiment framework, one has to do two things. First, one needs to develop a configuration for the experiment set (the domain and the range of f). Second, one needs to write the Java code that evaluates a single point of the experiment set, i.e., that evaluates f in a single point.

Creating the configuration consists of two steps in turn. The experiment package assumes that experiment sets are configured using the owner framework. Therefore, one needs to create an interface extending the `IExperimentSetConfig` interface. The interface may remain empty, because all configuration variables necessary to conduct the experiments are already defined in `IExperimentSetConfig`. In this interface, however, it is also possible to define other configuration variables that are specific to the experiment set. In any case, one needs to define the concrete file where the configuration is stored using the owner annotation `@ConfigurationFile`. An example of a configuration file is the following:

```
mem.max = 4098
cpu.max = 4

db.host = yourserver.com
db.username = example
db.password = example
db.database = dbname
db.table = myexperiments

datasets = abalone, car, amazon
classifiers = J48, SMO, RandomForest
seeds = 1,2,3,4,5,6,7,8,9,10

keyfields = seeds, datasets,classifiers
resultfields = loss

datasetfolder = ./datasets
```

The fields `keyfields` and `resultfields` define the *names* of the independent variables $I_1, .., I_n$ and the dependent variables $D_1, .., D_m$ respectively. The possible *values* of independent variables are defined separately as lists for each of them; there is one property for each of them. In the database, they will be stored without the last character, because this is, by convention, assumed

to be a pluralizing s. The values for dependent variables do not—and usually *cannot*— be specified explicitly.

The respective interface file could look as follows:

```
@Sources({ "file:./conf/mlexperiments.properties" })
public interface IMyExperimentConfig extends IExperimentSetConfig {
    public static final String DATASETS = "datasets";
    public static final String SEEDS = "seeds";
    public static final String TIMEOUTS_IN_SECONDS = "timeouts";
    public static final String datasetFolder = "datasetfolder";

    @Key(DATASETS)
    public List<String> getDatasets();

    @Key(SEEDS)
    public List<String> getSeeds();

    @Key(TIMEOUTS_IN_SECONDS)
    public List<String> getTimeouts();

    @Key(datasetFolder)
    public String getDatasetFolder();
}
```

Writing the point-wise evaluator means to design a class with a main method that uses the `ExperimentRunner` class. The `ExperimentRunner` class is the one that organizes the evaluation of f and hence, is the core of the framework. The usual usage is to create an object of the `ExperimentRunner` class and run the `randomlyConductExperiments` method on it, which will evaluate the entire function unless the process is killed meanwhile. Using the `randomlyConductExperiments` method makes sure that parallelized executions of the program do not focus on the same region of the independent variables.

When creating a `ExperimentRunner` object, one needs to specify, as a lambda function, the program logic that is executed when a point is evaluated. The method receives three inputs: An object describing the experiment, a handle to the database, and a processor for the computed results. That is, the results are not *returned* by this lambda function, but they can, one by one or all results together, be registered using the result processor (see a discussion of why this is so below). The lambda function itself is void, i.e., it does not return anything via the *return* statement.

The result processor has only one relevant method, which is `processResults`. It receives an object of type `Map<String,Object>` where the keys must be among the values specified for `resultfields` in the configuration file and values are the respective values to be stored (will always be evaluated to strings using the `toString` method).

The database handle may or may not be used. This is only a handle for the case that the researcher wants to store additional information (in another

table). There is then a convenient handle to do that without needing to open a new database connection.

A code example for such an experiment (that does not use the database handle) is as follows:

```
public class MyExperiment {
  public static void main(String[] args) {
    IMyExperimentConfig m = ConfigCache.getOrCreate(IMyExperimentConfig.class);
    if (m.getDatasetFolder() == null || (!new File(m.getDatasetFolder()).exists()))
         throw new IllegalArgumentException("config specifies invalid dataset folder " +

    switch (args[0].toLowerCase()) {
      case "getmemorylimit":
        System.out.println(m.getMemoryLimit());
        return;
      case "run":
         ExperimentRunner runner = new ExperimentRunner(m, (ExperimentDBEntry experiment
           Map<String, String> description = experiment.getExperiment().getValuesOfKeyFi
           Classifier c = AbstractClassifier.forName(description.get("classifier"), null
           Instances data = new Instances(new BufferedReader(new FileReader(new File(m.g
           data.setClassIndex(data.numAttributes() - 1);
           int seed = Integer.valueOf(description.get("seed"));

           System.out.println(c.getClass().getName());
           Map<String, Object> results = new HashMap<>();
           MulticlassEvaluator eval = new MulticlassEvaluator(new Random(seed));
           double loss = eval.getErrorRateForRandomSplit(c, data, .7f);

           results.put("loss", loss);
           processor.processResults(results);
         });
         runner.randomlyConductExperiments();
      }
    }
  }
}
```