User Guide for Auto-WEKA 0.5

Chris Thornton & Frank Hutter
Department of Computer Science
University of British Columbia
Vancouver, BC V6T 1Z4, Canada
{cwthornt,hutter}@cs.ubc.ca

November 12, 2013

Contents

1 Introduction

Auto-WEKA is a tool that performs combined algorithm selection and hyperparmeter optimisation over the classification and regression algorithms implements in WEKA. More specifically, given a specific dataset, Auto-WEKA explores hyperparameter settings for many algorithms and recommends to a user which method will likely have good generalization performance, using model based optimisation techniques.

1.1 License

Auto-WEKA is open source software issued under the GNU General Public License. Note that each of the optimisation methods that Auto-WEKA have their own license that govern their use.

1.2 Prerequisites

Auto-WEKA itself requires only Java 6 or newer to run, while the underlying optimisers that Auto-WEKA uses may have other requirements. Auto-WEKA was developed on Unix-compatible operating systems, but also runs on Windows. Auto-WEKA can use any version of WEKA, but it has been targeted against 3.7.9.

Auto-WEKA makes use of a few modifications to the algorithms in WEKA, detailed inside the `autoweka.patch` provided. You can apply the patch to your own WEKA distribution by running `patch -Np1 < autoweka.patch` from the WEKA source directory. The changes in this patch just add a support for algorithms to detect if the thread that they have been running in has been interrupted, and then break out of their training phase at their earliest convenience. We have provided a pre-compiled version of WEKA with the patches applied for you to use if you do not wish to compile your own version.

Note that you can still use an unmodified version of WEKA, just that you will likely get inferior performance; if Auto-WEKA tries to run a method that takes more than your allotted time budget, it will be equivalent to a method that wasn't able to get a single correct result, while the modified method will report a result that may not be trained to optimality.

1.3 Included Versions

The Auto-WEKA distribution comes with a "standalone" and "light" version of Auto-WEKA. The standalone version (found in `autoweka.jar`) contains the patched version of WEKA, along with the dependencies that are needed for the GUI. The light version (found in `autoweka-light.jar`) contains just the Auto-WEKA classes, so you will have to add the WEKA classes (and optionally the classes for the GUI) manually on the class path. Sample scripts that demonstrate how to do this can be found in `scripts/autoweka` depending on your platform.

Auto-WEKA makes use of a number of configuration files provided in the `params` directory, so you should ensure that you should keep this directory in the same place as either `autoweka.jar` or `autoweka-light.jar`.

2 Auto-WEKA Overview

Using Auto-WEKA can be broken down into three main steps. First, you have to build your experiment definition, which tells Auto-WEKA what dataset(s) to run on, as well as what kind of hyperparameter search will be done (either through a model based method, or through something like grid search). Once the definition has been written, the experiment needs to be fully instantiated by having Auto-WEKA detect what kind of classifiers can be used given the definition you wrote. At this stage, Auto-WEKA also resolves all path names to absolute paths, so instantiated experiments may have some difficulty when being moved between different computation environments.

Once an experiment has been produced, it actually has to be executed. Auto-WEKA takes advantage of multiple cores by running the same experiment with different random seeds, the only requirement is that all the experiments have a similar file system (since Auto-WEKA relies on absolute path names). The user has to start a new run of the experiment for each seed/core that they want to take advantage of.

After the experiment has been executed, the analysis phase occurs. When Auto-WEKA uses a model based optimisation method, it produces a trajectory of hyperparameters that were identified by the optimisation method as being the best at a particular point in time. The simplest form of analysis looks at the best hyperparameters that were found across all seeds, and uses the trained model to make predictions on a new dataset. Additional experiments can be performed on these trajectory points, for example to see if all the trajectory points have a similar performance on a new set of data that the optimisation method did not have access to.

The above three steps can all be completed through the use of a GUI (see Section 3), or through the more flexible command line interface (see Sections 4-7).

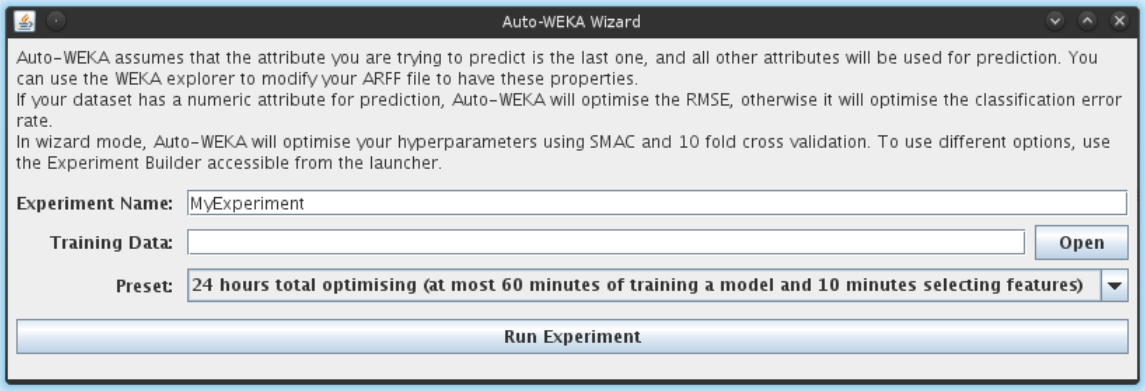
3 Using the GUI

Auto-WEKA contains a GUI to allow for easy execution of experiments. To launch the GUI, you need to execute the class `autoweka.ui.Launcher`. This can be done by running the command `java -jar autoweka.jar`, or using one of the Auto-WEKA scripts. This brings up the following

window that allows you to configure your experiment, run your experiment, and finally get the best set of hyperparameters that were found. Auto-WEKA features a wizard mode that uses presets to help get experiments running as quick as possible. Additionally, the launcher also provides a shortcut to launching the standard WEKA UI.

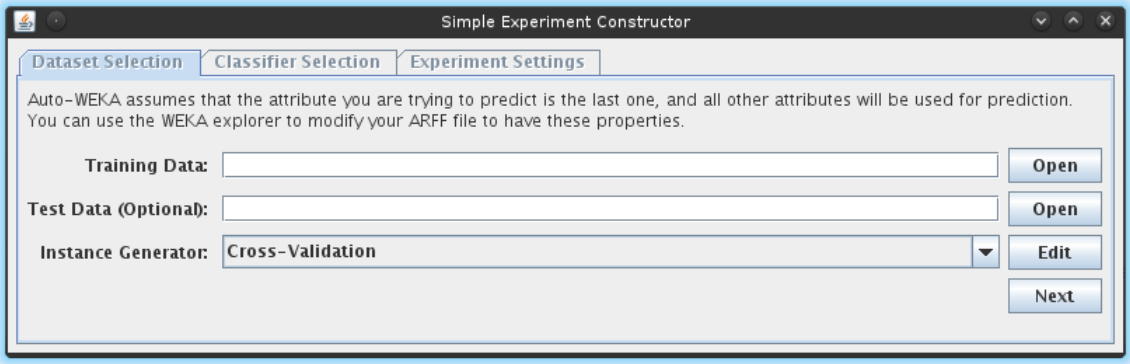
3.1 Wizard

The wizard allows for easy execution of experiments, targeted towards use cases where minimal input is required from the user. After pointing the wizard at the training data (in WEKA's ARFF data format), the user just has to pick the preset that best suits their resource limits. (Note that for larger datasets, you will not achieve good results by using very small time limits). Pressing the "Run Experiment" button will cause Auto-WEKA to use the optimisation method SMAC (See Section 4.3) to find hyperparemter settings with good accuracy (for classification) or RMSE (for regression). Once the optimisation has completed, the wizard will indicate what method and hyperparameters were selected, and allow for predictions on new datasets using the trained method.

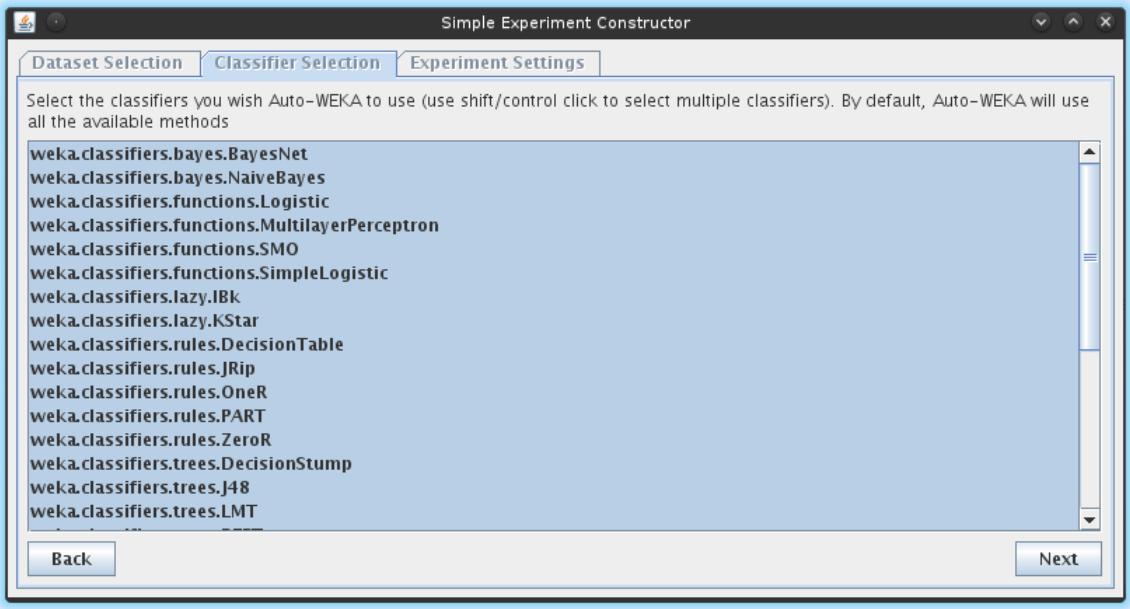


3.2 Experiment Builder

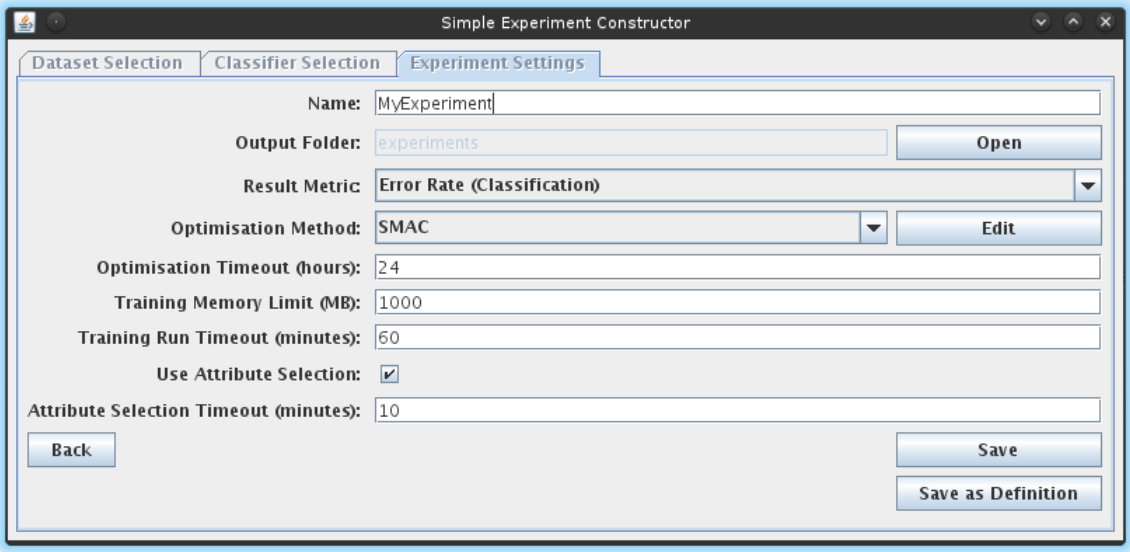
The Experiment Builder has three pages. On the first page, you specify the the location of the training data in the ARFF format. Auto-WEKA GUI assumes that the attribute you are attempting to predict is last attribute in the file, you may need to use the WEKA UI to reorder your dataset. Additionally, you can specify the test data to use (this is only ever used in the analysis stage). If you do not specify any test data, Auto-WEKA just uses the training data instead. You can also set up the way that the optimisation method will use the training data, through the use of an "instance generator". After picking the desired method, you can fine tune the method's settings by clicking on the "Edit" button. A more detailed explanation of each instance generator and its associated options can be found in Section 4.2. Once you have made your choices, press the "Next" button.



Next Auto-WEKA allows you to limit the choice of classifiers that it will consider during for the SMBO method. The second page allows you to select which methods you would like, by control/shift clicking on the different class names to limit your selection. We recommend that you should keep all the methods selected unless you have a very specific reason for not using a particular method. Detailed about which classifiers are not applicable are printed to the system console. Once you have made your selections, press the "Next" button.



The last page allows you to specify your experiment settings. The "output folder" is the folder where the sub-folder containing all of the experiment data will be produced. Change the "Result Metric" to one that is applicable to your dataset. After you select your optimisation method, you can fine tune specific settings by clicking on the edit button. These settings are detailed for each method in Section 4.3. The Optimisation timeout is the number of hours that you are willing to give the optimiser to perform the hyperparameter search. The training memory limit and training run timeout are the resource constraints that are placed on the individual run of a classification or regression method that the optimisation method will perform (and it should be allowed to perform many of them). The attribute selection check box indicates to Auto-WEKA if the hyperparameter search should also be performed over WEKA's methods for transforming/selecting attributes in the dataset. The attribute selection timeout is only used when attribute selection is enabled, and is the maximum number of minutes that will be used to perform the attribute selection for each run of a hyperparameter configuration.



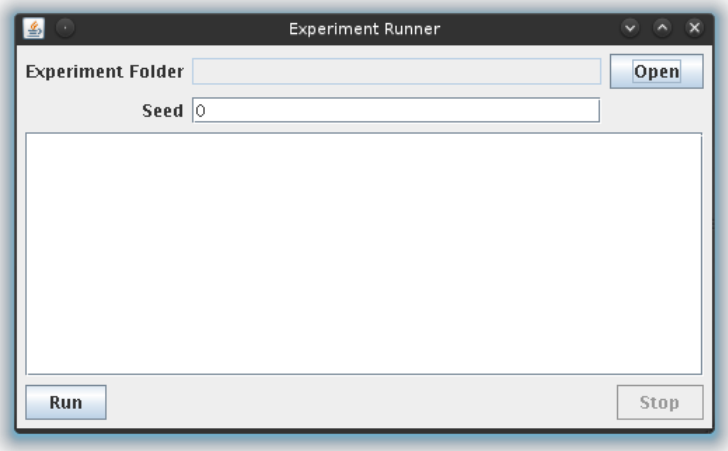
Finally, you can either choose to fully instantiate the experiment by clicking on the "Save" button (which converts all paths to absolute paths) and sets up the folders for the experiment, or you can choose to create a portable experiment definition by clicking the "Save as Definition" button. The experiment definition file can then be moved to a different environment, or be modified to facilitate batch experiments. See Section 4.1 for more details.

3.3 Experiment Runner

The experiment runner provides an easy way to run a fully instantiated Auto-WEKA experiment. Simply select the folder containing the `.experiment` file, enter in a integer for a seed, and push the run button. For example, if your directory structure looks as follows:

```
experiments/
   MyExperiment/
      MyExperiment.experiment
      ....
```

The folder `experiments/MyExperiment` should be selected in the open file dialog.



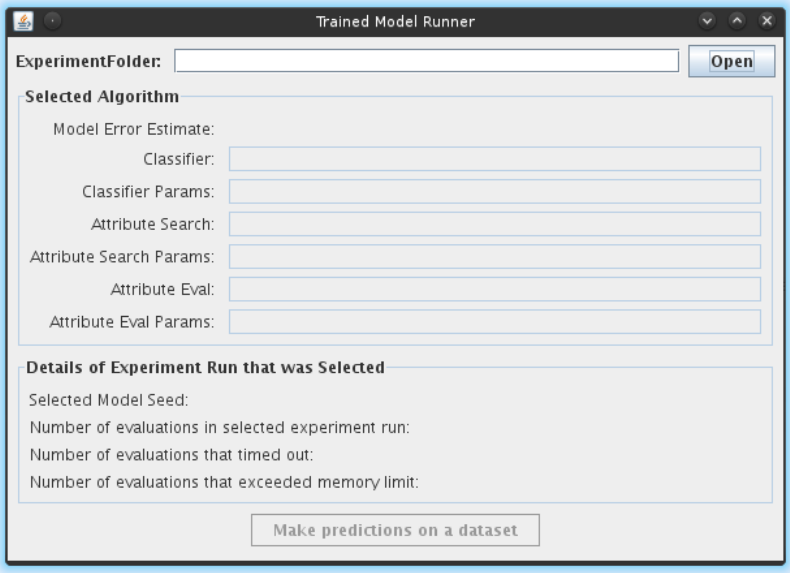
After pressing the run button, the log screen will begin filling up with the output of the optimisation method. Consult the documentation that came with the optimisation method for an explanation of what appears.

Once the time budget has been exhausted, Auto-WEKA will begin training a model using all the training data and the best hyperparameters. If a testing set was provided, Auto-WEKA will print out some basic statistics on the performance of the trained model on the data.

Sometimes the Stop button doesn't succeed in terminating the running process - if this is the case, you'll have to use the various tools of your OS to ensure that the optimisation process is terminated.

3.4 Trained Model Runner

Since Auto-WEKA exploits parallelism by running the optimisation method with different seeds, after running your experiment a few times with unique seeds, you'll want to use the best hyperparameters found to make predictions on new data. After opening up the folder containing the `.experiment`, Auto-WEKA selects the hyperparameters that have the best error estimate. The hyperparameters are then displayed, so you can use them in WEKA's explorer (if you right click on any of the text boxes that contain arguments, WEKA allows you to insert the command line string into it).



Additionally, you can provide new datasets (in ARFF format), and use the trained model to make predictions (the results are saved as a CSV). If attribute selection was used, this will be applied to the dataset before it is passed off to the model. Note that the format of the testing set needs to match the format of the training data exactly, otherwise you will likely get very strange predictions/cryptic error messages.

4 Defining Experiments

If not using the GUI, Auto-WEKA can define experiments either on the command line through a rather tedious process (only documented in the code), or 'experiment definition batches' can be made. An experiment definition batch is an XML file that contains a list of datasets to perform experiments on, and one or more experiment prototypes that contain information such as the type of optimisation method to use or the way to partition the provided training data. These files were designed for performing large comparisons of different settings on multiple datasets, but they can easily be applied to situations where you have a single dataset. Section 4.1 provides a description of how to write these files, but you can also look at the `autoweka.ExperimentBatch` class and its JavaDocs to see how to use them. The GUI also provides a way to generate a simple version of these files, which you can then modify by hand. These experiment definitions also select the choice of optimisation method (see section 4.3), and the way to partition the training data using an 'InstanceGenerator' (see section 4.2).

Once the experiment definition has been created, Auto-WEKA needs to know what type of classification or regression algorithms and feature/attribute selection methods to used. These are all specified in `.param` files, detailed in section 4.4.

After the experiment definition is fully written, the main method of `autoweka.ExperimentConstructor`

can be invoked on the experiment definition file to determine which algorithms can be used on your datasets, and will generate the files needed under the `experiments` folder from the current working directory.

Using the provided script, this command can be executed as follows:
`scripts/autoweka autoweka.ExperimentConstructor path/to/experimentbatch.xml`

4.1 Experiment Definition Files

Experiment definitions are done in XML files, with a root node of an `experimentBatch`. Inside the experiment batch, there can be any number of `datasetComponent` and `experimentComponent` elements - this allows for quickly making experiments with multiple different settings on a variety of datasets. You can produce a template file by running the main method `autoweka.ExperimentBatch` with the name of the template file you wish to create, or saving a definition from the GUI.

4.1.1 datasetComponent

The `datasetComponent` defines a dataset that you want to perform experiments on. Auto-WEKA can take two formats for input. The first requires that datasets are a compressed zip containing exactly two files, `train.arff` and `test.arff`. Auto-WEKA will perform all of its experiments on the training data in `train.arff`, and will use the test data in `test.arff` for analysis after the experiment has completed. (If you don't have any test data, you can just create a dummy dataset file or use the training data again). A `datasetComponent` has the following form in the XML file with self explanatory names

```
<datasetComponent>
  <zipFile>path/to/zip/file.zip</zipFile>
  <name>DatasetName</name>
</datasetComponent>
```

Additionally, Auto-WEKA can be pointed to the `train.arff` and `test.arff` individually. These datasetCompoments have the following form

```
<datasetComponent>
  <trainArff>path/to/train.arff</trainArff>
  <testArff>path/to/train.arff</testArff>
  <name>DatasetName</name>
</datasetComponent>
```

4.1.2 experimentComponent

The `experimentComponent` element contains the parameters that you want to use in an experiment on all of the `datasetComponent`s defined in the same file.

```
<experimentComponent>
```

```
  <name>SMAC-CV10</name>
  <resultMetric>errorRate</resultMetric>
  <experimentConstructor>autoweka.smac.SMACExperimentConstructor</experimentConstructor>
  <instanceGenerator>autoweka.instancegenerators.CrossValidation</instanceGenerator>
  <instanceGeneratorArgs>numFolds=10:seed=0</instanceGeneratorArgs>
  <tunerTimeout>108000</tunerTimeout>
  <trainTimeout>9000.0</trainTimeout>
  <attributeSelection>true</attributeSelection>
  <attributeSelectionTimeout>900</attributeSelectionTimeout>
  <memory>3072m</memory>
  <extraProps></extraProps>
</experimentComponent>
```

name The name of the experiment component, this will be combined with the data set when the full experiment is created.

resultMetric The resultMetric to use. Most likely `errorRate` for classification. `rmse`, `rrse`, `meanAbsoluteErrorMetric` and `relativeAbsoluteErrorMetric` are also supported. See the source for `autoweka.ClassifierResult` for more.

experimentConstructor The name of the class to be used to build the experiment - see 4.3 for a list of what values are supported.

instanceGenerator The name of the class to be used for partitioning the training data, see 4.2 for what classes are implemented.

instanceGeneratorArgs A property string (`var1=val1:var2=val2...`) with arguments to the instance generator.

tunerTimeout The number of seconds to run the optimisation method.

trainTimeout The number of seconds to spend training an algorithm with a set of hyperparameters on a given partition of the training set.

attributeSelection `true` if you want to consider using different feature/attribute selectors, `false` otherwise (or don't include it).

attributeSelectionTimeout Number of seconds to spend doing feature/attribute selection.

memory The memory limit that is passed to the Java instance that is training the classifier or regression method

extraProps Optional extra arguments that can be passed to an experiment

allowedClassifiers Optional argument that restricts what methods can be considered. Specify it multiple times to include a subset of methods, or don't specify it at all to make Auto-WEKA chose all possible methods.

It is also possible to indicate in an experiment if some extra computations should be done on all points in the trajectory during the analysis stage. In your `experimentComponent`, you can create any number of `trajectoryPointExtras`:

```
<trajectoryPointExtras>
  <name>humanReadableName</name>
  <instance>instanceString</instance>
</trajectoryPointExtras>
```

Here, the `name` is a human readable name the corresponds to the `instanceString` that is being executed. The instance string is what is passed to the instance generator, covered in section 4.2. For example, if you want to see how the selected hyperparameters perform over time, you can add the following into your `experimentComponent`.

```
<trajectoryPointExtras>
  <name>Test Performance</name>
  <instance>default</instance>
</trajectoryPointExtras>
```

4.2 Instance Generators

Auto-WEKA uses an instance generator to partition the training and test data provided into new sets of training and test data for the optimisation method to use, e.g. the training data is broken up into 10 folds in cross validation. Each partition of the data is called an 'instance', and can be specified through a string, often in the form of a property string (`var1=val1:var2=val2:...`). Additionally, each Instance Generator also has a string of `instanceGeneratorArgs` that determine what instances will be created in your experiment. For the most common instance generators Auto-WEKA provides, these strings are detailed below.

The instance string `default` however has a the special meaning that the training and test data are unmodified.

4.2.1 Default

The most boring of instance generators - it does nothing to the input training and test data, it ignores any arguments and returns the unmodified partition of instances.

4.2.2 Cross Validation

Performs k-fold cross validation on the training set. Implemented in `autoweka.instancegenerators.CrossValidation`.

Generator Args:
A property string containing the following two elements:

seed The seed to use for randomizing the dataset

numFolds The number of folds to generate

Instance String

A property string containing the following three elements:

seed The seed to use for randomizing the dataset

numFolds The number of folds total

fold The instance's fold number

4.2.3 Random Sub-Sampling

Performs generates an arbitrary number of folds by randomly making a partition of the training data of a fixed percentage. Implemented in `autoweka.instancegenerators.RandomSubSampling`.

Generator Args:
A property string containing the following the following elements:

startingSeed The seed to use for randomizing the dataset

numSamples The number of subsamples to generate

percent The percent of the training data to use as 'new training data'

bias Optional: The bias towards a uniform class distribution

Instance String
A property string containing the following three elements:

seed The seed to use for randomizing the dataset

percent The percent of the training data to use as 'new training data'

bias Optional: The bias towards a uniform class distribution

4.2.4 Termination Holdout

A meta instance generator that removes a random percentage of the training data before passing it on to the target instance generator. Implemented in `autoweka.instancegenerators.TerminationHoldout`.

Generator Args:
A string with three components, separated by `[$]`. The first component contains the parameters for the holdout method (described below), while the second component contains the name of the target instance generator that will receive the subsampled dataset. The final component contains all the arguments that will be passed on to the generator of the target A property string containing the following the following elements:

terminationSeed The seed to use for randomizing the dataset

terminationPercent The percent of the training data to use hold back

terminationBias Optional: The bias towards a uniform class distribution

Instance String

A string with three components, separated by `[$]`. The first component contains the parameters for the holdout method (described below), while the second component contains the name of the target instance generator that will receive the subsampled dataset. The final component contains all the arguments that will be passed on to the generator of the target

terminationSeed The seed to use for randomizing the dataset

terminationPercent The percent of the training data to use as 'new training data'

terminationBias Optional: The bias towards a uniform class distribution

4.3 Optimisation Methods

Currently Auto-WEKA supports three different optimisation methods, the Tree based Parzen Estimator (TPE), Sequential Model-based Algorithm Configuration (SMAC) and Iterated F-Race (IRace). Each method requires some initial set up with Auto-WEKA so that it can be used smoothly, namely by creating `.properties` files that tell Auto-WEKA where to find each method (These files must be in the current working directory when you invoke any of the experiment constructor commands. The syntax of a properties file is of the form `var=value`, with one variable per line). Each of the following sections mentions how to tell Auto-WEKA to use SMBO method, as well as the name and contents of the `.properties` file that must be created.

4.3.1 SMAC

SMAC was designed for algorithm configuration, but can easily be used in other cases of black box optimisation. The Auto-WEKA distribution comes with a development version of SMAC v2.06.01. To build an experiment with SMAC, you use the `autoweka.smac.SMACExperimentConstructor`.

autoweka.smac.SMACExperimentConstructor.properties

`smacexecutable` : The path to the `smac` script (with no `.bat` extension on Windows) inside the SMAC distribution.

A few of SMAC's options are also supported by Auto-WEKA (e.g. `initialIncumbent` and `executionMode`), look inside the
`autoweka.smac.SMACExperimentConstructor` class to see what variables are supported in the `extraProps` of an experiment definition.

4.3.2 TPE

TPE is provided by the Hyperopt project, written for Python 2.7. To build an experiment with TPE, you use the `autoweka.tpe.TPEExperimentConstructor`.

autoweka.smac.SMACExperimentConstructor.properties

`tperunner` : Path to the `tperunner.py` file in the Auto-WEKA source directory

`python` (Optional): The Python you want to use - defaults to trying to find a `python` on the system path.

`pythonpath` (Optional): Sets the PYTHONPATH environment variable before invoking python (useful if you don't have hyperopt inside your site-packages).

4.3.3 IRace

IRace is another tool that was designed for algorithm configuration, with an implementation in R. To build an experiment with IRace, you use the `autoweka.irace.IRaceExperimentConstructor`.

Note that IRace defines its budget in terms of the number of evaluations of different hyperparmeter settings, not in total time. As such, the number of seconds that Auto-WEKA uses for the `tunerTimeout` will be used as the number of evaluations.

---

autoweka.smac.IRaceExperimentConstructor.properties

`iraceexecutable` : Path to the `irace` script inside the R package.

4.4 Parameter Files

Auto-WEKA groups classification and regression algorithms into 3 categories: base, meta and ensemble. Meta methods are methods that take a single base method and use it to perform classification (like AdaBoost), while ensemble methods use a number of base methods to perform classification. Additionally, Auto-WEKA supports feature/attribute selection, through the use of search and evaluator methods. Many of these methods have parameters that influence their behaviour, and these parameters are exposed through the use of `.param` files. The name of the file contains the full class name of the method that we are exposing to Auto-WEKA (so for WEKA's SVM implementation, the file would be called `weka.classifiers.functions.SMO.params`), and it is placed inside the subfolder of the `params` directory corresponding to the method (so the SVM implementation goes inside the `base` subfolder, while the param file for AdaBoost would go inside the `meta` subfolder).

The contents of these files can be broken down into two parts: parameter definitions and conditional statements. For examples for a number of different classification, regression and feature selection methods, see the the provided `.param` files that come with Auto-WEKA.

4.4.1 Parameter Definitions

For each parameter exposed to Auto-WEKA is written on its own line, and has the following form.

`<FLAGS>_<NAME> <DOMAIN> <DEFAULT><TYPEFLAGS>`

`FLAGS` These will all be stripped by Auto-WEKA before they are passed to WEKA. The flags are defined up until the last underscore before the `NAME`. Summary of valid flags:

HIDDEN The parameter will be never be seen by the WEKA method.

INT The parameter will be forced to an integer when it is passed on to WEKA.

QUOTE_START A quote character will be inserted after this parameter, up until the next QUOTE_END

QUOTE_END Inserts a quotation character and removes the parameter

DASHDASH Inserts a double dash (--) into the call string

NAME The name of the argument that WEKA expects on the command line. Note that in most cases, these are single capital letters (and never contain an underscore)

DOMAIN The domain of this parameter, which is either numeric or categorical.

Categorical The domain is specified as a comma separated list of strings in between two curly braces, eg. {v1, v2, ... vk}

Numeric The domain is specified as two comma separated numbers for the lower and upper range of the domain between two square brackets, eg. [0.1, 10]

DEFAULT The default value of the parameter (which must be inside the domain) is specified between two square brackets eg. [1.0]

TYPEFLAGS Optional type flags for numeric domains. If you want to recommend to the SMBO method that this parameter should be treated as an integer, add an i. If the parameter should be sampled on a logarithmic scale, add a l

For example, the parameter file for random forests contains the line:

```
INT_I [2, 256][10]il
```

This indicates that there is a parameter I that must be treated as an integer, with values ranging between 2 and 256 (a default value of 10), and should be sampled log-uniformly.

Additionally, Auto-WEKA defines special treatment to the categorical domain of {REMOVED, REMOVE_PREV}, which can be best demonstarted through an example. Suppose we have a parameter M which is a flag of a classifier that enables aggressive memory caching. If M is set to REMOVED, by Auto-WEKA, then WEKA will receive the argument -M. In the case that M is set to REMOVE_PREV, then Auto-WEKA will completely hide the -M flag from the WEKA classifier.

Note: Arguments are sorted alphabetically by Auto-WEKA before they are passed on to the WEKA method.

4.4.2 Conditionals

For many methods, only some parameters make sense once another parameter takes on a certain value. If this is the case, after all the parameters have been defined in the .param file, you need to inform Auto-WEKA of these conditionals. All conditionals must appear after a Conditionals: line. The format of a conditional line is as follows

<PARAMETER> | <PARENT> in {<VALUE1>, <VALUE2>, ...}

PARAMETER The name of the child parameter that is active based on the value of the parent

PARENT The name of the parent parameter that the conditional depends on

VALUE* If the parent parameter takes on one of the values in this list, then the child parameter will be enabled, Otherwise, the child parameter is disabled.

5 Running Experiments Using the CLI

After generating your experiment using the main method of the ExperimentConstructor, execute the main method of autoweka.Experiment with two parameters - the path to the experiment folder and an initial seed for the random number generator of the SMBO method. Auto-WEKA has been designed to execute many optimisation runs in parallel, simply change the seed that you pass to each invocation of autoweka.Experiment. Auto-WEKA will now grind away for a while until the tunerTimeout has been hit as specified in the experiment definition.

Optionally, you can instead invoke the autoweka.tools.ExperimentRunner with the same arguments as above. This class does the extra step at the end of an experiment to produce a trained model on the entire training dataset so that it can be easily used to make further predictions. (This class performs the same operations as the Experiment Runner window in the GUI).

Once your generate the experiment by running the ExperimentConstructor, Auto-WEKA tries to resolve path names fully, so it is unlikely that you can move these folders around and still run the experiment.

---

6 Analyzing Experiments Using the CLI

The first step in doing any analysis of experiments is to generate the trajectories of the optimisation run. This can be done by invoking the autoweka.TrajectoryParser class, with the arguments of the experiment folder and the seed you want to parse. This step is only required if you did not use the autoweka.tools.ExperimentRunner to execute your experiment.

For each seed that the experiment was run with, there should be a file with the naming scheme of <ExperimentName>.trajectories.<Seed>. If you defined any trajectoryPointExtras in your experiment definition, you'll want to run the main method of autoweka.TrajectoryPointExtraRunner with the arguments
<ExperimentFolder>/<ExperimentName>.trajectories.<Seed> before performing any other analysis. This does runs of the classifer/hyperparameters identified in the trajectory on whatever instance strings you've specified, and stores the result of the error metric and timing information into the trajectory file.

Once all the individual trajectory files are complete, each of these files needs to be merged into a single trajectory group for analysis. Run the main method of autoweka.TrajectoryMerger, with a

single argument of the experiment's directory. This produces a single file `<ExperimentName>.trajectories` inside the experiment's folder.

Finally, to get the best hyper-parameters and method that Auto-WEKA has found on the dataset, run the main method of `autoweka.tools.GetBestFromTrajectoryGroup`, with the single command line argument pointing at the `.trajectories` file that was produced in the last step. This will print out information on how many trajectories were used to select the best, what Auto-WEKA thinks the performance of the selected method on the training set is, as well as which algorithm was chosen and the command line arguments that should be given to the algorithm.

7 CLI Sample Experiment Walkthrough

Include in the distribution is a sample experiment definition that runs SMAC on the German Credit dataset from the UCI repository, this section will show you how to run a typical Auto-WEKA experiment.

First, navigate to the `autoweka` directory that contains the `autoweka.jar` file. First, we need to build the actual experiment by running the `ExperimentConstructor`

For all the following Java commands, we assume that you run them inside the `autoweka` directory, and add both `autoweka.jar` and `weka.jar` to the Java's class path.

```
java -cp autoweka.jar  autoweka.ExperimentConstructor sampleexperiment.xml
```

This will load the dataset in `sampledata/creditg.zip`, determine what classifiers and feature selectors that are defined in the `params` directory can be used, and write out the experiment into the `experiments` directory.

Note that you will have to modify `autoweka.smac.SMACExperimentConstructor.properties` to point to your distribution of SMAC, if you are not using the version that came bundled with Auto-WEKA.

Next, we need to run our experiments by invoking the `Experiment` with different seeds

```
java -cp autoweka.jar autoweka.Experiment experiments/SMAC-CV10-GermanCredit 0
java -cp autoweka.jar autoweka.Experiment experiments/SMAC-CV10-GermanCredit 1
java -cp autoweka.jar autoweka.Experiment experiments/SMAC-CV10-GermanCredit 2
...
```

After watching some paint dry (and the experiments have completed), we now need to generate and combine all the trajectories into a single file using the `TrajectoryMerger`

```
java -cp autoweka.jar autoweka.TrajectoryParser experiments/SMAC-CV10-GermanCredit 0
java -cp autoweka.jar autoweka.TrajectoryParser experiments/SMAC-CV10-GermanCredit 1
java -cp autoweka.jar autoweka.TrajectoryParser experiments/SMAC-CV10-GermanCredit 2
...
java -cp autoweka.jar autoweka.TrajectoryMerger experiments/SMAC-CV10-GermanCredit
```

If you've defined a number of `trajectoryPointExtras` in your experiment definition, you'll want to invoke

```
java -cp autoweka.jar  autoweka.TrajectoryPointExtraRunner \
            experiments/SMAC-CV10-GermanCredit/SMAC-CV10-GermanCredit.trajectories.0
```

for each completed trajectory before you run the merger.

Now, you can perform any kind of analysis on this merged trajectory file, but the most common operation you'd want is to find the classifier/hyperparameters that have the best performance using `GetBestFromTrajectoryGroup`

```
java -cp autoweka.jar  autoweka.tools.GetBestFromTrajectoryGroup \
           experiments/SMAC-CV10-GermanCredit/SMAC-CV10-GermanCredit.trajectory
```

Additionally, if you use the `autoweka.tools.ExperimentRunner` to execute your experiments, you can use the class `autoweka.tools.TrainedModelPredictionRunner` to make predictions on new data:

```
java -cp autoweka.jar  autoweka.tools.TrainedModelPredictionRunner \
           -model path/to/trained.SEED.model \
           -attributeselection path/to/trained.SEED.attributeselection \
           -dataset path/to/test.arff \
           -predictionpath path/to/predictions.csv
```

8 Extending Auto-WEKA

Auto-WEKA has been designed to be relatively easy to extend with new optimisation methods/instance generators/machine learning algorithms. The core classes in Auto-WEKA all have JavaDoc, and comments throughout the code that should help explain what each bit does.

To add a new optimisation method, you need to provide three classes, an `ExperimentConstructor`, a `TrajectoryParser`, and a `Wrapper`. The `ExperimentConstructor` converts an experiment definition to an actual experiment file (by generating any extra data that is needed by the optimiser), while the `TrajectoryParser` extracts the results of the optimiser into a format that can readily be used by the rest of the Auto-WEKA tools. The `Wrapper` class provides a way to convert parameters from the optimiser into something that can be understood by Auto-WEKA (which in then turn is passed on to WEKA). This class is also responsible for reporting the error rate back to the optimiser, along with the time it took to train the classifier or regression method. Looking at the provided implementations for SMAC, TPE and IRace should be sufficient in determining how to write your own methods.

New instance generators can be created by extending `autoweka.InstanceGenerator`, and just ensuring that they are on the classpath when you invoke the `ExperimentConstructor`. Looking at the provided generators should be sufficient for creating your own, (which would allow you to build generators that don't require the entire dataset loaded into RAM).

Adding a new machine learning algorithm into Auto-WEKA is as simple as creating a new `.param` file in the appropriate subfolder under the `params` directory, and ensuring that your classifier is on Java's classpath when you invoke the `ExperimentConstructor` or UI.