Sparkle Documentation

Release 0.9.0

ADA Research Group, AIM @ RWTH Aachen

Nov 05, 2024

Platform

> A Programming by Optimisation (PbO)-based problem-solving platform designed to enable the widespread and effective use of PbO techniques for improving the state-of-the-art in solving a broad range of prominent AI problems, including SAT and AI Planning.

Specifically, Sparkle facilitates the use of:

- Automated algorithm configuration
- Automated algorithm selection

Furthermore, Sparkle handles various tasks for the user such as:

- Algorithm meta information collection and statistics calculation
- Instance/Data Set management and feature extraction
- Compute cluster job submission and monitoring
- Log file collection

CHAPTER 1

Installation

The quick and full installation of Sparkle can be done using Conda (For Conda installation see here).

Simply download the `environment.yml` file from the Github with wget:

```
wget https://raw.githubusercontent.com/ADA-research/Sparkle/main/environment.yml
```

and run:

```
conda env create -f environment.yml
```

The installation of the environment may take up to five minutes depending on your internet connection. Once the environment has been created it can be activated by:

```
conda activate sparkle
```

Note

The creation of the Conda environment also takes care of the installation of the Sparkle package itself.

Note

You will need to reactivate the environment every time you start the terminal, before using Sparkle.

Sparkle can also be installed as a standalone package using Pip. We recommend creating a new virtual environment (For example, venv) before to ensure no clashes between dependencies occur.

```
pip install SparkleAI
```

Note that a direct installation through Pip does not handle certain dependencies of the Sparkle CLI, such as the required libraries for compiling RunSolver.

1.1 Install dependencies

Asside from several package dependencies, Sparkle's package / CLI relies on a few user supplied executables:

- `LaTex` compiler (pdflatex) for report generation
- `Java`, tested with version 1.8.0_402, in order to use SMAC2
- `R` 4.3.1, in order to use IRACE

Other dependencies are handled by the Conda environment, but if that is not an option for you please ensure you have the following:

- libnuma and numactl for Runsolver compilation which sparkle uses to measure solvers meta data. This is restricted to Linux based systems.
- Swig 4.0.2 for SMAC3, which is in turn used by AutoFolio.

For detailed installation instructions see the documentation: https://ada-research.github.io/Sparkle/

1.2 Developer installation

The file `dev-env.yml` is used for developer mode of the Sparkle package and contains several extra packages for testing.

The two environments can be created in parallel since one is named `sparkle` and the other `sparkle-dev`. If you want to update an environment it is better to do a clean installation by removing and recreating it. For example:

```
conda deactivate
conda env remove -n sparkle
conda env create -f environment.yml
conda activate sparkle
```

This should be fast as both `conda` and `pip` use local cache for the packages.

1.2.1 Examples

See the `Examples` directory for some examples on how to use `Sparkle`. All Sparkle CLI commands need to be executed from the root of the initialised Sparkle directory.

1.2.2 Documentation

The documentation can be read at https://ada-research.github.io/Sparkle/.

A `PDF` is also available in the repository.

1.2.3 Licensing

Sparkle is distributed under the MIT licence

Component licences

Sparkle is distributed with a number of external components, solvers, and instance sets. Descriptions and licensing information for each these are included in the `sparkle/Components` and `Examples/Resources/` directories.

The SATzilla 2012 feature extractor is used from `http://www.cs.ubc.ca/labs/beta/Projects/SATzilla/` with some modifications. The main modification of this component is to disable calling the SAT instance preprocessor called SatELite. It is located in: `Examples/Resources/Extractors/SAT-features-competition2012_revised_without_SatELite_sparkle/`

1.3 Citation

If you use Sparkle for one of your papers and want to cite it, please cite our paper describing Sparkle: K. van der Blom, H. H. Hoos, C. Luo and J. G. Rook, **Sparkle: Toward Accessible Meta-Algorithmics for Improving the State of the Art in Solving Challenging Problems**, in *IEEE Transactions on Evolutionary Computation*, vol. 26, no. 6, pp. 1351-1364, Dec. 2022, doi: 10.1109/TEVC.2022.3215013.

```
@article{BloEtAl22,
  title={Sparkle: Toward Accessible Meta-Algorithmics for Improving the State of the Art␣
↪in Solving Challenging Problems},
  author={van der Blom, Koen and Hoos, Holger H. and Luo, Chuan and Rook, Jeroen G.},
  journal={IEEE Transactions on Evolutionary Computation},
  year={2022},
  volume={26},
  number={6},
  pages={1351--1364},
  doi={10.1109/TEVC.2022.3215013}
}
```

1.4 Maintainers

Thijs Snelleman, Jeroen Rook, Holger H. Hoos,

1.5 Contributors

Chuan Luo, Richard Middelkoop, Jérémie Gobeil, Sam Vermeulen, Marcel Baumann, Jakob Bossek, Tarek Junied, Yingliu Lu, Malte Schwerin, Aaron Berger, Marie Anastacio, Aaron Berger Koen van der Blom, Noah Peil, Brian Schiller

1.6 Contact

sparkle@aim.rwth-aachen.de

1.7 Sponsors

The development of Sparkle is partially sponsored by the Alexander von Humboldt foundation.

1.7.1 Quick Start

The Sparkle package offers an extensive *Command Line Interface* (CLI) to allow for easy interaction with the platform. If you haven't installed the package already, see the *Install Sparkle* page.

Note

Sparkle currently relies on Slurm, but in some cases works locally as well. Sparkle also relies on RunSolver, which is a Linux based program. Thus Sparkle can only run on Linux based systems in many cases.

To initialise a new Sparkle platform, select a (preferably new / empty) directory, and run in the terminal:

```
sparkle initialise
```

This sets up various default directories and files for Sparkle to use that you can customise later. Note that if you wish to download the files for the *Tutorials*, you can run the command with the flag:

```
sparkle initialise --download-examples
```

Due to the examples containing various algorithms and their executables, but also entire datasets, it is about 300MB large. Now you might receive a few warnings during the inialisation, lets go through a few of them:

- *"Could not find Java as an executable"*: One of the algorithm configurators Sparkle has to offer (SMAC2) is build on Java. Make sure your system has Java version 1.8.0_402 installed.
- *"R is not installed, which is required for the IRACE"*: Sparkle offers the algorithm configurator IRACE for its users, but as this runs in R, the user needs to supply this in their environment. The currently tested version is 4.3.1 ("Beagle Scouts")
- *"RunSolver was not compiled succesfully"*: Sparkle uses RunSolver to monitor algorithms and their meta statistics, such as CPU and Wallclock time. RunSolver needs various libraries to be compiled. You can run the make file in the Sparkle package components section (sparkle/Components/runsolver/src/Makefile) to inspect what your system is missing for the compilation to work.

1.7.2 Tutorials

In this section we demonstrate the usage of the platform for Algorithm Configuration, the creation of Algorithm Portfolios and Algorithm Selection.

Setting up Sparkle

Before running Sparkle, you probably want to have a look at the settings described in the *Platform* section. In particular, the default Slurm settings should be reconfigured to work with your cluster, for example by specifying a partition to run on.

Recompilation of example Solvers

Although the examples come precompiled with the download, in some cases they may not directly work on your target system due to certain target-system specific choices that are made during compilation. You can follow the steps below to re-compile.

CSCCSat

The CSCCSat Solver can be recompiled as follows in the `Examples/Resources/Solvers/CSCCSat/` directory:

```
unzip src.zip
cd src/CSCCSat_source_codes/
make
cp CSCCSat ../../
```

MiniSAT

The MiniSAT solver can be recompiled as follows in the `Examples/Resources/Solvers/MiniSAT/` directory:

```
unzip src.zip
cd minisat-master/
make
cp build/release/bin/minisat ../
```

PbO-CCSAT

The PbO-CCSAT solver can be recompiled as follows in the `Examples/Resources/Solvers/PbO-CCSAT-Generic/` directory:

```
unzip src.zip
cd PbO-CCSAT-master/PbO-CCSAT_process_oriented_version_source_code/
make
cp PbO-CCSAT ../../
```

TCA and FastCA

The TCA and FastCA solvers, require `GLIBCXX_3.4.21`. This library comes with `GCC 5.1.0` (or greater). Following installation you may have to update environment variables such as `LD_LIBRARY_PATH, LD_RUN_PATH, CPATH` to point to your installation directory.

TCA can be recompiled as follows in the `Examples/Resources/CCAG/Solvers/TCA/` directory:

```
unzip src.zip
cd TCA-master/
make clean
make
cp TCA ../
```

FastCA can be recompiled as follows in the `Examples/Resources/CCAG/Solvers/FastCA/` directory:

```
unzip src.zip
cd fastca-master/fastCA/
make clean
make
cp FastCA ../../
```

VRP_SISRs

VRP_SISRs solver can be recompiled as follows in the `Examples/Resources/CVRP/Solvers/VRP_SISRs/` directory:

```
unzip src.zip
cd src/
make
cp VRP_SISRs ../
```

Algorithm Runtime Configuration

These steps can also be found as a Bash script in `Examples/configuration.sh`

Initialise the Sparkle platform

```
sparkle initialise
```

Add instances

Add train, and optionally test, instances (in this case in CNF format) in a given directory, without running solvers or feature extractors yet

```
sparkle add_instances Examples/Resources/Instances/PTN/
sparkle add_instances Examples/Resources/Instances/PTN2/
```

Add a configurable solver

Add a configurable solver (here for SAT solving) with a wrapper containing the executable name of the solver and a string of command line parameters, without running the solver yet

The solver directory should contain the solver executable, the `sparkle_solver_wrapper` wrapper, and a `.pcs` file describing the configurable parameters

```
sparkle add_solver Examples/Resources/Solvers/PbO-CCSAT-Generic/
```

If needed solvers can also include additional files or scripts in their directory, but keeping additional files to a minimum speeds up copying.

Configure the solver

To perform configuration on the solver to obtain a target configuration we run:

```
sparkle configure_solver --solver Solvers/PbO-CCSAT-Generic/ --instance-set-train␣
↪Instances/PTN/
```

This step should take about ~10 minutes, although it is of course very cluster / slurm settings dependant.

Validate the configuration

To make sure configuration is completed before running validation you can use the `wait` command

```
sparkle wait
```

Now we can validate the performance of the best found parameter configuration against the default configuration specified in the PCS file. The test set is optional.

```
sparkle validate_configured_vs_default --solver Solvers/PbO-CCSAT-Generic/ --instance-
↪set-train Instances/PTN/ --instance-set-test Instances/PTN2/
```

Generate a report

Wait for validation to be completed

```
sparkle wait
```

Generate a report detailing the results on the training (and optionally testing) set. This includes the experimental procedure and performance information; this will be located in a `Configuration_Reports/` subdirectory for the solver, training set, and optionally test set like `PbO-CCSAT-Generic_PTN/Sparkle-latex-generator-for-configuration/`

```
sparkle generate_report
```

By default the `generate_report` command will create a report for the most recent solver and instance set(s). To generate a report for older solver-instance set combinations, the desired solver can be specified with `--solver Solvers/PbO-CCSAT-Generic/`, the training instance set with `--instance-set-train Instances/PTN/`, and the testing instance set with `--instance-set-test Instances/PTN2/`.

Run ablation

We can run ablation to determine parameter importance based on default (from the `.pcs` file) and configured parameters. To run ablation using the training instances and validate the parameter importance with the test set

```
sparkle run_ablation --solver Solvers/PbO-CCSAT-Generic/ --instance-set-train Instances/
↪PTN/ --instance-set-test Instances/PTN2/
```

Generate a report

Wait for ablation to be completed

```
sparkle wait
```

Generate a report including ablation, and as before the results on the train (and optionally test) set, the experimental procedure and performance information; this will be located in a `Configuration_Reports/` subdirectory for the solver, training set, and optionally test set like `PbO-CCSAT-Generic_PTN/Sparkle-latex-generator-for-configuration/`

```
sparkle generate_report
```

The ablation section can be suppressed with `--no-ablation`

Immediate ablation and validation after configuration

By adding `--ablation` and/or `--validate` to the `configure_solver` command, ablation and respectively validation will run directly after the configuration is finished.

There is no need to execute `run_ablation` and/or `validate_configured_vs_default` when these flags are given with the `configure_solver` command

Training set only

```
sparkle configure_solver --solver Solvers/PbO-CCSAT-Generic/ --instance-set-train␣
↪Instances/PTN/ --ablation --validate
```

Training and testing sets

Wait for the previous example to be completed

```
sparkle wait
sparkle configure_solver --solver Solvers/PbO-CCSAT-Generic/ --instance-set-train␣
↪Instances/PTN/ --instance-set-test Instances/PTN2/ --ablation --validate
```

Run configured solver

Run configured solver on a single instance

Now that we have a configured solver, we can run it on a single instance to get a result.

```
sparkle run_configured_solver Examples/Resources/Instances/PTN2/Ptn-7824-b20.cnf
```

Run configured solver on an instance directory

It is also possible to run a configured solver directly on an entire directory.

```
sparkle run_configured_solver Examples/Resources/Instances/PTN2
```

Algorithm Quality Configuration

We can configure an algorithm too based on some quality objective, that can be defined by the user. See the *SparkleObjective* page for all options regarding objective defintions. These steps can also be found as a Bash script in `Examples/configuration_qualty.sh`

Initialise the Sparkle platform

```
sparkle initialise
```

Add instances

Now we add train, and optionally test, instances for configuring our algorithm (in this case for the VRP). The instance sets are placed in a given directory.

```
sparkle add_instances Examples/Resources/CVRP/Instances/X-1-10/
sparkle add_instances Examples/Resources/CVRP/Instances/X-11-20/
```

Add a configurable solver

Add a configurable solver (In this tutorial its an algorithm for vehicle routing) with a wrapper containing the executable name of the solver and a string of command line parameters.

The solver directory should contain the `sparkle_solver_wrapper.py` wrapper, and a `.pcs` file describing the configurable parameters.

```
sparkle add_solver Examples/Resources/CVRP/Solvers/VRP_SISRs/
```

In this case the source directory also contains an executable, as the algorithm has been compiled from another programming language (C++). If needed solvers can also include additional files or scripts in their directory, but keeping additional files to a minimum speeds up copying.

Configure the solver

Perform configuration on the solver to obtain a target configuration. For the VRP we measure the absolute quality performance by setting the `--objectives` option, to avoid needing this for every command it can also be set in `Settings/sparkle_settings.ini`.

```
sparkle configure_solver --solver Solvers/VRP_SISRs/ --instance-set-train Instances/X-1-
↪10/ --objectives quality
```

Validate the configuration

To make sure configuration is completed before running validation you can use the `sparkle wait` command

```
sparkle wait
```

Validate the performance of the best found parameter configuration. The test set is optional. We again set the performance measure to absolute quality.

```
sparkle validate_configured_vs_default --solver Solvers/VRP_SISRs/ --instance-set-train␣
↪Instances/X-1-10/ --instance-set-test Instances/X-11-20/ --objective quality
```

Generate a report

Wait for validation to be completed

```
sparkle wait
```

Generate a report detailing the results on the training (and optionally testing) set. This includes the experimental procedure and performance information; this will be located in a `Configuration_Reports/` subdirectory for the solver, training set, and optionally test set like `VRP_SISRs_X-1-10_X-11-20/Sparkle-latex-generator-for-configuration/`. We again set the performance measure to absolute quality.

```
sparkle generate_report --objective quality
```

By default the `generate_report` command will create a report for the most recent solver and instance set(s). To generate a report for older solver-instance set combinations, the desired solver can be specified with `--solver Solvers/VRP_SISRs/`, the training instance set with `--instance-set-train Instances/X-1-10/`, and the testing instance set with `--instance-set-test Instances/X-11-20/`.

Configuring Random Forest on Iris

We can also use Sparkle for Machine Learning approaches, such as Random Forest for the Iris data set. Note that in this case, the entire data set is considered as being one instance.

Initialise the Sparkle platform

```
sparkle initialise
```

Add instances

```
sparkle add_instances Examples/Resources/Instances/Iris
```

Add solver

```
sparkle add_solver Examples/Resources/Solvers/RandomForest
```

Configure the solver on the data set

```
sparkle configure_solver --solver RandomForest --instance-set-train Iris --objectives␣
↪accuracy:max

sparkle wait
```

Validate the performance of the best found parameter configuration. The test set is optional.

```
sparkle validate_configured_vs_default --solver RandomForest --instance-set-train Iris --
↪objectives accuracy:max
```

Generate a report

Wait for validation to be completed

```
sparkle wait
```

Generate a report detailing the results on the training (and optionally testing) set.

```
sparkle generate_report --objectives accuracy:max
```

Running a Parallel Portfolio

In this tutorial we will measure the runtime performance of several algorithms in parallel. The general idea is that we consider the algorithms as a portfolio that we run in parallel (hence the name) and terminate all running algorithms once a solution is found.

Initialise the Sparkle platform

```
sparkle initialise
```

Add instances

First we add the instances to the platform that we want to use for our experiment. Note that if our instance set contains multiple instances, the portfolio will attempt to run them all in parallel. Note that you should use the full path to the directory containing the instance(s)

```
sparkle add_instances Examples/Resources/Instances/PTN/
```

Add solvers

Now we can add our solvers to the portfolio that we want to "race" in parallel against eachother. The path used should be the full path to the solver directory and should contain the solver executable and the `sparkle_solver_wrapper` wrapper. It is always a good idea to keep the amount of files in your solver directory to a minimum.

```
sparkle add_solver Examples/Resources/Solvers/CSCCSat/
sparkle add_solver Examples/Resources/Solvers/MiniSAT/
sparkle add_solver Examples/Resources/Solvers/PbO-CCSAT-Generic/
```

Run the portfolio

By running the portfolio a list of jobs will be created which will be executed by the cluster. Use the `--cutoff-time` option to specify the maximal time for which the portfolio is allowed to run. add `--portfolio-name` to specify a portfolio otherwise it will select the last constructed portfolio

The `--instance-path` option must be a path to a single instance file or an instance set directory. For example `--instance-path Instances/Instance_Set_Name/Single_Instance`.

If your solvers are non-deterministic (e.g. the random seed used to start your algorithm can have an impact on the runtime), you can set the amount of jobs that should start with a random seed per algorithm. Note that scaling up this variable has a significant impact on how many jobs will be run (Number of instances * number of solvers * number of seeds). We can set using the `--solver-seeds` argument followed by some positive integer.

```
sparkle run_parallel_portfolio --instance-path Instances/PTN/ --portfolio-name runtime_
↪experiment
```

Generate the report

The report details the experimental procedure and performance information. This will be located at `Output/Parallel_Portfolio/Sparkle_Report.pdf`

```
sparkle generate_report
```

Algorithm Selection

Sparkle also offers various tools to apply algorithm selection, where we, given an objective, train another algorithm to determine which solver is best to use based on an instance.

These steps can also be found as a Bash script in `Examples/selection.sh`

Initialise the Sparkle platform

```
sparkle initialise
```

Add instances

First, we add instance files (in this case in CNF format) to the platform by specifying the path.

```
sparkle add instances Examples/Resources/Instances/PTN/
```

Add solvers

Now we add solvers to the platform as possible options for our selection. Each solver directory should contain the solver wrapper.

```
sparkle add solver Examples/Resources/Solvers/CSCCSat/
sparkle add solver Examples/Resources/Solvers/PbO-CCSAT-Generic/
sparkle add solver Examples/Resources/Solvers/MiniSAT/
```

Add feature extractor

To run the selector, we need certain features to represent our instances. To that end, we add a feature extractor to the platform that creates vector representations of our instances.

```
sparkle add feature extractor Examples/Resources/Extractors/SAT-features-competition2012_
↪revised_without_SatELite_sparkle/
```

Compute features

Now we can run our features with the following command:

```
sparkle compute features
```

Run the solvers

Similarly, we can now also compute our objective values for our solvers, in this case PAR10. Note that we can at this point still specifiy multiple objectives by separating them with a comma, or denote them in our settings file.

```
sparkle run solvers --objective PAR10
```

Construct a portfolio selector

To make sure feature computation and solver performance computation are done before constructing the portfolio use the `wait` command

```
sparkle wait
```

Now we can construct a portfolio selector, using the previously computed features and the results of running the solvers. The `--selector-timeout` argument determines for how many seconds we will train our selector for. We can set the flag `--solver-ablation` for actual marginal contribution computation later.

```
sparkle construct portfolio selector --selector-timeout 1000 --solver-ablation
sparkle wait  # Wait for the constructor to complete its computations
```

Generate a report

Generate an experimental report detailing the experimental procedure and performance information; this will be located at `Output/Selection/Sparkle_Report.pdf`

```
sparkle generate report
```

Run the portfolio selector

Run on a single instance

Run the portfolio selector on a *single* testing instance; the result will be printed to the command line if you add `--run-on local` to the command.

```
sparkle run portfolio selector Examples/Resources/Instances/PTN2/plain7824.cnf
```

Run on an instance set

Run the portfolio selector on a testing instance *set*

```
sparkle run portfolio selector Examples/Resources/Instances/PTN2/
sparkle wait  # Wait for the portfolio selector to be done running on the testing instance set
```

Generate a report including results on the test set

Generate an experimental report that includes the results on the test set, and as before the experimental procedure and performance information; this will be located at `Output/Selection/Sparkle_Report_For_Test.pdf`

```
sparkle generate report
```

By default the `generate_report` command will create a report for the most recent instance set. To generate a report for an older instance set, the desired instance set can be specified with: `--test-case-directory Test_Cases/PTN2/`

Comparing against SATZilla 2024

If you wish to compare two feature extractors against one another, you need to remove the previous extractor from the platform (Or create a new platform from scratch) by running:

```
sparkle remove feature extractor SAT-features-competition2012_revised_without_SatELite_
↪sparkle
```

Otherwise, Sparkle will interpret adding the other feature extractor as creating a combined feature vector per instance from all present extractors in Sparkle. Now we can add SATZilla 2024 from the Examples directory Note that this feature extractor requires GCC (any version, tested with 13.2.0) to run.

```
sparkle add feature extractor Examples/Resources/Extractors/SAT-features-competition2024
```

We can also investigate a different data set, SAT Competition 2023 for which Sparkle has a subset.

```
sparkle remove instances PTN
sparkle add instances Examples/Resources/Instances/SATCOMP2023_SUB
```

We compute the features for the new extractor and new instances.

```
sparkle compute features
sparkle wait  # Wait for it to complete before continuing
```

And run the solvers on the new data set.

```
sparkle run solvers
sparkle wait
```

Now we can train a selector based on these features.

```
sparkle construct portfolio selector --selector-timeout 1000
sparkle wait  #Wait for the computation to be done
```

And generate the report. When running on the PTN/PTN2 data sets, you can compare the two to see the impact of different feature extractors.

```
sparkle generate report
```

Algorithm selection with multi-file instances

We can also run Sparkle on problems with instances that use multiple files. In this tutorial we will perform algorithm selection on instance sets with multiple files.

Initialise the Sparkle platform

```
sparkle initialise
```

Add instances

Add instance files in a given directory, without running solvers or feature extractors yet. In addition to the instance files, the directory should contain a file `sparkle_instance_list.txt` where each line contains a space separated list of files that together form an instance.

```
sparkle add_instances Examples/Resources/CCAG/Instances/CCAG/
```

Add solvers

Add solvers (here for the constrained covering array generation (CCAG) problem) with a wrapper containing the executable name of the solver and a string of command line parameters, without running the solvers yet

Each solver directory should contain the solver executable and a wrapper

```
sparkle add_solver Examples/Resources/CCAG/Solvers/TCA/
sparkle add_solver Examples/Resources/CCAG/Solvers/FastCA/
```

Add feature extractor

Similarly, add a feature extractor, without immediately running it on the instances

```
sparkle add_feature_extractor Examples/Resources/CCAG/Extractors/CCAG-features_sparkle/
```

Compute features

Compute features for all the instances

```
sparkle compute_features
```

Run the solvers

Run the solvers on all instances. For the CCAG (Constrained Covering Array Generation) problem we measure the quality objective by setting the `--objectives` option, to avoid needing this for every command it can also be set in `Settings/sparkle_settings.ini`.

```
sparkle run_solvers --objectives quality
```

Construct a portfolio selector

To make sure feature computation and solver performance computation are done before constructing the portfolio use the `wait` command

```
sparkle wait
```

Construct a portfolio selector, using the previously computed features and the results of running the solvers. We again set the objective measure to quality.

```
sparkle construct_portfolio_selector --objectives quality
```

Running the selector

Run on a single instance

Run the portfolio selector on a *single* testing instance; the result will be printed to the command line if you add `--run-on local` to the command. We again set the objective to quality.

```
sparkle run_portfolio_selector Examples/Resources/CCAG/Instances/CCAG2/Banking2.model␣
↪Examples/Resources/CCAG/Instances/CCAG2/Banking2.constraints --objectives quality
```

Run on an instance set

Run the portfolio selector on a testing instance *set*. We again set the objective to quality.

```
sparkle run_portfolio_selector Examples/Resources/CCAG/Instances/CCAG2/ --objectives␣
↪quality
```

Generate a report including results on the test set

Wait for the portfolio selector to be done running on the testing instance set

```
sparkle wait
```

Generate an experimental report that includes the results on the test set, and as before the experimental procedure and performance information; this will be located at `Components/Sparkle-latex-generator/Sparkle_Report_For_Test.pdf`. We again set the obejctive to quality.

```
sparkle generate_report --objectives quality`
```

By default the `generate_report` command will create a report for the most recent instance set. To generate a report for an older instance set, the desired instance set can be specified with: `--test-case-directory Test_Cases/CCAG2/`

1.7.3 Wrapping your Algorithm

When using Sparkle for your specific projects, you will want to plug in your own algorithms into the platform. To that end, a piece of wrapper code of about ~50 lines must be written to make sure the platform is able to submit calls to your algorithm, as well as parse the output. This should in general not take longer than five minutes to write.

A template for the wrapper that connects your algorithm with Sparkle is available at `Examples/Resources/Solvers/template/sparkle_solver_wrapper.py`. Within this template a number of `TODO`s are indicated where you are likely to need to make changes for your specific algorithm. You can also compare the different example solvers to get an idea for what kind of changes are needed.

Solver Wrapper Python script

The `sparkle_solver_wrapper.py` receives via commandline a dictionary as its inputs. This can be easily parsed using a Sparkle tool: `from sparkle.tools.solver_wrapper_parsing import parse_solver_wrapper_args`. After parsing it with the Sparkle tools, the dictionary should always have the following values:

```
solver_dir: Path
instance: Path,
objectives: list[str],
cutoff_time: float,
seed: int
```

The solver_dir specifies the Path to the Solver directory of your algorithm, where your optional additional files can be found. This can be empty, e.g. the cwd contains all your extra files. This can be useful when your algorithm is an executable that you need to run from the wrapper. The instance is the path to the instance we are going to run on. Cutoff time is the maximum amount of time your algorithm is allowed to run, which you set yourself in the `sparkle_settings.ini` under section `general` as option `target_cutoff_time`. Seed is the seed for this run.

When using Sparkle for algorithm configuration, this dictionary will also contain the (hyper)parameter values for your solver to use. These will all be in string format. See *Parameter configuration space* for more information.

A solver wrapper should always return a dictionary by printing it, containing the following values:

```
status: Enum,
objective: any,
...
solver_call: str (optional)
```

Status can hold the following various values such as `{SUCCESS, TIMEOUT, CRASHED}`, see *SolverStatus* for a description of the Enum. If the status is not known, reporting `SUCCESS` will allow Sparkle to continue, but may mean that Sparkle does not know when the algorithm crashed, and continues with faulty results. To return the values of your objectives, make sure to specify them with the exact same key string as they are specified in your Settings. This key is used to map it into the platform. If you have multiple objectives, simply place each key value pair in the dictionary. The solver_call is only used for logging purposes, to allow for easy inspection of the solver wrapper's subprocess.

PCS file

In order to use algorithm configuration, the algorithm configuration space must be specified in a PCS (Parameter configuration space) file.

> **Note**
>
> See the *tutorial* page for a walk-through on how to perform configuration with Sparkle.

The PCS (parameter configuration space) format is used to pass the possible parameter ranges of an algorithm to Sparkle in a `.pcs` file. For an example see e.g. `Examples/Resources/Solvers/PbO-CCSAT-Generic/PbO-CCSAT-params_test.pcs`.

In this file you should enter all configurable parameters of your algorithm. Note that parameters such as the random seed used by the algorithm should not be configured and therefore should also not be included in the PCS file.

> **Warning**
>
> Although you can specify *default* values for your parameters, it is not guaranteed each parameter will always be present in the input dictionary. It is therefore strongly encouraged to have the default/back up values available for each parameter in your wrapper.

1.7.4 Commands

Executing commands

Executing commands in Sparkle is as simple as running them terminal for example:

```
sparkle initialise
```

Do note that when running on a cluster additional arguments may be needed, for instance under the Slurm workload manager. All CLI entries are placed in the sparkle package `sparkle/CLI/$COMMANDNAME$.py` the above command would change to something like:

```
srun -N1 -n1 -c1 path/to/package/sparkle/CLI/initialise.py
```

In the `Examples/` directory a number of common command sequences are given. For instance, for configuration with specified training and testing sets see e.g. `Examples/configuration_runtime.sh` for an example of a sequence of commands to execute. Note that some command run in the background and need time to complete before the next command is executed. To see whether a command is still running the `wait` command can be used.

In the `Output/` directory paths to generated scripts and logs are gathered per executed command.

> **Note**
>
> When typing a sparkle command name that consists of multiple words, both spaces and underscores are accepted as seperators.

List of Commands

Currently the commands below are available in Sparkle (listed alphabetically). Every command can be called with the `-help` option to get a description of the required arguments and other options.

- cmd-about
- cmd-add-feature-extractor
- cmd-add-instances
- cmd-add-solver
- cmd-cancel
- cmd-cleanup
- cmd-compute-features
- cmd-compute-marginal-contribution
- cmd-configure-solver
- cmd-construct-portfolio-selector
- cmd-generate-report
- cmd-initialise
- cmd-load-snapshot
- cmd-remove-feature-extractor
- cmd-remove-instances
- cmd-remove-solver
- cmd-run-ablation
- cmd-run-configured-solver
- cmd-run-parallel-portfolio
- cmd-run-portfolio-selector
- cmd-run-solvers
- cmd-save-snapshot
- cmd-status
- cmd-validate-configured-vs-default
- cmd-wait

Note

Arguments in [square brackets] are optional, arguments without brackets are mandatory. Input in <chevrons> indicate required text input, {curly brackets} indicate a set of inputs to choose from.

about

```
usage: about [-h]
```

-h, --help
: show this help message and exit

add_feature_extractor

Add a feature extractor to the platform.

```
usage: add_feature_extractor [-h] [--run-extractor-now] [--nickname NICKNAME] extractor-
↪path
```

extractor-path
: path or nickname of the feature extractor

-h, --help
: show this help message and exit

--run-extractor-now
: immediately run the feature extractor(s) on all the instances

--nickname <nickname>
: set a nickname for the feature extractor

add_instances

Add instances to the platform.

```
usage: add_instances [-h] [--run-extractor-now] [--run-solver-now] [--nickname NICKNAME]␣
↪[--run-on {Runner.LOCAL,Runner.SLURM}] instances-path
```

instances-path
: path to the instance set

-h, --help
: show this help message and exit

--run-extractor-now
: immediately run the feature extractor(s) on all the instances

--run-solver-now
: immediately run the solver(s) on all instances

--nickname <nickname>
: set a nickname for the instance set

--run-on
: On which computer or cluster environment to execute the calculation.

add_solver

Add a solver to the Sparkle platform.

```
usage: add_solver [-h] [--deterministic] [--run-solver-now] [--nickname NICKNAME] [--run-
↪on {Runner.LOCAL,Runner.SLURM}] [--skip-checks] solver-path
```

solver-path
: path to the solver

-h, --help
: show this help message and exit

--deterministic
: Flag indicating the solver is deterministic

--run-solver-now
: immediately run the solver(s) on all instances

--nickname <nickname>
: set a nickname for the solver

--run-on
: On which computer or cluster environment to execute the calculation.

--skip-checks
: Checks the solver's functionality by testing it on an instance and the pcs file, when applicable.

cancel

Command to cancel running jobs.

```
usage: cancel [-h] [--job-ids JOB_IDS [JOB_IDS ...]] [--all]
```

-h, --help
: show this help message and exit

--job-ids <job_ids>
: job ID(s) to use for the command

--all
: use all known job ID(s) for the command

cleanup

Command to clean files from the platform.

```
usage: cleanup [-h] [--all] [--remove]
```

-h, --help
: show this help message and exit

--all
: clean all output files

--remove

remove all files in the platform, including user data such as InstanceSets and Solvers

compute_features

Sparkle command to Compute features for instances using added extractors and instances.

```
usage: compute_features [-h] [--recompute] [--settings-file SETTINGS_FILE] [--run-on
↪{Runner.LOCAL,Runner.SLURM}]
```

-h, --help

show this help message and exit

--recompute

Re-run feature extractor for instances with previously computed features

--settings-file

Specify the settings file to use in case you want to use one other than the default

--run-on

On which computer or cluster environment to execute the calculation.

compute_marginal_contribution

Command to compute the marginal contribution of solvers to the portfolio.

```
usage: compute_marginal_contribution [-h] [--perfect] [--actual] [--objectives␣
↪OBJECTIVES] [--settings-file SETTINGS_FILE]
```

-h, --help

show this help message and exit

--perfect

compute the marginal contribution for the perfect selector

--actual

compute the marginal contribution for the actual selector

--objectives <objectives>

the comma seperated objective(s) to use.

--settings-file

Specify the settings file to use in case you want to use one other than the default

configure_solver

Configure a solver in the platform.

```
usage: configure_solver [-h] [--configurator CONFIGURATOR] --solver SOLVER --instance-
↪set-train INSTANCE_SET_TRAIN [--instance-set-test INSTANCE_SET_TEST] [--objectives␣
↪OBJECTIVES]
                        [--target-cutoff-time TARGET_CUTOFF_TIME] [--solver-calls SOLVER_
↪CALLS] [--number-of-runs NUMBER_OF_RUNS] [--settings-file SETTINGS_FILE] [--use-
↪features] [--validate] [--ablation]
                        [--run-on {Runner.LOCAL,Runner.SLURM}]
```

-h, --help
    show this help message and exit

--configurator <configurator>
    name of the configurator

--solver <solver>
    path to solver

--instance-set-train <instance_set_train>
    path to training instance set

--instance-set-test <instance_set_test>
    path to test instance set (only for validating)

--objectives <objectives>
    the comma seperated objective(s) to use.

--target-cutoff-time <target_cutoff_time>
    cutoff time per target algorithm run in seconds

--solver-calls <solver_calls>
    number of solver calls to execute

--number-of-runs <number_of_runs>
    number of configuration runs to execute

--settings-file
    Specify the settings file to use in case you want to use one other than the default

--use-features
    use the training set's features for configuration

--validate
    validate after configuration

--ablation
    run ablation after configuration

--run-on
    On which computer or cluster environment to execute the calculation.

Note that the test instance set is only used if the `--ablation` or `--validation` flags are given

construct_portfolio_selector

Command to construct a portfolio selector over all known features solver performances.

```
usage: construct_portfolio_selector [-h] [--recompute-portfolio-selector] [--selector-
↪timeout SELECTOR_TIMEOUT] [--objectives OBJECTIVES] [--solver-ablation] [--run-on
↪{Runner.LOCAL,Runner.SLURM}]
                                    [--settings-file SETTINGS_FILE]
```

-h, --help
show this help message and exit

--recompute-portfolio-selector
force the construction of a new portfolio selector even when it already exists for the current feature and performance data. NOTE: This will also result in the computation of the marginal contributions of solvers to the new portfolio selector.

--selector-timeout <selector_timeout>
Cuttoff time (in seconds) for the algorithmselector construction

--objectives <objectives>
the comma seperated objective(s) to use.

--solver-ablation
construct a selector for each solver ablation combination

--run-on
On which computer or cluster environment to execute the calculation.

--settings-file
Specify the settings file to use in case you want to use one other than the default

generate_report

Without any arguments a report for the most recent algorithm selection or algorithm configuration procedure is generated.

```
usage: generate_report [-h] [--solver SOLVER] [--instance-set-train INSTANCE_SET_TRAIN]␣
↪[--instance-set-test INSTANCE_SET_TEST] [--no-ablation [FLAG_ABLATION]] [--selection]
                       [--test-case-directory TEST_CASE_DIRECTORY] [--objectives␣
↪OBJECTIVES] [--settings-file SETTINGS_FILE] [--only-json ONLY_JSON]
```

-h, --help
show this help message and exit

--solver <solver>
path to solver for an algorithm configuration report

--instance-set-train <instance_set_train>
path to training instance set included in Sparkle for an algorithm configuration report

--instance-set-test <instance_set_test>
path to testing instance set included in Sparkle for an algorithm configuration report

--no-ablation <flag_ablation>
turn off reporting on ablation for an algorithm configuration report

--selection
set to generate a normal selection report

--test-case-directory <test_case_directory>
Path to test case directory of an instance set for a selection report

--objectives <objectives>
the comma seperated objective(s) to use.

--settings-file
Specify the settings file to use in case you want to use one other than the default

--only-json <only_json>
if set to True, only generate machine readable output

Note that if a test instance set is given, the training instance set must also be given.

initialise

Initialise the Sparkle platform in the current directory.

```
usage: initialise [-h] [--download-examples | --no-download-examples]
```

-h, --help
show this help message and exit

--download-examples, --no-download-examples
Download the Examples into the directory. (default: False)

load_snapshot

Load a platform from a zip file.

```
usage: load_snapshot [-h] snapshot-file-path
```

snapshot-file-path
path to the snapshot file

-h, --help
show this help message and exit

remove_feature_extractor

Remove a feature extractor from the platform.

```
usage: remove_feature_extractor [-h] extractor-path
```

extractor-path
path or nickname of the feature extractor

-h, --help

show this help message and exit

remove_instances

Remove instances from the platform.

```
usage: remove_instances [-h] instances-path
```

instances-path

path to or nickname of the instance set

-h, --help

show this help message and exit

remove_solver

Remove a solver from the platform.

```
usage: remove_solver [-h] solver
```

solver

name, path to or nickname of the solver

-h, --help

show this help message and exit

run_ablation

Runs parameter importance between the default and configured parameters with ablation. This command requires a finished configuration for the solver instance pair.

```
usage: run_ablation [-h] [--solver SOLVER] [--instance-set-train INSTANCE_SET_TRAIN] [--
↪instance-set-test INSTANCE_SET_TEST] [--objectives OBJECTIVES] [--target-cutoff-time␣
↪TARGET_CUTOFF_TIME]
                    [--wallclock-time WALLCLOCK_TIME] [--number-of-runs NUMBER_OF_RUNS]␣
↪[--racing RACING] [--settings-file SETTINGS_FILE] [--run-on {Runner.LOCAL,Runner.SLURM}
↪]
```

-h, --help

show this help message and exit

--solver <solver>

path to solver

--instance-set-train <instance_set_train>

path to training instance set

--instance-set-test <instance_set_test>

path to test instance set

--objectives <objectives>
: the comma seperated objective(s) to use.

--target-cutoff-time
: cutoff time per target algorithm run in seconds

--wallclock-time <wallclock_time>
: configuration budget per configurator run in seconds (wallclock)

--number-of-runs
: Number of configuration runs to execute

--racing
: Performs abaltion analysis with racing

--settings-file
: Specify the settings file to use in case you want to use one other than the default

--run-on
: On which computer or cluster environment to execute the calculation.

Note that if no test instance set is given, the validation is performed on the training set.

run_configured_solver

Command to run a configured solver on an instance (set).

```
usage: run_configured_solver [-h] [--settings-file SETTINGS_FILE] [--objectives␣
↪OBJECTIVES] [--run-on {Runner.LOCAL,Runner.SLURM}] instance_path
```

instance_path
: Path to an instance (set)

-h, --help
: show this help message and exit

--settings-file
: Specify the settings file to use in case you want to use one other than the default

--objectives <objectives>
: the comma seperated objective(s) to use.

--run-on
: On which computer or cluster environment to execute the calculation.

run_parallel_portfolio

Run a portfolio of solvers on an instance set in parallel.

```
usage: run_parallel_portfolio [-h] --instance-path INSTANCE_PATH [--portfolio-name␣
↪PORTFOLIO_NAME] [--solvers SOLVERS [SOLVERS ...]] [--objectives OBJECTIVES] [--cutoff-
↪time CUTOFF_TIME]
                              [--solver-seeds SOLVER_SEEDS] [--run-on {Runner.LOCAL,
↪Runner.SLURM}] [--settings-file SETTINGS_FILE]
```

-h, --help
: show this help message and exit

--instance-path <instance_path>
: Path to an instance (set)

--portfolio-name <portfolio_name>
: Specify a name of the portfolio. If none is given, one will be generated.

--solvers <solvers>
: Specify the list of solvers to be used. If not specifed, all solvers known in Sparkle will be used.

--objectives <objectives>
: the comma seperated objective(s) to use.

--cutoff-time <cutoff_time>
: The duration the portfolio will run before the solvers within the portfolio will be stopped (default: 60)

--solver-seeds <solver_seeds>
: number of random seeds per solver to execute

--run-on
: On which computer or cluster environment to execute the calculation.

--settings-file
: Specify the settings file to use in case you want to use one other than the default

run_portfolio_selector

Run a portfolio selector on instance (set), determine which solver is most likely to perform well and run it on the instance (set).

```
usage: run_portfolio_selector [-h] [--run-on {Runner.LOCAL,Runner.SLURM}] [--settings-
↪file SETTINGS_FILE] [--objectives OBJECTIVES] instance_path
```

instance_path
: Path to an instance (set)

-h, --help
: show this help message and exit

--run-on
: On which computer or cluster environment to execute the calculation.

--settings-file
: Specify the settings file to use in case you want to use one other than the default

--objectives <objectives>
: the comma seperated objective(s) to use.

run_solvers

Run all solvers on all instances to get their performance data.

```
usage: run_solvers [-h] [--recompute] [--objectives OBJECTIVES] [--target-cutoff-time␣
↪TARGET_CUTOFF_TIME] [--also-construct-selector-and-report] [--run-on {Runner.LOCAL,
↪Runner.SLURM}]
                   [--settings-file SETTINGS_FILE]
```

-h, --help
show this help message and exit

--recompute
recompute the performance of all solvers on all instances

--objectives <objectives>
the comma seperated objective(s) to use.

--target-cutoff-time <target_cutoff_time>
cutoff time per target algorithm run in seconds

--also-construct-selector-and-report
after running the solvers also construct the selector and generate the report

--run-on
On which computer or cluster environment to execute the calculation.

--settings-file
Specify the settings file to use in case you want to use one other than the default

save_snapshot

Save the current platform in a .zip file.

```
usage: save_snapshot [-h] [--name NAME]
```

-h, --help
show this help message and exit

--name <name>
name of the snapshot

Can be loaded later with the load snapshot command.

status

Display the status of the platform.

```
usage: status [-h] [--verbose]
```

-h, --help
show this help message and exit

--verbose, -v
output status in verbose mode

validate_configured_vs_default

Test the performance of the configured solver and the default solver by doing validation experiments on the training and test sets.

```
usage: validate_configured_vs_default [-h] --solver SOLVER --instance-set-train INSTANCE_
↪SET_TRAIN [--instance-set-test INSTANCE_SET_TEST] [--configurator CONFIGURATOR] [--
↪objectives OBJECTIVES]
                                      [--target-cutoff-time TARGET_CUTOFF_TIME] [--
↪settings-file SETTINGS_FILE] [--run-on {Runner.LOCAL,Runner.SLURM}]
```

-h, --help
: show this help message and exit

--solver <solver>
: path to solver

--instance-set-train <instance_set_train>
: path to training instance set

--instance-set-test <instance_set_test>
: path to test instance set (only for validating)

--configurator <configurator>
: name of the configurator

--objectives <objectives>
: the comma seperated objective(s) to use.

--target-cutoff-time
: cutoff time per target algorithm run in seconds

--settings-file
: Specify the settings file to use in case you want to use one other than the default

--run-on
: On which computer or cluster environment to execute the calculation.

wait

Wait for async jobs to finish. Gives periodic updates in table format about each job.

```
usage: wait [-h] [--job-ids JOB_IDS [JOB_IDS ...]]
```

-h, --help
: show this help message and exit

--job-ids <job_ids>
: job ID(s) to use for the command

1.7.5 Platform

File structure

The platform automatically generates a file structure for both input and output upon initialisation.

Instance directory

The instance directory has the following structure:

```
Instances/
  Example_Instance_Set/
    instance_a.cnf
    instance_b.cnf
    ...        ...
    instance_z.cnf
```

Each directory under the Instances directory represents an `Instance Set` and each file is considered an instance. Note that if your dataset is a single file, it will be considered a single instance in the set.

For instances consisting of multiple files one additional file called `instances.csv` should be included in the `Example_Instance_Set` directory, describing which files together form an instance. The format is a single instance per line with each file separated by a space, as shown below.

```
instance_name_a instance_a_part_one.abc ... instance_a_part_n.xyz
instance_name_b instance_b_part_one.abc ... instance_b_part_n.xyz
...                     ...
instance_name_z instance_z_part_one.abc ... instance_z_part_n.xyz
```

Solver Directory

The solver directory has the following structure:

```
Solver/
  Example_Solver/
    sparkle_solver_wrapper.py
    parameters.pcs
    ...
```

The `sparkle_solver_wrapper.py` is a wrapper that Sparkle should call to run the solver with specific settings, and then returns a result for the configurator. In `parameters.pcs` the configurable parameters are described in the PCS format. Finally, when importing your Solver into Sparkle, a binary executable of the runsolver tool `runsolver` is added. This allows Sparkle to make fair time and computational cost measurements for all configuration experiments.

This same structure holds up for all other executables we refer to as `SparkleCallable` in the Sparkle package, such as Feature Extractors, which are placed in the `Extractor` directory.

The output directory

The output directory is located at the root of the Sparkle directory. Its structure is as follows:

```
Output/
  Logs/
    commandname_timestamp/
        log files
  Configuration/
    configurator/
        Raw_Data/
            configuration_scenario/
                related files
    Analysis/
  Parallel_Portfolio/
    Raw_Data/
        related files
    Analysis/
  Selection/
    selector/
        solver_scenario/
            related files
    Analysis/
```

The `Logs` directory should contain the history of commands and their output such that one can easily know what has been done in which order and find enough pointers to debug unwanted behaviour.

Other directories are cut into two subdirectories: `Raw_Data` contains the data produced by the main command, often time consuming to generate, handle with care; `Analysis` contains information extracted from the raw data, easy to generate, plots and reports.

For each type of task run by Sparkle, the `related files` differ. The aim is always to have all required files for reproducibility. A copy of the sparkle configuration file at the time of the run and of all files relevant to the run, a copy of any log or error file that could help with debugging or a link to it, and the output of the executed task.

For configuration the configuration trajectory if available, the training and testing sets, the default configuration and the final found configuration. The performance of those will be in the Analysis folder.

For parallel portfolio the resulting portfolio and its components. The performance of the portfolio will be in the Analysis folder.

For selection the algorithms and their performance on the training set, the model(s) generated if available and the resulting selector. The performance evaluation of the selector will be in the Analysis folder.

For analysis a link to the folder on which the analysis was performed (configuration, portfolio or selection), the performance evaluation from it and the report if it was generated.

Other directories

There are a few other special directories automatically generated by Sparkle.

- **Reference_Lists**: Here Sparkle keeps track of user-defined aliases
- **Snapshots**: Here Sparkle places your saved snapshots
- **Tmp**: Here temporary files are placed that are generated during commands, but should also be removed during the command
- Output/**Feature_Data**: Here Sparkle unifies all known/added Feature Extractors, the Instances and their features if calculated. When an extractor or instance is removed, they are also removed here.
- Output/**Performance_Data**: Here Sparkle unifies all known/added Solvers, the Instances and their recorded objectives if known. When a solver or instance is removed, they are also removed here.

Platform Settings

Most settings can be controlled through the `Settings` directory, specifically the `Settings/sparkle_settings.ini` file. Possible settings are summarised per category in *Options and possible values*. For any settings that are not provided the defaults will be used. Meaning, in the extreme case, that if the settings file is empty (and nothing is set through the command line) everything will run with default values.

For convenience after every command `Settings/latest.ini` is written with the used settings. Here any overrides by commandline arguments are reflected. This can, for instance, provide the same settings to the next command in a chain. E.g. for `validate_configured_vs_default` after `configure_solver`. The used settings are also recorded in the relevant `Output/` subdirectory. Note that when writing settings Sparkle always uses the name, and not an alias.

Note

When overriding settings in `sparkle_settings.ini` with the commandline arguments, this is considered as 'temporary' and only denoted in the latest_settings, but does not actually affect the values in sparkle_settings.ini

Example `sparkle_settings.ini`

This is a short example to show the format.

```
[general]
objective = RUNTIME
target_cutoff_time = 60

[configuration]
number_of_runs = 25

[slurm]
number_of_runs_in_parallel = 25
```

When initialising a new platform, the user is provided with a default settings.ini, which can be viewed here.

Sparkle Objectives

To define an objective for your algorithms, you can define them in the `general` section of your `Settings.ini` like the following:

```
[general]
objective = PAR10,loss,accuracy:max
```

In the above example we have defined three objectives: Penalised Average Runtime, the loss function value of our algorithm on the task, and the accuracy of our algorithm on the task. Note that objectives are by default assumed to be *minimised* and we must therefore specify `accuracy:max` to clarifiy this. The platform predefines for the user three objectives: cpu time, wallclock time and memory. These objectives will always recorded next to whatever the user may choose.

Note

Although the Platform supports multiple objectives to be registered for any Solver, not all used components, such as SMAC and Ablation Analysis, support Multi-Objective optimisation. In any such case, the first defined objective is considered the most important and used in these situations

Moreover, when aggregating an objective over various dimensions, Sparkle assumes the following:

- When aggregating multiple Solvers (Algorithms), we aggregate by taking the minimum/maximum value.
- When aggregating multiple runs on the same instances, we aggregate by taking the mean.
- When aggregating multiple instances, we aggregate by taking the mean.

It is possible to redefine these attributes for your specific objective. The platform looks for a file called `objective.py` in your Settings directory of the platform, and reads your own object definitions. These definitions can either add new objectives to the platform, but also can overwrite existing definitions in the library. E.g. when creating an objective definition with the same name of one that already exists in the library, the user definiton simply overrules the library definition. Note that there are a few constraints and details:

- The objective must inherit from the `SparkleObjective` class
- The classnames are constrained to the format of alphabetical letters followed by numericals
- The objective can be parametrised by an integer, such as `PAR` followed by `10` is interpreted as instantiating the `PAR` class with argument `10`
- If your objective is defined over time, you can indicate this using the `UseTime` enum, see the *types module*

Slurm

Slurm settings can be specified in the `Settings/settings.ini` file. Any setting in the Slurm section not internally recognised by Sparkle will be added to the `sbatch` or `srun` calls. It is advised to overwrite the default settings specific to your cluster, such as the option "–partition" with a valid value on your cluster. Also, you might have to adapt the default "–mem-per-cpu" value to your system. For example, your Slurm section in the `settings.ini` could look like:

```
[slurm]
partition = CPU
mem-per-cpu = 6000
...
time = 25:00
```

Discouraged options Currently these settings are inserted *as is* in any Slurm calls done by Sparkle. This means that any options exclusive to one or the other currently should not be used. The options below are exclusive to `sbatch` and are thus discouraged:

- `--array`
- `--clusters`
- `--wrap`

The options below are exclusive to `srun` and are thus discouraged:

- `--label`

Options and possible values

[general]

`objective`

aliases: `objective`

values: `str`, comma seperated for multiple

description: The type of objectives Sparkle considers, see *Sparkle Objective section* for more.

`configurator`

aliases: `configurator`

values: `SMAC2`

description: The name of the Configurator class implementation to use. Currently only supports SMAC2.

`selector`

aliases: `selector`

values: Path.

Note: Currently only AutoFolio is supported by Sparkle. This setting is soon to be deprecated for a more flexible solution.

Description: The Algorithm selector to use.

`solution_verifier`

aliases: N/A

values: `{NONE, SAT}`

note: Only available for SAT solving.

`target_cutoff_time`

aliases: `cutoff_time_each_solver_call`

values: integer

description: The time a solver is allowed to run before it is terminated.

`extractor_cutoff_time`

aliases: `cutoff_time_each_feature_computation`

values: integer

description: The time a feature extractor is allowed to run before it is terminated. In case of multiple feature extractors this budget is divided equally.

`run_on`

aliases: `run_on`

values: `LOCAL`, `SLURM`

description: On which compute to run the jobs on.

`verbosity`

aliases: `verbosity`

values: `QUIET`, `STANDARD`

description: The verbosity level of Sparkle when running CLI.

`check_interval`

aliases: `check_interval`

values: int

description: Specifically for the Wait command. The amount of seconds to wait in between refreshing the wait information.

[configuration]

`wallclock_time`

aliases: `wallclock_time`

values: integer

description: The wallclock time one configuration run is allowed to use for finding configurations.

`cpu_time`

aliases: `cpu_time`

values: integer

description: The cpu time one configuration run is allowed to use for finding configurations.

`solver_calls`

aliases: `solver_calls`

values: integer

description: The number of solver calls one configuration run is allowed to use for finding configurations.

`number_of_runs`

aliases: `number_of_runs`

values: integer

description: The number of separate configurations runs.

`target_cutoff_length`

aliases: `smac_each_run_cutoff_length`

values: `{max}` (other values: whatever is allowed by SMAC)

[slurm]

`number_of_jobs_in_parallel`

aliases: `num_job_in_parallel`

values: integer

description: The number of jobs runs that can run in parallel.

`max_parallel_runs_per_node`

aliases: `clis_per_node`

values: integer

description: The number of parallel processes that can be run on one compute node. In case a node has 32 cores and each solver uses 2 cores, the `max_parallel_runs_per_node` is at most 16.

[ablation]

`racing`

aliases: `ablation_racing`

values: boolean

description: Use racing when performing the ablation analysis between the default and configured parameters

---

[parallel_portfolio]

`check_interval`

aliases: `check_interval`

values: int

description: How many seconds the parallel portfolio waits to check whether jobs have completed. Decreasing the amount increases the accuracy of the report but also significantly increases computational load.

`num_seeds_per_solver`

aliases: `num_seeds_per_solver`

values: int

description: Only relevant for undeterministic solvers. The amount of solvers that will be started with a random seed.

Priorities

Sparkle has a large flexibility with passing along settings. Settings provided through different channels have different priorities as follows:

- Default - Default values will be overwritten if a value is given through any other mechanism;
- File – Settings form the `Settings/sparkle_settings.ini` overwrite default values, but are overwritten by settings given through the command line;
- Command line Settings file – Settings files provided through the command line, overwrite default values and other settings files.
- Command line - Settings given through the command line overwrite all other settings, including settings files provided through the command line.
- Configurators - Each configurator has its own option section and these values will take precedence of any value set in the general configurator section.

Reporting packages

The platform depends on the following user supplied packages to generate its reports:

- `pdflatex`
- `latex`
- `bibtex`

1.7.6 Configurators

Sparkle offers several configurators to use for Algorithm Configuration. Although they come with automatic installations and various default settings, you might need to set up a few details for your specific system and algorithm or data set to make sure everything works as intended.

SMAC2

Sequential Model-Based Optimization for General Algorithm Configuration*[1]*, or *SMAC* for short is a Java based algorithm configurator. *Note that this the second version, and not SMAC3 the Python version*. The original documentation of the configurator can be found here.

> **Note**
>
> SMAC2 is written in Java and therefore requires Java to be installed in your environment. The current tested version in Sparkle is 1.8.0_402

Budget

SMAC2 receives it budget in terms of `solver_calls`, which specify the maximum amount of times the target solver (e.g. your algorithm) may be run on a certain instance, or through `cpu_time` or `wallclock_time`. Note that in the case of using time as a budget, not only the solver time measurement is used for the budget but also that of SMAC itself. If you want only the execution time of the algorithm to be used for the budget, set `use_cpu_time_in_tunertime` to `False`.

IRACE

Iterated Racing for Automatic Algorithm Configuration*[2]*, or IRACE for short is an R based algorithm configurator. The full documentation of the configurator can be found here.

IRACE offers many parameters that can be set, but also automatically computed in accordance with their paper*[2]* and we recommend not deviating from those formulae as it may result in unexpected behaviour.

> **Note**
>
> IRACE is written in R and therefore requires R to be installed in your environment. The current tested version in Sparkle is R 4.3.1

Budget

In order to set a budget, IRACE offers two mutually exclusive parameters: `MaxExperiments` (Also known in Sparkle as `solver_calls`) and `MaxTime`.

> **Note**
>
> Since these two budgets are mutually exclusive, IRACE won't start if both are set. To avoid this, Sparkle will default to MaxExperiments.

MaxExperiments

The `MaxExperiments` parameter can be set through the `irace` section of the `sparkle_settings.ini` with the key word `max_experiments`. The user can fill in any value, but do note that IRACE pre-computes a minimum budget at runtime. Thus, setting a too low budget can cause the configurator to immediatly exit with an error.

> **Note**
>
> This parameter can also be set through `solver_calls` in the configuration section, but this value will be ignored if the IRACE section specifies either `max_time` or `max_experiments`

MaxTime

The MaxTime parameter specifies the budget in terms of maximum runtime of the target algorithm (e.g. your Solver in the Sparkle Platform). Sparkle measures the time spend of your solver using RunSolver, and passes the **CPU** time to IRACE to determine its spend budget. IRACE also tries to determine how much budget it has for the first run using the `budgetEstimation` parameter, which is by default set to 2%. Sparkle will attempt to recompute this based on `target_cutoff_time` (The time limit of your Solver in each call) and the `max_time` budget as `target_cutoff_time` / `max_time`, but only if the fraction is less than 1.0. **Note** that IRACE differs from SMAC2 in its time usage calculations, as it does not include the time used by IRACE itself to determine how much budget is left. See the SMAC2 section on how this can be changed.

References

[1] Sequential Model-Based Optimization for General Algorithm Configuration F. Hutter and H. H. Hoos and K. Leyton-Brown (2011) Proc.~of LION-5, 2011, p507–523

[2] The irace package: Iterated Racing for Automatic Algorithm Configuration, Manuel López-Ibáñez and Jérémie Dubois-Lacoste and Leslie Pérez Cáceres and Thomas Stützle and Mauro Birattari (2016) Operations Research Perspectives, Volume 3, p43–58

1.7.7 about

Helper module for information about Sparkle.

1.7.8 configurator

This package provides configurator support for Sparkle.

1.7.9 instance

This package provides instance set support for Sparkle.

class `sparkle.instance.`**`FileInstanceSet`**(*target: Path*)

Object representation of a set of single-file instances.

property name: str

Get instance set name.

class `sparkle.instance.`**`InstanceSet`**(*target: Path | list[str, Path]*)

Base object representation of a set of instances.

property all_paths: list[Path]

Returns all file paths in the instance set as a flat list.

get_path_by_name(*name: str*) → Path | list[Path]

Retrieves an instance paths by its name. Returns None upon failure.

property instance_names: list[str]

Get processed instance names for multi-file instances.

property instance_paths: list[Path]

Get processed instance paths.

property name: str

Get instance set name.

property size: int

Returns the number of instances in the set.

class sparkle.instance.**IterableFileInstanceSet**(*target: Path*)

Object representation of files containing multiple instances.

property size: int

Returns the number of instances in the set.

class sparkle.instance.**MultiFileInstanceSet**(*target: Path | list[str, Path]*)

Object representation of a set of multi-file instances.

property all_paths: list[Path]

Returns all file paths in the instance set as a flat list.

1.7.10 platform

This package provides platform support for Sparkle.

class sparkle.platform.**CommandName**(*value*)

Enum of all command names.

class sparkle.platform.**SettingState**(*value*)

Enum of possible setting states.

class sparkle.platform.**Settings**(*file_path: PurePath | None = None*)

Class to read, write, set, and get settings.

add_slurm_extra_option(*name: str*, *value: str*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Add additional Slurm options.

static check_settings_changes(*cur_settings:* Settings, *prev_settings:* Settings) → bool

Check if there are changes between the previous and the current settings.

Prints any section changes, printing None if no setting was found.

Args:

cur_settings: The current settings prev_settings: The previous settings

Returns:

True iff there are no changes.

`get_ablation_racing_flag()` → bool

Return a bool indicating whether the racing flag is set for ablation.

`get_configurator_max_iterations()` → int | None

Get the maximum number of configurator iterations.

`get_configurator_number_of_runs()` → int

Return the number of configuration runs.

`get_configurator_settings(`*configurator_name: str*`)` → dict[str, any]

Return the configurator settings.

`get_configurator_solver_calls()` → int | None

Return the maximum number of solver calls the configurator can do.

`get_general_check_interval()` → int

Return the general check interval.

`get_general_extractor_cutoff_time()` → int

Return the cutoff time in seconds for feature extraction.

`get_general_solution_verifier()` → object

Return the solution verifier to use.

`get_general_sparkle_configurator()` → Configurator

Return the configurator init method.

`get_general_sparkle_objectives()` → list[*SparkleObjective*]

Return the performance measure.

`get_general_sparkle_selector()` → *Selector*

Return the selector init method.

`get_general_target_cutoff_time()` → int

Return the cutoff time in seconds for target algorithms.

`get_general_verbosity()` → VerbosityLevel

Return the general verbosity.

`get_irace_first_test()` → int | None

Return the first test for IRACE.

Specifies how many instances are evaluated before the first elimination test. IRACE Default: 5. [firstTest]

`get_irace_max_experiments()` → int

Return the max number of experiments for IRACE.

`get_irace_max_iterations()` → int

Return the number of iterations for IRACE.

`get_irace_max_time()` → int

Return the max time in seconds for IRACE.

`get_irace_mu()` → int | None

Return the mu for IRACE.

Parameter used to define the number of configurations sampled and evaluated at each iteration. IRACE Default: 5. [mu]

`get_number_of_jobs_in_parallel()` → int
: Return the number of runs Sparkle can do in parallel.

`get_parallel_portfolio_check_interval()` → int
: Return the parallel portfolio check interval.

`get_parallel_portfolio_number_of_seeds_per_solver()` → int
: Return the parallel portfolio seeds per solver to start.

`get_run_on()` → Runner
: Return the compute on which to run.

`get_slurm_extra_options(`*as_args: bool = False*`)` → dict | list
: Return a dict with additional Slurm options.

`get_slurm_max_parallel_runs_per_node()` → int
: Return the number of algorithms Slurm can run in parallel per node.

`get_smac2_cpu_time()` → int | None
: Return the budget per configuration run in seconds (cpu).

`get_smac2_max_iterations()` → int | None
: Get the maximum number of SMAC2 iterations.

`get_smac2_target_cutoff_length()` → str
: Return the target algorithm cutoff length.

  'A domain specific measure of when the algorithm should consider itself done.'

  Returns:
  The target algorithm cutoff length.

`get_smac2_use_cpu_time_in_tunertime()` → bool
: Return whether to use CPU time in tunertime.

`get_smac2_wallclock_time()` → int | None
: Return the budget per configuration run in seconds (wallclock).

`read_settings_ini(`*file_path: PurePath = PurePosixPath('Settings/sparkle_settings.ini')*, *state:* SettingState *= SettingState.FILE*`)` → None
: Read the settings from an INI file.

`set_ablation_racing_flag(`*value: bool = False*, *origin:* SettingState = *SettingState.DEFAULT*`)` → None
: Set a flag indicating whether racing should be used for ablation.

`set_configurator_max_iterations(`*value: int | None = None*, *origin:* SettingState = *SettingState.DEFAULT*`)` → None
: Set the number of configuration runs.

`set_configurator_number_of_runs(`*value: int = 25*, *origin:* SettingState = *SettingState.DEFAULT*`)` → None
: Set the number of configuration runs.

`set_configurator_solver_calls(`*value: int = 100*, *origin:* SettingState = *SettingState.DEFAULT*`)` → None
: Set the number of solver calls.

set_general_check_interval(*value: int = 10*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the general check interval.

set_general_extractor_cutoff_time(*value: int = 60*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the cutoff time in seconds for feature extraction.

set_general_solution_verifier(*value: str = 'None'*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the solution verifier to use.

set_general_sparkle_configurator(*value: str = 'SMAC2'*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the Sparkle configurator.

set_general_sparkle_objectives(*value: list[~sparkle.types.objective.SparkleObjective] = [<sparkle.types.objective.PAR object>], origin: ~sparkle.platform.settings_objects.SettingState = SettingState.DEFAULT*) → None

Set the sparkle objective.

set_general_sparkle_selector(*value: Path = PosixPath('/home/snelleman/Sparkle/sparkle/Components/AutoFolio/scripts/autofolio')*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the Sparkle selector.

set_general_target_cutoff_time(*value: int = 60*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the cutoff time in seconds for target algorithms.

set_general_verbosity(*value: VerbosityLevel = VerbosityLevel.STANDARD*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the general verbosity to use.

set_irace_first_test(*value: int | None = None*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the first test for IRACE.

set_irace_max_experiments(*value: int = 0*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the max number of experiments for IRACE.

set_irace_max_iterations(*value: int | None = None*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the number of iterations for IRACE.

Maximum number of iterations to be executed. Each iteration involves the generation of new configurations and the use of racing to select the best configurations. By default (with 0), irace calculates a minimum number of iterations as N^iter = 2 + log2 N param, where N^param is the number of non-fixed parameters to be tuned. Setting this parameter may make irace stop sooner than it should without using all the available budget. IRACE recommends to use the default value (Empty).

set_irace_max_time(*value: int = 0*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the max time in seconds for IRACE.

set_irace_mu(*value: int | None = None*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the mu for IRACE.

set_number_of_jobs_in_parallel(*value: int = 25*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the number of runs Sparkle can do in parallel.

set_parallel_portfolio_check_interval(*value: int = 4*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the parallel portfolio check interval.

set_parallel_portfolio_number_of_seeds_per_solver(*value: int = 1*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the parallel portfolio seeds per solver to start.

set_run_on(*value: Runner = 'local'*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the compute on which to run.

set_slurm_max_parallel_runs_per_node(*value: int = 8*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the number of algorithms Slurm can run in parallel per node.

set_smac2_cpu_time(*value: int | None = None*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the budget per configuration run in seconds (cpu).

set_smac2_max_iterations(*value: int | None = None*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the maximum number of SMAC2 iterations.

set_smac2_target_cutoff_length(*value: str = 'max'*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the target algorithm cutoff length.

set_smac2_use_cpu_time_in_tunertime(*value: bool | None = None*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set whether to use CPU time in tunertime.

set_smac2_wallclock_time(*value: int | None = None*, *origin:* SettingState = *SettingState.DEFAULT*) → None

Set the budget per configuration run in seconds (wallclock).

write_settings_ini(*file_path: Path*) → None

Write the settings to an INI file.

write_used_settings() → None

Write the used settings to the default locations.

1.7.11 solver

This package provides solver support for Sparkle.

class sparkle.solver.**Extractor**(*directory: Path*, *runsolver_exec: Path | None = None*, *raw_output_directory: Path | None = None*)

Extractor base class for extracting features from instances.

build_cmd(*instance: Path | list[Path]*, *feature_group: str | None = None*, *output_file: Path | None = None*, *cutoff_time: int | None = None*, *log_dir: Path | None = None*) → list[str]

Builds a command line string seperated by space.

Args:

instance: The instance to run on feature_group: The optional feature group to run the extractor for. outputfile: Optional file to write the output to. runsolver_args: The arguments for runsolver. If not present,

will run the extractor without runsolver.

Returns:

The command seperated per item in the list.

property feature_groups: list[str]

Returns the various feature groups the Extractor has.

property features: list[tuple[str, str]]

Determines the features of the extractor.

get_feature_vector(*result: Path*, *runsolver_values: Path | None = None*) → list[str]

Extracts feature vector from an output file.

Args:

result: The raw output of the extractor runsolver_values: The output of runsolver.

Returns:

A list of features. Vector of missing values upon failure.

property groupwise_computation: bool

Determines if you can call the extractor per group for parallelisation.

property output_dimension: int

The size of the output vector of the extractor.

run(*instance: Path | list[Path]*, *feature_group: str | None = None*, *output_file: Path | None = None*, *cutoff_time: int | None = None*, *log_dir: Path | None = None*) → list | None

Runs an extractor job with Runrunner.

Args:

extractor_path: Path to the executable instance: Path to the instance to run on feature_group: The feature group to compute. Must be supported by the

extractor to use.

output_file: Target output. If None, piped to the RunRunner job. cutoff_time: CPU cutoff time in seconds log_dir: Directory to write logs. Defaults to self.raw_output_directory.

Returns:

The features or None if an output file is used, or features can not be found.

class sparkle.solver.**SATVerifier**

Class to handle the SAT verifier.

static sat_get_verify_string(*sat_output: str*) → *SolverStatus*

Return the status of the SAT verifier.

Four statuses are possible: "SAT", "UNSAT", "WRONG", "UNKNOWN"

static sat_judge_correctness_raw_result(*instance: Path*, *raw_result: Path*) → *SolverStatus*

Run a SAT verifier to determine correctness of a result.

Args:

instance: path to the instance raw_result: path to the result to verify

Returns:
The status of the solver on the instance

verify(*instance: Path*, *raw_result: Path*) → *SolverStatus*

Run a SAT verifier and return its status.

class sparkle.solver.**Selector**(*executable_path: Path*, *raw_output_directory: Path*)

The Selector class for handling Algorithm Selection.

build_cmd(*selector_path: Path*, *feature_vector: list | str*) → list[str | Path]

Builds the commandline call string for running the Selector.

build_construction_cmd(*target_file: Path*, *performance_data: Path*, *feature_data: Path*, *objective:* SparkleObjective, *runtime_cutoff: int | float | str | None = None*, *wallclock_limit: int | float | str | None = None*) → list[str | Path]

Builds the commandline call string for constructing the Selector.

Args:
target_file: Path to the file to save the Selector to. performance_data: Path to the performance data csv. feature_data: Path to the feature data csv. objective: The objective to optimize for selection. runtime_cutoff: Cutoff for the runtime in seconds. Defaults to None wallclock_limit: Cutoff for total wallclock in seconds. Defaults to None

Returns:
The command list for constructing the Selector.

construct(*target_file: Path | str*, *performance_data:* PerformanceDataFrame, *feature_data:* FeatureDataFrame, *objective:* SparkleObjective, *runtime_cutoff: int | float | str | None = None*, *wallclock_limit: int | float | str | None = None*, *run_on: Runner = Runner.SLURM*, *sbatch_options: list[str] | None = None*, *base_dir: Path = PosixPath('.')*) → Run

Construct the Selector.

Args:
target_file: Path to the file to save the Selector to. performance_data: Path to the performance data csv. feature_data: Path to the feature data csv. objective: The objective to optimize for selection. runtime_cutoff: Cutoff for the runtime in seconds. wallclock_limit: Cutoff for the wallclock time in seconds. run_on: Which runner to use. Defaults to slurm. sbatch_options: Additional options to pass to sbatch. base_dir: The base directory to run the Selector in.

Returns:
Path to the constructed Selector.

static **process_predict_schedule_output**(*output: str*) → list

Return the predicted algorithm schedule as a list.

run(*selector_path: Path*, *feature_vector: list | str*) → list

Run the Selector, returning the prediction schedule upon success.

class sparkle.solver.**SolutionVerifier**

Solution verifier base class.

verifiy() → *SolverStatus*

Verify the solution.

class sparkle.solver.**Solver**(*directory: Path*, *raw_output_directory: Path | None = None*, *runsolver_exec: Path | None = None*, *deterministic: bool | None = None*, *verifier:* SolutionVerifier | *None = None*)

Class to handle a solver and its directories.

build_cmd(*instance: str | list[str]*, *objectives: list[*SparkleObjective*]*, *seed: int*, *cutoff_time: int | None = None*, *configuration: dict | None = None*, *log_dir: Path | None = None*) → list[str]

Build the solver call on an instance with a configuration.

Args:
instance: Path to the instance. seed: Seed of the solver. cutoff_time: Cutoff time for the solver. configuration: Configuration of the solver.

Returns:
List of commands and arguments to execute the solver.

static config_str_to_dict(*config_str: str*) → dict[str, str]

Parse a configuration string to a dictionary.

get_forbidden(*port_type: PCSConvention*) → Path

Get the path to the file containing forbidden parameter combinations.

get_pcs() → dict[str, tuple[str, str, str]]

Get the parameter content of the PCS file.

get_pcs_file(*port_type: str | None = None*) → Path

Get path of the parameter file.

Returns:
Path to the parameter file. None if it can not be resolved.

static parse_solver_output(*solver_output: str*, *runsolver_configuration: list[str | Path] | None = None*) → dict[str, Any]

Parse the output of the solver.

Args:
solver_output: The output of the solver run which needs to be parsed runsolver_configuration: The runsolver configuration to wrap the solver

with. If runsolver was not used this should be None.

Returns:
Dictionary representing the parsed solver output

port_pcs(*port_type: PCSConvention*) → None

Port the parameter file to the given port type.

read_pcs_file() → bool

Checks if the pcs file can be read.

run(*instance: str | list[str] |* InstanceSet, *objectives: list[*SparkleObjective*]*, *seed: int*, *cutoff_time: int | None = None*, *configuration: dict | None = None*, *run_on: Runner = Runner.LOCAL*, *commandname: str = 'run_solver'*, *sbatch_options: list[str] | None = None*, *log_dir: Path | None = None*) → SlurmRun | list[dict[str, Any]] | dict[str, Any]

Run the solver on an instance with a certain configuration.

Args:

instance: The instance(s) to run the solver on, list in case of multi-file.
In case of an instance set, will run on all instances in the set.

seed: Seed to run the solver with. Fill with abitrary int in case of
determnistic solver.

cutoff_time: The cutoff time for the solver, measured through RunSolver.
If None, will be executed without RunSolver.

configuration: The solver configuration to use. Can be empty. log_dir: Path where to place output files. Defaults to

self.raw_output_directory.

Returns:
Solver output dict possibly with runsolver values.

`class sparkle.solver.Validator`(*out_dir: Path = PosixPath('.')*, *tmp_out_dir: Path = PosixPath('.')*)

Class to handle the validation of solvers on instance sets.

`append_entry_to_csv`(*solver: str*, *config_str: str*, *instance_set:* InstanceSet, *instance: str*, *solver_output: dict*, *subdir: Path | None = None*) → None

Append a validation result as a row to a CSV file.

`get_validation_results`(*solver:* Solver, *instance_set:* InstanceSet, *source_dir: Path | None = None*, *subdir: Path | None = None*, *config: str | None = None*) → list[list[str]]

Query the results of the validation of solver on instance_set.

Args:
solver: Solver object instance_set: Instance set source_dir: Path where to look for any unprocessed output.

By default, look in the solver's tmp dir.

subdir: Path where to place the .csv file subdir. By default will be
'self.outputdir/solver.name_instanceset.name/validation.csv'

config: Path to the configuration if the solver was configured, None
otherwise

Returns
A list of row lists with string values

`retrieve_raw_results`(*solver:* Solver, *instance_sets:* InstanceSet | *list[*InstanceSet*]*, *subdir: Path | None = None*, *log_dir: Path | None = None*) → None

Checks the raw results of a given solver for a specific instance_set.

Writes the raw results to a unified CSV file for the resolve/instance_set combination.

Args:
solver: The solver for which to check the raw result path instance_sets: The set of instances for which to retrieve the results subdir: Subdir where the CSV is to be placed, passed to the append method. log_dir: The directory to search for log files. If none, defaults to

the log directory of the Solver.

`validate`(*solvers: list[Path]* | *list[*Solver*]* | Solver | *Path*, *configurations: list[dict]* | *dict* | *Path*, *instance_sets: list[*InstanceSet*]*, *objectives: list[*SparkleObjective*]*, *cut_off: int*, *subdir: Path | None = None*, *dependency: list[Run] | Run | None = None*, *sbatch_options: list[str] = []*, *run_on: Runner = Runner.SLURM*) → Run

Validate a list of solvers (with configurations) on a set of instances.

Args:
solvers: list of solvers to validate configurations: list of configurations for each solver we validate.

If a path is supplied, will use each line as a configuration.

instance_sets: set of instance sets on which we want to validate each solver objectives: list of objectives to validate cut_off: maximum run time for the solver per instance subdir: The subdir where to place the output in the outputdir. If None, a semi-unique combination of solver_instanceset is created.

dependency: Jobs to wait for before executing the validation. sbatch_options: list of slurm batch options run_on: whether to run on SLURM or local

1.7.12 structures

This package provides Sparkle's wrappers for Pandas DataFrames.

class sparkle.structures.**FeatureDataFrame**(*csv_filepath: Path*, *instances: list[str] = []*, *extractor_data: dict[str, list[tuple[str, str]]] = {}*)

Class to manage feature data CSV files and common operations on them.

add_extractor(*extractor: str*, *extractor_features: list[tuple[str, str]]*, *values: list[list[float]] | None = None*) → None

Add an extractor and its feature names to the dataframe.

Arguments:

extractor: Name of the extractor extractor_features: Tuples of [FeatureGroup, FeatureName] values: Initial values of the Extractor per instance in the dataframe.

Defaults to FeatureDataFrame.missing_value.

add_instances(*instance: str | list[str]*, *values: list[float] | None = None*) → None

Add one or more instances to the dataframe.

property extractors: list[str]

Returns all unique extractors in the DataFrame.

get_feature_groups(*extractor: str | list[str] | None = None*) → list[str]

Retrieve the feature groups in the dataframe.

Args:

extractor: Optional. If extractor(s) are given,
yields only feature groups of that extractor.

Returns:

A list of feature groups.

get_instance(*instance: str*) → list[float]

Return the feature vector of an instance.

get_value(*instance: str*, *extractor: str*, *feature_group: str*, *feature_name: str*) → None

Return a value in the dataframe.

has_missing_value() → bool

Return whether there are missing values in the feature data.

has_missing_vectors() → bool

Returns True if there are any Extractors still to be run on any instance.

impute_missing_values() → None

Imputes all NaN values by taking the average feature value.

property instances: list[str]

Return the instances in the dataframe.

remaining_jobs() → list[tuple[str, str, str]]

Determines needed feature computations per instance/extractor/group.

Returns:

list: A list of tuples representing (Extractor, Instance, Feature Group).
that needs to be computed.

remove_extractor(*extractor: str*) → None

Remove an extractor from the dataframe.

remove_instances(*instances: str* | *list[str]*) → None

Remove an instance from the dataframe.

reset_dataframe() → bool

Resets all values to FeatureDataFrame.missing_value.

save_csv(*csv_filepath: Path* | *None = None*) → None

Write a CSV to the given path.

Args:
csv_filepath: String path to the csv file. Defaults to self.csv_filepath.

set_value(*instance: str*, *extractor: str*, *feature_group: str*, *feature_name: str*, *value: float*) → None

Set a value in the dataframe.

sort() → None

Sorts the DataFrame by Multi-Index for readability.

to_autofolio(*target: Path* | *None = None*) → Path

Port the data to a format acceptable for AutoFolio.

class sparkle.structures.**PerformanceDataFrame**(*csv_filepath: Path*, *solvers: list[str] = []*, *objectives: list[str* | SparkleObjective*]* | *None = None*, *instances: list[str] = []*, *n_runs: int = 1*, *init_df: bool = True*)

Class to manage performance data and common operations on them.

add_instance(*instance_name: str*, *initial_value: float* | *list[float]* | *None = None*) → None

Add and instance to the DataFrame.

add_solver(*solver_name: str*, *initial_value: float* | *list[float]* | *None = None*) → None

Add a new solver to the dataframe. Initializes value to None by default.

Args:
solver_name: The name of the solver to be added. initial_value: The value assigned for each index of the new solver.

If not None, must match the index dimension (n_obj * n_inst * n_runs).

best_instance_performance(*objective: str* | SparkleObjective | *None = None*, *run_id: int* | *None = None*, *exclude_solvers: list[str]* | *None = None*) → Series

Return the best performance for each instance in the portfolio.

Args:
objective: The objective for which we calculate the best performance run_id: The run for which we calculate the best performance. If None,

we consider all runs.

exclude_solvers: List of solvers to exclude in the calculation.

Returns:
The best performance for each instance in the portfolio.

`best_performance`(*exclude_solvers: list[str] = []*, *objective: str* | SparkleObjective | *None = None*) → float

Return the overall best performance of the portfolio.

Args:

exclude_solvers: List of solvers to exclude in the calculation.
Defaults to none.

objective: The objective for which we calculate the best performance

Returns:
The aggregated best performance of the portfolio over all instances.

`clean_csv`() → None

Set all values in Performance Data to None.

`copy`(*csv_filepath: Path* | *None = None*) → *PerformanceDataFrame*

Create a copy of this object.

Args:

csv_filepath: The new filepath to use for saving the object to.
Warning: If the original path is used, it could lead to dataloss!

`get_job_list`(*rerun: bool = False*) → list[tuple[str, str]]

Return a list of performance computation jobs there are to be done.

Get a list of tuple[instance, solver] to run from the performance data csv file. If rerun is False (default), get only the tuples that don't have a value in the table, else (True) get all the tuples.

Args:
rerun: Boolean indicating if we want to rerun all jobs

`get_solver_ranking`(*objective: str* | SparkleObjective | *None = None*) → list[tuple[str, float]]

Return a list with solvers ranked by average performance.

`get_value`(*solver: str*, *instance: str*, *objective: str* | *None = None*, *run: int* | *None = None*) → float

Index a value of the DataFrame and return it.

`get_values`(*solver: str*, *instance: str* | *None = None*, *objective: str* | *None = None*, *run: int* | *None = None*) → list[float]

Return a list of solver values.

`property has_missing_values: bool`

Returns True if there are any missing values in the dataframe.

`property instances: list[str]`

Return the instances as a Pandas Index object.

`marginal_contribution`(*objective: str* | SparkleObjective | *None = None*, *sort: bool = False*) → list[float]

Return the marginal contribution of the solvers on the instances.

Args:
objective: The objective for which we calculate the marginal contribution. sort: Whether to sort the results afterwards

Returns:
The marginal contribution of each solver.

mean(*objective: str | None = None*, *solver: str | None = None*, *instance: str | None = None*) → float

Return the mean value of a slice of the dataframe.

property multi_objective: bool

Return whether the dataframe represent MO or not.

property num_instances: int

Return the number of instances.

property num_objectives: int

Retrieve the number of objectives in the DataFrame.

property num_runs: int

Return the number of runs.

property num_solvers: int

Return the number of solvers.

property objective_names: list[str]

Return the objective names as a list of strings.

remaining_jobs() → dict[str, list[str]]

Return a dictionary for empty values per instance and solver combination.

remove_instance(*instance_name: str*) → None

Drop an instance from the Dataframe.

remove_solver(*solver_name: str | list[str]*) → None

Drop one or more solvers from the Dataframe.

reset_value(*solver: str*, *instance: str*, *objective: str | None = None*, *run: int | None = None*) → None

Reset a value in the dataframe.

save_csv(*csv_filepath: Path | None = None*) → None

Write a CSV to the given path.

Args:
csv_filepath: String path to the csv file. Defaults to self.csv_filepath.

schedule_performance(*schedule: dict[slice(<class 'str'>, list[tuple[str, float | None]], None)], target_solver: str | None = None, objective: str | ~sparkle.types.objective.SparkleObjective | None = None*) → float

Return the performance of a selection schedule on the portfolio.

Args:

schedule: Compute the best performance according to a selection schedule.
A dictionary with instances as keys and a list of tuple consisting of (solver, max_runtime) or solvers if no runtime prediction should be used.

target_solver: If not None, store the values in this solver of the DF. objective: The objective for which we calculate the best performance

Returns:
The performance of the schedule over the instances in the dictionary.

set_value(*value: float*, *solver: str*, *instance: str*, *objective: str | None = None*, *run: int | None = None*) → None

Setter method to assign a value to the Dataframe.

Args:

value: Float value to be assigned. solver: The solver that produced the value. instance: The instance that the value was produced on. objective: The objective for which the result was produced.

Optional in case of using single objective.

run: The run index for which the result was produced.
Optional in case of doing single run results.

property solvers: list[str]

Return the solver present as a list of strings.

to_autofolio(*objective:* SparkleObjective | *None* = *None*, *target: Path* | *None* = *None*) → Path

Port the data to a format acceptable for AutoFolio.

verify_indexing(*objective: str*, *run_id: int*) → tuple[str, int]

Method to check whether data indexing is correct.

Users are allowed to use the Performance Dataframe without the second and fourth dimension (Objective and Run respectively) in the case they only have one objective or only do one run. This method adjusts the indexing for those cases accordingly.

Args:

objective: The given objective name run_id: The given run index

Returns:

A tuple representing the (possibly adjusted) Objective and Run index.

verify_objective(*objective: str*) → str

Method to check whether the specified objective is valid.

Users are allowed to index the dataframe without specifying all dimensions. However, when dealing with multiple objectives this is not allowed and this is verified here. If we have only one objective this is returned. Otherwise, if an objective is specified by the user this is returned.

Args:

objective: The objective given by the user

verify_run_id(*run_id: int*) → int

Method to check whether run id is valid.

Similar to verify_objective but here we check the dimensionality of runs.

Args:

run_id: the run as specified by the user.

1.7.13 tools

Init for the tools module.

class sparkle.tools.PCSParser(*inherit:* PCSParser | *None* = *None*)

Base interface object for the parser.

It loads the pcs files into the generic pcs object. Once a parameter file is loaded, it can be exported to another file

check_validity() → bool

Check the validity of the pcs.

export(*destination: Path*, *convention: str = 'smac'*) → None

Main export function.

load(*filepath: Path*, *convention: str = 'smac'*) → None

Main import function.

class sparkle.tools.**SlurmBatch**(*srcfile: Path*)

Class to parse a Slurm batch file and get structured information.

Attributes

sbatch_options: list[str]
The SBATCH options. Ex.: ["–array=-22%250", "–mem-per-cpu=3000"]

cmd_params: list[str]
The parameters to pass to the command

cmd: str
The command to execute

srun_options: list[str]
A list of arguments to pass to srun. Ex.: ["-n1", "–nodes=1"]

file: Path
The loaded file Path

sparkle.tools.**get_solver_call_params**(*args_dict: dict*) → list[str]

Gather the additional parameters for the solver call.

Args:
args_dict: Dictionary mapping argument names to their currently held values

Returns:
A list of parameters for the solver call

sparkle.tools.**get_time_pid_random_string**() → str

Return a combination of time, Process ID, and random int as string.

Returns:
A random string composed of time, PID and a random positive integer value.

1.7.14 types

This package provides types for Sparkle applications.

class sparkle.types.**FeatureGroup**(*value*)

Various feature groups.

class sparkle.types.**FeatureSubgroup**(*value*)

Various feature subgroups. Only used for embedding in with feature names.

class sparkle.types.**FeatureType**(*value*)

Various feature types.

static with_subgroup(*subgroup:* FeatureSubgroup, *feature:* FeatureType) → str

Return a standardised string with a subgroup embedded.

class sparkle.types.**SolverStatus**(*value*)

Possible return states for solver runs.

class sparkle.types.**SparkleCallable**(*directory: Path*, *runsolver_exec: Path | None = None*, *raw_output_directory: Path | None = None*)

Sparkle Callable class.

build_cmd() → list[str | Path]

A method that builds the commandline call string.

run() → None

A method that runs the callable.

class sparkle.types.**SparkleObjective**(*name: str*, *run_aggregator: ~typing.Callable = <function mean>*, *instance_aggregator: ~typing.Callable = <function mean>*, *solver_aggregator: ~typing.Callable | None = None*, *minimise: bool = True*, *post_process: ~typing.Callable | None = None*, *use_time: ~sparkle.types.objective.UseTime = UseTime.NO*)

Objective for Sparkle specified by user.

property time: bool

Return whether the objective is time based.

class sparkle.types.**UseTime**(*value*)

Enum describing what type of time to use.

sparkle.types.**_check_class**(*candidate: Callable*) → bool

Verify whether a loaded class is a valid objective class.

sparkle.types.**resolve_objective**(*objective_name: str*) → *SparkleObjective*

Try to resolve the objective class by (case-sensitive) name.

convention: objective_name(variable-k)?(:[min|max])?

Order of resolving:
class_name of user defined SparkleObjectives class_name of sparkle defined SparkleObjectives default SparkleObjective with minimization unless specified as max

Args:
name: The name of the objective class. Can include parameter value k.

Returns:
Instance of the Objective class or None if not found.

Python Module Index

S