MadMiner Documentation

Release 0.2.8

Johann Brehmer, Felix Kling, Irina Espejo, and Kyle Cranmer

Mar 04, 2019

Sites

Johann Brehmer, Felix Kling, Irina Espejo, and Kyle Cranmer

An inference toolkit for LHC measurements

Note that this is a development version. Do not reply on anything being stable. If you have any questions, please contact us at johann.brehmer@nyu.edu.

CHAPTER 1

Introduction to MadMiner

Particle physics processes are usually modelled with complex Monte-Carlo simulations of the hard process, parton shower, and detector interactions. These simulators typically do not admit a tractable likelihood function: given a (potentially high-dimensional) set of observables, it is usually not possible to calculate the probability of these observables for some model parameters. Particle physicisists usually tackle this problem of "likelihood-free inference" by hand-picking a few "good" observables or summary statistics and filling histograms of them. But this conventional approach discards the information in all other observables and often does not scale well to high-dimensional problems.

In the three publications "Constraining Effective Field Theories With Machine Learning", "A Guide to Constraining Effective Field Theories With Machine Learning", and "Mining gold from implicit models to improve likelihood-free inference", a new approach has been developed. In a nut shell, additional information is extracted from the simulators. This "augmented data" can be used to train neural networks to efficiently approximate arbitrary likelihood ratios. We playfully call this process "mining gold" from the simulator, since this information may be hard to get, but turns out to be very valuable for inference.

But the gold does not have to be hard to mine. This package automates these inference strategies. It wraps around the simulators MadGraph and Pythia, with different options for the detector simulation. All steps in the analysis chain from the simulation to the extraction of the augmented data, their processing, and the training and evaluation of the neural estimators are implemented.

CHAPTER 2

Getting started

2.1 Simulator dependencies

Make sure the following tools are installed and running:

- MadGraph (we've tested our setup with MG5_aMC v2.6.2 and v2.6.5). See https://launchpad.net/mg5amcnlo for installation instructions. Note that MadGraph requires a Fortran compiler as well as Python 2.6 or 2.7. (Note that you can still run most MadMiner analysis steps with Python 3.)
- For the analysis of systematic uncertainties, LHAPDF6 has to be installed with Python support (see also the documentation of MadGraph's systematics tool).

For the detector simulation part, there are different options. For simple parton-level analyses, we provide a bare-bones option to calculate truth-level observables which do not require any additional packages.

We have also implemented a fast detector simulation based on Delphes with a flexible framework to calculate observables. Using this adds additional requirements:

- Pythia8 and the MG-Pythia interface, installed from within the MadGraph command line interface: execute `<MadGraph5_directory>/bin/mg5_aMC`, and then inside the MadGraph interface, run `install pythia8` and `install mg5amc_py8_interface`.
- Delphes. Again, you can (but this time you don't have to) install it from the MadGraph command line interface with `install Delphes`.

(These tools currently have a bug: the MG-Pythia interface and Delphes currently do not keep track of additional weights that are in the LHE file. This is not a big deal, MadMiner now offers an option to extract these weights from the LHE file. Alternatively, there is a unofficial patch for these tools that solves these issues. It is available upon request.)

Finally, Delphes can be replaced with another detector simulation, for instance a full detector simulation based with Geant4. In this case, the user has to implement code that runs the detector simulation, calculates the observables, and stores the observables and weights in the HDF5 file. The `DelphesProcessor` and `LHEProcessor` classes might provide some guidance for this.

We're currently working on a reference Docker image that has all these dependencies and the needed patches installed.

2.2 Install MadMiner

To install the MadMiner package with all its Python dependencies, run `pip install madminer`.

To get the latest development version as well as the tutorials, clone the GitHub repository and run `pip install -e .` from the repository main folder.

CHAPTER 3

Using MadMiner

3.1 Tutorials

In our GitHub repository we provide a set of tutorials that are probably a good way to get started with MadMiner.

As a starting point, we recommend to look at a tutorial based on a toy example. It demonstrates inference with MadMinier without spending much time on the more technical steps of running the simulation.

We then provide two sets of tutorials for the same real-world particle physics process. The difference between them is that the parton-level tutorial only requires running MadGraph. Instead of a proper shower and detector simulation, we describe detector effects through simple smearing functions. This reduces the runtime of the scripts quite a bit. In the Delphes tutorial, we finally switch to Pythia and Delphes; this tutorial is probably best suited as a starting point for phenomenological research projects. In most other aspects, the two tutorials are identical.

Other provided examples show MadMiner in action in different processes.

3.2 Package structure

- `madminer.core` contains the functions to set up the process, parameter space, morphing, and to steer MadGraph and Pythia.
- `madminer.lhe` and `madminer.delphes` contain two example implementations of a detector simulation and observable calculation. This part can easily be swapped out depending on the use case.
- In `madminer.sampling`, train and test samples for the machine learning part are generated and augmented with the joint score and joint ratio.
- `madminer.ml` contains an implementation of the machine learning part. The user can train and evaluate estimators for the likelihood ratio or score.
- Finally, `madminer.fisherinformation` contains functions to calculate the Fisher information, both on parton level or detector level, in the full process, individual observables, or the total cross section.

3.3 Technical documentation

The madminer API is documented on here as well, just look through the pages linked on the left.

CHAPTER 4

References

If you use MadMiner, please cite this code as

```
@misc{MadMiner,
      author          = "Brehmer, Johann and Kling, Felix and Espejo, Irina and␣
↪Cranmer, Kyle",
      title           = "{MadMiner}",
      doi             = "10.5281/zenodo.1489147",
      url             = {https://github.com/johannbrehmer/madminer}
}
```

For the inference methods, there are three main references. Two introduce most of the methods in a particle physics setting:

```
@article{Brehmer:2018kdj,
      author          = "Brehmer, Johann and Cranmer, Kyle and Louppe, Gilles and
                         Pavez, Juan",
      title           = "{Constraining Effective Field Theories with Machine
                         Learning}",
      journal         = "Phys. Rev. Lett.",
      volume          = "121",
      year            = "2018",
      number          = "11",
      pages           = "111801",
      doi             = "10.1103/PhysRevLett.121.111801",
      eprint          = "1805.00013",
      archivePrefix   = "arXiv",
      primaryClass    = "hep-ph",
}

@article{Brehmer:2018eca,
      author          = "Brehmer, Johann and Cranmer, Kyle and Louppe, Gilles and
                         Pavez, Juan",
      title           = "{A Guide to Constraining Effective Field Theories with
                         Machine Learning}",
```

(continues on next page)

(continued from previous page)

```
      journal         = "Phys. Rev.",
      volume          = "D98",
      year            = "2018",
      number          = "5",
      pages           = "052004",
      doi             = "10.1103/PhysRevD.98.052004",
      eprint          = "1805.00020",
      archivePrefix   = "arXiv",
      primaryClass    = "hep-ph",
}
```

In addition, the inference techniques are discussed in a more general setting, and the SCANDAL family of methods is added in:

```
@article{Brehmer:2018hga,
      author          = "Brehmer, Johann and Louppe, Gilles and Pavez, Juan and
                         Cranmer, Kyle",
      title           = "{Mining gold from implicit models to improve
                         likelihood-free inference}",
      year            = "2018",
      eprint          = "1805.12244",
      archivePrefix   = "arXiv",
      primaryClass    = "stat.ML",
      SLACcitation    = "%%CITATION = ARXIV:1805.12244;%%"
}
```

Some inference methods are introduced in other papers, including CARL, Masked Autoregressive Flows, and ALICE(S).

4.1 Acknowledgements

We are immensely grateful to all contributors and bug reporters! In particular, we would like to thank Zubair Bhatti, Alexander Held, and Duccio Pappadopulo. A big thanks to Lukas Heinrich for his help with workflows and Docker containers.

The SCANDAL inference method is based on Masked Autoregressive Flows, and our implementation is a pyTorch port of the original code by George Papamakarios et al., which is available at https://github.com/gpapamak/maf.

The setup.py was adapted from https://github.com/kennethreitz/setup.py.

CHAPTER 5

madminer.core module

class madminer.core.**MadMiner**

Bases: `object`

The central class to manage parameter spaces, benchmarks, and the generation of events through MadGraph and Pythia.

An instance of this class is the starting point of most MadMiner applications. It is typically used in four steps:

- Defining the parameter space through *MadMiner.add_parameter*
- Defining the benchmarks, i.e. the points at which the squared matrix elements will be evaluated in MadGraph, with *MadMiner.add_benchmark()* or, if operator morphing is used, with *MadMiner.set_benchmarks_from_morphing()*
- Saving this setup with *MadMiner.save()* (it can be loaded in a new instance with *MadMiner.load()*)
- Running MadGraph and Pythia with the appropriate settings with *MadMiner.run()* or *MadMiner.run_multiple()* (the latter allows the user to combine runs from multiple run cards and sampling points)

Please see the tutorial for a hands-on introduction to its methods.

Methods

`add_benchmark`(parameter_values[, . . .])	Manually adds an individual benchmark, that is, a parameter point that will be evaluated by MadGraph.
`add_parameter`(lha_block, lha_id[, . . .])	Adds an individual parameter.
`load`(filename[, disable_morphing])	Loads MadMiner setup from a file.
`run`(mg_directory, proc_card_file, . . . [, . . .])	High-level function that creates the the MadGraph process, all required cards, and prepares or runs the event generation for one combination of cards.

Continued on next page

Table 1 – continued from previous page

run_multiple(mg_directory, proc_card_file, . . .)	High-level function that creates the the MadGraph process, all required cards, and prepares or runs the event generation for multiple combinations of run_cards or importance samplings (*sample_benchmarks*).
save(filename)	Saves MadMiner setup into a file.
set_benchmarks([benchmarks, verbose])	Manually sets all benchmarks, that is, parameter points that will be evaluated by MadGraph.
set_morphing([max_overall_power, n_bases, . . .])	Sets up the morphing environment.
set_parameters([parameters])	Manually sets all parameters, overwriting previously added parameters.
set_systematics([scale_variation, scales, . . .])	Prepares the simulation of the effect of different nuisance parameters, including scale variations and PDF changes.

add_benchmark (*parameter_values*, *benchmark_name=None*, *verbose=True*)
Manually adds an individual benchmark, that is, a parameter point that will be evaluated by MadGraph.

If this command is called before

Parameters

parameter_values [dict] The keys of this dict should be the parameter names and the values the corresponding parameter values.

benchmark_name [str or None, optional] Name of benchmark. If None, a default name is used. Default value: None.

verbose [bool, optional] If True, prints output about each benchmark. Default value: True.

Returns

None

Raises

RuntimeError If a benchmark with the same name already exists, if parameter_values is not a dict, or if a key of parameter_values does not correspond to a defined parameter.

add_parameter (*lha_block*, *lha_id*, *parameter_name=None*, *param_card_transform=None*, *morphing_max_power=2*, *parameter_range=(0.0, 1.0)*)
Adds an individual parameter.

Parameters

lha_block [str] The name of the LHA block as used in the param_card. Case-sensitive.

lha_id [int] The LHA id as used in the param_card.

parameter_name [str or None] An internal name for the parameter. If None, a the default 'benchmark_i' is used.

morphing_max_power [int or tuple of int] The maximal power with which this parameter contributes to the squared matrix element of the process of interest. If a tuple is given, gives this maximal power for each of several operator configurations. Typically at tree level, this maximal number is 2 for parameters that affect one vertex (e.g. only production or only decay of a particle), and 4 for parameters that affect two vertices (e.g. production and decay). Default value: 2.

param_card_transform [None or str] Represents a one-parameter function mapping the parameter (*"theta"*) to the value that should be written in the parameter cards. This str is parsed by Python's *eval()* function, and *"theta"* is parsed as the parameter value. Default value: None.

parameter_range [tuple of float] The range of parameter values of primary interest. Only affects the basis optimization. Default value: (0., 1.).

Returns

None

load (*filename*, *disable_morphing=False*)

Loads MadMiner setup from a file. All parameters, benchmarks, and morphing settings are overwritten. See *save* for more details.

Parameters

filename [str] Path to the MadMiner file.

disable_morphing [bool, optional] If True, the morphing setup is not loaded from the file. Default value: False.

Returns

None

run (*mg_directory*, *proc_card_file*, *param_card_template_file*, *run_card_file=None*, *mg_process_directory=None*, *pythia8_card_file=None*, *sample_benchmark=None*, *is_background=False*, *only_prepare_script=False*, *ufo_model_directory=None*, *log_directory=None*, *temp_directory=None*, *initial_command=None*, *python2_override=False*)

High-level function that creates the the MadGraph process, all required cards, and prepares or runs the event generation for one combination of cards.

If *only_prepare_scripts=True*, the event generation is not run directly, but a bash script is created in *<process_folder>/madminer/run.sh* that will start the event generation with the correct settings.

High-level function that creates the the MadGraph process, all required cards, and prepares or runs the event generation for multiple combinations of run_cards or importance samplings (*sample_benchmarks*).

If *only_prepare_scripts=True*, the event generation is not run directly, but a bash script is created in *<process_folder>/madminer/run.sh* that will start the event generation with the correct settings.

Parameters

mg_directory [str] Path to the MadGraph 5 base directory.

proc_card_file [str] Path to the process card that tells MadGraph how to generate the process.

param_card_template_file [str] Path to a param card that will be used as template to create the appropriate param cards for these runs.

run_card_file [str] Paths to the MadGraph run card. If None, the default run_card is used.

mg_process_directory [str or None, optional] Path to the MG process directory. If None, MadMiner uses ./MG_process. Default value: None.

pythia8_card_file [str or None, optional] Path to the MadGraph Pythia8 card. If None, the card present in the process folder is used. Default value: None.

sample_benchmark [list of str or None, optional] Lists the names of benchmarks that should be used to sample events. A different sampling does not change the expected differential cross sections, but will change which regions of phase space have many events (small variance) or few events (high variance). If None, the benchmark added first is used. Default value: None.

is_background [bool, optional] Should be True for background processes, i.e. process in which the differential cross section does not depend on the parameters (i.e. is the same for all benchmarks). In this case, no reweighting is run, which can substantially speed up the event generation. Default value: False.

only_prepare_script [bool, optional] If True, the event generation is not started, but instead a run.sh script is created in the process directory. Default value: False.

only_prepare_script [bool, optional] If True, MadGraph is not executed, but instead a run.sh script is created in the process directory. Default value: False.

ufo_model_directory [str or None, optional] Path to an UFO model directory that should be used, but is not yet installed in mg_directory/models. The model will be copied to the MadGraph model directory before the process directory is generated. (Default value = None.

log_directory [str or None, optional] Directory for log files with the MadGraph output. If None, ./logs is used. Default value: None.

temp_directory [str or None, optional] Path to a temporary directory. If None, a system default is used. Default value: None.

initial_command [str or None, optional] Initial shell commands that have to be executed before MG is run (e.g. to load a virtual environment). Default value: None.

python2_override [bool, optional] If True, MadMiner explicitly calls "python2" instead of relying on the system Python version to be Python 2.6 or Python 2.7. If you use systematics, make sure that the python interface of LHAPDF was compiled with the Python version you are using. Default: False.

Returns

None

`run_multiple`(*mg_directory*, *proc_card_file*, *param_card_template_file*, *run_card_files*, *mg_process_directory=None*, *pythia8_card_file=None*, *sample_benchmarks=None*, *is_background=False*, *only_prepare_script=False*, *ufo_model_directory=None*, *log_directory=None*, *temp_directory=None*, *initial_command=None*, *python2_override=False*)

High-level function that creates the the MadGraph process, all required cards, and prepares or runs the event generation for multiple combinations of run_cards or importance samplings (*sample_benchmarks*).

If *only_prepare_scripts=True*, the event generation is not run directly, but a bash script is created in *<process_folder>/madminer/run.sh* that will start the event generation with the correct settings.

Parameters

mg_directory [str] Path to the MadGraph 5 base directory.

proc_card_file [str] Path to the process card that tells MadGraph how to generate the process.

param_card_template_file [str] Path to a param card that will be used as template to create the appropriate param cards for these runs.

run_card_files [list of str] Paths to the MadGraph run card.

mg_process_directory [str or None, optional] Path to the MG process directory. If None, MadMiner uses ./MG_process. Default value: None.

pythia8_card_file [str, optional] Path to the MadGraph Pythia8 card. If None, the card present in the process folder is used. Default value: None.

sample_benchmarks [list of str or None, optional] Lists the names of benchmarks that should be used to sample events. A different sampling does not change the expected differential cross sections, but will change which regions of phase space have many events (small variance) or few events (high variance). If None, a run is started for each of the benchmarks, which should map out all regions of phase space well. Default value: None.

is_background [bool, optional] Should be True for background processes, i.e. process in which the differential cross section does not depend on the parameters (i.e. is the same for all benchmarks). In this case, no reweighting is run, which can substantially speed up the event generation. Default value: False.

only_prepare_script [bool, optional] If True, the event generation is not started, but instead a run.sh script is created in the process directory. Default value: False.

only_prepare_script [bool, optional] If True, MadGraph is not executed, but instead a run.sh script is created in the process directory. Default value: False.

ufo_model_directory [str or None, optional] Path to an UFO model directory that should be used, but is not yet installed in mg_directory/models. The model will be copied to the MadGraph model directory before the process directory is generated. (Default value = None)

log_directory [str or None, optional] Directory for log files with the MadGraph output. If None, ./logs is used. Default value: None.

temp_directory [str or None, optional] Path to a temporary directory. If None, a system default is used. Default value: None.

initial_command [str or None, optional] Initial shell commands that have to be executed before MG is run (e.g. to load a virtual environment). Default value: None.

python2_override [bool, optional] If True, MadMiner explicitly calls "python2" instead of relying on the system Python version to be Python 2.6 or Python 2.7. If you use systematics, make sure that the python interface of LHAPDF was compiled with the Python version you are using. Default: False.

Returns

None

`save` (*filename*)

Saves MadMiner setup into a file.

The file format follows the HDF5 standard. The saved information includes:

- the parameter definitions,
- the benchmark points,
- the systematics setup (if defined), and
- the morphing setup (if defined).

This file is an important input to later stages in the analysis chain, including the processing of generated events, extraction of training samples, and calculation of Fisher information matrices. In these downstream tasks, additional information will be written to the MadMiner file, including the observations and event weights.

Parameters

filename [str] Path to the MadMiner file.

Returns

None

set_benchmarks(*benchmarks=None*, *verbose=True*)

Manually sets all benchmarks, that is, parameter points that will be evaluated by MadGraph. Calling this function overwrites all previously defined benchmarks.

Parameters

benchmarks [dict or list or None, optional] Specifies all benchmarks. If None, all benchmarks are reset. If dict, the keys are the benchmark names and the values are dicts of the form {parameter_name:value}. If list, the entries are dicts {parameter_name:value} (and the benchmark names are chosen automatically). Default value: None.

verbose [bool, optional] If True, prints output about each benchmark. Default value: True.

Returns

None

set_morphing(*max_overall_power=4*, *n_bases=1*, *include_existing_benchmarks=True*, *n_trials=100*, *n_test_thetas=100*)

Sets up the morphing environment.

Sets benchmarks, i.e. parameter points that will be evaluated by MadGraph, for a morphing algorithm, and calculates all information required for morphing. Morphing is a technique that allows MadMax to infer the full probability distribution *p(x_i | theta)* for each simulated event *x_i* and any *theta*, not just the benchmarks.

The morphing basis is optimized with respect to the expected mean squared morphing weights over the parameter region of interest. If keep_existing_benchmarks=True, benchmarks defined previously will be incorporated in the morphing basis and only the remaining basis points will be optimized.

Note that any subsequent call to *set_benchmarks* or *add_benchmark* will overwrite the morphing setup. The correct order is therefore to manually define benchmarks first, using *set_benchmarks* or *add_benchmark*, and then to create the morphing setup and complete the basis by calling *set_benchmarks_from_morphing(keep_existing_benchmarks=True)*.

Parameters

max_overall_power [int or tuple of int, optional] The maximal sum of powers of all parameters contributing to the squared matrix element. If a tuple is given, gives the maximal sum of powers for each of several operator configurations (see *add_parameter*). Typically, if parameters can affect the couplings at n vertices, this number is 2n. Default value: 4.

n_bases [int, optional] The number of morphing bases generated. If n_bases > 1, multiple bases are combined, and the weights for each basis are reduced by a factor 1 / n_bases. Currently only the default choice of 1 is fully implemented. Do not use any other value for now. Default value: 1.

include_existing_benchmarks [bool, optional] If True, the previously defined benchmarks are included in the morphing basis. In that case, the number of free parameters in the optimization routine is reduced. If False, the existing benchmarks will still be simulated, but are not part of the morphing routine. Default value: True.

n_trials [int, optional] Number of random basis configurations tested in the optimization procedure. A larger number will increase the run time of the optimization, but lead to better results. Default value: 100.

n_test_thetas [int, optional] Number of random parameter points used to evaluate the expected mean squared morphing weights. A larger number will increase the run time of the optimization, but lead to better results. Default value: 100.

Returns

None

set_parameters(*parameters=None*)

Manually sets all parameters, overwriting previously added parameters.

Parameters

parameters [dict or list or None, optional] If parameters is None, resets parameters. If parameters is an dict, the keys should be str and give the parameter names, and the values are tuples of the form (LHA_block, LHA_ID, morphing_max_power, param_min, param_max) or of the form (LHA_block, LHA_ID). If parameters is a list, the items should be tuples of the form (LHA_block, LHA_ID). Default value: None.

Returns

None

set_systematics(*scale_variation=None*, *scales='together'*, *pdf_variation=None*)

Prepares the simulation of the effect of different nuisance parameters, including scale variations and PDF changes.

Parameters

scale_variation [None or tuple of float, optional] If not None, the regularization and / or factorization scales are varied. A tuple like (0.5,1.,2.) specifies the factors with which they are varied. Default value: None.

scales [{“together”, “independent”, “mur”, “muf”}, optional] Whether only the regularization scale (“mur”), only the factorization scale (“muf”), both simultanously (“together”) or both independently (“independent”) are varied. Default value: “together”.

pdf_variation [None or str, optional] If not None, the PDFs are varied. The option is passed along to the *–pdf* option of MadGraph’s systematics module. See https://cp3.irmp.ucl.ac.be/projects/madgraph/wiki/Systematics for a list. The option “CT10” would, as an example, run over all the eigenvectors of the CTEQ10 set.

Returns

None

CHAPTER 6

madminer.delphes module

class madminer.delphes.**DelphesProcessor**(*filename*)

Bases: `object`

Detector simulation with Delphes and simple calculation of observables.

After setting up the parameter space and benchmarks and running MadGraph and Pythia, all of which is organized in the madminer.core.MadMiner class, the next steps are the simulation of detector effects and the calculation of observables. Different tools can be used for these tasks, please feel free to implement the detector simulation and analysis routine of your choice.

This class provides an example implementation based on Delphes. Its workflow consists of the following steps:

- Initializing the class with the filename of a MadMiner HDF5 file (the output of *madminer.core.MadMiner.save()*)
- Adding one or multiple event samples produced by MadGraph and Pythia in *DelphesProcessor.add_sample()*.
- Running Delphes on the samples that require it through *DelphesProcessor.run_delphes()*.
- Optionally, acceptance cuts for all visible particles can be defined with *DelphesProcessor.set_acceptance()*.
- Defining observables through *DelphesProcessor.add_observable()* or *DelphesProcessor.add_observable_from_function()*. A simple set of default observables is provided in *DelphesProcessor.add_default_observables()*
- Optionally, cuts can be set with *DelphesProcessor.add_cut()*
- Calculating the observables from the Delphes ROOT files with *DelphesProcessor.analyse_delphes_samples()*
- Saving the results with *DelphesProcessor.save()*

Please see the tutorial for a detailed walk-through.

Parameters

filename [str or None, optional] Path to MadMiner file (the output of *madminer.core.MadMiner.save()*). Default value: None.

Methods

`add_cut`(definition[, pass_if_not_parsed])	Adds a cut as a string that can be parsed by Python's *eval()* function and returns a bool.
`add_default_observables`([n_leptons_max, ...])	Adds a set of simple standard observables: the four-momenta (parameterized as E, pT, eta, phi) of the hardest visible particles, and the missing transverse energy.
`add_observable`(name, definition[, required, ...])	Adds an observable as a string that can be parsed by Python's *eval()* function.
`add_observable_from_function`(name, fn[, ...])	Adds an observable defined through a function.
`add_sample`(hepmc_filename, ...[, ...])	Adds a sample of simulated events.
`analyse_delphes_samples`([generator_truth, ...])	Main function that parses the Delphes samples (ROOT files), checks acceptance and cuts, and extracts the observables and weights.
`reset_cuts`()	Resets all cuts.
`reset_observables`()	Resets all observables.
`run_delphes`(delphes_directory, delphes_card)	Runs the fast detector simulation Delphes on all HepMC samples added so far for which it hasn't been run yet.
`save`(filename_out)	Saves the observable definitions, observable values, and event weights in a MadMiner file.
`set_acceptance`([pt_min_e, pt_min_mu, ...])	Sets acceptance cuts for all visible particles.

add_cut (*definition*, *pass_if_not_parsed=False*)

Adds a cut as a string that can be parsed by Python's *eval()* function and returns a bool.

Parameters

definition [str] An expression that can be parsed by Python's *eval()* function and returns a bool: True for the event to pass this cut, False for it to be rejected. In the definition, all visible particles can be used: *e*, *mu*, *j*, *a*, and *l* provide lists of electrons, muons, jets, photons, and leptons (electrons and muons combined), in each case sorted by descending transverse momentum. *met* provides a missing ET object. *visible* and *all* provide access to the sum of all visible particles and the sum of all visible particles plus MET, respectively. All these objects are instances of *MadMinerParticle*, which inherits from scikit-hep's [LorentzVector](http://scikit-hep.org/api/math.html#vector-classes). See the link for a documentation of their properties. In addition, *MadMinerParticle* have properties *charge* and *pdg_id*, which return the charge in units of elementary charges (i.e. an electron has *e[0].charge = -1.*), and the PDG particle ID. For instance, *"len(e) >= 2"* requires at least two electrons passing the acceptance cuts, while *"mu[0].charge > 0."* specifies that the hardest muon is positively charged.

pass_if_not_parsed [bool, optional] Whether the cut is passed if the observable cannot be parsed. Default value: False.

Returns

None

add_default_observables(*n_leptons_max=2*, *n_photons_max=2*, *n_jets_max=2*, *include_met=True*, *include_visible_sum=True*, *include_numbers=True*, *include_charge=True*)

Adds a set of simple standard observables: the four-momenta (parameterized as E, pT, eta, phi) of the hardest visible particles, and the missing transverse energy.

Parameters

n_leptons_max [int, optional] Number of hardest leptons for which the four-momenta are saved. Default value: 2.

n_photons_max [int, optional] Number of hardest photons for which the four-momenta are saved. Default value: 2.

n_jets_max [int, optional] Number of hardest jets for which the four-momenta are saved. Default value: 2.

include_met [bool, optional] Whether the missing energy observables are stored. Default value: True.

include_visible_sum [bool, optional] Whether observables characterizing the sum of all particles are stored. Default value: True.

include_numbers [bool, optional] Whether the number of leptons, photons, and jets is saved as observable. Default value: True.

include_charge [bool, optional] Whether the lepton charge is saved as observable. Default value: True.

Returns

None

add_observable(*name*, *definition*, *required=False*, *default=None*)

Adds an observable as a string that can be parsed by Python's *eval()* function.

Parameters

name [str] Name of the observable. Since this name will be used in *eval()* calls for cuts, this should not contain spaces or special characters.

definition [str] An expression that can be parsed by Python's *eval()* function. As objects, the visible particles can be used: *e*, *mu*, *j*, *a*, and *l* provide lists of electrons, muons, jets, photons, and leptons (electrons and muons combined), in each case sorted by descending transverse momentum. *met* provides a missing ET object. *visible* and *all* provide access to the sum of all visible particles and the sum of all visible particles plus MET, respectively. All these objects are instances of *MadMinerParticle*, which inherits from scikit-hep's [LorentzVector](http://scikit-hep.org/api/math.html#vector-classes). See the link for a documentation of their properties. In addition, *MadMinerParticle* have properties *charge* and *pdg_id*, which return the charge in units of elementary charges (i.e. an electron has *e[0].charge = -1.*), and the PDG particle ID. For instance, *"abs(j[0].phi() - j[1].phi())"* defines the azimuthal angle between the two hardest jets.

required [bool, optional] Whether the observable is required. If True, an event will only be retained if this observable is successfully parsed. For instance, any observable involving *"j[1]"* will only be parsed if there are at least two jets passing the acceptance cuts. Default value: False.

default [float or None, optional] If *required=False*, this is the placeholder value for observables that cannot be parsed. None is replaced with *np.nan*. Default value: None.

Returns

None

add_observable_from_function(*name*, *fn*, *required=False*, *default=None*)

Adds an observable defined through a function.

Parameters

name [str] Name of the observable. Since this name will be used in *eval()* calls for cuts, this should not contain spaces or special characters.

fn [function] A function with signature *observable(leptons, photons, jets, met)* where the input arguments are lists of MadMinerParticle instances and a float is returned. The function should raise a *RuntimeError* to signal that it is not defined.

required [bool, optional] Whether the observable is required. If True, an event will only be retained if this observable is successfully parsed. For instance, any observable involving *"j[1]"* will only be parsed if there are at least two jets passing the acceptance cuts. Default value: False.

default [float or None, optional] If *required=False*, this is the placeholder value for observables that cannot be parsed. None is replaced with *np.nan*. Default value: None.

Returns

None

add_sample(*hepmc_filename*, *sampled_from_benchmark*, *is_background=False*, *delphes_filename=None*, *lhe_filename=None*, *k_factor=1.0*, *weights='lhe'*)

Adds a sample of simulated events. A HepMC file (from Pythia) has to be provided always, since some relevant information is only stored in this file. The user can optionally provide a Delphes file, in this case run_delphes() does not have to be called.

By default, the weights are read out from the Delphes file and their names from the HepMC file. There are some issues with current MadGraph versions that lead to Pythia not storing the weights. As work-around, MadMiner supports reading weights from the LHE file (the observables still come from the Delphes file). To enable this, use weights="lhe".

Parameters

hepmc_filename [str] Path to the HepMC event file (with extension '.hepmc' or '.hepmc.gz').

sampled_from_benchmark [str] Name of the benchmark that was used for sampling in this event file (the keyword *sample_benchmark* of *madminer.core.MadMiner.run()*).

is_background [bool, optional] Whether the sample is a background sample (i.e. without benchmark reweighting).

delphes_filename [str or None, optional] Path to the Delphes event file (with extension '.root'). If None, the user has to call run_delphes(), which will create this file. Default value: None.

lhe_filename [None or str, optional] Path to the LHE event file (with extension '.lhe' or '.lhe.gz'). This is only needed if weights is "lhe".

k_factor [float, optional] Multiplies the cross sections found in the sample. Default value: 1.

weights [{"delphes", "lhe"}, optional] If "delphes", the weights are read out from the Delphes ROOT file, and their names are taken from the HepMC file. If "lhe" (and lhe_filename is not None), the weights are taken from the LHE file (and matched with the observables from the Delphes ROOT file). The "delphes" behaviour is generally better as it minimizes the risk of mismatching observables and weights, but for some MadGraph and Delphes versions there are issues with weights not being saved in the HepMC and Delphes ROOT files. In this case, setting weights to "lhe" and providing the unweighted LHE file from MadGraph may be an easy fix. Default value: "lhe".

Returns

None

analyse_delphes_samples(*generator_truth=False*, *delete_delphes_files=False*, *reference_benchmark=None*, *parse_lhe_events_as_xml=True*)

Main function that parses the Delphes samples (ROOT files), checks acceptance and cuts, and extracts the observables and weights.

Parameters

generator_truth [bool, optional] If True, the generator truth information (as given out by Pythia) will be parsed. Detector resolution or efficiency effects will not be taken into account.

delete_delphes_files [bool, optional] If True, the Delphes ROOT files will be deleted after extracting the information from them. Default value: False.

reference_benchmark [str or None, optional] The weights at the nuisance benchmarks will be rescaled to some reference theta benchmark: *dsigma(x|theta_sampling(x),nu) -> dsigma(x|theta_ref,nu) = dsigma(x|theta_sampling(x),nu) * dsigma(x|theta_ref,0) / dsigma(x|theta_sampling(x),0)*. This sets the name of the reference benchmark. If None, the first one will be used. Default value: None.

parse_lhe_events_as_xml [bool, optional] Decides whether the LHE events are parsed with an XML parser (more robust, but slower) or a text parser (less robust, faster). Default value: True.

Returns

None

reset_cuts()

Resets all cuts.

reset_observables()

Resets all observables.

run_delphes(*delphes_directory*, *delphes_card*, *initial_command=None*, *log_file=None*)

Runs the fast detector simulation Delphes on all HepMC samples added so far for which it hasn't been run yet.

Parameters

delphes_directory [str] Path to the Delphes directory.

delphes_card [str] Path to a Delphes card.

initial_command [str or None, optional] Initial bash commands that have to be executed before Delphes is run (e.g. to load the correct virtual environment). Default value: None.

log_file [str or None, optional] Path to log file in which the Delphes output is saved. Default value: None.

Returns

None

save(*filename_out*)

Saves the observable definitions, observable values, and event weights in a MadMiner file. The parameter, benchmark, and morphing setup is copied from the file provided during initialization. Nuisance benchmarks found in the HepMC file are added.

Parameters

filename_out [str] Path to where the results should be saved.

Returns

None

set_acceptance(*pt_min_e=None*, *pt_min_mu=None*, *pt_min_a=None*, *pt_min_j=None*, *eta_max_e=None*, *eta_max_mu=None*, *eta_max_a=None*, *eta_max_j=None*)

Sets acceptance cuts for all visible particles. These are taken into account before observables and cuts are calculated.

Parameters

pt_min_e [float or None, optional] Minimum electron transverse momentum in GeV. None means no acceptance cut. Default value: None.

pt_min_mu [float or None, optional] Minimum muon transverse momentum in GeV. None means no acceptance cut. Default value: None.

pt_min_a [float or None, optional] Minimum photon transverse momentum in GeV. None means no acceptance cut. Default value: None.

pt_min_j [float or None, optional] Minimum jet transverse momentum in GeV. None means no acceptance cut. Default value: None.

eta_max_e [float or None, optional] Maximum absolute electron pseudorapidity. None means no acceptance cut. Default value: None.

eta_max_mu [float or None, optional] Maximum absolute muon pseudorapidity. None means no acceptance cut. Default value: None.

eta_max_a [float or None, optional] Maximum absolute photon pseudorapidity. None means no acceptance cut. Default value: None.

eta_max_j [float or None, optional] Maximum absolute jet pseudorapidity. None means no acceptance cut. Default value: None.

Returns

None

CHAPTER 7

madminer.fisherinformation module

class madminer.fisherinformation.**FisherInformation**(*filename*, *include_nuisance_parameters=True*)

Bases: object

Functions to calculate expected Fisher information matrices.

After inializing a *FisherInformation* instance with the filename of a MadMiner file, different information matrices can be calculated:

- *FisherInformation.calculate_fisher_information_full_truth()* calculates the full truth-level Fisher information. This is the information in an idealized measurement where all parton-level particles with their charges, flavours, and four-momenta can be accessed with perfect accuracy.
- *FisherInformation.calculate_fisher_information_full_detector()* calculates the full Fisher information in realistic detector-level observations, estimated with neural networks. In addition to the MadMiner file, this requires a trained SALLY or SALLINO estimator as well as an unweighted evaluation sample.
- *FisherInformation.calculate_fisher_information_rate()* calculates the Fisher information in the total cross section.
- *FisherInformation.calculate_fisher_information_hist1d()* calculates the Fisher information in the histogram of one (parton-level or detector-level) observable.
- *FisherInformation.calculate_fisher_information_hist2d()* calculates the Fisher information in a two-dimensional histogram of two (parton-level or detector-level) observables.
- *FisherInformation.histogram_of_fisher_information()* calculates the full truth-level Fisher information in different slices of one observable (the "distribution of the Fisher information").

Finally, don't forget that in the presence of nuisance parameters the constraint terms also affect the Fisher information. This term is given by *FisherInformation.calculate_fisher_information_nuisance_constraints()*.

Parameters

filename [str] Path to MadMiner file (for instance the output of *madminer.delphes.DelphesProcessor.save()*).

include_nuisance_parameters [bool, optional] If True, nuisance parameters are taken into account. Default value: True.

Methods

calculate_fisher_information_full_detector(. . .)	Calculates the full Fisher information in realistic detector-level observations, estimated with neural networks.
calculate_fisher_information_full_truth(theta)	Calculates the full Fisher information at parton / truth level.
calculate_fisher_information_hist1d(theta, . . .)	Calculates the Fisher information in the one-dimensional histogram of an (parton-level or detector-level, depending on how the observations in the MadMiner file were calculated) observable.
calculate_fisher_information_hist2d(theta, . . .)	Calculates the Fisher information in a two-dimensional histogram of two (parton-level or detector-level, depending on how the observations in the MadMiner file were calculated) observables.
calculate_fisher_information_nuisance_constraints()	Builds the Fisher information term representing the Gaussian constraints on the nuisance parameters
calculate_fisher_information_rate(theta, . . .)	Calculates the Fisher information in a measurement of the total cross section (without any kinematic information).
extract_observables_and_weights(thetas)	Extracts observables and weights for given parameter points.
extract_raw_data([theta])	Returns all events together with the benchmark weights (if theta is None) or weights for a given theta.
histogram_of_fisher_information(theta, . . .)	Calculates the full and rate-only Fisher information in slices of one observable.

calculate_fisher_information_full_detector(*theta*, *model_file*, *unweighted_x_sample_file=None*, *luminosity=300000.0*, *include_xsec_info=True*, *mode='score'*, *uncertainty='ensemble'*, *ensemble_vote_expectation_weight=None*, *batch_size=100000*, *test_split=0.5*)

Calculates the full Fisher information in realistic detector-level observations, estimated with neural networks. In addition to the MadMiner file, this requires a trained SALLY or SALLINO estimator.

Nuisance parameter are taken into account automatically if the SALLY / SALLINO model was trained with them.

Parameters

theta [ndarray] Parameter point *theta* at which the Fisher information matrix *I_ij(theta)* is evaluated.

model_file [str] Filename of a trained local score regression model that was trained on samples from *theta* (see *madminer.ml.MLForge*).

unweighted_x_sample_file [str or None] Filename of an unweighted x sample that is sampled according to theta and obeys the cuts (see *mad-*

miner.sampling.SampleAugmenter.extract_samples_train_local()). If None, the Fisher information is instead calculated on the full, weighted samples (the data in the MadMiner file). Default value: None.

luminosity [float, optional] Luminosity in pb^-1. Default value: 300000.

include_xsec_info [bool, optional] Whether the rate information is included in the returned Fisher information. Default value: True.

mode [{"score", "information"}, optional] How the ensemble uncertainty on the kinematic Fisher information is calculated. If mode is "information", the Fisher information for each estimator is calculated individually and only then are the sample mean and covariance calculated. If mode is "score", the sample mean is calculated for the score for each event. Default value: "score".

uncertainty [{"ensemble", "expectation", "sum"}, optional] How the covariance matrix of the Fisher information estimate is calculated. With "ensemble", the ensemble covariance is used. With "expectation", the expectation of the score is used as a measure of the uncertainty of the score estimator, and this uncertainty is propagated through to the covariance matrix. With "sum", both terms are summed. Default value: "ensemble".

ensemble_vote_expectation_weight [float or list of float or None, optional] For ensemble models, the factor that determines how much more weight is given to those estimators with small expectation value. If a list is given, results are returned for each element in the list. If None, or if *EnsembleForge.calculate_expectation()* has not been called, all estimators are treated equal. Default value: None.

batch_size [int, optional] Batch size. Default value: 100000.

test_split [float or None, optional] If unweighted_x_sample_file is None, this determines the fraction of weighted events used for evaluation. If None, all events are used (this will probably include events used during training!). Default value: 0.5.

Returns

fisher_information [ndarray or list of ndarray] Estimated expected full detector-level Fisher information matrix with shape *(n_parameters, n_parameters)*. If more then one value ensemble_vote_expectation_weight is given, this is a list with results for all entries in ensemble_vote_expectation_weight.

fisher_information_uncertainty [ndarray or list of ndarray or None] Covariance matrix of the Fisher information matrix with shape *(n_parameters, n_parameters, n_parameters, n_parameters)*. If more then one value ensemble_vote_expectation_weight is given, this is a list with results for all entries in ensemble_vote_expectation_weight.

calculate_fisher_information_full_truth(*theta*, *luminosity=300000.0*, *cuts=None*, *efficiency_functions=None*, *include_nuisance_parameters=True*)

Calculates the full Fisher information at parton / truth level. This is the information in an idealized measurement where all parton-level particles with their charges, flavours, and four-momenta can be accessed with perfect accuracy, i.e. the latent variables *z_parton* can be measured directly.

Parameters

theta [ndarray] Parameter point *theta* at which the Fisher information matrix *I_ij(theta)* is evaluated.

luminosity [float] Luminosity in pb^-1.

cuts [None or list of str, optional] Cuts. Each entry is a parseable Python expression that returns a bool (True if the event should pass a cut, False otherwise). Default value: None.

efficiency_functions [list of str or None] Efficiencies. Each entry is a parseable Python expression that returns a float for the efficiency of one component. Default value: None.

include_nuisance_parameters [bool, optional] If True, nuisance parameters are taken into account. Default value: True.

Returns

fisher_information [ndarray] Expected full truth-level Fisher information matrix with shape *(n_parameters, n_parameters)*.

fisher_information_uncertainty [ndarray] Covariance matrix of the Fisher information matrix with shape *(n_parameters, n_parameters, n_parameters, n_parameters)*, calculated with plain Gaussian error propagation.

calculate_fisher_information_hist1d(*theta*, *luminosity*, *observable*, *nbins*, *histrange=None*, *cuts=None*, *efficiency_functions=None*, *n_events_dynamic_binning=None*)

Calculates the Fisher information in the one-dimensional histogram of an (parton-level or detector-level, depending on how the observations in the MadMiner file were calculated) observable.

Parameters

theta [ndarray] Parameter point *theta* at which the Fisher information matrix *I_ij(theta)* is evaluated.

luminosity [float] Luminosity in pb^-1.

observable [str] Expression for the observable to be histogrammed. The str will be parsed by Python's *eval()* function and can use the names of the observables in the MadMiner files.

nbins [int] Number of bins in the histogram, excluding overflow bins.

histrange [tuple of float or None, optional] Minimum and maximum value of the histogram in the form *(min, max)*. Overflow bins are always added. If None, variable-width bins with equal cross section are constructed automatically. Default value: None.

cuts [None or list of str, optional] Cuts. Each entry is a parseable Python expression that returns a bool (True if the event should pass a cut, False otherwise). Default value: None.

efficiency_functions [list of str or None] Efficiencies. Each entry is a parseable Python expression that returns a float for the efficiency of one component. Default value: None.

n_events_dynamic_binning [int or None, optional] Number of events used to calculate the dynamic binning (if histrange is None). If None, all events are used. Note that these events are not shuffled, so if the events in the MadMiner file are sorted, using a value different from None can cause issues. Default value: None.

Returns

fisher_information [ndarray] Expected Fisher information in the histogram with shape *(n_parameters, n_parameters)*.

fisher_information_uncertainty [ndarray] Covariance matrix of the Fisher information matrix with shape *(n_parameters, n_parameters, n_parameters, n_parameters)*, calculated with plain Gaussian error propagation.

calculate_fisher_information_hist2d(*theta*, *luminosity*, *observable1*, *nbins1*, *observable2*, *nbins2*, *histrange1=None*, *histrange2=None*, *cuts=None*, *efficiency_functions=None*, *n_events_dynamic_binning=None*)

Calculates the Fisher information in a two-dimensional histogram of two (parton-level or detector-level, depending on how the observations in the MadMiner file were calculated) observables.

Parameters

theta [ndarray] Parameter point *theta* at which the Fisher information matrix *I_ij(theta)* is evaluated.

luminosity [float] Luminosity in pb^-1.

observable1 [str] Expression for the first observable to be histogrammed. The str will be parsed by Python's *eval()* function and can use the names of the observables in the MadMiner files.

nbins1 [int] Number of bins along the first axis in the histogram, excluding overflow bins.

observable2 [str] Expression for the first observable to be histogrammed. The str will be parsed by Python's *eval()* function and can use the names of the observables in the MadMiner files.

nbins2 [int] Number of bins along the first axis in the histogram, excluding overflow bins.

histrange1 [tuple of float or None, optional] Minimum and maximum value of the first axis of the histogram in the form *(min, max)*. Overflow bins are always added. If None, variable-width bins with equal cross section are constructed automatically. Default value: None.

histrange2 [tuple of float or None, optional] Minimum and maximum value of the first axis of the histogram in the form *(min, max)*. Overflow bins are always added. If None, variable-width bins with equal cross section are constructed automatically. Default value: None.

cuts [None or list of str, optional] Cuts. Each entry is a parseable Python expression that returns a bool (True if the event should pass a cut, False otherwise). Default value: None.

efficiency_functions [list of str or None] Efficiencies. Each entry is a parseable Python expression that returns a float for the efficiency of one component. Default value: None.

n_events_dynamic_binning [int or None, optional] Number of events used to calculate the dynamic binning (if histrange is None). If None, all events are used. Note that these events are not shuffled, so if the events in the MadMiner file are sorted, using a value different from None can cause issues. Default value: None.

Returns

fisher_information [ndarray] Expected Fisher information in the histogram with shape *(n_parameters, n_parameters)*.

fisher_information_uncertainty [ndarray] Covariance matrix of the Fisher information matrix with shape *(n_parameters, n_parameters, n_parameters, n_parameters)*, calculated with plain Gaussian error propagation.

`calculate_fisher_information_nuisance_constraints``()`

Builds the Fisher information term representing the Gaussian constraints on the nuisance parameters

`calculate_fisher_information_rate``(`*theta*, *luminosity*, *cuts=None*, *efficiency_functions=None*, *include_nuisance_parameters=True*`)`

Calculates the Fisher information in a measurement of the total cross section (without any kinematic information).

Parameters

theta [ndarray] Parameter point *theta* at which the Fisher information matrix *I_ij(theta)* is evaluated.

luminosity [float] Luminosity in pb^-1.

cuts [None or list of str, optional] Cuts. Each entry is a parseable Python expression that returns a bool (True if the event should pass a cut, False otherwise). Default value: None.

efficiency_functions [list of str or None] Efficiencies. Each entry is a parseable Python expression that returns a float for the efficiency of one component. Default value: None.

include_nuisance_parameters [bool, optional] If True, nuisance parameters are taken into account. Default value: True.

Returns

fisher_information [ndarray] Expected Fisher information in the total cross section with shape *(n_parameters, n_parameters)*.

fisher_information_uncertainty [ndarray] Covariance matrix of the Fisher information matrix with shape *(n_parameters, n_parameters, n_parameters, n_parameters)*, calculated with plain Gaussian error propagation.

`extract_observables_and_weights`(*thetas*)

Extracts observables and weights for given parameter points.

Parameters

thetas [ndarray] Parameter points, with shape *(n_thetas, n_parameters)*.

Returns

x [ndarray] Observations *x* with shape *(n_events, n_observables)*.

weights [ndarray] Weights *dsigma(x|theta)* in pb with shape *(n_thetas, n_events)*.

`extract_raw_data`(*theta=None*)

Returns all events together with the benchmark weights (if theta is None) or weights for a given theta.

Parameters

theta [None or ndarray, optional] If None, the function returns the benchmark weights. Otherwise it uses morphing to calculate the weights for this value of theta. Default value: None.

Returns

x [ndarray] Observables with shape *(n_unweighted_samples, n_observables)*.

weights [ndarray] If theta is None, benchmark weights with shape *(n_unweighted_samples, n_benchmarks_phys)* in pb. Otherwise, weights for the given parameter theta with shape *(n_unweighted_samples,)* in pb.

`histogram_of_fisher_information`(*theta*, *observable*, *nbins*, *histrange*, *model_file=None*, *luminosity=300000.0*, *cuts=None*, *efficiency_functions=None*, *batch_size=100000*, *test_split=0.5*)

Calculates the full and rate-only Fisher information in slices of one observable. For the full information, it will return the truth-level information if model_file is None, and otherwise the detector-level information based on the SALLY-type score estimator saved in model_file.

Parameters

theta [ndarray] Parameter point *theta* at which the Fisher information matrix *I_ij(theta)* is evaluated.

observable [str] Expression for the observable to be sliced. The str will be parsed by Python's *eval()* function and can use the names of the observables in the MadMiner files.

nbins [int] Number of bins in the slicing, excluding overflow bins.

histrange [tuple of float] Minimum and maximum value of the slicing in the form *(min, max)*. Overflow bins are always added.

model_file [str or None, optional] If None, the truth-level Fisher information is calculated. If str, filename of a trained local score regression model that was trained on samples from *theta* (see *madminer.ml.MLForge*). Default value: None.

luminosity [float, optional] Luminosity in pb^-1. Default value: 300000.

cuts [None or list of str, optional] Cuts. Each entry is a parseable Python expression that returns a bool (True if the event should pass a cut, False otherwise). Default value: None.

efficiency_functions [list of str or None] Efficiencies. Each entry is a parseable Python expression that returns a float for the efficiency of one component. Default value: None.

batch_size [int, optional] If model_file is not None: Batch size. Default value: 100000.

test_split [float or None, optional] If model_file is not None: If unweighted_x_sample_file is None, this determines the fraction of weighted events used for evaluation. If None, all events are used (this will probably include events used during training!). Default value: 0.5.

Returns

bin_boundaries [ndarray] Observable slice boundaries.

sigma_bins [ndarray] Cross section in pb in each of the slices.

fisher_infos_rate [ndarray] Expected rate-only Fisher information for each slice. Has shape *(n_slices, n_parameters, n_parameters)*.

fisher_infos_full [ndarray] Expected full Fisher information for each slice. Has shape *(n_slices, n_parameters, n_parameters)*.

`madminer.fisherinformation.`**`profile_information`**(*fisher_information*, *remaining_components*, *covariance=None*, *error_propagation_n_ensemble=1000*, *error_propagation_factor=0.001*)

Calculates the profiled Fisher information matrix as defined in Appendix A.4 of arXiv:1612.05261.

Parameters

fisher_information [ndarray] Original n x n Fisher information.

remaining_components [list of int] List with m entries, each an int with 0 <= remaining_compoinents[i] < n. Denotes which parameters are kept, and their new order. All other parameters or profiled out.

covariance [ndarray or None, optional] The covariance matrix of the original Fisher information with shape (n, n, n, n). If None, the error on the profiled information is not calculated. Default value: None.

error_propagation_n_ensemble [int, optional] If covariance is not None, this sets the number of Fisher information matrices drawn from a normal distribution for the Monte-Carlo error propagation. Default value: 1000.

error_propagation_factor [float, optional] If covariance is not None, this factor multiplies the covariance of the distribution of Fisher information matrices. Smaller factors can avoid problems with ill-behaved Fisher information matrices. Default value: 1.e-3.

Returns

profiled_fisher_information [ndarray] Profiled m x m Fisher information, where the *i*-th row or column corresponds to the *remaining_components[i]*-th row or column of fisher_information.

profiled_fisher_information_covariance [ndarray] Covariance matrix of the profiled Fishere information matrix with shape (m, m, m, m).

`madminer.fisherinformation.`**`project_information`**(*fisher_information*, *remaining_components*, *covariance=None*)

Calculates projections of a Fisher information matrix, that is, "deletes" the rows and columns corresponding to some parameters not of interest.

Parameters

fisher_information [ndarray] Original n x n Fisher information.

remaining_components [list of int] List with m entries, each an int with 0 <= remaining_compoinents[i] < n. Denotes which parameters are kept, and their new order. All other parameters or projected out.

covariance [ndarray or None, optional] The covariance matrix of the original Fisher information with shape (n, n, n, n). If None, the error on the profiled information is not calculated. Default value: None.

Returns

projected_fisher_information [ndarray] Projected m x m Fisher information, where the *i*-th row or column corresponds to the *remaining_components[i]*-th row or column of fisher_information.

profiled_fisher_information_covariance [ndarray] Covariance matrix of the projected Fisher information matrix with shape (m, m, m, m). Only returned if covariance is not None.

CHAPTER 8

madminer.lhe module

class madminer.lhe.**LHEProcessor**(*filename*)

Bases: object

Detector simulation with smearing functions and simple calculation of observables.

After setting up the parameter space and benchmarks and running MadGraph and Pythia, all of which is organized in the madminer.core.MadMiner class, the next steps are the simulation of detector effects and the calculation of observables. Different tools can be used for these tasks, please feel free to implement the detector simulation and analysis routine of your choice.

This class provides a simple implementation in which detector effects are modeled with smearing functions. Its workflow consists of the following steps:

- Initializing the class with the filename of a MadMiner HDF5 file (the output of *madminer.core.MadMiner.save()*)
- Adding one or multiple event samples produced by MadGraph and Pythia in *LHEProcessor.add_sample()*.
- Running Delphes on the samples that require it through *LHEProcessor.run_delphes()*.
- Optionally, smearing functions for all visible particles can be defined with *LHEProcessor.set_smearing()*.
- Defining observables through *LHEProcessor.add_observable()* or *LHEProcessor.add_observable_from_function()*. A simple set of default observables is provided in *LHEProcessor.add_default_observables()*
- Optionally, cuts can be set with *LHEProcessor.add_cut()*
- Calculating the observables from the Delphes ROOT files with *LHEProcessor.analyse_delphes_samples()*
- Saving the results with *LHEProcessor.save()*

Please see the tutorial for a detailed walk-through.

Parameters

filename [str or None, optional] Path to MadMiner file (the output of *madminer.core.MadMiner.save()*). Default value: None.

Methods

add_cut(definition[, pass_if_not_parsed])	Adds a cut as a string that can be parsed by Python's *eval()* function and returns a bool.
add_default_observables([n_leptons_max, ...])	Adds a set of simple standard observables: the four-momenta (parameterized as E, pT, eta, phi) of the hardest visible particles, and the missing transverse energy.
add_observable(name, definition[, required, ...])	Adds an observable as a string that can be parsed by Python's *eval()* function.
add_observable_from_function(name, fn[, ...])	Adds an observable defined through a function.
add_sample(lhe_filename, sampled_from_benchmark)	Adds an LHE sample of simulated events.
analyse_samples([reference_benchmark, ...])	Main function that parses the LHE samples, applies detector effects, checks cuts, and extracts the observables and weights.
reset_cuts()	Resets all cuts.
reset_observables()	Resets all observables.
save(filename_out)	Saves the observable definitions, observable values, and event weights in a MadMiner file.
set_smearing([pdgids, ...])	Sets up the smearing of measured momenta from shower and detector effects.

add_cut (*definition*, *pass_if_not_parsed=False*)
Adds a cut as a string that can be parsed by Python's *eval()* function and returns a bool.

Parameters

definition [str] An expression that can be parsed by Python's *eval()* function and returns a bool: True for the event to pass this cut, False for it to be rejected. In the definition, all visible particles can be used: *e*, *mu*, *j*, *a*, and *l* provide lists of electrons, muons, jets, photons, and leptons (electrons and muons combined), in each case sorted by descending transverse momentum. *met* provides a missing ET object. *visible* and *all* provide access to the sum of all visible particles and the sum of all visible particles plus MET, respectively. All these objects are instances of *MadMinerParticle*, which inherits from scikit-hep's [LorentzVector](http://scikit-hep.org/api/math.html#vector-classes). See the link for a documentation of their properties. In addition, *MadMinerParticle* have properties *charge* and *pdg_id*, which return the charge in units of elementary charges (i.e. an electron has *e[0].charge = -1.*), and the PDG particle ID. For instance, *"len(e) >= 2"* requires at least two electrons passing the cuts, while *"mu[0].charge > 0."* specifies that the hardest muon is positively charged.

pass_if_not_parsed [bool, optional] Whether the cut is passed if the observable cannot be parsed. Default value: False.

Returns

None

add_default_observables (*n_leptons_max=2*, *n_photons_max=2*, *n_jets_max=2*, *include_met=True*, *include_visible_sum=True*, *include_numbers=True*, *include_charge=True*)
Adds a set of simple standard observables: the four-momenta (parameterized as E, pT, eta, phi) of the hardest visible particles, and the missing transverse energy.

Parameters

n_leptons_max [int, optional] Number of hardest leptons for which the four-momenta are saved. Default value: 2.

n_photons_max [int, optional] Number of hardest photons for which the four-momenta are saved. Default value: 2.

n_jets_max [int, optional] Number of hardest jets for which the four-momenta are saved. Default value: 2.

include_met [bool, optional] Whether the missing energy observables are stored. Default value: True.

include_visible_sum [bool, optional] Whether observables characterizing the sum of all particles are stored. Default value: True.

include_numbers [bool, optional] Whether the number of leptons, photons, and jets is saved as observable. Default value: True.

include_charge [bool, optional] Whether the lepton charge is saved as observable. Default value: True.

Returns

None

`add_observable`(*name*, *definition*, *required=False*, *default=None*)

Adds an observable as a string that can be parsed by Python's *eval()* function.

Parameters

name [str] Name of the observable. Since this name will be used in *eval()* calls for cuts, this should not contain spaces or special characters.

definition [str] An expression that can be parsed by Python's *eval()* function. As objects, all particles can be used: *e*, *mu*, *tau*, *j*, *a*, *l*, *v* provide lists of electrons, muons, taus, jets, photons, leptons (electrons and muons combined), and neutrinos, in each case sorted by descending transverse momentum. *met* provides a missing ET object. *p* gives all particles in the same order as in the LHE file (i.e. in the same order as defined in the MadGraph process card). All these objects are instances of *MadMinerParticle*, which inherits from scikit-hep's [LorentzVector](http://scikit-hep.org/api/math.html#vector-classes). See the link for a documentation of their properties. In addition, *MadMinerParticle* have properties *charge* and *pdg_id*, which return the charge in units of elementary charges (i.e. an electron has *e[0].charge = -1.*), and the PDG particle ID. For instance, *"abs(j[0].phi() - j[1].phi())"* defines the azimuthal angle between the two hardest jets.

required [bool, optional] Whether the observable is required. If True, an event will only be retained if this observable is successfully parsed. For instance, any observable involving *"j[1]"* will only be parsed if there are at least two jets passing the acceptance cuts. Default value: False.

default [float or None, optional] If *required=False*, this is the placeholder value for observables that cannot be parsed. None is replaced with *np.nan*. Default value: None.

Returns

None

`add_observable_from_function`(*name*, *fn*, *required=False*, *default=None*)

Adds an observable defined through a function.

Parameters

name [str] Name of the observable. Since this name will be used in *eval()* calls for cuts, this should not contain spaces or special characters.

fn [function] A function with signature *observable(particles)* where the input arguments are lists of MadMinerParticle instances (ordered in the same way as in the LHE file) and a float is returned. The function should raise a *RuntimeError* to signal that it is not defined.

required [bool, optional] Whether the observable is required. If True, an event will only be retained if this observable is successfully parsed. For instance, any observable involving *"j[1]"* will only be parsed if there are at least two jets passing the acceptance cuts. Default value: False.

default [float or None, optional] If *required=False*, this is the placeholder value for observables that cannot be parsed. None is replaced with *np.nan*. Default value: None.

Returns

None

add_sample(*lhe_filename*, *sampled_from_benchmark*, *is_background=False*, *k_factor=1.0*)
Adds an LHE sample of simulated events.

Parameters

lhe_filename [str] Path to the LHE event file (with extension '.lhe' or '.lhe.gz').

sampled_from_benchmark [str] Name of the benchmark that was used for sampling in this event file (the keyword *sample_benchmark* of *madminer.core.MadMiner.run()*).

is_background [bool, optional] Whether the sample is a background sample (i.e. without benchmark reweighting).

k_factor [float, optional] Multiplies the cross sections found in the sample. Default value: 1.

Returns

None

analyse_samples(*reference_benchmark=None*, *parse_events_as_xml=True*)
Main function that parses the LHE samples, applies detector effects, checks cuts, and extracts the observables and weights.

Parameters

reference_benchmark [str or None, optional] The weights at the nuisance benchmarks will be rescaled to some reference theta benchmark: *dsigma(x|theta_sampling(x),nu) -> dsigma(x|theta_ref,nu) = dsigma(x|theta_sampling(x),nu) * dsigma(x|theta_ref,0) / dsigma(x|theta_sampling(x),0)*. This sets the name of the reference benchmark. If None, the first one will be used. Default value: None.

parse_events_as_xml [bool, optional] Decides whether the LHE events are parsed with an XML parser (more robust, but slower) or a text parser (less robust, faster). Default value: True.

Returns

None

reset_cuts()
Resets all cuts.

reset_observables()
Resets all observables.

`save` (*filename_out*)

Saves the observable definitions, observable values, and event weights in a MadMiner file. The parameter, benchmark, and morphing setup is copied from the file provided during initialization. Nuisance benchmarks found in the LHE file are added.

Parameters

filename_out [str] Path to where the results should be saved.

Returns

None

`set_smearing` (*pdgids=None, energy_resolution_abs=0.0, energy_resolution_rel=0.0, pt_resolution_abs=0.0, pt_resolution_rel=0.0, eta_resolution_abs=0.0, eta_resolution_rel=0.0, phi_resolution_abs=0.0, phi_resolution_rel=0.0*)

Sets up the smearing of measured momenta from shower and detector effects.

This function can be called with pdgids=None, in which case the settinigs are used for all visible particles, or with pdgids set to a list of PDG ids representing particles, for instance [11, -11] for electrons (and positrons).

For all particles of this type, and for the energy, pT, phi, and eta, the measurement error is drawn from a Gaussian with mean 0 and standard deviation given by *(X_resolution_abs + X * X_resolution_rel)*. Here *X* is the quantity (E, pT, phi, eta) of interest and X_resolution_abs and X_resolution_rel are the corresponding keywords. In the case of energy and pT, values smaller than 0 will lead to a re-drawing of the measurement error.

Instead of such numerical values, either the energy or pT resolution (but not both!) may be set to None. In this case, this quantity is calculated from the mass of the particle and all other smeared particles. For instance, when pt_resolution_abs is None or pt_resolution_rel is None, for the given particles the energy, phi, and eta are smeared (according to their respective resolutions). Then the transverse momentum is calculated from the on-shell condition *p^2 = m^2*, or *pT = sqrt(E^2 - m^2) / cosh(eta)*. When this does not have a solution, the pT is set to zero. On the other hand, when energy_resolution_abs is None or energy_resolution_abs is None, for the given particles the pT, phi, and eta are smeared, and then the energy is calculated as *E = sqrt(pT * cosh(eta))^2 + m^2)*.

Parameters

pdgids [None or list of int, optional] Defines the particles these smearing functions affect. If None, all particles are affected. Note that if set_smearing() is called multiple times for a given particle, the earlier calls will be forgotten and only the last smearing function will take effect. Default value: None.

energy_resolution_abs [float or None, optional] Absolute measurement uncertainty for the energy in GeV. None means that the energy is not smeared directly, but calculated from the on-shell condition. Default value: 0.

energy_resolution_rel [float or None, optional] Relative measurement uncertainty for the energy. None means that the energy is not smeared directly, but calculated from the on-shell condition. Default value: 0.

pt_resolution_abs [float or None, optional] Absolute measurement uncertainty for the pT in GeV. None means that the pT is not smeared directly, but calculated from the on-shell condition. Default value: 0.

pt_resolution_rel [float or None, optional] Relative measurement uncertainty for the pT. None means that the pT is not smeared directly, but calculated from the on-shell condition. Default value: 0.

eta_resolution_abs [float, optional] Absolute measurement uncertainty for eta. Default value: 0.

eta_resolution_rel [float, optional] Relative measurement uncertainty for eta. Default value: 0.

phi_resolution_abs [float, optional] Absolute measurement uncertainty for phi. Default value: 0.

phi_resolution_rel [float, optional] Relative measurement uncertainty for phi. Default value: 0.

Returns

None

CHAPTER 9

madminer.ml module

class madminer.ml.**EnsembleForge**(*estimators=None*)

Bases: `object`

Ensemble methods for likelihood ratio and score information.

Generally, EnsembleForge instances can be used very similarly to MLForge instances:

- The initialization of EnsembleForge takes a list of (trained or untrained) MLForge instances.
- The methods *EnsembleForge.train_one()* and *EnsembleForge.train_all()* train the estimators (this can also be done outside of EnsembleForge).
- *EnsembleForge.calculate_expectation()* can be used to calculate the expectation of the estimation likelihood ratio or the expected estimated score over a validation sample. Ideally (and assuming the correct sampling), these expectation values should be close to zero. Deviations from zero therefore point out that the estimator is probably inaccurate.
- *EnsembleForge.evaluate()* and *EnsembleForge.calculate_fisher_information()* can then be used to calculate ensemble predictions. The user has the option to treat all estimators equally ('committee method') or to give those with expected score / ratio close to zero a higher weight.
- *EnsembleForge.save()* and *EnsembleForge.load()* can store all estimators in one folder.

The individual estimators in the ensemble can be trained with different methods, but they have to be of the same type: either all estimators are single-parameterized likelihood ratio estimators, or all estimators are doubly-parameterized likelihood estimators, or all estimators are local score regressors.

Parameters

estimators [None or int or list of (MLForge or str), optional] If int, sets the number of estimators that will be created as new MLForge instances. If list, sets the estimators directly, either from MLForge instances or filenames (that are then loaded with *MLForge.load()*). If None, the ensemble is initialized without estimators. Note that the estimators have to be consistent: either all of them are trained with a local score method ('sally' or 'sallino'); or all of them are trained with a single-parameterized method ('carl', 'rolr', 'rascal', 'scandal', 'alice', or 'alices'); or all of them are trained with a doubly parameterized method ('carl2', 'rolr2', 'rascal2', 'alice2', or 'alices2'). Mixing estimators of different types within one of these three categories is supported, but mixing estimators from different categories is not and will raise a RuntimeException. Default value: None.

Attributes

estimators [list of MLForge] The estimators in the form of MLForge instances.

Methods

add_estimator(estimator)	Adds an estimator to the ensemble.
calculate_expectation(x_filename[, …])	Calculates the expectation of the estimation likelihood ratio or the expected estimated score over a validation sample.
calculate_fisher_information(x[, …])	Calculates expected Fisher information matrices for an ensemble of SALLY estimators.
evaluate(x[, theta0_filename, …])	Evaluates the estimators of the likelihood ratio (or, if method is 'sally' or 'sallino', the score), and calculates the ensemble mean or variance.
load(folder)	Loads the estimator ensemble from a folder.
save(folder[, save_model])	Saves the estimator ensemble to a folder.
train_all(**kwargs)	Trains all estimators.
train_one(i, **kwargs)	Trains an individual estimator.

add_estimator(*estimator*)
Adds an estimator to the ensemble.

Parameters

estimator [MLForge or str] The estimator, either as MLForge instance or filename (which is then loaded with *MLForge.load()*).

Returns

None

calculate_expectation(*x_filename*, *theta0_filename=None*, *theta1_filename=None*)
Calculates the expectation of the estimation likelihood ratio or the expected estimated score over a validation sample. Ideally (and assuming the correct sampling), these expectation values should be close to zero. Deviations from zero therefore point out that the estimator is probably inaccurate.

Parameters

x_filename [str] Path to an unweighted sample of observations, as saved by the *madminer.sampling.SampleAugmenter* functions.

theta0_filename [str or None, optional] Path to an unweighted sample of numerator parameters, as saved by the *madminer.sampling.SampleAugmenter* functions. Required if the estimators were trained with the 'alice', 'alice2', 'alices', 'alices2', 'carl', 'carl2', 'nde', 'rascal', 'rascal2', 'rolr', 'rolr2', or 'scandal' method. Default value: None.

theta1_filename [str or None, optional] Path to an unweighted sample of denominator parameters, as saved by the *madminer.sampling.SampleAugmenter* functions. Required if the estimators were trained with the 'alice2', 'alices2', 'carl2', 'rascal2', or 'rolr2' method. Default value: None.

Returns

expectations [ndarray] Expected score (if the estimators were trained with the 'sally' or 'sallino' methods) or likelihood ratio (otherwise).

calculate_fisher_information(*x*, *obs_weights=None*, *n_events=1*, *mode='score'*, *uncertainty='ensemble'*, *vote_expectation_weight=None*, *return_individual_predictions=False*, *sum_events=True*)

Calculates expected Fisher information matrices for an ensemble of SALLY estimators.

There are two ways of calculating the ensemble average. In the default "score" mode, the ensemble average for the score is calculated for each event, and the Fisher information is calculated based on these mean scores. In the "information" mode, the Fisher information is calculated for each estimator separately and the ensemble mean is calculated only for the final Fisher information matrix. The "score" mode is generally assumed to be more precise and is the default.

In the "score" mode, the covariance matrix of the final result is calculated in the following way: - For each event *x* and each estimator *a*, the "shifted" predicted score is calculated as

*t_a'(x) = t(x) + 1/sqrt(n) * (t_a(x) - t(x))*. Here *t(x)* is the mean score (averaged over the ensemble) for this event, *t_a(x)* is the prediction of estimator *a* for this event, and *n* is the number of estimators. The ensemble variance of these shifted score predictions is equal to the uncertainty on the mean of the ensemble of original predictions.

- For each estimator *a*, the shifted Fisher information matrix *I_a'* is calculated from the shifted predicted scores.
- The ensemble covariance between all Fisher information matrices *I_a'* is calculated and taken as the measure of uncertainty on the Fisher information calculated from the mean scores.

In the "information" mode, the user has the option to treat all estimators equally ('committee method') or to give those with expected score close to zero (as calculated by *calculate_expectation()*) a higher weight. In this case, the ensemble mean *I* is calculated as *I = sum_i w_i I_i* with weights *w_i = exp(-vote_expectation_weight |E[t_i]|) / sum_j exp(-vote_expectation_weight |E[t_k]|)*. Here *I_i* are the individual estimators and *E[t_i]* is the expectation value calculated by *calculate_expectation()*.

Parameters

x [str or ndarray] Sample of observations, or path to numpy file with observations, as saved by the *madminer.sampling.SampleAugmenter* functions. Note that this sample has to be sampled from the reference parameter where the score is estimated with the SALLY / SALLINO estimator!

obs_weights [None or ndarray, optional] Weights for the observations. If None, all events are taken to have equal weight. Default value: None.

n_events [float, optional] Expected number of events for which the kinematic Fisher information should be calculated. Default value: 1.

mode [{"score", "information"}, optional] If mode is "information", the Fisher information for each estimator is calculated individually and only then are the sample mean and covariance calculated. If mode is "score", the sample mean is calculated for the score for each event. Default value: "score".

uncertainty [{"ensemble", "expectation", "sum", "none"}, optional] How the covariance matrix of the Fisher information estimate is calculate. With "ensemble", the ensemble covariance is used (only supported if mode is "information"). With "expectation", the expectation of the score is used as a measure of the uncertainty of the score estimator, and this uncertainty is propagated through to the covariance matrix. With "sum", both terms are summed (only supported if mode is "information"). With "none", no uncertainties are calculated. Default value: "ensemble".

vote_expectation_weight [float or list of float or None, optional] If mode is "information", this factor determines how much more weight is given to those estimators with small expectation value (as calculated by *calculate_expectation()*). If a list is given, results are returned for each element in the list. If None, or if *calculate_expectation()* has not been called, all estimators are treated equal. Default value: None.

return_individual_predictions [bool, optional] If mode is "information", sets whether the individual estimator predictions are returned. Default value: False.

sum_events [bool, optional] If True or mode is "information", the expected Fisher information summed over the events x is calculated. If False and mode is "score", the per-event Fisher information for each event is returned. Default value: True.

Returns

mean_prediction [ndarray or list of ndarray] Expected kinematic Fisher information matrix with shape *(n_events, n_parameters, n_parameters)* if sum_events is False and mode is "score", or *(n_parameters, n_parameters)* in any other case.

covariance [ndarray or list of ndarray] The covariance matrix of the Fisher information estimate. Its definition depends on the value of uncertainty; by default, the covariance is defined as the ensemble covariance (only supported if mode is "information"). This object has four indices, *cov_(ij)(i'j')*, ordered as i j i' j'. It has shape *(n_parameters, n_parameters, n_parameters, n_parameters)*. If more then one value vote_expectation_weight is given, this is a list with results for all entries in vote_expectation_weight.

weights [ndarray or list of ndarray] Only returned if return_individual_predictions is True. The estimator weights *w_i*. If more then one value vote_expectation_weight is given, this is a list with results for all entries in vote_expectation_weight.

individual_predictions [ndarray] Only returned if return_individual_predictions is True. The individual estimator predictions.

`evaluate`(*x*, *theta0_filename=None*, *theta1_filename=None*, *test_all_combinations=True*, *vote_expectation_weight=None*, *calculate_covariance=False*, *return_individual_predictions=False*)

Evaluates the estimators of the likelihood ratio (or, if method is 'sally' or 'sallino', the score), and calculates the ensemble mean or variance.

The user has the option to treat all estimators equally ('committee method') or to give those with expected score / ratio close to zero (as calculated by *calculate_expectation()*) a higher weight. In the latter case, the ensemble mean *f(x)* is calculated as *f(x) = sum_i w_i f_i(x)* with weights *w_i = exp(-vote_expectation_weight |E[f_i]|) / sum_j exp(-vote_expectation_weight |E[f_j]|)*. Here *f_i(x)* are the individual estimators and *E[f_i]* is the expectation value calculated by *calculate_expectation()*.

Parameters

x [str or ndarray] Sample of observations, or path to numpy file with observations, as saved by the *madminer.sampling.SampleAugmenter* functions. Note that this sample has to be sampled from the reference parameter where the score is estimated with the SALLY / SALLINO estimator!

theta0_filename [str or None, optional] Path to an unweighted sample of numerator parameters, as saved by the *madminer.sampling.SampleAugmenter* functions. Required if the estimator was trained with the 'alice', 'alice2', 'alices', 'alices2', 'carl', 'carl2', 'nde', 'rascal', 'rascal2', 'rolr', 'rolr2', or 'scandal' method. Default value: None.

theta1_filename [str or None, optional] Path to an unweighted sample of denominator parameters, as saved by the *madminer.sampling.SampleAugmenter* functions. Required if the estimator was trained with the 'alice2', 'alices2', 'carl2', 'rascal2', or 'rolr2' method. Default value: None.

test_all_combinations [bool, optional] If method is not 'sally' and not 'sallino': If False, the number of samples in the observable and theta files has to match, and the likelihood ratio is evaluated only for the combinations *r(x_i | theta0_i, theta1_i)*. If True, *r(x_i | theta0_j, theta1_j)* for all pairwise combinations *i, j* are evaluated. Default value: True.

vote_expectation_weight [float or list of float or None, optional] Factor that determines how much more weight is given to those estimators with small expectation value (as calculated by *calculate_expectation()*). If a list is given, results are returned for each element in the list. If None, or if *calculate_expectation()* has not been called, all estimators are treated equal. Default value: None.

calculate_covariance [bool, optional] Whether the covariance matrix is calculated. Default value: False.

return_individual_predictions [bool, optional] Whether the individual estimator predictions are returned. Default value: False.

Returns

mean_prediction [ndarray or list of ndarray] The (weighted) ensemble mean of the estimators. If the estimators were trained with *method='sally'* or *method='sallino'*, this is an array of the estimator for *t(x_i | theta_ref)* for all events *i*. Otherwise, the estimated likelihood ratio (if test_all_combinations is True, the result has shape *(n_thetas, n_x)*, otherwise, it has shape *(n_samples,)*). If more then one value vote_expectation_weight is given, this is a list with results for all entries in vote_expectation_weight.

covariance [None or ndarray or list of ndarray] The covariance matrix of the (flattened) predictions, defined as the ensemble covariance. If more then one value vote_expectation_weight is given, this is a list with results for all entries in vote_expectation_weight. If calculate_covariance is False, None is returned.

weights [ndarray or list of ndarray] Only returned if return_individual_predictions is True. The estimator weights *w_i*. If more then one value vote_expectation_weight is given, this is a list with results for all entries in vote_expectation_weight.

individual_predictions [ndarray] Only returned if return_individual_predictions is True. The individual estimator predictions.

`load`(*folder*)

Loads the estimator ensemble from a folder.

Parameters

folder [str] Path to the folder.

Returns

None

`save`(*folder*, *save_model=False*)

Saves the estimator ensemble to a folder.

Parameters

folder [str] Path to the folder.

save_model [bool, optional] If True, the whole model is saved in addition to the state dict. This is not necessary for loading it again with EnsembleForge.load(), but can be useful for debugging, for instance to plot the computational graph.

Returns

None

train_all(***kwargs*)

Trains all estimators. See *MLForge.train()*.

Parameters

kwargs [dict] Parameters for *MLForge.train()*. If a value in this dict is a list, it has to have length *n_estimators* and contain one value of this parameter for each of the estimators. Otherwise the value is used as parameter for the training of all the estimators.

Returns

None

train_one(*i*, ***kwargs*)

Trains an individual estimator. See *MLForge.train()*.

Parameters

i [int] The index *0 <= i < n_estimators* of the estimator to be trained.

kwargs [dict] Parameters for *MLForge.train()*.

Returns

None

class madminer.ml.**MLForge**

Bases: `object`

Estimating likelihood ratios and scores with machine learning.

Each instance of this class represents one neural estimator. The most important functions are:

- *MLForge.train()* to train an estimator. The keyword *method* determines the inference technique and whether a class instance represents a single-parameterized likelihood ratio estimator, a doubly-parameterized likelihood ratio estimator, or a local score estimator.
- *MLForge.evaluate()* to evaluate the estimator.
- *MLForge.save()* to save the trained model to files.
- *MLForge.load()* to load the trained model from files.

Please see the tutorial for a detailed walk-through.

Methods

calculate_fisher_information(x[, weights, ...])	Calculates the expected Fisher information matrix based on the kinematic information in a given number of events.
evaluate(x[, theta0_filename, ...])	Evaluates a trained estimator of the log likelihood ratio, the log likelihood, or the score, depending on the method.
evaluate_log_likelihood(x[, ...])	Evaluates a trained estimator of the log likelihood.
evaluate_log_likelihood_ratio(x[, ...])	Evaluates a trained estimator of the log likelihood ratio, the log likelihood, or the score, depending on the method.

Continued on next page

Table 2 – continued from previous page

`evaluate_score`(x[, return_grad_x])	Evaluates a trained estimator of the the score.
`load`(filename)	Loads a trained model from files.
`save`(filename[, save_model])	Saves the trained model to four files: a JSON file with the settings, a pickled pyTorch state dict file, and numpy files for the mean and variance of the inputs (used for input scaling).
`train`(method, x_filename[, y_filename, . . .])	Trains a neural network to estimate either the likelihood, the likelihood ratio, or the score.

`calculate_fisher_information`(*x*, *weights=None*, *n_events=1*, *sum_events=True*)

Calculates the expected Fisher information matrix based on the kinematic information in a given number of events. Currently only supported for estimators trained with *method='sally'* or *method='sallino'*.

Parameters

x [str or ndarray] Sample of observations, or path to numpy file with observations, as saved by the *madminer.sampling.SampleAugmenter* functions. Note that this sample has to be sampled from the reference parameter where the score is estimated with the SALLY / SALLINO estimator!

weights [None or ndarray, optional] Weights for the observations. If None, all events are taken to have equal weight. Default value: None.

n_events [float, optional] Expected number of events for which the kinematic Fisher information should be calculated. Default value: 1.

sum_events [bool, optional] If True, the expected Fisher information summed over the events x is calculated. If False, the per-event Fisher information for each event is returned. Default value: True.

Returns

fisher_information [ndarray] Expected kinematic Fisher information matrix with shape *(n_events, n_parameters, n_parameters)* if sum_events is False or *(n_parameters, n_parameters)* if sum_events is True.

`evaluate`(*x*, *theta0_filename=None*, *theta1_filename=None*, *test_all_combinations=True*, *evaluate_score=False*)

Evaluates a trained estimator of the log likelihood ratio, the log likelihood, or the score, depending on the method.

Parameters

x [str or ndarray] Sample of observations, or path to numpy file with observations, as saved by the *madminer.sampling.SampleAugmenter* functions.

theta0_filename [str or None, optional] Path to an unweighted sample of numerator parameters, as saved by the *madminer.sampling.SampleAugmenter* functions. Required if the estimator was trained with the 'alice', 'alice2', 'alices', 'alices2', 'carl', 'carl2', 'nde', 'rascal', 'rascal2', 'rolr', 'rolr2', or 'scandal' method. Default value: None.

theta1_filename [str or None, optional] Path to an unweighted sample of denominator parameters, as saved by the *madminer.sampling.SampleAugmenter* functions. Required if the estimator was trained with the 'alice2', 'alices2', 'carl2', 'rascal2', or 'rolr2' method. Default value: None.

test_all_combinations [bool, optional] If method is not 'sally' and not 'sallino': If False, the number of samples in the observable and theta files has to match, and the likelihood ratio is evaluated only for the combinations *r(x_i | theta0_i, theta1_i)*. If True, *r(x_i | theta0_j, theta1_j)* for all pairwise combinations *i, j* are evaluated. Default value: True.

evaluate_score [bool, optional] If method is not 'sally' and not 'sallino', this sets whether in addition to the likelihood ratio the score is evaluated. Default value: False.

return_grad_x [bool, optional] If True, *grad_x log r(x)* or *grad_x t(x)* (for 'sally' or 'sallino' estimators) are returned in addition to the other outputs. Default value: False.

Returns

sally_estimated_score [ndarray] Only returned if the network was trained with *method='sally'* or *method='sallino'*. In this case, an array of the estimator for *t(x_i | theta_ref)* is returned for all events *i*.

log_likelihood_ratio [ndarray] Only returned if the network was trained with neither *method='sally'* nor *method='sallino'*. The estimated log likelihood ratio. If test_all_combinations is True, the result has shape *(n_thetas, n_x)*. Otherwise, it has shape *(n_samples,)*.

score_theta0 [ndarray or None] Only returned if the network was trained with neither *method='sally'* nor *method='sallino'*. None if evaluate_score is False. Otherwise the derived estimated score at *theta0*. If test_all_combinations is True, the result has shape *(n_thetas, n_x, n_parameters)*. Otherwise, it has shape *(n_samples, n_parameters)*.

score_theta1 [ndarray or None] Only returned if the network was trained with neither *method='sally'* nor *method='sallino'*. None if evaluate_score is False, or the network was trained with any method other than 'alice2', 'alices2', 'carl2', 'rascal2', or 'rolr2'. Otherwise the derived estimated score at *theta1*. If test_all_combinations is True, the result has shape *(n_thetas, n_x, n_parameters)*. Otherwise, it has shape *(n_samples, n_parameters)*.

grad_x [ndarray] Only returned if return_grad_x is True.

`evaluate_log_likelihood`(*x*, *theta0_filename=None*, *test_all_combinations=True*, *evaluate_score=False*)

Evaluates a trained estimator of the log likelihood.

Parameters

x [str or ndarray] Sample of observations, or path to numpy file with observations, as saved by the *madminer.sampling.SampleAugmenter* functions.

theta0_filename [str or None, optional] Path to an unweighted sample of numerator parameters, as saved by the *madminer.sampling.SampleAugmenter* functions. Required if the estimator was trained with the 'alice', 'alice2', 'alices', 'alices2', 'carl', 'carl2', 'nde', 'rascal', 'rascal2', 'rolr', 'rolr2', or 'scandal' method. Default value: None.

test_all_combinations [bool, optional] If method is not 'sally' and not 'sallino': If False, the number of samples in the observable and theta files has to match, and the likelihood ratio is evaluated only for the combinations *r(x_i | theta0_i, theta1_i)*. If True, *r(x_i | theta0_j, theta1_j)* for all pairwise combinations *i, j* are evaluated. Default value: True.

evaluate_score [bool, optional] If method is not 'sally' and not 'sallino', this sets whether in addition to the likelihood ratio the score is evaluated. Default value: False.

Returns

log_likelihood [ndarray] The estimated log likelihood. If test_all_combinations is True, the result has shape *(n_thetas, n_x)*. Otherwise, it has shape *(n_samples,)*.

score_theta0 [ndarray or None] None if evaluate_score is False. Otherwise the derived estimated score at *theta0*. If test_all_combinations is True, the result has shape *(n_thetas, n_x, n_parameters)*. Otherwise, it has shape *(n_samples, n_parameters)*.

evaluate_log_likelihood_ratio(*x*, *theta0_filename=None*, *theta1_filename=None*, *test_all_combinations=True*, *evaluate_score=False*)

Evaluates a trained estimator of the log likelihood ratio, the log likelihood, or the score, depending on the method.

Parameters

x [str or ndarray] Sample of observations, or path to numpy file with observations, as saved by the *madminer.sampling.SampleAugmenter* functions.

theta0_filename [str or None, optional] Path to an unweighted sample of numerator parameters, as saved by the *madminer.sampling.SampleAugmenter* functions. Required if the estimator was trained with the 'alice', 'alice2', 'alices', 'alices2', 'carl', 'carl2', 'nde', 'rascal', 'rascal2', 'rolr', 'rolr2', or 'scandal' method. Default value: None.

theta1_filename [str or None, optional] Path to an unweighted sample of denominator parameters, as saved by the *madminer.sampling.SampleAugmenter* functions. Required if the estimator was trained with the 'alice2', 'alices2', 'carl2', 'rascal2', or 'rolr2' method. Default value: None.

test_all_combinations [bool, optional] If method is not 'sally' and not 'sallino': If False, the number of samples in the observable and theta files has to match, and the likelihood ratio is evaluated only for the combinations *r(x_i | theta0_i, theta1_i)*. If True, *r(x_i | theta0_j, theta1_j)* for all pairwise combinations *i, j* are evaluated. Default value: True.

evaluate_score [bool, optional] If method is not 'sally' and not 'sallino', this sets whether in addition to the likelihood ratio the score is evaluated. Default value: False.

Returns

log_likelihood_ratio [ndarray] Only returned if the network was trained with neither *method='sally'* nor *method='sallino'*. The estimated log likelihood ratio. If test_all_combinations is True, the result has shape *(n_thetas, n_x)*. Otherwise, it has shape *(n_samples,)*.

score_theta0 [ndarray or None] Only returned if the network was trained with neither *method='sally'* nor *method='sallino'*. None if evaluate_score is False. Otherwise the derived estimated score at *theta0*. If test_all_combinations is True, the result has shape *(n_thetas, n_x, n_parameters)*. Otherwise, it has shape *(n_samples, n_parameters)*.

score_theta1 [ndarray or None] Only returned if the network was trained with neither *method='sally'* nor *method='sallino'*. None if evaluate_score is False, or the network was trained with any method other than 'alice2', 'alices2', 'carl2', 'rascal2', or 'rolr2'. Otherwise the derived estimated score at *theta1*. If test_all_combinations is True, the result has shape *(n_thetas, n_x, n_parameters)*. Otherwise, it has shape *(n_samples, n_parameters)*.

evaluate_score(*x*, *return_grad_x=False*)

Evaluates a trained estimator of the the score.

Parameters

x [str or ndarray] Sample of observations, or path to numpy file with observations, as saved by the *madminer.sampling.SampleAugmenter* functions.

return_grad_x [bool, optional] If True, *grad_x log r(x)* or *grad_x t(x)* (for 'sally' or 'sallino' estimators) are returned in addition to the other outputs. Default value: False.

Returns

sally_estimated_score [ndarray] Only returned if the network was trained with *method='sally'* or *method='sallino'*. In this case, an array of the estimator for *t(x_i | theta_ref)* is returned for all events *i*.

grad_x [ndarray] Only returned if return_grad_x is True.

load (*filename*)
Loads a trained model from files.

Parameters

filename [str] Path to the files. '_settings.json' and '_state_dict.pl' will be added.

Returns

None

save (*filename*, *save_model=False*)
Saves the trained model to four files: a JSON file with the settings, a pickled pyTorch state dict file, and numpy files for the mean and variance of the inputs (used for input scaling).

Parameters

filename [str] Path to the files. '_settings.json' and '_state_dict.pl' will be added.

save_model [bool, optional] If True, the whole model is saved in addition to the state dict. This is not necessary for loading it again with MLForge.load(), but can be useful for debugging, for instance to plot the computational graph.

Returns

None

train (*method*, *x_filename*, *y_filename=None*, *theta0_filename=None*, *theta1_filename=None*, *r_xz_filename=None*, *t_xz0_filename=None*, *t_xz1_filename=None*, *features=None*, *nde_type='mafmog'*, *n_hidden=(100*, *100)*, *activation='tanh'*, *maf_n_mades=3*, *maf_batch_norm=False*, *maf_batch_norm_alpha=0.1*, *maf_mog_n_components=10*, *alpha=1.0*, *optimizer='amsgrad'*, *n_epochs=50*, *batch_size=200*, *initial_lr=0.001*, *final_lr=0.0001*, *nesterov_momentum=None*, *validation_split=0.25*, *early_stopping=True*, *scale_inputs=True*, *shuffle_labels=False*, *grad_x_regularization=None*, *limit_samplesize=None*, *verbose='some'*)
Trains a neural network to estimate either the likelihood, the likelihood ratio, or the score.

The keyword method determines the structure of the estimator that an instance of this class represents:

- For 'alice', 'alices', 'carl', 'nde', 'rascal', 'rolr', and 'scandal', the neural network models the likelihood ratio as a function of the observables *x* and the numerator hypothesis *theta0*, while the denominator hypothesis is kept at a fixed reference value ("single-parameterized likelihood ratio estimator"). In addition to the likelihood ratio, the estimator allows to estimate the score at *theta0*.
- For 'alice2', 'alices2', 'carl2', 'rascal2', and 'rolr2', the neural network models the likelihood ratio as a function of the observables *x*, the numerator hypothesis *theta0*, and the denominator hypothesis *theta1* ("doubly parameterized likelihood ratio estimator"). The score at *theta0* and *theta1* can also be evaluated.
- For 'sally' and 'sallino', the neural networks models the score evaluated at some reference hypothesis ("local score regression"). The likelihood ratio cannot be estimated directly from the neural network, but can be estimated in a second step through density estimation in the estimated score space.

Parameters

method [str] The inference method used. Allows values are 'alice', 'alices', 'carl', 'nde', 'rascal', 'rolr', and 'scandal' for a single-parameterized likelihood ratio estimator; 'alice2', 'alices2', 'carl2', 'rascal2', and 'rolr2' for a doubly-parameterized likelihood ratio estimator; and 'sally' and 'sallino' for local score regression.

x_filename [str] Path to an unweighted sample of observations, as saved by the *madminer.sampling.SampleAugmenter* functions. Required for all inference methods.

y_filename [str or None, optional] Path to an unweighted sample of class labels, as saved by the *madminer.sampling.SampleAugmenter* functions. Required for the 'alice', 'alice2', 'alices', 'alices2', 'carl', 'carl2', 'rascal', 'rascal2', 'rolr', and 'rolr2' methods. Default value: None.

theta0_filename [str or None, optional] Path to an unweighted sample of numerator parameters, as saved by the *madminer.sampling.SampleAugmenter* functions. Required for the 'alice', 'alice2', 'alices', 'alices2', 'carl', 'carl2', 'nde', 'rascal', 'rascal2', 'rolr', 'rolr2', and 'scandal' methods. Default value: None.

theta1_filename [str or None, optional] Path to an unweighted sample of denominator parameters, as saved by the *madminer.sampling.SampleAugmenter* functions. Required for the 'alice2', 'alices2', 'carl2', 'rascal2', and 'rolr2' methods. Default value: None.

r_xz_filename [str or None, optional] Path to an unweighted sample of joint likelihood ratios, as saved by the *madminer.sampling.SampleAugmenter* functions. Required for the 'alice', 'alice2', 'alices', 'alices2', 'rascal', 'rascal2', 'rolr', and 'rolr2' methods. Default value: None.

t_xz0_filename [str or None, optional] Path to an unweighted sample of joint scores at theta0, as saved by the *madminer.sampling.SampleAugmenter* functions. Required for the 'alices', 'alices2', 'rascal', 'rascal2', 'sallino', 'sally', and 'scandal' methods. Default value: None.

t_xz1_filename [str or None, optional] Path to an unweighted sample of joint scores at theta1, as saved by the *madminer.sampling.SampleAugmenter* functions. Required for the 'rascal2' and 'alices2' methods. Default value: None.

features [list of int or None, optional] Indices of observables (features) that are used as input to the neural networks. If None, all observables are used. Default value: None.

nde_type [{'maf', 'mafmog'}, optional] If the method is 'nde' or 'scandal', nde_type determines the architecture used in the neural density estimator. Currently supported are 'maf' for a Masked Autoregressive Flow with a Gaussian base density, or 'mafmog' for a Masked Autoregressive Flow with a mixture of Gaussian base densities. Default value: 'mafmog'.

n_hidden [tuple of int, optional] Units in each hidden layer in the neural networks. If method is 'nde' or 'scandal', this refers to the setup of each individual MADE layer. Default value: (100, 100).

activation [{'tanh', 'sigmoid', 'relu'}, optional] Activation function. Default value: 'tanh'.

maf_n_mades [int, optional] If method is 'nde' or 'scandal', this sets the number of MADE layers. Default value: 3.

maf_batch_norm [bool, optional] If method is 'nde' or 'scandal', switches batch normalization layers after each MADE layer on or off. Default: False.

maf_batch_norm_alpha [float, optional] If method is 'nde' or 'scandal' and maf_batch_norm is True, this sets the alpha parameter in the calculation of the running average of the mean and variance. Default value: 0.1.

maf_mog_n_components [int, optional] If method is 'nde' or 'scandal' and nde_type is 'mafmog', this sets the number of Gaussian base components. Default value: 10.

alpha [float, optional] Hyperparameter weighting the score error in the loss function of the 'alices', 'alices2', 'rascal', 'rascal2', and 'scandal' methods. Default value: 1.

optimizer [{"adam", "amsgrad", "sgd"}, optional] Optimization algorithm. Default value: "amsgrad".

n_epochs [int, optional] Number of epochs. Default value: 50.

batch_size [int, optional] Batch size. Default value: 200.

initial_lr [float, optional] Learning rate during the first epoch, after which it exponentially decays to final_lr. Default value: 0.001.

final_lr [float, optional] Learning rate during the last epoch. Default value: 0.0001.

nesterov_momentum [float or None, optional] If trainer is "sgd", sets the Nesterov momentum. Default value: None.

validation_split [float or None, optional] Fraction of samples used for validation and early stopping (if early_stopping is True). If None, the entire sample is used for training and early stopping is deactivated. Default value: 0.25.

early_stopping [bool, optional] Activates early stopping based on the validation loss (only if validation_split is not None). Default value: True.

scale_inputs [bool, optional] Scale the observables to zero mean and unit variance. Default value: True.

shuffle_labels [bool, optional] If True, the labels (*y*, *r_xz*, *t_xz*) are shuffled, while the observations (*x*) remain in their normal order. This serves as a closure test, in particular as cross-check against overfitting: an estimator trained with shuffle_labels=True should predict to likelihood ratios around 1 and scores around 0.

grad_x_regularization [None] Currently not supported.

limit_samplesize [int or None, optional] If not None, only this number of samples (events) is used to train the estimator. Default value: None.

verbose [{"all", "many", "some", "few", "none}, optional] Determines verbosity of training. Default value: "some".

Returns

None

CHAPTER 10

madminer.morphing module

class madminer.morphing.**Morpher**(*parameters_from_madminer=None*, *parameter_max_power=None*, *parameter_range=None*)

Bases: `object`

Morphing functionality for theory parameters. Morphing is a technique that allows MadMax to infer the full probability distribution *p(x_i | theta)* for each simulated event *x_i* and any *theta*, not just the benchmarks.

For a typical MadMiner application, it is not necessary to use the morphing classes directly. The other MadMiner classes use the morphing functions "under the hood" when needed. Only for an isolated study of the morphing setup (e.g. to optimize the morphing basis), the Morpher class itself may be of interest.

A typical morphing basis setup involves the following steps:

- The instance of the class is initialized with the parameter setup. The user can provide the parameters either in the format of *MadMiner.parameters*. Alternatively, human-friendly lists of the key properties can be provided.
- The function *find_components* can be used to find the relevant components, i.e. individual terms contributing to the squared matrix elements (alternatively they can be defined by the user with *set_components()*).
- The final step is the definition of the morphing basis, i.e. the benchmark points for which the squared matrix element will be evaluated before interpolating to other parameter points. Again the user can pick this basis manually with *set_basis()*. Alternatively, this class provides a basic optimization routine for the basis choice in *optimize_basis()*.

The class also provides helper functions that are important for working with morphing:

- ***calculate_morphing_matrix()* calculates the morphing matrix, i.e. the matrix that links the morphing basis to the**
  components.
- *calculate_morphing_weights()* calculates the morphing weights *w_b(theta)* for a given parameter point *theta* such that *p(theta) = sum_b w_b(theta) p(theta_b)*.
- *calculate_morphing_weight_gradient()* calculates the gradient of the morphing weights, *grad_theta w_b(theta)*.

Note that this class only implements the "theory morphing" (or, more specifically, "EFT morphing") of the physics parameters of interest. Nuisance parameter morphing is implemented in the NuisanceMorpher class.

Parameters

parameters_from_madminer [OrderedDict or None, optional] Parameters in the *MadMiner.parameters* convention. OrderedDict with keys equal to the parameter names and values equal to tuples (LHA_block, LHA_ID, morphing_max_power, param_min, param_max)

parameter_max_power [None or list of int or list of tuple of int, optional] Only used if parameters_from_madminer is None. Maximal power with which each parameter contributes to the squared matrix element. If tuples are given, gives this maximal power for each of several operator configurations. Typically at tree level, this maximal number is 2 for parameters that affect one vertex (e.g. only production or only decay of a particle), and 4 for parameters that affect two vertices (e.g. production and decay).

parameter_range [None or list of tuple of float, optional] Only used if parameters_from_madminer is None. Parameter range (param_min, param_max) for each parameter.

Methods

`calculate_morphing_matrix`([basis])	Calculates the morphing matrix that links the components to the basis benchmarks.
`calculate_morphing_weight_gradient`(theta[, ...])	Calculates the gradient of the morphing weights, *grad_i w_b(theta)*.
`calculate_morphing_weights`(theta[, basis, ...])	Calculates the morphing weights *w_b(theta)* for a given morphing basis *{theta_b}*.
`evaluate_morphing`([basis, morphing_matrix, ...])	Evaluates the expected sum of the squared morphing weights for a given basis.
`find_components`([max_overall_power])	Finds the components, i.e.
`optimize_basis`([n_bases, ...])	Optimizes the morphing basis.
`set_basis`([basis_from_madminer, ...])	Manually sets the basis benchmarks.
`set_components`(components)	Manually defines the components, i.e.

calculate_morphing_matrix(*basis=None*)
Calculates the morphing matrix that links the components to the basis benchmarks.

Parameters

basis [ndarray or None, optional] Manually specified morphing basis for which the morphing matrix is calculated. This array has shape *(n_basis_benchmarks, n_parameters)*. If None, the basis from the last call of *set_basis()* or *find_basis()* is used. Default value: None.

Returns

morphing_matrix [ndarray] Morphing matrix with shape *(n_basis_benchmarks, n_components)*

calculate_morphing_weight_gradient(*theta*, *basis=None*, *morphing_matrix=None*)
Calculates the gradient of the morphing weights, *grad_i w_b(theta)*.

Parameters

theta [ndarray] Parameter point *theta* with shape *(n_parameters,)*.

basis [ndarray or None, optional] Manually specified morphing basis for which the weights are calculated. This array has shape *(n_basis_benchmarks, n_parameters)*. If None, the basis from the last call of *set_basis()* or *find_basis()* is used. Default value: None.

morphing_matrix [ndarray or None, optional] Manually specified morphing matrix for the given morphing basis. This array has shape *(n_basis_benchmarks, n_components)*. If None, the morphing matrix is calculated automatically. Default value: None.

Returns

morphing_weight_gradients [ndarray] Morphing weights as an array with shape *(n_parameters, n_basis_benchmarks,)*, where the first component refers to the gradient direction.

`calculate_morphing_weights` (*theta*, *basis=None*, *morphing_matrix=None*)
Calculates the morphing weights *w_b(theta)* for a given morphing basis *{theta_b}*.

Parameters

theta [ndarray] Parameter point *theta* with shape *(n_parameters,)*.

basis [ndarray or None, optional] Manually specified morphing basis for which the weights are calculated. This array has shape *(n_basis_benchmarks, n_parameters)*. If None, the basis from the last call of *set_basis()* or *find_basis()* is used. Default value: None.

morphing_matrix [ndarray or None, optional] Manually specified morphing matrix for the given morphing basis. This array has shape *(n_basis_benchmarks, n_components)*. If None, the morphing matrix is calculated automatically. Default value: None.

Returns

morphing_weights [ndarray] Morphing weights as an array with shape *(n_basis_benchmarks,)*.

`evaluate_morphing` (*basis=None*, *morphing_matrix=None*, *n_test_thetas=100*, *return_weights_and_thetas=False*)
Evaluates the expected sum of the squared morphing weights for a given basis.

Parameters

basis [ndarray or None, optional] Manually specified morphing basis for which the weights are calculated. This array has shape *(n_basis_benchmarks, n_parameters)*. If None, the basis from the last call of *set_basis()* or *find_basis()* is used. Default value: None.

morphing_matrix [ndarray or None, optional] Manually specified morphing matrix for the given morphing basis. This array has shape *(n_basis_benchmarks, n_components)*. If None, the morphing matrix is calculated automatically. Default value: None.

n_test_thetas [int, optional] Number of random parameter points used to evaluate the expected mean squared morphing weights. A larger number will increase the run time of the optimization, but lead to better results. Default value: 100.

return_weights_and_thetas [bool, optional] If True, results for each evaluation theta are returned, rather than taking their average. Default value: False.

Returns

thetas_test [ndarray] Random parameter points used for evaluation. Only returned if *return_weights_and_thetas=True* is used.

squared_weights [ndarray] Squared summed morphing weights at each evaluation parameter point. Only returned if *return_weights_and_thetas=True* is used.

negative_expected_sum_squared_weights [float] Negative expected sum of the square of the morphing weights. Objective function in the optimization. Only returned with *return_weights_and_thetas=False*.

find_components(*max_overall_power=4*)

Finds the components, i.e. the individual terms contributing to the squared matrix element.

Parameters

max_overall_power [int or tuple of int, optional] The maximal sum of powers of all parameters contributing to the squared matrix element. If a tuple is given, gives the maximal sum of powers for each of several operator configurations (see constructor). Typically, if parameters can affect the couplings at n vertices, this number is 2n. Default value: 4.

Returns

components [ndarray] Array with shape (n_components, n_parameters), where each entry gives the power with which a parameter scales a given component.

optimize_basis(*n_bases=1*, *fixed_benchmarks_from_madminer=None*, *fixed_benchmarks_numpy=None*, *n_trials=100*, *n_test_thetas=100*)

Optimizes the morphing basis. If either fixed_benchmarks_from_maxminer or fixed_benchmarks_numpy are not None, then these will be used as fixed basis points and only the remaining part of the basis will be optimized.

Parameters

n_bases [int, optional] The number of morphing bases generated. If n_bases > 1, multiple bases are combined, and the weights for each basis are reduced by a factor 1 / n_bases. Currently only the default choice of 1 is fully implemented. Do not use any other value for now. Default value: 1.

fixed_benchmarks_from_madminer [OrderedDict or None, optional] Input basis vectors in the *MadMiner.benchmarks* conventions. Default value: None.

fixed_benchmarks_numpy [ndarray or None, optional] Input basis vectors as a ndarray with shape *(n_fixed_basis_points, n_parameters)*. Default value: None.

n_trials [int, optional] Number of random basis configurations tested in the optimization procedure. A larger number will increase the run time of the optimization, but lead to better results. Default value: 100.

n_test_thetas [int, optional] Number of random parameter points used to evaluate the expected mean squared morphing weights. A larger number will increase the run time of the optimization, but lead to better results. Default value: 100.

Returns

basis [OrderedDict or ndarray] Optimized basis in the same format (MadMiner or numpy) as the parameters provided during instantiation.

set_basis(*basis_from_madminer=None*, *basis_numpy=None*, *morphing_matrix=None*)

Manually sets the basis benchmarks.

Parameters

basis_from_madminer [OrderedDict or None, optional] Basis in the *MadMiner.benchmarks* conventions. Default value: None.

basis_numpy [ndarray or None, optional] Only used if basis_from_madminer is None. Basis as a ndarray with shape (n_components, n_parameters).

morphing_matrix [ndarray or None, optional] Manually provided morphing matrix. If None, the morphing matrix is calculated automatically. Default value: None.

Returns

None

set_components(*components*)

Manually defines the components, i.e. the individual terms contributing to the squared matrix element.

Parameters

components [ndarray] Array with shape (n_components, n_parameters), where each entry gives the power with which a parameter scales a given component. For instance, a typical signal, interference, background situation with one parameter might be described by the components [[2], [1], [0]].

Returns

None

class madminer.morphing.**NuisanceMorpher**(*nuisance_parameters_from_madminer*, *benchmark_names*, *reference_benchmark*)

Bases: `object`

Morphing functionality for nuisance parameters.

For a typical MadMiner application, it is not necessary to use the morphing classes directly. The other MadMiner classes use the morphing functions "under the hood" when needed.

Parameters

nuisance_parameters_from_madminer [OrderedDict] Nuisance parameters defined in the form {name: (benchmark_name_pos, benchmark_name_neg)}. Here benchmark_name_pos refers to the name of the benchmark with nu_i = 1, while benchmark_name_neg is either None or refers to the name of the benchmark with nu_i = -1.

benchmark_names [list] The names of the benchmarks.

reference_benchmark [str] Name of the reference benchmark.

Methods

calculate_a(benchmark_weights)	Calculates the first-order coefficients a_i(x) in *dsigma(x | theta, nu) / dsigma(x | theta, 0) = exp[sum_i (a_i(x) nu_i + b_i(x) nu_i^2)].*
calculate_b(benchmark_weights)	Calculates the second-order coefficients b_i(x) in *dsigma(x | theta, nu) / dsigma(x | theta, 0) = exp[sum_i (a_i(x) nu_i + b_i(x) nu_i^2)].*
calculate_nuisance_factors(. . .)	Calculates the rescaling of the event weights from non-central values of nuisance parameters.

calculate_a(*benchmark_weights*)

Calculates the first-order coefficients a_i(x) in *dsigma(x | theta, nu) / dsigma(x | theta, 0) = exp[sum_i (a_i(x) nu_i + b_i(x) nu_i^2)].*

Parameters

benchmark_weights [ndarray] Event weights *dsigma(x | theta_i, nu_i)* with shape *(n_events, n_benchmarks)*. The benchmarks are expected to be sorted in the same order as the keyword benchmark_names used during initialization, and the nuisance benchmarks are expected to be rescaled to have the same physics parameters theta as the reference_benchmark given during initialization.

Returns

a [ndarray] Coefficients a_i(x) with shape *(n_nuisance_parameters, n_events)*.

`calculate_b`(*benchmark_weights*)

Calculates the second-order coefficients b_i(x) in *dsigma(x | theta, nu) / dsigma(x | theta, 0) = exp[sum_i (a_i(x) nu_i + b_i(x) nu_i^2)]*.

Parameters

benchmark_weights [ndarray] Event weights *dsigma(x | theta_i, nu_i)* with shape *(n_events, n_benchmarks)*. The benchmarks are expected to be sorted in the same order as the keyword benchmark_names used during initialization, and the nuisance benchmarks are expected to be rescaled to have the same physics parameters theta as the reference_benchmark given during initialization.

Returns

b [ndarray] Coefficients b_i(x) with shape *(n_nuisance_parameters, n_events)*.

`calculate_nuisance_factors`(*nuisance_parameters*, *benchmark_weights*)

Calculates the rescaling of the event weights from non-central values of nuisance parameters.

Parameters

nuisance_parameters [ndarray] Values of the nuisance parameters *nu*, with shape *(n_nuisance_parameters,)*.

benchmark_weights [ndarray] Event weights *dsigma(x | theta_i, nu_i)* with shape *(n_events, n_benchmarks)*. The benchmarks are expected to be sorted in the same order as the keyword benchmark_names used during initialization, and the nuisance benchmarks are expected to be rescaled to have the same physics parameters theta as the reference_benchmark given during initialization.

Returns

nuisance_factors [ndarray] Nuisance factor *dsigma(x | theta, nu) / dsigma(x | theta, 0)* with shape *(n_events,)*.

CHAPTER 11

madminer.plotting module

madminer.plotting.**plot_2d_morphing_basis**(*morpher*, *xlabel='$\\theta_0$'*, *ylabel='$\\theta_1$'*, *xrange=(-1.0, 1.0)*, *yrange=(-1.0, 1.0)*, *crange=(1.0, 100.0)*, *resolution=100*)

Visualizes a morphing basis and morphing errors for problems with a two-dimensional parameter space.

Parameters

morpher [Morpher] Morpher instance with defined basis.

xlabel [str, optional] Label for the x axis. Default value: r'$ heta_0$'.

ylabel [str, optional] Label for the y axis. Default value: r'$ heta_1$'.

xrange [tuple of float, optional] Range *(min, max)* for the x axis. Default value: (-1., 1.).

yrange [tuple of float, optional] Range *(min, max)* for the y axis. Default value: (-1., 1.).

crange [tuple of float, optional] Range *(min, max)* for the color map. Default value: (1., 100.).

resolution [int, optional] Number of points per axis for the rendering of the squared morphing weights. Default value: 100.

Returns

figure [Figure] Plot as Matplotlib Figure instance.

madminer.plotting.**plot_distribution_of_information**(*xbins*, *xsecs*, *fisher_information_matrices*, *fisher_information_matrices_aux=None*, *xlabel=None*, *xmin=None*, *xmax=None*, *log_xsec=False*, *norm_xsec=True*, *epsilon=1e-09*)

Plots the distribution of the cross section together with the distribution of the Fisher information.

Parameters

xbins [list of float] Bin boundaries.

xsecs [list of float] Cross sections (in pb) per bin.

fisher_information_matrices [list of ndarray] Fisher information matrices for each bin.

fisher_information_matrices_aux [list of ndarray or None, optional] Additional Fisher information matrices for each bin (will be plotted with a dashed line).

xlabel [str or None, optional] Label for the x axis.

xmin [float or None, optional] Minimum value for the x axis.

xmax [float or None, optional] Maximum value for the x axis.

log_xsec [bool, optional] Whether to plot the cross section on a logarithmic y axis.

norm_xsec [bool, optional] Whether the cross sections are normalized to 1.

epsilon [float, optional] Numerical factor.

Returns

figure [Figure] Plot as Matplotlib Figure instance.

`madminer.plotting.`**`plot_distributions`**(*filename*, *observables=None*, *parameter_points=None*, *uncertainties='nuisance'*, *nuisance_parameters=None*, *draw_nuisance_toys=None*, *normalize=False*, *log=False*, *observable_labels=None*, *n_bins=50*, *line_labels=None*, *colors=None*, *linestyles=None*, *linewidths=1.5*, *toy_linewidths=0.5*, *alpha=0.15*, *toy_alpha=0.75*, *n_events=None*, *n_toys=100*, *n_cols=3*, *quantiles_for_range=(0.025, 0.975)*)

Plots one-dimensional histograms of observables in a MadMiner file for a given set of benchmarks.

Parameters

filename [str] Filename of a MadMiner HDF5 file.

observables [list of str or None, optional] Which observables to plot, given by a list of their names. If None, all observables in the file are plotted. Default value: None.

parameter_points [list of (str or ndarray) or None, optional] Which parameter points to use for histogramming the data. Given by a list, each element can either be the name of a benchmark in the MadMiner file, or an ndarray specifying any parameter point in a morphing setup. If None, all physics (non-nuisance) benchmarks defined in the MadMiner file are plotted. Default value: None.

uncertainties [{"nuisance", "none"}, optional] Defines how uncertainty bands are drawn. With "nuisance", the variation in cross section from all nuisance parameters is added in quadrature. With "none", no error bands are drawn.

nuisance_parameters [None or list of int, optional] If uncertainties is "nuisance", this can restrict which nuisance parameters are used to draw the uncertainty bands. Each entry of this list is the index of one nuisance parameter (same order as in the MadMiner file).

draw_nuisance_toys [None or int, optional] If not None and uncertainties is "nuisance", sets the number of nuisance toy distributions that are drawn (in addition to the error bands).

normalize [bool, optional] Whether the distribution is normalized to the total cross section. Default value: False.

log [bool, optional] Whether to draw the y axes on a logarithmic scale. Defaul value: False.

observable_labels [None or list of (str or None), optional] x-axis labels naming the observables. If None, the observable names from the MadMiner file are used. Default value: None.

n_bins [int, optional] Number of histogram bins. Default value: 50.

line_labels [None or list of (str or None), optional] Labels for the different parameter points. If None and if parameter_points is None, the benchmark names from the MadMiner file are used. Default value: None.

colors [None or str or list of str, optional] Matplotlib line (and error band) colors for the distributions. If None, uses default colors. Default value: None.

linestyles [None or str or list of str, optional] Matplotlib line styles for the distributions. If None, uses default linestyles. Default value: None.

linewidths [float or list of float, optional] Line widths for the contours. Default value: 1.5.

toy_linewidths [float or list of float or None, optional] Line widths for the toy replicas, if uncertainties is "nuisance" and draw_nuisance_toys is not None. If None, linewidths is used. Default value: 1.

alpha [float, optional] alpha value for the uncertainty bands. Default value: 0.25.

toy_alpha [float, optional] alpha value for the toy replicas, if uncertainties is "nuisance" and draw_nuisance_toys is not None. Default value: 0.75.

n_events [None or int, optional] If not None, sets the number of events from the MadMiner file that will be analyzed and plotted. Default value: None.

n_toys [int, optional] Number of toy nuisance parameter vectors used to estimate the systematic uncertainties. Default value: 100.

n_cols [int, optional] Number of columns of subfigures in the plot. Default value: 3.

quantiles_for_range [tuple of two float, optional] Tuple *(min_quantile, max_quantile)* that defines how the observable range is determined for each panel. Default: (0.025, 0.075).

Returns

figure [Figure] Plot as Matplotlib Figure instance.

`madminer.plotting.`**`plot_fisher_information_contours_2d`**(*fisher_information_matrices*, *fisher_information_covariances=None*, *reference_thetas=None*, *contour_distance=1.0*, *xlabel='$\\theta_0$'*, *ylabel='$\\theta_1$'*, *xrange=(-1.0, 1.0)*, *yrange=(-1.0, 1.0)*, *labels=None*, *inline_labels=None*, *resolution=500*, *colors=None*, *linestyles=None*, *linewidths=1.5*, *alphas=1.0*, *alphas_uncertainties=0.25*)

Visualizes 2x2 Fisher information matrices as contours of constant Fisher distance from a reference point *theta0*.

The local (tangent-space) approximation is used: distances *d(theta)* are given by *d(theta)^2 = (theta - theta0)_i I_ij (theta - theta0)_j*, summing over *i* and *j*.

Parameters

fisher_information_matrices [list of ndarray] Fisher information matrices, each with shape (2,2).

fisher_information_covariances [None or list of (ndarray or None), optional] Covariance matrices for the Fisher information matrices. Has to have the same length as fisher_information_matrices, and each entry has to be None (no uncertainty) or a tensor with shape (2,2,2,2). Default value: None.

reference_thetas [None or list of (ndarray or None), optional] Reference points from which the distances are calculated. If None, the origin (0,0) is used. Default value: None.

contour_distance [float, optional.] Distance threshold at which the contours are drawn. Default value: 1.

xlabel [str, optional] Label for the x axis. Default value: r'$ heta_0$'.

ylabel [str, optional] Label for the y axis. Default value: r'$ heta_1$'.

xrange [tuple of float, optional] Range *(min, max)* for the x axis. Default value: (-1., 1.).

yrange [tuple of float, optional] Range *(min, max)* for the y axis. Default value: (-1., 1.).

labels [None or list of (str or None), optional] Legend labels for the contours. Default value: None.

inline_labels [None or list of (str or None), optional] Inline labels for the contours. Default value: None.

resolution [int] Number of points per axis for the calculation of the distances. Default value: 500.

colors [None or str or list of str, optional] Matplotlib line (and error band) colors for the contours. If None, uses default colors. Default value: None.

linestyles [None or str or list of str, optional] Matploitlib line styles for the contours. If None, uses default linestyles. Default value: None.

linewidths [float or list of float, optional] Line widths for the contours. Default value: 1.5.

alphas [float or list of float, optional] Opacities for the contours. Default value: 1.

alphas_uncertainties [float or list of float, optional] Opacities for the error bands. Default value: 0.25.

Returns

figure [Figure] Plot as Matplotlib Figure instance.

madminer.plotting.**plot_fisherinfo_barplot**(*fisher_information_matrices*, *labels*, *determinant_indices=None*, *eigenvalue_colors=None*, *bar_colors=None*)

Parameters

fisher_information_matrices [list of ndarray] Fisher information matrices

labels [list of str] Labels for the x axis

determinant_indices [list of int or None, optional] If not None, the determinants will be based only on the indices given here. Default value: None.

eigenvalue_colors [None or list of str] Colors for the eigenvalue decomposition. If None, default colors are used. Default value: None.

bar_colors [None or list of str] Colors for the determinant bars. If None, default colors are used. Default value: None.

Returns

figure [Figure] Plot as Matplotlib Figure instance.

madminer.plotting.**plot_nd_morphing_basis_scatter**(*morpher*, *crange=(1.0, 100.0)*, *n_test_thetas=1000*)

Visualizes a morphing basis and morphing errors with scatter plots between each pair of operators.

Parameters

morpher [Morpher] Morpher instance with defined basis.

crange [tuple of float, optional] Range *(min, max)* for the color map. Default value: (1. 100.).

n_test_thetas [int, optional] Number of random points evaluated. Default value: 1000.

Returns

figure [Figure] Plot as Matplotlib Figure instance.

`madminer.plotting.`**`plot_nd_morphing_basis_slices`**(*morpher*, *crange=(1.0*, *100.0)*, *resolution=50*)

Visualizes a morphing basis and morphing errors with two-dimensional slices through parameter space.

Parameters

morpher [Morpher] Morpher instance with defined basis.

crange [tuple of float, optional] Range *(min, max)* for the color map.

resolution [int, optional] Number of points per panel and axis for the rendering of the squared morphing weights. Default value: 50.

Returns

figure [Figure] Plot as Matplotlib Figure instance.

`madminer.plotting.`**`plot_uncertainty`**(*filename*, *theta*, *observable*, *obs_label*, *obs_range*, *n_bins=50*, *nuisance_parameters=None*, *n_events=None*, *n_toys=100*, *linecolor='black'*, *bandcolor1='#CC002E'*, *bandcolor2='orange'*, *ratio_range=(0.8*, *1.2)*)

Plots absolute and relative uncertainty bands in a histogram of one observable in a MadMiner file.

Parameters

filename [str] Filename of a MadMiner HDF5 file.

theta [ndarray, optional] Which parameter points to use for histogramming the data.

observable [str] Which observable to plot, given by its name in the MadMiner file.

obs_label [str] x-axis label naming the observable.

obs_range [tuple of two float] Range to be plotted for the observable.

n_bins [int] Number of bins. Default value: 50.

nuisance_parameters [None or list of int, optional] This can restrict which nuisance parameters are used to draw the uncertainty bands. Each entry of this list is the index of one nuisance parameter (same order as in the MadMiner file).

n_events [None or int, optional] If not None, sets the number of events from the MadMiner file that will be analyzed and plotted. Default value: None.

n_toys [int, optional] Number of toy nuisance parameter vectors used to estimate the systematic uncertainties. Default value: 100.

linecolor [str, optional] Line color for central prediction. Default value: "black".

bandcolor1 [str, optional] Error band color for 1 sigma uncertainty. Default value: "#CC002E".

bandcolor2 [str, optional] Error band color for 2 sigma uncertainty. Default value: "orange".

ratio_range [tuple of two floar] y-axis range for the plots of the ratio to the central prediction. Default value: (0.8, 1.2).

Returns

figure [Figure] Plot as Matplotlib Figure instance.

CHAPTER 12

madminer.sampling module

class madminer.sampling.**SampleAugmenter**(*filename*, *disable_morphing=False*, *include_nuisance_parameters=True*)

Bases: `object`

Sampling and data augmentation.

After the generated events have been analyzed and the observables and weights have been saved into a MadMiner file, for instance with *madminer.delphes.DelphesProcessor* or *madminer.lhe.LHEProcessor*, the next step is typically the generation of training and evaluation data for the machine learning algorithms. This generally involves two (related) tasks: unweighting, i.e. the creation of samples that do not carry individual weights but follow some distribution, and the extraction of the joint likelihood ratio and / or joint score (the "augmented data").

After inializing *SampleAugmenter* with the filename of a MadMiner file, this is done with a single function call. Depending on the downstream inference algorithm, there are different possibilities:

- *SampleAugmenter.extract_samples_train_plain()* creates plain training samples without augmented data.
- *SampleAugmenter.extract_samples_train_local()* creates training samples for local methods based on the score, such as SALLY and SALLINO.
- *SampleAugmenter.extract_samples_train_global()* creates training samples for non-local methods based on density estimation and the score, such as SCANDAL.
- *SampleAugmenter.extract_samples_train_ratio()* creates training samples for non-local, ratio-based methods like RASCAL or ALICE.
- *SampleAugmenter.extract_samples_train_more_ratios()* does the same, but can extract joint ratios and scores at more parameter points. This additional information can be used efficiently in the setup with a "doubly parameterized" likelihood ratio estimator that models the dependence on both the numerator and denominator hypothesis.
- *SampleAugmenter.extract_samples_test()* creates evaluation samples for all methods.

Please see the tutorial for a walkthrough.

For the curious, let us explain these steps in a little bit more detail (assuming a morphing setup):

- The sample augmentation step starts from a set of events *(x_i, z_i)* together with corresponding weights for each morphing basis point *theta_b*, *p(x_i, z_i | theta_b)*.
- Morphing: Assume we want to generate data sampled from a parameter point theta, which is not necessarily one of the basis points theta_b. Using the morphing structure, the event weights for p(x_i, z_i | theta) can be calculated. Note that the events (phase-space points) *(x_i, z_i)* are not changed, only their weights.
- Unweighting: For the machine learning part, such a weighted event sample is not practical. Instead we aim for an unweighted one, in which events can appear multiple times. If the user request *N* events (which can be larger than the original number of events in the MadGraph runs), SampleAugmenter will draw *N* samples *(x_i, z_i)* from the discrete distribution *p(x_i, z_i | theta)*. In other words, it draws (with replacement) *N* of the original events from MadGraph, with probabilities given by the morphing setup before. This is similar to what *np.random.choice()* does.
- Augmentation: For each of the drawn samples, the morphing setup can be used to calculate the joint likelihood ratio and / or the joint score (this depends on which SampleAugmenter function is called).

  Parameters

  filename [str] Path to MadMiner file (for instance the output of *madminer.delphes.DelphesProcessor.save()*).

  disable_morphing [bool, optional] If True, the morphing setup is not loaded from the file. Default value: False.

  include_nuisance_parameters [bool, optional] If True, nuisance parameters are taken into account. Default value: True.

Methods

`extract_cross_sections`(theta)	Calculates the total cross sections for all specified thetas.
`extract_raw_data`([theta, derivative])	Returns all events together with the benchmark weights (if theta is None) or weights for a given theta.
`extract_samples_test`(theta, n_samples, . . .)	Extracts evaluation samples *x ~ p(x|theta)* without any augmented data.
`extract_samples_train_global`(theta, . . . [, . . .])	Extracts training samples x ~ p(x|theta) as well as the joint score t(x, z|theta), where theta is sampled from a prior.
`extract_samples_train_local`(theta, . . . [, . . .])	Extracts training samples x ~ p(x|theta) as well as the joint score t(x, z|theta).
`extract_samples_train_more_ratios`(theta0, . . .)	Extracts training samples *x ~ p(x|theta0)* and *x ~ p(x|theta1)* together with the class label *y*, the joint likelihood ratio *r(x,z|theta0, theta1)*, and the joint score *t(x,z|theta0)*.
`extract_samples_train_plain`(theta, . . . [, . . .])	Extracts plain training samples *x ~ p(x|theta)* without any augmented data.
`extract_samples_train_ratio`(theta0, theta1, . . .)	Extracts training samples *x ~ p(x|theta0)* and *x ~ p(x|theta1)* together with the class label *y*, the joint likelihood ratio *r(x,z|theta0, theta1)*, and, if morphing is set up, the joint score *t(x,z|theta0)*.

extract_cross_sections (*theta*)

Calculates the total cross sections for all specified thetas.

Parameters

theta [tuple] Tuple (type, value) that defines the parameter point or prior over parameter points at which the cross section is calculated. Pass the output of the functions *constant_benchmark_theta()*, *multiple_benchmark_thetas()*, *constant_morphing_theta()*, *multiple_morphing_thetas()*, or *random_morphing_thetas()*.

Returns

thetas [ndarray] Parameter points with shape *(n_thetas, n_parameters)*.

xsecs [ndarray] Total cross sections in pb with shape *(n_thetas,)*.

xsec_uncertainties [ndarray] Statistical uncertainties on the total cross sections in pb with shape *(n_thetas,)*.

extract_raw_data(*theta=None*, *derivative=False*)

Returns all events together with the benchmark weights (if theta is None) or weights for a given theta.

Parameters

theta [None or ndarray or str, optional] If None, the function returns all benchmark weights. If str, the function returns the weights for a given benchmark name. If ndarray, it uses morphing to calculate the weights for this value of theta. Default value: None.

derivative [bool, optional] If True and if theta is not None, the derivative of the weights with respect to theta are returned. Default value: False.

Returns

x [ndarray] Observables with shape *(n_unweighted_samples, n_observables)*.

weights [ndarray] If theta is None and derivative is False, benchmark weights with shape *(n_unweighted_samples, n_benchmarks_phys)* in pb. If theta is not None and derivative is True, the gradient of the weight for the given parameter with respect to theta with shape *(n_unweighted_samples, n_gradients)* in pb. Otherwise, weights for the given parameter theta with shape *(n_unweighted_samples,)* in pb.

extract_samples_test(*theta*, *n_samples*, *folder*, *filename*, *test_split=0.5*, *switch_train_test_events=False*)

Extracts evaluation samples *x ~ p(x|theta)* without any augmented data.

Parameters

theta [tuple] Tuple (type, value) that defines the parameter point or prior over parameter points for the sampling. Pass the output of the functions *constant_benchmark_theta()*, *multiple_benchmark_thetas()*, *constant_morphing_theta()*, *multiple_morphing_thetas()*, or *random_morphing_thetas()*.

n_samples [int] Total number of events to be drawn.

folder [str] Path to the folder where the resulting samples should be saved (ndarrays in .npy format).

filename [str] Filenames for the resulting samples. A prefix such as ‘x’ or ‘theta0’ as well as the extension ‘.npy’ will be added automatically.

test_split [float or None, optional] Fraction of events reserved for the evaluation sample (that will not be used for any training samples). Default value: 0.5.

switch_train_test_events [bool, optional] If True, this function generates a test sample from the events normally reserved for training samples. Default value: False.

Returns

x [ndarray] Observables with shape *(n_samples, n_observables)*. The same information is saved as a file in the given folder.

theta [ndarray] Parameter points used for sampling with shape *(n_samples, n_parameters)*. The same information is saved as a file in the given folder.

extract_samples_train_global(*theta*, *n_samples*, *folder*, *filename*, *test_split=0.5*, *switch_train_test_events=False*)

Extracts training samples x ~ p(x|theta) as well as the joint score t(x, z|theta), where theta is sampled from a prior. This can be used for inference methods such as SCANDAL.

Parameters

theta [tuple] Tuple (type, value) that defines the numerator parameter point or prior over parameter points for the sampling. Pass the output of the functions *constant_benchmark_theta()*, *multiple_benchmark_thetas()*, *constant_morphing_theta()*, *multiple_morphing_thetas()*, or *random_morphing_thetas()*.

n_samples [int] Total number of events to be drawn.

folder [str] Path to the folder where the resulting samples should be saved (ndarrays in .npy format).

filename [str] Filenames for the resulting samples. A prefix such as 'x' or 'theta0' as well as the extension '.npy' will be added automatically.

test_split [float or None, optional] Fraction of events reserved for the evaluation sample (that will not be used for any training samples). Default value: 0.5.

switch_train_test_events [bool, optional] If True, this function generates a training sample from the events normally reserved for test samples. Default value: False.

Returns

x [ndarray] Observables with shape *(n_samples, n_observables)*. The same information is saved as a file in the given folder.

theta [ndarray] Parameter points used for sampling (and evaluation of the joint score) with shape *(n_samples, n_parameters)*. The same information is saved as a file in the given folder.

t_xz [ndarray] Joint score evaluated at theta with shape *(n_samples, n_parameters)*. The same information is saved as a file in the given folder.

extract_samples_train_local(*theta*, *n_samples*, *folder*, *filename*, *nuisance_score=False*, *test_split=0.5*, *switch_train_test_events=False*, *log_message=True*)

Extracts training samples x ~ p(x|theta) as well as the joint score t(x, z|theta). This can be used for inference methods such as SALLY and SALLINO.

Parameters

theta [tuple] Tuple (type, value) that defines the parameter point for the sampling. This is also where the score is evaluated. Pass the output of the functions *constant_benchmark_theta()* or *constant_morphing_theta()*.

n_samples [int] Total number of events to be drawn.

folder [str] Path to the folder where the resulting samples should be saved (ndarrays in .npy format).

filename [str] Filenames for the resulting samples. A prefix such as 'x' or 'theta0' as well as the extension '.npy' will be added automatically.

nuisance_score [bool, optional] If True and if the sample contains nuisance parameters, the score with respect to the nuisance parameters (at the default position) will also be calculated. Otherwise, only the score with respect to the physics parameters is calculated. Default: False.

test_split [float or None, optional] Fraction of events reserved for the evaluation sample (that will not be used for any training samples). Default value: 0.5.

switch_train_test_events [bool, optional] If True, this function generates a training sample from the events normally reserved for test samples. Default value: False.

log_message [bool, optional] If True, logging output. This option is only designed for internal use.

Returns

x [ndarray] Observables with shape *(n_samples, n_observables)*. The same information is saved as a file in the given folder.

theta [ndarray] Parameter points used for sampling (and evaluation of the joint score) with shape *(n_samples, n_parameters)*. The same information is saved as a file in the given folder.

t_xz [ndarray] Joint score evaluated at theta with shape *(n_samples, n_parameters + n_nuisance_parameters)* (if nuisance_score is True) or *(n_samples, n_parameters)*. The same information is saved as a file in the given folder.

`extract_samples_train_more_ratios`(*theta0*, *theta1*, *n_samples*, *folder*, *filename*, *additional_thetas=None*, *test_split=0.5*, *switch_train_test_events=False*)

Extracts training samples *x ~ p(x|theta0)* and *x ~ p(x|theta1)* together with the class label *y*, the joint likelihood ratio *r(x,z|theta0, theta1)*, and the joint score *t(x,z|theta0)*. This information can be used in inference methods such as CARL, ROLR, CASCAL, and RASCAL.

With the keyword *additional_thetas*, this function allows to extract joint ratios and scores at more parameter points than just *theta0* and *theta1*. This additional information can be used efficiently in the setup with a "doubly parameterized" likelihood ratio estimator that models the dependence on both the numerator and denominator hypothesis.

Parameters

theta0 : Tuple (type, value) that defines the numerator parameter point or prior over parameter points for the sampling. Pass the output of the functions *constant_benchmark_theta()*, *multiple_benchmark_thetas()*, *constant_morphing_theta()*, *multiple_morphing_thetas()*, or *random_morphing_thetas()*.

theta1 : Tuple (type, value) that defines the denominator parameter point or prior over parameter points for the sampling. Pass the output of the functions *constant_benchmark_theta()*, *multiple_benchmark_thetas()*, *constant_morphing_theta()*, *multiple_morphing_thetas()*, or *random_morphing_thetas()*.

n_samples [int] Total number of events to be drawn.

folder [str] Path to the folder where the resulting samples should be saved (ndarrays in .npy format).

filename [str] Filenames for the resulting samples. A prefix such as 'x' or 'theta0' as well as the extension '.npy' will be added automatically.

additional_thetas [list of tuple or None] list of tuples *(type, value)* that defines additional theta points at which ratio and score are evaluated, and which are then used to create additional training data points. These can be efficiently used only in the "doubly parameterized" setup where a likelihood ratio estimator models the dependence of the likelihood ratio on both the numerator and denominator hypothesis. Pass the output of the helper functions *constant_benchmark_theta()*, *multiple_benchmark_thetas()*, *constant_morphing_theta()*, *multiple_morphing_thetas()*, or *random_morphing_thetas()*. Default value: None.

test_split [float or None, optional] Fraction of events reserved for the evaluation sample (that will not be used for any training samples). Default value: 0.5.

switch_train_test_events [bool, optional] If True, this function generates a training sample from the events normally reserved for test samples. Default value: False.

Returns

x [ndarray] Observables with shape *(n_samples, n_observables)*. The same information is saved as a file in the given folder.

theta0 [ndarray] Numerator parameter points with shape *(n_samples, n_parameters)*. The same information is saved as a file in the given folder.

theta1 [ndarray] Denominator parameter points with shape *(n_samples, n_parameters)*. The same information is saved as a file in the given folder.

y [ndarray] Class label with shape *(n_samples, n_parameters)*. *y=0* (*1*) for events sample from the numerator (denominator) hypothesis. The same information is saved as a file in the given folder.

r_xz [ndarray] Joint likelihood ratio with shape *(n_samples,)*. The same information is saved as a file in the given folder.

t_xz [ndarray] Joint score evaluated at theta0 with shape *(n_samples, n_parameters)*. The same information is saved as a file in the given folder.

`extract_samples_train_plain`(*theta*, *n_samples*, *folder*, *filename*, *test_split=0.5*, *switch_train_test_events=False*)

Extracts plain training samples *x ~ p(x|theta)* without any augmented data. This can be use for standard inference methods such as ABC, histograms of observables, or neural density estimation techniques. It can also be used to create validation or calibration samples.

Parameters

theta [tuple] Tuple (type, value) that defines the parameter point or prior over parameter points for the sampling. Pass the output of the functions *constant_benchmark_theta()*, *multiple_benchmark_thetas()*, *constant_morphing_theta()*, *multiple_morphing_thetas()*, or *random_morphing_thetas()*.

n_samples [int] Total number of events to be drawn.

folder [str] Path to the folder where the resulting samples should be saved (ndarrays in .npy format).

filename [str] Filenames for the resulting samples. A prefix such as 'x' or 'theta0' as well as the extension '.npy' will be added automatically.

test_split [float or None, optional] Fraction of events reserved for the evaluation sample (that will not be used for any training samples). Default value: 0.5.

switch_train_test_events [bool, optional] If True, this function generates a training sample from the events normally reserved for test samples. Default value: False.

Returns

x [ndarray] Observables with shape *(n_samples, n_observables)*. The same information is saved as a file in the given folder.

theta [ndarray] Parameter points used for sampling with shape *(n_samples, n_parameters)*. The same information is saved as a file in the given folder.

extract_samples_train_ratio(*theta0*, *theta1*, *n_samples*, *folder*, *filename*, *test_split=0.5*, *switch_train_test_events=False*)

Extracts training samples *x ~ p(x|theta0)* and *x ~ p(x|theta1)* together with the class label *y*, the joint likelihood ratio *r(x,z|theta0, theta1)*, and, if morphing is set up, the joint score *t(x,z|theta0)*. This information can be used in inference methods such as CARL, ROLR, CASCAL, and RASCAL.

Parameters

theta0 [tuple] Tuple (type, value) that defines the numerator parameter point or prior over parameter points for the sampling. Pass the output of the functions *constant_benchmark_theta()*, *multiple_benchmark_thetas()*, *constant_morphing_theta()*, *multiple_morphing_thetas()*, or *random_morphing_thetas()*.

theta1 [tuple] Tuple (type, value) that defines the denominator parameter point or prior over parameter points for the sampling. Pass the output of the functions *constant_benchmark_theta()*, *multiple_benchmark_thetas()*, *constant_morphing_theta()*, *multiple_morphing_thetas()*, or *random_morphing_thetas()*.

n_samples [int] Total number of events to be drawn.

folder [str] Path to the folder where the resulting samples should be saved (ndarrays in .npy format).

filename [str] Filenames for the resulting samples. A prefix such as 'x' or 'theta0' as well as the extension '.npy' will be added automatically.

test_split [float or None, optional] Fraction of events reserved for the evaluation sample (that will not be used for any training samples). Default value: 0.5.

switch_train_test_events [bool, optional] If True, this function generates a training sample from the events normally reserved for test samples. Default value: False.

Returns

x [ndarray] Observables with shape *(n_samples, n_observables)*. The same information is saved as a file in the given folder.

theta0 [ndarray] Numerator parameter points with shape *(n_samples, n_parameters)*. The same information is saved as a file in the given folder.

theta1 [ndarray] Denominator parameter points with shape *(n_samples, n_parameters)*. The same information is saved as a file in the given folder.

y [ndarray] Class label with shape *(n_samples, n_parameters)*. *y=0* (*1*) for events sample from the numerator (denominator) hypothesis. The same information is saved as a file in the given folder.

r_xz [ndarray] Joint likelihood ratio with shape *(n_samples,)*. The same information is saved as a file in the given folder.

t_xz [ndarray or None] If morphing is set up, the joint score evaluated at theta0 with shape *(n_samples, n_parameters)*. The same information is saved as a file in the given folder. If morphing is not set up, None is returned (and no file is saved).

madminer.sampling.**combine_and_shuffle**(*input_filenames*, *output_filename*, *k_factors=None*, *overwrite_existing_file=True*, *shuffle_sample=True*)

Combines multiple MadMiner files into one, and shuffles the order of the events.

Note that this function assumes that all samples are generated with the same setup, including identical benchmarks (and thus morphing setup). If it is used with samples with different settings, there will be wrong results!

There are no explicit cross checks in place yet!

Parameters

input_filenames [list of str] List of paths to the input MadMiner files.

output_filename [str] Path to the combined MadMiner file.

k_factors [float or list of float, optional] Multiplies the weights in input_filenames with a universal factor (if k_factors is a float) or with independent factors (if it is a list of float). Default value: None.

overwrite_existing_file [bool, optional] If True and if the output file exists, it is overwritten. Default value: True.

shuffle_sample [bool, optional] If True, the output shuffle will be shuffled. Default value: True.

Returns

None

`madminer.sampling.`**`constant_benchmark_theta`**(*benchmark_name*)

Utility function to be used as input to various SampleAugmenter functions, specifying a single parameter benchmark.

Parameters

benchmark_name [str] Name of the benchmark (as in *madminer.core.MadMiner.add_benchmark*)

Returns

output [tuple] Input to various SampleAugmenter functions

`madminer.sampling.`**`constant_morphing_theta`**(*theta*)

Utility function to be used as input to various SampleAugmenter functions, specifying a single parameter point theta in a morphing setup.

Parameters

theta [ndarray or list] Parameter point with shape *(n_parameters,)*

Returns

output [tuple] Input to various SampleAugmenter functions

`madminer.sampling.`**`multiple_benchmark_thetas`**(*benchmark_names*)

Utility function to be used as input to various SampleAugmenter functions, specifying multiple parameter benchmarks.

Parameters

benchmark_names [list of str] List of names of the benchmarks (as in *madminer.core.MadMiner.add_benchmark*)

Returns

output [tuple] Input to various SampleAugmenter functions

`madminer.sampling.`**`multiple_morphing_thetas`**(*thetas*)

Utility function to be used as input to various SampleAugmenter functions, specifying multiple parameter points theta in a morphing setup.

Parameters

thetas [ndarray or list of lists or list of ndarrays] Parameter points with shape *(n_thetas, n_parameters)*

Returns

output [tuple] Input to various SampleAugmenter functions

`madminer.sampling.`**`random_morphing_thetas`**(*n_thetas*, *priors*)

Utility function to be used as input to various SampleAugmenter functions, specifying random parameter points sampled from a prior in a morphing setup.

Parameters

n_thetas [int] Number of parameter points to be sampled

priors [list of tuples] Priors for each parameter is characterized by a tuple of the form *(prior_shape, prior_param_0, prior_param_1)*. Currently, the supported prior_shapes are *flat*, in which case the two other parameters are the lower and upper bound of the flat prior, and *gaussian*, in which case they are the mean and standard deviation of a Gaussian.

Returns

output [tuple] Input to various SampleAugmenter functions

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index