MadMiner Documentation

Johann Brehmer, Kyle Cranmer, and Felix Kling

Oct 25, 2018

CONTENTS:

CHAPTER

ONE

MADMINER.CORE MODULE

class madminer.core.**MadMiner**(*debug=False*)

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.

Parameters

debug [bool, optional] If True, additional detailed debugging output is printed. Default value: False.

Methods

add_benchmark(parameter_values[, benchmark_name])	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.
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.

Continued on next page

Table 1 – continued from previous page

set_benchmarks([benchmarks])	Manually sets all benchmarks, that is, parameter points that will be evaluated by MadGraph.
set_benchmarks_from_morphing([...])	Sets benchmarks, i.e.
set_parameters([parameters])	Manually sets all parameters, overwriting previously added parameters.

add_benchmark(*parameter_values*, *benchmark_name=None*)

Manually adds an individual benchmark, that is, a parameter point that will be evaluated by MadGraph.

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.

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*, *reweight_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*)

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.

reweight_card_template_file [str] Path to an empty reweight card that will be used as template to create the appropriate reweight 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.

Returns

None

run_multiple(*mg_directory*, *proc_card_file*, *param_card_template_file*, *reweight_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*)

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.

reweight_card_template_file [str] Path to an empty reweight card that will be used as template to create the appropriate reweight 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.

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, 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*)

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.

Returns

None

set_benchmarks_from_morphing(*max_overall_power=4*, *n_bases=1*, *keep_existing_benchmarks=True*, *n_trials=100*, *n_test_thetas=100*)

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.

keep_existing_benchmarks [bool, optional] If True, the previously defined benchmarks are included in the basis. In that case, the number of free parameters in the optimization routine is reduced. If False, all benchmarks are optimized and all previously defined benchmarks forgotten. 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

CHAPTER

TWO

MADMINER.DELPHES MODULE

class madminer.delphes.**DelphesProcessor**(*filename=None*, *debug=False*)

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 HepMC samples produced by Pythia in *DelphesProcessor.add_hepmc_sample()*
- Running Delphes on these samples through *DelphesProcessor.run_delphes()*
- Optionally, acceptance cuts for all visible particles can be defined with *DelphesProcessor.set_acceptance()*.
- Defining observables through *DelphesProcessor.add_observables()*. A simple set of default observables is provided with *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.

debug [bool, optional] If True, additional detailed debugging output is printed. Default value: False.

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.

Continued on next page

Table 1 – continued from previous page

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_hepmc_sample(filename, ...)	Adds simulated events in the HepMC format.
add_observable(name, definition[, required, ...])	Adds an observable as a string that can be parsed by Python's *eval()* function.
analyse_delphes_samples()	Main function that parses the Delphes samples (ROOT files), checks acceptance and cuts, and extracts the observables and weights.
run_delphes(delphes_directory, delphes_card)	Runs the fast detector simulation on all HepMC samples added so far.
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. 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 charged (i.e. an electron has *charge* = *-1.*), and the PDG particle ID. For instance, *"len(e) >= 2"* requires at least two electrons passing the acceptance cuts.

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*)

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.

Returns

None

add_hepmc_sample(*filename*, *sampled_from_benchmark*)
Adds simulated events in the HepMC format.

Parameters

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()*).

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. 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 charged (i.e. an electron has *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

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

Returns

None

run_delphes(*delphes_directory*, *delphes_card*, *initial_command=None*, *log_directory=None*)
Runs the fast detector simulation on all HepMC samples added so far.

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_directory [str or None, optional] Directory for log files 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.

Parameters

filename_out [str] Path to where the results should be saved. If the class was initialized with *filename=None*, this file is assumed to exist and contain the correct parameter, benchmark, and morphing setup.

Returns

None

set_acceptance(*pt_min_e=10.0*, *pt_min_mu=10.0*, *pt_min_a=0.0*, *pt_min_j=20.0*, *eta_max_e=2.5*, *eta_max_mu=2.5*, *eta_max_a=2.5*, *eta_max_j=5.0*)

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

Parameters

pt_min_e [float] Minimum electron transverse momentum in GeV. Default value: 10.

pt_min_mu [float] Minimum muon transverse momentum in GeV. Default value: 10.

pt_min_a [float] Minimum photon transverse momentum in GeV. Default value: 10.

pt_min_j [float] Minimum jet transverse momentum in GeV. Default value: 20.

eta_max_e [float] Maximum absolute electron pseudorapidity. Default value: 2.5.

eta_max_mu [float] Maximum absolute muon pseudorapidity. Default value: 2.5.

eta_max_a [float] Maximum absolute photon pseudorapidity. Default value: 2.5.

eta_max_j [float] Maximum absolute jet pseudorapidity. Default value: 5.

Returns

None

CHAPTER

THREE

MADMINER.FISHERINFORMATION MODULE

class madminer.fisherinformation.**FisherInformation**(*filename*, *debug=False*)

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").

Parameters

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

debug [bool, optional] If True, additional detailed debugging output is printed. Default value: False.

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()	Calculates the full Fisher information at parton / truth level.

Continued on next page

Table 1 – continued from previous page

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_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*, *luminosity=300000.0*, *cuts=None*, *return_error=None*, *ensemble_vote_expectation_weight=None*)

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.

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] Filename of an unweighted x sample that is sampled according to theta and obeys the cuts (see *madminer.sampling.SampleAugmenter.extract_samples_train_local()*)

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.

return_error [None or bool, optional] Whether an uncertainty of the Fisher information is returned together with the prediction. If None, it is returned only if model_file points to the directory of an ensemble. Default value: None.

ensemble_vote_expectation_weight [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 None, or if *EnsembleForge.calculate_expectation()* has not been called, all estimators are treated equal. Default value: None.

Returns

fisher_information [ndarray] Estimated expected full detector-level Fisher information matrix with shape *(n_parameters, n_parameters)*.

fisher_information_uncertainty [ndarray] Returned only if return_error is True, or if return_error is None and model_file is the directory of an ensemble. Uncertainty of the expected full detector-level Fisher information matrix with shape *(n_parameters, n_parameters)*.

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

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.

Returns

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

calculate_fisher_information_hist1d(*theta*, *luminosity*, *observable*, *nbins*, *histrange*, *cuts=None*, *efficiency_functions=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] Minimum and maximum value of the histogram in the form *(min, max)*. Overflow bins are always added.

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.

Returns

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

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

histrange1 [tuple of float] Minimum and maximum value of the first axis of the histogram in the form *(min, max)*. Overflow bins are always added.

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.

histrange2 [tuple of float] Minimum and maximum value of the first axis of the histogram in the form *(min, max)*. Overflow bins are always added.

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.

Returns

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

calculate_fisher_information_rate(*theta*, *luminosity*, *cuts=None*, *efficiency_functions=None*)

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.

Returns

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

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)* in pb. Otherwise, weights for the given parameter theta with shape *(n_unweighted_samples,)* in pb.

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

Calculates the full and rate-only Fisher information in slices of one 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 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.

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.

Returns

bin_boundaries [ndarray] Observable slice boundaries.

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

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

full_fisher_infos_truth [ndarray] Expected full truth-level Fisher information for each slice. Has shape *(n_slices, n_parameters, n_parameters)*.

madminer.fisherinformation.**profile_information**(*fisher_information*, *remaining_components*)

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 projected out.

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.

madminer.fisherinformation.**project_information**(*fisher_information*, *remaining_components*)

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.

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.

CHAPTER

FOUR

MADMINER.ML MODULE

class madminer.ml.**EnsembleForge**(*estimators=None*, *debug=False*)

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.

Note that currently EnsembleForge only supports SALLY and SALLINO estimators.

Parameters

estimators [None or int or list of MLForge, optional] If int, sets the number of estimators that will be created as new MLForge instances. If list of MLForge, sets the estimators directly. 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.

debug [bool, optional] If True, additional detailed debugging output is printed. Default value: False.

Methods

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_filename[, ...])	Calculates the expected Fisher information matrices for each estimator, and then returns the ensemble mean and variance.
evaluate(x_filename[, 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)	Saves the estimator ensemble to a folder.
train_all(**kwargs)	Trains all estimators.
train_one(i, **kwargs)	Trains an individual estimator.

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_filename*, *n_events=1*, *vote_expectation_weight=None*, *return_individual_predictions=False*)

Calculates the expected Fisher information matrices for each estimator, and then returns the ensemble mean and 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 I is calculated as $I = \sum_i w_i I_i$ with weights $w_i = \exp(-\text{vote\_expectation\_weight} \cdot |E[t_i]|) / \sum_j \exp(-\text{vote\_expectation\_weight} \cdot |E[t_k]|)$. Here I_i are the individual estimators and $E[t_i]$ is the expectation value calculated by *calculate_expectation()*.

Parameters

x_filename [str] Path to an unweighted sample of observations, as saved by the *madminer.sampling.SampleAugmenter* functions. Note that this sample has to be sample from the reference parameter where the score is estimated with the SALLY / SALLINO estimator!

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

vote_expectation_weight [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 None, or if *calculate_expectation()* has not been called, all estimators are treated equal. Default value: None.

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

Returns

mean_prediction [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,)*).

covariance [ndarray] The covariance matrix of the Fisher information estimate. 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)*.

weights [ndarray] Only returned if return_individual_predictions is True. The estimator weights *w_i*.

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

evaluate(*x_filename*, *theta0_filename=None*, *theta1_filename=None*, *test_all_combinations=True*, *vote_expectation_weight=None*, *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_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 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 None, optional] Factor that determines how much more weight is given to those estimators with small expectation value (as calculated by *calculate_expectation()*). If None, or if *calculate_expectation()* has not been called, all estimators are treated equal. Default value: None.

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

Returns

mean_prediction [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,)*).

covariance [ndarray] The covariance matrix of the (flattened) predictions.

weights [ndarray] Only returned if return_individual_predictions is True. The estimator weights *w_i*.

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*)

Saves the estimator ensemble to a folder.

Parameters

folder [str] Path to the folder.

Returns

None

train_all(***kwargs*)

Trains all estimators.

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.

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**(*debug=False*)

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.

Parameters

debug [bool, optional] If True, additional detailed debugging output is printed. Default value: False.

Methods

calculate_fisher_information(x_filename[, ...])	Calculates the expected Fisher information matrix based on the kinematic information in a given number of events.
evaluate(x_filename[, theta0_filename, ...])	Evaluates a trained estimator of the likelihood ratio (or, if method is 'sally' or 'sallino', the score).
load(filename)	Loads a trained model from files.
save(filename)	Saves the trained model to two files: a JSON file with the settings, as well as a pickled pyTorch state dict file.
train(method, x_filename[, y_filename, ...])	Trains a neural network to estimate either the likelihood ratio or, if method is 'sally' or 'sallino', the score.

calculate_fisher_information(*x_filename*, *n_events=1*)

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_filename [str] Path to an unweighted sample of observations, as saved by the *madminer.sampling.SampleAugmenter* functions. Note that this sample has to be sample from the reference parameter where the score is estimated with the SALLY / SALLINO estimator!

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

Returns

fisher_information [ndarray] Expected kinematic Fisher information matrix with shape *(n_parameters, n_parameters)*.

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

Evaluates a trained estimator of the likelihood ratio (or, if method is 'sally' or 'sallino', the score).

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 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

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 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)*.

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*)

Saves the trained model to two files: a JSON file with the settings, as well as a pickled pyTorch state dict file.

Parameters

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

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='maf'*, *n_hidden=(100, 100, 100)*, *activation='tanh'*, *maf_n_mades=3*, *maf_batch_norm=True*, *maf_batch_norm_alpha=0.1*, *maf_mog_n_components=10*, *alpha=1.0*, *n_epochs=20*, *batch_size=128*, *initial_lr=0.002*, *final_lr=0.0001*, *validation_split=0.2*, *early_stopping=True*)

Trains a neural network to estimate either the likelihood ratio or, if method is 'sally' or 'sallino', 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: 'maf'.

n_hidden [int, optional] Number of hidden layers in the neural networks. If method is 'nde' or 'scandal', this refers to the number of hidden layers in each individual MADE layer. Default value: 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: True.

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.

n_epochs [int, optional] Number of epochs. Default value: 20.

batch_size [int, optional] Batch size. Default value: 128.

initial_lr [float, optional] Learning rate during the first epoch, after which it exponentially decays to final_lr. Default value: 0.002.

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

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: 0.2.

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

Returns

None

CHAPTER

FIVE

MADMINER.MORPHING MODULE

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

Bases: object

Morphing functionality. 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)*.

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 not 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 not 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

CHAPTER

SIX

MADMINER.PLOTTING MODULE

madminer.plotting.**kinematic_distribution_of_information**(*xbins*, *xlabel*, *xmin*, *xmax*, *xsecs*, *matrices*, *matrices_aux*, *filename*, *ylabel_addition=''*, *log_xsec=False*, *norm_xsec=True*, *show_aux=False*, *show_labels=False*, *label_pos_information=(0.0, 0.0)*, *label_pos_sm=(0.0, 0.0)*, *label_pos_bsm=(0.0, 0.0)*, *label_pos_bkg=(0.0, 0.0)*, *label_bsm=''*)

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_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**(*matrices*, *matrices_for_determinants*, *labels*, *categories*, *operatorlabels*, *filename*, *additional_label=''*, *top_label=''*, *normalise_determinants=False*, *use_bar_colors=False*, *eigenvalue_operator_legend=True*)

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.

CHAPTER

SEVEN

MADMINER.SAMPLING MODULE

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

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_ratio()* creates training samples for non-local, ratio-based methods like RASCAL, ALICE, and SCANDAL.
- *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.

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.

debug [bool, optional] If True, additional detailed debugging output is printed. Default value: False.

Methods

extract_cross_sections(theta)	Calculates the total cross sections for all specified thetas.
extract_raw_data([theta])	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_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 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*)

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

Parameters

theta [None or ndarray] 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)* 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_local(*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). 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.

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_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 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 (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.

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.

madminer.sampling.**combine_and_shuffle**(*input_filenames*, *output_filename*, *overwrite_existing_file=True*, *debug=False*)

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.

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

debug [bool, optional] If True, additional detailed debugging output is printed. Default value: False.

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

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

m

INDEX