LUMIN

LUMIN Contributors

Feb 10, 2020

PACKAGE REFERENCE

Lumin Unifies Many Improvements for Networks

LUMIN aims to become a deep-learning and data-analysis ecosystem for High-Energy Physics, and perhaps other scientific domains in the future. Similar to Keras and fastai it is a wrapper framework for a graph computation library (PyTorch), but includes many useful functions to handle domain-specific requirements and problems. It also intends to provide easy access to to state-of-the-art methods, but still be flexible enough for users to inherit from base classes and override methods to meet their own demands.

CHAPTER

ONE

LUMIN.DATA_PROCESSING PACKAGE

1.1 Submodules

1.2 lumin.data_processing.file_proc module

`lumin.data_processing.file_proc.`**`save_to_grp`**(*arr*, *grp*, *name*)

Save Numpy array as a dataset in an h5py Group

Parameters

- **arr** (`ndarray`) – array to be saved
- **grp** (`Group`) – group in which to save arr
- **name** (`str`) – name of dataset to create

Return type `None`

`lumin.data_processing.file_proc.`**`fold2foldfile`**(*df*, *out_file*, *fold_idx*, *cont_feats*, *cat_feats*, *targ_feats*, *targ_type*, *misc_feats=None*, *wgt_feat=None*, *matrix_lookup=None*, *matrix_missing=None*, *matrix_shape=None*)

Save fold of data into an h5py Group

Parameters

- **df** (`DataFrame`) – Dataframe from which to save data
- **out_file** (`File`) – h5py file to save data in
- **fold_idx** (`int`) – ID for the fold; used name h5py group according to 'fold_{fold_idx}'
- **cont_feats** (`List[str]`) – list of columns in df to save as continuous variables
- **cat_feats** (`List[str]`) – list of columns in df to save as discreet variables
- **targ_feats** (`Union[str, List[str]]`) – (list of) column(s) in df to save as target feature(s)
- **targ_type** (`Any`) – type of target feature, e.g. int,'float32'
- **misc_feats** (`Optional[List[str]]`) – any extra columns to save
- **wgt_feat** (`Optional[str]`) – column to save as data weights
- **matrix_vecs** – list of objects for matrix encoding, i.e. feature prefixes

- **matrix_feats_per_vec** – list of features per vector for matrix encoding, i.e. feature suffixes. Features listed but not present in df will be replaced with NaN.
- **matrix_row_wise** – whether objects encoded as a matrix should be encoded row-wise (i.e. all the features associated with an object are in their own row), or column-wise (i.e. all the features associated with an object are in their own column)

Return type `None`

`lumin.data_processing.file_proc.`**`df2foldfile`**(*df*, *n_folds*, *cont_feats*, *cat_feats*, *targ_feats*, *savename*, *targ_type*, *strat_key=None*, *misc_feats=None*, *wgt_feat=None*, *cat_maps=None*, *matrix_vecs=None*, *matrix_feats_per_vec=None*, *matrix_row_wise=None*)

Convert dataframe into h5py file by splitting data into sub-folds to be accessed by a `FoldYielder`

Parameters

- **df** (`DataFrame`) – Dataframe from which to save data
- **n_folds** (`int`) – number of folds to split df into
- **cont_feats** (`List[str]`) – list of columns in df to save as continuous variables
- **cat_feats** (`List[str]`) – list of columns in df to save as discreet variables
- **targ_feats** (`Union[str, List[str]]`) – (list of) column(s) in df to save as target feature(s)
- **savename** (`Union[Path, str]`) – name of h5py file to create (.h5py extension not required)
- **targ_type** (`str`) – type of target feature, e.g. int,'float32'
- **strat_key** (`Optional[str]`) – column to use for stratified splitting
- **misc_feats** (`Optional[List[str]]`) – any extra columns to save
- **wgt_feat** (`Optional[str]`) – column to save as data weights
- **cat_maps** (`Optional[Dict[str, Dict[int, Any]]]`) – Dictionary mapping categorical features to dictionary mapping codes to categories
- **matrix_vecs** (`Optional[List[str]]`) – list of objects for matrix encoding, i.e. feature prefixes
- **matrix_feats_per_vec** (`Optional[List[str]]`) – list of features per vector for matrix encoding, i.e. feature suffixes. Features listed but not present in df will be replaced with NaN.
- **matrix_row_wise** (`Optional[bool]`) – whether objects encoded as a matrix should be encoded row-wise (i.e. all the features associated with an object are in their own row), or column-wise (i.e. all the features associated with an object are in their own column)

Return type `None`

`lumin.data_processing.file_proc.`**`add_meta_data`**(*out_file*, *feats*, *cont_feats*, *cat_feats*, *cat_maps*, *targ_feats*, *wgt_feat=None*, *matrix_vecs=None*, *matrix_feats_per_vec=None*, *matrix_row_wise=None*)

Adds meta data to foldfile containing information about the data: feature names, matrix information, etc.

`FoldYielder` objects will access this and automatically extract it to save the user from having to manually pass lists of features.

Parameters

- **out_file** (`File`) – h5py file to save data in
- **feats** (`List[str]`) – list of all features in data
- **cont_feats** (`List[str]`) – list of continuous features
- **cat_feats** (`List[str]`) – list of categorical features
- **cat_maps** (`Optional[Dict[str, Dict[int, Any]]]`) – Dictionary mapping categorical features to dictionary mapping codes to categories
- **targ_feats** (`Union[str, List[str]]`) – (list of) target feature(s)
- **wgt_feat** (`Optional[str]`) – name of weight feature
- **matrix_vecs** (`Optional[List[str]]`) – list of objects for matrix encoding, i.e. feature prefixes
- **matrix_feats_per_vec** (`Optional[List[str]]`) – list of features per vector for matrix encoding, i.e. feature suffixes. Features listed but not present in df will be replaced with NaN.
- **matrix_row_wise** (`Optional[bool]`) – whether objects encoded as a matrix should be encoded row-wise (i.e. all the features associated with an object are in their own row), or column-wise (i.e. all the features associated with an object are in their own column)

Return type `None`

1.3 lumin.data_processing.hep_proc module

`lumin.data_processing.hep_proc.`**`to_cartesian`**(*df*, *vec*, *drop=False*)

Vectoriesed conversion of 3-momenta to Cartesian coordinates inplace, optionally dropping old pT,eta,phi features

Parameters

- **df** (`DataFrame`) – DataFrame to alter
- **vec** (`str`) – column prefix of vector components to alter, e.g. 'muon' for columns ['muon_pt', 'muon_phi', 'muon_eta']
- **drop** (`bool`) – Whether to remove original columns and just keep the new ones

Return type `None`

`lumin.data_processing.hep_proc.`**`to_pt_eta_phi`**(*df*, *vec*, *drop=False*)

Vectorised conversion of 3-momenta to pT,eta,phi coordinates inplace, optionally dropping old px,py,pz features

Parameters

- **df** (`DataFrame`) – DataFrame to alter
- **vec** (`str`) – column prefix of vector components to alter, e.g. 'muon' for columns ['muon_px', 'muon_py', 'muon_pz']
- **drop** (`bool`) – Whether to remove original columns and just keep the new ones

Return type `None`

`lumin.data_processing.hep_proc.`**`delta_phi`**(*arr_a*, *arr_b*)

Vectorised computation of modulo 2pi angular seperation of array of angles b from array of angles a, in range [-pi,pi]

Parameters

- **arr_a** (Union[float, ndarray]) – reference angles
- **arr_b** (Union[float, ndarray]) – final angles

Return type Union[float, ndarray]

Returns angular separation as float or np.array

`lumin.data_processing.hep_proc.`**`twist`**(*dphi*, *deta*)

Vectorised computation of twist between vectors (https://arxiv.org/abs/1010.3698)

Parameters

- **dphi** (Union[float, ndarray]) – delta phi separations
- **deta** (Union[float, ndarray]) – delta eta separations

Return type Union[float, ndarray]

Returns angular separation as float or np.array

`lumin.data_processing.hep_proc.`**`add_abs_mom`**(*df*, *vec*, *z=True*)

Vectorised computation 3-momenta magnitude, adding new column in place. Currently only works for Cartesian vectors

Parameters

- **df** (DataFrame) – DataFrame to alter
- **vec** (str) – column prefix of vector components, e.g. 'muon' for columns ['muon_px', 'muon_py', 'muon_pz']
- **z** (bool) – whether to consider the z-component of the momenta

Return type None

`lumin.data_processing.hep_proc.`**`add_mass`**(*df*, *vec*)

Vectorised computation of mass of 4-vector, adding new column in place.

Parameters

- **df** (DataFrame) – DataFrame to alter
- **vec** (str) – column prefix of vector components, e.g. 'muon' for columns ['muon_px', 'muon_py', 'muon_pz']

Return type None

`lumin.data_processing.hep_proc.`**`add_energy`**(*df*, *vec*)

Vectorised computation of energy of 4-vector, adding new column in place.

Parameters

- **df** (DataFrame) – DataFrame to alter
- **vec** (str) – column prefix of vector components, e.g. 'muon' for columns ['muon_px', 'muon_py', 'muon_pz']

Return type None

`lumin.data_processing.hep_proc.`**`add_mt`**(*df*, *vec*, *mpt_name='mpt'*)

Vectorised computation of transverse mass of 4-vector with respect to missing transverse momenta, adding new column in place. Currently only works for pT, eta, phi vectors

Parameters

- **df** (`DataFrame`) – DataFrame to alter
- **vec** (`str`) – column prefix of vector components, e.g. 'muon' for columns ['muon_px', 'muon_py', 'muon_pz']
- **mpt_name** (`str`) – column prefix of vector of missing transverse momenta components, e.g. 'mpt' for columns ['mpt_pT', 'mpt_phi']

`lumin.data_processing.hep_proc.`**`get_vecs`**(*feats*, *strict=True*)

Filter list of features to get list of 3-momenta defined in the list. Works for both pT, eta, phi and Cartesian coordinates. If strict, return only vectors with all coordinates present in feature list.

Parameters

- **feats** (`List[str]`) – list of features to filter
- **strict** (`bool`) – whether to require all 3-momenta components to be present in the list

Return type `Set[str]`

Returns set of unique 3-momneta prefixes

`lumin.data_processing.hep_proc.`**`fix_event_phi`**(*df*, *ref_vec*)

Rotate event in phi such that ref_vec is at phi == 0. Performed inplace. Currently only works on vectors defined in pT, eta, phi

Parameters

- **df** (`DataFrame`) – DataFrame to alter
- **ref_vec** (`str`) – column prefix of vector components to use as reference, e.g. 'muon' for columns ['muon_pT', 'muon_eta', 'muon_phi']

Return type `None`

`lumin.data_processing.hep_proc.`**`fix_event_z`**(*df*, *ref_vec*)

Flip event in z-axis such that ref_vec is in positive z-direction. Performed inplace. Works for both pT, eta, phi and Cartesian coordinates.

Parameters

- **df** (`DataFrame`) – DataFrame to alter
- **ref_vec** (`str`) – column prefix of vector components to use as reference, e.g. 'muon' for columns ['muon_pT', 'muon_eta', 'muon_phi']

Return type `None`

`lumin.data_processing.hep_proc.`**`fix_event_y`**(*df*, *ref_vec_0*, *ref_vec_1*)

Flip event in y-axis such that ref_vec_1 has a higher py than ref_vec_0. Performed in place. Works for both pT, eta, phi and Cartesian coordinates.

Parameters

- **df** (`DataFrame`) – DataFrame to alter
- **ref_vec_0** (`str`) – column prefix of vector components to use as reference 0, e.g. 'muon' for columns ['muon_pT', 'muon_eta', 'muon_phi']

- **ref_vec_1** (str) – column prefix of vector components to use as reference 1, e.g. 'muon' for columns ['muon_pT', 'muon_eta', 'muon_phi']

Return type None

lumin.data_processing.hep_proc.**event_to_cartesian**(*df*, *drop=False*, *ignore=None*)

Convert entire event to Cartesian coordinates, except vectors listed in ignore. Optionally, drop old pT,eta,phi features. Perfomed inplace.

Parameters

- **df** (DataFrame) – DataFrame to alter
- **drop** (bool) – whether to drop old coordinates
- **ignore** (Optional[List[str]]) – vectors to ignore when converting

Return type None

lumin.data_processing.hep_proc.**proc_event**(*df*, *fix_phi=False*, *fix_y=False*, *fix_z=False*, *use_cartesian=False*, *ref_vec_0=None*, *ref_vec_1=None*, *keep_feats=None*, *default_vals=None*)

Process event: Pass data through inplace various conversions and drop uneeded columns. Data expected to consist of vectors defined in pT, eta, phi.

Parameters

- **df** (DataFrame) – DataFrame to alter
- **fix_phi** (bool) – whether to rotate events using fix_event_phi()
- **fix_y** – whether to flip events using fix_event_y()
- **fix_z** – whether to flip events using fix_event_z()
- **use_cartesian** – wether to convert vectors to Cartesian coordinates
- **ref_vec_0** (Optional[str]) – column prefix of vector components to use as reference (0) for :meth:~lumin.data_prcoessing.hep.proc.fix_event_phi', fix_event_y(), and fix_event_z() e.g. 'muon' for columns ['muon_pT', 'muon_eta', 'muon_phi']
- **ref_vec_1** (Optional[str]) – column prefix of vector components to use as reference 1 for fix_event_z(), e.g. 'muon' for columns ['muon_pT', 'muon_eta', 'muon_phi']
- **keep_feats** (Optional[List[str]]) – columns to keep which would otherwise be dropped
- **default_vals** (Optional[List[str]]) – list of default values which might be used to represent missing vector components. These will be replaced with np.nan.

Return type None

lumin.data_processing.hep_proc.**calc_pair_mass**(*df*, *masses*, *feat_map*)

Vectorised computation of invarient mass of pair of particles with given masses, using 3-momenta. Only works for vectors defined in Cartesian coordinates.

Parameters

- **df** (DataFrame) – DataFrame vector components
- **masses** (Union[Tuple[float, float], Tuple[ndarray, ndarray]]) – tuple of masses of particles (either constant or different pair of masses per pair of particles)
- **feat_map** (Dict[str, str]) – dictionary mapping of requested momentum components to the features in df

Return type ndarray

Returns np.array of invarient masses

`lumin.data_processing.hep_proc.boost`(*ref_vec*, *boost_vec*, *df=None*, *rescale_boost=False*)

Vectorised boosting of reference vectors along boosting vectors. N.B. Implementation adapted from ROOT (https://root.cern/)

Parameters

- **vec_0** – either (N,4) array of 4-momenta coordinates for starting vector, or prefix name for starting vector, i.e. columns should have names of the form [vec_0]_px, etc.
- **vec_1** – either (N,4) array of 4-momenta coordinates for boosting vector, or prefix name for boosting vector, i.e. columns should have names of the form [vec_1]_px, etc.
- **df** (Optional[DataFrame]) – DataFrame with data
- **rescale_boost** (bool) – whether to divide the boost vector by its energy

Return type ndarray

Returns (N,4) array of boosted vector in Cartesian coordinates

`lumin.data_processing.hep_proc.boost2cm`(*vec*, *df=None*)

Vectorised computation of boosting vector required to boost a vector to its centre-of-mass frame

Parameters

- **vec** (Union[<built-in function array>, str]) – either (N,4) array of 4-momenta coordinates for starting vector, or prefix name for starting vector, i.e. columns should have names of the form [vec]_px, etc.
- **df** (Optional[DataFrame]) – DataFrame with data is supplying a string *vec*

Return type <built-in function array>

Returns (N,3) array of boosting vector in Cartesian coordinates

`lumin.data_processing.hep_proc.get_momentum`(*df*, *vec*, *include_E=False*, *as_cart=False*)

Extracts array of 3- or 4-momenta coordinates from DataFrame columns

Parameters

- **df** (DataFrame) – DataFrame with data
- **vec** (str) – prefix name for vector, i.e. columns should have names of the form [vec]_px, etc.
- **as_cart** (bool) – if True will return momenta in Cartesian coordinates

Returns (px, py, pz, (E)) or (pT, phi, eta, (E))

Return type (N, 3|4) array with columns

`lumin.data_processing.hep_proc.cos_delta`(*vec_0*, *vec_1*, *df=None*, *name=None*, *inplace=False*)

Vectorised compututation of the cosine of the angular seperation of *vec_1* from *vec_0* If *vec_** are strings, then columns are extracted from DataFrame *df*. If inplace is True Cosine angle is added a new column to the DataFrame with name *cosdelta_[vec_0]_[vec_1]* or *cosdelta*, unless *name* is set

Parameters

- **vec_0** (Union[<built-in function array>, str]) – either (N,3) array of 3-momenta coordinates for vector 0, or prefix name for vector zero, i.e. columns should have names of the form [vec_0]_px, etc.

- **vec_1** (Union[<built-in function array>, str]) – either (N,3) array of 3-momenta coordinates for vector 1, or prefix name for vector one, i.e. columns should have names of the form [vec_1]_px, etc.
- **df** (Optional[DataFrame]) – DataFrame with data
- **name** (Optional[str]) – if set, will create a new column in df for cosdelta with given name, otherwise will generate a name
- **inplace** (bool) – if True will add new column to df, otherwise will return array of cos_deltas

Return type Union[None, <built-in function array>]

Returns array of cos deltas in not inplace

lumin.data_processing.hep_proc.**delta_r**(*dphi*, *deta*)

Vectorised computation of delta R separation for arrays of delta phi and delta eta (rapidity or pseudorapidity)

Parameters

- **dphi** (Union[float, ndarray]) – delta phi separations
- **deta** (Union[float, ndarray]) – delta eta separations

Return type Union[float, ndarray]

Returns delta R separation as float or np.array

lumin.data_processing.hep_proc.**delta_r_boosted**(*vec_0*, *vec_1*, *ref_vec*, *df=None*, *name=None*, *inplace=False*)

Vectorised compututation of the deltaR seperation of *vec_1* from *vec_0* in the rest-frame of another vector If *vec_** are strings, then columns are extracted from DataFrame *df*. If inplace is True deltaR is added a new column to the DataFrame with name *dR_[vec_0]_[vec_1]_boosted_[ref_vec]* or *dR_boosted*, unless *name* is set

Parameters

- **vec_0** (Union[<built-in function array>, str]) – either (N,4) array of 4-momenta coordinates for vector 0, in Cartesian coordinates or prefix name for vector zero, i.e. columns should have names of the form [vec_0]_px, etc.
- **vec_1** (Union[<built-in function array>, str]) – either (N,4) array of 4-momenta coordinates for vector 1, in Cartesian coordinates or prefix name for vector one, i.e. columns should have names of the form [vec_1]_px, etc.
- **ref_vec** (Union[<built-in function array>, str]) – either (N,4) array of 4-momenta coordinates for the vector in whos rest-frame deltaR should be computed, in Cartesian coordinates or prefix name for reference vector, i.e. columns should have names of the form [ref_vec]_px, etc.
- **df** (Optional[DataFrame]) – DataFrame with data
- **name** (Optional[str]) – if set, will create a new column in df for cosdelta with given name, otherwise will generate a name
- **inplace** (bool) – if True will add new column to df, otherwise will return array of cos_deltas

Return type Union[None, <built-in function array>]

Returns array of boosted deltaR in not inplace

1.4 lumin.data_processing.pre_proc module

`lumin.data_processing.pre_proc.`**`get_pre_proc_pipes`**(*norm_in=True*, *norm_out=False*, *pca=False*, *whiten=False*, *with_mean=True*, *with_std=True*, *n_components=None*)

Configure SKLearn Pipelines for processing inputs and targets with the requested transformations.

Parameters

- **norm_in** (bool) – whether to apply StandardScaler to inputs
- **norm_out** (bool) – whether to apply StandardScaler to outputs
- **pca** (bool) – whether to apply PCA to inputs. Perforemed prior to StandardScaler. No dimensionality reduction is applied, purely rotation.
- **whiten** (bool) – whether PCA should whiten inputs.
- **with_mean** (bool) – whether StandardScalers should shift means to 0
- **with_std** (bool) – whether StandardScalers should scale standard deviations to 1
- **n_components** (Optional[int]) – if set, causes PCA to reduce the dimensionality of the input data

Return type Tuple[Pipeline, Pipeline]

Returns Pipeline for input data Pipeline for target data

`lumin.data_processing.pre_proc.`**`fit_input_pipe`**(*df*, *cont_feats*, *savename=None*, *input_pipe=None*, *norm_in=True*, *pca=False*, *whiten=False*, *with_mean=True*, *with_std=True*, *n_components=None*)

Fit input pipeline to continuous features and optionally save.

Parameters

- **df** (DataFrame) – DataFrame with data to fit pipeline
- **cont_feats** (Union[str, List[str]]) – (list of) column(s) to use as input data for fitting
- **savename** (Optional[str]) – if set will save the fitted Pipeline to with that name as Pickle (.pkl extension added automatically)
- **input_pipe** (Optional[Pipeline]) – if set will fit, otherwise will instantiate a new Pipeline
- **norm_in** (bool) – whether to apply StandardScaler to inputs. Only used if input_pipe is not set.
- **pca** (bool) – whether to apply PCA to inputs. Perforemed prior to StandardScaler. No dimensionality reduction is applied, purely rotation. Only used if input_pipe is not set.
- **whiten** (bool) – whether PCA should whiten inputs. Only used if input_pipe is not set.
- **with_mean** (bool) – whether StandardScalers should shift means to 0. Only used if input_pipe is not set.
- **with_std** (bool) – whether StandardScalers should scale standard deviations to 1. Only used if input_pipe is not set.

- **n_components** (Optional[int]) – if set, causes PCA to reduce the dimensionality of the input data. Only used if input_pipe is not set.

Return type Pipeline

Returns Fitted Pipeline

lumin.data_processing.pre_proc.**fit_output_pipe**(*df*, *targ_feats*, *savename=None*, *output_pipe=None*, *norm_out=True*)

Fit output pipeline to target features and optionally save. Have you thought about using a y_range for regression instead?

Parameters

- **df** (DataFrame) – DataFrame with data to fit pipeline
- **targ_feats** (Union[str, List[str]]) – (list of) column(s) to use as input data for fitting
- **savename** (Optional[str]) – if set will save the fitted Pipeline to with that name as Pickle (.pkl extension added automatically)
- **output_pipe** (Optional[Pipeline]) – if set will fit, otherwise will instantiate a new Pipeline
- **norm_out** (bool) – whether to apply StandardScaler to outputs . Only used if output_pipe is not set.

Return type Pipeline

Returns Fitted Pipeline

lumin.data_processing.pre_proc.**proc_cats**(*train_df*, *cat_feats*, *val_df=None*, *test_df=None*)

Process categorical features in train_df to be valued 0->cardinality-1. Applied inplace. Applies same transformation to validation and testing data is passed. Will complain if validation or testing sets contain categories which are not present in the training data.

Parameters

- **train_df** (DataFrame) – DataFrame with the training data, which will also be used to specify all the categories to consider
- **cat_feats** (List[str]) – list of columns to use as categorical features
- **val_df** (Optional[DataFrame]) – if set will apply the same category to code mapping to the validation data as was performed on the training data
- **test_df** (Optional[DataFrame]) – if set will apply the same category to code mapping to the testing data as was performed on the training data

Return type Tuple[OrderedDict, OrderedDict]

Returns ordered dictionary mapping categorical features to dictionaries mapping categories to codes ordered dictionary mapping categorical features to their cardinalities

1.5 Module contents

CHAPTER

TWO

LUMIN.EVALUATION PACKAGE

2.1 Submodules

2.2 lumin.evaluation.ams module

`lumin.evaluation.ams.`**`calc_ams`**(*s*, *b*, *br=0*, *unc_b=0*)

Compute Approximate Median Significance (https://arxiv.org/abs/1007.1727)

Parameters

- **s** (`float`) – signal weight
- **b** (`float`) – background weight
- **br** (`float`) – background offset bias
- **unc_b** (`float`) – fractional systemtatic uncertainty on background

Return type `float`

Returns Approximate Median Significance if b > 0 else -1

`lumin.evaluation.ams.`**`calc_ams_torch`**(*s*, *b*, *br=0*, *unc_b=0*)

Compute Approximate Median Significance (https://arxiv.org/abs/1007.1727) using Tensor inputs

Parameters

- **s** (`Tensor`) – signal weight
- **b** (`Tensor`) – background weight
- **br** (`float`) – background offset bias
- **unc_b** (`float`) – fractional systemtatic uncertainty on background

Return type `Tensor`

Returns Approximate Median Significance if b > 0 else 1e-18 * s

`lumin.evaluation.ams.`**`ams_scan_quick`**(*df*, *wgt_factor=1*, *br=0*, *syst_unc_b=0*, *pred_name='pred'*, *targ_name='gen_target'*, *wgt_name='gen_weight'*)

Scan accross a range of possible prediction thresholds in order to maximise the Approximate Median Significance (https://arxiv.org/abs/1007.1727). Note that whilst this method is quicker than `ams_scan_slow()`, it sufferes from float precison. Not recommended for final evaluation.

Parameters

- **df** (`DataFrame`) – DataFrame containing prediction data

- **wgt_factor** (float) – factor to reweight signal and background weights
- **br** (float) – background offset bias
- **syst_unc_b** (float) – fractional systemtatic uncertainty on background
- **pred_name** (str) – column to use as predictions
- **targ_name** (str) – column to use as truth labels for signal and background
- **wgt_name** (str) – column to use as weights for signal and background events

Return type Tuple[float, float]

Returns maximum AMS prediction threshold corresponding to maximum AMS

lumin.evaluation.ams.**ams_scan_slow**(*df, wgt_factor=1, br=0, syst_unc_b=0, use_stat_unc=False, start_cut=0.9, min_events=10, pred_name='pred', targ_name='gen_target', wgt_name='gen_weight', show_prog=True*)

Scan accross a range of possible prediction thresholds in order to maximise the Approximate Median Significance (https://arxiv.org/abs/1007.1727). Note that whilst this method is slower than *ams_scan_quick()*, it does not suffer as much from float precison. Additionally it allows one to account for statistical uncertainty in AMS calculation.

Parameters

- **df** (DataFrame) – DataFrame containing prediction data
- **wgt_factor** (float) – factor to reweight signal and background weights
- **br** (float) – background offset bias
- **syst_unc_b** (float) – fractional systemtatic uncertainty on background
- **use_stat_unc** (bool) – whether to account for the statistical uncertainty on the background
- **start_cut** (float) – minimum prediction to consider; useful for speeding up scan
- **min_events** (int) – minimum number of background unscaled events required to pass threshold
- **pred_name** (str) – column to use as predictions
- **targ_name** (str) – column to use as truth labels for signal and background
- **wgt_name** (str) – column to use as weights for signal and background events
- **show_prog** (bool) – whether to display progress and ETA of scan

Return type Tuple[float, float]

Returns maximum AMS prediction threshold corresponding to maximum AMS

2.3 Module contents

CHAPTER

THREE

LUMIN.INFERENCE PACKAGE

3.1 Submodules

3.2 lumin.inference.summary_stat module

`lumin.inference.summary_stat.`**`bin_binary_class_pred`**(*df, max_unc, consider_samples=None, step_sz=0.001, pred_name='pred', sample_name='gen_sample', compact_samples=False, class_name='gen_target', add_pure_signal_bin=False, max_unc_pure_signal=0.1, verbose=True*)

Define bin-edges for binning particle process samples as a function of event class prediction (signal | background) such that the statistical uncertainties on per bin yields are below max_unc for each considered sample.

Parameters

- **df** (`DataFrame`) – DataFrame containing the data
- **max_unc** (`float`) – maximum fractional statisitcal uncertainty to allow when defining bins
- **consider_samples** (`Optional[List[str]]`) – if set, only listed samples are considered when defining bins
- **step_sz** (`float`) – resolution of scan along event prediction
- **pred_name** (`str`) – column to use as event class prediction
- **sample_name** (`str`) – column to use as particle process fo reach event
- **compact_samples** (`bool`) – if true, will not consider samples when computing bin edges, only the class
- **class_name** (`str`) – name of column to use as class indicator
- **add_pure_signal_bin** (`bool`) – if true will attempt to add a bin which oonly contains signal (class 1) if the fractional bin-fill uncertainty would be less than max_unc_pure_signal
- **max_unc_pure_signal** (`float`) – maximum fractional statisitcal uncertainty to allow when defining pure-signal bins
- **verbose** (`bool`) – whether to show progress bar

Return type `List[float]`

Returns list of bin edges

3.3 Module contents

CHAPTER

FOUR

LUMIN.NN PACKAGE

4.1 Subpackages

4.2 Module contents

CHAPTER

FIVE

LUMIN.OPTIMISATION PACKAGE

5.1 Submodules

5.2 lumin.optimisation.features module

`lumin.optimisation.features.`**`get_rf_feat_importance`**(*rf*, *inputs*, *targets*, *weights=None*)

Compute feature importance for a Random Forest model using rfpimp.

Parameters

- **`rf`** (`ForestRegressor`) – trained Random Forest model
- **`inputs`** (`DataFrame`) – input data as Pandas DataFrame
- **`targets`** (`ndarray`) – target data as Numpy array
- **`weights`** (`Optional[ndarray]`) – Optional data weights as Numpy array

Return type `DataFrame`

`lumin.optimisation.features.`**`rf_rank_features`**(*train_df*, *val_df*, *objective*, *train_feats*, *targ_name='gen_target'*, *wgt_name=None*, *importance_cut=0.0*, *n_estimators=40*, *rf_params=None*, *optimise_rf=True*, *n_rfs=1*, *n_max_display=30*, *plot_results=True*, *retrain_on_import_feats=True*, *verbose=True*, *savename=None*, *plot_settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Compute relative permutation importance of input features via using Random Forests. A reduced set of 'important features' is obtained by cutting on relative importance and a new model is trained and evaluated on this reduced set. RFs will have their hyper-parameters roughly optimised, both when training on all features and once when training on important features. Relative importances may be computed multiple times (via n_rfs) and averaged. In which case the standard error is also computed.

Parameters

- **`train_df`** (`DataFrame`) – training data as Pandas DataFrame
- **`val_df`** (`DataFrame`) – validation data as Pandas DataFrame
- **`objective`** (`str`) – string representation of objective: either 'classification' or 'regression'
- **`train_feats`** (`List[str]`) – complete list of training features

- **targ_name** (str) – name of column containing target data
- **wgt_name** (Optional[str]) – name of column containing weight data. If set, will use weights for training and evaluation, otherwise will not
- **importance_cut** (float) – minimum importance required to be considered an 'important feature'
- **n_estimators** (int) – number of trees to use in each forest
- **rf_params** (Optional[Dict[str, Any]]) – optional dictionary of keyword parameters for SK-Learn Random Forests Or ordered dictionary mapping parameters to optimise to list of values to consider If None and will optimise parameters using *lumin.optimisation.hyper_param.get_opt_rf_params()*
- **optimise_rf** (bool) – if true will optimise RF params, passing *rf_params* to *get_opt_rf_params()*
- **n_rfs** (int) – number of trainings to perform on all training features in order to compute importances
- **n_max_display** (int) – maximum number of features to display in importance plot
- **plot_results** (bool) – whether to plot the feature importances
- **retrain_on_import_feats** (bool) – whether to train a new model on important features to compare to full model
- **verbose** (bool) – whether to report results and progress
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **plot_settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type List[str]

Returns List of features passing importance_cut, ordered by decreasing importance

lumin.optimisation.features.**rf_check_feat_removal**(*train_df*, *objective*, *train_feats*, *check_feats*, *targ_name='gen_target'*, *wgt_name=None*, *val_df=None*, *subsample_rate=None*, *strat_key=None*, *n_estimators=40*, *n_rfs=1*, *rf_params=None*)

Checks whether features can be removed from the set of training features without degrading model performance using Random Forests Computes scores for model with all training features then for each feature listed in *check_feats* computes scores for a model trained on all training features except that feature E.g. if two features are highly correlated this function could be used to check whether one of them could be removed.

Parameters

- **train_df** (DataFrame) – training data as Pandas DataFrame
- **objective** (str) – string representation of objective: either 'classification' or 'regression'
- **train_feats** (List[str]) – complete list of training features
- **check_feats** (List[str]) – list of features to try removing
- **targ_name** (str) – name of column containing target data

- **wgt_name** (Optional[str]) – name of column containing weight data. If set, will use weights for training and evaluation, otherwise will not
- **val_df** (Optional[DataFrame]) – optional validation data as Pandas DataFrame. If set will compute validation scores in addition to Out Of Bag scores And will optimise RF parameters if *rf_params* is None
- **subsample_rate** (Optional[float]) – if set, will subsample the training data to the provided fraction. Subsample is repeated per Random Forest training
- **strat_key** (Optional[str]) – column name to use for stratified subsampling, if desired
- **n_estimators** (int) – number of trees to use in each forest
- **n_rfs** (int) – number of trainings to perform on all training features in order to compute importances
- **rf_params** (Optional[Dict[str, Any]]) – optional dictionary of keyword parameters for SK-Learn Random Forests If None and val_df is None will use default parameters of 'min_samples_leaf':3, 'max_features':0.5 Elif None and val_df is not None will optimise parameters using `lumin.optimisation.hyper_param.get_opt_rf_params()`

Return type Dict[str, float]

Returns Dictionary of results

lumin.optimisation.features.**repeated_rf_rank_features**(*train_df, val_df, n_reps, min_frac_import, objective, train_feats, targ_name='gen_target', wgt_name=None, strat_key=None, subsample_rate=None, resample_val=True, importance_cut=0.0, n_estimators=40, rf_params=None, optimise_rf=True, n_rfs=1, n_max_display=30, n_threads=1, savename=None, plot_settings=<lumin.plotting.plot_settings.PlotSettin object>*)

Runs `rf_rank_features()` multiple times on bootstrap resamples of training data and computes the fraction of times each feature passes the importance cut. Then returns a list features which are have a fractional selection as important great than some number. I.e. in cases where `rf_rank_features()` can be unstable (list of important features changes each run), this method can be used to help stabailse the list of important features

Parameters

- **train_df** (DataFrame) – training data as Pandas DataFrame
- **val_df** (DataFrame) – validation data as Pandas DataFrame
- **n_reps** (int) – number of times to resample and run `rf_rank_features()`
- **min_frac_import** (float) – minimum fraction of times feature must be selected as important by `rf_rank_features()` in order to be considered generally important

- **objective** (`str`) – string representation of objective: either 'classification' or 'regression'
- **train_feats** (`List[str]`) – complete list of training features
- **targ_name** (`str`) – name of column containing target data
- **wgt_name** (`Optional[str]`) – name of column containing weight data. If set, will use weights for training and evaluation, otherwise will not
- **strat_key** (`Optional[str]`) – name of column to use to stratify data when resampling
- **subsample_rate** (`Optional[float]`) – if set, will subsample the training data to the provided fraction. Subsample is repeated per Random Forest training
- **resample_val** (`bool`) – whether to also resample the validation set, or use the original set for all evaluations
- **importance_cut** (`float`) – minimum importance required to be considered an 'important feature'
- **n_estimators** (`int`) – number of trees to use in each forest
- **rf_params** (`Optional[Dict[str, Any]]`) – optional dictionary of keyword parameters for SK-Learn Random Forests Or ordered dictionary mapping parameters to optimise to list of values to consider If None and will optimise parameters using `lumin.optimisation.hyper_param.get_opt_rf_params()`
- **optimise_rf** (`bool`) – if true will optimise RF params, passing *rf_params* to `get_opt_rf_params()`
- **n_rfs** (`int`) – number of trainings to perform on all training features in order to compute importances
- **n_max_display** (`int`) – maximum number of features to display in importance plot
- **n_threads** (`int`) – number of rankings to run simultaneously
- **savename** (`Optional[str]`) – Optional name of file to which to save the plot of feature importances
- **plot_settings** (`PlotSettings`) – `PlotSettings` class to control figure appearance

Return type `Tuple[List[str], DataFrame]`

Returns

- List of features with fractional selection greater than min_frac_import, ordered by decreasing fractional selection
- DataFrame of number of selections and fractional selections for all features

`lumin.optimisation.features.`**`auto_filter_on_linear_correlation`**(*train_df*, *val_df*, *check_feats*, *objective*, *targ_name*, *strat_key=None*, *wgt_name=None*, *corr_threshold=0.8*, *n_estimators=40*, *rf_params=None*, *optimise_rf=True*, *n_rfs=5*, *subsample_rate=None*, *savename=None*, *plot_settings=<lumin.plotting.plot_setting object>*)

Filters a list of possible training features by identifying pairs of linearly correlated features and then attempting to remove either feature from each pair by checking whether doing so would not decrease the performance Random Forests trained to perform classification or regression.

Linearly correlated features are identified by computing Spearman's rank-order correlation coefficients for every pair of features. Hierachical clustering is then used to group features. Pairs with a correlation coefficient greater than a set threshold are candidates for removal. Candidate pairs are tested, in order of decreasing correlation, by computing the mean performance of a Random Forests trained on: all remaining training features; all remaining training features except the first feature in the pair; and all remaining training features except the second feature in the pair. If the RF trained on all remaining features consistently outperforms the other two trainings, then neither feature from the pair is removed, otherwise the feature whose removal causes the largest mean increase in performance is removed.

Since multiple features maybe correlated with one-another, but this function examines paris of features, it might be necessary/desirable to rerun it on the the previous results.

Since this function involves training many models, it can be slow on large datasets. In such cases one can use the *subsample_rate* argument to sample randomly a fraction of the whole dataset (with optionaly stratification). Resampling is performed prior to each RF training for maximum genralisation, and any weights in the data are automatically renormalised to the original weight sum (within each class).

> **Attention:** This function combines `plot_rank_order_dendrogram()` with `rf_check_feat_removal()`. This is purely for convenience and should not be treated as a 'black box'. We encourage users to convince themselves that it is really is reasonable to remove the features which are identified as redundant.

Parameters

- **`train_df`** (`DataFrame`) – training data as Pandas DataFrame
- **`val_df`** (`DataFrame`) – validation data as Pandas DataFrame
- **`check_feats`** (`List[str]`) – complete list of features to consider for training and removal
- **`objective`** (`str`) – string representation of objective: either 'classification' or 'regression'
- **`targ_name`** (`str`) – name of column containing target data
- **`strat_key`** (`Optional[str]`) – name of column to use to stratify data when resampling

- **wgt_name** (Optional[str]) – name of column containing weight data. If set, will use weights for training and evaluation, otherwise will not
- **corr_threshold** (float) – minimum threshold on Spearman's rank-order correlation coefficient for pairs to be considered 'correlated'
- **n_estimators** (int) – number of trees to use in each forest
- **rf_params** (Optional[Dict[~KT, ~VT]]) – either: a dictionare of keyword hyper-parameters to use for the Random Forests, if optimse_rf is False; or an *OrderedDict* of a range of hyper-parameters to test during optimisation. See *get_opt_rf_params()* for more details.
- **optimise_rf** (bool) – whether to optimise the Random Forest hyper-parameters for the (sub-sambled) dataset
- **n_rfs** (int) – number of trainings to perform during each perfromance impact test
- **subsample_rate** (Optional[float]) – float between 0 and 1. If set will subsample the trainng data to the requested fraction
- **savename** (Optional[str]) – Optional name of file to which to save the first plot of feature clustering
- **plot_settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type List[str]

Returns Filtered list of training features

lumin.optimisation.features.**auto_filter_on_mutual_dependence**(*train_df*, *val_df*, *check_feats*, *objective*, *targ_name*, *strat_key=None*, *wgt_name=None*, *md_threshold=0.8*, *n_estimators=40*, *rf_params=None*, *optimise_rf=True*, *n_rfs=5*, *subsample_rate=None*, *plot_settings=<lumin.plotting.plot_settings.object>*)

Filters a list of possible training features via mutual dependence: By identifying features whose values can be accurately predicted using the other features. Features with a high 'dependence' are then checked to see whether removing them would not decrease the performance Random Forests trained to perform classification or regression. For best results, the features to check should be supplied in order to decreasing importance.

Dependent features are identified by training Random Forest regressors on the other features. Features with a dependence greater than a set threshold are candidates for removal. Candidate features are tested, in order of increasing importance, by computing the mean performance of a Random Forests trained on: all remaining training features; and all remaining training features except the candidate feature. If the RF trained on all remaining features except the candidate feature consistently outperforms or matches the training which uses all remaining features, then the candidate feature is removed, otherwise the feature remains and is no longer tested.

Since evaluating the mutual dependence via regression then allows the important features used by the regressor to be identified, it is possible to test multiple feature removals at once, provided a removal candidate is not important for predicting another removal candidate.

Since this function involves training many models, it can be slow on large datasets. In such cases one can use the *subsample_rate* argument to sample randomly a fraction of the whole dataset (with optionaly stratification). Resampling is performed prior to each RF training for maximum genralisation, and any weights in the data are automatically renormalised to the original weight sum (within each class).

Attention: This function combines RFPImp's *feature_dependence_matrix* with `rf_check_feat_removal()`. This is purely for convenience and should not be treated as a 'black box'. We encourage users to convince themselves that it is really is reasonable to remove the features which are identified as redundant.

Note: Technicalities related to RFPImp's use of SVG for plots mean that the mutual dependence plots can have low resolution when shown or saved. Therefore this function does not take a *savename* argument. Users wiching to save the plots as PNG or PDF should compute the dependence matrix themselves using *feature_dependence_matrix* and then plot using *plot_dependence_heatmap*, calling *.save([savename])* on the retunred object. The plotting backend might need to be set to SVG, using: *%config InlineBackend.figure_format = 'svg'*.

Parameters

- **train_df** (`DataFrame`) – training data as Pandas DataFrame
- **val_df** (`DataFrame`) – validation data as Pandas DataFrame
- **check_feats** (`List[str]`) – complete list of features to consider for training and removal
- **objective** (`str`) – string representation of objective: either 'classification' or 'regression'
- **targ_name** (`str`) – name of column containing target data
- **strat_key** (`Optional[str]`) – name of column to use to stratify data when resampling
- **wgt_name** (`Optional[str]`) – name of column containing weight data. If set, will use weights for training and evaluation, otherwise will not
- **md_threshold** (`float`) – minimum threshold on the mutual dependence coefficient for a feature to be considered 'predictable'
- **n_estimators** (`int`) – number of trees to use in each forest
- **rf_params** (`Optional[OrderedDict]`) – either: a dictionare of keyword hyper-parameters to use for the Random Forests, if optimse_rf is False; or an *OrderedDict* of a range of hyper-parameters to test during optimisation. See `get_opt_rf_params()` for more details.
- **optimise_rf** (`bool`) – whether to optimise the Random Forest hyper-parameters for the (sub-sambled) dataset
- **n_rfs** (`int`) – number of trainings to perform during each perfromance impact test
- **subsample_rate** (`Optional[float]`) – float between 0 and 1. If set will subsample the trainng data to the requested fraction
- **plot_settings** (`PlotSettings`) – `PlotSettings` class to control figure appearance

Return type `List[str]`

Returns Filtered list of training features

5.3 lumin.optimisation.hyper_param module

`lumin.optimisation.hyper_param.`**`get_opt_rf_params`**(*x_trn, y_trn, x_val, y_val, objective, w_trn=None, w_val=None, params=None, n_estimators=40, verbose=True*)

Use an ordered parameter-scan to roughly optimise Random Forest hyper-parameters.

Parameters

- **x_trn** (ndarray) – training input data
- **y_trn** (ndarray) – training target data
- **x_val** (ndarray) – validation input data
- **y_val** (ndarray) – validation target data
- **objective** (str) – string representation of objective: either 'classification' or 'regression'
- **w_trn** (Optional[ndarray]) – training weights
- **w_val** (Optional[ndarray]) – validation weights
- **params** (Optional[OrderedDict]) – ordered dictionary mapping parameters to optimise to list of values to cosnider
- **n_estimators** (int) – number of trees to use in each forest
- **verbose** – Print extra information and show a live plot of model performance

Returns dictionary mapping parameters to their optimised values rf: best performing Random Forest

Return type params

`lumin.optimisation.hyper_param.`**`fold_lr_find`**(*fy, model_builder, bs, train_on_weights=True, shuffle_fold=True, n_folds=-1, lr_bounds=[1e-05, 10], callback_partials=None, plot_settings=<lumin.plotting.plot_settings.PlotSettings object>, bulk_move=True*)

Wrapper function for training using `LRFinder` which runs a Smith LR range test (https://arxiv.org/abs/1803.09820) using folds in `FoldYielder`. Trains models for 1 fold, interpolating LR between set bounds. This repeats for each fold in `FoldYielder`, and loss evolution is averaged.

Parameters

- **fy** (FoldYielder) – `FoldYielder` providing training data
- **model_builder** (ModelBuilder) – `ModelBuilder` providing networks and optimisers
- **bs** (int) – batch size
- **train_on_weights** (bool) – If weights are present, whether to use them for training
- **shuffle_fold** (bool) – whether to shuffle data in folds
- **n_folds** (int) – if >= 1, will only train n_folds number of models, otherwise will train one model per fold

- **lr_bounds** (Tuple[float, float]) – starting and ending LR values
- **callback_partials** (Optional[List[partial]]) – optional list of functools.partial, each of which will a instantiate Callback when called
- **plot_settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type List[LRFinder]

Returns List of LRFinder which were used for each model trained

5.4 lumin.optimisation.threshold module

lumin.optimisation.threshold.**binary_class_cut_by_ams**(*df*, *top_perc=5.0*, *min_pred=0.9*, *wgt_factor=1.0*, *br=0.0*, *syst_unc_b=0.0*, *pred_name='pred'*, *targ_name='gen_target'*, *wgt_name='gen_weight'*, *plot_settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Optimise a cut on a signal-background classifier prediction by the Approximate Median Significance Cut which should generalise better by taking the mean class prediction of the top top_perc percentage of points as ranked by AMS

Parameters

- **df** (DataFrame) – Pandas DataFrame containing data
- **top_perc** (float) – top percentage of events to consider as ranked by AMS
- **min_pred** (float) – minimum prediction to consider
- **wgt_factor** (float) – single multiplicative coeficient for rescaling signal and background weights before computing AMS
- **br** (float) – background offset bias
- **syst_unc_b** (float) – fractional systemtatic uncertainty on background
- **pred_name** (str) – column to use as predictions
- **targ_name** (str) – column to use as truth labels for signal and background
- **wgt_name** (str) – column to use as weights for signal and background events
- **plot_settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type Tuple[float, float, float]

Returns Optimised cut AMS at cut Maximum AMS

5.5 Module contents

CHAPTER

SIX

LUMIN.PLOTTING PACKAGE

6.1 Submodules

6.2 lumin.plotting.data_viewing module

`lumin.plotting.data_viewing.`**`plot_feat`**(*df, feat, wgt_name=None, cuts=None, labels=", plot_bulk=True, n_samples=100000, plot_params=None, size='mid', show_moments=True, ax_labels={'x': None, 'y': 'Density'}, savename=None, settings=<lumin.plotting.plot_settings.PlotSettings object>*)

A flexible function to provide indicative information about the 1D distribution of a feature. By default it will produce a weighted KDE+histogram for the [1,99] percentile of the data, as well as compute the mean and standard deviation of the data in this region. Distributions are weighted by sampling with replacement the data with probabilities propotional to the sample weights. By passing a list of cuts and labels, it will plot multiple distributions of the same feature for different cuts. Since it is designed to provide quick, indicative information, more specific functions (such as *plot_kdes_from_bs*) should be used to provide final results.

Parameters

- **df** (`DataFrame`) – Pandas DataFrame containing data
- **feat** (`str`) – column name to plot
- **wgt_name** (`Optional[str]`) – if set, will use column to weight data
- **cuts** (`Optional[List[Series]]`) – optional list of cuts to apply to feature. Will add one KDE+hist for each cut listed on the same plot
- **labels** (`Optional[List[str]]`) – optional list of labels for each KDE+hist
- **plot_bulk** (`bool`) – whether to plot the [1,99] percentile of the data, or all of it
- **n_samples** (`int`) – if plotting weighted distributions, how many samples to use
- **plot_params** (`Union[Dict[str, Any], List[Dict[str, Any]], None]`) – optional list of of arguments to pass to Seaborn Distplot for each KDE+hist
- **size** (`str`) – string to pass to `str2sz()` to determin size of plot
- **show_moments** (`bool`) – whether to compute and display the mean and standard deviation
- **ax_labels** (`Dict[str, Any]`) – dictionary of x and y axes labels

- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type None

lumin.plotting.data_viewing.**compare_events**(*events*)

Plot at least two events side by side in their transverse and longitudinal projections

Parameters **events** (list) – list of DataFrames containing vector coordinates for 3 momenta

Return type None

lumin.plotting.data_viewing.**plot_rank_order_dendrogram**(*df*, *threshold=0.8*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Plot dendrogram of features in df clustered via Spearman's rank correlation coefficient. Also returns a list pairs of features with correlation coefficients greater than the threshold

Parameters

- **df** (DataFrame) – Pandas DataFrame containing data
- **threshold** (float) – Threshold on correlation coefficient
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type List[List[str]]

Returns List of pairs of features with correlation coefficients greater than the threshold

lumin.plotting.data_viewing.**plot_kdes_from_bs**(*x*, *bs_stats*, *name2args*, *feat*, *units=None*, *moments=True*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Plot KDEs computed via *bootstrap_stats()*

Parameters

- **bs_stats** (Dict[str, Any]) – (filtered) dictionary retruned by *bootstrap_stats()*
- **name2args** (Dict[str, Dict[str, Any]]) – Dictionary mapping names of different distributions to arguments to pass to seaborn tsplot
- **feat** (str) – Name of feature being plotted (for axis lablels)
- **units** (Optional[str]) – Optional units to show on axes
- **moments** – whether to display mean and standard deviation of each distribution
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type None

6.3 lumin.plotting.interpretation module

`lumin.plotting.interpretation.`**`plot_importance`**(*df*, *feat_name='Feature'*, *imp_name='Importance'*, *unc_name='Uncertainty'*, *threshold=None*, *x_lbl='Importance via feature permutation'*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Plot feature importances as computted via *get_nn_feat_importance*, *get_ensemble_feat_importance*, or *rf_rank_features*

Parameters

- **df** (`DataFrame`) – DataFrame containing columns of features, importances and, optionally, uncertainties
- **feat_name** (`str`) – column name for features
- **imp_name** (`str`) – column name for importances
- **unc_name** (`str`) – column name for uncertainties (if present)
- **threshold** (`Optional[float]`) – if set, will draw a line at the threshold hold used for feature importance
- **x_lbl** (`str`) – label to put on the x-axis
- **savename** (`Optional[str]`) – Optional name of file to which to save the plot of feature importances
- **settings** (*`PlotSettings`*) – *`PlotSettings`* class to control figure appearance

Return type `None`

`lumin.plotting.interpretation.`**`plot_embedding`**(*embed*, *feat*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Visualise weights in provided categorical entity-embedding matrix

Parameters

- **embed** (`OrderedDict`) – state_dict of trained nn.Embedding
- **feat** (`str`) – name of feature embedded
- **savename** (`Optional[str]`) – Optional name of file to which to save the plot of feature importances
- **settings** (*`PlotSettings`*) – *`PlotSettings`* class to control figure appearance

Return type `None`

`lumin.plotting.interpretation.`**`plot_1d_partial_dependence`**(*model, df, feat, train_feats, ignore_feats=None, input_pipe=None, sample_sz=None, wgt_name=None, n_clusters=10, n_points=20, pdp_isolate_kargs=None, pdp_plot_kargs=None, y_lim=None, savename=None, settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Wrapper for PDPbox to plot 1D dependence of specified feature using provided NN or RF. If features have been preprocessed using an SK-Learn Pipeline, then that can be passed in order to rescale the x-axis back to its original values.

Parameters

- **model** (`Any`) – any trained model with a .predict method
- **df** (`DataFrame`) – DataFrame containing training data
- **feat** (`str`) – feature for which to evaluate the partial dependence of the model
- **train_feats** (`List[str]`) – list of all training features including ones which were later ignored, i.e. input features considered when input_pipe was fitted
- **ignore_feats** (`Optional[List[str]]`) – features present in training data which were not used to train the model (necessary to correctly deprocess feature using input_pipe)
- **input_pipe** (`Optional[Pipeline]`) – SK-Learn Pipeline which was used to process the training data
- **sample_sz** (`Optional[int]`) – if set, will only compute partial dependence on a random sample with replacement of the training data, sampled according to weights (if set). Speeds up computation and allows weighted partial dependencies to computed.
- **wgt_name** (`Optional[str]`) – Optional column name to use as sampling weights
- **n_points** (`int`) – number of points at which to evaluate the model output, passed to pdp_isolate as num_grid_points
- **n_clusters** (`Optional[int]`) – number of clusters in which to group dependency lines. Set to None to show all lines
- **pdp_isolate_kargs** (`Optional[Dict[str, Any]]`) – optional dictionary of keyword arguments to pass to pdp_isolate
- **pdp_plot_kargs** (`Optional[Dict[str, Any]]`) – optional dictionary of keyword arguments to pass to pdp_plot
- **y_lim** (`Union[Tuple[float, float], List[float], None]`) – If set, will limit y-axis plot range to tuple
- **savename** (`Optional[str]`) – Optional name of file to which to save the plot of feature importances
- **settings** (*`PlotSettings`*) – *`PlotSettings`* class to control figure appearance

Return type `None`

`lumin.plotting.interpretation.`**`plot_2d_partial_dependence`**(*model, df, feats, train_feats, ignore_feats=None, input_pipe=None, sample_sz=None, wgt_name=None, n_points=[20, 20], pdp_interact_kargs=None, pdp_interact_plot_kargs=None, savename=None, settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Wrapper for PDPbox to plot 2D dependence of specified pair of features using provided NN or RF. If features have been preprocessed using an SK-Learn Pipeline, then that can be passed in order to rescale them back to their original values.

Parameters

- **model** (`Any`) – any trained model with a .predict method
- **df** (`DataFrame`) – DataFrame containing training data
- **feats** (`Tuple[str, str]`) – pair of features for which to evaluate the partial dependence of the model
- **train_feats** (`List[str]`) – list of all training features including ones which were later ignored, i.e. input features considered when input_pipe was fitted
- **ignore_feats** (`Optional[List[str]]`) – features present in training data which were not used to train the model (necessary to correctly deprocess feature using input_pipe)
- **input_pipe** (`Optional[Pipeline]`) – SK-Learn Pipeline which was used to process the training data
- **sample_sz** (`Optional[int]`) – if set, will only compute partial dependence on a random sample with replacement of the training data, sampled according to weights (if set). Speeds up computation and allows weighted partial dependencies to computed.
- **wgt_name** (`Optional[str]`) – Optional column name to use as sampling weights
- **n_points** (`Tuple[int, int]`) – pair of numbers of points at which to evaluate the model output, passed to pdp_interact as num_grid_points
- **n_clusters** – number of clusters in which to group dependency lines. Set to None to show all lines
- **pdp_isolate_kargs** – optional dictionary of keyword arguments to pass to pdp_isolate
- **pdp_plot_kargs** – optional dictionary of keyword arguments to pass to pdp_plot
- **savename** (`Optional[str]`) – Optional name of file to which to save the plot of feature importances
- **settings** (*`PlotSettings`*) – *`PlotSettings`* class to control figure appearance

Return type `None`

`lumin.plotting.interpretation.`**`plot_multibody_weighted_outputs`**(*model, inputs, block_names=None, use_mean=False, savename=None, settings=<lumin.plotting.plot_settings.PlotSe object>*)

Interpret how a model relies on the outputs of each block in a :class:MultiBlock by plotting the outputs of each block as weighted by the tail block. This function currently only supports models whose tail block contains a single neuron in the first dense layer. Input data is passed through the model and the absolute sums of the weighted block outputs are computed per datum, and optionally averaged over the number of block outputs.

Parameters

- **model** (`AbsModel`) – model to interpret
- **inputs** (`Union[ndarray, Tensor]`) – input data to use for interpretation
- **block_names** (`Optional[List[str]]`) – names for each block to use when plotting
- **use_mean** (`bool`) – if True, will average the weighted outputs over the number of output neurons in each block
- **savename** (`Optional[str]`) – Optional name of file to which to save the plot of feature importances
- **settings** (*`PlotSettings`*) – *`PlotSettings`* class to control figure appearance

Return type `None`

`lumin.plotting.interpretation.`**`plot_bottleneck_weighted_inputs`**(*model, bottleneck_idx, inputs, log_y=True, savename=None, settings=<lumin.plotting.plot_settings.PlotSe object>*)

Interpret how a single-neuron bottleneck in a :class:MultiBlock relies on input features by plotting the absolute values of the features times their associated weight for a given set of input data.

Parameters

- **model** (`AbsModel`) – model to interpret
- **bottleneck_idx** (`int`) – index of the bottleneck to interpret, i.e. model.body.bottleneck_blocks[bottleneck_idx]
- **inputs** (`Union[ndarray, Tensor]`) – input data to use for interpretation
- **log_y** (`bool`) – whether to plot a log scale for the y-axis
- **savename** (`Optional[str]`) – Optional name of file to which to save the plot of feature importances
- **settings** (*`PlotSettings`*) – *`PlotSettings`* class to control figure appearance

Return type `None`

6.4 lumin.plotting.plot_settings module

class lumin.plotting.plot_settings.**PlotSettings**(***kargs*)

Bases: object

Class to provide control over plot appearances. Default parameters are set automatically, and can be adjusted by passing values as keyword arguments during initialisation (or changed after instantiation)

Parameters **arguments** (*keyword*) – used to set relevant plotting parameters

str2sz(*sz*, *ax*)

Used to map requested plot sizes to actual dimensions

Parameters

- **sz** (str) – string representation of size
- **ax** (str) – axis dimension requested

Return type float

Returns width of plot dimension

6.5 lumin.plotting.results module

lumin.plotting.results.**plot_roc**(*data*, *pred_name='pred'*, *targ_name='gen_target'*, *wgt_name=None*, *labels=None*, *plot_params=None*, *n_bootstrap=0*, *log_x=False*, *plot_baseline=True*, *savename=None*, *settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Plot receiver operating characteristic curve(s), optionally using booostrap resampling

Parameters

- **data** (Union[DataFrame, List[DataFrame]]) – (list of) DataFrame(s) from which to draw predictions and targets
- **pred_name** (str) – name of column to use as predictions
- **targ_name** (str) – name of column to use as targets
- **wgt_name** (Optional[str]) – optional name of column to use as sample weights
- **labels** (Union[str, List[str], None]) – (list of) label(s) for plot legend
- **plot_params** (Union[Dict[str, Any], List[Dict[str, Any]], None]) – (list of) dictionar[y/ies] of argument(s) to pass to line plot
- **n_bootstrap** (int) – if greater than 0, will bootstrap resample the data that many times when computing the ROC AUC. Currently, this does not affect the shape of the lines, which are based on computing the ROC for the entire dataset as is.
- **log_x** (bool) – whether to use a log scale for plotting the x-axis, useful for high AUC line
- **plot_baseline** (bool) – whether to plot a dotted line for AUC=0.5. Currently incompatable with log_x=True
- **savename** (Optional[str]) – Optional name of file to which to save the plot of feature importances
- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type `Dict[str, Union[float, Tuple[float, float]]]`

Returns Dictionary mapping data labels to aucs (and uncertainties if n_bootstrap > 0)

`lumin.plotting.results.`**`plot_binary_class_pred`**(*df, pred_name='pred', targ_name='gen_target', wgt_name=None, wgt_scale=1, log_y=False, lim_x=(0, 1), density=True, savename=None, settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Basic plotter for prediction distribution in a binary classification problem. Note that labels are set using the settings.targ2class dictionary, which by default is {0: 'Background', 1: 'Signal'}.

Parameters

- **df** (`DataFrame`) – DataFrame with targets and predictions
- **pred_name** (`str`) – name of column to use as predictions
- **targ_name** (`str`) – name of column to use as targets
- **wgt_name** (`Optional[str]`) – optional name of column to use as sample weights
- **wgt_scale** (`float`) – applies a global multiplicative rescaling to sample weights. Default 1 = no rescaling
- **log_y** (`bool`) – whether to use a log scale for the y-axis
- **lim_x** (`Tuple[float, float]`) – limit for plotting on the x-axis
- **density** – whether to normalise each distribution to one, or keep set to sum of weights / datapoints
- **savename** (`Optional[str]`) – Optional name of file to which to save the plot of feature importances
- **settings** (*`PlotSettings`*) – *`PlotSettings`* class to control figure appearance

Return type `None`

`lumin.plotting.results.`**`plot_sample_pred`**(*df, pred_name='pred', targ_name='gen_target', wgt_name='gen_weight', sample_name='gen_sample', wgt_scale=1, bins=35, log_y=True, lim_x=(0, 1), density=False, zoom_args=None, savename=None, settings=<lumin.plotting.plot_settings.PlotSettings object>*)

More advanceed plotter for prediction distirbution in a binary class problem with stacked distributions for backgrounds and user-defined binning Can also zoom in to specified parts of plot Note that plotting colours can be controled by seeting the settings.sample2col dictionary

Parameters

- **df** (`DataFrame`) – DataFrame with targets and predictions
- **pred_name** (`str`) – name of column to use as predictions
- **targ_name** (`str`) – name of column to use as targets
- **wgt_name** (`str`) – name of column to use as sample weights
- **sample_name** (`str`) – name of column to use as process names

- **wgt_scale** (`float`) – applies a global multiplicative rescaling to sample weights. Default 1 = no rescaling
- **bins** (`Union[int, List[int]]`) – either the number of bins to use for a uniform binning, or a list of bin edges for a variable-width binning
- **log_y** (`bool`) – whether to use a log scale for the y-axis
- **lim_x** (`Tuple[float, float]`) – limit for plotting on the x-axis
- **density** – whether to normalise each distribution to one, or keep set to sum of weights / datapoints
- **zoom_args** (`Optional[Dict[str, Any]]`) – arguments to control the optional zoomed in section, e.g. {'x':(0.4,0.45), 'y':(0.2, 1500), 'anchor':(0,0.25,0.95,1), 'width_scale':1, 'width_zoom':4, 'height_zoom':3}
- **savename** (`Optional[str]`) – Optional name of file to which to save the plot of feature importances
- **settings** (*`PlotSettings`*) – *`PlotSettings`* class to control figure appearance

Return type `None`

6.6 lumin.plotting.training module

`lumin.plotting.training.`**`plot_train_history`**(*histories, savename=None, ignore_trn=True, settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Plot histories object returned by `fold_train_ensemble()` showing the loss evolution over time per model trained.

Parameters

- **histories** (`List[Dict[str, List[float]]]`) – list of dictionaries mapping loss type to values at each (sub)-epoch
- **savename** (`Optional[str]`) – Optional name of file to which to save the plot of feature importances
- **ignore_trn** – whether to ignore training loss
- **settings** (*`PlotSettings`*) – *`PlotSettings`* class to control figure appearance

Return type `None`

`lumin.plotting.training.`**`plot_lr_finders`**(*lr_finders, lr_range=None, loss_range='auto', settings=<lumin.plotting.plot_settings.PlotSettings object>*)

Plot mean loss evolution against learning rate for several `fold_lr_find`.

Parameters

- **lr_finders** (`List[LRFinder]`) – list of `fold_lr_find`)
- **lr_range** (`Union[float, Tuple, None]`) – limits the range of learning rates plotted on the x-axis: if float, maximum LR; if tuple, minimum & maximum LR
- **loss_range** (`Union[float, Tuple, str, None]`) – limits the range of losses plotted on the x-axis: if float, maximum loss; if tuple, minimum & maximum loss; if None, no limits; if 'auto', computes an upper limit automatically

- **settings** (*PlotSettings*) – *PlotSettings* class to control figure appearance

Return type None

6.7 Module contents

CHAPTER

SEVEN

LUMIN.UTILS PACKAGE

7.1 Submodules

7.2 lumin.utils.data module

`lumin.utils.data.`**`check_val_set`**(*train*, *val*, *test=None*, *n_folds=None*)

Method to check validation set suitability by seeing whether Random Forests can predict whether events belong to one dataset or another. If a `FoldYielder` is passed, then trainings are run once per fold and averaged. Will compute the ROC AUC for set discrimination (should be close to 0.5) and compute the feature importances to aid removal of discriminating features.

Parameters

- **`train`** (`Union[DataFrame, ndarray, FoldYielder]`) – training data
- **`val`** (`Union[DataFrame, ndarray, FoldYielder]`) – validation data
- **`test`** (`Union[DataFrame, ndarray, FoldYielder, None]`) – optional testing data
- **`n_folds`** (`Optional[int]`) – if set and if passed a `FoldYielder`, will only use the first n_folds folds

Return type `None`

7.3 lumin.utils.misc module

`lumin.utils.misc.`**`to_np`**(*x*)

Convert Tensor x to a Numpy array

Parameters **`x`** (`Tensor`) – Tensor to convert

Return type `ndarray`

Returns x as a Numpy array

`lumin.utils.misc.`**`to_device`**(*x*, *device=device(type='cpu')*)

Recursively place Tensor(s) onto device

Parameters **`x`** (`Union[Tensor, List[Tensor]]`) – Tensor(s) to place on device

Return type `Union[Tensor, List[Tensor]]`

Returns Tensor(s) on device

`lumin.utils.misc.`**`to_tensor`**(*x*)

Convert Numpy array to Tensor with possibility of a None being passed

Parameters **x** (Optional[ndarray]) – Numpy array or None

Return type Optional[Tensor]

Returns x as Tensor or None

lumin.utils.misc.**str2bool**(*string*)

Convert string representation of Boolean to bool

Parameters **string** (Union[str, bool]) – string representation of Boolean (or a Boolean)

Return type bool

Returns bool if bool was passed else, True if lowercase string matches is in ("yes", "true", "t", "1")

lumin.utils.misc.**to_binary_class**(*df*, *zero_preds*, *one_preds*)

Map class precitions back to a binary prediction. The maximum prediction for features listed in zero_preds is treated as the prediction for class 0, vice versa for one_preds. The binary prediction is added to df in place as column 'pred'

Parameters

- **df** (DataFrame) – DataFrame containing prediction features
- **zero_preds** (List[str]) – list of column names for predictions associated with class 0
- **one_preds** (List[str]) – list of column names for predictions associated with class 0

Return type None

lumin.utils.misc.**ids2unique**(*ids*)

Map a permutaion of integers to a unique number, or a 2D array of integers to unique numbers by row. Returned numbers are unique for a given permutation of integers. This is achieved by computing the product of primes raised to powers equal to the integers. Beacause of this, it can be easy to produce numbers which are too large to be stored if many (large) integers are passed.

Parameters **ids** (Union[List[int], ndarray]) – (array of) permutation(s) of integers to map

Return type ndarray

Returns (Array of) unique id(s) for given permutation(s)

class lumin.utils.misc.**FowardHook**(*module*, *hook_fn=None*)

Bases: object

Create a hook for performing an action based on the forward pass thorugh a nn.Module

Parameters

- **module** – nn.Module to hook
- **hook_fn** – Optional function to perform. Default is to record input and output of module

Examples::

```
>>> hook = ForwardHook(model.tail.dense)
>>> model.predict(inputs)
>>> print(hook.inputs)
```

hook_fn(*module*, *input*, *output*)

Default hook function records inputs and outputs of module

Parameters

- **module** (Module) – nn.Module to hook

- **input** (Union[Tensor, Tuple[Tensor]]) – input tensor
- **output** (Union[Tensor, Tuple[Tensor]]) – output tensor of module

Return type None

remove()

Call when finished to remove hook

Return type None

lumin.utils.misc.**subsample_df**(*df*, *objective*, *targ_name*, *n_samples=None*, *replace=False*, *strat_key=None*, *wgt_name=None*)

Subsamples, or samples with replacement, a DataFrame. Will automatically reweight data such that weight sums remain the same as the original DataFrame (per class)

Parameters

- **df** (DataFrame) – DataFrame to sample
- **objective** (str) – string representation of objective: either 'classification' or 'regression'
- **targ_name** (str) – name of column containing target data
- **n_samples** (Optional[int]) – If set, will sample that number of data points, otherwise will sample with replacement a new DataFRame of the same size as the original
- **replace** (bool) – whether to sample with replacement
- **strat_key** (Optional[str]) – column name to use for stratified subsampling, if desired
- **wgt_name** (Optional[str]) – name of column containing weight data. If set, will reweight subsampled data, otherwise will not

Return type DataFrame

7.4 lumin.utils.multiprocessing module

lumin.utils.multiprocessing.**mp_run**(*args*, *func*)

Run multiple instances of function simultaneously by using a list of argument dictionaries Runs given function once per entry in args list.

Important: Function should put a dictionary of results into the *mp.Queue* and each result key should be unique otherwise they will overwrite one another.

Parameters

- **args** (List[Dict[Any, Any]]) – list of dictionaries of arguments
- **func** (Callable[[Any], Any]) – function to which to pass dictionary arguments

Return type Dict[Any, Any]

Returns DIctionary of results

7.5 lumin.utils.statistics module

`lumin.utils.statistics.`**`bootstrap_stats`**(*args*, *out_q=None*)

Computes statistics and KDEs of data via sampling with replacement

Parameters

- **`args`** (`Dict[str, Any]`) – dictionary of arguments. Possible keys are: data - data to resample name - name prepended to returned keys in result dict weights - array of weights matching length of data to use for weighted resampling n - number of times to resample data x - points at which to compute the kde values of resample data kde - whether to compute the kde values at x-points for resampled data mean - whether to compute the means of the resampled data std - whether to compute standard deviation of resampled data c68 - whether to compute the width of the absolute central 68.2 percentile of the resampled data
- **`out_q`** (`Optional[`<bound method BaseContext.Queue of <multiprocessing.context.DefaultContext object at 0x7fa26bd036a0>>`]`) – if using multiporcessing can place result dictionary in provided queue

Return type `Union[None, Dict[str, Any]]`

Returns Result dictionary if *out_q* is *None* else *None*.

`lumin.utils.statistics.`**`get_moments`**(*arr*)

Computes mean and std of data, and their associated uncertainties

Parameters **`arr`** (`ndarray`) – univariate data

Return type `Tuple[float, float, float, float]`

Returns

- mean
- statistical uncertainty of mean
- standard deviation
- statistical uncertainty of standard deviation

`lumin.utils.statistics.`**`uncert_round`**(*value*, *uncert*)

Round value according to given uncertainty using one significant figure of the uncertainty

Parameters

- **`value`** (`float`) – value to round
- **`uncert`** (`float`) – uncertainty of value

Return type `Tuple[float, float]`

Returns

- rounded value
- rounded uncertainty

7.6 Module contents

CHAPTER

EIGHT

PACKAGE DESCRIPTION

8.1 Distinguishing Characteristics

8.1.1 Data objects

- Use with large datasets: HEP data can become quite large, making training difficult:
  - The `FoldYielder` class provides on-demand access to data stored in HDF5 format, only loading into memory what is required.
  - Conversion from ROOT and CSV to HDF5 is easy to achieve using (see examples)
  - `FoldYielder` provides conversion methods to Pandas `DataFrame` for use with other internal methods and external packages
- Non-network-specific methods expect Pandas `DataFrame` allowing their use without having to convert to `FoldYielder`.

8.1.2 Deep learning

- PyTorch > 1.0
- Inclusion of recent deep learning techniques and practices, including:
  - Dynamic learning rate, momentum, beta_1:
    - Cyclical, Smith, 2015
    - Cosine annealed Loschilov & Hutter, 2016
    - 1-cycle, Smith, 2018
  - HEP-specific data augmentation during training and inference
  - Advanced ensembling methods:
    - Snapshot ensembles Huang et al., 2017
    - Fast geometric ensembles Garipov et al., 2018
    - Stochastic Weight Averaging Izmailov et al., 2018
  - Learning Rate Finders, Smith, 2015
  - Entity embedding of categorical features, Guo & Berkhahn, 2016
  - Label smoothing Szegedy et al., 2015
- Flexible architecture construction:

  - `ModelBuilder` takes parameters and modules to yield networks on-demand
  - Networks constructed from modular blocks:
    * Head - Takes input features
    * Body - Contains most of the hidden layers
    * Tail - Scales down the body to the desired number of outputs
    * Endcap - Optional layer for use post-training to provide further computation on model outputs; useful when training on a proxy objective
  - Easy loading and saving of pre-trained embedding weights
  - Modern architectures like:
    * Residual and dense(-like) networks (He et al. 2015 & Huang et al. 2016)
    * Graph nets for physics objects, e.g. Battaglia, Pascanu, Lai, Rezende, Kavukcuoglu, 2016 & Moreno et al., 2019
    * Recurrent layers for series of objects
    * 1D convolutional networks for series of objects
- Configurable initialisations, including LSUV Mishkin, Matas, 2016
- HEP-specific losses, e.g. Asimov loss Elwood & Krücker, 2018
- Easy training and inference of ensembles of models:
  - Default training method `fold_train_ensemble`, trains a specified number of models as well as just a single model
  - `Ensemble` class handles the (metric-weighted) construction of an ensemble, its inference, saving and loading, and interpretation
- Easy exporting of models to other libraries via Onnx
- Use with CPU and NVidia GPU
- Evaluation on domain-specific metrics such as Approximate Median Significance via `EvalMetric` class
- Keras-style callbacks

8.1.3 Feature selection methods

- Dendrograms of feature-pair monotonacity
- Feature importance via auto-optimised SK-Learn random forests
- Mutual dependance (via RFPImp)
- Automatic filtering and selection of features

8.1.4 Interpretation

- Feature importance for models and ensembles
- Embedding visualisation
- 1D & 2D partial dependency plots (via PDPbox)

8.1.5 Plotting

- Variety of domain-specific plotting functions
- Unified appearance via `PlotSettings` class - class accepted by every plot function providing control of plot appearance, titles, colour schemes, et cetera

8.1.6 Universal handling of sample weights

- HEP events are normally accompanied by weight characterising the acceptance and production cross-section of that particular event, or to flatten some distribution.
- Relevant methods and classes can take account of these weights.
- This includes training, interpretation, and plotting
- Expansion of PyTorch losses to better handle weights

8.1.7 Parameter optimisation

- Optimal learning rate via cross-validated range tests Smith, 2015
- Quick, rough optimisation of random forest hyper parameters
- Generalisable Cut & Count thresholds
- 1D discriminant binning with respect to bin-fill uncertainty

8.1.8 Statistics and uncertainties

- Integral to experimental science
- Quantitative results are accompanied by uncertainties
- Use of bootstrapping to improve precision of statistics estimated from small samples

8.1.9 Look and feel

- LUMIN aims to feel fast to use - liberal use of progress bars mean you're able to always know when tasks will finish, and get live updates of training
- Guaranteed to spark joy (in its current beta state, LUMIN may instead ignite rage, despair, and frustration - *dev.*)

8.2 Examples

Several examples are present in the form of Jupyter Notebooks in the `examples` folder. These can be run also on Google Colab to allow you to quickly try out the package here: https://github.com/GilesStrong/lumin#examples

8.3 Installation

Due to some strict version requirements on packages, it is recommended to install LUMIN in its own Python environment, e.g `conda create -n lumin python=3.6`

8.3.1 From PyPI

The main package can be installed via: `pip install lumin`

Full functionality requires two additional packages as described below.

8.3.2 From source

```
git clone git@github.com:GilesStrong/lumin.git
cd lumin
pip install .
```

Optionally, run pip install with `-e` flag for development installation. Full functionality requires an additional package as described below.

8.3.3 Additional modules

Full use of LUMIN requires the latest version of PDPbox, but this is not released yet on PyPI, so you'll need to install it from source, too:

- `git clone https://github.com/SauceCat/PDPbox.git && cd PDPbox && pip install -e .` note the `-e` flag to make sure the version number gets set properly.

8.4 Notes

8.4.1 Why use LUMIN

TMVA contained in CERN's ROOT system, has been the default choice for BDT training for analysis and reconstruction algorithms due to never having to leave ROOT format. With the gradual move to DNN approaches, more scientists are looking to move their data out of ROOT to use the wider selection of tools which are available. Keras appears to be the first stop due to its ease of use, however implementing recent methods in Keras can be difficult, and sometimes requires dropping back to the tensor library that it aims to abstract. Indeed, the prequel to LUMIN was a similar wrapper for Keras (HEPML_Tools) which involved some pretty ugly hacks. The fastai framework provides access to these recent methods, however doesn't yet support sample weights to the extent that HEP requires. LUMIN aims to provides the best of both, Keras-style sample weighting and fastai training methods, while focussing on columnar data and providing domain-specific metrics, plotting, and statistical treatment of results and uncertainties.

8.4.2 Data types

LUMIN is primarily designed for use on columnar data, and from version 0.5 onwards this also includes *matrix data*; ordered series and un-ordered groups of objects. With some extra work it can be used on other data formats, but at the moment it has nothing special to offer. Whilst recent work in HEP has made use of jet images and GANs, these normally hijack existing ideas and models. Perhaps once we get established, domain specific approaches which necessitate the use of a specialised framework, then LUMIN could grow to meet those demands, but for now I'd recommend checking out the fastai library, especially for image data.

With just one main developer, I'm simply focussing on the data types and applications I need for my own research and common use cases in HEP. If, however you would like to use LUMIN's other methods for your own work on other data formats, then you are most welcome to contribute and help to grow LUMIN to better meet the needs of the scientific community.

8.4.3 Future

The current priority is to imporve the documentation, add unit tests, and expand the examples.

The next step will be to try and increase the user base and number of contributors. I'm aiming to achieve this through presentations, tutorials, blog posts, and papers.

Further improvments will be in the direction of implementing new methods and (HEP-specific) architectures, as well as providing helper functions and data exporters to statistical analysis packages like Combine and PYHF.

8.4.4 Contributing & feedback

Contributions, suggestions, and feedback are most welcome! The issue tracker on this repo is probably the best place to report bugs et cetera.

8.4.5 Code style

Nope, the majority of the codebase does not conform to PEP8. PEP8 has its uses, but my understanding is that it primarily written for developers and maintainers of software whose users never need to read the source code. As a maths-heavy research framework which users are expected to interact with, PEP8 isn't the best style. Instead I'm aiming to follow more the style of fastai, which emphasises, in particular, reducing vertical space (useful for reading source code in a notebook) naming and abbreviating variables according to their importance and lifetime (easier to recognise which variables have a larger scope and permits easier writing of mathematical operations). A full list of the abbreviations used may be found in abbr.md

8.4.6 Why is LUMIN called LUMIN?

Aside from being a recursive accronym (and therefore the best kind of accronym) lumin is short for 'luminosity'. In high-energy physics, the integrated luminosity of the data collected by an experiment is the main driver in the results that analyses obtain. With the paradigm shift towards multivariate analyses, however, improved methods can be seen as providing 'artificial luminosity'; e.g. the gain offered by some DNN could be measured in terms of the amount of extra data that would have to be collected to achieve the same result with a more traditional analysis. Luminosity can also be connected to the fact that LUMIN is built around the python version of Torch.

8.4.7 Who develops LUMIN

Currently just me - Giles Strong; a British-born, Lisbon-based, PhD student in particle physics at IST, researcher at LIP-Lisbon, member of Marie Curie ITN AMVA4NewPhysics and the CMS collaboration.

Certainly more developers and contributors are welcome to join and help out!

8.4.8 Reference

If you have used LUMIN in your analysis work and wish to cite it, the preferred reference is: *Giles C. Strong, LUMIN, Zenodo (Mar. 2019), https://doi.org/10.5281/zenodo.2601857, Note: Please check https://github.com/GilesStrong/lumin/graphs/contributors for the full list of contributors*

@misc{giles_chatham_strong_2019_2601857, author = {Giles Chatham Strong}, title = {LUMIN}, month = mar, year = 2019, note = {{Please check https://github.com/GilesStrong/lumin/graphs/contributors for the full list of contributors}}, doi = {10.5281/zenodo.2601857}, url = {https://doi.org/10.5281/zenodo.2601857} }

CHAPTER

NINE

INDEX

- genindex

PYTHON MODULE INDEX

l

INDEX

U