Accelerated exploration of multinary systems

Release 1.1

Elise Garel, Jean-Luc Parouty

Feb 15, 2022

CONTENTS:

CHAPTER

ONE

EXPERIMENTSPLANNIFICATION

`modules.check_do_not_align`(*name_alignments*, *index*, *do_not_align*)
Check user condition to not align certain mixtures in the same gradient

Parameters

- **name_alignments** (`array(str)`) – name of points through which the gradients passes
- **index** (`int`) – index of alignments in the list of all alignments
- **do_not_align** (`cell(list(str))`) – list of mixtures that must not be aligned

Returns indicator: "ok" if alignement respects the user condition; else return "not ok".

Return type str

`modules.check_not_repeat`(*name_alignments*, *index*, *name_alignement_opt*, *not_repeat*)
Check user condition to not repeat certain mixtures in gradients set

Parameters

- **name_alignments** (`array(str)`) – name of points through which the gradients passes
- **index** (`int`) – index of alignments in the list of all alignments
- **name_alignement_opt** (`array(str)`) – gradients set that are already selected.
- **not_repeat** (`array(str)`) – mixtures to no repeat in the gradients set

Returns indicator: "ok" if alignement respects the user condition; else return "not ok".

Return type str

`modules.check_repeat_only`(*name_alignments*, *index*, *name_alignement_opt*, *repeat_only*)
Check user condition to not repeat certain mixtures in gradients set

Parameters

- **name_alignments** (`array(str)`) – name of points through which the gradients passes
- **index** (`int`) – index of alignments in the list of all alignments
- **name_alignement_opt** (`array(str)`) – gradients set that are allready selected.
- **repeat_only** (`array(str,int)`) – name of mixtures that must be repeated a limited number of time and this limited number of time

Returns indicator: "ok" if alignement respects the user condition; else return "not ok".

Return type str

modules.**compute_alignments**(*mixture*, *name_mixture*, *nb_type_mixture*)

For a reference mixture, the function calculates the vector coefficient between this reference mixture and all the other mixtures with same or higher order. Then it looks for equals vector coefficients for segments with a common point to determine which two other mixture points are aligned with the reference mixture

Parameters

- **mixture** (cell{array}) – coordinates of mixtures, cell index being the mixture order *eg:mixtures{2} contains the binaries coordinates*
- **name_mixture** (cell{str}) – name of mixture, cell index being the mixture order
- **nb_type_mixture** (int) – number of type/order of mixtures to explore

Returns

- alignments : coordinates of the mixtures through which the gradient pass (3x3 columns)
- name_alignments: mixture names through which the gradient pass

Return type array(float),array(str)

modules.**compute_planes**(*name_alignment*, *alignments*, *nb_type_mixture*)

From the gradients, planes are defines in the composition space made by 3 gradients with common points, encompassing 7 points of the mixtrure design. This means that the plane is centered on one of the point of the mixture design

eg: Nb-NbTi-Ti, Ti-TiZr-Zr and Nb-NbZr-Zr are forming a plane in a compositional space center on the ternary NbTiZr wich is a point of the mixture design: the plane is valid

Parameters

- **name_alignment** (array(str)) – points through which the gradient go
- **alignments** (array(float)) – coordinates of the points through which the gradient go (3x3 columns)
- **nb_type_mixture** (int) – number of type/order of mixtures to explore

Returns plane_points: mixture names encompassed by the planes

Returns plane_coord: coordinates of the mixtures encompassed by the planes (7x3 columns)

modules.**coordinates_name_centroid_points**(*nb_elements*, *name_elements*)

From the number and the name of the system elements, the function calculates the coordinates of the pure elements (standard uniform distribution in space) and of the equiolar mixtures of the Simplex Centroide mixture Design (all binaries, ternairies…).

Parameters **nb_elements** (int) – number of components

Name_element list(str) namer of components

Returns mixture: cell coordinates of all equimolar mixture

Return type cell

Returns name_mixture: containing the names of the equimolar mixtures.

Return type cell

modules.**count_occur**(*element*, *list*)

Count the numer of occurence of an element in a list

Parameters

- **element** – counted number or string

- **list** (list) – list in which the element is counted

Returns count: number of repetition of the element in list

Return type int

modules.**fix_nb_repetition**(*repeat_list*, *fig*, *position*)

This function is a callbacks of push buttons associated to listboxes When the buttons are pushed, the function identifies which mixture should be repeated Then it display in the interface the names of the mixtures that should be repeated and an edit box in which the user can enter the number of repetitions.

Parameters

- **repeat_list** (UIcontrol) – contains the mixture that should be repeated
- **fig** (figure) – working interface / window
- **position** (list(float)) – position features of repeated list

Returns nb_repet: edit boxes in which the user will enter the number of repetition of each mixture

modules.**get_elements**(*elements*, *fig1*)

Acquire the components name entered by the user

Parameters

- **elements** (UIcontrol) – edit boxes in which the user has entered the elements names
- **fig1** (figure) – interface window

Returns name_elements: name of elements

Return type list(str)

modules.**gradients_set**(*name_mixture*, *mixture*, *alignments*, *name_alignement*)

Selection of a gradients set that pass at least once through each point of the mixture design and that respect user condition inputs

Parameters

- **name_mixture** (cell(str)) – name of mixture, cell index being the mixture order
- **mixture** (cell(float)) – coordinates of mixtures, cell index being the mixture order *eg:mixtures{2} contains the binaries coordinates*

:param cell(float) alignments:coordinates of the points through which the gradient pass (3x3 columns) :param cell(str) name_alignement: points through which the gradient pass :return array(str) name_alignement_opt: name of mixture trhough whch the set of gradients pass :return array(float) alignement_opt: coordinates of mixture trhough whch the set of gradients pass

modules.**index_alignments**(*cell_coeff_dir*)

Called in `compute_alignments`_: we get cell structure with vector coefficient between one reference mixture and all the m
This function compares all the coefficients one by one to find equal ones

Parameters **cell_coeff_dir** (cell) – contains director coefficient of vectors between one reference mixtures and all the others with same or higher order.

Returns cell indice_cell, indice_list: indices of the cell and list where two coefficients are equals. Allow to identify pair of equal coefficient to identify aligned points.

modules.**kill_program**()

Kill the programis the user pushed STOP button

Returns display the message “kill” to indicate state

modules.**lineIntersect3D**(*PA*, *PB*)

Find intersection point of lines in 3D space, in the least squares sense.

Parameters

- **PA** – Nx3-matrix containing starting point of N lines
- **PB** – Nx3-matrix containing end point of N lines

Returns P_Intersect: Best intersection point of the N lines, in least squares sense.

Returns distances: Distances from intersection point to the input lines

Anders Eikenes (2022). Intersection point of lines in 3D space (https://www.mathworks.com/matlabcentral/fileexchange/37192-intersection-point-of-lines-in-3d-space), MATLAB Central File Exchange. Retrieved February 10, 2022.

modules.**nb_repetitions**()

modules.**listing_targets**(*name_alignement_opt*)

Lists the targets to use from the selected optimized set of gradients

Parameters **name_alignement_opt** (array(str)) – name of mixtures through which pass the gradients

Returns list(str) list_target: list the target compositions to use to deposit these gradients

modules.**listing_targets_3cath**(*name_planes_opt*)

Lists the targets to use from the selected optimized set of planar gradients

Parameters **name_planes_opt** (array(str)) – name of mixtures encompassed by planar gradients

Returns list(str) list_target: list the target compositions to use to deposit these gradients

modules.**plot_compo_space_gradients**(*nb_elements*, *mixture*, *name_mixture*, *name_elements*, *gradients*, *gradients_color*)

Plot the composition space with all the simplexe centroid points and linear gradients

Parameters

- **nb_elements** (int) – number of components
- **mixture** (cell(float)) – mixture points coordinates
- **name_mixture** (cell(str)) – mixture names

:param list(str) name_elements:name of the components :param array(str) gradients : coordinates of the gradients points :param str/list(float) gradients_color: color of the gradients for plot :return: fig: plot the compositions space dans gradients

modules.**plot_compo_space_planes**(*nb_elements*, *mixture*, *name_mixture*, *name_elements*, *plane_coord*, *plane_color*, *fignumber)%position*)

Plot the composition space with all the simplexe centroid points and planar gradients

Parameters

- **nb_elements** (int) – number of components

- **mixture** (`cell(float)`) – mixture points coordinates
- **name_mixture** (`cell(str)`) – mixture names

:param list(str) name_elements:name of the components :param array(str) plane_coord : coordinates of the planes points :param str/list(float) plane_color: color of the plane for plot :return: fig: plot the compositions space dans gradients

`modules.`**`parameters_file`**`()`

Write the users inputs and chosen parameters for one run of the interface in text file.

`modules.`**`planes_set`**`(`*name_mixture*, *mixture*, *planes*, *name_planes*`)`

Selection of a planes set that encompass at least once each point of the mixture design and that respect user condition inputs

Parameters

- **name_mixture** (`cell(str)`) – name of mixture, cell index being the mixture order
- **mixture** (`cell(float)`) – coordinates of mixtures, cell index being the mixture order *eg:mixtures{2} contains the binaries coordinates*

:param cell(float) plane:coordinates of the points through which the planes pass (3x3 columns) :param cell(str) name_planes: points through which the planes pass :return: array(str) name_planes_opt: name of mixture trhough whch the set of planes pass :return: array(float) planes_opt: coordinates of mixture trhough whch the set of planes pass

`modules.`**`price_calculation`**`(`*prices_list*, *target_list*`)`

Calculate the price of a set of experiment

Parameters

- **prices_list** (`list(str,float)`) – list of possible targets and associated price
- **targets** (`list(str)`) – list of targets associated to one set of linear gradients or planar gradients

Returns price: total price of the targets required for a set of linear gradients or planar gradients

Return type float

`modules.`**`vector_coeff`**`(`*A*, *B*`)`

Compute normed vector coefficients between two points.

Parameters **A,B** (`list(float)`) – coordinates of two points

Returns coordinates of the normed vector corresponding to (AB) line

CHAPTER

TWO

PYTERK PACKAGE

2.1 Module contents

PyTerK - A Python Iterated K-fold cross validation with shuffling

By E Garel / JL Parouty - SIMaP 2021

This package allows you to perform a **statistical evaluation** of different learning strategies (Keras/sklearn) by varying different (hyper)parameters.

## Description :

It is possible to combine the following (hyper)parameters :

- datasets
- models (with their characteristics…)
- batch size
- epochs
- iterations
- k fold
- seed (to control pseudo random generator)

It is possible, for example, to combine 3 datasets, with 3 models and to perform for each combination, 5 iterations of a cross validation of KFold type, with k=10. In this case, the total number of models to test would be 3x3x5x10=450 training sessions… So, be careful, the number of model.fit can quickly be very important !

The tasks will be run in **parallel** on the different CPUs/cores available.

## Documentation and examples :

Here is a basinc example, detailled in a notebook :

``` import pyterk.config as config import pyterk.reporter as reporter import pyterk.task_manager as task_manager

settings = config.load('settings_example.yml')

task_manager.add_combinational_iterative_manyfold(settings, run_key= 'Example-03.1') task_manager.run()

reporter.show_run_reports(settings) ```

This will retrieve all settings from *settings_example.yml*, prepare the different tasks and execute them. The last call, intended to be used from a Jupyter lab notebook, displays a complete execution report.

You can find **3 full example notebooks**, with a setting file :

- settings_example.yml
```

- 01-Example-01.ipynb
- 02-Example-02.ipynb
- 03-Example-03.ipynb

`pyterk.VERSION = 2.14`
    pyterk version

2.2 Submodules

2.3 pyterk.config module

Configuration management.

The settings files allow to specify datasets and models.

## Utilisation: Loading a settings file : ` `settings = config.load('settings_example.yml')` ` or: ``` settings = config.load('settings_example.yml',

    datasets_dir_env='MY_DATASETS_DIR')

```

where MY_DATASETS_DIR is an environment variable that will override *datasets_dir* directive in settings file.

`pyterk.config.datasets = None`
    datasets profiles

`pyterk.config.load`(*filename*, *datasets_dir_env='PYTERK_DATASETS_DIR'*, *run_dir_env='PYTERK_RUN_DIR'*, *verbose=0*)

**Load a setting file and dfined datasets.** If given, environment variable can be use to overide *datasets_dir* directive from setting file. Usefull for portability between several sites.

**Parameters**

- **filename** (*string*) – Filename of the yaml setting file
- **datasets_dir_env** (*string*) – Name of the overiding environment variable
- **verbose** (*int*) – verbose mode for loaded datasets (0).

**Returns** A dict from setting file, completed by datasets and more.

`pyterk.config.models = None`
    models profiles

`pyterk.config.run_dir = None`
    run_dir, the place to put all output directories

`pyterk.config.runs = None`
    dict of runs section

`pyterk.config.settings = None`
    Dict of settings
```

2.4 pyterk.models module

This module is for internal use only - You do not have to interact with ;-).

`pyterk.models.get_model`(*profile*)

Get a model from a model profile. The profile contains the module and function name of the model, and the arguments. The model will be retrieved by calling the function with the arguments. :param profile: a model profile :type profile: dict

Returns keras model as defined in the profile.

Return type model (keras model)

2.5 pyterk.reporter module

Module to generate execution reports.

During the run of the tasks, the bestmodel and results are saved in h5 and json files:

- *about.json* : information and description of the task
- *history.json* : history from model.fit()
- *evaluation.json* : evaluation from model.evaluate()
- *bestmodel.h5* : best model

### Example :

``` reporter.show_run_reports(settings,

args = [‘dataset_id’,’model_id’,’batch_size’], evaluation = [2])

```

This module will retrieve information from json files and generate a report.

`pyterk.reporter.plot_confusion`(*run_dir*, *predict_type='softmax'*, *normalize='pred'*, *figsize=(5, 5)*, *savefig=True*, *mplstyle='pyterk'*)

Plot a confusion matrix

Parameters

- **iterations_dir** – a directory with iterations subdirs (iter-000, iter-001, . . .)
- **predict_type** – sigmoid, softmax or classes
- **normalize** – true, pred, all or None (pred)
- **figsize** – figure size
- **savefig** – save fig (True) or not (False)

Returns Just plot the matrix and print report and hamming loss

`pyterk.reporter.plot_distribution`(*run_dir*, *metric_id=0*, *bins=10*, *min=None*, *max=None*, *figsize=(10, 8)*, *savefig=False*, *mplstyle='pyterk'*)

Plot distribution of a given metric from an evaluation.json saved file. For a kfold or an iterative kfold, all evaluation data are concatened in an evaluation.json file in main run_dir.

Parameters

- **run_dir** (*string*) – directory path of json evaluation file

- **metricid** (*int*) – number of metric to plot. Example : 2
- **min** (*int*) – min value
- **max** (*int*) – max value
- **bins** (*int*) – number of bins
- **figsize** (*tuple*) – figure size, default is (10,8)
- **savefig** (*boolean*) – if True, figure will be save in run_dir.
- **mplstyle** (*string*) – name of matplotlib style. default is 'pyterk', but all matplotlib are ok (default, bmh, . . .)

Returns Nothing, but display a beautifull distribution plot !

`pyterk.reporter.`**`plot_history`**(*run_dir*, *metric='val_mae'*, *min=None*, *max=None*, *figsize=(10, 8)*, *savefig=False*, *mplstyle='pyterk'*)

Plot history evolution from history.json saved file. For a kfold or an iterative kfold, all history data are concatened in history.json file in main run_dir. This will plot a curve for each one in a common plot.

Parameters

- **run_dir** (*string*) – directory path of json history file
- **metric** (*string*) – metric name to plot. Example : 'val_mae'
- **figsize** (*tuple*) – figure size, default is (10,8)
- **savefig** (*boolean*) – if True, figure will be save in run_dir.
- **mplstyle** (*string*) – name of matplotlib style. default is 'pyterk', but all matplotlib are ok (default, bmh, . . .)

Returns Nothing, but display a beautifull plot !

`pyterk.reporter.`**`plot_kfold_correlation`**(*run_dir*, *channel=0*, *figsize=(8, 6)*, *axes_min='auto'*, *axes_max='auto'*, *yy_deltamax=None*, *marker='o'*, *markersize=8*, *alpha=0.7*, *color='auto'*, *savefig=True*, *mplstyle='pyterk'*)

Plot a correlation for a (y_test, y_pred) saved json file.

Parameters

- **run_file** – a manyfold directory where kfold subdirectories are
- **channel** – composant of y to plot
- **figsize** (*tuple*) – figure size, default is (10,8)
- **axes_min** – min value for x and y axe. 'auto' or float
- **axes_max** – max value for x and y axe. 'auto' or float
- **mplstyle** (*string*) – name of matplotlib style. default is 'pyterk', but all matplotlib are ok (default, bmh, . . .)
- **marker** – marker, default is '.'
- **markersize** – marker size
- **alpha** – marker alpha
- **color** – plot color or 'auto'
- **savefig** – if True, save fig in run_dir

Returns -)

Return type Nothing, but display a beautifull correlation plot

`pyterk.reporter.show_report`(*run_dir*, *padding=''*, *sections=['title', 'context', 'args', 'settings', 'evaluation', 'monitoring', 'history', 'distribution', 'correlation']*, *context=['function', 'version', 'date', 'description', 'seed']*, *args=['run_dir', 'dataset_id', 'model_id', 'n_iter', 'k_fold', 'epochs', 'batch_size']*, *settings=['file', 'version', 'description', 'datasets_dir', 'run_dir']*, *evaluation=['all']*, *monitoring=['duration', 'used_data']*, *history=[{'metric': 'val_mae', 'min': None, 'max': None, 'figsize': (8, 6), 'savefig': True, 'mplstyle': 'pyterk'}]*, *distribution=[{'metric_id': 2, 'bins': 4, 'min': None, 'max': None, 'figsize': (8, 6), 'savefig': True, 'mplstyle': 'pyterk'}]*, *correlation=[{'axes_min': 'auto', 'axes_max': 'auto', 'figsize': (8, 6), 'marker': '.', 'markersize': 8, 'alpha': 0.7, 'color': 'auto', 'savefig': True, 'mplstyle': 'pyterk'}]*, *confusion=[{'normalize': 'pred', 'predict_type': 'softmax', 'figsize': (5, 5), 'savefig': True, 'mplstyle': 'pyterk'}]*)

Builds and displays a report from the json data of a given run_dir.

Parameters

- `run_dir` (*string*) – directory path of json report file
- `sections` (*list*) – list of sections to include in the report
- `context` (*list*) – informations to include in context section
- `args` (*list*) – informations to include in args section
- `settings` (*list*) – informations to include in settings section
- `evaluation` (*list*) – #metrics to include in evaluation section. ‘all’ mean all. Example : [0,1,2]
- `history` (*dict*) – parameters for history plot - see *plot_history*
- `distribution` (*dict*) – parameters for metrics distribution plot
- `correlation` (*dict*) – parameters for correlation plot
- `confusion` (*dict*) – parameters for confusion matrix (need yytest files)

`pyterk.reporter.show_run_reports`(*run_config*, *run_filter='.*'*, *sections=['title', 'context', 'args', 'settings', 'evaluation', 'monitoring', 'history', 'distribution', 'correlation', 'confusion']*, *context=['function', 'version', 'date', 'description', 'seed']*, *args=['run_dir', 'dataset_id', 'model_id', 'n_iter', 'k_fold', 'epochs', 'batch_size']*, *settings=['file', 'version', 'description', 'datasets_dir', 'run_dir']*, *evaluation=['all']*, *monitoring=['duration', 'used_data']*, *history=[{'metric': 'val_mae', 'min': None, 'max': None, 'figsize': (8, 6), 'savefig': True, 'mplstyle': 'pyterk'}]*, *distribution=[{'metric_id': 2, 'bins': 4, 'min': None, 'max': None, 'figsize': (8, 6), 'savefig': True, 'mplstyle': 'pyterk'}]*, *correlation=[{'axes_min': 'auto', 'axes_max': 'auto', 'figsize': (8, 6), 'marker': '.', 'markersize': 8, 'alpha': 0.7, 'color': 'auto', 'savefig': True, 'mplstyle': 'pyterk'}]*, *confusion=[{'normalize': 'pred', 'predict_type': 'softmax', 'figsize': (5, 5), 'savefig': True, 'mplstyle': 'pyterk'}]*)

Displays a full report in two parts, short and long, for all runs defined in the settings. Very simple to use…

Parameters

- `run_config` (*dict*) – settings, issued from config.load()
- `run_filter` (*regx*) – regex to filter run entries from yml settings file (.*)
- `sections` (*list*) – list of sections to include in the report

- **context** (*list*) – informations to include in context section
- **args** (*list*) – informations to include in args section
- **settings** (*list*) – informations to include in settings section
- **evaluation** (*list*) – #metrics to include in evaluation section. 'all' mean all. Example : [0,1,2]
- **history** (*dict*) – parameters for history plot - see *plot_history*
- **distribution** (*dict*) – parameters for metrics distribution plot
- **correlation** (*dict*) – parameters for correlation plot
- **confusion** (*dict*) – parameters for confusion matrix (need yytest files)

Returns Nothing, but display a short and long report, with index.

2.6 pyterk.task_manager module

Allows to generate tasks and to execute them in a distributed way.

See example notebook : *03-Example-03.ipynb*

Example : ``` task_manager.add_combinational_iterative_manyfold(settings = settings,

run_key= 'Example-03.3')

```

pyterk.task_manager.**add_combinational_iterative_manyfold**(*settings=None*, *run_key=None*, *verbose=1*)

Add tasks for a combinational iterative manyfold - *see 03-Example-03*.ipynb Generates all the tasks of the combinatorial described in the run section of the settings file. :param settings: settings :type settings: dict :param run_key: name of the config run section :type run_key: string :param verbose: verbosity of generated tasks :type verbose: int

**Returns** Nothings. Task are added to the pending taks queue.

pyterk.task_manager.**add_iterative_manyfold**(*settings=None*, *run_dir=None*, *dataset_id=None*, *model_id=None*, *n_iter=2*, *k_fold=10*, *epochs=10*, *batch_size=10*, *description=None*, *save_xxtest=False*, *save_yytest=False*, *verbose=1*)

Add tasks for an iterative manyfold - see *02-Example-02.ipynb* Generate n_ter*k_fold tasks, each iteration will generate a subdirectory in run_dir. :param settings: settings :type settings: dict :param run_dir: run directoty to output k results (json files and best model) :type run_dir: string :param dataset_id: datasets id in settings file :type dataset_id: string :param model_id: model id in settings file :type model_id: string :param n_iter: number of iteration :type n_iter: int :param k_fold: number of fold :type k_fold: int :param epochs: number of epochs :type epochs: int :param batch_size: size of batch :type batch_size: int :param description: description of the action :type description: string :param save_xxtest: save x_test as json file, or not :type save_xxtest: Boolean :param save_yytest: save y_test and y_pred as json file, or not :type save_yytest: Boolean :param verbose: verbosity of generated tasks :type verbose: int

**Returns** Nothings. Task are added to the pending taks queue.

pyterk.task_manager.**add_manyfold**(*settings=None*, *run_dir=None*, *dataset_id=None*, *model_id=None*, *k_fold=10*, *epochs=10*, *batch_size=10*, *description=None*, *save_xxtest=False*, *save_yytest=False*, *verbose=1*)

Add tasks for a manyfold - see *01-Example-01.ipynb* Generate k_fold tasks, each task will generate one subdirectory in run_dir. :param settings: settings :type settings: dict :param run_dir: run directoty to output k results (json
```

files and best model) :type run_dir: string :param dataset_id: datasets id in settings file :type dataset_id: string :param model_id: model id in settings file :type model_id: string :param k_fold: number of fold :type k_fold: int :param epochs: number of epochs :type epochs: int :param batch_size: size of batch :type batch_size: int :param description: description of the action :type description: string :param save_xxtest: save x_test as json file, or not :type save_xxtest: Boolean :param save_yytest: save y_test and y_pred as json file, or not :type save_yytest: Boolean :param verbose: verbosity of generated tasks :type verbose: int

Returns Nothings. Task are added to the pending taks queue.

`pyterk.task_manager.`**`reset`**`()`

Reset pending tasks. Suppress all of them !

`pyterk.task_manager.`**`run`**`(`*processes=None*, *maxtasksperchild=10*, *verbose=1*`)`

`pyterk.task_manager.`**`seed`**`(`*seed=None*`)`

Init random generators with given seed

`pyterk.task_manager.`**`show_tasks_size`**`()`

Print pending tasks size

2.7 pyterk.worker module

This module is for internal use only - You do not have to interact with ;-).

`pyterk.worker.`**`get_model_family`**`(`*model*`)`

Should return the model family : 'tensorflow', 'keras' or 'sklearn'

`pyterk.worker.`**`init`**`(`*s*, *l*, *v*`)`

`pyterk.worker.`**`model_fit`**`(`*run_dir=None*, *dataset_id=None*, *train_index=None*, *test_index=None*, *model_id=None*, *epochs=None*, *batch_size=None*, *seed=None*, *description=None*, *save_xxtest=False*, *save_yytest=False*`)`

`pyterk.worker.`**`model_fit_sklearn`**`(`*model*, *run_dir=None*, *x_train=None*, *y_train=None*, *x_test=None*, *y_test=None*, *save_xxtest=False*, *save_yytest=False*`)`

`pyterk.worker.`**`model_fit_tensorflow`**`(`*model*, *run_dir=None*, *x_train=None*, *y_train=None*, *x_test=None*, *y_test=None*, *epochs=None*, *batch_size=None*, *save_xxtest=False*, *save_yytest=False*`)`

CHAPTER THREE

MULTIPLEREGRESSION MODULE

Module to train Multiple Linear Regression with Scheffe ineteraction terms with iterative k-fold crossvalidation

Contains functions to :

- generate interactions
- train regression models
- plot iterative k-fold crossvalidation results

`MultipleRegression.`**`Scheffe_interactions_terms`**(*data*, *in_percent='True'*, *compo_columns=['Zr', 'Nb', 'Mo', 'Ti', 'Cr']*)

Shaping composition in percentage rate into percentage Compute interaction terms for all Scheffe interactions for quartic multiple regression and add it to dataframe data

Parameters **`panda.DataFrame`** – dataset that contains compositions in Zr, Nb, Mo, Ti, Cr in columns of the same name

Returns extended input dataset with interactions

Return type DataFrame

`MultipleRegression.`**`fit_outputs`**(*model_expression*, *k*, *nb_it*, *output*, *X*, *y*)

Takes an OLS model expression, and use it to perform regression between X and y . Model regression is performed using iterative k-fold crossvalidation Evaluation is performed through R2 and MAE computation

Parameters

- **`OLS-formula`** – contain OLS formula for regression
- **`k`** (*`int`*) – number of folds for iterative k-fold crossfvalidation
- **`nb_it`** (*`int`*) – number of iterations for iterative k-fold crossfvalidation
- **`output`** (*`str`*) – name of the Y output to fit
- **`X`** (*`panda.DtataFrame`*) – contains composition and interaction terms for regression input
- **`y`** (*`panda.DataFrame`*) – contains single column dataframe with regression output

Return model model coefficients and p-values

Return type statsmodels.regression.linear_model.RegressionResultsWrapper

Return MAE_list list of MAE for every run of iterative k-fold crossvalidation, between expected vs predicted value on test set

Return type list

Return R2_list list of R2 for every run of iterative k-fold crossvalidation, between expected vs predicted value on test set

Return type list

Returns Y_pred : list of predicted values on test set

Return type list

Returns Y_test : list of expected values on test set

Return type list

`MultipleRegression.`**`plot_result`**(*metric*, *output*, *val_metric*, *Y_pred*, *Y_test*, *min_hist*, *max_hist*, *iter*, *kfold*, *save_distri*, *save_regression*)

Plot metric histogram and regression between predictions and test values and save graphs

Parameters

- **`metric`** (*`str`*) – name of the metric distribution to plot
- **`output`** (*`str`*) – name of the Y output to fit
- **`val_metric`** (*`list`*) – list of MAE for every run of iterative k-fold crossvalidation, between expected vs predicted value on test set
- **`Y_pred`** (*`list`*) – list of predicted values on test set
- **`Y_test`** (*`list`*) – list of expected values on test set
- **`min_hist`** (*`int`*) – minimun of abscissa for metric distribution histogram
- **`max_hist`** (*`int`*) – maximum of abscissa for metric distribution histogram
- **`iter`** (*`int`*) – plot regression over a certain number of iterations
- **`save_distri`** (*`str`*) – path to save metric distribution
- **`save_regression`** (*`std`*) – path to save regression

Parm int kfold plot regression over a certain number of k-fold for each iteration

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

MATLAB MODULE INDEX

INDEX