

---

# Utilities Documentation

*Release 1.0.2*

**Brandon**

**May 01, 2020**



---

## Contents

---

|          |   |           |
|----------|---|-----------|
| <b>1</b> | <b>Introduction to pyllars</b>                  | <b>1</b>  |
| 1.1      | History . . . . .                               | 1         |
| <b>2</b> | <b>API</b>                                      | <b>3</b>  |
| 2.1      | Collection utilities . . . . .                  | 3         |
| 2.2      | Dask utilities . . . . .                        | 8         |
| 2.3      | Logging utilities . . . . .                     | 12        |
| 2.4      | Matrix utilities . . . . .                      | 15        |
| 2.5      | Machine learning utilities . . . . .            | 18        |
| 2.6      | Matplotlib utilities . . . . .                  | 29        |
| 2.7      | Natural language processing utilities . . . . . | 41        |
| 2.8      | Pandas utilities . . . . .                      | 41        |
| 2.9      | Statistics utilities . . . . .                  | 47        |
| 2.10     | String utilities . . . . .                      | 51        |
| <b>3</b> | <b>Domain-specific API</b>                      | <b>57</b> |
| 3.1      | Physionet utilities . . . . .                   | 57        |
| <b>4</b> | <b>Tutorials</b>                                | <b>65</b> |
| 4.1      | Cross-validation tutorial . . . . .             | 65        |
| 4.2      | Bayesian model selection tutorial . . . . .     | 65        |
| <b>5</b> | <b>Indices and tables</b>                       | <b>67</b> |
|          | <b>Python Module Index</b>                      | <b>69</b> |
|          | <b>Index</b>                                    | <b>71</b> |



# CHAPTER 1

---

## Introduction to pyllars

---

This package contains many supporting utilities I find useful for Python 3.

### 1.1 History

This project was called *pymisc-utils*. Due to significant changes in the API when moving from version *0.2.11* to version *1.0.0*, the name was also changed to avoid confusion.

The new name is also more fun... “pyllars”, “**supporting** utilities”... get it? I’m here all week, folks. Try the veal.



This is the API for the `pyllars` library. These utilities are generally useful across different problem domains.

## 2.1 Collection utilities

This module implements helpers for working with collections. In some cases, the iterable is restricted to a particular type, such as a list or set.

Many of the function names mention specific data structures, such as “list”s or “dict”s, in the names for historical reasons. In most cases, these functions work with any instance of the more general type (such as *Iterable* or *Mapping*). Please see the specific documentation for more details, though.

### 2.1.1 Iterable helpers

|  |   |
|--|---|
| <code>apply_no_return(items, func, *args, ...)</code>              | Apply <i>func</i> to each item in <i>items</i>                                      |
| <code>flatten_lists(list_of_lists)</code>                          | Flatten a list of iterables into a single list                                      |
| <code>is_iterator_exhausted(iterator, re-<br/>turn_element)</code> | Check if the iterator is exhausted  |
| <code>list_insert_list(l, to_insert, index)</code>                 | Insert <i>to_insert</i> into a shallow copy of <i>l</i> at position <i>index</i> .  |
| <code>list_remove_list(l, to_remove)</code>                        | Remove items in <i>to_remove</i> from <i>l</i>                                      |
| <code>list_to_dict(l, f)</code>                                    | Convert the list to a dictionary in which keys and values are adjacent in the list. |
| <code>remove_nones(l, return_np_array)</code>                      | Remove <i>None</i> ’s from <i>l</i>   |
| <code>replace_none_with_empty_iter(i)</code>                       | Return an empty iterator if <i>i</i> is <i>None</i> .                               |
| <code>wrap_in_list(maybe_sequence)</code>                          | If <i>maybe_sequence</i> is not a sequence, then wrap it in a list                  |
| <code>wrap_string_in_list(maybe_string)</code>                     | If <i>maybe_string</i> is a string, then wrap it in a list.                         |

## 2.1.2 Set helpers

|  |   |
|--|---|
| <code>wrap_in_set(maybe_set, wrap_string)</code>               | If <i>maybe_set</i> is not a set, then wrap it in a set.          |
| <code>get_set_pairwise_intersections(dict_of_sets, ...)</code> | Find the pairwise intersections among sets in <i>dict_of_sets</i> |
| <code>merge_sets(*set_args)</code>                             | Given any number of sets, merge them into a single set            |

## 2.1.3 Mapping helpers

|   |   |
|---|---|
| <code>reverse_dict(d)</code>            | Create a new dictionary in which the keys and values of <i>d</i> are switched |
| <code>sort_dict_keys_by_value(d)</code> | Sort the keys in <i>d</i> by their value and return as a list                 |

## 2.1.4 Definitions

This module implements helpers for working with collections. In some cases, the iterable is restricted to a particular type, such as a list or set.

```
pyllars.collection_utils.apply_no_return(items: Iterable, func: Callable, *args,
                                           progress_bar: bool = False, total_items:
                                           Optional[int] = None, **kwargs) → None
```

Apply *func* to each item in *items*

Unlike `map()`, this function does not return anything.

### Parameters

- **items** (*typing.Iterable*) – An iterable
- **func** (*typing.Callable*) – The function to apply to each item
- **args** – Positional arguments for *func*.
- **kwargs** – Keyword arguments to pass to *func*
- **progress\_bar** (*bool*) – Whether to show a progress bar when waiting for results.
- **total\_items** (*int or None*) – The number of items in *items*. If not given, *len* is used. Presumably, this is used when *items* is a generator and *len* does not work.

**Returns** `None` – If a return value is expected, use list comprehension instead.

**Return type** `None`

```
pyllars.collection_utils.flatten_lists(list_of_lists: Iterable) → List
```

Flatten a list of iterables into a single list

This function does not further flatten inner iterables.

**Parameters** **list\_of\_lists** (*typing.Iterable*) – The iterable to flatten

**Returns** **flattened\_list** – The flattened list

**Return type** *typing.List*

```
pyllars.collection_utils.get_set_pairwise_intersections(dict_of_sets: Mapping[str,
                                                                              Set], return_intersections:
                                                                              bool = True) → pandas.core.frame.DataFrame
```

Find the pairwise intersections among sets in *dict\_of\_sets*



**Parameters**

- **dict\_of\_sets** (*typing.Mapping[str, typing.Set]*) – A mapping in which the keys are the “names” of the sets and the values are the actual sets
- **return\_intersections** (*bool*) – Whether to include the actual set intersections in the return. If *False*, then only the intersection size will be included.

**Returns**

**df\_pairwise\_intersections** – A dataframe with the following columns:

- *set1* : the name of one set in the pair
- *set2* : the name of the second set in the pair
- *len(set1)* : the size of set1
- *len(set2)* : the size of set2
- *len(intersection)* : the size of the intersection
- *coverage\_small* : the fraction of the smaller of set1 or set2 in the intersection
- *coverage\_large* : the fraction of the larger of set1 or set2 in the intersection
- *intersection* : the intersection set. Only included if *return\_intersections* is *True*.

**Return type** `pandas.DataFrame`

`pyllars.collection_utils.is_iterator_exhausted` (*iterator: Iterable, return\_element: bool = False*) → `Tuple[bool, object]`

Check if the iterator is exhausted

N.B. THIS CONSUMES THE NEXT ELEMENT OF THE ITERATOR! The *return\_element* parameter can change this behavior.

This method is adapted from this SO question: <https://stackoverflow.com/questions/661603>

**Parameters**

- **iterator** (*typing.Iterable*) – The iterator
- **return\_element** (*bool*) – Whether to return the next element of the iterator

**Returns**

- **is\_exhausted** (*bool*) – Whether there was a next element in the iterator
- **[optional] next\_element** (*object*) – If *return\_element* is *True*, then the consumed element is also returned.

`pyllars.collection_utils.list_insert_list` (*l: Sequence, to\_insert: Sequence, index: int*) → `List`  
 Insert *to\_insert* into a shallow copy of *l* at position *index*.

This function is adapted from: <http://stackoverflow.com/questions/7376019/>

**Parameters**

- **l** (*typing.Sequence*) – An iterable
- **to\_insert** (*typing.Sequence*) – The items to insert
- **index** (*int*) – The location to begin the insertion

**Returns** `updated_l` – A list with *to\_insert* inserted into *l* at position *index*

**Return type** `typing.List`

`pyllars.collection_utils.list_remove_list` (*l*: *Iterable*, *to\_remove*: *Container*) → *List*  
Remove items in *to\_remove* from *l*

Note that “not in” is used to match items in *to\_remove*. Additionally, the return *is not* lazy.

**Parameters**

- *l* (*typing.Iterable*) – An iterable of items
- *to\_remove* (*typing.Container*) – The set of items to remove from *l*

**Returns** *copy\_of\_l* – A shallow copy of *l* without the items in *to\_remove*.

**Return type** *typing.List*

`pyllars.collection_utils.list_to_dict` (*l*: *Sequence*, *f*: *Optional[Callable]* = *None*) → *Dict*  
Convert the list to a dictionary in which keys and values are adjacent in the list. Optionally, a function *f* can be passed to apply to each value before adding it to the dictionary.

**Parameters**

- *l* (*typing.Sequence*) – The list of items
- *f* (*typing.Callable*) – A function to apply to each value before inserting it into the list. For example, *float* could be passed to convert each value to a float.

**Returns** *d* – The dictionary, defined as described above

**Return type** *typing.Dict*

## Examples

```
l = ["key1", "value1", "key2", "value2"]
list_to_dict(l, f) == {"key1": f("value1"), "key2": f("value2")}
```

`pyllars.collection_utils.merge_sets` (*\*set\_args*) → *Set*  
Given any number of sets, merge them into a single set

N.B. This function only performs a “shallow” merge. It does not handle nested containers within the “outer” sets.

**Parameters** *set\_args* (*typing.Iterable[typing.Container]*) – The sets to merge

**Returns** *merged\_set* – A single set containing unique elements from each of the input sets

**Return type** *typing.Set*

`pyllars.collection_utils.remove_nones` (*l*: *Iterable*, *return\_np\_array*: *bool* = *False*) → *List*  
Remove *None*’s from *l*

Compared to other single-function tests, this uses “is” and avoids strange behavior with data frames, lists of bools, etc.

This function returns a shallow copy and is not lazy.

N.B. This does not test nested lists. So, for example, a list of lists of *None* values would be unchanged by this function.

**Parameters**

- *l* (*typing.Iterable*) – The iterable
- *return\_np\_array* (*bool*) – If true, the filtered list will be wrapped in an *np.array*.

**Returns** *l\_no\_nones* – A list or *np.array* with the *None*’s removed from *l*

**Return type** `typing.List`

`pyllars.collection_utils.replace_none_with_empty_iter` (*i*: `Optional[Iterable]`) → `Iterable`

Return an empty iterator if *i* is *None*. Otherwise, return *i*.

The purpose of this function is to make iterating over results from functions which return either an iterator or *None* cleaner. This function does not verify that *i* is actually an iterator.

**Parameters** *i* (*None* or `typing.Iterable`) – The possibly-empty iterator

**Returns** *i* – An empty list if iterator is *None*, or the original iterator otherwise

**Return type** `typing.Iterable`

`pyllars.collection_utils.reverse_dict` (*d*: `Mapping`) → `Dict`

Create a new dictionary in which the keys and values of *d* are switched

In the case of duplicate values, it is arbitrary which will be retained.

**Parameters** *d* (`typing.Mapping`) – The mapping

**Returns** `reversed_d` – A dictionary in which the values of *d* now map to the keys

**Return type** `typing.Dict`

`pyllars.collection_utils.sort_dict_keys_by_value` (*d*: `Mapping`) → `List`

Sort the keys in *d* by their value and return as a list

This function uses *sorted*, so the values should be able to be sorted appropriately by that builtin function.

**Parameters** *d* (`typing.Mapping`) – The dictionary

**Returns** `sorted_keys` – The keys sorted by the associated values

**Return type** `typing.List`

`pyllars.collection_utils.wrap_in_list` (*maybe\_sequence*: `Any`) → `Sequence`

If *maybe\_sequence* is not a sequence, then wrap it in a list

See `pyllars.validation_utils.is_sequence()` for more details about what counts as a sequence.

**Parameters** *maybe\_sequence* (`typing.Any`) – An object which may be a sequence

**Returns** `list` – Either the original object, or *maybe\_sequence* wrapped in a list, if it was not already a sequence

**Return type** `typing.Sequence`

`pyllars.collection_utils.wrap_in_set` (*maybe\_set*: `Optional[Any]`, *wrap\_string*: `bool = True`) → `Set`

If *maybe\_set* is not a set, then wrap it in a set.

**Parameters**

- ***maybe\_set*** (`typing.Optional[typing.Any]`) – An object which may be a set
- ***wrap\_string*** (`bool`) – Whether to wrap *maybe\_set* as a singleton if it is a string. Otherwise, the string will be converted into a set of individual characters.

**Returns** *s* – Either the original object, or *maybe\_set* wrapped in a set, if it was not already a set. If *maybe\_set* was *None*, then an empty set is returned.

**Return type** `typing.Set`

`pyllars.collection_utils.wrap_string_in_list` (*maybe\_string*: `Any`) → `Sequence`

If *maybe\_string* is a string, then wrap it in a list.

The motivation for this function is that some functions return either a single string or multiple strings as a list. The return value of this function can be iterated over safely.

This function will fail if *maybe\_string* is not a string and it not a sequence.

**Parameters** *maybe\_string* (*typing.Any*) – An object which may be a string

**Returns** *l* – Either the original object, or *maybe\_string* wrapped in a list, if it was a string}

**Return type** *typing.Sequence*

## 2.2 Dask utilities

This module contains helpers for using dask: <https://dask.pydata.org/en/latest/>

### 2.2.1 Starting or connecting to a server

|  |   |
|--|---|
| <code>connect(args)</code>                                 | Connect to the dask cluster specified by the arguments in <i>args</i> |
| <code>add_dask_options(parser, num_procs, ...)</code>      | Add options for connecting to and/or controlling a local dask cluster |
| <code>add_dask_values_to_args(args, num_procs, ...)</code> | Add the options for a dask cluster to the given argparse namespace    |

### 2.2.2 Submitting jobs

|   |   |
|---|---|
| <code>apply_iter(it, client, func, *args, ...)</code>       | Distribute calls to <i>func</i> on each item in <i>it</i> across <i>client</i> .        |
| <code>apply_df(data_frame, client, func, *args, ...)</code> | Distribute calls to <i>func</i> on each row in <i>data_frame</i> across <i>client</i> . |
| <code>apply_groups(groups, client, func, *args, ...)</code> | Distribute calls to <i>func</i> on each group in <i>groups</i> across <i>client</i> .   |

### 2.2.3 Other dask helpers

|  |  |
|--|--|
| <code>check_status(f_list)</code>                        | Collect the status counts of a list of futures |
| <code>collect_results(f_list, finished_only, ...)</code> | Collect the results from a list of futures     |
| <code>cancel_all(f_list[, pending_only])</code>          | Cancel all (pending) tasks in the list         |

### 2.2.4 scikit-learn helpers

|   |   |
|---|---|
| <code>dask_pipeline(pipeline, dask_client)</code> | This class is a simple wrapper to submit an sklearn pipeline to a dask cluster for fitting. |
|---|---|

### 2.2.5 Definitions

This module contains helpers for using dask: <https://dask.pydata.org/en/latest/>

```
pyllars.dask_utils.add_dask_options(parser: argparse.ArgumentParser, num_procs: int = 1,
                                   num_threads_per_proc: int = 1, cluster_location: str =
                                   'LOCAL') → None
```

Add options for connecting to and/or controlling a local dask cluster

#### Parameters

- **parser** (*argparse.ArgumentParser*) – The parser to which the options will be added
- **num\_procs** (*int*) – The default number of processes for a local cluster
- **num\_threads\_per\_proc** (*int*) – The default number of threads for each process for a local cluster
- **cluster\_location** (*str*) – The default location of the cluster

**Returns** *None* – A “dask cluster options” group is added to the parser

**Return type** *None*

```
pyllars.dask_utils.add_dask_values_to_args(args: argparse.Namespace, num_procs: int
                                           = 1, num_threads_per_proc: int = 1, cluster_location: str = 'LOCAL',
                                           client_restart: bool = False) → None
```

Add the options for a dask cluster to the given argparse namespace

This function is mostly intended as a helper for use in ipython notebooks.

#### Parameters

- **args** (*argparse.Namespace*) – The namespace on which the arguments will be set
- **num\_procs** (*int*) – The number of processes for a local cluster
- **num\_threads\_per\_proc** (*int*) – The number of threads for each process for a local cluster
- **cluster\_location** (*str*) – The location of the cluster
- **client\_restart** (*bool*) – Whether to restart the client after connection

**Returns** *None* – The respective options will be set on the namespace

**Return type** *None*

```
pyllars.dask_utils.apply_df(data_frame: pandas.core.frame.DataFrame, client: distributed.client.Client,
                             func: Callable, *args, return_futures: bool = False,
                             progress_bar: bool = True, priority: int = 0, **kwargs) → List
```

Distribute calls to *func* on each row in *data\_frame* across *client*.

Additionally, *args* and *kwargs* are passed to the function call.

#### Parameters

- **data\_frame** (*pandas.DataFrame*) – A data frame
- **client** (*dask.distributed.Client*) – A dask client
- **func** (*typing.Callable*) – The function to apply to each row in *data\_frame*
- **args** – Positional arguments to pass to *func*
- **kwargs** – Keyword arguments to pass to *func*

- **return\_futures** (*bool*) – Whether to wait for the results (*False*, the default) or return a list of dask futures (when *True*). If a list of futures is returned, the *result* method should be called on each of them at some point before attempting to use the results.
- **progress\_bar** (*bool*) – Whether to show a progress bar when waiting for results. The parameter is only relevant when *return\_futures* is *False*.
- **priority** (*int*) – The priority of the submitted tasks. Please see the dask documentation for more details: <http://distributed.readthedocs.io/en/latest/priority.html>

**Returns results** – Either the result of each function call or a future which will give the result, depending on the value of *return\_futures*

**Return type** `typing.List`

```
pyllars.dask_utils.apply_groups (groups: pandas.core.groupby.generic.DataFrameGroupBy,  
                                client: distributed.client.Client, func: Callable, *args, re-  
                                turn_futures: bool = False, progress_bar: bool = True,  
                                priority: int = 0, **kwargs) → List
```

Distribute calls to *func* on each group in *groups* across *client*.

Additionally, *args* and *kwargs* are passed to the function call.

#### Parameters

- **groups** (*pandas.DataFrameGroupBy*) – The result of a call to *groupby* on a data frame
- **client** (*distributed.Client*) – A dask client
- **func** (*typing.Callable*) – The function to apply to each group in *groups*
- **args** – Positional arguments to pass to *func*
- **kwargs** – Keyword arguments to pass to *func*
- **return\_futures** (*bool*) – Whether to wait for the results (*False*, the default) or return a list of dask futures (when *True*). If a list of futures is returned, the *result* method should be called on each of them at some point before attempting to use the results.
- **progress\_bar** (*bool*) – Whether to show a progress bar when waiting for results. The parameter is only relevant when *return\_futures* is *False*.
- **priority** (*int*) – The priority of the submitted tasks. Please see the dask documentation for more details: <http://distributed.readthedocs.io/en/latest/priority.html>

**Returns results** – Either the result of each function call or a future which will give the result, depending on the value of *return\_futures*.

**Return type** `typing.List`

```
pyllars.dask_utils.apply_iter (it: Iterable, client: distributed.client.Client, func: Callable, *args,  
                               return_futures: bool = False, progress_bar: bool = True, priority:  
                               int = 0, **kwargs) → List
```

Distribute calls to *func* on each item in *it* across *client*.

#### Parameters

- **it** (*typing.Iterable*) – The inputs for *func*
- **client** (*dask.distributed.Client*) – A dask client
- **func** (*typing.Callable*) – The function to apply to each item in *it*
- **args** – Positional arguments to pass to *func*

- **kwargs** – Keyword arguments to pass to *func*
- **return\_futures** (*bool*) – Whether to wait for the results (*False*, the default) or return a list of dask futures (when *True*). If a list of futures is returned, the *result* method should be called on each of them at some point before attempting to use the results.
- **progress\_bar** (*bool*) – Whether to show a progress bar when waiting for results. The parameter is only relevant when *return\_futures* is *False*.
- **priority** (*int*) – The priority of the submitted tasks. Please see the dask documentation for more details: <http://distributed.readthedocs.io/en/latest/priority.html>

**Returns results** – Either the result of each function call or a future which will give the result, depending on the value of *return\_futures*

**Return type** `typing.List`

`pyllars.dask_utils.cancel_all(f_list: Iterable[distributed.client.Future], pending_only=True) → None`

Cancel all (pending) tasks in the list

By default, only pending tasks are cancelled.

**Parameters**

- **f\_list** (*Iterable[dask.distributed.client.Future]*) – The list of futures
- **pending\_only** (*bool*) – Whether to cancel only tasks whose status is ‘pending’

**Returns None** – The specified tasks are cancelled.

**Return type** `None`

`pyllars.dask_utils.check_status(f_list: Iterable[distributed.client.Future]) → collections.Counter`

Collect the status counts of a list of futures

This is primarily intended to check the status of jobs submitted with the various *apply* functions when *return\_futures* is *True*.

**Parameters f\_list** (*typing.List[dask.distributed.client.Future]*) – The list of futures

**Returns status\_counter** – The number of futures with each status

**Return type** `collections.Counter`

`pyllars.dask_utils.collect_results(f_list: Iterable[distributed.client.Future], finished_only: bool = True, progress_bar: bool = False) → List`

Collect the results from a list of futures

By default, only results from finished tasks will be collected. Thus, the function is (more or less) non-blocking.

**Parameters**

- **f\_list** (*typing.List[dask.distributed.client.Future]*) – The list of futures
- **finished\_only** (*bool*) – Whether to collect only results for jobs whose status is ‘finished’
- **progress\_bar** (*bool*) – Whether to show a progress bar when waiting for results. The parameter is only relevant when *return\_futures* is *False*.

**Returns results** – The results for each (finished, if specified) task

**Return type** `typing.List`

`pyllars.dask_utils.connect` (*args: argparse.Namespace*) → Tuple[`distributed.client.Client`, Optional[`distributed.deploy.local.LocalCluster`]]

Connect to the dask cluster specified by the arguments in *args*

Specifically, this function uses *args.cluster\_location* to determine whether to start a `dask.distributed.LocalCluster` (in case *args.cluster\_location* is “LOCAL”) or to (attempt to) connect to an existing cluster (any other value).

If a local cluster is started, it will use a number of worker processes equal to *args.num\_procs*. Each process will use *args.num\_threads\_per\_proc* threads. The scheduler for the local cluster will listen to a random port.

**Parameters** *args* (*argparse.Namespace*) – A namespace containing the following fields:

- *cluster\_location*
- *client\_restart*
- *num\_procs*
- *num\_threads\_per\_proc*

**Returns**

- **client** (*dask.distributed.Client*) – The client for the dask connection
- **cluster** (*dask.distributed.LocalCluster* or *None*) – If a local cluster is started, the reference to the local cluster object is returned. Otherwise, *None* is returned.

**class** `pyllars.dask_utils.dask_pipeline` (*pipeline, dask\_client*)

This class is a simple wrapper to submit an sklearn pipeline to a dask cluster for fitting.

## Examples

```
my_pipeline = sklearn.pipeline.Pipeline(steps)
d_pipeline = dask_pipeline(my_pipeline, dask_client)
d_pipeline_fit = d_pipeline.fit(X, y)
pipeline_fit = d_pipeline_fit.collect_results()
```

**collect\_results** ()

Collect the “fit” pipeline from *dask\_client*. Then, cleanup the references to the future and client.

**fit** (*X, y*)

Submit the call to *fit* of the underlying pipeline to *dask\_client*

`pyllars.dask_utils.get_dask_cmd_options` (*args: argparse.Namespace*) → List[str]

Extract the flags and options specified for dask from the parsed arguments.

Presumably, these were added with *add\_dask\_options*. This function returns the arguments as an array. Thus, they are suitable for use with *subprocess.run* and similar functions.

**Parameters** *args* (*argparse.Namespace*) – The parsed arguments

**Returns** *dask\_options* – The list of dask options and their values.

**Return type** `typing.List[str]`

## 2.3 Logging utilities

Utilities for interacting with the python logging module. Mostly, this module provides functions for easily adding command line options to an *argparse.ArgumentParser* and then setting logging parameters accordingly. More details



and examples for logging are given in the python documentation: \* **Introduction:** <https://docs.python.org/3/howto/logging.html> \* **Format string options:** <https://docs.python.org/3/library/logging.html#logrecord-attributes>

### 2.3.1 Command line helpers

|   |  |
|---|--|
| <code>add_logging_options(parser, ...)</code>           | Add options for controlling logging to an argument parser.   |
| <code>get_logging_cmd_options(args)</code>              | Extract the flags and options specified for logging from the parsed arguments.                           |
| <code>get_logging_options_string(args)</code>           | Extract the flags and options specified for logging from the parsed arguments and join them as a string. |
| <code>update_logging(args[, logger, format_str])</code> | Update <i>logger</i> to use the settings in <i>args</i>  |

### 2.3.2 Other helpers

|   |      |  |
|---|------|--|
| <code>get_ipython_logger([logging_level, mat_str])</code> | for- | Get a logger for use in jupyter notebooks  |
| <code>set_logging_values(**kwargs)</code>                 |      | Set the logging options for the default logger as given. This is intended for use in tests or other cases where a CLI is not easily available. |

### 2.3.3 Definitions

Utilities for interacting with the python logging module. Mostly, this module provides functions for easily adding command line options to an `argparse.ArgumentParser` and then setting logging parameters accordingly. More details and examples for logging are given in the python documentation: \* **Introduction:** <https://docs.python.org/3/howto/logging.html> \* **Format string options:** <https://docs.python.org/3/library/logging.html#logrecord-attributes>

```
pyllars.logging_utils.add_logging_options(parser:      argparse.ArgumentParser,
                                         default_log_file: str = "",
                                         default_logging_level: str = 'WARNING',
                                         default_specific_logging_level: str = 'NOT-SET') → None
```

Add options for controlling logging to an argument parser.

In particular, it adds options for logging to a file, stdout and stderr. In addition, it adds options for controlling the logging level of each of the loggers, and a general option for controlling all of the loggers.

#### Parameters

- **parser** (`argparse.ArgumentParser`) – An argument parser
- **default\_log\_file** (`str`) – The default for the `-log-file` flag
- **default\_logging\_level** (`str`) – The default for the `-logging-level` flag
- **default\_specific\_logging\_level** (`str`) – The default for the `-{file,stdout,stderr}-logging-level` flags

**Returns** `None` – The parser has the additional options added

**Return type** `None`

```
pyllars.logging_utils.add_logging_values_to_args (args: argparse.Namespace, log_file:
                                                    str = "", log_stdout: bool = False,
                                                    no_log_stderr: bool = False, log-
                                                    ging_level: str = 'WARNING',
                                                    file_logging_level: str = 'NOTSET',
                                                    stdout_logging_level: str = 'NOT-
                                                    SET', stderr_logging_level: str =
                                                    'NOTSET') → None
```

Add the options from `add_logging_options` to `args`. This is intended for use in notebooks or other settings where the logging option functionality is required, but a command line interface is not used. :param args: The namespace to which the options will be added :type args: `argparse.Namespace` :param log\_file: The path to a log file. If this is the empty string, then a log file

will not be used.

#### Parameters

- **log\_stdout** (*bool*) – Whether to log to stdout
- **no\_log\_stderr** (*bool*) – Whether to `_not_` log to stderr. So, if this is `True`, then logging statements `_will_` be written to stderr. (The negative is used because that is more natural for the command line arguments.)
- **logging\_level** (*str*) – The logging level for all loggers
- **{file, stdout, stderr}\_logging\_level** (*str*) – The logging level for the specific loggers. This overrides `logging_level` for the respective logger when given.

**Returns** `None` – The respective options will be set on the namespace

**Return type** `None`

```
pyllars.logging_utils.get_ipython_logger (logging_level='DEBUG',
                                           format_str='%(levelname)-8s : %(message)s')
```

Get a logger for use in jupyter notebooks

This function is useful because the default logger in notebooks has a number of handlers by default. This function removes those, so the logger behaves as expected.

#### Parameters

- **logging\_level** (*str*) – The logging level for the logger. This can be updated later.
- **format\_str** (*str*) – The logging format string. Please see the python logging documentation for examples and more description.

**Returns** `logger` – A logger suitable for use in a notebook

**Return type** `logging.Logger`

```
pyllars.logging_utils.get_logging_cmd_options (args: argparse.Namespace) → str
```

Extract the flags and options specified for logging from the parsed arguments.

Presumably, these were added with `add_logging_options`. Compared to `get_logging_options_string`, this function returns the arguments as an array. Thus, they are suitable for use with `subprocess.run` and similar functions.

**Parameters** **args** (*argparse.Namespace*) – The parsed arguments

**Returns** `logging_options` – The list of logging options and their values.

**Return type** `typing.List[str]`

`pyllars.logging_utils.get_logging_options_string(args: argparse.Namespace) → str`  
 Extract the flags and options specified for logging from the parsed arguments and join them as a string.

Presumably, these were added with `add_logging_options`. Compared to `get_logging_cmd_options`, this function returns the arguments as a single long string. Thus, they are suitable for use when building single strings to pass to the command line (such as with `subprocess.run` when `shell` is `True`).

**Parameters** `args` (`argparse.Namespace`) – The parsed arguments

**Returns** `logging_options_str` – A string containing all logging flags and options

**Return type** `str`

`pyllars.logging_utils.set_logging_values(**kwargs) → None`

Set the logging options for the default logger as given This is intended for use in tests or other cases where a CLI is not easily available. :param kwargs: These are passed unaltered to `add_logging_values_to_args`. Please see

that documentation for details on valid options and their effect.

**Returns** `None` – The respective options will be set for the default logger

**Return type** `None`

`pyllars.logging_utils.update_logging(args, logger=None, format_str='%(levelname)-8s %(name)-8s %(asctime)s : %(message)s')`

Update *logger* to use the settings in *args*

Presumably, the logging options were added with `add_logging_options`. :param args: A namespace with the arguments added by `add_logging_options` :type args: `argparse.Namespace` :param logger: The logger which will be updated. If `None` is given, then the default

logger will be updated.

**Parameters** `format_str` (`str`) – The logging format string. Please see the python logging documentation for examples and more description.

**Returns** the specified logging options

**Return type** `None`, but the default (or given) logger is updated to take into account

## 2.4 Matrix utilities

Helpers for working with (sparse) 2d matrices

### 2.4.1 Sparse matrix helpers

|  |  |
|--|--|
| <code>get_dense_row(matrix, row[, dtype])</code>           | Extract <i>row</i> from the sparse <i>matrix</i>                 |
| <code>sparse_matrix_to_dense(sparse_matrix)</code>         | Convert <i>sparse_matrix</i> to a dense numpy array              |
| <code>sparse_matrix_to_list(sparse_matrix)</code>          | Convert <i>sparse_matrix</i> to a list of “sparse row vectors”.  |
| <code>write_sparse_matrix(target, a, compress, ...)</code> | Write <i>a</i> to the file <i>target</i> in matrix market format |

### 2.4.2 Matrix operation helpers

|  |   |
|--|---|
| <code>col_op(m, op)</code>               | Apply op to each column in the matrix.                      |
| <code>col_sum(m)</code>                  | Calculate the sum of each column in the matrix.             |
| <code>col_sum_mean(m, return_var)</code> | Calculate the mean of the sum of each column in the matrix. |
| <code>normalize_columns(matrix)</code>   | Normalize the columns of the given (dense) matrix           |
| <code>row_op(m, op)</code>               | Apply op to each row in the matrix.                         |
| <code>row_sum(m)</code>                  | Calculate the sum of each row in the matrix.                |
| <code>row_sum_mean(m, var)</code>        | Calculate the mean of the sum of each row in the matrix.    |
| <code>normalize_rows(matrix)</code>      | Normalize the rows of the given (dense) matrix              |

### 2.4.3 Other helpers

|  |   |
|--|---|
| <code>matrix_multiply(m1, m2, m3)</code>       | Multiply the three matrices                 |
| <code>permute_matrix(m, is_flat, shape)</code> | Randomly permute the entries of the matrix. |

### 2.4.4 Definitions

Helpers for working with (sparse) 2d matrices

`pyllars.matrix_utils.col_op(m, op)`  
Apply op to each column in the matrix.

`pyllars.matrix_utils.col_sum(m)`  
Calculate the sum of each column in the matrix.

`pyllars.matrix_utils.col_sum_mean(m: numpy.ndarray, return_var: bool = False) → float`  
Calculate the mean of the sum of each column in the matrix.

Optionally, the variances of the column sums can also be calculated.

#### Parameters

- **m** (`numpy.ndarray`) – The (2d) matrix
- **var** (`bool`) – Whether to calculate the variances

#### Returns

- **mean** (`float`) – The mean of the column sums in the matrix
- **variance** (`float`) – If `return_var` is True, then the variance of the column sums

`pyllars.matrix_utils.get_dense_row(matrix: scipy.sparse.base.spmatrix, row: int, dtype=<class 'float'>, max_length: Optional[int] = None) → numpy.ndarray`

Extract *row* from the sparse *matrix*

#### Parameters

- **matrix** (`scipy.sparse.spmatrix`) – The sparse matrix
- **row** (`int`) – The 0-based row index
- **dtype** (`type`) – The base type of elements of *matrix*. This is used for the corner case where *matrix* is essentially a sparse column vector.
- **max\_length** (`typing.Optional[int]`) – The maximum number of columns to include in the returned row.

**Returns** **row** – The specified row (as a 1d numpy array)

**Return type** `numpy.ndarray`

`pyllars.matrix_utils.matrix_multiply(m1: numpy.ndarray, m2: numpy.ndarray, m3: numpy.ndarray) → numpy.ndarray`

Multiply the three matrices

This function performs the multiplications in an order such that the size of the intermediate matrix created by the first matrix multiplication is as small as possible.

**Parameters** `m{1, 2, 3}` (*numpy.ndarray*) – The (2d) matrices

**Returns** `product_matrix` – The product of the three input matrices

**Return type** `numpy.ndarray`

`pyllars.matrix_utils.normalize_columns(matrix: numpy.ndarray) → numpy.ndarray`

Normalize the columns of the given (dense) matrix

**Parameters** `m` (*numpy.ndarray*) – The (2d) matrix

**Returns** `normalized_matrix` – The matrix normalized such that all column sums are 1

**Return type** `numpy.ndarray`

`pyllars.matrix_utils.normalize_rows(matrix: numpy.ndarray) → numpy.ndarray`

Normalize the rows of the given (dense) matrix

**Parameters** `matrix` (*numpy.ndarray*) – The (2d) matrix

**Returns** `normalized_matrix` – The matrix normalized such that all row sums are 1

**Return type** `numpy.ndarray`

`pyllars.matrix_utils.permute_matrix(m: numpy.ndarray, is_flat: bool = False, shape: Optional[Tuple[int]] = None) → numpy.ndarray`

Randomly permute the entries of the matrix. The matrix is first flattened.

For reproducibility, the random seed of numpy should be set **before** calling this function.

**Parameters**

- `m` (*numpy.ndarray*) – The matrix (tensor, etc.)
- `is_flat` (*bool*) – Whether the matrix values have already been flattened. If they have been, then the desired shape must be passed.
- `shape` (*typing.Optional*[*typing.Tuple*]) – The shape of the output matrix, if `m` is already flattened

**Returns** `permuted_m` – A copy of `m` (with the same shape as `m`) with the values randomly permuted.

**Return type** `numpy.ndarray`

`pyllars.matrix_utils.row_op(m, op)`

Apply `op` to each row in the matrix.

`pyllars.matrix_utils.row_sum(m)`

Calculate the sum of each row in the matrix.

`pyllars.matrix_utils.row_sum_mean(m: numpy.ndarray, var: bool = False) → float`

Calculate the mean of the sum of each row in the matrix.

Optionally, the variances of the row sums can also be calculated.

**Parameters**

- `m` (*numpy.ndarray*) – The (2d) matrix

- **return\_var** (*bool*) – Whether to calculate the variances

**Returns**

- **mean** (*float*) – The mean of the row sums in the matrix
- **variance** (*float*) – If *return\_var* is *True*, then the variance of the row sums

`pyllars.matrix_utils.sparse_matrix_to_dense` (*sparse\_matrix*: *scipy.sparse.base.spmatrix*)  
→ *numpy.ndarray*

Convert *sparse\_matrix* to a dense numpy array

**Parameters** *sparse\_matrix* (*scipy.sparse.spmatrix*) – The sparse scipy matrix

**Returns** *dense\_matrix* – The dense (2d) numpy array

**Return type** *numpy.ndarray*

`pyllars.matrix_utils.sparse_matrix_to_list` (*sparse\_matrix*: *scipy.sparse.base.spmatrix*)  
→ *List*

Convert *sparse\_matrix* to a list of “sparse row vectors”.

In this context, a “sparse row vector” is simply a sparse matrix with dimensionality (1, *sparse\_matrix.shape[1]*).

**Parameters** *sparse\_matrix* (*scipy.sparse.spmatrix*) – The sparse scipy matrix

**Returns** *list\_of\_sparse\_row\_vectors* – The list of sparse row vectors

**Return type** *typing.List[scipy.sparse.spmatrix]*

`pyllars.matrix_utils.write_sparse_matrix` (*target*: *str*, *a*: *scipy.sparse.base.spmatrix*, *compress*: *bool* = *True*, *\*\*kwargs*) → *None*

Write *a* to the file *target* in matrix market format

This function is a drop-in replacement for `scipy.io.mmwrite`. The only difference is that it gzip compresses the output by default. It *does not* alter the file extension, which should likely end in “*mtx.gz*” except in special circumstances.

If *compress* is *True*, then this function imports *gzip*.

**Parameters**

- **target** (*str*) – The complete path to the output file, including file extension
- **a** (*scipy.sparse.spmatrix*) – The sparse matrix
- **compress** (*bool*) – Whether to compress the output
- **\*\*kwargs** (*<key>=<value> pairs*) – These are passed through to `scipy.io.mmwrite()`. Please see the scipy documentation for more details.

**Returns**

**Return type** *None*, but the matrix is written to disk

## 2.5 Machine learning utilities

This module contains utilities for common machine learning tasks.

In particular, this module focuses on tasks “surrounding” machine learning, such as cross-fold splitting, performance evaluation, etc. It does not include helpers for use directly in `sklearn.pipeline.Pipeline`.

### 2.5.1 Creating and managing cross-validation

|   |   |
|---|---|
| <code>get_cv_folds(y, num_splits, use_stratified, ...)</code> | Assign a split to each row based on the values of <code>y</code>      |
| <code>get_train_val_test_splits(df, ...)</code>               | Get the appropriate training, validation, and testing split masks     |
| <code>get_fold_data(df, target_field, m_train, ...)</code>    | Prepare a data frame for <i>sklearn</i> according to the given splits |

### 2.5.2 Evaluating results

|  |   |
|--|---|
| <code>collect_binary_classification_metrics(...)</code>      | Collect various binary classification performance metrics for the predictions               |
| <code>collect_multiclass_classification_metrics(...)</code>  | Calculate various multi-class classification performance metrics                            |
| <code>collect_regression_metrics(y_true, y_pred, ...)</code> | Collect various regression performance metrics for the predictions                          |
| <code>calc_hand_and_till_m_score(y_true, y_score)</code>     | Calculate the (multi-class AUC) $M$ score from Equation (7) of Hand and Till (2001).        |
| <code>calc_provost_and_domingos_auc(y_true, y_score)</code>  | Calculate the (multi-class AUC) $M$ score from Equation (7) of Provost and Domingos (2000). |

### 2.5.3 Data structures

|                          |   |
|--------------------------|---|
| <code>fold_data</code>   | A named tuple for holding train, validation, and test datasets suitable for use in <i>sklearn</i> .       |
| <code>split_masks</code> | A named tuple for holding boolean masks for the train, validation, and test splits of a complete dataset. |

### 2.5.4 Definitions

This module contains utilities for common machine learning tasks.

In particular, this module focuses on tasks “surrounding” machine learning, such as cross-fold splitting, performance evaluation, etc. It does not include helpers for use directly in `sklearn.pipeline.Pipeline`.

`pyllars.ml_utils._calc_hand_and_till_a_value` (`y_true`: `numpy.ndarray`, `y_score`: `numpy.ndarray, i: int, j: int`) → float

Calculate the  $\hat{A}$  value in Equation (3) of<sup>1</sup>. Specifically;

$$\hat{A}(i|j) = \frac{S_i - n_i * (n_i + 1)/2}{n_i * n_j},$$

where  $n_i, n_j$  are the count of instances of the respective classes and  $S_i$  is the (base-1) sum of the ranks of class  $i$ .

#### Parameters

- **y\_true** (`numpy.ndarray`) – The true label of each instance. The labels are assumed to be encoded with integers [0, 1, ... n\_classes-1]. The respective columns in `y_score` should give the probabilities of the matching label.

<sup>1</sup> Hand, D. & Till, R. A Simple Generalisation of the Area Under the ROC Curve for Multiple Class Classification Problems. Machine Learning, 2001, 45, 171-186. [Springer link](#).

This should have shape (n\_samples,).

- **y\_score** (*numpy.ndarray*) – The score predictions for each class, e.g., from *pred\_proba*, though they are not required to be probabilities.

This should have shape (n\_samples, n\_classes).

- **{i, j}** (*int*) – The class indices

**Returns** **a\_hat** – The  $\hat{A}$  value from Equation (3) referenced above. Specifically, this is the probability that a randomly drawn member of class  $j$  will have a lower estimated score for belonging to class  $i$  than a randomly drawn member of class  $i$ .

**Return type** *float*

## References

```
pyllars.ml_utils._train_and_evaluate(estimator, X_train, y_train, X_test, y_test,
                                     target_transform, target_inverse_transform,
                                     collect_metrics, collect_metrics_kwargs,
                                     use_predict_proba)
```

Train and evaluate *estimator* on the given datasets

This function is a helper for *evaluate\_hyperparameters*. It is not intended for external use.

```
pyllars.ml_utils.calc_hand_and_till_m_score(y_true: numpy.ndarray, y_score:
                                             numpy.ndarray) → float
```

Calculate the (multi-class AUC)  $M$  score from Equation (7) of Hand and Till (2001).

This is typically taken as a good multi-class extension of the AUC score. Please see<sup>2</sup> for more details about this score in particular and<sup>3</sup> for multi-class AUC in general.

**N.B.** In case *y\_score* contains any *np.nan* values, those will be removed before calculating the  $M$  score.

**N.B.** This function *can* handle unobserved labels, except for the label with the highest index. In particular, *y\_score.shape[1] != np.max(np.unique(y\_true)) + 1* causes an error.

## Parameters

- **y\_true** (*numpy.ndarray*) – The true label of each instance. The labels are assumed to be encoded with integers [0, 1, ..., n\_classes-1]. The respective columns in *y\_score* should give the scores of the matching label.

This should have shape (n\_samples,).

- **y\_score** (*numpy.ndarray*) – The score predictions for each class, e.g., from *pred\_proba*, though they are not required to be probabilities.

This should have shape (n\_samples, n\_classes).

**Returns** **m** – The “multi-class AUC” score referenced above

**Return type** *float*

**See also:**

[`\_calc\_hand\_and\_till\_a\_value\(\)`](#) for calculating the  $\hat{A}$  value

---

<sup>2</sup> Hand, D. & Till, R. A Simple Generalisation of the Area Under the ROC Curve for Multiple Class Classification Problems. Machine Learning, 2001, 45, 171-186. [Springer link](#).

<sup>3</sup> Fawcett, T. An introduction to ROC analysis. Pattern Recognition Letters, 2006, 27, 861 - 874. [Elsevier link](#).



## References

```
pyllars.ml_utils.calc_provost_and_domingos_auc(y_true: numpy.ndarray, y_score:
                                              numpy.ndarray) → float
```

Calculate the (multi-class AUC)  $M$  score from Equation (7) of Provost and Domingos (2000).

This is typically taken as a good multi-class extension of the AUC score. Please see<sup>4</sup> for more details about this score in particular and<sup>5</sup> for multi-class AUC in general.

N.B. This function *can* handle unobserved labels, except for the label with the highest index. In particular, `y_score.shape[1] != np.max(np.unique(y_true)) + 1` causes an error.

### Parameters

- **y\_true** (*numpy.ndarray*) – The true label of each instance. The labels are assumed to be encoded with integers [0, 1, ..., n\_classes-1]. The respective columns in y\_score should give the scores of the matching label.

This should have shape (n\_samples,).

- **y\_score** (*numpy.ndarray*) – The score predictions for each class, e.g., from *pred\_proba*, though they are not required to be probabilities.

This should have shape (n\_samples, n\_classes).

**Returns** **m** – The “multi-class AUC” score referenced above

**Return type** *float*

## References

```
pyllars.ml_utils.collect_binary_classification_metrics(y_true: numpy.ndarray,
                                                       y_probab_pred:
                                                       numpy.ndarray, thresh-
old: float = 0.5,
pos_label=1, k: int =
10, include_roc_curve: bool
= True, include_pr_curve:
bool = True, prefix: str = ")
→ Dict
```

Collect various binary classification performance metrics for the predictions

### Parameters

- **y\_true** (*numpy.ndarray*) – The true class of each instance.  
This should have shape (n\_samples,).
- **y\_probab\_pred** (*numpy.ndarray*) – The score of each prediction for each instance.  
This should have shape (n\_samples, n\_classes).
- **threshold** (*float*) – The score threshold to choose “positive” predictions
- **pos\_label** (*str* or *int*) – The “positive” class for some metrics
- **k** (*int*) – The value of  $k$  to use for *precision\_at\_k*
- **include\_roc\_curve** (*bool*) – Whether to include the fpr and trp points necessary to draw a roc curve

<sup>4</sup> Provost, F. & Domingos, P. Well-Trained PETs: Improving Probability Estimation Trees. Stern School of Business, NYU, Stern School of Business, NYU, 2000. [Citeseer link](#).

<sup>5</sup> Fawcett, T. An introduction to ROC analysis. Pattern Recognition Letters, 2006, 27, 861 - 874. [Elsevier link](#).

- **include\_pr\_curve** (*bool*) – Whether to include details on the precision-recall curve
- **prefix** (*str*) – An optional prefix for the keys in the *metrics* dictionary

### Returns

**metrics** – A mapping from the metric name to the respective value. Currently, the following metrics are included:

- `sklearn.metrics.cohen_kappa_score()`
- `sklearn.metrics.hinge_loss()`
- `sklearn.metrics.matthews_corrcoef()`
- `sklearn.metrics.accuracy_score()`
- `sklearn.metrics.f1_score()` (binary)
- `sklearn.metrics.f1_score()` (macro)
- `sklearn.metrics.f1_score()` (micro)
- `sklearn.metrics.hamming_loss()`
- `sklearn.metrics.jaccard_score()`
- `sklearn.metrics.log_loss()`
- `sklearn.metrics.precision_score()` (binary)
- `sklearn.metrics.precision_score()` (macro)
- `sklearn.metrics.precision_score()` (micro)
- `sklearn.metrics.recall_score()` (binary)
- `sklearn.metrics.recall_score()` (macro)
- `sklearn.metrics.recall_score()` (micro)
- `sklearn.metrics.zero_one_loss()`
- `sklearn.metrics.average_precision_score()` (macro)
- `sklearn.metrics.average_precision_score()` (micro)
- `sklearn.metrics.roc_auc_score()` (macro)
- `sklearn.metrics.roc_auc_score()` (micro)
- `pyllars.ml_utils.precision_at_k()`
- *auprc*: area under the PR curve
- *minpse*: See [Harutyunyan et al., 2019] for details
- *roc\_{fpr, tpr, thresholds}*: `sklearn.metrics.roc_curve()`
- *pr\_{precisions, recalls, thresholds}*: `sklearn.metrics.precision_recall_curve()`

**Return type** `dict`

```
pyllars.ml_utils.collect_multiclass_classification_metrics(y_true:
                                                         numpy.ndarray,
                                                         y_score:
                                                         numpy.ndarray,
                                                         prefix: str = "") → Dict
```

Calculate various multi-class classification performance metrics

#### Parameters

- **y\_true** (*numpy.ndarray*) – The true label of each instance. The labels are assumed to be encoded with integers [0, 1, ... n\_classes-1]. The respective columns in *y\_score* should give the scores of the matching label.

This should have shape (n\_samples,).

- **y\_score** (*numpy.ndarray*) – The score predictions for each class, e.g., from ‘pred\_proba’, though they are not required to be probabilities.

This should have shape (n\_samples, n\_classes).

- **prefix** (*str*) – An optional prefix for the keys in the *metrics* dictionary

#### Returns

**metrics** – A mapping from the metric name to the respective value. Currently, the following metrics are included:

- `sklearn.metrics.cohen_kappa_score()`
- `sklearn.metrics.accuracy_score()`
- `sklearn.metrics.f1_score()` (micro)
- `sklearn.metrics.f1_score()` (macro)
- `sklearn.metrics.hamming_loss()`
- `sklearn.metrics.precision_score()` (micro)
- `sklearn.metrics.precision_score()` (macro)
- `sklearn.metrics.recall_score()` (micro)
- `sklearn.metrics.recall_score()` (macro)
- `pyllars.ml_utils.calc_hand_and_till_m_score()`
- `pyllars.ml_utils.calc_provost_and_domingos_auc()`

**Return type** `typing.Dict`

```
pyllars.ml_utils.collect_regression_metrics(y_true:      numpy.ndarray,   y_pred:
                                             numpy.ndarray, prefix: str = "") → Dict
```

Collect various regression performance metrics for the predictions

#### Parameters

- **y\_true** (*numpy.ndarray*) – The true value of each instance
- **y\_pred** (*numpy.ndarray*) – The prediction for each instance
- **prefix** (*str*) – An optional prefix for the keys in the *metrics* dictionary

#### Returns

**metrics** – A mapping from the metric name to the respective value. Currently, the following metrics are included:

- `sklearn.metrics.explained_variance_score()`
- `sklearn.metrics.mean_absolute_error()`
- `sklearn.metrics.mean_squared_error()`
- `sklearn.metrics.median_absolute_error()`
- `sklearn.metrics.r2_score()`

Return type `typing.Dict`

**class** `pyllars.ml_utils.estimators_predictions_metrics`

A named tuple for holding fit estimators, predictions on the respective datasets, and results.

**estimator\_{val, test}**

Estimators fit on the respective datasets.

Type `sklearn.base.BaseEstimators`

**predictions\_{val, test}**

Predictions of the respective models.

Type `numpy.ndarray`

**metrics\_{val, test}**

Metrics for the respective datasets.

Type `typing.Dict`

**fold\_{train, val, test}**

The identifiers of the respective folds.

Type `typing.Any`

**hyperparameters\_{\_str}**

The hyperparameters (in a string format) for training the models.

Type `typing.Optional[typing.Dict]`

**\_asdict()**

Return a new `OrderedDict` which maps field names to their values.

**classmethod** **\_make** (*iterable*, *new=<built-in method \_\_new\_\_ of type object>*, *len=<built-in function len>*)

Make a new `estimators_predictions_metrics` object from a sequence or iterable

**\_replace** (*\*\*kws*)

Return a new `estimators_predictions_metrics` object replacing specified fields with new values

**estimator\_test**

Alias for field number 1

**estimator\_val**

Alias for field number 0

**fold\_test**

Alias for field number 10

**fold\_train**

Alias for field number 8

**fold\_val**

Alias for field number 9

**hyperparameters**

Alias for field number 11

**hyperparameters\_str**  
Alias for field number 12

**metrics\_test**  
Alias for field number 7

**metrics\_val**  
Alias for field number 6

**predictions\_test**  
Alias for field number 3

**predictions\_val**  
Alias for field number 2

**true\_test**  
Alias for field number 5

**true\_val**  
Alias for field number 4

```
pyllars.ml_utils.evaluate_hyperparameters(estimator_template:
    sklearn.base.BaseEstimator, hyperparameters: Dict, validation_folds: Any, test_folds:
    Any, data: pandas.core.frame.DataFrame, collect_metrics: Callable, use_predict_proba:
    bool = False, train_folds: Optional[Any] = None, split_field: str = 'fold', target_field:
    str = 'target', target_transform: Optional[Callable] = None, target_inverse_transform:
    Optional[Callable] = None, collect_metrics_kwargs: Optional[Dict] = None, attribute_fields:
    Optional[Iterable[str]] = None, fields_to_ignore: Optional[Container[str]] = None, attributes_are_np_arrays:
    bool = False) → pyllars.ml_utils.estimators_predictions_metrics
```

Evaluate *hyperparameters* for *fold*

**N.B.** This function is not particularly efficient with creating copies of data.

This function performs the following steps:

0. Create *estimator\_val* and *estimator\_test* based on *estimator\_template* and *hyperparameters*
1. Split *data* into *train*, *val*, *test* based on *validation\_fold* and *test\_fold*
2. Transform *target\_field* using the *target\_transform* function
3. Train *estimator\_val* using *train*
4. Evaluate the trained *estimator\_val* on *val* using *collect\_metrics*
5. Train *estimator\_test* using both *train* and *val*
6. Evaluate the trained *estimator\_test* on *test* using *collect\_metrics*

#### Parameters

- **estimator\_template** (*sklearn.base.BaseEstimator*) – The template for creating the *estimator*.

- **hyperparameters** (*typing.Dict*) – The hyperparameters for the model. These should be compatible with *estimator\_template.set\_params*.
- **validation\_folds** (*typing.Any*) – The fold(s) to use for validation. The validation fold will be selected based on *isin*. If *validation\_fold* is not a container, it will be cast as one.
- **test\_folds** (*typing.Any*) – The fold(s) to use for testing. The test fold will be selected based on *isin*. If *test\_fold* is not a container, it will be cast as one.
- **data** (*pandas.DataFrame*) – The data.
- **collect\_metrics** (*typing.Callable*) – The function for evaluating the model performance. It should have at least two arguments, *y\_true* and *y\_pred*, in that order. This function will eventually return whatever this function returns.
- **use\_predict\_proba** (*bool*) – Whether to use *predict* (when *False*, the default) or *predict\_proba* on the trained model.
- **train\_folds** (*typing.Optional[typing.Any]*) – The fold(s) to use for training. If not given, the training fold will be taken as all rows in *data* which are not part of the validation or testing set.
- **split\_field** (*str*) – The name of the column with the fold identifiers
- **target\_field** (*str*) – The name of the column with the target value
- **target\_transform** (*typing.Optional[typing.Callable]*) – A function for transforming the target before training models. Example: `numpy.log1p()`
- **target\_inverse\_transform** (*typing.Optional[typing.Callable]*) – A function for transforming model predictions back to the original domain. This should be a mathematical inverse of *target\_transform*. Example: `numpy.expml()` is the inverse of `numpy.log1p()`.
- **collect\_metrics\_kwargs** (*typing.Optional[typing.Dict]*) – Additional keyword arguments for *collect\_metrics*.
- **attribute\_fields** (*typing.Optional[typing.Iterable[str]]*) – The names of the columns to use for attributes (that is, X). If *None* (default), then all columns except the *target\_field* will be used as attributes.
- **fields\_to\_ignore** (*typing.Optional[typing.Container[str]]*) – The names of the columns to ignore.
- **attributes\_are\_np\_arrays** (*bool*) – Whether to stack the values from the individual rows. This should be set to *True* when some of the columns in *attribute\_fields* contain numpy arrays.

**Returns** *estimators\_predictions\_metrics* – The fit estimators, predictions on the respective datasets, and results from *collect\_metrics*.

**Return type** *typing.NamedTuple*

```
pyllars.ml_utils.get_cv_folds(y: numpy.ndarray, num_splits: int = 10, use_stratified: bool =
                                True, shuffle: bool = True, random_state: int = 8675309) →
                                numpy.ndarray
```

Assign a split to each row based on the values of y

#### Parameters

- **y** (*numpy.ndarray*) – The target variable for each row in a data frame. This is used to determine the stratification.

- **num\_splits** (*int*) – The number of stratified splits to use
- **use\_stratified** (*bool*) – Whether to use stratified cross-validation. For example, this may be set to *False* if choosing folds for regression.
- **shuffle** (*bool*) – Whether to shuffle during the split
- **random\_state** (*int*) – The state for the random number generator

**Returns** *splits* – The split of each row

**Return type** `numpy.ndarray`

```
pyllars.ml_utils.get_fold_data(df: pandas.core.frame.DataFrame, target_field: str, m_train:
                               numpy.ndarray, m_test: numpy.ndarray, m_validation:
                               Optional[numpy.ndarray] = None, attribute_fields: Op-
                               tional[Iterable[str]] = None, fields_to_ignore: Op-
                               tional[Iterable[str]] = None, attributes_are_np_arrays: bool =
                               False) → pyllars.ml_utils.fold_data
```

Prepare a data frame for *sklearn* according to the given splits

**N.B.** This function creates copies of the data, so it is not appropriate for very large datasets.

#### Parameters

- **df** (*pandas.DataFrame*) – A data frame
- **target\_field** (*str*) – The name of the column containing the target variable
- **m\_{train,test,validation}** (*np.ndarray*) – Boolean masks indicating the training, testing, and validation set rows. If *m\_validation* is *None* (default), then no validation set will be included.
- **attribute\_fields** (*typing.Optional[typing.Iterable[str]]*) – The names of the columns to use for attributes (that is, *X*). If *None* (default), then all columns except the *target\_field* will be used as attributes.
- **fields\_to\_ignore** (*typing.Optional[typing.Container[str]]*) – The names of the columns to ignore.
- **attributes\_are\_np\_arrays** (*bool*) – Whether to stack the values from the individual rows. This should be set to *True* when some of the columns in *attribute\_fields* contain numpy arrays.

**Returns** *fold\_data* – A named tuple with the given splits

**Return type** `pyllars.ml_utils.fold_data`

```
pyllars.ml_utils.get_train_val_test_splits(df: pandas.core.frame.DataFrame, train-
                                           ing_splits: Optional[Set] = None, valida-
                                           tion_splits: Optional[Set] = None, test_splits:
                                           Optional[Set] = None, split_field: str =
                                           'split') → pyllars.ml_utils.split_masks
```

Get the appropriate training, validation, and testing split masks

The *split\_field* column in *df* is used to assign each row to a particular split. Then, the splits specified in the parameters are assigned as indicated.

By default, all splits not in *validation\_splits* and *test\_splits* are assumed to belong to the training set. Thus, unless a particular training set is given, the returned masks will cover the entire dataset.

This function does not check whether the different splits overlap. So care should be taken, especially if specifying the training splits explicitly.

It is not necessary that the *split\_field* values are numeric. They must be compatible with *isin*, however.

**Parameters**

- **df** (*pandas.DataFrame*) – A data frame. It must contain a column named *split\_field*, but it is not otherwise validated.
- **training\_splits** (*typing.Optional[typing.Set]*) – The splits to use for the training set. By default, anything not in the *validation\_splits* or *test\_splits* will be placed in the training set.

If given, this container must be compatible with *isin*. Otherwise, it will be wrapped in a set.

- **{validation,test}\_splits** (*typing.Optional[typing.Set]*) – The splits to use for the validation and test sets, respectively.

If given, this container must be compatible with *isin*. Otherwise, it will be wrapped in a set.

- **split\_field** (*str*) – The name of the column indicating the split for each row.

**Returns** **split\_masks** – Masks for the respective sets. *True* positions indicate the rows which belong to the respective sets. All three masks are always returned, but a mask may be always *False* if the given split does not contain any rows.

**Return type** *pyllars.ml\_utils.split\_masks*

```
pyllars.ml_utils.precision_at_k(y_true, y_score, k=10, pos_label=1)
```

Precision at rank k

This code was adapted from this gist: <https://gist.github.com/mblondel/7337391>

**Parameters**

- **y\_true** (*array-like, shape = [n\_samples]*) – Ground truth (true relevance labels).
- **y\_score** (*array-like, shape = [n\_samples]*) – Predicted scores.
- **k** (*int*) – Rank.
- **pos\_label** (*int*) – The label for “positive” instances

**Returns** **precision @k**

**Return type** *float*

```
class pyllars.ml_utils.fold_data
```

A named tuple for holding train, validation, and test datasets suitable for use in *sklearn*.

This class can be more convenient than *pyllars.ml\_utils.split\_masks* for modest-sized datasets.

**X\_{train,test,validation}**

The X data (features) for the respective dataset splits

**Type** *numpy.ndarray*

**y\_{train,test,validation}**

The y data (target) for the respective dataset splits

**Type** *numpy.ndarray*

**{train\_test,validation}\_indices**

The row indices from the original dataset of the respective dataset splits

**Type** *numpy.ndarray*

```
class pyllars.ml_utils.split_masks
```

A named tuple for holding boolean masks for the train, validation, and test splits of a complete dataset.



These masks can be used to index `numpy.ndarray` or `pandas.DataFrame` objects to extract the relevant dataset split for `sklearn`. This class can be more appropriate than `pyllars.ml_utils.fold_data` for large objects since it avoids any copies of the data.

#### **training, test, validation**

Boolean masks for the respective dataset splits

Type `numpy.ndarray`

## 2.6 Matplotlib utilities

This module contains a number of helper functions for matplotlib. For details about various arguments, such as allowed key word arguments and how they will be interpreted, please consult the appropriate parts of the matplotlib documentation: \* **Lines:** [https://matplotlib.org/api/\\_as\\_gen/matplotlib.lines.Line2D.html#matplotlib.lines.Line2D](https://matplotlib.org/api/_as_gen/matplotlib.lines.Line2D.html#matplotlib.lines.Line2D) \* **Patches:** [https://matplotlib.org/api/\\_as\\_gen/matplotlib.patches.Patch.html#matplotlib.patches.Patch](https://matplotlib.org/api/_as_gen/matplotlib.patches.Patch.html#matplotlib.patches.Patch) \* **Scatter plots:** [https://matplotlib.org/api/\\_as\\_gen/matplotlib.pyplot.scatter.html#matplotlib.pyplot.scatter](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html#matplotlib.pyplot.scatter) \* **Text:** [https://matplotlib.org/api/text\\_api.html#matplotlib.text.Text](https://matplotlib.org/api/text_api.html#matplotlib.text.Text)

### 2.6.1 Adjusting axis properties

#### Fonts

|   |  |
|---|--|
| <code>set_legend_title_fontsize(ax, fontsize, str]</code>     | Set the font size of the title of the legend.  |
| <code>set_legend_fontsize(ax, fontsize, str]</code>           | Set the font size of the items of the legend.  |
| <code>set_title_fontsize(ax, fontsize, str]</code>            | Set the font size of the title of the axis.  |
| <code>set_label_fontsize(ax, fontsize, str], axis)</code>     | Set the font size of the labels of the axis.   |
| <code>set_ticklabels_fontsize(ax, fontsize, str], ...)</code> | Set the font size of the tick labels :param ax: The axis :type ax: matplotlib.axes.Axes :param fontsize: The size of the ticklabels :type fontsize: int, or a str recognized by matplotlib :param {axis,which}: Values passed to <code>matplotlib.axes.Axes.tick_params()</code> . |
| <code>set_ticklabel_rotation(ax, rotation, str], ...)</code>  | Set the rotation of the tick labels :param ax: The axis :type ax: matplotlib.axes.Axes :param rotation: The rotation of the labels :type rotation: int, or a string matplotlib recognizes :param {axis,which}: Values passed to <code>matplotlib.pyplot.setp()</code> .            |

#### Axes

|   |  |
|---|--|
| <code>center_splines(ax)</code>                             | Places the splines of <code>ax</code> in the center of the plot.                           |
| <code>hide_first_y_tick_label(ax)</code>                    | Hide the first tick label on the y-axis :param ax: The axis :type ax: matplotlib.axes.Axes |
| <code>hide_tick_labels_by_text(ax, to_remove_x, ...)</code> | Hide tick labels which match the given values.   |
| <code>hide_tick_labels_by_index(ax, keep_x, ...)</code>     | Hide the tick labels on both axes.   |

### 2.6.2 Creating standard plots

|   |  |
|---|--|
| <code>plot_simple_bar_chart(bars, ax, labels, ...)</code>     | Plot a simple bar chart based on the values in <i>bars</i>   |
| <code>plot_simple_scatter(x, y, ax, equal_aspect, ...)</code> | Plot a simple scatter plot of <i>x</i> vs.   |
| <code>plot_stacked_bar_graph(ax, data[, colors, ...])</code>  | Create a stacked bar plot with the given characteristics.  |
| <code>plot_sorted_values(values, ymin, ymax, ax, ...)</code>  | Sort <i>values</i> and plot them :param values: The values to plot :type values: typing.Sequence[float] :param y_{min,max}: The min and max values for the y-axis. |

## 2.6.3 Plotting standard machine learning and statistical results

|   |   |
|---|---|
| <code>plot_binary_prediction_scores(y_scores, ...)</code>     | Plot separate lines for the scores of the positives and negatives |
| <code>plot_confusion_matrix(confusion_matrix, ax, ...)</code> | Plot the given confusion matrix                                   |
| <code>plot_roc_curve(tpr, fpr, auc, show_points, ...)</code>  | Plot the ROC curve for the given <i>fpr</i> and <i>tpr</i> values |
| <code>plot_trend_line(x, intercept, slope, power, ...)</code> | Draw the trend line implied by the given coefficients.            |
| <code>plot_venn_diagram(sets, Sequence[, ax, ...])</code>     | Wrap the matplotlib_venn package.                                 |

## 2.6.4 Other helpers

|   |  |
|---|--|
| <code>add_fontsizes_to_args(args, ...)</code>               | Add reasonable default fontsize values to <i>args</i>  |
| <code>draw_rectangle(ax, base_x, base_y, width, ...)</code> | Draw a rectangle at the given <i>x</i> and <i>y</i> coordinates.                                   |
| <code>get_diff_counts(data_np)</code>                       | This function extracts the differential counts necessary for visualization with stacked_bar_graph. |

## 2.6.5 Definitions

This module contains a number of helper functions for matplotlib. For details about various arguments, such as allowed key word arguments and how they will be interpreted, please consult the appropriate parts of the matplotlib documentation: \* **Lines**: [https://matplotlib.org/api/\\_as\\_gen/matplotlib.lines.Line2D.html#matplotlib.lines.Line2D](https://matplotlib.org/api/_as_gen/matplotlib.lines.Line2D.html#matplotlib.lines.Line2D) \* **Patches**: [https://matplotlib.org/api/\\_as\\_gen/matplotlib.patches.Patch.html#matplotlib.patches.Patch](https://matplotlib.org/api/_as_gen/matplotlib.patches.Patch.html#matplotlib.patches.Patch) \* **Scatter plots**: [https://matplotlib.org/api/\\_as\\_gen/matplotlib.pyplot.scatter.html#matplotlib.pyplot.scatter](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html#matplotlib.pyplot.scatter) \* **Text**: [https://matplotlib.org/api/text\\_api.html#matplotlib.text.Text](https://matplotlib.org/api/text_api.html#matplotlib.text.Text)

```
pyllars.mpl_utils.VALID_AXIS_VALUES = {'both', 'x', 'y'}
    Valid axis values
```

```
pyllars.mpl_utils.VALID_WHICH_VALUES = {'both', 'major', 'minor'}
    Valid which values
```

```
pyllars.mpl_utils.X_AXIS_VALUES = {'both', 'x'}
    axis choices which affect the X axis
```

```
pyllars.mpl_utils.Y_AXIS_VALUES = {'both', 'y'}
    axis choices which affect the Y axis
```

```
pyllars.mpl_utils._get_fig_ax(ax: Optional[matplotlib.axes._axes.Axes])
    Grab a figure and axis from ax, or create a new one
```

```
pyllars.mpl_utils.add_fontsizes_to_args(args: argparse.Namespace, legend_title_fontsize:
    int = 12, legend_fontsize: int = 10, title_fontsize:
    int = 20, label_fontsize: int = 15, ticklabels_fontsize: int = 10)
    Add reasonable default fontsize values to args
```

`pyllars.mpl_utils.center_splines` (*ax: matplotlib.axes.\_axes.Axes*) → None

Places the splines of *ax* in the center of the plot.

This is useful for things like scatter plots where (0,0) should be in the center of the plot. :param ax: The axis  
:type ax: matplotlib.axes.Axes

### Returns

**Return type** None, but the splines are updated

`pyllars.mpl_utils.draw_rectangle` (*ax: matplotlib.axes.\_axes.Axes, base\_x: float, base\_y: float, width: float, height: float, center\_x: bool = False, center\_y: bool = False, \*\*kwargs*) → Tuple[matplotlib.figure.Figure, matplotlib.axes.\_axes.Axes]

Draw a rectangle at the given x and y coordinates.

Optionally, these can be adjusted such that they are the respective centers rather than edge values. :param ax: The axis on which the rectangle will be drawn :type ax: matplotlib.axes.Axes :param base\_{x,y}: The base x and y coordinates :type base\_{x,y}: float :param {width,height}: The width (change in x) and height (change in y) of the rectangle :type {width,height}: float :param center\_{x,y}: Whether to adjust the x and y coordinates such that they become the

center rather than lower left. In particular, if *center\_x* is *True*, then *base\_x* will be shifted left by *width/2*; likewise, if *center\_y* is *True*, then *base\_y* will be shifted down by *height/2*.

**Parameters** **\*\*kwargs** (*key=value pairs*) – Additional keywords are passed to the `patches.Rectangle` constructor. Please see the matplotlib documentation for more details: [https://matplotlib.org/api/\\_as\\_gen/matplotlib.patches.Rectangle.html](https://matplotlib.org/api/_as_gen/matplotlib.patches.Rectangle.html)

### Returns

- **fig** (*matplotlib.figure.Figure*) – The figure on which the rectangle was drawn
- **ax** (*matplotlib.axes.Axes*) – The axis on which the rectangle was drawn

`pyllars.mpl_utils.get_diff_counts` (*data\_np*)

This function extracts the differential counts necessary for visualization with `stacked_bar_graph`. It assumes the counts for each bar are given as a separate row in the numpy 2-d array. Within the rows, the counts are ordered in ascending order. That is, the first column contains the smallest count, the second column contains the next-smallest count, etc.

For example, if the columns represent some sort of filtering approach, then the last column would contain the unfiltered count, the next-to-last column would give the count after the first round of filtering, etc.

`pyllars.mpl_utils.hide_first_y_tick_label` (*ax: matplotlib.axes.\_axes.Axes*) → None

Hide the first tick label on the y-axis :param ax: The axis :type ax: matplotlib.axes.Axes

### Returns

**Return type** None, but the tick label is hidden

`pyllars.mpl_utils.hide_tick_labels` (*ax: matplotlib.axes.\_axes.Axes, axis: str = 'both'*) → None

Hide the tick labels on the specified axes.

Optionally, some can be preserved. :param ax: The axis :type ax: matplotlib.axes.Axes :param axis: Axis of the tick labels to hide :type axis: str in {both, x, y}

### Returns

**Return type** None, but the tick labels of the axis are removed, as specified

```
pyllars.mpl_utils.hide_tick_labels_by_index(ax: matplotlib.axes._axes.Axes, keep_x: Collection = {}, keep_y: Collection = {}, axis: str = 'both') → None
```

Hide the tick labels on both axes.

Optionally, some can be preserved. :param ax: The axis :type ax: matplotlib.axes.Axes :param keep\_{x,y}: The indices of any x-axis ticks to keep. The numbers are passed directly

as indices to the “ticks” arrays.

**Parameters** **axis** (str in {both, x, y}) – Axis of the tick labels to hide

**Returns**

**Return type** **None**, but the tick labels of the axis are removed, as specified

```
pyllars.mpl_utils.hide_tick_labels_by_text(ax: matplotlib.axes._axes.Axes, to_remove_x: Collection = {}, to_remove_y: Collection = {}) → None
```

Hide tick labels which match the given values. :param ax: The axis :type ax: matplotlib.axes.Axes :param to\_remove\_{x,y}: The values to remove :type to\_remove\_{x,y}: typing.Collection[str]

**Returns**

**Return type** **None**, but the specified tick labels are hidden

```
pyllars.mpl_utils.plot_binary_prediction_scores(y_scores: Sequence[float], y_true: Sequence[int], positive_label: int = 1, positive_line_color='g', negative_line_color='r', line_kwargs: Mapping = {}, positive_line_kwargs: Mapping = {}, negative_line_kwargs: Mapping = {}, title: Optional[str] = None, ylabel: Optional[str] = 'Score', xlabel: Optional[str] = 'Instance', title_font_size: int = 20, label_font_size: int = 15, ticklabels_font_size: int = 15, ax: Optional[matplotlib.axes._axes.Axes] = None) → Tuple[matplotlib.figure.Figure, matplotlib.axes._axes.Axes]
```

Plot separate lines for the scores of the positives and negatives

**Parameters**

- **y\_scores** (*typing.Sequence[float]*) – The predicted scores of the positive class. For example, this may be found using something like: `y_scores = y_proba_pred[:,1]` for probabilistic predictions from most *sklearn* classifiers.
- **y\_true** (*typing.Sequence[int]*) – The ground truth labels
- **positive\_label** (*int*) – The value for the “positive” class
- **{positive,negative}\_line\_color** (*color*) – Values to use for the color of the respective lines. These can be anything which *matplotlib.plot* can interpret. These values have precedent over the other *kwargs* parameters.
- **line\_kwargs** (*typing.Mapping*) – Other keyword arguments passed through to *plot* for both lines.

- **{positive,negative}\_line\_kwargs** (*typing.Mapping*) – Other keyword arguments pass through to *plot* for only the respective line.

These values have precedent over *line\_kwargs*.

- **title** (*typing.Optional[str]*) – If given, the title of the axis is set to this value
- **{y,x}label** (*typing.Optional[str]*) – Text for the respective labels
- **{title,label,ticklabels}\_font\_size** (*int*) – The font sizes for the respective elements.
- **ax** (*typing.Optional[matplotlib.axes.Axes]*) – The axis. If not given, then one will be created.

#### Returns

- **fig** (*matplotlib.figure.Figure*) – The figure on which the scores lines were plotted
- **ax** (*matplotlib.axes.Axes*) – The axis on which the score lines were plotted

```
pyllars.mpl_utils.plot_confusion_matrix(confusion_matrix: numpy.ndarray, ax: Optional[matplotlib.axes.Axes] = None,
                                         show_cell_labels: bool = True, show_colorbar: bool = True, title: Optional[str] = 'Confusion matrix',
                                         cmap: matplotlib.colors.Colormap = <matplotlib.colors.LinearSegmentedColormap object>,
                                         true_tick_labels: Optional[Sequence[str]] = None, predicted_tick_labels: Optional[Sequence[str]] = None,
                                         ylabel: Optional[str] = 'True labels', xlabel: Optional[str] = 'Predicted labels', title_font_size: int = 20,
                                         label_font_size: int = 15, true_tick_rotation: Union[str, int, None] = None, predicted_tick_rotation: Union[str, int, None] = None,
                                         out: Optional[str] = None) → Tuple[matplotlib.figure.Figure, matplotlib.axes.Axes]
```

Plot the given confusion matrix

#### Parameters

- **confusion\_matrix** (*numpy.ndarray*) – A 2-d array, presumably from *sklearn.metrics.confusion\_matrix()* or something similar. The rows (Y axis) are the “true” classes while the columns (X axis) are the “predicted” classes.
- **ax** (*typing.Optional[matplotlib.axes.Axes]*) – The axis. If not given, then one will be created.
- **show\_cell\_labels** (*bool*) – Whether to show the values within each cell
- **show\_colorbar** (*bool*) – Whether to show a color bar
- **title** (*typing.Optional[str]*) – If given, the title of the axis is set to this value
- **cmap** (*matplotlib.colors.Colormap*) – A colormap to determine the cell colors
- **{true,predicted}\_tick\_labels** (*typing.Optional[typing.Sequence[str]]*) – Text for the Y (true) and X (predicted) axis, respectively
- **{y,x}label** (*typing.Optional[str]*) – Text for the respective labels
- **{title,label}\_font\_size** (*int*) – The font sizes for the respective elements. The class labels (on the tick marks) use the *label\_font\_size*.

- **{true,predicted}\_tick\_rotation** (*typing.Optional[IntOrString]*) – The rotation arguments for the respective tick labels. Please see the matplotlib text documentation ([https://matplotlib.org/api/text\\_api.html#matplotlib.text.Text](https://matplotlib.org/api/text_api.html#matplotlib.text.Text)) for more details.
- **out** (*typing.Optional[str]*) – If given, the plot will be saved to this file.

### Returns

- **fig** (*matplotlib.figure.Figure*) – The figure on which the confusion matrix was plotted
- **ax** (*matplotlib.axes.Axes*) – The axis on which the confusion matrix was plotted

```
pyllars.mpl_utils.plot_mean_roc_curve (tprs: Sequence[Sequence[float]], fprs: Sequence[Sequence[float]], aucs: Optional[float] = None, label_note: Optional[str] = None, line_style: Mapping = {'alpha': 0.8, 'c': 'b', 'lw': 2}, fill_style: Mapping = {'alpha': 0.2, 'color': 'grey'}, show_xy_line: bool = True, xy_line_kwargs: Mapping = {'color': 'r', 'ls': '-', 'lw': 2}, ax: Optional[matplotlib.axes._axes.Axes] = None, title: Optional[str] = None, xlabel: Optional[str] = 'False positive rate', ylabel: Optional[str] = 'True positive rate', title_font_size: int = 25, label_font_size: int = 20, ticklabels_font_size: int = 20) → Tuple[matplotlib.figure.Figure, matplotlib.axes._axes.Axes]
```

Plot the mean plus/minus the standard deviation of the given ROC curves

### Parameters

- **tprs** (*typing.Sequence[typing.Sequence[float]]*) – The true positive rate at each threshold
- **fprs** (*typing.Sequence[typing.Sequence[float]]*) – The false positive rate at each threshold
- **aucs** (*typing.Optional[float]*) – The calculated area under the ROC curve
- **label\_note** (*typing.Optional[str]*) – A prefix for the label in the legend for this line.
- **{line,fill}\_style** (*typing.Mapping*) – Keyword arguments for plotting the line and *fill\_between*, respectively. Please see the mpl docs for more details.
- **show\_xy\_line** (*bool*) – Whether to draw the y=x line
- **xy\_line\_kwargs** (*typing.Mapping*) – Keyword arguments for plotting the x=y line.
- **title** (*typing.Optional[str]*) – If given, the title of the axis is set to this value
- **{x,y}label** (*typing.Optional[str]*) – Text for the respective labels
- **{title,label,ticklabels}\_font\_size** (*int*) – The font sizes for the respective elements
- **ax** (*typing.Optional[matplotlib.axes.Axes]*) – The axis. If not given, then one will be created.

### Returns

- **fig** (*matplotlib.figure.Figure*) – The figure on which the ROC curves were plotted
- **ax** (*matplotlib.axes.Axes*) – The axis on which the ROC curves were plotted

```
pyllars.mpl_utils.plot_roc_curve(tpr: Sequence[Sequence[float]], fpr: Sequence[Sequence[float]], auc: Optional[float] = None, show_points: bool = True, ax: Optional[matplotlib.axes.Axes] = None, method_names: Optional[Sequence[str]] = None, out: Optional[str] = None, line_colors: Optional[Sequence] = None, point_colors: Optional[Sequence] = None, alphas: Optional[Sequence[float]] = None, line_kwargs: Optional[Mapping] = None, point_kwargs: Optional[Mapping] = None, title: Optional[str] = 'Receiver operating characteristic curves', xlabel: Optional[str] = 'False positive rate', ylabel: Optional[str] = 'True positive rate', title_font_size: int = 20, label_font_size: int = 15, ticklabels_font_size: int = 15) → Tuple[matplotlib.figure.Figure, matplotlib.axes.Axes]
```

Plot the ROC curve for the given *fpr* and *tpr* values

Currently, this function plots multiple ROC curves.

Optionally, add a note of the *auc*.

### Parameters

- **tpr** (*typing.Sequence[typing.Sequence[float]]*) – The true positive rate at each threshold
- **fpr** (*typing.Sequence[typing.Sequence[float]]*) – The false positive rate at each threshold
- **auc** (*typing.Optional[float]*) – The calculated area under the ROC curve
- **show\_points** (*bool*) – Whether to plot points at each threshold
- **ax** (*typing.Optional[matplotlib.axes.Axes]*) – The axis. If not given, then one will be created.
- **method\_names** (*typing.Optional[typing.Sequence[str]]*) – The name of each method
- **out** (*typing.Optional[str]*) – If given, the plot will be saved to this file.
- **line\_colors** (*typing.Optional[typing.Sequence[color]]*) – The color of each ROC line
- **point\_colors** (*typing.Optional[typing.Sequence[color]]*) – The color of the points on each each ROC line
- **alphas** (*typing.Optional[typing.Sequence[float]]*) – An alpha value for each method
- **{line,point}\_kwargs** (*typing.Optional[typing.Mapping]*) – Additional keyword arguments for the respective elements
- **title** (*typing.Optional[str]*) – If given, the title of the axis is set to this value
- **{x,y}label** (*typing.Optional[str]*) – Text for the respective labels
- **{title,label,ticklabels}\_font\_size** (*int*) – The font sizes for the respective elements

### Returns

- **fig** (*matplotlib.figure.Figure*) – The figure on which the ROC curves were plotted
- **ax** (*matplotlib.axes.Axes*) – The axis on which the ROC curves were plotted



```
pyllars.mpl_utils.plot_simple_bar_chart (bars: Sequence[Sequence[float]], ax: Optional[matplotlib.axes._axes.Axes] = None, labels: Optional[Sequence[str]] = None, colors: Union[matplotlib.colors.Colormap, Sequence, int] = <matplotlib.colors.LinearSegmentedColormap object>, xticklabels: Union[str, Sequence[str], None] = 'default', xticklabels_rotation: Union[int, str] = 'vertical', xlabel: Optional[str] = None, ylabel: Optional[str] = None, spacing: float = 0, ymin: Optional[float] = None, ymax: Optional[float] = None, use_log_scale: bool = False, hide_first_tick: bool = True, show_legend: bool = False, title: Optional[str] = None, tick_fontsize: int = 12, label_fontsize: int = 12, legend_fontsize: int = 12, title_fontsize: int = 12, tick_offset: float = 0.5)
```

Plot a simple bar chart based on the values in *bars*

#### Parameters

- **bars** (*typing.Sequence[typing.Sequence[float]]*) – The heights of each bar. The “outer” sequence corresponds to each clustered group of bars, while the “inner” sequence gives the heights of each bar within the group.

As a data science example, the “outer” groups may correspond to different datasets, while the “inner” group corresponds to different methods.

- **ax** (*typing.Optional[matplotlib.axes.Axes]*) – The axis. If not given, then one will be created.
- **labels** (*typing.Optional[typing.Sequence[str]]*) – The label for each “outer” group in *bars*
- **colors** (*BarChartColorOptions*) – The colors of the bars for each “inner” group. The options and their interpretations are:
  - color map : the color of each bar will be taken as equi-distant colors sampled from the map. For example, if there are three bars in the inner group, then the colors will be: *colors(0.0)*, *colors(0.5)*, and *colors(1.0)*.
  - sequence of colors : the color of each bar will be taken from the respective position in the sequence.
  - scalar (int or str) : all bars will use this color
- **xticklabels** (*typing.Optional[typing.Union[str, typing.Sequence[str]]]*) – The tick labels for the “outer” groups. The options and their interpretations are:
  - None : no tick labels will be shown
  - “default” : the tick labels will be the numeric tick positions
  - sequence of strings : the tick labels will be the respective strings
- **xticklabels\_rotation** (*typing.Union[str, int]*) – The rotation for the *xticklabels*. If a string is given, it should be something which matplotlib can interpret as a rotation.
- **{x,y}label** (*typing.Optional[str]*) – Labels for the respective axes
- **spacing** (*float*) – The distance on the x axis between the “outer” groups.



- **y{min,max}** (*typing.Optional[float]*) – The min and max for the y axis. If not given, the default min is 0 (or 1 if a logarithmic scale is used, see option below), and the default max is 2 times the height of the highest bar in any group.
- **use\_log\_scale** (*bool*) – Whether to use a normal or logarithmic scale for the y axis
- **hide\_first\_ytick** (*bool*) – Whether to hide the first tick mark and label on the y axis. Typically, the first tick mark is either 0 or 1 (depending on the scale of the y axis). This can be distracting to see, so the default is to hide it.
- **show\_legend** (*bool*) – Whether to show the legend
- **title** (*typing.Optional[str]*) – A title for the axis
- **{tick,label,legend,title}\_fontsize** (*int*) – The font size for the respective elements
- **tick\_offset** (*float*) – The offset of the tick mark and label for the outer groups on the x axis

#### Returns

- **fig** (*matplotlib.figure.Figure*) – The figure on which the bars were plotted
- **ax** (*matplotlib.axes.Axes*) – The axis on which the bars were plotted

```
pyllars.mpl_utils.plot_simple_scatter(x: Sequence[float], y: Sequence[float], ax:
                                     Optional[matplotlib.axes._axes.Axes] = None,
                                     equal_aspect: bool = True, set_lim: bool = True,
                                     show_y_x_line: bool = True, xy_line_kwargs: dict
                                     = {}, **kwargs) → Tuple[matplotlib.figure.Figure,
                                     matplotlib.axes._axes.Axes]
```

Plot a simple scatter plot of *x* vs. *y* on *ax*

See the matplotlib documentation for more keyword arguments and details: [https://matplotlib.org/api/\\_as\\_gen/matplotlib.pyplot.scatter.html#matplotlib.pyplot.scatter](https://matplotlib.org/api/_as_gen/matplotlib.pyplot.scatter.html#matplotlib.pyplot.scatter)

#### Parameters

- **{x,y}** (*typing.Sequence[float]*) – The values to plot
- **ax** (*typing.Optional[matplotlib.axes.Axes]*) – The axis. If not given, then one will be created.
- **equal\_aspect** (*bool*) – Whether to set the aspect of the axis to *equal*
- **set\_lim** (*bool*) – Whether to automatically set the min and max axis limits
- **show\_y\_x\_line** (*bool*) – Whether to draw the *y=x* line. This will look weird if *set\_lim* is False.
- **xy\_line\_kwargs** (*typing.Mapping*) – keyword arguments for plotting the *y=x* line, if it plotting
- **\*\*kwargs** (*<key>=<value> pairs*) – Additional keyword arguments to pass to the scatter function. Some useful keyword arguments are:
  - *label* : the label for a legend
  - *marker* : [https://matplotlib.org/examples/lines\\_bars\\_and\\_markers/marker\\_reference.html](https://matplotlib.org/examples/lines_bars_and_markers/marker_reference.html)

#### Returns

- **fig** (*matplotlib.figure.Figure*) – The figure on which the scatter points were plotted
- **ax** (*matplotlib.axes.Axes*) – The axis on which the scatter points were plotted

```
pyllars.mpl_utils.plot_sorted_values (values: Sequence[float], ymin: Optional[float]
                                     = None, ymax: Optional[float] = None,
                                     ax: Optional[matplotlib.axes._axes.Axes] =
                                     None, scale_x: bool = False, **kwargs)
                                     → Tuple[matplotlib.figure.Figure, mat-
                                     plotlib.axes._axes.Axes]
```

Sort *values* and plot them :param values: The values to plot :type values: typing.Sequence[float] :param y\_{min,max}: The min and max values for the y-axis. If not given, then these

default to the minimum and maximum values in the list.

### Parameters

- **scale\_x** (*bool*) – If True, then the *x* values will be equally-spaced between 0 and 1. Otherwise, they will be the values 0 to len(values)
- **ax** (*typing.Optional[matplotlib.axes.Axes]*) – An axis for plotting. If this is not given, then a figure and axis will be created.
- **\*\*kwargs** (<key>=<value> pairs) – Additional keyword arguments to pass to the plot function. Some useful keyword arguments are:
  - *label* : the label for a legend
  - *lw* : the line width
  - *ls* : [https://matplotlib.org/gallery/lines\\_bars\\_and\\_markers/line\\_styles\\_reference.html](https://matplotlib.org/gallery/lines_bars_and_markers/line_styles_reference.html)
  - *marker* : [https://matplotlib.org/examples/lines\\_bars\\_and\\_markers/marker\\_reference.html](https://matplotlib.org/examples/lines_bars_and_markers/marker_reference.html)

### Returns

- **fig** (*matplotlib.figure.Figure*) – The Figure associated with *ax*, or a new Figure
- **ax** (*matplotlib.axes.Axes*) – Either *ax* or a new Axis

```
pyllars.mpl_utils.plot_stacked_bar_graph (ax, data, colors=
                                          <matplotlib.colors.LinearSegmentedColormap
                                          object>, x_tick_labels=None,
                                          stack_labels=None, y_ticks=None,
                                          y_tick_labels=None, hide_first_ytick=True,
                                          edge_colors=None, showFirst=-1, scale=False,
                                          widths=None, heights=None, y_title=None,
                                          x_title=None, gap=0.0, end_gaps=False,
                                          show_legend=True, legend_loc='best', leg-
                                          end_bbox_to_anchor=None, legend_ncol=-1,
                                          log=False, font_size=8, label_font_size=12,
                                          legend_font_size=8)
```

Create a stacked bar plot with the given characteristics.

This code is adapted from code by Michael Imelfort.

```
pyllars.mpl_utils.plot_trend_line (x: Sequence[float], intercept: float, slope: float,
                                   power: float, ax: Optional[matplotlib.axes._axes.Axes] =
                                   None, **kwargs) → Tuple[matplotlib.figure.Figure, mat-
                                   plotlib.axes._axes.Axes]
```

Draw the trend line implied by the given coefficients. :param x: The points at which the function will be evaluated and where

the line will be drawn

**Parameters**

- **{intercept, slope, power}** (*float*) – The coefficients of the trend line. Presumably, these come from `pyllars.stats_utils.fit_with_least_squares()` or something similar.
- **ax** (*typing.Optional[matplotlib.axes.Axes]*) – The axis. If not given, then one will be created.
- **\*\*kwargs** (*<key>=<value> pairs*) – Keyword arguments to pass to the `ax.plot` function (color, etc.). Please consult the matplotlib documentation for more details: [https://matplotlib.org/api/\\_as\\_gen/matplotlib.lines.Line2D.html#matplotlib.lines.Line2D](https://matplotlib.org/api/_as_gen/matplotlib.lines.Line2D.html#matplotlib.lines.Line2D)

**Returns**

- **fig** (*matplotlib.figure.Figure*) – The figure on which the trend line was plotted
- **ax** (*matplotlib.axes.Axes*) – The axis on which the trend line was plotted

```
pyllars.mpl_utils.plot_venn_diagram(sets: Union[Mapping, Sequence], ax: Optional[matplotlib.axes._axes.Axes] = None, set_labels: Optional[Sequence[str]] = None, weighted: bool = False, use_sci_notation: bool = False, sci_notation_limit: float = 999, labels_fontsize: int = 14, counts_fontsize: int = 12) → matplotlib_venn._common.VennDiagram
```

Wrap the `matplotlib_venn` package.

Please consult the package documentation for more details: <https://github.com/konstantint/matplotlib-venn>

**N.B.** Unlike most of the other high-level plotting helpers, this function returns the venn diagram object rather than the figure and axis objects.

**Parameters**

- **set** (*typing.Union[typing.Mapping, typing.Sequence]*) – If a dictionary, it must follow the conventions of `matplotlib_venn`. If a dictionary is given, the number of sets will be guessed based on the length of one of the entries.  
If a sequence is given, then it must be of length two or three.  
The type of venn diagram will be based on the number of sets.
- **ax** (*typing.Optional[matplotlib.axes.Axes]*) – The axis. If not given, then one will be created.
- **set\_labels** (*typing.Optional[typing.Sequence[str]]*) – The label for each set. The order of the labels must match the order of the sets.
- **weighted** (*bool*) – Whether the diagram is weighted (in which the size of the circles in the venn diagram are based on the number of elements) or unweighted (in which all circles are the same size)
- **use\_sci\_notation** (*bool*) – Whether to convert numbers to scientific notation
- **sci\_notation\_limit** (*float*) – The maximum number to show before switching to scientific notation
- **{labels, counts}\_fontsize** (*int*) – The respective font sizes

**Returns** `venn_diagram` – The venn diagram

**Return type** `matplotlib_venn._common.VennDiagram`

`pyllars.mpl_utils.set_label_fontsize` (*ax: matplotlib.axes.\_axes.Axes, fontsize: Union[int, str], axis: str = 'both'*) → None

Set the font size of the labels of the axis. :param ax: The axis :type ax: matplotlib.axes.Axes :param fontsize: The size of the label font :type fontsize: int, or a str recognized by matplotlib :param axis: Which label(s) to update :type axis: str in {both, x, y}

#### Returns

**Return type** None, but the respective label font sizes are updated

`pyllars.mpl_utils.set_legend_fontsize` (*ax: matplotlib.axes.\_axes.Axes, fontsize: Union[int, str]*) → None

Set the font size of the items of the legend. :param ax: The axis :type ax: matplotlib.axes.Axes :param fontsize: The size of the legend text :type fontsize: int, or a str recognized by matplotlib

#### Returns

**Return type** None, but the legend text fontsize is updated

`pyllars.mpl_utils.set_legend_title_fontsize` (*ax: matplotlib.axes.\_axes.Axes, fontsize: Union[int, str]*) → None

Set the font size of the title of the legend. :param ax: The axis :type ax: matplotlib.axes.Axes :param fontsize: The size of the legend title :type fontsize: int, or a str recognized by matplotlib

#### Returns

**Return type** None, but the legend title fontsize is updated

`pyllars.mpl_utils.set_ticklabel_rotation` (*ax: matplotlib.axes.\_axes.Axes, rotation: Union[int, str], axis: str = 'x', which: str = 'both'*)

Set the rotation of the tick labels :param ax: The axis :type ax: matplotlib.axes.Axes :param rotation: The rotation of the labels :type rotation: int, or a string matplotlib recognizes :param {axis, which}: Values passed to `matplotlib.pyplot.setp()`. Please see

the matplotlib documentation for more details.

#### Returns

**Return type** None, but the ticklabels are rotated

`pyllars.mpl_utils.set_ticklabels_fontsize` (*ax: matplotlib.axes.\_axes.Axes, fontsize: Union[int, str], axis: str = 'both', which: str = 'major'*)

Set the font size of the tick labels :param ax: The axis :type ax: matplotlib.axes.Axes :param fontsize: The size of the ticklabels :type fontsize: int, or a str recognized by matplotlib :param {axis, which}: Values passed to `matplotlib.axes.Axes.tick_params()`. Please see

the matplotlib documentation for more details.

#### Returns

**Return type** None, but the ticklabel font sizes are updated

`pyllars.mpl_utils.set_title_fontsize` (*ax: matplotlib.axes.\_axes.Axes, fontsize: Union[int, str]*) → None

Set the font size of the title of the axis. :param ax: The axis :type ax: matplotlib.axes.Axes :param fontsize: The size of the title font :type fontsize: int, or a str recognized by matplotlib

#### Returns

**Return type** None, but the title fontsize is updated

## 2.7 Natural language processing utilities

### 2.7.1 Parsing text

---

`clean_doc`

---

### 2.7.2 Definitions

## 2.8 Pandas utilities

This module contains utilities for data frame manipulation.

This module differs from *ml\_utils* and others because this module treats pandas data frames more like database tables which hold various types of records. The other modules tend to treat data frames as data matrices (in a statistical/machine learning sense).

### 2.8.1 Manipulating and processing data frames

|  |  |
|--|--|
| <code>apply(df, func, *args, progress_bar, **kwargs)</code>    | Apply <i>func</i> to each row in the data frame  |
| <code>apply_groups(groups, func, *args, ...)</code>            | Apply <i>func</i> to each group in <i>groups</i>   |
| <code>split_df(df, num_groups, chunk_size)</code>              | Split <i>df</i> into roughly equal-sized groups  |
| <code>join_df_list(dfs, join_col, List[str], ...)</code>       | Join a list of data frames on a common column  |
| <code>filter_rows(df_filter, df_to_keep, ...)</code>           | Filter rows from <i>df_to_keep</i> which have matches in <i>df_filter</i>  |
| <code>group_and_chunk_df(df, groupby_field, chunk_size)</code> | Group <i>df</i> using then given field, and then create “groups of groups” with <i>chunk_size</i> groups in each outer group |
| <code>get_group_extreme(df, ex_field, ex_type[, ...])</code>   | Find the row in each group of <i>df</i> with an extreme value for <i>ex_field</i>  |

### 2.8.2 Converting to and from data frames

|  |   |
|--|---|
| <code>dict_to_dataframe(dic, key_name, value_name)</code>  | Convert a dictionary into a two-column data frame using the given column names. |
| <code>dataframe_to_dict(df, key_field, value_field)</code> | Convert two columns of a data frame into a dictionary                           |

### 2.8.3 Other pandas helpers

|   |  |
|---|--|
| <code>get_series_union(*pd_series)</code> | Take the union of values from the list of series         |
| <code>groupby_to_generator(groups)</code> | Convert the groupby object to a generator of data frames |

### 2.8.4 Reading and writing data frames

|  |   |
|--|---|
| <code>read_df(filename, filetype, sheet, **kwargs)</code>  | Read a data frame from a file                       |
| <code>write_df(df, out, create_path, filetype, ...)</code> | Writes a data frame to a file of the specified type |
| <code>append_to_xlsx(df, xlsx[, sheet])</code>             | Append <i>df</i> to <i>xlsx</i>                     |

## 2.8.5 Definitions

This module contains utilities for data frame manipulation.

This module differs from `ml_utils` and others because this module treats pandas data frames more like database tables which hold various types of records. The other modules tend to treat data frames as data matrices (in a statistical/machine learning sense).

```
pyllars.pandas_utils.append_to_xlsx(df: pandas.core.frame.DataFrame, xlsx: str,
                                     sheet='Sheet_1', **kwargs) → None
```

Append `df` to `xlsx`

If the sheet already exists, it will be overwritten. If the file does not exist, it will be created.

**N.B. This will not work with an open file handle! The `xlsx` argument must be the path to the file.**

### Parameters

- **df** (`pandas.DataFrame`) – The data frame to write
- **xlsx** (`str`) – The path to the excel file
- **sheet** (`str`) – The name of the sheet, which will be truncated to 31 characters
- **kwargs** – Keyword arguments to pass to the appropriate “write” function.

**Returns** `None` – The sheet is appended to the excel file

**Return type** `None`

```
pyllars.pandas_utils.apply(df: pandas.core.frame.DataFrame, func: Callable, *args,
                             progress_bar: bool = False, **kwargs) → List
```

Apply `func` to each row in the data frame

Unlike `pandas.DataFrame.apply()`, this function does not attempt to “interpret” the results and cast them back to a data frame, etc.

### Parameters

- **df** (`pandas.DataFrame`) – the data frame
- **func** (`typing.Callable`) – The function to apply to each row in `data_frame`
- **args** – Positional arguments to pass to `func`
- **kwargs** – Keyword arguments to pass to `func`
- **progress\_bar** (`bool`) – Whether to show a progress bar when waiting for results.

**Returns** `results` – The result of each function call

**Return type** `typing.List`

```
pyllars.pandas_utils.apply_groups(groups: pandas.core.groupby.generic.DataFrameGroupBy,
                                   func: Callable, *args, progress_bar: bool = False,
                                   **kwargs) → List
```

Apply `func` to each group in `groups`

Unlike `pandas.core.groupby.GroupBy.apply()`, this function does not attempt to “interpret” the results by casting to a data frame, etc.

### Parameters

- **groups** (`pandas.core.groupby.GroupBy`) – The result of a call to `groupby` on a data frame

- **func** (*function pointer*) – The function to apply to each group in *groups*
- **args** – Positional arguments to pass to *func*
- **kwargs** – Keyword arguments to pass to *func*
- **progress\_bar** (*bool*) – Whether to show a progress bar when waiting for results.

**Returns** **results** – The result of each function call

**Return type** `typing.List`

```
pyllars.pandas_utils.dataframe_to_dict(df: pandas.core.frame.DataFrame, key_field: str,
                                       value_field: str) → Dict
```

Convert two columns of a data frame into a dictionary

**Parameters**

- **df** (*pandas.DataFrame*) – The data frame
- **key\_field** (*str*) – The field to use as the keys in the dictionary
- **value\_field** (*str*) – The field to use as the values

**Returns** **the\_dict** – A dictionary which has one entry for each row in the data frame, with the keys and values as indicated by the fields

**Return type** `typing.Dict`

```
pyllars.pandas_utils.dict_to_dataframe(dic: Dict, key_name: str = 'key', value_name: str =
                                       'value') → pandas.core.frame.DataFrame
```

Convert a dictionary into a two-column data frame using the given column names. Each entry in the data frame corresponds to one row.

**Parameters**

- **dic** (*typing.Dict*) – A dictionary
- **key\_name** (*str*) – The name to use for the column for the keys
- **value\_name** (*str*) – The name to use for the column for the values

**Returns** **df** – A data frame in which each row corresponds to one entry in *dic*

**Return type** `pandas.DataFrame`

```
pyllars.pandas_utils.filter_rows(df_filter: pandas.core.frame.DataFrame, df_to_keep:
                                pandas.core.frame.DataFrame, filter_on: List[str],
                                to_keep_on: List[str], drop_duplicates: bool = True)
                                → pandas.core.frame.DataFrame
```

Filter rows from *df\_to\_keep* which have matches in *df\_filter*

**N.B.** The order of the the columns in *filter\_on* and *to\_keep\_on* must match.

This is adapted from: <https://stackoverflow.com/questions/44706485>.

**Parameters**

- **df\_filter** (*pandas.DataFrame*) – The rows which will be used as the filter
- **df\_to\_keep** (*pandas.DataFrame*) – The rows which will be kept, unless they appear in *df\_filter*
- **filter\_on** (*typing.List[str]*) – The columns from *df\_filter* to use for matching
- **to\_keep\_on** (*typing.List[str]*) – The columns from *df\_to\_keep* to use for matching

- **drop\_duplicates** (*bool*) – Whether to remove duplicate rows from the filtered data frame

**Returns** **df\_filtered** – The rows of *df\_to\_keep* which do not appear in *df\_filter* (considering only the given columns)

**Return type** `pandas.DataFrame`

`pyllars.pandas_utils.get_group_extreme` (*df*: `pandas.core.frame.DataFrame`, *ex\_field*: *str*, *ex\_type*: *str* = 'max', *group\_fields*=None, *groups*: `pandas.core.groupby.groupby.GroupBy` = None) → `pandas.core.frame.DataFrame`

Find the row in each group of *df* with an extreme value for *ex\_field*

“ex\_type” must be either “max” or “min” and indicated which type of extreme to consider. Either the “group\_field” or “groups” must be given.

#### Parameters

- **df** (`pd.DataFrame`) – The original data frame. Even if the groups are created externally, the original data frame must be given.
- **ex\_field** (*str*) – The field to find for which to find the extreme values
- **ex\_type** (*str* {"max" or "min"}, *case-insensitive*) – The type of extreme to consider.
- **groups** (None or `pandas.core.groupby.GroupBy`) – If not None, then these groups will be used to find the maximum values.
- **group\_fields** (None or *str* or `typing.List[str]`) – If not None, then the field(s) by which to group the data frame. This value must be something which can be interpreted by `pd.DataFrame.groupby`.

**Returns** **ex\_df** – A data frame with rows which contain the extreme values for the indicated groups.

**Return type** `pandas.DataFrame`

`pyllars.pandas_utils.get_series_union` (\**pd\_series*) → Set

Take the union of values from the list of series

**Parameters** **pd\_series** (`typing.Iterable[pandas.Series]`) – The list of pandas series

**Returns** **set\_union** – The union of the values in all series

**Return type** `typing.Set`

`pyllars.pandas_utils.group_and_chunk_df` (*df*: `pandas.core.frame.DataFrame`, *groupby\_field*: *str*, *chunk\_size*: *int*) → `pandas.core.groupby.generic.DataFrameGroupBy`

Group *df* using then given field, and then create “groups of groups” with *chunk\_size* groups in each outer group

#### Parameters

- **df** (`pandas.DataFrame`) – The data frame
- **groupby\_field** (*str*) – The field for creating the initial grouping
- **chunk\_size** (*int*) – The size of each outer group

**Returns** **groups** – The groups

**Return type** `pandas.core.groupby.GroupBy`



`pyllars.pandas_utils.groupby_to_generator` (*groups:* `pandas.core.groupby.groupby.GroupBy`) → *pan-*  
Generator

Convert the groupby object to a generator of data frames

**Parameters** `groups` (`pandas.core.groupby.GroupBy`) – The groups

**Returns** `group_generator` – A generator over the data frames in *groups*

**Return type** `typing.Generator`

`pyllars.pandas_utils.join_df_list` (*dfs:* `List[pandas.core.frame.DataFrame]`, *join\_col:* `Union[str, List[str]]`, *\*args*, *\*\*kwargs*) → *pan-*  
`pandas.core.frame.DataFrame`

Join a list of data frames on a common column

**Parameters**

- `dfs` (`typing.Iterable[pandas.DataFrame]`) – The data frames
- `join_col` (`str` or `typing.List[str]`) – The name of the column(s) to use for joining. All of the data frames in *dfs* must have this column (or all columns in the list).
- `args` – Positional arguments to pass to `pandas.merge()`
- `kwargs` – Keyword arguments to pass to `pandas.merge()`

**Returns** `joined_df` – The data frame from joining all of those in the list on *join\_col*. This function does not especially handle other columns which appear in all data frames, and their names in the joined data frame will be adjusted according to the standard pandas suffix approach.

**Return type** `pandas.DataFrame`

`pyllars.pandas_utils.read_df` (*filename:* `str`, *filetype:* `str = 'AUTO'`, *sheet:* `str = None`, *\*\*kwargs*) → `pandas.core.frame.DataFrame`

Read a data frame from a file

By default, this function attempts to guess the type of the file based on its extension. Alternatively, the filetype can be explicitly specified. The supported types and extensions used for guessing are:

- excel: xls,xlsx
- hdf5: hdf, hdf5, h5, he5
- parquet: parq
- csv: all other extensions

**N.B.** In principle, matlab data files are hdf5, so this function should be able to read them. This has not been thoroughly tested, though.

**Parameters**

- `filename` (`str`) – The input file
- `filetype` (`str`) – The type of file, which determines which pandas read function will be called. If *AUTO*, the function uses the extensions mentioned above to guess the filetype.
- `sheet` (`str`) – For excel or hdf5 files, this will be passed to extract the desired information from the file. Please see `pandas.read_excel()` or `pandas.read_hdf()` for more information on how values are interpreted.
- `kwargs` – Keyword arguments to pass to the appropriate *read* function.

**Returns** `df` – The data frame

**Return type** `pandas.DataFrame`

```
pyllars.pandas_utils.split_df(df: pandas.core.frame.DataFrame, num_groups:
                               int = None, chunk_size: int = None) → pan-
                               das.core.groupby.generic.DataFrameGroupBy
```

Split *df* into roughly equal-sized groups

The size of the groups can be specified by either giving the number of groups (*num\_groups*) or the size of each group (*chunk\_size*).

The groups are contiguous rows in the data frame.

#### Parameters

- **df** (*pandas.DataFrame*) – The data frame
- **num\_groups** (*int*) – The number of groups
- **chunk\_size** (*int*) – The size of each group. If given, *num\_groups* groups has precedence over *chunk\_size*

**Returns** *groups* – The groups

**Return type** *pandas.core.groupby.GroupBy*

```
pyllars.pandas_utils.write_df(df: pandas.core.frame.DataFrame, out, create_path: bool = False,
                               filetype: str = 'AUTO', sheet: str = 'Sheet_1', compress: bool =
                               True, **kwargs) → None
```

Writes a data frame to a file of the specified type

Unless otherwise specified, csv files are gzipped when written. By default, the filetype will be guessed based on the extension. The supported types and extensions used for guessing are:

- excel: xls, xlsx
- hdf5: hdf, hdf5, h5, he5
- parquet: parq
- csv: all other extensions (e.g., “.gz” or “.bed”)

Additionally, the filetype can be specified as ‘excel\_writer’. In this case, the out object is taken to be a *pd.ExcelWriter*, and the df is appended to the writer. AUTO will also guess this correctly.

**N.B.** The hdf5 filetype has not been thoroughly tested.

#### Parameters

- **df** (*pandas.DataFrame*) – The data frame
- **out** (*str* or *pandas.ExcelWriter*) – The (complete) path to the file.  
The file name WILL NOT be modified. In particular, “.gz” WILL NOT be added if the file is to be zipped. As mentioned above, if the filetype is passed as ‘excel\_writer’, then this is taken to be a *pd.ExcelWriter* object.
- **create\_path** (*bool*) – Whether to create the path directory structure to the file if it does not already exist.  
N.B. This will not attempt to create the path to an excel\_writer since it is possible that it does not yet have one specified.
- **filetype** (*str*) – The type of output file to write. If AUTO, the function uses the extensions mentioned above to guess the filetype.
- **sheet** (*str*) – The name of the sheet (excel) or key (hdf5) to use when writing the file. This argument is not used for csv. For excel, the sheet is limited to 31 characters. It will be trimmed if necessary.

- **compress** (*bool*) – Whether to compress the output. This is only used for csv files.
- **kwargs** – Keyword arguments to pass to the appropriate “write” function.

**Returns** *None* – The file is created as specified

**Return type** *None*

## 2.9 Statistics utilities

This module contains helpers for various statistical calculations.

### 2.9.1 Analytic KL-divergence calculations

|   |  |
|---|--|
| <code>calculate_univariate_gaussian_kl(...)</code>        | Calculate the (asymmetric) KL-divergence between the univariate Gaussian distributions $p$ and $q$ |
| <code>calculate_symmetric_kl_divergence(p, q, ...)</code> | Calculates the symmetric KL-divergence between distributions $p$ and $q$                           |
| <code>symmetric_entropy(p, q)</code>                      | Calculate the symmetric <code>scipy.stats.entropy()</code> .                                       |
| <code>symmetric_gaussian_kl(p, q)</code>                  | Calculate the symmetric <code>pyllars.stats_utils.calculate_univariate_gaussian_kl()</code> .      |

### 2.9.2 Sufficient statistics and parameter estimation

|   |  |
|---|--|
| <code>get_population_statistics(...)</code>                   | Calculate the population size, mean and variance based on subpopulation statistics |
| <code>get_categorical_mle_estimates(observations, ...)</code> | Calculate the MLE estimates for the categorical observations                       |
| <code>fit_with_least_squares(x, y, w[, order])</code>         | Fit a polynomial relationship between $x$ and $y$ .                                |

### 2.9.3 Bayesian hypothesis testing

|  |  |
|--|--|
| <code>bayesian_proportion_test(x, int[, n, int], ...)</code> | Perform a Bayesian test to identify significantly different proportions. |
| <code>bayesian_means_test(x1, x2, ...)</code>                | Perform a Bayesian test to identify significantly different means.       |

### 2.9.4 Definitions

This module contains helpers for various statistical calculations.

`pyllars.stats_utils.bayesian_means_test` (*x1*: *numpy.ndarray*, *x2*: *numpy.ndarray*, *use\_jeffreys\_prior*: *bool* = *True*, *prior1*: *Optional[Tuple[float, float, float, float]]* = *None*, *prior2*: *Optional[Tuple[float, float, float, float]]* = *None*, *num\_samples*: *int* = *1000*, *seed*: *int* = *8675309*) → *Tuple[float, float, float]*

Perform a Bayesian test to identify significantly different means.

The test is based on a Gaussian conjugate model. (The normal-inverse-chi-square distribution is the prior.) It uses Monte Carlo simulation to estimate the posterior of the difference between the means of the populations, under the (probably dubious) assumption that the observations are Gaussian distributed. It also estimates the likelihood that  $\mu_1 > \mu_2$ , where  $\mu_i$  is the mean of each sample.

#### Parameters

- **x{1,2}** (*numpy.ndarray*) – The observations of each sample
- **use\_jeffreys\_prior** (*bool*) – Whether to use the Jeffreys prior. For more details, see:  
  
Murphy, K. Conjugate Bayesian analysis of the Gaussian distribution. Technical report, 2007.  
  
Briefly, the Jeffreys prior is: (sample mean,  $n$ ,  $n - 1$ , sample variance), according to a *pyllars.stats\_utils.normal\_inverse\_chi\_square()* distribution.
- **prior{1,2}** (*typing.Optional[typing.Tuple[float, float, float, float]]*) – If the Jeffreys prior is not used, then these parameters are used as the priors for the normal-inverse-chi-square. If only prior1 is given, then those values are also used for prior2, where prior\_i is taken as the prior for x\_i.
- **num\_samples** (*int*) – The number of simulations
- **seed** (*int*) – The seed for the random number generator

#### Returns

- **difference\_{mean,var}** (*float*) – The posterior mean and variance of the difference in the mean of the two samples. A negative difference\_mean indicates that the mean of x2 is higher.
- **p\_m1\_greater** (*float*) – The probability that  $\mu_1 > \mu_2$

```
pyllars.stats_utils.bayesian_proportion_test(x: Tuple[int, int], n: Tuple[int, int],
                                             prior: Tuple[float, float] = (0.5, 0.5),
                                             prior2: Optional[Tuple[float, float]] =
                                             None, num_samples: int = 1000, seed: int
                                             = 8675309) → Tuple[float, float, float]
```

Perform a Bayesian test to identify significantly different proportions.

This test is based on a beta-binomial conjugate model. It uses Monte Carlo simulations to estimate the posterior of the difference between the proportions, as well as the likelihood that  $\pi_1 > \pi_2$  (where  $\pi_i$  is the likelihood of success in sample  $i$ ).

#### Parameters

- **x** (*typing.Tuple[int, int]*) – The number of successes in each sample
- **n** (*typing.Tuple[int, int]*) – The number of trials in each sample
- **prior** (*typing.Tuple[float, float]*) – The parameters of the beta distribution used as the prior in the conjugate model for the first sample.
- **prior2** (*typing.Optional[typing.Tuple[float, float]]*) – The parameters of the beta distribution used as the prior in the conjugate model for the second sample. If this is not specified, then *prior* is used.
- **num\_samples** (*int*) – The number of simulations
- **seed** (*int*) – The seed for the random number generator

#### Returns

- **difference\_{mean,var}** (*float*) – The posterior mean and variance of the difference in the likelihood of success in the two samples. A negative mean indicates that the likelihood in sample 2 is higher.
- **p\_pi\_1\_greater** (*float*) – The probability that  $\pi_1 > \pi_2$

`pyllars.stats_utils.calculate_symmetric_kl_divergence` (*p: Any, q: Any, calculate\_kl\_divergence: Callable*) → *float*

Calculates the symmetric KL-divergence between distributions *p* and *q*

In particular, this function defines the symmetric KL-divergence to be:

$$D_{sym}(p||q) = \frac{D(p||q) + D(q||p)}{2}$$

#### Parameters

- **{p, q}** (*typing.Any*) – A representation of a distribution that can be used by the function *calculate\_kl\_divergence*
- **calculate\_kl\_divergence** (*typing.Callable*) – A function the calculates the KL-divergence between *p* and *q*

**Returns** *symmetric\_kl* – The symmetric KL-divergence between *p* and *q*

**Return type** *float*

`pyllars.stats_utils.calculate_univariate_gaussian_kl` (*mean\_p\_var\_p: Tuple[float, float], mean\_q\_var\_q: Tuple[float, float]*) → *float*

Calculate the (asymmetric) KL-divergence between the univariate Gaussian distributions *p* and *q*

That is, this calculates  $KL(p||q)$ .

**N.B.** This function uses the variance!

**N.B.** The parameters for each distribution are passed as pairs for easy use with *calculate\_symmetric\_kl\_divergence*.

See, for example,<sup>1</sup> for the formula.

**Parameters** **{mean\_p\_var\_p, mean\_q\_var\_q}** (*Typing.Tuple[float, float]*) – The parameters of the distributions.

**Returns** *kl\_divergence* – The KL divergence between the two distributions.

**Return type** *float*

## References

`pyllars.stats_utils.fit_with_least_squares` (*x: numpy.ndarray, y: numpy.ndarray, w: Optional[numpy.ndarray] = None, order=<polynomial\_order.linear: 1>*) → *Tuple[float, float, float, float]*

Fit a polynomial relationship between *x* and *y*.

Optionally, the values can be weighted.

#### Parameters

<sup>1</sup> Penny, W. “KL-Divergences of Normal, Gamma, Dirichlet and Wishart densities.” Wellcome Department of Cognitive Neurology, University College London, 2001.

- **{x, y}** (*numpy.ndarray*) – The input arrays
- **w** (*numpy.ndarray*) – A weighting of each of the (x,y) pairs for regression
- **order** (*polynomial\_order*) – The order of the fit

**Returns**

- **{intercept,slope,power}** (*float*) – The coefficients of the fit. power is 0 if the order is linear.
- **r\_sqr** (*float*) – The coefficient of determination

`pyllars.stats_utils.get_categorical_mle_estimates` (*observations: Iterable[int], cardinality: Optional[int] = None, use\_laplacian\_smoothing: bool = False, base\_1: bool = False*) → *numpy.ndarray*

Calculate the MLE estimates for the categorical observations

**Parameters**

- **observations** (*typing.Iterable[int]*) – The observed values. These are taken to already be “label encoded”, so they should be integers in [0,cardinality).
- **cardinality** (*typing.Optional[int]*) – The cardinality of the categorical variable. If *None*, then this is taken as the number of unique values in *observations*.
- **use\_laplacian\_smoothing** (*bool*) – Whether to use Laplacian (“add one”) smoothing for the estimates. This can also be interpreted as a symmetric Dirichlet prior with a concentration parameter of 1.
- **base\_1** (*bool*) – Whether the observations are base 1. If so, then the range is taken as [1, cardinality].

**Returns** *mle\_estimates* – The estimates. The size of the array is *cardinality*.

**Return type** *numpy.ndarray*

`pyllars.stats_utils.get_population_statistics` (*subpopulation\_sizes: numpy.ndarray, subpopulation\_means: numpy.ndarray, subpopulation\_variances: numpy.ndarray*) → *Tuple[float, float, float, float]*

Calculate the population size, mean and variance based on subpopulation statistics

This code is based on “Chap”’s answer here: <https://stats.stackexchange.com/questions/30495>

This calculation seems to underestimate the variance relative to `numpy.var()` on the entire dataset (determined by simulation). This may somehow relate to “biased” vs. “unbiased” variance estimates (basically, whether to subtract 1 from the population size). Still, naive approaches to correct for that do not produce variance estimates which exactly match those from `numpy.var()`.

**Parameters** *subpopulation\_{sizes,means,variances}* (*numpy.ndarray*) – The subpopulation sizes, means, and variances, respectively. These should all be the same size.

**Returns** *population\_{size,mean,variance,std}* – The respective statistics about the entire population

**Return type** *float*

`pyllars.stats_utils.normal_inverse_chi_square` (*m, k, r, s, size=1*)

Sample from a normal-inverse-chi-square distribution with parameters m, k, r, s.

This distribution is of interest because it is a conjugate prior for Gaussian observations.

Sampling is described in: <https://www2.stat.duke.edu/courses/Fall10/sta1114/notes15.pdf>

### Parameters

- **k** ( $m_r$ ) –  $m$  is the mean of the sampled mean;  $k$  relates to the variance of the sampled mean.
- **s** ( $r_r$ ) –  $r$  is the degrees of freedom in the chi-square distribution from which the variance is samples;  $s$  is something like a scaling factor.
- **size** (*int or tuple of ints, or None*) – Output shape. This shares the semantics as other numpy sampling functions.

`pyllars.stats_utils.symmetric_entropy(p, q) → float`

Calculate the symmetric `scipy.stats.entropy()`.

`pyllars.stats_utils.symmetric_gaussian_kl(p, q) → float`

Calculate the symmetric `pyllars.stats_utils.calculate_univariate_gaussian_kl()`.

**class** `pyllars.stats_utils.polynomial_order`

An enumeration.

## 2.10 String utilities

Utilities for working with strings

### 2.10.1 Encoding

|   |  |
|---|--|
| <code>encode_sequence(sequence, encoding_map, ...)</code> | Extract the amino acid properties of the given sequence          |
| <code>encode_all_sequences(sequences, ...)</code>         | Extract the amino acid feature vectors for each peptide sequence |

### 2.10.2 Length manipulation

|   |   |
|---|---|
| <code>pad_sequence(seq, max_seq_len, pad_value, align)</code> | Pad <i>seq</i> to <i>max_seq_len</i> with <i>value</i> based on the <i>align</i> strategy |
| <code>pad_trim_sequences(seq_vec, pad_value, ...)</code>      | Pad and/or trim a list of sequences to have common length                                 |
| <code>trim_sequence(seq, maxlen, align)</code>                | Trim <i>seq</i> to at most <i>maxlen</i> characters using <i>align</i> strategy           |

### 2.10.3 Other operations

|   |  |
|---|--|
| <code>simple_fill(text, width)</code>       | Split <i>text</i> into equal-sized chunks of length <i>width</i> |
| <code>split(s, delimiters, maxsplit)</code> | Split <i>s</i> on any of the given <i>delimiters</i>             |

### 2.10.4 Human-readable data type helpers

|                                     |  |
|-------------------------------------|--|
| <code>bytes2human(n, format)</code> | Convert <i>n</i> bytes to a human-readable format  |
| <code>human2bytes(s)</code>         | Convert a human-readable byte string to an integer |

Continued on next page

Table 32 – continued from previous page

|                                 |  |
|---------------------------------|--|
| <code>try_parse_float(s)</code> | Convert <i>s</i> to a float, if possible         |
| <code>try_parse_int(s)</code>   | Convert <i>s</i> to an integer, if possible      |
| <code>str2bool(s)</code>        | Convert <i>s</i> to a boolean value, if possible |

## 2.10.5 Definitions

Utilities for working with strings

`pyllars.string_utils.bytes2human(n: int, format: str = '%(value)i%(symbol)s') → str`  
 Convert *n* bytes to a human-readable format

This code is adapted from: <http://goo.gl/zeJZI>

### Parameters

- **n** (*int*) – The number of bytes
- **format** (*string*) – The format string

**Returns** `human_str` – A human-readable version of the number of bytes

**Return type** `string`

### Examples

```
>>> bytes2human(10000)
'9K'
>>> bytes2human(100001221)
'95M'
```

`pyllars.string_utils.encode_all_sequences(sequences: Iterable[str], encoding_map: Mapping[str, numpy.ndarray], maxlen: Optional[int] = None, align: str = 'start', pad_value: str = 'J', same_length: bool = False, flatten: bool = False, return_as_numpy: bool = True, swap_axes: bool = False, progress_bar: bool = True) → Union[numpy.ndarray, List]`

Extract the amino acid feature vectors for each peptide sequence

See `get_peptide_aa_features` for more details.

### Parameters

- **sequences** (*typing.Iterable[str]*) – The sequences
- **encoding\_map** (*typing.Mapping[str, numpy.ndarray]*) – The features for each character
- **maxlen** (*typing.Optional[int]*) –
- **align** (*str*) –
- **pad\_value** (*str*) –
- **same\_length** (*bool*) –
- **flatten** (*bool*) – Whether to (attempt to) convert the features of each peptide into a single long vector (*True*) or leave as a (presumably) 2d position-feature vector.



- **return\_as\_numpy** (*bool*) – Whether to return as a 2d or 3d numpy array (*True*) or a list containing 1d or 2d numpy arrays. (The dimensionality depends upon *flatten*.)
- **swap\_axes** (*bool*) – If the values are returned as a numpy tensor, swap axes 1 and 2.  
N.B. This flag is only compatible with *return\_as\_numpy=True* and *flatten=False*.
- **progress\_bar** (*bool*) – Whether to show a progress bar for collecting the features.

**Returns** **all\_encoded\_peptides** – The resulting features. See the *flatten* and *return\_as\_numpy* parameters for the expected output.

**Return type** `typing.Union[numpy.ndarray, typing.List]`

```
pyllars.string_utils.encode_sequence(sequence: str, encoding_map: Mapping[str,
                                numpy.ndarray], flatten: bool = False) →
                                numpy.ndarray
```

Extract the amino acid properties of the given sequence

This function is designed with the idea of mapping from a sequence to numeric features (such as chemical properties or BLOSUM features for amino acid sequences). It may fail if other features are included in *encoding\_map*.

#### Parameters

- **sequence** (*str*) – The sequence
- **encoding\_map** (*typing.Mapping[str, numpy.ndarray]*) – A mapping from each character to a set of features. Presumably, the features are numpy-like arrays, though they need not be.
- **flatten** (*bool*) – Whether to flatten the encoded sequence into a single, 1d array or leave them as-is.

**Returns** **encoded\_sequence** – A 1d or 2d np.array, depending on *flatten*. By default (*flatten=False*), this is a 1d array of objects, in which the outer dimension indexes the position in the epitope. If *flatten* is *True*, then the function attempts to reshape the features into a single long feature vector. This will likely fail if the *encoding\_map* values are not numpy-like arrays.

**Return type** `numpy.ndarray`

```
pyllars.string_utils.human2bytes(s: str) → int
```

Convert a human-readable byte string to an integer

This code is adapted from: <http://goo.gl/zeJZI>

**Parameters** **s** (*string*) – The human-readable byte string

**Returns** **num\_bytes** – The number of bytes

**Return type** `int`

#### Examples

```
>>> human2bytes('1M')
1048576
>>> human2bytes('1G')
1073741824
```

```
pyllars.string_utils.pad_sequence(seq: str, max_seq_len: int, pad_value: str = 'J', align: str
                                = 'end') → str
```

Pad *seq* to *max\_seq\_len* with *value* based on the *align* strategy

If *seq* is already of length *max\_seq\_len* or longer it will not be changed.

**Parameters**

- **seq** (*str*) – The character sequence
- **max\_seq\_len** (*int*) – The maximum length for a sequence
- **pad\_value** (*str*) – The value for padding. This should be a single character
- **align** (*str*) – The strategy for padding the string. Valid options are *start*, *end*, and *center*

**Returns** **padded\_seq** – The padded string. In case *seq* was already long enough or longer, it will not be changed. So *padded\_seq* could be longer than *max\_seq\_len*.

**Return type** *str*

```
pyllars.string_utils.pad_trim_sequences(seq_vec: Sequence[str], pad_value: str = 'J',
                                       maxlen: Optional[int] = None, align: str = 'start')
                                       → List[str]
```

Pad and/or trim a list of sequences to have common length

The procedure is as follows:

1. Pad the sequence with *pad\_value*
2. Trim the sequence

**Parameters**

- **seq\_vec** (*typing.Sequence[str]*) – List of sequences that can have various lengths
- **pad\_value** (*str*) – Neutral element with which to pad the sequence. This should be a single character.
- **maxlen** (*typing.Optional[int]*) – Length of padded/trimmed sequences. If *None*, *maxlen* is set to the longest sequence length.
- **align** (*str*) – To which end to align the sequences when trimming/padding. Valid options are *start*, *end*, *center*

**Returns** **padded\_sequences** – The padded and/or trimmed sequences

**Return type** *typing.List[str]*

```
pyllars.string_utils.simple_fill(text: str, width: int = 60) → str
```

Split *text* into equal-sized chunks of length *width*

This is a simplified version of *textwrap.fill*.

The code is adapted from: <http://stackoverflow.com/questions/11781261>

**Parameters**

- **text** (*string*) – The text to split
- **width** (*int*) – The (exact) length of each line after splitting

**Returns** **split\_str** – A single string with lines of length *width* (except possibly the last line)

**Return type** *string*

```
pyllars.string_utils.split(s: str, delimiters: Iterable[str], maxsplit: int = 0) → List[str]
```

Split *s* on any of the given *delimiters*

This code is adapted from: <http://stackoverflow.com/questions/4998629/>

**Parameters**

- **s** (*string*) – The string to split
- **delimiters** (*list of strings*) – The strings to use as delimiters
- **maxsplit** (*int*) – The maximum number of splits (or 0 for no limit)

**Returns** **splits** – the split strings

**Return type** list of strings

`pyllars.string_utils.str2bool(s: str) → bool`

Convert *s* to a boolean value, if possible

**Parameters** **s** (*string*) – A string which may represent a boolean value

**Returns** **bool\_s** – *True* if *s* is in `_TRUE_STRING`, and *False* otherwise

**Return type** boolean

`pyllars.string_utils.trim_sequence(seq: str, maxlen: int, align: str = 'end') → str`

Trim *seq* to at most *maxlen* characters using *align* strategy

**Parameters**

- **seq** (*str*) – The (amino acid) sequence
- **maxlen** (*int*) – The maximum length
- **align** (*str*) – The strategy for trimming the string. Valid options are *start*, *end*, and *center*

**Returns** **trimmed\_seq** – The trimmed string. In case *seq* was already an appropriate length, it will not be changed. So *trimmed\_seq* could be shorter than *maxlen*.

**Return type** *str*

`pyllars.string_utils.try_parse_float(s: str) → Optional[float]`

Convert *s* to a float, if possible

**Parameters** **s** (*string*) – A string which may represent a float

**Returns**

- **float\_s** (*float*) – A float
- — *OR* —
- *None* – If *s* cannot be parsed into a *float*.

`pyllars.string_utils.try_parse_int(s: str) → Optional[int]`

Convert *s* to an integer, if possible

**Parameters** **s** (*string*) – A string which may represent an integer

**Returns**

- **int\_s** (*int*) – An integer
- — *OR* —
- *None* – If *s* cannot be parsed into an *int*.



---

## Domain-specific API

---

This is the API for the domain-specific utilities in the pyllars library.

### 3.1 Physionet utilities

This module contains functions for working with datasets from physionet, including MIMIC and the Computing in Cardiology Challenge 2012 datasets.

In the future, this module may be renamed and updated to also work with the eICU dataset.

Please see the respective documentation for more details:

- MIMIC-III: <https://mimic.physionet.org/about/mimic/>
- CINC 2012: <https://physionet.org/challenge/2012/>
- eICU: <https://eicu-crd.mit.edu/about/eicu/>

#### 3.1.1 MIMIC-III utilities

---

`fix_mimic_icds(icds)`

Add the decimal to the correct location for the given ICD codes

---

#### 3.1.2 Definitions

This module contains functions for working with datasets from physionet, including MIMIC and the Computing in Cardiology Challenge 2012 datasets.

In the future, this module may be renamed and updated to also work with the eICU dataset.

Please see the respective documentation for more details:

- MIMIC-III: <https://mimic.physionet.org/about/mimic/>

- CINC 2012: <https://physionet.org/challenge/2012/>
- eICU: <https://eicu-crd.mit.edu/about/eicu/>

`pyllars.physionet_utils._fix_mimic_icd(icd)`

Add the decimal to the correct location for the ICD code

From the mimic documentation ([https://mimic.physionet.org/mimictables/diagnoses\\_icd/](https://mimic.physionet.org/mimictables/diagnoses_icd/)):

> The code field for the ICD-9-CM Principal and Other Diagnosis Codes > is six characters in length, with the decimal point implied between > the third and fourth digit for all diagnosis codes other than the V > codes. The decimal is implied for V codes between the second and third > digit.

`pyllars.physionet_utils._get_cinc_2012_record_descriptor(record_file_df)`

Given the record file data frame, use the first six rows to extract the descriptor information. See the documentation (<https://physionet.org/challenge/2012/>, “General descriptors”) for more details.

`pyllars.physionet_utils.create_followups_table(mimic_base, progress_bar=True)`

Create the FOLLOWUPS table, based on the admissions

**In particular, the table has the following columns:**

- `HADM_ID`
- `FOLLOWUP_HADM_ID`
- **FOLLOWUP\_TIME:** the difference between the discharge time of the first admission and the admit time of the second admission
- `SUBJECT_ID`

#### Parameters

- **mimic\_base** (*path-like*) – The path to the main MIMIC folder
- **progress\_bar** (*bool*) – Whether to show a progress bar for creating the table

**Returns** `df_followups` – The data frame constructed as described above. Currently, there is no need to create this table more than once. It can just be written to disk and loaded using `get_followups` after the initial creation.

**Return type** `pd.DataFrame`

`pyllars.physionet_utils.fix_mimic_icds(icds: Iterable[str]) → List[str]`

Add the decimal to the correct location for the given ICD codes

Since adding the decimals is a string-based operation, it can be somewhat slow. Thus, it may make sense to perform any filtering before fixing the (possibly much smaller number of) ICD codes.

**Parameters** `icds` (*typing.Iterable[str]*) – ICDs from the various mimic ICD columns

**Returns** `fixed_icds` – The ICD codes with decimals in the correct location

**Return type** `List[str]`

`pyllars.physionet_utils.get_admissions(mimic_base: str, **kwargs) → pandas.core.frame.DataFrame`

Load the ADMISSIONS table

This function automatically treats the following columns as date-times:

- `ADMITTIME`
- `DISCHTIME`
- `DEATHTIME`

- *EDREGTIME*
- *EDOUTTIME*

#### Parameters

- **mimic\_base** (*str*) – The path to the main MIMIC folder
- **kwargs** (*<key>=<value> pairs*) – Additional key words to pass to *read\_df*

**Returns** **admissions** – The admissions table as a pandas data frame

**Return type** `pandas.DataFrame`

```
pyllars.physionet_utils.get_cinc_2012_outcomes(cinc_2012_base, to_pandas=True)
```

Load the Outcomes-a.txt file.

N.B. This file is assumed to be named “Outcomes-a.txt” and located directly in the *cinc\_2012\_base* directory

#### Parameters

- **cinc\_2012\_base** (*path-like*) – The path to the main folder for this CinC challenge
- **to\_pandas** (*bool*) – Whether to read the table as a pandas (True) or dask (False) data frame

#### Returns

**cinc\_2012\_base** – The “Outcomes-a” table as either a pandas or dask data frame, depending on the value of *to\_pandas*. It contains the following columns:

- **HADM\_ID**: string, a key into the record table
- **SAPS-I**: integer, the SAPS-I score
- **SOFA**: integer, the SOFA score
- **ADMISSION\_ELAPSED\_TIME**: `pd.timedelta`, time in the hospital, in days
- **SURVIVAL: time between ICU admission and observed death**. If the patient survived (or the death was not recorded), then the value is `np.nan`.
- **EXPIRED**: bool, whether the patient died in the hospital

**Return type** `pd.DataFrame` or `dd.DataFrame`

```
pyllars.physionet_utils.get_cinc_2012_record(cinc_2012_base, record_id, wide=True)
```

Load the record file for the given id.

N.B. This file is assumed to be named “<record\_id>.txt” and located in the “<cinc\_2012\_base>/set-a” directory.

#### Parameters

- **cinc\_2012\_base** (*path-like*) – The path to the main folder for this CinC challenge
- **record\_id** (*string-like*) – The identifier for this record, e.g., “132539”
- **wide** (*bool*) – Whether to return a “long” or “wide” data frame
- **According to the specification (<https> (N.B.) –**  
**descriptors”), six descriptors are recorded only when the**  
**patients (“General) –**
- **admitted to the ICU and are included only once at the**  
**beginning of the (are) –**
- **record. –**

**Returns****record\_descriptors** –

The six descriptors:

- **HADM\_ID:** string, the record id. We call it “HADM\_ID” to keep the nomenclature consistent with the MIMIC data
- **ICU\_TYPE:** string [“coronary\_care\_unit”, “cardiac\_surgery\_recovery\_unit”, “medical\_icu”, “surgical\_icu”]
- **GENDER:** string [‘female’, ‘male’]
- **AGE:** float (or np.nan for missing)
- **WEIGHT:** float (or np.nan for missing)
- **HEIGHT:** float (or np.nan for missing)

**Return type** dictionary

**observations:** `pd.DataFrame` The remaining time series entries for this record. This is returned as either a “long” or “wide” data frame with columns:

- **HADM\_ID:** string (added for easy joining, etc.)
- **ELAPSED\_TIME:** `timedelta64[ns]`
- **MEASUREMENT:** the name of the measurement
- **VALUE:** the value of the measurement

For a wide data frame, there is instead one column for each measurement.

```
pyllars.physionet_utils.get_diagnosis_icds(mimic_base: str, drop_incomplete_records:
                                           bool = False, fix_icds: bool = False) → pandas.core.frame.DataFrame
```

Load the *DIAGNOSES\_ICDS* table

**Parameters**

- **mimic\_base** (*str*) – The path to the main MIMIC folder
- **drop\_incomplete\_records** (*bool*) – Some of the ICD codes are missing. If this flag is *True*, then those records will be removed.
- **fix\_icds** (*bool*) – Whether to add the decimal point in the correct position for the ICD codes

**Returns** `diagnosis_icds` – The diagnosis ICDs table as a pandas data frame

**Return type** `pandas.DataFrame`

```
pyllars.physionet_utils.get_followups(mimic_base: str) → pandas.core.frame.DataFrame
```

Load the (constructed) FOLLOWUPS table

**Parameters** **mimic\_base** (*str*) – The path to the main MIMIC folder

**Returns** `df_followups` – A data frame containing the followup information

**Return type** `pandas.DataFrame`

```
pyllars.physionet_utils.get_icu_stays(mimic_base, to_pandas=True, **kwargs)
```

Load the ICUSTAYS table

**Parameters**



- **mimic\_base** (*path-like*) – The path to the main MIMIC folder
- **to\_pandas** (*bool*) – Whether to read the table as a pandas (True) or dask (False) data frame
- **kwargs** (*key=value pairs*) – Additional keywords to pass to the appropriate *read* function

**Returns**

**patients** –

The patients table as either a pandas or dask data frame, depending on the value of `to_pandas`

**Return type** `pd.DataFrame` or `dd.DataFrame`

```
pyllars.physionet_utils.get_lab_events(mimic_base, to_pandas=True,
                                       drop_missing_admission=False, parse_dates=True,
                                       **kwargs)
```

Load the LABEVENTS table

**Parameters**

- **mimic\_base** (*path-like*) – The path to the main MIMIC folder
- **to\_pandas** (*bool*) – Whether to read the table as a pandas (True) or dask (False) data frame
- **drop\_missing\_admission** (*bool*) – About 20% of the lab events do not have an associated HADM\_ID. If this flag is True, then those will be removed.
- **parse\_dates** (*bool*) – Whether to directly parse *CHARTTIME* as a date. The main reason to skip this (when *parse\_dates* is False) is if the *CHARTTIME* column is skipped (using the *usecols* parameter).
- **kwargs** (*key=value pairs*) – Additional keywords to pass to the appropriate *read* function

**Returns**

**lab\_events** –

The notes table as either a pandas or dask data frame, depending on the value of `to_pandas`

**Return type** `pd.DataFrame` or `dd.DataFrame`

```
pyllars.physionet_utils.get_lab_items(mimic_base, to_pandas=True, **kwargs)
```

Load the D\_LABITEMS table

**Parameters**

- **mimic\_base** (*path-like*) – The path to the main MIMIC folder
- **to\_pandas** (*bool*) – Whether to read the table as a pandas (True) or dask (False) data frame
- **kwargs** (*key=value pairs*) – Additional keywords to pass to the appropriate *read* function

**Returns**

**diagnosis\_icds** –

The notes table as either a pandas or dask data frame, depending on the value of `to_pandas`

**Return type** `pd.DataFrame` or `dd.DataFrame`

`pyllars.physionet_utils.get_notes(mimic_base, to_pandas=True, **kwargs)`

Load the NOTEEVENTS table

#### Parameters

- **mimic\_base** (*path-like*) – The path to the main MIMIC folder
- **to\_pandas** (*bool*) – Whether to read the table as a pandas (True) or dask (False) data frame
- **kwargs** (*key=value pairs*) – Additional keywords to pass to the appropriate *read* function

#### Returns

**diagnosis\_icds** –

The notes table as either a pandas or dask data frame, depending on the value of to\_pandas

**Return type** `pd.DataFrame` or `dd.DataFrame`

`pyllars.physionet_utils.get_patients(mimic_base, to_pandas=True, **kwargs)`

Load the PATIENTS table

#### Parameters

- **mimic\_base** (*path-like*) – The path to the main MIMIC folder
- **to\_pandas** (*bool*) – Whether to read the table as a pandas (True) or dask (False) data frame
- **kwargs** (*key=value pairs*) – Additional keywords to pass to the appropriate *read* function

#### Returns

**patients** –

The patients table as either a pandas or dask data frame, depending on the value of to\_pandas

**Return type** `pd.DataFrame` or `dd.DataFrame`

`pyllars.physionet_utils.get_procedure_icds(mimic_base, to_pandas=True)`

Load the PROCEDURES\_ICD table

#### Parameters

- **mimic\_base** (*path-like*) – The path to the main MIMIC folder
- **to\_pandas** (*bool*) – Whether to read the table as a pandas (True) or dask (False) data frame

#### Returns

**procedure\_icds** –

The procedure ICDs table as either a pandas or dask data frame, depending on the value of to\_pandas

**Return type** `pd.DataFrame` or `dd.DataFrame`

`pyllars.physionet_utils.get_transfers(mimic_base, to_pandas=True, **kwargs)`

Load the TRANSFERS table

#### Parameters

- **mimic\_base** (*path-like*) – The path to the main MIMIC folder

- **to\_pandas** (*bool*) – Whether to read the table as a pandas (True) or dask (False) data frame

**kwargs: key=value pairs** Additional keywords to pass to the appropriate *read* function

#### Returns

**transfers** –

The transfers table as either a pandas or dask data frame, depending on the value of `to_pandas`

**Return type** `pd.DataFrame` or `dd.DataFrame`

`pyllars.physionet_utils.parse_rdsamp_datetime(fname, version=2)`

Extract the identifying information from the filename of the MIMIC-III header (\*.hea) files

In this project, we refer to each of these files as an “episode”.

**Parameters** **fname** (*string*) – The name of the file. It should be of the form:

**version 1:** `/path/to/my/s09870-2111-11-04-12-36.hea` `/path/to/my/s09870-2111-11-04-12-36n.hea`

**version 2:** `/path/to/my/p000020-2183-04-28-17-47.hea` `/path/to/my/p000020-2183-04-28-17-47n.hea`

#### Returns

**episode\_timestap** – A dictionary containing the time stamp and subject id for this episode. Specifically, it includes the following keys:

- **SUBJECT\_ID**: the patient identifier
- **EPISODE\_ID**: the identifier for this episode
- **EPISODE\_BEGIN\_TIME**: the beginning time for this episode

**Return type** `dict`



These are various tutorials.

#### **4.1 Cross-validation tutorial**

This tutorial gives an overview of evaluating models, including hyperparameter selection, using cross-validation.

#### **4.2 Bayesian model selection tutorial**

This tutorial introduces Bayesian model selection and describes how it can be used for unsupervised classification.



## CHAPTER 5

---

### Indices and tables

---

- `genindex`
- `modindex`
- `search`





### p

- `pyllars.collection_utils`, 4
- `pyllars.dask_utils`, 8
- `pyllars.logging_utils`, 13
- `pyllars.matrix_utils`, 16
- `pyllars.ml_utils`, 19
- `pyllars.mpl_utils`, 30
- `pyllars.pandas_utils`, 42
- `pyllars.physionet_utils`, 57
- `pyllars.stats_utils`, 47
- `pyllars.string_utils`, 52



## Symbols

`_asdict()` (*pyllars.ml\_utils.estimators\_predictions\_metrics* method), 24  
`_calc_hand_and_till_a_value()` (*in module pyllars.ml\_utils*), 19  
`_fix_mimic_icd()` (*in module pyllars.physionet\_utils*), 58  
`_get_cinc_2012_record_descriptor()` (*in module pyllars.physionet\_utils*), 58  
`_get_fig_ax()` (*in module pyllars.mpl\_utils*), 30  
`_make()` (*pyllars.ml\_utils.estimators\_predictions\_metrics* class method), 24  
`_replace()` (*pyllars.ml\_utils.estimators\_predictions\_metrics* method), 24  
`_train_and_evaluate()` (*in module pyllars.ml\_utils*), 20

## A

`add_dask_options()` (*in module pyllars.dask\_utils*), 8  
`add_dask_values_to_args()` (*in module pyllars.dask\_utils*), 9  
`add_fontsizes_to_args()` (*in module pyllars.mpl\_utils*), 30  
`add_logging_options()` (*in module pyllars.logging\_utils*), 13  
`add_logging_values_to_args()` (*in module pyllars.logging\_utils*), 13  
`append_to_xlsx()` (*in module pyllars.pandas\_utils*), 42  
`apply()` (*in module pyllars.pandas\_utils*), 42  
`apply_df()` (*in module pyllars.dask\_utils*), 9  
`apply_groups()` (*in module pyllars.dask\_utils*), 10  
`apply_groups()` (*in module pyllars.pandas\_utils*), 42  
`apply_iter()` (*in module pyllars.dask\_utils*), 10  
`apply_no_return()` (*in module pyllars.collection\_utils*), 4

## B

`bayesian_means_test()` (*in module pyllars.stats\_utils*), 47  
`bayesian_proportion_test()` (*in module pyllars.stats\_utils*), 48  
`bytes2human()` (*in module pyllars.string\_utils*), 52

## C

`calc_hand_and_till_m_score()` (*in module pyllars.ml\_utils*), 20  
`calc_provost_and_domingos_auc()` (*in module pyllars.ml\_utils*), 21  
`calculate_symmetric_kl_divergence()` (*in module pyllars.stats\_utils*), 49  
`calculate_univariate_gaussian_kl()` (*in module pyllars.stats\_utils*), 49  
`cancel_all()` (*in module pyllars.dask\_utils*), 11  
`center_splines()` (*in module pyllars.mpl\_utils*), 30  
`check_status()` (*in module pyllars.dask\_utils*), 11  
`col_op()` (*in module pyllars.matrix\_utils*), 16  
`col_sum()` (*in module pyllars.matrix\_utils*), 16  
`col_sum_mean()` (*in module pyllars.matrix\_utils*), 16  
`collect_binary_classification_metrics()` (*in module pyllars.ml\_utils*), 21  
`collect_multiclass_classification_metrics()` (*in module pyllars.ml\_utils*), 22  
`collect_regression_metrics()` (*in module pyllars.ml\_utils*), 23  
`collect_results()` (*in module pyllars.dask\_utils*), 11  
`collect_results()` (*pyllars.dask\_utils.dask\_pipeline* method), 12  
`connect()` (*in module pyllars.dask\_utils*), 11  
`create_followups_table()` (*in module pyllars.physionet\_utils*), 58

## D

`dask_pipeline` (*class in pyllars.dask\_utils*), 12

- `dataframe_to_dict()` (in module `pyllars.pandas_utils`), 43
- `dict_to_dataframe()` (in module `pyllars.pandas_utils`), 43
- `draw_rectangle()` (in module `pyllars.mpl_utils`), 31
- ## E
- `encode_all_sequences()` (in module `pyllars.string_utils`), 52
- `encode_sequence()` (in module `pyllars.string_utils`), 53
- `estimator_test` (`pyllars.ml_utils.estimators_predictions_metrics` attribute), 24
- `estimator_val` (`pyllars.ml_utils.estimators_predictions_metrics` attribute), 24
- `estimators_predictions_metrics` (class in `pyllars.ml_utils`), 24
- `evaluate_hyperparameters()` (in module `pyllars.ml_utils`), 25
- ## F
- `filter_rows()` (in module `pyllars.pandas_utils`), 43
- `fit()` (`pyllars.dask_utils.dask_pipeline` method), 12
- `fit_with_least_squares()` (in module `pyllars.stats_utils`), 49
- `fix_mimic_icds()` (in module `pyllars.physionet_utils`), 58
- `flatten_lists()` (in module `pyllars.collection_utils`), 4
- `fold_data` (class in `pyllars.ml_utils`), 28
- `fold_test` (`pyllars.ml_utils.estimators_predictions_metrics` attribute), 24
- `fold_train` (`pyllars.ml_utils.estimators_predictions_metrics` attribute), 24
- `fold_val` (`pyllars.ml_utils.estimators_predictions_metrics` attribute), 24
- ## G
- `get_admissions()` (in module `pyllars.physionet_utils`), 58
- `get_categorical_mle_estimates()` (in module `pyllars.stats_utils`), 50
- `get_cinc_2012_outcomes()` (in module `pyllars.physionet_utils`), 59
- `get_cinc_2012_record()` (in module `pyllars.physionet_utils`), 59
- `get_cv_folds()` (in module `pyllars.ml_utils`), 26
- `get_dask_cmd_options()` (in module `pyllars.dask_utils`), 12
- `get_dense_row()` (in module `pyllars.matrix_utils`), 16
- `get_diagnosis_icds()` (in module `pyllars.physionet_utils`), 60
- `get_diff_counts()` (in module `pyllars.mpl_utils`), 31
- `get_fold_data()` (in module `pyllars.ml_utils`), 27
- `get_followups()` (in module `pyllars.physionet_utils`), 60
- `get_group_extreme()` (in module `pyllars.pandas_utils`), 44
- `get_icu_stays()` (in module `pyllars.physionet_utils`), 60
- `get_ipython_logger()` (in module `pyllars.logging_utils`), 14
- `get_lab_events()` (in module `pyllars.physionet_utils`), 61
- `get_lab_items()` (in module `pyllars.physionet_utils`), 61
- `get_logging_cmd_options()` (in module `pyllars.logging_utils`), 14
- `get_logging_options_string()` (in module `pyllars.logging_utils`), 14
- `get_notes()` (in module `pyllars.physionet_utils`), 61
- `get_patients()` (in module `pyllars.physionet_utils`), 62
- `get_population_statistics()` (in module `pyllars.stats_utils`), 50
- `get_procedure_icds()` (in module `pyllars.physionet_utils`), 62
- `get_series_union()` (in module `pyllars.pandas_utils`), 44
- `get_set_pairwise_intersections()` (in module `pyllars.collection_utils`), 4
- `get_train_val_test_splits()` (in module `pyllars.ml_utils`), 27
- `get_transfers()` (in module `pyllars.physionet_utils`), 62
- `group_and_chunk_df()` (in module `pyllars.pandas_utils`), 44
- `groupby_to_generator()` (in module `pyllars.pandas_utils`), 44
- ## H
- `hide_first_y_tick_label()` (in module `pyllars.mpl_utils`), 31
- `hide_tick_labels()` (in module `pyllars.mpl_utils`), 31
- `hide_tick_labels_by_index()` (in module `pyllars.mpl_utils`), 31
- `hide_tick_labels_by_text()` (in module `pyllars.mpl_utils`), 32
- `human2bytes()` (in module `pyllars.string_utils`), 53
- `hyperparameters` (`pyllars.ml_utils.estimators_predictions_metrics` attribute), 24

hyperparameters\_str (pyllars.ml\_utils.estimators\_predictions\_metrics attribute), 24

**I**

is\_iterator\_exhausted() (in module pyllars.collection\_utils), 5

**J**

join\_df\_list() (in module pyllars.pandas\_utils), 45

**L**

list\_insert\_list() (in module pyllars.collection\_utils), 5

list\_remove\_list() (in module pyllars.collection\_utils), 5

list\_to\_dict() (in module pyllars.collection\_utils), 6

**M**

matrix\_multiply() (in module pyllars.matrix\_utils), 17

merge\_sets() (in module pyllars.collection\_utils), 6

metrics\_test (pyllars.ml\_utils.estimators\_predictions\_metrics attribute), 25

metrics\_val (pyllars.ml\_utils.estimators\_predictions\_metrics attribute), 25

**N**

normal\_inverse\_chi\_square() (in module pyllars.stats\_utils), 50

normalize\_columns() (in module pyllars.matrix\_utils), 17

normalize\_rows() (in module pyllars.matrix\_utils), 17

**P**

pad\_sequence() (in module pyllars.string\_utils), 53

pad\_trim\_sequences() (in module pyllars.string\_utils), 54

parse\_rdsamp\_datetime() (in module pyllars.physionet\_utils), 63

permute\_matrix() (in module pyllars.matrix\_utils), 17

plot\_binary\_prediction\_scores() (in module pyllars.mpl\_utils), 32

plot\_confusion\_matrix() (in module pyllars.mpl\_utils), 33

plot\_mean\_roc\_curve() (in module pyllars.mpl\_utils), 34

plot\_roc\_curve() (in module pyllars.mpl\_utils), 34

plot\_simple\_bar\_chart() (in module pyllars.mpl\_utils), 35

plot\_simple\_scatter() (in module pyllars.mpl\_utils), 37

plot\_sorted\_values() (in module pyllars.mpl\_utils), 37

plot\_stacked\_bar\_graph() (in module pyllars.mpl\_utils), 38

plot\_trend\_line() (in module pyllars.mpl\_utils), 38

plot\_venn\_diagram() (in module pyllars.mpl\_utils), 39

polynomial\_order (class in pyllars.stats\_utils), 51

precision\_at\_k() (in module pyllars.ml\_utils), 28

predictions\_test (pyllars.ml\_utils.estimators\_predictions\_metrics attribute), 25

predictions\_val (pyllars.ml\_utils.estimators\_predictions\_metrics attribute), 25

pyllars.collection\_utils (module), 4

pyllars.dask\_utils (module), 8

pyllars.logging\_utils (module), 13

pyllars.matrix\_utils (module), 16

pyllars.ml\_utils (module), 19

pyllars.mpl\_utils (module), 30

pyllars.pandas\_utils (module), 42

pyllars.physionet\_utils (module), 57

pyllars.stats\_utils (module), 47

pyllars.string\_utils (module), 52

**R**

read\_df() (in module pyllars.pandas\_utils), 45

remove\_nones() (in module pyllars.collection\_utils), 6

replace\_none\_with\_empty\_iter() (in module pyllars.collection\_utils), 7

reverse\_dict() (in module pyllars.collection\_utils), 7

row\_op() (in module pyllars.matrix\_utils), 17

row\_sum() (in module pyllars.matrix\_utils), 17

row\_sum\_mean() (in module pyllars.matrix\_utils), 17

**S**

set\_label\_fontsize() (in module pyllars.mpl\_utils), 39

set\_legend\_fontsize() (in module pyllars.mpl\_utils), 40

set\_legend\_title\_fontsize() (in module pyllars.mpl\_utils), 40

set\_logging\_values() (in module pyllars.logging\_utils), 15

set\_ticklabel\_rotation() (in module pyllars.mpl\_utils), 40

set\_ticklabels\_fontsize() (in module pyllars.mpl\_utils), 40

`set_title_fontsize()` (in module `pyllars.mpl_utils`), 40  
`simple_fill()` (in module `pyllars.string_utils`), 54  
`sort_dict_keys_by_value()` (in module `pyllars.collection_utils`), 7  
`sparse_matrix_to_dense()` (in module `pyllars.matrix_utils`), 18  
`sparse_matrix_to_list()` (in module `pyllars.matrix_utils`), 18  
`split()` (in module `pyllars.string_utils`), 54  
`split_df()` (in module `pyllars.pandas_utils`), 45  
`split_masks` (class in `pyllars.ml_utils`), 28  
`str2bool()` (in module `pyllars.string_utils`), 55  
`symmetric_entropy()` (in module `pyllars.stats_utils`), 51  
`symmetric_gaussian_kl()` (in module `pyllars.stats_utils`), 51

## T

`trim_sequence()` (in module `pyllars.string_utils`), 55  
`true_test` (`pyllars.ml_utils.estimators_predictions_metrics` attribute), 25  
`true_val` (`pyllars.ml_utils.estimators_predictions_metrics` attribute), 25  
`try_parse_float()` (in module `pyllars.string_utils`), 55  
`try_parse_int()` (in module `pyllars.string_utils`), 55

## U

`update_logging()` (in module `pyllars.logging_utils`), 15

## V

`VALID_AXIS_VALUES` (in module `pyllars.mpl_utils`), 30  
`VALID_WHICH_VALUES` (in module `pyllars.mpl_utils`), 30

## W

`wrap_in_list()` (in module `pyllars.collection_utils`), 7  
`wrap_in_set()` (in module `pyllars.collection_utils`), 7  
`wrap_string_in_list()` (in module `pyllars.collection_utils`), 7  
`write_df()` (in module `pyllars.pandas_utils`), 46  
`write_sparse_matrix()` (in module `pyllars.matrix_utils`), 18

## X

`X_AXIS_VALUES` (in module `pyllars.mpl_utils`), 30

## Y

`Y_AXIS_VALUES` (in module `pyllars.mpl_utils`), 30