hhpy

Release 0.1.6

Apr 22, 2020

Contents:

CHAPTER 1

python API reference

1.1 hhpy.main Module

1.1.1 hhpy.main.py

Contains basic calculation functions that are used in the more specialized versions of the package but can also be used on their own

1.1.2 Functions

today(date_format)	Returns today's date as string
size(byte, unit, dec)	Formats bytes as human readable string
mem_usage(pandas_obj, *args, **kwargs)	Get memory usage of a pandas object
tprint(*args, sep, r_loc, **kwargs)	Wrapper for print() but with a carriage return at the end.
fprint(*args, file, sep, mode, append_sep, ...)	Write the output of print to a file instead.
elapsed_time_init()	Resets reference time for elapsed_time()
elapsed_time(do_return, ref_t)	Get the elapsed time since reference time ref_time.
total_time(i, i_max)	Estimates total time of running operation by linear extrapolation using iteration counters.
remaining_time(i, i_max)	Estimates remaining time of running operation by linear extrapolation using iteration counters.
progressbar(i, i_max, symbol, empty_symbol, ...)	Prints a progressbar for the currently running process based on iteration counters.
time_to_str(t, time_format)	Wrapper for strftime
cf_vec(x, func, to_list, *args, **kwargs)	Pandas compatible vectorize function.
round_signif_i(x, digits)	Round to significant number of digits for a Scalar number
round_signif(x, *args, **kwargs)	Round to significant number of digits
floor_signif(x, digits)	Floor to significant number of digits

Continued on next page

Table 1 – continued from previous page

ceil_signif(x, digits)	Ceil to significant number of digits
concat_cols(df, columns, sep, to_int)	Concat a number of columns of a pandas DataFrame
list_unique(lst)	Returns unique elements from a list (dropping duplicates)
list_duplicate(lst)	Returns only duplicate elements from a list
list_flatten(lst)	Flatten a list of lists
list_merge(*args, unique, flatten)	Merges n lists together
list_intersection(lst, int, float, str, ...)	Returns common elements of n lists
list_exclude(lst, int, float, str, bytes, ...)	Returns a list that includes only those elements from the first list that are not in any subsequent list.
rand(shape, lower, upper, step, seed)	A seedable wrapper for numpy.random.random_sample that allows for boundaries and steps
dict_list(*args, dict_type)	Creates a dictionary of empty named lists.
append_to_dict_list(dct, ...)	Appends to a dictionary of named lists.
is_scalar(obj)	Checks if a given python object is scalar, i.e.
is_list_like(obj)	Checks if a given python object is list like.
assert_list(*args, default, int, float, str, ...)	Takes any python object(s) and turns them into an iterable list.
assert_tuple(*args, **kwargs)	Takes any python object(s) and turns them into an iterable tuple.
assert_scalar(obj, warn, default, float, ...)	Takes any python object and turns it into a scalar object.
qformat(value, int_format, float_format, ...)	Creates a human readable representation of a generic python object
to_hdf(df, file, groupby, List[str]] = None, ...)	saves a pandas DataFrame as h5 file, if groupby is supplied will save each group with a different key.
get_hdf_keys(file)	Reads all keys from an hdf file and returns as list
read_hdf(file, key, List[str]] = None, ...)	read a DataFrame from hdf file based on pandas.read_hdf but with default option to read all keys (since we're
rounddown(x, digits)	convenience wrapper for np.floor with digits option
roundup(x, digits)	convenience wrapper for np.ceil with digits option
reformat_string(string, case, replace, ...)	Function to quickly reformat a string to a specific convention.
dict_inv(dct, VT_co], key_as_str, duplicates)	Returns an inverted copy of a given dictionary (if it is invertible)
copy_function(f)	return a copy of a function, based on this StackOverflow answer https://stackoverflow.com/questions/13503079/how-to-create-a-copy-of-a-python-function
get_else_key(dct, VT_co], key, exclude, int, ...)	Returns a value from a dictionary if the key is present, if not returns the key

today

hhpy.main.**today**(*date_format: str = '%Y_%m_%d'*) → str

Returns today's date as string

Parameters **date_format** – The formating string for the date. Passed to strftime

Returns Formated String

Examples

```
>>> today()
'2020_01_14'
```

size

hhpy.main.**size**(*byte: int*, *unit: str = 'MB'*, *dec: int = 2*) → str

Formats bytes as human readable string

Parameters

- **byte** – The byte amount to be formated
- **unit** – The unit to display the output in, supports 'KB', 'MB', 'GB' and 'TB'
- **dec** – The number of decimals to use

Returns Formated bytes as string

Examples

```
>>> size(1024, unit='KB')
'1.0 KB'
```

```
>>> size(1024*1024*10, unit='MB')
'10.0 MB'
```

```
>>> size(10**10, unit='GB')
'9.31 GB'
```

mem_usage

hhpy.main.**mem_usage**(*pandas_obj*, **args*, ***kwargs*) → str

Get memory usage of a pandas object

Parameters

- **pandas_obj** – Pandas object to get the memory usage of
- **args** – passed to size()
- **kwargs** – passed to size()

Returns memory usage of a pandas object formated as string

Examples

```
>>> import seaborn as sns
>>> diamonds = sns.load_dataset('diamonds')
>>> mem_usage(diamonds)
'12.62 MB'
```

tprint

hhpy.main.**tprint**(**args*, *sep: str = ' '*, *r_loc: str = 'front'*, ***kwargs*)

Wrapper for print() but with a carriage return at the end. This results in the text being overwritten by the next print call. Can be used for progress bars and the like.

Parameters

- **args** – arguments to print
- **sep** – separator
- **r_loc** – where to put the carriage return, one of ['front', 'end']. Some interpreters (e.g. PyCharm) don't like end since they automatically clear the print area after each carriage return. When using front a regular print after a tprint will start at the end of the tprint. When using 'end' a regular print will overwrite the tprint output but will not clear the console so if it is . In either case a blank tprint() will clear the console and restore default print behaviour.
- **kwargs** – passed to print

Returns None

Examples

```
>>> tprint('Hello World')
'Hello World'
```

```
>>> tprint(1)
>>> tprint(2)
2
```

fprint

hhpy.main.**fprint**(**args*, *file: str = '_fprint.txt'*, *sep: str = ' '*, *mode: str = 'replace'*, *append_sep: str = '\n'*, *timestamp: bool = True*, *do_print: bool = False*, *do_tprint: bool = False*)

Write the output of print to a file instead. Supports also writing to console.

Parameters

- **args** – the arguments to print
- **file** – the name of the file to print to
- **sep** – separator
- **mode** – weather to append or replace the contents of the file
- **append_sep** – if mode=='append', use this separator
- **timestamp** – weather to include a timestamp in the print statement
- **do_print** – weather to also print to console
- **do_tprint** – weather to also print to console using tprint

Returns None

Examples

The below output gets written to a file called 'fprint.txt'

```
>>> fprint('Hello World', file='fprint.txt')
```

The below output gets written both to a file and to console

```
>>> fprint('Hello World', file='fprint.txt', do_print=True)
'Hello World'
```

elapsed_time_init

hhpy.main.**elapsed_time_init**() → None
Resets reference time for elapsed_time()

Returns None

Examples

see *elapsed_time()*

elapsed_time

hhpy.main.**elapsed_time**(*do_return: bool = True*, *ref_t: datetime.datetime = None*) → datetime.timedelta
Get the elapsed time since reference time ref_time.

Parameters

- **do_return** – Whether to return or print
- **ref_t** – Reference time. If None is provided the time elapsed_time_init() was last called is used.

Returns In case of do_return: Datetime object containing the elapsed time. Else calls tprint and returns None.

Examples

```
>>> from time import sleep
>>> elapsed_time_init()
>>> sleep(1)
>>> elapsed_time(do_return=False)
'0:00:01.0'
```

```
>>> from time import sleep
>>> elapsed_time_init()
>>> sleep(1)
>>> elapsed_time(do_return=True)
datetime.timedelta(0, 1, 1345)
```

total_time

hhpy.main.**total_time**(*i: int*, *i_max: int*) → datetime.timedelta
Estimates total time of running operation by linear extrapolation using iteration counters.

Parameters

- **i** – current iteration
- **i_max** – max iteration

Returns datetime object representing estimated total time of operation

remaining_time

hhpy.main.**remaining_time**(*i: int*, *i_max: int*) → datetime.timedelta
Estimates remaining time of running operation by linear extrapolation using iteration counters.

Parameters

- **i** – current iteration
- **i_max** – max iteration

Returns datetime object representing estimated remaining time of operation

progressbar

hhpy.main.**progressbar**(*i: int = 1*, *i_max: int = 1*, *symbol: str = '='*, *empty_symbol: str = '_'*, *mid: str = None*, *mode: str = 'perc'*, *print_prefix: str = ''*, *p_step: int = 1*, *printf: Callable = <function tprint>*, *persist: bool = False*, ***kwargs*)

Prints a progressbar for the currently running process based on iteration counters.

Parameters

- **i** – current iteration
- **i_max** – max iteration
- **symbol** – symbol that represents reached progress blocks
- **empty_symbol** – symbol that represents not yet reached progress blocks
- **mid** – what to write in the middle of the progressbar, if mid is passed mode is ignored
- **mode** – One of ['perc', 'remaining', 'elapsed'] If perc is passed writes percentage. If 'remaining' or 'elapsed' writes remaining or elapsed time respectively. [optional]
- **print_prefix** – what to write in front of the progressbar. Useful when calling progressbar multiple times from different functions.
- **p_step** – progressbar prints one symbol (progress block) per p_step
- **printf** – Using tprint by default. Use fprint to write to file instead.
- **persist** – Whether to persist the progressbar after reaching 100 percent.
- **kwargs** – Passed to print function

Returns

time_to_str

hhpy.main.**time_to_str**(*t: datetime.datetime*, *time_format: str = '%Y-%m-%d'*) → str

Wrapper for strftime

Parameters

- **t** – datetime object
- **time_format** – time format, passed to strftime

Returns formated datetime as string

cf_vec

hhpy.main.**cf_vec**(*x: Any*, *func: Callable*, *to_list: bool = True*, **args*, ***kwargs*) → Any

Pandas compatible vectorize function. In case a DataFrame is passed the function is applied to all columns.

Parameters

- **x** – Any vector like object
- **func** – Any function that should be vectorized
- **to_list** – Whether to cast the output to a list
- **args** – passed to func
- **kwargs** – passed to func

Returns Vector like object

round_signif_i

hhpy.main.**round_signif_i**(*x: numpy.number*, *digits: int = 1*) → float
Round to significant number of digits for a Scalar number

Parameters

- **x** – any number
- **digits** – integer amount of significant digits

Returns float rounded to significant digits

round_signif

hhpy.main.**round_signif**(*x: Any*, **args*, ***kwargs*) → Any
Round to significant number of digits

Parameters

- **x** – any vector like object of numbers
- **args** – passed to cf_vec
- **kwargs** – passed to cf_vec

Returns Vector like object of floats rounded to significant digits

floor_signif

hhpy.main.**floor_signif**(*x: Any*, *digits: int = 1*) → Any
Floor to significant number of digits

Parameters

- **x** – any vector like object of numbers
- **digits** – integer amount of significant digits

Returns float floored to significant digits

ceil_signif

hhpy.main.**ceil_signif**(*x: Any*, *digits: int = 1*) → Any
Ceil to significant number of digits

Parameters

- **x** – any vector like object of numbers

- **digits** – integer amount of significant digits

Returns float ceiled to significant digits

concat_cols

hhpy.main.**concat_cols**(*df: pandas.core.frame.DataFrame*, *columns: list*, *sep: str = '_'*, *to_int: bool = False*) → pandas.core.series.Series

Concat a number of columns of a pandas DataFrame

Parameters

- **df** – Pandas DataFrame
- **columns** – Names of the columns to be concat
- **sep** – Separator
- **to_int** – If true: Converts columns to int before concatting

Returns Pandas Series containing the concat columns

list_unique

hhpy.main.**list_unique**(*lst: Any*) → list

Returns unique elements from a list (dropping duplicates)

Parameters **lst** – any list like object

Returns list containing each element only once

list_duplicate

hhpy.main.**list_duplicate**(*lst: Any*) → list

Returns only duplicate elements from a list

Parameters **lst** – any list like object

Returns list of duplicates values

list_flatten

hhpy.main.**list_flatten**(*lst: Any*) → list

Flatten a list of lists

Parameters **lst** – list of lists

Returns flattened list

list_merge

hhpy.main.**list_merge**(**args*, *unique: bool = True*, *flatten: bool = False*) → list

Merges n lists together

Parameters

- **args** – The lists to be merged together
- **unique** – if True then duplicate elements will be dropped

- **flatten** – if True then the individual lists will be flatten before merging

Returns The merged list

list_intersection

hhpy.main.**list_intersection**(*lst: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]], *args*) → list

Returns common elements of n lists

Parameters

- **lst** – the first list
- **args** – the subsequent lists

Returns the list of common elements

list_exclude

hhpy.main.**list_exclude**(*lst: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]], *args*) → list

Returns a list that includes only those elements from the first list that are not in any subsequent list. Can also be called with non list args, then those elements are removed.

Parameters

- **lst** – the list to exclude from
- **args** – the subsequent lists

Returns the filtered list

rand

hhpy.main.**rand**(*shape: tuple = None*, *lower: int = None*, *upper: int = None*, *step: int = None*, *seed: int = None*) → numpy.array

A seedable wrapper for numpy.random.random_sample that allows for boundaries and steps

Parameters

- **shape** – A tuple containing the shape of the desired output array
- **lower** – Lower bound of random numbers
- **upper** – Upper bound of random numbers
- **step** – Minimum step between random numbers
- **seed** – Random Seed

Returns Numpy Array

dict_list

hhpy.main.**dict_list**(**args*, *dict_type: str = 'defaultdict'*) → dict

Creates a dictionary of empty named lists. Useful for iteratively creating a pandas DataFrame

Parameters

- **args** – The names of the lists

- **dict_type** – Whether to use a 'regular' or 'defaultdict' (default to empty list) type dictionary

Returns Dictionary of empty named lists

append_to_dict_list

hhpy.main.**append_to_dict_list**(*dct: Union[dict, collections.defaultdict], append: Union[dict, list], inplace: bool = True*) → Optional[dict]

Appends to a dictionary of named lists. Useful for iteratively creating a pandas DataFrame.

Parameters

- **dct** – Dictionary to append to
- **append** – List or dictionary of values to append
- **inplace** – Modify inplace or return modified copy

Returns None if inplace, else modified dictionary

is_scalar

hhpy.main.**is_scalar**(*obj: Any*) → bool

Checks if a given python object is scalar, i.e. one of int, float, str, bytes

Parameters **obj** – Any python object

Returns True if scaler, else False

is_list_like

hhpy.main.**is_list_like**(*obj: Any*) → bool

Checks if a given python object is list like. The conditions must be satisfied:

- not a string or bytes object
- one of (Sequence, 1d-array like Iterable)

Parameters **obj** – Any python object

Returns True if list like, else False

assert_list

hhpy.main.**assert_list**(**args, default: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None*) → list

Takes any python object(s) and turns them into an iterable list.

Parameters

- **args** – Any python object
- **default** – What to return if args are Empty or None

Returns List

assert_tuple

hhpy.main.**assert_tuple**(**args*, ***kwargs*) → tuple
Takes any python object(s) and turns them into an iterable tuple.

Parameters

- **args** – Any python object
- **kwargs** – Keyword arguments passed to :~func: assert_list

Returns List

assert_scalar

hhpy.main.**assert_scalar**(*obj: Any*, *warn: bool = True*, *default: Union[int, float, str, bytes, None] = None*) → Union[int, float, str, bytes, None]
Takes any python object and turns it into a scalar object.

Parameters

- **obj** – Any python object
- **warn** – Whether to trigger a warning when objects are being truncated
- **default** – What to return if obj is None

Returns List

qformat

hhpy.main.**qformat**(*value: Any*, *int_format: str = ', '*, *float_format: str = '.2f'*, *datetime_format: str = '%Y-%m-%d'*, *sep: str = ' - '*, *key_sep: str = ': '*, *print_key: bool = True*) → str
Creates a human readable representation of a generic python object

Parameters

- **value** – Any python object
- **int_format** – Format string for integer
- **float_format** – Format string for float
- **datetime_format** – Format string for datetime
- **sep** – Separator
- **key_sep** – Separator used between key and value if print_key is True
- **print_key** – Whether to print keys as well as values (if object has keys)

Returns Formated string

to_hdf

hhpy.main.**to_hdf**(*df: pandas.core.frame.DataFrame*, *file: str*, *groupby: Union[str, List[str]] = None*, *write_groupby: bool = True*, *key: str = None*, *replace: bool = False*, *format: str = 'table'*, *do_print=True*, ***kwargs*) → None
saves a pandas DataFrame as h5 file, if groupby is supplied will save each group with a different key. Needs with groupby OR key to be supplied. Extends on pandas.DataFrame.to_hdf.

Parameters

- **df** – DataFrame to save
- **file** – filename to save the DataFrame as
- **groupby** – if supplied will save each sub-DataFrame as a different key [optional]
- **write_groupby** – Whether groupby columns should be written to hdf [optional]
- **key** – The key to write as. Ignored if groupby is supplied [optional]
- **replace** – Whether to replace or append to existing files. Defaults to append [optional]
- **format** – Table format to use, passed to pandas.DataFrame.to_hdf. Defaults to 'table' while pandas defaults to 'fixed' [optional]
- **do_print** – Whether to print intermediate steps to console [optional]
- **kwargs** – Other keyword arguments passed to pandas.DataFrame.to_hdf [optional]

Returns None

get_hdf_keys

hhpy.main.**get_hdf_keys**(*file: str*) → List[str]

Reads all keys from an hdf file and returns as list

Parameters **file** – The path of the file to read the keys of

Returns List of keys

read_hdf

hhpy.main.**read_hdf**(*file: str*, *key: Union[str*, *List[str]] = None*, *sample: int = None*, *random_state: int = None*, *key_to_col: Union[bool*, *str] = False*, *do_print: bool = True*, *catch_error: bool = True*, ***kwargs*) → pandas.core.frame.DataFrame

read a DataFrame from hdf file based on pandas.read_hdf but with default option to read all keys (since we're expecting a DataFrame)

Parameters

- **file** – The path to the file to read from
- **key** – The key(s) to read, if not specified all keys are read [optional]
- **sample** – If specified will read sample keys at random from the file, ignored if key is specified [optional]
- **random_state** – Random state for sample [optional]
- **key_to_col** – Whether to save the key value to a column, if a string then used as column name [optional]
- **do_print** – Whether to print intermediate steps [optional]
- **catch_error** – Whether to catch errors when reading [optional]
- **kwargs** – Other keyword arguments passed to pandas.read_hdf [optional]

Returns pandas DataFrame

rounddown

hhpy.main.**rounddown**(*x: Any*, *digits: int*) → Any
convenience wrapper for np.floor with digits option

Parameters

- **x** – any python object that supports np.floor
- **digits** – amount of digits

Returns rounded x

roundup

hhpy.main.**roundup**(*x: Any*, *digits: int*) → Any
convenience wrapper for np.ceil with digits option

Parameters

- **x** – any python object that supports np.ceil
- **digits** – amount of digits

Returns rounded x

reformat_string

hhpy.main.**reformat_string**(*string: str*, *case: Optional[str] = 'lower'*, *replace: Optional[Mapping[str, str]] = None*, *lstrip: Optional[str] = ' '*, *rstrip: Optional[str] = ' '*, *demojize: bool = True*, *trans: bool = False*, *trans_dest: Optional[str] = 'en'*, *trans_src: Optional[str] = 'auto'*, *trans_sleep: Union[float, bool] = 0.4*, *warn: bool = True*) → str
Function to quickly reformat a string to a specific convention. The default convention is only lowercase, numbers and underscores. Also allows translation if optional dependency googletrans is installed.

Parameters

- **string** – input string to be reformatted
- **case** – casts string to specified case, one of ['lower', 'upper'] [optional]
- **replace** – Dictionary containing the replacements to be made passed to re.sub . Defaults to replacing any non [a-zA-Z0-9] string with '_'. Note that this means that special characters from other languages get replaced. If you don't want that set replace to False or specify your own mapping. Is applied **last** so make sure your conventions match [optional]
- **lstrip** – The leading characters to be removed, passed to string.lstrip [optional]
- **rstrip** – The training characters to be removed, passed to string.rstrip [optional]
- **demojize** – Whether to remove emojis using emoji.demojize [optional]
- **trans** – Whether to translate the string using googletrans.Translator.translate [optional]
- **trans_dest** – The language to translate from, passed to googletrans as dest=trans_dest [optional]
- **trans_src** – The language to translate to, passed to googletrans as src=trans_src [optional]

- **trans_sleep** – Amount of seconds to sleep before translating, should be at least .4 to avoid triggering google's rate limits. Set it to lower values / None / False for a speedup at your own risk [optional]
- **warn** – Whether to show UserWarnings triggered by this function. Set to False to suppress, other warnings will still be triggered [optional]

Returns reformatted string

dict_inv

hhpy.main.**dict_inv**(*dct: Mapping[KT, VT_co], key_as_str: bool = False, duplicates: str = 'keep'*) → dict

Returns an inverted copy of a given dictionary (if it is invertible)

Parameters

- **dct** – Dictionary to be inverted
- **key_as_str** – Whether all keys of the inverted dictionary should be forced to string
- **duplicates** – Whether to 'adjust' or 'drop' duplicates. In case of 'adjust' duplicates are suffixed with '_'

Returns Inverted dictionary

copy_function

hhpy.main.**copy_function**(*f: function*) → function

return a copy of a function, based on this StackOverflow answer https://stackoverflow.com/questions/13503079/how-to-create-a-copy-of-a-python-function

Parameters **f** – a function

Returns copy of function

get_else_key

hhpy.main.**get_else_key**(*dct: Mapping[KT, VT_co], key: Any, exclude: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None*) → Any

Returns a value from a dictionary if the key is present, if not returns the key

Parameters

- **dct** – dictionary or similar Mapping
- **key** – Key of value to attempt get
- **exclude** – Keys to not get the value from (always return as is)

Returns Value if key in dictionary keys, else key

1.1.3 Classes

BaseClass	Base class for various classes deriving from this.

BaseClass

class hhpy.main.**BaseClass**
Bases: object

Base class for various classes deriving from this. Implements __repr__, converting to dict as well as saving to pickle and restoring from pickle. Does NOT provide __init__ since it cannot be used by itself

Methods Summary

copy()	Uses copy.deepcopy to return a copy of the object
from_dict(dct, VT_co])	Restores self from a dictionary
load(filename, f)	Load self from file saved with *save()* using an arbitrary function that supports loading dictionaries.
read_pickle(*args, **kwargs)	Wrapper for *BaseClass.load()* using f = pandas.read_pickle
save(filename, f)	Save self to file using an arbitrary function that supports saving dictionaries.
to_dict()	Converts self to a dictionary
to_pickle(*args, **kwargs)	Wrapper for *save()* using f = pandas.to_pickle

Methods Documentation

copy()
Uses copy.deepcopy to return a copy of the object

Returns Copy of self

from_dict(*dct: Mapping[KT, VT_co]*)
Restores self from a dictionary

Parameters **dct** – Dictionary created from *to_dict()*

Returns None

load(*filename: str*, *f: Callable = <function read_pickle>*)
Load self from file saved with *save()* using an arbitrary function that supports loading dictionaries.

Parameters

- **filename** – filename (path) of the file
- **f** – function to be used [optional]

Returns None

read_pickle(**args*, ***kwargs*)
Wrapper for *BaseClass.load()* using f = pandas.read_pickle

Parameters

- **args** – passed to load [optional]
- **kwargs** – passed to load [optional]

Returns see load

save(*filename: str*, *f: Callable = <function to_pickle>*)
Save self to file using an arbitrary function that supports saving dictionaries. Note that the object is implicitly converted to a dictionary before saving.

Parameters

- **filename** – filename (path) to be used
- **f** – function to be used [optional]

Returns None

to_dict() → dict
Converts self to a dictionary

Returns Dictionary

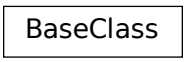
to_pickle(*args, **kwargs)
Wrapper for *save()* using f = pandas.to_pickle

Parameters

- **args** – passed to save [optional]
- **kwargs** – passed to save [optional]

Returns see save

1.1.4 Class Inheritance Diagram



1.2 hhpy.ds Module

1.2.1 hhpy.ds.py

Contains DataScience functions extending on pandas and sklearn

1.2.2 Functions

assert_df(df, groupby, int, float, str, ...)	assert that input is a pandas DataFrame, raise ValueError if it cannot be cast to DataFrame
optimize_pd(df, c_int, c_float, c_cat, ...)	optimize memory usage of a pandas df, automatically downcast all var types and converts objects to categories
get_df_corr(df, columns, target, groupby, ...)	Calculate Pearson Correlations for numeric columns, extends on pandas.DataFrame.corr but automatically melts the output.
drop_zero_cols(df)	Drop columns with all 0 or None Values from DataFrame.
get_duplicate_indices(df)	Returns duplicate indices from a pandas DataFrame

Continued on next page

Table 4 – continued from previous page

get_duplicate_cols(df)	Returns names of duplicate columns from a pandas DataFrame
drop_duplicate_indices(df, warn)	Drop duplicate indices from pandas DataFrame
drop_duplicate_cols(df, warn)	Drop duplicate columns from pandas DataFrame
change_span(s, steps)	return a True/False series around a changepoint, used for filtering stepwise data series in a pandas df must be properly sorted!
outlier_to_nan(df, col, groupby, ...)	this algorithm cuts off all points whose DELTA (avg diff to the prev and next point) is outside of the n std range
butter_pass_filter(data, cutoff, fs, order, ...)	Implementation of a highpass / lowpass filter using scipy.signal.butter
pass_by_group(df, col, groupby, list], ...)	allows applying a butter_pass filter by group
lfit(x, int, float, str, bytes, None, ...)	quick linear fit with numpy
rolling_lfit(x, int, float, str, bytes, ...)	Rolling version of lfit: for each row of the DataFrame / Series look at the previous window rows, then perform an lfit and use this value as a prediction for this row.
qf(df, fltr, pandas.core.series.Series, ...)	quickly filter a DataFrame based on equal criteria.
quantile_split(s, n, signif, na_to_med)	splits a numerical column into n quantiles.
acc(y_true, str], y_pred, str], df)	calculate accuracy for a categorical label
rel_acc(y_true, str], y_pred, str], df, ...)	relative accuracy of the prediction in comparison to predicting everything as the most common group :param y_true: true values as name of df or vector data :param y_pred: predicted values as name of df or vector data :param df: pandas DataFrame containing true and predicted values [optional] :param target_class: name of the target class, by default the most common one is used [optional] :return: accuracy difference as percent
cm(y_true, str], y_pred, str], df)	confusion matrix from pandas df :param y_true: true values as name of df or vector data :param y_pred: predicted values as name of df or vector data :param df: pandas DataFrame containing true and predicted values [optional] :return: Confusion matrix as pandas DataFrame
f1_pr(y_true, str], y_pred, str], df, ...)	get f1 score, true positive, true negative, missed positive and missed negative rate
f_score(y_true, str], y_pred, str], df, ...)	generic scoring function base on pandas DataFrame.
r2(*args, **kwargs)	wrapper for f_score using sklearn.metrics.r2_score
rmse(*args, **kwargs)	wrapper for f_score using numpy.sqrt(skearn.metrics.mean_squared_error)
mae(*args, **kwargs)	wrapper for f_score using skearn.metrics.mean_absolute_error
stdae(*args, **kwargs)	wrapper for f_score using the standard deviation of the absolute error
medae(*args, **kwargs)	wrapper for f_score using skearn.metrics.median_absolute_error
pae(*args, times_hundred, pmax, **kwargs)	wrapper for f_score using percentage absolute error
corr(*args, **kwargs)	wrapper for f_score using pandas.Series.corr
df_score(df, y_true, int, float, str, bytes, ...)	creates a DataFrame displaying various kind of scores
rmsd(x, df, group, return_df_paired, ...)	calculated the weighted root mean squared difference for a reference columns x by a specific group.

Continued on next page

Table 4 – continued from previous page

df_rmsd(x, df, groups, str] = None, hue, ...)	calculate *rmsd()* for reference column x with multiple other columns and return as DataFrame.
df_p(x, group, df, hue, agg_func, agg, ...)	returns a DataFrame with the p value.
col_to_front(df, cols, float, str, bytes, ...)	Brings one or more columns to the front (first n positions) of a DataFrame
df_split(df, split_by, str], return_type, ...)	Split a pandas DataFrame by column value and returns a list or dict
rank(df, rankby, int, float, str, bytes, ...)	creates a ranking (without duplicate ranks) based on columns of a DataFrame
mahalanobis(point, ...)	Calculates the Mahalanobis distance for a single point or a DataFrame of points
df_count(x, df, hue, sort_by_count, top_nr, ...)	Create a DataFrame of value counts.
top_n(s, n, str], w, n_max)	Select n elements form a categorical pandas series with the highest counts. Ties are broken by sorting
top_n_coding(s, n, other_name, na_to_other, ...)	Returns a modified version of the pandas series where all elements not in top_n become recoded as 'other'
k_split(df, k, groupby, str] = None, sortby, ...)	Splits a DataFrame into k (equal sized) parts that can be used for train test splitting or k_cross splitting
remove_unused_categories(df, inplace)	Remove unused categories from all categorical columns in the DataFrame
read_csv(path, nrows, encoding, errors, ...)	wrapper for pandas.read_csv that reads the file into an IOString first.
get_columns(df, dtype, int, float, str, ...)	A quick way to get the columns of a certain dtype.
reformat_columns(df, printf, **kwargs)	A quick way to clean the column names of a DataFrame

assert_df

hhpy.ds.**assert_df**(*df: Any, groupby: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co], bool] = False, name: str = 'df'*) → Union[pandas.core.frame.DataFrame, Tuple[pandas.core.frame.DataFrame, List[T]]]

assert that input is a pandas DataFrame, raise ValueError if it cannot be cast to DataFrame

Parameters

- **df** – Object to be cast to DataFrame
- **groupby** – column to use as groupby
- **name** – name to use in the ValueError message, useful when calling from another function

Returns pandas DataFrame

optimize_pd

hhpy.ds.**optimize_pd**(*df: pandas.core.frame.DataFrame, c_int: bool = True, c_float: bool = True, c_cat: bool = True, cat_frac: float = 0.5, convert_dtypes: bool = True, drop_all_na_cols: bool = False*) → pandas.core.frame.DataFrame

optimize memory usage of a pandas df, automatically downcast all var types and converts objects to categories

Parameters

- **df** – pandas DataFrame to be optimized. Other objects are implicitly cast to DataFrame
- **c_int** – Whether to downcast integers [optional]

- **c_float** – Whether to downcast floats [optional]
- **c_cat** – Whether to cast objects to categories. Uses cat_frac as condition [optional]
- **cat_frac** – If c_cat: If the column has less than cat_frac percent unique values it will be cast to category [optional]
- **convert_dtypes** – Whether to call convert dtypes (pandas 1.0.0+) [optional]
- **drop_all_na_cols** – Whether to drop columns that contain only missing values [optional]

Returns the optimized pandas DataFrame

get_df_corr

hhpy.ds.**get_df_corr**(*df: pandas.core.frame.DataFrame*, *columns: List[str] = None*, *target: str = None*, *groupby: Union[str, list] = None*) → pandas.core.frame.DataFrame

Calculate Pearson Correlations for numeric columns, extends on pandas.DataFrame.corr but automatically melts the output. Used by *corrplot_bar()*

Parameters

- **df** – input pandas DataFrame. Other objects are implicitly cast to DataFrame
- **columns** – Column to calculate the correlation for, defaults to all numeric columns [optional]
- **target** – Returns only correlations that involve the target column [optional]
- **groupby** – Returns correlations for each level of the group [optional]

Returns pandas DataFrame containing all pearson correlations in a melted format

drop_zero_cols

hhpy.ds.**drop_zero_cols**(*df: pandas.core.frame.DataFrame*) → pandas.core.frame.DataFrame

Drop columns with all 0 or None Values from DataFrame. Useful after applying one hot encoding.

Parameters **df** – pandas DataFrame

Returns pandas DataFrame without 0 columns.

get_duplicate_indices

hhpy.ds.**get_duplicate_indices**(*df: pandas.core.frame.DataFrame*) → pandas.core.series.Series

Returns duplicate indices from a pandas DataFrame

Parameters **df** – pandas DataFrame

Returns List of indices that are duplicate

get_duplicate_cols

hhpy.ds.**get_duplicate_cols**(*df: pandas.core.frame.DataFrame*) → pandas.core.series.Series

Returns names of duplicate columns from a pandas DataFrame

Parameters **df** – pandas DataFrame

Returns List of column names that are duplicate

drop_duplicate_indices

hhpy.ds.**drop_duplicate_indices**(*df: pandas.core.frame.DataFrame*, *warn: bool = True*) → pandas.core.frame.DataFrame

Drop duplicate indices from pandas DataFrame

Parameters

- **df** – pandas DataFrame
- **warn** – Whether to trigger a warning if duplicate indices are dropped

Returns pandas DataFrame without the duplicates indices

drop_duplicate_cols

hhpy.ds.**drop_duplicate_cols**(*df: pandas.core.frame.DataFrame*, *warn: bool = True*) → pandas.core.frame.DataFrame

Drop duplicate columns from pandas DataFrame

Parameters

- **df** – pandas DataFrame
- **warn** – Whether to trigger a warning if duplicate columns are dropped

Returns pandas DataFrame without the duplicates columns

change_span

hhpy.ds.**change_span**(*s: pandas.core.series.Series*, *steps: int = 5*) → pandas.core.series.Series

return a True/False series around a changepoint, used for filtering stepwise data series in a pandas df must be properly sorted!

Parameters

- **s** – pandas Series or similar
- **steps** – number of steps around the changepoint to flag as true

Returns pandas Series of dtype Boolean

outlier_to_nan

hhpy.ds.**outlier_to_nan**(*df: pandas.core.frame.DataFrame*, *col: str*, *groupby: Union[list, str] = None*, *std_cutoff: numpy.number = 3*, *reps: int = 1*, *do_print: bool = False*) → pandas.core.frame.DataFrame

this algorithm cuts off all points whose DELTA (avg diff to the prev and next point) is outside of the n std range

Parameters

- **df** – pandas DataFrame
- **col** – column to be filtered
- **groupby** – if provided: applies std filter by group
- **std_cutoff** – the number of standard deviations outside of which to set values to None
- **reps** – how many times to repeat the algorithm
- **do_print** – whether to print steps to console

Returns pandas Series with outliers set to nan

butter_pass_filter

hhpy.ds.**butter_pass_filter**(*data: pandas.core.series.Series*, *cutoff: int*, *fs: int*, *order: int*, *btype: str = None*, *shift: bool = False*)

Implementation of a highpass / lowpass filter using scipy.signal.butter

Parameters

- **data** – pandas Series or 1d numpy Array
- **cutoff** – cutoff
- **fs** – critical frequencies
- **order** – order of the fit
- **btype** – The type of filter. Passed to scipy.signal.butter. Default is 'lowpass'. One of {'lowpass', 'highpass', 'bandpass', 'bandstop'}
- **shift** – whether to shift the data to start at 0

Returns 1d numpy array containing the filtered data

pass_by_group

hhpy.ds.**pass_by_group**(*df: pandas.core.frame.DataFrame*, *col: str*, *groupby: Union[str, list]*, *btype: str*, *shift: bool = False*, *cutoff: int = 1*, *fs: int = 20*, *order: int = 5*)

allows applying a butter_pass filter by group

Parameters

- **df** – pandas DataFrame
- **col** – column to filter
- **groupby** – columns to groupby
- **btype** – The type of filter. Passed to scipy.signal.butter. Default is 'lowpass'. One of {'lowpass', 'highpass', 'bandpass', 'bandstop'}
- **shift** – shift: whether to shift the data to start at 0
- **cutoff** – cutoff
- **fs** – critical frequencies
- **order** – order of the filter

Returns filtered DataFrame

lfit

hhpy.ds.**lfit**(*x: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]]*, *y: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None*, *w: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None*, *df: pandas.core.frame.DataFrame = None*, *groupby: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None*, *do_print: bool = True*, *catch_error: bool = False*, *return_df: bool = False*, *extrapolate: int = None*) → Union[pandas.core.series.Series, pandas.core.frame.DataFrame]

quick linear fit with numpy

Parameters

- **x** – names of x variables in df or vector data, if y is None treated as target and fit against the index
- **y** – names of y variables in df or vector data [optional]
- **w** – names of weight variables in df or vector data [optional]
- **df** – pandas DataFrame containing x,y,w data [optional]
- **groupby** – If specified the linear fit is applied by group [optional]
- **do_print** – whether to print steps to console
- **catch_error** – whether to keep going in case of error [optional]
- **return_df** – whether to return a DataFrame or Series [optional]
- **extrapolate** – how many iteration to extrapolate [optional]

Returns if return_df is True: pandas DataFrame, else: pandas Series

rolling_lfit

hhpy.ds.**rolling_lfit**(*x: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]], window: int, df: pandas.core.frame.DataFrame = None, groupby: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None*)

Rolling version of lfit: for each row of the DataFrame / Series look at the previous window rows, then perform an lfit and use this value as a prediction for this row. Useful as naive predictor for time series Data.

Parameters

- **x** – Main variable, name of a column in the DataFrame or vector data
- **window** – Size of the rolling window, see pandas.Series.rolling [optional]
- **df** – Pandas DataFrame containing the data, other objects are implicitly cast to DataFrame

:param groupby:The columns used for grouping, passed to pandas.DataFrame.groupby [optional] :return: pandas Series containing the fitted values

qf

hhpy.ds.**qf**(*df: pandas.core.frame.DataFrame, fltr: Union[pandas.core.frame.DataFrame, pandas.core.series.Series, Mapping[KT, VT_co]], rem_unused_categories: bool = True, reset_index: bool = False*)

quickly filter a DataFrame based on equal criteria. All columns of fltr present in df are filtered to be equal to the first entry in filter_df.

Parameters

- **df** – pandas DataFrame to be filtered
- **fltr** – filter condition as DataFrame or Mapping or Series
- **rem_unused_categories** – whether to remove unused categories from categorical dtype after filtering
- **reset_index** – whether to reset index after filtering

Returns filtered pandas DataFrame

quantile_split

hhpy.ds.**quantile_split**(*s: pandas.core.series.Series*, *n: int*, *signif: int = 2*, *na_to_med: bool = False*)

splits a numerical column into n quantiles. Useful for mapping numerical columns to categorical columns

Parameters

- **s** – pandas Series to be split
- **n** – number of quantiles to split into
- **signif** – number of significant digits to round to
- **na_to_med** – whether to fill na values with median values

Returns pandas Series of dtype category

acc

hhpy.ds.**acc**(*y_true: Union[pandas.core.series.Series, str]*, *y_pred: Union[pandas.core.series.Series, str]*, *df: pandas.core.frame.DataFrame = None*) → float

calculate accuracy for a categorical label

Parameters

- **y_true** – true values as name of df or vector data
- **y_pred** – predicted values as name of df or vector data
- **df** – pandas DataFrame containing true and predicted values [optional]

Returns accuracy a percentage

rel_acc

hhpy.ds.**rel_acc**(*y_true: Union[pandas.core.series.Series, str]*, *y_pred: Union[pandas.core.series.Series, str]*, *df: pandas.core.frame.DataFrame = None*, *target_class: str = None*)

relative accuracy of the prediction in comparison to predicting everything as the most common group :param y_true: true values as name of df or vector data :param y_pred: predicted values as name of df or vector data :param df: pandas DataFrame containing true and predicted values [optional] :param target_class: name of the target class, by default the most common one is used [optional] :return: accuracy difference as percent

cm

hhpy.ds.**cm**(*y_true: Union[pandas.core.series.Series, str]*, *y_pred: Union[pandas.core.series.Series, str]*, *df: pandas.core.frame.DataFrame = None*) → pandas.core.frame.DataFrame

confusion matrix from pandas df :param y_true: true values as name of df or vector data :param y_pred: predicted values as name of df or vector data :param df: pandas DataFrame containing true and predicted values [optional] :return: Confusion matrix as pandas DataFrame

f1_pr

hhpy.ds.**f1_pr**(*y_true: Union[pandas.core.series.Series, str]*, *y_pred: Union[pandas.core.series.Series, str]*, *df: pandas.core.frame.DataFrame = None*, *target: str = None*, *factor: int = 100*) → pandas.core.frame.DataFrame

get f1 score, true positive, true negative, missed positive and missed negative rate

Parameters

- **y_true** – true values as name of df or vector data
- **y_pred** – predicted values as name of df or vector data
- **df** – pandas DataFrame containing true and predicted values [optional]
- **target** – level for which to return the rates, by default all levels are returned [optional]
- **factor** – factor by which to scale results, default 100 [optional]

Returns pandas DataFrame containing f1 score, true positive, true negative, missed positive and missed negative rate

f_score

hhpy.ds.**f_score**(*y_true: Union[pandas.core.series.Series, str], y_pred: Union[pandas.core.series.Series, str], df: pandas.core.frame.DataFrame = None, dropna: bool = True, f: Callable = <function r2_score>, groupby: Union[list, str] = None, f_name: str = None*) → Union[pandas.core.frame.DataFrame, float]

generic scoring function base on pandas DataFrame.

Parameters

- **y_true** – true values as name of df or vector data
- **y_pred** – predicted values as name of df or vector data
- **df** – pandas DataFrame containing true and predicted values [optional]
- **dropna** – whether to dropna values [optional]
- **f** – scoreing function to apply, default is sklearn.metrics.r2_score, should return a scalar value. [optional]
- **groupby** – if supplied then the result is returned for each group level [optional]
- **f_name** – name of the scoreing function, by default uses .__name__ property of function [optional]

Returns if groupby is supplied: pandas DataFrame, else: scalar value

r2

hhpy.ds.**r2**(**args*, ***kwargs*) → Union[pandas.core.frame.DataFrame, float]

wrapper for f_score using sklearn.metrics.r2_score

Parameters

- **args** – passed to f_score
- **kwargs** – passed to f_score

Returns if groupby is supplied: pandas DataFrame, else: scalar value

rmse

hhpy.ds.**rmse**(**args*, ***kwargs*) → Union[pandas.core.frame.DataFrame, float]

wrapper for f_score using numpy.sqrt(skearn.metrics.mean_squared_error)

Parameters

- **args** – passed to f_score
- **kwargs** – passed to f_score

Returns if groupby is supplied: pandas DataFrame, else: scalar value

mae

hhpy.ds.**mae**(**args*, ***kwargs*) → Union[pandas.core.frame.DataFrame, float]
wrapper for f_score using skearn.metrics.mean_absolute_error

Parameters

- **args** – passed to f_score
- **kwargs** – passed to f_score

Returns if groupby is supplied: pandas DataFrame, else: scalar value

stdae

hhpy.ds.**stdae**(**args*, ***kwargs*) → Union[pandas.core.frame.DataFrame, float]
wrapper for f_score using the standard deviation of the absolute error

Parameters

- **args** – passed to f_score
- **kwargs** – passed to f_score

Returns if groupby is supplied: pandas DataFrame, else: scalar value

medae

hhpy.ds.**medae**(**args*, ***kwargs*) → Union[pandas.core.frame.DataFrame, float]
wrapper for f_score using skearn.metrics.median_absolute_error

Parameters

- **args** – passed to f_score
- **kwargs** – passed to f_score

Returns if groupby is supplied: pandas DataFrame, else: scalar value

pae

hhpy.ds.**pae**(**args*, *times_hundred: bool = True*, *pmax: int = 999*, ***kwargs*) → Union[pandas.core.frame.DataFrame, float]
wrapper for f_score using percentage absolute error

Parameters

- **args** – passed to f_score
- **times_hundred** – Whether to multiply by 100 for human readable percentages
- **pmax** – Max value for the percentage absolute error, used as a fallback because pae can go to infinity as y_true approaches zero
- **kwargs** – passed to f_score

Returns if groupby is supplied: pandas DataFrame, else: scalar value

corr

hhpy.ds.**corr**(**args*, ***kwargs*) → Union[pandas.core.frame.DataFrame, float]
wrapper for f_score using pandas.Series.corr

Parameters

- **args** – passed to f_score
- **kwargs** – passed to f_score

Returns if groupby is supplied: pandas DataFrame, else: scalar value

df_score

hhpy.ds.**df_score**(*df: pandas.core.frame.DataFrame, y_true: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]], y_pred: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, pred_suffix: list = None, scores: List[Callable] = None, pivot: bool = True, scale: Union[dict, list, int] = None, groupby: Union[list, str] = None, multi: int = None, dropna: bool = True*) → pandas.core.frame.DataFrame
creates a DataFrame displaying various kind of scores

Parameters

- **df** – pandas DataFrame containing the true, pred data
- **y_true** – name of the true variable(s) inside df
- **y_pred** – name of the pred variable(s) inside df, specify either this or pred_suffix
- **pred_suffix** – name of the predicted variable suffixes. Supports multiple predictions. By default assumed suffix 'pred' [optional]
- **scores** – scoring functions to be used [optional]
- **pivot** – whether to pivot the DataFrame for easier readability [optional]
- **scale** – a scale for multiplying the scores, default 1 [optional]
- **groupby** – if supplied then the scores are calculated by group [optional]
- **multi** – how many multi outputs are there [optional]
- **dropna** – whether to drop na [optional]

Returns pandas DataFrame containing al the scores

rmsd

hhpy.ds.**rmsd**(*x: str, df: pandas.core.frame.DataFrame, group: str, return_df_paired: bool = False, agg_func: str = 'median', standardize: bool = False, to_abs: bool = False*) → Union[float, pandas.core.frame.DataFrame]
calculated the weighted root mean squared difference for a reference columns x by a specific group. For a multi group DataFrame see *df_rmsd()*. For a plot see *hhpy.plotting.rmsdplot()*

Parameters

- **x** – name of the column to calculate the rmsd for

- **df** – pandas DataFrame
- **group** – groups for which to calculate the rmsd
- **return_df_paired** – whether to return the paired DataFrame
- **agg_func** – which aggregation to use for the group value, passed to pd.DataFrame.agg
- **standardize** – whether to apply Standardization before calculating the rmsd
- **to_abs** – whether to cast x to abs before calculating the rmsd

Returns if return_df_paired pandas DataFrame, else rmsd as float

Examples

Check out the example notebook

df_rmsd

hhpy.ds.**df_rmsd**(*x: str*, *df: pandas.core.frame.DataFrame*, *groups: Union[list, str] = None*, *hue: str = None*, *hue_order: list = None*, *sort_by_hue: bool = True*, *n_quantiles: int = 10*, *signif: int = 2*, *include_rmsd: bool = True*, ***kwargs*) → pandas.core.frame.DataFrame

calculate *rmsd()* for reference column x with multiple other columns and return as DataFrame. For a plot see *rmsdplot()*

Parameters

- **x** – name of the column to calculate the rmsd for
- **df** – pandas DataFrame containing the data
- **groups** – groups to calculate the rmsd or, defaults to all other columns in the DataFrame [optional]
- **hue** – further calculate the rmsd for each hue level [optional]
- **hue_order** – sort the hue levels in this order [optional]
- **sort_by_hue** – sort the values by hue rather than by group [optional]
- **n_quantiles** – numeric columns will be automatically split into this many quantiles [optional]
- **signif** – how many significant digits to use in quantile splitting [optional]
- **include_rmsd** – if False provide only a grouped DataFrame but don't actually calculate the rmsd, you can use include_rmsd=False to save computation time if you only need the maxperc (used in plotting)
- **kwargs** – passed to *rmsd()*

Returns None

Examples

Check out the example notebook

df_p

hhpy.ds.**df_p**(*x: str*, *group: str*, *df: pandas.core.frame.DataFrame*, *hue: str = None*, *agg_func: str = 'mean'*, *agg: bool = False*, *n_quantiles: int = 10*)

returns a DataFrame with the p value. See hypothesis testing. :param x: name of column to evaluate :param group: name of grouping column :param df: pandas DataFrame :param hue: further split by hue level :param

agg_func: standard agg function, passed to pd.DataFrame.agg :param agg: whether to include standard aggregation :param n_quantiles: numeric columns will be automatically split into this many quantiles [optional] :return: pandas DataFrame containing p values

col_to_front

hhpy.ds.**col_to_front**(*df: pandas.core.frame.DataFrame, cols: Union[Sequence[Union[int, float, str, bytes, None]], int, float, str, bytes, None], inplace: bool = False*) → pandas.core.frame.DataFrame

Brings one or more columns to the front (first n positions) of a DataFrame

Parameters

- **df** – Pandas DataFrame containing the data, other objects are implicitly cast to DataFrame
- **cols** – One or more column names to be brought to the front
- **inplace** – Whether to modify the DataFrame inplace [optional]

Returns Modified copy of the DataFrame

df_split

hhpy.ds.**df_split**(*df: pandas.core.frame.DataFrame, split_by: Union[List[str], str], return_type: str = 'dict', print_key: bool = False, sep: str = '_', key_sep: str = '=='*) → Union[list, dict]

Split a pandas DataFrame by column value and returns a list or dict

Parameters

- **df** – pandas DataFrame to be split
- **split_by** – Column(s) to split by, creates a sub-DataFrame for each level
- **return_type** – one of ['list', 'dict'], if list returns a list of sub-DataFrame, if dict returns a dictionary with each level as keys
- **print_key** – whether to include the column names in the key labels
- **sep** – separator to use in the key labels between columns
- **key_sep** – separator to use in the key labels between key and value

Returns see return_type

rank

hhpy.ds.**rank**(*df: pandas.core.frame.DataFrame, rankby: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]], groupby: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, rank_ascending: bool = True, sortby: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, sortby_ascending: Union[bool, List[bool]] = None*) → pandas.core.series.Series

creates a ranking (without duplicate ranks) based on columns of a DataFrame

Parameters

- **df** – Pandas DataFrame containing the data, other objects are implicitly cast to DataFrame
- **rankby** – the column(s) to rankby

- **groupby** – The columns used for grouping, passed to pandas.DataFrame.groupby [optional]
- **rank_ascending** – Whether to rank in ascending order [optional]
- **sortby** – After the rankby column(s) the sortby columns will be sorted to break ties [optional]
- **sortby_ascending** – The sorting preference for each sortby column [optional]

Returns pandas Series containing the rank (no duplicates)

mahalanobis

hhpy.ds.**mahalanobis**(*point: Union[pandas.core.frame.DataFrame, pandas.core.series.Series, numpy.ndarray], df: pandas.core.frame.DataFrame = None, params: List[str] = None, do_print: bool = True*) → Union[float, List[float]]

Calculates the Mahalanobis distance for a single point or a DataFrame of points

Parameters

- **point** – The point(s) to calculate the Mahalanobis distance for
- **df** – The reference DataFrame against which to calculate the Mahalanobis distance
- **params** – The columns to calculate the Mahalanobis distance for
- **do_print** – Whether to print intermediate steps to the console

Returns if a single point is passed: Mahalanobis distance as float, else a list of floats

df_count

hhpy.ds.**df_count**(*x: str, df: pandas.core.frame.DataFrame, hue: Optional[str] = None, sort_by_count: bool = True, top_nr: int = 5, x_base: Optional[float] = None, x_min: Optional[float] = None, x_max: Optional[float] = None, other_name: str = 'other', other_to_na: bool = False, na: Union[bool, str] = 'drop'*) → pandas.core.frame.DataFrame

Create a DataFrame of value counts. Supports hue levels and is therefore useful for plots, for an application see *countplot()*

Parameters

- **x** – Main variable, name of a column in the DataFrame or vector data
- **df** – Pandas DataFrame containing the data, other objects are implicitly cast to DataFrame
- **hue** – Name of the column to split by level [optional]
- **sort_by_count** – Whether to sort the DataFrame by value counts [optional]
- **top_nr** – Number of unique levels to keep when applying *top_n_coding()* [optional]
- **x_base** – if supplied: cast x to integer multiples of x_base, useful when you have float data that would result in many unique counts for close numbers [optional]
- **x_min** – limit the range of valid numeric x values to be greater than or equal to x_min [optional]
- **x_max** – limit the range of valid numeric x values to be less than or equal to x_max [optional]
- **other_name** – Name of the levels grouped inside other [optional]

- **other_to_na** – Whether to cast all other elements to NaN [optional]
- **na** – whether to keep (True, 'keep') na values and implicitly cast to string or drop (False, 'drop') them [optional]

Returns pandas DataFrame containing the counts by x (and by hue if it is supplied)

top_n

hhpy.ds.**top_n**(*s: Sequence[T_co], n: Union[int, str], w: Optional[Sequence[T_co]] = None, n_max: int = 20*) → list

Select n elements form a categorical pandas series with the highest counts. Ties are broken by sorting s ascending

Parameters

- **s** – pandas Series to select from
- **n** – how many elements to return, you can pass a percentage to return the top n %
- **w** – weights, if given the weights are summed instead of just counting entries in s [optional]
- **n_max** – how many elements to return at max if n is a percentage, set to None for no max [optional]

Returns List of top n elements

top_n_coding

hhpy.ds.**top_n_coding**(*s: Sequence[T_co], n: int, other_name: str = 'other', na_to_other: bool = False, other_to_na: bool = False, w: Optional[Sequence[T_co]] = None*) → pandas.core.series.Series

Returns a modified version of the pandas series where all elements not in top_n become recoded as 'other'

Parameters

- **s** – Pandas Series to adjust
- **n** – How many unique elements to keep
- **other_name** – Name of the other element [optional]
- **na_to_other** – Whether to cast missing elements to other [optional]
- **other_to_na** – Whether to cast all other elements to NaN [optional]
- **w** – Weights, if given the weights are summed instead of just counting entries in s [optional]

Returns Adjusted pandas Series

k_split

hhpy.ds.**k_split**(*df: pandas.core.frame.DataFrame, k: int = 5, groupby: Union[Sequence[T_co], str] = None, sortby: Union[Sequence[T_co], str] = None, random_state: int = None, do_print: bool = True, return_type: Union[str, int] = 1*) → Union[pandas.core.series.Series, tuple]

Splits a DataFrame into k (equal sized) parts that can be used for train test splitting or k_cross splitting

Parameters

- **df** – pandas DataFrame to be split
- **k** – how many (equal sized) parts to split the DataFrame into [optional]
- **groupby** – passed to pandas.DataFrame.groupby before splitting, ensures that each group will be represented equally in each split part [optional]
- **sortby** – if True the DataFrame is ordered by these column(s) and then sliced into parts from the top if False the DataFrame is sorted randomly before slicing [optional]
- **random_state** – random_state to be used in random sorting, ignore if sortby is True [optional]
- **do_print** – whether to print steps to console [optional]
- **return_type** – if one of ['Series', 's'] returns a pandas Series containing the k indices range(k) if a positive integer < k returns tuple of shape (df_train, df_test) where the return_type'th part is equal to df_test and the other parts are equal to df_train

Returns depending on return_type either a pandas Series or a tuple

remove_unused_categories

hhpy.ds.**remove_unused_categories**(*df: pandas.core.frame.DataFrame*, *inplace: bool = False*) → Optional[pandas.core.frame.DataFrame]

Remove unused categories from all categorical columns in the DataFrame

Parameters

- **df** – Pandas DataFrame containing the data, other objects are implicitly cast to DataFrame
- **inplace** – Whether to modify the DataFrame inplace [optional]

Returns pandas DataFrame with the unused categories removed

read_csv

hhpy.ds.**read_csv**(*path: str*, *nrows: int = None*, *encoding: str = None*, *errors: str = 'replace'*, *kws_open: Mapping[KT, VT_co] = None*, ***kwargs*)

wrapper for pandas.read_csv that reads the file into an IOString first. This enables one to use the error handling params of open which is very useful when opening a file with an uncertain encoding or illegal characters that would trigger an encoding error in pandas.read_csv

Parameters

- **path** – path to file
- **nrows** – how many rows to read, defaults to all [optional]
- **encoding** – encoding to pass to open [optional]
- **errors** – how to handle errors, see open [optional]
- **kws_open** – other keyword arguments passed to open [optional]
- **kwargs** – other keyword arguments passed to pandas.read_csv [optional]

Returns

get_columns

hhpy.ds.**get_columns**(*df: pandas.core.frame.DataFrame, dtype: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co], numpy.number] = None, to_list: bool = False*) → Union[list, pandas.core.indexes.base.Index]

A quick way to get the columns of a certain dtype. I added this because in pandas 1.0.0 pandas.DataFrame.select_dtypes('string') sometimes throws an error when the column does not contain correctly formated data.

Parameters

- **df** – Pandas DataFrame containing the data, other objects are implicitly cast to DataFrame
- **dtype** – dtype to filter for, mimics behaviour of pandas.DataFrame.select_dtypes
- **to_list** – Whether to return a list instead of a pandas.Index

Returns object containing the column names - if to_list: list, else pandas.Index

reformat_columns

hhpy.ds.**reformat_columns**(*df: pandas.core.frame.DataFrame, printf: Callable = None, **kwargs*) → pandas.core.frame.DataFrame

A quick way to clean the column names of a DataFrame

Parameters

- **df** – Pandas DataFrame containing the data, other objects are implicitly cast to DataFrame
- **printf** – Printing Function to use for steps [optional]
- **kwargs** – Additional keyword arguments passed to DFMapping [optional]

Returns DataFrame with reformated column names

1.2.3 Classes

DFMapping(df, dict, str] = None, **kwargs)	Mapping object bound to a pandas DataFrame that standardizes column names and values according to the chosen conventions.

DFMapping

class hhpy.ds.**DFMapping**(*df: Union[pandas.core.frame.DataFrame, dict, str] = None, **kwargs*)

Bases: *hhpy.main.BaseClass*

Mapping object bound to a pandas DataFrame that standardizes column names and values according to the chosen conventions. Also implements google translation. Can be used like an sklearn scalar object. The mapping can be saved and later used to restore the original shape of the DataFrame. Note that the index is exempt.

Parameters

- **name** – name of the object [Optional]
- **df** – a DataFrame to init on or path to a saved DFMapping object [Optional]
- **kwargs** – other arguments passed to the respective init function

Methods Summary

fit(*args, **kwargs)	Alias for *from_df()* to be inline with sklearn conventions
fit_transform(df, col_names, values, ...)	First applies *DFMapping.from_df()* (which has alias fit) and then *DFMapping.transform()*
from_df(df, col_names, values, columns, ...)	Initialize the DFMapping from a pandas DataFrame.
from_excel(path)	Init the DFMapping object from an excel file.
inverse_transform(*args, **kwargs)	wrapper for *DFMapping.transform()* with inverse=True
to_excel(path, if_exists)	Save the DFMapping object as an excel file.
transform(df, col_names, values, columns, ...)	Apply a mapping created using `create_df_mapping()`.

Methods Documentation

fit(**args*, ***kwargs*) → Optional[Tuple[dict, dict]]

Alias for *from_df()* to be inline with sklearn conventions

Parameters

- **args** – passed to from_df
- **kwargs** – passed to from_df

Returns see from_df

fit_transform(*df: pandas.core.frame.DataFrame*, *col_names: bool = True*, *values: bool = True*, *columns: Optional[List[str]] = None*, *kwargs_fit: Mapping[KT, VT_co] = None*, ***kwargs*) → Optional[pandas.core.frame.DataFrame]

First applies *DFMapping.from_df()* (which has alias fit) and then *DFMapping.transform()*

Parameters

- **df** – pandas DataFrame to fit against and then transform.
- **col_names** – Whether to transform the column names [optional]
- **values** – Whether to transform the column values [optional]
- **columns** – Columns to transform, defaults to all columns [optional]
- **kwargs** – passed to transform
- **kwargs_fit** – passed to fit

Returns see transform

from_df(*df: pandas.core.frame.DataFrame*, *col_names: bool = True*, *values: bool = True*, *columns: Optional[List[str]] = None*, *return_type: str = 'self'*, *printf: Callable = <function tprint>*, *duplicate_limit: int = 10*, *warn: bool = True*, ***kwargs*) → Optional[Tuple[dict, dict]]

Initialize the DFMapping from a pandas DataFrame.

Parameters

- **df** – Pandas DataFrame containing the data, other objects are implicitly cast to DataFrame
- **col_names** – Whether to transform the column names [optional]
- **values** – Whether to transform the column values [optional]
- **columns** – Columns to transform, defaults to all columns [optional]

- **return_type** – if 'self': writes to self, 'tuple' returns (col_mapping, value_mapping) [optional]
- **printf** – The function used for printing in-function messages. Set to None or False to suppress printing [optional]
- **duplicate_limit** – allowed number of reformated duplicates per column, each duplicate is suffixed with '_' but if you have too many you likely have a column of non allowed character strings and the mapping would take a very long time. The duplicate handling therefore stops and a warning is triggered since the transformation is no longer invertible. Consider excluding the column or using cat codes [optional]
- **warn** – Whether to show UserWarnings triggered by this function. Set to False to suppress, other warnings will still be triggered [optional]
- **kwargs** – Other keyword arguments passed to *reformat_string()* [optional]

Returns see return_type

from_excel(*path: str*) → None

Init the DFMapping object from an excel file. For example you could auto generate a DFMapping using googletrans and then adjust the translations you feel are inappropriate in the excel file. Then regenerate the object from the edited excel file.

Parameters **path** – Path to the excel file

Returns None

inverse_transform(**args*, ***kwargs*) → Optional[pandas.core.frame.DataFrame]

wrapper for *DFMapping.transform()* with inverse=True

Parameters

- **args** – passed to transform
- **kwargs** – passed to transform

Returns see transform

to_excel(*path: str*, *if_exists: str = 'error'*) → None

Save the DFMapping object as an excel file. Useful if you want to edit the results of the automatically generated object to fit your specific needs.

Parameters

- **path** – Path to save the excel file to
- **if_exists** – One of %(DFMapping__to_excel__if_exists)s, if 'error' raises exception, if 'replace' replaces existing files and if 'append' appends to file (while checking for duplicates)

Returns None

transform(*df: pandas.core.frame.DataFrame*, *col_names: bool = True*, *values: bool = True*, *columns: Optional[List[str]] = None*, *inverse: bool = False*, *inplace: bool = False*) → Optional[pandas.core.frame.DataFrame]

Apply a mapping created using `create_df_mapping()`. Intended to make a DataFrame standardized and human readable. The same mapping can also be applied with inverse=True to restore the original form of the transformed DataFrame.

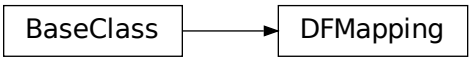
Parameters

- **df** – Pandas DataFrame containing the data, other objects are implicitly cast to DataFrame
- **col_names** – Whether to transform the column names [optional]

- **values** – Whether to transform the column values [optional]
- **columns** – Columns to transform, defaults to all columns [optional]
- **inverse** – Whether to apply the mapping in inverse order to restore the original form of the DataFrame [optional]
- **inplace** – Whether to modify the DataFrame inplace [optional]

Returns if inplace: None, else: Transformed DataFrame

1.2.4 Class Inheritance Diagram



1.3 hhpy.ipython Module

1.3.1 hhpy.ipython.py

Contains convenience wrappers for ipython

1.3.2 Functions

wide_notebook(width)	makes the jupyter notebook wider by appending html code to change the width,
hide_code()	hides the code and introduces a toggle button
display_full(*args[, rows, cols])	wrapper to display a pandas DataFrame with all rows and columns
pd_display(*args[, number_format, full])	wrapper to display a pandas DataFrame with a specified number format
display_df(df[, int_format, float_format, ...])	Wrapper to display a pandas DataFrame with separate options for int / float, also adds an option to exclude columns
highlight_max(df, color)	highlights the largest value in each column of a pandas DataFrame
highlight_min(df, color)	highlights the smallest value in each column of a pandas DataFrame
highlight_max_min(df, max_color, min_color)	highlights the largest and smallest value in each column of a pandas DataFrame

wide_notebook

hhpy.ipython.**wide_notebook**(*width: int = 90*)

makes the jupyter notebook wider by appending html code to change the width, based on https://stackoverflow.com/questions/21971449/how-do-i-increase-the-cell-width-of-the-jupyter-ipython-notebook-in-my-browser

Param width in percent, default 90 [optional]

Returns None

hide_code

hhpy.ipython.**hide_code**()

hides the code and introduces a toggle button based on https://stackoverflow.com/questions/27934885/how-to-hide-code-from-cells-in-ipython-notebook-visualized -with-nbviewer

Returns None

display_full

hhpy.ipython.**display_full**(**args*, *rows=None*, *cols=None*, ***kwargs*)

wrapper to display a pandas DataFrame with all rows and columns

Parameters

- **rows** – number of rows to display, defaults to all
- **cols** – number of columns to display, defaults to all
- **args** – passed to display
- **kwargs** – passed to display

Returns None

pd_display

hhpy.ipython.**pd_display**(**args*, *number_format='{:, .2f}'*, *full=True*, ***kwargs*)

wrapper to display a pandas DataFrame with a specified number format

Parameters

- **args** – passed to display
- **number_format** – the number format to apply
- **full** – whether to use *display_full()* (True) or standard display (False)
- **kwargs** – passed to display

Returns None

display_df

hhpy.ipython.**display_df**(*df*, *int_format=' '*, *float_format='.2f'*, *exclude=None*, *full=True*, ***kwargs*)

Wrapper to display a pandas DataFrame with separate options for int / float, also adds an option to exclude columns

Parameters

- **df** – pandas DataFrame to display
- **int_format** – format for integer columns
- **float_format** – format for float columns
- **exclude** – columns to exclude
- **full** – whether to show all rows and columns or keep default behaviour
- **kwargs** – passed to display

Returns None

highlight_max

hhpy.ipython.**highlight_max**(*df: pandas.core.frame.DataFrame*, *color: str = 'xkcd:cyan'*) → pandas.core.frame.DataFrame

highlights the largest value in each column of a pandas DataFrame

Parameters

- **df** – pandas DataFrame
- **color** – color used for highlighting

Returns the pandas DataFrame with the style applied to it

highlight_min

hhpy.ipython.**highlight_min**(*df: pandas.core.frame.DataFrame*, *color: str = 'xkcd:light red'*) → pandas.core.frame.DataFrame

highlights the smallest value in each column of a pandas DataFrame

Parameters

- **df** – pandas DataFrame
- **color** – color used for highlighting

Returns the pandas DataFrame with the style applied to it

highlight_max_min

hhpy.ipython.**highlight_max_min**(*df: pandas.core.frame.DataFrame*, *max_color: str = 'xkcd:cyan'*, *min_color: str = 'xkcd:light red'*)

highlights the largest and smallest value in each column of a pandas DataFrame

Parameters

- **df** – pandas DataFrame
- **max_color** – color used for highlighting largest value
- **min_color** – color used for highlighting smallest value

Returns the pandas DataFrame with the style applied to it

1.4 hhpy.modelling Module

1.4.1 hhpy.modelling.py

Contains a model class that is based on pandas DataFrames and wraps around sklearn and other frameworks to provide convenient train test functions.

1.4.2 Functions

assert_array(a, return_name, name_default)	Take any python object and turn it into a 2d numpy array (if possible).
dict_to_model(dic, VT_co])	restore a Model object from a dictionary
assert_model(model)	takes any Model, model object or dictionary and converts to Model
get_coefs(model, y, int, float, str, bytes, ...)	get coefficients of a linear regression in a sorted data frame
get_feature_importance(model, predictors, ...)	get feature importance of a decision tree like model in a sorted data frame
to_keras_3d(x, numpy.ndarray], window, y, ...)	reformat a DataFrame / 2D array to become a keras compatible 3D array.

assert_array

hhpy.modelling.**assert_array**(*a: Any*, *return_name: bool = False*, *name_default: str = 'name'*) → Union[Tuple[numpy.ndarray, str], numpy.ndarray]

Take any python object and turn it into a 2d numpy array (if possible). Useful for training neural networks.

Parameters

- **a** – any python object
- **return_name** – Whether the name should be returned
- **name_default** – The name to fall back to if the object has no name attribute.

Returns numpy array, if return_name: Tuple [numpy.array, name]

dict_to_model

hhpy.modelling.**dict_to_model**(*dic: Mapping[KT, VT_co]*) → hhpy.modelling.Model

restore a Model object from a dictionary

Parameters **dic** – dictionary containing the model definition

Returns Model

assert_model

hhpy.modelling.**assert_model**(*model: Any*) → hhpy.modelling.Model

takes any Model, model object or dictionary and converts to Model

Parameters **model** – Mapping or object containing a model

Returns Model

get_coefs

hhpy.modelling.**get_coefs**(*model: Any, y: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]]*)

get coefficients of a linear regression in a sorted data frame

Parameters

- **model** – model object
- **y** – name of the coefficients

Returns pandas DataFrame containing the coefficient names and values

get_feature_importance

hhpy.modelling.**get_feature_importance**(*model: object, predictors: Union[Sequence[T_co], str], features_to_sum: Mapping[KT, VT_co] = None*) → pandas.core.frame.DataFrame

get feature importance of a decision tree like model in a sorted data frame

Parameters

- **model** – model object
- **predictors** – names of the predictors properly sorted
- **features_to_sum** – if you want to sum features please provide name mappings

Returns pandas DataFrame containing the feature importances

to_keras_3d

hhpy.modelling.**to_keras_3d**(*x: Union[pandas.core.frame.DataFrame, numpy.ndarray], window: int, y: Union[pandas.core.frame.DataFrame, numpy.ndarray] = None, groupby: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, groupby_to_dummy: bool = False, dropna: bool = True, reshape: bool = True*) → Union[numpy.ndarray, Tuple[numpy.ndarray, numpy.ndarray]]

reformat a DataFrame / 2D array to become a keras compatible 3D array. If dropna is True the first window observations get dropped since they will contain NaN values in the required shifted elements.

Parameters

- **x** – numpy array or DataFrame
- **window** – series-window, how many iterations to convolve
- **y** – accompanying target / label DataFrame or numpy 2d array. If specified a modified version of y will be returned to match x's shape where the first window elements have been dropped. [optional]
- **groupby** – column to group by (shift observations in each group) [optional]
- **groupby_to_dummy** – Whether to include the groupby value as pandas Dummy [optional]
- **dropna** – Whether to drop na rows [optional]
- **reshape** – Whether to reshape to keras format observations - timestamps - features [optional]

Returns if y is None: x as 3d array, else: Tuple[x, y]

1.4.3 Classes

Model(model, name, X_ref, int, float, str, ...)	A unified modeling class that is extended from sklearn, accepts any model that implements .fit and .predict
Models(*args, df, X_ref, int, float, str, ...)	Collection of Models that allow for fitting and predicting with multiple Models at once, comparing accuracy and creating Ensembles

Model

class hhpy.modelling.**Model**(*model: Any = None, name: str = 'pred', X_ref: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, y_ref: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, groupby: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None*)

Bases: *hhpy.main.BaseClass*

A unified modeling class that is extended from sklearn, accepts any model that implements .fit and .predict

Parameters

- **model** – Any model object that implements .fit and .predict
- **name** – Name of the model, used for naming columns [optional]
- **X_ref** – List of features (predictors) used for training the model
- **y_ref** – List of labels (targets) to be predicted
- **groupby** – The columns used for grouping, passed to pandas.DataFrame.groupby [optional]

Methods Summary

fit(X, numpy.ndarray, Sequence[T_co], int, ...)	generalized fit method extending on model.fit
predict(X, numpy.ndarray, Sequence[T_co], ...)	Generalized predict method based on model.predict

Methods Documentation

fit(*X: Union[pandas.core.frame.DataFrame, numpy.ndarray, Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, y: Union[pandas.core.frame.DataFrame, numpy.ndarray, Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, df: pandas.core.frame.DataFrame = None, dropna: bool = True, X_test: Union[pandas.core.frame.DataFrame, numpy.ndarray, Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, y_test: Union[pandas.core.frame.DataFrame, numpy.ndarray, Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, df_test: pandas.core.frame.DataFrame = None, groupby: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, k: int = 0*) → None
generalized fit method extending on model.fit

Parameters

- **X** – The feature (predictor) data used for training as DataFrame, np.array or column names

- **y** – The label (target) data used for training as DataFrame, np.array or column names
- **df** – Pandas DataFrame containing the training data, optional if array like data is passed for X/y
- **dropna** – Whether to drop rows containing NA in the training data [optional]
- **X_test** – The feature (predictor) data used for testing as DataFrame, np.array or column names
- **y_test** – The label (target) data used for testing as DataFrame, np.array or column names
- **df_test** – Pandas DataFrame containing the testing data, optional if array like data is passed for X/y test
- **groupby** – The columns used for grouping, passed to pandas.DataFrame.groupby [optional]
- **k** – index of the model to fit

Returns None

predict(*X: Union[pandas.core.frame.DataFrame, numpy.ndarray, Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, y: Union[pandas.core.frame.DataFrame, numpy.ndarray, Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, df: pandas.core.frame.DataFrame = None, return_type: str = 'y', k_index: pandas.core.series.Series = None, groupby: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, handle_na: bool = True, multi: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None*) → Union[pandas.core.series.Series, pandas.core.frame.DataFrame]

Generalized predict method based on model.predict

Parameters

- **X** – The feature (predictor) data used for training as DataFrame, np.array or column names
- **y** – The label (target) data used for training as DataFrame, np.array or column names
- **df** – Pandas DataFrame containing the training and testing data. Can be saved to the Model object or supplied on an as needed basis.
- **return_type** – one of ['y', 'df', 'DataFrame'], if 'y' returns a pandas Series / DataFrame with only the predictions, if one of 'df','DataFrame' returns the full DataFrame with predictions added
- **k_index** – If specified and model is k_cross split: return only the predictions for each test subset
- **groupby** – The columns used for grouping, passed to pandas.DataFrame.groupby [optional]
- **handle_na** – Whether to handle NaN values (prediction will be NaN) [optional]
- **multi** – Postfixes to use for multi output models [optional]

Returns see return_type

Models

class hhpy.modelling.**Models**(**args, df: pandas.core.frame.DataFrame = None, X_ref: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, y_ref: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, groupby: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, scaler_X: Any = None, scaler_y: Any = None, printf: Callable = <function tprint>*)

Bases: *hhpy.main.BaseClass*

Collection of Models that allow for fitting and predicting with multiple Models at once, comparing accuracy and creating Ensembles

Parameters

- **args** – multiple Model objects that will form a Models Collection
- **name** – name of the collection
- **df** – Pandas DataFrame containing the training and testing data. Can be saved to the Model object or supplied on an as needed basis.
- **X_ref** – List of features (predictors) used for training the model
- **y_ref** – List of labels (targets) to be predicted
- **scaler_X** – Scalar object that implements .transform and .inverse_transform, applied to the features (predictors)before training and inversely after predicting [optional]
- **scaler_y** – Scalar object that implements .transform and .inverse_transform, applied to the labels (targets)before training and inversely after predicting [optional]
- **printf** – print function to use for logging [optional]

Methods Summary

fit(fit_type, k_test, groupby, int, float, ...)	fit all Model objects in collection
k_split(**kwargs)	apply hhpy.ds.k_split to self to create train-test or k-cross ready data
model_by_name(name, str])	extract a list of Models from the collection by their names
predict(X, numpy.ndarray, Sequence[T_co], ...)	predict with all models in collection
score(return_type, pivot, groupby, int, ...)	calculate score of the Model predictions
scoreplot([x, y, hue, hue_order, row, ...])	plot the score(s) using sns.barplot
train(df, k, groupby, str] = None, sortby, ...)	wrapper method that combined k_split, train, predict and score

Methods Documentation

fit(*fit_type: str = 'train_test', k_test: Optional[int] = 0, groupby: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, do_print: bool = True, **kwargs*)
fit all Model objects in collection

Parameters

- **fit_type** – one of ['train_test', 'k_cross', 'final']
- **k_test** – which k_index to use as test data

- **groupby** – The columns used for grouping, passed to pandas.DataFrame.groupby [optional]
- **do_print** – Whether to print the steps to console [optional]
- **kwargs** – Other keyword arguments passed to *fit()*

Returns None

k_split(***kwargs*)
apply hhpy.ds.k_split to self to create train-test or k-cross ready data

Parameters **kwargs** – keyword arguments passed to *k_split()*

Returns None

model_by_name(*name: Union[list, str]*) → Union[hhpy.modelling.Model, list]
extract a list of Models from the collection by their names

Parameters **name** – name of the Model

Returns list of Models

predict(*X: Union[pandas.core.frame.DataFrame, numpy.ndarray, Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, y: Union[pandas.core.frame.DataFrame, numpy.ndarray, Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, df: pandas.core.frame.DataFrame = None, return_type: str = 'self', ensemble: bool = False, k_predict_type: str = 'test', groupby: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, multi: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, do_print: bool = True*) → Union[pandas.core.series.Series, pandas.core.frame.DataFrame, None]
predict with all models in collection

Parameters

- **X** – The feature (predictor) data used for predicting as DataFrame, np.array or column names
- **y** – The label (target) data used for predicting as DataFrame, np.array or column names. Specifying y is only necessary for convolutional or time-series type models [optional]
- **df** – Pandas DataFrame containing the predict data, optional if array like data is passed for X_predict
- **return_type** – one of ['y', 'df', 'DataFrame', 'self']
- **ensemble** – if True also predict with Ensemble like combinations of models. If True or mean calculatemean of individual predictions. If median calculate median of individual predictions. [optional]
- **k_predict_type** – 'test' or 'all'
- **groupby** – The columns used for grouping, passed to pandas.DataFrame.groupby [optional]
- **multi** – postfixes to use for multi output [optional]
- **do_print** – Whether to print the steps to console [optional]

Returns if return_type is self: None, else see Model.predict

score(*return_type: str = 'self', pivot: bool = False, groupby: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, do_print: bool = True, display_score: bool = True, display_format: str = ',.3f', **kwargs*) → Optional[pandas.core.frame.DataFrame]
calculate score of the Model predictions

Parameters

- **return_type** – one of ['self', 'df', 'DataFrame']
- **pivot** – whether to pivot the DataFrame for easier readability [optional]
- **do_print** – Whether to print the steps to console [optional]
- **display_score** – Whether to display the score DataFrame [optional]
- **display_format** – Format to use when displaying the score DataFrame [optional]
- **groupby** – The columns used for grouping, passed to pandas.DataFrame.groupby [optional]
- **kwargs** – other keyword arguments passed to *df_score()*

Returns if return_type is 'self': None, else: pandas DataFrame containing the scores

scoreplot(*x='y_ref'*, *y='value'*, *hue='model'*, *hue_order=None*, *row='score'*, *row_order=None*, *palette=None*, *width=16*, *height=4.5*, *scale=None*, *query=None*, *return_fig_ax=False*, ***kwargs*) → Optional[tuple]

plot the score(s) using sns.barplot

Parameters

- **x** – Name of the x variable in data or vector data
- **y** – Name of the y variable in data or vector data
- **hue** – Further split the plot by the levels of this variable [optional]
- **hue_order** – Either a string describing how the (hue) levels or to be ordered or an explicit list of levels to be used for plotting. Accepted strings are:

- `sorted`: following python standard sorting conventions (alphabetical for string, ascending for value)
- `inv`: following python standard sorting conventions but in inverse order
- `count`: sorted by value counts
- `mean`, `mean_ascending`, `mean_descending`: sorted by mean value, defaults to descending
- `median`, `mean_ascending`, `median_descending`: sorted by median value, defaults to descending

Parameters

- **row** – the variable to wrap around the rows [optional]
- **row_order** – Either a string describing how the (hue) levels or to be ordered or an explicit list of levels to be used for plotting. Accepted strings are:

- `sorted`: following python standard sorting conventions (alphabetical for string, ascending for value)
- `inv`: following python standard sorting conventions but in inverse order
- `count`: sorted by value counts
- `mean`, `mean_ascending`, `mean_descending`: sorted by mean value, defaults to descending
- `median`, `mean_ascending`, `median_descending`: sorted by median value, defaults to descending

Parameters

- **palette** – Collection of colors to be used for plotting. Can be a dictionary for with names for each level or a list of colors or an individual color name. Must be valid colors known to pyplot [optional]
- **width** – Width of each individual subplot [optional]
- **height** – Height of each individual subplot [optional]
- **scale** – scale the values [optional
- **query** – query to be passed to pd.DataFrame.query before plotting [optional]
- **return_fig_ax** – Whether to return the figure and axes objects as tuple to be captured as fig,ax = . . . , If False pyplot.show() is called and the plot returns None [optional]
- **kwargs** – other keyword arguments passed to sns.barplot

Returns see return_fig_ax

train(*df: pandas.core.frame.DataFrame = None, k: int = 5, groupby: Union[Sequence[T_co], str] = None, sortby: Union[Sequence[T_co], str] = None, random_state: int = None, fit_type: str = 'train_test', k_test: Optional[int] = 0, ensemble: bool = False, scores: Union[Sequence[T_co], str, Callable] = None, multi: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, scale: float = None, do_predict: bool = True, do_score: bool = True, do_split: bool = True, do_fit: bool = True, do_print: bool = True, display_score: bool = True*) → None

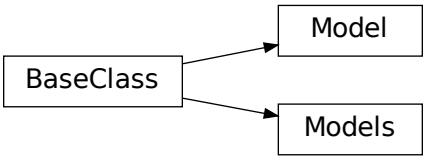
wrapper method that combined k_split, train, predict and score

Parameters

- **df** – Pandas DataFrame containing the training and testing data. Can be saved to the Model object or supplied on an as needed basis.
- **k** – see hhpy.ds.k_split see hhpy.ds.k_split
- **groupby** – The columns used for grouping, passed to pandas.DataFrame.groupby [optional]
- **sortby** – see hhpy.ds.k_split
- **random_state** – see hhpy.ds.k_split
- **fit_type** – see .fit
- **k_test** – see .fit
- **ensemble** – if True also predict with Ensemble like combinations of models. If True or mean calculatemean of individual predictions. If median calculate median of individual predictions. [optional]
- **scores** – see .score [optional]
- **multi** – postfixes to use for multi output [optional]
- **scale** – see .score
- **do_print** – Whether to print the steps to console [optional]
- **display_score** – Whether to display the score DataFrame [optional]
- **do_split** – whether to apply k_split [optional]
- **do_fit** – whether to fit the Models [optional]
- **do_predict** – whether to add predictions to DataFrame [optional]
- **do_score** – whether to create self.df_score [optional]

Returns None

1.4.4 Class Inheritance Diagram



1.5 hhpy.plotting Module

1.5.1 hhpy.plotting.py

Contains plotting functions using matplotlib.pyplot

1.5.2 Functions

heatmap(x, y, z, data, ax, cmap, agg_func, ...)	Wrapper for seaborn heatmap in x-y-z format
corrplot(data, annotations, number_format[, ax])	function to create a correlation plot using a seaborn heatmap based on: https://www.linkedin.com/pulse/generating-correlation-heatmaps-seaborn-python-andrew-holt
corrplot_bar(data, target, columns, ...)	Correlation plot as barchart based on *get_df_corr()*
pairwise_corrplot(data, corr_cutoff, ...)	print a pairwise_corrplot to for all variables in the df, by default only plots those with a correlation coefficient of >= corr_cutoff
distplot(x, str], data, hue, hue_order, ...)	Similar to seaborn.distplot but supports hues and some other things.
hist_2d(x, y, data, bins, std_cutoff, ...)	generic 2d histogram created by splitting the 2d area into equal sized cells, counting data points in them and drawn using pyplot.pcolormesh
paired_plot(data, cols, color, cmap, alpha, ...)	create a facet grid to analyze various aspects of correlation between two variables using seaborn.PairGrid
q_plim(s, q_min, q_max, offset_perc, ... [, ...])	returns quick x limits for plotting (cut off data not in q_min to q_max quantile)
levelplot(data, level, cols, str], hue, ...)	Plots a plot for each specified column for each level of a certain column plus a summary plot
get_legends(ax)	returns all legends on a given axis, useful if you have a secaxis
facet_wrap(func, data, facet, str], *args, ...)	modeled after r's facet_wrap function.

Continued on next page

Table 12 – continued from previous page

get_subax(ax, numpy.ndarray], row, col, ...)	shorthand to get around the fact that ax can be a 1D array or a 2D array (for subplots that can be 1x1,1xn,nx1)
ax_as_list(ax, numpy.ndarray])	takes any Axes and turns them into a list
ax_as_array(ax, numpy.ndarray])	takes any Axes and turns them into a numpy 2D array
rmsdplot(x, data, groups, str] = None, hue, ...)	creates a seaborn.barplot showing the rmsd calculating *df_rmsd()*
insert_linebreak(s, pos, frac, max_breaks)	used to insert linebreaks in strings, useful for formatting axes labels
ax_tick_linebreaks(ax, x, y, **kwargs)	uses insert_linebreaks to insert linebreaks into the axes ticklabels
annotate_barplot(ax, x, y, ci, ci_newline, ...)	automatically annotates a barplot with bar values and error bars (if present).
animplot(data, x, y, t, lines, ...)	wrapper for FuncAnimation to be used with pandas DataFrames.
legend_outside(ax, width, loc, legend_space, ...)	draws a legend outside of the subplot
set_ax_sym(ax, x, y)	automatically sets the select axes to be symmetrical
custom_legend(colors, str], labels, str][, ...])	uses patches to create a custom legend with the specified colors
stemplot(x, y[, data, ax, color, baseline, ...])	modeled after pyplot.stemplot but more customizeable
get_twin(ax)	get the twin axis from an Axes object
get_axlim(ax, xy)	Wrapper function to get x limits, y limits or both with one function call
set_axlim(ax, lim, Mapping[KT, VT_co]], xy)	Wrapper function to set both x and y limits with one call
share_xy(ax, x, y, mode, adj_twin_ax)	set the subplots on the Axes to share x and/or y limits WITHOUT sharing x and y legends.
share_legend(ax, keep_i)	removes all legends except for i from an Axes object
barplot_err(x, y, xerr, yerr, data, **kwargs)	extension on seaborn barplot that allows for plotting errorbars with preprocessed data.The idea is based on this StackOverflow question.
countplot(x, str] = None, data, hue, ax, ...)	Based on seaborn barplot but with a few more options, uses *df_count()*
quantile_plot(x, str], data, qs, ...)	plots the specified quantiles of a Series using seaborn.barplot
plotly_aggplot(data, x, float, str, bytes, ...)	create a (grouped) plotly aggplot that let's you select the groupby categories

heatmap

hhpy.plotting.**heatmap**(*x: str*, *y: str*, *z: str*, *data: pandas.core.frame.DataFrame*, *ax: matplotlib.axes._axes.Axes = None*, *cmap: object = None*, *agg_func: str = 'mean'*, *invert_y: bool = True*, ***kwargs*) → matplotlib.axes._axes.Axes

Wrapper for seaborn heatmap in x-y-z format

Parameters

- **x** – Variable name for x axis value
- **y** – Variable name for y axis value
- **z** – Variable name for z value, used to color the heatmap
- **data** – Pandas DataFrame containing named data, optional if vector data is used
- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]
- **cmap** – Color map to use [optional]

- **agg_func** – If more than one z value per x,y pair exists agg_func is used to aggregate the data. Must be a function name understood by pandas.DataFrame.agg
- **invert_y** – Whether to call ax.invert_yaxis (orders the heatmap as expected)
- **kwargs** – Other keyword arguments passed to seaborn heatmap

Returns The matplotlib.pyplot.Axes object with the plot on it

corrplot

hhpy.plotting.**corrplot**(*data: pandas.core.frame.DataFrame*, *annotations: bool = True*, *number_format: str = '.2f'*, *ax=None*)

function to create a correlation plot using a seaborn heatmap based on: https://www.linkedin.com/pulse/generating-correlation-heatmaps-seaborn-python-andrew-holt

Parameters

- **number_format** – The format string used for annotations [optional]
- **data** – Pandas DataFrame containing named data
- **annotations** – Whether to display annotations [optional]
- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]

Returns The matplotlib.pyplot.Axes object with the plot on it

corrplot_bar

hhpy.plotting.**corrplot_bar**(*data: pandas.core.frame.DataFrame*, *target: str = None*, *columns: List[str] = None*, *corr_cutoff: float = 0*, *corr_as_alpha: bool = False*, *xlim: tuple = (-1, 1)*, *ax: matplotlib.axes._axes.Axes = None*)

Correlation plot as barchart based on *get_df_corr()*

Parameters

- **data** – Pandas DataFrame containing named data, optional if vector data is used
- **target** – Target variable name, if specified only correlations with the target are shown [optional]
- **columns** – Columns for which to calculate the correlations, defaults to all numeric columns [optional]
- **corr_cutoff** – Filter all correlation whose absolute value is below the cutoff [optional]
- **corr_as_alpha** – Whether to set alpha value of bars to scale with correlation [optional]
- **xlim** – xlim scale for plot, defaults to (-1, 1) to show the absolute scale of the correlations. set to None if you want the plot x limits to scale to the highest correlation values [optional]
- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]

Returns The matplotlib.pyplot.Axes object with the plot on it

pairwise_corrplot

hhpy.plotting.**pairwise_corrplot**(*data: pandas.core.frame.DataFrame, corr_cutoff: float = 0, col_wrap: int = 4, hue: str = None, hue_order: Union[list, str] = None, width: float = 7, height: float = 7, trendline: bool = True, alpha: float = 0.75, ax: matplotlib.axes._axes.Axes = None, target: str = None, palette: Union[Mapping[KT, VT_co], Sequence[T_co], str] = ['xkcd:blue', 'xkcd:red', 'xkcd:green', 'xkcd:cyan', 'xkcd:magenta', 'xkcd:golden yellow', 'xkcd:dark cyan', 'xkcd:red orange', 'xkcd:dark yellow', 'xkcd:easter green', 'xkcd:baby blue', 'xkcd:light brown', 'xkcd:strong pink', 'xkcd:light navy blue', 'xkcd:deep blue', 'xkcd:deep red', 'xkcd:ultramarine blue', 'xkcd:sea green', 'xkcd:plum', 'xkcd:old pink', 'xkcd:lawn green', 'xkcd:amber', 'xkcd:green blue', 'xkcd:yellow green', 'xkcd:dark mustard', 'xkcd:bright lime', 'xkcd:aquamarine', 'xkcd:very light blue', 'xkcd:light grey blue', 'xkcd:dark sage', 'xkcd:dark peach', 'xkcd:shocking pink'], max_n: int = 10000, random_state: int = None, sample_warn: bool = True, return_fig_ax: bool = True, **kwargs*) → Optional[tuple]

print a pairwise_corrplot to for all variables in the df, by default only plots those with a correlation coefficient of >= corr_cutoff

Parameters

- **data** – Pandas DataFrame containing named data
- **corr_cutoff** – Filter all correlation whose absolute value is below the cutoff [optional]
- **col_wrap** – After how many columns to create a new line of subplots [optional]
- **hue** – Further split the plot by the levels of this variable [optional]
- **hue_order** – Either a string describing how the (hue) levels or to be ordered or an explicit list of levels to be used for plotting. Accepted strings are:
  - `sorted`: following python standard sorting conventions (alphabetical for string, ascending for value)
  - `inv`: following python standard sorting conventions but in inverse order
  - `count`: sorted by value counts
  - `mean`, `mean_ascending`, `mean_descending`: sorted by mean value, defaults to descending
  - `median`, `mean_ascending`, `median_descending`: sorted by median value, defaults to descending
- **width** – Width of each individual subplot [optional]
- **height** – Height of each individual subplot [optional]
- **trendline** – Whether to add a trendline [optional]
- **alpha** – Alpha transparency level [optional]
- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]
- **target** – Target variable name, if specified only correlations with the target are shown [optional]

- **palette** – Collection of colors to be used for plotting. Can be a dictionary for with names for each level or a list of colors or an individual color name. Must be valid colors known to pyplot [optional]
- **max_n** – Maximum number of samples to be used for plotting, if this number is exceeded max_n samples are drawn ‘ ‘at random from the data which triggers a warning unless sample_warn is set to False. ‘ ‘Set to False or None to use all samples for plotting. [optional]
- **random_state** – Random state (seed) used for drawing the random samples [optional]
- **sample_warn** – Whether to trigger a warning if the data has more samples than max_n [optional]
- **return_fig_ax** – Whether to return the figure and axes objects as tuple to be captured as fig,ax = . . . , If False pyplot.show() is called and the plot returns None [optional]
- **kwargs** – other keyword arguments passed to pyplot.subplots

Returns if return_fig_ax: figure and axes objects as tuple, else None

distplot

hhpy.plotting.**distplot**(*x: Union[Sequence[T_co], str], data: pandas.core.frame.DataFrame = None, hue: str = None, hue_order: Union[Sequence[T_co], str] = 'sorted', palette: Union[Mapping[KT, VT_co], Sequence[T_co], str] = None, linecolor: str = 'black', edgecolor: str = 'black', alpha: float = None, bins: Union[Sequence[T_co], int] = 40, perc: bool = None, top_nr: int = None, other_name: str = 'other', title: bool = True, title_prefix: str = '', std_cutoff: float = None, hist: bool = None, distfit: Union[str, bool, None] = 'kde', fill: bool = True, legend: bool = True, legend_loc: str = None, legend_space: float = 0.1, legend_ncol: int = 1, agg_func: str = 'mean', number_format: str = '.2f', kde_steps: int = 1000, max_n: int = 100000, random_state: int = None, sample_warn: bool = True, xlim: Sequence[T_co] = None, linestyle: str = None, label_style: str = 'mu_sigma', x_offset_perc: float = 0.025, ax: matplotlib.axes._axes.Axes = None, **kwargs*) → matplotlib.axes._axes.Axes

Similar to seaborn.distplot but supports hues and some other things. Plots a combination of a histogram and a kernel density estimation.

Parameters

- **x** – the name of the variable(s) in data or vector data, if data is provided and x is a list of columns the DataFrame is automatically melted and the newly generated column used as hue. i.e. you plot the distributions of multiple columns on the same axis
- **data** – Pandas DataFrame containing named data, optional if vector data is used
- **hue** – Further split the plot by the levels of this variable [optional]
- **hue_order** – Either a string describing how the (hue) levels or to be ordered or an explicit list of levels to be used for plotting. Accepted strings are:
  - `sorted`: following python standard sorting conventions (alphabetical for string, ascending for value)
  - `inv`: following python standard sorting conventions but in inverse order
  - `count`: sorted by value counts
  - `mean`, `mean_ascending`, `mean_descending`: sorted by mean value, defaults to descending

  - median, mean_ascending, median_descending: sorted by median value, defaults to descending

- **palette** – Collection of colors to be used for plotting. Can be a dictionary for with names for each level or a list of colors or an individual color name. Must be valid colors known to pyplot [optional]
- **linecolor** – Color of the kde fit line, overwritten with palette by hue level if hue is specified [optional]
- **edgecolor** – Color of the histogram edges [optional]
- **alpha** – Alpha transparency level [optional]
- **bins** – Nr of bins of the histogram [optional]
- **perc** – Whether to display the y-axes as percentage, if false count is displayed. Defaults if hue: True, else False [optional]
- **top_nr** – limit hue to top_nr levels using hhpy.ds.top_n, the rest will be cast to other [optional]
- **other_name** – name of the other group created by hhpy.ds.top_n [optional]
- **title** – whether to set the plot title equal to x's name [optional]
- **title_prefix** – prefix to be used in plot title [optional]
- **std_cutoff** – automatically cutoff data outside of the std_cutoff standard deviations range, by default this is off but a recommended value for a good visual experience without outliers is 3 [optional]
- **hist** – whether to show the histogram, default False if hue else True [optional]
- **distfit** – one of ['kde', 'gauss', 'False', 'None']. If 'kde' fits a kernel density distribution to the data. If gauss fits a gaussian distribution with the observed mean and std to the data. [optional]
- **fill** – whether to fill the area under the distfit curve, ignored if hist is True [optional]
- **legend** – Whether to show a legend [optional]
- **legend_loc** – Location of the legend, one of [bottom, right] or accepted value of pyplot.legendIf in [bottom, right] legend_outside is used, else pyplot.legend [optional]
- **legend_space** – Only valid if legend_loc is bottom. The space between the main plot and the legend [optional]
- **legend_ncol** – Number of columns to use in legend [optional]
- **agg_func** – one of ['mean', 'median']. The agg function used to find the center of the distribution [optional]
- **number_format** – The format string used for annotations [optional]
- **kde_steps** – Nr of steps the range is split into for kde fitting [optional]
- **max_n** – Maximum number of samples to be used for plotting, if this number is exceeded max_n samples are drawn ' 'at random from the data which triggers a warning unless sample_warn is set to False. ' 'Set to False or None to use all samples for plotting. [optional]
- **random_state** – Random state (seed) used for drawing the random samples [optional]
- **sample_warn** – Whether to trigger a warning if the data has more samples than max_n [optional]

- **xlim** – X limits for the axis as tuple, passed to ax.set_xlim() [optional]
- **linestyle** – Linestyle used, must a valid linestyle recognized by pyplot.plot [optional]
- **label_style** – one of ['mu_sigma', 'plain']. If mu_sigma then the mean (or median) and std value are displayed inside the label [optional]
- **x_offset_perc** – the amount whitespace to display next to x_min and x_max in percent of x_range [optional]
- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]
- **kwargs** – additional keyword arguments passed to pyplot.plot

Returns The matplotlib.pyplot.Axes object with the plot on it

hist_2d

hhpy.plotting.**hist_2d**(*x: str*, *y: str*, *data: pandas.core.frame.DataFrame*, *bins: int = 100*, *std_cutoff: int = 3*, *cutoff_perc: float = 0.01*, *cutoff_abs: float = 0*, *cmap: str = 'rainbow'*, *ax: matplotlib.axes._axes.Axes = None*, *color_sigma: str = 'xkcd:red'*, *draw_sigma: bool = True*, ***kwargs*) → matplotlib.axes._axes.Axes

generic 2d histogram created by splitting the 2d area into equal sized cells, counting data points in them and drawn using pyplot.pcolormesh

Parameters

- **x** – Name of the x variable in data or vector data
- **y** – Name of the y variable in data or vector data
- **data** – Pandas DataFrame containing named data, optional if vector data is used
- **bins** – Nr of bins of the histogram [optional]
- **std_cutoff** – Remove data outside of std_cutoff standard deviations, for a good visual experience try 3 [optional]
- **cutoff_perc** – if less than this percentage of data points is in the cell then the data is ignored [optional]
- **cutoff_abs** – if less than this amount of data points is in the cell then the data is ignored [optional]
- **cmap** – Color map to use [optional]
- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]
- **color_sigma** – color to highlight the sigma range in, must be a valid pyplot.plot color [optional]
- **draw_sigma** – whether to highlight the sigma range [optional]
- **kwargs** – other keyword arguments passed to pyplot.pcolormesh [optional]

Returns The matplotlib.pyplot.Axes object with the plot on it

paired_plot

hhpy.plotting.**paired_plot**(*data: pandas.core.frame.DataFrame*, *cols: Sequence[T_co]*, *color: str = None*, *cmap: str = None*, *alpha: float = 1*, ***kwargs*) → seaborn.axisgrid.FacetGrid

create a facet grid to analyze various aspects of correlation between two variables using seaborn.PairGrid

Parameters

- **data** – Pandas DataFrame containing named data, optional if vector data is used
- **cols** – list of exactly two variables to be compared
- **color** – Color used for plotting, must be known to matplotlib.pyplot [optional]
- **cmap** – Color map to use [optional]
- **alpha** – Alpha transparency level [optional]
- **kwargs** – other arguments passed to seaborn.PairGrid

Returns seaborn FacetGrid object with the plots on it

q_plim

hhpy.plotting.**q_plim**(*s: pandas.core.series.Series*, *q_min: float = 0.1*, *q_max: float = 0.9*, *offset_perc: float = 0.1*, *limit_min_max: bool = False*, *offset=True*) → tuple

returns quick x limits for plotting (cut off data not in q_min to q_max quantile)

Parameters

- **s** – pandas Series to truncate
- **q_min** – lower bound quantile [optional]
- **q_max** – upper bound quantile [optional]
- **offset_perc** – percentage of offset to the left and right of the quantile boundaries
- **limit_min_max** – whether to truncate the plot limits at the data limits
- **offset** – whether to apply the offset

Returns a tuple containing the x limits

levelplot

hhpy.plotting.**levelplot**(*data: pandas.core.frame.DataFrame*, *level: str*, *cols: Union[list, str]*, *hue: str = None*, *order: Union[list, str] = None*, *hue_order: Union[list, str] = None*, *func: Callable = <function distplot>*, *summary_title: bool = True*, *level_title: bool = True*, *do_print: bool = False*, *width: int = None*, *height: int = None*, *return_fig_ax: bool = None*, *kwargs_subplots_adjust: Mapping[KT, VT_co] = None*, *kwargs_summary: Mapping[KT, VT_co] = None*, ***kwargs*) → Union[None, tuple]

Plots a plot for each specified column for each level of a certain column plus a summary plot

Parameters

- **data** – Pandas DataFrame containing named data, optional if vector data is used
- **level** – the name of the column to split the plots by, must be in data
- **cols** – the columns to create plots for, defaults to all numeric columns [optional]
- **hue** – Further split the plot by the levels of this variable [optional]
- **order** – Either a string describing how the (hue) levels or to be ordered or an explicit list of levels to be used for plotting. Accepted strings are:
  - `sorted`: following python standard sorting conventions (alphabetical for string, ascending for value)

  - `inv`: following python standard sorting conventions but in inverse order
  - `count`: sorted by value counts
  - `mean`, `mean_ascending`, `mean_descending`: sorted by mean value, defaults to descending
  - `median`, `mean_ascending`, `median_descending`: sorted by median value, defaults to descending
- **`hue_order`** – Either a string describing how the (hue) levels or to be ordered or an explicit list of levels to be used for plotting. Accepted strings are:
  - `sorted`: following python standard sorting conventions (alphabetical for string, ascending for value)
  - `inv`: following python standard sorting conventions but in inverse order
  - `count`: sorted by value counts
  - `mean`, `mean_ascending`, `mean_descending`: sorted by mean value, defaults to descending
  - `median`, `mean_ascending`, `median_descending`: sorted by median value, defaults to descending
- **`func`** – function to use for plotting, must support 1 positional argument, data, hue, ax and kwargs [optional]
- **`summary_title`** – whether to automatically set the summary plot title [optional]
- **`level_title`** – whether to automatically set the level plot title [optional]
- **`do_print`** – Whether to print intermediate steps to console [optional]
- **`width`** – Width of each individual subplot [optional]
- **`height`** – Height of each individual subplot [optional]
- **`return_fig_ax`** – Whether to return the figure and axes objects as tuple to be captured as fig,ax = . . . , If False pyplot.show() is called and the plot returns None [optional]
- **`kwargs_subplots_adjust`** – other keyword arguments passed to pyplot.subplots_adjust [optional]
- **`kwargs_summary`** – other keyword arguments passed to summary distplot, if None uses kwargs [optional]
- **`kwargs`** – other keyword arguments passed to func [optional]

Returns see return_fig_ax

get_legends

`hhpy.plotting.`**`get_legends`**(*ax: matplotlib.axes._axes.Axes = None*) → list

returns all legends on a given axis, useful if you have a secaxis

Parameters **`ax`** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]

Returns list of legends

facet_wrap

hhpy.plotting.**facet_wrap**(*func: Callable, data: pandas.core.frame.DataFrame, facet: Union[list, str], *args, facet_type: str = None, col_wrap: int = 4, width: int = None, height: int = None, catch_error: bool = True, return_fig_ax: bool = None, sharex: bool = False, sharey: bool = False, show_xlabel: bool = True, x_tick_rotation: int = None, y_tick_rotation: int = None, ax_title: str = 'set', order: Union[list, str] = None, subplots_kws: Mapping[KT, VT_co] = None, **kwargs*)

modeled after r's facet_wrap function. Wraps a number of subplots onto a 2d grid of subplots while creating a new line after col_wrap columns. Uses a given plot function and creates a new plot for each facet level.

Parameters

- **func** – Any plot function. Must support keyword arguments data and ax
- **data** – Pandas DataFrame containing named data, optional if vector data is used
- **facet** – The column / list of columns to facet over.
- **args** – passed to func
- **facet_type** – one of ['group', 'cols', None]. If group facet is treated as the column creating the facet levels and a subplot is created for each level. If cols each facet is in turn passed as the first positional argument to the plot function func. If None then the facet_type is inferred: a single facet value will be treated as group and multiple facet values will be treated as cols.
- **col_wrap** – After how many columns to create a new line of subplots [optional]
- **width** – Width of each individual subplot [optional]
- **height** – Height of each individual subplot [optional]
- **catch_error** – whether to keep going in case of an error being encountered in the plot function [optional]
- **return_fig_ax** – Whether to return the figure and axes objects as tuple to be captured as fig,ax = . . . , If False pyplot.show() is called and the plot returns None [optional]
- **sharex** – Whether to share the x axis [optional]
- **sharey** – Whether to share the y axis [optional]
- **show_xlabel** – whether to show the x label for each subplot
- **x_tick_rotation** – x tick rotation for each subplot
- **y_tick_rotation** – y tick rotation for each subplot
- **ax_title** – one of ['set','hide'], if set sets axis title to facet name, if hide forcefully hids axis title
- **order** – Either a string describing how the (hue) levels or to be ordered or an explicit list of levels to be used for plotting. Accepted strings are:
  - `sorted`: following python standard sorting conventions (alphabetical for string, ascending for value)
  - `inv`: following python standard sorting conventions but in inverse order
  - `count`: sorted by value counts
  - `mean`, `mean_ascending`, `mean_descending`: sorted by mean value, defaults to descending

- `median`, `mean_ascending`, `median_descending`: sorted by median value, defaults to descending

- **subplots_kws** – other keyword arguments passed to pyplot.subplots
- **kwargs** – other keyword arguments passed to func

Returns if return_fig_ax: figure and axes objects as tuple, else None

Examples

Check out the example notebook

get_subax

hhpy.plotting.**get_subax**(*ax: Union[matplotlib.axes._axes.Axes, numpy.ndarray], row: int = None, col: int = None, rows_prio: bool = True*) → matplotlib.axes._axes.Axes

shorthand to get around the fact that ax can be a 1D array or a 2D array (for subplots that can be 1x1,1xn,nx1)

Parameters

- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]
- **row** – Row index [optional]
- **col** – Column index [optional]
- **rows_prio** – decides if to use row or col in case of a 1xn / nx1 shape (False means cols get priority)

Returns The matplotlib.pyplot.Axes object with the plot on it

ax_as_list

hhpy.plotting.**ax_as_list**(*ax: Union[matplotlib.axes._axes.Axes, numpy.ndarray]*) → list

takes any Axes and turns them into a list

Parameters **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]

Returns List containing the subaxes

ax_as_array

hhpy.plotting.**ax_as_array**(*ax: Union[matplotlib.axes._axes.Axes, numpy.ndarray]*) → numpy.ndarray

takes any Axes and turns them into a numpy 2D array

Parameters **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]

Returns Numpy 2D array containing the subaxes

rmsdplot

hhpy.plotting.**rmsdplot**(*x: str, data: pandas.core.frame.DataFrame, groups: Union[Sequence[T_co], str] = None, hue: str = None, hue_order: Union[Sequence[T_co], str] = None, cutoff: float = 0, ax: matplotlib.axes._axes.Axes = None, color_as_balance: bool = False, balance_cutoff: float = None, rmsd_as_alpha: bool = False, sort_by_hue: bool = False, palette=None, barh_kws=None, **kwargs*)

creates a seaborn.barplot showing the rmsd calculating *df_rmsd()*

Parameters

- **x** – Name of the x variable in data or vector data
- **data** – Pandas DataFrame containing named data, optional if vector data is used
- **groups** – the columns to calculate the rmsd for, defaults to all columns [optional]
- **hue** – Further split the plot by the levels of this variable [optional]
- **hue_order** – Either a string describing how the (hue) levels or to be ordered or an explicit list of levels to be used for plotting. Accepted strings are:
  - `sorted`: following python standard sorting conventions (alphabetical for string, ascending for value)
  - `inv`: following python standard sorting conventions but in inverse order
  - `count`: sorted by value counts
  - `mean`, `mean_ascending`, `mean_descending`: sorted by mean value, defaults to descending
  - `median`, `mean_ascending`, `median_descending`: sorted by median value, defaults to descending
- **cutoff** – drop rmsd values smaller than cutoff [optional]
- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]
- **color_as_balance** – Whether to color the bars based on how balanced (based on maxperc values) the levels are [optional]
- **balance_cutoff** – If specified: all bars with worse balance (based on maxperc values) than cutoff are shown in red [optional]
- **rmsd_as_alpha** – Whether to use set the alpha values of the columns based on the rmsd value [optional]
- **sort_by_hue** – Whether to sort the plot by hue value [optional]
- **palette** – Collection of colors to be used for plotting. Can be a dictionary for with names for each level or a list of colors or an individual color name. Must be valid colors known to pyplot [optional]
- **barh_kws** – other keyword arguments passed to seaborn.barplot [optional]
- **kwargs** – other keyword arguments passed to `hhpy.ds.rf_rmsd()` [optional]

Returns The matplotlib.pyplot.Axes object with the plot on it

Examples

Check out the example notebook

insert_linebreak

hhpy.plotting.**insert_linebreak**(*s: str*, *pos: int = None*, *frac: float = None*, *max_breaks: int = None*) → str

used to insert linebreaks in strings, useful for formatting axes labels

Parameters

- **s** – string to insert linebreaks into
- **pos** – inserts a linebreak every pos characters [optional]
- **frac** – inserts a linebreak after frac percent of characters [optional]
- **max_breaks** – maximum number of linebreaks to insert [optional]

Returns string with the linebreaks inserted

ax_tick_linebreaks

hhpy.plotting.**ax_tick_linebreaks**(*ax: matplotlib.axes._axes.Axes = None*, *x: bool = True*, *y: bool = True*, ***kwargs*) → None

uses insert_linebreaks to insert linebreaks into the axes ticklabels

Parameters

- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]
- **x** – whether to insert linebreaks into the x axis label [optional]
- **y** – whether to insert linebreaks into the y axis label [optional]
- **kwargs** – other keyword arguments passed to insert_linebreaks

Returns None

annotate_barplot

hhpy.plotting.**annotate_barplot**(*ax: matplotlib.axes._axes.Axes = None*, *x: Sequence[T_co] = None*, *y: Sequence[T_co] = None*, *ci: bool = True*, *ci_newline: bool = True*, *adj_ylim: float = 0.05*, *nr_format: str = None*, *ha: str = 'center'*, *va: str = 'center'*, *offset: int = None*, ***kwargs*) → matplotlib.axes._axes.Axes

automatically annotates a barplot with bar values and error bars (if present). Currently does not work with ticks!

Parameters

- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]
- **x** – Name of the x variable in data or vector data
- **y** – Name of the y variable in data or vector data
- **ci** – whether to annotate error bars [optional]
- **ci_newline** – whether to add a newline between values and error bar values [optional]
- **adj_ylim** – whether to automatically adjust the plot y limits to fit the annotations [optional]
- **nr_format** – The format string used for annotations [optional]
- **ha** – horizontal alignment [optional]

- **va** – vertical alignment [optional]
- **offset** – offset between bar top and annotation center, defaults to rcParams[font.size] [optional]
- **kwargs** – other keyword arguments passed to pyplot.annotate

Returns The matplotlib.pyplot.Axes object with the plot on it

animplot

hhpy.plotting.**animplot**(*data: pandas.core.frame.DataFrame = None, x: str = 'x', y: str = 'y', t: str = 't', lines: Mapping[KT, VT_co] = None, max_interval: int = None, time_per_frame: int = 200, mode: str = None, title: bool = True, title_prefix: str = '', t_format: str = None, fig: matplotlib.figure.Figure = None, ax: matplotlib.axes._axes.Axes = None, color: str = None, label: str = None, legend: bool = False, legend_out: bool = False, legend_kws: Mapping[KT, VT_co] = None, xlim: tuple = None, ylim: tuple = None, ax_facecolor: Union[str, Mapping[KT, VT_co]] = None, grid: bool = False, vline: Union[Sequence[T_co], float] = None, **kwargs*) → Union[IPython.core.display.HTML, matplotlib.animation.FuncAnimation]

wrapper for FuncAnimation to be used with pandas DataFrames. Assumes that you have a DataFrame containing one data point for each x-y-t combination.

If mode is set to jshtml the function is optimized for use with Jupyter Notebook and returns an Interactive JavaScript Widget.

Parameters

- **data** – Pandas DataFrame containing named data, optional if vector data is used
- **x** – Name of the x variable in data
- **y** – Name of the y variable in data
- **t** – Name of the t variable in data
- **lines** – you can also pass lines that you want to animate. Details to follow [optional]
- **max_interval** – max interval at which to abort the animation [optional]
- **time_per_frame** – time per frame [optional]
- **mode** – one of the below [optional]
  - `matplotlib`: Return the matplotlib FuncAnimation object
  - `html`: Returns an HTML5 movie (You need to install ffmpeg for this to work)
  - `jshtml`: Returns an interactive Javascript Widget
- **title** – whether to set the time as plot title [optional]
- **title_prefix** – title prefix to be put in front of the time if title is true [optional]
- **t_format** – format string used to format the time variable in the title [optional]
- **fig** – figure to plot on [optional]
- **ax** – axes to plot on [optional]
- **color** – Color used for plotting, must be known to matplotlib.pyplot [optional]
- **label** – Label to use for the data [optional]

- **legend** – Whether to show a legend [optional]
- **legend_out** – Whether to draw the legend outside of the axis, can also be a location string [optional]
- **legend_kws** – Other keyword arguments passed to pyplot.legend [optional]
- **xlim** – X limits for the axis as tuple, passed to ax.set_xlim() [optional]
- **ylim** – Y limits for the axis as tuple, passed to ax.set_ylim() [optional]
- **ax_facecolor** – passed to ax.set_facecolor, can also be a conditional mapping to change the facecolor at specific timepoints t [optional]
- **grid** – Whether to toggle ax.grid() [optional]
- **vline** – A list of x positions to draw vlines at [optional]
- **kwargs** – other keyword arguments passed to pyplot.plot

Returns see mode

Examples

Check out the example notebook

legend_outside

hhpy.plotting.**legend_outside**(*ax: matplotlib.axes._axes.Axes = None*, *width: float = 0.85*, *loc: str = 'right'*, *legend_space: float = None*, *offset_x: float = 0*, *offset_y: float = 0*, *loc_warn: bool = True*, ***kwargs*)

draws a legend outside of the subplot

Parameters

- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]
- **width** – how far to shrink down the subplot if loc=='right'
- **loc** – one of ['right','bottom'], where to put the legend
- **legend_space** – how far below the subplot to put the legend if loc=='bottom'
- **offset_x** – x offset for the legend
- **offset_y** – y offset for the legend
- **loc_warn** – Whether to trigger a warning if legend loc is not recognized
- **kwargs** – other keyword arguments passed to pyplot.legend

Returns None

set_ax_sym

hhpy.plotting.**set_ax_sym**(*ax: matplotlib.axes._axes.Axes*, *x: bool = True*, *y: bool = True*)

automatically sets the select axes to be symmetrical

Parameters

- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]
- **x** – whether to set x axis to be symmetrical
- **y** – whether to set y axis to be symmetrical

Returns None

custom_legend

hhpy.plotting.**custom_legend**(*colors: Union[list, str], labels: Union[list, str], do_show=True*) → Optional[list]

uses patches to create a custom legend with the specified colors

Parameters

- **colors** – list of matplotlib colors to use for the legend
- **labels** – list of labels to use for the legend
- **do_show** – whether to show the created legend

Returns if do_show: None, else handles

stemplot

hhpy.plotting.**stemplot**(*x*, *y*, *data=None*, *ax=None*, *color='xkcd:blue'*, *baseline=0*, *kwline=None*, ***kwargs*)

modeled after pyplot.stemplot but more customizeable

Parameters

- **x** – Name of the x variable in data or vector data
- **y** – Name of the y variable in data or vector data
- **data** – Pandas DataFrame containing named data, optional if vector data is used
- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]
- **color** – Color used for plotting, must be known to matplotlib.pyplot [optional]
- **baseline** – where to draw the baseline for the stemplot
- **kwline** – other keyword arguments passed to pyplot.plot
- **kwargs** – other keyword arguments passed to pyplot.scatter

Returns The matplotlib.pyplot.Axes object with the plot on it

get_twin

hhpy.plotting.**get_twin**(*ax: matplotlib.axes._axes.Axes*) → Optional[matplotlib.axes._axes.Axes]

get the twin axis from an Axes object

Parameters **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]

Returns the twin axis if it exists, else None

get_axlim

hhpy.plotting.**get_axlim**(*ax: matplotlib.axes._axes.Axes*, *xy: Optional[str] = None*) → Union[tuple, Mapping[KT, VT_co]]

Wrapper function to get x limits, y limits or both with one function call

Parameters

- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]
- **xy** – one of [‘x’, ‘y’, ‘xy’, None]

Returns if xy is ‘xy’ or None returns a dictionary else returns x or y lims as tuple

set_axlim

hhpy.plotting.**set_axlim**(*ax: matplotlib.axes._axes.Axes, lim: Union[Sequence[T_co], Mapping[KT, VT_co]], xy: Optional[str] = None*)

Wrapper function to set both x and y limits with one call

Parameters

- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]
- **lim** – axes limits as tuple or Mapping
- **xy** – one of [‘x’, ‘y’, ‘xy’, None]

Returns None

share_xy

hhpy.plotting.**share_xy**(*ax: matplotlib.axes._axes.Axes*, *x: bool = True*, *y: bool = True*, *mode: str = 'all'*, *adj_twin_ax: bool = True*)

set the subplots on the Axes to share x and/or y limits WITHOUT sharing x and y legends. If you want that please use pyplot.subplots(share_x=True,share_y=True) when creating the plots.

Parameters

- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]
- **x** – whether to share x limits [optional]
- **y** – whether to share y limits [optional]
- **mode** – one of [‘all’, ‘row’, ‘col’], if all shares across all subplots, else just across rows / columns
- **adj_twin_ax** – whether to also adjust twin axes

Returns None

share_legend

hhpy.plotting.**share_legend**(*ax: matplotlib.axes._axes.Axes*, *keep_i: int = None*)

removes all legends except for i from an Axes object

Parameters

- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]
- **keep_i** – index of the plot whose legend you want to keep

Returns None

barplot_err

hhpy.plotting.**barplot_err**(*x: str, y: str, xerr: str = None, yerr: str = None, data: pandas.core.frame.DataFrame = None, **kwargs*) → matplotlib.axes._axes.Axes

extension on seaborn barplot that allows for plotting errorbars with preprocessed data. The idea is based on this StackOverflow question

Parameters

- **x** – Name of the x variable in data
- **y** – Name of the y variable in data
- **xerr** – variable to use as x error bars [optional]
- **yerr** – variable to use as y error bars [optional]
- **data** – Pandas DataFrame containing named data
- **kwargs** – other keyword arguments passed to seaborn barplot

Returns The matplotlib.pyplot.Axes object with the plot on it

countplot

hhpy.plotting.**countplot**(*x: Union[Sequence[T_co], str] = None, data: pandas.core.frame.DataFrame = None, hue: str = None, ax: matplotlib.axes._axes.Axes = None, order: Union[Sequence[T_co], str] = None, hue_order: Union[Sequence[T_co], str] = None, normalize_x: bool = False, normalize_hue: bool = False, palette: Union[Mapping[KT, VT_co], Sequence[T_co], str] = None, x_tick_rotation: int = None, count_twinx: bool = False, hide_legend: bool = False, annotate: bool = True, annotate_format: str = ',.0f', legend_loc: str = 'upper right', barplot_kws: Mapping[KT, VT_co] = None, count_twinx_kws: Mapping[KT, VT_co] = None, **kwargs*)

Based on seaborn barplot but with a few more options, uses `df_count()`

Parameters

- **x** – Name of the x variable in data or vector data
- **data** – Pandas DataFrame containing named data, optional if vector data is used
- **hue** – Further split the plot by the levels of this variable [optional]
- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]
- **order** – Either a string describing how the (hue) levels or to be ordered or an explicit list of levels to be used for plotting. Accepted strings are:
  - `sorted`: following python standard sorting conventions (alphabetical for string, ascending for value)
  - `inv`: following python standard sorting conventions but in inverse order
  - `count`: sorted by value counts
  - `mean`, `mean_ascending`, `mean_descending`: sorted by mean value, defaults to descending
  - `median`, `mean_ascending`, `median_descending`: sorted by median value, defaults to descending

- **hue_order** – Either a string describing how the (hue) levels or to be ordered or an explicit list of levels to be used for plotting. Accepted strings are:
  - `sorted`: following python standard sorting conventions (alphabetical for string, ascending for value)
  - `inv`: following python standard sorting conventions but in inverse order
  - `count`: sorted by value counts
  - `mean`, `mean_ascending`, `mean_descending`: sorted by mean value, defaults to descending
  - `median`, `mean_ascending`, `median_descending`: sorted by median value, defaults to descending
- **normalize_x** – Whether to normalize x, causes the sum of each x group to be 100 percent [optional]
- **normalize_hue** – Whether to normalize hue, causes the sum of each hue group to be 100 percent [optional]
- **palette** – Collection of colors to be used for plotting. Can be a dictionary for with names for each level or a list of colors or an individual color name. Must be valid colors known to pyplot [optional]
- **x_tick_rotation** – Set x tick label rotation to this value [optional]
- **count_twinx** – Whether to plot the count values on the second axis (if using normalize) [optional]
- **hide_legend** – Whether to hide the legend [optional]
- **annotate** – Whether to use annotate_barplot [optional]
- **annotate_format** – The format string used for annotations [optional]
- **legend_loc** – Location of the legend, one of [bottom, right] or accepted value of pyplot.legendIf in [bottom, right] legend_outside is used, else pyplot.legend [optional]
- **barplot_kws** – Additional keyword arguments passed to seaborn.barplot [optional]
- **count_twinx_kws** – Additional keyword arguments passed to pyplot.plot [optional]
- **kwargs** – Additional keyword arguments passed to *df_count()* [optional]

Returns The matplotlib.pyplot.Axes object with the plot on it

quantile_plot

hhpy.plotting.**quantile_plot**(*x: Union[Sequence[T_co], str], data: pandas.core.frame.DataFrame = None, qs: Union[Sequence[T_co], float] = None, x2: str = None, hue: str = None, hue_order: Union[Sequence[T_co], str] = None, to_abs: bool = False, ax: matplotlib.axes._axes.Axes = None, **kwargs*) → matplotlib.axes._axes.Axes

plots the specified quantiles of a Series using seaborn.barplot

Parameters

- **x** – Name of the x variable in data or vector data
- **data** – Pandas DataFrame containing named data, optional if vector data is used
- **qs** – Quantile levels [optional]

- **x2** – if specified: subtracts x2 from x before calculating quantiles [optional]
- **hue** – Further split the plot by the levels of this variable [optional]
- **hue_order** – Either a string describing how the (hue) levels or to be ordered or an explicit list of levels to be used for plotting. Accepted strings are:
  - `sorted`: following python standard sorting conventions (alphabetical for string, ascending for value)
  - `inv`: following python standard sorting conventions but in inverse order
  - `count`: sorted by value counts
  - `mean`, `mean_ascending`, `mean_descending`: sorted by mean value, defaults to descending
  - `median`, `mean_ascending`, `median_descending`: sorted by median value, defaults to descending
- **to_abs** – Whether to cast the values to absolute before proceeding [optional]
- **ax** – The matplotlib.pyplot.Axes object to plot on, defaults to current axis [optional]
- **kwargs** – other keyword arguments passed to seaborn.barplot

Returns The matplotlib.pyplot.Axes object with the plot on it

plotly_aggplot

hhpy.plotting.**plotly_aggplot**(*data: pandas.core.frame.DataFrame, x: Union[int, float, str, bytes, None], y: Union[int, float, str, bytes, None], hue: Union[int, float, str, bytes, None] = None, groupby: Union[Sequence[T_co], int, float, str, bytes, None, AbstractSet[T_co]] = None, sep: str = ';', agg: str = 'sum', hue_order: Union[list, str] = None, x_min: Union[int, float, str, bytes, None] = None, x_max: Union[int, float, str, bytes, None] = None, y_min: Union[int, float, str, bytes, None] = None, y_max: Union[int, float, str, bytes, None] = None, mode: str = 'lines+markers', title: str = None, xaxis_title: str = None, yaxis_title: str = None, label_maxchar: int = 15, direction: str = 'up', showactive: bool = True, dropdown_x: float = 0, dropdown_y: float = -0.1, fig: plotly.graph_objs._figure.Figure = None, do_print: bool = True, kws_dropdown: Mapping[KT, VT_co] = None, kws_fig: Mapping[KT, VT_co] = None, **kwargs*) → plotly.graph_objs._figure.Figure

create a (grouped) plotly aggplot that let's you select the groupby categories

Parameters

- **data** – Pandas DataFrame containing named data, optional if vector data is used
- **x** – Name of the x variable in data
- **y** – Name of the y variable in data
- **hue** – Further split the plot by the levels of this variable [optional]
- **groupby** – Column name(s) to split the plot by [optional]
- **sep** – Separator used for groupby columns [optional]
- **agg** – Aggregate function to use [optional]

- **hue_order** – Either a string describing how the (hue) levels or to be ordered or an explicit list of levels to be used for plotting. Accepted strings are:
  - `sorted`: following python standard sorting conventions (alphabetical for string, ascending for value)
  - `inv`: following python standard sorting conventions but in inverse order
  - `count`: sorted by value counts
  - `mean`, `mean_ascending`, `mean_descending`: sorted by mean value, defaults to descending
  - `median`, `mean_ascending`, `median_descending`: sorted by median value, defaults to descending
- **x_min** – Lower limit for the x axis [optional]
- **x_max** – Upper limit for the x axis [optional]
- **y_min** – Lower limit for the y axis [optional]
- **y_max** – Upper limit for the y axis [optional]
- **mode** – plotly mode [optional]
- **title** – Figure title, passed to plotly.Figure.update_layout [optional]
- **xaxis_title** – x axis title, passed to plotly.Figure.update_layout [optional]
- **yaxis_title** – y axis title, passed to plotly.Figure.update_layout [optional]
- **label_maxchar** – Maximum allowed number of characters of the labels [optional]
- **direction** – One of ['up', 'down'] , direction of the dropdown [optional]
- **showactive** – Whether to show the active selection in the dropdown [optional]
- **dropdown_x** – x position of the first dropdown [optional]
- **dropdown_y** – y position of the first dropdown [optional]
- **fig** – The plotly.Figure object to draw the plot on [optional]
- **do_print** – Whether to print intermediate steps to console [optional]
- **kws_dropdown** – Other keyword arguments passed to the dropdown updatemenu [optional]
- **kws_fig** – other keyword arguments passed to plotly.graph_objects.Figure [optional]
- **kwargs** – other keyword arguments passed to plotly.graph_objects.scatter [optional]

Returns plotly Figure with the plot on it

CHAPTER 2

quickstart

Try these functions to get a feeling of what the package does

Main

Quick convenience

- *tprint*
- *is_list_like*
- force_list

DS

Quick ways to filter, score and split DataFrames

- *qf*
- *f_score*
- *k_split*
- *df_split*

Modelling

Modelling provides an extension on any statistical model that implements .fit and .predict that allows for easy multi modelling.

- *Model*
- *Models*

Plotting

Animated plots and distribution plots

- *animplot*
- *distplot*
- *facet_wrap*

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

h

Index

H

I

K

L

M

O

P

Q

R

S

T

W