bff

Release 0.2.7+14.g26dad64

Axel Fahy

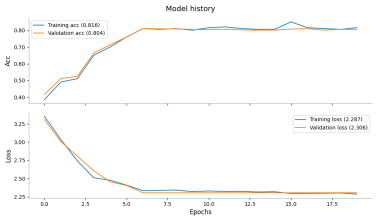
Oct 12, 2020

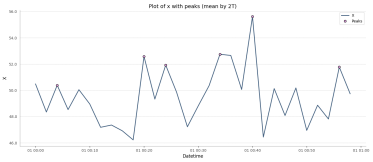
GETTING STARTED

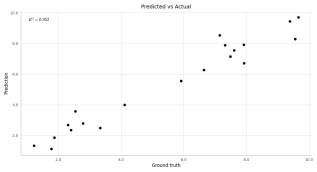
Best Fancy Functions, your Best Friend Forever

The bff package contains some utility functions from plots to data manipulations and could become your new bff.

The goal of this package is to have easy access of the functions I am using frequently on projects.







This is still a work in progress, contributions are welcome.

CHAPTER

ONE

QUICK START

1.1 Install bff

If you use `pip`, you can install it with:

```
pip install bff
```

1.2 Examples

Here are some examples of possible plots from the *plot* module.

1.2.1 Examples of plots

This notebook presents examples of the `bff.plot` module.

For each function, the `ax` can be provided and is returned by the function. This allow to plot multiple things on the same axis and modify it if needed.

```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

import bff.plot as bplt

np.random.seed(42)
```

```
# Variables with fake data to display.
history = {
    'loss': [2.3517270615352457, 2.3737808328178063, 2.342552079627262,
             2.310529179309481, 2.3773420348239305, 2.3290258640020935,
             2.3345777257603015, 2.336566770496081, 2.34276949460782,
             2.321525989465378, 2.3300879552735756, 2.3224288386915197,
             2.324129374183003, 2.3158747431021838, 2.3194296072475873,
             2.2962934024369894, 2.296843618603807, 2.298411148876401,
             2.302087271033819, 2.2869889256942213],
    'acc': [0.085427135, 0.09045226, 0.110552765, 0.110552765, 0.06030151,
            0.14070351, 0.110552765, 0.10552764, 0.09045226, 0.10050251,
            0.11557789, 0.12060302, 0.110552765, 0.10552764, 0.1356784,
            0.15075377, 0.11557789, 0.12060302, 0.10552764, 0.14572865],
    'val_loss': [2.3074077556791077, 2.306745302662272, 2.3061152659403104,
```

(continues on next page)

(continued from previous page)

```
                  2.3056061252970226, 2.30513324273213, 2.3046621198808954,
                  2.304321059871107, 2.304280655512054, 2.3042611346560324,
                  2.3042683235268466, 2.3044002410326705, 2.304716517416279,
                  2.3049982415602894, 2.305085456921962, 2.3051163034046187,
                  2.3052417696192022, 2.3052861982219377, 2.305426104982545,
                  2.305481707112173, 2.3055578968795793],
    'val_acc': [0.11610487, 0.11111111, 0.11485643, 0.11360799, 0.11360799,
                0.11985019, 0.11111111, 0.10861423, 0.10861423, 0.10486891,
                0.10362048, 0.096129835, 0.09238452, 0.09113608, 0.09113608,
                0.08739076, 0.08988764, 0.09113608, 0.096129835, 0.09363296]
}

y_true = [1.87032178, 1.2272566 , 9.38496685, 7.91451104, 7.60794146,
          9.65912261, 2.5405396 , 7.31815866, 5.91692937, 2.78676838,
          7.9258648 , 2.31337877, 1.78432016, 9.5559698 , 6.64471696,
          3.33907423, 7.49321025, 7.14822795, 4.11686499, 2.40202043]

y_pred = [1.85161709, 1.33317135, 9.45246137, 7.9198675 , 7.54877922,
          9.7153202 , 3.56777447, 7.88673475, 5.56090322, 2.78851836,
          6.70636033, 2.67531555, 1.13061356, 8.29287223, 6.27275223,
          2.4957286 , 7.14305019, 8.53578604, 3.99890533, 2.35510298]
```

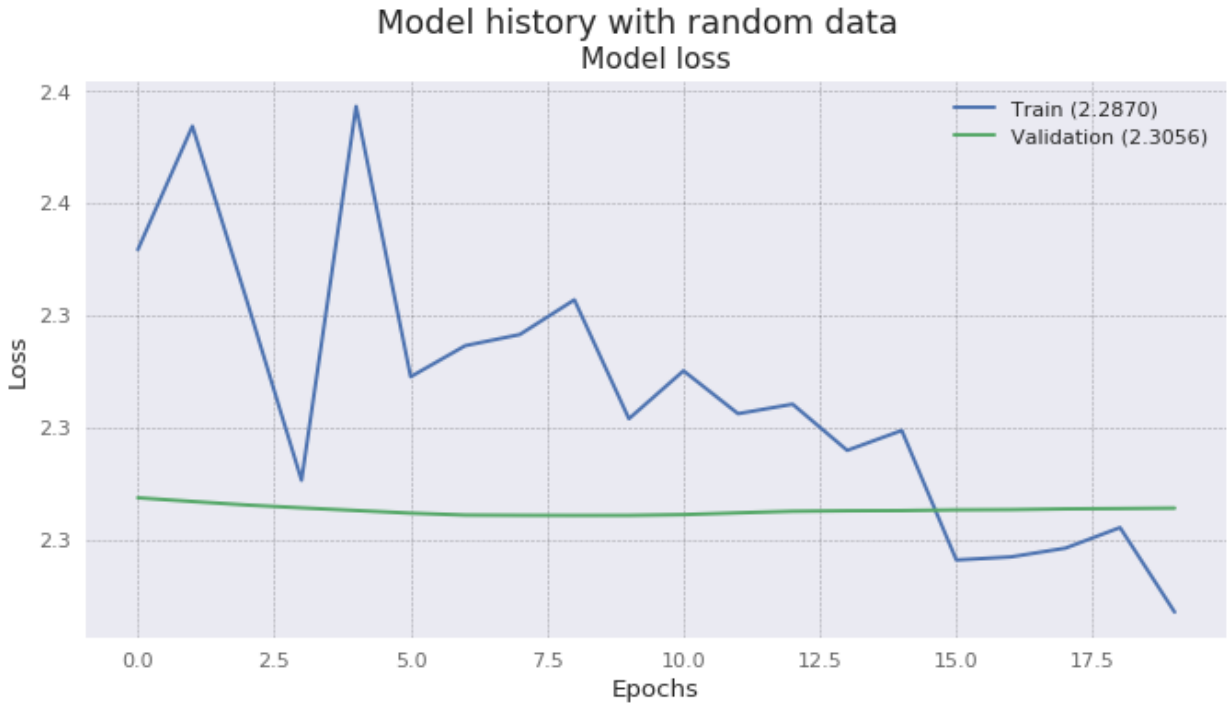
Plot of history

The `plot_history` function can plot the loss and a metric from the `history.history` dictionary usually returned by a Keras model.

Plot of only loss with grid and a different style for matplotlib.

```
bplt.plot_history(history, title='Model history with random data', grid='both',␣
↪figsize=(10, 5), style='seaborn')
```

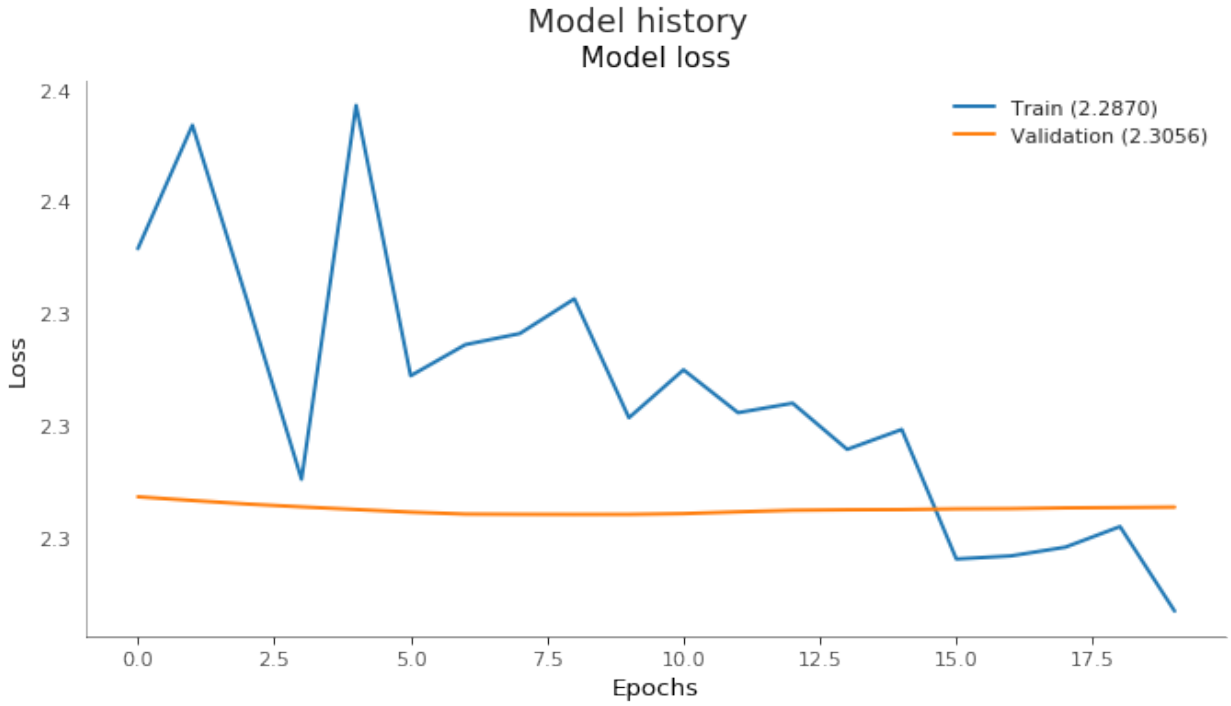
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3f297d78d0>
```



Plot of history using a previously created axis.

```
__, ax = plt.subplots(1, 1, figsize=(10, 5), dpi=80)
bplt.plot_history(history, axes=ax, style='seaborn')
```

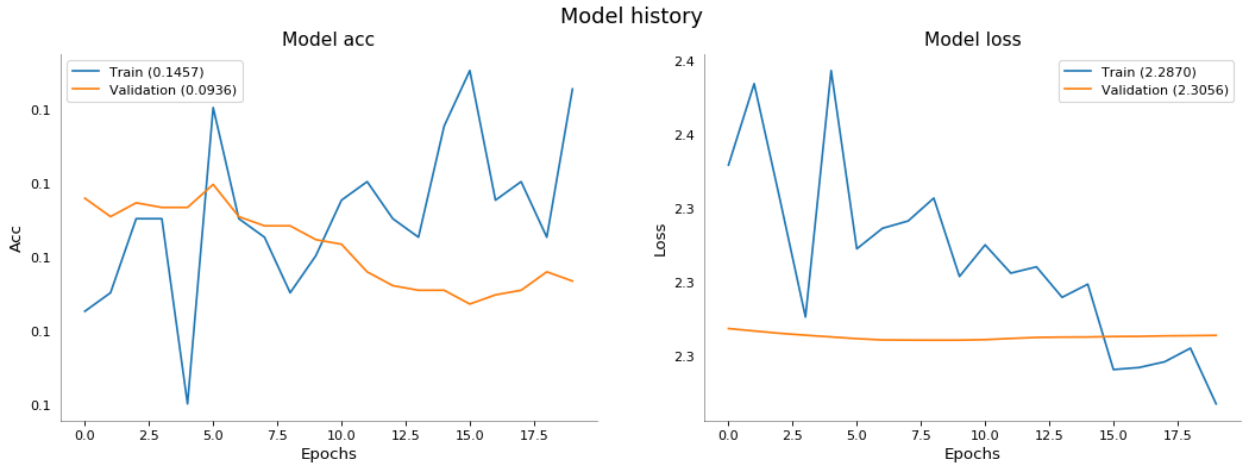
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3f297c34a8>
```



Plot of history with loss and acc.

```
bplt.plot_history(history, metric='acc')
```

```
array([<matplotlib.axes._subplots.AxesSubplot object at 0x7f0fd8b12ac8>,
       <matplotlib.axes._subplots.AxesSubplot object at 0x7f0fd84749b0>],
      dtype=object)
```

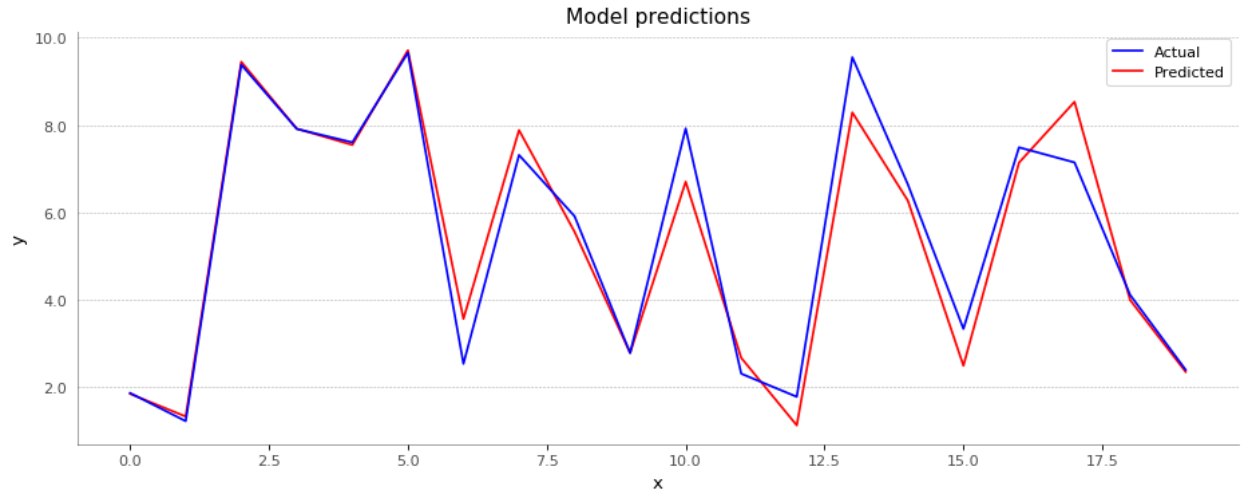


Plot of predictions

Plot of actual and predicted values on the same axis.

```
bplt.plot_predictions(y_true, y_pred)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0fd8c823c8>
```



Plot series

Series can either be plot on the same axis or in different ones in the same figure.

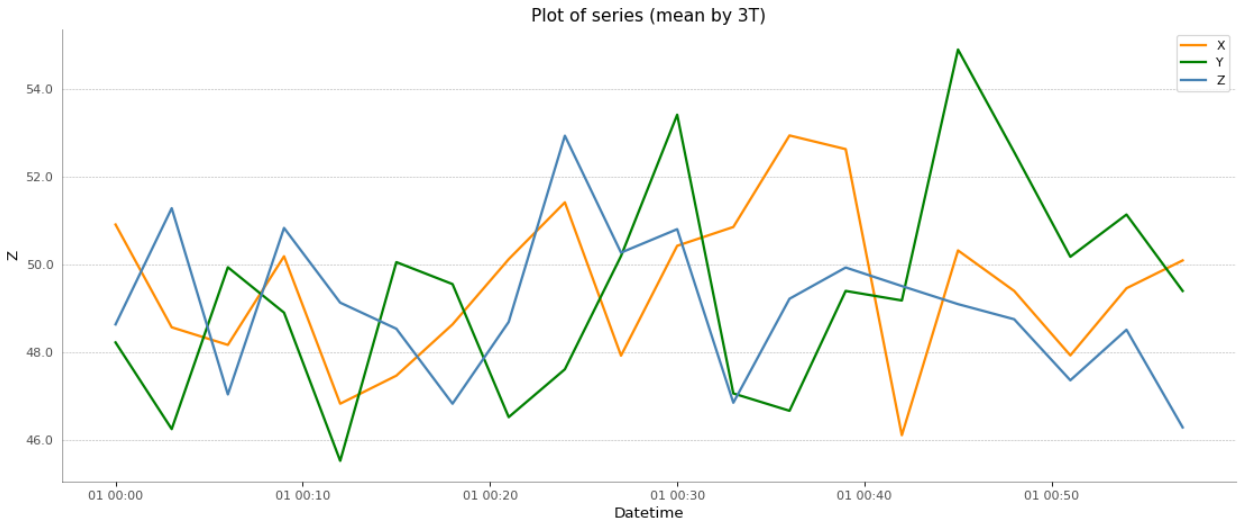
Some fake data are created in a `DataFrame`. The index must be named `datetime`. A different color is assigned to each of the acceleration.

```
AXIS = {'x': 'darkorange', 'y': 'green', 'z': 'steelblue'}
data = (pd.DataFrame(np.random.randint(0, 100, size=(60 * 60, 3)), columns=AXIS.
↪keys())
        .set_index(pd.date_range('2018-01-01', periods=60 * 60, freq='S'))
        .rename_axis('datetime'))

data_miss = (data
             .drop(pd.date_range('2018-01-01 00:05', '2018-01-01 00:07', freq='S'))
             .drop(pd.date_range('2018-01-01 00:40', '2018-01-01 00:41', freq='S'))
             .drop(pd.date_range('2018-01-01 00:57', '2018-01-01 00:59', freq='S'))
             )
```

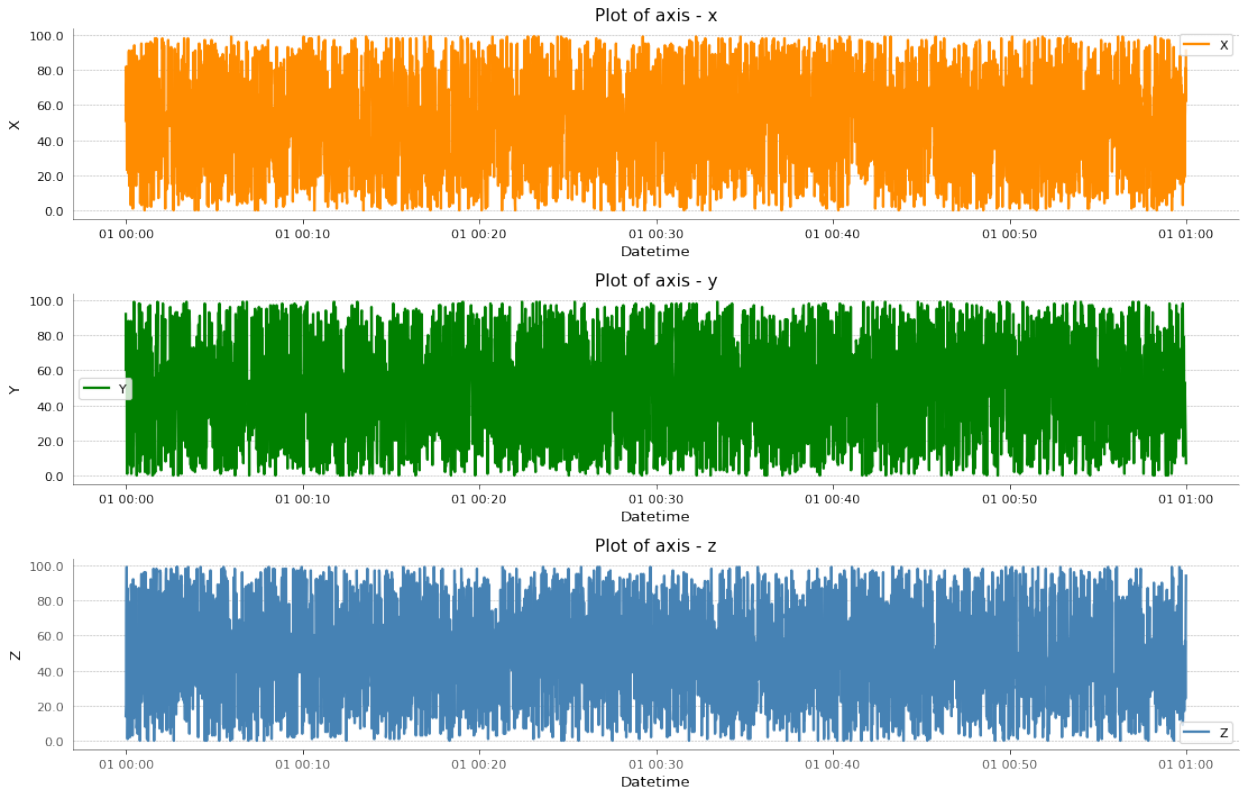
Plot of x, y and z acceleration on the same axis. The function is returning the axis so it can be used in the next plot.

```
ax = bplt.plot_series(data, 'x', groupby='3T', title=f'Plot of all axis', color=AXIS[
↪'x'])
for k in list(AXIS.keys())[1:]:
    bplt.plot_series(data, k, groupby='3T', ax=ax, color=AXIS[k])
```



This time, accelerations are plot on a figure containing a different axis for each acceleration.

```
_, axes = plt.subplots(nrows=len(AXIS), ncols=1, figsize=(14, len(AXIS) * 3), dpi=80)
for i, k in enumerate(AXIS.keys()):
    bplt.plot_series(data, k, ax=axes[i], title=f'Plot of axis - {k}', color=AXIS[k])
```



A resampling of the data is done by averaging each 2 minutes (`2T`). A peak detection is done as well.

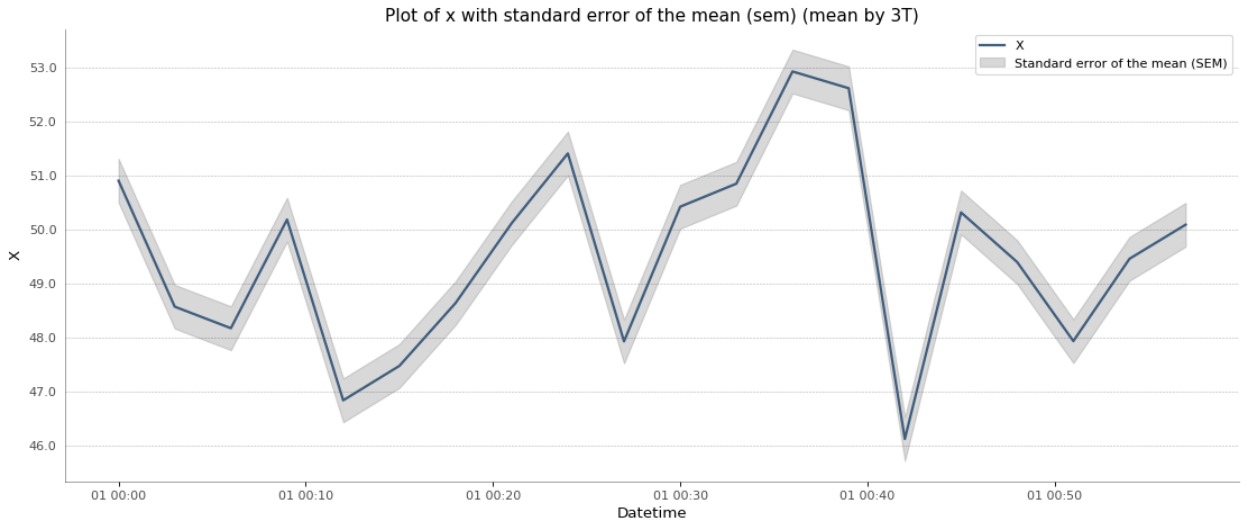
```
bplt.plot_series(data, 'x', groupby='2T', with_peaks=True, title=f'Plot of x with␣
↪peaks')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3f2ad91f98>
```

A resampling of the data is done by averaging each 3 minutes (`2T`). The standard error of the mean (SEM) is plotted as well. This is usefull to see if the data are close to the mean or not since there was a resampling.

```
bplt.plot_series(data, 'x', groupby='3T', with_sem=True, title=f'Plot of x with␣
↪standard error of the mean (sem)')
```

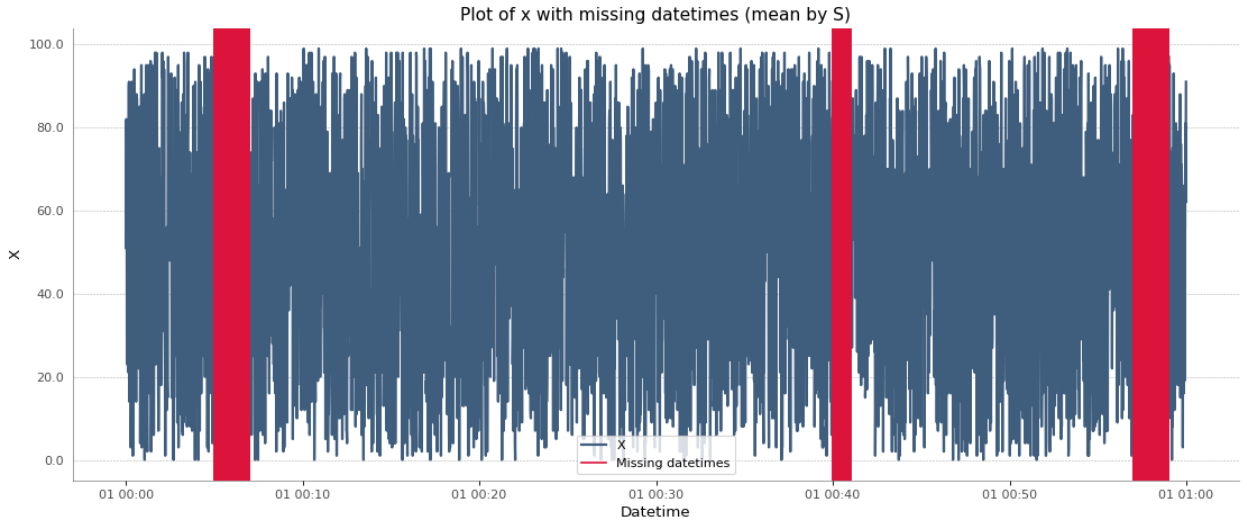
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3f29c38668>
```



Plot of a serie with missing data. By specifying the resampling, we can easily see if some of the datetime are missing.

```
bplt.plot_series(data_miss, 'x', groupby='S', with_missing_datetimes=True,
                 title=f'Plot of x with missing datetimes')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3f2a2e58d0>
```



Same as the previous plot, but with a group by minute (`T`). Since this is regroup by minute, there are less data missing.

```
bplt.plot_series(data_miss, 'x', groupby='T', with_missing_datetimes=True,
                 title=f'Plot of x with missing datetimes')
```

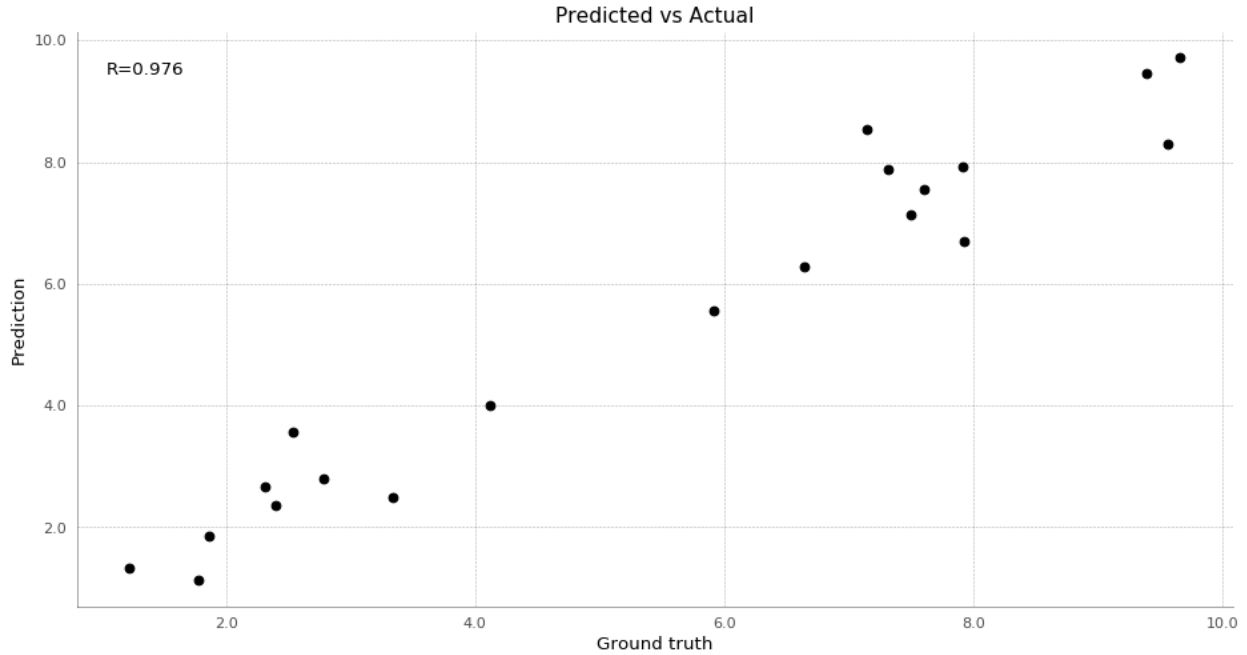
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3f2a30c748>
```

Plot true vs pred

Plot the real data against the predictions. The correlation (`R`) can be calculated or not using the `with_correlation` option.

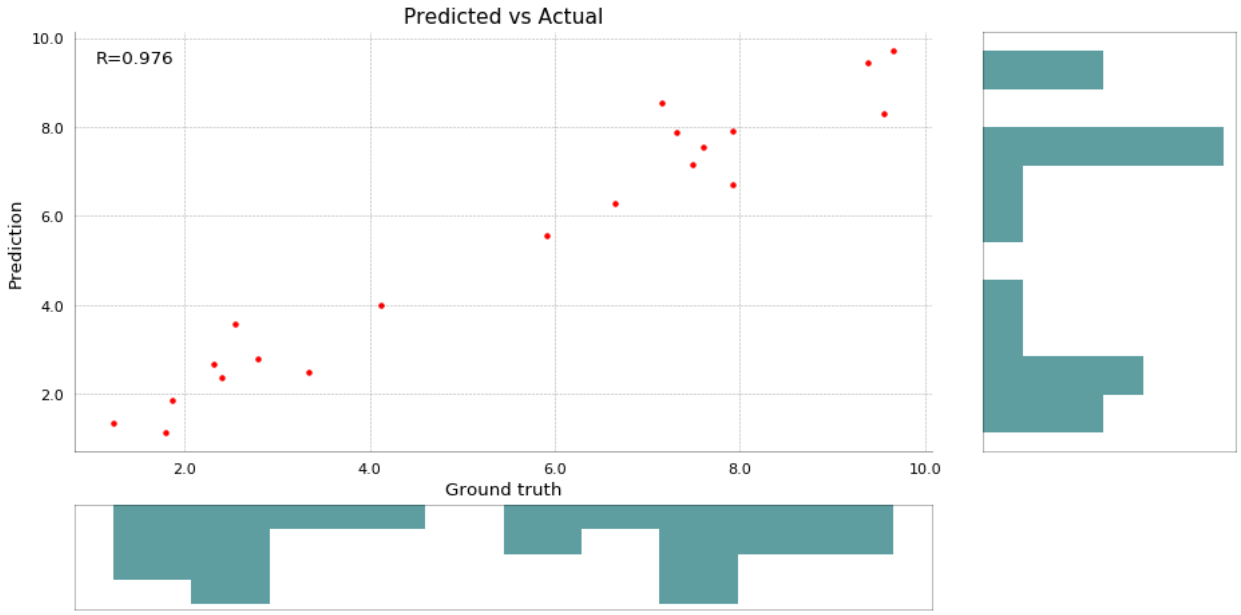
```
bplt.plot_true_vs_pred(y_true, y_pred)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f0fdae202b0>
```



Using the `with_histograms` option, the function will plot histograms on the side, showing the distribution of the data.

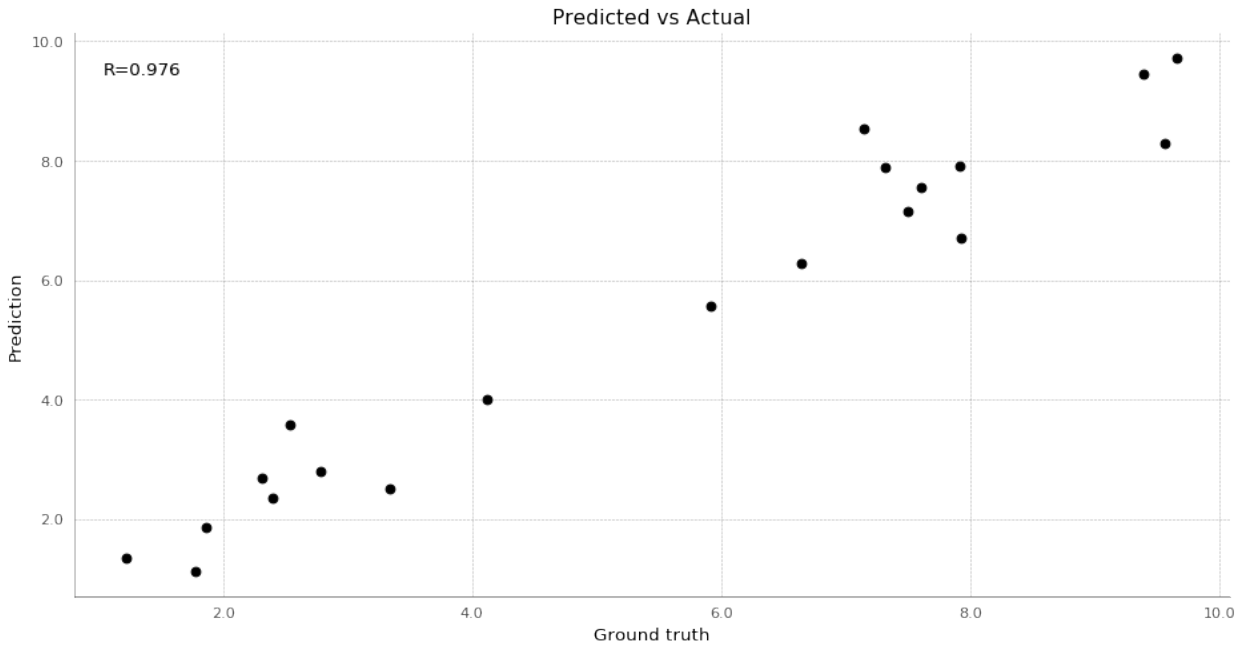
```
ax = bplt.plot_true_vs_pred(y_true, y_pred, with_histograms=True, marker='.', c='r')
```



Plot using a previously created axis.

```
__, ax = plt.subplots(1, 1, figsize=(14, 7), dpi=80)
bplt.plot_true_vs_pred(y_true, y_pred, ax=ax)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3f29b84518>
```



1.3 Development

1.3.1 Setup

The developement environment can be installed as follow:

```
git clone https://github.com/axelfahy/bff.git
cd bff
python -m venv venv-dev
source venv-dev/bin/activate
pip install -r requirements_dev.txt
pip install -e .
```

1.3.2 Unittest

You can run the test using:

```
make all
```

This will run unittests for code and code style checks.

To test plots, images with baseline should be placed in *tests/baseline* and can be generated using `make build-baseline`.

As of *v0.2*, plots are not yet tested in the travis build.

1.4 Contributing

Contributions are welcome!

If you want to contribute, you should proceed as follows:

```
1. Fork it (<https://github.com/yourname/yourproject/fork>)
2. Create your feature branch (`git checkout -b feature/fooBar`)
3. Commit your changes (`git commit -am 'Add some fooBar'`)
4. Push to the branch (`git push origin feature/fooBar`)
5. Create a new Pull Request
```

If this is supposed to be a new realease, the new version must be set in the tag:

```
git tag vx.y.z
git push --tags
```

CHAPTER

TWO

BFF

All of bff's functions.

bff.avg_dicts	Average all the values in the given dictionaries.
bff.cast_to_category_pd	Automatically converts columns of pandas DataFrame that are worth stored as `category` dtype.
bff.concat_with_categories	Concatenation of Pandas DataFrame having categorical columns.
bff.get_peaks	Get the peaks of a time series having datetime as index.
bff.idict	Invert a dictionary.
bff.kwargs_2_list	Convert all single values from keyword arguments into lists.
bff.log_df	Log information on a DataFrame before returning it.
bff.mem_usage_pd	Calculate the memory usage of a pandas object.
bff.normalization_pd	Normalize columns of a pandas DataFrame using the given scaler.
bff.parse_date	Cast str date into datetime format.
bff.pipe_multiprocessing_pd	Compute function on DataFrame with *nb_proc* processes.
bff.plot.plot_correlation	Plot the correlation between variables of a pandas DataFrame.
bff.plot.plot_counter	Plot the values of a counter as a bar plot.
bff.plot.plot_history	Plot the history of the model trained using Keras.
bff.plot.plot_pca_explained_variance_ratio	Plot the explained variance ratio of PCA.
bff.plot.plot_predictions	Plot the predictions of the model.
bff.plot.plot_series	Plot time series with datetime with the given resample (*groupby*).
bff.plot.plot_true_vs_pred	Plot the ground truth against the predictions of the model.
bff.plot.set_thousands_separator	Set thousands separator on the axes.
bff.read_sql_by_chunks	Read SQL query by chunks into a DataFrame.
bff.size_2_square	Return the size of the side to create a square able to contain n elements.
bff.sliding_window	Apply a sliding window over the sequence.
bff.value_2_list	Convert a single value into a list with a single value.

2.1 bff.avg_dicts

bff.**avg_dicts**(**args*)

Average all the values in the given dictionaries.

Dictionaries must only have numerical values. If a key is not present in one of the dictionary, the value is 0.

Parameters ***args** – Dictionaries to average, as positional arguments.

Returns Dictionary with the average of all inputs.

Return type dict

Raises **TypeError** – If a value is not a number.

2.2 bff.cast_to_category_pd

bff.**cast_to_category_pd**(*df*, *deep=True*)

Automatically converts columns of pandas DataFrame that are worth stored as `category` dtype.

To be casted a column must not be numerical, must be hashable and must have less than 50% of unique values.

Parameters

- **df** (*pd.DataFrame*) – DataFrame with the columns to cast.
- **deep** (*bool, default True*) – Whether to perform a deep copy of the original DataFrame.

Returns Optimized copy of the input DataFrame.

Return type pd.DataFrame

Examples

```
>>> import pandas as pd
>>> columns = ['name', 'age', 'country']
>>> df = pd.DataFrame([['John', 24, 'China'],
...                    ['Mary', 20, 'China'],
...                    ['Jane', 25, 'Switzerland'],
...                    ['Greg', 23, 'China'],
...                    ['James', 28, 'China']],
...                   columns=columns)
>>> df
    name  age      country
0   John   24        China
1   Jane   25  Switzerland
2  James   28        China
>>> df.dtypes
name       object
age         int64
country    object
dtype: object
>>> df_optimized = cast_to_category_pd(df)
>>> df_optimized.dtypes
name       object
age         int64
```

(continues on next page)

(continued from previous page)

```
country  category
dtype: object
```

2.3 bff.concat_with_categories

bff.**concat_with_categories**(*df_left*, *df_right*, ***kwargs*)

Concatenation of Pandas DataFrame having categorical columns.

With the *concat* function from Pandas, when merging two DataFrames having categorical columns, categories not present in both DataFrames and with the same code are lost. Columns are cast to *object*, which takes more memory.

In this function, a union of categorical values from both DataFrames is done and both DataFrames are recategorized with the complete list of categorical values before the concatenation. This way, the category field is preserved.

Original DataFrame are copied, hence preserved.

Parameters

- **df_left** (*pd.DataFrame*) – Left DataFrame to merge.
- **df_right** (*pd.DataFrame*) – Right DataFrame to merge.
- ****kwargs** – Additional keyword arguments to be passed to the *pd.concat* function.

Returns Concatenation of both DataFrames.

Return type pd.DataFrame

Examples

```
>>> import pandas as pd
>>> column_types = {'name': 'object',
...                 'color': 'category',
...                 'country': 'category'}
>>> columns = list(column_types.keys())
>>> df_left = pd.DataFrame([['John', 'red', 'China'],
...                         ['Jane', 'blue', 'Switzerland']],
...                        columns=columns).astype(column_types)
>>> df_right = pd.DataFrame([['Mary', 'yellow', 'France'],
...                          ['Fred', 'blue', 'Italy']],
...                         columns=columns).astype(column_types)
>>> df_left
   name color      country
0  John   red        China
1  Jane  blue  Switzerland
>>> df_left.dtypes
name         object
color      category
country    category
dtype: object
```

The following concatenation shows the issue when using the *concat* function from pandas:

```
>>> res_fail = pd.concat([df_left, df_right], ignore_index=True)
>>> res_fail
   name   color      country
0  John     red        China
1  Jane    blue  Switzerland
2  Mary  yellow       France
3  Fred    blue       Italy
>>> res_fail.dtypes
name       object
color      object
country    object
dtype: object
```

All types are back to *object* since not all categorical values were present in both DataFrames.

With this custom implementation, the categorical type is preserved:

```
>>> res_ok = concat_with_categories(df_left, df_right, ignore_index=True)
>>> res_ok
   name   color      country
0  John     red        China
1  Jane    blue  Switzerland
2  Mary  yellow       France
3  Fred    blue       Italy
>>> res_ok.dtypes
name         object
color      category
country    category
dtype: object
```

2.4 bff.get_peaks

bff.**get_peaks**(*s*, *distance_scale=0.04*)

Get the peaks of a time series having datetime as index.

Only the peaks having a height higher than 0.75 quantile are returned and a distance between two peaks at least `df.shape[0]*distance_scale`.

Return the dates and the corresponding value of the peaks.

Parameters

- **s** (*pd.Series*) – Series to get the peaks from, with datetime as index.
- **distance_scale** (*str, default 0.04*) – Scaling for the minimal distances between two peaks. Multiplication of the length of the DataFrame with the *distance_scale* value.

Returns

- **dates** (*np.ndarray*) – Dates when the peaks occur.
- **heights** (*np.ndarray*) – Heights of the peaks at the corresponding dates.

2.5 bff.idict

bff.**idict**(*d*)

Invert a dictionary.

Keys will be become values and values will become keys.

Parameters **d** (*dict of any to hashable*) – Dictionary to invert.

Returns Inverted dictionary.

Return type dict of hashable to any

Raises **TypeError** – If original values are not Hashable.

Examples

```
>>> idict({1: 4, 2: 5})
{4: 1, 5: 2}
>>> idict({1: 4, 2: 4, 3: 6})
{4: 2, 6: 3}
```

2.6 bff.kwargs_2_list

bff.**kwargs_2_list**(***kwargs*)

Convert all single values from keyword arguments into lists.

For each argument provided, if the type is not a sequence, convert the single value into a list. Strings are not considered as a sequence in this scenario.

Parameters ****kwargs** – Parameters passed to the function.

Returns Dictionary with the single values put into a list.

Return type dict

Raises **TypeError** – If a non-keyword argument is passed to the function.

Examples

```
>>> kwargs_2_list(name='John Doe', age=42, children=('Jane Doe', 14))
{'name': ['John Doe'], 'age': [42], 'children': ('Jane Doe', 14)}
>>> kwargs_2_list(countries=['Swiss', 'Spain'])
{'countries': ['Swiss', 'Spain']}
```

2.7 bff.log_df

bff.**log_df**(*df*, *f=<function <lambda>>*, *msg=''*)

Log information on a DataFrame before returning it.

The given function is applied and the result is printed. The original DataFrame is returned, unmodified.

This allows printing debug information in method chaining.

Parameters

- **df** (*pd.DataFrame*) – DataFrame to log.
- **f** (*Callable, default is the shape of the DataFrame*) – Function to apply on the DataFrame and to log.

Returns The DataFrame, unmodified.

Return type pd.DataFrame

Examples

```
>>> import pandas as pd
>>> import pandas.util.testing as tm
>>> df = tm.makeDataFrame().head()
>>> df_res = (df.pipe(log_df)
...           .assign(E=2)
...           .pipe(log_df, f=lambda x: x.head(), msg='My df: \n')
...           .pipe(log_df, lambda x: x.shape, 'New shape=')
...          )
2019-11-04 13:31:34,742 [INFO   ] bff.fancy: (5, 4)
2019-11-04 13:31:34,758 [INFO   ] bff.fancy: My df:
                   A         B         C         D  E
7t93kTGSqJ -0.104845 -1.296579 -0.487572  0.928964  2
P8CEEHf07x -0.462075 -2.426990 -0.538038  0.487148  2
0DlwZOOj83 -1.964108 -1.272991  0.622618 -0.562890  2
LcrsmbFAjk -0.827403 -0.015269 -0.970148  0.683915  2
kHfxaURF8t  0.654381  0.353666 -0.830602  1.788581  2
2019-11-04 13:31:34,758 [INFO   ] bff.fancy: New shape=(5, 5)
```

2.8 bff.mem_usage_pd

bff.**mem_usage_pd**(*pd_obj*, *index=True*, *deep=True*, *details=True*)

Calculate the memory usage of a pandas object.

If *details*, returns a dictionary with the memory usage and type of each column (DataFrames only). Key=column, value=(memory, type). Else returns a dictionary with the total memory usage. Key=`total`, value=memory.

Parameters

- **pd_obj** (*pd.DataFrame or pd.Series*) – DataFrame or Series to calculate the memory usage.
- **index** (*bool, default True*) – If True, include the memory usage of the index.
- **deep** (*bool, default True*) – If True, introspect the data deeply by interrogating object dtypes for system-level memory consumption.

- **details** (*bool, default True*) – If True and a DataFrame is given, give the detail (memory and type) of each column.

Returns Dictionary with the column or total as key and the memory usage as value (with 'MB').

Return type dict of str to str

Raises **AttributeError** – If argument is not a pandas object.

Examples

```
>>> import pandas as pd
>>> df = pd.DataFrame({'A': [f'value{i}' for i in range(100_000)],
...                    'B': [i for i in range(100_000)],
...                    'C': [float(i) for i in range(100_000)]}).set_index('A')
>>> mem_usage_pd(df)
{'total': '7.90 MB'}
>>> mem_usage_pd(df, details=True)
{'Index': {'6.38 MB', 'Index type'},
 'B': {'0.76 MB', dtype('int64')},
 'C': {'0.76 MB', dtype('float64')},
 'total': '7.90 MB'}
>>> serie = df.reset_index()['B']
>>> mem_usage_pd(serie)
{'total': '0.76 MB'}
>>> mem_usage_pd(serie, details=True)
2019-06-24 11:23:39,500 Details is only available for DataFrames.
{'total': '0.76 MB'}
```

2.9 bff.normalization_pd

bff.**normalization_pd**(*df*, *scaler=None*, *columns=None*, *suffix=None*, *new_type=<class 'numpy.float32'>*, ***kwargs*)

Normalize columns of a pandas DataFrame using the given scaler.

If the columns are not provided, will normalize all the numerical columns.

If the original columns are integers (*RangeIndex*), it is not possible to replace them. This will create new columns having the same integer, but as a string name.

By default, if the suffix is not provided, columns are overridden.

Parameters

- **df** (*pd.DataFrame*) – DataFrame to normalize.
- **scaler** (*TransformerMixin, default MinMaxScaler*) – Scaler of sklearn to use for the normalization.
- **columns** (*sequence of str, default None*) – Columns to normalize. If None, normalize all numerical columns.
- **suffix** (*str, default None*) – If provided, create the normalization in new columns having this suffix.
- **new_type** (*np.dtype, default np.float32*) – New type for the columns.
- ****kwargs** – Additional keyword arguments to be passed to the scaler function from sklearn.

Returns DataFrame with the normalized columns.

Return type pd.DataFrame

Examples

```
>>> import numpy as np
>>> import pandas as pd
>>> from sklearn.preprocessing import StandardScaler
>>> data = {'x': [123, 27, 38, 45, 67], 'y': [456, 45.4, 32, 34, 90]}
>>> df = pd.DataFrame(data)
>>> df
     x      y
0  123  456.0
1   27   45.4
2   38   32.0
3   45   34.0
4   67   90.0
>>> df_std = df.pipe(normalization_pd, columns=['x'], scaler=StandardScaler)
>>> df_std
          x      y
0  1.847198  456.0
1 -0.967580   45.4
2 -0.645053   32.0
3 -0.439809   34.0
4  0.205244   90.0
>>> df_min_max = normalization_pd(df, suffix='_norm', feature_range=(0, 2),
...                               new_type=np.float64)
>>> df_min_max
     x      y    x_norm    y_norm
0  123  456.0  2.000000  2.000000
1   27   45.4  0.000000  0.063208
2   38   32.0  0.229167  0.000000
3   45   34.0  0.375000  0.009434
4   67   90.0  0.833333  0.273585
```

2.10 bff.parse_date

bff.**parse_date**(*func=None*, *date_fields='date'*)

Cast str date into datetime format.

This decorator casts string arguments of a function to datetime.datetime type. This allows to specify either string of datetime format for a function argument. The name of the parameters to cast must be specified in the *date_fields*.

The cast is done using the *parse* function from the dateutil package. All supported format are those from the library and may evolve.

In order to use the decorator both with or without parenthesis when calling it without parameter, the *date_fields* argument is keyword only. This allows checking if the parameter was given or not.

Parameters

- **func** (*Callable*) – Function with the arguments to parse.
- **date_fields** (*Sequence of str, default 'date'*) – Sequence containing the fields with dates.

Returns Function with the date fields cast to datetime.datetime type.

Return type Callable

Examples

```
>>> @parse_date
... def dummy_function(**kwargs):
...     print(f'Args {kwargs}')
...
>>> dummy_function(date='20190325')
Args {'date': datetime.datetime(2019, 3, 25, 0, 0)}
>>> dummy_function(date='Mon, 21 March, 2015')
Args {'date': datetime.datetime(2015, 3, 21, 0, 0)}
>>> dummy_function(date='2019-03-09 08:03:00')
Args {'date': datetime.datetime(2019, 3, 9, 8, 3)}
>>> dummy_function(date='March 27 2019')
Args {'date': datetime.datetime(2019, 3, 27, 0, 0)}
>>> dummy_function(date='wrong string')
Value `wrong string` for field `date` is not convertible to a date format.
Args {'date': 'wrong string'}
```

2.11 bff.pipe_multiprocessing_pd

bff.**pipe_multiprocessing_pd**(*df*, *func*, *, *nb_proc=None*, ***kwargs*)

Compute function on DataFrame with *nb_proc* processes.

The given function must return a new DataFrame. Rows must be independant and not depend from a value generated using the whole DataFrame.

The function uses as many processes as cpu available on the machine.

The DataFrame is splitted in *nb_proc* processes and then each splitted DataFrame is computed by a different process. The results are then concatenated an returned.

Parameters

- **df** (*pd.DataFrame*) – DataFrame that must be computed by the function.
- **func** (*function*) – Function that takes the DataFrame as input.
- **nb_proc** (*Union[int, None], default None*) – Number of processor to use. If not provided, uses *multiprocessing.cpu_count()* number of processes.
- ****kwargs** – Additional keyword arguments to be passed to *func*.

Returns Return the DataFrame computed by *func*.

Return type pd.DataFrame

2.12 bff.plot.plot_correlation

bff.plot.**plot_correlation**(*df*, *already_computed=False*, *method='pearson'*, *title='Correlation between variables'*, *ax=None*, *rotation_xticks=90*, *rotation_yticks=None*, *figsize=13, 10*, *dpi=80*, *style='white'*, ***kwargs*)

Plot the correlation between variables of a pandas DataFrame.

The computing of the correlation can be done either in the function or before.

Parameters

- **df** (*pd.DataFrame*) – DataFrame with the values or the correlations.
- **already_computed** (*bool, default False*) – Set to True if the DataFrame already contains the correlations.
- **method** (*str, default 'pearson'*) – Type of normalization. See pandas.DataFrame.corr for possible values.
- **title** (*str, default 'Correlation between variables'*) – Title for the plot (axis level).
- **ax** (*plt.axes, optional*) – Axes from matplotlib, if None, new figure and axes will be created.
- **rotation_xticks** (*float or None, default 90*) – Rotation of x ticks if any.
- **rotation_yticks** (*float, optional*) – Rotation of x ticks if any. Set to 90 to put them vertically.
- **figsize** (*Tuple[int, int], default (13, 10)*) – Size of the figure to plot.
- **dpi** (*int, default 80*) – Resolution of the figure.
- **style** (*str, default 'white'*) – Style to use for seaborn.axes_style. The style is use only in this context and not applied globally.
- ****kwargs** – Additional keyword arguments to be passed to the *sns.heatmap* function from seaborn.

Returns Axes returned by the *plt.subplots* function.

Return type plt.axes

2.13 bff.plot.plot_counter

bff.plot.**plot_counter**(*counter*, *label_x='x'*, *label_y='y'*, *title='Bar chart'*, *width=0.9*, *threshold=0*, *vertical=True*, *ax=None*, *rotation_xticks=None*, *grid='y'*, *figsize=14, 5*, *dpi=80*, *style='default'*, ***kwargs*)

Plot the values of a counter as a bar plot.

Values above the ratio are written as text on top of the bar.

Parameters

- **counter** (*collections.Counter or dictionary*) – Counter or dictionary to plot.
- **label_x** (*str, default 'x'*) – Label for x axis.
- **label_y** (*str, default 'y'*) – Label for y axis.

- **title** (*str, default 'Bar chart'*) – Title for the plot (axis level).
- **width** (*float, default 0.9*) – Width of the bar. If below 1.0, there will be space between them.
- **threshold** (*int, default = 0*) – Threshold above which the value is written on the plot as text. By default, all bar have their text.
- **vertical** (*bool, default True*) – By default, vertical bar. If set to False, will plot using *plt.barh* and inverse the labels.
- **ax** (*plt.axes, optional*) – Axes from matplotlib, if None, new figure and axes will be created.
- **loc** (*str or int, default 'best'*) – Location of the legend on the plot. Either the legend string or legend code are possible.
- **rotation_xticks** (*float, optional*) – Rotation of x ticks if any. Set to 90 to put them vertically.
- **grid** (*str or None, default 'y'*) – Axis where to activate the grid ('both', 'x', 'y'). To turn off, set to None.
- **figsize** (*Tuple[int, int], default (14, 5)*) – Size of the figure to plot.
- **dpi** (*int, default 80*) – Resolution of the figure.
- **style** (*str, default 'default'*) – Style to use for matplotlib.pyplot. The style is use only in this context and not applied globally.
- ****kwargs** – Additional keyword arguments to be passed to the *plt.plot* function from matplotlib.

Returns Axes returned by the *plt.subplots* function.

Return type plt.axes

Examples

```
>>> from collections import Counter
>>> counter = Counter({'red': 4, 'blue': 2})
>>> plot_counter(counter, title='MyTitle', rotation_xticks=90)
```

2.14 bff.plot.plot_history

bff.plot.**plot_history**(*history*, *metric=None*, *twinx=False*, *title='Model history'*, *axes=None*, *loc='best'*, *grid=None*, *figsize=12, 7*, *dpi=80*, *style='default'*, ***kwargs*)

Plot the history of the model trained using Keras.

Parameters

- **history** (*dict*) – Dictionary from the history object of the training.
- **metric** (*str, optional*) – Metric to plot. If no metric is provided, will only print the loss.
- **twinx** (*bool, default False*) – If metric and twinx, plot the loss and the metric on the same axis. Four colors will be used for the plot.
- **title** (*str, default 'Model history'*) – Main title for the plot (figure level).

- **axes** (*plt.axes, optional*) – Axes from matplotlib, if None, new figure and axes will be created. If metric is provided, need to have at least 2 axes.
- **loc** (*str or int, default 'best'*) – Location of the legend on the plot. Either the legend string or legend code are possible.
- **grid** (*str, optional*) – Axis where to activate the grid ('both', 'x', 'y'). To turn off, set to None.
- **figsize** (*Tuple[int, int], default (12, 7)*) – Size of the figure to plot.
- **dpi** (*int, default 80*) – Resolution of the figure.
- **style** (*str, default 'default'*) – Style to use for matplotlib.pyplot. The style is use only in this context and not applied globally.
- ****kwargs** – Additional keyword arguments to be passed to the *plt.plot* function from matplotlib.

Returns Axes object or array of Axes objects returned by the *plt.subplots* function.

Return type plt.axes

Examples

```
>>> history = model.fit(...)
>>> plot_history(history.history, metric='acc', title='MyTitle', linestyle=':')
```

2.15 bff.plot.plot_pca_explained_variance_ratio

bff.plot.**plot_pca_explained_variance_ratio**(*pca*, *label_x='Number of components'*, *label_y='Cumulative explained variance'*, *title='PCA explained variance ratio'*, *hline=None*, *ax=None*, *lim_x=None*, *lim_y=None*, *grid=None*, *figsize=10, 4*, *dpi=80*, *style='default'*, ***kwargs*)

Plot the explained variance ratio of PCA.

PCA must be already fitted.

Parameters

- **pca** (*sklearn.decomposition.PCA*) – PCA object to plot.
- **label_x** (*str, default 'Number of components'*) – Label for x axis.
- **label_y** (*str, default 'Cumulative explained variance'*) – Label for y axis.
- **title** (*str, default 'PCA explained variance ratio'*) – Title for the plot (axis level).
- **hline** (*float, optional*) – Horizontal line (darkorange) to draw on the plot (e.g. at 0.8 to see the number of components needed to keep 80% of the variance).
- **ax** (*plt.axes, optional*) – Axes from matplotlib, if None, new figure and axes will be created.
- **lim_x** (*Tuple[TNum, TNum], optional*) – Limit for the x axis.

- **lim_y** (*Tuple[TNum, TNum], optional*) – Limit for the y axis.
- **grid** (*str, optional*) – Axis where to activate the grid ('both', 'x', 'y'). To turn off, set to None.
- **figsize** (*Tuple[int, int], default (10, 4)*) – Size of the figure to plot.
- **dpi** (*int, default 80*) – Resolution of the figure.
- **style** (*str, default 'default'*) – Style to use for matplotlib.pyplot. The style is use only in this context and not applied globally.
- ****kwargs** – Additional keyword arguments to be passed to the *plt.plot* function from matplotlib.

Returns Axes returned by the *plt.subplots* function.

Return type plt.axes

Examples

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=20)
>>> pca.fit(data)
>>> plot_pca_explained_variance_ratio(pca)
```

2.16 bff.plot.plot_predictions

bff.plot.**plot_predictions**(*y_true*, *y_pred*, *x_true=None*, *x_pred=None*, *label_true='Actual'*, *label_pred='Predicted'*, *label_x='x'*, *label_y='y'*, *title='Model predictions'*, *ax=None*, *loc='best'*, *rotation_xticks=None*, *grid='y'*, *figsize=14, 5*, *dpi=80*, *style='default'*, ***kwargs*)

Plot the predictions of the model.

If a DataFrame is provided, it must only contain one column.

Parameters

- **y_true** (*np.array, pd.Series or pd.DataFrame*) – Actual values.
- **y_pred** (*np.array, pd.Series or pd.DataFrame*) – Predicted values by the model.
- **x_true** (*np.array, pd.Series, pd.DataFrame, optional*) – X coordinates for actual values. If not given, will be integer starting from 0.
- **x_pred** (*np.array, pd.Series, pd.DataFrame, optional*) – X coordinates for predicted values. If not given, will be integer starting from 0.
- **label_true** (*str, default 'Actual'*) – Label for the actual values.
- **label_pred** (*str, default 'Predicted'*) – Label for the predicted values.
- **label_x** (*str, default 'x'*) – Label for x axis.
- **label_y** (*str, default 'y'*) – Label for y axis.
- **title** (*str, default 'Model predictions'*) – Title for the plot (axis level).
- **ax** (*plt.axes, default None*) – Axes from matplotlib, if None, new figure and axes will be created.

- **loc** (*str or int, default 'best'*) – Location of the legend on the plot. Either the legend string or legend code are possible.
- **rotation_xticks** (*float, optional*) – Rotation of x ticks if any. Set to 90 to put them vertically.
- **grid** (*str or None, default 'y'*) – Axis where to activate the grid ('both', 'x', 'y'). To turn off, set to None.
- **figsize** (*Tuple[int, int], default (14, 5)*) – Size of the figure to plot.
- **dpi** (*int, default 80*) – Resolution of the figure.
- **style** (*str, default 'default'*) – Style to use for matplotlib.pyplot. The style is use only in this context and not applied globally.
- ****kwargs** – Additional keyword arguments to be passed to the *plt.plot* function from matplotlib.

Returns Axes returned by the *plt.subplots* function.

Return type plt.axes

Examples

```
>>> y_pred = model.predict(x_test, ...)
>>> plot_predictions(y_true, y_pred, title='MyTitle', linestyle=':')
```

2.17 bff.plot.plot_series

bff.plot.**plot_series**(*df*, *column*, *groupby=None*, *with_sem=False*, *with_peaks=False*, *with_missing_datetimes=False*, *distance_scale=0.04*, *label_x='Datetime'*, *label_y=None*, *title='Plot of series'*, *ax=None*, *color='#3F5D7D'*, *loc='best'*, *rotation_xticks=None*, *grid='y'*, *figsize=14, 6*, *dpi=80*, *style='default'*, ***kwargs*)

Plot time series with datetime with the given resample (*groupby*).

Parameters

- **df** (*pd.DataFrame*) – DataFrame to plot, with datetime as index.
- **column** (*str*) – Column of the DataFrame to display.
- **groupby** (*str, optional*) – Grouping for the resampling by mean of the data. For example, can resample from seconds ('S') to minutes ('T'). By default, no resampling is applied.
- **with_sem** (*bool, default False*) – Display the standard error of the mean (SEM) if set to true. Only possible if a resampling has been done.
- **with_peaks** (*bool, default False*) – Display the peaks of the serie if set to true.
- **with_missing_datetimes** (*bool, default False*) – Display the missing datetimes with vertical red lines. Only possible if a resampling has been done.
- **distance_scale** (*float, default 0.04*) – Scaling for the minimal distance between peaks. Only used if *with_peaks* is set to true.
- **label_x** (*str, default 'Datetime'*) – Label for x axis.

- **label_y** (*str, default None*) – Label for y axis. If None, will take the column name as label.
- **title** (*str, default 'Plot of series'*) – Title for the plot (axis level).
- **ax** (*plt.axes, optional*) – Axes from matplotlib, if None, new figure and ax will be created.
- **color** (*str, default '#3F5D7D'*) – Default color for the plot.
- **loc** (*str or int, default 'best'*) – Location of the legend on the plot. Either the legend string or legend code are possible.
- **rotation_xticks** (*float, optional*) – Rotation of x ticks if any. Set to 90 to put them vertically.
- **grid** (*str or None, default 'y'*) – Axis where to activate the grid ('both', 'x', 'y'). To turn off, set to None.
- **figsize** (*Tuple[int, int], default (14, 6)*) – Size of the figure to plot.
- **dpi** (*int, default 80*) – Resolution of the figure.
- **style** (*str, default 'default'*) – Style to use for matplotlib.pyplot. The style is use only in this context and not applied globally.
- ****kwargs** – Additional keyword arguments to be passed to the *plt.plot* function from matplotlib.

Returns Axes object or array of Axes objects returned by the *plt.subplots* function.

Return type plt.axes

Examples

```
>>> df_acceleration = fake.get_data_with_datetime_index(...)
>>> _, axes = plt.subplots(nrows=3, ncols=1, figsize=(14, 20), dpi=80)
>>> colors = {'x': 'steelblue', 'y': 'darkorange', 'z': 'darkgreen'}
>>> for i, acc in enumerate(['x', 'y', 'z']):
...     plot_series(df_acceleration, acc, groupby='T',
...                 ax=axes[i], color=colors[acc])
```

2.18 bff.plot.plot_true_vs_pred

bff.plot.**plot_true_vs_pred**(*y_true*, *y_pred*, *with_correlation=False*, *with_determination=True*, *with_histograms=False*, *with_identity=False*, *label_x='Ground truth'*, *label_y='Prediction'*, *title='Predicted vs Actual'*, *ax=None*, *lim_x=None*, *lim_y=None*, *grid='both'*, *figsize=14, 7*, *dpi=80*, *style='default'*, ***kwargs*)

Plot the ground truth against the predictions of the model.

If a DataFrame is provided, it must only contain one column.

Parameters

- **y_true** (*np.array or pd.DataFrame*) – Actual values.
- **y_pred** (*np.array or pd.DataFrame*) – Predicted values by the model.

- **with_correlation** (*bool, default False*) – If true, print correlation coefficient in the top left corner.
- **with_determination** (*bool, default True*) – If true, print the determination coefficient in the top left corner. If both *with_correlation* and *with_determination* are set to true, the correlation coefficient is printed.
- **with_histograms** (*bool, default False*) – If true, plot histograms of *y_true* and *y_pred* on the sides. Not possible if the *ax* is provided.
- **with_identity** (*bool, default False*) – If true, plot the identity line on the scatter plot.
- **label_x** (*str, default 'Ground truth'*) – Label for x axis.
- **label_y** (*str, default 'Prediction'*) – Label for y axis.
- **title** (*str, default 'Predicted vs Actual'*) – Title for the plot (axis level).
- **ax** (*plt.axes, optional*) – Axes from matplotlib, if None, new figure and axes will be created.
- **lim_x** (*Tuple[TNum, TNum], optional*) – Limit for the x axis. If None, automatically calculated according to the limits of the data, with an extra 5% for readability.
- **lim_y** (*Tuple[TNum, TNum], optional*) – Limit for the y axis. If None, automatically calculated according to the limits of the data, with an extra 5% for readability.
- **grid** (*str or None, default 'both'*) – Axis where to activate the grid ('both', 'x', 'y'). To turn off, set to None.
- **figsize** (*Tuple[int, int], default (14, 7)*) – Size of the figure to plot.
- **dpi** (*int, default 80*) – Resolution of the figure.
- **style** (*str, default 'default'*) – Style to use for matplotlib.pyplot. The style is use only in this context and not applied globally.
- ****kwargs** – Additional keyword arguments to be passed to the *plt.plot* function from matplotlib.

Returns Axes returned by the *plt.subplots* function. If *with_histograms*, return the three axes.

Return type plt.axes

Examples

```
>>> y_pred = model.predict(x_test, ...)
>>> plot_true_vs_pred(y_true, y_pred, title='MyTitle', linestyle=':')
```

2.19 bff.plot.set_thousands_separator

bff.plot.**set_thousands_separator**(*axes*, *which='both'*, *nb_decimals=1*)

Set thousands separator on the axes.

Parameters

- **axes** (*plt.axes*) – Axes from matplotlib, can be a single ax or an array of axes.

- **which** (*str, default 'both'*) – Which axis to format with the thousand separator ('both', 'x', 'y').
- **nb_decimals** (*int, default 1*) – Number of decimals to use for the number.

Returns Axis with the formatted axes.

Return type plt.axes

Examples

```
>>> fig, ax = plt.subplots(1, 1)
>>> ax.plot(...)
>>> set_thousands_separator(ax, which='x', nb_decimals=3)
```

2.20 bff.read_sql_by_chunks

bff.**read_sql_by_chunks**(*sql*, *cnxn*, *params=None*, *chunksize=8000000*, *column_types=None*, ***kwargs*)

Read SQL query by chunks into a DataFrame.

This function uses the *read_sql* from Pandas with the *chunksize* option.

The columns of the DataFrame are cast in order to be memory efficient and preserved when adding the several chunks of the iterator.

Parameters

- **sql** (*str*) – SQL query to be executed.
- **cnxn** (*SQLAlchemy connectable (engine/connection) or database string URI*) – Connection object representing a single connection to the database.
- **params** (*list or dict, default None*) – List of parameters to pass to execute method.
- **chunksize** (*int, default 8,000,000*) – Number of rows to include in each chunk.
- **column_types** (*dict, default None*) – Dictionary with the name of the column as key and the type as value. No cast is done if None.
- ****kwargs** – Additional keyword arguments to be passed to the *pd.read_sql* function.

Returns DataFrame with the concatenation of the chunks in the wanted type.

Return type pd.DataFrame

2.21 bff.size_2_square

bff.**size_2_square**(*n*)

Return the size of the side to create a square able to contain n elements.

This is mainly used to have the correct sizes of the sides when creating squared subplots.

Parameters **n** (*int*) – Number of elements that need to fit inside the square.

Returns Tuple of int with the size of each part of the square. Both element of the tuple are similar.

Return type Tuple of int

2.22 bff.sliding_window

bff.**sliding_window**(*sequence*, *window_size*, *step*)

Apply a sliding window over the sequence.

Each window is yielded. If there is a remainder, the remainder is yielded last, and will be smaller than the other windows.

Parameters

- **sequence** (*Sequence*) – Sequence to apply the sliding window on (can be str, list, numpy.array, etc.).
- **window_size** (*int*) – Size of the window to apply on the sequence.
- **step** (*int*) – Step for each sliding window.

Yields *Sequence* – Sequence generated.

Examples

```
>>> list(sliding_window('abcdef', 2, 1))
['ab', 'bc', 'cd', 'de', 'ef']
>>> list(sliding_window(np.array([1, 2, 3, 4, 5, 6]), 5, 5))
[array([1, 2, 3, 4, 5]), array([6])]
```

2.23 bff.value_2_list

bff.**value_2_list**(*value*)

Convert a single value into a list with a single value.

If the value is alredy a sequence, it is returned without modification. Type *np.ndarray* is not put inside another sequence.

Strings are not considered as a sequence in this scenario.

Parameters **value** (Any) – Value to convert to a sequence.

Returns Value put into a sequence.

Return type sequence

Examples

```
>>> value_2_list(42)
[42]
>>> value_2_list('Swiss')
['Swiss']
>>> value_2_list(['Swiss'])
['Swiss']
```

CHAPTER

THREE

FANCYCONFIG

Class to load the configuration file.

This class behaves like a dictionary that loads a configuration file in yaml format.

If the configuration file does not exist, creates it from template.

Examples

```
>>> config = FancyConfig()
>>> print(config)
{ 'database': { 'host': '127.0.0.1',
                'name': 'porgs',
                'port': 3306,
                'pwd': 'bacca',
                'user': 'Chew'},
  'env': 'prod',
  'imports': {'star_wars': ['ewok', 'bantha']}}
```

bff.FancyConfig.**__init__**(*self*, *path_config_to_load=PosixPath('/home/docs/.config/fancyconfig.yml')*, *default_config_path=PosixPath('/home/docs/checkouts/readthedocs.org/user_builds/bff/envs/latest/lib/python3.7/site-packages/bff-0.2.7+14.g26dad64-py3.7.egg/bff/config.yml')*)

Initialization of configuration.

If the folder to store the configuration does not exist, create it. If configuration file does not exist, copy it from default one.

Parameters

- **path_config_to_load** (*Path, default '~/.config/'*) – Directory to store the configuration file and load the configuration from.
- **default_config_path** (*Path, default 'config.yml' current directory.*) – Name of the configuration file.

INDEX