mlqa

Release 0.1.1

Dogan Askan

Dec 13, 2020

CONTENTS:

Welcome to mlqa's documentation. Get started with Introduction and then get an overview with the Quickstart. The rest of the documentation describes each component of MLQA in detail in the *API Reference* section.

CHAPTER

ONE

USER'S GUIDE

1.1 Introduction

1.1.1 What is it?

MLQA is a Python package that is created to help data scientists, analysts and developers to perform quality assurance (i.e. QA) on pandas dataframes and 1d arrays, especially for machine learning modeling data flows. It's designed to work with logging library to log and notify QA steps in a descriptive way. It includes stand alone functions (i.e. checkers) for different QA activities and DiffChecker class for integrated QA capabilities on data.

1.1.2 Installation

You can install MLQA with pip.

```
pip install mlqa
```

MLQA depends on Pandas and Numpy.

1.2 Quickstart

Here, you can see some quick examples on how to utilize the package. For more details, refer to API Reference.

1.2.1 DiffChecker Basics

DiffChecker is designed to perform QA on data flows for ML. You can easily save statistics from the origin data such as missing value rate, mean, min/max, percentile, outliers, etc., then to compare against the new data. This is especially important if you want to keep the prediction data under the same assumptions with the training data.

Below is a quick example on how it works, just initiate and save statistics from the input data.

```
>>> from mlqa.identifiers import DiffChecker
>>> import pandas as pd
>>> dc = DiffChecker()
>>> dc.fit(pd.DataFrame({'mean_col':[1, 2]*50, 'na_col':[None]*50+[1]*50}))
```

Then, you can check on new data if it's okay for given criteria. Below, you can see some data that is very similar in column *mean_col* but increased NA count in column *na_col*. The default threshold is 0.5 which means it should be okay if NA rate is 50% more than the origin data. NA rate is 50% in the origin data so up to 75% (i.e. 50*(1+0.5)) should be okay. NA rate is 70% in the new data and, as expected, the QA passes.

```
>>> dc.check(pd.DataFrame({'mean_col':[.99, 2.1]*50, 'na_col':[None]*70+[1]*30}))
True
```

If you think the threshold is too loose, you can adjust as you wish with set_threshold method. And, now the same returns *False* indicating the QA has failed.

```
>>> dc.set_threshold(0.1)
>>> dc.check(pd.DataFrame({'mean_col':[.99, 2.1]*50, 'na_col':[None]*70+[1]*30}))
False
```

1.2.2 DiffChecker Details

As default, DiffChecker is initialized with *qa_level='loose'*. Different values can also be given.

```
>>> from mlqa.identifiers import DiffChecker
>>> dc = DiffChecker()
>>> dc.threshold
0.5
>>> dc = DiffChecker(qa_level='mid')
>>> dc.threshold
0.2
>>> dc = DiffChecker(qa_level='strict')
>>> dc.threshold
0.1
```

To be more precise, you can set both threshold and stats individually.

```
>>> import pandas as pd
>>> import numpy as np
>>> dc = DiffChecker()
>>> dc.set_threshold(0.2)
>>> dc.set_stats(['mean', 'max', np.sum])
>>> dc.fit(pd.DataFrame({'col1':[1, 2, 3, 4], 'col2':[1]*4}))
>>> dc.check(pd.DataFrame({'col1':[1, 2, 3, 4], 'col2':[0]*4}))
False
>>> dc.check(pd.DataFrame({'col1':[1, 2.1, 3.2, 4.2], 'col2':[1.1]*4}))
True
```

You can even be more detailed in set_threshold.

```
>>> dc = DiffChecker()
>>> dc.set_stats(['mean', 'max'])
>>> dc.set_threshold(0.1) # to reset all thresholds
>>> print(dc.threshold)
0.1
>>> dc.fit(pd.DataFrame({'col1':[1, 2, 3, 4], 'col2':[0]*4}))
>>> dc.set_threshold({'col1':0.2, 'col2':0.1}) # to set in column level
>>> print(dc.threshold_df)
      col1  col2
mean   0.2   0.1
max    0.2   0.1
>>> dc.set_threshold({'col1':{'mean':0.1}}) # to set in column-stat level
>>> print(dc.threshold_df)
      col1  col2
mean   0.1   0.1
max    0.2   0.1
```

You can also pickle the object to be used later with to_pickle method.

```
>>> dc1 = DiffChecker()
>>> dc1.fit(pd.DataFrame({'col1':[1, 2, 3, 4], 'col2':[0]*4}))
>>> dc1.to_pickle(path='DiffChecker.pkl')
```

Then, to load the same object later.

```
>>> import pickle
>>> pkl_file = open('DiffChecker.pkl', 'rb')
>>> dc2 = pickle.load(pkl_file)
>>> pkl_file.close()
```

1.2.3 DiffChecker with Logging

If you enable logging functionality, you can get detailed description of what column failed for which stat and why. You can even log DiffChecker steps.

Just initiate the class with *logger='<your-logger-name>.log'* argument.

```
>>> from mlqa.identifiers import DiffChecker
>>> import pandas as pd
>>> dc = DiffChecker(logger='mylog.log')
>>> dc.fit(pd.DataFrame({'mean_col':[1, 2]*50, 'na_col':[None]*50+[1]*50}))
>>> dc.set_threshold(0.1)
>>> dc.check(pd.DataFrame({'mean_col':[1, 1.5]*50, 'na_col':[None]*70+[1]*30}))
False
```

If you open *mylog.log*, you'll see something like below.

```
WARNING|2020-05-31 15:56:48,146|mean value (i.e. 1.25) is not in the range of [1.35,␣
↪1.65] for mean_col
WARNING|2020-05-31 15:56:48,147|na_rate value (i.e. 0.7) is not in the range of [0.45,
↪ 0.55] for na_col
```

If you initiate the class with also *log_info=True* argument, then the other class steps (e.g. set_threshold, check) would be logged, too.

Note: Although DiffChecker is able to create a Logger object by just passing a file name (i.e. *logger='mylog.log'*), creating the Logger object externally then passing accordingly (i.e. *logger=<mylogger>*) is highly recommended.

1.2.4 Checkers with Logging

There are also checkers to provide other kind of QA functionalities such as outliers detection, pd.DataFrame comparison or some categorical value QA. You can use these individually or combining with DiffChecker's logger.

Let's say you initiated DiffChecker with some logger already.

```
>>> from mlqa.identifiers import DiffChecker
>>> dc = DiffChecker(logger='mylog.log')
```

Then, you can just pass *logger* attribute of the object when calling checkers. Here is an example of qa_outliers.

```
>>> import mlqa.checkers as ch
>>> import numpy as np
>>> import pandas as pd
>>> np.random.seed(123)
>>> df = pd.DataFrame({
...     'col1':np.random.normal(0, 0.1, 100),
...     'col2':np.random.normal(0, 1.0, 100)})
>>> ch.qa_outliers(df, std=0.5, logger=dc.logger)
False
```

This should log something like below.

```
WARNING|2020-05-31 17:54:13,426|70 outliers detected within inlier range (i.e. [-0.
↪053985309527773806, 0.059407124225845764]) for col1
WARNING|2020-05-31 17:54:13,428|53 outliers detected within inlier range (i.e. [-0.
↪5070058315486367, 0.46793470772834406]) for col2
```

You can also compare multiple datasets from the same population with qa_df_set.

```
>>> df1 = pd.DataFrame({'col1':[1, 2]*10, 'col2':[0, 4]*10})
>>> df2 = pd.DataFrame({'col1':[1, 9]*10, 'col2':[0, -4]*10})
>>> ch.qa_df_set([df1, df2], logger=dc.logger)
False
```

This should log something like below.

```
INFO|2020-05-31 18:09:47,581|df sets QA initiated with threshold 0.1
WARNING|2020-05-31 18:09:47,598|mean of col1 not passed. Values are 1.5 and 5.0
WARNING|2020-05-31 18:09:47,599|mean of col2 not passed. Values are 2.0 and -2.0
WARNING|2020-05-31 18:09:47,599|std of col1 not passed. Values are 0.51299 and 4.10391
WARNING|2020-05-31 18:09:47,599|min of col2 not passed. Values are 0.0 and -4.0
WARNING|2020-05-31 18:09:47,599|25% of col2 not passed. Values are 0.0 and -4.0
WARNING|2020-05-31 18:09:47,599|50% of col1 not passed. Values are 1.5 and 5.0
WARNING|2020-05-31 18:09:47,600|50% of col2 not passed. Values are 2.0 and -2.0
WARNING|2020-05-31 18:09:47,600|75% of col1 not passed. Values are 2.0 and 9.0
WARNING|2020-05-31 18:09:47,600|75% of col2 not passed. Values are 4.0 and 0.0
WARNING|2020-05-31 18:09:47,600|max of col1 not passed. Values are 2.0 and 9.0
WARNING|2020-05-31 18:09:47,600|max of col2 not passed. Values are 4.0 and 0.0
INFO|2020-05-31 18:09:47,600|df sets QA done with threshold 0.1
```

For categorical values, you can check its distribution on a numeric column with qa_category_distribution_on_value.

```
>>> df1 = pd.DataFrame({'Gender': ['Male', 'Male', 'Female', 'Female'],
...                     'Weight': [200, 250, 100, 125]})
>>> ch.qa_category_distribution_on_value(df1,
...                                      'Gender',
...                                      {'Male':.5, 'Female':.5},
...                                      'Weight',
...                                      logger=dc.logger)
False
```

This should log something like below.

```
WARNING|2020-05-31 18:21:20,019|Gender distribution looks wrong, check Weight for␣
↪Gender=Male. Expected=0.5, Actual=0.6666666666666666
WARNING|2020-05-31 18:21:20,019|Gender distribution looks wrong, check Weight for␣
↪Gender=Female. Expected=0.5, Actual=0.3333333333333333
```

Note: Although DiffChecker is able to create a Logger object by just passing a file name (i.e. *logger='mylog.log'*), creating the Logger object externally then passing accordingly (i.e. *logger=<mylogger>*) is highly recommended.

CHAPTER

TWO

API REFERENCE

2.1 mlqa.checkers

This module includes individual QA functions of mlqa.

checkers.**qa_outliers**(*data*, *std*, *logger=None*, *log_level=30*)

QA check for outliers as wrapper of *qa_outliers_1d*.

If there are values in the *data* outside of [mean-*std*, mean+`std`] range, returns *False*, otherwise *True*. If a pd.DataFrame given, then it checks each column individually.

Parameters

- **data** (*pd.DataFrame or iter*) – data to check
- **std** (*list or float*) – distance from mean for outliers, can be 2 elements iterable for different lower and upper bounds
- **logger** (*logging.Logger or None*) – Python logging object https://docs.python.org/3/library/logging.html#logging.Logger
- **log_level** (*int*) – https://docs.python.org/3/library/logging.html#logging-levels

Returns is QA passed or not

Return type bool

Example

Check for 1d:

```
>>> qa_outliers([1, 2, 3, 4], std=0.1)
False
>>> qa_outliers([1, 2, 3, 4], std=3)
True
```

Check for pd.DataFrame:

```
>>> import numpy as np
>>> import pandas as pd
>>> np.random.seed(123)
>>> df = pd.DataFrame({
...     'col1':np.random.normal(0, 0.1, 100),
...     'col2':np.random.normal(0, 1.0, 100)})
>>> qa_outliers(df, std=0.5)
False
```

See also:

qa_outliers_1d: same but only for 1d

checkers.**qa_outliers_1d**(*array*, *std*, *logger=None*, *log_level=30*, *name=None*)

QA check for outliers for 1d iterable.

If there are values in the *array* outside of [mean-*std*, mean+`std`] range, returns *False*, otherwise *True*.

Parameters

- **array** (*iter*) – 1d array to check
- **std** (*list or float*) – distance from mean for outliers, can be 2 elements iterable for different lower and upper bounds
- **logger** (*logging.Logger or None*) – Python logging object https://docs.python.org/3/library/logging.html#logging.Logger
- **log_level** (*int*) – https://docs.python.org/3/library/logging.html#logging-levels
- **name** (*str*) – optional array name for logger

Returns is QA passed or not

Return type bool

Example

```
>>> qa_outliers_1d([1, 2, 3, 4], std=0.1)
False
>>> qa_outliers_1d([1, 2, 3, 4], std=3)
True
```

See also:

qa_outliers: wrapper to be used in pd.DataFrame

checkers.**qa_missing_values**(*data*, *n=None*, *frac=None*, *threshold=0.1*, *limit=False, True*, *logger=None*, *log_level=30*)

QA check for missing values as wrapper of *qa_missing_values_1d* to also use in pd.DataFrame.

If array na count is within given condition, returns *True*, *False* otherwise. If a pd.DataFrame given, then it checks each column individually.

Parameters

- **data** (*pd.DataFrame or iter*) – data to check
- **n** (*int or None*) – expected missing value count
- **frac** (*float or None*) – expected missing value percentage
- **threshold** (*float*) – percentage threshold for upper or lower limit
- **limit** (*tuple*) – limit direction, which side of na limit to check
- **logger** (*logging.Logger or None*) – Python logging object https://docs.python.org/3/library/logging.html#logging.Logger
- **log_level** (*int*) – https://docs.python.org/3/library/logging.html#logging-levels

Returns is QA passed or not

Return type bool

See also:

qa_missing_values_1d: same but only for 1d

checkers.**qa_missing_values_1d**(*array*, *n=None*, *frac=None*, *threshold=0.1*, *limit=False, True*, *logger=None*, *log_level=30*, *name=None*)

QA check for missing values of 1D array.

If array na count is within given condition, returns *True*, *False* otherwise.

Parameters

- **array** (*iter*) – 1d array to check
- **n** (*int or None*) – expected missing value count
- **frac** (*float or None*) – expected missing value percentage
- **threshold** (*float*) – percentage threshold for upper or lower limit
- **limit** (*tuple*) – limit direction, which side of na limit to check
- **logger** (*logging.Logger or None*) – Python logging object https://docs.python.org/3/library/logging.html#logging.Logger
- **log_level** (*int*) – https://docs.python.org/3/library/logging.html#logging-levels
- **name** (*str*) – optional array name for logger

Returns is QA passed or not

Return type bool

Example

```
>>> qa_missing_values_1d([1, 2, None, None], n=1)
False
>>> qa_missing_values_1d([1, 2, None, None], n=2)
True
>>> qa_missing_values_1d([1, None, None, None], n=2, threshold=0.5)
True
```

See also:

qa_missing_values: wrapper to be used in pd.DataFrame

checkers.**qa_df_set**(*dfs*, *threshold=0.1*, *ignore_min=None*, *ignore_max=None*, *stats_to_exclude=None*, *columns_to_exclude=None*, *error_columns=None*, *logger=None*, *name=None*)

Wrapper for *qa_df_pair()* to apply 2 length subsequences of *dfs*.

QA datasets' statistics by utilizing describe() method of pd.DataFrame. Ignores non-numeric columns.

Parameters

- **dfs** (*iter*) – set of pd.DataFrame
- **threshold** (*float*) – percentage threshold for absolute percentage error between statistics
- **ignore_min** (*None or float*) – ignore stats less or equal than this to handle division errors or extreme values
- **ignore_max** (*None or float*) – ignore stats greater or equal than this to handle extreme values

- **stats_to_exclude** (*None or list*) – statistics to exclude as list of strings, e.g. ['count', 'mean', 'std', 'min', '25%', '50%', '75%', 'max']
- **columns_to_exclude** (*None or list*) – columns to exclude as list of strings
- **error_columns** (*None or list*) – error columns for error, if given, then test results for non error columns would be ignored. Only these columns are logged with level 40.
- **logger** (*logging.Logger or None*) – Python logging object https://docs.python.org/3/library/logging.html#logging.Logger
- **name** (*str*) – optional array name for logger

Returns is QA passed or not

Return type bool

Example

```
>>> df1 = pd.DataFrame({'col1':[1, 2]*10, 'col2':[0, 4]*10})
>>> df2 = pd.DataFrame({'col1':[1, 9]*10, 'col2':[0, -4]*10})
>>> qa_df_set([df1, df2])
False
```

See also:

qa_df_pair: same but only for 2 pd.DataFrame

checkers.**qa_df_pair**(*df1*, *df2*, *threshold=0.1*, *ignore_min=None*, *ignore_max=None*, *stats_to_exclude=None*, *columns_to_exclude=None*, *error_columns=None*, *logger=None*, *name=None*)

QA two datasets' statistics by utilizing describe() method of pd.DataFrame. Ignores non-numeric columns.

Parameters

- **df1** (*pd.DataFrame*) – test dataframe
- **df2** (*pd.DataFrame*) – test dataframe
- **threshold** (*float*) – percentage threshold for absolute percentage error between statistics
- **ignore_min** (*None or float*) – ignore stats less or equal than this to handle division errors or extreme values
- **ignore_max** (*None or float*) – ignore stats greater or equal than this to handle extreme values
- **stats_to_exclude** (*None or list*) – statistics to exclude as list of strings, e.g. ['count', 'mean', 'std', 'min', '25%', '50%', '75%', 'max']
- **columns_to_exclude** (*None or list*) – columns to exclude as list of strings
- **error_columns** (*None or list*) – error columns for error, if given, then test results for non error columns would be ignored. Only these columns are logged with level 40.
- **logger** (*logging.Logger or None*) – Python logging object https://docs.python.org/3/library/logging.html#logging.Logger
- **name** (*str*) – optional array name for logger

Returns is QA passed or not

Return type bool

See also:

qa_df_set: wrapper to use more than 2 pd.DataFrame

checkers.**qa_preds**(*preds*, *warn_range*, *error_range=None*, *logger=None*, *name=None*)

Wrapper for *qa_array_statistics* for stats *min* and *max* only.

It should be mainly used to also log QA steps and prediction statistics. Use *qa_array_statistics* for detailed QA on prediction array.

Parameters

- **preds** – array, shape (n_samples, 1)
- **warn_range** (*iter*) – 2 elements iterable, e.g. [min, max] to warn
- **error_range** (*iter or None*) – 2 elements iterable or None, e.g. [min, max] for error, should involve warn_range. If not None, QA result by *warn_range* is ignored.
- **logger** (*logging.Logger or None*) – Python logging object https://docs.python.org/3/library/logging.html#logging.Logger If *None* given, no practical use of this function. Use *qa_array_statistics* instead.
- **name** (*str*) – optional array name for logger

Returns is QA passed or not

Return type bool

Example

```
>>> qa_preds([1, 2, 3, 4], warn_range=[1.5, 5])
False
>>> qa_preds([1, 2, 3, 4], warn_range=[1.5, 5], error_range=[0, 5.5])
True
```

checkers.**qa_category_distribution_on_value**(*df*, *category_column_name*, *distribution*, *value_column_name*, *threshold=0.1*, *logger=None*, *log_level=30*)

QA check for the distribution of category-value pairs in a pd.DataFrame.

Gender

Parameters

- **df** (*pd.DataFrame*) – input data
- **category_column_name** (*str*) – column name for the category, (e.g. 'Gender')
- **distribution** (*dict*) – expected value distribution of the category (e.g. {'Male':.05, 'Female':.14, 'Undefined':.81})
- **value_column_name** (*str*) – numeric column name to check distribution, (e.g. 'Weight')
- **threshold** (*float*) – percentage threshold for absolute percentage error
- **logger** (*logging.Logger or None*) – Python logging object https://docs.python.org/3/library/logging.html#logging.Logger
- **log_level** (*int*) – https://docs.python.org/3/library/logging.html#logging-levels

Returns is QA passed or not

Return type bool

Example

```
>>> df1 = pd.DataFrame({'Gender': ['Male', 'Male', 'Female', 'Female'],
...                     'Weight': [200, 250, 100, 125]})
>>> qa_category_distribution_on_value(df1,
...                                   'Gender',
...                                   {'Male':.66, 'Female':.33},
...                                   'Weight',
...                                   0.1)
True
>>> qa_category_distribution_on_value(df1,
...                                   'Gender',
...                                   {'Male':.5, 'Female':.5},
...                                   'Weight',
...                                   0.1)
False
>>> qa_category_distribution_on_value(df1,
...                                   'Gender',
...                                   {'Male':.5, 'Female':.5},
...                                   'Weight',
...                                   0.5)
True
```

checkers.**qa_preds_by_metric**(*y_true*, *y_pred*, *metric*, *check_range*, *logger=None*, *log_level=30*)

QA check for model's predictions by selected metric (e.g. R2, AUC).

Parameters

- **y_true** (*iter*) – shape (n_samples, 1)
- **y_pred** (*iter*) – shape (n_samples, 1)
- **metric** (*func*) – sklearn like metric function. https://scikit-learn.org/stable/modules/classes.html#module-sklearn.metrics
- **check_range** (*list*) – list of 2 float, i.e. [*lower_limit*, *upper_limit*], either of elements can be *None* if no limit is set for that direction.
- **logger** (*logging.Logger or None*) – Python logging object https://docs.python.org/3/library/logging.html#logging.Logger
- **log_level** (*int*) – https://docs.python.org/3/library/logging.html#logging-levels

Returns is QA passed or not

Return type bool

Example

```
>>> y_true = pd.Series([1, 2, 3, 4])
>>> y_pred = pd.Series([1, 3, 3, 3])
>>> mae = lambda x, y: abs(x - y).mean()
>>> qa_preds_by_metric(y_true, y_pred, mae, [None, 0.6])
True
>>> qa_preds_by_metric(y_true, y_pred, mae, [0.4, 0.6])
True
>>> qa_preds_by_metric(y_true, y_pred, mae, [0.6, None])
False
```

checkers.**qa_array_statistics**(*array*, *stats*, *logger=None*, *log_level=30*, *name=None*)

QA check for 1D array statistics such as mean, count.

Parameters

- **array** (*iter*) – shape (n_samples, 1)
- **stats** (*dict*) – stats to qa (e.g. {'mean':[0.1, 0.99], 'count':[100, None]}(Options for keys are ['mean', 'min', 'max', 'sum', 'count', 'std'] or function such as *np.mean*.
- **logger** (*logging.Logger or None*) – Python logging object https://docs.python.org/3/library/logging.html#logging.Logger
- **log_level** (*int*) – https://docs.python.org/3/library/logging.html#logging-levels
- **name** (*str*) – optional array name for logger

Returns is QA passed or not

Return type bool

Example

```
>>> qa_array_statistics([1, 2, 3, 4], {'count':[3, 5], 'min':[None, 1.5]})
True
>>> qa_array_statistics([1, 2, 3, 4], {'count':[3, 5], 'max':[None, 1.5]})
False
```

checkers.**is_value_in_range**(*value*, *check_range*, *logger=None*, *log_level=None*, *log_msg=None*)

Checks if a *value* is in given *check_range*.

Parameters

- **value** (*float*) – value to check
- **check_range** (*list*) – acceptable lower and upper bounds for *value*
- **logger** (*logging.Logger or None*) – Python logging object https://docs.python.org/3/library/logging.html#logging.Logger
- **log_level** (*int*) – https://docs.python.org/3/library/logging.html#logging-levels
- **log_msg** – str or None, custom log message for *logger*

Returns is QA passed or not

Return type bool

Example

```
>>> is_value_in_range(5.0, [3, 10])
True
>>> is_value_in_range(5.0, [None, 1])
False
```

checkers.**na_rate**(*array*)

Aggregate function to calculate na rate in pd.Series.

Parameters **array** (*pd.Series*) – input array

Returns na count / array length

Return type float

Example

```
>>> na_rate(pd.Series([1, None, 2, 3]))
0.25
```

2.2 mlqa.identifiers

This module is for DiffChecker class.

class identifiers.**DiffChecker**(*qa_level='loose'*, *logger=None*, *qa_log_level=None*, *log_info=False*)

Bases: object

Integrated QA performer on pd.DataFrame with logging functionality.

It only works in numerical columns.

Parameters

- **qa_level** (*str*) – quick set for QA level, can be one of ['loose', 'mid', 'strict']
- **logger** (*str or logging.Logger*) – 'print' for print only, every other str creates a file for logging. using external logging.Logger object is highly recommended, i.e. logger=<mylogger>.
- **qa_log_level** (*int*) – qa message logging level
- **log_info** (*bool*) – *True* if method calls or arguments also need to be logged

Notes

Although DiffChecker is able to create a Logger object by just passing a file name (i.e. *logger='mylog.log'*), creating the Logger object externally then passing accordingly (i.e. *logger=<mylogger>*) is highly recommended.

Example

Basic usage:

```
>>> dc = DiffChecker()
>>> dc.fit(pd.DataFrame({'mean_col':[1, 2]*50, 'na_col':[None]*50+[1]*50}))
>>> dc.check(pd.DataFrame({'mean_col':[.99, 2.1]*50, 'na_col':[None]*70+[1]*30}))
True
>>> dc.set_threshold(0.1)
>>> dc.check(pd.DataFrame({'mean_col':[.99, 2.1]*50, 'na_col':[None]*70+[1]*30}))
False
```

Quick set for *qa_level*:

```
>>> dc = DiffChecker()
>>> dc.threshold
0.5
>>> dc = DiffChecker(qa_level='mid')
>>> dc.threshold
0.2
>>> dc = DiffChecker(qa_level='strict')
>>> dc.threshold
0.1
```

Logger can also be initiated:

```
>>> dc = DiffChecker(logger='mylog.log')
>>> dc.fit(pd.DataFrame({'mean_col':[1, 2]*50, 'na_col':[None]*50+[1]*50}))
>>> dc.set_threshold(0.1)
>>> dc.check(pd.DataFrame({'mean_col':[1, 1.5]*50, 'na_col':[None]*70+[1]*30}))
False
```

stats = []

threshold = 0.0

threshold_df = Empty DataFrame Columns: [] Index: []

df_fit_stats = Empty DataFrame Columns: [] Index: []

set_stats (*funcs*)
Sets statistic functions list to check by.

Parameters **funcs** (*list*) – list of functions and/or function names, e.g. [np.sum, 'mean']

See also:

add_stat: just to add one

add_stat (*func*)
Appends a statistic function into the existing list (i.e. *stats*).

Parameters **func** (*func*) – function name (e.g. np.sum or 'mean')

See also:

set_stats: to reset all

set_threshold (*threshold*)
Sets threshold for statistic-column pairs.

Parameters **threshold** (*float or dict*) – can be used to set for all or column statistic pairs.

Example

```
>>> dc = DiffChecker()
>>> dc.set_stats(['mean', 'max'])
>>> dc.set_threshold(0.1) # to reset all thresholds
>>> print(dc.threshold)
0.1
>>> dc.fit(pd.DataFrame({'col1':[1, 2, 3, 4], 'col2':[0]*4}))
>>> dc.set_threshold({'col1':0.2, 'col2':0.1}) # to set in column level
>>> print(dc.threshold_df)
      col1  col2
mean   0.2   0.1
max    0.2   0.1
>>> dc.set_threshold({'col1':{'mean':0.3}}) # to set in column-stat level
>>> print(dc.threshold_df)
      col1  col2
mean   0.3   0.1
max    0.2   0.1
```

fit (*df*)

Fits given *df*.

Based on given *df* and *stats* attribute, this method constructs *df_fit_stats* attribute to store column statistics. This is later to be used by *check* method. Only works in numerical columns.

Parameters **df** (*pd.DataFrame*) – data to be fit

Example

```
>>> dc = DiffChecker()
>>> dc.set_stats(['mean', 'max'])
>>> dc.fit(pd.DataFrame({'col1':[1, 2, 3, 4], 'col2':[0]*4}))
>>> print(dc.df_fit_stats)
      col1  col2
mean   2.5   0.0
max    4.0   0.0
```

check (*df_to_check*, *columns=None*, *columns_to_exclude=None*)

Checks given *df_to_check* based on fitted *df* stats.

For each column stat pairs, it checks if stat is in given threshold by utilizing qa_array_statistics. If any stat qa fails, returns *False*, *True otherwise*.

Parameters

- **df_to_check** (*pd.DataFrame*) – data to check
- **columns** (*None or list*) – if given, only these columns will be considered for qa
- **columns_to_exclude** (*None or list*) – columns to exclude from qa

Returns is QA passed or not

Return type bool

Example

```
>>> dc = DiffChecker()
>>> dc.set_threshold(0.2)
>>> dc.set_stats(['mean', 'max', np.sum])
>>> dc.fit(pd.DataFrame({'col1':[1, 2, 3, 4], 'col2':[1]*4}))
>>> dc.check(pd.DataFrame({'col1':[1, 2, 3, 4], 'col2':[0]*4}))
False
>>> dc.check(pd.DataFrame({'col1':[1, 2.1, 3.2, 4.2], 'col2':[1.1]*4}))
True
```

to_pickle(*path='DiffChecker.pkl'*)

Pickle (serialize) object to a file.

Parameters **path** (*str*) – file path where the pickled object will be stored

Example

To save a **.pkl* file:

```
>>> dc1 = DiffChecker()
>>> dc1.fit(pd.DataFrame({'col1':[1, 2, 3, 4], 'col2':[0]*4}))
>>> dc1.to_pickle(path='DiffChecker.pkl')
```

To load the same object later:

```
>>> import pickle
>>> pkl_file = open('DiffChecker.pkl', 'rb')
>>> dc2 = pickle.load(pkl_file)
>>> pkl_file.close()
>>> os.remove('DiffChecker.pkl')
```

_method_init_logger(*args*, *exclude=['self']*)

Logs method initiation with given arguments.

Parameters

- **args** (*dict*) – local arguments, i.e. *locals()*
- **exclude** (*list*) – arguments to exclude, e.g. *self*

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX