traval

Release 0.3.1

Artesia

Nov 08, 2022

CONTENTS:

Python package for applying automatic error detection algorithms to timeseries.

This module is set up to provide tools for applying any error detection algorithm to any timeseries. The module consists of three main components:

- *Detector*: a data management object for storing timeseries and error detection results.
- *RuleSet*: the RuleSet object is a highly flexible object for defining error detection algorithms based on (user-defined) functions.
- *SeriesComparison**: objects for comparing timeseries. These objects include plots for visualizing the comparisons.

The general workflow consists of the following steps:

1. Define error detection algorithm(s).
2. Load data, i.e. raw timeseries data and optionally timeseries representing the "truth" to see how well the algorithms perform.
3. Initialize Detector objects and apply algorithms to timeseries.
4. Store and analyze the results.

For more detailed information and examples, please refer to the notebooks in the examples directory.

CHAPTER

ONE

GETTING STARTED

1.1 Installation

To install traval, a working version of Python 3.7 or 3.8 has to be installed on your computer. We recommend using the Anaconda Distribution with Python 3.7 as it includes most of the python package dependencies and the Jupyter Notebook software to run the notebooks. However, you are free to install any Python distribution you want.

To install traval, use:

```
pip install traval
```

To install in development mode, clone the repository, then type the following from the module root directory:

```
pip install -e .
```

1.2 Usage

The basic usage of the module is described below. To start using the module, import the package:

```
import traval
```

The first step is generally to define an error detection algorithm. This is done with the *RuleSet* object:

```
ruleset = traval.RuleSet("my_first_algorithm")
```

Add a detection rule (using a general rule from the library contained within the module). In this case the rule states any value above 10.0 is suspect:

```
ruleset.add_rule("rule1",
                 traval.rulelib.rule_ufunc_threshold ,
                 apply_to=0,
                 kwargs={"ufunc": (np.greater,), "threshold": 10.0}
                 )
```

Take a look at the ruleset by just typing *ruleset*:

```
ruleset
```

```
RuleSet: 'my_first_algorithm'
   step: name            apply_to
      1: rule1                  0
```

Next define a Detector object. This object is designed to store a timeseries and the intermediate and final results after applying an error detection algorithm. Initialize the Detector object with some timeseries. In this example we assume there is a timeseries called *raw_series*:

```
detect = traval.Detector(raw_series)
```

Apply our first algorithm to the timeseries.

```
detect.apply_ruleset(ruleset)
```

By default, the result of each step in the algorithm is compared to the original series and stored in the *detect.comparisons* attribute. Take a look at the comparison between the raw data and the result of the error detection algorithm.

Since we only defined one step, step 1 represents the final result.

```
cp = detect.comparisons[1]  # result of step 1 = final result
```

The *SeriesComparison** objects contain methods to visualize the comparison, or summarize the number of observations in each category:

```
cp.plots.plot_series_comparison()  # plot a comparison
cp.summary  # series containing number of observations in each category
```

For more detailed explanation and more complex examples, see the notebook(s) in the examples directory.

CHAPTER

TWO

EXAMPLES

The following notebooks contain examples showcasing traval.

The first example shows how to apply the tools contained in traval to detect errors in a single timeseries. The second example shows how the same can be done for a full dataset with lots of timeseries.

2.1 Example 1: Applying an automatic error detection algorithm to a timeseries

Created by Davíd Brakenhoff, Artesia, May 2020

This notebook contains a simple example how to set up an automatic error detection algorithm based on a few simple rules and applies those rules to a groundwater timeseries.

First import the requisite packages:

[1]:
```
import os
import numpy as np
import pandas as pd

import traval
from traval import rulelib as rlib
```

2.1.1 Data

Load the data. The following information is available: - The raw groundwater levels (with all the measurement errors still in the timeseries). - The manually validated timeseries (this is what we consider as the 'truth'). Of course the manual validation isn't always perfect, but since it's all we have to compare with we're using this as our 'truth'. We assume our manual validation is good, so hopefully our error detection algorithm yields similar results to the manual validation. - The sensor level in meters relative to NAP, for checking whether the sensor is above the groundwater level. - The elevation of the top of the piezometer, for checking whether the groundwater level is above this level.

[2]:
```
datadir = "../data/"
raw = pd.read_csv(os.path.join(datadir, "raw_series.csv"), index_col=[0], parse_
↪dates=True).squeeze()
truth = pd.read_csv(os.path.join(datadir, "manual_validation_series.csv"), index_col=[0],
↪ parse_dates=True).squeeze()
truth.name = "manual validation"
sensor_level = pd.read_csv(os.path.join(datadir, "sensor_level_nap.csv"), index_col=[0],␣
↪parse_dates=True).squeeze()
```

(continues on next page)

(continued from previous page)

```
top_piezometer = pd.read_csv(os.path.join(datadir, "top_piezometer_level.csv"), index_
↪col=[0], parse_dates=True).squeeze()
```

2.1.2 The error detection algorithm and the `RuleSet` object

The detection algorithm consists of 3 checks and an extra step to combine the results of those three checks: 1. Check for spikes (when groundwater level suddenly changes and returns to its original level one timestep later). 2. Check if sensor is above groundwater level. 3. Check if groundwater level is above top of piezometer (if the piezometer is closed this is possible, but in this case we know it isn't). 4. Combine the results of checks 1 to 3 to yield the final result.

The error detection algorithm is entered using the `RuleSet` object. This is an object that can hold any number of error detection rules and apply those to a timeseries. With the `RuleSet.add_rule()` method, rules can be added to the algorithm. Ading a rule requires the following input: - **name**: name of the rule (user-specified) - **func**: the function to apply to the timeseries - **apply_to**: an integer indicating to which timeseries the rule should be applied. The original timeseries is 0, the outcome from step 1 is 1, etc. - **kwargs**: a dictionary containing any other arguments required by the functions that are passed.

The final rule we add doesn't check for errors but combines the results from the previous three steps to create one final timeseries that includes the outcome from each of the preceding rules. In this case `apply_to` is a tuple of ints referencing the results that should be combined. In this case it says to combine the results from steps 1, 2 and 3.

[4]:
```
# initialize RuleSet object
rset = traval.RuleSet(name="basic")

# add rules
rset.add_rule("spikes", rlib.rule_spike_detection, apply_to=0,
              kwargs={"threshold": 0.15, "spike_tol": 0.15, "max_gap": "7D"})
rset.add_rule("dry", rlib.rule_ufunc_threshold, apply_to=0,
              kwargs={"ufunc": (np.less,), "threshold": sensor_level, "offset": 0.025})
rset.add_rule("hardmax", rlib.rule_ufunc_threshold, apply_to=0,
              kwargs={"ufunc": (np.greater,), "threshold": top_piezometer})
rset.add_rule("combine", rlib.rule_combine_nan_or, apply_to=(1, 2, 3))
```

The view of the object shows which rules have been added:

[5]:
```
# view object
rset
```

[5]:
```
RuleSet: 'basic'
  step: name            apply_to
     1: spikes                 0
     2: dry                    0
     3: hardmax                0
     4: combine        (1, 2, 3)
```

The `RuleSet` object can be stored as pickle file or as JSON. - `to_pickle`: This option has full support for custom functions and is the most flexible and is therefore recommended. The file is not human readable however. - `to_json`: Storing as a JSON file has the advantage of creating a human readable file, but it only supports default functions from `traval.rulelib`. So custom functions will not be preserved when saving in this format.

[6]:
```
rset.to_pickle('test.pkl')
rset2 = traval.RuleSet.from_pickle("test.pkl")
```

```
RuleSet written to file: 'test.pkl'
```

[7]:
```
rset.to_json("test.json")
rset3 = traval.RuleSet.from_json("test.json")
```

```
RuleSet written to file: 'test.json'
```

```
/home/david/Github/traval/traval/ruleset.py:436: UserWarning: Custom functions will not␣
↪be preserved when storing RuleSet as JSON file!
  warnings.warn(msg)
```

Delete the two files we just created.

[8]:
```
for f in ["test.json", "test.pkl"]:
    os.remove(f)
```

2.1.3 The Detector object

The `Detector` object provides tools for storing a timeseries, applying an algorithm built with the `RuleSet` object and processing the outcomes. We initialize the objet with the raw data timeseries in which we want to find the erroneous measurements. Optionally we can add a “truth” series to compare the outcome of our error detection algorithm to.

[9]:
```
detector = traval.Detector(raw, truth=truth)
detector
```

[9]:
```
Detector: <DEUR033_G>
```

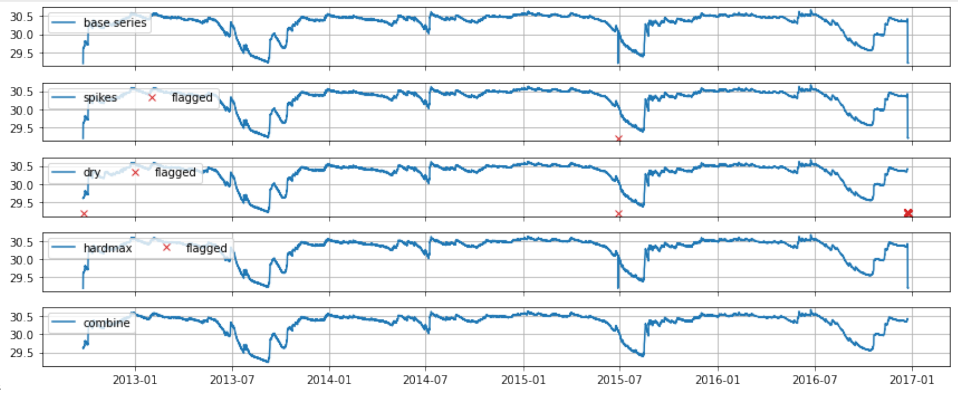
Apply our custom algorithm. The `compare=True` creates `SeriesComparison` objects for the result of each step in the algorithm. These objects compare the result of a error detection step with the original timeseries or the ‘truth’ if available. The object also includes methods to plot the comparison results.

[10]:
```
detector.apply_ruleset(rset, compare=True)
```

Plot an overview of the results. This creates one plot per rule and highlights which points were marked as suspect based on that rule.

[11]:
```
axes = detector.plot_overview()
```

nbsphinx-code-borderwhite



The results with the flagged values as NaNs in the original series can be obtained with `detector.get_results_dataframe()`.

```
[12]: results = detector.get_results_dataframe()
      results.head()
```

```
[12]:                      base series   spikes      dry  hardmax  combine
      index
      2012-09-24 15:00:00      29.1959  29.1959      NaN  29.1959      NaN
      2012-09-24 16:00:00      29.6104  29.6104  29.6104  29.6104  29.6104
      2012-09-24 17:00:00      29.6082  29.6082  29.6082  29.6082  29.6082
      2012-09-24 18:00:00      29.6170  29.6170  29.6170  29.6170  29.6170
      2012-09-24 19:00:00      29.6158  29.6158  29.6158  29.6158  29.6158
```

The corrections are only stored for timestamps where corrections were deemed relevant.

```
[13]: corrections = detector.get_corrections_dataframe()
      corrections
```

```
[13]:                      spikes  dry  hardmax  combine
      2012-09-24 15:00:00     0.0  NaN      0.0      0.0
      2015-06-27 14:30:41     NaN  NaN      0.0      0.0
      2016-12-23 10:00:00     0.0  NaN      0.0      0.0
      2016-12-23 11:00:00     0.0  NaN      0.0      0.0
      2016-12-23 12:00:00     0.0  NaN      0.0      0.0
      2016-12-23 13:00:00     0.0  NaN      0.0      0.0
      2016-12-23 14:00:00     0.0  NaN      0.0      0.0
      2016-12-23 15:00:00     0.0  NaN      0.0      0.0
      2016-12-23 16:00:00     0.0  NaN      0.0      0.0
      2016-12-23 17:00:00     0.0  NaN      0.0      0.0
      2016-12-23 18:00:00     0.0  NaN      0.0      0.0
      2016-12-23 19:00:00     0.0  NaN      0.0      0.0
      2016-12-23 20:00:00     0.0  NaN      0.0      0.0
      2016-12-23 21:00:00     0.0  NaN      0.0      0.0
      2016-12-23 22:00:00     0.0  NaN      0.0      0.0
      2016-12-23 23:00:00     0.0  NaN      0.0      0.0
      2016-12-24 00:00:00     0.0  NaN      0.0      0.0
      2016-12-24 01:00:00     0.0  NaN      0.0      0.0
      2016-12-24 02:00:00     0.0  NaN      0.0      0.0
      2016-12-24 03:00:00     0.0  NaN      0.0      0.0
      2016-12-24 04:00:00     0.0  NaN      0.0      0.0
      2016-12-24 05:00:00     0.0  NaN      0.0      0.0
      2016-12-24 06:00:00     0.0  NaN      0.0      0.0
      2016-12-24 07:00:00     0.0  NaN      0.0      0.0
      2016-12-24 08:00:00     0.0  NaN      0.0      0.0
```

The comparison objects are stored as a dictionary under `detector.comparisons`:

```
[14]: detect = detector
```

```
[15]: detect.comparisons
```

```
[15]: {1: <traval.ts_comparison.SeriesComparisonRelative at 0x7f670b7b3890>,
       2: <traval.ts_comparison.SeriesComparisonRelative at 0x7f66e2fab7d0>,
       3: <traval.ts_comparison.SeriesComparisonRelative at 0x7f66e2fab4d0>,
       4: <traval.ts_comparison.SeriesComparisonRelative at 0x7f66e2fb73d0>}
```

Let’s take a look at the comparison of the final result with manual validation, relative to the raw data.

[16]:
```
comp = detector.comparisons[4]
```

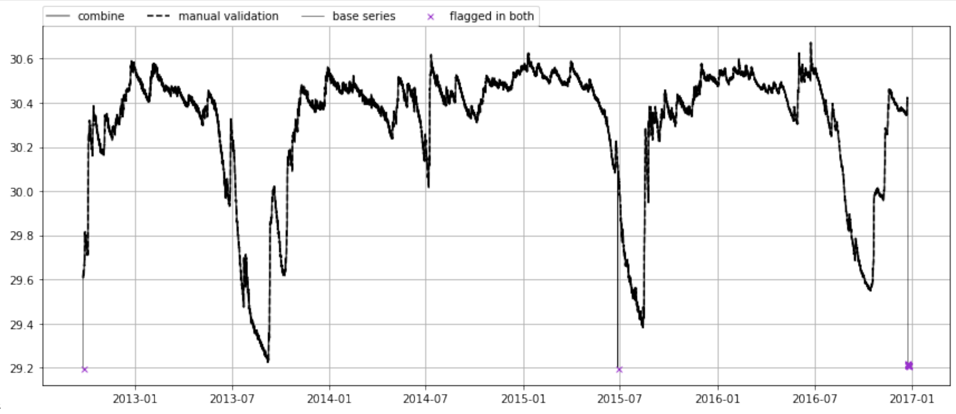
The `SeriesComparisonRelative` object contains a summary of the different categories measurements can fall into, e.g. “kept in both” means the observations was deemed valid in both the detection result and the “truth” timeseries.

[17]:
```
comp.summary_base_comparison
```

[17]:
```
kept_in_both           37212
flagged_in_s1              0
flagged_in_s2              0
flagged_in_both           25
in_all_nan                 0
introduced_in_s1           0
introduced_in_s2           0
introduced_in_both         0
Name: N_obs, dtype: int64
```

Plot the comparison overview. In this comparison the manual validation and the result of our error detection algorithm are compared to each other relative to the raw data. In this case we can see that all the points that were marked as suspect in the manual validation were also identified by our algorithm.

[18]:
```
ax = comp.plots.plot_relative_comparison(mark_different=False, mark_identical=False)
leg = ax.legend(loc=(0, 1), ncol=4)
```



nbsphinx-code-borderwhite

2.1.4 Binary Classification statistics and the confusion matrix

All methods for calculating statistics related to binary classification are available through the `comp.bc` attribute.

[19]:
```
comp.bc?
```

```
Type:           BinaryClassifier
String form:    <traval.binary_classifier.BinaryClassifier object at 0x7f66e2fb9850>
File:           ~/Github/traval/traval/binary_classifier.py
Docstring:      Class for calculating binary classification statistics.
Init docstring:
Initialize class for calculating binary classification statistics.

Parameters
----------
tp : int
```

(continues on next page)

(continued from previous page)

```
    number of True Positives (TP)
fp : int
    number of False Positives (FP)
tn : int
    number of True Negatives (TN)
fn : int
    number of False Negatives (FN)
```

The comp.bc.confustion_matrix method is used to calculate the confusion matrix. The number of times the algorithm is correct (as compared to the “truth”) is given on the diagonal. The number of times the algorithm is wrong is shown outside the diagonal.

[20]:
```
comp.bc.confusion_matrix?
```

```
Signature: comp.bc.confusion_matrix(as_array=False)
Docstring:
Calculate confusion matrix.

Confusion matrix shows the performance of the algorithm given a
certain truth. An abstract example of the confusion matrix:

                |     Algorithm     |
                |-------------------|
                |  error  | correct |
------|---------|---------|---------|
      |  error  |   TP    |   FN    |
Truth |---------|---------|---------|
      | correct |   FP    |   TN    |
------|---------|---------|---------|

where:
- TP: True Positives  = errors correctly detected by algorithm
- TN: True Negatives  = correct values correctly not flagged by algorithm
- FP: False Positives = correct values marked as errors by algorithm
- FN: False Negatives = errors not detected by algorithm

Parameters
----------
as_array : bool, optional
    return data as array instead of DataFrame, by default False

Returns
-------
data : pd.DataFrame or np.array
    confusion matrix
File:      ~/Github/traval/traval/binary_classifier.py
Type:      method
```

[21]:
```
cmat = comp.bc.confusion_matrix()
cmat
```

[21]:
```
                Algorithm
                    error correct
"Truth" error          25       0
        correct         0   37212
```

Binary classification statistics are also available: - **sensitivity** or **true positive rate**: number of positives identified relative to total number of positives. This says something about the avoidance of false negatives. - **specificity** or **true negative rate**: number of negatives identified relative to total number of negatives. This says something about the avoidance of false positives. - **false positive rate**: (= 1 - specificity) - **false negative rate**: (= 1 - sensitivity)

More explanation is available in de docstrings of these properties:

[22]:
```
comp.bc.specificity?
```

```
Type:         property
String form: <property object at 0x7f66e3d4e2f0>
Docstring:
Specificity or True Negative Rate.

Statistic describing ratio of true negatives identified,
which also says something about the avoidance of false positives.

    Specificity = TN / (TN + FP)

where
- TN : True Negatives
- FP : False Positives
```

[23]:
```
print(f"- True Positive Rate  = {comp.bc.true_positive_rate} =       Sensitivity = {comp.
↪bc.sensitivity}")
print(f"- True Negative Rate  = {comp.bc.true_negative_rate} =       Specificity = {comp.
↪bc.specificity}")
print(f"- False Positive Rate = {comp.bc.false_positive_rate} = (1 - Specificity) = {1 -␣
↪comp.bc.specificity}")
print(f"- False Negative Rate = {comp.bc.false_negative_rate} = (1 - Sensitivity) = {1 -␣
↪comp.bc.sensitivity}")
```

```
- True Positive Rate  = 1.0 =       Sensitivity = 1.0
- True Negative Rate  = 1.0 =       Specificity = 1.0
- False Positive Rate = 0.0 = (1 - Specificity) = 0.0
- False Negative Rate = 0.0 = (1 - Sensitivity) = 0.0
```

Matthews Correlation Coefficient (MCC)

The MCC is seen as a balanced measure for classification that also works when classes have very different sizes. It is essentially a measure between -1 and 1 that attempts to summarize the confusion matrix in one value. A coefficient of +1 represents a perfect prediction, 0 no better than random prediction and 1 indicates total disagreement between prediction and observation. Though there is no perfect way of summarizing the confusion matrix in one measure, the MCC is seen as one of the best of such measures (Source: Wikipedia).

[25]:
```
comp.bc.mcc
```

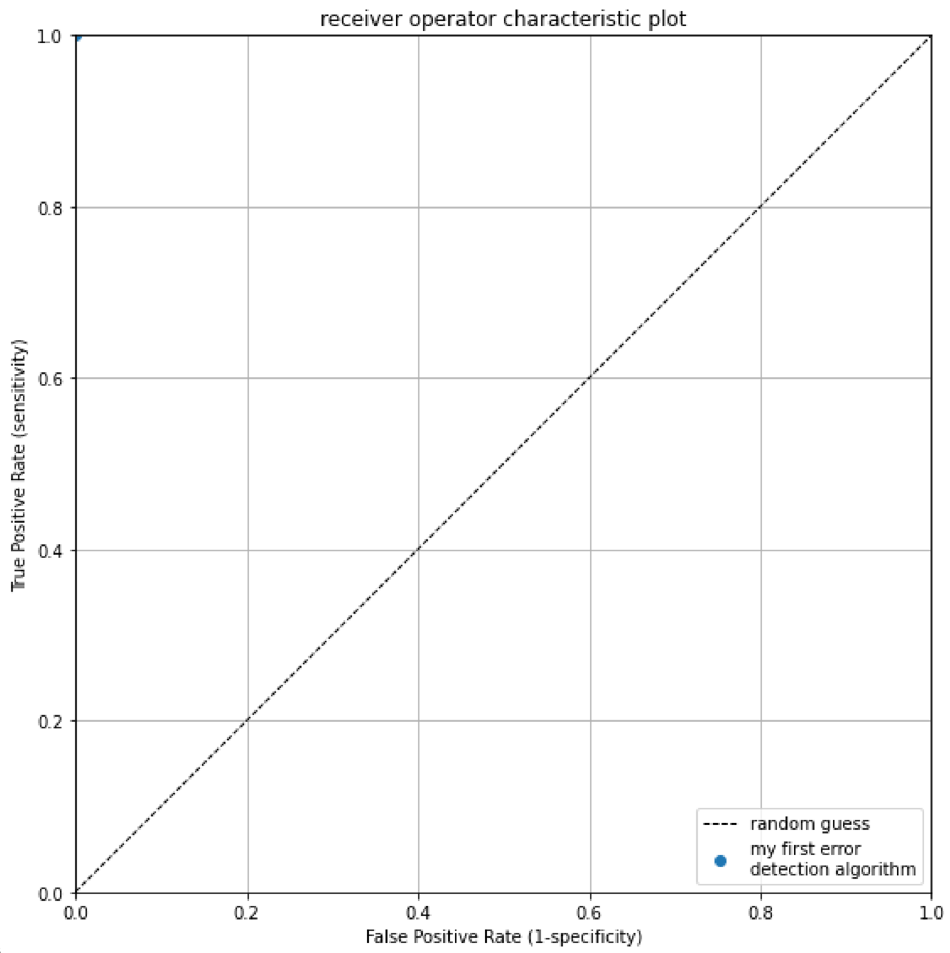
[25]:
```
1.0
```

2.1.5 Receiver Operator Characteristic plot

Finally, we can plot the so-called "receiver operator characteristic curve", which shows the performance of our algorithm relative to the line corresponding to a random guess. The plot shows the False Positive Rate versus the True Positive Rate. The higher the True Positive Rate the better our algorithm is able to identify errors. The lower the False Positive Rate, the lower the number of False Positives it yields. The perfect score is therefore the top left corner of the plot.

Our algorithm scores perfectly in this example.

```
[29]: label = "my first error \ndetection algorithm"
      ax = traval.plots.roc_plot(comp.bc.true_positive_rate,
                                 comp.bc.false_positive_rate,
                                 labels=label)
```



nbsphinx-code-borderwhite

2.2 Example 2: Applying an error detection algorithm to a full dataset

Created by Davíd Brakenhoff, Artesia, May 2020

Use Aa en Maas divers dataset consisting of 484 piezometers to test new traval module. - Requires pystore - Requires hydropandas - Requires data in pystore format as prepared by scripts in the `traval_data` module

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import pystore
from tqdm.notebook import tqdm
import hydropandas as hpd
import traval
from traval import rulelib as rlib
```

2.2.1 Load data

[2]:
```
pystore_path = "/home/david/Github/traval-data/extracted_data/pystore"
pystore_name = "aaenmaas"
```

[3]:
```
items = ["GW.meting.ruw"]
raw_obs = hpd.ObsCollection.from_pystore(pystore_name,
                                        pystore_path,
                                        collection_names=None,
                                        item_names=items,
                                        nameby="collection",
                                        read_series=True,
                                        verbose=False,
                                        progressbar=True)

items = ["GW.meting.totaalcorrectie"]
val_obs = hpd.ObsCollection.from_pystore(pystore_name,
                                        pystore_path,
                                        collection_names=None,
                                        item_names=items,
                                        nameby="collection",
                                        read_series=True,
                                        verbose=False,
                                        progressbar=True)
```

```
100%|| 484/484 [00:11<00:00, 42.06it/s]
100%|| 484/484 [00:17<00:00, 27.88it/s]
```

2.2.2 Helper functions for obtaining additional timeseries

[4]:
```python
pystore.set_path(pystore_path)
store = pystore.store(pystore_name)

def get_mph_series(name):
    """Get piezometer height from pystore.
    """
    coll = store.collection(name)
    meetpuntNAP_df = coll.item('Meetpunt.hoogte').to_pandas()
    return meetpuntNAP_df.value


def get_threshold_series(name):
    """Get level below which sensor is dry from pystore."""
    coll = store.collection(name)
    inhangdiepte_df = coll.item('Inhang.diepte').to_pandas()
    inhangdiepte = inhangdiepte_df.value.iloc[0]
    meetpuntNAP_df = coll.item('Meetpunt.hoogte').to_pandas()
    meetpuntNAP = meetpuntNAP_df.value.iloc[0]
    threshold_series = meetpuntNAP_df.value - inhangdiepte_df.value
    return threshold_series
```

2.2.3 Define error detection algorithm

BASIC method from TRAVAL study.

[6]:
```python
# initialize RuleSet object
rset = traval.RuleSet(name="basic")

# add rules
rset.add_rule("spikes", rlib.rule_spike_detection, apply_to=0, kwargs={"threshold": 0.15,
↪ "spike_tol": 0.15, "max_gap": "7D"})
rset.add_rule("dry", rlib.rule_ufunc_threshold, apply_to=0, kwargs={"ufunc": (np.less,),
↪"threshold": get_threshold_series, "offset": 0.05})
rset.add_rule("hardmax", rlib.rule_ufunc_threshold, apply_to=0, kwargs={"ufunc": (np.
↪greater,), "threshold": get_mph_series})
rset.add_rule("combine", rlib.rule_combine_nan_or, apply_to=(1, 2, 3))

# view object
rset
```

[6]:
```
RuleSet: 'basic'
  step: name            apply_to
     1: spikes                 0
     2: dry                    0
     3: hardmax                0
     4: combine        (1, 2, 3)
```

2.2.4 Error detection

Do some pre-processing on timeseries prior to running error detection: - Create synthetic raw timeseries (remove unlabeled adjustments to timeseries) - Ensure all labeled errors are set to np.nan in truth series (this means they are counted as erroneous observations). - Recategorize some comments.

[7]:
```
dlist = {}

for name in tqdm(raw_obs.index.intersection(val_obs.index)):

    # get raw data
    raw = raw_obs.loc[name, "obs"]["value"]
    raw.name = name

    # get truth
    truth = val_obs.loc[name, "obs"].loc[:, ["value", "comment"]]

    # set all commented data to np.NaN
    truth.loc[truth.comment != "", "value"] = np.nan
    truth.loc[truth.comment == "vorst", "comment"] = "onbetrouwbare meting"

    # rename columns
    truth.columns = ["manual validation", "comment"]

    # create synthetic raw (only keeps values for labeled changes)
    synth_raw = traval.ts_utils.create_synthetic_raw_timeseries(raw, truth["manual␣
↪validation"], truth["comment"])
    synth_raw.name = name

    # create detector object and apply algorithm
    detector = traval.Detector(synth_raw, truth=truth)
    detector.apply_ruleset(rset, compare=[-1])

    # store object
    dlist[name] = detector
```

```
  0%|          | 0/481 [00:00<?, ?it/s]
```

2.2.5 Calculate statistics

For full dataset and for individual timeseries

[8]:
```
fpr = []
tpr = []

# initialize empty BinaryClassifier
bc_sum = traval.BinaryClassifier(0, 0, 0, 0)

for k, dct in dlist.items():

    # get TPR and FPR
    itpr = dct.comparisons[4].bc.true_positive_rate
    ifpr = dct.comparisons[4].bc.false_positive_rate
```

(continues on next page)

(continued from previous page)

```
    tpr.append(itpr)
    fpr.append(ifpr)

    # add binary classification result,
    # the '+' is overloaded to allow adding of the two
    bc_sum = bc_sum + dct.comparisons[4].bc
```

Calculate confusion matrix for full dataset.

[9]:
```
bc_sum.confusion_matrix?
```

```
Signature: bc_sum.confusion_matrix(as_array=False)
Docstring:
Calculate confusion matrix.

Confusion matrix shows the performance of the algorithm given a
certain truth. An abstract example of the confusion matrix:

                |     Algorithm     |
                |-------------------|
                |  error  | correct |
------|---------|---------|---------|
      |  error  |   TP    |   FN    |
Truth |---------|---------|---------|
      | correct |   FP    |   TN    |
------|---------|---------|---------|

where:
- TP: True Positives  = errors correctly detected by algorithm
- TN: True Negatives  = correct values correctly not flagged by algorithm
- FP: False Positives = correct values marked as errors by algorithm
- FN: False Negatives = errors not detected by algorithm

Parameters
----------
as_array : bool, optional
    return data as array instead of DataFrame, by default False

Returns
-------
data : pd.DataFrame or np.array
    confusion matrix
File:      ~/Github/traval/traval/binary_classifier.py
Type:      method
```

The confusion matrix summarizes the performance of the algorithm. The diagonal entries show in how many cases the algorithm was correct in identifying errors or correct measurements, as compared to the “truth”: the manually validated timeseries. The off-diagonal entries show the cases where the algorithm was incorrect, either by erroneously classifying a good measurement as an error, or not identifying an erroneous measurement.

[10]:
```
bc_sum.confusion_matrix()
```

[10]:
```
                 Algorithm
                     error   correct
"Truth" error       442444    411828
        correct     319029  24651161
```

ROC-plot

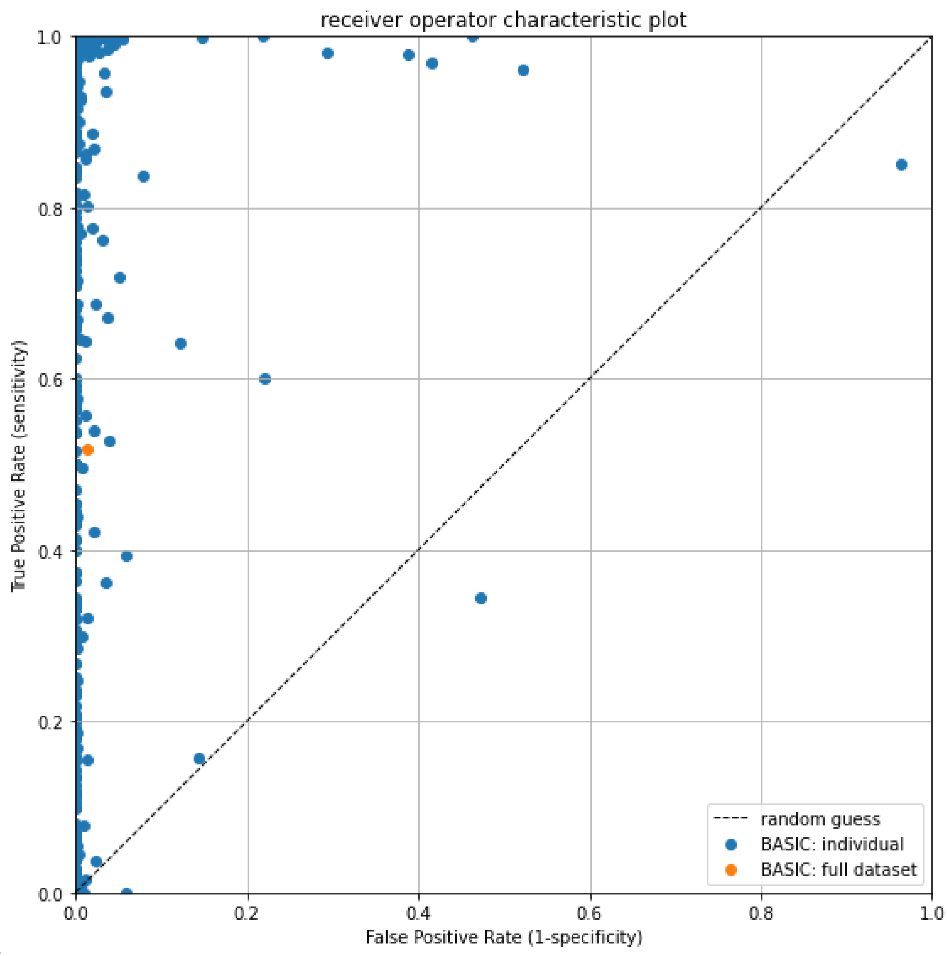
The receiver-operator characteristic plot shows the performance of the algorithm relative to the line representing a random guess. - The x-axis shows the False Positive Rate (FPR), which says something about how many false positives occur on a scale from 0.0 to 1.0. E.g. at an FPR of 25% it means that 75% of true negatives are correctly not flagged as suspect, which in turn, means that 25% are marked as suspect and represent False Positives. - The y-axis shows the True Positive Rate or Sensitivity which says something about how many erroroneous observations are correctly identified on a scale from 0.0 to 1.0.

The perfect score is the top-left corner, which indicates that 100% of values are correctly classified, and there are 0% False Positives. Values on the black diagonal dotted line are no better than a random guess. Values below the black dotted line indicate that the inverse of classification of the algorithm is better at classifying measurements than its current implementation.

Each blue dot represents the algorithm score for a single timeseries. The orange dot is the performance of the algorithm for the whole dataset. The plot shows that the False Positive rate is generally quite low, but that there is a significant spread in the Sensitivity. On average, about 50% of erroneous measurements are correctly identified.

[12]:
```
ax = traval.plots.roc_plot([np.array(tpr)],
                           [np.array(fpr)],
                           labels=["BASIC: individual"])
traval.plots.roc_plot(bc_sum.true_positive_rate,
                      bc_sum.false_positive_rate,
                      labels="BASIC: full dataset",
                      ax=ax,
                      plot_diagonal=False)
```

[12]:
```
<AxesSubplot:title={'center':'receiver operator characteristic plot'}, xlabel='False␣
↪Positive Rate (1-specificity)', ylabel='True Positive Rate (sensitivity)'>
```



nbsphinx-code-borderwhite

If we check the True Positive Rate we see that it is indeed about 50%:

[13]:
```
bc_sum.true_positive_rate
```

[13]:
```
0.517919351213665
```

Matthews Correlation Coefficient (MCC) for the full dataset.

The MCC is seen as a balanced measure for classification that also works when classes have very different sizes. It is essentially a measure between -1 and 1 that attempts to summarize the confusion matrix in one value. A coefficient of +1 represents a perfect prediction, 0 no better than random prediction and 1 indicates total disagreement between prediction and observation. Though there is no perfect way of summarizing the confusion matrix in one measure, the MCC is seen as one of the best of such measures (Source: Wikipedia).

[14]:
```
bc_sum.mcc
```

[14]:
```
0.5340471787404938
```

Get statistics per comment

[15]:
```
summ = None
for d in tqdm(dlist.values()):
    if summ is None:
        summ = d.comparisons[4].compare_to_base_by_comment()
```

(continues on next page)

(continued from previous page)

```
    else:
        summ = summ.add(d.comparisons[4].compare_to_base_by_comment(), fill_value=0.0)
```

```
  0%|          | 0/481 [00:00<?, ?it/s]
```

[16]:
```
summ = summ.dropna(how="all", axis=0)
summ
```

[16]:
```
                              drift logger     droog  foute meting   inhang  \
kept_in_both     24651161.0           NaN       NaN           NaN      NaN
flagged_in_s1      319029.0           NaN       NaN           NaN      NaN
flagged_in_s2           NaN      299331.0   24234.0       21559.0   9111.0
flagged_in_both         NaN       31367.0  303194.0       23006.0  34652.0

                   lucht  onbetrouwbare meting    piek
kept_in_both         NaN                   NaN     NaN
flagged_in_s1        NaN                   NaN     NaN
flagged_in_s2     3361.0               48017.0  6215.0
flagged_in_both  15343.0               24959.0  9923.0
```

[17]:
```
ratio = summ / summ.sum()
ratio
```

[17]:
```
                            drift logger     droog  foute meting   inhang  \
kept_in_both     0.987224           NaN       NaN           NaN      NaN
flagged_in_s1    0.012776           NaN       NaN           NaN      NaN
flagged_in_s2         NaN      0.905149  0.074013      0.483765  0.20819
flagged_in_both       NaN      0.094851  0.925987      0.516235  0.79181

                    lucht  onbetrouwbare meting      piek
kept_in_both          NaN                   NaN       NaN
flagged_in_s1         NaN                   NaN       NaN
flagged_in_s2    0.179694              0.657983  0.385116
flagged_in_both  0.820306              0.342017  0.614884
```

Plot barchart showing error detection rate per comment.

[18]:
```
lbls = list((ratio.columns.str.replace(" ", "\n") +
             summ.sum().apply(lambda s: "\n({0:>6.2%})".format(s/bc_sum.n_obs))).values)

# create figure
fig, ax = plt.subplots(1, 1, figsize=(12, 6))

bar = ax.bar(range(ratio.columns.size), ratio.loc["flagged_in_both"].values,
       align="center", width=0.6, edgecolor="k", label=rset.name)

ax.set_xticklabels(lbls)
ax.set_yticks(np.arange(0, 1.1, 0.1))
ax.set_ylabel("Ratio (-)")
ax.grid(b=True, axis="y")
ax.legend(loc="upper right")
ax.set_title("Ratio data correctly identified as suspect "
             "(manual validation as truth) "
```

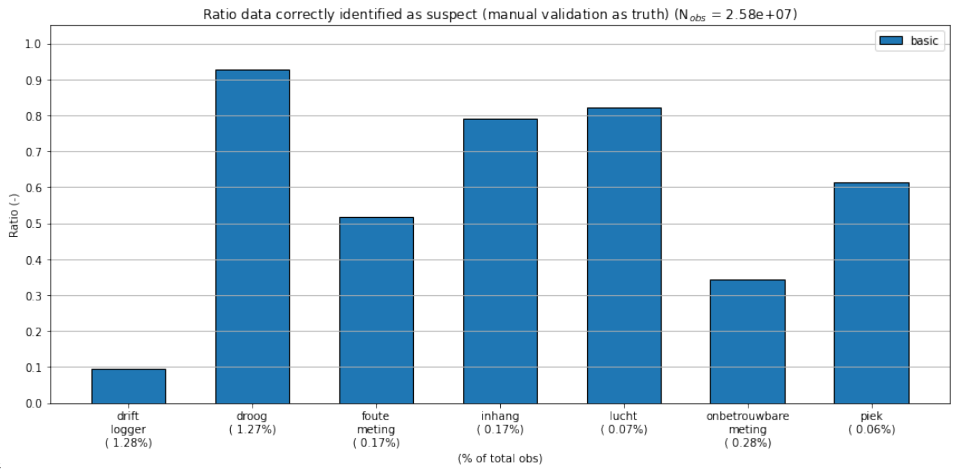
(continues on next page)

(continued from previous page)

```
             "(N$_{{obs}}$ = {0:5.2e})".format(summ.sum().sum()))
plt.xticks(rotation=0, ha="center")
ax.set_ylim(0, 1.05)
ax.set_xlabel("(% of total obs)")

fig.tight_layout()
```

```
/home/david/anaconda3/lib/python3.7/site-packages/ipykernel_launcher.py:10: UserWarning:␣
↪FixedFormatter should only be used together with FixedLocator
  # Remove the CWD from sys.path while we load stuff.
```



nbsphinx-code-borderwhite

```
[ ]:
```

CHAPTER

THREE

API DOCUMENTATION

3.1 Detector

class traval.detector.**Detector**(*series*, *truth=None*)

Detector object for applying error detection algorithms to timeseries.

The Detector is used to apply error detection algorithms to a timeseries and optionally contains a 'truth' series, to which the error detection result can be compared. An example of a 'truth' series is a manually validated timeseries. Custom error detection algorithms can be defined using the RuleSet object.

Parameters

- **series** (*pd.Series or pd.DataFrame*) – timeseries to check
- **truth** (*pd.Series or pd.DataFrame, optional*) – series that represents the 'truth', i.e. a benchmark to which the error detection result can be compared, by default None

Examples

Given a timeseries 'series' and some ruleset 'rset':

```
>>> d = Detector(series)
>>> d.apply_ruleset(rset)
>>> d.plot_overview()
```

See also:

traval.RuleSet
object for defining detection algorithms

static _validate_input_series(*series*)

Internal method for checking type and dtype of series.

Parameters
series (*object*) – timeseries to check, must be pd.Series or pd.DataFrame. Datatype of series or first column of DataFrame must be float.

Raises
TypeError – if series or dtype of series does not comply

apply_ruleset(*ruleset*, *compare=True*)

Apply RuleSet to series.

Parameters

- **ruleset** (*traval.RuleSet*) – RuleSet object containing detection rules
- **compare** (*bool or list of int, optional*) – if True, compare all results to original series and store in dictionary under comparisons attribute, default is True. If False, do not store comparisons. If list of int, store only those step numbers as comparisons. Note: value of -1 refers to last step for convenience.

See also:

traval.RuleSet
object for defining detection algorithms

confusion_matrix(*steps=None*, *truth=None*)

Calculate confusion matrix stats for detection rules.

Note: the calculated statistics per rule contain overlapping counts, i.e. multiple rules can mark the same observatin as suspect.

Parameters

- **steps** (*int, list of int or None, optional*) – steps for which to calculate confusion matrix statistics, by default None which uses all steps.
- **truth** (*pd.Series or pd.DataFrame, optional*) – series representing the "truth", i.e. a benchmark to which the resulting series is compared. By default None, which uses the stored truth series. Argument is included so a different truth can be passed.

Returns
df – dataframe containing confusion matrix data, i.e. counts of true positives, false positives, true negatives and false negatives.

Return type
pd.DataFrame

get_comment_series(*steps=None*)

get_corrections_comparison(*truth=None*)

get_corrections_dataframe()

Get DataFrame containing corrections.

Returns
df – DataFrame containing corrections. NaN means value is flagged as suspicious, 0.0 means no correction.

Return type
pandas.DataFrame

get_final_result()

Get final timeseries with flagged values set to NaN.

Returns
series – Timeseries produced by final step in RuleSet with flagged values set to NaN.

Return type
pandas.Series

get_indices(*category*, *step*, *truth=None*) `get_results_dataframe()`

Get results as DataFrame.

Returns
df – results with flagged values set to NaN per applied rule.

Return type
pandas.DataFrame

`get_series`(*step*, *category=None*)

`plot_overview`(*mark_suspects=True*, ***kwargs*)

Plot timeseries with flagged values per applied rule.

Parameters
mark_suspects (*bool, optional*) – mark suspect values with red X, by default True

Returns
ax – axes objects

Return type
list of matplotlib.pyplot.Axes

`reset()`

Reset Detector object.

`set_truth`(*truth*)

Set 'truth' series.

Used for comparison with detection result.

Parameters
truth (*pd.Series or pd.DataFrame*) – Series or DataFrame containing the "truth", i.e. a benchmark to compare the detection result to.

`stats_per_comment`(*step=None*, *truth=None*)

`uniqueness`(*truth=None*)

Calculate unique contribution per rule to stats.

Note: the calculated statistics per rule contain an undercount, i.e. when multiple rules mark the same observatin as suspect it is not contained in this result.

Parameters

- **steps** (*int, list of int or None, optional*) – steps for which to calculate confusion matrix statistics, by default None which uses all steps.
- **truth** (*pd.Series or pd.DataFrame, optional*) – series representing the "truth", i.e. a benchmark to which the resulting series is compared. By default None, which uses the stored truth series. Argument is included so a different truth can be passed.

Returns
df – dataframe containing confusion matrix data, i.e. unique counts of true positives, false positives, true negatives and false negatives.

Return type
pd.DataFrame

3.2 RuleSet

class `traval.ruleset.`**`RuleSet`**(*name=None*)

Create RuleSet object for storing detection rules.

The RuleSet object stores detection rules and other relevant information in a dictionary. The order in which rules are carried out, the functions that parse the timeseries, the extra arguments required by those functions are all stored together.

The detection functions must take a series as the first argument, and return a series with corrections based on the detection rule. In the corrections series invalid values are set to np.nan, and adjustments are defined with a float. No change is defined as 0. Extra keyword arguments for the function can be passed through a kwargs dictionary. These kwargs are also allowed to contain functions. These functions must return some value based on the name of the series.

Parameters

name (*str, optional*) – name of the RuleSet, by default None

Examples

Given two detection functions 'foo' and 'bar':

```
>>> rset = RuleSet(name="foobar")
>>> rset.add_rule("foo", foo, apply_to=0)  # add rule 1
>>> rset.add_rule("bar", bar, apply_to=1, kwargs={"n": 2})  # add rule 2
>>> print(rset)  # print overview of rules
```

`add_rule`(*name*, *func*, *apply_to=None*, *kwargs=None*)

Add rule to RuleSet.

Parameters

- **name** (*str*) – name of the rule
- **func** (*callable*) – function that takes series as input and returns a correction series.
- **apply_to** (*int or tuple of ints, optional*) – series to apply the rule to, by default None, which defaults to the original series. E.g. 0 is the original series, 1 is the result of step 1, etc. If a tuple of ints is passed, the results of those steps are collected and passed to func.
- **kwargs** (*dict, optional*) – dictionary of additional keyword arguments for func, by default None. Additional arguments can be functions as well, in which case they must return some value based on the name of the series to which the RuleSet will be applied.

`del_rule`(*name*)

Delete rule from RuleSet.

Parameters

name (*str*) – name of the rule to delete

classmethod **`from_json`**(*fname*)

Load RuleSet object from JSON file.

Attempts to load functions in the RuleSet by searching for the function name in traval.rulelib. If the function cannot be found, only the name of the function is preserved. This means a RuleSet with custom functions will not be fully functional when loaded from a JSON file.

Parameters
`fname` (*str*) – filename or path to file

Returns
RuleSet object

Return type
RuleSet

See also:

to_json
store RuleSet as JSON file (does not support custom functions)

to_pickle
store RuleSet as pickle (supports custom functions)

from_pickle
load RuleSet from pickle file

`classmethod from_pickle`(*fname*)

Load RuleSet object form pickle file.

Parameters
`fname` (*str*) – filename or path to file

Returns
RuleSet object, including custom functions and parameters

Return type
RuleSet

See also:

to_pickle
store RuleSet as pickle (supports custom functions)

to_json
store RuleSet as json file (does not support custom functions)

from_json
load RuleSet from json file

`to_dataframe()`

Convert RuleSet to pandas.DataFrame.

Returns
rdf – DataFrame containing all the information from the RuleSet

Return type
pandas.DataFrame

`to_json`(*fname*, *verbose=True*)

Write RuleSet to disk as json file.

Note that it is not possible to write custom functions to a JSON file. When writing the JSON only the name of the function is stored. When loading a JSON file, the function name is used to search within *traval.rulelib*. If the function can be found, it loads that function. A RuleSet making use of functions in the default rulelib.

Parameters

- **fname** (*str*) – filename or path to file
- **verbose** (*bool, optional*) – prints message when operation complete, default is True

See also:

from_json
load RuleSet from json file

to_pickle
store RuleSet as pickle (supports custom functions)

from_pickle
load RuleSet from pickle file

`to_pickle`(*fname*, *verbose=True*)

Write RuleSet to disk as pickle.

Parameters

- **fname** (*str*) – filename or path of file
- **verbose** (*bool, optional*) – prints message when operation complete, default is True

See also:

from_pickle
load RuleSet from pickle file

to_json
store RuleSet as json file (does not support custom functions)

from_json
load RuleSet from json file

`update_rule`(*name*, *func*, *apply_to=None*, *kwargs=None*)

Update rule in RuleSet.

Parameters

- **name** (*str*) – name of the rule
- **func** (*callable*) – function that takes series as input and returns a correction series.
- **apply_to** (*int or tuple of ints, optional*) – series to apply the rule to, by default None, which defaults to the original series. E.g. 0 is the original series, 1 is the result of step 1, etc. If a tuple of ints is passed, the results of those steps are collected and passed to func.
- **kwargs** (*dict, optional*) – dictionary of additional keyword arguments for func, by default None. Additional arguments can be functions as well, in which case they must return some value based on the name of the series to which the RuleSet will be applied.

class traval.ruleset.**RuleSetEncoder**(**, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, sort_keys=False, indent=None, separators=None, default=None*)

`default`(*o*)

Implement this method in a subclass such that it returns a serializable object for `o`, or calls the base implementation (to raise a `TypeError`).

For example, to support arbitrary iterators, you could implement default like this:

```
def default(self, o):
    try:
        iterable = iter(o)
    except TypeError:
        pass
    else:
        return list(iterable)
    # Let the base class default method raise the TypeError
    return JSONEncoder.default(self, o)
```

3.3 Rule Library

`traval.rulelib.`**`rule_combine_nan_and`**`(*args)`

Combination rule, combine NaN values for any number of timeseries.

Used for combining intermediate results in branching algorithm trees to create one final result, i.e. (s1.isna() AND s2.isna())

Returns

corrections – a series with same index as the input timeseries containing corrections. Contains NaNs where any of the input series values is NaN.

Return type

pd.Series

`traval.rulelib.`**`rule_combine_nan_or`**`(*args)`

Combination rule, combine NaN values for any number of timeseries.

Used for combining intermediate results in branching algorithm trees to create one final result, i.e. (s1.isna() OR s2.isna())

Returns

corrections – a series with same index as the input timeseries containing corrections. Contains NaNs where any of the input series values is NaN.

Return type

pd.Series

`traval.rulelib.`**`rule_diff_outside_of_n_sigma`**`(series, n, max_gap='7D')`

Detection rule, calculate diff of series and identify suspect.

observations based on values outside of n * standard deviation of the difference.

Parameters

- **series** (*pd.Series*) – timeseries in which suspect values are identified
- **n** (*float, optional*) – number of standard deviations to use, by default 2
- **max_gap** (*str, optional*) – only considers observations within this maximum gap between measurements to calculate diff, by default "7D".

Returns

corrections – a series with same index as the input timeseries containing corrections. Suspect values are set to np.nan.

Return type

pd.Series

`traval.rulelib.`**`rule_diff_ufunc_threshold`**(*series*, *ufunc*, *threshold*, *max_gap='7D'*)

Detection rule, flag values based on diff, operator and threshold.

Calculate diff of series and identify suspect observations based on comparison with threshold value.

The argument ufunc is a tuple containing a function, e.g. an operator function (i.e. '>', '<', '>=', '<='). These are passed using their named equivalents, e.g. in numpy: np.greater, np.less, np.greater_equal, np.less_equal. This function essentially does the following: ufunc(series, threshold_series). The argument is passed as a tuple to bypass RuleSet logic.

Parameters

- **`series`** (*`pd.Series`*) – timeseries in which suspect values are identified
- **`ufunc`** (*`tuple`*) – tuple containing ufunc (i.e. (numpy.greater_equal,)). The function must be callable according to *ufunc(series, threshold)*. The function is passed as a tuple to bypass RuleSet logic.
- **`threshold`** (*`float`*) – value to compare diff of timeseries to
- **`max_gap`** (*`str, optional`*) – only considers observations within this maximum gap between measurements to calculate diff, by default "7D".

Returns

corrections – a series with same index as the input timeseries containing corrections. Suspect values are set to np.nan.

Return type

pd.Series

`traval.rulelib.`**`rule_flat_signal`**(*series*, *window*, *min_obs*, *std_threshold=0.0075*, *qbelow=None*, *qabove=None*, *hbelow=None*, *habove=None*)

Detection rule, flag values based on dead signal in rolling window.

Flag values when variation in signal within a window falls below a certain threshold value. Optionally provide quantiles below or above which to look for dead/flat signals.

Parameters

- **`series`** (*`pd.Series`*) – timeseries to analyse
- **`window`** (*`int`*) – number of days in window
- **`min_obs`** (*`int`*) – minimum number of observations in window to calculate standard deviation
- **`std_threshold`** (*`float, optional`*) – standard deviation threshold value, by default 7.5e-3
- **`qbelow`** (*`float, optional`*) – quantile value between 0 and 1, signifying an upper limit. Only search for flat signals below this limit. By default None.
- **`qabove`** (*`float, optional`*) – quantile value between 0 and 1, signifying a lower limit. Only search for flat signals above this limit. By default None.
- **`hbelow`** (*`float, optional`*) – absolute value in units of timeseries signifying an upper limit. Only search for flat signals below this limit. By default None.
- **`habove`** (*`float, optional`*) – absolute value in units of timeseries signifying a lower limit. Only search for flat signals above this limit. By default None.

Returns

corrections – a series with same index as the input timeseries containing corrections. Contains NaNs where the signal is considered flat or dead.

Return type
pd.Series

`traval.rulelib.`**`rule_funcdict_to_nan`**(*series*, *funcdict*)

Detection rule, flag values with dictionary of functions.

Use dictionary of functions to identify suspect values and set them to NaN.

Parameters

- **series** (*pd.Series*) – timeseries in which suspect values are identified
- **funcdict** (*dict*) – dictionary with function names as keys and functions/methods as values. Each function is applied to each value in the timeseries using *series.apply(func)*. Suspect values are those where the function evaluates to True.

Returns
corrections – a series with same index as the input timeseries containing corrections. Suspect values (according to the provided functions) are set to np.nan.

Return type
pd.Series

`traval.rulelib.`**`rule_keep_comments`**(*series*, *keep_comments*, *comment_series*, *other_series*)

Filter rule, modify timeseries to keep data with certain comments.

This rule was invented to extract timeseries only containing certain types of errors, based on labeled data. For example, to get only erroneous observations caused by sensors above the groundwater level:

- series: the raw timeseries
- keep_comments: list of comments to keep, e.g. [‘dry sensor’]
- comment_series: timeseries containing the comments for erroneous obs
- other_series: the validated timeseries where the commented observations were removed (set to NaN).

Parameters

- **series** (*pd.Series*) – timeseries to filter
- **keep_comments** (*list of str*) – list of comments to keep
- **comment_series** (*pd.Series*) – timeseries containing comments, should have same index as series
- **other_series** (*pd.Series*) – timeseries containing corrected/adjusted values corresponding to the commmented entries.

Returns
corrections – timeseries containing NaN values where comment is in keep_comments and 0 otherwise.

Return type
pd.Series

`traval.rulelib.`**`rule_max_gradient`**(*series*, *max_step=0.5*, *max_timestep='1D'*)

Detection rule, flag values when maximum gradient exceeded.

Set values tot NaN when maximum gradient between two observations is exceeded.

Parameters

- **series** (*pd.Series*) – timeseries in which suspect values are identified

- **max_step** (*float, optional*) – max jump between two observations within given timestep, by default 0.5
- **timestep** (*str, optional*) – maximum timestep to consider, by default "1D". The gradient is not calculated for values that lie farther apart.

Returns
corrections – a series with same index as the input timeseries containing corrections. Suspect values are set to np.nan.

Return type
pd.Series

traval.rulelib.**rule_offset_detection**(*series*, *threshold=0.15*, *updown_diff=0.1*, *max_gap='7D'*, *return_df=False*)

Detection rule, detect periods with an offset error.

This rule looks for jumps in both positive and negative direction that are larger than a particular threshold. It then tries to match jumps in upward direction to one in downward direction of a similar size. If this is possible, all observations between two matching but oppposite jumps are set to NaN.

Parameters

- **series** (*pd.Series*) – timeseries in which to look for offset errors
- **threshold** (*float, optional*) – minimum jump to consider as offset error, by default 0.35
- **updown_diff** (*float, optional*) – the maximum difference between two opposite jumps to consider them matching, by default 0.1
- **max_gap** (*str, optional*) – only considers observations within this maximum gap between measurements to calculate diff, by default "7D".
- **return_df** (*bool, optional*) – return the dataframe containing the potential offsets, by default False

Returns
corrections – a series with same index as the input timeseries containing corrections. Suspect values are set to np.nan.

Return type
pd.Series

traval.rulelib.**rule_other_ufunc_threshold**(*series*, *other*, *ufunc*, *threshold*)

Detection rule, flag values based on other series and threshold.

Set values to Nan based on comparison of another timeseries with a threshold value.

The argument ufunc is a tuple containing an operator function (i.e. '>', '<', '>=', '<='). These are passed using their named equivalents, e.g. in numpy: np.greater, np.less, np.greater_equal, np.less_equal. This function essentially does the following: ufunc(series, threshold_series). The argument is passed as a tuple to bypass RuleSet logic.

Parameters

- **series** (*pd.Series*) – timeseries in which suspect values are identified, only used to test if index of other overlaps
- **other** (*pd.Series*) – other timeseries based on which suspect values are identified

- **ufunc** (*tuple*) – tuple containing ufunc (i.e. (numpy.greater_equal,)). The function must be callable according to *ufunc(series, threshold)*. The function is passed as a tuple to bypass RuleSet logic.
- **threshold** (*float*) – value to compare timeseries to

Returns
corrections – a series with same index as the input timeseries containing corrections. Suspect values are set to np.nan.

Return type
pd.Series

`traval.rulelib.`**`rule_outside_bandwidth`**`(`*series*, *lowerbound*, *upperbound*`)`

Detection rule, set suspect values to NaN if outside bandwidth.

Parameters

- **series** (*pd.Series*) – timeseries in which suspect values are identified
- **lowerbound** (*pd.Series*) – timeseries containing the lower bound, if bound values are less frequent than series, bound is interpolated to series.index
- **upperbound** (*pd.Series*) – timeseries containing the upper bound, if bound values are less frequent than series, bound is interpolated to series.index

Returns
corrections – a series with same index as the input timeseries containing corrections. Suspect values are set to np.nan.

Return type
pd.Series

`traval.rulelib.`**`rule_outside_n_sigma`**`(`*series*, *n=2.0*`)`

Detection rule, set values outside of n * standard deviation to NaN

Parameters

- **series** (*pd.Series*) – timeseries in which suspect values are identified
- **n** (*float, optional*) – number of standard deviations to use, by default 2

Returns
corrections – a series with same index as the input timeseries containing corrections. Suspect values are set to np.nan.

Return type
pd.Series

`traval.rulelib.`**`rule_pastas_outside_pi`**`(`*series*, *ml*, *ci=0.95*, *min_ci=None*, *smoothfreq=None*, *tmin=None*, *tmax=None*, *savedir=None*, *verbose=False*`)`

Detection rule, flag values based on pastas model prediction interval.

Flag suspect outside prediction interval calculated by pastas timeseries model. Uses a pastas.Model and a confidence interval as input.

Parameters

- **series** (*pd.Series*) – timeseries to identify suspect observations in
- **ml** (*pastas.Model*) – timeseries model for series

- **ci** (*float, optional*) – confidence interval for calculating bandwidth, by default 0.95. Higher confidence interval means that bandwidth is wider and more observations will fall within the bounds.
- **min_ci** (*float, optional*) – value indicating minimum distance between upper and lower bounds, if ci does not meet this requirement, this value is added to the bounds. This can be used to prevent extremely narrow prediction intervals. Default is None.
- **smoothfreq** (*str, optional*) – str indicating no. of periods and frequency str (i.e. "1D") for smoothing upper and lower bounds only used if smoothbounds=True, default is None.
- **tmin** (*str or pd.Timestamp, optional*) – set tmin for model simulation
- **tmax** (*str or pd.Timestamp, optional*) – set tmax for model simulation
- **savedir** (*str, optional*) – save calculated prediction interval to folder as pickle file.

Returns
corrections – a series with same index as the input timeseries containing corrections. Suspect values are set to np.nan.

Return type
pd.Series

traval.rulelib.**rule_shift_to_manual_obs**(*series*, *hseries*, *method='linear'*, *max_dt='1D'*, *reset_dates=None*)

Adjustment rule, for shifting timeseries onto manual observations.

Used for shifting timeseries based on sensor observations onto manual verification measurements. By default uses linear interpolation between two manual verification observations.

Parameters

- **series** (*pd.Series*) – timeseries to adjust
- **hseries** (*pd.Series*) – timeseries containing manual observations
- **method** (*str, optional*) – method to use for interpolating between two manual observations, by default "linear". Other options are those that are accepted by series.reindex(): 'bfill', 'ffill', 'nearest'.
- **max_dt** (*str, optional*) – maximum amount of time between manual observation and value in series, by default "1D"
- **reset_dates** (*list, optional*) – list of dates (as str or pd.Timestamp) on which to reset the adjustments to 0.0, by default None. Useful for resetting the adjustments when the sensor is replaced, for example.

Returns
adjusted_series – timeseries containing adjustments to shift series onto manual observations.

Return type
pd.Series

traval.rulelib.**rule_spike_detection**(*series*, *threshold=0.15*, *spike_tol=0.15*, *max_gap='7D'*)

Detection rule, identify spikes in timeseries and set to NaN.

Spikes are sudden jumps in the value of a timeseries that last 1 timestep. They can be both negative or positive.

Parameters

- **series** (*pd.Series*) – timeseries in which suspect values are identified

- **threshold** (*float, optional*) – the minimum size of the jump to qualify as a spike, by default 0.15
- **spike_tol** (*float, optional*) – offset between value of timeseries before spike and after spike, by default 0.15. After a spike, the value of the timeseries is usually close to but not identical to the value that preceded the spike. Use this parameter to control how close the value has to be.
- **max_gap** (*str, optional*) – only considers observations within this maximum gap between measurements to calculate diff, by default "7D".

Returns
corrections – a series with same index as the input timeseries containing corrections. Suspect values are set to np.nan.

Return type
pd.Series

traval.rulelib.**rule_ufunc_threshold**(*series*, *ufunc*, *threshold*, *offset=0.0*)

Detection rule, flag values based on operator and threshold value.

Set values to Nan based on operator function and threshold value. The argument ufunc is a tuple containing an operator function (i.e. '>', '<', '>=', '<='). These are passed using their named equivalents, e.g. in numpy: np.greater, np.less, np.greater_equal, np.less_equal. This function essentially does the following: ufunc(series, threshold).

Parameters

- **series** (*pd.Series*) – timeseries in which suspect values are identified
- **ufunc** (*tuple*) – tuple containing ufunc (i.e. (numpy.greater_equal,)). The function must be callable according to *ufunc(series, threshold)*. The function is passed as a tuple to bypass RuleSet logic.
- **threshold** (*float or pd.Series*) – value or timeseries to compare series with
- **offset** (*float, optional*) – value that is added to the threshold, e.g. if some extra tolerance is allowable. Default value is 0.0.

Returns
corrections – a series with same index as the input timeseries containing corrections. Suspect values are set to np.nan.

Return type
pd.Series

3.4 Timeseries Comparison

class traval.ts_comparison.**DateTimeIndexComparison**(*idx1*, *idx2*)

Helper class for comparing two DateTimeIndexes.

idx_in_both()

Index members in both DateTimeIndexes.

Returns
index with entries in both

Return type
DateTimeIndex

`idx_in_idx1()`

Index members only in Index #1.

Returns
index with entries only in index #1

Return type
DateTimeIndex

`idx_in_idx2()`

Index members only in Index #2.

Returns
index with entries only in index #2

Return type
DateTimeIndex

`class traval.ts_comparison.SeriesComparison(`*s1*, *s2*, *names=None*, *diff_threshold=0.0*`)`

Object for comparing two timeseries.

Comparison yields the following categories:

- in_both_identical: in both series and difference <= than diff_threshold
- in_both_different: in both series and difference > than diff_threshold
- in_s1: only in series #1
- in_s2: only in series #2
- in_both_nan: NaN in both

Parameters

- **s1** (*pd.Series or pd.DataFrame*) – first series to compare
- **s2** (*pd.Series or pd.DataFrame*) – second series to compare
- **diff_threshold** (*float, optional*) – value beyond which a difference is considered significant, by default 0.0. Two values whose difference is smaller than threshold are considered identical.

`compare_by_comment()`

Compare series per comment.

Returns
comparison – series containing the possible comparison outcomes, but split into categories, one for each unique comment. Comments must be passed via series2.

Return type
pd.Series

Raises
`ValueError` – if no comment series is found

`comparison_series()`

Create series that indicates what happend to a value.

Series index is the union of s1 and s2 with a value indicating the status of the comparison:

- -1: value is modified
- 0: value stays the same

- 1: value only in series 1
- 2: value only in series 2
- -9999: value is NaN in both series

Returns
s – series containing status of value from comparison

Return type
pd.Series

class `traval.ts_comparison.`**`SeriesComparisonRelative`**(*s1*, *truth*, *base*, *diff_threshold=0.0*)

Object for comparing two timeseries relative to a third timeseries.

Extends the SeriesComparison object to include a comparison between two timeseries and a third base timeseries. This is used for example, when comparing the results of two error detection outcomes to the original raw timeseries.

Comparison yields both the results from SeriesComparison as well as the following categories for the relative comparison to the base timeseries:

- kept_in_both: both timeseries and the base timeseries contain values
- flagged_in_s1: value is NaN/missing in series #1
- flagged_in_s2: value is NaN/missing in series #2
- flagged_in_both: value is NaN/missing in both series #1 and series #2
- in_all_nan: value is NaN in all timeseries (series #1, #2 and base)
- introduced_in_s1: value is NaN/missing in base but has value in series #1
- introduced_in_s2: value is NaN/missing in base but has value in series #2
- introduced_in_both: value is NaN/missing in base but has value in both timeseries

Parameters

- **s1** (*pd.Series or pd.DataFrame*) – first series to compare
- **truth** (*pd.Series or pd.DataFrame*) – second series to compare, if a "truth" timeseries is available pass it as the second timeseries. Stored in object as 's2'.
- **base** (*pd.Series or pd.DataFrame*) – timeseries to compare other two series with
- **diff_threshold** (*float, optional*) – value beyond which a difference is considered significant, by default 0.0. Two values whose difference is smaller than threshold are considered identical.

See also:

SeriesComparison
Comparison of two timeseries relative to each other

`compare_to_base_by_comment`()

Compare two series to base series per comment.

Returns
comparison – Series containing the number of observations in each possible comparison category, but split per unique comment. Comments must be provided via 'truth' series (series2).

Return type
pd.Series

Raises
`ValueError` – if no comment series is available.

3.5 Timeseries Utilities

`traval.ts_utils.`**`bandwidth_moving_avg_n_sigma`**`(`*series*, *window*, *n*`)`

Calculate bandwidth around timeseries based moving average + n * std.

Parameters

- **`series`** (*`pd.Series`*) – series to calculate bandwidth for
- **`window`** (*`int`*) – number of observations to consider for moving average
- **`n`** (*`float`*) – number of standard deviations from moving average for bandwidth

Returns
bandwidth – dataframe with 2 columns, with lower and upper bandwidth

Return type
pd.DataFrame

`traval.ts_utils.`**`create_synthetic_raw_timeseries`**`(`*raw_series*, *truth_series*, *comments*`)`

Create synthetic raw timeseries.

Updates 'truth_series' (where values are labelled with a comment) with values from raw_series. Used for removing unlabeled changes between a raw and validated timeseries.

Parameters

- **`raw_series`** (*`pd.Series`*) – timeseries with raw data
- **`truth_series`** (*`pd.Series`*) – timeseries with validated data
- **`comments`** (*`pd.Series`*) – timeseries with comments. Index must be same as 'truth_series'. When value does not have a comment it must be an empty string: ''.

Returns
s – synthetic raw timeseries, same as truth_series but updated with raw_series where value has been commented.

Return type
pd.Series

`traval.ts_utils.`**`diff_with_gap_awareness`**`(`*series*, *max_gap='7D'*`)`

Get diff of timeseries with a limit on gap between two values.

Parameters

- **`series`** (*`pd.Series`*) – timeseries to calculate diff for
- **`max_gap`** (*`str, optional`*) – maximum period between two observations for calculating diff, otherwise set value to NaN, by default "7D"

Returns
diff – timeseries with diff, with NaNs whenever two values are farther apart than max_gap.

Return type
pd.Series

`traval.ts_utils.`**`interpolate_series_to_new_index`**`(`*series*, *new_index*`)`

Interpolate timeseries to new DateTimeIndex.

Parameters

- **`series`** (*`pd.Series`*) – original series
- **`new_index`** (*`DateTimeIndex`*) – new index to interpolate series to

Returns
si – new series with new index, with interpolated values

Return type
pd.Series

`traval.ts_utils.`**`mask_corrections_as_nan`**`(`*series*, *mask*`)`

Get corrections series with NaNs where mask == True.

Parameters

- **`series`** (*`pd.Series`*) – timeseries to provide corrections for
- **`mask`** (*`DateTimeIndex or boolean np.array`*) – DateTimeIndex containing timestamps where value should be set to NaN, or boolean array with same length as series set to True where value should be set to NaN. (Uses pandas .loc[mask] to set values.)

Returns
c – return corrections series

Return type
pd.Series

`traval.ts_utils.`**`resample_short_series_to_long_series`**`(`*short_series*, *long_series*`)`

Resample a short timeseries to index from a longer timeseries.

First uses 'ffill' then 'bfill' to fill new series.

Parameters

- **`short_series`** (*`pd.Series`*) – short timeseries
- **`long_series`** (*`pd.Series`*) – long timeseries

Returns
new_series – series with index from long_series and data from short_series

Return type
pd.Series

`traval.ts_utils.`**`spike_finder`**`(`*series*, *threshold=0.15*, *spike_tol=0.15*, *max_gap='7D'*`)`

Find spikes in timeseries.

Spikes are sudden jumps in the value of a timeseries that last 1 timestep. They can be both negative or positive.

Parameters

- **`series`** (*`pd.Series`*) – timeseries to find spikes in
- **`threshold`** (*`float, optional`*) – the minimum size of the jump to qualify as a spike, by default 0.15
- **`spike_tol`** (*`float, optional`*) – offset between value of timeseries before spike and after spike, by default 0.15. After a spike, the value of the timeseries is usually close to but not identical to the value that preceded the spike. Use this parameter to control how close the value has to be.

- **max_gap** (*str, optional*) – only considers observations within this maximum gap between measurements to calculate diff, by default "7D".

Returns
: **upspikes, downspikes** – pandas DateTimeIndex objects containing timestamps of upward and downward spikes.

Return type
: pandas.DateTimeIndex

`traval.ts_utils.`**`unique_nans_in_series`**`(`*series*, **args*`)`

Get mask where NaNs in series are unique compared to other series.

Parameters

- **series** (*pd.Series*) – identify unique NaNs in series
- ***args** – any number of pandas.Series

Returns
: **mask** – mask with value True where NaN is unique to series

Return type
: pd.Series

3.6 Binary Classification

`class traval.binary_classifier.`**`BinaryClassifier`**`(`*tp*, *fp*, *tn*, *fn*`)`

Class for calculating binary classification statistics.

`property accuracy`

Accuracy of binary classification.

ACC = (TP + TN) / (TP + FP + FN + TN)

where - TP : True Positives - TN : True Negatives - FP : False Positives - FN : False Negatives

`confusion_matrix``(`*as_array=False*`)`

Calculate confusion matrix.

Confusion matrix shows the performance of the algorithm given a certain truth. An abstract example of the confusion matrix:

Algorithm |

|-------------------| | error | correct |

——|---------|———|---------|

error | TP | FN |

Truth |---------|———|---------|

correct | FP | TN |

——|---------|———|---------|

where: - TP: True Positives = errors correctly detected by algorithm - TN: True Negatives = correct values correctly not flagged by algorithm - FP: False Positives = correct values marked as errors by algorithm - FN: False Negatives = errors not detected by algorithm

Parameters
`as_array` (*bool, optional*) – return data as array instead of DataFrame, by default False

Returns
data – confusion matrix

Return type
pd.DataFrame or np.array

property false_discovery_rate

False discovery rate.

FDR = 1 - PPV = FP / (FP + TP)

where - TP : True Positives - FP : False Positives

property false_negative_rate

False Negative Rate = (1 - sensitivity).

FNR = FN / (FN + TP)

where - FN : False Negatives - TP : True Positives

property false_omission_rate

False omission rate.

FOR = 1 - NPV = FN / (TN + FN)

where - TN : True Negatives - FN : False Negatives

property false_positive_rate

False Positive Rate = (1 - specificity).

FPR = FP / (FP + TN)

where - FP : False Positives - TN : True Negatives

classmethod from_confusion_matrix(*cmat*)

Create BinaryClassifier from confusion matrix.

Note: Confusion Matrix must be passed as an np.array or pd.DataFrame corresponding to: [[TP, FN], [FP, TN]], like the one returned by *BinaryClassifier.confusion_matrix*

Parameters
`cmat` (*np.array or pd.DataFrame*) –

a 2x2 dataset with structure [[TP, FN],
[FP, TN]]

Returns
BinaryClassifier object based on values in confusion matrix.

Return type
BinaryClassifier

See also:

BinaryClassifier.confusion_matrix
for explanation (of abbreviations)

classmethod from_series_comparison_relative(*comparison*)

Binary Classification object from SeriesComparisonRelative object.

Parameters
comparison (*traval.SeriesComparisonRelative*) – object comparing two timeseries with base timeseries

Returns
object for calculating binary classification statistics

Return type
BinaryClassifier

get_all_statistics(*use_abbreviations=True*)

Get all statistics in pandas.Series.

Parameters
use_abbreviations (*bool, optional*) – whether to use abbreviations or full names for index, by default True

Returns
s – series containing all statistics

Return type
pandas.Series

property informedness

Informedness statistic (a.k.a. Youden's J statistic).

Measure of diagnostic performance, and has a zero value when a diagnostic test gives the same proportion of positive results for groups with and without a condition, i.e the test is useless. A value of 1 indicates that there are no false positives or false negatives, i.e. the test is perfect.

Calculated as:

informedness = specificity + sensitivity - 1.

property matthews_correlation_coefficient

Matthews correlation coefficient (MCC).

The MCC is in essence a correlation coefficient between the observed and predicted binary classifications; it returns a value between 1 and +1. A coefficient of +1 represents a perfect prediction, 0 no better than random prediction and 1 indicates total disagreement between prediction and observation.

Returns
phi – the Matthews correlation coefficient

Return type
float

See also:

mcc
convenience method for calculating MCC

`property mcc`

Convenience method for calculating Matthews correlation coefficient.

Returns
phi – the Matthews correlation coefficient

Return type
float

See also:

matthews_correlation_coefficient
more information about the statistic

`property negative_predictive_value`

Negative predictive value.

NPV = TN / (TN + FN)

where - TN : True Negatives - FN : False Negatives

`property positive_predictive_value`

Positive predictive value (a.k.a. precision).

PPV = TP / (TP + FP)

where - TP : True Positives - FP : False Positives

`property prevalence`

Prevalance of true errors in total population.

Prevalence = (TP + FN) / (TP + FP + FN + TN)

where - TP : True Positives - TN : True Negatives - FP : False Positives - FN : False Negatives

`property sensitivity`

Sensitivity or True Positive Rate.

Statistic describing ratio of true positives identified, which also says something about the avoidance of false negatives.

Sensitivity = TP / (TP + FN)

where - TP : True Positives - FN : False Negatives

`property specificity`

Specificity or True Negative Rate.

Statistic describing ratio of true negatives identified, which also says something about the avoidance of false positives.

Specificity = TN / (TN + FP)

where - TN : True Negatives - FP : False Positives

`property true_negative_rate`

True Negative Rate. Synonym for specificity.

See specificity for description.

`property true_positive_rate`

True Positive Rate. Synonym for sensitivity.

See sensitiviy for description.

3.7 Plots

`class traval.plots.ComparisonPlots`(*cp*)

Mix-in class for plots for comparing timeseries.

`plot_relative_comparison`(*mark_unique=True*, *mark_different=True*, *mark_identical=True*, *mark_introduced=False*, *ax=None*)

Plot comparison between two timeseries relative to base timeseries.

Parameters

- **mark_unique** (*bool, optional*) – mark unique observations with colored X's, by default True
- **mark_different** (*bool, optional*) – highlight where series are different in red, by default True
- **mark_identical** (*bool, optional*) – highlight where series are identical with green, by default True
- **mark_introduced** (*bool, optional*) – mark observations that are not in the base timeseries with X's, by default False
- **ax** (*axis, optional*) – axis to plot on, by default None

Returns
ax – axis handle

Return type
axis

`plot_series_comparison`(*mark_unique=True*, *mark_different=True*, *mark_identical=True*, *ax=None*)

Plot comparison between two timeseries.

Parameters

- **mark_unique** (*bool, optional*) – mark unique values with colored X's, by default True
- **mark_different** (*bool, optional*) – highlight where timeseries differ with red, by default True
- **mark_identical** (*bool, optional*) – highlight where timeseries are identical with green, by default True
- **ax** (*axis, optional*) – axis object to plot on, by default None

Returns
ax – axis object

Return type
axis

`reset_color_dict`()

Reset color_dict to default values.

`update_color_dict`(*key*, *color=None*, *alpha=None*)

Update colors for plots.

Parameters

- **key** (*str*) – name of category to update, see *ComparisonPlots.color_dict.keys()* for options
- **color** (*str, optional*) – color name, by default None

- **alpha** (*float, optional*) – alpha value, by default None

`traval.plots.`**`det_plot`**(*fpr*, *fnr*, *labels*, *ax=None*, ***kwargs*)

Detection Error Tradeoff plot.

Adapted from scikitlearn *DetCurveDisplay*.

Parameters

- **fpr** (*list or value or array*) – false positive rate. If passed as a list loops through each entry and plots it. Otherwise just plots the array or value.
- **fnr** (*list or value or array*) – false negative rate. If passed as a list loops through each entry and plots it. Otherwise just plots the array or value.
- **labels** (*list or str*) – label for each fpr/fnr entry.
- **ax** (*matplotlib.pyplot.Axes, optional*) – axes handle to plot on, by default None, which creates a new figure

Returns
ax – axes handle

Return type
matplotlib.pyplot.Axes

`traval.plots.`**`roc_plot`**(*tpr*, *fpr*, *labels*, *colors=None*, *ax=None*, *plot_diagonal=True*, *colorbar_label=None*, ***kwargs*)

Receiver operator characteristic plot.

Plots the false positive rate (x-axis) versus the true positive rate (y-axis). The 'tpr' and 'fpr' can be passed as: - values: outcome of a single error detection algorithm - arrays: outcomes of error detection algorithm in which a detection

parameter is varied.

- lists: for passing multiple results, entries can be values or arrays, as listed above.

Parameters

- **tpr** (*list or value or array*) – true positive rate. If passed as a list loops through each entry and plots it. Otherwise just plots the array or value.
- **fpr** (*list or value or array*) – false positive rate. If passed as a list loops through each entry and plots it. Otherwise just plots the array or value.
- **labels** (*list or str*) – label for each tpr/fpr entry.
- **ax** (*matplotlib.pyplot.Axes, optional*) – axes to plot on, default is None, which creates new figure
- **plot_diagonal** (*bool, optional*) – whether to plot the diagonal (useful for combining multiple ROC plots)
- ****kwargs** – passed to ax.scatter

Returns
ax – axes instance

Return type
matplotlib.pyplot.Axes

CHAPTER

FOUR

INDICES AND TABLES

- genindex

PYTHON MODULE INDEX

INDEX