# Package: scoringutils (via r-universe)

July 4, 2024

**Title** Utilities for Scoring and Assessing Predictions

**Version** 1.2.2.9000

**Language** en-GB

**Description** `scoringutils` facilitates the evaluation of forecasts in a convenient framework based on `data.table`. It allows user to to check their forecasts and diagnose issues, to visualise forecasts and missing data, to transform data before scoring, to handle missing forecasts, to aggregate scores, and to visualise the results of the evaluation. The package mostly focuses on the evaluation of probabilistic forecasts and allows evaluating several different forecast types and input formats. Find more information about the package in the Vignettes as well as in the accompanying paper (<doi:10.48550/arXiv.2205.07090>).

**License** MIT + file LICENSE

**Encoding** UTF-8

**LazyData** true

**Imports** checkmate, cli, data.table, ggplot2 (>= 3.4.0), methods, Metrics, scoringRules, stats

**Suggests** ggdist, kableExtra, knitr, magrittr, rmarkdown, testthat (>= 3.1.9), vdiffr

**Config/Needs/website** r-lib/pkgdown, amirmasoudabdol/preferably

**Config/testthat/edition** 3

**RoxygenNote** 7.3.1

**URL** https://doi.org/10.48550/arXiv.2205.07090, https://epiforecasts.io/scoringutils/, https://github.com/epiforecasts/scoringutils

**BugReports** https://github.com/epiforecasts/scoringutils/issues

**VignetteBuilder** knitr

**Depends** R (>= 4.0)

**Roxygen** list(markdown = TRUE)

**Repository** https://epiforecasts.r-universe.dev

**RemoteUrl** https://github.com/epiforecasts/scoringutils

**RemoteRef** HEAD

**RemoteSha** 4ef767a2789a6b210395be69f35c8dd7e2fedd25

# Contents

## add_relative_skill *Add relative skill scores based on pairwise comparisons*

### Description

Adds a columns with relative skills computed by running pairwise comparisons on the scores. For more information on the computation of relative skill, see `get_pairwise_comparisons()`. Relative skill will be calculated for the aggregation level specified in `by`.

### Usage

```
add_relative_skill(
  scores,
  by = "model",
  metric = intersect(c("wis", "crps", "brier_score"), names(scores)),
  baseline = NULL
)
```

### Arguments

| | |
|---|---|
| `scores` | An object of class `scores` (a data.table with scores and an additional attribute `metrics` as produced by `score()`). |
| `by` | Character vector with column names that define the grouping level for the pairwise comparisons. By default (`model`), there will be one relative skill score per model. If, for example, `by = c("model", "location")`. Then you will get a separate relative skill score for every model in every location. Internally, the data.table with scores will be split according `by` (removing "model" before splitting) and the pairwise comparisons will be computed separately for the split data.tables. |
| `metric` | A string with the name of the metric for which a relative skill shall be computed. By default this is either "crps", "wis" or "brier_score" if any of these are available. |
| `baseline` | A string with the name of a model. If a baseline is given, then a scaled relative skill with respect to the baseline will be returned. By default (`NULL`), relative skill will not be scaled with respect to a baseline model. |

## ae_median_quantile *Absolute error of the median (quantile-based version)*

### Description

Compute the absolute error of the median calculated as

$$\text{abs}(\text{observed} - \text{median prediction})$$

The median prediction is the predicted value for which quantile_level == 0.5, the function therefore requires 0.5 to be among the quantile levels in `quantile_level`.

### Usage

```
ae_median_quantile(observed, predicted, quantile_level)
```

### Arguments

| | |
|---|---|
| `observed` | Numeric vector of size n with the observed values. |
| `predicted` | Numeric nxN matrix of predictive quantiles, n (number of rows) being the number of forecasts (corresponding to the number of observed values) and N (number of columns) the number of quantiles per forecast. If `observed` is just a single number, then predicted can just be a vector of size N. |
| `quantile_level` | Vector of of size N with the quantile levels for which predictions were made. |

### Value

Numeric vector of length N with the absolute error of the median.

### See Also

`ae_median_sample()`

### Examples

```
observed <- rnorm(30, mean = 1:30)
predicted_values <- replicate(3, rnorm(30, mean = 1:30))
ae_median_quantile(
  observed, predicted_values, quantile_level = c(0.2, 0.5, 0.8)
)
```

---

## `ae_median_sample` *Absolute error of the median (sample-based version)*

---

### Description

Absolute error of the median calculated as

$$\text{abs}(\text{observevd} - \text{median\_prediction})$$

### Usage

```
ae_median_sample(observed, predicted)
```

### Arguments

| | |
|---|---|
| `observed` | A vector with observed values of size n |
| `predicted` | nxN matrix of predictive samples, n (number of rows) being the number of data points and N (number of columns) the number of Monte Carlo samples. Alternatively, `predicted` can just be a vector of size n. |

### Value

vector with the scoring values

### See Also

ae_median_quantile()

### Examples

```
observed <- rnorm(30, mean = 1:30)
predicted_values <- matrix(rnorm(30, mean = 1:30))
ae_median_sample(observed, predicted_values)
```

---

## assert_dims_ok_point *Assert Inputs Have Matching Dimensions*

---

### Description

Function assesses whether input dimensions match. In the following, n is the number of observations / forecasts. Scalar values may be repeated to match the length of the other input. Allowed options are therefore:

- `observed` is vector of length 1 or length n
- `predicted` is:
  - a vector of of length 1 or length n
  - a matrix with n rows and 1 column

### Usage

```
assert_dims_ok_point(observed, predicted)
```

### Arguments

| | |
|---|---|
| `observed` | Input to be checked. Should be a factor of length n with exactly two levels, holding the observed values. The highest factor level is assumed to be the reference level. This means that `predicted` represents the probability that the observed value is equal to the highest factor level. |
| `predicted` | Input to be checked. `predicted` should be a vector of length n, holding probabilities. Alternatively, `predicted` can be a matrix of size n x 1. Values represent the probability that the corresponding value in `observed` will be equal to the highest available factor level. |

### Value

Returns NULL invisibly if the assertion was successful and throws an error otherwise.

`assert_forecast` *Assert that input is a forecast object and passes validations*

## Description

Assert that an object is a forecast object (i.e. a `data.table` with a class `forecast` and an additional class `forecast_*` corresponding to the forecast type).

## Usage

```
assert_forecast(forecast, forecast_type = NULL, verbose = TRUE, ...)

## Default S3 method:
assert_forecast(forecast, forecast_type = NULL, verbose = TRUE, ...)

## S3 method for class 'forecast_binary'
assert_forecast(forecast, forecast_type = NULL, verbose = TRUE, ...)

## S3 method for class 'forecast_point'
assert_forecast(forecast, forecast_type = NULL, verbose = TRUE, ...)

## S3 method for class 'forecast_quantile'
assert_forecast(forecast, forecast_type = NULL, verbose = TRUE, ...)

## S3 method for class 'forecast_sample'
assert_forecast(forecast, forecast_type = NULL, verbose = TRUE, ...)
```

## Arguments

| | |
|---|---|
| `forecast` | A forecast object (a validated data.table with predicted and observed values, see `as_forecast()`) |
| `forecast_type` | (optional) The forecast type you expect the forecasts to have. If the forecast type as determined by `scoringutils` based on the input does not match this, an error will be thrown. If `NULL` (the default), the forecast type will be inferred from the data. |
| `verbose` | Logical. If `FALSE` (default is `TRUE`), no messages and warnings will be created. |
| `...` | Additional arguments |

## Value

Returns `NULL` invisibly.

## Forecast types and input formats

Various different forecast types / forecast formats are supported. At the moment, those are:

- point forecasts
- binary forecasts ("soft binary classification")
- Probabilistic forecasts in a quantile-based format (a forecast is represented as a set of predictive quantiles)
- Probabilistic forecasts in a sample-based format (a forecast is represented as a set of predictive samples)

Forecast types are determined based on the columns present in the input data. Here is an overview of the required format for each forecast type:

| Forecast type | | | column | type |
|---|---|---|---|---|
| All forecast types | | | `observed`<br>`predicted`<br>`model` | |
| Classification | Binary | Soft classification (prediction is probability) | `observed`<br>`predicted` | factor with 2 levels<br>numeric [0,1] |
| Point forecasts | Discrete, Continuous | | `observed`<br>`predicted` | numeric<br>numeric |
| Probabilistic forecast | Discrete, Continuous | Sample format | `observed`<br>`predicted`<br>`sample_id` | numeric<br>numeric<br>numeric |
| | | Quantile format | `observed`<br>`predicted`<br>`quantile_level` | numeric<br>numeric<br>numeric [0,1] |

*All forecast types* require a data.frame or similar with columns `observed` `predicted`, and `model`.

*Point forecasts* require a column `observed` of type numeric and a column `predicted` of type numeric.

*Binary forecasts* require a column `observed` of type factor with exactly two levels and a column `predicted` of type numeric with probabilities, corresponding to the probability that `observed` is equal to the second factor level. See details here for more information.

*Quantile-based forecasts* require a column `observed` of type numeric, a column `predicted` of type numeric, and a column `quantile_level` of type numeric with quantile-levels (between 0 and 1).

*Sample-based forecasts* require a column `observed` of type numeric, a column `predicted` of type numeric, and a column `sample_id` of type numeric with sample indices.

For more information see the vignettes and the example data (example_quantile, example_sample_continuous, example_sample_discrete, `example_point()`, and example_binary).

## Examples

```
forecast <- as_forecast(example_binary)
assert_forecast(forecast)
```

## assert_forecast_generic *Validation common to all forecast types*

### Description

The function runs input checks that apply to all input data, regardless of forecast type. The function

- asserts that the forecast is a data.table which has columns `observed` and `predicted`, as well as a column called `model`.
- checks the forecast type and forecast unit
- checks there are no duplicate forecasts
- if appropriate, checks the number of samples / quantiles is the same for all forecasts.

### Usage

```
assert_forecast_generic(data, verbose = TRUE)
```

### Arguments

| | |
|---|---|
| `data` | A data.table with forecasts and observed values that should be validated. |
| `verbose` | Logical. If `FALSE` (default is `TRUE`), no messages and warnings will be created. |

### Value

returns the input

## assert_forecast_type *Assert that forecast type is as expected*

### Description

Assert that forecast type is as expected

### Usage

```
assert_forecast_type(data, actual = get_forecast_type(data), desired = NULL)
```

### Arguments

| | |
|---|---|
| `data` | A forecast object as produced by `as_forecast()`. |
| `actual` | The actual forecast type of the data |
| `desired` | The desired forecast type of the data |

### Value

Returns NULL invisibly if the assertion was successful and throws an error otherwise.

---

`assert_input_binary` *Assert that inputs are correct for binary forecast*

---

### Description

Function assesses whether the inputs correspond to the requirements for scoring binary forecasts.

### Usage

```
assert_input_binary(observed, predicted)
```

### Arguments

| | |
|---|---|
| `observed` | Input to be checked. Should be a factor of length n with exactly two levels, holding the observed values. The highest factor level is assumed to be the reference level. This means that `predicted` represents the probability that the observed value is equal to the highest factor level. |
| `predicted` | Input to be checked. `predicted` should be a vector of length n, holding probabilities. Alternatively, `predicted` can be a matrix of size n x 1. Values represent the probability that the corresponding value in `observed` will be equal to the highest available factor level. |

### Value

Returns NULL invisibly if the assertion was successful and throws an error otherwise.

---

`assert_input_interval`
*Assert that inputs are correct for interval-based forecast*

---

### Description

Function assesses whether the inputs correspond to the requirements for scoring interval-based forecasts.

### Usage

```
assert_input_interval(observed, lower, upper, interval_range)
```

### Arguments

| | |
|---|---|
| `observed` | Input to be checked. Should be a numeric vector with the observed values of size n. |
| `lower` | Input to be checked. Should be a numeric vector of size n that holds the predicted value for the lower bounds of the prediction intervals. |
| `upper` | Input to be checked. Should be a numeric vector of size n that holds the predicted value for the upper bounds of the prediction intervals. |
| `interval_range` | Input to be checked. Should be a vector of size n that denotes the interval range in percent. E.g. a value of 50 denotes a (25%, 75%) prediction interval. |

### Value

Returns NULL invisibly if the assertion was successful and throws an error otherwise.

---

## `assert_input_point` *Assert that inputs are correct for point forecast*

---

### Description

Function assesses whether the inputs correspond to the requirements for scoring point forecasts.

### Usage

```
assert_input_point(observed, predicted)
```

### Arguments

| | |
|---|---|
| `observed` | Input to be checked. Should be a numeric vector with the observed values of size n. |
| `predicted` | Input to be checked. Should be a numeric vector with the predicted values of size n. |

### Value

Returns NULL invisibly if the assertion was successful and throws an error otherwise.

## assert_input_quantile

*Assert that inputs are correct for quantile-based forecast*

### Description

Function assesses whether the inputs correspond to the requirements for scoring quantile-based forecasts.

### Usage

```
assert_input_quantile(
  observed,
  predicted,
  quantile_level,
  unique_quantile_levels = TRUE
)
```

### Arguments

| | |
|---|---|
| `observed` | Input to be checked. Should be a numeric vector with the observed values of size n. |
| `predicted` | Input to be checked. Should be nxN matrix of predictive quantiles, n (number of rows) being the number of data points and N (number of columns) the number of quantiles per forecast. If `observed` is just a single number, then predicted can just be a vector of size N. |
| `quantile_level` | Input to be checked. Should be a vector of size N that denotes the quantile levels corresponding to the columns of the prediction matrix. |
| `unique_quantile_levels` | Whether the quantile levels are required to be unique (`TRUE`, the default) or not (`FALSE`). |

### Value

Returns NULL invisibly if the assertion was successful and throws an error otherwise.

## assert_input_sample

*Assert that inputs are correct for sample-based forecast*

### Description

Function assesses whether the inputs correspond to the requirements for scoring sample-based forecasts.

### Usage

```
assert_input_sample(observed, predicted)
```

### Arguments

| | |
|---|---|
| `observed` | Input to be checked. Should be a numeric vector with the observed values of size n. |
| `predicted` | Input to be checked. Should be a numeric nxN matrix of predictive samples, n (number of rows) being the number of data points and N (number of columns) the number of samples per forecast. If `observed` is just a single number, then predicted values can just be a vector of size N. |

### Value

Returns NULL invisibly if the assertion was successful and throws an error otherwise.

---

## `as_forecast` *Create a* `forecast` *object*

---

### Description

Process and validate a data.frame (or similar) or similar with forecasts and observations. If the input passes all input checks, it will be converted to a `forecast` object. A forecast object is a `data.table` with a class `forecast` and an additional class that depends on the forecast type. See the details section below for more information on the expected input formats.

`as_forecast()` gives users some control over how their data is parsed. Using the arguments `observed`, `predicted`, and `model`, users can rename existing columns of their input data to match the required columns for a forecast object. Using the argument `forecast_unit`, users can specify the the columns that uniquely identify a single forecast (and remove the others, see `set_forecast_unit()` for details).

### Usage

```
as_forecast(data, ...)

## Default S3 method:
as_forecast(
  data,
  forecast_unit = NULL,
  forecast_type = NULL,
  observed = NULL,
  predicted = NULL,
  model = NULL,
  quantile_level = NULL,
  sample_id = NULL,
  ...
)
```

### Arguments

| | |
|---|---|
| `data` | A data.frame (or similar) with predicted and observed values. See the details section of `as_forecast()` for additional information on required input formats. |
| `...` | Additional arguments |
| `forecast_unit` | (optional) Name of the columns in `data` (after any renaming of columns done by `as_forecast()`) that denote the unit of a single forecast. See `get_forecast_unit()` for details. If `NULL` (the default), all columns that are not required columns are assumed to form the unit of a single forecast. If specified, all columns that are not part of the forecast unit (or required columns) will be removed. |
| `forecast_type` | (optional) The forecast type you expect the forecasts to have. If the forecast type as determined by `scoringutils` based on the input does not match this, an error will be thrown. If `NULL` (the default), the forecast type will be inferred from the data. |
| `observed` | (optional) Name of the column in `data` that contains the observed values. This column will be renamed to "observed". |
| `predicted` | (optional) Name of the column in `data` that contains the predicted values. This column will be renamed to "predicted". |
| `model` | (optional) Name of the column in `data` that contains the names of the models/forecasters that generated the predicted values. This column will be renamed to "model". |
| `quantile_level` | (optional) Name of the column in `data` that contains the quantile level of the predicted values. This column will be renamed to "quantile_level". Only applicable to quantile-based forecasts. |
| `sample_id` | (optional) Name of the column in `data` that contains the sample id. This column will be renamed to "sample_id". Only applicable to sample-based forecasts. |

### Value

Depending on the forecast type, an object of the following class will be returned:

- `forecast_binary` for binary forecasts
- `forecast_point` for point forecasts
- `forecast_sample` for sample-based forecasts
- `forecast_quantile` for quantile-based forecasts

### Forecast types and input formats

Various different forecast types / forecast formats are supported. At the moment, those are:

- point forecasts
- binary forecasts ("soft binary classification")

- Probabilistic forecasts in a quantile-based format (a forecast is represented as a set of predictive quantiles)
- Probabilistic forecasts in a sample-based format (a forecast is represented as a set of predictive samples)

Forecast types are determined based on the columns present in the input data. Here is an overview of the required format for each forecast type:

| Forecast type | | | column | type |
|---|---|---|---|---|
| All forecast types | | | `observed`<br>`predicted`<br>`model` | |
| Classification | Binary | Soft classification (prediction is probability) | `observed`<br>`predicted` | factor with 2 levels<br>numeric [0,1] |
| Point forecasts | Discrete, Continuous | | `observed`<br>`predicted` | numeric<br>numeric |
| Probabilistic forecast | Discrete, Continuous | Sample format | `observed`<br>`predicted`<br>`sample_id` | numeric<br>numeric<br>numeric |
| | | Quantile format | `observed`<br>`predicted`<br>`quantile_level` | numeric<br>numeric<br>numeric [0,1] |

*All forecast types* require a data.frame or similar with columns `observed` `predicted`, and `model`.

*Point forecasts* require a column `observed` of type numeric and a column `predicted` of type numeric.

*Binary forecasts* require a column `observed` of type factor with exactly two levels and a column `predicted` of type numeric with probabilities, corresponding to the probability that `observed` is equal to the second factor level. See details here for more information.

*Quantile-based forecasts* require a column `observed` of type numeric, a column `predicted` of type numeric, and a column `quantile_level` of type numeric with quantile-levels (between 0 and 1).

*Sample-based forecasts* require a column `observed` of type numeric, a column `predicted` of type numeric, and a column `sample_id` of type numeric with sample indices.

For more information see the vignettes and the example data (example_quantile, example_sample_continuous, example_sample_discrete, `example_point()`, and example_binary).

### Forecast unit

In order to score forecasts, `scoringutils` needs to know which of the rows of the data belong together and jointly form a single forecasts. This is easy e.g. for point forecast, where there is one row per forecast. For quantile or sample-based forecasts, however, there are multiple rows that belong to a single forecast.

The *forecast unit* or *unit of a single forecast* is then described by the combination of columns that uniquely identify a single forecast. For example, we could have forecasts made by different models in various locations at different time points, each for several weeks into the future. The forecast unit could then be described as `forecast_unit = c("model", "location", "forecast_date", "forecast_horizon")`. `scoringutils` automatically tries to determine the unit of a single forecast. It uses all existing columns for

this, which means that no columns must be present that are unrelated to the forecast unit. As a very simplistic example, if you had an additional row, "even", that is one if the row number is even and zero otherwise, then this would mess up scoring as `scoringutils` then thinks that this column was relevant in defining the forecast unit.

In order to avoid issues, we recommend setting the forecast unit explicitly, usually through the `forecast_unit` argument in `as_forecast()`. This will drop unneeded columns, while making sure that all necessary, 'protected columns' like "predicted" or "observed" are retained.

### Examples

```
as_forecast(example_binary)
as_forecast(
  example_quantile,
  forecast_unit = c("model", "target_type", "target_end_date",
                    "horizon", "location")
)
```

---

## bias_quantile *Determines bias of quantile forecasts*

---

### Description

Determines bias from quantile forecasts. For an increasing number of quantiles this measure converges against the sample based bias version for integer and continuous forecasts.

### Usage

```
bias_quantile(observed, predicted, quantile_level, na.rm = TRUE)
```

### Arguments

| | |
|---|---|
| `observed` | Numeric vector of size n with the observed values. |
| `predicted` | Numeric nxN matrix of predictive quantiles, n (number of rows) being the number of forecasts (corresponding to the number of observed values) and N (number of columns) the number of quantiles per forecast. If `observed` is just a single number, then predicted can just be a vector of size N. |
| `quantile_level` | Vector of of size N with the quantile levels for which predictions were made. Note that if this does not contain the median (0.5) then the median is imputed as being the mean of the two innermost quantiles. |
| `na.rm` | Logical. Should missing values be removed? |

### Details

For quantile forecasts, bias is measured as

$$B_t = (1-2 \cdot \max\{i | q_{t,i} \in Q_t \wedge q_{t,i} \leq x_t\}) \mathbf{1}(x_t \leq q_{t,0.5}) + (1-2 \cdot \min\{i | q_{t,i} \in Q_t \wedge q_{t,i} \geq x_t\}) 1(x_t \geq q_{t,0.5}),$$

where $Q_t$ is the set of quantiles that form the predictive distribution at time $t$ and $x_t$ is the observed value. For consistency, we define $Q_t$ such that it always includes the element $q_{t,0} = -\infty$ and $q_{t,1} = \infty$. 1() is the indicator function that is 1 if the condition is satisfied and 0 otherwise.

In clearer terms, bias $B_t$ is:

- $1-2\cdot$ the maximum percentile rank for which the corresponding quantile is still smaller than or equal to the observed value, *if the observed value is smaller than the median of the predictive distribution.*
- $1-2\cdot$ the minimum percentile rank for which the corresponding quantile is still larger than or equal to the observed value *if the observed value is larger than the median of the predictive distribution..*
- 0 *if the observed value is exactly the median* (both terms cancel out)

Bias can assume values between -1 and 1 and is 0 ideally (i.e. unbiased).

Note that if the given quantiles do not contain the median, the median is imputed as the mean of the two innermost quantiles.

For a large enough number of quantiles, the percentile rank will equal the proportion of predictive samples below the observed value, and the bias metric coincides with the one for continuous forecasts (see `bias_sample()`).

### Value

scalar with the quantile bias for a single quantile prediction

### Examples

```
predicted <- matrix(c(1.5:23.5, 3.3:25.3), nrow = 2, byrow = TRUE)
quantile_level <- c(0.01, 0.025, seq(0.05, 0.95, 0.05), 0.975, 0.99)
observed <- c(15, 12.4)
bias_quantile(observed, predicted, quantile_level)
```

---

## `bias_sample` *Determine bias of forecasts*

---

### Description

Determines bias from predictive Monte-Carlo samples. The function automatically recognises, whether forecasts are continuous or integer valued and adapts the Bias function accordingly.

### Usage

```
bias_sample(observed, predicted)
```

### Arguments

| | |
|---|---|
| `observed` | A vector with observed values of size n |
| `predicted` | nxN matrix of predictive samples, n (number of rows) being the number of data points and N (number of columns) the number of Monte Carlo samples. Alternatively, `predicted` can just be a vector of size n. |

### Details

For continuous forecasts, Bias is measured as

$$B_t(P_t, x_t) = 1 - 2 * (P_t(x_t))$$

where $P_t$ is the empirical cumulative distribution function of the prediction for the observed value $x_t$. Computationally, $P_t(x_t)$ is just calculated as the fraction of predictive samples for $x_t$ that are smaller than $x_t$.

For integer valued forecasts, Bias is measured as

$$B_t(P_t, x_t) = 1 - (P_t(x_t) + P_t(x_t + 1))$$

to adjust for the integer nature of the forecasts.

In both cases, Bias can assume values between -1 and 1 and is 0 ideally.

### Value

Numeric vector of length n with the biases of the predictive samples with respect to the observed values.

### References

The integer valued Bias function is discussed in Assessing the performance of real-time epidemic forecasts: A case study of Ebola in the Western Area region of Sierra Leone, 2014-15 Funk S, Camacho A, Kucharski AJ, Lowe R, Eggo RM, et al. (2019) Assessing the performance of real-time epidemic forecasts: A case study of Ebola in the Western Area region of Sierra Leone, 2014-15. PLOS Computational Biology 15(2): e1006785. doi:10.1371/journal.pcbi.1006785

### Examples

```
## integer valued forecasts
observed <- rpois(30, lambda = 1:30)
predicted <- replicate(200, rpois(n = 30, lambda = 1:30))
bias_sample(observed, predicted)

## continuous forecasts
```

```
observed <- rnorm(30, mean = 1:30)
predicted <- replicate(200, rnorm(30, mean = 1:30))
bias_sample(observed, predicted)
```

---

`check_columns_present` *Check column names are present in a data.frame*

---

## Description

The functions loops over the column names and checks whether they are present. If an issue is encountered, the function immediately stops and returns a message with the first issue encountered.

## Usage

```
check_columns_present(data, columns)
```

## Arguments

| | |
|---|---|
| `data` | A data.frame or similar to be checked |
| `columns` | A character vector of column names to check |

## Value

Returns TRUE if the check was successful and a string with an error message otherwise.

---

`check_dims_ok_point` *Check Inputs Have Matching Dimensions*

---

## Description

Function assesses whether input dimensions match. In the following, n is the number of observations / forecasts. Scalar values may be repeated to match the length of the other input. Allowed options are therefore:

- `observed` is vector of length 1 or length n
- `predicted` is:
  - a vector of of length 1 or length n
  - a matrix with n rows and 1 column

## Usage

```
check_dims_ok_point(observed, predicted)
```

### Arguments

| | |
|---|---|
| `observed` | Input to be checked. Should be a factor of length n with exactly two levels, holding the observed values. The highest factor level is assumed to be the reference level. This means that `predicted` represents the probability that the observed value is equal to the highest factor level. |
| `predicted` | Input to be checked. `predicted` should be a vector of length n, holding probabilities. Alternatively, `predicted` can be a matrix of size n x 1. Values represent the probability that the corresponding value in `observed` will be equal to the highest available factor level. |

### Value

Returns TRUE if the check was successful and a string with an error message otherwise.

---

## `check_duplicates` *Check that there are no duplicate forecasts*

---

### Description

Runs `get_duplicate_forecasts()` and returns a message if an issue is encountered

### Usage

```
check_duplicates(data)
```

### Arguments

| | |
|---|---|
| `data` | A data.frame as used for `score()` |

### Value

Returns TRUE if the check was successful and a string with an error message otherwise.

---

## `check_input_binary` *Check that inputs are correct for binary forecast*

---

### Description

Function assesses whether the inputs correspond to the requirements for scoring binary forecasts.

### Usage

```
check_input_binary(observed, predicted)
```

### Arguments

| | |
|---|---|
| `observed` | Input to be checked. Should be a factor of length n with exactly two levels, holding the observed values. The highest factor level is assumed to be the reference level. This means that `predicted` represents the probability that the observed value is equal to the highest factor level. |
| `predicted` | Input to be checked. `predicted` should be a vector of length n, holding probabilities. Alternatively, `predicted` can be a matrix of size n x 1. Values represent the probability that the corresponding value in `observed` will be equal to the highest available factor level. |

### Value

Returns TRUE if the check was successful and a string with an error message otherwise.

---

## `check_input_interval` *Check that inputs are correct for interval-based forecast*

---

### Description

Function assesses whether the inputs correspond to the requirements for scoring interval-based forecasts.

### Usage

```
check_input_interval(observed, lower, upper, interval_range)
```

### Arguments

| | |
|---|---|
| `observed` | Input to be checked. Should be a numeric vector with the observed values of size n. |
| `lower` | Input to be checked. Should be a numeric vector of size n that holds the predicted value for the lower bounds of the prediction intervals. |
| `upper` | Input to be checked. Should be a numeric vector of size n that holds the predicted value for the upper bounds of the prediction intervals. |
| `interval_range` | Input to be checked. Should be a vector of size n that denotes the interval range in percent. E.g. a value of 50 denotes a (25%, 75%) prediction interval. |

### Value

Returns TRUE if the check was successful and a string with an error message otherwise.

## check_input_point *Check that inputs are correct for point forecast*

### Description

Function assesses whether the inputs correspond to the requirements for scoring point forecasts.

### Usage

```
check_input_point(observed, predicted)
```

### Arguments

| | |
|---|---|
| observed | Input to be checked. Should be a numeric vector with the observed values of size n. |
| predicted | Input to be checked. Should be a numeric vector with the predicted values of size n. |

### Value

Returns TRUE if the check was successful and a string with an error message otherwise.

## check_input_quantile *Check that inputs are correct for quantile-based forecast*

### Description

Function assesses whether the inputs correspond to the requirements for scoring quantile-based forecasts.

### Usage

```
check_input_quantile(observed, predicted, quantile_level)
```

### Arguments

| | |
|---|---|
| observed | Input to be checked. Should be a numeric vector with the observed values of size n. |
| predicted | Input to be checked. Should be nxN matrix of predictive quantiles, n (number of rows) being the number of data points and N (number of columns) the number of quantiles per forecast. If observed is just a single number, then predicted can just be a vector of size N. |
| quantile_level | Input to be checked. Should be a vector of size N that denotes the quantile levels corresponding to the columns of the prediction matrix. |

### Value

Returns TRUE if the check was successful and a string with an error message otherwise.

---

`check_input_sample` *Check that inputs are correct for sample-based forecast*

---

### Description

Function assesses whether the inputs correspond to the requirements for scoring sample-based forecasts.

### Usage

```
check_input_sample(observed, predicted)
```

### Arguments

| | |
|---|---|
| `observed` | Input to be checked. Should be a numeric vector with the observed values of size n. |
| `predicted` | Input to be checked. Should be a numeric nxN matrix of predictive samples, n (number of rows) being the number of data points and N (number of columns) the number of samples per forecast. If `observed` is just a single number, then predicted values can just be a vector of size N. |

### Value

Returns TRUE if the check was successful and a string with an error message otherwise.

---

`check_number_per_forecast`
*Check that all forecasts have the same number of quantiles or samples*

---

### Description

Function checks the number of quantiles or samples per forecast. If the number of quantiles or samples is the same for all forecasts, it returns TRUE and a string with an error message otherwise.

### Usage

```
check_number_per_forecast(data, forecast_unit)
```

### Arguments

| | |
|---|---|
| `data` | A data.frame or similar to be checked |
| `forecast_unit` | Character vector denoting the unit of a single forecast. |

### Value

Returns TRUE if the check was successful and a string with an error message otherwise.

---

## `check_numeric_vector` *Check whether an input is an atomic vector of mode 'numeric'*

---

### Description

Helper function to check whether an input is a numeric vector.

### Usage

```
check_numeric_vector(x, ...)
```

### Arguments

| | |
|---|---|
| `x` | input to check |
| `...` | Arguments passed on to `checkmate::check_numeric` |

`lower` [`numeric(1)`]
: Lower value all elements of `x` must be greater than or equal to.

`upper` [`numeric(1)`]
: Upper value all elements of `x` must be lower than or equal to.

`finite` [`logical(1)`]
: Check for only finite values? Default is `FALSE`.

`any.missing` [`logical(1)`]
: Are vectors with missing values allowed? Default is `TRUE`.

`all.missing` [`logical(1)`]
: Are vectors with no non-missing values allowed? Default is `TRUE`. Note that empty vectors do not have non-missing values.

`len` [`integer(1)`]
: Exact expected length of `x`.

`min.len` [`integer(1)`]
: Minimal length of `x`.

`max.len` [`integer(1)`]
: Maximal length of `x`.

`unique` [`logical(1)`]
: Must all values be unique? Default is `FALSE`.

`sorted` [`logical(1)`]
: Elements must be sorted in ascending order. Missing values are ignored.

`names` [`character(1)`]
: Check for names. See `checkNamed` for possible values. Default is "any" which performs no check at all. Note that you can use `checkSubset` to check for a specific set of names.

`typed.missing` [`logical(1)`]
: If set to `FALSE` (default), all types of missing values (`NA`, `NA_integer_`, `NA_real_`, `NA_character_` or `NA_character_`) as well as empty vectors are allowed while type-checking atomic input. Set to `TRUE` to enable strict type checking.

`null.ok` [`logical(1)`]
: If set to `TRUE`, `x` may also be `NULL`. In this case only a type check of `x` is performed, all additional checks are disabled.

## Value

Returns TRUE if the check was successful and a string with an error message otherwise.

---

`check_try` *Helper function to convert assert statements into checks*

---

## Description

Tries to execute an expression. Internally, this is used to see whether assertions fail when checking inputs (i.e. to convert an `assert_*()` statement into a check). If the expression fails, the error message is returned. If the expression succeeds, `TRUE` is returned.

## Usage

```
check_try(expr)
```

## Arguments

| | |
|---|---|
| `expr` | an expression to be evaluated |

## Value

Returns TRUE if the check was successful and a string with an error message otherwise.

## crps_sample *(Continuous) ranked probability score*

### Description

Wrapper around the `crps_sample()` function from the **scoringRules** package. Can be used for continuous as well as integer valued forecasts

### Usage

```
crps_sample(observed, predicted, ...)
```

### Arguments

| | |
|---|---|
| `observed` | A vector with observed values of size n |
| `predicted` | nxN matrix of predictive samples, n (number of rows) being the number of data points and N (number of columns) the number of Monte Carlo samples. Alternatively, `predicted` can just be a vector of size n. |
| `...` | Additional arguments passed to crps_sample() from the scoringRules package. |

### Value

Vector with scores.

### References

Alexander Jordan, Fabian Krüger, Sebastian Lerch, Evaluating Probabilistic Forecasts with scoringRules, https://www.jstatsoft.org/article/view/v090i12

### Examples

```
observed <- rpois(30, lambda = 1:30)
predicted <- replicate(200, rpois(n = 30, lambda = 1:30))
crps_sample(observed, predicted)
```

## customise_metric *Customises a metric function with additional arguments.*

### Description

This function takes a metric function and additional arguments, and returns a new function that includes the additional arguments when calling the original metric function.

This is the expected way to pass additional arguments to a metric when evaluating a forecast using `score()`: To evaluate a forecast using a metric with an additional argument, you need to create a custom version of the scoring function with the argument included. You then need to create an updated version of the list of scoring functions that includes your customised metric and pass this to `score()`.

### Usage

```
customise_metric(metric, ...)

customize_metric(metric, ...)
```

### Arguments

| | |
|---|---|
| metric | The metric function to be customised. |
| ... | Additional arguments to be included when calling the metric function. |

### Value

A customised metric function.

### Examples

```
# Create a customised metric function
custom_metric <- customise_metric(mean, na.rm = TRUE)

# Use the customised metric function
values <- c(1, 2, NA, 4, 5)
custom_metric(values)

# Updating metrics list to calculate 70% coverage in `score()`
interval_coverage_70 <- customise_metric(
  interval_coverage, interval_range = 70
)
updated_metrics <- c(
  metrics_quantile(),
  "interval_coverage_70" = interval_coverage_70
)
score(
  as_forecast(example_quantile),
  metrics = updated_metrics
)
```

---

## dss_sample *Dawid-Sebastiani score*

---

### Description

Wrapper around the dss_sample() function from the **scoringRules** package.

### Usage

```
dss_sample(observed, predicted, ...)
```

### Arguments

| | |
|---|---|
| `observed` | A vector with observed values of size n |
| `predicted` | nxN matrix of predictive samples, n (number of rows) being the number of data points and N (number of columns) the number of Monte Carlo samples. Alternatively, `predicted` can just be a vector of size n. |
| `...` | Additional arguments passed to dss_sample() from the scoringRules package. |

### Value

Vector with scores.

### References

Alexander Jordan, Fabian Krüger, Sebastian Lerch, Evaluating Probabilistic Forecasts with scoringRules, https://www.jstatsoft.org/article/view/v090i12

### Examples

```
observed <- rpois(30, lambda = 1:30)
predicted <- replicate(200, rpois(n = 30, lambda = 1:30))
dss_sample(observed, predicted)
```

---

## ensure_model_column *Assure that data has a* `model` *column*

---

### Description

Check whether the data.table has a column called `model`. If not, a column called `model` is added with the value `Unspecified model`.

### Usage

```
ensure_model_column(data)
```

### Arguments

| | |
|---|---|
| `data` | A data.frame (or similar) with predicted and observed values. See the details section of `as_forecast()` for additional information on required input formats. |

### Value

The data.table with a column called `model`

`example_binary` *Binary forecast example data*

## Description

A data set with binary predictions for COVID-19 cases and deaths constructed from data submitted to the European Forecast Hub.

## Usage

```
example_binary
```

## Format

A data frame with the following columns:

**location** the country for which a prediction was made

**location_name** name of the country for which a prediction was made

**target_end_date** the date for which a prediction was made

**target_type** the target to be predicted (cases or deaths)

**observed** A factor with observed values

**forecast_date** the date on which a prediction was made

**model** name of the model that generated the forecasts

**horizon** forecast horizon in weeks

**predicted** predicted value

## Details

Predictions in the data set were constructed based on the continuous example data by looking at the number of samples below the mean prediction. The outcome was constructed as whether or not the actually observed value was below or above that mean prediction. This should not be understood as sound statistical practice, but rather as a practical way to create an example data set.

The data was created using the script create-example-data.R in the inst/ folder (or the top level folder in a compiled package).

## Source

https://github.com/european-modelling-hubs/covid19-forecast-hub-europe/commit/a42867b1ea152c57e25b04f9faa26cfd4bfd8fa6/

`example_point` *Point forecast example data*

## Description

A data set with predictions for COVID-19 cases and deaths submitted to the European Forecast Hub. This data set is like the quantile example data, only that the median has been replaced by a point forecast.

## Usage

```
example_point
```

## Format

A data frame with the following columns:

**location** the country for which a prediction was made

**target_end_date** the date for which a prediction was made

**target_type** the target to be predicted (cases or deaths)

**observed** observed values

**location_name** name of the country for which a prediction was made

**forecast_date** the date on which a prediction was made

**predicted** predicted value

**model** name of the model that generated the forecasts

**horizon** forecast horizon in weeks

## Details

The data was created using the script create-example-data.R in the inst/ folder (or the top level folder in a compiled package).

## Source

https://github.com/european-modelling-hubs/covid19-forecast-hub-europe/commit/a42867b1ea152c57e25b04f9faa26cfd4bfd8fa6/

`example_quantile` *Quantile example data*

## Description

A data set with predictions for COVID-19 cases and deaths submitted to the European Forecast Hub.

## Usage

```
example_quantile
```

## Format

A data frame with the following columns:

**location** the country for which a prediction was made

**target_end_date** the date for which a prediction was made

**target_type** the target to be predicted (cases or deaths)

**observed** Numeric: observed values

**location_name** name of the country for which a prediction was made

**forecast_date** the date on which a prediction was made

**quantile_level** quantile level of the corresponding prediction

**predicted** predicted value

**model** name of the model that generated the forecasts

**horizon** forecast horizon in weeks

## Details

The data was created using the script create-example-data.R in the inst/ folder (or the top level folder in a compiled package).

## Source

https://github.com/european-modelling-hubs/covid19-forecast-hub-europe/commit/a42867b1ea152c57e25b04f9faa26cfd4bfd8fa6/

`example_sample_continuous` *Continuous forecast example data*

## Description

A data set with continuous predictions for COVID-19 cases and deaths constructed from data submitted to the European Forecast Hub.

## Usage

```
example_sample_continuous
```

## Format

A data frame with the following columns:

**location** the country for which a prediction was made

**target_end_date** the date for which a prediction was made

**target_type** the target to be predicted (cases or deaths)

**observed** observed values

**location_name** name of the country for which a prediction was made

**forecast_date** the date on which a prediction was made

**model** name of the model that generated the forecasts

**horizon** forecast horizon in weeks

**predicted** predicted value

**sample_id** id for the corresponding sample

## Details

The data was created using the script create-example-data.R in the inst/ folder (or the top level folder in a compiled package).

## Source

https://github.com/european-modelling-hubs/covid19-forecast-hub-europe/commit/a42867b1ea152c57e25b04f9faa26cfd4bfd8fa6/

`example_sample_discrete` *Discrete forecast example data*

## Description

A data set with integer predictions for COVID-19 cases and deaths constructed from data submitted to the European Forecast Hub.

## Usage

```
example_sample_discrete
```

## Format

A data frame with the following columns:

**location** the country for which a prediction was made

**target_end_date** the date for which a prediction was made

**target_type** the target to be predicted (cases or deaths)

**observed** observed values

**location_name** name of the country for which a prediction was made

**forecast_date** the date on which a prediction was made

**model** name of the model that generated the forecasts

**horizon** forecast horizon in weeks

**predicted** predicted value

**sample_id** id for the corresponding sample

## Details

The data was created using the script create-example-data.R in the inst/ folder (or the top level folder in a compiled package).

## Source

https://github.com/european-modelling-hubs/covid19-forecast-hub-europe/commit/a42867b1ea152c57e25b04f9faa26cfd4bfd8fa6/

## get_correlations *Calculate correlation between metrics*

### Description

Calculate the correlation between different metrics for a data.frame of scores as produced by `score()`.

### Usage

```
get_correlations(scores, metrics = get_metrics(scores), ...)
```

### Arguments

| | |
|---|---|
| `scores` | An object of class `scores` (a data.table with scores and an additional attribute `metrics` as produced by `score()`). |
| `metrics` | A character vector with the metrics to show. If set to `NULL` (default), all metrics present in `scores` will be shown. |
| `...` | Additional arguments to pass down to `cor()`. |

### Value

An object of class `scores` (a data.table with an additional attribute `metrics` holding the names of the scores) with correlations between different metrics

### Examples

```
scores <- score(as_forecast(example_quantile))
get_correlations(scores)
```

## get_coverage *Get quantile and interval coverage values for quantile-based forecasts*

### Description

For a validated forecast object in a quantile-based format (see `as_forecast()` for more information), this function computes:

- interval coverage of central prediction intervals
- quantile coverage for predictive quantiles
- the deviation between desired and actual coverage (both for interval and quantile coverage)

Coverage values are computed for a specific level of grouping, as specified in the `by` argument. By default, coverage values are computed per model.

**Interval coverage**

Interval coverage for a given interval range is defined as the proportion of observations that fall within the corresponding central prediction intervals. Central prediction intervals are symmetric around the median and formed by two quantiles that denote the lower and upper bound. For example, the 50% central prediction interval is the interval between the 0.25 and 0.75 quantiles of the predictive distribution.

**Quantile coverage**

Quantile coverage for a given quantile level is defined as the proportion of observed values that are smaller than the corresponding predictive quantile. For example, the 0.5 quantile coverage is the proportion of observed values that are smaller than the 0.5 quantile of the predictive distribution. Just as above, for a single observation and the quantile of a single predictive distribution, the value will either be `TRUE` or `FALSE`.

**Coverage deviation**

The coverage deviation is the difference between the desired coverage (can be either interval or quantile coverage) and the actual coverage. For example, if the desired coverage is 90% and the actual coverage is 80%, the coverage deviation is -0.1.

## Usage

```
get_coverage(forecast, by = "model")
```

## Arguments

| | |
|---|---|
| `forecast` | A forecast object (a validated data.table with predicted and observed values, see `as_forecast()`) |
| `by` | character vector that denotes the level of grouping for which the coverage values should be computed. By default (`"model"`), one coverage value per model will be returned. |

## Value

A data.table with columns as specified in `by` and additional columns for the coverage values described above

a data.table with columns "interval_coverage", "interval_coverage_deviation", "quantile_coverage", "quantile_coverage_deviation" and the columns specified in `by`.

## Examples

```
library(magrittr) # pipe operator
example_quantile %>%
  as_forecast() %>%
  get_coverage(by = "model")
```

## get_duplicate_forecasts *Find duplicate forecasts*

### Description

Helper function to identify duplicate forecasts, i.e. instances where there is more than one forecast for the same prediction target.

### Usage

```
get_duplicate_forecasts(data, counts = FALSE)
```

### Arguments

| | |
|---|---|
| data | A data.frame as used for score() |
| counts | Should the output show the number of duplicates per forecast unit instead of the individual duplicated rows? Default is FALSE. |

### Value

A data.frame with all rows for which a duplicate forecast was found

### Examples

```
example <- rbind(example_quantile, example_quantile[1000:1010])
get_duplicate_forecasts(example)
```

## get_forecast_counts *Count number of available forecasts*

### Description

Given a data set with forecasts, this function counts the number of available forecasts. The level of grouping can be specified using the by argument (e.g. to count the number of forecasts per model, or the number of forecasts per model and location). This is useful to determine whether there are any missing forecasts.

### Usage

```
get_forecast_counts(
  forecast,
  by = get_forecast_unit(forecast),
  collapse = c("quantile_level", "sample_id")
)
```

### Arguments

| | |
|---|---|
| `forecast` | A forecast object (a validated data.table with predicted and observed values, see `as_forecast()`) |
| `by` | character vector or `NULL` (the default) that denotes the categories over which the number of forecasts should be counted. By default this will be the unit of a single forecast (i.e. all available columns (apart from a few "protected" columns such as 'predicted' and 'observed') plus "quantile_level" or "sample_id" where present). |
| `collapse` | character vector (default: `c("quantile_level", "sample_id")`) with names of categories for which the number of rows should be collapsed to one when counting. For example, a single forecast is usually represented by a set of several quantiles or samples and collapsing these to one makes sure that a single forecast only gets counted once. Setting `collapse = c()` would mean that all quantiles / samples would be counted as individual forecasts. |

### Value

A data.table with columns as specified in `by` and an additional column "count" with the number of forecasts.

### Examples

```
get_forecast_counts(
  as_forecast(example_quantile),
  by = c("model", "target_type")
)
```

---

## get_forecast_type *Infer forecast type from data*

---

### Description

Helper function to infer the forecast type based on a data.frame or similar with forecasts and observed values. See the details section below for information on the different forecast types.

### Usage

```
get_forecast_type(data)
```

### Arguments

| | |
|---|---|
| `data` | A data.frame (or similar) with predicted and observed values. See the details section of `as_forecast()` for additional information on required input formats. |

## Value

Character vector of length one with either "binary", "quantile", "sample" or "point".

## Forecast types and input formats

Various different forecast types / forecast formats are supported. At the moment, those are:

- point forecasts
- binary forecasts ("soft binary classification")
- Probabilistic forecasts in a quantile-based format (a forecast is represented as a set of predictive quantiles)
- Probabilistic forecasts in a sample-based format (a forecast is represented as a set of predictive samples)

Forecast types are determined based on the columns present in the input data. Here is an overview of the required format for each forecast type:

| Forecast type | | | column | type |
|---|---|---|---|---|
| All forecast types | | | `observed`<br>`predicted`<br>`model` | |
| Classification | Binary | Soft classification (prediction is probability) | `observed`<br>`predicted` | factor with 2 levels<br>numeric [0,1] |
| Point forecasts | Discrete, Continuous | | `observed`<br>`predicted` | numeric<br>numeric |
| Probabilistic forecast | Discrete, Continuous | Sample format | `observed`<br>`predicted`<br>`sample_id` | numeric<br>numeric<br>numeric |
| | | Quantile format | `observed`<br>`predicted`<br>`quantile_level` | numeric<br>numeric<br>numeric [0,1] |

*All forecast types* require a data.frame or similar with columns `observed` `predicted`, and `model`.

*Point forecasts* require a column `observed` of type numeric and a column `predicted` of type numeric.

*Binary forecasts* require a column `observed` of type factor with exactly two levels and a column `predicted` of type numeric with probabilities, corresponding to the probability that `observed` is equal to the second factor level. See details here for more information.

*Quantile-based forecasts* require a column `observed` of type numeric, a column `predicted` of type numeric, and a column `quantile_level` of type numeric with quantile-levels (between 0 and 1).

*Sample-based forecasts* require a column `observed` of type numeric, a column `predicted` of type numeric, and a column `sample_id` of type numeric with sample indices.

For more information see the vignettes and the example data (example_quantile, example_sample_continuous, example_sample_discrete, `example_point()`, and example_binary).

`get_forecast_unit` *Get unit of a single forecast*

## Description

Helper function to get the unit of a single forecast, i.e. the column names that define where a single forecast was made for. This just takes all columns that are available in the data and subtracts the columns that are protected, i.e. those returned by `get_protected_columns()` as well as the names of the metrics that were specified during scoring, if any.

## Usage

```
get_forecast_unit(data)
```

## Arguments

| | |
|---|---|
| `data` | A data.frame (or similar) with predicted and observed values. See the details section of `as_forecast()` for additional information on required input formats. |

## Value

A character vector with the column names that define the unit of a single forecast

## Forecast unit

In order to score forecasts, `scoringutils` needs to know which of the rows of the data belong together and jointly form a single forecasts. This is easy e.g. for point forecast, where there is one row per forecast. For quantile or sample-based forecasts, however, there are multiple rows that belong to a single forecast.

The *forecast unit* or *unit of a single forecast* is then described by the combination of columns that uniquely identify a single forecast. For example, we could have forecasts made by different models in various locations at different time points, each for several weeks into the future. The forecast unit could then be described as `forecast_unit = c("model", "location", "forecast_date", "forecast_horizon")`. `scoringutils` automatically tries to determine the unit of a single forecast. It uses all existing columns for this, which means that no columns must be present that are unrelated to the forecast unit. As a very simplistic example, if you had an additional row, "even", that is one if the row number is even and zero otherwise, then this would mess up scoring as `scoringutils` then thinks that this column was relevant in defining the forecast unit.

In order to avoid issues, we recommend setting the forecast unit explicitly, usually through the `forecast_unit` argument in `as_forecast()`. This will drop unneeded columns, while making sure that all necessary, 'protected columns' like "predicted" or "observed" are retained.

`get_metrics` *Get names of the metrics that were used for scoring*

## Description

When applying a scoring rule via `score()`, the names of the scoring rules become column names of the resulting data.table. In addition, an attribute `metrics` will be added to the output, holding the names of the scores as a vector.

This is done so that functions like `get_forecast_unit()` or `summarise_scores()` can still identify which columns are part of the forecast unit and which hold a score.

`get_metrics()` accesses and returns the `metrics` attribute. If there is no attribute, the function will return `NULL` (or, if `error = TRUE` will produce an error instead). In addition, it checks the column names of the input for consistency with the data stored in the `metrics` attribute.

**Handling a missing or inconsistent `metrics` attribute**:

If the metrics attribute is missing or is not consistent with the column names of the data.table, you can either

- run `score()` again, specifying names for the scoring rules manually, or
- add/update the attribute manually using `attr(scores, "metrics") <- c("names", "of", "your", "scores")` (the order does not matter).

## Usage

```
get_metrics(scores, error = FALSE)
```

## Arguments

| | |
|---|---|
| `scores` | A data.table with an attribute `metrics`. |
| `error` | Throw an error if there is no attribute called `metrics`? Default is FALSE. |

## Value

Character vector with the names of the scoring rules that were used for scoring or `NULL` if no scores were computed previously.

`get_pairwise_comparisons` *Obtain pairwise comparisons between models*

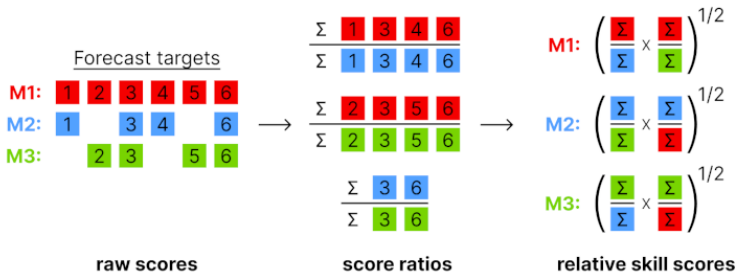
## Description

Compare scores obtained by different models in a pairwise tournament. All combinations of two models are compared against each other based on the overlapping set of available forecasts common to both models.

The input should be a `scores` object as produced by `score()`. Note that adding additional unrelated columns can unpredictably change results, as all present columns are taken into account when determining the set of overlapping forecasts between two models.

The output of the pairwise comparisons is a set of mean score ratios, relative skill scores and p-values.

The following illustrates the pairwise comparison process:




*Mean score ratios*

For every pair of two models, a mean score ratio is computed. This is simply the mean score of the first model divided by the mean score of the second. Mean score ratios are computed based on the set of overlapping forecasts between the two models. That means that only scores for those targets are taken into account for which both models have submitted a forecast.

*(Scaled) Relative skill scores*

The relative score of a model is the geometric mean of all mean score ratios which involve that model. If a baseline is provided, scaled relative skill scores will be calculated as well. Scaled relative skill scores are simply the relative skill score of a model divided by the relative skill score of the baseline model.

*p-values*

In addition, the function computes p-values for the comparison between two models (again based on the set of overlapping forecasts). P-values can be computed in two ways: based on a nonparametric Wilcoxon signed-rank test (internally using `wilcox.test()` with `paired = TRUE`) or based on a permutation test. The permutation test is based on the difference in mean scores between two models. The default null hypothesis is that the mean score

difference is zero (see permutation_test()). Adjusted p-values are computed by calling p.adjust() on the raw p-values.

The code for the pairwise comparisons is inspired by an implementation by Johannes Bracher. The implementation of the permutation test follows the function permutationTest from the surveillance package by Michael Höhle, Andrea Riebler and Michaela Paul.

## Usage

```
get_pairwise_comparisons(
  scores,
  by = "model",
  metric = intersect(c("wis", "crps", "brier_score"), names(scores)),
  baseline = NULL,
  ...
)
```

## Arguments

| | |
|---|---|
| scores | An object of class scores (a data.table with scores and an additional attribute metrics as produced by score()). |
| by | Character vector with column names that define the grouping level for the pairwise comparisons. By default (model), there will be one relative skill score per model. If, for example, by = c("model", "location"). Then you will get a separate relative skill score for every model in every location. Internally, the data.table with scores will be split according by (removing "model" before splitting) and the pairwise comparisons will be computed separately for the split data.tables. |
| metric | A string with the name of the metric for which a relative skill shall be computed. By default this is either "crps", "wis" or "brier_score" if any of these are available. |
| baseline | A string with the name of a model. If a baseline is given, then a scaled relative skill with respect to the baseline will be returned. By default (NULL), relative skill will not be scaled with respect to a baseline model. |
| ... | Additional arguments for the comparison between two models. See compare_two_models() for more information. |

## Value

A data.table with the results of pairwise comparisons containing the mean score ratios (mean_scores_ratio), unadjusted (pval) and adjusted (adj_pval) p-values, and relative skill values of each model (..._relative_skill). If a baseline model is given then the scaled relative skill is reported as well (..._scaled_relative_skill).

## Author(s)

Nikos Bosse <nikosbosse@gmail.com>

Johannes Bracher, <johannes.bracher@kit.edu>

### Examples

```
scores <- score(as_forecast(example_quantile))
pairwise <- get_pairwise_comparisons(scores, by = "target_type")
pairwise2 <- get_pairwise_comparisons(
  scores, by = "target_type", baseline = "EuroCOVIDhub-baseline"
)

library(ggplot2)
plot_pairwise_comparisons(pairwise, type = "mean_scores_ratio") +
  facet_wrap(~target_type)
```

---

## get_pit *Probability integral transformation (data.frame version)*

---

### Description

Compute the Probability Integral Transformation (PIT) for validated forecast objects.

### Usage

```
get_pit(forecast, by, n_replicates = 100)
```

### Arguments

| | |
|---|---|
| `forecast` | A forecast object (a validated data.table with predicted and observed values, see `as_forecast()`) |
| `by` | Character vector with the columns according to which the PIT values shall be grouped. If you e.g. have the columns 'model' and 'location' in the input data and want to have a PIT histogram for every model and location, specify `by = c("model", "location")`. |
| `n_replicates` | The number of draws for the randomised PIT for discrete predictions. Will be ignored if forecasts are continuous. |

### Value

A data.table with PIT values according to the grouping specified in `by`.

### References

Sebastian Funk, Anton Camacho, Adam J. Kucharski, Rachel Lowe, Rosalind M. Eggo, W. John Edmunds (2019) Assessing the performance of real-time epidemic forecasts: A case study of Ebola in the Western Area region of Sierra Leone, 2014-15, doi:10.1371/journal.pcbi.1006785

### Examples

```
result <- get_pit(as_forecast(example_sample_continuous), by = "model")
plot_pit(result)

# example with quantile data
result <- get_pit(as_forecast(example_quantile), by = "model")
plot_pit(result)
```

---

## get_type *Get type of a vector or matrix of observed values or predictions*

---

### Description

Internal helper function to get the type of a vector (usually of observed or predicted values). The function checks whether the input is a factor, or else whether it is integer (or can be coerced to integer) or whether it's continuous.

### Usage

```
get_type(x)
```

### Arguments

| | |
|---|---|
| x | Input the type should be determined for. |

### Value

Character vector of length one with either "classification", "integer", or "continuous".

---

## interval_coverage *Interval coverage (for quantile-based forecasts)*

---

### Description

Check whether the observed value is within a given central prediction interval. The prediction interval is defined by a lower and an upper bound formed by a pair of predictive quantiles. For example, a 50% prediction interval is formed by the 0.25 and 0.75 quantiles of the predictive distribution.

### Usage

```
interval_coverage(observed, predicted, quantile_level, interval_range = 50)
```

### Arguments

| | |
|---|---|
| `observed` | Numeric vector of size n with the observed values. |
| `predicted` | Numeric nxN matrix of predictive quantiles, n (number of rows) being the number of forecasts (corresponding to the number of observed values) and N (number of columns) the number of quantiles per forecast. If `observed` is just a single number, then predicted can just be a vector of size N. |
| `quantile_level` | Vector of of size N with the quantile levels for which predictions were made. |
| `interval_range` | A single number with the range of the prediction interval in percent (e.g. 50 for a 50% prediction interval) for which you want to compute interval coverage. |

### Value

A vector of length n with elements either TRUE, if the observed value is within the corresponding prediction interval, and FALSE otherwise.

### Examples

```
observed <- c(1, -15, 22)
predicted <- rbind(
  c(-1, 0, 1, 2, 3),
  c(-2, 1, 2, 2, 4),
   c(-2, 0, 3, 3, 4)
)
quantile_level <- c(0.1, 0.25, 0.5, 0.75, 0.9)
interval_coverage(observed, predicted, quantile_level)
```

---

## `interval_coverage_deviation` *Interval coverage deviation (for quantile-based forecasts)*

---

### Description

Check the agreement between desired and actual interval coverage of a forecast.

The function is similar to `interval_coverage()`, but takes all provided prediction intervals into account and compares nominal interval coverage (i.e. the desired interval coverage) with the actual observed interval coverage.

A central symmetric prediction interval is defined by a lower and an upper bound formed by a pair of predictive quantiles. For example, a 50% prediction interval is formed by the 0.25 and 0.75 quantiles of the predictive distribution. Ideally, a forecaster should aim to cover about 50% of all observed values with their 50% prediction intervals, 90% of all observed values with their 90% prediction intervals, and so on.

For every prediction interval, the deviation is computed as the difference between the observed interval coverage and the nominal interval coverage For a single observed value and a single prediction interval, coverage is always either 0 or 1 (`FALSE` or `TRUE`). This is not the case for a single observed value and multiple prediction intervals, but it still doesn't make that much sense to compare nominal (desired) coverage and actual coverage for a single observation. In that sense coverage deviation only really starts to make sense as a metric when averaged across multiple observations).

Positive values of interval coverage deviation are an indication for underconfidence, i.e. the forecaster could likely have issued a narrower forecast. Negative values are an indication for overconfidence, i.e. the forecasts were too narrow.

$$\text{interval coverage deviation} = \mathbf{1}(\text{observed value falls within interval}) - \text{nominal interval coverage}$$

The interval coverage deviation is then averaged across all prediction intervals. The median is ignored when computing coverage deviation.

## Usage

```
interval_coverage_deviation(observed, predicted, quantile_level)
```

## Arguments

| | |
|---|---|
| `observed` | Numeric vector of size n with the observed values. |
| `predicted` | Numeric nxN matrix of predictive quantiles, n (number of rows) being the number of forecasts (corresponding to the number of observed values) and N (number of columns) the number of quantiles per forecast. If `observed` is just a single number, then predicted can just be a vector of size N. |
| `quantile_level` | Vector of of size N with the quantile levels for which predictions were made. |

## Value

A numeric vector of length n with the interval coverage deviation for each forecast (with the forecast itself comprising one or multiple prediction intervals).

## Examples

```
observed <- c(1, -15, 22)
predicted <- rbind(
  c(-1, 0, 1, 2, 3),
  c(-2, 1, 2, 2, 4),
  c(-2, 0, 3, 3, 4)
)
quantile_level <- c(0.1, 0.25, 0.5, 0.75, 0.9)
interval_coverage_deviation(observed, predicted, quantile_level)
```

`interval_score` *Interval score*

## Description

Proper Scoring Rule to score quantile predictions, following Gneiting and Raftery (2007). Smaller values are better.

The score is computed as

$$\text{score} = (\text{upper} - \text{lower}) + \frac{2}{\alpha}(\text{lower} - \text{observed}) * \mathbf{1}(\text{observed} < \text{lower}) + \frac{2}{\alpha}(\text{observed} - \text{upper}) * \mathbf{1}(\text{observed} > \text{upper})$$

where $\mathbf{1}()$ is the indicator function and indicates how much is outside the prediction interval. $\alpha$ is the decimal value that indicates how much is outside the prediction interval.

To improve usability, the user is asked to provide an interval range in percentage terms, i.e. interval_range = 90 (percent) for a 90 percent prediction interval. Correspondingly, the user would have to provide the 5% and 95% quantiles (the corresponding alpha would then be 0.1). No specific distribution is assumed, but the interval has to be symmetric around the median (i.e you can't use the 0.1 quantile as the lower bound and the 0.7 quantile as the upper bound). Non-symmetric quantiles can be scored using the function `quantile_score()`.

## Usage

```
interval_score(
  observed,
  lower,
  upper,
  interval_range,
  weigh = TRUE,
  separate_results = FALSE
)
```

## Arguments

| | |
|---|---|
| `observed` | A vector with observed values of size n |
| `lower` | Vector of size n with the prediction for the lower quantile of the given interval range. |
| `upper` | Vector of size n with the prediction for the upper quantile of the given interval range. |
| `interval_range` | Numeric vector (either a single number or a vector of size n) with the range of the prediction intervals. For example, if you're forecasting the 0.05 and 0.95 quantile, the interval range would be 90. The interval range corresponds to $(100 - \alpha)/100$, where $\alpha$ is the decimal value that indicates how much is outside the prediction interval (see e.g. Gneiting and Raftery (2007)). |

`weigh` Logical. If `TRUE` (the default), weigh the score by $\alpha/2$, so it can be averaged into an interval score that, in the limit (for an increasing number of equally spaced quantiles/prediction intervals), corresponds to the CRPS. $\alpha$ is the value that corresponds to the $(\alpha/2)$ or $(1 - \alpha/2)$, i.e. it is the decimal value that represents how much is outside a central prediction interval (E.g. for a 90 percent central prediction interval, alpha is 0.1).

`separate_results` Logical. If `TRUE` (default is `FALSE`), then the separate parts of the interval score (dispersion penalty, penalties for over- and under-prediction get returned as separate elements of a list). If you want a `data.frame` instead, simply call `as.data.frame()` on the output.

## Value

Vector with the scoring values, or a list with separate entries if `separate_results` is `TRUE`.

## References

Strictly Proper Scoring Rules, Prediction,and Estimation, Tilmann Gneiting and Adrian E. Raftery, 2007, Journal of the American Statistical Association, Volume 102, 2007 - Issue 477

Evaluating epidemic forecasts in an interval format, Johannes Bracher, Evan L. Ray, Tilmann Gneiting and Nicholas G. Reich, https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1008618 # nolint

## Examples

```
observed <- rnorm(30, mean = 1:30)
interval_range <- rep(90, 30)
alpha <- (100 - interval_range) / 100
lower <- qnorm(alpha / 2, rnorm(30, mean = 1:30))
upper <- qnorm((1 - alpha / 2), rnorm(30, mean = 11:40))

scoringutils:::interval_score(
  observed = observed,
  lower = lower,
  upper = upper,
  interval_range = interval_range
)

# gives a warning, as the interval_range should likely be 50 instead of 0.5
scoringutils:::interval_score(
  observed = 4, upper = 8, lower = 2, interval_range = 0.5
)

# example with missing values and separate results
scoringutils:::interval_score(
  observed = c(observed, NA),
  lower = c(lower, NA),
  upper = c(NA, upper),
  separate_results = TRUE,
```

```
  interval_range = 90
)
```

---

`is_forecast` *Test whether an object is a forecast object*

---

## Description

Test whether an object is a forecast object (see `as_forecast()` for more information).

You can test for a specific `forecast_*` class using the appropriate `is_forecast_*` function.

## Usage

```
is_forecast(x)

is_forecast_sample(x)

is_forecast_binary(x)

is_forecast_point(x)

is_forecast_quantile(x)
```

## Arguments

| | |
|---|---|
| `x` | An R object. |

## Value

`is_forecast`: `TRUE` if the object is of class `forecast`, `FALSE` otherwise.

`is_forecast_*`: `TRUE` if the object is of class `forecast_*` in addition to class `forecast`, `FALSE` otherwise.

## Examples

```
forecast_binary <- as_forecast(example_binary)
is_forecast(forecast_binary)
```

`logs_sample` *Logarithmic score (sample-based version)*

## Description

This function is a wrapper around the `logs_sample()` function from the **scoringRules** package.

The function should be used to score continuous predictions only. While the Log Score is in theory also applicable to discrete forecasts, the problem lies in the implementation: The Log score needs a kernel density estimation, which is not well defined with integer-valued Monte Carlo Samples. The Log score can be used for specific discrete probability distributions. See the scoringRules package for more details.

## Usage

```
logs_sample(observed, predicted, ...)
```

## Arguments

| | |
|---|---|
| `observed` | A vector with observed values of size n |
| `predicted` | nxN matrix of predictive samples, n (number of rows) being the number of data points and N (number of columns) the number of Monte Carlo samples. Alternatively, `predicted` can just be a vector of size n. |
| `...` | Additional arguments passed to logs_sample() from the scoringRules package. |

## Value

Vector with scores.

## References

Alexander Jordan, Fabian Krüger, Sebastian Lerch, Evaluating Probabilistic Forecasts with scoringRules, https://www.jstatsoft.org/article/view/v090i12

## Examples

```
observed <- rpois(30, lambda = 1:30)
predicted <- replicate(200, rpois(n = 30, lambda = 1:30))
logs_sample(observed, predicted)
```

`log_shift` *Log transformation with an additive shift*

## Description

Function that shifts a value by some offset and then applies the natural logarithm to it.

## Usage

```
log_shift(x, offset = 0, base = exp(1))
```

## Arguments

| | |
|---|---|
| `x` | vector of input values to be transformed |
| `offset` | Number to add to the input value before taking the natural logarithm. |
| `base` | A positive number: the base with respect to which logarithms are computed. Defaults to e = exp(1). |

## Details

The output is computed as log(x + offset)

## Value

A numeric vector with transformed values

## References

Transformation of forecasts for evaluating predictive performance in an epidemiological context Nikos I. Bosse, Sam Abbott, Anne Cori, Edwin van Leeuwen, Johannes Bracher, Sebastian Funk medRxiv 2023.01.23.23284722 doi:10.1101/2023.01.23.23284722 https://www.medrxiv.org/content/10.1101/2023.01.23.23284722v1 # nolint

## Examples

```
log_shift(1:10)
log_shift(0:9, offset = 1)

transform_forecasts(
  as_forecast(example_quantile)[observed > 0, ],
  fun = log_shift,
  offset = 1
 )
```

`mad_sample` *Determine dispersion of a probabilistic forecast*

## Description

Sharpness is the ability of the model to generate predictions within a narrow range and dispersion is the lack thereof. It is a data-independent measure, and is purely a feature of the forecasts themselves.

Dispersion of predictive samples corresponding to one single observed value is measured as the normalised median of the absolute deviation from the median of the predictive samples. For details, see mad() and the explanations given in Funk et al. (2019)

## Usage

```
mad_sample(observed = NULL, predicted, ...)
```

## Arguments

| | |
|---|---|
| `observed` | Place holder, argument will be ignored and exists only for consistency with other scoring functions. The output does not depend on any observed values. |
| `predicted` | nxN matrix of predictive samples, n (number of rows) being the number of data points and N (number of columns) the number of Monte Carlo samples. Alternatively, `predicted` can just be a vector of size n. |
| `...` | Additional arguments passed to mad(). |

## Value

Vector with dispersion values.

## References

Funk S, Camacho A, Kucharski AJ, Lowe R, Eggo RM, Edmunds WJ (2019) Assessing the performance of real-time epidemic forecasts: A case study of Ebola in the Western Area region of Sierra Leone, 2014-15. PLoS Comput Biol 15(2): e1006785. doi:10.1371/journal.pcbi.1006785

## Examples

```
predicted <- replicate(200, rpois(n = 30, lambda = 1:30))
mad_sample(predicted = predicted)
```

## metrics_binary *Default metrics and scoring rules for binary forecasts*

### Description

Helper function that returns a named list of default scoring rules suitable for binary forecasts.

The default scoring rules are:

- "brier_score" = `brier_score()`
- "log_score" = `logs_binary()`

### Usage

```
metrics_binary(select = NULL, exclude = NULL)
```

### Arguments

| | |
|---|---|
| `select` | A character vector of scoring rules to select from the list. If `select` is `NULL` (the default), all possible scoring rules are returned. |
| `exclude` | A character vector of scoring rules to exclude from the list. If `select` is not `NULL`, this argument is ignored. |

### Value

A list of scoring rules.

### Examples

```
metrics_binary()
metrics_binary(select = "brier_score")
metrics_binary(exclude = "log_score")
```

## metrics_point *Default metrics and scoring rules for point forecasts*

### Description

Helper function that returns a named list of default scoring rules suitable for point forecasts.

The default scoring rules are:

- "ae_point" = ae()
- "se_point" = se()
- "ape" = ape()

### Usage

```
metrics_point(select = NULL, exclude = NULL)
```

### Arguments

| | |
|---|---|
| `select` | A character vector of scoring rules to select from the list. If `select` is `NULL` (the default), all possible scoring rules are returned. |
| `exclude` | A character vector of scoring rules to exclude from the list. If `select` is not `NULL`, this argument is ignored. |

### Value

A list of scoring rules.

### Examples

```
metrics_point()
metrics_point(select = "ape")
```

---

## metrics_quantile *Default metrics and scoring rules for quantile-based forecasts*

---

### Description

Helper function that returns a named list of default scoring rules suitable for forecasts in a quantile-based format.

The default scoring rules are:

- "wis" = wis
- "overprediction" = `overprediction()`
- "underprediction" = `underprediction()`
- "dispersion" = `dispersion()`
- "bias" = `bias_quantile()`
- "interval_coverage_50" = `interval_coverage()`
- "interval_coverage_90" = customise_metric( interval_coverage, interval_range = 90 )
- "interval_coverage_deviation" = `interval_coverage_deviation()`,
- "ae_median" = `ae_median_quantile()`

Note: The `interval_coverage_90` scoring rule is created by modifying `interval_coverage()`, making use of the function `customise_metric()`. This construct allows the function to deal with arbitrary arguments in `...`, while making sure that only those that `interval_coverage()` can accept get passed on to it. `interval_range = 90` is set in the function definition, as passing an argument `interval_range = 90` to `score()` would mean it would also get passed to `interval_coverage_50`.

### Usage

```
metrics_quantile(select = NULL, exclude = NULL)
```

### Arguments

| | |
|---|---|
| `select` | A character vector of scoring rules to select from the list. If `select` is `NULL` (the default), all possible scoring rules are returned. |
| `exclude` | A character vector of scoring rules to exclude from the list. If `select` is not `NULL`, this argument is ignored. |

### Value

A list of scoring rules.

### Examples

```
metrics_quantile()
metrics_quantile(select = "wis")
```

---

## `metrics_sample` *Default metrics and scoring rules sample-based forecasts*

---

### Description

Helper function that returns a named list of default scoring rules suitable for forecasts in a sample-based format.

The default scoring rules are:

- "crps" = `crps_sample()`
- "log_score" = `logs_sample()`
- "dss" = `dss_sample()`
- "mad" = `mad_sample()`
- "bias" = `bias_sample()`
- "ae_median" = `ae_median_sample()`
- "se_mean" = `se_mean_sample()`

### Usage

```
metrics_sample(select = NULL, exclude = NULL)
```

### Arguments

| | |
|---|---|
| `select` | A character vector of scoring rules to select from the list. If `select` is `NULL` (the default), all possible scoring rules are returned. |
| `exclude` | A character vector of scoring rules to exclude from the list. If `select` is not `NULL`, this argument is ignored. |

### Value

A list of scoring rules.

### Examples

```
metrics_sample()
metrics_sample(select = "mad")
```

---

`pit_sample` *Probability integral transformation (sample-based version)*

---

### Description

Uses a Probability integral transformation (PIT) (or a randomised PIT for integer forecasts) to assess the calibration of predictive Monte Carlo samples.

### Usage

```
pit_sample(observed, predicted, n_replicates = 100)
```

### Arguments

| | |
|---|---|
| `observed` | A vector with observed values of size n |
| `predicted` | nxN matrix of predictive samples, n (number of rows) being the number of data points and N (number of columns) the number of Monte Carlo samples. Alternatively, `predicted` can just be a vector of size n. |
| `n_replicates` | The number of draws for the randomised PIT for discrete predictions. Will be ignored if forecasts are continuous. |

### Details

Calibration or reliability of forecasts is the ability of a model to correctly identify its own uncertainty in making predictions. In a model with perfect calibration, the observed data at each time point look as if they came from the predictive probability distribution at that time.

Equivalently, one can inspect the probability integral transform of the predictive distribution at time t,

$$u_t = F_t(x_t)$$

where $x_t$ is the observed data point at time $t$ in $t_1, , t_n$, n being the number of forecasts, and $F_t$ is the (continuous) predictive cumulative probability distribution at time t. If the true probability distribution of outcomes at time t is $G_t$ then the forecasts $F_t$ are said to be ideal if $F_t = G_t$ at all times t. In that case, the probabilities $u_t$ are distributed uniformly.

In the case of discrete outcomes such as incidence counts, the PIT is no longer uniform even when forecasts are ideal. In that case a randomised PIT can be used instead:

$$u_t = P_t(k_t) + v * (P_t(k_t) - P_t(k_t - 1))$$

where $k_t$ is the observed count, $P_t(x)$ is the predictive cumulative probability of observing incidence k at time t, $P_t(-1) = 0$ by definition and v is standard uniform and independent of k. If $P_t$ is the true cumulative probability distribution, then $u_t$ is standard uniform.

The function checks whether integer or continuous forecasts were provided. It then applies the (randomised) probability integral and tests the values $u_t$ for uniformity using the Anderson-Darling test.

As a rule of thumb, there is no evidence to suggest a forecasting model is miscalibrated if the p-value found was greater than a threshold of p >= 0.1, some evidence that it was miscalibrated if 0.01 < p < 0.1, and good evidence that it was miscalibrated if p <= 0.01. However, the AD-p-values may be overly strict and there actual usefulness may be questionable. In this context it should be noted, though, that uniformity of the PIT is a necessary but not sufficient condition of calibration.

## Value

A vector with PIT-values. For continuous forecasts, the vector will correspond to the length of `observed`. For integer forecasts, a randomised PIT will be returned of length `length(observed) * n_replicates`.

## References

Sebastian Funk, Anton Camacho, Adam J. Kucharski, Rachel Lowe, Rosalind M. Eggo, W. John Edmunds (2019) Assessing the performance of real-time epidemic forecasts: A case study of Ebola in the Western Area region of Sierra Leone, 2014-15, doi:10.1371/journal.pcbi.1006785

## See Also

`get_pit()`

## Examples

```
## continuous predictions
observed <- rnorm(20, mean = 1:20)
predicted <- replicate(100, rnorm(n = 20, mean = 1:20))
pit <- pit_sample(observed, predicted)
plot_pit(pit)

## integer predictions
observed <- rpois(20, lambda = 1:20)
predicted <- replicate(100, rpois(n = 20, lambda = 1:20))
pit <- pit_sample(observed, predicted, n_replicates = 30)
plot_pit(pit)
```

## plot_correlations *Plot correlation between metrics*

### Description

Plots a heatmap of correlations between different metrics.

### Usage

```
plot_correlations(correlations, digits = NULL)
```

### Arguments

| | |
|---|---|
| `correlations` | A data.table of correlations between scores as produced by `get_correlations()`. |
| `digits` | A number indicating how many decimal places the correlations should be rounded to. By default (`digits = NULL`) no rounding takes place. |

### Value

A ggplot object showing a coloured matrix of correlations between metrics.

A ggplot object with a visualisation of correlations between metrics

### Examples

```
scores <- score(as_forecast(example_quantile))
correlations <- get_correlations(
 summarise_scores(scores)
)
plot_correlations(correlations, digits = 2)
```

## plot_forecast_counts *Visualise the number of available forecasts*

### Description

Visualise Where Forecasts Are Available.

### Usage

```
plot_forecast_counts(
  forecast_counts,
  x,
  y = "model",
  x_as_factor = TRUE,
  show_counts = TRUE
)
```

### Arguments

| | |
|---|---|
| forecast_counts | A data.table (or similar) with a column `count` holding forecast counts, as produced by `get_forecast_counts()`. |
| x | Character vector of length one that denotes the name of the column to appear on the x-axis of the plot. |
| y | Character vector of length one that denotes the name of the column to appear on the y-axis of the plot. Default is "model". |
| x_as_factor | Logical (default is `TRUE`). Whether or not to convert the variable on the x-axis to a factor. This has an effect e.g. if dates are shown on the x-axis. |
| show_counts | Logical (default is `TRUE`) that indicates whether or not to show the actual count numbers on the plot. |

### Value

A ggplot object with a plot of forecast counts

### Examples

```
library(ggplot2)
forecast_counts <- get_forecast_counts(
  as_forecast(example_quantile),
  by = c("model", "target_type", "target_end_date")
)
plot_forecast_counts(
 forecast_counts, x = "target_end_date", show_counts = FALSE
) +
 facet_wrap("target_type")
```

---

## plot_heatmap *Create a heatmap of a scoring metric*

---

### Description

This function can be used to create a heatmap of one metric across different groups, e.g. the interval score obtained by several forecasting models in different locations.

### Usage

```
plot_heatmap(scores, y = "model", x, metric)
```

### Arguments

| | |
|---|---|
| scores | A data.frame of scores based on quantile forecasts as produced by `score()`. |
| y | The variable from the scores you want to show on the y-Axis. The default for this is "model" |

| | |
|---|---|
| x | The variable from the scores you want to show on the x-Axis. This could be something like "horizon", or "location" |
| metric | String, the metric that determines the value and colour shown in the tiles of the heatmap. |

### Value

A ggplot object showing a heatmap of the desired metric

### Examples

```
scores <- score(as_forecast(example_quantile))
scores <- summarise_scores(scores, by = c("model", "target_type"))
scores <- summarise_scores(
  scores, by = c("model", "target_type"),
  fun = signif, digits = 2
)

plot_heatmap(scores, x = "target_type", metric = "bias")
```

---

## plot_interval_coverage

*Plot interval coverage*

---

### Description

Plot interval coverage values (see `get_coverage()` for more information).

### Usage

```
plot_interval_coverage(coverage, colour = "model")
```

### Arguments

| | |
|---|---|
| coverage | A data frame of coverage values as produced by `get_coverage()`. |
| colour | According to which variable shall the graphs be coloured? Default is "model". |

### Value

ggplot object with a plot of interval coverage

### Examples

```
coverage <- get_coverage(as_forecast(example_quantile), by = "model")
plot_interval_coverage(coverage)
```

## plot_pairwise_comparisons *Plot heatmap of pairwise comparisons*

### Description

Creates a heatmap of the ratios or pvalues from a pairwise comparison between models.

### Usage

```
plot_pairwise_comparisons(
  comparison_result,
  type = c("mean_scores_ratio", "pval")
)
```

### Arguments

| | |
|---|---|
| comparison_result | A data.frame as produced by get_pairwise_comparisons(). |
| type | Character vector of length one that is either "mean_scores_ratio" or "pval". This denotes whether to visualise the ratio or the p-value of the pairwise comparison. Default is "mean_scores_ratio". |

### Value

A ggplot object with a heatmap of mean score ratios from pairwise comparisons.

### Examples

```
library(ggplot2)
scores <- score(as_forecast(example_quantile))
pairwise <- get_pairwise_comparisons(scores, by = "target_type")
plot_pairwise_comparisons(pairwise, type = "mean_scores_ratio") +
  facet_wrap(~target_type)
```

## plot_pit *PIT histogram*

### Description

Make a simple histogram of the probability integral transformed values to visually check whether a uniform distribution seems likely.

### Usage

```
plot_pit(pit, num_bins = "auto", breaks = NULL)
```

### Arguments

| | |
|---|---|
| pit | Either a vector with the PIT values, or a data.table as produced by get_pit(). |
| num_bins | The number of bins in the PIT histogram, default is "auto". When num_bins == "auto", plot_pit() will either display 10 bins, or it will display a bin for each available quantile in case you passed in data in a quantile-based format. You can control the number of bins by supplying a number. This is fine for sample-based pit histograms, but may fail for quantile-based formats. In this case it is preferred to supply explicit breaks points using the breaks argument. |
| breaks | Numeric vector with the break points for the bins in the PIT histogram. This is preferred when creating a PIT histogram based on quantile-based data. Default is NULL and breaks will be determined by num_bins. If breaks is used, num_bins will be ignored. |

### Value

A ggplot object with a histogram of PIT values

### Examples

```
# PIT histogram in vector based format
observed <- rnorm(30, mean = 1:30)
predicted <- replicate(200, rnorm(n = 30, mean = 1:30))
pit <- pit_sample(observed, predicted)
plot_pit(pit)

# quantile-based pit
pit <- get_pit(as_forecast(example_quantile), by = "model")
plot_pit(pit, breaks = seq(0.1, 1, 0.1))

# sample-based pit
pit <- get_pit(as_forecast(example_sample_discrete), by = "model")
plot_pit(pit)
```

---

## plot_quantile_coverage

*Plot quantile coverage*

---

### Description

Plot quantile coverage values (see get_coverage() for more information).

### Usage

```
plot_quantile_coverage(coverage, colour = "model")
```

### Arguments

| | |
|---|---|
| coverage | A data frame of coverage values as produced by get_coverage(). |
| colour | String, according to which variable shall the graphs be coloured? Default is "model". |

### Value

A ggplot object with a plot of interval coverage

### Examples

```
coverage <- get_coverage(as_forecast(example_quantile), by = "model")
plot_quantile_coverage(coverage)
```

---

## plot_wis *Plot contributions to the weighted interval score*

---

### Description

Visualise the components of the weighted interval score: penalties for over-prediction, under-prediction and for high dispersion (lack of sharpness).

### Usage

```
plot_wis(scores, x = "model", relative_contributions = FALSE, flip = FALSE)
```

### Arguments

| | |
|---|---|
| scores | A data.table of scores based on quantile forecasts as produced by score() and summarised using summarise_scores(). |
| x | The variable from the scores you want to show on the x-Axis. Usually this will be "model". |
| relative_contributions | Logical. Show relative contributions instead of absolute contributions? Default is FALSE and this functionality is not available yet. |
| flip | Boolean (default is FALSE), whether or not to flip the axes. |

### Value

A ggplot object showing a contributions from the three components of the weighted interval score.

A ggplot object with a visualisation of the WIS decomposition

### References

Bracher J, Ray E, Gneiting T, Reich, N (2020) Evaluating epidemic forecasts in an interval format. https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1008618

### Examples

```
library(ggplot2)
scores <- score(as_forecast(example_quantile))
scores <- summarise_scores(scores, by = c("model", "target_type"))

plot_wis(scores,
  x = "model",
  relative_contributions = TRUE
) +
  facet_wrap(~target_type)
plot_wis(scores,
  x = "model",
  relative_contributions = FALSE
) +
  facet_wrap(~target_type, scales = "free_x")
```

---

## print.forecast *Print information about a forecast object*

---

### Description

This function prints information about a forecast object, including "Forecast type", "Score columns", "Forecast unit".

### Usage

```
## S3 method for class 'forecast'
print(x, ...)
```

### Arguments

| | |
|---|---|
| x | A forecast object (a validated data.table with predicted and observed values, see as_forecast()). |
| ... | Additional arguments for print(). |

### Value

Returns x invisibly.

### Examples

```
dat <- as_forecast(example_quantile)
print(dat)
```

`quantile_score` *Quantile score*

## Description

Proper Scoring Rule to score quantile predictions. Smaller values are better. The quantile score is closely related to the interval score (see `wis()`) and is the quantile equivalent that works with single quantiles instead of central prediction intervals.

The quantile score, also called pinball loss, for a single quantile level $\tau$ is defined as

$$\mathrm{QS}_\tau(F, y) = 2 \cdot \{\mathbf{1}(y \leq q_\tau) - \tau\} \cdot (q_\tau y) = \begin{cases} 2 \cdot (1 - \tau) * q_\tau - y, & \text{if } y \leq q_\tau \\ 2 \cdot \tau * |q_\tau - y|, & \text{if } y > q_\tau, \end{cases}$$

with $q_\tau$ being the $\tau$-quantile of the predictive distribution $F$, and $\mathbf{1}(\cdot)$ the indicator function.

The weighted interval score for a single prediction interval can be obtained as the average of the quantile scores for the lower and upper quantile of that prediction interval:

$$\mathrm{WIS}_\alpha(F, y) = \frac{\mathrm{QS}_{\alpha/2}(F, y) + \mathrm{QS}_{1-\alpha/2}(F, y)}{2}.$$

See the SI of Bracher et al. (2021) for more details.

`quantile_score()` returns the average quantile score across the quantile levels provided. For a set of quantile levels that form pairwise central prediction intervals, the quantile score is equivalent to the interval score.

## Usage

```
quantile_score(observed, predicted, quantile_level, weigh = TRUE)
```

## Arguments

| | |
|---|---|
| `observed` | Numeric vector of size n with the observed values. |
| `predicted` | Numeric nxN matrix of predictive quantiles, n (number of rows) being the number of forecasts (corresponding to the number of observed values) and N (number of columns) the number of quantiles per forecast. If `observed` is just a single number, then predicted can just be a vector of size N. |
| `quantile_level` | Vector of of size N with the quantile levels for which predictions were made. |
| `weigh` | Logical. If `TRUE` (the default), weigh the score by $\alpha/2$, so it can be averaged into an interval score that, in the limit (for an increasing number of equally spaced quantiles/prediction intervals), corresponds to the CRPS. $\alpha$ is the value that corresponds to the $(\alpha/2)$ or $(1 - \alpha/2)$, i.e. it is the decimal value that represents how much is outside a central prediction interval (E.g. for a 90 percent central prediction interval, alpha is 0.1). |

## Value

Numeric vector of length n with the quantile score. The scores are averaged across quantile levels if multiple quantile levels are provided (the result of calling `rowMeans()` on the matrix of quantile scores that is computed based on the observed and predicted values).

## References

Strictly Proper Scoring Rules, Prediction,and Estimation, Tilmann Gneiting and Adrian E. Raftery, 2007, Journal of the American Statistical Association, Volume 102, 2007 - Issue 477

Evaluating epidemic forecasts in an interval format, Johannes Bracher, Evan L. Ray, Tilmann Gneiting and Nicholas G. Reich, 2021, https://journals.plos.org/ploscompbiol/article?id=10.1371/journal.pcbi.1008618

## Examples

```
observed <- rnorm(10, mean = 1:10)
alpha <- 0.5

lower <- qnorm(alpha / 2, observed)
upper <- qnorm((1 - alpha / 2), observed)

qs_lower <- quantile_score(observed,
  predicted = matrix(lower),
  quantile_level = alpha / 2
)
qs_upper <- quantile_score(observed,
  predicted = matrix(upper),
  quantile_level = 1 - alpha / 2
)
interval_score <- (qs_lower + qs_upper) / 2
interval_score2 <- quantile_score(
  observed,
  predicted = cbind(lower, upper),
  quantile_level = c(alpha / 2, 1 - alpha / 2)
)

# this is the same as the following
wis(
  observed,
  predicted = cbind(lower, upper),
  quantile_level = c(alpha / 2, 1 - alpha / 2)
)
```

---

`run_safely` *Run a function safely*

---

### Description

This is a wrapper function designed to run a function safely when it is not completely clear what arguments could be passed to the function.

All named arguments in `...` that are not accepted by `fun` are removed. All unnamed arguments are passed on to the function. In case `fun` errors, the error will be converted to a warning and `run_safely` returns `NULL`.

`run_safely` can be useful when constructing functions to be used as metrics in `score()`.

### Usage

```
run_safely(..., fun, metric_name)
```

### Arguments

| | |
|---|---|
| `...` | Arguments to pass to `fun`. |
| `fun` | A function to execute. |
| `metric_name` | A character string with the name of the metric. Used to provide a more informative warning message in case `fun` errors. |

### Value

The result of `fun` or `NULL` if `fun` errors

### Examples

```
f <- function(x) {x}
scoringutils:::run_safely(2, fun = f, metric_name = "f")
scoringutils:::run_safely(2, y = 3, fun = f, metric_name = "f")
scoringutils:::run_safely(fun = f, metric_name = "f")
scoringutils:::run_safely(y = 3, fun = f, metric_name = "f")
```

---

## `sample_to_quantile` *Change data from a sample based format to a quantile format*

---

### Description

Transform data from a format that is based on predictive samples to a format based on plain quantiles.

### Usage

```
sample_to_quantile(
  forecast,
  quantile_level = c(0.05, 0.25, 0.5, 0.75, 0.95),
  type = 7
)
```

### Arguments

| | |
|---|---|
| forecast | A forecast object of class forecast_sample (a validated data.table with predicted and observed values, see as_forecast()). |
| quantile_level | A numeric vector of quantile levels for which quantiles will be computed. |
| type | Type argument passed down to the quantile function. For more information, see quantile(). |

### Value

a data.table in a long interval range format

### Examples

```
sample_to_quantile(as_forecast(example_sample_discrete))
```

---

## score *Evaluate forecasts*

---

### Description

score() applies a selection of scoring metrics to a forecast object (a data.table with forecasts and observations) as produced by as_forecast(). score() is a generic that dispatches to different methods depending on the class of the input data.

See the details section for more information on forecast types and input formats. For additional help and examples, check out the Getting Started Vignette as well as the paper Evaluating Forecasts with scoringutils in R.

### Usage

```
score(forecast, metrics, ...)

## S3 method for class 'forecast_binary'
score(forecast, metrics = metrics_binary(), ...)

## S3 method for class 'forecast_point'
score(forecast, metrics = metrics_point(), ...)

## S3 method for class 'forecast_sample'
score(forecast, metrics = metrics_sample(), ...)

## S3 method for class 'forecast_quantile'
score(forecast, metrics = metrics_quantile(), ...)
```

## Arguments

| | |
|---|---|
| `forecast` | A forecast object (a validated data.table with predicted and observed values, see `as_forecast()`) |
| `metrics` | A named list of scoring functions. Names will be used as column names in the output. See `metrics_point()`, `metrics_binary()`, `metrics_quantile()`, and `metrics_sample()` for more information on the default metrics used. Note that if you want to pass arguments to any given metric, you should do that through the function `customise_metric()` and pass an updated list of functions with your custom metric to the `metrics` argument in `score()`. |
| `...` | Additional arguments. Currently unused but allows for future extensions. If you want to pass arguments to individual metrics, use `customise_metric()`. |

## Value

An object of class `scores`. This object is a data.table with unsummarised scores (one score per forecast) and has an additional attribute `metrics` with the names of the metrics used for scoring. See `summarise_scores()`) for information on how to summarise scores.

## Forecast types and input formats

Various different forecast types / forecast formats are supported. At the moment, those are:

- point forecasts
- binary forecasts ("soft binary classification")
- Probabilistic forecasts in a quantile-based format (a forecast is represented as a set of predictive quantiles)
- Probabilistic forecasts in a sample-based format (a forecast is represented as a set of predictive samples)

Forecast types are determined based on the columns present in the input data. Here is an overview of the required format for each forecast type:

| Forecast type | | | column | type |
|---|---|---|---|---|
| All forecast types | | | `observed`<br>`predicted`<br>`model` | |
| Classification | Binary | Soft classification (prediction is probability) | `observed`<br>`predicted` | factor with 2 levels<br>numeric [0,1] |
| Point forecasts | Discrete, Continuous | | `observed`<br>`predicted` | numeric<br>numeric |
| Probabilistic forecast | Discrete, Continuous | Sample format | `observed`<br>`predicted`<br>`sample_id` | numeric<br>numeric<br>numeric |
| | | Quantile format | `observed`<br>`predicted`<br>`quantile_level` | numeric<br>numeric<br>numeric [0,1] |

*All forecast types* require a data.frame or similar with columns `observed` `predicted`, and `model`.

*Point forecasts* require a column `observed` of type numeric and a column `predicted` of type numeric.

*Binary forecasts* require a column `observed` of type factor with exactly two levels and a column `predicted` of type numeric with probabilities, corresponding to the probability that `observed` is equal to the second factor level. See details here for more information.

*Quantile-based forecasts* require a column `observed` of type numeric, a column `predicted` of type numeric, and a column `quantile_level` of type numeric with quantile-levels (between 0 and 1).

*Sample-based forecasts* require a column `observed` of type numeric, a column `predicted` of type numeric, and a column `sample_id` of type numeric with sample indices.

For more information see the vignettes and the example data (example_quantile, example_sample_continuous, example_sample_discrete, `example_point()`, and example_binary).

## Forecast unit

In order to score forecasts, `scoringutils` needs to know which of the rows of the data belong together and jointly form a single forecasts. This is easy e.g. for point forecast, where there is one row per forecast. For quantile or sample-based forecasts, however, there are multiple rows that belong to a single forecast.

The *forecast unit* or *unit of a single forecast* is then described by the combination of columns that uniquely identify a single forecast. For example, we could have forecasts made by different models in various locations at different time points, each for several weeks into the future. The forecast unit could then be described as `forecast_unit = c("model", "location", "forecast_date", "forecast_horizon")`. `scoringutils` automatically tries to determine the unit of a single forecast. It uses all existing columns for this, which means that no columns must be present that are unrelated to the forecast unit. As a very simplistic example, if you had an additional row, "even", that is one if the row number is even and zero otherwise, then this would mess up scoring as `scoringutils` then thinks that this column was relevant in defining the forecast unit.

In order to avoid issues, we recommend setting the forecast unit explicitly, usually through the `forecast_unit` argument in `as_forecast()`. This will drop unneeded columns, while making sure that all necessary, 'protected columns' like "predicted" or "observed" are retained.

## Author(s)

Nikos Bosse <nikosbosse@gmail.com>

## References

Bosse NI, Gruson H, Cori A, van Leeuwen E, Funk S, Abbott S (2022) Evaluating Forecasts with scoringutils in R. doi:10.48550/arXiv.2205.07090

## Examples

```
library(magrittr) # pipe operator
```

```
validated <- as_forecast(example_quantile)
score(validated) %>%
  summarise_scores(by = c("model", "target_type"))

# set forecast unit manually (to avoid issues with scoringutils trying to
# determine the forecast unit automatically)
example_quantile %>%
  as_forecast(
    forecast_unit = c(
      "location", "target_end_date", "target_type", "horizon", "model"
    )
  ) %>%
  score()

# forecast formats with different metrics
## Not run:
score(as_forecast(example_binary))
score(as_forecast(example_quantile))
score(as_forecast(example_point))
score(as_forecast(example_sample_discrete))
score(as_forecast(example_sample_continuous))

## End(Not run)
```

---

`scoring-functions-binary`

*Metrics for binary outcomes*

---

## Description

### Brier score

The Brier Score is the mean squared error between the probabilistic prediction and the observed outcome. The Brier score is a proper scoring rule. Small values are better (best is 0, the worst is 1).

$$\text{Brier\_Score} = (\text{prediction} - \text{outcome})^2,$$

where outcome $\in \{0, 1\}$, and prediction $\in [0, 1]$ represents the probability that the outcome is equal to 1.

### Log score for binary outcomes

The Log Score is the negative logarithm of the probability assigned to the observed value. It is a proper scoring rule. Small values are better (best is zero, worst is infinity).

## Usage

```
brier_score(observed, predicted)

logs_binary(observed, predicted)
```

### Arguments

| | |
|---|---|
| `observed` | A factor of length n with exactly two levels, holding the observed values. The highest factor level is assumed to be the reference level. This means that `predicted` represents the probability that the observed value is equal to the highest factor level. |
| `predicted` | A numeric vector of length n, holding probabilities. Values represent the probability that the corresponding outcome is equal to the highest level of the factor `observed`. |

### Details

#### Input formats

The functions require users to provide observed values as a factor in order to distinguish its input from the input format required for scoring point forecasts. Internally, however, factors will be converted to numeric values. A factor `observed = factor(c(0, 1, 1, 0, 1)` with two levels (`0` and `1`) would internally be coerced to a numeric vector (in this case this would result in the numeric vector `c(1, 2, 2, 1, 1)`). After subtracting 1, the resulting vector (`c(0, 1, 1, 0)` in this case) is used for internal calculations. All predictions are assumed represent the probability that the outcome is equal of the highest factor level (in this case that the outcome is equal to 1).

You could alternatively also provide a vector like `observed = factor(c("a", "b", "b", "a"))` (with two levels, `a` and `b`), which would result in exactly the same internal representation. Probabilities then represent the probability that the outcome is equal to "b". If you want your predictions to be probabilities that the outcome is "a", then you could of course make `observed` a factor with levels swapped, i.e. `observed = factor(c("a", "b", "b", "a"), levels = c("b", "a"))`

### Value

A numeric vector of size n with the Brier scores

A numeric vector of size n with log scores

### Examples

```
observed <- factor(sample(c(0, 1), size = 30, replace = TRUE))
predicted <- runif(n = 30, min = 0, max = 1)

brier_score(observed, predicted)
logs_binary(observed, predicted)
```

---

## `select_metrics` *Select metrics from a list of functions*

---

### Description

Helper function to return only the scoring rules selected by the user from a list of possible functions.

### Usage

```
select_metrics(metrics, select = NULL, exclude = NULL)
```

### Arguments

| | |
|---|---|
| `metrics` | A list of scoring functions. |
| `select` | A character vector of scoring rules to select from the list. If `select` is `NULL` (the default), all possible scoring rules are returned. |
| `exclude` | A character vector of scoring rules to exclude from the list. If `select` is not `NULL`, this argument is ignored. |

### Value

A list of scoring rules.

### Examples

```
select_metrics(
  metrics = metrics_binary(),
  select = "brier_score"
)
select_metrics(
  metrics = metrics_binary(),
  exclude = "log_score"
)
```

---

## `set_forecast_unit` *Set unit of a single forecast manually*

---

### Description

Helper function to set the unit of a single forecast (i.e. the combination of columns that uniquely define a single forecast) manually. This simple function keeps the columns specified in `forecast_unit` (plus additional protected columns, e.g. for observed values, predictions or quantile levels) and removes duplicate rows. `set_forecast_unit()` will mainly be called from `as_forecast()` through the `forecast_unit` argument.

If not done explicitly, `scoringutils` attempts to determine the unit of a single forecast automatically by simply assuming that all column names are relevant to determine the forecast unit. This may lead to unexpected behaviour, so setting the forecast unit explicitly can help make the code easier to debug and easier to read.

### Usage

```
set_forecast_unit(data, forecast_unit)
```

### Arguments

| | |
|---|---|
| `data` | A data.frame (or similar) with predicted and observed values. See the details section of `as_forecast()` for additional information on required input formats. |
| `forecast_unit` | Character vector with the names of the columns that uniquely identify a single forecast. |

### Value

A data.table with only those columns kept that are relevant to scoring or denote the unit of a single forecast as specified by the user.

### Examples

```
set_forecast_unit(
  example_quantile,
  c("location", "target_end_date", "target_type", "horizon", "model")
)
```

---

## `se_mean_sample` *Squared error of the mean (sample-based version)*

---

### Description

Squared error of the mean calculated as

$$\text{mean}(\text{observed} - \text{mean prediction})^2$$

The mean prediction is calculated as the mean of the predictive samples.

### Usage

```
se_mean_sample(observed, predicted)
```

### Arguments

| | |
|---|---|
| `observed` | A vector with observed values of size n |
| `predicted` | nxN matrix of predictive samples, n (number of rows) being the number of data points and N (number of columns) the number of Monte Carlo samples. Alternatively, `predicted` can just be a vector of size n. |

### Examples

```
observed <- rnorm(30, mean = 1:30)
predicted_values <- matrix(rnorm(30, mean = 1:30))
se_mean_sample(observed, predicted_values)
```

`summarise_scores` *Summarise scores as produced by* `score()`

## Description

Summarise scores as produced by `score()`.

`summarise_scores` relies on a way to identify the names of the scores and distinguish them from columns that denote the unit of a single forecast. Internally, this is done via a stored attribute, `metrics` that stores the names of the scores. This means, however, that you need to be careful with renaming scores after they have been produced by `score()`. If you do, you also have to manually update the attribute by calling `attr(scores, "metrics") <- new_names`.

## Usage

```
summarise_scores(scores, by = "model", fun = mean, ...)

summarize_scores(scores, by = "model", fun = mean, ...)
```

## Arguments

| | |
|---|---|
| `scores` | An object of class `scores` (a data.table with scores and an additional attribute `metrics` as produced by `score()`). |
| `by` | Character vector with column names to summarise scores by. Default is `model`, meaning that there will be one score per model in the output. |
| `fun` | A function used for summarising scores. Default is `mean()`. |
| `...` | Additional parameters that can be passed to the summary function provided to `fun`. For more information see the documentation of the respective function. |

## Value

A data.table with summarised scores. Scores are summarised according to the names of the columns of the original data specified in `by` using the `fun` passed to `summarise_scores()`.

## Examples

```
library(magrittr) # pipe operator
scores <- score(as_forecast(example_sample_continuous))

# get scores by model
summarise_scores(scores,by = "model")

# get scores by model and target type
summarise_scores(scores, by = c("model", "target_type"))
```

```
# get standard deviation
summarise_scores(scores, by = "model", fun = sd)

# round digits
summarise_scores(scores,by = "model") %>%
  summarise_scores(fun = signif, digits = 2)
```

---

`test_columns_not_present` *Test whether column names are NOT present in a data.frame*

---

## Description

The function checks whether all column names are NOT present. If none of the columns are present, the function returns TRUE. If one or more columns are present, the function returns FALSE.

## Usage

```
test_columns_not_present(data, columns)
```

## Arguments

| | |
|---|---|
| `data` | A data.frame or similar to be checked |
| `columns` | A character vector of column names to check |

## Value

Returns TRUE if none of the columns are present and FALSE otherwise

---

`test_columns_present` *Test whether all column names are present in a data.frame*

---

## Description

The function checks whether all column names are present. If one or more columns are missing, the function returns FALSE. If all columns are present, the function returns TRUE.

## Usage

```
test_columns_present(data, columns)
```

## Arguments

| | |
|---|---|
| `data` | A data.frame or similar to be checked |
| `columns` | A character vector of column names to check |

### Value

Returns TRUE if all columns are present and FALSE otherwise

---

`test_forecast_type_is_binary`
*Test whether data could be a binary forecast.*

---

### Description

Checks type of the necessary columns.

### Usage

```
test_forecast_type_is_binary(data)
```

### Arguments

| | |
|---|---|
| `data` | A data.frame or similar to be checked |

### Value

Returns TRUE if basic requirements are satisfied and FALSE otherwise

---

`test_forecast_type_is_point`
*Test whether data could be a point forecast.*

---

### Description

Checks type of the necessary columns.

### Usage

```
test_forecast_type_is_point(data)
```

### Arguments

| | |
|---|---|
| `data` | A data.frame or similar to be checked |

### Value

Returns TRUE if basic requirements are satisfied and FALSE otherwise

---

`test_forecast_type_is_quantile`
*Test whether data could be a quantile forecast.*

---

## Description

Checks type of the necessary columns.

## Usage

```
test_forecast_type_is_quantile(data)
```

## Arguments

| | |
|---|---|
| `data` | A data.frame or similar to be checked |

## Value

Returns TRUE if basic requirements are satisfied and FALSE otherwise

---

`test_forecast_type_is_sample`
*Test whether data could be a sample-based forecast.*

---

## Description

Checks type of the necessary columns.

## Usage

```
test_forecast_type_is_sample(data)
```

## Arguments

| | |
|---|---|
| `data` | A data.frame or similar to be checked |

## Value

Returns TRUE if basic requirements are satisfied and FALSE otherwise

## theme_scoringutils *Scoringutils ggplot2 theme*

### Description

A theme for ggplot2 plots used in `scoringutils`.

### Usage

```
theme_scoringutils()
```

### Value

A ggplot2 theme

## transform_forecasts *Transform forecasts and observed values*

### Description

Function to transform forecasts and observed values before scoring.

### Usage

```
transform_forecasts(
  forecast,
  fun = log_shift,
  append = TRUE,
  label = "log",
  ...
)
```

### Arguments

| | |
|---|---|
| `forecast` | A forecast object (a validated data.table with predicted and observed values, see `as_forecast()`) |
| `fun` | A function used to transform both observed values and predictions. The default function is `log_shift()`, a custom function that is essentially the same as `log()`, but has an additional arguments (`offset`) that allows you add an offset before applying the logarithm. This is often helpful as the natural log transformation is not defined at zero. A common, and pragmatic solution, is to add a small offset to the data before applying the log transformation. In our work we have often used an offset of 1 but the precise value will depend on your application. |

| | |
|---|---|
| append | Logical, defaults to TRUE. Whether or not to append a transformed version of the data to the currently existing data (TRUE). If selected, the data gets transformed and appended to the existing data, making it possible to use the outcome directly in score(). An additional column, 'scale', gets created that denotes which rows or untransformed ('scale' has the value "natural") and which have been transformed ('scale' has the value passed to the argument label). |
| label | A string for the newly created 'scale' column to denote the newly transformed values. Only relevant if append = TRUE. |
| ... | Additional parameters to pass to the function you supplied. For the default option of log_shift() this could be the offset argument. |

## Details

There are a few reasons, depending on the circumstances, for why this might be desirable (check out the linked reference for more info). In epidemiology, for example, it may be useful to log-transform incidence counts before evaluating forecasts using scores such as the weighted interval score (WIS) or the continuous ranked probability score (CRPS). Log-transforming forecasts and observations changes the interpretation of the score from a measure of absolute distance between forecast and observation to a score that evaluates a forecast of the exponential growth rate. Another motivation can be to apply a variance-stabilising transformation or to standardise incidence counts by population.

Note that if you want to apply a transformation, it is important to transform the forecasts and observations and then apply the score. Applying a transformation after the score risks losing propriety of the proper scoring rule.

## Value

A forecast object with either a transformed version of the data, or one with both the untransformed and the transformed data. includes the original data as well as a transformation of the original data. There will be one additional column, 'scale', present which will be set to "natural" for the untransformed forecasts.

## Author(s)

Nikos Bosse <nikosbosse@gmail.com>

## References

Transformation of forecasts for evaluating predictive performance in an epidemiological context Nikos I. Bosse, Sam Abbott, Anne Cori, Edwin van Leeuwen, Johannes Bracher, Sebastian Funk medRxiv 2023.01.23.23284722 doi:10.1101/2023.01.23.23284722 https://www.medrxiv.org/content/10.1101/2023.01.23.23284722v1

## Examples

```
library(magrittr) # pipe operator

# transform forecasts using the natural logarithm
```

```
# negative values need to be handled (here by replacing them with 0)
example_quantile %>%
  .[, observed := ifelse(observed < 0, 0, observed)] %>%
  as_forecast() %>%
# Here we use the default function log_shift() which is essentially the same
# as log(), but has an additional arguments (offset) that allows you add an
# offset before applying the logarithm.
  transform_forecasts(append = FALSE) %>%
  head()

# alternatively, integrating the truncation in the transformation function:
example_quantile %>%
  as_forecast() %>%
 transform_forecasts(
   fun = function(x) {log_shift(pmax(0, x))}, append = FALSE
 ) %>%
 head()

# specifying an offset for the log transformation removes the
# warning caused by zeros in the data
example_quantile %>%
  as_forecast() %>%
  .[, observed := ifelse(observed < 0, 0, observed)] %>%
  transform_forecasts(offset = 1, append = FALSE) %>%
  head()

# adding square root transformed forecasts to the original ones
example_quantile %>%
  .[, observed := ifelse(observed < 0, 0, observed)] %>%
  as_forecast() %>%
  transform_forecasts(fun = sqrt, label = "sqrt") %>%
  score() %>%
  summarise_scores(by = c("model", "scale"))

# adding multiple transformations
example_quantile %>%
  as_forecast() %>%
  .[, observed := ifelse(observed < 0, 0, observed)] %>%
  transform_forecasts(fun = log_shift, offset = 1) %>%
  transform_forecasts(fun = sqrt, label = "sqrt") %>%
  head()
```

---

`validate_forecast` *Re-validate an existing forecast object*

---

## Description

The function re-validates an existing forecast object. It is similar to `assert_forecast()`, but returns the input data instead of an invisible `NULL`. See `as_forecast()` for details on the expected input formats.

### Usage

```
validate_forecast(forecast, forecast_type = NULL, verbose = TRUE)
```

### Arguments

| | |
|---|---|
| forecast | A forecast object (a validated data.table with predicted and observed values, see as_forecast()) |
| forecast_type | (optional) The forecast type you expect the forecasts to have. If the forecast type as determined by scoringutils based on the input does not match this, an error will be thrown. If NULL (the default), the forecast type will be inferred from the data. |
| verbose | Logical. If FALSE (default is TRUE), no messages and warnings will be created. |

### Value

Returns NULL invisibly.

### Examples

```
forecast <- as_forecast(example_binary)
assert_forecast(forecast)
```

---

## validate_metrics *Validate metrics*

---

### Description

This function validates whether the list of metrics is a list of valid functions.

The function is used in score() to make sure that all metrics are valid functions.

### Usage

```
validate_metrics(metrics)
```

### Arguments

| | |
|---|---|
| metrics | A named list with metrics. Every element should be a scoring function to be applied to the data. |

### Value

A named list of metrics, with those filtered out that are not valid functions

`wis` *Weighted interval score (WIS)*

## Description

The WIS is a proper scoring rule used to evaluate forecasts in an interval- / quantile-based format. See Bracher et al. (2021). Smaller values are better.

As the name suggest the score assumes that a forecast comes in the form of one or multiple central prediction intervals. A prediction interval is characterised by a lower and an upper bound formed by a pair of predictive quantiles. For example, a 50% central prediction interval is formed by the 0.25 and 0.75 quantiles of the predictive distribution.

**Interval score**

The interval score (IS) is the sum of three components: overprediction, underprediction and dispersion. For a single prediction interval only one of the components is non-zero. If for a single prediction interval the observed value is below the lower bound, then the interval score is equal to the absolute difference between the lower bound and the observed value ("underprediction"). "Overprediction" is defined analogously. If the observed value falls within the bounds of the prediction interval, then the interval score is equal to the width of the prediction interval, i.e. the difference between the upper and lower bound. For a single interval, we therefore have:

$$\text{IS} = (\text{upper} - \text{lower}) + \frac{2}{\alpha}(\text{lower} - \text{observed}) * \mathbf{1}(\text{observed} < \text{lower}) + \frac{2}{\alpha}(\text{observed} - \text{upper}) * \mathbf{1}(\text{observed} > \text{upper})$$

where $\mathbf{1}()$ is the indicator function and indicates how much is outside the prediction interval. $\alpha$ is the decimal value that indicates how much is outside the prediction interval. For a 90% prediction interval, for example, $\alpha$ is equal to 0.1. No specific distribution is assumed, but the interval formed by the quantiles has to be symmetric around the median (i.e you can't use the 0.1 quantile as the lower bound and the 0.7 quantile as the upper bound). Non-symmetric quantiles can be scored using the function `quantile_score()`.

Usually the interval score is weighted by a factor that makes sure that the average score across an increasing number of equally spaced quantiles, converges to the continuous ranked probability score (CRPS). This weighted score is called the weighted interval score (WIS). The weight commonly used is $\alpha/2$.

**Quantile score**

In addition to the interval score, there also exists a quantile score (QS) (see `quantile_score()`), which is equal to the so-called pinball loss. The quantile score can be computed for a single quantile (whereas the interval score requires two quantiles that form an interval). However, the intuitive decomposition into overprediction, underprediction and dispersion does not exist for the quantile score.

**Two versions of the weighted interval score**

There are two ways to conceptualise the weighted interval score across several quantiles / prediction intervals and the median.

In one view, you would treat the WIS as the average of quantile scores (and the median as 0.5-quantile) (this is the default for `wis()`). In another view, you would treat the WIS as the average of several interval scores + the difference between the observed value and median forecast. The effect of that is that in contrast to the first view, the median has twice as much weight (because it is weighted like a prediction interval, rather than like a single quantile). Both are valid ways to conceptualise the WIS and you can control the behaviour with the `count_median_twice`-argument.

**WIS components**: WIS components can be computed individually using the functions `overprediction`, `underprediction`, and `dispersion`.

## Usage

```
wis(
  observed,
  predicted,
  quantile_level,
  separate_results = FALSE,
  weigh = TRUE,
  count_median_twice = FALSE,
  na.rm = TRUE
)

dispersion(observed, predicted, quantile_level, ...)

overprediction(observed, predicted, quantile_level, ...)

underprediction(observed, predicted, quantile_level, ...)
```

## Arguments

| | |
|---|---|
| `observed` | Numeric vector of size n with the observed values. |
| `predicted` | Numeric nxN matrix of predictive quantiles, n (number of rows) being the number of forecasts (corresponding to the number of observed values) and N (number of columns) the number of quantiles per forecast. If `observed` is just a single number, then predicted can just be a vector of size N. |
| `quantile_level` | Vector of of size N with the quantile levels for which predictions were made. |
| `separate_results` | Logical. If `TRUE` (default is `FALSE`), then the separate parts of the interval score (dispersion penalty, penalties for over- and under-prediction get returned as separate elements of a list). If you want a `data.frame` instead, simply call `as.data.frame()` on the output. |
| `weigh` | Logical. If `TRUE` (the default), weigh the score by $\alpha/2$, so it can be averaged into an interval score that, in the limit (for an increasing number of equally spaced quantiles/prediction intervals), corresponds to the CRPS. $\alpha$ is the value that corresponds to the $(\alpha/2)$ or $(1 - \alpha/2)$, i.e. it is the decimal value that represents how much is outside a central prediction interval (E.g. for a 90 percent central prediction interval, alpha is 0.1). |

`count_median_twice`
: If TRUE, count the median twice in the score.

`na.rm`
: If TRUE, ignore NA values when computing the score.

`...`
: Additional arguments passed on to `wis()` from functions `overprediction()`, `underprediction()` and `dispersion()`.

## Value

`wis()`: a numeric vector with WIS values of size n (one per observation), or a list with separate entries if `separate_results` is `TRUE`.

`dispersion()`: a numeric vector with dispersion values (one per observation).

`overprediction()`: a numeric vector with overprediction values (one per observation).

`underprediction()`: a numeric vector with underprediction values (one per observation)

## Examples

```
observed <- c(1, -15, 22)
predicted <- rbind(
  c(-1, 0, 1, 2, 3),
  c(-2, 1, 2, 2, 4),
  c(-2, 0, 3, 3, 4)
)
quantile_level <- c(0.1, 0.25, 0.5, 0.75, 0.9)
wis(observed, predicted, quantile_level)
```

# Index