Package 'forecastsuite'

November 3, 2026

Title Multi-Model Time Series Forecasting Toolkit and Local App

Version 1.0.3

Description Fits, compares and visualizes univariate time series forecasts through an extensible model registry built largely on the 'forecast' package: AutoRegressive Integrated Moving Average (ARIMA) and Seasonal ARIMA (SARIMA) models (Hyndman and Khandakar (2008) <doi:10.18637/jss.v027.i03>), Error, Trend, Seasonal (ETS) exponential smoothing (Hyndman et al. (2002) <doi:10.1016/S0169-2070(01)00110-8>), Trigonometric seasonality, Box-Cox transformation, AutoRegressive Moving Average (ARMA) errors, Trend and Seasonal components (TBATS) models (De Livera, Hyndman and Snyder (2011) <doi:10.1198/jasa.2011.tm09771>), Neural Network AutoRegression (NNETAR) and Holt-Winters exponential smoothing (Winters (1960) <doi:10.1287/mnsc.6.3.324>), plus optional 'Prophet' models via the 'prophet' package (Taylor and Letham (2018) <doi:10.1080/00031305.2017.1380080>) and Long Short-Term Memory (LSTM) networks via the 'torch' package (Hochreiter and Schmidhuber (1997) <doi:10.1162/neco.1997.9.8.1735>). Also provides grouped series with bottom-up and top-down reconciliation, rolling-origin cross-validation, ensembles and residual diagnostics, following Hyndman and Athanasopoulos (2021) <https://otexts.com/fpp3/>, and a bundled local 'shiny' application exposing every model.

License MIT + file LICENSE

URL https://github.com/SpaRroW410/Forecast_dashboard/tree/main/forecastsuite

BugReports https://github.com/SpaRroW410/Forecast_dashboard/issues

Encoding UTF-8

Depends R (>= 4.1)

Collate 'forecastsuite-package.R' 'globals.R' 'registry.R' 'horizon_utils.R' 'data_utils.R' 'aggregate_period.R' 'grouping_utils.R' 'reconciliation.R' 'ensemble.R' 'date_parts.R' 'google_sheets.R' 'metric_utils.R' 'multi_window_metrics.R' 'rolling_cv.R' 'holidays_helpers.R' 'contingency.R' 'model_shared.R' 'model_prophet.R' 'model_arima.R' 'model_sarima.R' 'model_ets.R' 'model_tbats.R'

'model_nnetar.R' 'model_holtwinters.R' 'model_lstm.R'
'plot_generic.R' 'plot_png.R' 'plot_diagnostics.R'
'recommend.R' 'diagnostics.R' 'bias_drift.R' 'suggest_params.R'
'code_gen.R' 'app_guide.R' 'app_workflow_guide.R'
'app_holidays.R' 'ui_helpers.R' 'app_ui.R' 'project_io.R'
'app_server.R' 'run_app.R' 'zzz_register_builtins.R'

Imports shiny, dplyr, lubridate, tibble, tidyr, stringr, plotly, DT, forecast, zoo, shinycssloaders, shinyjs, colourpicker, stats, utils, grDevices, graphics

Suggests prophet, torch, readxl, writexl, testthat (>= 3.0.0), withr, rmarkdown, knitr

Config/testthat/edition 3

VignetteBuilder knitr

Config/roxygen2/version 8.1.0

NeedsCompilation no

Author Mukul Maheshwari [aut, cre, cph] (ORCID: <https://orcid.org/0000-0002-1918-5100>)

Maintainer Mukul Maheshwari <ryan00700@gmail.com>

Repository CRAN

Date/Publication 2026-11-03 17:20:27 UTC

Contents

always_zero_dates . . . 3
analyze_series . . . 4
build_cv_folds . . . 5
collapse_to_period . . . 6
compose_date_parts . . . 7
compute_forecast_bias . . . 8
compute_forecast_metrics . . . 8
compute_group_value_map . . . 9
compute_interval_coverage . . . 10
compute_multi_window_metrics . . . 10
compute_residual_diagnostics . . . 11
convert_months_to_horizon . . . 12
count_duplicate_periods . . . 12
decompose_series . . . 13
detect_anomalies . . . 14
detect_bias_drift . . . 15
ensemble_forecasts . . . 15
fixed_holiday_catalog . . . 16
get_model . . . 17
googlesheet_csv_url . . . 17
group_correlation_matrix . . . 18
holidays_with_data . . . 19

holiday_contingency . . . 19
holiday_limitation_note . . . 20
impute_anomalies . . . 20
list_models . . . 21
lstm_available . . . 22
plot_forecast_generic . . . 22
plot_model_comparison . . . 23
process_uploaded_data . . . 24
prophet_available . . . 25
read_google_sheet . . . 26
recommend_model . . . 26
reconcile_bottom_up . . . 27
reconcile_top_down . . . 28
register_model . . . 29
run_app . . . 30
run_backtest_leaderboard . . . 31
safe_compute_metrics . . . 32
split_by_group . . . 33
suggest_parameters . . . 33
suggest_parameters_for . . . 34
ts_frequency_for . . . 35

Index **36**

`always_zero_dates` *Find calendar dates that are always zero or missing*

Description

Calendar dates (month-day) that are zero or missing in every year present – these look like de facto holidays/closures that were never declared, so a holiday-aware model can't account for them.

Usage

```
always_zero_dates(data, holidays = NULL)
```

Arguments

`data`	A tibble with ds/y columns, the finalized dataset.
`holidays`	Optional tibble with a ds column, the compiled holiday list – used only to flag which suspicious dates are already declared.

Value

A tibble with columns `ds`, `flag` (`"Zero every year"` or `"Missing every year"`), and `declared_holiday` (logical) – the undeclared rows are the actual gap.

Examples

```
data <- tibble::tibble(
  ds = c(as.Date("2023-01-01"), as.Date("2024-01-01"), as.Date("2023-06-01")),
  y = c(0, 0, 5)
)
always_zero_dates(data)
```

`analyze_series` *Analyze a series' shape for model recommendation*

Description

A lightweight, no-model-fitting heuristic: measures trend/seasonal strength (via STL decomposition), differencing needed (`forecast::ndiffs`/`nsdiffs`), missing-data ratio, and a Box-Jenkins ACF/PACF-based (p,q)(P,Q) order suggestion. Feeds `recommend_model()`'s scoring and the app's suggested-hyperparameters table.

Usage

```
analyze_series(df, date_agg = "day", robust = FALSE)
```

Arguments

`df`	A tibble with ds/y columns.
`date_agg`	Aggregation frequency: one of `"hour"`, `"day"`, `"week"`, `"month"`, `"quarter"`, `"year"`.
`robust`	Logical; use robust (outlier-downweighted) STL when a seasonal decomposition is attempted internally. Default `FALSE`.

Value

A list with `n_obs`, `detected_freq`, `n_cycles`, `trend_strength`, `seasonal_strength`, `ndiffs_needed`, `nsdiffs_needed`, `missing_ratio`, `arima_p`, `arima_q`, `arima_P`, `arima_Q`.

Examples

```
df <- tibble::tibble(ds = as.Date("2024-01-01") + 0:29, y = 1:30 + rnorm(30))
analyze_series(df, date_agg = "day")
```

`build_cv_folds` *Build rolling-origin (walk-forward) cross-validation folds*

Description

Builds up to `k_requested` chronological, non-overlapping walk-forward folds from one `ds`-ordered series. Each fold's test window is exactly `horizon_periods` rows long. Degrades gracefully: if the series can't support `k_requested` folds, fewer are returned instead of erroring – possibly zero, for a very short series.

Usage

```
build_cv_folds(
  df,
  horizon_periods,
  k_requested,
  min_train = 2,
  window = c("expanding", "rolling"),
  train_window = NULL
)
```

Arguments

`df`	A data frame with a `ds` column, in any row order (it's sorted internally).
`horizon_periods`	Integer, the length of each fold's test window, in the same units as `df$ds`'s spacing.
`k_requested`	Integer, the number of folds requested (may return fewer).
`min_train`	Integer, the minimum number of training rows a fold must have to be included. Default `2`.
`window`	`"expanding"` (the default – training windows always start from the beginning of the series, growing fold over fold) or `"rolling"` (each fold's training window is capped to the most recent `train_window` rows instead, a fixed width – useful for testing robustness to concept drift/regime change separately from an ever-larger training set).
`train_window`	Integer, the fixed training-window width used when `window = "rolling"`. Ignored for `"expanding"`. Defaults to `NULL`, which uses what would be the smallest (earliest) fold's expanding training size – a sensible width with no extra input needed.

Value

A list of folds, oldest-first (fold 1 has the smallest training window under `"expanding"`, or the same fixed width under `"rolling"`), each a list with `train` and `test` data frames.

Examples

```
df <- data.frame(ds = as.Date("2024-01-01") + 0:29, y = 1:30)
folds <- build_cv_folds(df, horizon_periods = 5, k_requested = 3)
length(folds)
rolling_folds <- build_cv_folds(df, horizon_periods = 5, k_requested = 3,
                                window = "rolling", train_window = 10)
length(rolling_folds)
```

---

collapse_to_period *Collapse multiple rows sharing a period into one*

Description

Snaps each row's ds to the start of its period (per date_agg) and combines rows that land on the same period (and, if group_col is supplied, the same group) – e.g. case data split by district/sex/age with many rows per Year+Quarter.

Usage

```
collapse_to_period(
  df,
  date_agg = "day",
  fun = c("sum", "mean", "median", "none"),
  group_col = NULL
)
```

Arguments

df	A data frame with ds (date-like) and y (numeric) columns.
date_agg	Aggregation frequency: one of "hour", "day", "week", "month", "quarter", "year".
fun	How to combine rows sharing a period: "sum" (the default, for counts), "mean"/"median" (for rates/indices, where summing across strata would be meaningless), or "none" (return df unchanged).
group_col	Optional character scalar; rows only collapse together if they share both a period and this column's value.

Value

A tibble with one row per period (and group, if group_col is supplied), columns ds/y(/group_col).

Examples

```
df <- data.frame(ds = as.Date(c("2024-01-01", "2024-01-02")), y = c(3, 5))
collapse_to_period(df, date_agg = "month")
```

`compose_date_parts` *Assemble a date from separate Year/Quarter/Month/Day columns*

Description

Real-world tables often split the period across columns (a Year column plus a Quarter or Month column, sometimes plus a Day column) rather than carrying one parseable date. This assembles those into a single `Date` and reports the finest granularity present, so the caller can aggregate at that level (Year+Quarter -> quarterly, Year+Month -> monthly, and so on) instead of guessing.

Usage

```
compose_date_parts(
  df,
  year_col,
  quarter_col = NULL,
  month_col = NULL,
  day_col = NULL
)
```

Arguments

`df`	A data frame containing the part columns.
`year_col`	Character scalar, the Year column's name. Required.
`quarter_col`	Optional character scalar, a Quarter column (accepts `1-4`, `"Q1"-"Q4"`, `"Quarter 2"`, etc.). Supply at most one of `quarter_col`/`month_col`.
`month_col`	Optional character scalar, a Month column (accepts `1-12`, `"Jan"`, `"January"`, etc.).
`day_col`	Optional character scalar, a Day-of-month column. Only meaningful alongside `month_col`.

Value

A list with `ds` (a `Date` vector) and `granularity` (one of `"year"`, `"quarter"`, `"month"`, `"day"`, the finest part supplied).

Examples

```
df <- data.frame(Year = c(2023, 2023, 2024), Month = c("Jan", "Feb", "Jan"))
compose_date_parts(df, year_col = "Year", month_col = "Month")
```

compute_forecast_bias *Signed forecast error (bias)*

Description

MASE/sMAPE/RMSE are all magnitude-only, so a model that's consistently over- or under-forecasting can still look "fine" on those alone. This surfaces the signed error directly. Generic across every registered model – only needs `fc_tib`'s `ds`/`yhat` and `test_df`'s `ds`/`y`.

Usage

```
compute_forecast_bias(fc_tib, test_df)
```

Arguments

`fc_tib`	A tibble with `ds`/`yhat` columns – a model's forecast.
`test_df`	A tibble with `ds`/`y` columns – the held-out actuals.

Value

A list with `bias` (mean signed error, `y - yhat`), `pct_bias` (`bias` as a percentage of mean `|y|`), and `n`. All `NA`/`0` if there's no overlap.

Examples

```
ds <- as.Date("2024-01-01") + 0:4
fc_tib <- tibble::tibble(ds = ds, yhat = rep(8, 5))
test_df <- tibble::tibble(ds = ds, y = rep(10, 5))
compute_forecast_bias(fc_tib, test_df)
```

compute_forecast_metrics *Compute MASE, sMAPE, and RMSE for a forecast*

Description

Compute MASE, sMAPE, and RMSE for a forecast

Usage

```
compute_forecast_metrics(actual_df, forecast_df, label = "Test")
```

Arguments

actual_df	A tibble with ds/y columns – the held-out actuals.
forecast_df	A tibble with ds/yhat columns – the forecast to evaluate.
label	Character scalar identifying this evaluation (shown as the Set column), e.g. "Test".

Value

A tibble with columns Set, Metric ("MASE", "sMAPE (%)", "RMSE"), and Value – NA for all three if fewer than 2 rows overlap.

Examples

```
ds <- as.Date("2024-01-01") + 0:9
actual_df <- tibble::tibble(ds = ds, y = 1:10)
forecast_df <- tibble::tibble(ds = ds, yhat = 1:10 + 0.5)
compute_forecast_metrics(actual_df, forecast_df, label = "Test")
```

---

compute_group_value_map

Build a raw-value -> canonical-label map for a grouping column

---

Description

Real-world grouping columns are often messy: "female"/"Female"/ "FEMALE" in the same column, meant to be one group, not three. Builds a raw-value -> canonical-label mapping (one row per distinct raw value) that the Import tab's "Merge / relabel values" table starts from and the user can further edit by hand.

Usage

```
compute_group_value_map(raw_values, merge_case = TRUE)
```

Arguments

raw_values	The full grouping column, with repeats – frequency matters, since the most common casing/spacing variant becomes each cluster's canonical label (ties broken by whichever appears first).
merge_case	Logical; if TRUE (the default), values identical after trimws()+tolower() collapse into one label; if FALSE, every distinct raw value keeps its own label.

Value

A tibble with columns raw (every distinct input value) and label (its canonical label).

Examples

```
compute_group_value_map(c("female", "Female", "male"))
```

compute_interval_coverage

Empirical prediction-interval coverage

Description

Whether a model's uncertainty bounds are honest, not just its point forecast: what fraction of actuals actually fell inside `yhat_lower`/`yhat_upper`, versus the interval's nominal confidence level (Prophet and the `forecast`-package adapters default to an 80% interval).

Usage

```
compute_interval_coverage(fc_tib, test_df, nominal_level = 0.8)
```

Arguments

`fc_tib`	A tibble with `ds`/`yhat` columns, and typically `yhat_lower`/`yhat_upper`. Models with no interval columns (e.g. NNETAR without `PI = TRUE`) return `NA` coverage rather than erroring.
`test_df`	A tibble with `ds`/`y` columns – the held-out actuals.
`nominal_level`	Numeric, the interval's nominal confidence level. Default `0.8`.

Value

A list with `empirical_coverage`, `nominal_level`, `gap` (`empirical_coverage - nominal_level`), and `n`.

Examples

```
ds <- as.Date("2024-01-01") + 0:9
fc_tib <- tibble::tibble(ds = ds, yhat = 10, yhat_lower = 9, yhat_upper = 11)
test_df <- tibble::tibble(ds = ds, y = c(rep(10, 8), 20, 20))
compute_interval_coverage(fc_tib, test_df)
```

compute_multi_window_metrics

Score a forecast over several evaluation windows at once

Description

A single aggregate score hides where a model actually fails – it can look strong on the full training range while drifting badly over the most recent months. Scores Train / Last 6 Months / Last 2 Years / Test (held out) using `safe_compute_metrics()`, skipping windows the series is too short to contain rather than reporting them as `NA`.

Usage

```
compute_multi_window_metrics(forecast_df, train_df, test_df, date_agg = "day")
```

Arguments

forecast_df	A tibble with ds/yhat columns.
train_df	A tibble with ds/y columns – the training data.
test_df	A tibble with ds/y columns – the held-out test data.
date_agg	Aggregation frequency (currently unused directly, kept for interface consistency with the rest of the package).

Value

A tibble with columns Set, Metric, Value – one block per window that applied.

Examples

```
ds <- as.Date("2024-01-01") + 0:29
train_df <- tibble::tibble(ds = ds[1:20], y = 1:20)
test_df <- tibble::tibble(ds = ds[21:30], y = 21:30)
forecast_df <- tibble::tibble(ds = ds, yhat = 1:30 + 0.5)
compute_multi_window_metrics(forecast_df, train_df, test_df)
```

---

compute_residual_diagnostics

Residual diagnostics for a forecast

Description

Whether a fitted model's error looks like unpredictable noise, or still has structure left in it: a Ljung-Box test for residual autocorrelation, a Shapiro-Wilk normality test, and the residual ACF with its Box-Jenkins confidence band. Generic across every registered model – only needs fc_tib's ds/yhat and test_df's ds/y.

Usage

```
compute_residual_diagnostics(fc_tib, test_df)
```

Arguments

fc_tib	A tibble with ds/yhat columns – a model's forecast.
test_df	A tibble with ds/y columns – the held-out actuals.

Value

A list with residuals (tibble of ds/resid), ljung_box_p, shapiro_p (each NA if there isn't enough overlapping data), acf_df (tibble of lag/acf), ci (the confidence bound), and n.

Examples

```
ds <- as.Date("2024-01-01") + 0:29
fc_tib <- tibble::tibble(ds = ds, yhat = 100)
test_df <- tibble::tibble(ds = ds, y = 100 + rnorm(30))
compute_residual_diagnostics(fc_tib, test_df)
```

`convert_months_to_horizon` *Convert a forecast horizon in months to periods*

Description

Convert a forecast horizon in months to periods

Usage

```
convert_months_to_horizon(months, aggregation)
```

Arguments

`months`	Numeric, the horizon in months (as set by the app's "Forecast Horizon (months)" slider).
`aggregation`	One of `"hour"`, `"day"`, `"week"`, `"month"`, `"quarter"`, `"year"` – the series' aggregation frequency.

Value

A numeric horizon in the same units as `aggregation` (e.g. days, for `aggregation = "day"`), suitable for a model's `forecast(model, h)` call.

Examples

```
convert_months_to_horizon(12, "day")
convert_months_to_horizon(12, "month")
```

`count_duplicate_periods` *Count how many rows share a period*

Description

Used to tell the user what `collapse_to_period()` would do, before doing it. With `group_col` supplied, rows only count as duplicates if they share both a period and a group – two different districts on the same date are not a collision.

Usage

```
count_duplicate_periods(df, date_agg = "day", group_col = NULL)
```

Arguments

df	A data frame with a ds (date-like) column.
date_agg	Aggregation frequency: one of "hour", "day", "week", "month", "quarter", "year".
group_col	Optional character scalar, a grouping column.

Value

A single integer: how many rows would be removed by collapsing (0 if none).

Examples

```
df <- data.frame(ds = as.Date(c("2024-01-01", "2024-01-02")), y = c(3, 5))
count_duplicate_periods(df, date_agg = "month")
```

---

decompose_series *Seasonal decomposition of a series*

Description

Surfaces the STL trend/seasonal/remainder split that analyze_series() already computes internally to drive the recommendation heuristic, as a tidy tibble a caller can plot directly.

Usage

```
decompose_series(df, date_agg = "day", robust = FALSE)
```

Arguments

df	A tibble with ds/y columns.
date_agg	Aggregation frequency: one of "hour", "day", "week", "month", "quarter", "year".
robust	Logical; use robust (outlier-downweighted) STL – stats::stl(..., robust = TRUE) – instead of ordinary STL, so a handful of spikes/dips don't skew the trend/seasonal split. Default FALSE.

Value

A tibble with columns ds, observed, trend, seasonal, remainder, or NULL (not an error) when no candidate frequency has enough cycles of data, or date_agg has no seasonal candidates at all (e.g. "year").

Examples

```
n <- 60
df <- tibble::tibble(ds = as.Date("2024-01-01") + 0:(n - 1),
                     y = 100 + 10 * sin(2 * pi * seq_len(n) / 7) + rnorm(n))
decompose_series(df, date_agg = "day")
```

`detect_anomalies` *Flag unusual points in a series*

Description

A statistical check on the values themselves – distinct from the Holidays tab's consistency check (calendar-vs-declared-holiday mismatch). Uses the STL remainder when `decompose_series()` finds one (isolates unusual points from normal trend/seasonal movement, which a raw-value check can't); falls back to deviation from a running median when no seasonal decomposition applies (short series, yearly aggregation, etc.), so it still works on every series shape.

Usage

```
detect_anomalies(
  df,
  date_agg = "day",
  method = c("iqr", "zscore"),
  threshold = 1.5
)
```

Arguments

`df`	A tibble with ds/y columns.
`date_agg`	Aggregation frequency: one of `"hour"`, `"day"`, `"week"`, `"month"`, `"quarter"`, `"year"`.
`method`	`"iqr"` (the default) flags points outside `threshold * IQR` from the quartiles; `"zscore"` flags points more than `threshold` standard deviations from the mean.
`threshold`	Numeric sensitivity threshold. Default `1.5`.

Value

A tibble with columns `ds`, `y`, `score` (the value the threshold was applied to), and `is_anomaly` (logical).

Examples

```
df <- tibble::tibble(ds = as.Date("2024-01-01") + 0:29, y = c(rep(10, 29), 100))
detect_anomalies(df, date_agg = "day", method = "iqr")
```

detect_bias_drift *Check whether forecast bias is growing across the test window*

Description

Splits the test window into `n_splits` sequential, roughly-equal chunks (oldest first) and computes `compute_forecast_bias()` in each – a simple, explainable drift signal (is `|bias|` growing chunk over chunk), not a full statistical trend test.

Usage

```
detect_bias_drift(fc_tib, test_df, n_splits = 2)
```

Arguments

`fc_tib`	A tibble with `ds`/`yhat` columns – a model's forecast.
`test_df`	A tibble with `ds`/`y` columns – the held-out actuals.
`n_splits`	Integer, how many sequential chunks to split the window into. Default `2`. Capped so every chunk gets at least one row rather than erroring on a short window.

Value

A list with `chunks` (a tibble of `chunk`/`bias`/`pct_bias`/`n`, one row per chunk) and `drifting` (`TRUE` if `|bias|` strictly increases chunk over chunk, `FALSE` if not, `NA` if there isn't enough data for a read).

Examples

```
ds <- as.Date("2024-01-01") + 0:9
fc_tib <- tibble::tibble(ds = ds, yhat = c(rep(10, 5), rep(6, 5)))
test_df <- tibble::tibble(ds = ds, y = rep(10, 10))
detect_bias_drift(fc_tib, test_df, n_splits = 2)
```

ensemble_forecasts *Combine several models' forecasts into one*

Description

Averages (simple or inverse-error-weighted) several already-fit models' forecasts – no re-fitting. Models that don't share every `ds` are handled by renormalizing weights per-row over whichever models actually have a value there.

Usage

```
ensemble_forecasts(
  forecasts,
  method = c("mean", "inverse_error"),
  errors = NULL
)
```

Arguments

`forecasts`	A named list of `ds`/`yhat` tibbles (names are the model labels).
`method`	`"mean"` (the default) weights every model equally; `"inverse_error"` weights each model by `1/errors[[name]]`. A model missing from `errors`, or with a non-positive/`NA` error, falls back to equal weight rather than dividing by zero.
`errors`	Required for `method = "inverse_error"`: a named list or vector of each model's own test error (e.g. RMSE or MASE, from `safe_compute_metrics()` – not re-derived here), keyed by the same names as `forecasts`.

Value

A tibble with columns `ds` and `yhat` – the combined forecast.

Examples

```
ds <- as.Date("2024-01-01") + 0:2
forecasts <- list(
  A = tibble::tibble(ds = ds, yhat = c(10, 10, 10)),
  B = tibble::tibble(ds = ds, yhat = c(20, 20, 20))
)
ensemble_forecasts(forecasts, method = "mean")
ensemble_forecasts(forecasts, method = "inverse_error", errors = list(A = 1, B = 5))
```

`fixed_holiday_catalog` *Built-in fixed-date holiday catalog*

Description

The Holidays tab's checklist of recurring, fixed-date holidays. Matches the hosted Forecast Dashboard app's catalog exactly.

Usage

```
fixed_holiday_catalog()
```

Value

A named character vector: names are `"MM-DD"` dates, values are the holiday label.

Examples

```
fixed_holiday_catalog()
```

`get_model` *Look up a registered model*

Description

Look up a registered model

Usage

```
get_model(key)
```

Arguments

`key`	Character scalar, the registry key (e.g. `"prophet"`, `"arima"`). See `list_models()` for every currently registered key.

Value

The registered model's list (`key`, `label`, `fit`, `forecast`, `to_tibble`, `supports_holidays`, `requires`, `annotate`).

Examples

```
model <- get_model("arima")
model$label
```

`googlesheet_csv_url` *Build a Google Sheets CSV export URL*

Description

Turns a link-shared Google Sheets URL (or a bare spreadsheet ID) into its CSV export endpoint, which works for any sheet shared as "Anyone with the link can view" – no OAuth, no credential storage.

Usage

```
googlesheet_csv_url(url, gid = NULL)
```

Arguments

`url`	The pasted Google Sheets link (several shapes accepted, e.g. `.../spreadsheets/d/<ID>/edit#gid=123` or a bare spreadsheet ID.
`gid`	Optional worksheet gid. If `NULL` and `url` carries one, that one is reused; otherwise the default worksheet is used.

Value

A character scalar, the CSV export URL.

Examples

```
googlesheet_csv_url("https://docs.google.com/spreadsheets/d/abc123/edit#gid=456")
```

`group_correlation_matrix` *Pairwise correlation between grouped series*

Description

Correlation between every pair of configured groups' series, aligned on `ds` (pairwise-complete, so groups that don't share every period still compare on their overlap).

Usage

```
group_correlation_matrix(grouped_series_list)
```

Arguments

`grouped_series_list`	A named list of `ds`/`y` tibbles, one per group (e.g. the app's `grouped_series()`).

Value

A tibble with columns `group_a`, `group_b`, `correlation` – upper triangle only, no self-pairs, one row per distinct pair. Empty if fewer than 2 groups are supplied.

Examples

```
ds <- as.Date("2024-01-01") + 0:9
groups <- list(
  A = tibble::tibble(ds = ds, y = 1:10),
  B = tibble::tibble(ds = ds, y = 10:1)
)
group_correlation_matrix(groups)
```

holidays_with_data *Find declared holidays with non-zero data*

Description

Days declared as holidays that nonetheless carry non-zero values – usually means the holiday wasn't actually observed at that site, or data entry continued anyway.

Usage

```
holidays_with_data(holidays, data)
```

Arguments

holidays	A tibble with a ds column (and typically holiday), the compiled holiday list.
data	A tibble with ds/y columns, the finalized dataset.

Value

A tibble with columns ds, holiday, y, sorted by y descending; empty if holidays/data are NULL/empty.

Examples

```
holidays <- tibble::tibble(ds = as.Date("2024-01-01"), holiday = "New Year")
data <- tibble::tibble(ds = as.Date("2024-01-01"), y = 5)
holidays_with_data(holidays, data)
```

holiday_contingency *Run both holiday consistency checks at once*

Description

Bundles holidays_with_data() and always_zero_dates() – the app's full "consistency check."

Usage

```
holiday_contingency(holidays, data)
```

Arguments

holidays	A tibble with a ds column, the compiled holiday list.
data	A tibble with ds/y columns, the finalized dataset.

Value

A list with `with_data` (see `holidays_with_data()`) and `always_zero` (see `always_zero_dates()`).

Examples

```
holidays <- tibble::tibble(ds = as.Date("2024-01-01"), holiday = "New Year")
data <- tibble::tibble(ds = as.Date("2024-01-01"), y = 5)
holiday_contingency(holidays, data)
```

`holiday_limitation_note` *Holiday-limitation note for a model*

Description

Shown in the UI whenever the selected model's `supports_holidays` is `FALSE`, mirroring the wording/style already used in the hosted app's ARIMA tab.

Usage

```
holiday_limitation_note(model_label)
```

Arguments

`model_label`	Character scalar, the model's display label.

Value

A one-line character string.

Examples

```
holiday_limitation_note("ARIMA")
```

`impute_anomalies` *Impute detected anomalies before fitting*

Description

Runs `detect_anomalies()`, replaces flagged points with `NA`, and re-interpolates with the same fill idiom used throughout this package (`zoo::na.approx()`, remaining edge `NA`s filled with the series mean) – a cleaned series to fit on instead of the raw one, so a handful of spikes/dips don't distort the fitted model the way they would untouched. Distinct from `decompose_series()`'s `robust` argument, which downweights outliers during decomposition rather than removing them before fitting; the two can be used independently or together.

Usage

```
impute_anomalies(
  df,
  date_agg = "day",
  method = c("iqr", "zscore"),
  threshold = 1.5
)
```

Arguments

df	A tibble with ds/y columns.
date_agg	Aggregation frequency: one of "hour", "day", "week", "month", "quarter", "year".
method	"iqr" (the default) or "zscore" – passed through to detect_anomalies().
threshold	Numeric sensitivity threshold, passed through to detect_anomalies(). Default 1.5.

Value

A tibble with columns ds, y (the cleaned series). The count of points that were flagged and replaced is attached as attr(result, "n_imputed").

Examples

```
df <- tibble::tibble(ds = as.Date("2024-01-01") + 0:29, y = c(rep(10, 29), 100))
impute_anomalies(df, date_agg = "day", method = "iqr")
```

---

list_models *List every registered model*

Description

List every registered model

Usage

```
list_models(available_only = TRUE)
```

Arguments

available_only	Logical; if TRUE (the default), omit any model whose requires package (see register_model()) isn't installed – e.g. "lstm" is omitted until torch is available.

Value

A list of registered model entries (see get_model() for the shape of each one).

Examples

```
vapply(list_models(available_only = TRUE), function(m) m$key, character(1))
```

`lstm_available` *Check whether the LSTM model is available*

Description

The `"lstm"` model requires the optional `torch` package. This checks whether it's installed, without loading it.

Usage

```
lstm_available()
```

Value

`TRUE` if `torch` is installed, `FALSE` otherwise.

Examples

```
lstm_available()
```

`plot_forecast_generic` *Plot a single model's forecast*

Description

Model-agnostic plotly plotting: works off just `ds`/`yhat`(+intervals) for models that only produce that, and additionally draws Prophet's trend/holiday/changepoint decomposition when `forecast_df` still has those columns (pass the adapter's raw `forecast()` output for that; pass the standardized `to_tibble()` output to metrics functions instead).

Usage

```
plot_forecast_generic(
  forecast_df,
  train_df = NULL,
  model_obj = NULL,
  subtitle = NULL,
  title = "Forecast",
  show_trend = TRUE,
  show_uncertainty = TRUE,
  show_holidays = FALSE,
  show_changepoints = FALSE,
  max_marker_lines = 60,
```

```
  color_actual = "#1b9e77",
  color_forecast = "#d95f02",
  color_trend = "#7570b3",
  color_ci = "#1b9e77"
)
```

Arguments

forecast_df	A tibble with ds/yhat (and optionally yhat_lower/yhat_upper/trend/holidays) columns.
train_df	Optional tibble with ds/y columns, drawn as the "Actual" line.
model_obj	Optional fitted model object; only used to read $changepoints when show_changepoints = TRUE.
subtitle	Optional character scalar, e.g. an ARIMA/SARIMA annotate() result like "ARIMA(2,1,1)(1,0,0)[12]".
title	Character scalar, the plot title. Default "Forecast".
show_trend, show_uncertainty, show_holidays, show_changepoints	Logical toggles for each optional layer.
max_marker_lines	Integer cap on how many holiday/changepoint marker lines are drawn (evenly subsampled beyond this), so a long horizon with recurring holidays doesn't render as unreadable clutter.
color_actual, color_forecast, color_trend, color_ci	Hex color strings for each line/ribbon.

Value

A plotly htmlwidget.

Examples

```
ds <- as.Date("2024-01-01") + 0:9
forecast_df <- tibble::tibble(ds = ds, yhat = 1:10)
plot_forecast_generic(forecast_df)
```

---

plot_model_comparison *Overlay several models' forecasts on one plot*

Description

Used for the "Compare Selected Models" flow (and forecast ensembling, which adds one more named entry).

Usage

```
plot_model_comparison(forecasts, train_df = NULL, title = "Model Comparison")
```

Arguments

forecasts	A named list of ds/yhat tibbles; names are used as the legend labels.
train_df	Optional tibble with ds/y columns, drawn as the "Actual" line.
title	Character scalar, the plot title. Default "Model Comparison".

Value

A plotly htmlwidget.

Examples

```
ds <- as.Date("2024-01-01") + 0:9
forecasts <- list(
  ARIMA = tibble::tibble(ds = ds, yhat = 1:10),
  ETS = tibble::tibble(ds = ds, yhat = 1:10 + 1)
)
plot_model_comparison(forecasts)
```

---

process_uploaded_data *Turn an uploaded table into a ds/y forecasting series*

Description

The Import tab's core transform: renames the user's chosen date/value columns to ds/y, aggregates individual-observation data into periods, and optionally normalizes by a population table to produce incidence rather than absolute counts.

Usage

```
process_uploaded_data(
  data,
  type = c("agg", "individual"),
  date_col,
  value_col = NULL,
  group_col = NULL,
  pop_df = NULL,
  pop_date_col = NULL,
  pop_value_col = NULL,
  pop_group_col = NULL,
  unit_divisor = 1,
  pop_multiplier = 1,
  date_agg = "day",
  pop_freq = "year"
)
```

Arguments

data	A data frame – the uploaded main dataset.
type	"agg" (each row is already one value per period) or "individual" (each row is a single event, counted per period).
date_col	Character scalar, the name of data's date column.
value_col	Character scalar, the name of data's value column (required when type = "agg", ignored for "individual").
group_col	Optional character scalar, a grouping column (e.g. District) to carry through untouched alongside ds/y.
pop_df	Optional population data frame for normalization.
pop_date_col, pop_value_col	Character scalars naming pop_df's key and value columns (required together with pop_df to normalize).
pop_group_col	Optional character scalar, pop_df's grouping column, if it has one.
unit_divisor, pop_multiplier	Numeric scaling applied to the population figure (e.g. unit_divisor = 100000 for per-100k).
date_agg	Aggregation frequency: one of "hour", "day", "week", "month", "quarter", "year".
pop_freq	Whether pop_df's value is given "year"ly or "month"ly.

Value

A tibble with columns ds, y, and group_col if supplied.

Examples

```
df <- data.frame(when = as.Date("2024-01-01") + 0:9, count = 1:10)
process_uploaded_data(df, type = "agg", date_col = "when", value_col = "count")
```

---

prophet_available *Check whether the Prophet model is available*

Description

The "prophet" model requires the optional prophet package. This checks whether it's installed, without loading it.

Usage

```
prophet_available()
```

Value

TRUE if prophet is installed, FALSE otherwise.

Examples

```
prophet_available()
```

`read_google_sheet` *Read a link-shared Google Sheet as a data frame*

Description

Requires the sheet be shared as "Anyone with the link can view." Makes a real network request via `googlesheet_csv_url()` – not runnable on CRAN's offline check machines.

Usage

```
read_google_sheet(url, gid = NULL)
```

Arguments

`url`	The pasted Google Sheets link, or a bare spreadsheet ID.
`gid`	Optional worksheet gid (see `googlesheet_csv_url()`).

Value

A data frame, as read by `utils::read.csv()`.

Examples

```
## Not run:
read_google_sheet("https://docs.google.com/spreadsheets/d/abc123/edit")

## End(Not run)
```

`recommend_model` *Rank registered models for a series*

Description

Scores every candidate model against `analyze_series()`'s measured trend/seasonality/regularity, with short human-readable reasons – no model is actually fit, so this is fast enough to run right after a dataset is finalized.

Usage

```
recommend_model(analysis, holidays_configured = FALSE, candidates = NULL)
```

Arguments

analysis	The list returned by analyze_series().
holidays_configured	Logical; whether holidays are configured for this dataset. Every model except Prophet is scored down when TRUE, since none of the others model holiday effects.
candidates	Character vector of registry keys to score (see list_models()), or NULL (the default) to score every currently available registered model.

Value

A data frame with columns model (label), key, score, and reason, sorted by score descending.

Examples

```
df <- tibble::tibble(ds = as.Date("2024-01-01") + 0:59, y = 1:60 + rnorm(60))
recommend_model(analyze_series(df, "day"))
```

---

reconcile_bottom_up *Bottom-up hierarchical reconciliation*

Description

Sums already-fit per-group forecasts (and their actuals) instead of fitting the aggregate separately – the sum of individually-coherent group forecasts is itself always coherent by construction. Point forecasts sum exactly; prediction intervals are summed too, under the standard (if approximate) bottom-up assumption that group forecast errors are perfectly correlated – flagged as a known simplification, not a bug.

Usage

```
reconcile_bottom_up(fits)
```

Arguments

fits	A named list of fit-result lists (one per group), each with $fc_tib (a tibble with ds, yhat, and optionally yhat_lower/yhat_upper) and $train/$test (tibbles with ds, y). A NULL entry is dropped; a present entry missing a field is skipped for that field only, rather than erroring the whole reconciliation.

Value

A list with fc_tib, train, test (each summed by ds across groups, or NULL if no group supplied that field), and components (the names of the groups that were actually combined).

Examples

```
ds <- as.Date("2024-01-01") + 0:2
fits <- list(
  A = list(fc_tib = tibble::tibble(ds = ds, yhat = c(10, 11, 12)),
           train = tibble::tibble(ds = ds, y = c(9, 10, 11))),
  B = list(fc_tib = tibble::tibble(ds = ds, yhat = c(1, 2, 3)),
           train = tibble::tibble(ds = ds, y = c(1, 1, 1)))
)
reconcile_bottom_up(fits)
```

---

reconcile_top_down *Top-down hierarchical reconciliation (historical proportions)*

Description

Disaggregates one already-fit aggregate-level forecast down to each group by that group's fixed historical share of the total, instead of fitting (or summing) each group's own forecast shape the way `reconcile_bottom_up()` does. Coherent by construction (the shares sum to 1), and a meaningfully different result from bottom-up whenever the aggregate-level model captures dynamics the per-group models miss (or vice versa) – not just the same total recomputed a different way.

Usage

```
reconcile_top_down(fits, aggregate_forecast)
```

Arguments

`fits`	A named list of fit-result lists (one per group), each with `$train` (a tibble with ds/y) used only to compute historical proportions – `$fc_tib` is not used here (unlike `reconcile_bottom_up()`, which sums it). A `NULL` entry is dropped. A group whose historical total is zero/NA falls back to an equal share rather than a zero share.
`aggregate_forecast`	A tibble with ds/yhat columns (and optionally `yhat_lower`/`yhat_upper`) – a forecast fit directly on the summed series, not derived from `fits`.

Value

A list with `fc_tib` (`aggregate_forecast`'s own columns, unchanged – the top-down total *is* the aggregate-level forecast), `train`/`test` (`NULL`; the caller already has the aggregate-level train/test that produced `aggregate_forecast` and should use those directly for display), `components` (the names of the groups whose historical shares were used), and `proportions` (a named list, one fixed share per group, summing to 1).

Examples

```
ds <- as.Date("2024-01-01") + 0:2
fits <- list(
  A = list(train = tibble::tibble(ds = ds, y = c(9, 10, 11))),
  B = list(train = tibble::tibble(ds = ds, y = c(1, 1, 1)))
)
aggregate_forecast <- tibble::tibble(ds = ds, yhat = c(10, 11, 12))
reconcile_top_down(fits, aggregate_forecast)
```

`register_model` *Register a forecasting model*

Description

Adds a model to the package's pluggable registry, keyed by `key`. Every model implements the same three-function contract (`fit`/`forecast`/ `to_tibble`), so once registered it works everywhere in the bundled app (Model tab, Compare Selected Models, Cross-Validation, the Backtest Leaderboard, the Show Code panel) with no other code changes.

Usage

```
register_model(
  key,
  label,
  fit,
  forecast,
  to_tibble,
  supports_holidays = FALSE,
  requires = NULL,
  annotate = NULL
)
```

Arguments

`key`	Character scalar, a unique registry key (e.g. `"arima"`).
`label`	Character scalar, the human-readable name shown in the UI.
`fit`	Function with signature `fit(train_df, date_agg, ...)` returning a fitted model object.
`forecast`	Function with signature `forecast(model_obj, h, ...)` returning the model's raw forecast object.
`to_tibble`	Function with signature `to_tibble(raw_forecast_obj, test_df)` returning a tibble with columns `ds`, `yhat`, and optionally `yhat_lower`/`yhat_upper`.
`supports_holidays`	Logical; whether `fit()` accepts a `holidays_df` argument and models holiday effects. Default `FALSE`.

requires	Character scalar naming an optional package this model needs (e.g. "torch"), or NULL if it has no extra dependency. list_models(available_only = TRUE) omits the model until that package is installed.
annotate	Optional function with signature annotate(model_obj) returning a one-line character description of the fitted model (e.g. "ARIMA(2,1,1)(1,0,0)[12]"), shown as a plot subtitle.

Value

TRUE, invisibly.

Examples

```
register_model(
  key = "naive_last",
  label = "Naive (last value)",
  fit = function(train_df, date_agg = "day", ...) utils::tail(train_df$y, 1),
  forecast = function(model_obj, h, ...) list(mean = rep(model_obj, h)),
  to_tibble = function(fc_obj, test_df) {
    tibble::tibble(ds = test_df$ds[seq_along(fc_obj$mean)], yhat = fc_obj$mean)
  }
)
```

---

run_app *Launch the bundled local forecastsuite Shiny app*

Description

Unlike the hosted Forecast Dashboard app, this exposes every registered, available model (Prophet, ARIMA, SARIMA, ETS, TBATS, NNETAR, Holt-Winters, and LSTM if torch is installed) with Plotly always on and no memory-conserving toggles, since it targets local/offline use rather than a free hosting tier.

Usage

```
run_app(...)
```

Arguments

...	Passed through to shiny::shinyApp(options = list(...)).

Details

The Data Import tab's "Global environment" source only reads data frames; it never writes. By default it lists them from the global environment. Set options(forecastsuite.data_env = <environment>) to point it at a different environment instead.

Value

A shiny.appobj, as returned by shiny::shinyApp(). Printing it (e.g. calling run_app() at the console, or letting it auto-print) launches the app; it does not block on its own.

Examples

```
if (interactive()) {
  run_app()
}
```

---

run_backtest_leaderboard *Multi-model rolling-origin backtest leaderboard*

Description

Runs build_cv_folds() once and refits every model in model_keys across every fold, ranking them by mean test MASE – a real multi-model extension of single-model cross-validation. A fold a model fails on is dropped for that model rather than failing the whole row; a model with zero successful folds (including an invalid model_keys entry) gets an NA row instead of being dropped, so it's still visible in the result as "failed."

Usage

```
run_backtest_leaderboard(
  model_keys,
  df,
  date_agg,
  horizon_periods,
  k_requested,
  build_args = function(model_key, train_df) {
     list(train_df = train_df, date_agg =
    date_agg)
 },
  window = c("expanding", "rolling"),
  train_window = NULL
)
```

Arguments

model_keys	Character vector of registry keys (see list_models()) to include in the leaderboard.
df	A data frame with ds/y columns – the series to backtest.
date_agg	Aggregation frequency (e.g. "day", "month"), passed through to each model's fit().
horizon_periods	Integer, each fold's test window length (see build_cv_folds()).

k_requested	Integer, the number of folds requested (may return fewer; see build_cv_folds()).
build_args	A function function(model_key, train_df) returning the full argument list for that model's fit() call (train_df/ date_agg included). The default passes train_df/date_agg through with no model-specific tuning; callers wanting each model's currently-configured parameters (e.g. the bundled app, which passes its own build_fit_args()) can supply their own.
window	"expanding" (the default) or "rolling", passed through to build_cv_folds() – see there for what each means.
train_window	Integer or NULL, passed through to build_cv_folds() when window = "rolling".

Value

A tibble with one row per model in model_keys, columns model (label), key, n_folds, mean_mase, sd_mase, mean_smape, mean_rmse, ranked ascending by mean_mase (NA last).

Examples

```
df <- data.frame(ds = as.Date("2024-01-01") + 0:59,
                 y = 100 + seq_len(60) * 0.2 + stats::rnorm(60, sd = 2))

run_backtest_leaderboard(c("arima", "ets"), df, date_agg = "day",
                         horizon_periods = 5, k_requested = 2)
```

---

safe_compute_metrics *Compute forecast metrics, guarding against a too-short test set*

Description

Same as compute_forecast_metrics(), but checks actual_df's row count first so callers don't need their own length guard before calling it.

Usage

```
safe_compute_metrics(forecast_df, actual_df, label)
```

Arguments

forecast_df	A tibble with ds/yhat columns.
actual_df	A tibble with ds/y columns.
label	Character scalar identifying this evaluation.

Value

A tibble with columns Set, Metric, Value (see compute_forecast_metrics()).

Examples

```
ds <- as.Date("2024-01-01") + 0:9
forecast_df <- tibble::tibble(ds = ds, yhat = 1:10 + 0.5)
actual_df <- tibble::tibble(ds = ds, y = 1:10)
safe_compute_metrics(forecast_df, actual_df, label = "Test")
```

---

split_by_group *Split a ds/y/group tibble into one tibble per group*

Description

Split a ds/y/group tibble into one tibble per group

Usage

```
split_by_group(df, group_col)
```

Arguments

df	A data frame with ds, y, and group_col columns.
group_col	Character scalar, the grouping column's name, or NULL (returns NULL – no grouping).

Value

A named list of ds/y tibbles, one per distinct value of group_col (names sorted), or NULL if group_col is NULL or not present in df.

Examples

```
df <- data.frame(ds = as.Date("2024-01-01") + 0:3,
                 y = 1:4, District = c("A", "A", "B", "B"))
split_by_group(df, "District")
```

---

suggest_parameters *Suggest starting hyperparameters for one model*

Description

Derives starting-point hyperparameters from analyze_series()'s diagnostics rather than fitting a grid of candidates. These are informed starting points, not optimal values – for ARIMA/SARIMA, auto.arima() still searches the order space better than this heuristic; the suggested p/q matter mainly for hand-tuning, while d/D come straight from ndiffs()/nsdiffs().

Usage

```
suggest_parameters(analysis, model_key)
```

Arguments

analysis	The list returned by analyze_series().
model_key	A registry key, e.g. "prophet", "arima", "sarima". Models with no tunable parameters return an empty suggestion.

Value

A list with params (a named list of suggested argument values, matching that model's fit() parameters), text (a one-line human-readable summary), and reasons (a character vector explaining each suggestion).

Examples

```
df <- tibble::tibble(ds = as.Date("2024-01-01") + 0:59, y = 1:60 + rnorm(60))
suggest_parameters(analyze_series(df, "day"), "arima")
```

---

suggest_parameters_for

Suggest starting hyperparameters for several models at once

Description

Vectorized suggest_parameters(), for the recommendation table's "suggested settings" column.

Usage

```
suggest_parameters_for(analysis, model_keys)
```

Arguments

analysis	The list returned by analyze_series().
model_keys	Character vector of registry keys.

Value

A character vector, the same length as model_keys, of each model's suggestion text.

Examples

```
df <- tibble::tibble(ds = as.Date("2024-01-01") + 0:59, y = 1:60 + rnorm(60))
suggest_parameters_for(analyze_series(df, "day"), c("arima", "prophet"))
```

`ts_frequency_for` *Map an aggregation frequency to a ts() seasonal frequency*

Description

Map an aggregation frequency to a ts() seasonal frequency

Usage

```
ts_frequency_for(date_agg)
```

Arguments

`date_agg`	One of `"hour"`, `"day"`, `"week"`, `"month"`, `"quarter"`, `"year"`; anything else (including `NULL`) defaults to `"day"`.

Value

A single integer, suitable for `stats::ts(y, frequency = ...)`.

Examples

```
ts_frequency_for("month")
```

Index

always_zero_dates, 3
always_zero_dates(), *19, 20*
analyze_series, 4
analyze_series(), *26, 27*, *33, 34*

build_cv_folds, 5
build_cv_folds(), *31, 32*

collapse_to_period, 6
collapse_to_period(), *12*
compose_date_parts, 7
compute_forecast_bias, 8
compute_forecast_bias(), *15*
compute_forecast_metrics, 8
compute_forecast_metrics(), *32*
compute_group_value_map, 9
compute_interval_coverage, 10
compute_multi_window_metrics, 10
compute_residual_diagnostics, 11
convert_months_to_horizon, 12
count_duplicate_periods, 12

decompose_series, 13
decompose_series(), *14*, *20*
detect_anomalies, 14
detect_anomalies(), *20, 21*
detect_bias_drift, 15

ensemble_forecasts, 15

fixed_holiday_catalog, 16

get_model, 17
get_model(), *21*
googlesheet_csv_url, 17
googlesheet_csv_url(), *26*
group_correlation_matrix, 18

holiday_contingency, 19
holiday_limitation_note, 20
holidays_with_data, 19
holidays_with_data(), *19, 20*

impute_anomalies, 20

list_models, 21
list_models(), *17*, *27*, *31*
lstm_available, 22

plot_forecast_generic, 22
plot_model_comparison, 23
process_uploaded_data, 24
prophet_available, 25

read_google_sheet, 26
recommend_model, 26
recommend_model(), *4*
reconcile_bottom_up, 27
reconcile_bottom_up(), *28*
reconcile_top_down, 28
register_model, 29
register_model(), *21*
run_app, 30
run_backtest_leaderboard, 31

safe_compute_metrics, 32
safe_compute_metrics(), *10*, *16*
split_by_group, 33
suggest_parameters, 33
suggest_parameters(), *34*
suggest_parameters_for, 34

ts_frequency_for, 35