Package ‘modeltime’

December 17, 2025

Title The Tidymodels Extension for Time Series Modeling

Version 1.3.3

Description The time series forecasting framework for use with the 'tidymodels' ecosystem. Models include ARIMA, Exponential Smoothing, and additional time series models from the 'forecast' and 'prophet' packages. Refer to ``Forecasting Principles & Practice, Second edition'' (<https://otexts.com/fpp2/>). Refer to ``Prophet: forecasting at scale'' (<https://research.facebook.com/blog/2017/02/prophet-forecasting-at-scale/>.).

URL https://github.com/business-science/modeltime,
https://business-science.github.io/modeltime/

BugReports https://github.com/business-science/modeltime/issues

License MIT + file LICENSE

Encoding UTF-8

LazyData true

Depends R (>= 3.5.0)

Imports StanHeaders, timetk (>= 2.8.1), parsnip (>= 0.2.1), dials, yardstick (>= 0.0.8), workflows (>= 1.0.0), hardhat (>= 1.0.0), rlang (>= 0.1.2), glue, plotly, reactable, gt, ggplot2, tibble, tidyr, dplyr (>= 1.1.0), purrr, stringr, forcats, scales, janitor, parallel, parallelly, doParallel, foreach, magrittr, forecast, xgboost (>= 1.2.0.1), prophet, methods, cli, tidymodels

Suggests rstan, slider, sparklyr, doRNG, workflowsets, recipes, rsample, tune (>= 0.2.0), lubridate, testthat, kernlab, glmnet, thief, smooth, greybox, earth, randomForest, trelliscopejs, knitr, rmarkdown (>= 2.9), webshot, qpdf, TSrepr, future, doFuture

VignetteBuilder knitr

RoxygenNote 7.3.3

NeedsCompilation no

Author Matt Dancho [aut, cre],
Business Science [cph]

Maintainer Matt Dancho <mdancho@business-science.io>

Repository CRAN

Date/Publication 2025-12-17 06:40:30 UTC

Contents

`adam_params` *Tuning Parameters for ADAM Models*

Description

Tuning Parameters for ADAM Models

Usage

```
ets_model(values = c("ZZZ", "XXX", "YYY", "CCC", "PPP", "FFF"))

loss(
  values = c("likelihood", "MSE", "MAE", "HAM", "LASSO", "RIDGE", "TMSE", "GTMSE",
    "MSEh", "MSCE")
)

use_constant(values = c(FALSE, TRUE))

regressors_treatment(values = c("use", "select", "adapt"))

outliers_treatment(values = c("ignore", "use", "select"))
```

```
probability_model(
  values = c("none", "auto", "fixed", "general", "odds-ratio", "inverse-odds-ratio",
    "direct")
)

distribution(
  values = c("default", "dnorm", "dlaplace", "ds", "dgnorm", "dlnorm", "dinvgauss",
    "dgamma")
)

information_criteria(values = c("AICc", "AIC", "BICc", "BIC"))

select_order(values = c(FALSE, TRUE))
```

Arguments

`values`	A character string of possible values.

Details

The main parameters for ADAM models are:

- `ets_model`:
  - model="ZZZ" means that the model will be selected based on the chosen information criteria type. The Branch and Bound is used in the process.
  - model="XXX" means that only additive components are tested, using Branch and Bound.
  - model="YYY" implies selecting between multiplicative components.
  - model="CCC" triggers the combination of forecasts of models using information criteria weights (Kolassa, 2011).
  - combinations between these four and the classical components are also accepted. For example, model="CAY" will combine models with additive trend and either none or multiplicative seasonality.
  - model="PPP" will produce the selection between pure additive and pure multiplicative models. "P" stands for "Pure". This cannot be mixed with other types of components.
  - model="FFF" will select between all the 30 types of models. "F" stands for "Full". This cannot be mixed with other types of components.
  - The parameter model can also be a vector of names of models for a finer tuning (pool of models). For example, model=c("ANN","AAA") will estimate only two models and select the best of them.
- `loss`:
  - likelihood - the model is estimated via the maximization of the likelihood of the function specified in distribution;
  - MSE (Mean Squared Error),
  - MAE (Mean Absolute Error),
  - HAM (Half Absolute Moment),
  - LASSO - use LASSO to shrink the parameters of the model;

  - RIDGE - use RIDGE to shrink the parameters of the model;
  - TMSE - Trace Mean Squared Error,
  - GTMSE - Geometric Trace Mean Squared Error,
  - MSEh - optimisation using only h-steps ahead error,
  - MSCE - Mean Squared Cumulative Error.
- `non_seasonal_ar`: The order of the non-seasonal auto-regressive (AR) terms.
- `non_seasonal_differences`: The order of integration for non-seasonal differencing.
- `non_seasonal_ma`: The order of the non-seasonal moving average (MA) terms.
- `seasonal_ar`: The order of the seasonal auto-regressive (SAR) terms.
- `seasonal_differences`: The order of integration for seasonal differencing.
- `seasonal_ma`: The order of the seasonal moving average (SMA) terms.
- `use_constant`: Logical, determining, whether the constant is needed in the model or not.
- `regressors_treatment`: The variable defines what to do with the provided explanatory variables.
- `outliers_treatment`: Defines what to do with outliers.
- `probability_model`: The type of model used in probability estimation.
- `distribution`: What density function to assume for the error term.
- `information_criteria`: The information criterion to use in the model selection / combination procedure.
- `select_order`: If TRUE, then the function will select the most appropriate order.

Value

A `dials` parameter

A parameter

A parameter

A parameter

A parameter

A parameter

A parameter

A parameter

A parameter

A parameter

Examples

```
use_constant()

regressors_treatment()

distribution()
```

adam_reg *General Interface for ADAM Regression Models*

Description

`adam_reg()` is a way to generate a *specification* of an ADAM model before fitting and allows the model to be created using different packages. Currently the only package is `smooth`.

Usage

```
adam_reg(
  mode = "regression",
  ets_model = NULL,
  non_seasonal_ar = NULL,
  non_seasonal_differences = NULL,
  non_seasonal_ma = NULL,
  seasonal_ar = NULL,
  seasonal_differences = NULL,
  seasonal_ma = NULL,
  use_constant = NULL,
  regressors_treatment = NULL,
  outliers_treatment = NULL,
  outliers_ci = NULL,
  probability_model = NULL,
  distribution = NULL,
  loss = NULL,
  information_criteria = NULL,
  seasonal_period = NULL,
  select_order = NULL
)
```

Arguments

`mode` A single character string for the type of model. The only possible value for this model is "regression".

`ets_model` The type of ETS model. The first letter stands for the type of the error term ("A" or "M"), the second (and sometimes the third as well) is for the trend ("N", "A", "Ad", "M" or "Md"), and the last one is for the type of seasonality ("N", "A" or "M").

`non_seasonal_ar` The order of the non-seasonal auto-regressive (AR) terms. Often denoted "p" in pdq-notation.

`non_seasonal_differences` The order of integration for non-seasonal differencing. Often denoted "d" in pdq-notation.

non_seasonal_ma
: The order of the non-seasonal moving average (MA) terms. Often denoted "q" in pdq-notation.

seasonal_ar
: The order of the seasonal auto-regressive (SAR) terms. Often denoted "P" in PDQ-notation.

seasonal_differences
: The order of integration for seasonal differencing. Often denoted "D" in PDQ-notation.

seasonal_ma
: The order of the seasonal moving average (SMA) terms. Often denoted "Q" in PDQ-notation.

use_constant
: Logical, determining, whether the constant is needed in the model or not. This is mainly needed for ARIMA part of the model, but can be used for ETS as well.

regressors_treatment
: The variable defines what to do with the provided explanatory variables: "use" means that all of the data should be used, while "select" means that a selection using ic should be done, "adapt" will trigger the mechanism of time varying parameters for the explanatory variables.

outliers_treatment
: Defines what to do with outliers: "ignore", so just returning the model, "detect" outliers based on specified level and include dummies for them in the model, or detect and "select" those of them that reduce ic value.

outliers_ci
: What confidence level to use for detection of outliers. Default is 99%.

probability_model
: The type of model used in probability estimation. Can be "none" - none, "fixed" - constant probability, "general" - the general Beta model with two parameters, "odds-ratio" - the Odds-ratio model with b=1 in Beta distribution, "inverse-odds-ratio" - the model with a=1 in Beta distribution, "direct" - the TSB-like (Teunter et al., 2011) probability update mechanism a+b=1, "auto" - the automatically selected type of occurrence model.

distribution
: what density function to assume for the error term. The full name of the distribution should be provided, starting with the letter "d" - "density".

loss
: The type of Loss Function used in optimization.

information_criteria
: The information criterion to use in the model selection / combination procedure.

seasonal_period
: A seasonal frequency. Uses "auto" by default. A character phrase of "auto" or time-based phrase of "2 weeks" can be used if a date or date-time variable is provided. See Fit Details below.

select_order
: If `TRUE`, then the function will select the most appropriate order. The values list(ar=...,i=...,ma=...) specify the maximum orders to check in this case.

Details

The data given to the function are not saved and are only used to determine the *mode* of the model. For `adam_reg()`, the mode will always be "regression".

The model can be created using the `fit()` function using the following *engines*:

- "auto_adam" (default) - Connects to smooth::auto.adam()
- "adam" - Connects to smooth::adam()

Main Arguments

The main arguments (tuning parameters) for the model are:

- seasonal_period: The periodic nature of the seasonality. Uses "auto" by default.
- non_seasonal_ar: The order of the non-seasonal auto-regressive (AR) terms.
- non_seasonal_differences: The order of integration for non-seasonal differencing.
- non_seasonal_ma: The order of the non-seasonal moving average (MA) terms.
- seasonal_ar: The order of the seasonal auto-regressive (SAR) terms.
- seasonal_differences: The order of integration for seasonal differencing.
- seasonal_ma: The order of the seasonal moving average (SMA) terms.
- ets_model: The type of ETS model.
- use_constant: Logical, determining, whether the constant is needed in the model or not.
- regressors_treatment: The variable defines what to do with the provided explanatory variables.
- outliers_treatment: Defines what to do with outliers.
- probability_model: The type of model used in probability estimation.
- distribution: what density function to assume for the error term.
- loss: The type of Loss Function used in optimization.
- information_criteria: The information criterion to use in the model selection / combination procedure.

These arguments are converted to their specific names at the time that the model is fit.

Other options and argument can be set using set_engine() (See Engine Details below).

If parameters need to be modified, update() can be used in lieu of recreating the object from scratch.

auto_adam (default engine)

The engine uses smooth::auto.adam().

Function Parameters:

```
#> function (data, model = "ZXZ", lags = c(frequency(data)), orders = list(ar = c(3,
#>   3), i = c(2, 1), ma = c(3, 3), select = TRUE), formula = NULL, regressors = c("use",
#>     "select", "adapt"), occurrence = c("none", "auto", "fixed", "general",
#>     "odds-ratio", "inverse-odds-ratio", "direct"), distribution = c("dnorm",
#>   "dlaplace", "ds", "dgnorm", "dlnorm", "dinvgauss", "dgamma"), outliers = c("ignore",
#>    "use", "select"), level = 0.99, h = 0, holdout = FALSE, persistence = NULL,
#>    phi = NULL, initial = c("backcasting", "optimal", "two-stage", "complete"),
#>     arma = NULL, ic = c("AICc", "AIC", "BIC", "BICc"), bounds = c("usual",
#>      "admissible", "none"), silent = TRUE, parallel = FALSE, ets = c("conventional",
#>         "adam"), ...)
```

The *MAXIMUM* nonseasonal ARIMA terms (max.p, max.d, max.q) and seasonal ARIMA terms (max.P, max.D, max.Q) are provided to forecast::auto.arima() via arima_reg() parameters. Other options and argument can be set using set_engine().

Parameter Notes:

- All values of nonseasonal pdq and seasonal PDQ are maximums. The smooth::auto.adam() model will select a value using these as an upper limit.
- xreg - This is supplied via the parsnip / modeltime fit() interface (so don't provide this manually). See Fit Details (below).

adam

The engine uses smooth::adam().

Function Parameters:

```
#> function (data, model = "ZXZ", lags = c(frequency(data)), orders = list(ar = c(0),
#>     i = c(0), ma = c(0), select = FALSE), constant = FALSE, formula = NULL,
#>     regressors = c("use", "select", "adapt"), occurrence = c("none", "auto",
#>         "fixed", "general", "odds-ratio", "inverse-odds-ratio", "direct"),
#>     distribution = c("default", "dnorm", "dlaplace", "ds", "dgnorm", "dlnorm",
#>         "dinvgauss", "dgamma"), loss = c("likelihood", "MSE", "MAE", "HAM",
#>         "LASSO", "RIDGE", "MSEh", "TMSE", "GTMSE", "MSCE"), outliers = c("ignore",
#>         "use", "select"), level = 0.99, h = 0, holdout = FALSE, persistence = NULL,
#>     phi = NULL, initial = c("backcasting", "optimal", "two-stage", "complete"),
#>     arma = NULL, ic = c("AICc", "AIC", "BIC", "BICc"), bounds = c("usual",
#>         "admissible", "none"), silent = TRUE, ets = c("conventional", "adam"),
#>     ...)
```

The nonseasonal ARIMA terms (orders) and seasonal ARIMA terms (orders) are provided to smooth::adam() via adam_reg() parameters. Other options and argument can be set using set_engine().

Parameter Notes:

- xreg - This is supplied via the parsnip / modeltime fit() interface (so don't provide this manually). See Fit Details (below).

Fit Details

Date and Date-Time Variable

It's a requirement to have a date or date-time variable as a predictor. The fit() interface accepts date and date-time features and handles them internally.

- fit(y ~ date)

Seasonal Period Specification

The period can be non-seasonal (seasonal_period = 1 or "none") or yearly seasonal (e.g. For monthly time stamps, seasonal_period = 12, seasonal_period = "12 months", or seasonal_period = "yearly"). There are 3 ways to specify:

1. seasonal_period = "auto": A seasonal period is selected based on the periodicity of the data (e.g. 12 if monthly)

2. `seasonal_period = 12`: A numeric frequency. For example, 12 is common for monthly data
3. `seasonal_period = "1 year"`: A time-based phrase. For example, "1 year" would convert to 12 for monthly data.

Univariate (No xregs, Exogenous Regressors):

For univariate analysis, you must include a date or date-time feature. Simply use:

- Formula Interface (recommended): `fit(y ~ date)` will ignore xreg's.

Multivariate (xregs, Exogenous Regressors)

The `xreg` parameter is populated using the `fit()` function:

- Only `factor`, `ordered factor`, and `numeric` data will be used as xregs.
- Date and Date-time variables are not used as xregs
- `character` data should be converted to factor.

Xreg Example: Suppose you have 3 features:

1. `y` (target)
2. `date` (time stamp),
3. `month.lbl` (labeled month as a ordered factor).

The `month.lbl` is an exogenous regressor that can be passed to the `arima_reg()` using `fit()`:

- `fit(y ~ date + month.lbl)` will pass `month.lbl` on as an exogenous regressor.

Note that date or date-time class values are excluded from `xreg`.

See Also

`fit.model_spec()`, `set_engine()`

Examples

```
library(dplyr)
library(parsnip)
library(rsample)
library(timetk)
library(smooth)

# Data
m750 <- m4_monthly %>% filter(id == "M750")
m750

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.8)

# ---- AUTO ADAM ----
```

```
# Model Spec
model_spec <- adam_reg() %>%
    set_engine("auto_adam")

# Fit Spec
model_fit <- model_spec %>%
    fit(log(value) ~ date, data = training(splits))
model_fit


# ---- STANDARD ADAM ----

# Model Spec
model_spec <- adam_reg(
        seasonal_period          = 12,
        non_seasonal_ar          = 3,
        non_seasonal_differences = 1,
        non_seasonal_ma          = 3,
        seasonal_ar              = 1,
        seasonal_differences     = 0,
        seasonal_ma              = 1
    ) %>%
    set_engine("adam")

# Fit Spec
model_fit <- model_spec %>%
    fit(log(value) ~ date, data = training(splits))
model_fit
```

`add_modeltime_model` *Add a Model into a Modeltime Table*

Description

Add a Model into a Modeltime Table

Usage

```
add_modeltime_model(object, model, location = "bottom")
```

Arguments

`object`	Multiple Modeltime Tables (class `mdl_time_tbl`)
`model`	A model of class `model_fit` or a fitted `workflow` object
`location`	Where to add the model. Either "top" or "bottom". Default: "bottom".

See Also

- combine_modeltime_tables(): Combine 2 or more Modeltime Tables together
- add_modeltime_model(): Adds a new row with a new model to a Modeltime Table
- drop_modeltime_model(): Drop one or more models from a Modeltime Table
- update_modeltime_description(): Updates a description for a model inside a Modeltime Table
- update_modeltime_model(): Updates a model inside a Modeltime Table
- pull_modeltime_model(): Extracts a model from a Modeltime Table

Examples

```
library(tidymodels)

model_fit_ets <- exp_smoothing() %>%
    set_engine("ets") %>%
    fit(value ~ date, training(m750_splits))

m750_models %>%
    add_modeltime_model(model_fit_ets)
```

---

arima_boost *General Interface for "Boosted" ARIMA Regression Models*

Description

arima_boost() is a way to generate a *specification* of a time series model that uses boosting to improve modeling errors (residuals) on Exogenous Regressors. It works with both "automated" ARIMA (auto.arima) and standard ARIMA (arima). The main algorithms are:

- Auto ARIMA + XGBoost Errors (engine = auto_arima_xgboost, default)
- ARIMA + XGBoost Errors (engine = arima_xgboost)

Usage

```
arima_boost(
  mode = "regression",
  seasonal_period = NULL,
  non_seasonal_ar = NULL,
  non_seasonal_differences = NULL,
  non_seasonal_ma = NULL,
  seasonal_ar = NULL,
  seasonal_differences = NULL,
  seasonal_ma = NULL,
  mtry = NULL,
```

```
    trees = NULL,
    min_n = NULL,
    tree_depth = NULL,
    learn_rate = NULL,
    loss_reduction = NULL,
    sample_size = NULL,
    stop_iter = NULL
)
```

Arguments

mode	A single character string for the type of model. The only possible value for this model is "regression".
seasonal_period	A seasonal frequency. Uses "auto" by default. A character phrase of "auto" or time-based phrase of "2 weeks" can be used if a date or date-time variable is provided. See Fit Details below.
non_seasonal_ar	The order of the non-seasonal auto-regressive (AR) terms. Often denoted "p" in pdq-notation.
non_seasonal_differences	The order of integration for non-seasonal differencing. Often denoted "d" in pdq-notation.
non_seasonal_ma	The order of the non-seasonal moving average (MA) terms. Often denoted "q" in pdq-notation.
seasonal_ar	The order of the seasonal auto-regressive (SAR) terms. Often denoted "P" in PDQ-notation.
seasonal_differences	The order of integration for seasonal differencing. Often denoted "D" in PDQ-notation.
seasonal_ma	The order of the seasonal moving average (SMA) terms. Often denoted "Q" in PDQ-notation.
mtry	A number for the number (or proportion) of predictors that will be randomly sampled at each split when creating the tree models (specific engines only).
trees	An integer for the number of trees contained in the ensemble.
min_n	An integer for the minimum number of data points in a node that is required for the node to be split further.
tree_depth	An integer for the maximum depth of the tree (i.e. number of splits) (specific engines only).
learn_rate	A number for the rate at which the boosting algorithm adapts from iteration-to-iteration (specific engines only). This is sometimes referred to as the shrinkage parameter.
loss_reduction	A number for the reduction in the loss function required to split further (specific engines only).

`sample_size`	number for the number (or proportion) of data that is exposed to the fitting routine.
`stop_iter`	The number of iterations without improvement before stopping (`xgboost` only).

Details

The data given to the function are not saved and are only used to determine the *mode* of the model. For `arima_boost()`, the mode will always be "regression".

The model can be created using the `fit()` function using the following *engines*:

- "auto_arima_xgboost" (default) - Connects to `forecast::auto.arima()` and xgboost::xgb.train
- "arima_xgboost" - Connects to `forecast::Arima()` and xgboost::xgb.train

Main Arguments

The main arguments (tuning parameters) for the **ARIMA model** are:

- `seasonal_period`: The periodic nature of the seasonality. Uses "auto" by default.
- `non_seasonal_ar`: The order of the non-seasonal auto-regressive (AR) terms.
- `non_seasonal_differences`: The order of integration for non-seasonal differencing.
- `non_seasonal_ma`: The order of the non-seasonal moving average (MA) terms.
- `seasonal_ar`: The order of the seasonal auto-regressive (SAR) terms.
- `seasonal_differences`: The order of integration for seasonal differencing.
- `seasonal_ma`: The order of the seasonal moving average (SMA) terms.

The main arguments (tuning parameters) for the model **XGBoost model** are:

- `mtry`: The number of predictors that will be randomly sampled at each split when creating the tree models.
- `trees`: The number of trees contained in the ensemble.
- `min_n`: The minimum number of data points in a node that are required for the node to be split further.
- `tree_depth`: The maximum depth of the tree (i.e. number of splits).
- `learn_rate`: The rate at which the boosting algorithm adapts from iteration-to-iteration.
- `loss_reduction`: The reduction in the loss function required to split further.
- `sample_size`: The amount of data exposed to the fitting routine.
- `stop_iter`: The number of iterations without improvement before stopping.

These arguments are converted to their specific names at the time that the model is fit.

Other options and argument can be set using `set_engine()` (See Engine Details below).

If parameters need to be modified, `update()` can be used in lieu of recreating the object from scratch.

Engine Details

The standardized parameter names in modeltime can be mapped to their original names in each engine:

Model 1: ARIMA:

modeltime	forecast::auto.arima	forecast::Arima
seasonal_period	ts(frequency)	ts(frequency)
non_seasonal_ar, non_seasonal_differences, non_seasonal_ma	max.p(5), max.d(2), max.q(5)	order = c(p(0), d(0), q(0))
seasonal_ar, seasonal_differences, seasonal_ma	max.P(2), max.D(1), max.Q(2)	seasonal = c(P(0), D(0), Q(

Model 2: XGBoost:

modeltime	xgboost::xgb.train
tree_depth	max_depth (6)
trees	nrounds (15)
learn_rate	eta (0.3)
mtry	colsample_bynode (1)
min_n	min_child_weight (1)
loss_reduction	gamma (0)
sample_size	subsample (1)
stop_iter	early_stop

Other options can be set using set_engine().

auto_arima_xgboost (default engine)

Model 1: Auto ARIMA (forecast::auto.arima):

```
#> function (y, d = NA, D = NA, max.p = 5, max.q = 5, max.P = 2, max.Q = 2,
#>     max.order = 5, max.d = 2, max.D = 1, start.p = 2, start.q = 2, start.P = 1,
#>     start.Q = 1, stationary = FALSE, seasonal = TRUE, ic = c("aicc", "aic",
#>       "bic"), stepwise = TRUE, nmodels = 94, trace = FALSE, approximation = (length(x) >
#>         150 | frequency(x) > 12), method = NULL, truncate = NULL, xreg = NULL,
#>     test = c("kpss", "adf", "pp"), test.args = list(), seasonal.test = c("seas",
#>         "ocsb", "hegy", "ch"), seasonal.test.args = list(), allowdrift = TRUE,
#>     allowmean = TRUE, lambda = NULL, biasadj = FALSE, parallel = FALSE,
#>     num.cores = 2, x = y, ...)
```

Parameter Notes:

- All values of nonseasonal pdq and seasonal PDQ are maximums. The auto.arima will select a value using these as an upper limit.
- xreg - This should not be used since XGBoost will be doing the regression

Model 2: XGBoost (xgboost::xgb.train):

```
#> function (params = xgb.params(), data, nrounds, evals = list(), objective = NULL,
#>     custom_metric = NULL, verbose = 1, print_every_n = 1L, early_stopping_rounds = NULL,
#>     maximize = NULL, save_period = NULL, save_name = "xgboost.model", xgb_model = NULL,
#>     callbacks = list(), ...)
```

Parameter Notes:

- XGBoost uses a `params = list()` to capture. Parsnip / Modeltime automatically sends any args provided as `...` inside of `set_engine()` to the `params = list(...)`.

Fit Details

Date and Date-Time Variable

It's a requirement to have a date or date-time variable as a predictor. The `fit()` interface accepts date and date-time features and handles them internally.

- `fit(y ~ date)`

Seasonal Period Specification

The period can be non-seasonal (`seasonal_period = 1`) or seasonal (e.g. `seasonal_period = 12` or `seasonal_period = "12 months"`). There are 3 ways to specify:

1. `seasonal_period = "auto"`: A period is selected based on the periodicity of the data (e.g. 12 if monthly)
2. `seasonal_period = 12`: A numeric frequency. For example, 12 is common for monthly data
3. `seasonal_period = "1 year"`: A time-based phrase. For example, "1 year" would convert to 12 for monthly data.

Univariate (No xregs, Exogenous Regressors):

For univariate analysis, you must include a date or date-time feature. Simply use:

- Formula Interface (recommended): `fit(y ~ date)` will ignore xreg's.
- XY Interface: `fit_xy(x = data[,"date"], y = data$y)` will ignore xreg's.

Multivariate (xregs, Exogenous Regressors)

The `xreg` parameter is populated using the `fit()` or `fit_xy()` function:

- Only `factor`, `ordered factor`, and `numeric` data will be used as xregs.
- Date and Date-time variables are not used as xregs
- `character` data should be converted to factor.

Xreg Example: Suppose you have 3 features:

1. `y` (target)
2. `date` (time stamp),
3. `month.lbl` (labeled month as a ordered factor).

The `month.lbl` is an exogenous regressor that can be passed to the `arima_boost()` using `fit()`:

- `fit(y ~ date + month.lbl)` will pass `month.lbl` on as an exogenous regressor.
- `fit_xy(data[,c("date", "month.lbl")], y = data$y)` will pass x, where x is a data frame containing `month.lbl` and the `date` feature. Only `month.lbl` will be used as an exogenous regressor.

Note that date or date-time class values are excluded from `xreg`.

See Also

`fit.model_spec()`, `set_engine()`

Examples

```
library(dplyr)
library(lubridate)
library(parsnip)
library(rsample)
library(timetk)

# Data
m750 <- m4_monthly %>% filter(id == "M750")

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.9)

# MODEL SPEC ----

# Set engine and boosting parameters
model_spec <- arima_boost(

    # ARIMA args
    seasonal_period = 12,
    non_seasonal_ar = 0,
    non_seasonal_differences = 1,
    non_seasonal_ma = 1,
    seasonal_ar     = 0,
    seasonal_differences = 1,
    seasonal_ma     = 1,

    # XGBoost Args
    tree_depth = 6,
    learn_rate = 0.1
) %>%
    set_engine(engine = "arima_xgboost")

# FIT ----

# Boosting - Happens by adding numeric date and month features
model_fit_boosted <- model_spec %>%
    fit(value ~ date + as.numeric(date) + month(date, label = TRUE),
        data = training(splits))
model_fit_boosted
```

`arima_params` *Tuning Parameters for ARIMA Models*

Description

Tuning Parameters for ARIMA Models

Usage

```
non_seasonal_ar(range = c(0L, 5L), trans = NULL)

non_seasonal_differences(range = c(0L, 2L), trans = NULL)

non_seasonal_ma(range = c(0L, 5L), trans = NULL)

seasonal_ar(range = c(0L, 2L), trans = NULL)

seasonal_differences(range = c(0L, 1L), trans = NULL)

seasonal_ma(range = c(0L, 2L), trans = NULL)
```

Arguments

`range`	A two-element vector holding the *defaults* for the smallest and largest possible values, respectively. If a transformation is specified, these values should be in the *transformed units*.
`trans`	A `trans` object from the `scales` package, such as `scales::transform_log10()` or `scales::transform_reciprocal()`. If not provided, the default is used which matches the units used in `range`. If no transformation, `NULL`.

Details

The main parameters for ARIMA models are:

- `non_seasonal_ar`: The order of the non-seasonal auto-regressive (AR) terms.
- `non_seasonal_differences`: The order of integration for non-seasonal differencing.
- `non_seasonal_ma`: The order of the non-seasonal moving average (MA) terms.
- `seasonal_ar`: The order of the seasonal auto-regressive (SAR) terms.
- `seasonal_differences`: The order of integration for seasonal differencing.
- `seasonal_ma`: The order of the seasonal moving average (SMA) terms.

Examples

```
ets_model()

non_seasonal_ar()

non_seasonal_differences()

non_seasonal_ma()
```

---

arima_reg *General Interface for ARIMA Regression Models*

Description

`arima_reg()` is a way to generate a *specification* of an ARIMA model before fitting and allows the model to be created using different packages. Currently the only package is `forecast`.

Usage

```
arima_reg(
  mode = "regression",
  seasonal_period = NULL,
  non_seasonal_ar = NULL,
  non_seasonal_differences = NULL,
  non_seasonal_ma = NULL,
  seasonal_ar = NULL,
  seasonal_differences = NULL,
  seasonal_ma = NULL
)
```

Arguments

`mode` A single character string for the type of model. The only possible value for this model is "regression".

`seasonal_period` A seasonal frequency. Uses "auto" by default. A character phrase of "auto" or time-based phrase of "2 weeks" can be used if a date or date-time variable is provided. See Fit Details below.

`non_seasonal_ar` The order of the non-seasonal auto-regressive (AR) terms. Often denoted "p" in pdq-notation.

`non_seasonal_differences` The order of integration for non-seasonal differencing. Often denoted "d" in pdq-notation.

`non_seasonal_ma`
: The order of the non-seasonal moving average (MA) terms. Often denoted "q" in pdq-notation.

`seasonal_ar`
: The order of the seasonal auto-regressive (SAR) terms. Often denoted "P" in PDQ-notation.

`seasonal_differences`
: The order of integration for seasonal differencing. Often denoted "D" in PDQ-notation.

`seasonal_ma`
: The order of the seasonal moving average (SMA) terms. Often denoted "Q" in PDQ-notation.

Details

The data given to the function are not saved and are only used to determine the *mode* of the model. For `arima_reg()`, the mode will always be "regression".

The model can be created using the `fit()` function using the following *engines*:

- "auto_arima" (default) - Connects to `forecast::auto.arima()`
- "arima" - Connects to `forecast::Arima()`

Main Arguments

The main arguments (tuning parameters) for the model are:

- `seasonal_period`: The periodic nature of the seasonality. Uses "auto" by default.
- `non_seasonal_ar`: The order of the non-seasonal auto-regressive (AR) terms.
- `non_seasonal_differences`: The order of integration for non-seasonal differencing.
- `non_seasonal_ma`: The order of the non-seasonal moving average (MA) terms.
- `seasonal_ar`: The order of the seasonal auto-regressive (SAR) terms.
- `seasonal_differences`: The order of integration for seasonal differencing.
- `seasonal_ma`: The order of the seasonal moving average (SMA) terms.

These arguments are converted to their specific names at the time that the model is fit.

Other options and argument can be set using `set_engine()` (See Engine Details below).

If parameters need to be modified, `update()` can be used in lieu of recreating the object from scratch.

Engine Details

The standardized parameter names in `modeltime` can be mapped to their original names in each engine:

modeltime	forecast::auto.arima	forecast::Arima
seasonal_period	ts(frequency)	ts(frequency)
non_seasonal_ar, non_seasonal_differences, non_seasonal_ma	max.p(5), max.d(2), max.q(5)	order = c(p(0), d(0), q(0))
seasonal_ar, seasonal_differences, seasonal_ma	max.P(2), max.D(1), max.Q(2)	seasonal = c(P(0), D(0), Q(

Other options can be set using `set_engine()`.

auto_arima (default engine)

The engine uses `forecast::auto.arima()`.

Function Parameters:

```
#> function (y, d = NA, D = NA, max.p = 5, max.q = 5, max.P = 2, max.Q = 2,
#>     max.order = 5, max.d = 2, max.D = 1, start.p = 2, start.q = 2, start.P = 1,
#>     start.Q = 1, stationary = FALSE, seasonal = TRUE, ic = c("aicc", "aic",
#>      "bic"), stepwise = TRUE, nmodels = 94, trace = FALSE, approximation = (length(x) >
#>         150 | frequency(x) > 12), method = NULL, truncate = NULL, xreg = NULL,
#>    test = c("kpss", "adf", "pp"), test.args = list(), seasonal.test = c("seas",
#>         "ocsb", "hegy", "ch"), seasonal.test.args = list(), allowdrift = TRUE,
#>     allowmean = TRUE, lambda = NULL, biasadj = FALSE, parallel = FALSE,
#>     num.cores = 2, x = y, ...)
```

The *MAXIMUM* nonseasonal ARIMA terms (`max.p`, `max.d`, `max.q`) and seasonal ARIMA terms (`max.P`, `max.D`, `max.Q`) are provided to `forecast::auto.arima()` via `arima_reg()` parameters. Other options and argument can be set using `set_engine()`.

Parameter Notes:

- All values of nonseasonal pdq and seasonal PDQ are maximums. The `forecast::auto.arima()` model will select a value using these as an upper limit.
- `xreg` - This is supplied via the parsnip / modeltime `fit()` interface (so don't provide this manually). See Fit Details (below).

arima

The engine uses `forecast::Arima()`.

Function Parameters:

```
#> function (y, order = c(0, 0, 0), seasonal = c(0, 0, 0), xreg = NULL, include.mean = TRUE,
#>    include.drift = FALSE, include.constant, lambda = model$lambda, biasadj = FALSE,
#>     method = c("CSS-ML", "ML", "CSS"), model = NULL, x = y, ...)
```

The nonseasonal ARIMA terms (`order`) and seasonal ARIMA terms (`seasonal`) are provided to `forecast::Arima()` via `arima_reg()` parameters. Other options and argument can be set using `set_engine()`.

Parameter Notes:

- `xreg` - This is supplied via the parsnip / modeltime `fit()` interface (so don't provide this manually). See Fit Details (below).
- `method` - The default is set to "ML" (Maximum Likelihood). This method is more robust at the expense of speed and possible selections may fail unit root inversion testing. Alternatively, you can add `method = "CSS-ML"` to evaluate Conditional Sum of Squares for starting values, then Maximium Likelihood.

Fit Details

Date and Date-Time Variable

It's a requirement to have a date or date-time variable as a predictor. The `fit()` interface accepts date and date-time features and handles them internally.

- `fit(y ~ date)`

Seasonal Period Specification

The period can be non-seasonal (`seasonal_period = 1 or "none"`) or yearly seasonal (e.g. For monthly time stamps, `seasonal_period = 12`, `seasonal_period = "12 months"`, or `seasonal_period = "yearly"`). There are 3 ways to specify:

1. `seasonal_period = "auto"`: A seasonal period is selected based on the periodicity of the data (e.g. 12 if monthly)
2. `seasonal_period = 12`: A numeric frequency. For example, 12 is common for monthly data
3. `seasonal_period = "1 year"`: A time-based phrase. For example, "1 year" would convert to 12 for monthly data.

Univariate (No xregs, Exogenous Regressors):

For univariate analysis, you must include a date or date-time feature. Simply use:

- Formula Interface (recommended): `fit(y ~ date)` will ignore xreg's.
- XY Interface: `fit_xy(x = data[,"date"], y = data$y)` will ignore xreg's.

Multivariate (xregs, Exogenous Regressors)

The `xreg` parameter is populated using the `fit()` or `fit_xy()` function:

- Only `factor`, `ordered factor`, and `numeric` data will be used as xregs.
- Date and Date-time variables are not used as xregs
- `character` data should be converted to factor.

Xreg Example: Suppose you have 3 features:

1. `y` (target)
2. `date` (time stamp),
3. `month.lbl` (labeled month as a ordered factor).

The `month.lbl` is an exogenous regressor that can be passed to the `arima_reg()` using `fit()`:

- `fit(y ~ date + month.lbl)` will pass `month.lbl` on as an exogenous regressor.
- `fit_xy(data[,c("date", "month.lbl")], y = data$y)` will pass x, where x is a data frame containing `month.lbl` and the `date` feature. Only `month.lbl` will be used as an exogenous regressor.

Note that date or date-time class values are excluded from `xreg`.

See Also

`fit.model_spec()`, `set_engine()`

Examples

```
library(dplyr)
library(parsnip)
library(rsample)
library(timetk)

# Data
m750 <- m4_monthly %>% filter(id == "M750")
m750

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.8)

# ---- AUTO ARIMA ----

# Model Spec
model_spec <- arima_reg() %>%
    set_engine("auto_arima")

# Fit Spec
model_fit <- model_spec %>%
    fit(log(value) ~ date, data = training(splits))
model_fit


# ---- STANDARD ARIMA ----

# Model Spec
model_spec <- arima_reg(
        seasonal_period          = 12,
        non_seasonal_ar          = 3,
        non_seasonal_differences = 1,
        non_seasonal_ma          = 3,
        seasonal_ar              = 1,
        seasonal_differences     = 0,
        seasonal_ma              = 1
    ) %>%
    set_engine("arima")

# Fit Spec
model_fit <- model_spec %>%
    fit(log(value) ~ date, data = training(splits))
model_fit
```

`combine_modeltime_tables` *Combine multiple Modeltime Tables into a single Modeltime Table*

Description

Combine multiple Modeltime Tables into a single Modeltime Table

Usage

```
combine_modeltime_tables(...)
```

Arguments

...	Multiple Modeltime Tables (class mdl_time_tbl)

Details

This function combines multiple Modeltime Tables.

- The .model_id will automatically be renumbered to ensure each model has a unique ID.
- Only the .model_id, .model, and .model_desc columns will be returned.

Re-Training Models on the Same Datasets

One issue can arise if your models are trained on different datasets. If your models have been trained on different datasets, you can run modeltime_refit() to train all models on the same data.

Re-Calibrating Models

If your data has been calibrated using modeltime_calibrate(), the .test and .calibration_data columns will be removed. To re-calibrate, simply run modeltime_calibrate() on the newly combined Modeltime Table.

See Also

- combine_modeltime_tables(): Combine 2 or more Modeltime Tables together
- add_modeltime_model(): Adds a new row with a new model to a Modeltime Table
- drop_modeltime_model(): Drop one or more models from a Modeltime Table
- update_modeltime_description(): Updates a description for a model inside a Modeltime Table
- update_modeltime_model(): Updates a model inside a Modeltime Table
- pull_modeltime_model(): Extracts a model from a Modeltime Table

Examples

```
library(tidymodels)
library(timetk)
library(dplyr)
library(lubridate)

# Setup
m750 <- m4_monthly %>% filter(id == "M750")

splits <- time_series_split(m750, assess = "3 years", cumulative = TRUE)
```

```
model_fit_arima <- arima_reg() %>%
    set_engine("auto_arima") %>%
    fit(value ~ date, training(splits))

model_fit_prophet <- prophet_reg() %>%
    set_engine("prophet") %>%
    fit(value ~ date, training(splits))

# Multiple Modeltime Tables
model_tbl_1 <- modeltime_table(model_fit_arima)
model_tbl_2 <- modeltime_table(model_fit_prophet)

# Combine
combine_modeltime_tables(model_tbl_1, model_tbl_2)
```

control_modeltime *Control aspects of the training process*

Description

These functions are matched to the associated training functions:

- control_refit(): Used with modeltime_refit()
- control_fit_workflowset(): Used with modeltime_fit_workflowset()
- control_nested_fit(): Used with modeltime_nested_fit()
- control_nested_refit(): Used with modeltime_nested_refit()
- control_nested_forecast(): Used with modeltime_nested_forecast()

Usage

```
control_refit(verbose = FALSE, allow_par = FALSE, cores = 1, packages = NULL)

control_fit_workflowset(
  verbose = FALSE,
  allow_par = FALSE,
  cores = 1,
  packages = NULL
)

control_nested_fit(
  verbose = FALSE,
  allow_par = FALSE,
  cores = 1,
  packages = NULL
)
```

```
control_nested_refit(
  verbose = FALSE,
  allow_par = FALSE,
  cores = 1,
  packages = NULL
)

control_nested_forecast(
  verbose = FALSE,
  allow_par = FALSE,
  cores = 1,
  packages = NULL
)
```

Arguments

verbose	Logical to control printing.
allow_par	Logical to allow parallel computation. Default: FALSE (single threaded).
cores	Number of cores for computation. If -1, uses all available physical cores. Default: 1.
packages	An optional character string of additional R package names that should be loaded during parallel processing. • Packages in your namespace are loaded by default • Key Packages are loaded by default: tidymodels, parsnip, modeltime, dplyr, stats, lubridate and timetk.

Value

A List with the control settings.

See Also

- Setting Up Parallel Processing: parallel_start(), [parallel_stop())]
- Training Functions: [modeltime_refit()], [modeltime_fit_workflowset()], [modeltime_nested_fit()], [modeltime_nested_refit()]

[parallel_stop())]: R:parallel_stop()) [modeltime_refit()]: R:modeltime_refit() [modeltime_fit_workflowset()]: R:modeltime_fit_workflowset() [modeltime_nested_fit()]: R:modeltime_nested_fit() [modeltime_nested_refit()]: R:modeltime_nested_refit()

Examples

```
# No parallel processing by default
control_refit()

# Allow parallel processing and use all cores
control_refit(allow_par = TRUE, cores = -1)

# Set verbosity to show additional training information
```

```
control_refit(verbose = TRUE)

# Add additional packages used during modeling in parallel processing
# - This is useful if your namespace does not load all needed packages
#   to run models.
# - An example is if I use `temporal_hierarchy()`, which depends on the `thief` package
control_refit(allow_par = TRUE, packages = "thief")
```

---

create_model_grid — *Helper to make* parsnip *model specs from a* dials *parameter grid*

Description

Helper to make parsnip model specs from a dials parameter grid

Usage

```
create_model_grid(grid, f_model_spec, engine_name, ..., engine_params = list())
```

Arguments

grid	A tibble that forms a grid of parameters to adjust
f_model_spec	A function name (quoted or unquoted) that specifies a parsnip model specification function
engine_name	A name of an engine to use. Gets passed to parsnip::set_engine().
...	Static parameters that get passed to the f_model_spec
engine_params	A list of additional parameters that can be passed to the engine via parsnip::set_engine(...).

Details

This is a helper function that combines dials grids with parsnip model specifications. The intent is to make it easier to generate workflowset objects for forecast evaluations with modeltime_fit_workflowset().

The process follows:

1. Generate a grid (hyperparemeter combination)
2. Use create_model_grid() to apply the parameter combinations to a parsnip model spec and engine.

The output contains ".model" column that can be used as a list of models inside the workflow_set() function.

Value

Tibble with a new colum named .models

See Also

- `dials::grid_regular()`: For making parameter grids.
- `workflowsets::workflow_set()`: For creating a `workflowset` from the `.models` list stored in the ".models" column.
- `modeltime_fit_workflowset()`: For fitting a `workflowset` to forecast data.

Examples

```
library(tidymodels)

# Parameters that get optimized
grid_tbl <- grid_regular(
    learn_rate(),
    levels = 3
)

# Generate model specs
grid_tbl %>%
    create_model_grid(
        f_model_spec = boost_tree,
        engine_name  = "xgboost",
        # Static boost_tree() args
        mode = "regression",
        # Static set_engine() args
        engine_params = list(
            max_depth = 5
        )
    )
```

---

create_xreg_recipe *Developer Tools for preparing XREGS (Regressors)*

Description

These functions are designed to assist developers in extending the `modeltime` package. `create_xregs_recipe()` makes it simple to automate conversion of raw un-encoded features to machine-learning ready features.

Usage

```
create_xreg_recipe(
  data,
  prepare = TRUE,
  clean_names = TRUE,
  dummy_encode = TRUE,
  one_hot = FALSE
)
```

Arguments

`data`	A data frame
`prepare`	Whether or not to run `recipes::prep()` on the final recipe. Default is to prepare. User can set this to FALSE to return an un prepared recipe.
`clean_names`	Uses `janitor::clean_names()` to process the names and improve robustness to failure during dummy (one-hot) encoding step.
`dummy_encode`	Should `factors` (categorical data) be
`one_hot`	If `dummy_encode = TRUE`, should the encoding return one column for each feature or one less column than each feature. Default is `FALSE`.

Details

The default recipe contains steps to:

1. Remove date features
2. Clean the column names removing spaces and bad characters
3. Convert ordered factors to regular factors
4. Convert factors to dummy variables
5. Remove any variables that have zero variance

Value

A `recipe` in either prepared or un-prepared format.

Examples

```
library(dplyr)
library(timetk)
library(recipes)
library(lubridate)

predictors <- m4_monthly %>%
    filter(id == "M750") %>%
    select(-value) %>%
    mutate(month = month(date, label = TRUE))
predictors

# Create default recipe
xreg_recipe_spec <- create_xreg_recipe(predictors, prepare = TRUE)

# Extracts the preprocessed training data from the recipe (used in your fit function)
juice_xreg_recipe(xreg_recipe_spec)

# Applies the prepared recipe to new data (used in your predict function)
bake_xreg_recipe(xreg_recipe_spec, new_data = predictors)
```

drop_modeltime_model *Drop a Model from a Modeltime Table*

Description

Drop a Model from a Modeltime Table

Usage

```
drop_modeltime_model(object, .model_id)
```

Arguments

object	A Modeltime Table (class mdl_time_tbl)
.model_id	A numeric value matching the .model_id that you want to drop

See Also

- combine_modeltime_tables(): Combine 2 or more Modeltime Tables together
- add_modeltime_model(): Adds a new row with a new model to a Modeltime Table
- drop_modeltime_model(): Drop one or more models from a Modeltime Table
- update_modeltime_description(): Updates a description for a model inside a Modeltime Table
- update_modeltime_model(): Updates a model inside a Modeltime Table
- pull_modeltime_model(): Extracts a model from a Modeltime Table

Examples

```
library(tidymodels)


m750_models %>%
    drop_modeltime_model(.model_id = c(2,3))
```

`exp_smoothing` *General Interface for Exponential Smoothing State Space Models*

Description

`exp_smoothing()` is a way to generate a *specification* of an Exponential Smoothing model before fitting and allows the model to be created using different packages. Currently the only package is `forecast`. Several algorithms are implemented:

- ETS - Automated Exponential Smoothing
- CROSTON - Croston's forecast is a special case of Exponential Smoothing for intermittent demand
- Theta - A special case of Exponential Smoothing with Drift that performed well in the M3 Competition

Usage

```
exp_smoothing(
  mode = "regression",
  seasonal_period = NULL,
  error = NULL,
  trend = NULL,
  season = NULL,
  damping = NULL,
  smooth_level = NULL,
  smooth_trend = NULL,
  smooth_seasonal = NULL
)
```

Arguments

`mode` A single character string for the type of model. The only possible value for this model is "regression".

`seasonal_period` A seasonal frequency. Uses "auto" by default. A character phrase of "auto" or time-based phrase of "2 weeks" can be used if a date or date-time variable is provided. See Fit Details below.

`error` The form of the error term: "auto", "additive", or "multiplicative". If the error is multiplicative, the data must be non-negative.

`trend` The form of the trend term: "auto", "additive", "multiplicative" or "none".

`season` The form of the seasonal term: "auto", "additive", "multiplicative" or "none".

`damping` Apply damping to a trend: "auto", "damped", or "none".

`smooth_level` This is often called the "alpha" parameter used as the base level smoothing factor for exponential smoothing models.

`smooth_trend` This is often called the "beta" parameter used as the trend smoothing factor for exponential smoothing models.

`smooth_seasonal`
This is often called the "gamma" parameter used as the seasonal smoothing factor for exponential smoothing models.

Details

Models can be created using the following *engines*:

- "ets" (default) - Connects to `forecast::ets()`
- "croston" - Connects to `forecast::croston()`
- "theta" - Connects to `forecast::thetaf()`
- "smooth_es" - Connects to `smooth::es()`

Engine Details

The standardized parameter names in `modeltime` can be mapped to their original names in each engine:

modeltime	forecast::ets	forecast::croston()	forecast::thetaf()	smooth::es()
seasonal_period()	ts(frequency)	ts(frequency)	ts(frequency)	ts(frequency)
error(), trend(), season()	model ('ZZZ')	NA	NA	model('ZZZ')
damping()	damped (NULL)	NA	NA	phi
smooth_level()	alpha (NULL)	alpha (0.1)	NA	persistence(alpha)
smooth_trend()	beta (NULL)	NA	NA	persistence(beta)
smooth_seasonal()	gamma (NULL)	NA	NA	persistence(gamma)

Other options can be set using `set_engine()`.

ets (default engine)

The engine uses `forecast::ets()`.

Function Parameters:

```
#> function (y, model = "ZZZ", damped = NULL, alpha = NULL, beta = NULL, gamma = NULL,
#>     phi = NULL, additive.only = FALSE, lambda = NULL, biasadj = FALSE,
#>    lower = c(rep(1e-04, 3), 0.8), upper = c(rep(0.9999, 3), 0.98), opt.crit = c("lik",
#>         "amse", "mse", "sigma", "mae"), nmse = 3, bounds = c("both", "usual",
#>      "admissible"), ic = c("aicc", "aic", "bic"), restrict = TRUE, allow.multiplicative.trend = FALS
#>     use.initial.values = FALSE, na.action = c("na.contiguous", "na.interp",
#>         "na.fail"), ...)
```

The main arguments are `model` and `damped` are defined using:

- `error()` = "auto", "additive", and "multiplicative" are converted to "Z", "A", and "M"
- `trend()` = "auto", "additive", "multiplicative", and "none" are converted to "Z","A","M" and "N"

- `season()` = "auto", "additive", "multiplicative", and "none" are converted to "Z","A","M" and "N"
- `damping()` - "auto", "damped", "none" are converted to NULL, TRUE, FALSE
- `smooth_level()`, `smooth_trend()`, and `smooth_seasonal()` are automatically determined if not provided. They are mapped to "alpha", "beta" and "gamma", respectively.

By default, all arguments are set to "auto" to perform automated Exponential Smoothing using *in-sample data* following the underlying `forecast::ets()` automation routine.

Other options and argument can be set using `set_engine()`.

Parameter Notes:

- `xreg` - This model is not set up to use exogenous regressors. Only univariate models will be fit.

croston

The engine uses `forecast::croston()`.

Function Parameters:

```
#> function (y, h = 10, alpha = 0.1, x = y)
```

The main arguments are defined using:

- `smooth_level()`: The "alpha" parameter

Parameter Notes:

- `xreg` - This model is not set up to use exogenous regressors. Only univariate models will be fit.

theta

The engine uses `forecast::thetaf()`

Parameter Notes:

- `xreg` - This model is not set up to use exogenous regressors. Only univariate models will be fit.

smooth_es

The engine uses `smooth::es()`.

Function Parameters:

```
#> function (y, model = "ZXZ", lags = c(frequency(y)), persistence = NULL,
#>     phi = NULL, initial = c("backcasting", "optimal", "two-stage", "complete"),
#>     initialSeason = NULL, ic = c("AICc", "AIC", "BIC", "BICc"), loss = c("likelihood",
#>         "MSE", "MAE", "HAM", "MSEh", "TMSE", "GTMSE", "MSCE"), h = 10,
#>     holdout = FALSE, bounds = c("usual", "admissible", "none"), silent = TRUE,
#>     xreg = NULL, regressors = c("use", "select"), initialX = NULL, ...)
```

The main arguments `model` and `phi` are defined using:

- `error()` = "auto", "additive" and "multiplicative" are converted to "Z", "A" and "M"
- `trend()` = "auto", "additive", "multiplicative", "additive_damped", "multiplicative_damped" and "none" are converted to "Z", "A", "M", "Ad", "Md" and "N".
- `season()` = "auto", "additive", "multiplicative", and "none" are converted "Z", "A","M" and "N"
- `damping()` - Value of damping parameter. If NULL, then it is estimated.
- `smooth_level()`, `smooth_trend()`, and `smooth_seasonal()` are automatically determined if not provided. They are mapped to "persistence"("alpha", "beta" and "gamma", respectively).

By default, all arguments are set to "auto" to perform automated Exponential Smoothing using *in-sample data* following the underlying `smooth::es()` automation routine.

Other options and argument can be set using `set_engine()`.

Parameter Notes:

- `xreg` - This is supplied via the parsnip / modeltime `fit()` interface (so don't provide this manually). See Fit Details (below).

Fit Details

Date and Date-Time Variable

It's a requirement to have a date or date-time variable as a predictor. The `fit()` interface accepts date and date-time features and handles them internally.

- `fit(y ~ date)`

Seasonal Period Specification

The period can be non-seasonal (`seasonal_period = 1` or `"none"`) or seasonal (e.g. `seasonal_period = 12` or `seasonal_period = "12 months"`). There are 3 ways to specify:

1. `seasonal_period = "auto"`: A period is selected based on the periodicity of the data (e.g. 12 if monthly)
2. `seasonal_period = 12`: A numeric frequency. For example, 12 is common for monthly data
3. `seasonal_period = "1 year"`: A time-based phrase. For example, "1 year" would convert to 12 for monthly data.

Univariate:

For univariate analysis, you must include a date or date-time feature. Simply use:

- Formula Interface (recommended): `fit(y ~ date)` will ignore xreg's.
- XY Interface: `fit_xy(x = data[,"date"], y = data$y)` will ignore xreg's.

Multivariate (xregs, Exogenous Regressors)

Just for `smooth` engine.

The `xreg` parameter is populated using the `fit()` or `fit_xy()` function:

- Only `factor`, `ordered factor`, and `numeric` data will be used as xregs.
- Date and Date-time variables are not used as xregs

- `character` data should be converted to factor.

Xreg Example: Suppose you have 3 features:

1. `y` (target)
2. `date` (time stamp),
3. `month.lbl` (labeled month as a ordered factor).

The `month.lbl` is an exogenous regressor that can be passed to the `arima_reg()` using `fit()`:

- `fit(y ~ date + month.lbl)` will pass `month.lbl` on as an exogenous regressor.
- `fit_xy(data[,c("date", "month.lbl")], y = data$y)` will pass x, where x is a data frame containing `month.lbl` and the `date` feature. Only `month.lbl` will be used as an exogenous regressor.

Note that date or date-time class values are excluded from `xreg`.

See Also

`fit.model_spec()`, `set_engine()`

Examples

```
library(dplyr)
library(parsnip)
library(rsample)
library(timetk)
library(smooth)

# Data
m750 <- m4_monthly %>% filter(id == "M750")
m750

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.8)

# ---- AUTO ETS ----

# Model Spec - The default parameters are all set
# to "auto" if none are provided
model_spec <- exp_smoothing() %>%
    set_engine("ets")

# Fit Spec
model_fit <- model_spec %>%
    fit(log(value) ~ date, data = training(splits))
model_fit


# ---- STANDARD ETS ----

# Model Spec
```

```
model_spec <- exp_smoothing(
        seasonal_period  = 12,
        error            = "multiplicative",
        trend            = "additive",
        season           = "multiplicative"
    ) %>%
    set_engine("ets")

# Fit Spec
model_fit <- model_spec %>%
    fit(log(value) ~ date, data = training(splits))
model_fit


# ---- CROSTON ----

# Model Spec
model_spec <- exp_smoothing(
        smooth_level = 0.2
    ) %>%
    set_engine("croston")

# Fit Spec
model_fit <- model_spec %>%
    fit(log(value) ~ date, data = training(splits))
model_fit




# ---- THETA ----

#' # Model Spec
model_spec <- exp_smoothing() %>%
    set_engine("theta")

# Fit Spec
model_fit <- model_spec %>%
    fit(log(value) ~ date, data = training(splits))
model_fit





#' # ---- SMOOTH ----

#' # Model Spec
model_spec <- exp_smoothing(
               seasonal_period  = 12,
               error            = "multiplicative",
               trend            = "additive_damped",
               season           = "additive"
```

```
        ) %>%
    set_engine("smooth_es")

# Fit Spec
model_fit <- model_spec %>%
    fit(value ~ date, data = training(splits))
model_fit
```

`exp_smoothing_params` *Tuning Parameters for Exponential Smoothing Models*

Description

Tuning Parameters for Exponential Smoothing Models

Usage

```
error(values = c("additive", "multiplicative"))

trend(values = c("additive", "multiplicative", "none"))

trend_smooth(
  values = c("additive", "multiplicative", "none", "additive_damped",
    "multiplicative_damped")
)

season(values = c("additive", "multiplicative", "none"))

damping(values = c("none", "damped"))

damping_smooth(range = c(0, 2), trans = NULL)

smooth_level(range = c(0, 1), trans = NULL)

smooth_trend(range = c(0, 1), trans = NULL)

smooth_seasonal(range = c(0, 1), trans = NULL)
```

Arguments

`values`	A character string of possible values.
`range`	A two-element vector holding the *defaults* for the smallest and largest possible values, respectively. If a transformation is specified, these values should be in the *transformed units*.
`trans`	A `trans` object from the `scales` package, such as `scales::transform_log10()` or `scales::transform_reciprocal()`. If not provided, the default is used which matches the units used in `range`. If no transformation, `NULL`.

Details

The main parameters for Exponential Smoothing models are:

- `error`: The form of the error term: additive", or "multiplicative". If the error is multiplicative, the data must be non-negative.
- `trend`: The form of the trend term: "additive", "multiplicative" or "none".
- `season`: The form of the seasonal term: "additive", "multiplicative" or "none"..
- `damping`: Apply damping to a trend: "damped", or "none".
- `smooth_level`: This is often called the "alpha" parameter used as the base level smoothing factor for exponential smoothing models.
- `smooth_trend`: This is often called the "beta" parameter used as the trend smoothing factor for exponential smoothing models.
- `smooth_seasonal`: This is often called the "gamma" parameter used as the seasonal smoothing factor for exponential smoothing models.

Examples

```
error()

trend()

season()
```

`get_arima_description` *Get model descriptions for Arima objects*

Description

Get model descriptions for Arima objects

Usage

```
get_arima_description(object, padding = FALSE)
```

Arguments

`object`	Objects of class `Arima`
`padding`	Whether or not to include padding

Source

- Forecast R Package, `forecast:::arima.string()`

Examples

```
library(forecast)

arima_fit <- forecast::Arima(1:10)

get_arima_description(arima_fit)
```

---

get_model_description *Get model descriptions for parsnip, workflows & modeltime objects*

Description

Get model descriptions for parsnip, workflows & modeltime objects

Usage

```
get_model_description(object, indicate_training = FALSE, upper_case = TRUE)
```

Arguments

object	Parsnip or workflow objects
indicate_training	Whether or not to indicate if the model has been trained
upper_case	Whether to return upper or lower case model descriptions

Examples

```
library(dplyr)
library(timetk)
library(parsnip)

# Model Specification ----

arima_spec <- arima_reg() %>%
    set_engine("auto_arima")

get_model_description(arima_spec, indicate_training = TRUE)

# Fitted Model ----

m750 <- m4_monthly %>% filter(id == "M750")

arima_fit <- arima_spec %>%
    fit(value ~ date, data = m750)

get_model_description(arima_fit, indicate_training = TRUE)
```

get_tbats_description *Get model descriptions for TBATS objects*

Description

Get model descriptions for TBATS objects

Usage

```
get_tbats_description(object)
```

Arguments

object	Objects of class tbats

Source

- Forecast R Package, `forecast:::as.character.tbats()`

log_extractors *Log Extractor Functions for Modeltime Nested Tables*

Description

Extract logged information calculated during the `modeltime_nested_fit()`, `modeltime_nested_select_best()`, and `modeltime_nested_refit()` processes.

Usage

```
extract_nested_test_accuracy(object)

extract_nested_test_forecast(object, .include_actual = TRUE, .id_subset = NULL)

extract_nested_error_report(object)

extract_nested_best_model_report(object)

extract_nested_future_forecast(
  object,
  .include_actual = TRUE,
  .id_subset = NULL
)

extract_nested_modeltime_table(object, .row_id = 1)

extract_nested_train_split(object, .row_id = 1)

extract_nested_test_split(object, .row_id = 1)
```

Arguments

`object`	A nested modeltime table
`.include_actual`	Whether or not to include the actual data in the extracted forecast. Default: TRUE.
`.id_subset`	Can supply a vector of id's to extract forcasts for one or more id's, rather than extracting all forecasts. If `NULL`, extracts forecasts for all id's.
`.row_id`	The row number to extract from the nested data.

`m750` *The 750th Monthly Time Series used in the M4 Competition*

Description

The 750th Monthly Time Series used in the M4 Competition

Usage

```
m750
```

Format

A `tibble` with 306 rows and 3 variables:

- `id` Factor. Unique series identifier
- `date` Date. Timestamp information. Monthly format.
- `value` Numeric. Value at the corresponding timestamp.

Source

- M4 Competition Website: https://www.unic.ac.cy/iff/research/forecasting/m-competitions/m4/

Examples

```
m750
```

m750_models *Three (3) Models trained on the M750 Data (Training Set)*

Description

Three (3) Models trained on the M750 Data (Training Set)

Usage

```
m750_models
```

Format

An `time_series_cv` object with 6 slices of Time Series Cross Validation resamples made on the `training(m750_splits)`

Details

```
m750_models <- modeltime_table(
    wflw_fit_arima,
    wflw_fit_prophet,
    wflw_fit_glmnet
)
```

Examples

```
m750_models
```

m750_splits *The results of train/test splitting the M750 Data*

Description

The results of train/test splitting the M750 Data

Usage

```
m750_splits
```

Format

An `rsplit` object split into approximately 23.5-years of training data and 2-years of testing data

Details

```
library(timetk)
m750_splits <- time_series_split(m750, assess = "2 years", cumulative = TRUE)
```

Examples

```
library(rsample)

m750_splits

training(m750_splits)
```

`m750_training_resamples`
The Time Series Cross Validation Resamples the M750 Data (Training Set)

Description

The Time Series Cross Validation Resamples the M750 Data (Training Set)

Usage

```
m750_training_resamples
```

Format

An `time_series_cv` object with 6 slices of Time Series Cross Validation resamples made on the `training(m750_splits)`

Details

```
library(timetk)
m750_training_resamples <- time_series_cv(
    data        = training(m750_splits),
    assess      = "2 years",
    skip        = "2 years",
    cumulative  = TRUE,
    slice_limit = 6
)
```

Examples

```
library(rsample)

m750_training_resamples
```

`maape` *Mean Arctangent Absolute Percentage Error*

Description

Useful when MAPE returns Inf (e.g., intermittent data with zeros).

Usage

```
maape(data, ...)

## S3 method for class 'data.frame'
maape(data, truth, estimate, na_rm = TRUE, case_weights = NULL, ...)
```

Arguments

`data`	A data frame containing the truth and estimate columns.
`...`	Additional arguments (not used).
`truth`	The column identifier for the true results (numeric).
`estimate`	The column identifier for the predicted results (numeric).
`na_rm`	Logical, whether to remove missing values before computation.
`case_weights`	Optional column identifier for non-negative case weights (not used by `maape`).

Value

A tibble with columns `.metric`, `.estimator`, and `.estimate`.

`maape_vec` *Mean Arctangent Absolute Percentage Error*

Description

Wrapper around `TSrepr::maape()` with yardstick-compatible interface.

Usage

```
maape_vec(truth, estimate, na_rm = TRUE, case_weights = NULL, ...)
```

Arguments

`truth`	Numeric vector of ground-truth values.
`estimate`	Numeric vector of predicted values.
`na_rm`	Logical, whether to remove missing values before computation.
`case_weights`	Optional numeric vector of non-negative case weights (not used by `maape_vec`).
`...`	Additional arguments (not used).

Value

A numeric value representing the Mean Arctangent Absolute Percentage Error.

`metric_sets` *Forecast Accuracy Metrics Sets*

Description

This is a wrapper for `metric_set()` with several common forecast / regression accuracy metrics included. These are the default time series accuracy metrics used with `modeltime_accuracy()`.

Usage

```
default_forecast_accuracy_metric_set(...)

extended_forecast_accuracy_metric_set(...)
```

Arguments

`...`	Add additional `yardstick` metrics

Default Forecast Accuracy Metric Set

The primary purpose is to use the default accuracy metrics to calculate the following forecast accuracy metrics using `modeltime_accuracy()`:

- MAE - Mean absolute error, `mae()`
- MAPE - Mean absolute percentage error, `mape()`
- MASE - Mean absolute scaled error, `mase()`
- SMAPE - Symmetric mean absolute percentage error, `smape()`
- RMSE - Root mean squared error, `rmse()`
- RSQ - R-squared, `rsq()`

Adding additional metrics is possible via `...`.

Extended Forecast Accuracy Metric Set

Extends the default metric set by adding:

- MAAPE - Mean Arctangent Absolute Percentage Error, `maape()`. MAAPE is designed for intermittent data where MAPE returns `Inf`.

See Also

- `yardstick::metric_tweak()` - For modifying `yardstick` metrics

Examples

```
library(tibble)
library(dplyr)
library(timetk)
library(yardstick)

fake_data <- tibble(
  y = c(1:12, 2*1:12),
  yhat = c(1 + 1:12, 2*1:12 - 1)
)

# ---- HOW IT WORKS ----

# Default Forecast Accuracy Metric Specification
default_forecast_accuracy_metric_set()

# Create a metric summarizer function from the metric set
calc_default_metrics <- default_forecast_accuracy_metric_set()

# Apply the metric summarizer to new data
calc_default_metrics(fake_data, y, yhat)

# ---- ADD MORE PARAMETERS ----

# Can create a version of mase() with seasonality = 12 (monthly)
mase12 <- metric_tweak(.name = "mase12", .fn = mase, m = 12)

# Add it to the default metric set
my_metric_set <- default_forecast_accuracy_metric_set(mase12)
my_metric_set

# Apply the newly created metric set
my_metric_set(fake_data, y, yhat)
```

---

modeltime_accuracy *Calculate Accuracy Metrics*

Description

This is a wrapper for `yardstick` that simplifies time series regression accuracy metric calculations from a fitted `workflow` (trained workflow) or `model_fit` (trained parsnip model).

Usage

```
modeltime_accuracy(
  object,
  new_data = NULL,
  metric_set = default_forecast_accuracy_metric_set(),
```

```
  acc_by_id = FALSE,
  quiet = TRUE,
  ...
)
```

Arguments

`object`	A Modeltime Table
`new_data`	A `tibble` to predict and calculate residuals on. If provided, overrides any calibration data.
`metric_set`	A `yardstick::metric_set()` that is used to summarize one or more forecast accuracy (regression) metrics.
`acc_by_id`	Should a global or local model accuracy be produced? (Default: FALSE) • When `FALSE`, a global model accuracy is provided. • If `TRUE`, a local accuracy is provided group-wise for each time series ID. To enable local accuracy, an `id` must be provided during `modeltime_calibrate()`.
`quiet`	Hide errors (`TRUE`, the default), or display them as they occur?
`...`	If `new_data` is provided, these parameters are passed to `modeltime_calibrate()`

Details

The following accuracy metrics are included by default via `default_forecast_accuracy_metric_set()`:

- MAE - Mean absolute error, `mae()`
- MAPE - Mean absolute percentage error, `mape()`
- MASE - Mean absolute scaled error, `mase()`
- SMAPE - Symmetric mean absolute percentage error, `smape()`
- RMSE - Root mean squared error, `rmse()`
- RSQ - R-squared, `rsq()`

Value

A tibble with accuracy estimates.

Examples

```
library(tidymodels)
library(dplyr)
library(lubridate)
library(timetk)


# Data
m750 <- m4_monthly %>% filter(id == "M750")

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.8)
```

```
# --- MODELS ---

# Model 1: prophet ----
model_fit_prophet <- prophet_reg() %>%
    set_engine(engine = "prophet") %>%
    fit(value ~ date, data = training(splits))


# ---- MODELTIME TABLE ----

models_tbl <- modeltime_table(
    model_fit_prophet
)

# ---- ACCURACY ----

models_tbl %>%
    modeltime_calibrate(new_data = testing(splits)) %>%
    modeltime_accuracy(
        metric_set = metric_set(mae, rmse, rsq)
    )
```

`modeltime_calibrate` *Preparation for forecasting*

Description

Calibration sets the stage for accuracy and forecast confidence by computing predictions and residuals from out of sample data.

Usage

```
modeltime_calibrate(object, new_data, id = NULL, quiet = TRUE, ...)
```

Arguments

`object`	A fitted model object that is either: 1. A modeltime table that has been created using `modeltime_table()` 2. A workflow that has been fit by `fit.workflow()` or 3. A parsnip model that has been fit using `fit.model_spec()`
`new_data`	A test data set `tibble` containing future information (timestamps and actual values).
`id`	A quoted column name containing an identifier column identifying time series that are grouped.
`quiet`	Hide errors (`TRUE`, the default), or display them as they occur?
`...`	Additional arguments passed to `modeltime_forecast()`.

Details

The results of calibration are used for:

- **Forecast Confidence Interval Estimation**: The out of sample residual data is used to calculate the confidence interval. Refer to `modeltime_forecast()`.
- **Accuracy Calculations:** The out of sample actual and prediction values are used to calculate performance metrics. Refer to `modeltime_accuracy()`

The calibration steps include:

1. If not a Modeltime Table, objects are converted to Modeltime Tables internally
2. Two Columns are added:

- `.type`: Indicates the sample type. This is:
  - "Test" if predicted, or
  - "Fitted" if residuals were stored during modeling.
- `.calibration_data`:
  - Contains a tibble with Timestamps, Actual Values, Predictions and Residuals calculated from `new_data` (Test Data)
  - If `id` is provided, will contain a 5th column that is the identifier variable.

Value

A Modeltime Table (`mdl_time_tbl`) with nested `.calibration_data` added

Examples

```
library(dplyr)
library(lubridate)
library(timetk)
library(parsnip)
library(rsample)

# Data
m750 <- m4_monthly %>% filter(id == "M750")

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.8)

# --- MODELS ---

# Model 1: prophet ----
model_fit_prophet <- prophet_reg() %>%
    set_engine(engine = "prophet") %>%
    fit(value ~ date, data = training(splits))


# ---- MODELTIME TABLE ----

models_tbl <- modeltime_table(
```

```
    model_fit_prophet
)

# ---- CALIBRATE ----

calibration_tbl <- models_tbl %>%
    modeltime_calibrate(
        new_data = testing(splits)
    )

# ---- ACCURACY ----

calibration_tbl %>%
    modeltime_accuracy()

# ---- FORECAST ----

calibration_tbl %>%
    modeltime_forecast(
        new_data    = testing(splits),
        actual_data = m750
    )
```

---

modeltime_fit_workflowset

Fit a `workflowset` object to one or multiple time series

Description

This is a wrapper for `fit()` that takes a `workflowset` object and fits each model on one or multiple time series either sequentially or in parallel.

Usage

```
modeltime_fit_workflowset(
  object,
  data,
  ...,
  control = control_fit_workflowset()
)
```

Arguments

`object`	A workflow_set object, generated with the workflowsets::workflow_set function.
`data`	A `tibble` that contains data to fit the models.
`...`	Not currently used.
`control`	An object used to modify the fitting process. See `control_fit_workflowset()`.

Value

A Modeltime Table containing one or more fitted models.

See Also

control_fit_workflowset()

Examples

```
library(tidymodels)
library(workflowsets)
library(dplyr)
library(lubridate)
library(timetk)

data_set <- m4_monthly

# SETUP WORKFLOWSETS

rec1 <- recipe(value ~ date + id, data_set) %>%
    step_mutate(date_num = as.numeric(date)) %>%
    step_mutate(month_lbl = lubridate::month(date, label = TRUE)) %>%
    step_dummy(all_nominal(), one_hot = TRUE)

mod1 <- linear_reg() %>% set_engine("lm")

mod2 <- prophet_reg() %>% set_engine("prophet")

wfsets <- workflowsets::workflow_set(
    preproc = list(rec1 = rec1),
    models  = list(
        mod1 = mod1,
        mod2 = mod2
    ),
    cross   = TRUE
)

# FIT WORKFLOWSETS
# - Returns a Modeltime Table with fitted workflowsets

wfsets %>% modeltime_fit_workflowset(data_set)
```

`modeltime_forecast` *Forecast future data*

Description

The goal of `modeltime_forecast()` is to simplify the process of forecasting future data.

Usage

```
modeltime_forecast(
  object,
  new_data = NULL,
  h = NULL,
  actual_data = NULL,
  conf_interval = 0.95,
  conf_by_id = FALSE,
  conf_method = "conformal_default",
  keep_data = FALSE,
  arrange_index = FALSE,
  ...
)
```

Arguments

`object`	A Modeltime Table
`new_data`	A `tibble` containing future information to forecast. If `NULL`, forecasts the calibration data.
`h`	The forecast horizon (can be used instead of `new_data` for time series with no exogenous regressors). Extends the calibration data `h` periods into the future.
`actual_data`	Reference data that is combined with the output tibble and given a `.key = "actual"`
`conf_interval`	An estimated confidence interval based on the calibration data. This is designed to estimate future confidence from *out-of-sample prediction error*.
`conf_by_id`	Whether or not to produce confidence interval estimates by an ID feature. • When `FALSE`, a global model confidence interval is provided. • If `TRUE`, a local confidence interval is provided group-wise for each time series ID. To enable local confidence interval, an `id` must be provided during `modeltime_calibrate()`.
`conf_method`	Algorithm used to produce confidence intervals. All CI's are Conformal Predictions. Choose one of: • `conformal_default`: Uses `qnorm()` to compute quantiles from out-of-sample (test set) residuals. • `conformal_split`: Uses the split method split conformal inference method described by Lei *et al* (2018)
`keep_data`	Whether or not to keep the `new_data` and `actual_data` as extra columns in the results. This can be useful if there is an important feature in the `new_data` and `actual_data` needed when forecasting. Default: `FALSE`.
`arrange_index`	Whether or not to sort the index in rowwise chronological order (oldest to newest) or to keep the original order of the data. Default: `FALSE`.
`...`	Not currently used

Details

The modeltime_forecast() function prepares a forecast for visualization with with plot_modeltime_forecast(). The forecast is controlled by new_data or h, which can be combined with existing data (controlled by actual_data). Confidence intervals are included if the incoming Modeltime Table has been calibrated using modeltime_calibrate(). Otherwise confidence intervals are not estimated.

New Data

When forecasting you can specify future data using new_data. This is a future tibble with date column and columns for xregs extending the trained dates and exogonous regressors (xregs) if used.

- **Forecasting Evaluation Data**: By default, the new_data will use the .calibration_data if new_data is not provided. This is the equivalent of using rsample::testing() for getting test data sets.
- **Forecasting Future Data**: See timetk::future_frame() for creating future tibbles.
- **Xregs**: Can be used with this method

H (Horizon)

When forecasting, you can specify h. This is a phrase like "1 year", which extends the .calibration_data (1st priority) or the actual_data (2nd priority) into the future.

- **Forecasting Future Data**: All forecasts using h are **extended after the calibration data or actual_data**.
- Extending .calibration_data - Calibration data is given 1st priority, which is desirable *after refitting* with modeltime_refit(). Internally, a call is made to timetk::future_frame() to expedite creating new data using the date feature.
- Extending actual_data - If h is provided, and the modeltime table has not been calibrated, the "actual_data" will be extended into the future. This is useful in situations where you want to go directly from modeltime_table() to modeltime_forecast() without calibrating or refitting.
- **Xregs**: Cannot be used because future data must include new xregs. If xregs are desired, build a future data frame and use new_data.

Actual Data

This is reference data that contains the true values of the time-stamp data. It helps in visualizing the performance of the forecast vs the actual data.

When h is used and the Modeltime Table has *not been calibrated*, then the actual data is extended into the future periods that are defined by h.

Confidence Interval Estimation

Confidence intervals (.conf_lo, .conf_hi) are estimated based on the normal estimation of the testing errors (out of sample) from modeltime_calibrate(). The out-of-sample error estimates are then carried through and applied to applied to any future forecasts.

The confidence interval can be adjusted with the conf_interval parameter. The algorithm used to produce confidence intervals can be changed with the conf_method parameter.

Conformal Default Method:

When `conf_method = "conformal_default"` (default), this method uses `qnorm()` to produce a 95% confidence interval by default. It estimates a normal (Gaussian distribution) based on the out-of-sample errors (residuals).

The confidence interval is *mean-adjusted*, meaning that if the mean of the residuals is non-zero, the confidence interval is adjusted to widen the interval to capture the difference in means.

Conformal Split Method:

When `conf_method = "conformal_split`, this method uses the split conformal inference method described by Lei *et al* (2018). This is also implemented in the `probably` R package's `int_conformal_split()` function.

What happens to the confidence interval after refitting models?

Refitting has no affect on the confidence interval since this is calculated independently of the refitted model. New observations typically improve future accuracy, which in most cases makes the out-of-sample confidence intervals conservative.

Keep Data

Include the new data (and actual data) as extra columns with the results of the model forecasts. This can be helpful when the new data includes information useful to the forecasts. An example is when forecasting *Panel Data* and the new data contains ID features related to the time series group that the forecast belongs to.

Arrange Index

By default, `modeltime_forecast()` keeps the original order of the data. If desired, the user can sort the output by `.key`, `.model_id` and `.index`.

Value

A tibble with predictions and time-stamp data. For ease of plotting and calculations, the column names are transformed to:

- `.key`: Values labeled either "prediction" or "actual"
- `.index`: The timestamp index.
- `.value`: The value being forecasted.

Additionally, if the Modeltime Table has been previously calibrated using `modeltime_calibrate()`, you will gain confidence intervals.

- `.conf_lo`: The lower limit of the confidence interval.
- `.conf_hi`: The upper limit of the confidence interval.

Additional descriptive columns are included:

- `.model_id`: Model ID from the Modeltime Table
- `.model_desc`: Model Description from the Modeltime Table

Unnecessary columns are *dropped* to save space:

- `.model`
- `.calibration_data`

References

Lei, Jing, et al. "Distribution-free predictive inference for regression." *Journal of the American Statistical Association* 113.523 (2018): 1094-1111.

Examples

```
library(dplyr)
library(timetk)
library(parsnip)
library(rsample)

# Data
m750 <- m4_monthly %>% filter(id == "M750")

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.9)

# --- MODELS ---

# Model 1: prophet ----
model_fit_prophet <- prophet_reg() %>%
    set_engine(engine = "prophet") %>%
    fit(value ~ date, data = training(splits))


# ---- MODELTIME TABLE ----

models_tbl <- modeltime_table(
    model_fit_prophet
)

# ---- CALIBRATE ----

calibration_tbl <- models_tbl %>%
    modeltime_calibrate(new_data = testing(splits))

# ---- ACCURACY ----

calibration_tbl %>%
    modeltime_accuracy()

# ---- FUTURE FORECAST ----

calibration_tbl %>%
    modeltime_forecast(
        new_data    = testing(splits),
        actual_data = m750
    )

# ---- ALTERNATIVE: FORECAST WITHOUT CONFIDENCE INTERVALS ----
# Skips Calibration Step, No Confidence Intervals
```

```
models_tbl %>%
    modeltime_forecast(
        new_data    = testing(splits),
        actual_data = m750
    )

# ---- KEEP NEW DATA WITH FORECAST ----
# Keeps the new data. Useful if new data has information
#  like ID features that should be kept with the forecast data

calibration_tbl %>%
    modeltime_forecast(
        new_data      = testing(splits),
        keep_data     = TRUE
    )
```

`modeltime_nested_fit` *Fit Tidymodels Workflows to Nested Time Series*

Description

Fits one or more `tidymodels` workflow objects to nested time series data using the following process:

1. Models are iteratively fit to training splits.
2. Accuracy is calculated on testing splits and is logged. Accuracy results can be retrieved with `extract_nested_test_accuracy()`
3. Any model that returns an error is logged. Error logs can be retrieved with `extract_nested_error_report()`
4. Forecast is predicted on testing splits and is logged. Forecast results can be retrieved with `extract_nested_test_forecast()`

Usage

```
modeltime_nested_fit(
  nested_data,
  ...,
  model_list = NULL,
  metric_set = default_forecast_accuracy_metric_set(),
  conf_interval = 0.95,
  conf_method = "conformal_default",
  control = control_nested_fit()
)
```

Arguments

`nested_data`	Nested time series data
`...`	Tidymodels `workflow` objects that will be fit to the nested time series data.
`model_list`	Optionally, a `list()` of Tidymodels `workflow` objects can be provided
`metric_set`	A `yardstick::metric_set()` that is used to summarize one or more forecast accuracy (regression) metrics.
`conf_interval`	An estimated confidence interval based on the calibration data. This is designed to estimate future confidence from *out-of-sample prediction error*.
`conf_method`	Algorithm used to produce confidence intervals. All CI's are Conformal Predictions. Choose one of: • `conformal_default`: Uses `qnorm()` to compute quantiles from out-of-sample (test set) residuals. • `conformal_split`: Uses the split method split conformal inference method described by Lei *et al* (2018)
`control`	Used to control verbosity and parallel processing. See `control_nested_fit()`.

Details

Preparing Data for Nested Forecasting:

Use `extend_timeseries()`, `nest_timeseries()`, and `split_nested_timeseries()` for preparing data for Nested Forecasting. The structure must be a nested data frame, which is suppplied in `modeltime_nested_fit(nested_data)`.

Fitting Models:

Models must be in the form of `tidymodels workflow` objects. The models can be provided in two ways:

1. Using `...` (dots): The workflow objects can be provided as dots.
2. Using `model_list` parameter: You can supply one or more workflow objects that are wrapped in a `list()`.

Controlling the fitting process:

A `control` object can be provided during fitting to adjust the verbosity and parallel processing. See `control_nested_fit()`.

`modeltime_nested_forecast`

Modeltime Nested Forecast

Description

Make a new forecast from a Nested Modeltime Table.

Usage

```
modeltime_nested_forecast(
  object,
  h = NULL,
  include_actual = TRUE,
  conf_interval = 0.95,
  conf_method = "conformal_default",
  id_subset = NULL,
  control = control_nested_forecast()
)
```

Arguments

`object`	A Nested Modeltime Table
`h`	The forecast horizon. Extends the "trained on" data "h" periods into the future.
`include_actual`	Whether or not to include the ".actual_data" as part of the forecast. If FALSE, just returns the forecast predictions.
`conf_interval`	An estimated confidence interval based on the calibration data. This is designed to estimate future confidence from *out-of-sample prediction error*.
`conf_method`	Algorithm used to produce confidence intervals. All CI's are Conformal Predictions. Choose one of: • `conformal_default`: Uses `qnorm()` to compute quantiles from out-of-sample (test set) residuals. • `conformal_split`: Uses the split method split conformal inference method described by Lei *et al* (2018)
`id_subset`	A sequence of ID's from the modeltime table to subset the forecasting process. This can speed forecasts up.
`control`	Used to control verbosity and parallel processing. See `control_nested_forecast()`.

Details

This function is designed to help users that want to make new forecasts other than those that are created during the logging process as part of the Nested Modeltime Workflow.

Logged Forecasts:

The logged forecasts can be extracted using:

- `extract_nested_future_forecast()`: Extracts the future forecast created after refitting with `modeltime_nested_refit()`.
- `extract_nested_test_forecast()`: Extracts the test forecast created after initial fitting with `modeltime_nested_fit()`.

The problem is that these forecasts are static. The user would need to redo the fitting, model selection, and refitting process to obtain new forecasts. This is why `modeltime_nested_forecast()` exists. So you can create a new forecast without retraining any models.

Nested Forecasts:

The main arguments is `h`, which is a horizon that specifies how far into the future to make the new forecast.

- If h = NULL, a logged forecast will be returned
- If h = 12, a new forecast will be generated that extends each series 12-periods into the future.
- If h = "2 years", a new forecast will be generated that extends each series 2-years into the future.

Use the id_subset to filter the Nested Modeltime Table object to just the time series of interest.

Use the conf_interval to override the logged confidence interval. Note that this will have no effect if h = NULL as logged forecasts are returned. So be sure to provide h if you want to update the confidence interval.

Use the control argument to apply verbosity during the forecasting process and to run forecasts in parallel. Generally, parallel is better if many forecasts are being generated.

---

modeltime_nested_refit

Refits a Nested Modeltime Table

Description

Refits a Nested Modeltime Table to actual data using the following process:

1. Models are iteratively refit to .actual_data.
2. Any model that returns an error is logged. Errors can be retrieved with extract_nested_error_report()
3. Forecast is predicted on future_data and is logged. Forecast can be retrieved with extract_nested_future_forecast(

Usage

```
modeltime_nested_refit(object, control = control_nested_refit())
```

Arguments

object	A Nested Modeltime Table
control	Used to control verbosity and parallel processing. See control_nested_refit().

---

modeltime_nested_select_best

Select the Best Models from Nested Modeltime Table

Description

Finds the best models for each time series group in a Nested Modeltime Table using a metric that the user specifies.

- Logs the best results, which can be accessed with extract_nested_best_model_report()
- If filter_test_forecasts = TRUE, updates the test forecast log, which can be accessed extract_nested_test_forecast()

Usage

```
modeltime_nested_select_best(
  object,
  metric = "rmse",
  minimize = TRUE,
  filter_test_forecasts = TRUE
)
```

Arguments

`object` A Nested Modeltime Table

`metric` A metric to minimize or maximize. By default available metrics are:

- "rmse" (default)
- "mae"
- "mape"
- "mase"
- "smape"
- "rsq"

`minimize` Whether to minimize or maximize. Default: TRUE (minimize).

`filter_test_forecasts` Whether or not to update the test forecast log to filter only the best forecasts. Default: TRUE.

`modeltime_refit` *Refit one or more trained models to new data*

Description

This is a wrapper for `fit()` that takes a Modeltime Table and retrains each model on *new data* re-using the parameters and preprocessing steps used during the training process.

Usage

```
modeltime_refit(object, data, ..., control = control_refit())
```

Arguments

`object` A Modeltime Table

`data` A `tibble` that contains data to retrain the model(s) using.

`...` Additional arguments to control refitting.

Ensemble Model Spec (`modeltime.ensemble`)**:**

When making a meta-learner with `modeltime.ensemble::ensemble_model_spec()`, used to pass `resamples` argument containing results from `modeltime.resample::modeltime_fit_resa`

`control` Used to control verbosity and parallel processing. See `control_refit()`.

Details

Refitting is an important step prior to forecasting time series models. The `modeltime_refit()` function makes it easy to recycle models, retraining on new data.

Recycling Parameters

Parameters are recycled during retraining using the following criteria:

- **Automated models** (e.g. "auto arima") will have parameters recalculated.
- **Non-automated models** (e.g. "arima") will have parameters preserved.
- All preprocessing steps will be reused on the data

Refit

The `modeltime_refit()` function is used to retrain models trained with `fit()`.

Refit XY

The XY format is not supported at this time.

Value

A Modeltime Table containing one or more re-trained models.

See Also

`control_refit()`

Examples

```
library(dplyr)
library(lubridate)
library(timetk)
library(parsnip)
library(rsample)

# Data
m750 <- m4_monthly %>% filter(id == "M750")

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.9)

# --- MODELS ---

model_fit_prophet <- prophet_reg() %>%
    set_engine(engine = "prophet") %>%
    fit(value ~ date, data = training(splits))


# ---- MODELTIME TABLE ----

models_tbl <- modeltime_table(
    model_fit_prophet
)
```

```
# ---- CALIBRATE ----
# - Calibrate on training data set

calibration_tbl <- models_tbl %>%
    modeltime_calibrate(new_data = testing(splits))


# ---- REFIT ----
# - Refit on full data set

refit_tbl <- calibration_tbl %>%
    modeltime_refit(m750)
```

`modeltime_residuals` *Extract Residuals Information*

Description

This is a convenience function to unnest model residuals

Usage

```
modeltime_residuals(object, new_data = NULL, quiet = TRUE, ...)
```

Arguments

`object`	A Modeltime Table
`new_data`	A `tibble` to predict and calculate residuals on. If provided, overrides any calibration data.
`quiet`	Hide errors (`TRUE`, the default), or display them as they occur?
`...`	Not currently used.

Value

A tibble with residuals.

Examples

```
library(dplyr)
library(lubridate)
library(timetk)
library(parsnip)
library(rsample)

# Data
```

```
m750 <- m4_monthly %>% filter(id == "M750")

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.9)

# --- MODELS ---

# Model 1: prophet ----
model_fit_prophet <- prophet_reg() %>%
    set_engine(engine = "prophet") %>%
    fit(value ~ date, data = training(splits))


# ---- MODELTIME TABLE ----

models_tbl <- modeltime_table(
    model_fit_prophet
)

# ---- RESIDUALS ----

# In-Sample
models_tbl %>%
    modeltime_calibrate(new_data = training(splits)) %>%
    modeltime_residuals() %>%
    plot_modeltime_residuals(.interactive = FALSE)

# Out-of-Sample
models_tbl %>%
    modeltime_calibrate(new_data = testing(splits)) %>%
    modeltime_residuals() %>%
    plot_modeltime_residuals(.interactive = FALSE)
```

modeltime_residuals_test

Apply Statistical Tests to Residuals

Description

This is a convenience function to calculate some statistical tests on the residuals models. Currently, the following statistics are calculated: the shapiro.test to check the normality of the residuals, the box-pierce and ljung-box tests and the durbin watson test to check the autocorrelation of the residuals. In all cases the p-values are returned.

Usage

```
modeltime_residuals_test(object, new_data = NULL, lag = 1, fitdf = 0, ...)
```

Arguments

`object`	A `tibble` extracted from modeltime::modeltime_residuals().
`new_data`	A `tibble` to predict and calculate residuals on. If provided, overrides any calibration data.
`lag`	The statistic will be based on lag autocorrelation coefficients. Default: 1 (Applies to Box-Pierce, Ljung-Box, and Durbin-Watson Tests)
`fitdf`	Number of degrees of freedom to be subtracted. Default: 0 (Applies Box-Pierce and Ljung-Box Tests)
`...`	Not currently used

Details

Shapiro-Wilk Test

The Shapiro-Wilk tests the Normality of the residuals. The Null Hypothesis is that the residuals are normally distributed. A low P-Value below a given significance level indicates the values are NOT Normally Distributed.

If the **p-value > 0.05 (good)**, this implies that the distribution of the data are not significantly different from normal distribution. In other words, we can assume the normality.

Box-Pierce and Ljung-Box Tests Tests

The Ljung-Box and Box-Pierce tests are methods that test for the absense of autocorrelation in residuals. A low p-value below a given significance level indicates the values are autocorrelated.

If the **p-value > 0.05 (good)**, this implies that the residuals of the data are are independent. In other words, we can assume the residuals are not autocorrelated.

For more information about the parameters associated with the Box Pierce and Ljung Box tests check ?Box.Test

Durbin-Watson Test

The Durbin-Watson test is a method that tests for the absense of autocorrelation in residuals. The Durbin Watson test reports a test statistic, with a value from 0 to 4, where:

- **2 is no autocorrelation (good)**
- From 0 to <2 is positive autocorrelation (common in time series data)
- From >2 to 4 is negative autocorrelation (less common in time series data)

Value

A tibble with with the p-values of the calculated statistical tests.

See Also

`stats::shapiro.test()`, `stats::Box.test()`

Examples

```
library(dplyr)
library(timetk)
library(parsnip)
library(rsample)

# Data
m750 <- m4_monthly %>% filter(id == "M750")

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.9)

# --- MODELS ---

# Model 1: prophet ----
model_fit_prophet <- prophet_reg() %>%
    set_engine(engine = "prophet") %>%
    fit(value ~ date, data = training(splits))


# ---- MODELTIME TABLE ----

models_tbl <- modeltime_table(
    model_fit_prophet
)

# ---- RESIDUALS ----

# In-Sample
models_tbl %>%
    modeltime_calibrate(new_data = training(splits)) %>%
    modeltime_residuals() %>%
    modeltime_residuals_test()

# Out-of-Sample
models_tbl %>%
    modeltime_calibrate(new_data = testing(splits)) %>%
    modeltime_residuals() %>%
    modeltime_residuals_test()
```

`modeltime_table` *Scale forecast analysis with a Modeltime Table*

Description

Designed to perform forecasts at scale using models created with `modeltime`, `parsnip`, `workflows`, and regression modeling extensions in the `tidymodels` ecosystem.

Usage

```
modeltime_table(...)

as_modeltime_table(.l)
```

Arguments

`...`	Fitted `parsnip` model or `workflow` objects
`.l`	A list containing fitted `parsnip` model or `workflow` objects

Details

`modeltime_table()`:

1. Creates a table of models
2. Validates that all objects are models (parsnip or workflows objects) and all models have been fitted (trained)
3. Provides an ID and Description of the models

`as_modeltime_table()`:

Converts a `list` of models to a modeltime table. Useful if programatically creating Modeltime Tables from models stored in a `list`.

Examples

```
library(dplyr)
library(timetk)
library(parsnip)
library(rsample)

# Data
m750 <- m4_monthly %>% filter(id == "M750")

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.9)

# --- MODELS ---

# Model 1: prophet ----
model_fit_prophet <- prophet_reg() %>%
    set_engine(engine = "prophet") %>%
    fit(value ~ date, data = training(splits))


# ---- MODELTIME TABLE ----

# Make a Modeltime Table
models_tbl <- modeltime_table(
    model_fit_prophet
)
```

```
# Can also convert a list of models
list(model_fit_prophet) %>%
    as_modeltime_table()

# ---- CALIBRATE ----

calibration_tbl <- models_tbl %>%
    modeltime_calibrate(new_data = testing(splits))

# ---- ACCURACY ----

calibration_tbl %>%
    modeltime_accuracy()

# ---- FORECAST ----

calibration_tbl %>%
    modeltime_forecast(
        new_data    = testing(splits),
        actual_data = m750
    )
```

`naive_reg` *General Interface for NAIVE Forecast Models*

Description

`naive_reg()` is a way to generate a *specification* of an NAIVE or SNAIVE model before fitting and allows the model to be created using different packages.

Usage

```
naive_reg(mode = "regression", id = NULL, seasonal_period = NULL)
```

Arguments

`mode`	A single character string for the type of model. The only possible value for this model is "regression".
`id`	An optional quoted column name (e.g. "id") for identifying multiple time series (i.e. panel data).
`seasonal_period`	SNAIVE only. A seasonal frequency. Uses "auto" by default. A character phrase of "auto" or time-based phrase of "2 weeks" can be used if a date or date-time variable is provided. See Fit Details below.

Details

The data given to the function are not saved and are only used to determine the *mode* of the model. For `naive_reg()`, the mode will always be "regression".

The model can be created using the `fit()` function using the following *engines*:

- "naive" (default) - Performs a NAIVE forecast
- "snaive" - Performs a Seasonal NAIVE forecast

Engine Details

naive (default engine)

- The engine uses `naive_fit_impl()`
- The NAIVE implementation uses the last observation and forecasts this value forward.
- The `id` can be used to distinguish multiple time series contained in the data
- The `seasonal_period` is not used but provided for consistency with the SNAIVE implementation

snaive (default engine)

- The engine uses `snaive_fit_impl()`
- The SNAIVE implementation uses the last seasonal series in the data and forecasts this sequence of observations forward
- The `id` can be used to distinguish multiple time series contained in the data
- The `seasonal_period` is used to determine how far back to define the repeated series. This can be a numeric value (e.g. 28) or a period (e.g. "1 month")

Fit Details

Date and Date-Time Variable

It's a requirement to have a date or date-time variable as a predictor. The `fit()` interface accepts date and date-time features and handles them internally.

- `fit(y ~ date)`

ID features (Multiple Time Series, Panel Data)

The `id` parameter is populated using the `fit()` or `fit_xy()` function:

ID Example: Suppose you have 3 features:

1. `y` (target)
2. `date` (time stamp),
3. `series_id` (a unique identifer that identifies each time series in your data).

The `series_id` can be passed to the `naive_reg()` using `fit()`:

- `naive_reg(id = "series_id")` specifes that the `series_id` column should be used to identify each time series.

- `fit(y ~ date + series_id)` will pass `series_id` on to the underlying naive or snaive functions.

Seasonal Period Specification (snaive)

The period can be non-seasonal (`seasonal_period = 1 or "none"`) or yearly seasonal (e.g. For monthly time stamps, `seasonal_period = 12`, `seasonal_period = "12 months"`, or `seasonal_period = "yearly"`). There are 3 ways to specify:

1. `seasonal_period = "auto"`: A seasonal period is selected based on the periodicity of the data (e.g. 12 if monthly)
2. `seasonal_period = 12`: A numeric frequency. For example, 12 is common for monthly data
3. `seasonal_period = "1 year"`: A time-based phrase. For example, "1 year" would convert to 12 for monthly data.

External Regressors (Xregs)

These models are univariate. No xregs are used in the modeling process.

See Also

`fit.model_spec()`, `set_engine()`

Examples

```
library(dplyr)
library(parsnip)
library(rsample)
library(timetk)

# Data
m750 <- m4_monthly %>% filter(id == "M750")
m750

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.8)

# ---- NAIVE ----

# Model Spec
model_spec <- naive_reg() %>%
    set_engine("naive")

# Fit Spec
model_fit <- model_spec %>%
    fit(log(value) ~ date, data = training(splits))
model_fit


# ---- SEASONAL NAIVE ----

# Model Spec
model_spec <- naive_reg(
```

```
        id = "id",
        seasonal_period = 12
    ) %>%
    set_engine("snaive")

# Fit Spec
model_fit <- model_spec %>%
    fit(log(value) ~ date + id, data = training(splits))
model_fit
```

`new_modeltime_bridge` *Constructor for creating modeltime models*

Description

These functions are used to construct new `modeltime` bridge functions that connect the `tidymodels` infrastructure to time-series models containing date or date-time features.

Usage

```
new_modeltime_bridge(class, models, data, extras = NULL, desc = NULL)
```

Arguments

`class`	A class name that is used for creating custom printing messages
`models`	A list containing one or more models
`data`	A data frame (or tibble) containing 4 columns: (date column with name that matches input data), .actual, .fitted, and .residuals.
`extras`	An optional list that is typically used for transferring preprocessing recipes to the predict method.
`desc`	An optional model description to appear when printing your modeltime objects

Examples

```
library(dplyr)
library(lubridate)
library(timetk)

lm_model <- lm(value ~ as.numeric(date) + hour(date) + wday(date, label = TRUE),
               data = taylor_30_min)

data = tibble(
   date      = taylor_30_min$date, # Important - The column name must match the modeled data
    # These are standardized names: .actual, .fitted, .residuals
    .actual     = taylor_30_min$value,
    .fitted     = lm_model$fitted.values %>% as.numeric(),
    .residuals  = lm_model$residuals %>% as.numeric()
```

```
)

new_modeltime_bridge(
    class  = "lm_time_series_impl",
    models = list(model_1 = lm_model),
    data   = data,
    extras = NULL
)
```

`nnetar_params` *Tuning Parameters for NNETAR Models*

Description

Tuning Parameters for NNETAR Models

Usage

```
num_networks(range = c(1L, 100L), trans = NULL)
```

Arguments

`range`	A two-element vector holding the *defaults* for the smallest and largest possible values, respectively. If a transformation is specified, these values should be in the *transformed units*.
`trans`	A `trans` object from the `scales` package, such as `scales::transform_log10()` or `scales::transform_reciprocal()`. If not provided, the default is used which matches the units used in `range`. If no transformation, `NULL`.

Details

The main parameters for NNETAR models are:

- `non_seasonal_ar`: Number of non-seasonal auto-regressive (AR) lags. Often denoted "p" in pdq-notation.
- `seasonal_ar`: Number of seasonal auto-regressive (SAR) lags. Often denoted "P" in PDQ-notation.
- `hidden_units`: An integer for the number of units in the hidden model.
- `num_networks`: Number of networks to fit with different random starting weights. These are then averaged when producing forecasts.
- `penalty`: A non-negative numeric value for the amount of weight decay.
- `epochs`: An integer for the number of training iterations.

See Also

`non_seasonal_ar()`, `seasonal_ar()`, `dials::hidden_units()`, `dials::penalty()`, `dials::epochs()`

Examples

```
num_networks()
```

`nnetar_reg` *General Interface for NNETAR Regression Models*

Description

`nnetar_reg()` is a way to generate a *specification* of an NNETAR model before fitting and allows the model to be created using different packages. Currently the only package is `forecast`.

Usage

```
nnetar_reg(
  mode = "regression",
  seasonal_period = NULL,
  non_seasonal_ar = NULL,
  seasonal_ar = NULL,
  hidden_units = NULL,
  num_networks = NULL,
  penalty = NULL,
  epochs = NULL
)
```

Arguments

`mode`	A single character string for the type of model. The only possible value for this model is "regression".
`seasonal_period`	A seasonal frequency. Uses "auto" by default. A character phrase of "auto" or time-based phrase of "2 weeks" can be used if a date or date-time variable is provided. See Fit Details below.
`non_seasonal_ar`	The order of the non-seasonal auto-regressive (AR) terms. Often denoted "p" in pdq-notation.
`seasonal_ar`	The order of the seasonal auto-regressive (SAR) terms. Often denoted "P" in PDQ-notation.
`hidden_units`	An integer for the number of units in the hidden model.
`num_networks`	Number of networks to fit with different random starting weights. These are then averaged when producing forecasts.
`penalty`	A non-negative numeric value for the amount of weight decay.
`epochs`	An integer for the number of training iterations.

Details

The data given to the function are not saved and are only used to determine the *mode* of the model. For `nnetar_reg()`, the mode will always be "regression".

The model can be created using the `fit()` function using the following *engines*:

- "nnetar" (default) - Connects to `forecast::nnetar()`

Main Arguments

The main arguments (tuning parameters) for the model are the parameters in `nnetar_reg()` function. These arguments are converted to their specific names at the time that the model is fit.

Other options and argument can be set using `set_engine()` (See Engine Details below).

If parameters need to be modified, `update()` can be used in lieu of recreating the object from scratch.

Engine Details

The standardized parameter names in `modeltime` can be mapped to their original names in each engine:

modeltime	forecast::nnetar
seasonal_period	ts(frequency)
non_seasonal_ar	p (1)
seasonal_ar	P (1)
hidden_units	size (10)
num_networks	repeats (20)
epochs	maxit (100)
penalty	decay (0)

Other options can be set using `set_engine()`.

nnetar

The engine uses `forecast::nnetar()`.

Function Parameters:

```
#> function (y, p, P = 1, size, repeats = 20, xreg = NULL, lambda = NULL,
#>     model = NULL, subset = NULL, scale.inputs = TRUE, x = y, ...)
```

Parameter Notes:

- `xreg` - This is supplied via the parsnip / modeltime `fit()` interface (so don't provide this manually). See Fit Details (below).
- `size` - Is set to 10 by default. This differs from the `forecast` implementation
- `p` and `P` - Are set to 1 by default.
- `maxit` and `decay` are `nnet::nnet` parameters that are exposed in the `nnetar_reg()` interface. These are key tuning parameters.

Fit Details

Date and Date-Time Variable

It's a requirement to have a date or date-time variable as a predictor. The `fit()` interface accepts date and date-time features and handles them internally.

- `fit(y ~ date)`

Seasonal Period Specification

The period can be non-seasonal (`seasonal_period = 1 or "none"`) or yearly seasonal (e.g. For monthly time stamps, `seasonal_period = 12`, `seasonal_period = "12 months"`, or `seasonal_period = "yearly"`). There are 3 ways to specify:

1. `seasonal_period = "auto"`: A seasonal period is selected based on the periodicity of the data (e.g. 12 if monthly)
2. `seasonal_period = 12`: A numeric frequency. For example, 12 is common for monthly data
3. `seasonal_period = "1 year"`: A time-based phrase. For example, "1 year" would convert to 12 for monthly data.

Univariate (No xregs, Exogenous Regressors):

For univariate analysis, you must include a date or date-time feature. Simply use:

- Formula Interface (recommended): `fit(y ~ date)` will ignore xreg's.
- XY Interface: `fit_xy(x = data[,"date"], y = data$y)` will ignore xreg's.

Multivariate (xregs, Exogenous Regressors)

The `xreg` parameter is populated using the `fit()` or `fit_xy()` function:

- Only `factor`, `ordered factor`, and `numeric` data will be used as xregs.
- Date and Date-time variables are not used as xregs
- `character` data should be converted to factor.

Xreg Example: Suppose you have 3 features:

1. `y` (target)
2. `date` (time stamp),
3. `month.lbl` (labeled month as a ordered factor).

The `month.lbl` is an exogenous regressor that can be passed to the `nnetar_reg()` using `fit()`:

- `fit(y ~ date + month.lbl)` will pass `month.lbl` on as an exogenous regressor.
- `fit_xy(data[,c("date", "month.lbl")], y = data$y)` will pass x, where x is a data frame containing `month.lbl` and the `date` feature. Only `month.lbl` will be used as an exogenous regressor.

Note that date or date-time class values are excluded from `xreg`.

See Also

`fit.model_spec()`, `set_engine()`

Examples

```
library(dplyr)
library(parsnip)
library(rsample)
library(timetk)

# Data
m750 <- m4_monthly %>% filter(id == "M750")
m750

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.8)

# ---- NNETAR ----

# Model Spec
model_spec <- nnetar_reg() %>%
    set_engine("nnetar")

# Fit Spec
set.seed(123)
model_fit <- model_spec %>%
    fit(log(value) ~ date, data = training(splits))
model_fit
```

`panel_tail` *Filter the last N rows (Tail) for multiple time series*

Description

Filter the last N rows (Tail) for multiple time series

Usage

```
panel_tail(data, id, n)
```

Arguments

data	A data frame
id	An "id" feature indicating which column differentiates the time series panels
n	The number of rows to filter

Value

A data frame

See Also

- `recursive()` - used to generate recursive autoregressive models

Examples

```
library(timetk)

# Get the last 6 observations from each group
m4_monthly %>%
    panel_tail(id = id, n = 6)
```

---

parallel_start   *Start parallel clusters / plans*

Description

Start parallel clusters / plans

Usage

```
parallel_start(
  ...,
  .method = c("parallel", "spark", "future"),
  .export_vars = NULL,
  .packages = NULL
)

parallel_stop()
```

Arguments

`...`	Parameters passed to underlying functions (See Details Section)
`.method`	The method to create the parallel backend. Supports: • "future" - Uses the `future` package; foreach bridged via `doFuture` • "parallel" - Uses the `parallel` + `doParallel` packages • "spark" - Uses the `sparklyr` package
`.export_vars`	Environment variables that can be sent to the workers (not needed for "future")
`.packages`	Packages that can be sent to the workers (auto-handled by "future")

Details

future (`.method = "future"`):

Sets a `future::multisession` plan (portable across OSes) and registers a foreach backend via `doFuture::registerDoFuture()`. This avoids the tune foreach/future warning.

- Pass the first unnamed `...` argument as worker count (numeric) or omit to default to `parallelly::availableCores(l = FALSE)` or 2 if unknown.

parallel (`.method = "parallel"`):

1. `parallel::makeCluster(...)` 2) `doParallel::registerDoParallel(cl)`
2. Set `.libPaths()` on workers; optional `clusterExport` and package loads.

spark (`.method = "spark"`):

Requires `sparklyr::spark_connect()`; registers foreach via `sparklyr::registerDoSpark(...)`.

Examples

```
# Starts 2 clusters
parallel_start(2)

# Returns to sequential processing
parallel_stop()
```

`parse_index` *Developer Tools for parsing date and date-time information*

Description

These functions are designed to assist developers in extending the `modeltime` package.

Usage

```
parse_index_from_data(data)

parse_period_from_index(data, period)
```

Arguments

`data`	A data frame
`period`	A period to calculate from the time index. Numeric values are returned as-is. "auto" guesses a numeric value from the index. A time-based phrase (e.g. "7 days") calculates the number of timestamps that typically occur within the time-based phrase.

Value

- parse_index_from_data(): Returns a tibble containing the date or date-time column.
- parse_period_from_index(): Returns the numeric period from a tibble containing the index.

Examples

```
library(dplyr)
library(timetk)

predictors <- m4_monthly %>%
    filter(id == "M750") %>%
    select(-value)

index_tbl <- parse_index_from_data(predictors)
index_tbl

period <- parse_period_from_index(index_tbl, period = "1 year")
period
```

---

plot_modeltime_forecast

Interactive Forecast Visualization

Description

This is a wrapper for `timetk::plot_time_series()` that generates an interactive (`plotly`) or static (`ggplot2`) plot with the forecasted data.

Usage

```
plot_modeltime_forecast(
  .data,
  .conf_interval_show = TRUE,
  .conf_interval_fill = "grey20",
  .conf_interval_alpha = 0.2,
  .smooth = FALSE,
  .legend_show = TRUE,
  .legend_max_width = 40,
  .facet_ncol = 1,
  .facet_nrow = 1,
  .facet_scales = "free_y",
  .title = "Forecast Plot",
  .x_lab = "",
  .y_lab = "",
  .color_lab = "Legend",
  .interactive = TRUE,
```

```
    .plotly_slider = FALSE,
    .trelliscope = FALSE,
    .trelliscope_params = list(),
    ...
  )
```

Arguments

`.data` A `tibble` that is the output of `modeltime_forecast()`

`.conf_interval_show`
: Logical. Whether or not to include the confidence interval as a ribbon.

`.conf_interval_fill`
: Fill color for the confidence interval

`.conf_interval_alpha`
: Fill opacity for the confidence interval. Range (0, 1).

`.smooth` Logical - Whether or not to include a trendline smoother. Uses See `smooth_vec()` to apply a LOESS smoother.

`.legend_show` Logical. Whether or not to show the legend. Can save space with long model descriptions.

`.legend_max_width`
: Numeric. The width of truncation to apply to the legend text.

`.facet_ncol` Number of facet columns.

`.facet_nrow` Number of facet rows (only used for `.trelliscope = TRUE`)

`.facet_scales` Control facet x & y-axis ranges. Options include "fixed", "free", "free_y", "free_x"

`.title` Title for the plot

`.x_lab` X-axis label for the plot

`.y_lab` Y-axis label for the plot

`.color_lab` Legend label if a `color_var` is used.

`.interactive` Returns either a static (`ggplot2`) visualization or an interactive (`plotly`) visualization

`.plotly_slider` If `TRUE`, returns a plotly date range slider.

`.trelliscope` Returns either a normal plot or a trelliscopejs plot (great for many time series) Must have `trelliscopejs` installed.

`.trelliscope_params`
: Pass parameters to the `trelliscopejs::facet_trelliscope()` function as a `list()`. The only parameters that cannot be passed are:

- `ncol`: use `.facet_ncol`
- `nrow`: use `.facet_nrow`
- `scales`: use `facet_scales`
- `as_plotly`: use `.interactive`

`...` Additional arguments passed to `timetk::plot_time_series()`.

Value

A static ggplot2 plot or an interactive plotly plot containing a forecast

Examples

```
library(dplyr)
library(lubridate)
library(timetk)
library(parsnip)
library(rsample)

# Data
m750 <- m4_monthly %>% filter(id == "M750")

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.9)

# --- MODELS ---

# Model 1: prophet ----
model_fit_prophet <- prophet_reg() %>%
    set_engine(engine = "prophet") %>%
    fit(value ~ date, data = training(splits))


# ---- MODELTIME TABLE ----

models_tbl <- modeltime_table(
    model_fit_prophet
)

# ---- FORECAST ----

models_tbl %>%
    modeltime_calibrate(new_data = testing(splits)) %>%
    modeltime_forecast(
        new_data    = testing(splits),
        actual_data = m750
    ) %>%
    plot_modeltime_forecast(.interactive = FALSE)
```

`plot_modeltime_residuals` *Interactive Residuals Visualization*

Description

This is a wrapper for examining residuals using:

- Time Plot: `timetk::plot_time_series()`
- ACF Plot: `timetk::plot_acf_diagnostics()`
- Seasonality Plot: `timetk::plot_seasonal_diagnostics()`

Usage

```
plot_modeltime_residuals(
  .data,
  .type = c("timeplot", "acf", "seasonality"),
  .smooth = FALSE,
  .legend_show = TRUE,
  .legend_max_width = 40,
  .title = "Residuals Plot",
  .x_lab = "",
  .y_lab = "",
  .color_lab = "Legend",
  .interactive = TRUE,
  ...
)
```

Arguments

`.data`	A `tibble` that is the output of `modeltime_residuals()`
`.type`	One of "timeplot", "acf", or "seasonality". The default is "timeplot".
`.smooth`	Logical - Whether or not to include a trendline smoother. Uses See `smooth_vec()` to apply a LOESS smoother.
`.legend_show`	Logical. Whether or not to show the legend. Can save space with long model descriptions.
`.legend_max_width`	Numeric. The width of truncation to apply to the legend text.
`.title`	Title for the plot
`.x_lab`	X-axis label for the plot
`.y_lab`	Y-axis label for the plot
`.color_lab`	Legend label if a `color_var` is used.
`.interactive`	Returns either a static (`ggplot2`) visualization or an interactive (`plotly`) visualization
`...`	Additional arguments passed to: • Time Plot: `timetk::plot_time_series()` • ACF Plot: `timetk::plot_acf_diagnostics()` • Seasonality Plot: `timetk::plot_seasonal_diagnostics()`

Value

A static `ggplot2` plot or an interactive `plotly` plot containing residuals vs time

Examples

```
library(dplyr)
library(timetk)
library(parsnip)
library(rsample)

# Data
m750 <- m4_monthly %>% filter(id == "M750")

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.9)

# --- MODELS ---

# Model 1: prophet ----
model_fit_prophet <- prophet_reg() %>%
    set_engine(engine = "prophet") %>%
    fit(value ~ date, data = training(splits))


# ---- MODELTIME TABLE ----

models_tbl <- modeltime_table(
    model_fit_prophet
)

# ---- RESIDUALS ----

residuals_tbl <- models_tbl %>%
    modeltime_calibrate(new_data = testing(splits)) %>%
    modeltime_residuals()

residuals_tbl %>%
    plot_modeltime_residuals(
        .type = "timeplot",
        .interactive = FALSE
    )
```

`pluck_modeltime_model` *Extract model by model id in a Modeltime Table*

Description

The `pull_modeltime_model()` and `pluck_modeltime_model()` functions are synonymns.

Usage

```
pluck_modeltime_model(object, .model_id)
```

```
## S3 method for class 'mdl_time_tbl'
pluck_modeltime_model(object, .model_id)

pull_modeltime_model(object, .model_id)
```

Arguments

object	A Modeltime Table
.model_id	A numeric value matching the .model_id that you want to update

See Also

- combine_modeltime_tables(): Combine 2 or more Modeltime Tables together
- add_modeltime_model(): Adds a new row with a new model to a Modeltime Table
- drop_modeltime_model(): Drop one or more models from a Modeltime Table
- update_modeltime_description(): Updates a description for a model inside a Modeltime Table
- update_modeltime_model(): Updates a model inside a Modeltime Table
- pull_modeltime_model(): Extracts a model from a Modeltime Table

Examples

```
m750_models %>%
    pluck_modeltime_model(2)
```

---

prep_nested *Prepared Nested Modeltime Data*

Description

A set of functions to simplify preparation of nested data for iterative (nested) forecasting with Nested Modeltime Tables.

Usage

```
extend_timeseries(.data, .id_var, .date_var, .length_future, ...)

nest_timeseries(.data, .id_var, .length_future, .length_actual = NULL)

split_nested_timeseries(.data, .length_test, .length_train = NULL, ...)
```

Arguments

`.data` A data frame or tibble containing time series data. The data should have:

- identifier (.id_var): Identifying one or more time series groups
- date variable (.date_var): A date or date time column
- target variable (.value): A column containing numeric values that is to be forecasted

`.id_var` An id column

`.date_var` A date or datetime column

`.length_future` Varies based on the function:

- `extend_timeseries()`: Defines how far into the future to extend the time series by each time series group.
- `nest_timeseries()`: Defines which observations should be split into the `.future_data`.

`...` Additional arguments passed to the helper function. See details.

`.length_actual` Can be used to slice the `.actual_data` to a most recent number of observations.

`.length_test` Defines the length of the test split for evaluation.

`.length_train` Defines the length of the training split for evaluation.

Details

Preparation of nested time series follows a 3-Step Process:

Step 1: Extend the Time Series:

`extend_timeseries()`: A wrapper for `timetk::future_frame()` that extends a time series group-wise into the future.

- The group column is specified by `.id_var`.
- The date column is specified by `.date_var`.
- The length into the future is specified with `.length_future`.
- The `...` are additional parameters that can be passed to `timetk::future_frame()`

Step 2: Nest the Time Series:

`nest_timeseries()`: A helper for nesting your data into `.actual_data` and `.future_data`.

- The group column is specified by `.id_var`
- The `.length_future` defines the length of the `.future_data`.
- The remaining data is converted to the `.actual_data`.
- The `.length_actual` can be used to slice the `.actual_data` to a most recent number of observations.

The result is a "nested data frame".

Step 3: Split the Actual Data into Train/Test Splits:

`split_nested_timeseries()`: A wrapper for `timetk::time_series_split()` that generates training/testing splits from the `.actual_data` column.

- The `.length_test` is the primary argument that identifies the size of the testing sample. This is typically the same size as the `.future_data`.
- The `.length_train` is an optional size of the training data.
- The `...` (dots) are additional arguments that can be passed to `timetk::time_series_split()`.

Helpers:

`extract_nested_train_split()` and `extract_nested_test_split()` are used to simplify extracting the training and testing data from the actual data. This can be helpful when making preprocessing recipes using the `recipes` package.

Examples

```
library(dplyr)
library(timetk)


nested_data_tbl <- walmart_sales_weekly %>%
    select(id, date = Date, value = Weekly_Sales) %>%

    # Step 1: Extends the time series by id
    extend_timeseries(
        .id_var     = id,
        .date_var   = date,
        .length_future = 52
    ) %>%

    # Step 2: Nests the time series into .actual_data and .future_data
    nest_timeseries(
        .id_var     = id,
        .length_future = 52
    ) %>%

    # Step 3: Adds a column .splits that contains training/testing indices
    split_nested_timeseries(
        .length_test = 52
    )

nested_data_tbl

# Helpers: Getting the Train/Test Sets
extract_nested_train_split(nested_data_tbl, .row_id = 1)
```

`prophet_boost` *General Interface for Boosted PROPHET Time Series Models*

Description

`prophet_boost()` is a way to generate a *specification* of a Boosted PROPHET model before fitting and allows the model to be created using different packages. Currently the only package is `prophet`.

Usage

```
prophet_boost(
  mode = "regression",
  growth = NULL,
  changepoint_num = NULL,
  changepoint_range = NULL,
  seasonality_yearly = NULL,
  seasonality_weekly = NULL,
  seasonality_daily = NULL,
  season = NULL,
  prior_scale_changepoints = NULL,
  prior_scale_seasonality = NULL,
  prior_scale_holidays = NULL,
  logistic_cap = NULL,
  logistic_floor = NULL,
  mtry = NULL,
  trees = NULL,
  min_n = NULL,
  tree_depth = NULL,
  learn_rate = NULL,
  loss_reduction = NULL,
  sample_size = NULL,
  stop_iter = NULL
)
```

Arguments

`mode` A single character string for the type of model. The only possible value for this model is "regression".

`growth` String 'linear' or 'logistic' to specify a linear or logistic trend.

`changepoint_num` Number of potential changepoints to include for modeling trend.

`changepoint_range` Adjusts the flexibility of the trend component by limiting to a percentage of data before the end of the time series. 0.80 means that a changepoint cannot exist after the first 80% of the data.

`seasonality_yearly` One of "auto", TRUE or FALSE. Toggles on/off a seasonal component that models year-over-year seasonality.

`seasonality_weekly` One of "auto", TRUE or FALSE. Toggles on/off a seasonal component that models week-over-week seasonality.

`seasonality_daily` One of "auto", TRUE or FALSE. Toggles on/off a seasonal componet that models day-over-day seasonality.

`season` 'additive' (default) or 'multiplicative'.

prior_scale_changepoints
: Parameter modulating the flexibility of the automatic changepoint selection. Large values will allow many changepoints, small values will allow few changepoints.

prior_scale_seasonality
: Parameter modulating the strength of the seasonality model. Larger values allow the model to fit larger seasonal fluctuations, smaller values dampen the seasonality.

prior_scale_holidays
: Parameter modulating the strength of the holiday components model, unless overridden in the holidays input.

logistic_cap
: When growth is logistic, the upper-bound for "saturation".

logistic_floor
: When growth is logistic, the lower-bound for "saturation".

mtry
: A number for the number (or proportion) of predictors that will be randomly sampled at each split when creating the tree models (specific engines only).

trees
: An integer for the number of trees contained in the ensemble.

min_n
: An integer for the minimum number of data points in a node that is required for the node to be split further.

tree_depth
: An integer for the maximum depth of the tree (i.e. number of splits) (specific engines only).

learn_rate
: A number for the rate at which the boosting algorithm adapts from iteration-to-iteration (specific engines only). This is sometimes referred to as the shrinkage parameter.

loss_reduction
: A number for the reduction in the loss function required to split further (specific engines only).

sample_size
: number for the number (or proportion) of data that is exposed to the fitting routine.

stop_iter
: The number of iterations without improvement before stopping (`xgboost` only).

Details

The data given to the function are not saved and are only used to determine the *mode* of the model. For `prophet_boost()`, the mode will always be "regression".

The model can be created using the `fit()` function using the following *engines*:

- "prophet_xgboost" (default) - Connects to `prophet::prophet()` and `xgboost::xgb.train()`

Main Arguments

The main arguments (tuning parameters) for the **PROPHET** model are:

- `growth`: String 'linear' or 'logistic' to specify a linear or logistic trend.
- `changepoint_num`: Number of potential changepoints to include for modeling trend.
- `changepoint_range`: Range changepoints that adjusts how close to the end the last changepoint can be located.
- `season`: 'additive' (default) or 'multiplicative'.

- `prior_scale_changepoints`: Parameter modulating the flexibility of the automatic changepoint selection. Large values will allow many changepoints, small values will allow few changepoints.
- `prior_scale_seasonality`: Parameter modulating the strength of the seasonality model. Larger values allow the model to fit larger seasonal fluctuations, smaller values dampen the seasonality.
- `prior_scale_holidays`: Parameter modulating the strength of the holiday components model, unless overridden in the holidays input.
- `logistic_cap`: When growth is logistic, the upper-bound for "saturation".
- `logistic_floor`: When growth is logistic, the lower-bound for "saturation".

The main arguments (tuning parameters) for the model **XGBoost model** are:

- `mtry`: The number of predictors that will be randomly sampled at each split when creating the tree models.
- `trees`: The number of trees contained in the ensemble.
- `min_n`: The minimum number of data points in a node that are required for the node to be split further.
- `tree_depth`: The maximum depth of the tree (i.e. number of splits).
- `learn_rate`: The rate at which the boosting algorithm adapts from iteration-to-iteration.
- `loss_reduction`: The reduction in the loss function required to split further.
- `sample_size`: The amount of data exposed to the fitting routine.
- `stop_iter`: The number of iterations without improvement before stopping.

These arguments are converted to their specific names at the time that the model is fit.

Other options and argument can be set using `set_engine()` (See Engine Details below).

If parameters need to be modified, `update()` can be used in lieu of recreating the object from scratch.

Engine Details

The standardized parameter names in `modeltime` can be mapped to their original names in each engine:

Model 1: PROPHET:

modeltime	prophet
growth	growth ('linear')
changepoint_num	n.changepoints (25)
changepoint_range	changepoints.range (0.8)
seasonality_yearly	yearly.seasonality ('auto')
seasonality_weekly	weekly.seasonality ('auto')
seasonality_daily	daily.seasonality ('auto')
season	seasonality.mode ('additive')
prior_scale_changepoints	changepoint.prior.scale (0.05)
prior_scale_seasonality	seasonality.prior.scale (10)

prior_scale_holidays	holidays.prior.scale (10)
logistic_cap	df$cap (NULL)
logistic_floor	df$floor (NULL)

Model 2: XGBoost:

modeltime	xgboost::xgb.train
tree_depth	max_depth (6)
trees	nrounds (15)
learn_rate	eta (0.3)
mtry	colsample_bynode (1)
min_n	min_child_weight (1)
loss_reduction	gamma (0)
sample_size	subsample (1)
stop_iter	early_stop

Other options can be set using `set_engine()`.

prophet_xgboost

Model 1: PROPHET (`prophet::prophet`):

```
#> function (df = NULL, growth = "linear", changepoints = NULL, n.changepoints = 25,
#>     changepoint.range = 0.8, yearly.seasonality = "auto", weekly.seasonality = "auto",
#>     daily.seasonality = "auto", holidays = NULL, seasonality.mode = "additive",
#>     seasonality.prior.scale = 10, holidays.prior.scale = 10, changepoint.prior.scale = 0.05,
#>     mcmc.samples = 0, interval.width = 0.8, uncertainty.samples = 1000,
#>     fit = TRUE, ...)
```

Parameter Notes:

- `df`: This is supplied via the parsnip / modeltime `fit()` interface (so don't provide this manually). See Fit Details (below).
- `holidays`: A data.frame of holidays can be supplied via `set_engine()`
- `uncertainty.samples`: The default is set to 0 because the prophet uncertainty intervals are not used as part of the Modeltime Workflow. You can override this setting if you plan to use prophet's uncertainty tools.

Logistic Growth and Saturation Levels:

- For `growth = "logistic"`, simply add numeric values for `logistic_cap` and / or `logistic_floor`. There is *no need* to add additional columns for "cap" and "floor" to your data frame.

Limitations:

- `prophet::add_seasonality()` is not currently implemented. It's used to specify non-standard seasonalities using fourier series. An alternative is to use `step_fourier()` and supply custom seasonalities as Extra Regressors.

Model 2: XGBoost (`xgboost::xgb.train`):

```
#> function (params = xgb.params(), data, nrounds, evals = list(), objective = NULL,
#>    custom_metric = NULL, verbose = 1, print_every_n = 1L, early_stopping_rounds = NULL,
#>    maximize = NULL, save_period = NULL, save_name = "xgboost.model", xgb_model = NULL,
#>      callbacks = list(), ...)
```

Parameter Notes:

- XGBoost uses a `params = list()` to capture. Parsnip / Modeltime automatically sends any args provided as `...` inside of `set_engine()` to the `params = list(...)`.

Fit Details

Date and Date-Time Variable

It's a requirement to have a date or date-time variable as a predictor. The `fit()` interface accepts date and date-time features and handles them internally.

- `fit(y ~ date)`

Univariate (No Extra Regressors):

For univariate analysis, you must include a date or date-time feature. Simply use:

- Formula Interface (recommended): `fit(y ~ date)` will ignore xreg's.
- XY Interface: `fit_xy(x = data[,"date"], y = data$y)` will ignore xreg's.

Multivariate (Extra Regressors)

Extra Regressors parameter is populated using the `fit()` or `fit_xy()` function:

- Only `factor`, `ordered factor`, and `numeric` data will be used as xregs.
- Date and Date-time variables are not used as xregs
- `character` data should be converted to factor.

Xreg Example: Suppose you have 3 features:

1. `y` (target)
2. `date` (time stamp),
3. `month.lbl` (labeled month as a ordered factor).

The `month.lbl` is an exogenous regressor that can be passed to the `arima_reg()` using `fit()`:

- `fit(y ~ date + month.lbl)` will pass `month.lbl` on as an exogenous regressor.
- `fit_xy(data[,c("date", "month.lbl")], y = data$y)` will pass x, where x is a data frame containing `month.lbl` and the `date` feature. Only `month.lbl` will be used as an exogenous regressor.

Note that date or date-time class values are excluded from `xreg`.

See Also

`fit.model_spec()`, `set_engine()`

Examples

```
library(dplyr)
library(lubridate)
library(parsnip)
library(rsample)
library(timetk)

# Data
m750 <- m4_monthly %>% filter(id == "M750")
m750

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.8)

# ---- PROPHET ----

# Model Spec
model_spec <- prophet_boost(
    learn_rate = 0.1
) %>%
    set_engine("prophet_xgboost")

# Fit Spec

model_fit <- model_spec %>%
    fit(log(value) ~ date + as.numeric(date) + month(date, label = TRUE),
        data = training(splits))
model_fit
```

---

prophet_params — *Tuning Parameters for Prophet Models*

Description

Tuning Parameters for Prophet Models

Usage

```
growth(values = c("linear", "logistic"))

changepoint_num(range = c(0L, 50L), trans = NULL)

changepoint_range(range = c(0.6, 0.9), trans = NULL)

seasonality_yearly(values = c(TRUE, FALSE))
```

```
seasonality_weekly(values = c(TRUE, FALSE))

seasonality_daily(values = c(TRUE, FALSE))

prior_scale_changepoints(range = c(-3, 2), trans = log10_trans())

prior_scale_seasonality(range = c(-3, 2), trans = log10_trans())

prior_scale_holidays(range = c(-3, 2), trans = log10_trans())
```

Arguments

values	A character string of possible values.
range	A two-element vector holding the *defaults* for the smallest and largest possible values, respectively. If a transformation is specified, these values should be in the *transformed units*.
trans	A `trans` object from the `scales` package, such as `scales::transform_log10()` or `scales::transform_reciprocal()`. If not provided, the default is used which matches the units used in `range`. If no transformation, `NULL`.

Details

The main parameters for Prophet models are:

- `growth`: The form of the trend: "linear", or "logistic".
- `changepoint_num`: The maximum number of trend changepoints allowed when modeling the trend
- `changepoint_range`: The range affects how close the changepoints can go to the end of the time series. The larger the value, the more flexible the trend.
- Yearly, Weekly, and Daily Seasonality:
  - *Yearly*: `seasonality_yearly` - Useful when seasonal patterns appear year-over-year
  - *Weekly*: `seasonality_weekly` - Useful when seasonal patterns appear week-over-week (e.g. daily data)
  - *Daily*: `seasonality_daily` - Useful when seasonal patterns appear day-over-day (e.g. hourly data)
- `season`:
  - The form of the seasonal term: "additive" or "multiplicative".
  - See `season()`.
- "Prior Scale": Controls flexibility of
  - *Changepoints:* `prior_scale_changepoints`
  - *Seasonality:* `prior_scale_seasonality`
  - *Holidays:* `prior_scale_holidays`
  - The `log10_trans()` converts priors to a scale from 0.001 to 100, which effectively weights lower values more heavily than larger values.

Examples

```
growth()

changepoint_num()

season()

prior_scale_changepoints()
```

---

prophet_reg   *General Interface for PROPHET Time Series Models*

Description

`prophet_reg()` is a way to generate a *specification* of a PROPHET model before fitting and allows the model to be created using different packages. Currently the only package is `prophet`.

Usage

```
prophet_reg(
  mode = "regression",
  growth = NULL,
  changepoint_num = NULL,
  changepoint_range = NULL,
  seasonality_yearly = NULL,
  seasonality_weekly = NULL,
  seasonality_daily = NULL,
  season = NULL,
  prior_scale_changepoints = NULL,
  prior_scale_seasonality = NULL,
  prior_scale_holidays = NULL,
  logistic_cap = NULL,
  logistic_floor = NULL
)
```

Arguments

`mode` A single character string for the type of model. The only possible value for this model is "regression".

`growth` String 'linear' or 'logistic' to specify a linear or logistic trend.

`changepoint_num` Number of potential changepoints to include for modeling trend.

`changepoint_range` Adjusts the flexibility of the trend component by limiting to a percentage of data before the end of the time series. 0.80 means that a changepoint cannot exist after the first 80% of the data.

`seasonality_yearly`
: One of "auto", TRUE or FALSE. Toggles on/off a seasonal component that models year-over-year seasonality.

`seasonality_weekly`
: One of "auto", TRUE or FALSE. Toggles on/off a seasonal component that models week-over-week seasonality.

`seasonality_daily`
: One of "auto", TRUE or FALSE. Toggles on/off a seasonal componet that models day-over-day seasonality.

`season`
: 'additive' (default) or 'multiplicative'.

`prior_scale_changepoints`
: Parameter modulating the flexibility of the automatic changepoint selection. Large values will allow many changepoints, small values will allow few changepoints.

`prior_scale_seasonality`
: Parameter modulating the strength of the seasonality model. Larger values allow the model to fit larger seasonal fluctuations, smaller values dampen the seasonality.

`prior_scale_holidays`
: Parameter modulating the strength of the holiday components model, unless overridden in the holidays input.

`logistic_cap`
: When growth is logistic, the upper-bound for "saturation".

`logistic_floor`
: When growth is logistic, the lower-bound for "saturation".

Details

The data given to the function are not saved and are only used to determine the *mode* of the model. For `prophet_reg()`, the mode will always be "regression".

The model can be created using the `fit()` function using the following *engines*:

- "prophet" (default) - Connects to `prophet::prophet()`

Main Arguments

The main arguments (tuning parameters) for the model are:

- `growth`: String 'linear' or 'logistic' to specify a linear or logistic trend.
- `changepoint_num`: Number of potential changepoints to include for modeling trend.
- `changepoint_range`: Range changepoints that adjusts how close to the end the last changepoint can be located.
- `season`: 'additive' (default) or 'multiplicative'.
- `prior_scale_changepoints`: Parameter modulating the flexibility of the automatic changepoint selection. Large values will allow many changepoints, small values will allow few changepoints.
- `prior_scale_seasonality`: Parameter modulating the strength of the seasonality model. Larger values allow the model to fit larger seasonal fluctuations, smaller values dampen the seasonality.

- `prior_scale_holidays`: Parameter modulating the strength of the holiday components model, unless overridden in the holidays input.
- `logistic_cap`: When growth is logistic, the upper-bound for "saturation".
- `logistic_floor`: When growth is logistic, the lower-bound for "saturation".

These arguments are converted to their specific names at the time that the model is fit.

Other options and argument can be set using `set_engine()` (See Engine Details below).

If parameters need to be modified, `update()` can be used in lieu of recreating the object from scratch.

Engine Details

The standardized parameter names in `modeltime` can be mapped to their original names in each engine:

modeltime	prophet
growth	growth ('linear')
changepoint_num	n.changepoints (25)
changepoint_range	changepoints.range (0.8)
seasonality_yearly	yearly.seasonality ('auto')
seasonality_weekly	weekly.seasonality ('auto')
seasonality_daily	daily.seasonality ('auto')
season	seasonality.mode ('additive')
prior_scale_changepoints	changepoint.prior.scale (0.05)
prior_scale_seasonality	seasonality.prior.scale (10)
prior_scale_holidays	holidays.prior.scale (10)
logistic_cap	df$cap (NULL)
logistic_floor	df$floor (NULL)

Other options can be set using `set_engine()`.

prophet

The engine uses `prophet::prophet()`.

Function Parameters:

```
#> function (df = NULL, growth = "linear", changepoints = NULL, n.changepoints = 25,
#>     changepoint.range = 0.8, yearly.seasonality = "auto", weekly.seasonality = "auto",
#>     daily.seasonality = "auto", holidays = NULL, seasonality.mode = "additive",
#>     seasonality.prior.scale = 10, holidays.prior.scale = 10, changepoint.prior.scale = 0.05,
#>     mcmc.samples = 0, interval.width = 0.8, uncertainty.samples = 1000,
#>     fit = TRUE, ...)
```

Parameter Notes:

- `df`: This is supplied via the parsnip / modeltime `fit()` interface (so don't provide this manually). See Fit Details (below).
- `holidays`: A data.frame of holidays can be supplied via `set_engine()`

- `uncertainty.samples`: The default is set to 0 because the prophet uncertainty intervals are not used as part of the Modeltime Workflow. You can override this setting if you plan to use prophet's uncertainty tools.

Regressors:

- Regressors are provided via the `fit()` or `recipes` interface, which passes regressors to `prophet::add_regressor()`
- Parameters can be controlled in `set_engine()` via: `regressors.prior.scale`, `regressors.standardize`, and `regressors.mode`
- The regressor prior scale implementation default is `regressors.prior.scale = 1e4`, which deviates from the `prophet` implementation (defaults to holidays.prior.scale)

Logistic Growth and Saturation Levels:

- For `growth = "logistic"`, simply add numeric values for `logistic_cap` and / or `logistic_floor`. There is *no need* to add additional columns for "cap" and "floor" to your data frame.

Limitations:

- `prophet::add_seasonality()` is not currently implemented. It's used to specify non-standard seasonalities using fourier series. An alternative is to use `step_fourier()` and supply custom seasonalities as Extra Regressors.

Fit Details

Date and Date-Time Variable

It's a requirement to have a date or date-time variable as a predictor. The `fit()` interface accepts date and date-time features and handles them internally.

- `fit(y ~ date)`

Univariate (No Extra Regressors):

For univariate analysis, you must include a date or date-time feature. Simply use:

- Formula Interface (recommended): `fit(y ~ date)` will ignore xreg's.
- XY Interface: `fit_xy(x = data[,"date"], y = data$y)` will ignore xreg's.

Multivariate (Extra Regressors)

Extra Regressors parameter is populated using the `fit()` or `fit_xy()` function:

- Only `factor`, `ordered factor`, and `numeric` data will be used as xregs.
- Date and Date-time variables are not used as xregs
- `character` data should be converted to factor.

Xreg Example: Suppose you have 3 features:

1. `y` (target)
2. `date` (time stamp),
3. `month.lbl` (labeled month as a ordered factor).

The `month.lbl` is an exogenous regressor that can be passed to the `arima_reg()` using `fit()`:

- `fit(y ~ date + month.lbl)` will pass `month.lbl` on as an exogenous regressor.
- `fit_xy(data[,c("date", "month.lbl")], y = data$y)` will pass x, where x is a data frame containing `month.lbl` and the `date` feature. Only `month.lbl` will be used as an exogenous regressor.

Note that date or date-time class values are excluded from `xreg`.

See Also

`fit.model_spec()`, `set_engine()`

Examples

```
library(dplyr)
library(parsnip)
library(rsample)
library(timetk)

# Data
m750 <- m4_monthly %>% filter(id == "M750")
m750

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.8)

# ---- PROPHET ----

# Model Spec
model_spec <- prophet_reg() %>%
    set_engine("prophet")

# Fit Spec
model_fit <- model_spec %>%
    fit(log(value) ~ date, data = training(splits))
model_fit
```

---

pull_modeltime_residuals

Extracts modeltime residuals data from a Modeltime Model

Description

If a modeltime model contains `data` with residuals information, this function will extract the data frame.

Usage

```
pull_modeltime_residuals(object)
```

Arguments

`object`	A fitted `parsnip` / `modeltime` model or `workflow`

Value

A `tibble` containing the model timestamp, actual, fitted, and residuals data

---

`pull_parsnip_preprocessor` *Pulls the Formula from a Fitted Parsnip Model Object*

Description

Pulls the Formula from a Fitted Parsnip Model Object

Usage

```
pull_parsnip_preprocessor(object)
```

Arguments

`object`	A fitted parsnip model `model_fit` object

Value

A formula using `stats::formula()`

`recipe_helpers` *Developer Tools for processing XREGS (Regressors)*

Description

Wrappers for using `recipes::bake` and `recipes::juice` to process data returning data in either `data frame` or `matrix` format (Common formats needed for machine learning algorithms).

Usage

```
juice_xreg_recipe(recipe, format = c("tbl", "matrix"))

bake_xreg_recipe(recipe, new_data, format = c("tbl", "matrix"))
```

Arguments

recipe — A prepared recipe

format — One of:

- `tbl`: Returns a tibble (data.frame)
- `matrix`: Returns a matrix

new_data — Data to be processed by a recipe

Value

Data in either the `tbl` (data.frame) or `matrix` formats

Examples

```
library(dplyr)
library(timetk)
library(recipes)
library(lubridate)

predictors <- m4_monthly %>%
    filter(id == "M750") %>%
    select(-value) %>%
    mutate(month = month(date, label = TRUE))
predictors

# Create default recipe
xreg_recipe_spec <- create_xreg_recipe(predictors, prepare = TRUE)

# Extracts the preprocessed training data from the recipe (used in your fit function)
juice_xreg_recipe(xreg_recipe_spec)

# Applies the prepared recipe to new data (used in your predict function)
bake_xreg_recipe(xreg_recipe_spec, new_data = predictors)
```

---

recursive — *Create a Recursive Time Series Model from a Parsnip or Workflow Regression Model*

Description

Create a Recursive Time Series Model from a Parsnip or Workflow Regression Model

Usage

```
recursive(object, transform, train_tail, id = NULL, chunk_size = 1, ...)
```

Arguments

object	An object of `model_fit` or a fitted `workflow` class
transform	A transformation performed on `new_data` after each step of recursive algorithm. • **Transformation Function:** Must have one argument `data` (see examples)
train_tail	A tibble with tail of training data set. In most cases it'll be required to create some variables based on dependent variable.
id	(Optional) An identifier that can be provided to perform a panel forecast. A single quoted column name (e.g. `id = "id"`).
chunk_size	The size of the smallest lag used in `transform`. If the smallest lag necessary is n, the forecasts can be computed in chunks of n, which can dramatically improve performance. Defaults to 1. Non-integers are coerced to integer, e.g. `chunk_size = 3.5` will be coerced to integer via `as.integer()`.
...	Not currently used.

Details

What is a Recursive Model?

A *recursive model* uses predictions to generate new values for independent features. These features are typically lags used in autoregressive models. It's important to understand that a recursive model is only needed when the **Lag Size < Forecast Horizon.**

Why is Recursive needed for Autoregressive Models with Lag Size < Forecast Horizon?

When the lag length is less than the forecast horizon, a problem exists were missing values (`NA`) are generated in the future data. A solution that `recursive()` implements is to iteratively fill these missing values in with values generated from predictions.

Recursive Process

When producing forecast, the following steps are performed:

1. Computing forecast for first row of new data. The first row cannot contain NA in any required column.
2. Filling i-th place of the dependent variable column with already computed forecast.
3. Computing missing features for next step, based on already calculated prediction. These features are computed with on a tibble object made from binded `train_tail` (i.e. tail of training data set) and `new_data` (which is an argument of predict function).
4. Jumping into point 2., and repeating rest of steps till the for-loop is ended.

Recursion for Panel Data

Panel data is time series data with multiple groups identified by an ID column. The `recursive()` function can be used for Panel Data with the following modifications:

1. Supply an `id` column as a quoted column name
2. Replace `tail()` with `panel_tail()` to use tails for each time series group.

Value

An object with added `recursive` class

See Also

- `panel_tail()` - Used to generate tails for multiple time series groups.

Examples

```
# Libraries & Setup ----
library(tidymodels)
library(dplyr)
library(tidyr)
library(timetk)
library(slider)

# ---- SINGLE TIME SERIES (NON-PANEL) -----

m750

FORECAST_HORIZON <- 24

m750_extended <- m750 %>%
    group_by(id) %>%
    future_frame(
        .length_out = FORECAST_HORIZON,
        .bind_data  = TRUE
    ) %>%
    ungroup()

# TRANSFORM FUNCTION ----
# - Function runs recursively that updates the forecasted dataset
lag_roll_transformer <- function(data){
    data %>%
        # Lags
        tk_augment_lags(value, .lags = 1:12) %>%
        # Rolling Features
        mutate(rolling_mean_12 = lag(slide_dbl(
            value, .f = mean, .before = 12, .complete = FALSE
        ), 1))
}

# Data Preparation
m750_rolling <- m750_extended %>%
    lag_roll_transformer() %>%
    select(-id)

train_data <- m750_rolling %>%
    drop_na()

future_data <- m750_rolling %>%
    filter(is.na(value))

# Modeling
```

```
# Straight-Line Forecast
model_fit_lm <- linear_reg() %>%
    set_engine("lm") %>%
    # Use only date feature as regressor
    fit(value ~ date, data = train_data)

# Autoregressive Forecast
model_fit_lm_recursive <- linear_reg() %>%
    set_engine("lm") %>%
    # Use date plus all lagged features
    fit(value ~ ., data = train_data) %>%
    # Add recursive() w/ transformer and train_tail
    recursive(
        transform  = lag_roll_transformer,
        train_tail = tail(train_data, FORECAST_HORIZON)
    )

model_fit_lm_recursive

# Forecasting
modeltime_table(
    model_fit_lm,
    model_fit_lm_recursive
) %>%
    update_model_description(2, "LM - Lag Roll") %>%
    modeltime_forecast(
        new_data    = future_data,
        actual_data = m750
    ) %>%
    plot_modeltime_forecast(
        .interactive        = FALSE,
        .conf_interval_show = FALSE
    )

# MULTIPLE TIME SERIES (PANEL DATA) -----

m4_monthly

FORECAST_HORIZON <- 24

m4_extended <- m4_monthly %>%
    group_by(id) %>%
    future_frame(
        .length_out = FORECAST_HORIZON,
        .bind_data  = TRUE
    ) %>%
    ungroup()

# TRANSFORM FUNCTION ----
# - NOTE - We create lags by group
lag_transformer_grouped <- function(data){
    data %>%
        group_by(id) %>%
```

```
        tk_augment_lags(value, .lags = 1:FORECAST_HORIZON) %>%
        ungroup()
}

m4_lags <- m4_extended %>%
    lag_transformer_grouped()

train_data <- m4_lags %>%
    drop_na()

future_data <- m4_lags %>%
    filter(is.na(value))

# Modeling Autoregressive Panel Data
model_fit_lm_recursive <- linear_reg() %>%
    set_engine("lm") %>%
    fit(value ~ ., data = train_data) %>%
    recursive(
        id         = "id", # We add an id = "id" to specify the groups
        transform  = lag_transformer_grouped,
        # We use panel_tail() to grab tail by groups
        train_tail = panel_tail(train_data, id, FORECAST_HORIZON)
    )

modeltime_table(
    model_fit_lm_recursive
) %>%
    modeltime_forecast(
        new_data    = future_data,
        actual_data = m4_monthly,
        keep_data   = TRUE
    ) %>%
    group_by(id) %>%
    plot_modeltime_forecast(
        .interactive = FALSE,
        .conf_interval_show = FALSE
    )
```

`seasonal_reg` *General Interface for Multiple Seasonality Regression Models (TBATS, STLM)*

Description

`seasonal_reg()` is a way to generate a *specification* of an Seasonal Decomposition model before fitting and allows the model to be created using different packages. Currently the only package is `forecast`.

Usage

```
seasonal_reg(
  mode = "regression",
  seasonal_period_1 = NULL,
  seasonal_period_2 = NULL,
  seasonal_period_3 = NULL
)
```

Arguments

`mode` A single character string for the type of model. The only possible value for this model is "regression".

`seasonal_period_1` (required) The primary seasonal frequency. Uses `"auto"` by default. A character phrase of "auto" or time-based phrase of "2 weeks" can be used if a date or date-time variable is provided. See Fit Details below.

`seasonal_period_2` (optional) A second seasonal frequency. Is `NULL` by default. A character phrase of "auto" or time-based phrase of "2 weeks" can be used if a date or date-time variable is provided. See Fit Details below.

`seasonal_period_3` (optional) A third seasonal frequency. Is `NULL` by default. A character phrase of "auto" or time-based phrase of "2 weeks" can be used if a date or date-time variable is provided. See Fit Details below.

Details

The data given to the function are not saved and are only used to determine the *mode* of the model. For `seasonal_reg()`, the mode will always be "regression".

The model can be created using the `fit()` function using the following *engines*:

- "tbats" - Connects to `forecast::tbats()`
- "stlm_ets" - Connects to `forecast::stlm()`, `method = "ets"`
- "stlm_arima" - Connects to `forecast::stlm()`, `method = "arima"`

Engine Details

The standardized parameter names in `modeltime` can be mapped to their original names in each engine:

modeltime	forecast::stlm	forecast::tbats
seasonal_period_1, seasonal_period_2, seasonal_period_3	msts(seasonal.periods)	msts(seasonal.periods)

Other options can be set using `set_engine()`.

The engines use `forecast::stlm()`.

Function Parameters:

```
#> function (y, s.window = 7 + 4 * seq(6), robust = FALSE, method = c("ets",
#>   "arima"), modelfunction = NULL, model = NULL, etsmodel = "ZZN", lambda = NULL,
#>     biasadj = FALSE, xreg = NULL, allow.multiplicative.trend = FALSE, x = y,
#>     ...)
```

tbats

- **Method:** Uses `method = "tbats"`, which by default is auto-TBATS.
- **Xregs:** Univariate. Cannot accept Exogenous Regressors (xregs). Xregs are ignored.

stlm_ets

- **Method:** Uses `method = "stlm_ets"`, which by default is auto-ETS.
- **Xregs:** Univariate. Cannot accept Exogenous Regressors (xregs). Xregs are ignored.

stlm_arima

- **Method:** Uses `method = "stlm_arima"`, which by default is auto-ARIMA.
- **Xregs:** Multivariate. Can accept Exogenous Regressors (xregs).

Fit Details

Date and Date-Time Variable

It's a requirement to have a date or date-time variable as a predictor. The `fit()` interface accepts date and date-time features and handles them internally.

- `fit(y ~ date)`

Seasonal Period Specification

The period can be non-seasonal (`seasonal_period = 1 or "none"`) or yearly seasonal (e.g. For monthly time stamps, `seasonal_period = 12`, `seasonal_period = "12 months"`, or `seasonal_period = "yearly"`). There are 3 ways to specify:

1. `seasonal_period = "auto"`: A seasonal period is selected based on the periodicity of the data (e.g. 12 if monthly)
2. `seasonal_period = 12`: A numeric frequency. For example, 12 is common for monthly data
3. `seasonal_period = "1 year"`: A time-based phrase. For example, "1 year" would convert to 12 for monthly data.

Univariate (No xregs, Exogenous Regressors):

For univariate analysis, you must include a date or date-time feature. Simply use:

- Formula Interface (recommended): `fit(y ~ date)` will ignore xreg's.
- XY Interface: `fit_xy(x = data[,"date"], y = data$y)` will ignore xreg's.

Multivariate (xregs, Exogenous Regressors)

- The `tbats` engine *cannot* accept Xregs.
- The `stlm_ets` engine *cannot* accept Xregs.

- The `stlm_arima` engine *can* accept Xregs

The `xreg` parameter is populated using the `fit()` or `fit_xy()` function:

- Only `factor`, `ordered factor`, and `numeric` data will be used as xregs.
- Date and Date-time variables are not used as xregs
- `character` data should be converted to factor.

Xreg Example: Suppose you have 3 features:

1. `y` (target)
2. `date` (time stamp),
3. `month.lbl` (labeled month as a ordered factor).

The `month.lbl` is an exogenous regressor that can be passed to the `seasonal_reg()` using `fit()`:

- `fit(y ~ date + month.lbl)` will pass `month.lbl` on as an exogenous regressor.
- `fit_xy(data[,c("date", "month.lbl")], y = data$y)` will pass x, where x is a data frame containing `month.lbl` and the `date` feature. Only `month.lbl` will be used as an exogenous regressor.

Note that date or date-time class values are excluded from `xreg`.

See Also

`fit.model_spec()`, `set_engine()`

Examples

```
library(dplyr)
library(parsnip)
library(rsample)
library(timetk)

# Data
taylor_30_min

# Split Data 80/20
splits <- initial_time_split(taylor_30_min, prop = 0.8)

# ---- STLM ETS ----

# Model Spec
model_spec <- seasonal_reg() %>%
    set_engine("stlm_ets")

# Fit Spec
model_fit <- model_spec %>%
    fit(log(value) ~ date, data = training(splits))
model_fit
```

```
# ---- STLM ARIMA ----

# Model Spec
model_spec <- seasonal_reg() %>%
    set_engine("stlm_arima")

# Fit Spec
model_fit <- model_spec %>%
    fit(log(value) ~ date, data = training(splits))
model_fit
```

---

summarize_accuracy_metrics   *Summarize Accuracy Metrics*

Description

This is an internal function used by `modeltime_accuracy()`.

Usage

```
summarize_accuracy_metrics(data, truth, estimate, metric_set)
```

Arguments

`data`	A `data.frame` containing the truth and estimate columns.
`truth`	The column identifier for the true results (numeric).
`estimate`	The column identifier for the predicted results (numeric).
`metric_set`	A `yardstick::metric_set` object specifying the metrics to compute.

Value

A tibble with columns `.metric`, `.estimator`, and `.estimate`, pivoted wider by metric and grouped by any grouping variables in the input data.

Examples

```
library(dplyr)

predictions_tbl <- tibble(
  group = c("model 1", "model 1", "model 1",
            "model 2", "model 2", "model 2"),
  truth = c(1, 2, 3,
            1, 2, 3),
  estimate = c(1.2, 2.0, 2.5,
               0.9, 1.9, 3.3)
)
```

```
predictions_tbl %>%
  dplyr::group_by(group) %>%
  summarize_accuracy_metrics(
    truth, estimate,
    metric_set = default_forecast_accuracy_metric_set()
  )
```

---

table_modeltime_accuracy
Interactive Accuracy Tables

Description

Converts results from `modeltime_accuracy()` into either interactive (`reactable`) or static (`gt`) tables.

Usage

```
table_modeltime_accuracy(
  .data,
  .round_digits = 2,
  .sortable = TRUE,
  .show_sortable = TRUE,
  .searchable = TRUE,
  .filterable = FALSE,
  .expand_groups = TRUE,
  .title = "Accuracy Table",
  .interactive = TRUE,
  ...
)
```

Arguments

`.data`	A `tibble` that is the output of `modeltime_accuracy()`
`.round_digits`	Rounds accuracy metrics to a specified number of digits. If `NULL`, rounding is not performed.
`.sortable`	Allows sorting by columns. Only applied to `reactable` tables. Passed to `reactable(sortable)`.
`.show_sortable`	Shows sorting. Only applied to `reactable` tables. Passed to `reactable(showSortable)`.
`.searchable`	Adds search input. Only applied to `reactable` tables. Passed to `reactable(searchable)`.
`.filterable`	Adds filters to table columns. Only applied to `reactable` tables. Passed to `reactable(filterable)`.
`.expand_groups`	Expands groups dropdowns. Only applied to `reactable` tables. Passed to `reactable(defaultExpanded)`.

.title	A title for static (gt) tables.
.interactive	Return interactive or static tables. If TRUE, returns reactable table. If FALSE, returns static gt table.
...	Additional arguments passed to reactable::reactable() or gt::gt() (depending on .interactive selection).

Details

Groups

The function respects dplyr::group_by() groups and thus scales with multiple groups.

Reactable Output

A reactable() table is an interactive format that enables live searching and sorting. When .interactive = TRUE, a call is made to reactable::reactable().

table_modeltime_accuracy() includes several common options like toggles for sorting and searching. Additional arguments can be passed to reactable::reactable() via ....

GT Output

A gt table is an HTML-based table that is "static" (e.g. non-searchable, non-sortable). It's commonly used in PDF and Word documents that does not support interactive content.

When .interactive = FALSE, a call is made to gt::gt(). Arguments can be passed via ....

Table customization is implemented using a piping workflow (%>%). For more information, refer to the GT Documentation.

Value

A static gt table or an interactive reactable table containing the accuracy information.

Examples

```
library(dplyr)
library(lubridate)
library(timetk)
library(parsnip)
library(rsample)

# Data
m750 <- m4_monthly %>% filter(id == "M750")

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.9)

# --- MODELS ---

# Model 1: prophet ----
model_fit_prophet <- prophet_reg() %>%
    set_engine(engine = "prophet") %>%
    fit(value ~ date, data = training(splits))
```

```
# ---- MODELTIME TABLE ----

models_tbl <- modeltime_table(
    model_fit_prophet
)

# ---- ACCURACY ----

models_tbl %>%
    modeltime_calibrate(new_data = testing(splits)) %>%
    modeltime_accuracy() %>%
    table_modeltime_accuracy()
```

---

temporal_hierarchy   *General Interface for Temporal Hierarchical Forecasting (THIEF) Models*

Description

`temporal_hierarchy()` is a way to generate a *specification* of an Temporal Hierarchical Forecasting model before fitting and allows the model to be created using different packages. Currently the only package is `thief`. Note this function requires the `thief` package to be installed.

Usage

```
temporal_hierarchy(
  mode = "regression",
  seasonal_period = NULL,
  combination_method = NULL,
  use_model = NULL
)
```

Arguments

`mode` A single character string for the type of model. The only possible value for this model is "regression".

`seasonal_period` A seasonal frequency. Uses "auto" by default. A character phrase of "auto" or time-based phrase of "2 weeks" can be used if a date or date-time variable is provided. See Fit Details below.

`combination_method` Combination method of temporal hierarchies, taking one of the following values:

- "struc" - Structural scaling: weights from temporal hierarchy
- "mse" - Variance scaling: weights from in-sample MSE

- "ols" - Unscaled OLS combination weights
- "bu" - Bottom-up combination – i.e., all aggregate forecasts are ignored.
- "shr" - GLS using a shrinkage (to block diagonal) estimate of residuals
- "sam" - GLS using sample covariance matrix of residuals

`use_model` Model used for forecasting each aggregation level:

- "ets" - exponential smoothing
- "arima" - arima
- "theta" - theta
- "naive" - random walk forecasts
- "snaive" - seasonal naive forecasts, based on the last year of observed data

Details

Models can be created using the following *engines*:

- "thief" (default) - Connects to `thief::thief()`

Engine Details

The standardized parameter names in `modeltime` can be mapped to their original names in each engine:

modeltime	thief::thief()
combination_method	comb
use_model	usemodel

Other options can be set using `set_engine()`.

thief (default engine)

The engine uses `thief::thief()`.

Function Parameters:

```
#> function (y, m = frequency(y), h = m * 2, comb = c("struc", "mse", "ols",
#>     "bu", "shr", "sam"), usemodel = c("ets", "arima", "theta", "naive",
#>     "snaive"), forecastfunction = NULL, aggregatelist = NULL, ...)
```

Other options and argument can be set using `set_engine()`.

Parameter Notes:

- `xreg` - This model is not set up to use exogenous regressors. Only univariate models will be fit.

Fit Details

Date and Date-Time Variable

It's a requirement to have a date or date-time variable as a predictor. The `fit()` interface accepts date and date-time features and handles them internally.

- `fit(y ~ date)`

Univariate:

For univariate analysis, you must include a date or date-time feature. Simply use:

- Formula Interface (recommended): `fit(y ~ date)` will ignore xreg's.
- XY Interface: `fit_xy(x = data[,"date"], y = data$y)` will ignore xreg's.

Multivariate (xregs, Exogenous Regressors)

This model is not set up for use with exogenous regressors.

References

- For forecasting with temporal hierarchies see: Athanasopoulos G., Hyndman R.J., Kourentzes N., Petropoulos F. (2017) Forecasting with Temporal Hierarchies. *European Journal of Operational research*, **262**(**1**), 60-74.
- For combination operators see: Kourentzes N., Barrow B.K., Crone S.F. (2014) Neural network ensemble operators for time series forecasting. *Expert Systems with Applications*, **41**(**9**), 4235-4244.

See Also

`fit.model_spec()`, `set_engine()`

Examples

```
library(dplyr)
library(parsnip)
library(rsample)
library(timetk)
library(thief)

# Data
m750 <- m4_monthly %>% filter(id == "M750")
m750

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.8)

# ---- HIERARCHICAL ----

# Model Spec - The default parameters are all set
# to "auto" if none are provided
model_spec <- temporal_hierarchy() %>%
    set_engine("thief")
```

```
# Fit Spec
model_fit <- model_spec %>%
    fit(log(value) ~ date, data = training(splits))
model_fit
```

`temporal_hierarchy_params` *Tuning Parameters for TEMPORAL HIERARCHICAL Models*

Description

Tuning Parameters for TEMPORAL HIERARCHICAL Models

Usage

```
combination_method(values = c("struc", "mse", "ols", "bu", "shr", "sam"))

use_model()
```

Arguments

`values`	A character string of possible values.

Details

The main parameters for Temporal Hierarchical models are:

- `combination_method`: Combination method of temporal hierarchies.
- `use_model`: Model used for forecasting each aggregation level.

Examples

```
combination_method()

use_model()
```

time_series_params — *Tuning Parameters for Time Series (ts-class) Models*

Description

Tuning Parameters for Time Series (ts-class) Models

Usage

```
seasonal_period(values = c("none", "daily", "weekly", "yearly"))
```

Arguments

`values`	A time-based phrase

Details

Time series models (e.g. `Arima()` and `ets()`) use `stats::ts()` or `forecast::msts()` to apply seasonality. We can do the same process using the following general time series parameter:

- `period`: The periodic nature of the seasonality.

It's usually best practice to *not* tune this parameter, but rather set to obvious values based on the seasonality of the data:

- **Daily Seasonality:** Often used with **hourly data** (e.g. 24 hourly timestamps per day)
- **Weekly Seasonality:** Often used with **daily data** (e.g. 7 daily timestamps per week)
- **Yearly Seasonalty:** Often used with **weekly, monthly, and quarterly data** (e.g. 12 monthly observations per year).

However, in the event that users want to experiment with period tuning, you can do so with `seasonal_period()`.

Examples

```
seasonal_period()
```

`update_modeltime_model` *Update the model by model id in a Modeltime Table*

Description

Update the model by model id in a Modeltime Table

Usage

```
update_modeltime_model(object, .model_id, .new_model)
```

Arguments

`object`	A Modeltime Table
`.model_id`	A numeric value matching the .model_id that you want to update
`.new_model`	A fitted workflow, model_fit, or mdl_time_ensmble object

See Also

- `combine_modeltime_tables()`: Combine 2 or more Modeltime Tables together
- `add_modeltime_model()`: Adds a new row with a new model to a Modeltime Table
- `drop_modeltime_model()`: Drop one or more models from a Modeltime Table
- `update_modeltime_description()`: Updates a description for a model inside a Modeltime Table
- `update_modeltime_model()`: Updates a model inside a Modeltime Table
- `pull_modeltime_model()`: Extracts a model from a Modeltime Table

Examples

```
library(tidymodels)

model_fit_ets <- exp_smoothing() %>%
    set_engine("ets") %>%
    fit(value ~ date, training(m750_splits))

m750_models %>%
    update_modeltime_model(1, model_fit_ets)
```

update_model_description *Update the model description by model id in a Modeltime Table*

Description

The `update_model_description()` and `update_modeltime_description()` functions are synonyms.

Usage

```
update_model_description(object, .model_id, .new_model_desc)

update_modeltime_description(object, .model_id, .new_model_desc)
```

Arguments

`object`	A Modeltime Table
`.model_id`	A numeric value matching the .model_id that you want to update
`.new_model_desc`	Text describing the new model description

See Also

- `combine_modeltime_tables()`: Combine 2 or more Modeltime Tables together
- `add_modeltime_model()`: Adds a new row with a new model to a Modeltime Table
- `drop_modeltime_model()`: Drop one or more models from a Modeltime Table
- `update_modeltime_description()`: Updates a description for a model inside a Modeltime Table
- `update_modeltime_model()`: Updates a model inside a Modeltime Table
- `pull_modeltime_model()`: Extracts a model from a Modeltime Table

Examples

```
m750_models %>%
    update_modeltime_description(2, "PROPHET - No Regressors")
```

window_reg *General Interface for Window Forecast Models*

Description

window_reg() is a way to generate a *specification* of a window model before fitting and allows the model to be created using different backends.

Usage

```
window_reg(mode = "regression", id = NULL, window_size = NULL)
```

Arguments

mode	A single character string for the type of model. The only possible value for this model is "regression".
id	An optional quoted column name (e.g. "id") for identifying multiple time series (i.e. panel data).
window_size	A window to apply the window function. By default, the window uses the full data set, which is rarely the best choice.

Details

A time series window regression is derived using window_reg(). The model can be created using the fit() function using the following *engines*:

- **"window_function" (default)** - Performs a Window Forecast applying a window_function (engine parameter) to a window of size defined by window_size

Engine Details

function (default engine)

The engine uses window_function_fit_impl(). A time series window function applies a window_function to a window of the data (last N observations).

- The function can return a scalar (single value) or multiple values that are repeated for each window
- Common use cases:
  - **Moving Average Forecasts:** Forecast forward a 20-day average
  - **Weighted Average Forecasts:** Exponentially weighting the most recent observations
  - **Median Forecasts:** Forecasting forward a 20-day median
  - **Repeating Forecasts:** Simulating a Seasonal Naive Forecast by broadcasting the last 12 observations of a monthly dataset into the future

The key engine parameter is the window_function. A function / formula:

- If a function, e.g. mean, the function is used with any additional arguments, ... in set_engine().

- If a formula, e.g. `~ mean(., na.rm = TRUE)`, it is converted to a function.

This syntax allows you to create very compact anonymous functions.

Fit Details

Date and Date-Time Variable

It's a requirement to have a date or date-time variable as a predictor. The `fit()` interface accepts date and date-time features and handles them internally.

- `fit(y ~ date)`

ID features (Multiple Time Series, Panel Data)

The `id` parameter is populated using the `fit()` or `fit_xy()` function:

ID Example: Suppose you have 3 features:

1. `y` (target)
2. `date` (time stamp),
3. `series_id` (a unique identifer that identifies each time series in your data).

The `series_id` can be passed to the `window_reg()` using `fit()`:

- `window_reg(id = "series_id")` specifes that the `series_id` column should be used to identify each time series.
- `fit(y ~ date + series_id)` will pass `series_id` on to the underlying functions.

Window Function Specification (window_function)

You can specify a function / formula using `purrr` syntax.

- If a function, e.g. `mean`, the function is used with any additional arguments, `...` in `set_engine()`.
- If a formula, e.g. `~ mean(., na.rm = TRUE)`, it is converted to a function.

This syntax allows you to create very compact anonymous functions.

Window Size Specification (window_size)

The period can be non-seasonal (`window_size = 1 or "none"`) or yearly seasonal (e.g. For monthly time stamps, `window_size = 12`, `window_size = "12 months"`, or `window_size = "yearly"`). There are 3 ways to specify:

1. `window_size = "all"`: A seasonal period is selected based on the periodicity of the data (e.g. 12 if monthly)
2. `window_size = 12`: A numeric frequency. For example, 12 is common for monthly data
3. `window_size = "1 year"`: A time-based phrase. For example, "1 year" would convert to 12 for monthly data.

External Regressors (Xregs)

These models are univariate. No xregs are used in the modeling process.

See Also

fit.model_spec(), set_engine()

Examples

```
library(dplyr)
library(parsnip)
library(rsample)
library(timetk)

# Data
m750 <- m4_monthly %>% filter(id == "M750")
m750

# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.8)

# ---- WINDOW FUNCTION -----

# Used to make:
# - Mean/Median forecasts
# - Simple repeating forecasts

# Median Forecast ----

# Model Spec
model_spec <- window_reg(
        window_size     = 12
    ) %>%
    # Extra parameters passed as: set_engine(...)
    set_engine(
        engine          = "window_function",
        window_function = median,
        na.rm           = TRUE
    )

# Fit Spec
model_fit <- model_spec %>%
    fit(log(value) ~ date, data = training(splits))
model_fit

# Predict
# - The 12-month median repeats going forward
predict(model_fit, testing(splits))


# ---- PANEL FORECAST - WINDOW FUNCTION ----

# Weighted Average Forecast
model_spec <- window_reg(
        # Specify the ID column for Panel Data
        id          = "id",
```

```
        window_size = 12
    ) %>%
    set_engine(
        engine = "window_function",
        # Create a Weighted Average
        window_function = ~ sum(tail(.x, 3) * c(0.1, 0.3, 0.6)),
    )

# Fit Spec
model_fit <- model_spec %>%
    fit(log(value) ~ date + id, data = training(splits))
model_fit

# Predict: The weighted average (scalar) repeats going forward
predict(model_fit, testing(splits))

# ---- BROADCASTING PANELS (REPEATING) ----

# Simulating a Seasonal Naive Forecast by
# broadcasted model the last 12 observations into the future
model_spec <- window_reg(
        id          = "id",
        window_size = Inf
    ) %>%
    set_engine(
        engine          = "window_function",
        window_function = ~ tail(.x, 12),
    )

# Fit Spec
model_fit <- model_spec %>%
    fit(log(value) ~ date + id, data = training(splits))
model_fit

# Predict: The sequence is broadcasted (repeated) during prediction
predict(model_fit, testing(splits))
```

Index