

Package ‘modeltime’

June 13, 2021

Title The Tidymodels Extension for Time Series Modeling

Version 0.6.1

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.fb.com/blog/2017/02/prophet-forecasting-at-scale/>).

URL <https://github.com/business-science/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 (>= 2.6.0), parsnip (>= 0.1.6), dials, yardstick (>= 0.0.8), workflows (>= 0.1.3), hardhat, rlang (>= 0.1.2), glue, plotly, reactable, gt, ggplot2, tibble, tidyr, dplyr, purrr, stringr, forcats, scales, janitor, parallel, doParallel, foreach, magrittr, forecast, xgboost (>= 1.2.0.1), prophet, methods, cli

Suggests rstan, slider, tidymodels, workflowsets, recipes, rsample, tune, tidyverse, lubridate, progress, testthat, roxygen2, kernlab, earth, randomForest, tidyquant, knitr, rmarkdown, webshot, qpdf, covr

VignetteBuilder knitr

RoxygenNote 7.1.1

NeedsCompilation no

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

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

Repository CRAN

Date/Publication 2021-06-13 12:20:02 UTC

R topics documented:

.prepare_transform	4
add_modeltime_model	4
arima_boost	5
Arima_fit_impl	10
arima_params	12
Arima_predict_impl	13
arima_reg	13
arima_xgboost_fit_impl	18
arima_xgboost_predict_impl	20
auto_arima_fit_impl	20
auto_arima_xgboost_fit_impl	21
combine_modeltime_tables	24
control_fit_workflowset	26
control_refit	27
create_model_grid	28
create_xreg_recipe	29
croston_fit_impl	31
croston_predict_impl	31
default_forecast_accuracy_metric_set	32
ets_fit_impl	33
ets_predict_impl	34
exp_smoothing	35
exp_smoothing_params	39
get_arima_description	40
get_model_description	41
get_tbats_description	42
is_calibrated	42
is_modeltime_model	43
is_modeltime_table	43
is_residuals	43
load_namespace	44
m750	44
m750_models	45
m750_splits	45
m750_training_resamples	46
modeltime_accuracy	47
modeltime_calibrate	48
modeltime_fit_workflowset	50
modeltime_forecast	52
modeltime_refit	56
modeltime_residuals	57
modeltime_residuals_test	59

modeltime_table	61
naive_fit_impl	63
naive_predict_impl	63
naive_reg	64
new_modeltime_bridge	66
nnetar_fit_impl	67
nnetar_params	68
nnetar_predict_impl	69
nnetar_reg	70
panel_tail	73
parallel_start	74
parse_index	75
plot_modeltime_forecast	76
plot_modeltime_residuals	78
pluck_modeltime_model	80
predict.recursive	81
predict.recursive_panel	82
prophet_boost	83
prophet_fit_impl	88
prophet_params	90
prophet_predict_impl	92
prophet_reg	92
prophet_xgboost_fit_impl	97
prophet_xgboost_predict_impl	99
pull_modeltime_residuals	100
pull_parsnip_preprocessor	100
recipe_helpers	101
recursive	102
seasonal_reg	106
snaive_fit_impl	109
snaive_predict_impl	110
stlm_arima_fit_impl	110
stlm_arima_predict_impl	111
stlm_ets_fit_impl	111
stlm_ets_predict_impl	112
summarize_accuracy_metrics	113
table_modeltime_accuracy	114
tbats_fit_impl	116
tbats_predict_impl	117
theta_fit_impl	117
theta_predict_impl	118
time_series_params	118
type_sum.mdl_time_tbl	119
update_modeltime_model	119
update_model_description	120
window_function_fit_impl	121
window_function_predict_impl	122
window_reg	122

xgboost_impl	126
xgboost_predict	127

Index	128
--------------	------------

<code>.prepare_transform</code>	<i>Prepare Recursive Transformations</i>
---------------------------------	--

Description

Prepare Recursive Transformations

Usage

```
.prepare_transform(.transform)
.prepare_panel_transform(.transform)
```

Arguments

`.transform` A transformation function

Value

A function that applies a recursive transformation

<code>add_modeltime_model</code>	<i>Add a Model into a Modeltime Table</i>
----------------------------------	---

Description

Add a Model into a Modeltime Table

Usage

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

Arguments

<code>object</code>	Multiple Modeltime Tables (class <code>mdl_time_tbl</code>)
<code>model</code>	A model of class <code>model_fit</code> or a fitted workflow object
<code>location</code>	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
- `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 (xgboost 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) (xgboost only).
learn_rate	A number for the rate at which the boosting algorithm adapts from iteration-to-iteration (xgboost only).
loss_reduction	A number for the reduction in the loss function required to split further (xgboost 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
<code>seasonal_period</code>	<code>ts(frequency)</code>	<code>ts(frequency)</code>
<code>non_seasonal_ar, non_seasonal_differences, non_seasonal_ma</code>	<code>max.p(5), max.d(2), max.q(5)</code>	<code>order = c(p(0), d(0), q(0))</code>
<code>seasonal_ar, seasonal_differences, seasonal_ma</code>	<code>max.P(2), max.D(1), max.Q(2)</code>	<code>seasonal = c(P(0), D(0), Q(0))</code>

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 = list(), data, nrounds, watchlist = list(), obj = NULL,
##   feval = 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(tidyverse)
library(lubridate)
library(parsnip)
library(rsample)
library(timetk)
library(modeltime)

# 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 ----

## Not run:
# 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

## End(Not run)

```

Description

Low-Level ARIMA function for translating modeltime to forecast

Usage

```
Arima_fit_impl(
  x,
  y,
  period = "auto",
  p = 0,
  d = 0,
  q = 0,
  P = 0,
  D = 0,
  Q = 0,
  ...
)
```

Arguments

x	A dataframe of xreg (exogenous regressors)
y	A numeric vector of values to fit
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.
p	The order of the non-seasonal auto-regressive (AR) terms. Often denoted "p" in pdq-notation.
d	The order of integration for non-seasonal differencing. Often denoted "d" in pdq-notation.
q	The order of the non-seasonal moving average (MA) terms. Often denoted "q" in pdq-notation.
P	The order of the seasonal auto-regressive (SAR) terms. Often denoted "P" in PDQ-notation.
D	The order of integration for seasonal differencing. Often denoted "D" in PDQ-notation.
Q	The order of the seasonal moving average (SMA) terms. Often denoted "Q" in PDQ-notation.
...	Additional arguments passed to <code>forecast::Arima</code>

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 <i>defaults</i> for the smallest and largest possible values, respectively.
trans	A trans object from the scales package, such as <code>scales::log10_trans()</code> or <code>scales::reciprocal_trans()</code> . 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

```
non_seasonal_ar()
```

```
non_seasonal_differences()
```

```
non_seasonal_ma()
```

Arima_predict_impl *Bridge prediction function for ARIMA models*

Description

Bridge prediction function for ARIMA models

Usage

```
Arima_predict_impl(object, new_data, ...)
```

Arguments

object	An object of class <code>model_fit</code>
new_data	A rectangular data object, such as a data frame.
...	Additional arguments passed to <code>forecast::Arima()</code>

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:

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

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("aic", "aicc",
##     "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 Maximum 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)
library(modeltime)

# 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

```

```
arima_xgboost_fit_impl
```

Bridge ARIMA-XGBoost Modeling function

Description

Bridge ARIMA-XGBoost Modeling function

Usage

```

arima_xgboost_fit_impl(
  x,
  y,
  period = "auto",
  p = 0,
  d = 0,
  q = 0,
  P = 0,
  D = 0,
  Q = 0,
  include.mean = TRUE,
  include.drift = FALSE,
  include.constant,
  lambda = model$lambda,
  biasadj = FALSE,
  method = c("CSS-ML", "ML", "CSS"),
  model = NULL,
  max_depth = 6,
  nrounds = 15,
  eta = 0.3,
  colsample_bytree = NULL,
  colsample_bynode = NULL,
  min_child_weight = 1,
  gamma = 0,
  subsample = 1,
  validation = 0,
  early_stop = NULL,
  ...
)

```

Arguments

x	A dataframe of xreg (exogenous regressors)
y	A numeric vector of values to fit
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.
p	The order of the non-seasonal auto-regressive (AR) terms.
d	The order of integration for non-seasonal differencing.
q	The order of the non-seasonal moving average (MA) terms.
P	The order of the seasonal auto-regressive (SAR) terms.
D	The order of integration for seasonal differencing.
Q	The order of the seasonal moving average (SMA) terms.
include.mean	Should the ARIMA model include a mean term? The default is TRUE for undifferenced series, FALSE for differenced ones (where a mean would not affect the fit nor predictions).
include.drift	Should the ARIMA model include a linear drift term? (i.e., a linear regression with ARIMA errors is fitted.) The default is FALSE.
include.constant	If TRUE, then include.mean is set to be TRUE for undifferenced series and include.drift is set to be TRUE for differenced series. Note that if there is more than one difference taken, no constant is included regardless of the value of this argument. This is deliberate as otherwise quadratic and higher order polynomial trends would be induced.
lambda	Box-Cox transformation parameter. If lambda="auto", then a transformation is automatically selected using BoxCox.lambda. The transformation is ignored if NULL. Otherwise, data transformed before model is estimated.
biasadj	Use adjusted back-transformed mean for Box-Cox transformations. If transformed data is used to produce forecasts and fitted values, a regular back transformation will result in median forecasts. If biasadj is TRUE, an adjustment will be made to produce mean forecasts and fitted values.
method	Fitting method: maximum likelihood or minimize conditional sum-of-squares. The default (unless there are missing values) is to use conditional-sum-of-squares to find starting values, then maximum likelihood.
model	Output from a previous call to Arima. If model is passed, this same model is fitted to y without re-estimating any parameters.
max_depth	An integer for the maximum depth of the tree.
nrounds	An integer for the number of boosting iterations.
eta	A numeric value between zero and one to control the learning rate.
colsample_bytree	Subsampling proportion of columns.
colsample_bynode	Subsampling proportion of columns for each node within each tree. See the counts argument below. The default uses all columns.

min_child_weight	A numeric value for the minimum sum of instance weights needed in a child to continue to split.
gamma	A number for the minimum loss reduction required to make a further partition on a leaf node of the tree
subsample	Subsampling proportion of rows.
validation	A positive number. If on [0, 1) the value, validation is a random proportion of data in x and y that are used for performance assessment and potential early stopping. If 1 or greater, it is the <i>number</i> of training set samples use for these purposes.
early_stop	An integer or NULL. If not NULL, it is the number of training iterations without improvement before stopping. If validation is used, performance is base on the validation set; otherwise the training set is used.
...	Additional arguments passed to <code>xgboost::xgb.train</code>

arima_xgboost_predict_impl

Bridge prediction Function for ARIMA-XGBoost Models

Description

Bridge prediction Function for ARIMA-XGBoost Models

Usage

```
arima_xgboost_predict_impl(object, new_data, ...)
```

Arguments

object	An object of class <code>model_fit</code>
new_data	A rectangular data object, such as a data frame.
...	Additional arguments passed to <code>predict.xgb.Booster()</code>

auto_arima_fit_impl *Low-Level ARIMA function for translating modeltime to forecast*

Description

Low-Level ARIMA function for translating modeltime to forecast

Usage

```

auto_arima_fit_impl(
  x,
  y,
  period = "auto",
  max.p = 5,
  max.d = 2,
  max.q = 5,
  max.P = 2,
  max.D = 1,
  max.Q = 2,
  ...
)

```

Arguments

x	A dataframe of xreg (exogenous regressors)
y	A numeric vector of values to fit
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.
max.p	The maximum order of the non-seasonal auto-regressive (AR) terms.
max.d	The maximum order of integration for non-seasonal differencing.
max.q	The maximum order of the non-seasonal moving average (MA) terms.
max.P	The maximum order of the seasonal auto-regressive (SAR) terms.
max.D	The maximum order of integration for seasonal differencing.
max.Q	The maximum order of the seasonal moving average (SMA) terms.
...	Additional arguments passed to forecast::auto.arima

```
auto_arima_xgboost_fit_impl
```

Bridge ARIMA-XGBoost Modeling function

Description

Bridge ARIMA-XGBoost Modeling function

Usage

```

auto_arima_xgboost_fit_impl(
  x,
  y,
  period = "auto",
  max.p = 5,

```

```

max.d = 2,
max.q = 5,
max.P = 2,
max.D = 1,
max.Q = 2,
max.order = 5,
d = NA,
D = NA,
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,
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,
max_depth = 6,
nrounds = 15,
eta = 0.3,
colsample_bytree = NULL,
colsample_bynode = NULL,
min_child_weight = 1,
gamma = 0,
subsample = 1,
validation = 0,
early_stop = NULL,
...
)

```

Arguments

x	A dataframe of xreg (exogenous regressors)
y	A numeric vector of values to fit
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.
max.p	The maximum order of the non-seasonal auto-regressive (AR) terms.
max.d	The maximum order of integration for non-seasonal differencing.
max.q	The maximum order of the non-seasonal moving average (MA) terms.
max.P	The maximum order of the seasonal auto-regressive (SAR) terms.
max.D	The maximum order of integration for seasonal differencing.
max.Q	The maximum order of the seasonal moving average (SMA) terms.
max.order	Maximum value of $p+q+P+Q$ if model selection is not stepwise.
d	Order of first-differencing. If missing, will choose a value based on <code>test</code> .
D	Order of seasonal-differencing. If missing, will choose a value based on <code>season.test</code> .
start.p	Starting value of p in stepwise procedure.
start.q	Starting value of q in stepwise procedure.
start.P	Starting value of P in stepwise procedure.
start.Q	Starting value of Q in stepwise procedure.
stationary	If TRUE, restricts search to stationary models.
seasonal	If FALSE, restricts search to non-seasonal models.
ic	Information criterion to be used in model selection.
stepwise	If TRUE, will do stepwise selection (faster). Otherwise, it searches over all models. Non-stepwise selection can be very slow, especially for seasonal models.
nmodels	Maximum number of models considered in the stepwise search.
trace	If TRUE, the list of ARIMA models considered will be reported.
approximation	If TRUE, estimation is via conditional sums of squares and the information criteria used for model selection are approximated. The final model is still computed using maximum likelihood estimation. Approximation should be used for long time series or a high seasonal period to avoid excessive computation times.
method	fitting method: maximum likelihood or minimize conditional sum-of-squares. The default (unless there are missing values) is to use conditional-sum-of-squares to find starting values, then maximum likelihood. Can be abbreviated.
truncate	An integer value indicating how many observations to use in model selection. The last <code>truncate</code> values of the series are used to select a model when <code>truncate</code> is not NULL and <code>approximation=TRUE</code> . All observations are used if either <code>truncate=NULL</code> or <code>approximation=FALSE</code> .
test	Type of unit root test to use. See ndiffs for details.
test.args	Additional arguments to be passed to the unit root test.
seasonal.test	This determines which method is used to select the number of seasonal differences. The default method is to use a measure of seasonal strength computed from an STL decomposition. Other possibilities involve seasonal unit root tests.
seasonal.test.args	Additional arguments to be passed to the seasonal unit root test. See nsdiffs for details.

allowdrift	If TRUE, models with drift terms are considered.
allowmean	If TRUE, models with a non-zero mean are considered.
lambda	Box-Cox transformation parameter. If lambda="auto", then a transformation is automatically selected using BoxCox.lambda. The transformation is ignored if NULL. Otherwise, data transformed before model is estimated.
biasadj	Use adjusted back-transformed mean for Box-Cox transformations. If transformed data is used to produce forecasts and fitted values, a regular back transformation will result in median forecasts. If biasadj is TRUE, an adjustment will be made to produce mean forecasts and fitted values.
max_depth	An integer for the maximum depth of the tree.
nrounds	An integer for the number of boosting iterations.
eta	A numeric value between zero and one to control the learning rate.
colsample_bytree	Subsampling proportion of columns.
colsample_bynode	Subsampling proportion of columns for each node within each tree. See the counts argument below. The default uses all columns.
min_child_weight	A numeric value for the minimum sum of instance weights needed in a child to continue to split.
gamma	A number for the minimum loss reduction required to make a further partition on a leaf node of the tree
subsample	Subsampling proportion of rows.
validation	A positive number. If on [0, 1) the value, validation is a random proportion of data in x and y that are used for performance assessment and potential early stopping. If 1 or greater, it is the <i>number</i> of training set samples use for these purposes.
early_stop	An integer or NULL. If not NULL, it is the number of training iterations without improvement before stopping. If validation is used, performance is base on the validation set; otherwise the training set is used.
...	Additional arguments passed to xgboost::xgb.train

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
- `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(modeltime)
library(tidymodels)
library(tidyverse)
library(timetk)
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_fit_workflowset

Control aspects of the modeltime_fit_workflowset() process.

Description

Control aspects of the modeltime_fit_workflowset() process.

Usage

```
control_fit_workflowset(
  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. <ul style="list-style-type: none">• 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

[modeltime_fit_workflowset\(\)](#)

Examples

```
#' # No parallel processing
control_fit_workflowset()

# With parallel processing
control_fit_workflowset(allow_par = TRUE)
```

control_refit *Control aspects of the modeltime_refit() process.*

Description

Control aspects of the modeltime_refit() process.

Usage

```
control_refit(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. <ul style="list-style-type: none"> • 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

[modeltime_refit\(\)](#)

Examples

```
# No parallel processing
control_refit()

# With parallel processing
control_refit(allow_par = TRUE)
```

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 <code>parsnip::set_engine()</code> .
...	Static parameters that get passed to the <code>f_model_spec</code>
engine_params	A list of additional parameters that can be passed to the engine via <code>parsnip::set_engine(...)</code> .

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 (hyperparameter 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 column 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)
library(modeltime)

# 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

<code>data</code>	A data frame
<code>prepare</code>	Whether or not to run <code>recipes::prep()</code> on the final recipe. Default is to prepare. User can set this to <code>FALSE</code> to return an un prepared recipe.

<code>clean_names</code>	Uses <code>janitor::clean_names()</code> to process the names and improve robustness to failure during dummy (one-hot) encoding step.
<code>dummy_encode</code>	Should factors (categorical data) be
<code>one_hot</code>	If <code>dummy_encode = TRUE</code> , should the encoding return one column for each feature or one less column than each feature. Default is <code>FALSE</code> .

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)
```

croston_fit_impl	<i>Low-Level Exponential Smoothing function for translating modeltime to forecast</i>
------------------	---

Description

Low-Level Exponential Smoothing function for translating modeltime to forecast

Usage

```
croston_fit_impl(x, y, alpha = 0.1, ...)
```

Arguments

x	A dataframe of xreg (exogenous regressors)
y	A numeric vector of values to fit
alpha	Value of alpha. Default value is 0.1.
...	Additional arguments passed to forecast::ets

croston_predict_impl	<i>Bridge prediction function for CROSTON models</i>
----------------------	--

Description

Bridge prediction function for CROSTON models

Usage

```
croston_predict_impl(object, new_data, ...)
```

Arguments

object	An object of class model_fit
new_data	A rectangular data object, such as a data frame.
...	Additional arguments passed to stats::predict()

```
default_forecast_accuracy_metric_set
      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(...)
```

Arguments

```
...          Add additional yardstick metrics
```

Details

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

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)
```

ets_fit_impl

*Low-Level Exponential Smoothing function for translating modeltime
to forecast*

Description

Low-Level Exponential Smoothing function for translating modeltime to forecast

Usage

```
ets_fit_impl(  
  x,  
  y,  
  period = "auto",  
  error = "auto",  
  trend = "auto",  
  season = "auto",  
  damping = "auto",  
  alpha = NULL,  
  beta = NULL,  
  gamma = NULL,  
  ...  
)
```

Arguments

x	A dataframe of xreg (exogenous regressors)
y	A numeric vector of values to fit
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.
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".
alpha	Value of alpha. If NULL, it is estimated.
beta	Value of beta. If NULL, it is estimated.
gamma	Value of gamma. If NULL, it is estimated.
...	Additional arguments passed to forecast::ets

 ets_predict_impl

Bridge prediction function for Exponential Smoothing models

Description

Bridge prediction function for Exponential Smoothing models

Usage

```
ets_predict_impl(object, new_data, ...)
```

Arguments

object	An object of class model_fit
new_data	A rectangular data object, such as a data frame.
...	Additional arguments passed to forecast::ets()

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 for intermittent demand

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()`

Engine Details

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

<code>modeltime</code>	<code>forecast::ets</code>	<code>forecast::croston()</code>
<code>seasonal_period()</code>	<code>ts(frequency)</code>	<code>ts(frequency)</code>
<code>error(), trend(), season()</code>	<code>model('ZZZ')</code>	NA
<code>damping()</code>	<code>damped(NULL)</code>	NA
<code>smooth_level()</code>	<code>alpha(NULL)</code>	<code>alpha(0.1)</code>
<code>smooth_trend()</code>	<code>beta(NULL)</code>	NA
<code>smooth_seasonal()</code>	<code>gamma(NULL)</code>	NA

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 = FALSE,
##   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.

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)

This model is not set up for use with exogenous regressors.

See Also

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

Examples

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

# 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

```

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"))

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

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

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 <i>defaults</i> for the smallest and largest possible values, respectively.
trans	A trans object from the scales package, such as scales::log10_trans() or scales::reciprocal_trans(). 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_arma_description *Get model descriptions for Arima objects*

Description

Get model descriptions for Arima objects

Usage

```
get_arma_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)
library(modeltime)

# Model Specification ----

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

```

get_model_description(arma_spec, indicate_training = TRUE)

# Fitted Model ----

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

arma_fit <- arma_spec %>%
  fit(value ~ date, data = m750)

get_model_description(arma_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()`

`is_calibrated` *Test if a Modeltime Table has been calibrated*

Description

This function returns TRUE for objects that contains columns ".type" and ".calibration_data"

Usage

```
is_calibrated(object)
```

Arguments

`object` An object to detect if is a Calibrated Modeltime Table

is_modeltime_model	<i>Test if object contains a fitted modeltime model</i>
--------------------	---

Description

This function returns TRUE for trained workflows and parsnip objects that contain modeltime models

Usage

```
is_modeltime_model(object)
```

Arguments

object	An object to detect if contains a fitted modeltime model
--------	--

is_modeltime_table	<i>Test if object is a Modeltime Table</i>
--------------------	--

Description

This function returns TRUE for objects that contain class mdl_time_tbl

Usage

```
is_modeltime_table(object)
```

Arguments

object	An object to detect if is a Modeltime Table
--------	---

is_residuals	<i>Test if a table contains residuals.</i>
--------------	--

Description

This function returns TRUE for objects that contains the column name '.residuals'.

Usage

```
is_residuals(object)
```

Arguments

object	An object to detect if it provides from modeltime::modeltime_residuals().
--------	---

load_namespace	<i>These are not intended for use by the general public.</i>
----------------	--

Description

These are not intended for use by the general public.

Usage

```
load_namespace(x, full_load)
```

Arguments

x	A vector
full_load	A vector

Value

Control information

m750	<i>The 750th Monthly Time Series used in the M4 Competition</i>
------	---

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](#)

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

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

Examples

```
library(modeltime)

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
```

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(),
  quiet = TRUE,
  ...
)
```

Arguments

<code>object</code>	A Modeltime Table
<code>new_data</code>	A tibble to predict and calculate residuals on. If provided, overrides any calibration data.
<code>metric_set</code>	A <code>yardstick::metric_set()</code> that is used to summarize one or more forecast accuracy (regression) metrics.
<code>quiet</code>	Hide errors (TRUE, the default), or display them as they occur?
<code>...</code>	Not currently used

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(tidyverse)
library(lubridate)
library(timetk)

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

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

# --- MODELS ---

# Model 1: auto_arima ----
model_fit_arima <- arima_reg() %>%
  set_engine(engine = "auto_arima") %>%
  fit(value ~ date, data = training(splits))

# ---- MODELTIME TABLE ----

models_tbl <- modeltime_table(
  model_fit_arima
)

# ---- 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, quiet = TRUE, ...)
```

Arguments

object	A fitted model object that is either: <ol style="list-style-type: none"> 1. A modeltime table that has been created using <code>modeltime_table()</code> 2. A workflow that has been fit by <code>fit.workflow()</code> or 3. A parsnip model that has been fit using <code>fit.model_spec()</code>
new_data	A test data set tibble containing future information (timestamps and actual values).
quiet	Hide errors (TRUE, the default), or display them as they occur?
...	Additional arguments passed to <code>modeltime_forecast()</code> .

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. Only "Test" is currently available.
 - `.calibration_data`: Contains a tibble with Timestamps, Actual Values, Predictions and Residuals calculated from `new_data` (Test Data)

Value

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

Examples

```
library(tidyverse)
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: auto_arima ----
model_fit_arima <- arima_reg() %>%
  set_engine(engine = "auto_arima") %>%
  fit(value ~ date, data = training(splits))

# ---- MODELTIME TABLE ----

models_tbl <- modeltime_table(
  model_fit_arima
)

# ---- 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(modeltime)
library(workflowsets)
library(tidyverse)
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,
  keep_data = FALSE,
  arrange_index = FALSE,
  ...
)
```

Arguments

<code>object</code>	A Modeltime Table
<code>new_data</code>	A tibble containing future information to forecast. If NULL, forecasts the calibration data.
<code>h</code>	The forecast horizon (can be used instead of <code>new_data</code> for time series with no exogenous regressors). Extends the calibration data <code>h</code> periods into the future.
<code>actual_data</code>	Reference data that is combined with the output tibble and given a <code>.key = "actual"</code>
<code>conf_interval</code>	An estimated confidence interval based on the calibration data. This is designed to estimate future confidence from <i>out-of-sample prediction error</i> .
<code>keep_data</code>	Whether or not to keep the <code>new_data</code> and <code>actual_data</code> as extra columns in the results. This can be useful if there is an important feature in the <code>new_data</code> and <code>actual_data</code> needed when forecasting. Default: FALSE.
<code>arrange_index</code>	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.
<code>...</code>	Not currently used

Details

The `modeltime_forecast()` function prepares a forecast for visualization 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 exogenous 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 any future forecasts.

The confidence interval can be adjusted with the `conf_interval` parameter. An 80% confidence interval estimates a normal (Gaussian distribution) that assumes that 80% of the future data will fall within the upper and lower confidence limits.

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.

Refitting has no effect on the confidence interval since this is calculated independently of the refitted model (on data with a smaller sample size). 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`

Examples

```
library(tidyverse)
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: auto_arima ----
model_fit_arima <- arima_reg() %>%
  set_engine(engine = "auto_arima") %>%
  fit(value ~ date, data = training(splits))

# ---- MODELTIME TABLE ----

models_tbl <- modeltime_table(
  model_fit_arima
)

# ---- 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_refit	<i>Refit one or more trained models to new data</i>
-----------------	---

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 (<code>modeltime.ensemble</code>):
	When making a meta-learner with <code>modeltime.ensemble::ensemble_model_spec()</code> , used to pass <code>resamples</code> argument containing results from <code>modeltime.resample::modeltime_fit_resa</code>
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(tidyverse)
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_auto_arima <- arima_reg() %>%
  set_engine(engine = "auto_arima") %>%
  fit(value ~ date, data = training(splits))

# ---- MODELTIME TABLE ----

models_tbl <- modeltime_table(
  model_fit_auto_arima
)

# ---- 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(tidyverse)
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: auto_arima ----
model_fit_arima <- arima_reg() %>%
  set_engine(engine = "auto_arima") %>%
  fit(value ~ date, data = training(splits))

# ---- MODELTIME TABLE ----

models_tbl <- modeltime_table(
  model_fit_arima
)

# ---- 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 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 absence 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(tidyverse)
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: auto_arima ----
model_fit_arima <- arima_reg() %>%
  set_engine(engine = "auto_arima") %>%
  fit(value ~ date, data = training(splits))

# ---- MODELTIME TABLE ----

models_tbl <- modeltime_table(
  model_fit_arima
)

# ---- 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	<i>Scale forecast analysis with a Modeltime Table</i>
-----------------	---

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 <code>parsnip</code> model or <code>workflow</code> objects
.l	A list containing fitted <code>parsnip</code> model or <code>workflow</code> 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(tidyverse)
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: auto_arima ----
model_fit_arima <- arima_reg() %>%
  set_engine(engine = "auto_arima") %>%
  fit(value ~ date, data = training(splits))

# ---- MODELTIME TABLE ----

# Make a Modeltime Table
models_tbl <- modeltime_table(
  model_fit_arima
)

# Can also convert a list of models
list(model_fit_arima) %>%
  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_fit_impl	<i>Low-Level NAIVE Forecast</i>
----------------	---------------------------------

Description

Low-Level NAIVE Forecast

Usage

```
naive_fit_impl(x, y, id = NULL, seasonal_period = "auto", ...)
```

Arguments

x	A dataframe of xreg (exogenous regressors)
y	A numeric vector of values to fit
id	An optional ID feature to identify different time series. Should be a quoted name.
seasonal_period	Not used for NAIVE forecast but here for consistency with SNAIVE
...	Not currently used

naive_predict_impl	<i>Bridge prediction function for NAIVE Models</i>
--------------------	--

Description

Bridge prediction function for NAIVE Models

Usage

```
naive_predict_impl(object, new_data)
```

Arguments

object	An object of class <code>model_fit</code>
new_data	A rectangular data object, such as a data frame.

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 identifier 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")` specifies 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)
library(modeltime)

# 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(stats)
library(tidyverse)
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_fit_impl

Low-Level NNETAR function for translating modeltime to forecast

Description

Low-Level NNETAR function for translating modeltime to forecast

Usage

```
nnetar_fit_impl(
  x,
  y,
  period = "auto",
  p = 1,
  P = 1,
  size = 10,
  repeats = 20,
  decay = 0,
  maxit = 100,
  ...
)
```

Arguments

x	A dataframe of xreg (exogenous regressors)
y	A numeric vector of values to fit
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.
p	Embedding dimension for non-seasonal time series. Number of non-seasonal lags used as inputs. For non-seasonal time series, the default is the optimal number of lags (according to the AIC) for a linear AR(p) model. For seasonal time series, the same method is used but applied to seasonally adjusted data (from an stl decomposition).
P	Number of seasonal lags used as inputs.
size	Number of nodes in the hidden layer. Default is half of the number of input nodes (including external regressors, if given) plus 1.
repeats	Number of networks to fit with different random starting weights. These are then averaged when producing forecasts.
decay	Parameter for weight decay. Default 0.
maxit	Maximum number of iterations. Default 100.
...	Additional arguments passed to forecast::nnetar

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 <i>defaults</i> for the smallest and largest possible values, respectively.
trans	A trans object from the scales package, such as scales::log10_trans() or scales::reciprocal_trans(). 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_predict_impl *Bridge prediction function for ARIMA models*

Description

Bridge prediction function for ARIMA models

Usage

```
nnetar_predict_impl(object, new_data, ...)
```

Arguments

object	An object of class model_fit
new_data	A rectangular data object, such as a data frame.
...	Additional arguments passed to forecast::forecast()

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:

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

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)
library(modeltime)

# 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	<i>Start parallel clusters using parallel package</i>
----------------	---

Description

Start parallel clusters using parallel package

Usage

```
parallel_start(...)

parallel_stop()
```

Arguments

... Parameters passed to `parallel::makeCluster()`

Details

Performs 3 Steps:

1. Makes clusters using `parallel::makeCluster()`
2. Registers clusters using `doParallel::registerDoParallel()`
3. Adds `.libPaths()` using `parallel::clusterCall()`

Examples

```
# Starts 2 clusters
parallel_start(2)

# Returns to sequential processing
parallel_stop()
```

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

<code>data</code>	A data frame
<code>period</code>	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 `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,
  .title = "Forecast Plot",
  .x_lab = "",
  .y_lab = "",
  .color_lab = "Legend",
  .interactive = TRUE,
  .plotly_slider = FALSE,
  ...
)
```

Arguments

<code>.data</code>	A tibble that is the output of <code>modeltime_forecast()</code>
<code>.conf_interval_show</code>	Logical. Whether or not to include the confidence interval as a ribbon.
<code>.conf_interval_fill</code>	Fill color for the confidence interval
<code>.conf_interval_alpha</code>	Fill opacity for the confidence interval. Range (0, 1).
<code>.smooth</code>	Logical - Whether or not to include a trendline smoother. Uses See <code>smooth_vec()</code> to apply a LOESS smoother.
<code>.legend_show</code>	Logical. Whether or not to show the legend. Can save space with long model descriptions.
<code>.legend_max_width</code>	Numeric. The width of truncation to apply to the legend text.
<code>.title</code>	Title for the plot
<code>.x_lab</code>	X-axis label for the plot

<code>.y_lab</code>	Y-axis label for the plot
<code>.color_lab</code>	Legend label if a <code>color_var</code> is used.
<code>.interactive</code>	Returns either a static (ggplot2) visualization or an interactive (plotly) visualization
<code>.plotly_slider</code>	If TRUE, returns a plotly date range slider.
<code>...</code>	Additional arguments passed to <code>timetk::plot_time_series()</code> .

Value

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

Examples

```
library(tidyverse)
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: auto_arima ----
model_fit_arima <- arima_reg() %>%
  set_engine(engine = "auto_arima") %>%
  fit(value ~ date, data = training(splits))

# ---- MODELTIME TABLE ----

models_tbl <- modeltime_table(
  model_fit_arima
)

# ---- 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: [plot_time_series\(\)](#)
- ACF Plot: [plot_acf_diagnostics\(\)](#)
- Seasonality Plot: [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

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

... Additional arguments passed to:

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

Value

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

Examples

```
library(tidyverse)
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: auto_arima ----
model_fit_arima <- arima_reg() %>%
  set_engine(engine = "auto_arima") %>%
  fit(value ~ date, data = training(splits))

# ---- MODELTIME TABLE ----

models_tbl <- modeltime_table(
  model_fit_arima
)

# ---- 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 synonyms.

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

<code>object</code>	A Modeltime Table
<code>.model_id</code>	A numeric value matching the <code>.model_id</code> 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
- [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)
```

predict.recursive *Recursive Model Predictions*

Description

Make predictions from a recursive model.

Usage

```
## S3 method for class 'recursive'  
predict(object, new_data, type = NULL, opts = list(), ...)
```

Arguments

object	An object of class <code>model_fit</code>
new_data	A rectangular data object, such as a data frame.
type	A single character value or NULL. Possible values are "numeric", "class", "prob", "conf_int", "pred_int", "quantile", "time", "hazard", "survival", or "raw". When NULL, <code>predict()</code> will choose an appropriate value based on the model's mode.
opts	A list of optional arguments to the underlying predict function that will be used when <code>type = "raw"</code> . The list should not include options for the model object or the new data being predicted.
...	Arguments to the underlying model's prediction function cannot be passed here (see <code>opts</code>). There are some <code>parSNIP</code> related options that can be passed, depending on the value of <code>type</code> . Possible arguments are: <ul style="list-style-type: none">• <code>level</code>: for types of "conf_int" and "pred_int" this is the parameter for the tail area of the intervals (e.g. confidence level for confidence intervals). Default value is 0.95.• <code>std_error</code>: add the standard error of fit or prediction (on the scale of the linear predictors) for types of "conf_int" and "pred_int". Default value is FALSE.• <code>quantile</code>: the quantile(s) for quantile regression (not implemented yet)• <code>time</code>: the time(s) for hazard and survival probability estimates.

Details

Refer to [recursive\(\)](#) for further details and examples.

Value

Numeric values for the recursive panel prediction

predict.recursive_panel

Recursive Model Predictions

Description

Make predictions from a recursive model.

Usage

```
## S3 method for class 'recursive_panel'
predict(object, new_data, type = NULL, opts = list(), ...)
```

Arguments

object	An object of class <code>model_fit</code>
new_data	A rectangular data object, such as a data frame.
type	A single character value or NULL. Possible values are "numeric", "class", "prob", "conf_int", "pred_int", "quantile", "time", "hazard", "survival", or "raw". When NULL, <code>predict()</code> will choose an appropriate value based on the model's mode.
opts	A list of optional arguments to the underlying predict function that will be used when <code>type = "raw"</code> . The list should not include options for the model object or the new data being predicted.
...	Arguments to the underlying model's prediction function cannot be passed here (see <code>opts</code>). There are some <code>parSNIP</code> related options that can be passed, depending on the value of <code>type</code> . Possible arguments are: <ul style="list-style-type: none"> • <code>level</code>: for types of "conf_int" and "pred_int" this is the parameter for the tail area of the intervals (e.g. confidence level for confidence intervals). Default value is 0.95. • <code>std_error</code>: add the standard error of fit or prediction (on the scale of the linear predictors) for types of "conf_int" and "pred_int". Default value is FALSE. • <code>quantile</code>: the quantile(s) for quantile regression (not implemented yet) • <code>time</code>: the time(s) for hazard and survival probability estimates.

Details

Refer to [recursive\(\)](#) for further details and examples.

Value

Numeric values for the recursive panel prediction

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 component 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 (xgboost 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) (xgboost only).
learn_rate	A number for the rate at which the boosting algorithm adapts from iteration-to-iteration (xgboost only).
loss_reduction	A number for the reduction in the loss function required to split further (xgboost 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:

<code>modeltime</code>	<code>prophet</code>
<code>growth</code>	<code>growth ('linear')</code>

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 = list(), data, nrounds, watchlist = list(), obj = NULL,
##   feval = 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)
library(modeltime)

# 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
## Not run:
model_fit <- model_spec %>%
  fit(log(value) ~ date + as.numeric(date) + month(date, label = TRUE),
      data = training(splits))
model_fit

## End(Not run)
```


Description

Low-Level PROPHET function for translating modeltime to PROPHET

Usage

```
prophet_fit_impl(
  x,
  y,
  growth = "linear",
  n.changepoints = 25,
  changepoint.range = 0.8,
  yearly.seasonality = "auto",
  weekly.seasonality = "auto",
  daily.seasonality = "auto",
  seasonality.mode = "additive",
  changepoint.prior.scale = 0.05,
  seasonality.prior.scale = 10,
  holidays.prior.scale = 10,
  regressors.prior.scale = 10000,
  regressors.standardize = "auto",
  regressors.mode = NULL,
  logistic_cap = NULL,
  logistic_floor = NULL,
  ...
)
```

Arguments

<code>x</code>	A dataframe of xreg (exogenous regressors)
<code>y</code>	A numeric vector of values to fit
<code>growth</code>	String 'linear', 'logistic', or 'flat' to specify a linear, logistic or flat trend.
<code>n.changepoints</code>	Number of potential changepoints to include. Not used if input 'changepoints' is supplied. If 'changepoints' is not supplied, then n.changepoints potential changepoints are selected uniformly from the first 'changepoint.range' proportion of df\$ds.
<code>changepoint.range</code>	Proportion of history in which trend changepoints will be estimated. Defaults to 0.8 for the first 80 'changepoints' is specified.
<code>yearly.seasonality</code>	Fit yearly seasonality. Can be 'auto', TRUE, FALSE, or a number of Fourier terms to generate.
<code>weekly.seasonality</code>	Fit weekly seasonality. Can be 'auto', TRUE, FALSE, or a number of Fourier terms to generate.
<code>daily.seasonality</code>	Fit daily seasonality. Can be 'auto', TRUE, FALSE, or a number of Fourier terms to generate.

seasonality.mode	'additive' (default) or 'multiplicative'.
changepoint.prior.scale	Parameter modulating the flexibility of the automatic changepoint selection. Large values will allow many changepoints, small values will allow few changepoints.
seasonality.prior.scale	Parameter modulating the strength of the seasonality model. Larger values allow the model to fit larger seasonal fluctuations, smaller values dampen the seasonality. Can be specified for individual seasonalities using add_seasonality.
holidays.prior.scale	Parameter modulating the strength of the holiday components model, unless overridden in the holidays input.
regressors.prior.scale	Float scale for the normal prior. Default is 10,000. Gets passed to prophet::add_regressor(prior.scale)
regressors.standardize	Bool, specify whether this regressor will be standardized prior to fitting. Can be 'auto' (standardize if not binary), True, or False. Gets passed to prophet::add_regressor(standardize)
regressors.mode	Optional, 'additive' or 'multiplicative'. Defaults to seasonality.mode.
logistic_cap	When growth is logistic, the upper-bound for "saturation".
logistic_floor	When growth is logistic, the lower-bound for "saturation".
...	Additional arguments passed to prophet::prophet

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 <i>defaults</i> for the smallest and largest possible values, respectively.
trans	A trans object from the scales package, such as <code>scales::log10_trans()</code> or <code>scales::reciprocal_trans()</code> . 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_predict_impl Bridge prediction function for PROPHET models
```

Description

Bridge prediction function for PROPHET models

Usage

```
prophet_predict_impl(object, new_data, ...)
```

Arguments

object	An object of class <code>model_fit</code>
new_data	A rectangular data object, such as a data frame.
...	Additional arguments passed to <code>prophet::predict()</code>

```
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 component 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 change-point can be located.
- `season`: 'additive' (default) or 'multiplicative'.
- `prior_scale_changepoints`: Parameter modulating the flexibility of the automatic change-point 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:

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

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 `arma_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(modeltime)

# 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
```

`prophet_xgboost_fit_impl`

Low-Level PROPHET function for translating modeltime to Boosted PROPHET

Description

Low-Level PROPHET function for translating modeltime to Boosted PROPHET

Usage

```
prophet_xgboost_fit_impl(  
  x,  
  y,  
  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,  
  logistic_cap = NULL,  
  logistic_floor = NULL,  
  mcmc.samples = 0,  
  interval.width = 0.8,  
  uncertainty.samples = 1000,  
  fit = TRUE,  
  max_depth = 6,  
  nrounds = 15,  
  eta = 0.3,  
  colsample_bytree = NULL,  
  colsample_bynode = NULL,  
  min_child_weight = 1,  
  gamma = 0,  
  subsample = 1,  
  validation = 0,  
  early_stop = NULL,  
  ...  
)
```

Arguments

x	A dataframe of xreg (exogenous regressors)
y	A numeric vector of values to fit
df	(optional) Dataframe containing the history. Must have columns ds (date type) and y, the time series. If growth is logistic, then df must also have a column cap that specifies the capacity at each ds. If not provided, then the model object will be instantiated but not fit; use fit.prophet(m, df) to fit the model.
growth	String 'linear', 'logistic', or 'flat' to specify a linear, logistic or flat trend.
changepoints	Vector of dates at which to include potential changepoints. If not specified, potential changepoints are selected automatically.
n.changepoints	Number of potential changepoints to include. Not used if input 'changepoints' is supplied. If 'changepoints' is not supplied, then n.changepoints potential changepoints are selected uniformly from the first 'changepoint.range' proportion of df\$ds.
changepoint.range	Proportion of history in which trend changepoints will be estimated. Defaults to 0.8 for the first 80 'changepoints' is specified.
yearly.seasonality	Fit yearly seasonality. Can be 'auto', TRUE, FALSE, or a number of Fourier terms to generate.
weekly.seasonality	Fit weekly seasonality. Can be 'auto', TRUE, FALSE, or a number of Fourier terms to generate.
daily.seasonality	Fit daily seasonality. Can be 'auto', TRUE, FALSE, or a number of Fourier terms to generate.
holidays	data frame with columns holiday (character) and ds (date type)and optionally columns lower_window and upper_window which specify a range of days around the date to be included as holidays. lower_window=-2 will include 2 days prior to the date as holidays. Also optionally can have a column prior_scale specifying the prior scale for each holiday.
seasonality.mode	'additive' (default) or 'multiplicative'.
seasonality.prior.scale	Parameter modulating the strength of the seasonality model. Larger values allow the model to fit larger seasonal fluctuations, smaller values dampen the seasonality. Can be specified for individual seasonalities using add_seasonality.
holidays.prior.scale	Parameter modulating the strength of the holiday components model, unless overridden in the holidays input.
changepoint.prior.scale	Parameter modulating the flexibility of the automatic changepoint selection. Large values will allow many changepoints, small values will allow few changepoints.

logistic_cap	When growth is logistic, the upper-bound for "saturation".
logistic_floor	When growth is logistic, the lower-bound for "saturation".
mcmc.samples	Integer, if greater than 0, will do full Bayesian inference with the specified number of MCMC samples. If 0, will do MAP estimation.
interval.width	Numeric, width of the uncertainty intervals provided for the forecast. If mcmc.samples=0, this will be only the uncertainty in the trend using the MAP estimate of the extrapolated generative model. If mcmc.samples>0, this will be integrated over all model parameters, which will include uncertainty in seasonality.
uncertainty.samples	Number of simulated draws used to estimate uncertainty intervals. Settings this value to 0 or False will disable uncertainty estimation and speed up the calculation.
fit	Boolean, if FALSE the model is initialized but not fit.
max_depth	An integer for the maximum depth of the tree.
nrounds	An integer for the number of boosting iterations.
eta	A numeric value between zero and one to control the learning rate.
colsample_bytree	Subsampling proportion of columns.
colsample_bynode	Subsampling proportion of columns for each node within each tree. See the counts argument below. The default uses all columns.
min_child_weight	A numeric value for the minimum sum of instance weights needed in a child to continue to split.
gamma	A number for the minimum loss reduction required to make a further partition on a leaf node of the tree
subsample	Subsampling proportion of rows.
validation	A positive number. If on [0, 1) the value, validation is a random proportion of data in x and y that are used for performance assessment and potential early stopping. If 1 or greater, it is the <i>number</i> of training set samples use for these purposes.
early_stop	An integer or NULL. If not NULL, it is the number of training iterations without improvement before stopping. If validation is used, performance is base on the validation set; otherwise the training set is used.
...	Additional arguments passed to xgboost::xgb.train

 prophet_xgboost_predict_impl

Bridge prediction function for Boosted PROPHET models

Description

Bridge prediction function for Boosted PROPHET models

Usage

```
prophet_xgboost_predict_impl(object, new_data, ...)
```

Arguments

object	An object of class <code>model_fit</code>
new_data	A rectangular data object, such as a data frame.
...	Additional arguments passed to <code>prophet::predict()</code>

`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 <code>model_fit</code> object
--------	--

Value

A formula using `stats::formula()`

recipe_helpers	<i>Developer Tools for processing XREGS (Regressors)</i>
----------------	--

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: <ul style="list-style-type: none"> • <code>tbl</code>: Returns a tibble (<code>data.frame</code>) • <code>matrix</code>: 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	<i>Create a Recursive Time Series Model from a Parsnip or Workflow Regression Model</i>
-----------	---

Description

Create a Recursive Time Series Model from a Parsnip or Workflow Regression Model

Usage

```
recursive(object, transform, train_tail, id = NULL, ...)
```

Arguments

object	An object of <code>model_fit</code> or a fitted workflow class
transform	A transformation performed on <code>new_data</code> after each step of recursive algorithm. <ul style="list-style-type: none"> • Transformation Function: Must have one argument <code>data</code> (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. <code>id = "id"</code>).
...	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(modeltime)
library(tidymodels)
library(tidyverse)
library(lubridate)
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	<i>General Interface for Multiple Seasonality Regression Models (TBATS, STLTM)</i>
--------------	--

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

<code>mode</code>	A single character string for the type of model. The only possible value for this model is "regression".
<code>seasonal_period_1</code>	(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.
<code>seasonal_period_2</code>	(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.
<code>seasonal_period_3</code>	(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:

<code>modeltime</code>	<code>forecast::stlm</code>	<code>forecast::tbats</code>
<code>seasonal_period_1</code> , <code>seasonal_period_2</code> , <code>seasonal_period_3</code>	<code>msts(seasonal.periods)</code>	<code>msts(seasonal.periods)</code>

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)
library(modeltime)

# 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
```

`snaive_fit_impl`*Low-Level SNAIVE Forecast*

Description

Low-Level SNAIVE Forecast

Usage`snaive_fit_impl(x, y, id = NULL, seasonal_period = "auto", ...)`

Arguments

x	A dataframe of xreg (exogenous regressors)
y	A numeric vector of values to fit
id	An optional ID feature to identify different time series. Should be a quoted name.
seasonal_period	The seasonal period to forecast into the future
...	Not currently used

snaive_predict_impl *Bridge prediction function for SNAIVE Models*

Description

Bridge prediction function for SNAIVE Models

Usage

```
snaive_predict_impl(object, new_data)
```

Arguments

object	An object of class model_fit
new_data	A rectangular data object, such as a data frame.

stlm_arma_fit_impl *Low-Level stlm function for translating modeltime to forecast*

Description

Low-Level stlm function for translating modeltime to forecast

Usage

```
stlm_arma_fit_impl(
  x,
  y,
  period_1 = "auto",
  period_2 = NULL,
  period_3 = NULL,
  ...
)
```

Arguments

x	A dataframe of xreg (exogenous regressors)
y	A numeric vector of values to fit
period_1	(required) First 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.
period_2	(optional) First 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.
period_3	(optional) First 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.
...	Additional arguments passed to <code>forecast::stlm()</code>

stlm_arma_predict_impl

Bridge prediction function for ARIMA models

Description

Bridge prediction function for ARIMA models

Usage

```
stlm_arma_predict_impl(object, new_data, ...)
```

Arguments

object	An object of class <code>model_fit</code>
new_data	A rectangular data object, such as a data frame.
...	Additional arguments passed to <code>forecast::forecast()</code>

stlm_ets_fit_impl

Low-Level stlm function for translating modeltime to forecast

Description

Low-Level stlm function for translating modeltime to forecast

Usage

```

stlm_ets_fit_impl(
  x,
  y,
  period_1 = "auto",
  period_2 = NULL,
  period_3 = NULL,
  ...
)

```

Arguments

x	A dataframe of xreg (exogenous regressors)
y	A numeric vector of values to fit
period_1	(required) First 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.
period_2	(optional) First 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.
period_3	(optional) First 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.
...	Additional arguments passed to <code>forecast::stlm()</code>

stlm_ets_predict_impl *Bridge prediction function for ARIMA models*

Description

Bridge prediction function for ARIMA models

Usage

```

stlm_ets_predict_impl(object, new_data, ...)

```

Arguments

object	An object of class <code>model_fit</code>
new_data	A rectangular data object, such as a data frame.
...	Additional arguments passed to <code>forecast::forecast()</code>

`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

<code>data</code>	A <code>data.frame</code> containing the truth and estimate columns.
<code>truth</code>	The column identifier for the true results (that is numeric).
<code>estimate</code>	The column identifier for the predicted results (that is also numeric).
<code>metric_set</code>	A <code>yardstick::metric_set()</code> that is used to summarize one or more forecast accuracy (regression) metrics.

Examples

```
library(tibble)
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 %>%
  group_by(group) %>%
  summarize_accuracy_metrics(
    truth, estimate,
    metric_set = default_forecast_accuracy_metric_set()
  )
```

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

<code>.data</code>	A tibble that is the output of <code>modeltime_accuracy()</code>
<code>.round_digits</code>	Rounds accuracy metrics to a specified number of digits. If NULL, rounding is not performed.
<code>.sortable</code>	Allows sorting by columns. Only applied to reactable tables. Passed to <code>reactable(sortable)</code> .
<code>.show_sortable</code>	Shows sorting. Only applied to reactable tables. Passed to <code>reactable(showSortable)</code> .
<code>.searchable</code>	Adds search input. Only applied to reactable tables. Passed to <code>reactable(searchable)</code> .
<code>.filterable</code>	Adds filters to table columns. Only applied to reactable tables. Passed to <code>reactable(filterable)</code> .
<code>.expand_groups</code>	Expands groups dropdowns. Only applied to reactable tables. Passed to <code>reactable(defaultExpanded)</code> .
<code>.title</code>	A title for static (gt) tables.
<code>.interactive</code>	Return interactive or static tables. If TRUE, returns reactable table. If FALSE, returns static gt table.
<code>...</code>	Additional arguments passed to <code>reactable::reactable()</code> or <code>gt::gt()</code> (depending on <code>.interactive</code> 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(tidyverse)
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: auto_arima ----
model_fit_arima <- arima_reg() %>%
  set_engine(engine = "auto_arima") %>%
  fit(value ~ date, data = training(splits))

# ---- MODELTIME TABLE ----

models_tbl <- modeltime_table(
  model_fit_arima
)

# ---- ACCURACY ----
```

```
models_tbl %>%
  modeltime_calibrate(new_data = testing(splits)) %>%
  modeltime_accuracy() %>%
  table_modeltime_accuracy()
```

<code>tbats_fit_impl</code>	<i>Low-Level tbats function for translating modeltime to forecast</i>
-----------------------------	---

Description

Low-Level tbats function for translating modeltime to forecast

Usage

```
tbats_fit_impl(
  x,
  y,
  period_1 = "auto",
  period_2 = NULL,
  period_3 = NULL,
  use.parallel = length(y) > 1000,
  ...
)
```

Arguments

<code>x</code>	A dataframe of xreg (exogenous regressors)
<code>y</code>	A numeric vector of values to fit
<code>period_1</code>	(required) First 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.
<code>period_2</code>	(optional) First 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.
<code>period_3</code>	(optional) First 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.
<code>use.parallel</code>	TRUE/FALSE indicates whether or not to use parallel processing.
<code>...</code>	Additional arguments passed to <code>forecast::tbats()</code>

tbats_predict_impl *Bridge prediction function for ARIMA models*

Description

Bridge prediction function for ARIMA models

Usage

```
tbats_predict_impl(object, new_data, ...)
```

Arguments

object	An object of class <code>model_fit</code>
new_data	A rectangular data object, such as a data frame.
...	Additional arguments passed to <code>forecast::forecast()</code>

theta_fit_impl *Low-Level Exponential Smoothing function for translating modeltime to forecast*

Description

Low-Level Exponential Smoothing function for translating modeltime to forecast

Usage

```
theta_fit_impl(x, y, ...)
```

Arguments

x	A dataframe of <code>xreg</code> (exogenous regressors)
y	A numeric vector of values to fit
...	Additional arguments passed to <code>forecast::ets</code>

theta_predict_impl	<i>Bridge prediction function for THETA models</i>
--------------------	--

Description

Bridge prediction function for THETA models

Usage

```
theta_predict_impl(object, new_data, ...)
```

Arguments

object	An object of class <code>model_fit</code>
new_data	A rectangular data object, such as a data frame.
...	Additional arguments passed to <code>stats::predict()</code>

time_series_params	<i>Tuning Parameters for Time Series (ts-class) Models</i>
--------------------	--

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 Seasonality**: 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()
```

type_sum.mdl_time_tbl *Succinct summary of Modeltime Tables*

Description

type_sum controls how objects are shown when inside tibble columns.

Usage

```
## S3 method for class 'mdl_time_tbl'
type_sum(x)
```

Arguments

x A mdl_time_tbl object to summarise.

Value

A character value.

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
- `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

<code>object</code>	A Modeltime Table
<code>.model_id</code>	A numeric value matching the <code>.model_id</code> that you want to update
<code>.new_model_desc</code>	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
- `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_function_fit_impl
```

Low-Level Window Forecast

Description

Low-Level Window Forecast

Usage

```
window_function_fit_impl(
  x,
  y,
  id = NULL,
  window_size = "all",
  window_function = NULL,
  ...
)
```

Arguments

<code>x</code>	A dataframe of xreg (exogenous regressors)
<code>y</code>	A numeric vector of values to fit
<code>id</code>	An optional ID feature to identify different time series. Should be a quoted name.
<code>window_size</code>	The period to apply the window function to
<code>window_function</code>	A function to apply to the window. The default is <code>mean()</code> .
<code>...</code>	Additional arguments for the <code>window_function</code> . For example, it's common to pass <code>na.rm = TRUE</code> for the mean forecast.

window_function_predict_impl
Bridge prediction function for window Models

Description

Bridge prediction function for window Models

Usage

```
window_function_predict_impl(object, new_data)
```

Arguments

object	An object of class <code>model_fit</code>
new_data	A rectangular data object, such as a data frame.

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 identifier 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")` specifies 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)
library(modeltime)

# 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
# broadcasting 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))
```

```
xgboost_impl
```

```
Wrapper for parsnip::xgb_train
```

Description

Wrapper for `parsnip::xgb_train`

Usage

```
xgboost_impl(
  x,
  y,
  max_depth = 6,
  nrounds = 15,
  eta = 0.3,
  colsample_bynode = NULL,
  colsample_bytree = NULL,
  min_child_weight = 1,
  gamma = 0,
  subsample = 1,
  validation = 0,
  early_stop = NULL,
  objective = NULL,
  counts = TRUE,
  event_level = c("first", "second"),
  ...
)
```

Arguments

<code>x</code>	A data frame or matrix of predictors
<code>y</code>	A vector (factor or numeric) or matrix (numeric) of outcome data.
<code>max_depth</code>	An integer for the maximum depth of the tree.
<code>nrounds</code>	An integer for the number of boosting iterations.
<code>eta</code>	A numeric value between zero and one to control the learning rate.
<code>colsample_bynode</code>	Subsampling proportion of columns for each node within each tree. See the <code>counts</code> argument below. The default uses all columns.
<code>colsample_bytree</code>	Subsampling proportion of columns for each tree. See the <code>counts</code> argument below. The default uses all columns.

min_child_weight	A numeric value for the minimum sum of instance weights needed in a child to continue to split.
gamma	A number for the minimum loss reduction required to make a further partition on a leaf node of the tree
subsample	Subsampling proportion of rows. By default, all of the training data are used.
validation	A positive number. If on [0, 1) the value, validation is a random proportion of data in x and y that are used for performance assessment and potential early stopping. If 1 or greater, it is the <i>number</i> of training set samples use for these purposes.
early_stop	An integer or NULL. If not NULL, it is the number of training iterations without improvement before stopping. If validation is used, performance is base on the validation set; otherwise the training set is used.
objective	A single string (or NULL) that defines the loss function that xgboost uses to create trees. See <code>xgboost::xgb.train()</code> for options. If left NULL, an appropriate loss function is chosen.
counts	A logical. If FALSE, colsample_bynode and colsample_bytree are both assumed to be <i>proportions</i> of the proportion of columns affects (instead of counts).
event_level	For binary classification, this is a single string of either "first" or "second" to pass along describing which level of the outcome should be considered the "event".
...	Other options to pass to <code>xgb.train</code> .

xgboost_predict	<i>Wrapper for xgboost::predict</i>
-----------------	-------------------------------------

Description

Wrapper for `xgboost::predict`

Usage

```
xgboost_predict(object, newdata, ...)
```

Arguments

object	a model object for which prediction is desired.
newdata	New data to be predicted
...	additional arguments affecting the predictions produced.

Index

- * **datasets**
 - m750, [44](#)
 - m750_models, [45](#)
 - m750_splits, [45](#)
 - m750_training_resamples, [46](#)
- .prepare_panel_transform
 - (.prepare_transform), [4](#)
- .prepare_transform, [4](#)
- add_modeltime_model, [4](#)
- add_modeltime_model(), [5](#), [25](#), [80](#), [120](#), [121](#)
- arima_boost, [5](#)
- Arima_fit_impl, [10](#)
- arima_params, [12](#)
- Arima_predict_impl, [13](#)
- arima_reg, [13](#)
- arima_xgboost_fit_impl, [18](#)
- arima_xgboost_predict_impl, [20](#)
- as_modeltime_table(modeltime_table), [61](#)
- auto_arima_fit_impl, [20](#)
- auto_arima_xgboost_fit_impl, [21](#)
- bake_xreg_recipe(recipe_helpers), [101](#)
- changepoint_num(prophet_params), [90](#)
- changepoint_range(prophet_params), [90](#)
- combine_modeltime_tables, [24](#)
- combine_modeltime_tables(), [5](#), [25](#), [80](#), [120](#), [121](#)
- control_fit_workflowset, [26](#)
- control_fit_workflowset(), [51](#)
- control_refit, [27](#)
- control_refit(), [56](#)
- create_model_grid, [28](#)
- create_xreg_recipe, [29](#)
- croston_fit_impl, [31](#)
- croston_predict_impl, [31](#)
- damping(exp_smoothing_params), [39](#)
- default_forecast_accuracy_metric_set,
[32](#)
- default_forecast_accuracy_metric_set(),
[47](#)
- dials::epochs(), [69](#)
- dials::grid_regular(), [28](#)
- dials::hidden_units(), [69](#)
- dials::penalty(), [69](#)
- error(exp_smoothing_params), [39](#)
- ets_fit_impl, [33](#)
- ets_predict_impl, [34](#)
- exp_smoothing, [35](#)
- exp_smoothing_params, [39](#)
- fit.model_spec(), [9](#), [17](#), [38](#), [49](#), [65](#), [72](#), [88](#),
[96](#), [108](#), [124](#)
- fit.workflow(), [49](#)
- forecast::Arima(), [7](#), [14–16](#)
- forecast::auto.arima(), [7](#), [14](#), [15](#)
- forecast::croston(), [36](#), [37](#)
- forecast::ets(), [36](#)
- forecast::msts(), [118](#)
- forecast::nnetar(), [71](#)
- forecast::thetaf(), [36](#), [37](#)
- get_arima_description, [40](#)
- get_model_description, [41](#)
- get_tbats_description, [42](#)
- growth(prophet_params), [90](#)
- gt::gt(), [114](#), [115](#)
- is_calibrated, [42](#)
- is_modeltime_model, [43](#)
- is_modeltime_table, [43](#)
- is_residuals, [43](#)
- juice_xreg_recipe(recipe_helpers), [101](#)
- load_namespace, [44](#)
- m750, [44](#)
- m750_models, [45](#)

- m750_splits, 45
- m750_training_resamples, 46
- mae(), 32, 47
- mape(), 32, 47
- mase(), 32, 47
- metric_set(), 32
- modeltime_accuracy, 47
- modeltime_accuracy(), 32, 49, 114
- modeltime_calibrate, 48
- modeltime_calibrate(), 25, 53, 54
- modeltime_fit_workflowset, 50
- modeltime_fit_workflowset(), 26, 28
- modeltime_forecast, 52
- modeltime_forecast(), 49, 76
- modeltime_refit, 56
- modeltime_refit(), 25, 27, 53
- modeltime_residuals, 57
- modeltime_residuals(), 78
- modeltime_residuals_test, 59
- modeltime_table, 61
- modeltime_table(), 49

- naive_fit_impl, 63
- naive_fit_impl(), 64
- naive_predict_impl, 63
- naive_reg, 64
- ndiffs, 23
- new_modeltime_bridge, 66
- nnetar_fit_impl, 67
- nnetar_params, 68
- nnetar_predict_impl, 69
- nnetar_reg, 70
- non_seasonal_ar (arima_params), 12
- non_seasonal_ar(), 69
- non_seasonal_differences (arima_params), 12
- non_seasonal_ma (arima_params), 12
- nsdiffs, 23
- num_networks (nnetar_params), 68

- panel_tail, 73
- panel_tail(), 103
- parallel_start, 74
- parallel_stop (parallel_start), 74
- parse_index, 75
- parse_index_from_data (parse_index), 75
- parse_period_from_index (parse_index), 75
- plot_acf_diagnostics(), 78, 79

- plot_modeltime_forecast, 76
- plot_modeltime_forecast(), 53
- plot_modeltime_residuals, 78
- plot_seasonal_diagnostics(), 78, 79
- plot_time_series(), 76, 78, 79
- pluck_modeltime_model, 80
- predict_recursive, 81
- predict_recursive_panel, 82
- prior_scale_changepoints (prophet_params), 90
- prior_scale_holidays (prophet_params), 90
- prior_scale_seasonality (prophet_params), 90
- prophet::prophet(), 84, 93, 94
- prophet_boost, 83
- prophet_fit_impl, 88
- prophet_params, 90
- prophet_predict_impl, 92
- prophet_reg, 92
- prophet_xgboost_fit_impl, 97
- prophet_xgboost_predict_impl, 99
- pull_modeltime_model (pluck_modeltime_model), 80
- pull_modeltime_model(), 5, 25, 80, 120, 121
- pull_modeltime_residuals, 100
- pull_parsnip_preprocessor, 100

- reactable::reactable(), 114, 115
- recipe_helpers, 101
- recursive, 102
- recursive(), 74, 81, 82
- rmse(), 32, 47
- rsq(), 32, 47

- season (exp_smoothing_params), 39
- season(), 91
- seasonal_ar (arima_params), 12
- seasonal_ar(), 69
- seasonal_differences (arima_params), 12
- seasonal_ma (arima_params), 12
- seasonal_period (time_series_params), 118
- seasonal_reg, 106
- seasonality_daily (prophet_params), 90
- seasonality_weekly (prophet_params), 90
- seasonality_yearly (prophet_params), 90

set_engine(), [9](#), [17](#), [38](#), [65](#), [72](#), [88](#), [96](#), [108](#),
[124](#)
 smape(), [32](#), [47](#)
 smooth_level(exp_smoothing_params), [39](#)
 smooth_seasonal(exp_smoothing_params),
[39](#)
 smooth_trend(exp_smoothing_params), [39](#)
 smooth_vec(), [76](#), [78](#)
 snaive_fit_impl, [109](#)
 snaive_fit_impl(), [64](#)
 snaive_predict_impl, [110](#)
 stats::Box.test(), [60](#)
 stats::shapiro.test(), [60](#)
 stats::ts(), [118](#)
 stlm_arma_fit_impl, [110](#)
 stlm_arma_predict_impl, [111](#)
 stlm_ets_fit_impl, [111](#)
 stlm_ets_predict_impl, [112](#)
 summarize_accuracy_metrics, [113](#)

table_modeltime_accuracy, [114](#)
 tail(), [103](#)
 tbats_fit_impl, [116](#)
 tbats_predict_impl, [117](#)
 theta_fit_impl, [117](#)
 theta_predict_impl, [118](#)
 time_series_params, [118](#)
 timetk::plot_time_series(), [77](#)
 trend(exp_smoothing_params), [39](#)
 type_sum.mdl_time_tbl, [119](#)

update_model_description, [120](#)
 update_modeltime_description
 (update_model_description), [120](#)
 update_modeltime_description(), [5](#), [25](#),
[80](#), [120](#), [121](#)
 update_modeltime_model, [119](#)
 update_modeltime_model(), [5](#), [25](#), [80](#), [120](#),
[121](#)

window_function_fit_impl, [121](#)
 window_function_fit_impl(), [123](#)
 window_function_predict_impl, [122](#)
 window_reg, [122](#)
 workflowsets::workflow_set(), [28](#)

xgboost::xgb.train, [7](#)
 xgboost::xgb.train(), [84](#), [127](#)
 xgboost_impl, [126](#)
 xgboost_predict, [127](#)
 yardstick::metric_tweak(), [32](#)