Package ‘modeltime’

January 12, 2022

Title The Tidymodels Extension for Time Series Modeling

Version 1.1.1

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

https://business-science.github.io/modeltime/

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

License MIT + file LICENSE

Encoding UTF-8

LazyData true

Depends R (>= 3.5.0)

Imports StanHeaders, timetk (>= 2.6.0), 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, sparklyr, tidymodels, workflowsets, recipes, rsample, tune, tidyverse, lubridate, progress, testthat, roxygen2, kernlab, glmnet, thief, smooth, greybox, earth, randomForest, tidyquant, knitr, rmarkdown (>= 2.9), webshot, qpfd, covr, TSEpr

VignetteBuilder knitr

RoxygenNote 7.1.2

NeedsCompilation no
Author  Matt Dancho [aut, cre],
        Business Science [cph]
Maintainer  Matt Dancho <mdancho@business-science.io>
Repository  CRAN
Date/Publication  2022-01-12 16:22:42 UTC

R topics documented:

.prepare_transform .................................................. 4
adam_fit_impl .......................................................... 5
adam_params ............................................................ 6
Adam_predict_impl ..................................................... 8
adam_reg ............................................................... 8
add_modeltime_model .................................................. 14
arima_boost ............................................................ 15
Arima_fit_impl ......................................................... 20
arima_params ........................................................... 21
Arima_predict_impl .................................................... 22
arima_reg ............................................................... 23
arima_xgboost_fit_impl ............................................... 27
arima_xgboost_predict_impl .......................................... 30
auto_adam_fit_impl .................................................... 30
Auto_adam_predict_impl ............................................... 32
auto_arima_fit_impl ................................................... 32
auto_arima_xgboost_fit_impl ......................................... 33
combine_modeltime_tables ............................................. 36
control_modeltime ..................................................... 38
create_model_grid ...................................................... 40
create_xreg_recipe ...................................................... 41
croston_fit_impl ....................................................... 43
croston_predict_impl ................................................... 43
ets_fit_impl ............................................................ 44
ets_predict_impl ........................................................ 45
exp_smoothing .......................................................... 45
exp_smoothing_params .................................................. 51
get_arima_description .................................................. 53
get_model_description .................................................. 53
get_tbats_description .................................................. 54
is_calibrated ........................................................... 55
is_modeltime_model ..................................................... 55
is_modeltime_table ..................................................... 55
is_residuals ............................................................. 56
load_namespace .......................................................... 56
log_extractors .......................................................... 57
m750 ................................................................. 58
m750_models ............................................................ 58
m750_splits ............................................................. 59
R topics documented:

m750_training_resamples ................................................. 60
maape ................................................................. 60
maape.data.frame ....................................................... 61
maape_vec ............................................................. 61
make_ts_splits .......................................................... 62
metric_sets ........................................................... 62
modeltime_accuracy ..................................................... 64
modeltime_calibrate ..................................................... 66
modeltime_fit_workflowset .............................................. 68
modeltime_forecast ..................................................... 69
modeltime_nested_fit ................................................... 73
modeltime_nested_forecast ............................................. 74
modeltime_nested_refit ............................................... 76
modeltime_nested_select_best ....................................... 76
modeltime_refit ....................................................... 77
modeltime_residuals ..................................................... 79
modeltime_residuals_test ............................................... 80
modeltime_table ....................................................... 82
naive_fit_impl ........................................................ 84
naive_predict_impl .................................................... 85
naive_reg ............................................................... 85
new_modeltime_bridge .................................................. 88
nnetar_fit_impl ......................................................... 89
nnetar_params .......................................................... 90
nnetar_predict_impl ................................................... 91
nnetar_reg ............................................................. 91
panel_tail .............................................................. 94
parallel_start .......................................................... 95
parse_index ............................................................ 96
plot_modeltime_forecast ................................................. 97
plot_modeltime_residuals ................................................. 99
pluck_modeltime_model .................................................. 101
predict.recursive ...................................................... 102
predict.recursive_panel ............................................... 103
prep_nested ........................................................... 104
prophet_boost .......................................................... 106
prophet_fit_impl ...................................................... 112
prophet_params ......................................................... 113
prophet_predict_impl .................................................. 115
prophet_reg ........................................................... 115
prophet_xgboost_fit_impl .............................................. 120
prophet_xgboost_predict_impl ........................................ 123
pull_modeltime_residuals ............................................... 123
pull_parsnip_preprocessor ............................................ 124
recipe_helpers ........................................................ 124
recursive ............................................................... 125
seasonal_reg ........................................................... 129
smooth_fit_impl ........................................................ 133
Prepare Recursive Transformations

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
**adam_fit_impl**  
*Low-Level ADAM function for translating modeltime to forecast*

**Description**
Low-Level ADAM function for translating modeltime to forecast

**Usage**

```r
adam_fit_impl(
  x,
  y,
  period = "auto",
  p = 0,
  d = 0,
  q = 0,
  P = 0,
  D = 0,
  Q = 0,
  model = "ZXZ",
  constant = FALSE,
  regressors = c("use", "select", "adapt"),
  outliers = c("ignore", "use", "select"),
  level = 0.99,
  occurrence = c("none", "auto", "fixed", "general", "odds-ratio",
  "inverse-odds-ratio", "direct"),
  distribution = c("default", "dnorm", "dlaplace", "ds", "dgnorm", "dlnorm",
  "dinvgauss", "dgamma"),
  loss = c("likelihood", "MSE", "MAE", "HAM", "LASSO", "RIDGE", "MSEh", "TMSE",
  "GTMSE", "MSCE"),
  ic = c("AICc", "AIC", "BIC", "BICc"),
  select_order = FALSE,
  ...
)
```

**Arguments**

- **x**: A data.frame of predictors
- **y**: A vector with outcome
- **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.
Adam\_params

Tuning Parameters for ADAM Models

### Description

Tuning Parameters for ADAM Models

### Usage

```
use\_constant(values = c(FALSE, TRUE))

regressors\_treatment(values = c("use", "select", "adapt"))

outliers\_treatment(values = c("ignore", "use", "select"))

probability\_model(
  values = c("none", "auto", "fixed", "general", "odds\_ratio", "inverse\_odds\_ratio", "direct")
)

distribution(
```
adam_params

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

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

select_order(values = c(FALSE, TRUE))

Arguments

values A character string of possible values.

Details

The main parameters for ADAM models are:

- **non_seasonal_ar**: The order of the non-seasonal auto-regressive (AR) terms.
- **non_seasonal_differences**: The order of integration for non-seasonal differencing.
- **non_seasonal_ma**: The order of the non-seasonal moving average (MA) terms.
- **seasonal_ar**: The order of the seasonal auto-regressive (SAR) terms.
- **seasonal_differences**: The order of integration for seasonal differencing.
- **seasonal_ma**: The order of the seasonal moving average (SMA) terms.
- **use_constant**: Logical, determining, whether the constant is needed in the model or not.
- **regressors_treatment**: The variable defines what to do with the provided explanatory variables.
- **outliers_treatment**: Defines what to do with outliers.
- **probability_model**: The type of model used in probability estimation.
- **distribution**: What density function to assume for the error term.
- **information_criteria**: The information criterion to use in the model selection / combination procedure.
- **select_order**: If TRUE, then the function will select the most appropriate order.

Value

A dials parameter
A parameter
A parameter
A parameter
A parameter
A parameter
A parameter
A parameter
A parameter
A parameter
**Examples**

```r
use_constant()
regressors_treatment()
distribution()
```

---

**Adam_predict_impl**  
*Bridge prediction function for ADAM models*

**Description**

Bridge prediction function for ADAM models

**Usage**

```r
Adam_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 `smooth::adam()`

---

**adam_reg**  
*General Interface for ADAM Regression Models*

**Description**

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

**Usage**

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

Arguments

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

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

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

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

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

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

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

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

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

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

outliers_treatment Defines what to do with outliers: "ignore", so just returning the model, "detect" outliers based on specified level and include dummies for them in the model, or detect and "select" those of them that reduce ic value.
outliers_ci  What confidence level to use for detection of outliers. Default is 99%.

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

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

loss  The type of Loss Function used in optimization.

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

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

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

Details

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

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

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

Main Arguments

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

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

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

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

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

**auto_adam (default engine)**
The engine uses `smooth::auto.adam()`.

Function Parameters:

```r
## Registered S3 method overwritten by 'greybox':
## method from
## print.pcor lava

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

The **MAXIMUM** nonseasonal ARIMA terms (max.p, max.d, max.q) and seasonal ARIMA terms (max.P, max.D, max.Q) are provided to `forecast::auto.arima()` via `arima_reg()` parameters.

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

Parameter Notes:

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

**adam**
The engine uses `smooth::adam()`.

Function Parameters:

```r
## function (data, model = \"ZXZ\", lags = c(frequency(data)), orders = list(ar = c(0),
##   i = c(0), ma = c(0), select = FALSE), formula = NULL, constant = FALSE, formula = NULL,
##   outliers = c(\"ignore\", \"use\", \"select\"), level = 0.99, occurrence = c(\"none\", \"auto\", \"fixed",
##   \"select\"), level = 0.99, occurrence = c(\"none\", \"auto\", \"fixed", ...)
```
The nonseasonal ARIMA terms (orders) and seasonal ARIMA terms (orders) are provided to `smooth::adam()` via `adam_reg()` parameters. Other options and arguments can be set using `set_engine()`.

Parameter Notes:

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

**Fit Details**

**Date and Date-Time Variable**

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

- `fit(y ~ date)`

**Seasonal Period Specification**

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

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

**Univariate (No xregs, Exogenous Regressors):**

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

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

**Multivariate (xregs, Exogenous Regressors)**

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

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

**Xreg Example:** Suppose you have 3 features:

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

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

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

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

See Also
fit.model_spec(), set_engine()

Examples

```r
## Not run:
library(dplyr)
library(parsnip)
library(rsample)
library(timetk)
library(modeltime)
library(smooth)

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

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

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

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

# ---- STANDARD ADAM ----
# Model Spec
model_spec <- adam_reg(
  seasonal_period = 12,
  non_seasonal_ar = 3,
  non_seasonal_differences = 1,
  non_seasonal_ma = 3,
  seasonal_ar = 1,
  seasonal_differences = 0,
  seasonal_ma = 1)
```
add_modeltime_model

Add a Model into a Modeltime Table

Description
Add a Model into a Modeltime Table

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

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

See Also
- combine_modeltime_tables(): Combine 2 or more Modeltime Tables together
- add_modeltime_model(): Adds a new row with a new model to a Modeltime Table
- 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 %>%

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

## End(Not run)
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 (\texttt{auto.arima}) and standard ARIMA (\texttt{arima}). The main algorithms are:

- Auto ARIMA + XGBoost Errors (\texttt{engine = auto_arima_xgboost}, default)
- ARIMA + XGBoost Errors (\texttt{engine = arima_xgboost})

## Usage

```r
arima_boost(
  mode = "regression",
  seasonal_period = NULL,
  non_seasonal_ar = NULL,
  non_seasonal_differences = NULL,
  non_seasonal_ma = NULL,
  seasonal_ar = NULL,
  seasonal_differences = NULL,
  seasonal_ma = NULL,
  mtry = NULL,
  trees = NULL,
  min_n = NULL,
  tree_depth = NULL,
  learn_rate = NULL,
  loss_reduction = NULL,
  sample_size = NULL,
  stop_iter = NULL
)
```

## Arguments

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

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

- **non_seasonal_ar** The order of the non-seasonal auto-regressive (AR) terms. Often denoted "p" in pdq-notation.
non_seasonal_differences
The order of integration for non-seasonal differencing. Often denoted "d" in pdq-notation.

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

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

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

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

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

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

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

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

learn_rate
A number for the rate at which the boosting algorithm adapts from iteration-to-iteration (specific engines only).

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

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

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

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

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

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

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

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

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

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

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

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

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

Engine Details

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

Model 1: ARIMA:

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

Model 2: XGBoost:

<table>
<thead>
<tr>
<th>modeltime</th>
<th>xgboost::xgb.train</th>
</tr>
</thead>
<tbody>
<tr>
<td>tree_depth</td>
<td>max_depth (6)</td>
</tr>
<tr>
<td>trees</td>
<td>nrounds (15)</td>
</tr>
<tr>
<td>learn_rate</td>
<td>eta (0.3)</td>
</tr>
<tr>
<td>mtry</td>
<td>colsample_bynode (1)</td>
</tr>
<tr>
<td>min_n</td>
<td>min_child_weight (1)</td>
</tr>
<tr>
<td>loss_reduction</td>
<td>gamma (0)</td>
</tr>
<tr>
<td>sample_size</td>
<td>subsample (1)</td>
</tr>
<tr>
<td>stop_iter</td>
<td>early_stop</td>
</tr>
</tbody>
</table>
Other options can be set using `set_engine()`.

**auto_arima_xgboost (default engine)**

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

```r
## 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, ...)
```

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`):

```r
## 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

```r
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
```

Arima_fit_impl <- 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)

Arima_fit_impl  

\textit{Low-Level ARIMA function for translating modeltime to forecast}

\textbf{Description}

Low-Level ARIMA function for translating modeltime to forecast

\textbf{Usage}

\begin{verbatim}
Arima_fit_impl(
  x,
  y,
  period = "auto",
  p = 0,
  d = 0,
  q = 0,
  P = 0,
  D = 0,
\end{verbatim}
\( Q = 0, \)
\( ... \)

### 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 `forecast::Arima`

---

### Description

Tuning Parameters for ARIMA Models

### Usage

- `non_seasonal_ar(range = c(0L, 5L), trans = NULL)`
- `non_seasonal_differences(range = c(0L, 2L), trans = NULL)`
- `non_seasonal_ma(range = c(0L, 5L), trans = NULL)`
- `seasonal_ar(range = c(0L, 2L), trans = NULL)`
- `seasonal_differences(range = c(0L, 1L), trans = NULL)`
- `seasonal_ma(range = c(0L, 2L), trans = NULL)`
Arguments

- **range**: A two-element vector holding the *defaults* for the smallest and largest possible values, respectively.
- **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 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 `model_fit`
- **new_data**: A rectangular data object, such as a data frame.
- **...**: Additional arguments passed to `forecast::Arima()`
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:

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

Other options can be set using set_engine().

auto_arima (default engine)
The engine uses forecast::auto.arima().

Function Parameters:

```r
## 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,
```
## arima_reg

```r
## 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:

```r
## 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 an 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**

```r
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**

```r
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 number 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
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 `model_fit`
- **new_data**: A rectangular data object, such as a data frame.
- **...**: Additional arguments passed to `predict.xgb.Booster()`

auto_adam_fit_impl

_Low-Level ADAM function for translating modeltime to forecast_

**Description**

Low-Level ADAM function for translating modeltime to forecast

**Usage**

```
auto_adam_fit_impl(
  x,
  y,
  period = "auto",
  p = 0,
  d = 0,
  q = 0,
  P = 0,
  D = 0,
  Q = 0,
  model = "ZXZ",
  constant = FALSE,
  regressors = c("use", "select", "adapt"),
  outliers = c("ignore", "use", "select"),
  level = 0.99,
  occurrence = c("none", "auto", "fixed", "general", "odds-ratio",
                 "inverse-odds-ratio", "direct"),
  distribution = c("default", "dnorm", "dlaplace", "ds", "dgnorm", "dlnorm",
```

...
Arguments

x  A data.frame of predictors
y  A vector with outcome
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.
model  The type of ETS model.
constant  Logical, determining, whether the constant is needed in the model or not.
regressors  The variable defines what to do with the provided explanatory variables.
outliers  Defines what to do with outliers.
level  What confidence level to use for detection of outliers.
occurrence  The type of model used in probability estimation.
distribution  what density function to assume for the error term.
loss  The type of Loss Function used in optimization.
ic  The information criterion to use in the model selection / combination procedure.
select_order  If TRUE, then the function will select the most appropriate order using a mech-
              anism similar to auto.msarima(), but implemented in auto.adam(). The values
               list(ar=...,i=...,ma=...) specify the maximum orders to check in this case.
...  Additional arguments passed to smooth::auto.adam
**Auto_adam_predict_impl**

*Bridge prediction function for AUTO ADAM models*

**Description**

Bridge prediction function for AUTO ADAM models

**Usage**

Auto_adam_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 `smooth::auto.adam()`

---

**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,
    ...
)
auto_arima_xgboost_fit_impl

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

```r
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 test.
D Order of seasonal-differencing. If missing, will choose a value based on season.test.
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.
truncat e  An integer value indicating how many observations to use in model selection. The last truncate values of the series are used to select a model when truncate is not NULL and approximation=TRUE. All observations are used if either truncate=NULL or approximation=FALSE.
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.
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

```r
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_modeltime  Control aspects of the training process

Description

These functions are matched to the associated training functions:

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

Usage

control_refit(verbos = FALSE, allow_par = FALSE, cores = -1, packages = NULL)

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

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

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

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

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

Value

A List with the control settings.

See Also

• Setting Up Parallel Processing: parallel_start(), parallel_stop()
• Training Functions: modeltime_refit(), modeltime_fit_workflowset(), modeltime_nested_fit(), modeltime_nested_refit()
  [parallel_stop()]: R:parallel_stop()
  [modeltime_refit()]: R:modelltime_refit()
  [modeltime_fit_workflowset()]: R:modelltime_fit_workflowset()
  [modeltime_nested_fit()]: R:modelltime_nested_fit()
  [modeltime_nested_refit()]: R:modelltime_nested_refit()

Examples

# No parallel processing by default
control_refit()

# Allow parallel processing
control_refit(allow_par = TRUE)

# Set verbosity to show additional training information
control_refit(verbos = TRUE)

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

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

Description

Helper to make parsnip model specs from a dials parameter grid

Usage

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

Arguments

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

Details

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

The process follows:

1. Generate a grid (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.
create_xreg_recipe

Examples

```r
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_xreg_recipe()` makes it simple to automate conversion of raw un-encoded features to machine-learning ready features.

Usage

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

Arguments

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

**clean_names**
Uses `janitor::clean_names()` to process the names and improve robustness to failure during dummy (one-hot) encoding step.

**dummy_encode**
Should factors (categorical data) be

**one_hot**
If `dummy_encode = TRUE`, should the encoding return one column for each feature or one less column than each feature. Default is `FALSE`.

**Details**
The default recipe contains steps to:

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

**Value**
A recipe in either prepared or un-prepared format.

**Examples**

```r
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

Low-Level Exponential Smoothing function for translating modeltime to forecast

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

Bridge prediction function for CROSTON models

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()`
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
**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()`

**exp_smoothing**

*General Interface for Exponential Smoothing State Space Models*

**Description**

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

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

**Usage**

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

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

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

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

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

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

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

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

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

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

Details

Models can be created using the following engines:

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

Other options can be set using set_engine().

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

Function Parameters:

```r
## 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("aic", "aicc", "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:

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

The main arguments are defined using:

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

Parameter Notes:

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

**theta**

The engine uses `forecast::thetaf()`

Parameter Notes:
• xreg - This model is not set up to use exogenous regressors. Only univariate models will be fit.

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

Function Parameters:

```r
## function (y, model = "ZZZ", persistence = NULL, phi = NULL, initial = c("optimal",
## "backcasting"), initialSeason = NULL, ic = c("AICc", "AIC", "BIC",
## "BICc"), loss = c("likelihood", "MSE", "MAE", "HAM", "MSEh", "TMSE",
## "GTMSE", "MSCE"), h = 10, holdout = FALSE, cumulative = FALSE, interval = c("none",
## "parametric", "likelihood", "semiparametric", "nonparametric"), level = 0.95,
## bounds = c("usual", "admissible", "none"), silent = c("all", "graph",
## "legend", "output", "none"), xreg = NULL, xregDo = c("use", "select"),
## initialX = NULL, ...)
```

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

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

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

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

Parameter Notes:

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

**Fit Details**

**Date and Date-Time Variable**

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

```r
fit(y ~ date)
```

**Seasonal Period Specification**

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

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

**Univariate:**

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

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

**Multivariate (xregs, Exogenous Regressors)**

Just for smooth engine.

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

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

**Xreg Example:** Suppose you have 3 features:

1. y (target)
2. date (timestamp),
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**

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

# Data
m750 <- m4_monthly %>% filter(id == "M750")
m750
```
# Split Data 80/20
splits <- initial_time_split(m750, prop = 0.8)

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

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

# ---- STANDARD ETS ----
# Model Spec
model_spec <- exp_smoothing(
  seasonal_period = 12,
  error = "multiplicative",
  trend = "additive",
  season = "multiplicative"
) %>%
  set_engine("ets")

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

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

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

# ---- THETA ----
```r
set_engine("theta")

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

# ---- SMOOTH ----

# Model Spec
model_spec <- exp_smoothing(
  seasonal_period = 12,
  error = "multiplicative",
  trend = "additive_damped",
  season = "additive"
) %>%
  set_engine("smooth_es")

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

---

**exp_smoothing_params**: Tuning Parameters for Exponential Smoothing Models

**Description**

Tuning Parameters for Exponential Smoothing Models

**Usage**

```r
error(values = c("additive", "multiplicative"))
trend(values = c("additive", "multiplicative", "none"))
trend_smooth(
  values = c("additive", "multiplicative", "none", "additive_damped",
  "multiplicative_damped")
)
season(values = c("additive", "multiplicative", "none"))
```
damping(values = c("damped", "none"))

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

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

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

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

**Arguments**

- **values**: A character string of possible values.
- **range**: A two-element vector holding the *defaults* for the smallest and largest possible values, respectively.
- **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**

```r
error()
trend()
season()
```
**get_arima_description**  
*Get model descriptions for Arima objects*

---

**Description**
Get model descriptions for Arima objects

**Usage**

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

**Arguments**

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

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

**Examples**

```r
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**

```r
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

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

# Model Specification ----
arima_spec <- arima_reg() %>%
    set_engine("auto_arima")

get_model_description(arima_spec, indicate_training = TRUE)

# Fitted Model ----
m750 <- m4_monthly %>% filter(id == "M750")
arima_fit <- arima_spec %>%
    fit(value ~ date, data = m750)

get_model_description(arima_fit, indicate_training = TRUE)
```

---

`get_tbats_description`  
*Get model descriptions for TBATS objects*

Description

Get model descriptions for TBATS objects

Usage

`get_tbats_description(object)`

Arguments

- `object`  
  Objects of class `tbats`

Source

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

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  

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  

Description  
Test if a table contains residuals.

Usage  
is_residuals(object)

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

load_namespace  

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
Description

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

Usage

extract_nested_test_accuracy(object)
extract_nested_test_forecast(object, .include_actual = TRUE, .id_subset = NULL)
extract_nested_error_report(object)
extract_nested_best_model_report(object)

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

extract_nested_modeltime_table(object, .row_id = 1)
extract_nested_train_split(object, .row_id = 1)
extract_nested_test_split(object, .row_id = 1)

Arguments

object A nested modeltime table
.include_actual Whether or not to include the actual data in the extracted forecast. Default: TRUE.
.id_subset Can supply a vector of id’s to extract forecasts for one or more id’s, rather than extracting all forecasts. If NULL, extracts forecasts for all id’s.
.row_id The row number to extract from the nested data.
m750

The 750th Monthly Time Series used in the M4 Competition

Description

The 750th Monthly Time Series used in the M4 Competition

Usage

m750

Format

A tibble with 306 rows and 3 variables:

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

Source

- M4 Competition Website

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)
m750_splits

Details

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

Examples

library(modeltime)
m750_models

\[
m750_splits \quad \text{The results of train/test splitting the M750 Data}\]

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

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

Examples

```r
library(rsample)
m750_training_resamples
```

maape

Mean Arctangent Absolute Percentage Error

Description

Useful when MAPE returns Inf typically due to intermittent data containing zeros. This is a wrapper to the function of TSrepr::maape().
maape.data.frame

Usage

maape(data, ...)

Arguments

data A data.frame containing the truth and estimate columns.
... Not currently in use.

maape.data.frame  Mean Arctangent Absolute Percentage Error

Description

This is basically a wrapper to the function of TSrepr::maape().

Usage

## S3 method for class 'data.frame'
maape(data, truth, estimate, na.rm = TRUE, ...)

Arguments

data A data.frame containing the truth and estimate columns.
truth The column identifier for the true results (that is numeric).
estimate The column identifier for the predicted results (that is also numeric).
na_rm Not in use...NA values managed by TSrepr::maape
... Not currently in use

maape_vec  Mean Arctangent Absolute Percentage Error

Description

This is basically a wrapper to the function of TSrepr::maape().

Usage

maape_vec(truth, estimate, na.rm = TRUE, ...)

Arguments

truth The column identifier for the true results (that is numeric).
estimate The column identifier for the predicted results (that is also numeric).
na_rm Not in use...NA values managed by TSrepr::maape
... Not currently in use
**Make Ts Splits**

Generate a Time Series Train/Test Split Indicies

**Description**

Makes fast train/test split indicies for time series.

**Usage**

```r
make_ts_splits(.data, .length_test, .length_train = NULL)
```

**Arguments**

- `.data` A data frame containing ordered time series data (ascending)
- `.length_test` The number of rows to include in the test set
- `.length_train` Optional. The number of rows to include in the training set. If NULL, returns all remaining row indicies.

**Value**

A list containing train_idx and test_idx

---

**Metric Sets**

Forecast Accuracy Metrics Sets

**Description**

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

**Usage**

```r
default_forecast_accuracy_metric_set(...)

extended_forecast_accuracy_metric_set(...)
```

**Arguments**

- `...` Add additional yardstick metrics
Default Forecast Accuracy Metric Set

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

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

Adding additional metrics is possible via . . .

Extended Forecast Accuracy Metric Set

Extends the default metric set by adding:

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

See Also

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

Examples

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

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

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

# Default Forecast Accuracy Metric Specification
default_forecast_accuracy_metric_set()

calc_default_metrics <- default_forecast_accuracy_metric_set()

calc_default_metrics(fake_data, y, yhat)

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

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

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

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

---

**modeltime_accuracy**  
*Calculate Accuracy Metrics*

**Description**

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

**Usage**

```r
modeltime_accuracy(
  object,
  new_data = NULL,
  metric_set = default_forecast_accuracy_metric_set(),
  acc_by_id = FALSE,
  quiet = TRUE,
  ...
)
```

**Arguments**

- **object**  
  A Modeltime Table

- **new_data**  
  A tibble to predict and calculate residuals on. If provided, overrides any calibration data.

- **metric_set**  
  A yardstick::metric_set() that is used to summarize one or more forecast accuracy (regression) metrics.

- **acc_by_id**  
  Should a global or local model accuracy be produced? (Default: FALSE)
  - When FALSE, a global model accuracy is provided.
  - If TRUE, a local accuracy is provided group-wise for each time series ID. To enable local accuracy, an id must be provided during modeltime_calibrate().

- **quiet**  
  Hide errors (TRUE, the default), or display them as they occur?

- **...**  
  If new_data is provided, these parameters are passed to modeltime_calibrate()
Details

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

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

Value

A tibble with accuracy estimates.

Examples

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

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

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

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

# MODELTME TABLE ----
models_tbl <- modeltime_table(
  model_fit_prophet
)

# ACCURACY ----
models_tbl %>%
  modeltime_calibrate(new_data = testing(splits)) %>%
  modeltime_accuracy(
    metric_set = metric_set(mae, rmse, rsq)
  )
```
**modeltime_calibrate**  
*Preparation for forecasting*

**Description**

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

**Usage**

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

**Arguments**

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

**Details**

The results of calibration are used for:

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

The calibration steps include:

1. If not a Modeltime Table, objects are converted to Modeltime Tables internally
2. Two Columns are added:
   - `.type`: Indicates the sample type. This is:
     - "Test" if predicted, or
     - "Fitted" if residuals were stored during modeling.
   - `.calibration_data`:
     - Contains a `tibble` with Timestamps, Actual Values, Predictions and Residuals calculated from `new_data` (Test Data)
     - If `id` is provided, will contain a 5th column that is the identifier variable.
Value

A Modeltime Table (mdl_time_tbl) with nested .calibration_data added

Examples

```r
library(tidyverse)
library(lubridate)
library(timetk)
library(parsnip)
library(rsample)
library(modeltime)

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

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

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

# ---- MODELTIME TABLE ----
models_tbl <- modeltime_table(model_fit_prophet)

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

# ---- ACCURACY ----
calibration_tbl %>%
  modeltime_accuracy()

# ---- FORECAST ----
calibration_tbl %>%
  modeltime_forecast(new_data = testing(splits),
                     actual_data = m750)
```
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
```r
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
```r
library(tidymodels)
library(modeltime)
library(workflowsets)
library(tidyverse)
library(lubridate)
library(timetk)

data_set <- m4_monthly

# SETUP WORKFLOWSETS
```
modeltime_forecast

Forecast future data

Description

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

Usage

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

Arguments

object A Modeltime Table
new_data A tibble containing future information to forecast. If NULL, forecasts the calibration data.

h The forecast horizon (can be used instead of new_data for time series with no exogenous regressors). Extends the calibration data h periods into the future.

actual_data Reference data that is combined with the output tibble and given a .key = "actual"

conf_interval An estimated confidence interval based on the calibration data. This is designed to estimate future confidence from out-of-sample prediction error.

conf_by_id Whether or not to produce confidence interval estimates by an ID feature.
  • When FALSE, a global model confidence interval is provided.
  • If TRUE, a local confidence interval is provided group-wise for each time series ID. To enable local confidence interval, an id must be provided during modeltime_calibrate().

keep_data Whether or not to keep the new_data and actual_data as extra columns in the results. This can be useful if there is an important feature in the new_data and actual_data needed when forecasting. Default: FALSE.

arrange_index Whether or not to sort the index in rowwise chronological order (oldest to newest) or to keep the original order of the data. Default: FALSE.

... Not currently used

Details

The modeltime_forecast() function prepares a forecast for visualization with 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 modetime 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 modetime_table() to modetime_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 Modetime 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 modetime_calibrate(). The out-of-sample error estimates are then carried through and applied to applied to any future forecasts.

The confidence interval can be adjusted with the conf_interval parameter. 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 affect 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, modetime_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 Modetime Table has been previously calibrated using modetime_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

```r
library(tidyverse)
library(lubridate)
library(timetk)
library(parsnip)
library(rsample)
library(modeltime)

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

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

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

# ---- MODELTIME TABLE ----
models_tbl <- modeltime_table(model_fit_prophet)

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

# ---- ACCURACY ----
calibration_tbl %>%
  modeltime_accuracy()

# ---- FUTURE FORECAST ----
```
modeltime_nested_fit

Fit Tidymodels Workflows to Nested Time Series

Description

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

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

Usage

modeltime_nested_fit(
  nested_data,
  ...
)
conf_interval = 0.95,  
control = control_nested_fit()  
)

Arguments

nested_data  Nested time series data

...  Tidymodels workflow objects that will be fit to the nested time series data.

model_list  Optionally, a list() of Tidymodels workflow objects can be provided

metric_set  A yardstick::metric_set() that is used to summarize one or more forecast accuracy (regression) metrics.

conf_interval  An estimated confidence interval based on the calibration data. This is designed to estimate future confidence from out-of-sample prediction error.

control  Used to control verbosity and parallel processing. See control_nested_fit().

Details

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

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

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

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

Description

Make a new forecast from a Nested Modeltime Table.
Usage

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

Arguments

- **object**: A Nested Modeltime Table
- **h**: The forecast horizon. Extends the "trained on" data "h" periods into the future.
- **include_actual**: Whether or not to include the ".actual_data" as part of the forecast. If FALSE, just returns the forecast predictions.
- **conf_interval**: An estimated confidence interval based on the calibration data. This is designed to estimate future confidence from **out-of-sample prediction error**.
- **id_subset**: A sequence of ID’s from the modeltime table to subset the forecasting process. This can speed forecasts up.
- **control**: Used to control verbosity and parallel processing. See `control_nested_forecast()`.

Details

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

**Logged Forecasts:**

The logged forecasts can be extracted using:

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

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

**Nested Forecasts:**

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

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

Use the id_subset to filter the Nested Modeltime Table object to just the time series of interest. Use the conf_interval to override the logged confidence interval. Note that this will have no effect if h = NULL as logged forecasts are returned. So be sure to provide h if you want to update the confidence interval.

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

modeltime_nested_refit

Refits a Nested Modeltime Table

Description

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

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

Usage

modeltime_nested_refit(object, control = control_nested_refit())

Arguments

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

modeltime_nested_select_best

Select the Best Models from Nested Modeltime Table

Description

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

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

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

Arguments

object A Nested Modeltime Table
metric A metric to minimize or maximize. By default available metrics are:
  • "rmse" (default)
  • "mae"
  • "mape"
  • "mase"
  • "smape"
  • "rsq"
minimize Whether to minimize or maximize. Default: TRUE (minimize).
filter_test_forecasts Whether or not to update the test forecast log to filter only the best forecasts.
  Default: TRUE.

modeltime_refit Refit one or more trained models to new data

Description

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

Usage

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

Arguments

object A Modeltime Table
data A tibble that contains data to retrain the model(s) using.
... Additional arguments to control refitting.
Ensemble Model Spec (modeltime.ensemble):
When making a meta-learner with modeltime.ensemble::ensemble_model_spec(),
used to pass resamples argument containing results from modeltime.resample::modeltime_fit_resamples()
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

```r
library(tidyverse)
library(lubridate)
library(timetk)
library(parsnip)
library(rsample)
library(modeltime)

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

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

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

# ---- MODELTIME TABLE ----
models_tbl <- modeltime_table(model_fit_prophet)
```
# ---- CALIBRATE ----
# - Calibrate on training data set

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

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

refit_tbl <- calibration_tbl %>%
  modeltime_refit(m750)

---

### `modeltime_residuals` 
*Extract Residuals Information*

**Description**

This is a convenience function to unnest model residuals

**Usage**

```r
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**

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

# ---- MODELTIME TABLE ----
models_tbl <- modeltime_table(
  model_fit_prophet
)

# ---- RESIDUALS ----

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

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

---

**modeltime_residuals_test**

*Apply Statistical Tests to Residuals*

**Description**

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

**Usage**

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

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

Details

**Shapiro-Wilk Test**

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

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

**Box-Pierce and Ljung-Box Tests Tests**

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

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

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

**Durbin-Watson Test**

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

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

**Value**

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

**See Also**

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

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

# ---- MODELTIME TABLE ----
models_tbl <- modeltime_table(model_fit_prophet)

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

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

---

**Description**

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

```r
modeltime_table(...)
```

```r
as_modeltime_table(.l)
```

**Arguments**

... Fitted `parsnip` model or `workflow` objects

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

**Details**

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

**as_modeltime_table()**:  
Converts a list of models to a modeltime table. Useful if programatically creating Modeltime Tables from models stored in a list.

**Examples**

```r
library(tidyverse)
library(lubridate)
library(timetk)
library(parsnip)
library(rsample)
library(modeltime)

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

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

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

# ---- MODELTIME TABLE ----
# Make a Modeltime Table
models_tbl <- modeltime_table(
```
naive_fit_impl

Low-Level NAIVE Forecast

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**  
*Bridge prediction function for NAIVE Models*

**Description**

Bridge prediction function for NAIVE Models

**Usage**

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

**Arguments**

- **object**  
  An object of class `model_fit`
- **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**

```r
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**

```r
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

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
Tuning Parameters for NNETAR Models

Usage

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

Arguments

range A two-element vector holding the defaults for the smallest and largest possible values, respectively.

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**

```r
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()`

**nnetar_reg**  
*General Interface for NNETAR Regression Models*

**Description**

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

**Usage**

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

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

An integer for the number of units in the hidden model.

Number of networks to fit with different random starting weights. These are then averaged when producing forecasts.

A non-negative numeric value for the amount of weight decay.

An integer for the number of training iterations.

Details

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

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

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

Main Arguments

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

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

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

Engine Details

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

- modeltime
- seasonal_period
- non_seasonal_ar
- seasonal_ar
- hidden_units
- num_networks
- epochs
- penalty

forecast::nnetar

ts(frequency)
p (1)
P (1)
size (10)
repeats (20)
maxit (100)
decay (0)

Other options can be set using set_engine().

nnetar

The engine uses forecast::nnetar().

Function Parameters:
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
**parallel_start**  
Start parallel clusters using the parallel package

**Description**  
Start parallel clusters using parallel package

**Usage**  

```r
parallel_start(..., .method = c("parallel", "spark"))
```

**Arguments**  

- `...` Parameters passed to underlying functions (See Details Section)
- `method` The method to create the parallel backend. Supports:
  - "parallel" - Uses the parallel and doParallel packages
  - "spark" - Uses the sparklyr package

**Examples**

```r
library(timetk)

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

Performs 3 Steps:

1. Makes clusters using parallel::makeCluster(...). The parallel_start(...) are passed to parallel::makeCluster(...).
2. Registers clusters using doParallel::registerDoParallel().
3. Adds .libPaths() using parallel::clusterCall().

Spark (.method = "spark")

- Important, make sure to create a spark connection using sparklyr::spark_connect().
- Pass the connection object as the first argument. For example, parallel_start(sc,.method = "spark").
- The parallel_start(...) are passed to sparklyr::registerDoSpark(...).

Examples

```r
# Starts 2 clusters
parallel_start(2)

# Returns to sequential processing
parallel_stop()
```

parse_index  

Developer Tools for parsing date and date-time information

Description

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

Usage

```r
parse_index_from_data(data)

parse_period_from_index(data, period)
```

Arguments

data  

A data frame

period  

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

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

```r
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

```r
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,
  ...
)
```
plot_modeltime_forecast

Arguments

- `.data` A tibble that is the output of `modeltime_forecast()`
- `.conf_interval_show` Logical. Whether or not to include the confidence interval as a ribbon.
- `.conf_interval_fill` Fill color for the confidence interval
- `.conf_interval_alpha` Fill opacity for the confidence interval. Range (0, 1).
- `.smooth` Logical - Whether or not to include a trendline smoother. Uses See `smooth_vec()` to apply a LOESS smoother.
- `.legend_show` Logical. Whether or not to show the legend. Can save space with long model descriptions.
- `.legend_max_width` Numeric. The width of truncation to apply to the legend text.
- `.title` Title for the plot
- `.x_lab` X-axis label for the plot
- `.y_lab` Y-axis label for the plot
- `.color_lab` Legend label if a `color_var` is used.
- `.interactive` Returns either a static (ggplot2) visualization or an interactive (plotly) visualization
- `.plotly_slider` If TRUE, returns a plotly date range slider.
- `...` Additional arguments passed to `timetk::plot_time_series()`.

Value

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

Examples

```r
library(tidyverse)
library(lubridate)
library(timetk)
library(parsnip)
library(rsample)
library(modeltime)

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

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

# --- MODELS ---

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

# ---- MODELTIME TABLE ----
models_tbl <- modeltime_table(
  model_fit_prophet
)

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

plot_modeltime_residuals

Interactive Residuals Visualization

Description

This is a wrapper for examining residuals using:

- Time Plot: 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**

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

... Additional arguments passed to:

- **Time Plot**: `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**

```r
library(tidyverse)
library(lubridate)
library(timetk)
library(parsnip)
library(rsample)
library(modeltime)

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

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

# --- MODELS ---

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

# ---- RESIDUALS ----
residuals_tbl <- models_tbl %>%
  modeltime_calibrate(new_data = testing(splits)) %>%
  modeltime_residuals()
residuals_tbl %>%
  plot_modeltime_residuals(
    .type = "timeplot",
    .interactive = FALSE
  )

pluck_modeltime_model  Extract model by model id in a Modeltime Table

Description
The pull_modeltime_model() and pluck_modeltime_model() functions are 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
  object      A Modeltime Table
  .model_id   A numeric value matching the .model_id that you want to update

See Also
- combine_modeltime_tables(): Combine 2 or more Modeltime Tables together
- add_modeltime_model(): Adds a new row with a new model to a Modeltime Table
- 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

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

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

### Arguments

- **object**
  - An object of class `model_fit`.

- **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`, `predict()` 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 `type = "raw"`. 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 `opts`). There are some `parsnip` related options that can be passed, depending on the value of `type`. Possible arguments are:
    - level: 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.
    - std_error: 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`.
    - quantile: the quantile(s) for quantile regression (not implemented yet)
    - time: 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.
Recursive Model Predictions

Description

Make predictions from a recursive model.

Usage

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

Arguments

- `object`: An object of class `model_fit`
- `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`, `predict()` 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 `type = "raw"`. 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 `opts`). There are some `parsnip` related options that can be passed, depending on the value of `type`. Possible arguments are:
  - `level`: 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.
  - `std_error`: 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.
  - `quantile`: the quantile(s) for quantile regression (not implemented yet)
  - `time`: 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
Prepared Nested Modeltime Data

Description

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

Usage

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

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

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

Arguments

- `.data` A data frame or tibble containing time series data. The data should have:
  - identifier `.id_var`: Identifying one or more time series groups
  - date variable `.date_var`: A date or date time column
  - target variable `.value`: A column containing numeric values that is to be forecasted
- `.id_var` An id column
- `.date_var` A date or datetime column
- `.length_future` Varies based on the function:
  - `extend_timeseries()`: Defines how far into the future to extend the time series by each time series group.
  - `nest_timeseries()`: Defines which observations should be split into the `.future_data`.
- `...` Additional arguments passed to the helper function. See details.
- `.length_actual` Can be used to slice the `.actual_data` to a most recent number of observations.
- `.length_test` Defines the length of the test split for evaluation.
- `.length_train` Defines the length of the training split for evaluation.

Details

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

**Step 1: Extend the Time Series:**

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

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

**Step 2: Nest the Time Series:**
nest_timeseries(): A helper for nesting your data into `.actual_data` and `.future_data`.
• The group column is specified by `.id_var`
• The `.length_future` defines the length of the `.future_data`.
• The remaining data is converted to the `.actual_data`.
• The `.length_actual` can be used to slice the `.actual_data` to a most recent number of observations.

The result is a "nested data frame".

**Step 3: Split the Actual Data into Train/Test Splits:**
split_nested_timeseries(): A wrapper for `timetk::time_series_split()` that generates training/testing splits from the `.actual_data` column.
• The `.length_test` is the primary argument that identifies the size of the testing sample. This is typically the same size as the `.future_data`.
• The `.length_train` is an optional size of the training data.
• The ... (dots) are additional arguments that can be passed to `timetk::time_series_split()`.

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

**Examples**

```r
library(tidyverse)
library(timetk)
library(modeltime)

nested_data_tbl <- walmart_sales_weekly %>%
  select(id, Date, Weekly_Sales) %>%
  set_names(c("id", "date", "value")) %>%

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

  # Step 2: Nests the time series into .actual_data and .future_data
  nest_timeseries(
    .id_var = id,
    .length_future = 52
  ) %>%
```
# Step 3: Adds a column .splits that contains training/testing indices

```r
split_nested_timeseries(
  .length_test = 52
)
```

```r
nested_data_tbl
```

# Helpers: Getting the Train/Test Sets

```r
evaluate_nested_train_split(nested_data_tbl, .row_id = 1)
```

---

### prophet_boost

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

**Description**

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

**Usage**

```r
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 (specific engines only)

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

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

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

learn_rate
A number for the rate at which the boosting algorithm adapts from iteration-to-iteration (specific engines only).

loss_reduction
A number for the reduction in the loss function required to split further (specific engines only).
sample_size number for the number (or proportion) of data that is exposed to the fitting routine.
stop_iter The number of iterations without improvement before stopping (xgboost only).

Details
The data given to the function are not saved and are only used to determine the mode of the model. For prophet_boost(), the mode will always be "regression".
The model can be created using the fit() function using the following engines:
- "prophet_xgboost" (default) - Connects to \texttt{prophet::prophet()} and \texttt{xgboost::xgb.train()}

Main Arguments
The main arguments (tuning parameters) for the \texttt{PROPHET} model are:
- \texttt{growth}: String 'linear' or 'logistic' to specify a linear or logistic trend.
- \texttt{changepoint_num}: Number of potential changepoints to include for modeling trend.
- \texttt{changepoint_range}: Range changepoints that adjusts how close to the end the last changepoint can be located.
- \texttt{season}: 'additive' (default) or 'multiplicative'.
- \texttt{prior_scale_changepoints}: Parameter modulating the flexibility of the automatic changepoint selection. Large values will allow many changepoints, small values will allow few changepoints.
- \texttt{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.
- \texttt{prior_scale_holidays}: Parameter modulating the strength of the holiday components model, unless overridden in the holidays input.
- \texttt{logistic_cap}: When growth is logistic, the upper-bound for "saturation".
- \texttt{logistic_floor}: When growth is logistic, the lower-bound for "saturation".

The main arguments (tuning parameters) for the model \texttt{XGBoost model} are:
- \texttt{mtry}: The number of predictors that will be randomly sampled at each split when creating the tree models.
- \texttt{trees}: The number of trees contained in the ensemble.
- \texttt{min_n}: The minimum number of data points in a node that are required for the node to be split further.
- \texttt{tree_depth}: The maximum depth of the tree (i.e. number of splits).
- \texttt{learn_rate}: The rate at which the boosting algorithm adapts from iteration-to-iteration.
- \texttt{loss_reduction}: The reduction in the loss function required to split further.
- \texttt{sample_size}: The amount of data exposed to the fitting routine.
- \texttt{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 \texttt{set_engine()} (See Engine Details below).
If parameters need to be modified, \texttt{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:

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

Model 2: XGBoost:

- `modetime` → `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):

```r
## 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 / modetime 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):**

```r
## 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)
prophet_fit_impl  

**Description**

Low-Level PROPHET function for translating modeltime to PROPHET

**Usage**

```r
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**

- **x**: A dataframe of xreg (exogenous regressors)
- **y**: A numeric vector of values to fit
- **growth**: String 'linear', 'logistic', or 'flat' to specify a linear, logistic or flat trend.
- **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.

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**

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

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

changepoint_range(range = c(0.6, 0.9), trans = NULL)
```
seasonality_yearly(values = c(TRUE, FALSE))
seasonality_weekly(values = c(TRUE, FALSE))
seasonality_daily(values = c(TRUE, FALSE))
prior_scale_changepoints(range = c(-3, 2), trans = log10_trans())
prior_scale_seasonality(range = c(-3, 2), trans = log10_trans())
prior_scale_holidays(range = c(-3, 2), trans = log10_trans())

Arguments

values       A character string of possible values.
range        A two-element vector holding the defaults for the smallest and largest possible values, respectively.
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 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 model_fit
new_data A rectangular data object, such as a data frame.
... Additional arguments passed to prophet::predict()

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,
prophet_reg

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

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

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

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

Engine Details

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

<table>
<thead>
<tr>
<th>modeltime</th>
<th>prophet</th>
</tr>
</thead>
<tbody>
<tr>
<td>growth</td>
<td>growth ('linear')</td>
</tr>
<tr>
<td>changepoint_num</td>
<td>n.changepoints (25)</td>
</tr>
<tr>
<td>changepoint_range</td>
<td>changepoints.range (0.8)</td>
</tr>
<tr>
<td>seasonality_yearly</td>
<td>yearly.seasonality ('auto')</td>
</tr>
<tr>
<td>seasonality_weekly</td>
<td>weekly.seasonality ('auto')</td>
</tr>
<tr>
<td>seasonality_daily</td>
<td>daily.seasonality ('auto')</td>
</tr>
<tr>
<td>season</td>
<td>seasonality.mode ('additive')</td>
</tr>
<tr>
<td>prior_scale_changepoints</td>
<td>changepoint.prior.scale (0.05)</td>
</tr>
<tr>
<td>prior_scale_seasonality</td>
<td>seasonality.prior.scale (10)</td>
</tr>
</tbody>
</table>
Other options can be set using `set_engine()`.

**prophet**

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

Function Parameters:

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

Parameter Notes:

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

Regressors:

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

Logistic Growth and Saturation Levels:

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

Limitations:

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

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

- fit(y ~ date)

Univariate (No Extra Regressors):
For univariate analysis, you must include a date or date-time feature. Simply use:

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

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

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

Xreg Example: Suppose you have 3 features:
1. y (target)
2. date (time stamp),
3. month.lbl (labeled month as a ordered factor).

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

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

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

See Also

- fit.model_spec(), set_engine()

Examples

library(dplyr)
library(parsnip)
library(rsample)
library(timetk)
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**

```r
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 season-
   ality. 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 change-
   points.
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 num-
   ber 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 ex-
   trapolated 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 calcula-
   tion.
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 number of training set samples use for these
   purposes.
**prophet_xgboost_predict_impl**

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 based 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 model_fit
- **new_data**
  - A rectangular data object, such as a data frame.
- **...**
  - Additional arguments passed to prophet::predict()

---

**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

```r
pull_parsnip_preprocessor(object)
```

### Arguments

- **object**: A fitted parsnip model `model_fit` object

### Value

A formula using `stats::formula()`

---

## recipe_helpers

*Developer Tools for processing XREGS (Regressors)*

### Description

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

### Usage

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

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

### Arguments

- **recipe**: A prepared recipe
- **format**: One of:
  - `tbl`: Returns a tibble (data.frame)
  - `matrix`: Returns a matrix
- **new_data**: Data to be processed by a recipe

### Value

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

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 model_fit or a fitted workflow class
transform A transformation performed on new_data after each step of recursive algorithm.
  • Transformation Function: Must have one argument data (see examples)
train_tail A tibble with tail of training data set. In most cases it'll be required to create some variables based on dependent variable.
id (Optional) An identifier that can be provided to perform a panel forecast. A single quoted column name (e.g. id = “id”).
... 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 where 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

```r
# Libraries & Setup ----
library(modeltime)
library(tidymodels)
library(tidyverse)
library(lubridate)
library(timetk)
library(sluder)
```
# ---- 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") %>%
  modeltimetime_forecast(
    new_data = future_data,
    actual_data = m750
  ) %>%
  plot_modeltimetime_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
seasonal_reg

General Interface for Multiple Seasonality Regression Models (TBATS, STLM)

Description
seasonal_reg() is a way to generate a specification of an Seasonal Decomposition model before fitting and allows the model to be created using different packages. Currently the only package is forecast.

Usage
seasonal_reg(
  mode = "regression",
  seasonal_period_1 = NULL,
  seasonal_period_2 = NULL,
  seasonal_period_3 = NULL
)

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

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
)

plot_modeltime_forecast(
  .interactive = FALSE,
  .conf_interval_show = FALSE
)
seasonal_period_1
(required) The primary seasonal frequency. Uses "auto" by default. A character phrase of "auto" or time-based phrase of "2 weeks" can be used if a date or date-time variable is provided. See Fit Details below.

seasonal_period_2
(optional) A second seasonal frequency. Is NULL by default. A character phrase of "auto" or time-based phrase of "2 weeks" can be used if a date or date-time variable is provided. See Fit Details below.

seasonal_period_3
(optional) A third seasonal frequency. Is NULL by default. A character phrase of "auto" or time-based phrase of "2 weeks" can be used if a date or date-time variable is provided. See Fit Details below.

Details

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

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

- "tbats" - Connects to forecast::tbats()
- "stlm_ets" - Connects to forecast::stlm(), method = "ets"
- "stlm_arima" - Connects to forecast::stlm(), method = "arima"

Engine Details

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

<table>
<thead>
<tr>
<th>modeltime</th>
<th>forecast::stlm</th>
<th>forecast::tbats</th>
</tr>
</thead>
<tbody>
<tr>
<td>seasonal_period_1, seasonal_period_2, seasonal_period_3</td>
<td>msts(seasonal.periods)</td>
<td>msts(seasonal.periods)</td>
</tr>
</tbody>
</table>

Other options can be set using set_engine().

The engines use forecast::stlm().

Function Parameters:

```r
## 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**

```r
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
```
smooth_fit_impl

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

**Description**

Low-Level Exponential Smoothing function for translating modeltime to forecast

**Usage**

```r
smooth_fit_impl(
  x,
  y,
  period = "auto",
  error = "auto",
  trend = "auto",
  season = "auto",
  damping = NULL,
  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 `smooth::es`
smooth_predict_impl  

Bridge prediction function for Exponential Smoothing models

Description

Bridge prediction function for Exponential Smoothing models

Usage

smooth_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 smooth::es()

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_arima_fit_impl  
Low-Level stlm function for translating modeltime to forecast

Description

Low-Level stlm function for translating modeltime to forecast

Usage

stlm_arima_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.
stlm_ets_fit_impl

(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 forecast::stlm()

stlm_arima_predict_impl

_Bridge prediction function for ARIMA models_

Description

Bridge prediction function for ARIMA models

Usage

stlm_arima_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()

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,
  ...
)

...
**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 `model_fit`
- `new_data`  
  A rectangular data object, such as a data frame.
- `...`  
  Additional arguments passed to `forecast::stlm()`

---

**summarize_accuracy_metrics**  
Summarize Accuracy Metrics

**Description**

This is an internal function used by `modeltime_accuracy()`.

**Usage**

```
summarize_accuracy_metrics(data, truth, estimate, metric_set)`
Arguments

data A data.frame containing the truth and estimate columns.
truth The column identifier for the true results (that is numeric).
estimate The column identifier for the predicted results (that is also numeric).
metric_set A yardstick::metric_set() 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()
)

package: yardstick

	able_modeltime_accuracy

Interactive Accuracy Tables

Description

Converts results from modeltime_accuracy() into either interactive (reactable) or static (gt) tables.

Usage

table_modeltime_accuracy(
  .data, 
  .round_digits = 2, 
  .sortable = TRUE, 
  .show_sortable = TRUE, 
  .searchable = TRUE, 
  .filterable = FALSE,
### table_modeltime_accuracy

```r
.data, .round_digits, .sortable, .show_sortable, .searchable, .filterable, .expand_groups, .title = "Accuracy Table", .interactive = TRUE,
...
```

#### Arguments

- **.data**
  A tibble that is the output of `modeltime_accuracy()`

- **.round_digits**
  Rounds accuracy metrics to a specified number of digits. If `NULL`, rounding is not performed.

- **.sortable**
  Allows sorting by columns. Only applied to `reactable` tables. Passed to `reactable(sortable)`.

- **.show_sortable**
  Shows sorting. Only applied to `reactable` tables. Passed to `reactable(showSortable)`.

- **.searchable**
  Adds search input. Only applied to `reactable` tables. Passed to `reactable(searchable)`.

- **.filterable**
  Adds filters to table columns. Only applied to `reactable` tables. Passed to `reactable(filterable)`.

- **.expand_groups**
  Expands groups dropdowns. Only applied to `reactable` tables. Passed to `reactable(defaultExpanded)`.

- **.title**
  A title for static (gt) tables.

- **.interactive**
  Return interactive or static tables. If `TRUE`, returns `reactable` table. If `FALSE`, returns static `gt` table.

- **...**
  Additional arguments passed to `reactable::reactable()` or `gt::gt()` (depending on `.interactive` selection).

#### Details

**Groups**

The function respects `dplyr::group_by()` groups and thus scales with multiple groups.

**Reactable Output**

A `reactable()` table is an interactive format that enables live searching and sorting. When `.interactive` = `TRUE`, a call is made to `reactable::reactable()`.

`table_modeltime_accuracy()` includes several common options like toggles for sorting and searching. Additional arguments can be passed to `reactable::reactable()` via `...`.

**GT Output**

A `gt` table is an HTML-based table that is "static" (e.g. non-searchable, non-sortable). It’s commonly used in PDF and Word documents that does not support interactive content.

When `.interactive` = `FALSE`, a call is made to `gt::gt()`. Arguments can be passed via `...`.

Table customization is implemented using a piping workflow (`%>%`). For more information, refer to the [GT Documentation](#).

#### Value

A static `gt` table or an interactive `reactable` table containing the accuracy information.
Examples

```r
library(tidyverse)
library(lubridate)
library(timetk)
library(parsnip)
library(rsample)
library(modeltime)

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

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

# --- MODELS ---

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

# ---- MODELTIME TABLE ----
models_tbl <- modeltime_table(model_fit_prophet)

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

tbats_fit_impl

**Low-Level tbats function for translating modeltime to forecast**

**Description**

Low-Level tbats function for translating modeltime to forecast

**Usage**

```r
.tbats_fit_impl(
  x,
  y,
```
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.

**use.parallel**  
TRUE/FALSE indicates whether or not to use parallel processing.

...  
Additional arguments passed to forecast::tbats()
**temporal_hierarchy**  
*General Interface for Temporal Hierarchical Forecasting (THIEF)*  
*Models*

**Description**

temporal_hierarchy() is a way to generate a *specification* of an Temporal Hierarchical Forecasting model before fitting and allows the model to be created using different packages. Currently the only package is *thief*. Note this function requires the thief package to be installed.

**Usage**

temporal_hierarchy(
  mode = "regression",
  seasonal_period = NULL,
  combination_method = NULL,
  use_model = NULL
)

**Arguments**

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

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

- **combination_method**  
  Combination method of temporal hierarchies, taking one of the following values:
  - "struc" - Structural scaling: weights from temporal hierarchy
  - "mse" - Variance scaling: weights from in-sample MSE
  - "ols" - Unscaled OLS combination weights
  - "bu" - Bottom-up combination – i.e., all aggregate forecasts are ignored.
  - "shr" - GLS using a shrinkage (to block diagonal) estimate of residuals
  - "sam" - GLS using sample covariance matrix of residuals

- **use_model**  
  Model used for forecasting each aggregation level:
  - "ets" - exponential smoothing
  - "arima" - arima
  - "theta" - theta
  - "naive" - random walk forecasts
  - "snaive" - seasonal naive forecasts, based on the last year of observed data
Details

Models can be created using the following engines:

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

Engine Details

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

```
modeltime                  thief::thief()
combination_method         comb
use_model                  usemodel
```

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

**thief (default engine)**

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

Function Parameters:

```r
## function (y, m = frequency(y), h = m * 2, comb = c("struc", "mse", "ols",
## "bu", "shr", "sam"), usemodel = c("ets", "arima", "theta", "naive",
## "snaive"), forecastfunction = NULL, aggregatelist = NULL, ...)
```

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

Parameter Notes:

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

Fit Details

**Date and Date-Time Variable**

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

- `fit(y ~ date)`

**Univariate:**

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

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

**Multivariate (xregs, Exogenous Regressors)**

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


See Also

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

Examples

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

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

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

# ---- HIERARCHICAL ----

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

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

Description

Tuning Parameters for TEMPORAL HIERARCHICAL Models

Usage

combination_method()

use_model()

Details

The main parameters for Temporal Hierarchical models are:

- combination_method: Combination method of temporal hierarchies.
- use_model: Model used for forecasting each aggregation level.

Examples

combination_method()

use_model()


__________________________________________________________

temporal_hier_fit_impl

Low-Level Temporaral Hierarchical function for translating model-time to forecast

__________________________________________________________

Description

Low-Level Temporaral Hierarchical function for translating model-time to forecast

Usage

temporal_hier_fit_impl(
    x,
    y,
    period = "auto",
    comb = c("struc", "mse", "ols", "bu", "shr", "sam"),
    usemodel = c("ets", "arima", "theta", "naive", "snaive"),
    ...
)
### theta_fit_impl

**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.
- `comb`: Combination method of temporal hierarchies
- `usemodel`: Model used for forecasting each aggregation level
- `...`: Additional arguments passed to `forecast::ets`

### temporal_hier_predict_impl

**Description**

Bridge prediction function for TEMPORAL HIERARCHICAL models

**Usage**

```
 temporal_hier_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()`

### theta_fit_impl

**Description**

Low-Level Exponential Smoothing function for translating modeltime to forecast

**Usage**

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

**Arguments**

- `x`: A dataframe of xreg (exogenous regressors)
- `y`: A numeric vector of values to fit
- `...`: Additional arguments passed to `forecast::ets`
Bridge prediction function for THETA models

Usage

theta_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()

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'

```r
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

```r
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_ensemble` 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

```r
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**

```r
update_model_description(object, .model_id, .new_model_desc)
update_modeltime_description(object, .model_id, .new_model_desc)
```

**Arguments**

- `object` A Modeltime Table
- `.model_id` A numeric value matching the `.model_id` that you want to update
- `.new_model_desc` Text describing the new model description
See Also

- `combine_modetime_tables()`: Combine 2 or more Modetime Tables together
- `add_modetime_model()`: Adds a new row with a new model to a Modetime Table
- `update_modetime_description()`: Updates a description for a model inside a Modetime Table
- `update_modetime_model()`: Updates a model inside a Modetime Table
- `pull_modetime_model()`: Extracts a model from a Modetime Table

Examples

```r
m750_models %>%
  update_modetime_description(2, "PROPHET - No Regressors")
```

---

`window_function_fit_impl`

Low-Level Window Forecast

Description

Low-Level Window Forecast

Usage

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

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.
- `window_size`: The period to apply the window function to
- `window_function`: A function to apply to the window. The default is `mean()`.
- `...`: Additional arguments for the `window_function`. For example, it's common to pass `na.rm = TRUE` 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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>object</td>
<td>An object of class model_fit</td>
</tr>
<tr>
<td>new_data</td>
<td>A rectangular data object, such as a data frame.</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mode</td>
<td>A single character string for the type of model. The only possible value for this model is &quot;regression&quot;.</td>
</tr>
<tr>
<td>id</td>
<td>An optional quoted column name (e.g. &quot;id&quot;) for identifying multiple time series (i.e. panel data).</td>
</tr>
<tr>
<td>window_size</td>
<td>A window to apply the window function. By default, the window uses the full data set, which is rarely the best choice.</td>
</tr>
</tbody>
</table>

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**

```r
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
) %>%
  set_engine(
    engine = "window_function",
    window_function = median,
    na.rm = TRUE

define two words: "window_function", "median"
# Fit Spec
model_fit <- model_spec %>%
  fit(log(value) ~ date, data = training(splits))
model_fit

# Predict
# - The 12-month median repeats going forward
predict(model_fit, testing(splits))

# ---- PANEL FORECAST - WINDOW FUNCTION ----

# Weighted Average Forecast
model_spec <- window_reg(
  # Specify the ID column for Panel Data
  id = "id",
  window_size = 12
) %>%
  set_engine(
    engine = "window_function",
    # Create a Weighted Average
    window_function = ~ sum(tail(.x, 3) * c(0.1, 0.3, 0.6)),
  )

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

# Predict: The weighted average (scalar) repeats going forward
predict(model_fit, testing(splits))

# ---- BROADCASTING PANELS (REPEATING) ----

# Simulating a Seasonal Naive Forecast by
# broadcasted model the last 12 observations into the future
model_spec <- window_reg(
  id = "id",
  window_size = Inf
) %>%
  set_engine(
    engine = "window_function",
    window_function = ~ tail(.x, 12),
  )

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

# Predict: The sequence is broadcasted (repeated) during prediction
predict(model_fit, testing(splits))

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

x A data frame or matrix of predictors
y A vector (factor or numeric) or matrix (numeric) of outcome data.
max_depth An integer for the maximum depth of the tree.
nrounds An integer for the number of boosting iterations.
etta A numeric value between zero and one to control the learning rate.
colsample_bynode Subsampling proportion of columns for each node within each tree. See the counts argument below. The default uses all columns.
colsample_bytree Subsampling proportion of columns for 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. 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 \textit{number} 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 \texttt{xgboost} uses to create trees. See \texttt{xgboost::xgb.train()} for options. If left NULL, an appropriate loss function is chosen.

counts
A logical. If FALSE, \texttt{colsample_bynode} and \texttt{colsample_bytree} are both assumed to be \textit{proportions} 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 \texttt{xgb.train}.

\begin{verbatim}
xgboost_predict   Wrapper for xgboost::predict
\end{verbatim}

\section*{Description}
Wrapper for \texttt{xgboost::predict}

\section*{Usage}
\texttt{xgboost_predict(object, newdata, ...)}

\section*{Arguments}
\begin{itemize}
  \item \texttt{object} a model object for which prediction is desired.
  \item \texttt{newdata} New data to be predicted
  \item \texttt{...} additional arguments affecting the predictions produced.
\end{itemize}
Index

* datasets
  m750, 58
  m750_models, 58
  m750_splits, 59
  m750_training_resamples, 60
  prepare_panel_transform (.prepare_transform), 4
  prepare_transform, 4
  adam_fit_impl, 5
  adam_params, 6
  Adam_predict_impl, 8
  adam_reg, 8
  add_modeltime_model, 14
  add_modeltime_model(), 14, 37, 101, 149, 150
  arima_boost, 15
  Arima_fit_impl, 20
  arima_params, 21
  Arima_predict_impl, 22
  arima_reg, 23
  arima_xgboost_fit_impl, 27
  arima_xgboost_predict_impl, 30
  as_modeltime_table (modeltime_table), 82
  auto_adam_fit_impl, 30
  Auto_adam_predict_impl, 32
  auto_arima_fit_impl, 32
  auto_arima_xgboost_fit_impl, 33
  bake_xreg_recipe (recipe_helpers), 124
  changepoint_num (prophet_params), 113
  changepoint_range (prophet_params), 113
  combination_method (temporal_hierarchy_params), 144
  combine_modeltime_tables, 36
  combine_modeltime_tables(), 14, 37, 101, 149, 150
  control_fit_workflowset (control_modeltime), 38
  control_fit_workflowset(), 68
  control_modeltime, 38
  control_nested_fit (control_modeltime), 38
  control_nested_fit(), 74
  control_nested_forecast (control_modeltime), 38
  control_nested_forecast(), 75
  control_nested_refit (control_modeltime), 38
  control_nested_refit(), 76
  control_refit (control_modeltime), 38
  control_refit(), 77, 78
  create_model_grid, 40
  create_xreg_recipe, 41
  croston_fit_impl, 43
  croston_predict_impl, 43
  damping (exp_smoothing_params), 51
  damping_smooth (exp_smoothing_params), 51
  default_forecast_accuracy_metric_set (metric_sets), 62
  default_forecast_accuracy_metric_set(), 65
  dials::epochs(), 90
  dials::grid_regular(), 40
  dials::hidden_units(), 90
  dials::penalty(), 90
  distribution (adam_params), 6
  error (exp_smoothing_params), 51
  ets_fit_impl, 44
  ets_predict_impl, 45
  exp_smoothing, 45
  exp_smoothing_params, 51
  extend_timeseries (prep_nested), 104
  extend_timeseries(), 74
extended_forecast_accuracy_metric_set (metric_sets), 62
extract_nested_best_model_report (log_extractors), 57
extract_nested_best_model_report(), 76
extract_nested_error_report (log_extractors), 57
extract_nested_error_report(), 73, 76
extract_nested_future_forecast (log_extractors), 57
extract_nested_future_forecast(), 75, 76
extract_nested_modeltime_table (log_extractors), 57
extract_nested_test_accuracy (log_extractors), 57
extract_nested_test_accuracy(), 73
extract_nested_test_forecast (log_extractors), 57
extract_nested_test_forecast(), 73, 75, 76
extract_nested_test_split (log_extractors), 57
extract_nested_test_split(), 105
extract_nested_train_split (log_extractors), 57
extract_nested_train_split(), 105
fit.model_spec(), 13, 19, 26, 49, 66, 87, 94, 111, 119, 132, 144, 153
fit.workflow(), 66
forecast::Arima(), 16, 24, 25
forecast::auto.arima(), 11, 16, 24, 25
forecast::croston(), 46, 47
forecast::ets(), 46, 47
forecast::msts(), 147
forecast::nnetar(), 92
forecast::thetaf(), 46, 47
get_arima_description, 53
get_model_description, 53
growth (prophet_params), 113
gt::gt(), 139
information_criteria (adam_params), 6
is_calibrated, 55
is_modeltime_model, 55
is_modeltime_table, 55
is_residuals, 56
juice_xreg_recipe (recipe_helpers), 124
load_namespace, 56
log_extractors, 57
m750, 58
m750_models, 58
m750_splits, 59
m750_training_resamples, 60
maape, 60
maape(), 63
maape.data.frame, 61
maape_vec, 61
mae(), 63, 65
make_ts_splits, 62
mape(), 63, 65
mase(), 63, 65
metric_set(), 62
metric_sets, 62
modeltime_accuracy, 64
modeltime_accuracy(), 62, 63, 66, 138, 139
modeltime_calibrate, 66
modeltime_calibrate(), 37, 70, 71
modeltime_fit_workflowset, 68
modeltime_fit_workflowset(), 38, 40
modeltime_forecast, 69
modeltime_forecast(), 66, 98
modeltime_nested_fit, 73
modeltime_nested_fit(), 38
modeltime_nested_forecast, 74
modeltime_nested_forecast(), 38
modeltime_nested_refit, 76
modeltime_nested_refit(), 38
modeltime_nested_select_best, 76
modeltime_refit, 77
modeltime_refit(), 37, 38, 70
modeltime_residuals, 79
modeltime_residuals(), 100
modeltime_residuals_test, 80
modeltime_table, 82
modeltime_table(), 66
naive_fit_impl, 84
naive_fit_impl(), 86
naive_predict_impl, 85
naive_reg, 85
ndiffs, 35
nest_timeseries (prep_nested), 104
nest_timeseries(), 74
new_modeltime_bridge, 88
nnetar_fit_impl, 89
nnetar_params, 90
nnetar_predict_impl, 91
nnetar_reg, 91
non_seasonal_ar (arima_params), 21	non_seasonal_ar(), 90	non_seasonal_differences (arima_params), 21	non_seasonal_ma (arima_params), 21
nsdiffs, 35
num_networks (nnetar_params), 90
outliers_treatment (adam_params), 6
panel_tail, 94
panel_tail(), 126
parallel_start, 95
parallel_start(), 39
parallel_stop (parallel_start), 95
parse_index, 96
parse_index_from_data (parse_index), 96
parse_period_from_index (parse_index), 96
plot_acf_diagnostics(), 99, 100
plot_modeltime_forecast, 97
plot_modeltime_forecast(), 70
plot_modeltime_residuals, 99
plot_seasonal_diagnostics(), 99, 100
plot_time_series(), 97, 99, 100
pluck_modeltime_model, 101
predict.recursive, 102
predict.recursive_panel, 103
prep_nested, 104
prior_scale_changepoints (prophet_params), 113
prior_scale_holidays (prophet_params), 113
prior_scale_seasonality (prophet_params), 113
probability_model (adam_params), 6
prophet::prophet(), 108, 117, 118
prophet_boost, 106
prophet_fit_impl, 112
prophet_params, 113
prophet_predict_impl, 115
prophet_reg, 115
prophet_xgboost_fit_impl, 120
prophet_xgboost_predict_impl, 123
pull_modeltime_model (pluck_modeltime_model), 101
pull_modeltime_model(), 14, 37, 101, 149, 150
pull_modeltime_residuals, 123
pull_parsnip_preprocessor, 124
reactable::reactable(), 139
recipe_helpers, 124
recursive, 125
recursive(), 95, 102, 103
regressors_treatment (adam_params), 6
rmse(), 63, 65
rsq(), 63, 65
season (exp_smoothing_params), 51
season(), 114
seasonal_ar (arima_params), 21
seasonal_ar(), 90
seasonal_differences (arima_params), 21
seasonal_ma (arima_params), 21
seasonal_period (time_series_params), 147
seasonal_reg, 129
seasonality_daily (prophet_params), 113
seasonality_weekly (prophet_params), 113
seasonality_yearly (prophet_params), 113
select_order (adam_params), 6
set_engine(), 13, 19, 26, 49, 87, 94, 111, 119, 132, 144, 153
smape(), 63, 65
smooth::adam(), 10–12
smooth::auto.adam(), 10, 11
smooth::es(), 46, 48
smooth_fit_impl, 133
smooth_level (exp_smoothing_params), 51
smooth_predict_impl, 134
smooth_seasonal (exp_smoothing_params), 51
smooth_trend (exp_smoothing_params), 51
smooth_vec(), 98, 100
snaive_fit_impl, 134
snaive_fit_impl(), 86
snaive_predict_impl, 135
split_nested_timeseries (prep_nested), 104
split_nested_timeseries(), 74