Package ‘healthyR.ts’

August 23, 2021

**Title**  The Time Series Modeling Companion to 'healthyR'

**Version**  0.1.3

**Description**  Hospital time series data analysis workflow tools, modeling, and automations. This library provides many useful tools to review common administrative time series hospital data. Some of these include average length of stay, and readmission rates. The aim is to provide a simple and consistent verb framework that takes the guesswork out of everything.

**License**  MIT + file LICENSE

**Encoding**  UTF-8

**RoxygenNote**  7.1.1

**URL**  https://github.com/spsanderson/healthyR.ts

**BugReports**  https://github.com/spsanderson/healthyR.ts/issues

**Imports**  magrittr, rlang (>= 0.1.2), tibble, timetk, tidyr, tidyquant, dplyr, purrr, ggplot2, lubridate, plotly, recipes, modelltime, cowplot, graphics, forcats, stringi, parsnip, workflowsets, earth

**Suggests**  knitr, rmarkdown, roxygen2, scales, rsample, healthyR.data, stringr, forecast, tidymodels, tidyverse, cli, crayon, glue, rstudioapi, xts, zoo

**VignetteBuilder**  knitr

**Depends**  R (>= 2.10)

**NeedsCompilation**  no

**Author**  Steven Sanderson [aut, cre],
Steven Sanderson [cph]

**Maintainer**  Steven Sanderson <spsanderson@gmail.com>

**Repository**  CRAN

**Date/Publication**  2021-08-23 07:11:31 UTC
**R topics documented:**

- `calibrate_and_plot` .................................................. 2
- `model_extraction_helper` .............................................. 4
- `ts_auto_recipe` ...................................................... 5
- `ts_calendar_heatmap_plot` .......................................... 7
- `ts_compare_data` .................................................... 10
- `ts_forecast_simulator` ................................................ 11
- `ts_ma_plot` .......................................................... 13
- `ts_qc_run_chart` ..................................................... 15
- `ts_random_walk` ..................................................... 17
- `ts_random_walk_ggplot_layers` .................................... 18
- `ts_splits_plot` ....................................................... 19
- `ts_wfs_lin_reg` ..................................................... 21
- `ts_wfs_mars` .......................................................... 23
- `ts_wfs_svm_poly` ..................................................... 25
- `ts_wfs_svm_rbf` ...................................................... 27

**Index**

| calibrate_and_plot | Helper function - Calibrate and Plot |

**Description**

This function is a helper function. It will take in a set of workflows and then perform the `modeltime::modeltime_calibrate()` and `modeltime::plot_modeltime_forecast()`.

**Usage**

```r
calibrate_and_plot(
  ..., .type = "testing",
  .splits_obj, .data,
  .print_info = TRUE,
  .interactive = FALSE
)
```

**Arguments**

- `...` The workflow(s) you want to add to the function.
- `type` Either the training(splits) or testing(splits) data.
- `splits_obj` The splits object.
- `data` The full data set.
- `print_info` The default is TRUE and will print out the calibration accuracy tibble and the resulting plotly plot.
The defaults is FALSE. This controls if a forecast plot is interactive or not via plotly.

Details

This function expects to take in workflows fitted with training data.

Value

The original time series, the simulated values and a some plots

Author(s)

Steven P. Sanderson II, MPH

Examples

```r
## Not run:
suppressPackageStartupMessages(library(timetk))
suppressPackageStartupMessages(library(dplyr))
suppressPackageStartupMessages(library(healthyR.data))
suppressPackageStartupMessages(library(tidymodels))

data <- healthyR_data %>%
  filter(ip_op_flag == "I") %>%
  select(visit_end_date_time) %>%
  rename(date_col = visit_end_date_time) %>%
timetk::filter_by_time(
  .date_var = date_col,
  .start_date = "2015",
  .end_date = "2019"
) %>%
timetk::summarise_by_time(
  .date_var = date_col,
  .by = "month",
  value = n()
)

splits <- timetk::time_series_split(
  data,
  date_col,
  assess = 12,
  skip = 3,
  cumulative = TRUE
)

rec_obj <- recipe(value ~ ., data = training(splits))

model_spec <- linear_reg(
  mode = "regression",
  penalty = 0.1,
  mixture = 0.5
)```
```r
wflw <- workflow() %>%
  add_recipe(rec_obj) %>%
  add_model(model_spec) %>%
  fit(training(splits))

output <- calibrate_and_plot(
  wflw,
  .type = "training",
  .splits_obj = splits,
  .data = data,
  .print_info = FALSE,
  .interactive = FALSE
)

## End(Not run)
```

---

**model_extraction_helper**

*Model Method Extraction Helper*

**Description**

This takes in a model fit and returns the method of the fit object.

**Usage**

```r
model_extraction_helper(.fit_object)
```

**Arguments**

- `.fit_object` A time-series fitted model

**Details**

Currently supports forecasting model of one of the following from the `forecast` package:

- `Arima`
- `auto.arima`
- `ets`
- `nnetar`

**Value**

A model description
ts_auto_recipe

Build a Time Series Recipe

Description

Automatically builds generic time series recipe objects from a given tibble.

Usage

    ts_auto_recipe(
        .data, 
        .date_col,


```r
.
.pred_col,
.step_ts_sig = TRUE,
.step_ts_rm_misc = TRUE,
.step_ts_dummy = TRUE,
.step_ts_fourier = TRUE,
.step_ts_fourier_period = NULL,
.K = 1,
.step_ts_yeo = TRUE,
.step_ts_nzv = TRUE
)
```

### Arguments

- `.data` The data that is going to be modeled. You must supply a tibble.

- `.date_col` The column that holds the date for the time series.

- `.pred_col` The column that is to be predicted.

- `.step_ts_sig` A Boolean indicating should the `timetk::step_timeseries_signature()` be added, default is TRUE.

- `.step_ts_rm_misc` A Boolean indicating should the following items be removed from the time series signature, default is TRUE.
  - iso$
  - xts$
  - hour
  - min
  - sec
  - am.pm

- `.step_ts_dummy` A Boolean indicating if all_nominal_predictors() should be dummied and with one hot encoding.

- `.step_ts_fourier` A Boolean indicating if `timetk::step_fourier()` should be added to the recipe.

- `.step_ts_fourier_period` A number such as 365/12, 365/4 or 365 indicting the period of the fourier term. The numeric period for the oscillation frequency.

- `.K` The number of orders to include for each sine/cosine fourier series. More orders increase the number of fourier terms and therefore the variance of the fitted model at the expense of bias. See details for examples of K specification.

- `.step_ts_yeo` A Boolean indicating if the `recipes::step_YeoJohnson()` should be added to the recipe.

- `.step_ts_nzv` A Boolean indicating if the `recipes::step_nzv()` should be run on all predictors.

### Details

This will build out a couple of generic recipe objects and return those items in a list.
Author(s)

Steven P. Sanderson II, MPH

Examples

```r
library(healthyR.data)
library(timetk)
library(healthyR.ts)
library(recipes)
library(dplyr)
library(rsample)

data_tbl <- healthyR_data %>%
  filter_by_time(.date_var = visit_end_date_time,
                 .start_date = "2012",
                 .end_date = "2020")
  %>%
  filter(payer_grouping != "?")
  %>%
  select(visit_end_date_time, ip_op_flag)
  %>%
  summarise_by_time(.date_var = visit_end_date_time,
                   .by = "week",
                   value = n())

splits <- rsample::initial_time_split(data_tbl, prop = 0.8, cumulative = TRUE)

ts_auto_recipe(.data = data_tbl,
               .date_col = visit_end_date_time,
               .pred_col = value)

ts_auto_recipe(.data = training(splits),
               .date_col = visit_end_date_time,
               .pred_col = value)
```

---

*Time Series Calendar Heatmap*
**ts_calendar_heatmap_plot**

**Description**

Takes in data that has been aggregated to the day level and makes a calendar heatmap.

**Usage**

```r
ts_calendar_heatmap_plot(
  .data,
  .date_col,
  .value_col,
  .low = "red",
  .high = "green",
  .plt_title = "",
  .interactive = TRUE
)
```

**Arguments**

- `.data` The time-series data with a date column and value column.
- `.date_col` The column that has the datetime values
- `.value_col` The column that has the values
- `.low` The color for the low value, must be quoted like "red". The default is "red"
- `.high` The color for the high value, must be quoted like "green". The default is "green"
- `.plt_title` The title of the plot
- `.interactive` Default is TRUE to get an interactive plot using `plotly::ggplotly()`. It can be set to FALSE to get a ggplot plot.

**Details**

The data provided must have been aggregated to the day level, if not funky output could result and it is possible nothing will be output but errors. There must be a date column and a value column, those are the only items required for this function to work.

This function is intentionally inflexible, it complains more and does less in order to force the user to supply a clean data-set.

**Value**

A ggplot2 plot or if interactive a plotly plot

**Author(s)**

Steven P. Sanderson II, MPH
Examples

suppressPackageStartupMessages(library(dplyr))
suppressPackageStartupMessages(library(ggplot2))
suppressPackageStartupMessages(library(timetk))
suppressPackageStartupMessages(library(lubridate))
suppressPackageStartupMessages(library(zoo))
suppressPackageStartupMessages(library(healthyR.data))
suppressPackageStartupMessages(library(stringi))
suppressPackageStartupMessages(library(plotly))
suppressPackageStartupMessages(library(purrr))
suppressPackageStartupMessages(library(forcats))

data <- healthyR_data %>%
  filter(ip_op_flag == "O") %>%
  filter(substr(visit_id, 1, 1) == "8") %>%
  select(visit_start_date_time) %>%
  filter_by_time(.date_var = visit_start_date_time,
    .start_date = "2014",
    .end_date = "2016") %>%
  summarise_by_time(.date_var = visit_start_date_time,
    value = n()) %>%
  set_names("date_col","value") %>%
  tk_augment_timeseries_signature(.date_var = date_col) %>%
  select(date_col,
    value,
    year,
    month,
    week,
    wday.lbl) %>%
  mutate(yearmonth_fct = as.yearmon(date_col) %>% factor()) %>%
  mutate(wday.lbl = fct_rev(wday.lbl)) %>%
  select(date_col, year, yearmonth_fct, everything()) %>%
  arrange(date_col) %>%
  mutate(week_of_month = stri_datetime_fields(date_col)$WeekOfMonth) %>%
  rename("week_day" = "wday.lbl")

ts_calendar_heatmap_plot(
  .data = data,
  .date_col = date_col,
  .value_col = value,
  .interactive = FALSE
)
ts_compare_data

Description

Given a tibble/data.frame, you can get date from two different but comparative date ranges. Let's say you want to compare visits in one year to visits from 2 years before without also seeing the previous 1 year. You can do that with this function.

Usage

\[
\text{ts_compare_data}(., \text{date_col}, \text{start_date}, \text{end_date}, \text{periods_back})
\]

Arguments

- \text{.data} The date.frame/tibble that holds the data
- \text{.date_col} The column with the date value
- \text{.start_date} The start of the period you want to analyze
- \text{.end_date} The end of the period you want to analyze
- \text{.periods_back} How long ago do you want to compare data too. Time units are collapsed using \text{lubridate::floor_date()}. The value can be:
  - second
  - minute
  - hour
  - day
  - week
  - month
  - bimonth
  - quarter
  - season
  - halfyear
  - year

Arbitrary unique English abbreviations as in the \text{lubridate::period()} constructor are allowed.

Details

- Uses the \text{timetk::filter_by_time()} function in order to filter the date column.
- Uses the \text{timetk::subtract_time()} function to subtract time from the start date.

Value

A tibble.
ts_forecast_simulator

Author(s)
Steven P. Sanderson II, MPH

Examples

```r
suppressPackageStartupMessages(library(healthyR.data))
suppressPackageStartupMessages(library(dplyr))
suppressPackageStartupMessages(library(timetk))
ts_compare_data(
  .data = healthyR_data,
  .date_col = visit_start_date_time,
  .start_date = "2019-01-01",
  .end_date = "2019-12-31",
  .periods_back = "2 years"
) %>%
  select(visit_start_date_time) %>%
  summarise_by_time(
    .date_var = visit_start_date_time,
    .by = "year",
    visits = n()
  )

ts_compare_data(
  .data = healthyR_data,
  .date_col = visit_end_date_time,
  .start_date = "2019-01-01",
  .end_date = "2019-12-31",
  .periods_back = "2 years"
)
```

---

**ts_forecast_simulator**  Time-series Forecasting Simulator

Description

Creating different forecast paths for forecast objects (when applicable), by utilizing the underlying model distribution with the `simulate` function.

Usage

```r
ts_forecast_simulator(
  .model,
  .horizon = 4,
  .iterations = 25,
  .sim_color = "steelblue",
  .alpha = 0.05,
  .data
)
```
Arguments

.model  A forecasting model of one of the following from the forecast package:

- Arima
- auto.arima
- ets
- nnetar

.horizon  An integer defining the forecast horizon.

.iterations  An integer, set the number of iterations of the simulation.

.sim_color  Set the color of the simulation paths lines.

.alpha  Set the opacity level of the simulation path lines.

.data  The data that is used for the .model parameter. This is used with timetk::tk_index()

Details

This function expects to take in a model of either Arima, auto.arima, ets or nnetar from the forecast package. You can supply a forecasting horizon, iterations and a few other items.

Value

The original time series, the simulated values and a some plots

Author(s)

Steven P. Sanderson II, MPH

Examples

suppressPackageStartupMessages(library(forecast))
suppressPackageStartupMessages(library(healthyR.data))
suppressPackageStartupMessages(library(dplyr))
suppressPackageStartupMessages(library(timetk))
suppressPackageStartupMessages(library(ggplot2))
suppressPackageStartupMessages(library(plotly))
suppressPackageStartupMessages(library(purrr))
suppressPackageStartupMessages(library(tidyquant))
suppressPackageStartupMessages(library(tidyr))

data <- healthyR_data %>%
  filter(ip_op_flag == "I") %>%
  select(visit_end_date_time) %>%
  rename(date_col = visit_end_date_time) %>%
  summarise_by_time(
    .date_var = date_col,
    .by = "month",
    value = n()
  ) %>%
  filter_by_time(  
    .date_var = date_col
  )
**ts_ma_plot**

`ts_ma_plot` is a function that produces two moving average plots. One is created using `xts::plot.xts()` and the other is a `ggplot2` plot. Users can choose which type is best for their needs. The plots are stacked so each graph is on top of the other.

### Description

This function will produce two plots. Both of these are moving average plots. One of the plots is from `xts::plot.xts()` and the other a `ggplot2` plot. This is done so that the user can choose which type is best for them. The plots are stacked so each graph is on top of the other.

### Usage

```r
ts_ma_plot(
  .data,  
  .date_col,  
  .value_col,  
  .ts_frequency = "monthly",  
  .main_title = NULL,
)```
Arguments

.data The data you want to visualize. This should be pre-processed and the aggregation should match the .frequency argument.
.date_col The data column from the .data argument.
.value_col The value column from the .data argument
.ts_frequency The frequency of the aggregation, quoted, ie. "monthly", anything else will default to weekly, so it is very important that the data passed to this function be in either a weekly or monthly aggregation.
.main_title The title of the main plot.
.secondary_title The title of the second plot.
.tertiary_title The title of the third plot.

Details

This function expects to take in a data.frame/tibble. It will return a list object so it is a good idea to save the output to a variable and extract from there.

Value

The original time series, the simulated values and a some plots

Author(s)

Steven P. Sanderson II, MPH

Examples

```r
suppressPackageStartupMessages(library(timetk))
suppressPackageStartupMessages(library(tidyverse))
suppressPackageStartupMessages(library(tidyquant))
suppressPackageStartupMessages(library(xts))
suppressPackageStartupMessages(library(cowplot))
suppressPackageStartupMessages(library(healthyR.data))

data_tbl <- healthyR_data %>%
  select(visit_end_date_time) %>%
  summarise_by_time(
    .date_var = visit_end_date_time,
    by = "month",
    value = n()
  ) %>%
  set_names("date_col","value") %>%
```
filter_by_time(
  .date_var = date_col,
  .start_date = "2013",
  .end_date = "2020"
)

output <- ts_ma_plot(
  .data = data_tbl,
  .date_col = date_col,
  .value_col = value
)

output$pgrid
output$xts_plt
output$data_summary_tbl %>% head()

data_tbl <- healthyR_data %>%
  select(visit_end_date_time) %>%
  summarise_by_time(
    .date_var = visit_end_date_time,
    .by = "week",
    value = n()
  ) %>%
  set_names("date_col","value") %>%
  filter_by_time(
    .date_var = date_col,
    .start_date = "2013",
    .end_date = "2020"
  )

output <- ts_ma_plot(
  .data = data_tbl,
  .date_col = date_col,
  .value_col = value,
  .ts_frequency = "week"
)

output$pgrid
output$xts_plt
output$data_summary_tbl %>% head()

---

**ts_qc_run_chart**  
*Quality Control Run Chart*

**Description**

A control chart is a specific type of graph that shows data points between upper and lower limits over a period of time. You can use it to understand if the process is in control or not. These charts commonly have three types of lines such as upper and lower specification limits, upper and lower
limits and planned value. By the help of these lines, Control Charts show the process behavior over time.

Usage

ts_qc_run_chart(
  .data,
  .date_col,
  .value_col,
  .interactive = FALSE,
  .median = TRUE,
  .cl = TRUE,
  .mcl = TRUE,
  .ucl = TRUE,
  .lc = FALSE,
  .lmcl = FALSE,
  .llcl = FALSE
)

Arguments

.data The data.frame/tibble to be passed.
.date_col The column holding the timestamp.
.value_col The column with the values to be analyzed.
.interactive Default is FALSE, TRUE for an interactive plotly plot.
.median Default is TRUE. This will show the median line of the data.
.cl This is the first upper control line
.mcl This is the second sigma control line positive
.ucl This is the third sigma control line positive
.lc This is the first negative control line
.lmcl This is the second sigma negative control line
.llcl This is the third sigma negative control line

Details

• Expects a time-series tibble/data.frame
• Expects a date column and a value column

Value

A static ggplot2 graph or if .interactive is set to TRUE a plotly plot

Author(s)

Steven P. Sanderson II, MPH
Examples

```r
library(healthyR.data)
library(timetk)
library(dplyr)
library(stringr)

df <- healthyR_data

df_monthly_tbl <- df %>%
  mutate(ip_op_flag = str_squish(ip_op_flag)) %>%
  filter(ip_op_flag == "I") %>%
  select(visit_end_date_time, length_of_stay) %>%
  arrange(visit_end_date_time) %>%
  summarise_by_time(
    .date_var = visit_end_date_time,
    .by = "month",
    alos = round(mean(length_of_stay, na.rm = TRUE), 2),
    .type = "ceiling"
  ) %>%
  mutate(
    visit_end_date_time = visit_end_date_time %>%
    subtract_time("1 day")
  )

df_monthly_tbl %>%
  ts_qc_run_chart(
    .date_col = visit_end_date_time,
    .value_col = alos,
    .llcl = TRUE
  )
```

---

### ts_random_walk

**Random Walk Function**

**Description**

This function takes in four arguments and returns a tibble of random walks.

**Usage**

```r
ts_random_walk(
  .mean = 0,
  .sd = 0.1,
  .num_walks = 100,
  .periods = 100,
  .initial_value = 1000
)
```
ts_random_walk_ggplot_layers

Arguments

- **.mean**: The desired mean of the random walks
- **.sd**: The standard deviation of the random walks
- **.num_walks**: The number of random walks you want generated
- **.periods**: The length of the random walk(s) you want generated
- **.initial_value**: The initial value where the random walks should start

Details

Monte Carlo simulations were first formally designed in the 1940's while developing nuclear weapons, and since have been heavily used in various fields to use randomness solve problems that are potentially deterministic in nature. In finance, Monte Carlo simulations can be a useful tool to give a sense of how assets with certain characteristics might behave in the future. While there are more complex and sophisticated financial forecasting methods such as ARIMA (Auto-Regressive Integrated Moving Average) and GARCH (Generalised Auto-Regressive Conditional Heteroskedasticity) which attempt to model not only the randomness but underlying macro factors such as seasonality and volatility clustering, Monte Carlo random walks work surprisingly well in illustrating market volatility as long as the results are not taken too seriously.

Value

A tibble

Author(s)

Steven P. Sanderson II, MPH

Examples

```r
ts_random_walk(
  .mean = 6,
  .sd = 1,
  .num_walks = 25,
  .periods = 180,
  .initial_value = 6
)
```

---

**ts_random_walk_ggplot_layers**

Get Random Walk ggplot2 layers

Description

Get layers to add to a ggplot graph from the `ts_random_walk()` function.
Usage

`ts_random_walk_ggplot_layers(.data)`

Arguments

.data The data passed to the function.

Details

- Set the intercept of the initial value from the random walk
- Set the max and min of the cumulative sum of the random walks

Value

A `ggplot2` layers object

Author(s)

Steven P. Sanderson II, MPH

Examples

```r
library(ggplot2)

df <- ts_random_walk()

df %>%
ggplot(
  mapping = aes(
    x = x,
    y = cum_y,
    color = factor(run),
    group = factor(run)
  )
)
  +
geom_line(alpha = 0.8) +
ts_random_walk_ggplot_layers(df)
```

Description

Sometimes we want to see the training and testing data in a plot. This is a simple wrapper around a couple of functions from the `timetk` package.
Usage

```r
ts_splits_plot(.splits_obj, .date_col, .value_col)
```

Arguments

- `.splits_obj`  The predefined splits object.
- `.date_col`    The date column for the time series.
- `.value_col`   The value column of the time series.

Details

You should already have a splits object defined. This function takes in three parameters, the splits object, a date column and the value column.

Value

A time series cv plan plot

Author(s)

Steven P. Sanderson II, MPH

See Also


Examples

```r
suppressPackageStartupMessages(library(modeltime))
suppressPackageStartupMessages(library(timetk))
suppressPackageStartupMessages(library(dplyr))
suppressPackageStartupMessages(library(healthyR.data))

data <- healthyR_data %>%
  filter(ip_op_flag == "I") %>%
  select(visit_end_date_time) %>%
  rename(date_col = visit_end_date_time) %>%
  summarise_by_time(
    .date_var = date_col,
    .by = "month",
    value = n()
  ) %>%
  filter_by_time(
    .date_var = date_col
  )
```
ts_wfs_lin_reg


time_series_split(data, date_col, assess = 12, skip = 3, cumulative = TRUE)

ts_splits_plot(.splits_obj = splits, .date_col = date_col, .value_col = value)

---

**ts_wfs_lin_reg**  
*Linear Regression Workflowset Function*

**Description**

This function is used to quickly create a workflowsets object.

**Usage**

```r
ts_wfs_lin_reg(.model_type, .recipe_list, .penalty = 1, .mixture = 0.5)
```

**Arguments**

- **.model_type**  
  This is where you will set your engine. It uses `parsnip::linear_reg()` under the hood and can take one of the following:
  - "lm"
  - "glmnet"
  - "all_engines" - This will make a model spec for all available engines.

Not yet implemented are:
- "stan"
- "spark"
- "keras"

- **.recipe_list**  
  You must supply a list of recipes. list(rec_1, rec_2, ...)

- **.penalty**  
  The penalty parameter of the glmnet. The default is 1

- **.mixture**  
  The mixture parameter of the glmnet. The default is 0.5
Details

This function expects to take in the recipes that you want to use in the modeling process. This is an automated workflow process. There are sensible defaults set for the \texttt{glmnet} model specification, but if you choose you can set them yourself if you have a good understanding of what they should be.

Value

Returns a workflowsets object.

Author(s)

Steven P. Sanderson II, MPH

See Also

https://workflowsets.tidymodels.org/\texttt{(workflowsets)}

Examples

```r
suppressPackageStartupMessages(library(modeltime))
suppressPackageStartupMessages(library(timetk))
suppressPackageStartupMessages(library(dplyr))
suppressPackageStartupMessages(library(healthyR.data))
suppressPackageStartupMessages(library(tidymodels))

data <- healthyR_data %>%
  filter(ip_op_flag == "I") %>%
  select(visit_end_date_time) %>%
  rename(date_col = visit_end_date_time) %>%
  summarise_by_time(
    .date_var = date_col,
    .by = "month",
    value = n()
  ) %>%
  filter_by_time(
    .date_var = date_col,
    .start_date = "2012",
    .end_date = "2019"
  )

splits <- time_series_split(
  data,
  date_col,
  assess = 12,
  skip = 3,
  cumulative = TRUE
)

rec_objs <- ts_auto_recipe(
  .data = training(splits)
)
ts_wfs_mars

, .date_col = date_col
, .pred_col = value
)

wf_sets <- ts_wfs_lin_reg("all_engines", rec_objs)
wf_sets

ts_wfs_mars            MARS (Earth) Workflowset Function

Description
This function is used to quickly create a workflowsets object.

Usage

ts_wfs_mars(
  .model_type = "earth",
  .recipe_list,
  .num_terms = 200,
  .prod_degree = 1,
  .prune_method = "backward"
)

Arguments

.model_type          This is where you will set your engine. It uses parsnip::mars() under the
                     hood and can take one of the following:
                     • "earth"
.recipe_list         You must supply a list of recipes. list(rec_1, rec_2, ...)
.num_terms           The number of features that will be retained in the final model, including the
                     intercept.
.prod_degree         The highest possible interaction degree.
.prune_method        The pruning method. This is a character, the default is "backward". You can
                     choose from one of the following:
                     • "backward"
                     • "none"
                     • "exhaustive"
                     • "forward"
                     • "seqrep"
                     • "cv"
Details

This function expects to take in the recipes that you want to use in the modeling process. This is an automated workflow process. There are sensible defaults set for the model specification, but if you choose you can set them yourself if you have a good understanding of what they should be. The mode is set to "regression".

This only uses the option set_engine("earth") and therefore the .model_type is not needed. The parameter is kept because it is possible in the future that this could change, and it keeps with the framework of how other functions are written.

Value

Returns a workflowsets object.

Author(s)

Steven P. Sanderson II, MPH

See Also

https://workflowsets.tidymodels.org/
https://parsnip.tidymodels.org/reference/mars.html

Examples

```r
suppressPackageStartupMessages(library(modeltime))
suppressPackageStartupMessages(library(timetk))
suppressPackageStartupMessages(library(dplyr))
suppressPackageStartupMessages(library(healthyR.data))
suppressPackageStartupMessages(library(tidymodels))
suppressPackageStartupMessages(library(earth))

data <- healthyR_data %>%
  filter(ip_op_flag == "I") %>%
  select(visit_end_date_time) %>%
  rename(date_col = visit_end_date_time) %>%
  summarise_by_time(
    .date_var = date_col,
    .by = "month",
    value = n()
  ) %>%
  filter_by_time(
    .date_var = date_col,
    .start_date = "2012",
    .end_date = "2019"
  )

splits <- time_series_split(
  data = data,
  date_col = date_col,
  assess = 12
)
ts_wfs_svm_poly

, skip = 3
, cumulative = TRUE
)

rec_objs <- ts_auto_recipe(
  .data = training(splits)
  , .date_col = date_col
  , .pred_col = value
)

wf_sets <- ts_wfs_mars("earth", rec_objs)
wf_sets

---

**ts_wfs_svm_poly**  
*SVM Poly (Kernlab) Workflowset Function*

**Description**

This function is used to quickly create a workflowsets object.

**Usage**

```r
ts_wfs_svm_poly(
  .model_type = "kernlab",
  .recipe_list,  
  .cost = 1,
  .degree = 1,
  .scale_factor = 1,
  .margin = 0.1
)
```

**Arguments**

- `.model_type`  
  This is where you will set your engine. It uses `parsnip::svm_poly()` under the hood and can take one of the following:
  - "kernlab"

- `.recipe_list`  
  You must supply a list of recipes. list(rec_1, rec_2, ...)

- `.cost`  
  A positive number for the cost of predicting a sample within or on the wrong side of the margin.

- `.degree`  
  A positive number for polynomial degree.

- `.scale_factor`  
  A positive number for the polynomial scaling factor.

- `.margin`  
  A positive number for the epsilon in the SVM insensitive loss function (regression only.)
Details

This function expects to take in the recipes that you want to use in the modeling process. This is an automated workflow process. There are sensible defaults set for the model specification, but if you choose you can set them yourself if you have a good understanding of what they should be. The mode is set to "regression".

This only uses the option `set_engine("kernlab")` and therefore the `.model_type` is not needed. The parameter is kept because it is possible in the future that this could change, and it keeps with the framework of how other functions are written.

`parsnip::svm_poly()` defines a support vector machine model. For classification, the model tries to maximize the width of the margin between classes. For regression, the model optimizes a robust loss function that is only affected by very large model residuals.

This SVM model uses a nonlinear function, specifically a polynomial function, to create the decision boundary or regression line.

Value

Returns a workflowsets object.

Author(s)

Steven P. Sanderson II, MPH

See Also

https://workflowsets.tidymodels.org/
https://parsnip.tidymodels.org/reference/svm_poly.html

Examples

```r
suppressPackageStartupMessages(library(modeltime))
suppressPackageStartupMessages(library(timetk))
suppressPackageStartupMessages(library(dplyr))
suppressPackageStartupMessages(library(healthyR.data))
suppressPackageStartupMessages(library(tidymodels))
suppressPackageStartupMessages(library(earth))

data <- healthyR_data %>%
  filter(ip_op_flag == "I") %>%
  select(visit_end_date_time) %>%
  rename(date_col = visit_end_date_time) %>%
  summarise_by_time(
    .date_var = date_col,
    .by = "month",
    value = n()
  ) %>%
  filter_by_time(
    .date_var = date_col,
    .start_date = "2012",
    .end_date = "2019"
  )
```


`splits <- time_series_split(
  data
  , date_col
  , assess = 12
  , skip = 3
  , cumulative = TRUE
)

rec_objs <- ts_auto_recipe(
  .data = training(splits)
  , .date_col = date_col
  , .pred_col = value
)

wf_sets <- ts_wfs_svm_poly("kernlab", rec_objs)
wf_sets

---

**ts_wfs_svm_rbf**

*SVM RBF (Kernlab) Workflowset Function*

**Description**

This function is used to quickly create a workflowsets object.

**Usage**

```r
ts_wfs_svm_rbf(
  .model_type = "kernlab",
  .recipe_list,
  .cost = 1,
  .rbf_sigma = 0.01,
  .margin = 0.1
)
```

**Arguments**

- **.model_type** This is where you will set your engine. It uses `parsnip::svm_rbf()` under the hood and can take one of the following:
  - "kernlab"
- **.recipe_list** You must supply a list of recipes. `list(rec_1, rec_2, ...)`
- **.cost** A positive number for the cost of predicting a sample within or on the wrong side of the margin.
- **.rbf_sigma** A positive number for the radial basis function.
- **.margin** A positive number for the epsilon in the SVM insensitive loss function (regression only).
Details

This function expects to take in the recipes that you want to use in the modeling process. This is an automated workflow process. There are sensible defaults set for the model specification, but if you choose you can set them yourself if you have a good understanding of what they should be. The mode is set to "regression".

This only uses the option set_engine("kernlab") and therefore the .model_type is not needed. The parameter is kept because it is possible in the future that this could change, and it keeps with the framework of how other functions are written.

`parsnip::svm_rbf()` defines a support vector machine model. For classification, the model tries to maximize the width of the margin between classes. For regression, the model optimizes a robust loss function that is only affected by very large model residuals.

This SVM model uses a nonlinear function, specifically a polynomial function, to create the decision boundary or regression line.

Value

Returns a workflowsets object.

Author(s)

Steven P. Sanderson II, MPH

See Also

https://workflowsets.tidymodels.org/
https://parsnip.tidymodels.org/reference/svm_rbf.html

Examples

```r
suppressPackageStartupMessages(library(modeltime))
suppressPackageStartupMessages(library(timetk))
suppressPackageStartupMessages(library(dplyr))
suppressPackageStartupMessages(library(healthyR.data))
suppressPackageStartupMessages(library(tidymodels))
suppressPackageStartupMessages(library(earth))

data <- healthyR_data %>%
  filter(ip_op_flag == "I") %>%
  select(visit_end_date_time) %>%
  rename(date_col = visit_end_date_time) %>%
  summarise_by_time(
    .date_var = date_col,
    .by = "month",
    value = n()
  ) %>%
  filter_by_time(  
    .date_var = date_col,
    .start_date = "2012",
    .end_date = "2019"
  )
```
splits <- time_series_split(
  data
  , date_col
  , assess = 12
  , skip = 3
  , cumulative = TRUE
)

rec_objs <- ts_auto_recipe(
  .data = training(splits)
  , .date_col = date_col
  , .pred_col = value
)

wf_sets <- ts_wfs_svm_rbf("kernlab", rec_objs)
wf_sets
Index

Arima, 4, 12
calibrate_and_plot, 2
ets, 4, 12
model_extraction_helper, 4
modeltime::modeltime_calibrate(), 2
modeltime::plot_modeltime_forecast(), 2
nnetar, 4, 12
parsnip::linear_reg(), 21
parsnip::mars(), 23
parsnip::svm_poly(), 25, 26
parsnip::svm_rbf(), 27, 28
plotly::ggplotly(), 8
recipes::step_nzv(), 6
recipes::step_YeoJohnson(), 6
simulate, 11
timetk::step_fourier(), 6
timetk::step_timeseries_signature(), 6
timetk::tk_index(), 12
ts_auto_recipe, 5
ts_calendar_heatmap_plot, 7
ts_compare_data, 10
ts_forecast_simulator, 11
ts_ma_plot, 13
ts_qc_run_chart, 15
ts_random_walk, 17
ts_random_walk(), 18
ts_random_walk_ggplot_layers, 18
ts_splits_plot, 19
ts_wfs_lin_reg, 21
ts_wfs_mars, 23
ts_wfs_svm_poly, 25

tstime::plot.xts(), 13