**Speed up** forecasting with modeltime’s new built-in parallel processing.

Fitting many time series models can be an expensive process. To help speed up computation, modeltime now includes **parallel processing**, which is support for high-performance computing by spreading the model fitting steps across multiple CPUs or clusters.

# Highlights

We now have a new workflow for forecast model fitting with parallel processing that is **much faster** when creating many forecast models.

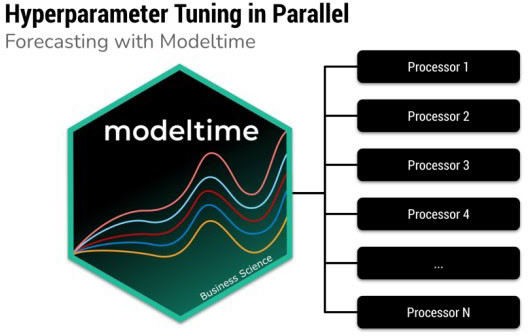
With 2-cores we got an immediate **30%-40% boost** in performance. With more expensive processes and more CPU cores we get even more performance.

It’s perfect for **hyperparameter tuning.** See create\_model\_grid() for filling model specs with hyperparameters.

The workflow is **simple.** Just use parallel\_start(6) to fire up 6-cores. Just use control\_fit\_workflowsets(allow\_par = TRUE) to tell the modeltime\_fit\_workflowset() to run in parallel.

# Forecast Hyperparameter Tuning Tutorial

## Speed up forecasting



Speed up forecasting using multiple processors

In this tutorial, we go through a common **Hyperparameter Tuning** workflow that shows off the modeltime parallel processing integration and support for workflowsets from the tidymodels ecosystem. Hyperparameter tuning is an expensive process that can benefit from parallelization.

# What is Modeltime?

## A growing ecosystem for tidymodels forecasting

Modeltime is a **growing** ecosystem of forecasting packages used to develop scalable forecasting systems for your business.

The Modeltime Ecosystem **extends** tidymodels, which means any machine learning algorithm can now become a forecasting algorithm.

# Out-of-the-Box

## Parallel Processing Functionality Included

The newest feature of the modeltime package is **parallel processing functionality.** Modeltime comes with:

Use of parallel\_start() and parallel\_stop() to simplify the parallel processing setup.

Use of create\_model\_grid() to help generate parsnip model specs from dials parameter grids.

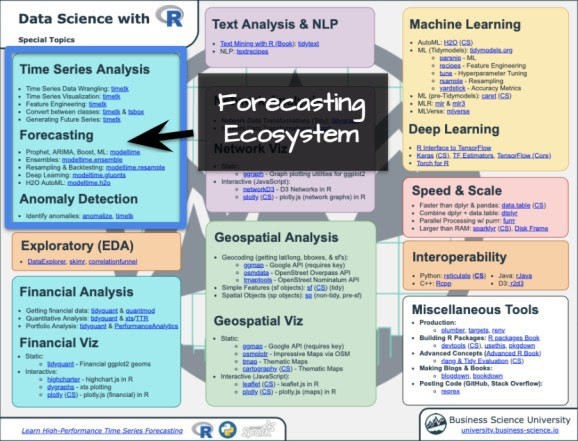
Use of modeltime\_fit\_workflowset() for initial fitting many models in parallel using workflowsets from the tidymodels ecosystem.

Use of modeltime\_refit() to refit models in parallel.

Use of control\_fit\_workflowset() and control\_refit() for controlling the fitting and refitting of many models.

# Download the Cheat Sheet

As you go through this tutorial, it may help to use the Ultimate R Cheat Sheet. Page 3 covers the *Modeltime Forecasting Ecosystem* with links to key documentation.



Forecasting Ecosystem Links (Ultimate R Cheat Sheet)

# How to Use Parallel Processing

Let’s go through a common **Hyperparameter Tuning** workflow that shows off the modeltime **parallel processing integration** and support for workflowsets from the tidymodels ecosystem.

### Libraries

Load the following libraries. Note that the new parallel processing functionality is available in Modeltime 0.6.1 (or greater).

# Machine Learning

library(modeltime) # Requires version >= 0.6.1 library(tidymodels)

library(workflowsets)

# Core library(tidyverse) library(timetk)

### Setup Parallel Backend

I’ll set up this tutorial to use two (2) cores.

To simplify creating clusters, modeltime includes parallel\_start(). We can simply supply the number of cores we’d like to use. To detect how many physical cores you have, you can run parallel::detectCores(logical = FALSE).

parallel\_start(2

### Load Data

We’ll use the walmart\_sales\_weeekly dataset from timetk. It has seven (7) time series that represent weekly sales demand by department.

dataset\_tbl <- walmart\_sales\_weekly %>% select(id, Date, Weekly\_Sales

dataset\_tbl %>% group\_by(id) %>% plot\_time\_series(

.date\_var = Date,

.value = Weekly\_Sales,

.facet\_ncol = 2,

.interactive = FALSE



### Train / Test Splits

Use time\_series\_split() to make a temporal split for all seven time series.

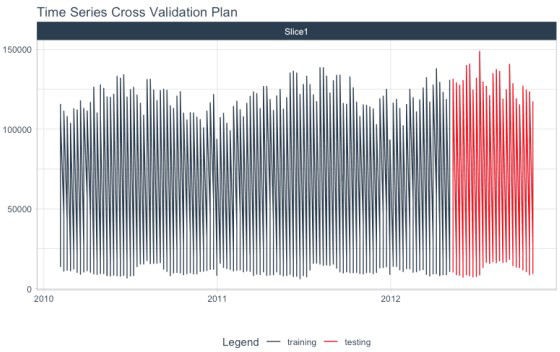
splits <- time\_series\_split( dataset\_tbl,

assess = "6 months", cumulative = TRUE

)

splits %>% tk\_time\_series\_cv\_plan() %>%

plot\_time\_series\_cv\_plan(Date, Weekly\_Sales, .interactive = F)



### Recipe

Make a preprocessing recipe that generates time series features.

recipe\_spec\_1 <- recipe(Weekly\_Sales ~ ., data = training(splits)) %>% step\_timeseries\_signature(Date) %>%

step\_rm(Date) %>% step\_normalize(Date\_index.num) %>% step\_zv(all\_predictors()) %>%

step\_dummy(all\_nominal\_predictors(), one\_hot = TRUE)

### Model Specifications

We’ll make 6 xgboost model specifications using boost\_tree() and the “xgboost” engine. These will be combined with the recipe from the previous step using a workflow\_set() in the next section.

#### The general idea

We can vary the learn\_rate parameter to see it’s effect on forecast error.

# XGBOOST MODELS

model\_spec\_xgb\_1 <- boost\_tree(learn\_rate = 0.001) %>% set\_engine("xgboost"

model\_spec\_xgb\_2 <- boost\_tree(learn\_rate = 0.010) %>% set\_engine("xgboost"

model\_spec\_xgb\_3 <- boost\_tree(learn\_rate = 0.100) %>% set\_engine("xgboost"

model\_spec\_xgb\_4 <- boost\_tree(learn\_rate = 0.350) %>% set\_engine("xgboost"

model\_spec\_xgb\_5 <- boost\_tree(learn\_rate = 0.500) %>% set\_engine("xgboost"

model\_spec\_xgb\_6 <- boost\_tree(learn\_rate = 0.650) %>% set\_engine("xgboost"

#### A faster way

You may notice that this is a lot of repeated code to adjust the learn\_rate. To simplify this process, we can use create\_model\_grid(). model\_tbl <- tibble(

learn\_rate = c(0.001, 0.010, 0.100, 0.350, 0.500, 0.650)

) %>%

create\_model\_grid( f\_model\_spec = boost\_tree, engine\_name = "xgboost", mode = "regression"

model\_tbl

## # A tibble: 6 x 2 ## learn\_rate .models ## <dbl> <list>

## 1 0.001 <spec[+]>

## 2 0.01 <spec[+]>

## 3 0.1 <spec[+]>

## 4 0.35 <spec[+]>

## 5 0.5 <spec[+]>

## 6 0.65 <spec[+]>

#### Extracting the model list

We can extract the model list for use with our workflowset next. This is the same result if we would have placed the manually generated 6 model specs into a list(). model\_list <- model\_tbl$.models

model\_list

## [[1]]

## Boosted Tree Model Specification (regression) ##

## Main Arguments:

## learn\_rate = 0.001 ##

## Computational engine: xgboost ##

##

## [[2]]

## Boosted Tree Model Specification (regression) ##

## Main Arguments:

## learn\_rate = 0.01 ##

## Computational engine: xgboost ##

##

## [[3]]

## Boosted Tree Model Specification (regression) ##

## Main Arguments:

## learn\_rate = 0.1 ##

## Computational engine: xgboost ##

##

## [[4]]

## Boosted Tree Model Specification (regression) ##

## Main Arguments:

## learn\_rate = 0.35 ##

## Computational engine: xgboost ##

##

## [[5]]

## Boosted Tree Model Specification (regression) ##

## Main Arguments:

## learn\_rate = 0.5 ##

## Computational engine: xgboost ##

##

## [[6]]

## Boosted Tree Model Specification (regression) ##

## Main Arguments:

## learn\_rate = 0.65 ##

## Computational engine: xgboost

### Workflowsets

With the workflow\_set() function, we can combine the 6 xgboost models with the 1 recipe to return six (6) combinations of recipe and model specifications. These are currently untrained (unfitted).

model\_wfset <- workflow\_set( preproc = list(

recipe\_spec\_1

)

models = model\_list, cross = TRUE

)

model\_wfset

## # A workflow set/tibble: 6 x 4

## wflow\_id info option result ## <chr> <list> <list> <list>

## 1 recipe\_boost\_tree\_1 <tibble [1 × 4]> <wrkflw > <list [0]>

## 2 recipe\_boost\_tree\_2 <tibble [1 × 4]> <wrkflw > <list [0]>

## 3 recipe\_boost\_tree\_3 <tibble [1 × 4]> <wrkflw > <list [0]>

## 4 recipe\_boost\_tree\_4 <tibble [1 × 4]> <wrkflw > <list [0]>

## 5 recipe\_boost\_tree\_5 <tibble [1 × 4]> <wrkflw > <list [0]>

## 6 recipe\_boost\_tree\_6 <tibble [1 × 4]> <wrkflw > <list [0]>

### Parallel Training (Fitting)

We can train each of the combinations in parallel.

#### Controlling the Fitting Proces

Each fitting function in modeltime has a “control” function:

control\_fit\_workflowset() for modeltime\_fit\_workflowset() control\_refit() for modeltime\_refit()

The control functions help the user control the verbosity (adding remarks while training) and set up parallel processing. We can see the output when verbose = TRUE and allow\_par = TRUE.

**allow\_par:** Whether or not the user has indicated that parallel processing should be used.

If the user has set up parallel processing externally, the clusters will be reused.

If the user has not set up parallel processing, the fitting (training) process will set up parallel processing internally and shutdown. Note that this is more expensive, and usually costs around 10-15 seconds to set up.

**verbose:** Will return important messages showing the progress of the fitting operation.

**cores:** The cores that the user has set up. Since we’ve already set up doParallel to use 2 cores, the control recognizes this.

**packages:** The packages are packages that will be sent to each of the workers.

control\_fit\_workflowset( verbose = TRUE allow\_par = TRUE

)

## workflowset control object ## ## allow\_par : TRUE

## cores : 2

## verbose : TRUE

## packages : modeltime parsnip dplyr stats lubridate tidymodels timetk forcats stringr readr tidyverse yardstick workflowsets workflows tune tidyr tibble rsample recipes purrr modeldata infer ggplot2 dials scales broom graphics grDevices utils datasets methods base

#### Fitting Using Parallel Backend

We use the modeltime\_fit\_workflowset() and control\_fit\_workflowset() together to train the unfitted workflowset in parallel.

model\_parallel\_tbl <- model\_wfset %>% modeltime\_fit\_workflowset(

data = training(splits)

control = control\_fit\_workflowset( verbose = TRUE

allow\_par = TRUE

## Using existing parallel backend with 2 clusters (cores)... ## Beginning Parallel Loop | 0.006 seconds

## Finishing parallel backend. Clusters are remaining open. | 12.458 seconds ## Close clusters by running: `parallel\_stop()`.

## Total time | 12.459 seconds

This returns a modeltime table.

model\_parallel\_tbl

## # Modeltime Table ## # A tibble: 6 x 3

## .model\_id .model .model\_desc ## <int> <list> <chr>

## 1 1 <workflow> XGBOOST

## 2 2 <workflow> XGBOOST

## 3 3 <workflow> XGBOOST

## 4 4 <workflow> XGBOOST

## 5 5 <workflow> XGBOOST

## 6 6 <workflow> XGBOOST

#### Comparison to Sequential Backend

We can compare to a sequential backend. We have a slight perfomance boost. Note that this performance benefit increases with the size of the training task.

model\_sequential\_tbl <- model\_wfset %>% modeltime\_fit\_workflowset(

data = training(splits)

control = control\_fit\_workflowset( verbose = TRUE

allow\_par = FALSE

## ℹ Fitting Model: 1

## ✓ Model Successfully Fitted: 1 ## ℹ Fitting Model: 2

## ✓ Model Successfully Fitted: 2 ## ℹ Fitting Model: 3

## ✓ Model Successfully Fitted: 3 ## ℹ Fitting Model: 4

## ✓ Model Successfully Fitted: 4 ## ℹ Fitting Model: 5

## ✓ Model Successfully Fitted: 5 ## ℹ Fitting Model: 6

## ✓ Model Successfully Fitted: 6 ## Total time | 15.781 seconds

### Accuracy Assessment

We can review the forecast accuracy. We can see that Model 5 has the lowest MAE.

model\_parallel\_tbl %>% modeltime\_calibrate(testing(splits)) %>% modeltime\_accuracy() %>% table\_modeltime\_accuracy(.interactive = FALSE)

##### Accuracy Table

**.model\_id .model\_desc .type mae mape mase smape rmse rsq**

|  |  |  |
| --- | --- | --- |
| 1 | XGBOOST | Test 55572.50 98.52 1.63 194.17 66953.92 0.96 |
| 2 | XGBOOST | Test 48819.23 86.15 1.43 151.49 58992.30 0.96 |
| 3 | XGBOOST | Test 13426.89 21.69 0.39 25.06 17376.53 0.98 |
| 4 | XGBOOST | Test 3699.94 8.94 0.11 8.68 5163.37 0.98 |
| 5 | XGBOOST | Test 3296.74 7.30 0.10 7.37 5166.48 0.98 |
| 6 | XGBOOST | Test 3612.70 8.15 0.11 8.24 5308.19 0.98 |

### Forecast Assessment

We can visualize the forecast.

model\_parallel\_tbl %>% modeltime\_forecast(

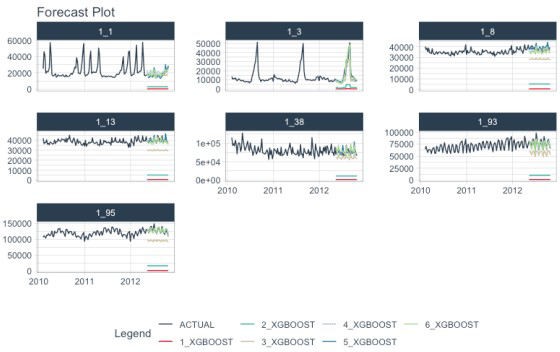
new\_data = testing(splits), actual\_data = dataset\_tbl, keep\_data = TRUE

) %>%

group\_by(id) %>% plot\_modeltime\_forecast(

.facet\_ncol = 3

.interactive = FALSE



### Closing Clusters

We can close the parallel clusters using parallel\_stop(). parallel\_stop(

# It gets better

## You’ve just scratched the surface, here’s what’s coming…

The Modeltime Ecosystem functionality is much more feature-rich than what we’ve covered here (I couldn’t possibly cover everything in this post).  Here’s what I didn’t cover:

**Feature Engineering:** We can make this forecast much more accurate by including features from competition-winning strategies

**Ensemble Modeling:** We can stack H2O Models with other models not included in H2O like GluonTS Deep Learning.

**Deep Learning:** We can use GluonTS Deep Learning for developing high-performance, scalable forecasts.