Let’s rewind for a moment: in the previous post, we looked at how we can combine econometric techniques like differencing with machine learning (ML) algorithms like random forests to predict a time series with a high degree of accuracy.

Since we covered quite some ground in the last post, there wasn’t much room for other topics. This is where this post comes in. Today, we take a look at how we can tune the hyperparameters of a random forest when dealing with time series data.

Any takers? Alright, then let’s do this! If you read the last post, feel free to skip over section 1 and move right on to 2.

**The setup**

# load packages

suppressPackageStartupMessages(require(tidyverse))

suppressPackageStartupMessages(require(tsibble))

suppressPackageStartupMessages(require(randomForest))

suppressPackageStartupMessages(require(forecast))

# specify the path to the csv file (your path here)

file <- "/Users/manueltilgner/Library/Mobile Documents/com~apple~CloudDocs/Artificial Intelligence/10\_Blog/RF/tax.csv"

# read in the csv file

tax\_tbl <- readr::read\_delim(

file = file,

delim = ";",

col\_names = c("Year", "Type", month.abb),

skip = 1,

col\_types = "iciiiiiiiiiiii",

na = c("...")

) %>%

select(-Type) %>%

gather(Date, Value, -Year) %>%

unite("Date", c(Date, Year), sep = " ") %>%

mutate(

Date = Date %>%

lubridate::parse\_date\_time("m y") %>%

yearmonth()

) %>%

drop\_na() %>%

as\_tsibble(index = "Date") %>%

filter(Date <= "2018-12-01")

# convert to ts format

tax\_ts <- as.ts(tax\_tbl)

Again, all this is from last time, I just have it here so you can get up and running quickly (just copy paste).

# pretend we're in December 2017 and have to forecast the next twelve months

tax\_ts\_org <- window(tax\_ts, end = c(2017, 12))

# estimate the required order of differencing

n\_diffs <- nsdiffs(tax\_ts\_org)

# log transform and difference the data

tax\_ts\_trf <- tax\_ts\_org %>%

log() %>%

diff(n\_diffs)

# embed the matrix

lag\_order <- 6 # the desired number of lags (six months)

horizon <- 12 # the forecast horizon (twelve months)

tax\_ts\_mbd <- embed(tax\_ts\_trf, lag\_order + 1) # embedding magic!

# do a train test split

y\_train <- tax\_ts\_mbd[, 1] # the target

X\_train <- tax\_ts\_mbd[, -1] # everything but the target

y\_test <- window(tax\_ts, start = c(2018, 1), end = c(2018, 12)) # the year 2018

X\_test <- tax\_ts\_mbd[nrow(tax\_ts\_mbd), c(1:lag\_order)] # the test set consisting

# of the six most recent values (we have six lags) of the training set. It's the

# same for all models.

**Cross-validation and its (mis-)contents**

How do you tune the hyperparameters of an ML model when you’re dealing with time series data? Viewer discretion advised: the following material might be disturbing to some audiences.

Before we answer the question above, let’s first answer the following: how do we tune the hyperparameters of an ML model on data that has no time dimension?

Generally, we proceed as follows:

1. pick a handful of hyperparameters,
2. select a sensible range of values for each, and finally,
3. determine which configuration is ‚best‘, e.g., in the sense of minimizing an error metric.

Steps 1) and 2) are a matter of trial and error. In most cases, there’s no analytic way of deriving which hyperparameters in which configuration work best. While some practical insights have emerged over the years, generally, you just have to experiment and see what works with your data.

What about step 3)? Finding the ‚best‘ configuration can be achieved with the help of different resampling schemes, a popular one of which k-fold cross-validation. Popular because k-fold cross-validation tends to provide fairly reliable model performance estimates and makes efficient use of the data. How so?

Let’s refresh: k-fold cross-validation works by splitting the data into folds of roughly equal size. Each fold constitutes a subset of the data with observations. It then fits the model on folds and computes its loss (e.g., RMSE) th left out fold, which serves as a validation set. This process repeats times until each fold has served once as a validation set.

From the description, it’s clear why this sounds like a bad idea for time series data. If we do k-fold cross-validation, we invariably end up fitting a model on future data to predict the past! I don’t know about you, but somehow this doesn’t feel right. No need to mention, of course, that if we shuffle the data before splitting it into folds, we completely destroy the time structure.

**Hold out, not so fast**

What to do instead? In time series econometrics, a traditional approach is this: reserve the last part of your time series as a holdout set. That is, fit your model on the first part of the series and evaluate it on the last part. In essence, this is nothing more than a validation set strategy.

This approach is simple and intuitive. Moreover, it preserves the order of the time series. But it also has a significant downside: it does not provide robust performance estimates since we test our model only once and on a ‚test‘ set that is arbitrary.

Is it a problem?

It definitely can be: time series above all tend to represent phenomena that are ever-changing and evolving. As such, the test set (i.e., the ‚future‘) may be systematically different than the train set (i.e., the ‚past‘). This change in the underlying data generating process over time is known as ‚concept drift‘ and poses a serious challenge in time series forecasting.

But we’re getting off track! What is it that we’re trying to achieve here? We want to get reliable performance estimates of our ML model on time series data. Only then we can decide with some degree of certitude which hyperparameters to choose for our model. Yet, the traditional approach of doing a single train/test split doesn’t really cut it for us.

Much better would be multiple train/test splits. In the context of time series, this means sliding a fixed or steadily expanding window over our series, training on one part of the data, and predicting the next, then computing the loss. Next, move the window ahead an observation (or enlarge it) and repeat.

This approach, called time series cross-validation is effective, but also computationally expensive. Imagine this, if you have 10 hyperparameter configurations and you test each of them with 20 train/test splits, you end up calculating two hundred models. Depending on the model and the amount of data you have, this can take its sweet time.

Since we’re busy people living a busy world, let’s stick with the holdout strategy for now. It sounds simple, but how do you do it? Personally, I like to use the createTimeSlices function from the caret package. This makes sense if you have a caret workflow or work with many different series. If not, simply slice your (training) series so that the last part is reserved as a validation set.

To see how the createTimeSlices function works, run it on its own first.

# we hold out the last 12 observations from our train set because this is also how far we want to predict into the future later

caret::createTimeSlices(

1:nrow(X\_train),

initialWindow = nrow(X\_train) - horizon,

horizon = horizon,

fixedWindow = TRUE

)

You see that it splits our train set into a train and validation set, based on the forecast horizon we specified (12 months). Ok, now let’s set it up in such a way that we can use it in a train workflow:

tr\_control <- caret::trainControl(

method = 'timeslice',

initialWindow = nrow(X\_train) - horizon,

fixedWindow = TRUE

)

With trainControl in place, let us next set up a tuning grid. While we can get super fancy here, for random forests, it often boils down to two hyperparameters that matter: the number of trees (ntree) and the number of predictors (mtry) that get sampled at each split in the tree.

Good values for ntree are a few hundred to a thousand. More trees can give you a bump in accuracy, but usually not much. Since random forests do not run a high risk of overfitting, the question of how many trees you use really comes down to how much computing power (or time) you have. Since tuning with time series tends to be computationally expensive, let’s pick 500 trees.

mtry, on the other hand, tends to be the parameter where the party’s at. It represents the number of predictors that get considered as splitting candidates at each node in the tree. James et al. (2013) recommend or (where is the number of predictors) as a guideline. I’m also going to throw in , which amounts to bagging

# caret actually only allows us to put one hyperparameter here

tune\_grid <- expand.grid(

mtry = c(

ncol(X\_train), # p

ncol(X\_train) / 3, # p / 3

ceiling(sqrt(ncol(X\_train))) # square root of p

)

)

# let's see which hyperparameter the holdout method recommends

holdout\_result <- caret::train(

data.frame(X\_train),

y\_train,

method = 'rf',

trControl = tr\_control,

tuneGrid = tune\_grid

)

This method recommends mtry = 6. So, should we stop and fit our model already? Not just yet!

**Cross-validation again?**

Remember all the bad things we said about k-fold cross-validation (CV) with time series data? All of it is still technically true. But now comes the weird part. Bergmeir et al. (2018) found that k-fold CV *can* actually be applied with time series models. But only if they are purely autoregressive. In other words, we can use k-fold CV on time series data, but only if the predictors in our model are lagged versions of the response.

How come? The authors note that in order for k-fold CV to be valid, the errors of the model must be uncorrelated. This condition is met when the model that we train fits the data well. In other words, k-fold CV is valid if our model nests a good model, i.e., a model whose weights are a subset of the model weights we train. When this is the case, k-fold cross-validation is actually a better choice than the holdout strategy!

I don’t know about you, but for me, this was a surprise. Since it’s a mighty useful insight, it’s worth repeating one more time: **if the residuals of our (purely autoregressive) time series model are serially uncorrelated, then k-fold cross-validation can and should be used over the holdout strategy.**

Are you as curious as I am now? Then let’s put it work for us and not just once, but repeatedly!

tr\_control <- trainControl(

method = 'repeatedcv',

number = 10,

repeats = 3

)

kfold\_result <- caret::train(

data.frame(X\_train),

y\_train,

method = 'rf',

trControl = tr\_control,

tuneGrid = tune\_grid

)

Using k-fold CV on this time series suggests a value of mtry = 2.

Interesting! Let’s train our random forest twice now, once with mtry = 2and once with mtry = 6. Then, we compare which one gives a better prediction!

**mtrying it out!**

# set up our empty forecast tibble

forecasts\_rf <- tibble(

mtry\_holdout = rep(NA, horizon),

mtry\_kfold = rep(NA, horizon)

)

# collect the two mtry values from the tuning step

mtrys <- c(

holdout\_result$bestTune[[1]],

kfold\_result$bestTune[[1]]

)

# train the model in a double loop

for (i in seq\_len(length(mtrys))) {

# start fresh for each mtry run

y\_train <- tax\_ts\_mbd[, 1]

X\_train <- tax\_ts\_mbd[, -1]

# train the models

for (j in seq\_len(horizon)) {

# set seed

set.seed(2019)

# fit the model

fit\_rf <- randomForest(X\_train, y\_train, mtry = mtrys[i])

# predict using the test set

forecasts\_rf[j, i] <- predict(fit\_rf, X\_test)

# here is where we repeatedly reshape the training data to reflect the time # distance corresponding to the current forecast horizon.

y\_train <- y\_train[-1]

X\_train <- X\_train[-nrow(X\_train), ]

}

}

Again, we need to back-transform our forecasts to calculate the accuracy on our test set. For this, I’m going to use purrr.

last\_observation <- as.vector(tail(tax\_ts\_org, 1))

forecasts <- forecasts\_rf %>%

purrr::map\_df(function(x) exp(cumsum(x)) \* last\_observation)

accuracies <- forecasts %>%

purrr::map(function(x) accuracy(x, as.vector(y\_test))) %>%

do.call(rbind, .)

And what do you know! k-fold CV proved indeed better than our holdout approach. It reduced both RMSE and MAPE. We even validated our result from last time, where we also had a MAPE of 2.6. So at least here, using random forest out of the box was totally fine.

|  | **ME** | **RMSE** | **MAE** | **MPE** | **MAPE** |
| --- | --- | --- | --- | --- | --- |
| holdout | 429258.4 | **484154.7** | 429258.4 | 4.709705 | **4.709705** |
| k-fold | 198307.5 | **352789.9** | 238652.6 | 2.273785 | **2.607773** |

Let’s visualize it as well:

plot <- tax\_tbl %>%

filter(Date >= "2018-01-01") %>%

mutate(

Ground\_Truth = Value / 10000,

Forecast\_Holdout = forecasts$mtry\_holdout / 10000,

Forecast\_Kfold = forecasts$mtry\_kfold / 10000,

) %>%

ggplot(aes(x = Date)) +

geom\_line(aes(y = Value / 10000, linetype = "Truth")) +

geom\_line(aes(y = Forecast\_Holdout, color = "Holdout")) +

geom\_line(aes(y = Forecast\_Kfold, color = "k-fold CV")) +

theme\_minimal()+

labs(

title = "Forecast of the German Wage and Income Tax for the Year 2018",

x = "Months",

y = "Euros"

) +

scale\_color\_manual(

name = "Forecasts",

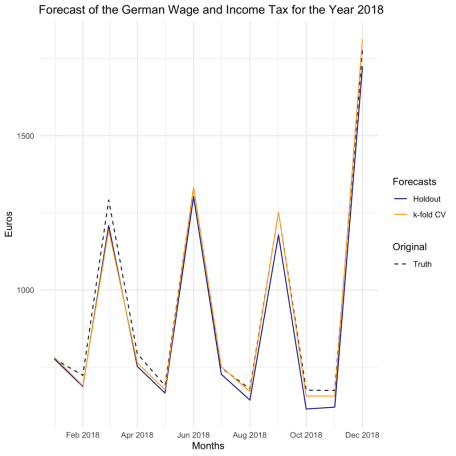
values = c("Truth" = "black", "Holdout" = "darkblue", "k-fold CV" = "orange")

) +

scale\_linetype\_manual(name = "Original", values = c("Truth" = "dashed")) +

scale\_x\_date(date\_labels = "%b %Y", date\_breaks = "2 months")

We see that the orange line, which represents the forecasts from the k-fold CV model, tends to hug the true values more snugly at several points.



**Conclusion (TL;DR)**

Tuning ML models on time series data can be expensive, but it needn’t be. If the model you’re fitting uses only endogenous predictors, i.e., lags of the response, you’re in luck! You can go ahead and use the known and beloved k-fold cross-validation strategy to tune your hyperparameters. If you want to go deeper, check out the original paper in the reference. Otherwise, go pick a time a time series of your choice and see if you can improve your model with a bit of tuning. No matter the result, it’ll always beat doing taxes!