Benjamin Franklin said that only two things are certain in life: death and taxes.

What could taxes and the outdoors possibly have in common? Well, I asked myself: can we predict tax revenue using random forest? (wildly creative, I know). When dealing with tax revenue, we enter the realm of time series, ruled by fantastic beasts like ARIMA, VAR, STLM, and others. These are tried and proven methods, so why use random forests?

Well, you and I may both agree that random forest is one of the most awesome algorithms around: it’s simple, flexible, and powerful. So much so, that Wyner et al. (2015) call it *the* ‚off-the-shelf‘ tool for most data science applications. Long story short, it’s one of those algorithms that just works

**Random forest is a hammer, but is time series data a nail?**

You probably used random forest for regression and classification before, but time series forecasting? Hold up you’re going to say; time series data is special! And you’re right. When it comes to data that has a time dimension, applying machine learning (ML) methods becomes a little tricky.

How come? Well, random forests, like most ML methods, have no awareness of time. On the contrary, they take observations to be independent and identically distributed. This assumption is obviously violated in time series data which is characterized by serial dependence.

What’s more, random forests or decision tree based methods are unable to predict a trend, i.e., they do not extrapolate. To understand why, recall that trees operate by if-then rules that recursively split the input space. Thus, they’re unable to predict values that fall outside the range of values of the target in the training set.

So, should we go back to ARIMA? Not just yet! With a few tricks, we *can* do time series forecasting with random forests. All it takes is a little pre- and (post-)processing. This blog post will show you how you can harness random forests for forecasting!

Let it be said that there are different ways to go about this. Here’s how we are going to pull it off: We’ll raid the time series econometrics toolbox for some old but gold techniques – differencing and statistical transformations. These are cornerstones of ARIMA modeling, but who says we can’t use them for random forests as well?

To stick with the topic, we’ll use a time series from the German Statistical Office on the German wage and income tax revenue from 1999 – 2018 (after tax redistribution). You can download the data [here](https://www-genesis.destatis.de/genesis/online/data;sid=324194910261427BD227A6DA6E868E7B.GO_1_1?operation=abruftabelleAbrufen&selectionname=71211-0006&levelindex=0&levelid=1560495247902&index=24). Let’s do it!

**Getting ready for machine learning or what’s in a time series anyway?**

Essentially, a (univariate) time series is a vector of values indexed by time. In order to make it ‚learnable‘ we need to do some pre-processing. This can include some or all of the following:

* Statistical transformations (Box-Cox transform, log transform, etc.)
* Detrending (differencing, STL, SEATS, etc.)
* Time Delay Embedding (more on this below)
* Feature engineering (lags, rolling statistics, Fourier terms, time dummies, etc.)

For brevity and clarity, we’ll focus on steps one to three in this post.

Ok, let’s structure this a bit: in order to use random forest for time series data we do TDE: transform, difference and embed. Let’s fire up R and load the required packages plus our data.

# load the packages

suppressPackageStartupMessages(require(tidyverse))

suppressPackageStartupMessages(require(tsibble))

suppressPackageStartupMessages(require(randomForest))

suppressPackageStartupMessages(require(forecast))

# specify the csv file (your path here)

file <- ".../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)

Before we dive into the analysis, let’s quickly check for implicit and explicit missings in the data. The tsibble package has some handy functions to do just that:

# implicit missings

has\_gaps(tax\_tbl)

# explicit missings

colSums([is.na](http://is.na" \t "_blank)(tax\_tbl[, "Value"]))

Nope, looks good! So what kind of time series are we dealing with?

# visualize

plot\_org <- tax\_tbl %>%

ggplot(aes(Date, Value / 1000)) + # to get the axis on a more manageable scale

geom\_line() +

theme\_minimal() +

labs(title = "German Wage and Income Taxes 1999 - 2018", x = "Year", y = "Euros")



**Differencing can make all the difference**

If you’ve worked with classical time series models before, you likely stumbled across the concept of differencing. The reason for this is that classical time series models require the data to be stationary.

Stationarity means that the mean and variance of the series is finite and does not change over time. Thus, it implies some stability in the statistical properties of the time series. As we can see in the plot, our time series is far from it! There is an upward trend as well as a distinct seasonal pattern in the series.

How is it done? Here, we simply take the first differences of the data, i.e., the difference between consecutive observations ​. This also works with a seasonal lag ​, which amounts to taking the difference between an observation and a previous observation from the same season, e.g., November this year and November last year.

How do these two concepts – differencing and stationarity – relate? You probably already know or guessed it: differencing is one way to make non-stationary time series data stationary. That’s nice to know, but right now we care more about the fact that differencing removes changes in the level of a series and, with it, the trend. Just what we need for our random forest!

How is it done? Here, we simply take the first differences of the data, i.e., the difference between consecutive observations y_t' = y_t - y_{t-1}. This also works with a seasonal lag y_t' = y_t - y_{t-s}, which amounts to taking the difference between an observation and a previous observation from the same season, e.g., November this year and November last year.

Whereas differencing can stabilize the mean of a time series, a Box-Cox or log transformation can stabilize the variance. The family of Box-Cox transformations revolves around the parameter lambda:

\[\tilde{y}_t = \begin{cases} \log(y_t) & \lambda = 0\\ (y_t^\lambda - 1)/ \lambda & \lambda \neq 0 \end{cases}\]

When lambda is zero, the Box-Cox transformation amounts to taking logs. We choose this value to make the back-transformation of our forecasts straightforward. But don’t hesitate to experiment with different values of lambda or estimate the ‚best‘ value with the help of the forecast package.

# 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)

# check out the difference! (pun)

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

autoplot() +

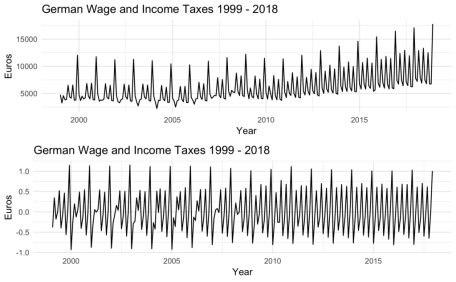
xlab("Year") +

ylab("Euros") +

ggtitle("German Wage and Income Taxes 1999 - 2018") +

theme\_minimal()

gridExtra::grid.arrange(plot\_org, plot\_trf)



Let’s sum up what we’ve done so far: we first took logs of the data to stabilize the variance. Then, we differenced the data once to make it stationary in the mean. Together, these rather simple transformations took us from non-stationary to stationary.

What’s next? Well, now we use the data thus transformed to train our random forest and to make forecasts. Once we obtain the forecasts, we reverse the transformations to get them on the original scale of the data.

Just one more step before we get to the modeling part: we still only have a vector. How do we cast this data in a shape, that an ML algorithm can handle?

**Enter the matrix: Time Delay Embedding**

To feed our random forest the transformed data, we need to turn what is essentially a vector into a matrix, i.e., a structure that an ML algorithm can work with. For this, we make use of a concept called time delay embedding.

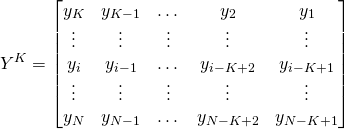
Time delay embedding represents a time series in a Euclidean space with the embedding dimension K. To do this in R, use the base function embed(). All you have to do is plug in the time series object and set the embedding dimension as one greater than the desired number of lags.

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!

When you check out the tax\_ts\_mbd object, you’ll see that you get a matrix where the dependent variable in the first column is regressed on its lags in the remaining columns:



Time delay embedding allows us to use any linear or non-linear regression method on time series data, be it random forest, gradient boosting, support vector machines, etc. I decided to go with a lag of six months, but you can play around with other lags. Moreover, the forecast horizon is twelve as we’re forecasting the tax revenue for the year 2018.

**When it comes to forecasting, I’m pretty direct**

In this post, we make use of the direct forecasting strategy. That means that we estimate Hseparate models f_H, one for each forecast horizon. In other words, we train a separate model for each time distance in the data.

The direct forecasting strategy is less efficient than the recursive forecasting strategy, which estimates only one model fand, as the name suggests, re-uses it Htimes. Recursive, in this case, means that we feed back each forecast as input back to the model to get the next forecast.

Despite this drawback, the direct strategy has two key advantages: First, it does not suffer from an accumulation of forecast errors, and second, it makes it straightforward to include exogenous predictors.

How to implement the direct forecasting strategy is nicely demonstrated in the before-mentioned post, so I don’t want rehash it here. If you’re short on time, the **tl;dr** is this: we use the direct forecasting strategy to generate multi-step ahead forecasts. This entails training a model for each forecast horizon by progressively reshaping the training data to reflect the time distance between observations.

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.

If you followed me here, kudos, we’re almost done! For now, we get to the fun part: letting our random forest loose on this data. We train the model in a loop, where each iteration fits one model, one for each forecast horizon h = 1 \dots H.

**The random forest forecast: things are looking good**

Below I’m using the random forest straight out of the box, not even bothering tuning it (a topic to which I’d like to dedicate a post in the future). It may seem lazy (and probably is), but I stripped the process down to its bare bones in the hope of showing most clearly what is going on here.

forecasts\_rf <- numeric(horizon)

for (i in 1:horizon){

# set seed

set.seed(2019)

# fit the model

fit\_rf <- randomForest(X\_train, y\_train)

# predict using the test set

forecasts\_rf[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), ]

}

Alright, the loop’s done. We just trained twelve models and got twelve forecasts. Since we transformed our time series before training, we need to transform the forecasts back.

**Back to the former or how we get forecasts on the original scale**

As we took the log transform earlier, the back-transform is rather straightforward. We roll back the process from the inside out, i.e., we first reverse the differencing and then the log transform. We do this by exponentiating the cumulative sum of our transformed forecasts and multiplying the result with the last observation of our time series. In other words, we calculate:

\[y_{t+H} = y_{t}\exp\left(\sum_{h = 1}^H \tilde{y}_{t+h}\right)\]

# calculate the exp term

exp\_term <- exp(cumsum(forecasts\_rf))

# extract the last observation from the time series (y\_t)

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

# calculate the final predictions

backtransformed\_forecasts <- last\_observation \* exp\_term

# convert to ts format

y\_pred <- ts(

backtransformed\_forecasts,

start = c(2018, 1),

frequency = 12

)

# add the forecasts to the original tibble

tax\_tbl <- tax\_tbl %>%

mutate(Forecast = c(rep(NA, length(tax\_ts\_org)), y\_pred))

# visualize the forecasts

plot\_fc <- tax\_tbl %>%

ggplot(aes(x = Date)) +

geom\_line(aes(y = Value / 1000)) +

geom\_line(aes(y = Forecast / 1000), color = "blue") +

theme\_minimal() +

labs(

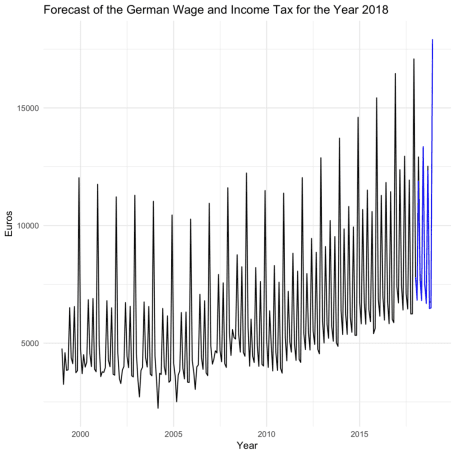
title = "Forecast of the German Wage and Income Tax for the Year 2018",

x = "Year",

y = "Euros"

)

accuracy(y\_pred, y\_test)



|  | **ME** | **RMSE** | **MAE** | **MPE** | **MAPE** | **ACF1** | **Theil’s U** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Test set | 198307.5 | 352789.9 | 238652.6 | 2.273785 | 2.607773 | 0.257256 | 0.0695694 |

It looks like our forecast is pretty good! We achieved a MAPE of 2.6 percent. But since one should never rely on accuracy metrics alone, let’s quickly calculate a simple benchmark like the seasonal naive model. That’s a low hurdle to pass, so if our model doesn’t beat it, in the bin it goes.

benchmark <- forecast(snaive(tax\_ts\_org), h = horizon)

tax\_ts %>%

autoplot() +

autolayer(benchmark, PI = FALSE)

accuracy(benchmark, y\_test)

|  | **ME** | **RMSE** | **MAE** | **MPE** | **MAPE** | **MASE** | **ACF1** | **Theil’s U** |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Training set | 216938.7 | 470874.8 | 388618.5 | 2.852533 | 6.648711 | 1.000000 | 0.6303982 | NA |
| Test set | 485006.8 | 495096.8 | 485006.8 | 5.507943 | 5.507943 | 1.248028 | 0.1382461 | 0.0926723 |

The error metrics are much higher here, so it’s safe to say our random forest did a good job.

Where do we go from here? Well, we haven’t tried it yet, but we may further improve our forecasts with some hyperparameter tuning. Also, just between the two of us, maybe random forest is pretty good, but not the best model for the job. We could try others or an ensemble of models.

If you take away one thing from this post today, let it be this: We can do effective time series forecasting with machine learning without whipping out big guns like recurrent neural networks. All it takes is a little pre- and post-processing. So why not include random forest in your arsenal the next time you do forecasting (or procrastinate on doing your taxes)? 