Time series cross-validation is important part of the toolkit for good evaluation of forecasting models. forecast::tsCV makes it straightforward to implement, even with different combinations of explanatory regressors in the different candidate models for evaluation.

Suprious correlation between time series is a well documented and mocked problem, with [Tyler Vigen’s educational website on the topic](https://www.tylervigen.com/spurious-correlations) (“per capita cheese consumption correlated with number of people dying by becoming entangled in their bedsheets”) even spawning a whole book of humourous examples.

Identifying genuinely-correlated series can be immensely helpful for time series forecasting. Forecasting is hard, and experience generally shows that complex causal models don’t do as well as much simpler methods. However, a well chosen small set of “x regressors” can improve forecasting performance in many situations. I have been investigating one of those situations for a future blog post on forecasting unemployment rates. One method I’ll be using is time series cross-validation, as well described in [this Rob Hyndman post](https://robjhyndman.com/hyndsight/tscv/). The implementation was sufficiently non-trivial in my real-data case that I’m writing today a separate post with simulated data to be sure I’m doing it right.

**Simulated data**

So here’s my simulated data. I have made three time series which are causally related to eachother:

* x is an ARIMA(1, 1, 1) process, which means that if you took its first difference (the change from day to day rather than the original observation) it would be an ARMA(1, 1) process with an autoregression coefficient of 0.5 and a moving average coefficient of 0.5
* y has a random component which is generated in a similar way to x, and a structural component which is a simple multiplier of x
* z has a similar random component again and a structural component which is a simple multiplier of the lagged values of y.

In other words, x causes y and y causes z via a delay. Our job is to forecast y. We would expect the best model to do this to be one that uses x as an explanatory variable and which ignores z. We would expect z to be a tempting but ultimately unhelpful explanatory variable in a model of y.

Here’s what the data looks like:

Generated with this:

library(forecast)

library(ggplot2)

# Simulate data:

set.seed(125)

n <- 100

x <- ts(cumsum(arima.sim(list(ar = 0.5, ma = 0.5), n = n)))

y = ts(0.6 \* x + cumsum(arima.sim(list(ar = 0.5, ma = 0.5), n = n)))

z = ts(0.6 \* y + cumsum(arima.sim(list(ar = 0.5, ma = 0.5), n = n)))

z <- c(0, z[1:(n - 1)])

d <- cbind(x, y, z)

# Exploratory plot:

autoplot(d) +

ggtitle("Three simulated, related, time-series",

"x causes y and y causes z; our job is to forecast y.") +

labs(colour = "", y ="")

If this were a real analysis, I would always start with the partial auto correlations and cross correlations. Auto-correlation means the correlation of a time series with lagged versions of itself. The “partial” means we look at these correlations after controlling for the higher order lags (for example, we look at the autocorrelation at lag of 2 time periods, after controlling for the autocorrelation at lag 1). Cross correlations are the same statistics, but with the variously lagged versions of another variable instead of with itself. Here are the partial auto correlations of our three simulated series:

To the experienced eye it is immediately obvious from this PACF plot, if not from the original simple plot, that these time series are non-stationary and are heavily correlated with themselves – that is, knowing its value at time t gives you a strong indication of its location time t + 1. The giveaway in the PACF plots is the tall line indicating high autocorrelation (close to 1.0) at lag 1 for each of the three series. This non-stationarity hides the other relationships in the data and gets in the way of effective modelling, and the standard response is to try “differencing” the series. This technique alone would eliminate the big majority of the mis-inferences in the “spurious correlation” time series collection.

Here are the PACF plots for the first-differenced versions of these series:

The series are still heavily correlated with themselves – each with an autocorrelation of about 0.6 at lag 1 – but no longer so much so that they are obviously non-stationary. We can now see some of the secondary relationships, including correlations between the various variables at several lags that suggest there could be something going on between them (as in fact we know is the case).

Those PACF plots use the handy ggPacf function that comes with Hyndman’s forecast R package.

ggPacf(d) +

theme(panel.grid = element\_blank()) +

ggtitle("Partial autocorrelations and cross correlations",

"Original series")

ggPacf(diff(d)) +

theme(panel.grid = element\_blank()) +

ggtitle("Partial autocorrelations and cross correlations",

"First differenced series")

**Modelling**

When I first learnt time series analysis in the 1990s we used to use PACF plots to try to infer the order of the auto-regressive and moving-average components of a worthwhile model to fit. These days we just use highly effective and tested automated algorithms such as that behind the forecast::auto.arima function. I’m going to fit four models for my hypothetical forecast job:

1. univariate model just using y
2. x as an explanatory variable
3. z as an explanatory variable
4. both x and z as explanatory variables

For each model I am going to manually specify the level of differencing as one lag only, so my resulting models can all be compared on the same basis.

Because I made up the data, I happen to know that model 2 is the “correct” specification. I was pleased to see that fitting the four models and comparing their AIC agreed with this foreknown correct answer. The AIC of mod2 is lowest, correctly implying that any reduction in deviance from including z in the model is not justified by the use of an additional degree of freedom, whether z is included by itself or in addition to x:

|  | **df** | **AIC** |
| --- | --- | --- |
| mod1 | 3 | 313.1799 |
| mod2 | 5 | 290.6071 |
| mod3 | 4 | 315.0472 |
| mod4 | 6 | 292.3268 |

The models were fit with the code below – the perfect example of fitting enormously complex models with a single line of code each:

mod1 <- auto.arima(y, d = 1)

mod2 <- auto.arima(y, xreg = x, d = 1)

mod3 <- auto.arima(y, xreg = z, d = 1)

mod4 <- auto.arima(y, xreg = cbind(x, z), d = 1)

knitr::kable(AIC(mod1, mod2, mod3, mod4))

**Time series cross-validation**

OK, so the simple expedient of comparing AIC values worked in this case, but my actual motivation for today was to check that time series cross-validation would similarly pick the known-best model in a situation comparing time series forecasting models with different numbers (or no) explanatory variables. Cross-validation for time series is more complex than for cross-sectional data because we can’t simply divide the data into training and test sets without taking into account the intrinsic connections of data in each time period with its neighbours on either side. It is well explained in [the Hyndman blog post I linked to earlier](https://robjhyndman.com/hyndsight/tscv/). The intuition is well presented in this image from that post:



Basically you use the data from the blue dots to forecast the red dots.

The forecasts::tsCV function implements this approach. It takes as its main argument a function that returns an object of class forecast. The complication (such as it is) in my case comes from the need to write a function that combines fitting an auto.arima model and then converting it to a forecast object. Building on an example provided by (again) Hyndman, here’s my version of this which I think I will find myself re-using and hence will put in the frs package for future use:

#---------Cross validation---------------

#' auto.arima forecast function for time series cross validation

#'

#' adapted from https://gist.github.com/robjhyndman/d9eb5568a78dbc79f7acc49e22553e96

aafc <- function(y, h, xreg = NULL, ...){

if(!is.null(xreg)){

ncol <- NCOL(xreg)

X <- matrix(xreg[1:length(y), ], ncol = ncol)

if(NROW(xreg) < length(y) + h)

stop("Not enough xreg data for forecasting")

newX <- matrix(xreg[length(y) + (1:h), ], ncol = ncol)

fit <- auto.arima(y, xreg=X, ...)

return(forecast(fit, xreg = newX, h = h))

} else {

fit <- auto.arima(y, ...)

return(forecast(fit, h = h))

}

}

# this CV takes about 50 seconds

system.time({

aa1\_cv <- tsCV(y, aafc, d = 1)

aa2\_cv <- tsCV(y, aafc, xreg = x, d = 1)

aa3\_cv <- tsCV(y, aafc, xreg = z, d = 1)

aa4\_cv <- tsCV(y, aafc, xreg = cbind(x, z), d = 1)

})

The tsCV function returns a numerical time series object containing the forecast errors as a vector (if forecasting is for only one period forward, as is the case in my example). Interpreting and presenting the results takes a bit of care. All my models return some NAs in the forecast errors after cross-validation, including the final observation in each case. The tsCV helpfile includes the helpful comment that in the output of tsCV:

The time index corresponds to the last period of the training data

So if we want to line up the forecast errors to the actual values they apply to, we need to be careful! In my case, I need to knock out the *first* value of the original time series, and the *last* value of the forecast errors, if I want to line them up (eg for use in the accuracy function). Here’s how I did that:

rbind(accuracy(ts(y[-1] - aa1\_cv[-n]), ts(y[-1])),

accuracy(ts(y[-1] - aa2\_cv[-n]), ts(y[-1])),

accuracy(ts(y[-1] - aa3\_cv[-n]), ts(y[-1])),

accuracy(ts(y[-1] - aa4\_cv[-n]), ts(y[-1]))) %>%

as.data.frame() %>%

mutate(model = c("Univariate",

"With x regressor",

"With z regressor",

"Both x and z as regressors")) %>%

dplyr::select(model, everything()) %>%

knitr::kable()

I’m not sure this is the best way to evaluate those forecast errors (ie by using forecast::accuracy) but it’s a method that fits in with how I think about it and returns lots of performance metrics at once. Here are the results:

| **model** | **ME** | **RMSE** | **MAE** | **MPE** | **MAPE** | **ACF1** | **Theil’s U** |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Univariate | -0.0238600 | 1.249621 | 0.9706499 | 28.28211 | 47.15084 | 0.0725987 | 0.4626243 |
| With x regressor | -0.0560274 | 1.177935 | 0.9507106 | 20.91153 | 45.24956 | 0.1673370 | 0.3950919 |
| With z regressor | -0.0156043 | 1.245264 | 0.9824904 | 29.08816 | 47.88510 | 0.0453596 | 0.5016008 |
| Both x and z as regressors | -0.0909384 | 1.212325 | 0.9823389 | 22.85555 | 46.54281 | 0.1491128 | 0.4872987 |

Model 2, with just the x regressor, has the lowest root mean square error, mean absolute error, mean percentage error, mean absolute percentage error and Theil’s U. The autocorrelation of its errors is relatively high, but I don’t think that matters. It has more biased forecasts (mean error) than the models that don’t include x but depending on context we might accept that in return for being closer on average. Overall, this method comes up with good support for the correct underlying model.

Just to check I haven’t mangled anything, gotten the signs the wrong way around, etc I calculated a few of these stats by hand directly from the tsCV output:

> # mean error of univariate model:

> mean(aa1\_cv, na.rm = TRUE)

[1] -0.02386004

>

> # root mean square error of x regressor model

> sqrt(mean(aa2\_cv ^ 2, na.rm = TRUE))

[1] 1.177935

>

> # mean absolute error of z regressor model

> mean(abs(aa3\_cv), na.rm = TRUE)

[1] 0.9824904

Final word on this is that I tried running this program with various other settings (eg just changing the random seed from 125 to another number, or making the relationships between the variables a little weaker) and sometimes the AIC was slightly better at picking the correct model than was the cross-validation. Neither method is fool-proof – remember, forecasting is hard, there is always a lot of noise around the signal!

OK, c’est tout. Today’s key thought again: Time series cross-validation is important part of the toolkit, and forecast::tsCV makes it straightforward to implement.