It has recently been confronted to a kind of data set and problem that I was not even aware existed:  
intermittent demand data. Intermittent demand arises when the demand for a certain good arrives  
sporadically. Let’s take a look at an example, by analyzing the number of downloads for the {RDieHarder}  
package:

library(tidyverse)

library(tsintermittent)

library(nnfor)

library(cranlogs)

library(brotools)

rdieharder <- cran\_downloads("RDieHarder", from = "2017-01-01")

ggplot(rdieharder) +

geom\_line(aes(y = count, x = date), colour = "#82518c") +

theme\_blog()

Let’s take a look at just one month of data, because the above plot is not very clear, because of  
the outlier just before 2019… I wonder now, was that on Christmas day?

rdieharder %>%

filter(count == max(count))

## date count package

## 1 2018-12-21 373 RDieHarder

Not exactly on Christmas day, but almost! Anyways, let’s look at one month of data:

january\_2018 <- rdieharder %>%

filter(between(date, as.Date("2018-01-01"), as.Date("2018-02-01")))

ggplot(january\_2018) +

geom\_line(aes(y = count, x = date), colour = "#82518c") +

theme\_blog()

Now, it is clear that this will be tricky to forecast. There is no discernible pattern,  
no trend, no seasonality… nothing that would make it “easy” for a model to learn how to forecast  
such data.

This is typical intermittent demand data. Specific methods have been developed to forecast such  
data, the most well-known being Croston

A function to estimate such models is available in the {tsintermittent} package, and another package, {nnfor}, which uses Neural Networks to forecast time series data.  
I am going to use both to try to forecast the intermittent demand for the {RDieHarder} package  
for the year 2019.

Let’s first load these packages:

library(tsintermittent)

library(nnfor)

And as usual, split the data into training and testing sets:

train\_data <- rdieharder %>%

filter(date < as.Date("2019-01-01")) %>%

pull(count) %>%

ts()

test\_data <- rdieharder %>%

filter(date >= as.Date("2019-01-01"))

Let’s consider three models; a naive one, which simply uses the mean of the training set as the  
forecast for all future periods, Croston’s method, and finally a Neural Network from the {nnfor}  
package:

naive\_model <- mean(train\_data)

croston\_model <- crost(train\_data, h = 163)

nn\_model <- mlp(train\_data, reps = 1, hd.auto.type = "cv")

## Warning in preprocess(y, m, lags, keep, difforder, sel.lag,

## allow.det.season, : No inputs left in the network after pre-selection,

## forcing AR(1).

nn\_model\_forecast <- forecast(nn\_model, h = 163)

The crost() function estimates Croston’s model, and the h argument produces the  
forecast for the next 163 days. mlp() trains a multilayer perceptron, and the hd.auto.type = "cv"  
argument means that 5-fold cross-validation will be used to find the best number of hidden nodes. I  
then obtain the forecast using the forecast() function. As you can read from the Warning message  
above, the Neural Network was replaced by an auto-regressive model, AR(1), because no inputs were  
left after pre-selection… I am not exactly sure what that means, but if I remove the big outlier  
from before, this warning message disappears, and a Neural Network is successfully trained.

Here is the function:

mase <- function(train\_ts, test\_ts, outsample\_forecast){

naive\_insample\_forecast <- stats::lag(train\_ts)

insample\_mae <- mean(abs(train\_ts - naive\_insample\_forecast), na.rm = TRUE)

error\_outsample <- test\_ts - outsample\_forecast

ase <- error\_outsample / insample\_mae

mean(abs(ase), na.rm = TRUE)

}

It is now easy to compute the models’ accuracies:

mase(train\_data, test\_data$count, naive\_model)

## [1] 1.764385

mase(train\_data, test\_data$count, croston\_model$component$c.out[1])

## [1] 1.397611

mase(train\_data, test\_data$count, nn\_model\_forecast$mean)

## [1] 1.767357

Croston’s method is the one that performs best from the three. Maybe surprisingly, the naive method  
performs just as well as the Neural Network! (or rather, the AR(1) model) Let’s also plot the predictions  
with the true values from the test set:

test\_data <- test\_data %>%

mutate(naive\_model\_forecast = naive\_model,

croston\_model\_forecast = croston\_model$component$c.out[1],

nn\_model\_forecast = nn\_model\_forecast$mean) %>%

select(-package) %>%

rename(actual\_value = count)

test\_data\_longer <- test\_data %>%

gather(models, value,

actual\_value, naive\_model\_forecast, croston\_model\_forecast, nn\_model\_forecast)

## Warning: attributes are not identical across measure variables;

## they will be dropped

ggplot(test\_data\_longer) +

geom\_line(aes(y = value, x = date, colour = models)) +

theme\_blog()

Just to make sure I didn’t make a mistake when writing the mase() function, let’s use the  
accuracy() function from the {forecast} package and compare the result for the Neural Network:

library(forecast)

accuracy(nn\_model\_forecast, x = test\_data$actual\_value)

## ME RMSE MAE MPE MAPE MASE ACF1

## Training set 0.001929409 14.81196 4.109577 NaN Inf 0.8437033 0.05425074

## Test set 8.211758227 12.40199 8.635563 -Inf Inf 1.7673570 NA

The result is the same, so it does seem like the naive method is not that bad, actually! Now, in  
general, intermittent demand series have a lot of 0 values, which is not really the case here. I  
still think that the methodology fits to this particular data set.