Alternative Models for Part Demand Forecasting

Introduction

This notebook follows the "Parts Forecasting Exploration" notebook and must be read in that context.

That notebook explored the parts sales history of the company and considered basic approaches to forecasting sales of the thousands of part-lines in stock. Most have very few sales per year. The obvious approach was to use simple weighted averages [(3 x yr2) + (2 x yr2) + 1 x yr3)]/6 for parts with few sales and build an ARIMA forecast where possible. The simple weighted averages turned out to be as good as ARIMA forecasts even for higher selling parts, so that weighted average has become the benchmark forecast to be bettered.

Since simple weighted averages have been so succesful this notebook investigates other statistical models which are not strictly 'time aware'. It does not investigate deep learning, there is a separate notebook dedicated to that approach.

Executive Summary

Various GLM models are developed and compared with a Bagged CART and an improved approach to weighted averages. All are tested on a test set, but are found not to be substantially better than a simple weighted average. The report recommends time be spent investigating a deep learning model.

Input Data

The data for this notebook is from the "Parts Forecasting Exploration" notebook. Please view that notebook for an introduction to the data.

```
setwd("C:\\Users\\User\\OneDrive\\Oliver\\0_OM\\Training\\DeepLearning_Udacity\\LSTM")
library(dplyr)
library(readr)
forecasts_fromAllMonths <- read_csv("pca_parts_forecasts_fromAllMonths.csv")
parts_decode            <- read_csv("pca_parts_parts_decode.csv")
parts_frequent          <- read_csv("pca_parts_parts_frequent.csv")
clusters_fromAllMonths_revised <- read_csv("pca_parts_clusters_fromAllMonths_revised.csv")
clusters_fromAllMonths_revised <- clusters_fromAllMonths_revised %>% ungroup()
```

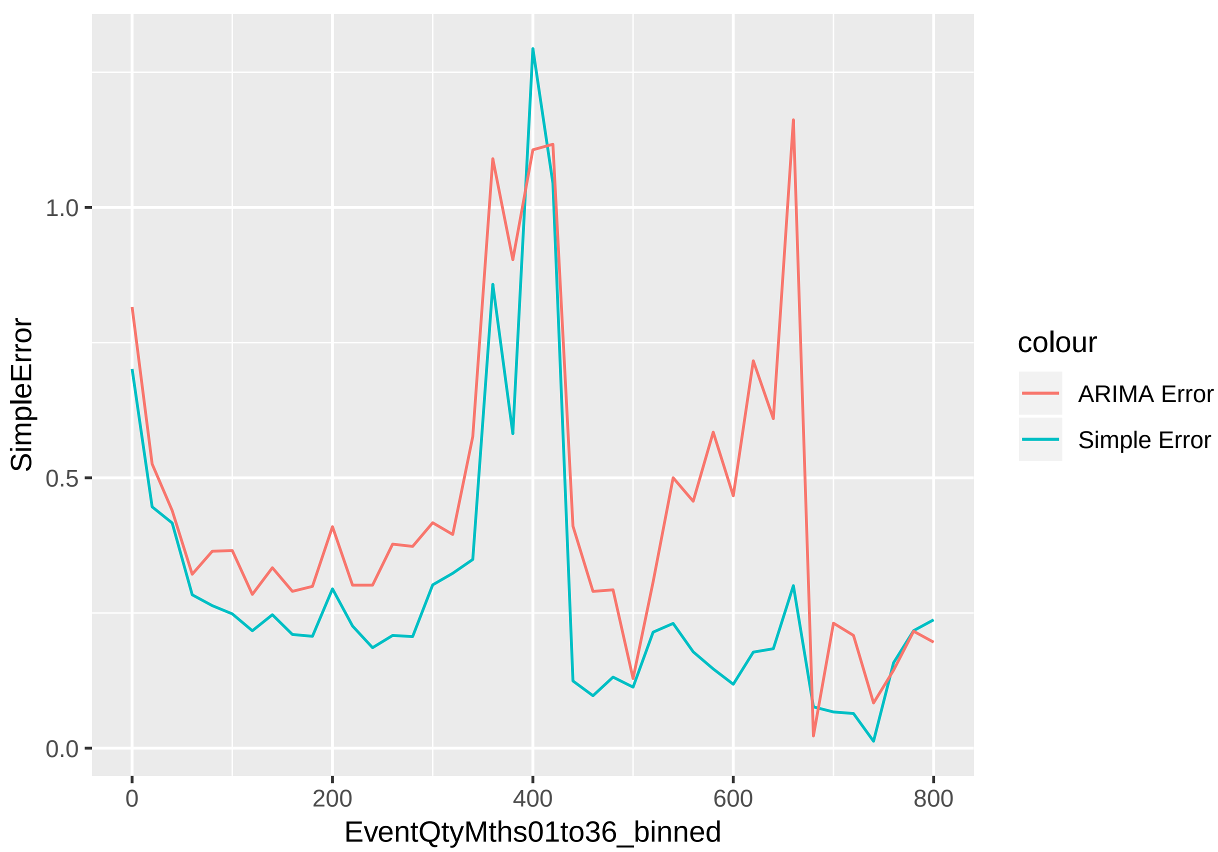
Reformat the data so we have easy access to the simple forecasts and the respective correct future sales.

```
#replace NA with zero
library(dplyr)
library(tidyr)
library(lubridate)
library(ggplot2)
library(caret)

#lots of packages, ensure dplyr gets priority
select    <- dplyr::select
filter    <- dplyr::filter
summarise <- dplyr::summarise

#prepare data
```

```r
simple_model_dat      <- forecasts_fromAllMonths %>% ungroup()
simple_model_format   <- sapply(simple_model_dat, class)
#Set NA to zero
for (i in 1:ncol(simple_model_dat)){
  if(simple_model_format[i] != "Date"){
  isna_idx <- is.na(simple_model_dat[,i])
  simple_model_dat[isna_idx, i] <- 0
  }
}

simple_model_dat <- simple_model_dat %>%
  # get RRP per part_number
  inner_join(y = parts_decode %>% select(PART_NUMBER, R_R_PRICE),
           by="PART_NUMBER") %>%
  # join to self, but 12mths later, sales0to12 for that period = our correct forecasts
  inner_join(y  = simple_model_dat %>% select(PART_NUMBER,
                                              ForecastFromMth = ForecastFromMthLess12,
                                              CorrectForecast = SalesQtyMths01to12),
             by = c("ForecastFromMth", "PART_NUMBER")) %>%

  # get years in use, then create all the new columns for analysis
  left_join(y=parts_frequent %>% select(PART_NUMBER, FIRST_SALE), by="PART_NUMBER") %>%

    # Log the data from Poisson to Gaussian (ish)


    # get required fields
    mutate(YearsInUse  = round(as.numeric(difftime(ForecastFromMth,
                                                     FIRST_SALE,
                                                     units="days"))/365,0),
           ValueToOrder= R_R_PRICE * CorrectForecast,
           SimpleError = R_R_PRICE * abs(SimpleForecast - CorrectForecast),
           ARIMAError  = R_R_PRICE * abs(ARIMAforecast  - CorrectForecast),

           # we use bins for charting, each being 20 invoices (events) in the previous 3yrs
           # in other words, the numbe rof data points available to the models.
           EventQtyMths01to36_binned = ((EventQtyMths01to12+
                                         EventQtyMths13to24+
                                         EventQtyMths25to36) %/% 20)*20,

           # add a single column to represent the trend, as these simple models have no time concept
           Trend = case_when(   SalesQtyMths01to12 > 1.1 * SalesQtyMths13to24
                              & SalesQtyMths13to24 > 1.1 * SalesQtyMths25to36 ~ "Up",
                                SalesQtyMths01to12 < 0.9 * SalesQtyMths13to24
                              & SalesQtyMths13to24 < 0.9 * SalesQtyMths25to36 ~ "Down",
                              TRUE ~ "None"),

           # forecasts will be different according to season, so need a new feature, month as integer
           MonthOfForecast = month(ForecastFromMth)
           )%>%

            # exclude attempts at predicting the first year's sales. Always impossible.
            # Now, because we rounded YearsInUse (as opposed to ceiling or floor) this filter
```

```r
         # effectively demands at least 6months sales before presenting data for forecasting
         # approx 6,000 rows are removed from 90,000, so a considerable number.
         filter(YearsInUse>0) %>%
         select(PART_NUMBER, CorrectForecast, R_R_PRICE, ValueToOrder,
                ARIMAforecast, ARIMAError, ARIMAsigma2, SimpleForecast, SimpleError,
                EventQtyMths01to36_binned, MonthOfForecast, ForecastFromMth,
                YearsInUse, Trend,
                SalesQtyMths01to12, SalesQtyMths13to24, SalesQtyMths25to36, # sales history
                EventQtyMths01to12, EventQtyMths13to24, EventQtyMths25to36) # event history

#get sum error within each bin, for all part_numbers
simple_model_sumry <- simple_model_dat %>%
                      group_by(EventQtyMths01to36_binned) %>%
                        summarise(SimpleError= sum(SimpleError)/sum(ValueToOrder),
                                   ARIMAError = sum(ARIMAError) /sum(ValueToOrder))%>%
                          arrange(EventQtyMths01to36_binned)
```

```r
# chart cumulative error vs number of sales vents over past 3yrs (qty of data points)
p <- ggplot(data = simple_model_sumry, aes(x=EventQtyMths01to36_binned)) +
     geom_line(aes(y = SimpleError, colour = "Simple Error"))  +
     geom_line(aes(y = ARIMAError,  colour = "ARIMA Error"))

p
```



The simple 3,2,1 model is surprisingly good, mostly being within 10% of the actual sales demand. Extraordinary. Whereas the ARIMA models are consistently under ordering. It is curious that as the number of data points exceeds 400, both models order more parts. By 600 data points in the past 36mths, both models are substantially over ordering. This could be simple regression to the mean, frequently sold parts tend to fall out of usage more often then carry on being popular.

```
errors_sumry <- simple_model_dat %>%
                  group_by(PART_NUMBER) %>%
                    summarise(ARIMAError  = sum(ARIMAError),
                              SimpleError = sum(SimpleError)) %>%
                      arrange(desc(ARIMAError))

paste0("How concentrated are the errors in selected part numbers? (Simple Forecast Error)")
```

```
## [1] "How concentrated are the errors in selected part numbers? (Simple Forecast Error)"
```

```
quantile(errors_sumry$SimpleError, seq(from=0,to=1,by=0.1))
```

```
##         0%        10%        20%        30%        40%        50%
##      0.000    170.870    596.272   1219.655   2346.552   4082.775
##        60%        70%        80%        90%       100%
##   6992.832  13522.867  26793.448  53080.505 547282.080
```

```
cat("\n")
```

```
paste0("How concentrated are the errors in selected part numbers? (ARIMA Forecast Error)")
```

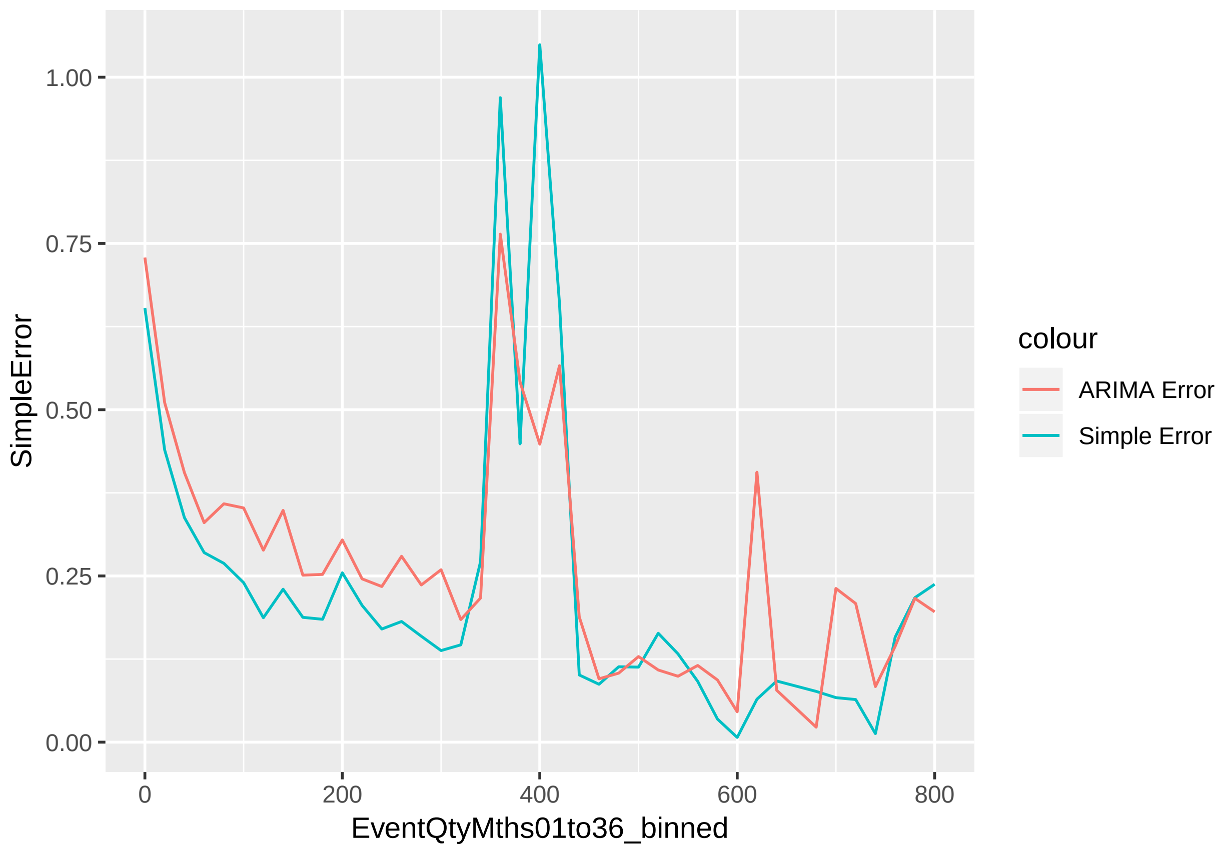
```
## [1] "How concentrated are the errors in selected part numbers? (ARIMA Forecast Error)"
```

```
quantile(errors_sumry$ARIMAError,  seq(from=0,to=1,by=0.1))
```

```
##          0%         10%         20%         30%         40%         50%
##      0.0000    213.6587    658.7458   1474.4825   2825.7256   4638.0576
##         60%         70%         80%         90%        100%
##   8233.2171  15157.4362  30211.6133  63960.4233 965994.7030
```

The vast majority of the financial error is in just 10% of parts. For ARIMA this is more extreme. ARIMA appears to be getting a small number of part numbers very wrong. Let's revisit the chart, but excluding the top 10 part numbers by error (out if thousands of part numbers)

```
q <- ggplot(data = simple_model_dat %>%
                    #exclude top 10 errored part numbers!
                    filter(!(PART_NUMBER %in% errors_sumry$PART_NUMBER[1:10])) %>%
                      group_by(EventQtyMths01to36_binned) %>%
                          summarise(SimpleError= sum(SimpleError)/sum(CorrectForecast*R_R_PRICE),
                                    ARIMAError = sum(ARIMAError) /sum(CorrectForecast*R_R_PRICE))%>%
                            arrange(EventQtyMths01to36_binned),
            aes(x=EventQtyMths01to36_binned)) +
     geom_line(aes(y = SimpleError, colour = "Simple Error"))  +
     geom_line(aes(y = ARIMAError,  colour = "ARIMA Error"))

q
```



OK, that fixed the majority of the ARIMA error anomaly. 10 faulty forecasts out of thousands.

Regression Models

Interestly, we are not required to use a time dependent model for forecasting. For example, we could construct a model which is fed with the following parameters:

It would be interesting to see how a simple random forest would work. A random forest model has been built for comparison with the wavenet model. We'll make this model PART_NUMBER agnostic, it will have to forecast using sales, invoices etc, but not the PART_NUMBER.

```
# select data for the models, exclude other forecasts already in the data. This is not an ensemble mode
model_dat <- simple_model_dat %>%
                mutate(ValueToOrder = CorrectForecast * R_R_PRICE)

# Use 75% of data for training, 25% for test.
# Randomly sample for training part numbers
model_dat_partnums  <- model_dat %>% distinct(PART_NUMBER)
train_size          <- round(nrow(model_dat_partnums)*0.75,0)
set.seed(20180820)
train_partsnums     <- sample(model_dat_partnums$PART_NUMBER, train_size, replace=FALSE)

#get train and test sets
model_dat_train <- model_dat %>%
                     filter(PART_NUMBER %in% train_partsnums)
```

```
model_dat_test  <- model_dat %>%
                     filter(!(PART_NUMBER %in% train_partsnums))
```

Function to Train the GLM

Let's build a function to test the various ways we could build a glm for this data:

```
library(car)
library(caret)
library(MASS)

# MASS doesn't play nice with other packages.
# So ensure dplyr is default for important commands
select    <- dplyr::select
filter    <- dplyr::filter
summarise <- dplyr::summarise

get_model_trained <- function(training_data_df,
                              # poisson or gaussian residuals ?
                              residuals_family_chr,
                              # whether we are modelling CorrectForecast or ValueToOrder
                              target_chr,
                              # whether to log(x+1) transform counts?
                              # NB CorrectForecast, SalesQty, SalesEvent are counts
                              log_of_counts_lc,
                              #whether we provide the model with simple forecasts, like stacked model
                              provide_simples_lc){
                              # returns model object
  require(car)
  require(MASS)
  #arrange data
  if(target_chr == "CorrectForecast"){
    training_data_df$Target <- training_data_df$CorrectForecast
  }else{
    training_data_df$Target <- training_data_df$ValueToOrder
  }

  if(log_of_counts_lc){
    training_data_df$Target             <- log(training_data_df$Target+1)
    training_data_df$SalesQtyMths01to12 <- log(training_data_df$SalesQtyMths01to12+1)
    training_data_df$SalesQtyMths13to24 <- log(training_data_df$SalesQtyMths13to24+1)
    training_data_df$SalesQtyMths25to36 <- log(training_data_df$SalesQtyMths25to36+1)
    training_data_df$EventQtyMths01to12 <- log(training_data_df$EventQtyMths01to12+1)
    training_data_df$EventQtyMths13to24 <- log(training_data_df$EventQtyMths13to24+1)
    training_data_df$EventQtyMths25to36 <- log(training_data_df$EventQtyMths25to36+1)
  }


  if(provide_simples_lc){
    training_data_df <- training_data_df %>% select(-PART_NUMBER,     -ForecastFromMth,
                                                    -ARIMAforecast,   -ARIMAError,  -ARIMAsigma2,
                                                    -SimpleError,
                                                    -CorrectForecast, -ValueToOrder)
```

```r
  }else{
    training_data_df <- training_data_df %>% select(-PART_NUMBER,     -ForecastFromMth,
                                                   -ARIMAforecast,   -ARIMAError,  -ARIMAsigma2,
                                                   -SimpleError,     -SimpleForecast,
                                                   -CorrectForecast, -ValueToOrder)
  }

  if(log_of_counts_lc & provide_simples_lc){
    training_data_df$SimpleForecast <- log(training_data_df$SimpleForecast+1)
  }

  if(residuals_family_chr == "poisson"){
    #must pass integer to poisson family.
    training_data_df$Target <- round(training_data_df$Target,0)
  }

  #get first model:
  suppressMessages(model_step1 <- glm(Target ~.,
                                      family = residuals_family_chr,
                                      data   = training_data_df))

  #get top 100 outliers from first model
  outliers <- as.numeric(attributes(outlierTest(model_step1, n.max=100)$p)$names)

  #Rebuild model without outliers
  suppressMessages(model_step2 <- glm(Target ~.,
                                      family = residuals_family_chr,
                                      data   = training_data_df[-outliers,]))

  #optimise
  model_step3 <- stepAIC(model_step2, direction=c("both"), trace=FALSE)

  return(model_step3)
}
```

Function to Test GLMs

```r
# Let's see how the model performs on the test data vs the simple forecast and ARIMA
## prepare the test data...

## function to generate chart comparing results of the model vs the simple and ARIMA forecasts
get_model_tested <- function(model_object,
                             testing_data_df,
                             target_chr,
                             log_of_counts_lc){
                             #returns dataframe

  #log the test data, if required
  if(log_of_counts_lc){
    testing_data_df$SalesQtyMths01to12 <- log(testing_data_df$SalesQtyMths01to12+1)
    testing_data_df$SalesQtyMths13to24 <- log(testing_data_df$SalesQtyMths13to24+1)
    testing_data_df$SalesQtyMths25to36 <- log(testing_data_df$SalesQtyMths25to36+1)
```

```r
    testing_data_df$EventQtyMths01to12 <- log(testing_data_df$EventQtyMths01to12+1)
    testing_data_df$EventQtyMths13to24 <- log(testing_data_df$EventQtyMths13to24+1)
    testing_data_df$EventQtyMths25to36 <- log(testing_data_df$EventQtyMths25to36+1)
    testing_data_df$SimpleForecast     <- log(testing_data_df$SimpleForecast+1)
  }

  ## get the predictions
  model_results <- predict(object  = model_object,
                           newdata = testing_data_df)

  # un log the results, where required
  if(log_of_counts_lc){
    model_results <- exp(model_results)-1
  }

  #round results
  model_results <- round(model_results,0)

  # union data with predictions
  model_results <- cbind(testing_data_df, model_results)
  colnames(model_results) <- c(colnames(testing_data_df),"Modelled")

  if(target_chr=="CorrectForecast"){
    model_results <- model_results %>%
                     mutate(model_error = abs(Modelled-CorrectForecast)*R_R_PRICE)
  }else{
    model_results <- model_results %>%
                     mutate(model_error = abs(Modelled-ValueToOrder))
  }

  #get sum error within each binned, for all part_numbers
  model_results_sumry <- model_results %>%
                          summarise(Simple_Error= sum(SimpleError)/sum(ValueToOrder),
                                    ARIMA_Error = sum(ARIMAError) /sum(ValueToOrder),
                                    Model_Error = sum(model_error)/sum(ValueToOrder))
  return(model_results_sumry)
}
```

Compile some model configurations to examine

```r
# training data and test data always same, so exclude from configurations.
Configurations <- NULL
Configurations <- data.frame(residuals_family_chr  = character(),
                             target_chr            = character(),
                             log_of_counts_lc      = logical(),
                             provide_simples_lc    = logical(),
                             Simple_Error          = numeric(),
                             ARIMA_Error           = numeric(),
                             Model_Error           = numeric(),
                             stringsAsFactors      = FALSE)

                            # family    target            log    stacked simple arima model
Configurations[ 1,] <- list("gaussian", "CorrectForecast", FALSE, FALSE,  NA,    NA,   NA)
Configurations[ 2,] <- list("gaussian", "CorrectForecast", FALSE, TRUE,   NA,    NA,   NA)
```

```r
Configurations[ 3,] <- list("gaussian", "CorrectForecast", TRUE,  FALSE,  NA,    NA,   NA)
Configurations[ 4,] <- list("gaussian", "CorrectForecast", TRUE,  TRUE,   NA,    NA,   NA)
Configurations[ 5,] <- list("gaussian", "ValueToOrder",    FALSE, FALSE,  NA,    NA,   NA)
Configurations[ 6,] <- list("gaussian", "ValueToOrder",    FALSE, TRUE,   NA,    NA,   NA)
Configurations[ 7,] <- list("gaussian", "ValueToOrder",    TRUE,  FALSE,  NA,    NA,   NA)
Configurations[ 8,] <- list("gaussian", "ValueToOrder",    TRUE,  TRUE,   NA,    NA,   NA)
Configurations[ 9,] <- list("poisson",  "CorrectForecast", FALSE, FALSE,  NA,    NA,   NA)
Configurations[10,] <- list("poisson",  "CorrectForecast", FALSE, TRUE,   NA,    NA,   NA)
Configurations[11,] <- list("poisson",  "ValueToOrder",    FALSE, FALSE,  NA,    NA,   NA)
Configurations[12,] <- list("poisson",  "ValueToOrder",    FALSE, TRUE,   NA,    NA,   NA)

#NB, cannot use poisson family on log of counts data, poisson family expects integer.
```

Use the functions on the range of configurations

```r
pb <- txtProgressBar(min = 1, max = nrow(Configurations), style = 3)

for(i in 1:nrow(Configurations)){
  #print(i)
  setTxtProgressBar(pb, i)

  model_object <- get_model_trained(
                      training_data_df     = model_dat_train,
                      residuals_family_chr = Configurations[i,]$residuals_family_chr,
                      target_chr           = Configurations[i,]$target_chr,
                      log_of_counts_lc     = Configurations[i,]$log_of_counts_lc,
                      provide_simples_lc   = Configurations[i,]$provide_simples_lc
                      )

  model_results_sumry <- get_model_tested(
                         model_object     = model_object,
                         testing_data_df  = model_dat_test,
                         target_chr       = Configurations[i,]$target_chr,
                         log_of_counts_lc = Configurations[i,]$log_of_counts_lc)

  Configurations[i,]$Simple_Error <- round(model_results_sumry$Simple_Error,4)
  Configurations[i,]$ARIMA_Error  <- round(model_results_sumry$ARIMA_Error,4)
  Configurations[i,]$Model_Error  <- round(model_results_sumry$Model_Error,4)
}
```

```r
library(knitr)
kable(Configurations %>% arrange(Model_Error) %>% select(-ARIMA_Error))
```

residuals_family_chr	target_chr	log_of_counts_lc	provide_simples_lc	Simple_Error	Model_Error
gaussian	CorrectForecast	TRUE	TRUE	0.3675	0.3865
gaussian	CorrectForecast	TRUE	FALSE	0.3675	0.3884
gaussian	CorrectForecast	FALSE	TRUE	0.3675	0.5247
gaussian	CorrectForecast	FALSE	FALSE	0.3675	0.6390
gaussian	ValueToOrder	TRUE	FALSE	0.3675	0.7736
gaussian	ValueToOrder	TRUE	TRUE	0.3675	0.7758
gaussian	ValueToOrder	FALSE	FALSE	0.3675	0.9441
gaussian	ValueToOrder	FALSE	TRUE	0.3675	0.9461
poisson	ValueToOrder	FALSE	FALSE	0.3675	0.9936
poisson	ValueToOrder	FALSE	TRUE	0.3675	0.9936

residuals_family_chr	target_chr	log_of_counts_lc	provide_simples_lc	Simple_Error	Model_Error
poisson	CorrectForecast	FALSE	TRUE	0.3675	1.3009
poisson	CorrectForecast	FALSE	FALSE	0.3675	1.3042

Ouch! The simple model is still the best. Providing the Simple forecast to the glm did not help. The best glm without taking the simple forecast as a feature was good ol' fashioned log(x+1) of inputs, then use gaussian model. Poisson family models were all awful.

Modelling the CorrectForecast qty (then multiplying by RRP) appears a much better strategy than modelling ValueToOrder directly.

Other models and features are now worth investigating.

New Features: Clusters

We'll rebuild the glm model with the optimal configuration, but this time with access to the cluster feature as a factor. We also return the outliers to the dataset before recalculating outliers.

```
# Get clusters
model_dat_clust <- simple_model_dat %>%
                      left_join(y = clusters_fromAllMonths_revised,
                                by=c("PART_NUMBER", "ForecastFromMth")) %>%
                        #replace NA cluster with 0
                        mutate(CLUSTER = ifelse(is.na(CLUSTER),0,CLUSTER))

#make cluster a factor
model_dat_clust$CLUSTER <- as.factor(model_dat_clust$CLUSTER)

model_dat_clust_train <- model_dat_clust %>% filter(PART_NUMBER %in% train_partsnums)
model_dat_clust_test  <- model_dat_clust %>% filter(!(PART_NUMBER %in% train_partsnums))
```

```
# Make cluster a 'factor'
model_object_clust        <- get_model_trained(
                                   training_data_df     = model_dat_clust_train,
                                   residuals_family_chr = "gaussian",
                                   target_chr           = "CorrectForecast",
                                   log_of_counts_lc     = TRUE,
                                   provide_simples_lc   = FALSE)

model_results_sumry_clust <- get_model_tested(
                                   model_object     = model_object_clust,
                                   testing_data_df  = model_dat_clust_test,
                                   target_chr       = "CorrectForecast",
                                   log_of_counts_lc = TRUE)
```

```
print(summary(model_object_clust))
```

```
##
## Call:
## glm(formula = Target ~ R_R_PRICE + EventQtyMths01to36_binned +
##     YearsInUse + Trend + SalesQtyMths01to12 + SalesQtyMths13to24 +
##     SalesQtyMths25to36 + EventQtyMths01to12 + EventQtyMths13to24 +
##     EventQtyMths25to36 + CLUSTER, family = residuals_family_chr,
```

```
##     data = training_data_df[-outliers, ])
##
## Deviance Residuals:
##     Min       1Q   Median       3Q      Max
## -4.3603  -0.4158   0.0259   0.4667   4.2999
##
## Coefficients:
##                            Estimate Std. Error t value Pr(>|t|)
## (Intercept)               6.412e-01  2.922e-02  21.946  < 2e-16 ***
## R_R_PRICE                -3.921e-04  3.398e-05 -11.541  < 2e-16 ***
## EventQtyMths01to36_binned  1.588e-03  9.956e-05  15.950  < 2e-16 ***
## YearsInUse               -6.614e-02  1.785e-03 -37.051  < 2e-16 ***
## TrendNone                -3.462e-01  1.515e-02 -22.853  < 2e-16 ***
## TrendUp                  -6.680e-01  1.922e-02 -34.763  < 2e-16 ***
## SalesQtyMths01to12        6.181e-01  7.365e-03  83.924  < 2e-16 ***
## SalesQtyMths13to24        2.428e-01  8.558e-03  28.371  < 2e-16 ***
## SalesQtyMths25to36        7.275e-02  8.382e-03   8.679  < 2e-16 ***
## EventQtyMths01to12        2.477e-01  1.104e-02  22.425  < 2e-16 ***
## EventQtyMths13to24       -1.648e-01  1.305e-02 -12.624  < 2e-16 ***
## EventQtyMths25to36       -1.244e-01  1.261e-02  -9.868  < 2e-16 ***
## CLUSTER1                  2.416e-01  3.129e-02   7.721 1.17e-14 ***
## CLUSTER2                  3.102e-01  3.237e-02   9.583  < 2e-16 ***
## CLUSTER3                  2.891e-01  3.220e-02   8.978  < 2e-16 ***
## CLUSTER4                  3.936e-01  3.166e-02  12.431  < 2e-16 ***
## CLUSTER5                  3.146e-01  3.234e-02   9.729  < 2e-16 ***
## CLUSTER6                  3.150e-01  3.163e-02   9.958  < 2e-16 ***
## CLUSTER7                  3.411e-01  3.193e-02  10.682  < 2e-16 ***
## CLUSTER8                  3.969e-01  3.163e-02  12.548  < 2e-16 ***
## CLUSTER9                  3.250e-01  3.247e-02  10.009  < 2e-16 ***
## CLUSTER10                 2.754e-01  3.218e-02   8.558  < 2e-16 ***
## CLUSTER11                 3.033e-01  3.223e-02   9.413  < 2e-16 ***
## CLUSTER12                 2.639e-01  3.234e-02   8.162 3.37e-16 ***
## CLUSTER13                 2.758e-01  3.165e-02   8.714  < 2e-16 ***
## CLUSTER14                 2.646e-01  3.176e-02   8.332  < 2e-16 ***
## CLUSTER15                 2.760e-01  3.156e-02   8.746  < 2e-16 ***
## CLUSTER16                 2.125e-01  3.178e-02   6.687 2.30e-11 ***
## CLUSTER17                 2.155e-01  3.151e-02   6.839 8.07e-12 ***
## CLUSTER18                 3.354e-01  3.119e-02  10.751  < 2e-16 ***
## CLUSTER19                 2.927e-01  3.144e-02   9.309  < 2e-16 ***
## CLUSTER20                 2.634e-01  3.107e-02   8.476  < 2e-16 ***
## CLUSTER21                 2.419e-01  3.165e-02   7.644 2.13e-14 ***
## CLUSTER22                 3.257e-01  3.089e-02  10.545  < 2e-16 ***
## CLUSTER23                 2.593e-01  3.104e-02   8.352  < 2e-16 ***
## CLUSTER24                 2.948e-01  3.100e-02   9.508  < 2e-16 ***
## CLUSTER25                 3.073e-01  3.183e-02   9.654  < 2e-16 ***
## CLUSTER26                 2.629e-01  3.182e-02   8.262  < 2e-16 ***
## CLUSTER27                 3.277e-01  3.102e-02  10.564  < 2e-16 ***
## CLUSTER28                 2.738e-01  3.064e-02   8.935  < 2e-16 ***
## CLUSTER29                 2.397e-01  3.041e-02   7.881 3.31e-15 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for gaussian family taken to be 0.7596306)
##
```

```
##     Null deviance: 193575  on 61506  degrees of freedom
## Residual deviance:  46691  on 61466  degrees of freedom
## AIC: 157683
##
## Number of Fisher Scoring iterations: 2
```

```r
kable(model_results_sumry_clust)
```

Simple_Error	ARIMA_Error	
0.3674597	0.4731738	
Some of the clu	sters appear t	o be highly significant, although we don't know whether this is spurious. However the o

Tuning the Simple Weighted Average

Perhaps we're being too clever, the weighted average forecast is very successful and has not been tuned at all. Can we tune those 3,2,1 parameters and make it better?

6 Parameters for tuning are:

for forecasting sales qty: - i: Sales Qty, mths 25to36 - j: Sales Qty, mths 13to24 - k: Sales Qty, mths 01to12

i,j,k can usefully be 0, 1, 2, 3 or 4

For setting forecast to zero where not sensible: - x: Sales Qty, mths 01to12 - y: Event Qty, mths 01to12 - z: Event Qty, mths 25to36

x,y,z can usefully be 0, 1, 3, 6

Hence, 8000 permutations to be tested!

We could also subdidivd our data by some feature, say - Price, modelling high value items differently to low value or - Frequently sold items to be modelled differently to frequently sold

so those 8000 permutations need to be tested over different model configurations.

For this we need a number of functions:

```r
get_error_given_params <-function(i,j,k, x,y,z, model_dat_df, getdetail_lc = FALSE){
  #print("started get_error_given_params")
  # calculate simple forecast
      error_detail <- model_dat_df %>%
                      mutate(SimpleForecast_Adj  =
                               case_when(YearsInUse %in% c(0,1) ~
                                                     SalesQtyMths01to12,
                                         YearsInUse == 2 ~
                                           round((k*SalesQtyMths01to12+
                                                  j*SalesQtyMths13to24)/(k+j)),
                                         YearsInUse >= 3 ~
                                           round((k*SalesQtyMths01to12+
                                                  j*SalesQtyMths13to24+
                                                  i*SalesQtyMths25to36)/(i+j+k)),
                                         TRUE ~ 0))

      # apply common sense rules
      setForecastToZero <- which(

                         (  error_detail$SalesQtyMths01to12 == 0)
```

```r
                              |
                              (   error_detail$SalesQtyMths01to12 <   x
                               & error_detail$SalesQtyMths13to24 == 0)
                                |
                              (   error_detail$EventQtyMths01to12 <   y
                               & error_detail$EventQtyMths13to24 == 0)
                              |
                              (   error_detail$SalesQtyMths01to12 <= z
                               & error_detail$SalesQtyMths13to24 <= z
                               & error_detail$SalesQtyMths25to36 == 0)
                              |
                              (   error_detail$EventQtyMths01to12 <= 1
                               & error_detail$EventQtyMths13to24 <= 1
                               & error_detail$EventQtyMths25to36 == 0)
      )

      setForecastToZero <- unique(setForecastToZero)

      if(length(setForecastToZero)>0){
        error_detail$SimpleForecast_Adj[setForecastToZero] <- 0
      }

      # summarise
      error_sumry  <- error_detail %>%
                mutate(SimpleError     = R_R_PRICE * abs(SimpleForecast     - CorrectForecast),
                       SimpleError_Adj = R_R_PRICE * abs(SimpleForecast_Adj - CorrectForecast)) %>%
                  summarise(SimpleError     = sum(SimpleError    )/sum(ValueToOrder),
                            SimpleError_Adj = sum(SimpleError_Adj)/sum(ValueToOrder))
      if(getdetail_lc){
        return(list(error_sumry, error_detail))
      }else{
        return(error_sumry)
      }
}
```

```r
get_data_filteredtorange <- function(model_dat_df, event_range_vc=NULL, price_range_vc=NULL){

  # handle NULLs
  if(is.null(event_range_vc)){
    event_range_vc <- c(0,100000)
  }
  if(is.null(price_range_vc)){
    price_range_vc <- c(0,100000)
  }
  # filter to EventRange
  model_dat_df <- model_dat_df %>%
                  mutate(EventQtyMths0To36 = EventQtyMths01to12 +
                                             EventQtyMths13to24 +
                                             EventQtyMths25to36 ) %>%
                  filter(EventQtyMths0To36 >= event_range_vc[1]
                         &
                         EventQtyMths0To36 <= event_range_vc[2])

  # filter to PriceRange
```

```
  model_dat_df <- model_dat_df %>%
                  filter(R_R_PRICE >= price_range_vc[1]
                         &
                         R_R_PRICE <= price_range_vc[2])

  return(model_dat_df)
}
```

```
get_optimal_ijk <- function(model_dat_df,
                            permutations_df,
                            event_range_vc = NULL,
                            price_range_vc = NULL,
                            getdetail_lc   = FALSE){

  if(is.null(event_range_vc)){
    event_range_vc <- c(0,100000)
  }
  if(is.null(price_range_vc)){
    price_range_vc <- c(0,100000)
  }

  error_detail_all <- NULL
  model_dat_df     <- get_data_filteredtorange(model_dat_df   = model_dat_df,
                                               event_range_vc = event_range_vc,
                                               price_range_vc = price_range_vc)

  pb <- txtProgressBar(min = 1, max = 4, style = 3)
  for(i in c(0, 1, 2, 3, 4)){
    setTxtProgressBar(pb, i)
    for(j in c(0, 1, 2, 3, 4)){
      for(k in c(0, 1, 2, 3, 4)){
        for(x in c(0, 1, 3, 6)){
          for(y in c(0, 1, 3, 6)){
            for(z in c(0, 1, 3, 6)){ #8,000 permutations to test!

            # run model
            error <- get_error_given_params(i,j,k,x,y,z, model_dat_df, getdetail_lc)

            if(getdetail_lc){
              error_sumry  <- error[[1]]
              error_detail <- error[[2]]
            }else{
              error_sumry  <- error
            }

            rm(error)

            #format results
            permutations_df_add <- data_frame(EventQtyFrom = event_range_vc[1],
                                              EventQtyTo   = event_range_vc[2],
                                              PriceFrom    = price_range_vc[1],
                                              PriceTo      = price_range_vc[2],
                                              Sales25to36x = i,
                                              Sales13to24x = j,
```

```
                                                  Sales01to12x = k,
                                                  x = x,
                                                  y = y,
                                                  z = z,
                                                  SimpleError_New = error_sumry$SimpleError_Adj,
                                                  SimpleError_Old = error_sumry$SimpleError)

            # save results
            permutations_df  <- bind_rows(permutations_df,
                                          permutations_df_add)

            #get detail if required
            if(getdetail_lc){
              error_detail_all <- bind_rows(error_detail_all,
                                            error_detail)
            }
            }
          }
        }
      }
    }
  }

  #format results
  permutations_df <- as.data.frame(permutations_df)
  colnames(permutations_df) <- c("EventQtyFrom", "EventQtyTo",
                                 "PriceFrom",    "PriceTo",
                                 "Sales25to36x", "Sales13to24x", "Sales01to12x",
                                 "x","y","z",
                                 "SimpleError_New","SimpleError_Old")
  return(list(permutations_df, error_detail_all))
}
```

Optimal parameters for data as a whole

Now we can use these funcitons to test the model configurations. first is to test 'no configuration', ie no subdivision by price or qty. Parameters will be optimised for the data set as a whole. Explore parameters based on training data only, later we will compare models on the test data.

```
# get training
evaluation_train_dat <- simple_model_dat %>% # test set
                         filter(PART_NUMBER %in% train_partsnums)

#prep table to hold the permuations results:
permutations <-data_frame(
                  EventQtyFrom = numeric(), EventQtyTo   = numeric(),
                  PriceFrom    = numeric(), PriceTo      = numeric(),
                  Sales25to36x = numeric(), Sales13to24x = numeric(), Sales01to12x = numeric(),
                  x            = numeric(), y            = numeric(), z            = numeric(),
                  SimpleError_New = numeric(), SimpleError_Old = numeric())

# find optimal values using above functions
permutations_list <- get_optimal_ijk(model_dat_df    = evaluation_train_dat,
```

```
                                        permutations_df = permutations,
                                        getdetail_lc    = FALSE)

permutations <- permutations_list[[1]]
```

```
#show top results:
kable(  (permutations %>%
          select(Sales25to36x, Sales13to24x, Sales01to12x, x,y,z, SimpleError_New, SimpleError_Old) %>%
            arrange(SimpleError_New))[1:10,]
     )
```

Sales25to36x	Sales13to24x	Sales01to12x	x	y	z	SimpleError_New	SimpleError_Old
0	1	4	0	0	0	0.4047716	0.4255312
0	1	4	0	0	1	0.4047716	0.4255312
0	1	4	0	1	0	0.4047716	0.4255312
0	1	4	0	1	1	0.4047716	0.4255312
0	1	4	1	0	0	0.4047716	0.4255312
0	1	4	1	0	1	0.4047716	0.4255312
0	1	4	1	1	0	0.4047716	0.4255312
0	1	4	1	1	1	0.4047716	0.4255312
0	1	3	0	0	0	0.4059852	0.4255312
0	1	3	0	0	1	0.4059852	0.4255312

Basically, this says the best strategy is to order (4 x Sales01to12 + 1 x Sales13to24)/5. We then get a small improvement over the original 3,2,1 strategy. But, surely the more data points we have, the better forecast we can make?

Sub-divide the data by Event Qty

Let's run the optimisation function again, but this time we derive i,j,k optimally for each range of EventQty (ie data points). So we expect a different i,j,k where we have lots of data to those examples where we have few data points to analyse. Train on training data only, later we will compare models on the test data.

```
permutations_eventqty <- permutations[0,]

#begin loops
for(i in list(c(0,50), c(51,100), c(101,250), c(251,500), c(501,10000))){

 permutations_eventqty_list <- get_optimal_ijk(model_dat_df    = evaluation_train_dat,
                                               permutations_df = permutations_eventqty,
                                               event_range_vc  = i,
                                               getdetail_lc    = FALSE)

  permutations_eventqty <- permutations_eventqty_list[[1]]
}

#present top result for each range
params_by_eventqty <- permutations_eventqty %>%
                        group_by(EventQtyFrom, EventQtyTo) %>%
                          mutate(Sequence = row_number(SimpleError_New)) %>%
                            filter(Sequence == 1) %>%
                                arrange(EventQtyFrom)
```

```r
kable(params_by_eventqty %>% select(-PriceFrom, -PriceTo))
```

EventQtyFrom	EventQtyTo	Sales25to36x	Sales13to24x	Sales01to12x	x	y	z	SimpleError_New	Simp
0	50	0	1	2	0	0	0	0.5105258	
51	100	2	0	3	0	0	0	0.2609826	
101	250	1	2	4	0	0	0	0.1770179	
251	500	0	1	3	0	0	0	0.1408958	
501	10000	1	3	1	0	0	0	0.0841114	
We're getting	small improve	ments over the	traditional wei	ghted averages.					

Sub-divide the data by Price

Let's also see the results when the data is subdivided by price. Train on training data only, later we will compare models on the test data.

```r
#prep data, blank copy of previous table
permutations_price <- permutations[0,]

for(i in list(c(0,10), c(11,50), c(51,200), c(201,500), c(501,10000))){
  permutations_price_list <- get_optimal_ijk(model_dat_df    = evaluation_train_dat,
                                             permutations_df = permutations_price,
                                             price_range_vc  = i,
                                             getdetail_lc    = FALSE)

  permutations_price <- permutations_price_list[[1]]
}

#present top result for each range
params_by_price <- permutations_price %>%
                    group_by(PriceFrom, PriceTo) %>%
                      mutate(Sequence = row_number(SimpleError_New)) %>%
                        filter(Sequence == 1) %>%
                          arrange(PriceFrom)
```

```r
kable(params_by_price %>% select(-EventQtyFrom, -EventQtyTo))
```

PriceFrom	PriceTo	Sales25to36x	Sales13to24x	Sales01to12x	x	y	z	SimpleError_New	SimpleE
0	10	1	1	3	0	0	0	0.3604959	
11	50	1	1	4	0	0	0	0.3031537	
51	200	0	1	2	0	0	0	0.4087287	
201	500	0	1	3	0	0	0	0.5074186	
501	10000	3	0	4	0	0	0	0.2767026	
Again, we se	e small im	provements over	the weighted a	verage approach	.				

Evaluate on the Test Data

And which performs best overall on the test set for the ValueToOrder, We need a function to make that comparison

```
evaluate_scheme <- function(paramtype_chr, params_df, model_dat_df){

  errors_all <- NULL

  #cycle around ranges
  for(loop in 1:nrow(params_df)){
    i <- params_df$Sales25to36x[loop]
    j <- params_df$Sales13to24x[loop]
    k <- params_df$Sales01to12x[loop]
    x <- params_df$x[loop]
    y <- params_df$y[loop]
    z <- params_df$z[loop]

    if(paramtype_chr == "EventQty"){
      event_range_vc <- c(params_df$EventQtyFrom[loop], params_df$EventQtyTo[loop])
      price_range_vc <- c(0, 100000)
    }else{
      event_range_vc <- c(0, 100000)
      price_range_vc <- c(params_df$PriceFrom[loop],    params_df$PriceTo[loop])
    }

    dat       <- get_data_filteredtorange(model_dat_df   = model_dat_df,
                                          event_range_vc = event_range_vc,
                                          price_range_vc = price_range_vc)

    error_list <- get_error_given_params(i,j,k, x,y,z, dat, getdetail = TRUE)
    errors_all <- bind_rows(errors_all, error_list[[2]])

  }
  return(errors_all)
}
```

We'll evaluate these two approaches on the test data used for the LM model.

```
# Evaluate schemes
# by price
errors_all_price    <- evaluate_scheme(paramtype_chr = "Price",
                                       params_df     = params_by_price,
                                       model_dat_df  = evaluation_test_dat)
# by eventqty
errors_all_eventqty <- evaluate_scheme(paramtype_chr = "EventQty",
                                       params_df     = params_by_eventqty,
                                       model_dat_df  = evaluation_test_dat)
# summarise results
error_sumry_price   <- errors_all_price %>%
                 mutate(SimpleError     = R_R_PRICE * abs(SimpleForecast     - CorrectForecast),
                        SimpleError_Adj = R_R_PRICE * abs(SimpleForecast_Adj - CorrectForecast))%>%
                   summarise(SimpleError     = sum(SimpleError    )/sum(ValueToOrder),
                             SimpleError_Adj = sum(SimpleError_Adj)/sum(ValueToOrder))

error_sumry_eventqty<- errors_all_eventqty %>%
                  mutate(SimpleError     = R_R_PRICE * abs(SimpleForecast     - CorrectForecast),
                         SimpleError_Adj = R_R_PRICE * abs(SimpleForecast_Adj - CorrectForecast))%>%
                    summarise(SimpleError     = sum(SimpleError    )/sum(ValueToOrder),
```

```r
                              SimpleError_Adj = sum(SimpleError_Adj)/sum(ValueToOrder))
# View results
evaluation <- bind_rows(c(Type="Price",error_sumry_price),
                        c(Type="EventQty",error_sumry_eventqty))
```

```r
kable(evaluation)
```

Type	SimpleError	SimpleError_Adj
Price	0.3660166	0.3532567
EventQty	0.3674597	0.3557654

SimpleError_Adj is the error resulting from the new models. SimpleError is the error from the traditional 3,2,1 method

Overall, these models offer approximately the same outcome as the simple weighted average. Not worth applying the additional complexity for such small gains. We'll look at a couple more models before settling with the traditional 3,2,1 method, which we will then pit against deep learning.

Bagged CART

Let's try a different type of model, the Bagged CART. Useful for regression, being a tree model it has very different architecture to linear regression.

```r
model_tb1 <- train(log(CorrectForecast+1) ~.,
                   method    = "treebag",
                   data      = model_dat_train %>%
                               # get log of data, we know that helps!
                               mutate(SalesQtyMths01to12 = log(SalesQtyMths01to12+1),
                                      SalesQtyMths13to24 = log(SalesQtyMths13to24+1),
                                      SalesQtyMths25to36 = log(SalesQtyMths25to36+1),
                                      EventQtyMths01to12 = log(EventQtyMths01to12+1),
                                      EventQtyMths13to24 = log(EventQtyMths13to24+1),
                                      EventQtyMths25to36 = log(EventQtyMths25to36+1))%>%
                               select(-PART_NUMBER, -ForecastFromMth, -SimpleForecast, -SimpleError,
                                      -ARIMAforecast,   -ARIMAError,  -ARIMAsigma2, -ValueToOrder),
                   metric    = "RMSE",
                   #trControl = train_control_lm,
                   trace     = FALSE)

#Test
model_results_sumry_tree <- get_model_tested(
                               model_object     = model_tb1,
                               testing_data_df  = model_dat_test,
                               target_chr       = "CorrectForecast",
                               log_of_counts_lc = TRUE)
```

```r
kable(model_results_sumry_tree)
```

Simple_Error	ARIMA_Error	Model_Error
0.3674597	0.4731738	0.4323573

Not bad, but the simple weighted average is proving extra-ordinarily difficult to beat!

Let's try another architecture, XG Boost, which is a random forest tool very popular in Kaggle. https://machinelearningmastery.com/gentle-introduction-xgboost-applied-machine-learning/

```
# Start the clock!
ptm <- proc.time()

model_xg <- train(log(CorrectForecast+1) ~.,
                  method    = "xgbTree",
                  data      = model_dat_train %>%
                              # get log of data, we know that helps!
                              mutate(SalesQtyMths01to12 = log(SalesQtyMths01to12+1),
                                     SalesQtyMths13to24 = log(SalesQtyMths13to24+1),
                                     SalesQtyMths25to36 = log(SalesQtyMths25to36+1),
                                     EventQtyMths01to12 = log(EventQtyMths01to12+1),
                                     EventQtyMths13to24 = log(EventQtyMths13to24+1),
                                     EventQtyMths25to36 = log(EventQtyMths25to36+1))%>%
                              select(-PART_NUMBER, -ForecastFromMth, -SimpleForecast, -SimpleError,
                                     -ARIMAforecast,   -ARIMAError,  -ARIMAsigma2, -ValueToOrder),
                  metric    = "RMSE",
                  #trControl = train_control_lm,
                  trace     = FALSE)

# Stop the clock
time_diff <- proc.time() - ptm
kable(time_diff)

#Test
model_results_sumry_xg <- get_model_tested(
                              model_object     = model_xg,
                              testing_data_df  = model_dat_test,
                              target_chr       = "CorrectForecast",
                              log_of_counts_lc = TRUE)
```

```
library(knitr)
kable(model_results_sumry_xg)
```

Simple_Error	ARIMA_Error	Model_Error
0.3674597	0.4731738	0.4391442

Still no better than simple weighted average! Nevertheless, XGboost appears to be the best of the machine learning models, and has potential to be better because it has yet to be tuned.

The mlr package is useful for automating tuning of XGboost models: http://mlr-org.github.io/How-to-win-a-drone-in-20-lines-of-R-code/

Caret can also tune XgbTree using a tuning grid: https://analyticsdataexploration.com/xgboost-model-tuning-in-crossvalidation

```
library(mlr)
xg_dat_train <- model_lm1_dat_train %>% select(-PART_NUMBER, -ForecastFromMth, -CorrectForecast)
xg_dat_train$Trend <- as.factor(xg_dat_train$Trend)

xg_dat_test  <- model_lm1_dat_test  %>% select(-PART_NUMBER, -ForecastFromMth, -CorrectForecast)
xg_dat_test$Trend  <- as.factor(xg_dat_test$Trend)
```

```
trainTask <- makeRegrTask(data     = xg_dat_train,
                          target   = "ValueToOrder")

testTask  <- makeRegrTask(data     = xg_dat_test,
                          target   = "ValueToOrder")
```

Deep Learning

The 'simple' models have failed to improve upon the weighted average, suggesting there is very little information in the data. Nevertheless, a deep learning model will be attempted in another notebook.