Time Series Forecasting

Forecast with Distributed Lags

Marc Kullmann

2019-06-20

Contents

1 Introduction

This reposition is a refurbished project from my master-lecture Computational Methods in Economics, held by Prof. Zhentao SHI.

The purpose of this project is to illustrate how well machine learning methods predicts against mixed data sampling. To see how well each algorithm performs, we use in a numerical example a Data Generating Process (DGP) to generate random normal distributed numbers and let our algorithm run. After comparing each in section 3.1.8, we use a real data example, the data-set "rvsp500", which contains the realized volatility of the S&P500. The method is analogue to the numerical one.

This report is structured as the folllowing: First we define our functions in the order nlopR, decicion tree's, gradient boosting and LASSO. In section 2 we start our numerical evaluation with the Monte Carlo simulation and then the Real Date example. Each section and subsection shares the same structure, first we define the parameters, second we start the evaluation with the order as the definions (nlopR, decision tree's, gradient boosting and LASSO). The last section (Real Data example) has an attional subsection with the plotting of our results.

2 Define Functions

The approach of all functions work basically the same except nloptR. For desicion tree's, boosting method's and LASSO, we write a function with variables of the training data set (train.dgp), test data set (test.dgp), declare the true dependent variable (test.y), and finally specify the train control variables (trncntrl). Inside the functions we have a repetitive function body:

1. record the time to get an idea how time consuming (and therefore computer-power) each method is.
2. train the testing data set with our train variables and simple specifications such as `verbose = FALSE`
3. Record the time consuming work into `@@@_time`
4. Save the results `(@@@_res)`. Coninuing with concatenation and printing the best tune, observation.
5. Predict our trained variables, safe them.
6. Calculating the out-of-sample RMSE
7. `return()` a list of all important variables.

2.0.1 nloptR

For nloptR, NLopt is a free/open-source library for nonlinear optimization, NLopt includes implementations of a number of different optimization algorithms. The one we used below is from the Local gradient-based optimization family, SpeciBied in NLopt as NLOPT_LD_SLSQP, this is a sequential quadratic programming (SQP) algorithm for nonlinearly constrained gradient-based optimization (supporting both inequality and equality constraints).

Y_predict following the form when you substitute the exponential Almon speciBication beta into the general equation of Y.Then we deBine the data generating process function a head of time here for later use.The objective function is valued as the difference between y and y_predict to the power of two then dividied by the total number of the observations.The gradient of objective function is separately derivative to all the 20 betas, to rho,to alpha1 and alpha2.And the almon function

also the gradient of almon function are also set here to boost the calculation time latterly.Then we combine all the things above into a new function as a list form for easier call.After setting thoses parameters of the true value x0, Starting points x1,also optimization options ,we can Binally call nloptr from the packages, the results also returns the calculation time and also Optimal value of objective function and the controls.

```
# \hat{y}
y_pred = function(theta) {
  y = theta[4] + theta[1] * X %*% almon(theta[2], theta[3], 1:N)
  return(y)
}

# Data Generating Process
DGP = function(TT, N, true_weight) {
  X = matrix(rnorm(TT * N), TT)
  y = betaN + rho * X %*% true_weight + rnorm(TT)
  return(list(X = X, y = y))
}

# almon function
almon = function(alpha1, alpha2,  j) {
  exp(alpha1 * j + alpha2 * j ^ 2)
}
## gradient of almon function
# derivative to alpha1
almon.dalpha1 = function(alpha1, alpha2,  j) {
  j * exp(alpha1 * j + alpha2 * j ^ 2)
}
# derivative to alpha2
almon.dalpha2 = function(alpha1, alpha2,  j) {
  j ^ 2 * exp(alpha1 * j + alpha2 * j ^ 2)
}

#### Performance optimization
eval_f_list <- function(x) {
  common_term =  y_pred(x) - y
  return(list(
    "objective" = (1 / TT) * (sum(common_term)) ^ 2,
    "gradient" = c(
      2 / TT * sum (common_term * (X %*% almon(alpha1 = x[2],
                                               alpha2 = x[3],
                                               0:(N - 1)))
                    ),
      2 / TT * sum (common_term * (X %*% almon.dalpha1(alpha1 = x[2],
                                                       alpha2 = x[3],
                                                       0:(N - 1)))
                    ),
      2 / TT * sum (common_term * (X %*% almon.dalpha2(alpha1 = x[2],
```

```
                                                    alpha2 = x[3],
                                                    0:(N - 1)))
                    ),
      2 / TT * sum (common_term)
    )
  ))
}

# Constraints
## no constraints

################ Alogrithm 1 (nloptr)############################

midas_nloptr <- function(X, y, x0, x1) {
  ### nlopt optimization option
  opts = list(
    "algorithm" = "NLOPT_LD_SLSQP",
    "xtol_rel" = 1.0e-12,
    "ftol_rel" = 1.0e-8,
    "maxeval" = 10000
  )

  t <- Sys.time()
  nloptr_res <- nloptr(
    x0 = x1,
    eval_f = eval_f_list,
    opts = opts
  )
  cat(
    "Time Cost of the nloptr optimization:",
      Sys.time() - t , "\n",
    "Starting points:", x1 , "\n",
    "True values:", x0 ,"\n",
    "Optimal value of objective function :",
      nloptr_res[["objective"]], "\n",
    "Optimal value of controls are",
      nloptr_res[["solution"]], "\n"
  )
    return(nloptr_res)
}
```

2.0.2 Decision Tree's

In this section we compare two decisiontree methods, the singleTree method and the randomForest method. Simply to understand by how much randomForest differs to the singleTree method in a IID DGP and real data analysis. For this purpose, the following function combines the singleTree method, by trainig the data and then predicting the declared testing data and finally calculating

the out-of-sample RMSE.

2.0.2.1 Single Tree

```
################ Alogrithm 2.1 (midas_rpart) ############################
midas_rpart <- function(train.dgp, test.dgp, test.y, trncntrl){
  t <- Sys.time()

  rpart_tree <<- train(
    y ~ .,
    data = train.dgp,
    method = "rpart",
    trControl = trncntrl,
    preProc = c("center", "scale")
  )

  rpart_time = Sys.time()-t
  rpart_tree_res <<- rpart_tree$results
  minRMSE <-min(rpart_tree_res$RMSE)
  whichObser <- which.min(rpart_tree_res$RMSE)
  cat(
    "Time Cost of Finding Best Tuning Parameters:",
      rpart_time, "\n",
    "The Minimum Value of (in-sample) RMSE:",
      minRMSE, "of Observation:", whichObser,"\n"
      )

  yhat.rpart_tree <<- predict(
    rpart_tree,
    newdata = test.dgp
  )

  outsample_RMSE <- mean(sqrt((yhat.rpart_tree - test.y)^2))
  cat(
    "The out-sample RMSE:",  outsample_RMSE, "\n"
  )
  rpart_list <- list(
    "rpart_tree" = rpart_tree,
    "rpart_tree_res" = rpart_tree_res,
    "yhat.rpart_tree" = yhat.rpart_tree,
    "rpart_time" = rpart_time,
    "minRMSE"= minRMSE,
    "minObser" = whichObser,
    "outsample_RMSE" = outsample_RMSE
  )
  return(rpart_list)
}
```

2.0.2.2 Random Forest

RandomForest is a wildely used method to predict and forcast various data. Our approach is similar to the singleTree method, first we train, then predict and then compare.

```
midas_rF <- function(train.dgp, test.dgp, test.y, trncntrl) {
  t <- Sys.time()
  set.seed(5170)
  rf_tree <<- train(
    y ~ .,
    data = train.dgp,
    method = "rf",
    tuneLength = 30,
    trControl = trncntrl,
    ntrees = c(1:10 * 100),
    preProc = c("center", "scale")
  )
  rf_time = Sys.time() - t
  rf_tree_res <<- rf_tree$results
  minRMSE <- min(rf_tree_res$RMSE)
  whichObser <- which.min(rf_tree_res$RMSE)
  cat(
    "Time Cost of Finding Best Tuning Parameters:",
      rf_time, "\n",
    "The Minimum Value of (in-sample) RMSE:",
      minRMSE, "of Observation:", whichObser,"\n"
  )

  yhat.rf_tree <<- predict(
    rf_tree,
    newdata = test.dgp
    )

  outsample_RMSE <-  mean(sqrt((yhat.rf_tree - test.y) ^ 2))
  cat("The out-sample RMSE:",outsample_RMSE, "\n")
  rF_list <-
    list(
      "rf_tree" = rf_tree,
      "rf_tree_res" = rf_tree_res,
      "yhat.rf_tree" = yhat.rf_tree,
      "rf_time" = rf_time,
      "minRMSE"= minRMSE,
      "whichObser" = whichObser,
      "outsample_RMSE" = outsample_RMSE
      )

  return(rF_list)
}
```

2.0.3 Gradient Boosting Method

For generalized boosted models, The only difference between random forest and the boosted regression is that the former uses the same dependent variable and simple average of the trees whereas the latter uses the residual of the previous step and adding up the new tree. Here we try to compare the auto tuned version with the default settings with a tuned version of it. #### Auto Tuned Boosting

```
################ Algorithm 3 (midas_gbm) ############################

### Auto GBM
midas_gbm_auto <- function(train.dgp, test.dgp, test.y, trncntrl) {
  t <- Sys.time()
  set.seed(5170)

  gbm_tree_auto <<- train(
    y ~ .,
    data = train.dgp,
    method = "gbm",
    distribution = "gaussian",
    trControl = trncntrl,
    verbose = FALSE,
    preProc = c("center", "scale")
  )

  gbm_auto_res <<- gbm_tree_auto$results
  gbm_auto_time <- Sys.time() - t
  cat(
    "Time Cost of Finding Best Tuning Parameters:",
      gbm_auto_time, "\n",
    "The Minimum Value of (in-sample) RMSE:",
      min(gbm_auto_res$RMSE), "of Observation:",
        which.min(gbm_auto_res$RMSE), "\n"
  )

  yhat.midas_gbm_auto <<- predict(
    gbm_tree_auto,
    newdata = test.dgp
    )
  outsample_RMSE <- mean(sqrt((yhat.midas_gbm_auto - test.y) ^ 2))
  cat(
    "The out-sample RMSE:", outsample_RMSE, "\n"
    )
  gbm_auto_list <-
    list(
      "gbm_auto_res" = gbm_auto_res,
      "yhat.midas_gbm_auto" = yhat.midas_gbm_auto,
      "gbm_auto_time" = gbm_auto_time,
```

```
      "gbm_tree_auto" = gbm_tree_auto,
      "outsample_RMSE" = outsample_RMSE
      )
  return(gbm_auto_list)
}
```

2.0.3.1 Tuned Boosting

The tuned boosting differs in a way that we add the variable tuneGrid, to modify the following variables variables: n.trees, shrinkage, interaction.depth, and n.minobsinnode.

```
midas_gbm_tuned <- function(train.dgp, test.dgp, test.y, trncntrl, grd) {
  t <- Sys.time()
  set.seed(5170)
  gbm_tree_tune <<- train(
    y ~ .,
    data = train.dgp,
    method = "gbm",
    distribution = "gaussian",
    trControl = trncntrl,
    verbose = FALSE,
    tuneGrid = grd,
    preProc = c("center", "scale")
  )

  time_gbm_tune <- Sys.time() - t
  gbm_tune_res <<- gbm_tree_tune$results

  cat(
    "Time Cost of Finding Best Tuning Parameters:",
      Sys.time() - t, "\n",
    "The Minimum Value of (in-sample) RMSE:",
      min(gbm_tune_res$RMSE), "of Observation:",
      which.min(gbm_tune_res$RMSE), "\n"
  )

  yhat.midas_gbm_tune <<- predict(
    gbm_tree_tune,
    newdata = test.dgp
    )
  outsample_RMSE <- mean(sqrt((yhat.midas_gbm_tune - test.y)^2))
  cat("The out-sample RMSE:", outsample_RMSE, "\n")

  gbm_tune_list <-
    list(
      "gbm_tree_tune" = gbm_tree_tune,
      "gbm_tune_res" = gbm_tune_res,
      "yhat.midas_gbm_tune" = yhat.midas_gbm_tune,
```

```
      "time_gbm_tune" = time_gbm_tune,
      "outsample_RMSE" = outsample_RMSE
    )
  return(gbm_tune_list)
}
```

2.0.4 LASSO

Following the procedure as described, here we implement the glmnet LASSO into caret. As the tuned GBM, we have to add a traning control variable and grid, to specify the way LASSO optimizes

```
midas_LASSO_caret <- function(train.dgp, test.dgp, test.y, trncntrl, grd){
  t <- Sys.time()
  set.seed(5170)
  lasso_caret_train <<- train(y ~ ., data = train.dgp,
                              method = "glmnet",
                              tuneGrid = grd,
                              tuneLength = 100, metric = "RMSE",
                              trControl = trncntrl,
                              preProc = c("center", "scale")
  )
  time_lasso_caret <- Sys.time() - t
  lasso_caret_res <<- lasso_caret_train$results

  cat(
    "Time Cost of Finding Best Tuning Parameters:",
      Sys.time() - t, "\n",
    "The Minimum Value of (in-sample) RMSE:",
      min(lasso_caret_res$RMSE), "of Observation:",
      which.min(lasso_caret_res$RMSE), "\n"
  )
  plot(lasso_caret_train)

  yhat.lasso_caret <<- predict(
    lasso_caret_train,
    newdata = test.dgp
  )

  outsample_RMSE <- mean(sqrt((yhat.lasso_caret - test.y)^2))
  cat("The out-sample RMSE:", outsample_RMSE, "\n")

  lasso_caret_list <-
    list(
      "lasso_train" = lasso_caret_train,
      "time_lasso_caret" = time_lasso_caret,
      "outsample_RMSE" = outsample_RMSE,
      "yhat.lasso_caret" = yhat.lasso_caret
    )
```

```
  return(lasso_caret_list)
}
```

3 Numerical Evaluation

3.1 Monte Carlo simulation

3.1.1 Define parameters

```
alpha1  <- 1.1
alpha2  <- -0.5
j       <- 0:19
betaN   <- 0
almon0  <- almon(alpha1 = alpha1, alpha2 = alpha2, j = j)
rho     <- 1 / (sum (almon0))

TT      <-  200 # total length of the time series
N       <-  20  # number of regressors
```

3.1.2 Data Generating Precesses

3.1.2.1 DGP 1: correct specification

```
almon_weight  <- almon(alpha1 = alpha1, alpha2 = alpha2, j = j)

TT = 200 # total length of the time series
N = 20 # number of regressors

set.seed(5170)
data_cor_specification_1.1 = DGP(TT, N, almon_weight)

X_1.1 = data_cor_specification_1.1$X
y_1.1 = data_cor_specification_1.1$y
yX_1.1 <- data.frame(y = y_1.1, X = X_1.1)

set.seed(51700)
data_cor_specification_1.2 = DGP(TT, N, almon_weight)

X_1.2 = data_cor_specification_1.2$X
y_1.2 = data_cor_specification_1.2$y
yX_1.2 <- data.frame(y = y_1.2, X = X_1.2)
```

3.1.2.2 DGP 2: almon lag misspecification

```
set.seed(5170)
non_almon_weight = c(exp(seq(0.1, 1.5, length.out = 4)),
                     exp(seq(1.5,-2.3, length.out = 8))[-1],
                     seq(0.1, 0.0, length.out = 3),
```

```
                    rep(0, N - 4 - 7 - 3))

data_misspecification_2.1 = DGP(TT, N, non_almon_weight)

X_2.1 = data_misspecification_2.1$X
y_2.1 = data_misspecification_2.1$y
y_raw_2.1 = y_2.1
yX_2.1 <- data.frame(y = y_2.1, X = X_2.1)
```

3.1.3 nloptR optimization option

```
opts = list(
  "algorithm" = "NLOPT_LD_SLSQP",
  "xtol_rel" = 1.0e-9,
  "ftol_rel" = 1.0e-6,
  "maxeval" = 500
)

### initial value X0
x0 = c(rho, alpha1, alpha2, betaN)

### random value X1
x1 = c((rho + 0.101), (alpha1 + 1.1), (alpha2 - 1.01), betaN)

### Solution for 2.1 nloptr

#### Special treatment for nloptR -> copy X_@@ to X
## Optimizing Data 1:
X <- X_1.1
y <- y_1.1

nlopres_true <- midas_nloptr(X, y, x0 = x0, x1 = x1)
nlopres_true_sol <- nlopres_true[["solution"]]

## Optimizing Data 2:
y <- y_1.2
y.hat_nloptr_true_sol <- y_pred(nlopres_true_sol)

cat(
  "Out-sample RMSE:", mean(sqrt((y.hat_nloptr_true_sol - y_1.2) ^ 2)), "\n"
  )
nloptr_correct_outsample_rmse <- mean(sqrt((y.hat_nloptr_true_sol - y_1.2) ^ 2))

## False
X <- X_2.1
y <- y_2.1
```

```
nlopres_true <- midas_nloptr(X, y, x0 = x0, x1 = x1)
nlopres_false_sol <- nlopres_true[["solution"]]

y.hat_nloptr_false_sol <- y_pred(nlopres_false_sol)

cat(
  "Out-sample RMSE:", mean(sqrt((y.hat_nloptr_false_sol - y_1.2) ^ 2)), "\n"
  )
nloptr_false_outsample_rmse <- mean(sqrt((y.hat_nloptr_false_sol - y_1.2) ^ 2))
```

3.1.4 Train Control

```
myTrainControl <-
  trainControl(method = "repeatedcv",
               number = 10,
               repeats = 10)
```

Specifying our train control variable to the repeated crossvalidation for the four machine learning algorithms we choose to follow the settings from the given shell: ten fold CV, ten times.

3.1.5 Decision Tree's

3.1.5.1 Single Tree

```
mcs_rpart <- midas_rpart(
  yX_1.1,
  yX_1.2,
  y_1.2,
  trncntrl = myTrainControl
  )
```

3.1.5.2 Random Forest

```
mcs_rF <- midas_rF(
  yX_1.1,
  yX_1.2,
  y_1.2,
  trncntrl = myTrainControl
  )

print(mcs_rF$rf_tree$finalModel)

plot(mcs_rF$rf_tree$finalModel)
```

Suprisingly, our best RMSE (in-sample) is better than the suggested one from `$final model`. However, the out-sample RMSE is also surprisingly low, compared to the in-sample RMSE. We added ten different values of trees (100 - 1000) to have at least a small bandwidth of comparison. The final model suggests, 500 trees and 15 variables give the best result.

3.1.6 Gradient Boosting Method

3.1.6.1 Auto Tuned Boosting

```
mcs_gbm_auto <- midas_gbm_auto(
  train.dgp = yX_1.1,
  test.dgp = yX_1.2,
  test.y = y_1.2,
  trncntrl = myTrainControl
  )

print(mcs_gbm_auto$gbm_tree_auto$finalModel)
plot(mcs_gbm_auto$gbm_tree_auto)
```

The auto tuned GBM function returns a positive correlation of boosting iterations and RMSE. However, the in-sample RMSE as well as out-sample RMSE seem both relatively low. To compensate and make use of the tuning values the next subsection provides us with the tuned GBM.

3.1.6.2 Tuned GBM

```
grid_gbm <- expand.grid(
  n.trees = (9:12 * 1000),
  shrinkage = c(0.0001, 0.001, 0.01),
  interaction.depth = c(1, 16, 20),
  n.minobsinnode = c(1)
)

mcs_gbm_tune <- midas_gbm_tuned(
  yX_1.1,
  yX_1.2,
  y_1.2,
  trncntrl = myTrainControl,
  grd = grid_gbm
  )

mcs_gbm_tune$gbm_tune_res %>%
  group_by(shrinkage) %>%
  ggplot(aes(x = n.trees,
             y = RMSE, group = shrinkage,
             color = shrinkage)) +
    scale_colour_gradient(low = "blue", high = "red") +
    geom_line() +
    facet_grid(rows = vars(interaction.depth))
```

The last plot gives us a hint how the proposed grid interacts with our IID sample. We can see the higher interaction.depth returns a much lower RMSE, interestingly the more terminal nodes we have the better the estimation. Which seems on the first hand not that intuitive, as we are working in a IID-sample case. Differently one would think it could have a correlation with specific lags in a time series analysis with q lags. Furthermore, the lowest shrinkage (0.0001) and the highest shrinkage

(0.01) seem to have a positive correlation with the number of trees.

3.1.7 LASSO

3.1.7.1 Caret

```
grid_L <- expand.grid(
  alpha=1,
  lambda=seq(0, 100, by = 0.1)
  )

mcs_lasso_caret <- midas_LASSO_caret(
  yX_1.1,
  yX_1.2,
  y_1.2,
  trncntrl = myTrainControl,
  grd = grid_L
  )

plot(mcs_lasso_caret$lasso_train)
```

3.1.7.2 GLMNET

```
mcs_lasso <- midas_LASSO(
  train.x = X_1.1,
  train.y = y_1.1,
  test.x = X_1.2,
  test.y = y_1.2
)
```

3.1.8 Comparison

```
resamps_2.1 <- resamples(list(singleTree = mcs_rpart$rpart_tree,
                              randomForest = mcs_rF$rf_tree,
                              GBM_auto = mcs_gbm_auto$gbm_tree_auto,
                              GBM_tuned = mcs_gbm_tune$gbm_tree_tune,
                              LASSO_caret = mcs_lasso_caret$lasso_train)
                         )
summary(resamps_2.1)
cat(
  "The out-sample RMSE of nloptR correct specification: ",
    nloptr_correct_outsample_rmse, "\n",
  "The out-sample RMSE of nloptR miss specification: ",
    nloptr_false_outsample_rmse, "\n"
)
```

3.2 Real Data Example

3.2.1 Data Processing

```
data("rvsp500", package = "midasr")
spx2_rvol <- 100 * sqrt(252 * as.numeric(rvsp500[, "SPX2.rv"]))

d_rv = data.frame(y = spx2_rvol, X = mls(spx2_rvol, 1:20, 1)) %>% na.omit()
TT = nrow(d_rv)

d_rv_train = d_rv[1:3000,]
d_rv_test  = d_rv[3021:TT,] # starting from 3021 so the training data
                            # and testing data have no overlap

# time series validation
# notice that the time series validation is different from iid case
# the train data is cut into consecutive small subsamples of length 201
# the first 200 observations are used for estimation and the last 1
# observation for forecast
```

3.2.2 Time Train Control

```
myTimeControl <- trainControl(
  method = "timeslice",
  initialWindow = 200,
  horizon = 1,
  fixedWindow = TRUE
)
```

3.2.3 nloptR:

```
#### Special treatment for nloptR -> copy X_@@ to X
X <- data.matrix(select(d_rv_train, -c(y)))
y <- data.matrix(select(d_rv_train, c(y)))

#### Correct Specification
cat("Correct Specification: \n")
algo1.1 <- midas_nloptr(X = X, y = y, x0 = x0, x1 = x1)
algo1.1_sol <- algo1.1[["solution"]]
y.hat_nloptr_corr_insample_sol <- y_pred(algo1.1_sol)
nloptr22_corr_insample_rmse <-
  sqrt(mean((y.hat_nloptr_corr_insample_sol - d_rv_train[y]) ^ 2))


X <- data.matrix(select(d_rv_test, -c(y)))
y <- data.matrix(select(d_rv_test, c(y)))
y.hat_nloptr_corr_outsample_sol <- y_pred(algo1.1_sol)
```

```
nloptr22_corr_outsample_rmse <-
  mean(sqrt((y.hat_nloptr_corr_outsample_sol - d_rv_test[y]) ^ 2))
cat(
  "in-sample RMSE:", nloptr22_corr_insample_rmse, "\n",
  "out-sample RMSE:", nloptr22_corr_outsample_rmse, "\n"
)


#### Misspecification
cat("Misspecification: \n")
X <- data.matrix(select(d_rv_train, -c(y)))
y <- data.matrix(select(d_rv_train, c(y)))
algo1.2 <- midas_nloptr(X = X, y = y, x0 = x0, x1 = x1)
algo1.2_sol <- algo1.2[["solution"]]
y.hat_nloptr_false_insample_sol <- y_pred(algo1.2_sol)
nloptr22_false_insample_rmse <-
  sqrt(mean((y.hat_nloptr_false_insample_sol - d_rv_train[y]) ^ 2))


X <- data.matrix(select(d_rv_test, -c(y)))
y <- data.matrix(select(d_rv_test, c(y)))
y.hat_nloptr_false_outsample_sol <- y_pred(algo1.2_sol)
nloptr22_false_outsample_rmse <-
  mean(sqrt((y.hat_nloptr_false_outsample_sol - d_rv_test[y]) ^ 2))
cat(
  "in-sample RMSE:", nloptr22_false_insample_rmse, "\n",
  "out-sample RMSE:", nloptr22_false_outsample_rmse, "\n"
)
```

3.2.4 Decision Tree's

3.2.4.1 Single Tree

```
algo2.1 <- midas_rpart(
  train.dgp = d_rv_train,
  test.dgp = d_rv_test,
  test.y = d_rv_test["y"],
  trncntrl = myTimeControl)

algo2.1$rpart_tree$results
```

Here we can observe, that with the default settings, no convergence was found. The variance within certain parameters were zero.

3.2.4.2 randomForest

```
algo2.2 <- midas_rF(
  train.dgp = d_rv_train,
  test.dgp = d_rv_test,
```

```
  test.y = d_rv_test$y,
  trncntrl = myTimeControl
  )
```

3.2.5 Gradient Boosting Method

3.2.5.1 Auto Tuned Boosting

```
algo3.1 <- midas_gbm_auto(
  train.dgp = d_rv_train,
  test.dgp = d_rv_test,
  test.y = d_rv_test$y,
  trncntrl = myTimeControl
  )
```

3.2.5.2 Tuned Boosting

```
grid_gbm <- expand.grid(
  n.trees = (9:12 * 1000),
  shrinkage = c(0.0001, 0.001, 0.01),
  interaction.depth = c(1, 16),
  n.minobsinnode = c(1)
)

algo3.2 <- midas_gbm_tuned(
  train.dgp = d_rv_train,
  test.dgp = d_rv_test,
  test.y = d_rv_test$y,
  trncntrl = myTimeControl,
  grd = grid_gbm
  )
```

3.2.6 LASSO

3.2.6.1 Caret LASSO

```
grid_L <- expand.grid(
  alpha=1,
  lambda=seq(0, 100, by = 0.1)
  )

algo4.1 <- midas_LASSO_caret(
  train.dgp = d_rv_train,
  test.dgp = d_rv_test,
  test.y = d_rv_test$y,
  trncntrl = myTimeControl,
  grd = grid_L)
```

3.2.6.2 GLMNET LASSO

```
#### Special Treatment for LASSO
rv_train.X <- data.matrix(select(d_rv_train, -c(y)))
rv_train.y <- data.matrix(select(d_rv_train, c(y)))

rv_test.X <- data.matrix(select(d_rv_test, -c(y)))
rv_test.y <- data.matrix(select(d_rv_test, c(y)))

algo4.2 <- midas_LASSO(
  train.x = rv_train.X,
  train.y = rv_train.y,
  test.x = rv_test.X,
  test.y = rv_test.y)
```

3.2.7 Comparison

```
cat(
  "The out-sample RMSE of nloptR correct specification: ",
    nloptr22_corr_outsample_rmse, "\n",
  "The out-sample RMSE of nloptR miss specification: ",
    nloptr22_false_outsample_rmse, "\n",
  "The out-sample RMSE of singleTree method: ",
    algo2.1$outsample_RMSE, "\n",
  "The out-sample RMSE of randomForest method: ",
    algo2.2$outsample_RMSE, "\n",
  "The out-sample RMSE of auto tuned GBM: ",
    algo3.1$outsample_RMSE, "\n",
  "The out-sample RMSE of tuned GBM: ",
    algo3.2$outsample_RMSE, "\n",
  "The out-sample RMSE of LASSO caret: ",
    algo4.1$outsample_RMSE, "\n",
  "The out-sample RMSE of LASSO glmnet: ",
    algo4.2$outsample_RMSE, "\n"
)
```

Similar to results from Monte Carlo simulations, machine learning methods turn out to be more accurate in terms of smaller RMSE. In which, LASSO caret has the least out of sample RMSE. This is followed by random forest and auto tuned GBM. This is saying, with the aid of computing power, we are able to obtain better forecasts than non linear optimisation method.

3.2.8 Plotting

```
snp <- rvsp500 %>% transform(DateID = as.Date(as.character(DateID), "%Y%m%d")) %>%
  mutate(spx2_rvol = 100 * sqrt(252 * as.numeric(rvsp500[, "SPX2.rv"])))

snp_test <- snp[3021:TT,]

ggplot(data = snp_test, aes(x = DateID, y = spx2_rvol)) +
```

```
  geom_line() +
  geom_line(aes(y = y.hat_nloptr_corr_outsample_sol,
                colour = "red",
                linetype = "1" )) +
  geom_line(aes(y = y.hat_nloptr_false_outsample_sol,
                colour = "blue",
                linetype = "2")) +
  scale_color_manual(labels = c("Correct", "Missspecified"),
                     values = c("blue", "red")) +
  labs(title = "nloptR - Comparison",
       subtitle = "True Values Against Predicted",
       y = "Volatility",
       x = "Time")

ggplot(data = snp_test, aes(x = DateID, y = spx2_rvol)) +
  geom_line() +
  geom_line(aes(y = algo2.1$yhat.rpart_tree,
                color = "blue",
                linetype = "1" )) +
  geom_line(aes(y = algo2.2$yhat.rf_tree,
                color = "red",
                linetype = "2")) +
  scale_color_manual(labels = c("SingleTree", "RandomForest"),
                     values = c("blue", "red")) +
  labs(title = "Decision Tree - Comparison",
     subtitle = "True Values Against Predicted",
     y = "Volatility",
     x = "Time")

ggplot(data = snp_test, aes(x = DateID, y = spx2_rvol)) +
  geom_line() +
  geom_line(aes(y = algo3.1$yhat.midas_gbm_auto,
                color = "blue",
                linetype = "1" )) +
  geom_line(aes(y = algo3.2$yhat.midas_gbm_tune,
                color = "red",
                linetype = "2")) +
  scale_color_manual(labels = c("Auto Tuned", "Tuned"),
                     values = c("blue", "red")) +
  labs(title = "Boosting - Comparison",
     subtitle = "True Values Against Predicted",
     y = "Volatility",
     x = "Time")

ggplot(data = snp_test, aes(x = DateID, y = spx2_rvol)) +
  geom_line()+
  geom_line(aes(y = algo4.1$yhat.lasso_caret,
```

```
              color = "blue",
              linetype = "1" )) +
  geom_line(aes(y = algo4.2$y.hat.lasso_glmnet,
              color = "red",
              linetype = "2")) +
  scale_color_manual(labels = c("Caret", "glmnet"),
                     values = c("blue", "red")) +
  labs(title = "LASSO - Comparison",
      subtitle = "True Values Against Predicted",
      y = "Volatility",
      x = "Time")
```