Postshock

Qiyang Wang

2025-04-09

Contents

Postshock package

This report presents a unified framework that integrates Synthetic Control methods with time series forecasting models (such as ARIMA, GARCH, and HAR) to assess and predict the impact of exogenous "shock events" on financial time series.

In many real-world financial and economic scenarios, individual assets—such as stocks, indices, or volatility measures—may experience abrupt changes due to news, policies, or market events. The objective of this framework is to: 1. Construct a synthetic predictor using unaffected "donor" time series. 2. Extract the fixed effects (shock responses) from the donor pool using regression-based methods. 3. Fit a time series model to the target asset and adjust the prediction using the estimated aggregate shock effect. 4. Visualize and evaluate model predictions against ground truth using statistical loss metrics (e.g., MSE, Quasi-Likelihood Loss).

The key components of the framework include: - **SynthPrediction**: Forecasting log-returns using ARIMA combined with synthetic control. - **SynthVolForecast**: Forecasting volatility using GARCH and synthetic fixed effects from donors. - **HAR**: Modeling high-frequency realized volatility using Heterogeneous Autoregressive structures.

This report will walk through the implementation of each model using either simulated or real data, demonstrating their effectiveness in capturing and adjusting for post-shock behavior.

```
### START QL_loss_function
QL_loss_function <- function(pred, gt){gt/pred - log(gt/pred) - 1}
### END QL_loss_function

#We specify some functions for transforming y

#mean_square_y will be used in garch models
mean_square_y <- function(y){return((y-mean(y))**2)}

#identity function will be used in synthetic prediction models
id <- function(y){return(y)}
```

Function1:distance-based weighting function

```
### START dbw
dbw <- function(X                          # List of covariates for time series; X[[1]] is target, X[[2:
                ,dbw_indices               # Indices of covariates to use
                ,shock_time_vec            # Vector of shock times for each series
                ,scale = FALSE             # Whether to scale data for standardization
                ,center = FALSE            # Whether to center data (subtract mean)
                ,sum_to_1 = 1              # Whether weights must sum to 1
                ,bounded_below_by = 0      # Lower bound for weights, default is 0 (non-negative)
                ,bounded_above_by = 1      # Upper bound for weights, default is 1
                ,princ_comp_count = NULL   # Number of principal components to use, NULL means no dimens
                ,normchoice = c('l1', 'l2')[2] # Choice of norm, default is L2
                ,penalty_normchoice = c('l1', 'l2')[1] # Choice of penalty norm, default is L1
                ,penalty_lambda = 0        # Regularization parameter, default 0 means no regularization
                ,Y = NULL                  # Optional additional time series data
                ,Y_lookback_indices = list(seq(1,1,1)) # Lag structure indices for Y
                ,X_lookback_indices = rep(list(c(1)),length(dbw_indices)) # Lag structure indices for X
                ,inputted_transformation   # Transformation function to apply to Y (e.g., mean_square_y
) { # Reference: https://github.com/DEck13/synthetic_prediction/blob/master/prevalence_testing/numerica

  # Calculate size of donor pool (total time series minus 1 for target)
  n <- length(X) - 1

  # Extract number from normchoice string ('l1'->1, 'l2'->2) for norm function
  normchoice_number <- unlist(strsplit(normchoice, split = ""))[2]

  # Check if model is overparameterized: number of variables p > number of donors n
  p <- length(dbw_indices)
  print('Here is the number of covariates used in dbw:') # Print number of covariates used
  print(p)
  if (p > n){cat('p > n, i.e. system is overdetermined from an unconstrained point-of-view.')} # Warnin
```

```r
## Now perform the complex task of extracting specified lag structures
## for each time series i=1,2,...,n+1 and each covariate

# Define function to extract specified columns from dataframes
col_returner <- function(df){return(df[,dbw_indices])}
print('col_returner succeeded')

# Apply col_returner to all time series X to extract covariates of interest
X_subset1 <- lapply(X, col_returner)
print('col_returner with lapply succeeded')

# If Y lookback indices are provided, process Y data and merge with X data
if (is.null(Y_lookback_indices) == FALSE){

  print('User has provided Y_lookback_indices, so we include them.')
  X_lookback_indices <- c(Y_lookback_indices, X_lookback_indices) # Combine Y and X lookback indices

  # Define function to transform Y and merge with X
  X_Y_combiner <- function(y,x) {
        # Print transformation function info for debugging
        print('We print the transformation and its class')
        print(inputted_transformation)
        print(class(inputted_transformation))

        # Apply specified transformation to y (like mean_square_y or id)
        transformed_series <- inputted_transformation(y)

        # Merge transformed y with x data, all=FALSE means keep only common timepoints
        return(merge(transformed_series,x, all = FALSE))
  } #end X_Y_combiner

  # Print class and length info for Y and X data
  print('We are about to combine X_subset1 and Y...')
  print(class(Y))
  print(length(Y))
  print(class(X_subset1))
  print(length(X_subset1))

  # Apply X_Y_combiner to Y and X data in parallel using mapply
  combined_X <- mapply(X_Y_combiner
                       , y = Y
                       , x = X_subset1
                       , SIMPLIFY = FALSE)
}
else{
  # If no Y data, just use X_subset1
  combined_X <- X_subset1
}

# Define function to extract data up to shock time for each series
row_returner <- function(df, stv){
  print(paste('Shock occurs at ', stv, sep = ''))
  print(paste('Row count of the series is ', nrow(df), sep = ''))
```

```
  return(df[1:(stv),]) # Return all rows from 1 to shock time
}

# Apply row_returner to all merged data, and using the simplify to make it a list
X_subset2 <- mapply(row_returner, df = combined_X, stv = shock_time_vec, SIMPLIFY=FALSE)

# Define function to extract covariate values according to specified lag indices
cov_extractor <- function(X_df){
      # Get row count of dataframe
      len <- nrow(X_df)

      # Define inner function to create boolean vector with TRUE at specified indices
      padded_vector_maker <- function(x)
          {
          vec <- rep(FALSE,len) # Create all-FALSE vector
          vec[x] <- TRUE        # Set specified indices to TRUE
          vec_reversed <- rev(vec) # Reverse vector (count from end)
          return(vec_reversed)
          }

      # Apply padded_vector_maker to each lag index to create list of boolean vectors
      covariate_vals_in_list <- lapply(X_lookback_indices, padded_vector_maker)
      # Convert list of boolean vectors to matrix
      boolmat <- as.matrix(do.call(data.frame, covariate_vals_in_list))

      # Extract specified values from dataframe using boolean matrix
      return(as.matrix(X_df)[boolmat])
}

# Apply cov_extractor to all data
X_subset <- lapply(X_subset2, cov_extractor)

# Bind all processed data into a single matrix
dat <- do.call('rbind', X_subset)
print('Pre-scaling') # Print data before scaling
print(dat)

# Apply scaling and centering according to parameters
if (scale == TRUE) {cat('User has chosen to scale covariates.')}
if (center == TRUE) {cat('User has chosen to center covariates.')}

# Apply scale function to each column for standardization and centering
dat <- apply(dat, 2, function(x) scale(x, center = center, scale = scale))
print('Post-scaling (nothing will happen if center and scale are set to FALSE).')
print(dat) # Print processed data

# Compute singular value decomposition for principal component analysis
dat.svd <- svd(dat)
sing_vals <- dat.svd$d / sum(dat.svd$d) # Calculate proportion of variance explained
print('Singular value percentages for the donor pool X data:')
print(paste(round(100 * sing_vals,2), "%", sep = "")) # Print singular value percentages

# If principal component count specified, perform dimension reduction
```

```r
if (is.null(princ_comp_count) == FALSE){
  print(paste('We are using ', princ_comp_count, ' principal components.', sep = ''))
  print(dat.svd$v) # Print right singular vectors (principal component directions)
  # Project data onto first princ_comp_count principal components
  dat <- dat %*% dat.svd$v[,1:princ_comp_count]
}

# Separate data into target series and donor series
X1 <- dat[1, , drop = FALSE] # Target series (first row)
X0 <- split(dat[-1,], seq(nrow(dat[-1,]))) # Donor series (remaining rows), split into list

print('We inspect X1.') # Print target series
print(X1)
print('We inspect X0.') # Print donor series
print(X0)

# Define objective function: distance between weighted combination of donors and target
weightedX0 <- function(W) {
  # W is weight vector with length equal to X0's length
  n <- length(W)
  p <- ncol(X1)
  XW <- matrix(0, nrow = 1, ncol = p) # Initialize weighted combination matrix

  # Calculate weighted sum
  for (i in 1:n) {
    XW <- XW + W[i] * X0[[i]]
  } #end of loop

  # Calculate distance between target and weighted combination using specified norm
  norm_output <- as.numeric(norm(matrix(X1 - XW), type = normchoice_number))

  # Add regularization penalty
  if (penalty_normchoice == 'l1' & penalty_lambda > 0) {
    # L1 regularization (like LASSO)
    norm_output <- norm_output + penalty_lambda * norm(as.matrix(W), type = "1")
  }
  else if (penalty_normchoice == 'l2' & penalty_lambda > 0) {
    # L2 regularization (like Ridge)
    norm_output <- norm_output + penalty_lambda * as.numeric(crossprod(matrix(W)))
  }
  else {norm_output <- norm_output} # No regularization

  return(norm_output)
} #end objective function

# Optimization to find optimal weights

# The function has been enhanced with features:
# 1) Option to remove sum-to-1 constraint
# 2) Option to change lower bound to -1 or NA (no limit)
# 3) Option to change upper bound to NA (no limit)
# 4) Option to choose L1 or L2 norm as distance function
```

```r
# Set up constraints according to parameters

# Set up sum-to-1 constraint
if (is.na(sum_to_1) == FALSE) {
  eq_constraint <- function(W) sum(W) - 1 # Sum of weights minus 1
}
else{
  eq_constraint = NULL # No sum-to-1 constraint
}

# Set up lower bound constraint
if (is.na(bounded_below_by) == FALSE) {
  lower_bound = rep(bounded_below_by, n) # Same lower bound for all weights
}
else if (is.na(bounded_below_by) == TRUE) {
  lower_bound = NULL # No lower bound
}

# Set up upper bound constraint
if (is.na(bounded_above_by) == FALSE) {
  upper_bound = rep(bounded_above_by, n) # Same upper bound for all weights
}
else if (is.na(bounded_above_by) == TRUE) {
  upper_bound = NULL # No upper bound
}

# Solve constrained optimization problem using Rsolnp package
object_to_return <- Rsolnp::solnp(
                          par = rep(1/n, n),         # Initial values: equal weights
                          fun = weightedX0,          # Objective function
                          eqfun = eq_constraint,     # Equality constraint function
                          eqB = 0,                  # Right-hand side of equality constraint
                          LB = lower_bound, UB = upper_bound, # Bounds
                          control = list(trace = 1   # Output optimization process
                                         , 1.0e-12   # Control parameter
                                         , tol = 1e-27 # Tolerance
                                         , outer.iter = 1000000000 # Max outer iterations
                                         , inner.iter = 10000000)) # Max inner iterations

# Calculate final loss value
loss <- round(norm(X1 - object_to_return$pars %*% dat[-1,], type = normchoice_number),3)

# Print loss information
print(paste('The loss of distanced-based weighting is ', loss, ',',
            ' which is ',
            100*round(loss/norm(X1, type=normchoice_number),3),
            "% of the norm of the vector we are trying to approximate.", sep = ""))

# Check if optimization converged
if (object_to_return$convergence == 0){
  convergence <- 'convergence' # Convergence successful
}
else {
```

```
    convergence <- 'failed_convergence' # Convergence failed
  }

  # Create return result list
  pair_to_return <- list(object_to_return$pars, convergence)

  # Add names to result list
  names(pair_to_return) <- c('opt_params', 'convergence')

  # Return optimal weights and convergence status
  return(pair_to_return)

} #END dbw function
### END dbw
```

Distance-Based Weighting (DBW) Function

Overview The DBW function is a sophisticated optimization tool that finds optimal donor weights for synthetic volatility forecasting. It combines dimensionality reduction techniques with constrained optimization to create weighted combinations of donor time series that best match target series characteristics.

Key Components

1. **Data Preprocessing**
   - Extracts relevant covariates
   - Applies user-specified transformations
   - Handles lag structures
   - Trims data to pre-shock periods
2. **Dimensionality Reduction**
   - Optional standardization and centering
   - SVD-based principal component analysis
   - Projects data onto principal components
   - Separates target and donor series
3. **Weight Optimization**
   - Objective function minimizes distance between weighted donors and target
   - Supports L1/L2 regularization
   - Flexible constraints (sum-to-1, non-negativity)
   - Solves using sequential quadratic programming
4. **Results Evaluation**
   - Calculates final loss metrics
   - Reports convergence status
   - Returns optimal weights

Implementation Features

- Adjustable norm choice for distance measurement
- Tunable regularization parameters
- Flexible constraint configuration
- Comprehensive diagnostics

Financial Applications In volatility forecasting, this function significantly improves predictions during market shocks by: - Learning from historical similar events - Reducing prediction bias in traditional volatility models - Providing more reliable risk estimations - Adapting to various market conditions and asset classes

Function2:plot_maker_garch why here is no donor plot?

```
### START GARCH plot_maker_garch
#only accept the volatility
plot_maker_garch <- function(fitted_vol
                             ,shock_time_labels = NULL
                             ,shock_time_vec #mk
                             ,shock_length_vec
                             ,unadjusted_pred
                             ,w_hat
                             #,omega_star_hat(this variable is not used)erdf
                             ,omega_star_hat_vec
                             ,omega_star_std_err_hat_vec
                             ,adjusted_pred
                             ,arithmetic_mean_based_pred
                             ,ground_truth_vec){

  if (is.character(shock_time_labels) == FALSE | is.null(shock_time_labels) == TRUE){
    shock_time_labels <- 1:length(shock_time_vec)
  }

  par(mfrow = c(1,3), mar=c(15,9,4,2))

  barplot_colors <- RColorBrewer::brewer.pal(length(w_hat),'Set3')

  #PLOT ON THE LEFT:
print('We plot the weights.')
  # Plot donor weights
  barplot(w_hat
          , main = 'Donor Pool Weights'
          , names.arg = shock_time_labels[-1]
          , cex.names=1.3
          , cex.main=1.5
          , las=2
          , col = barplot_colors
          )

  #PLOT IN THE MIDDLE
  print('We plot the FE estimates.')
  #Plot FE estimates
    bp <- barplot(omega_star_hat_vec
          , main = 'Donor-Pool-Supplied\n FE Estimates\nand Standard Errors Estimates'
          , names.arg = shock_time_labels[-1]
          , cex.names=1.4
          , cex.main=1.5
          , las=2
          , col = barplot_colors
          , ylim = c(0, 1.4 * max(omega_star_hat_vec)) )
```

```
  # Add the labels with some offset to be above the bar
  print('We print the std errors')
  print(omega_star_std_err_hat_vec)
  #omega_star_std_err_hat_vec <- ifelse(omega_star_std_err_hat_vec, nan)

  #https://stackoverflow.com/questions/65057352/how-to-add-labels-above-the-bar-of-barplot-graphics
  text(x = bp,
              ,y = omega_star_hat_vec + .00029
              ,cex = 1.3
              ,labels = round(omega_star_std_err_hat_vec, 5)
              , srt= 90)

title(ylab = expression(sigma^2), line = 3.05, cex.lab = 1.99) # Add y-axis text

#Plot target series and prediction

thing_to_get_max_of <- c(as.numeric(fitted_vol)
                      , unadjusted_pred
                      , adjusted_pred
                      , ground_truth_vec
                      , arithmetic_mean_based_pred
                      )

max_for_y_lim <- max(thing_to_get_max_of)

x_ax_first_point_of_shock <- index(fitted_vol)[shock_time_vec[1]-1] + 1 #do I use this?
x_ax_end_point <- index(fitted_vol)[shock_time_vec[1]-1] + length(adjusted_pred)

#PLOT ON THE RIGHT:
print('We plot the fitted volatility series.')
plot(y = fitted_vol[1:shock_time_vec[1]], #mk
        x = index(fitted_vol)[1:shock_time_vec[1]],
     main = 'Post-Shock Volatility Forecast', #mk can improve this title
     cex.main=1.5,
     ylab = '',
     type="l",
     xlab = '',
     xlim =  as.Date(c(index(fitted_vol)[1], x_ax_end_point)),
     ylim = c(min(0, fitted_vol),  max_for_y_lim))

title(ylab = expression(sigma^2), line = 2.05, cex.lab = 1.99) # Add y-axis text

# Here is the color scheme we will use
# https://colorbrewer2.org/?type=diverging&scheme=RdYlBu&n=4
colors_for_adjusted_pred <- c('#d7191c','#fdae61','#abd9e9','#2c7bb6')

# Let's add the plain old GARCH prediction
points(y = unadjusted_pred
       ,x = x_ax_first_point_of_shock:x_ax_end_point
       ,col = colors_for_adjusted_pred[1]
       ,cex = 3.5
       ,pch = 15)
```

```
  # Now plot the adjusted predictions
  points(y = adjusted_pred
         ,x = x_ax_first_point_of_shock:x_ax_end_point
         ,col = colors_for_adjusted_pred[2]
         ,cex = 3.5
         ,pch = 18)

  # Now plot the arithmetic mean-based predictions
  points(y = arithmetic_mean_based_pred
         ,x = x_ax_first_point_of_shock:x_ax_end_point
         ,col = colors_for_adjusted_pred[3]
         ,cex = 3.5
         ,pch = 19)

  # Now plot Ground Truth tk
  if (is.null(ground_truth_vec) == FALSE)
    {
    points(y = ground_truth_vec
           ,x = x_ax_first_point_of_shock:x_ax_end_point
           ,col = colors_for_adjusted_pred[4]
           ,cex = 3.5
           ,pch = 17)
  }

  labels_for_legend <- c('GARCH (unadjusted)'
                         , 'Adjusted'
                         , 'Arithmetic Mean'
                         , 'Ground Truth'
                         )

  legend(x = "topleft",  # Coordinates (x also accepts keywords) #mk
         legend = labels_for_legend,
         1:length(labels_for_legend), # Vector with the name of each group
         colors_for_adjusted_pred,   # Creates boxes in the legend with the specified colors
         title = 'Prediction Method',      # Legend title,
         cex = .9)

  #par(mfrow = c(1,1), mar=c(15,6,4,2))

}
### END plot_maker_garch
```

Plotting Function for Synthetic Prediction The `plot_maker_garch` function creates a three-panel visualization for evaluating synthetic volatility forecasts:

1. **Left panel**: Bar plot showing optimal donor weights
2. **Middle panel**: Fixed effect estimates with standard errors
3. **Right panel**: Comparison of volatility forecasts including:
    - Pre-shock fitted volatility (line)
    - Standard GARCH predictions
    - Synthetic-adjusted predictions
    - Arithmetic mean-adjusted predictions

- Ground truth values (when available)

The function handles missing labels automatically, calculates appropriate axis bounds, and uses a consistent color scheme across panels. It provides a comprehensive visualization tool for assessing model performance during market shocks.

Key inputs include fitted volatility series, shock timing information, donor weights, fixed effect estimates, and various prediction values.

```
### START plot_maker_synthprediction
plot_maker_synthprediction <- function(
    Y,                      # List of time series data
    shock_time_labels = NULL,  # Optional labels for shock times
    shock_time_vec,         # Vector of shock times for each series
    shock_length_vec,       # Vector of shock lengths
    unadjusted_pred,        # Unadjusted predictions
    w_hat,                  # Donor pool weights
    #omega_star_hat,         # Unused in the function body
    omega_star_hat_vec,     # Donor pool fixed effects estimates
    adjusted_pred,          # Adjusted predictions
    display_ground_truth = FALSE  # Option to show actual data
){


  if (is.character(shock_time_labels) == FALSE | is.null(shock_time_labels) == TRUE){
    shock_time_labels <- 1:length(shock_time_vec)
  }

  n <- length(Y) - 1

  #First print donor series
  par(mfrow = c(round(sqrt(n)),ceiling(sqrt(n))))

  for (i in 2:(n+1)){
    plot.ts(Y[[i]][1:shock_time_vec[i]]
            ,xlab = ' '
            ,ylab = 'Log Return'
            ,main = paste('Donor ', i,': ', shock_time_labels[i], sep = '')
            ,xlim = c(0, shock_time_vec[i] + 5)
            ,ylim = c(min(Y[[i]]),  max(Y[[i]]))
            )

    if (display_ground_truth == TRUE){

      lines(y = Y[[i]][shock_time_vec[i]:(shock_time_vec[i] + shock_length_vec[i])]
            ,x = shock_time_vec[i]:(shock_time_vec[i] + shock_length_vec[i])
            ,col = 'purple'
            ,cex = 1.1
            ,lty = 3)

      points(y = Y[[i]][(shock_time_vec[i]+1):(shock_time_vec[i] + shock_length_vec[i])]
             ,x = (shock_time_vec[i]+1):(shock_time_vec[i] + shock_length_vec[i])
             # ,col = 'red'
             ,cex = 1.1
```

```
            ,pch = 17)

  }
}

#Now print time series under study
par(mfrow = c(1,3), mar=c(15,6,4,2))

barplot_colors <- RColorBrewer::brewer.pal(length(w_hat),'Set3')

#PLOT ON THE LEFT:
#Plot donor weights
barplot(w_hat
        , main = 'Donor Pool Weights'
        , names.arg = shock_time_labels[-1]
        , cex.names=.95
        , las=2
        , col = barplot_colors)

#PLOT IN THE MIDDLE

#Plot FE estimates
barplot(omega_star_hat_vec
        , main = 'Donor-Pool-Supplied \n FE Estimates'
        , names.arg = shock_time_labels[-1]
        , cex.names=.95
        , las=2
        , col = barplot_colors)

#Plot target series and prediction

thing_to_get_max_of <- c(as.numeric(Y[[1]]), unadjusted_pred, adjusted_pred)

max_for_y_lim <- max(thing_to_get_max_of)

#PLOT ON THE RIGHT:
plot.ts(Y[[1]][1:shock_time_vec[1]], #mk
        main = 'Post-shock Forecasts',
        ylab = '',
        xlab = ' ',
        xlim = c(0, shock_time_vec[1] + 5), #mk
        ylim = c(min(0, Y[[1]]),  max_for_y_lim))

title(ylab = 'Log Return', line = 2.05, cex.lab = 1.99) # Add y-axis text

# Here is the color scheme we will use
#https://colorbrewer2.org/?type=diverging&scheme=RdYlBu&n=4
colors_for_adjusted_pred <- c('#d7191c','#fdae61','#abd9e9')

# Let's add the plain old GARCH prediction
points(y = unadjusted_pred
       ,x = (shock_time_vec[1]+1):(shock_time_vec[1]+shock_length_vec[1])
       ,col = colors_for_adjusted_pred[1]
```

```
        ,cex = .9
        ,pch = 15)

  # Now plot the adjusted predictions
  points(y = adjusted_pred
        ,x = (shock_time_vec[1]+1):(shock_time_vec[1]+shock_length_vec[1])
        ,col = colors_for_adjusted_pred[2]
        ,cex = 2
        ,pch = 19)

  if (display_ground_truth == TRUE){

    lines(y = Y[[1]][shock_time_vec[1]:(shock_time_vec[1] + shock_length_vec[1])]
          ,x = shock_time_vec[1]:(shock_time_vec[1] + shock_length_vec[1])
          ,col = colors_for_adjusted_pred[3]
          ,cex = 1.1
          ,lty = 3)

    points(y = Y[[1]][(shock_time_vec[1]+1):(shock_time_vec[1] + shock_length_vec[1])]
          ,x = (shock_time_vec[1]+1):(shock_time_vec[1] + shock_length_vec[1])
          ,col = colors_for_adjusted_pred[3]
          ,cex = 1.1
          ,pch = 24)

  }

  labels_for_legend <- c('ARIMA (unadjusted)', 'Adjusted Prediction', 'Actual')

  legend(x = "topleft",  # Coordinates (x also accepts keywords) #mk
         legend = labels_for_legend,
         1:length(labels_for_legend), # Vector with the name of each group
         colors_for_adjusted_pred,   # Creates boxes in the legend with the specified colors
         title = 'Prediction Method',      # Legend title,
         cex = .9)

}
### END plot_maker_synthprediction
```

Function3:plot_maker_HAR

```
### HAR (Heterogeneous Autoregressive) Plot Maker Function
# Purpose: Visualize HAR model predictions and donor series analysis
plot_maker_HAR <- function(
    Y,                      # List of time series data
    shock_time_labels = NULL,  # Optional labels for shock times
    shock_time_vec,         # Vector of shock times for each series
    shock_length_vec = 1,  # Vector of shock lengths (default 1)
    unadjusted_pred,        # Initial (unadjusted) predictions
    w_hat,                  # Donor pool weights
    omega_star_hat_vec,     # Donor pool fixed effect estimates
    adjusted_pred,          # Adjusted predictions
    display_ground_truth = FALSE  # Option to show actual data
```

```
){
    # Calculate number of donor series
    n <- length(shock_time_vec) - 1

    # Prepare shock time labels
    # If no labels provided or invalid, generate default numeric labels
    if (is.character(shock_time_labels) == FALSE | is.null(shock_time_labels) == TRUE){
        shock_time_labels <- 1:(n+1)
    }

    # Set up grid layout for donor series plots
    # Create near-square grid based on number of donor series
    par(mfrow = c(round(sqrt(n)),ceiling(sqrt(n))))

    # Plot each donor series
    for (i in 2:(n+1)){
        plot.ts(Y[[i]][1:shock_time_vec[i]]
                ,xlab = ' '
                ,ylab = 'Realized Measure of Volatility'
                ,main = paste('Donor ', i,': ', shock_time_labels[i], sep = '')
                ,xlim = c(0, shock_time_vec[i] + 5)
                ,ylim = c(min(Y[[i]]),  max(Y[[i]]))
        )

        # Optionally display ground truth data
        if (display_ground_truth == TRUE){
            # Plot actual data line
            lines(y = Y[[i]][shock_time_vec[i]:(shock_time_vec[i] + shock_length_vec[i])]
                  ,x = shock_time_vec[i]:(shock_time_vec[i] + shock_length_vec[i])
                  ,col = 'purple'
                  ,cex = 1.1
                  ,lty = 3)

            # Plot actual data points
            points(y = Y[[i]][(shock_time_vec[i]+1):(shock_time_vec[i] + shock_length_vec[i])]
                   ,x = (shock_time_vec[i]+1):(shock_time_vec[i] + shock_length_vec[i])
                   ,cex = 1.1
                   ,pch = 17)
        }
    }

    # Set up main visualization panel
    par(mfrow = c(1,3), mar=c(15,6,4,2))

    # Generate color palette
    barplot_colors <- RColorBrewer::brewer.pal(length(w_hat),'Set3')

    # LEFT PLOT: Donor Pool Weights
    barplot(w_hat
            , main = 'Donor Pool Weights'
            , names.arg = shock_time_labels[-1]
            , cex.names=.95
            , las=2
```

```
        , col = barplot_colors)

# MIDDLE PLOT: Fixed Effects Estimates
barplot(omega_star_hat_vec
        , main = 'Donor-Pool-Supplied \n FE Estimates'
        , names.arg = shock_time_labels[-1]
        , cex.names=.95
        , las=2
        , col = barplot_colors)

# Prepare data for target series plot
Y_to_plot <- Y[[1]][(shock_time_vec[1]-45):(shock_time_vec[1]-1)]
x_ax_first_point_of_shock <- index(Y[[1]])[(shock_time_vec[1]-45)]
x_ax_end_point <- index(Y[[1]])[shock_time_vec[1]] + 5

# Calculate y-axis limit
thing_to_get_max_of <- c(as.numeric(Y[[1]])
                         , unadjusted_pred
                         , adjusted_pred)
max_for_y_lim <- max(thing_to_get_max_of)

# RIGHT PLOT: Target Series and Predictions
plot(y = Y_to_plot
     ,x = index(Y_to_plot)
     ,main = 'Post-Shock Volatility Forecast'
     ,cex.main=1.5
     ,ylab = ''
     ,type="l"
     ,xlab = ''
     ,xlim = as.Date(c(x_ax_first_point_of_shock, x_ax_end_point))
     ,ylim = c(min(0, Y_to_plot),  max_for_y_lim)
)
title(ylab = 'Realized Measure of Volatility', line = 2.05, cex.lab = 1.99)

# Color scheme for predictions
colors_for_adjusted_pred <- c('#d7191c','#fdae61','#abd9e9')

# Add unadjusted HAR prediction
points(y = unadjusted_pred
       ,x = index(Y[[1]])[shock_time_vec[1]]
       ,col = colors_for_adjusted_pred[1]
       ,cex = 2
       ,pch = 19)

# Add adjusted HAR prediction
points(y = adjusted_pred
       ,x = index(Y[[1]])[shock_time_vec[1]]
       ,col = colors_for_adjusted_pred[2]
       ,cex = 2
       ,pch = 19)

# Optionally display ground truth
if (display_ground_truth == TRUE){
```

```
        Y_not_yet_plotted <- Y[[1]][shock_time_vec[1]:(shock_time_vec[1]+1)]
        connecting_the_two <- Y[[1]][(shock_time_vec[1]-1):shock_time_vec[1]]

        # Connect previous and current points
        lines(y = connecting_the_two
              ,x = index(connecting_the_two)
              ,col = 'black'
              ,cex = 1.1
              ,lty = 3
              ,lwd = 3)

        # Plot actual data points
        points(y = Y_not_yet_plotted[1:length(adjusted_pred)]
               ,x = index(Y_not_yet_plotted)[1:length(adjusted_pred)]
               ,col = colors_for_adjusted_pred[3]
               ,cex = 2
               ,pch = 19)
    }

    # Add legend
    labels_for_legend <- c('HAR (unadjusted)', 'Adjusted HAR Prediction', 'Actual')
    legend(x = "topleft"
           ,legend = labels_for_legend
           ,1:length(labels_for_legend)
           ,colors_for_adjusted_pred
           ,title = 'Prediction Method'
           ,cex = 1.3)
}
### END plot_maker_HAR
```

Plotting Function for HAR Model The plot_maker_HAR function is a comprehensive visualization tool for Heterogeneous Autoregressive (HAR) model predictions. It provides:

Multi-panel visualization of:

Donor series time plots Donor pool weights Fixed effects estimates Target series predictions

Key features:

Flexible labeling Optional ground truth display Color-coded predictions Adaptive plotting layout

Function4: SynthPrediction

```
### SynthPrediction Function
### Purpose: Synthetic Prediction for Time Series with Shock Analysis

SynthPrediction <- function(Y_series_list
                            ,covariates_series_list
                            ,shock_time_vec
                            ,shock_length_vec
                            ,k = 1
                            ,dbw_scale = TRUE
                            ,dbw_center = TRUE
```

```
                              ,dbw_indices = NULL
                              ,princ_comp_input = min(length(shock_time_vec), ncol(covariates_series_list
                              ,covariate_indices = NULL
                              ,geometric_sets = NULL
                              ,days_before_shocktime_vec = NULL
                              ,arima_order = NULL
                              ,user_ic_choice = c('aicc','aic','bic')[1]
                              ,plots = TRUE
                              ,display_ground_truth_choice = FALSE
){
  ### Function Overview:
  # This function performs synthetic prediction for time series data,
  # specifically designed to analyze the impact of shocks on a target series.
  # It uses a combination of ARIMA modeling, distance-based weighting,
  # and donor series to estimate and adjust predictions.

  ### Input Parameters:
  # Y_series_list: List of time series, with the first series being the target
  # covariates_series_list: List of covariate time series
  # shock_time_vec: Vector of shock times for each series
  # shock_length_vec: Vector of shock durations for each series
  # k: Forecast horizon (default 1)
  # dbw_scale/center: Scaling and centering options for distance-based weighting
  # dbw_indices: Indices for distance-based weighting
  # covariate_indices: Specific covariates to use in the model
  # arima_order: ARIMA model order (default c(1,1,1))
  # user_ic_choice: Information criterion for model selection
  # plots: Whether to generate visualization plots
  # display_ground_truth_choice: Whether to display ground truth in plots

  ### Preprocessing and Initialization
  # Determine number of donor series
  n <- length(Y_series_list) - 1

  # Set default ARIMA order if not specified
  if (is.null(arima_order) == TRUE) {
    arima_order <- c(1,1,1)
  }

  # Set default distance-based weighting indices
  if (is.null(dbw_indices) == TRUE) {
    dbw_indices <- 1:ncol(covariates_series_list[[1]])
  }

  ### Convert Shock Times to Integer Indices
  # Handle both character (date) and numeric shock time inputs
  integer_shock_time_vec <- c()
  integer_shock_time_vec_for_convex_hull_based_optimization <- c()
  for (i in 1:(n+1)){
    if (is.character(shock_time_vec[i]) == TRUE){
      # Convert date strings to integer indices
      integer_shock_time_vec[i] <- which(index(Y[[i]]) == shock_time_vec[i])
      integer_shock_time_vec_for_convex_hull_based_optimization[i] <- which(index(covariates_series_lis
```

```
  }
  else{
    # Use numeric indices directly
    integer_shock_time_vec[i] <- shock_time_vec[i]
    integer_shock_time_vec_for_convex_hull_based_optimization[i] <- shock_time_vec[i]
  }
}

### Estimate Fixed Effects for Donor Series
# Compute shock effects for each donor series using ARIMA models
omega_star_hat_vec <- c()
order_of_arima <- list()
for (i in 2:(n+1)){
  # Create indicator variable for post-shock periods
  vec_of_zeros <- rep(0, integer_shock_time_vec[i])
  vec_of_ones <- rep(1, shock_length_vec[i])
  post_shock_indicator <- c(vec_of_zeros, vec_of_ones)
  last_shock_point <- integer_shock_time_vec[i] + shock_length_vec[i]

  # Prepare covariates for ARIMA modeling
  if (is.null(covariate_indices) == TRUE) {
    X_i_penultimate <- cbind(Y_series_list[[i]][1:last_shock_point], post_shock_indicator)
    X_i_final <- X_i_penultimate[,2]
  }
  else {
    X_i_subset <- covariates_series_list[[i]][1:last_shock_point,covariate_indices]
    X_i_with_indicator <- cbind(X_i_subset, post_shock_indicator)
    X_i_final <- X_i_with_indicator
  }

  # Fit ARIMA model to donor series
  arima <- forecast::auto.arima(Y_series_list[[i]][1:last_shock_point]
                                ,xreg=X_i_final
                                ,ic = user_ic_choice)

  # Extract fixed effect (shock coefficient)
  order_of_arima[[i]] <- arima$arma
  coef_test <- lmtest::coeftest(arima)
  extracted_fixed_effect <- coef_test[nrow(coef_test),1]
  omega_star_hat_vec <- c(omega_star_hat_vec, extracted_fixed_effect)
}

### Compute Distance-Based Weights
# Use distance-based weighting to combine donor series
dbw_output <- dbw(covariates_series_list
                  ,dbw_indices
                  ,integer_shock_time_vec
                  ,scale = TRUE
                  ,center = TRUE
                  ,sum_to_1 = TRUE
                  ,bounded_below_by = 0
                  ,bounded_above_by = 1
                  ,Y = Y_series_list
```

```
                  ,inputted_transformation = id
)

# Compute weighted average of shock effects
w_hat <- dbw_output[[1]]
omega_star_hat <- as.numeric(w_hat %*% omega_star_hat_vec)

### Forecast Target Series
# Fit ARIMA model to target series and generate predictions
if (is.null(covariate_indices) == TRUE){
  # Simple ARIMA without external regressors
  arima <- forecast::auto.arima(Y_series_list[[1]][1:(integer_shock_time_vec[1])]
                                ,xreg = NULL
                                ,ic = user_ic_choice)
  unadjusted_pred <- predict(arima, n.ahead = shock_length_vec[1])
}
else{
  # ARIMA with lagged covariates
  X_lagged <- lag.xts(covariates_series_list[[1]][1:integer_shock_time_vec[1],covariate_indices])
  arima <- forecast::auto.arima(Y_series_list[[1]][1:integer_shock_time_vec[1]]
                                ,xreg = X_lagged
                                ,ic = user_ic_choice)

  # Prepare forecast with last observed covariate values
  X_to_use_in_forecast <- covariates_series_list[[1]][integer_shock_time_vec[1],covariate_indices]
  X_replicated_for_forecast_length <- matrix(rep(X_to_use_in_forecast, k)
                                             , nrow = shock_length_vec[1]
                                             , byrow = TRUE)
  forecast_period <- (integer_shock_time_vec[1]+1):(integer_shock_time_vec[1]+shock_length_vec[1])
  mat_X_for_forecast <- cbind(Y_series_list[[1]][forecast_period]
                              , X_replicated_for_forecast_length)
  unadjusted_pred <- predict(arima
                             , n.ahead = shock_length_vec[1]
                             , newxreg = mat_X_for_forecast[,-1])
}

### Adjust Prediction with Estimated Shock Effect
adjusted_pred <- unadjusted_pred$pred + omega_star_hat

### Prepare Output
list_of_linear_combinations <- list(w_hat)
list_of_forecasts <- list(unadjusted_pred, adjusted_pred)
names(list_of_forecasts) <- c('unadjusted_pred', 'adjusted_pred')
output_list <- list(list_of_linear_combinations, list_of_forecasts)
names(output_list) <- c('linear_combinations', 'predictions')

### Verbose Output
cat('SynthPrediction Details','\n',
    '-------------------------------------------------------------\n',
    'Donors:', n, '\n',
    'Shock times:', shock_time_vec, '\n',
    'Lengths of shock times:', shock_length_vec, '\n',
    'Optimization Success:', dbw_output[[2]], '\n', '\n',
```

```
      'Convex combination',w_hat,'\n',
      'Shock estimates provided by donors:', omega_star_hat_vec, '\n',
      'Aggregate estimated shock effect:', omega_star_hat, '\n',
      'Actual change in stock price at T* + 1:', Y_series_list[[1]][integer_shock_time_vec[1]+1],'\n',
      'Unadjusted forecasted change in stock price at T*+1:', unadjusted_pred$pred,'\n',
      'MSE unadjusted:', (as.numeric(Y_series_list[[1]][integer_shock_time_vec[1]+1])-unadjusted_pred$p
      'Adjusted forecasted change in stock price at T*+1:', adjusted_pred,'\n',
      'MSE adjusted:', (as.numeric(Y_series_list[[1]][integer_shock_time_vec[1]+1])-adjusted_pred)**2,'
  )

  ### Optional Plotting
  if (plots == TRUE){
    cat('User has opted to produce plots.','\n')
    plot_maker_synthprediction(Y_series_list
                               ,shock_time_vec
                               ,integer_shock_time_vec
                               ,shock_length_vec
                               ,unadjusted_pred$pred
                               ,w_hat
                               ,omega_star_hat
                               ,omega_star_hat_vec
                               ,adjusted_pred
                               ,display_ground_truth = display_ground_truth_choice
    )
  }

  return(output_list)
}
```

SynthPrediction Function SynthPrediction Function Summary The SynthPrediction function is a sophisticated time series forecasting method designed to estimate the impact of shocks on a target time series by leveraging multiple donor series. Key features include:

Core Methodology:

Uses ARIMA (Autoregressive Integrated Moving Average) modeling Applies distance-based weighting to combine donor series Estimates shock effects across multiple time series

Main Steps:

Preprocess input time series and shock times Estimate shock effects for donor series Compute weights for donor series using distance-based method Generate unadjusted and shock-adjusted predictions Provide detailed output and optional visualization

Key Inputs:

Target time series Donor time series Shock times and durations Optional covariate series

Key Outputs:

Unadjusted and adjusted predictions Linear combination weights Detailed statistical information Optional visualization of results

Advanced Features:

Handles both numeric and date-based time indices Supports flexible ARIMA model selection Provides multiple information criteria for model selection Generates comprehensive diagnostic output

(from here some coding is not done yet)

```
### SynthVolForecast Function
### Purpose: Advanced Volatility Forecasting with Synthetic Weighting
SynthVolForecast <- function(
    Y_series_list,              # List of time series (target first)
    covariates_series_list,     # List of covariate time series
    shock_time_vec,             # Vector of shock times for each series
    shock_length_vec,           # Vector of shock durations
    k = 1,                      # Forecast horizon
    dbw_scale = TRUE,           # Scale distance-based weights
    dbw_center = TRUE,          # Center distance-based weights
    dbw_indices = NULL,         # Indices for distance-based weighting
    dbw_Y_lookback = c(0),      # Lookback period for target series
    dbw_princ_comp_input = NULL, # Principal component input for weighting
    covariate_indices = NULL,   # Specific covariate indices to use
    geometric_sets = NULL,      # Placeholder for geometric sets
    days_before_shocktime_vec = NULL, # Days before shock time
    garch_order = NULL,         # GARCH model order
    common_series_assumption = FALSE, # Assume common series effect
    plots = TRUE,               # Generate plots
    shock_time_labels = NULL,   # Labels for shock times
    ground_truth_vec = NULL,    # Ground truth values
    Y_lookback_indices_input = list(seq(1,30,1)), # Lookback indices for target
    X_lookback_indices_input = rep(list(c(1)),length(dbw_indices)) # Lookback indices for covariates
){
    ### 1. Initialize and Set Default Parameters
    # Calculate number of donor series
    n <- length(Y_series_list) - 1

    # Set default GARCH model order if not provided
    if (is.null(garch_order) == TRUE) {
        garch_order <- list(length = n+1)
        for (i in 1:(n+1)){
            garch_order[[i]] <- c(1,1,0)  # Default: GARCH(1,1,0)
        }
    }

    # Set default distance-based weight indices
    if (is.null(dbw_indices) == TRUE) {
        dbw_indices <- 1:ncol(covariates_series_list[[1]])
    }

    ### 2. Convert Shock Times to Integer Indices
    integer_shock_time_vec <- c()
    integer_shock_time_vec_for_convex_hull_based_optimization <- c()

    for (i in 1:(n+1)){
        # Handle both character (date) and numeric shock times
        if (is.character(shock_time_vec[i]) == TRUE){
            integer_shock_time_vec[i] <- which(index(Y[[i]]) == shock_time_vec[i])
            integer_shock_time_vec_for_convex_hull_based_optimization[i] <-
                which(index(covariates_series_list[[i]]) == shock_time_vec[i])
```

```
        }
        else{
            integer_shock_time_vec[i] <- shock_time_vec[i]
            integer_shock_time_vec_for_convex_hull_based_optimization[i] <- shock_time_vec[i]
        }
    }

    ### 3. Calculate Distance-Based Weights
    dbw_output <- dbw(
        covariates_series_list,
        dbw_indices,
        integer_shock_time_vec_for_convex_hull_based_optimization,
        scale = dbw_scale,
        center = dbw_center,
        sum_to_1 = TRUE,
        princ_comp_count = dbw_princ_comp_input,
        bounded_below_by = 0,
        bounded_above_by = 1,
        Y = Y_series_list,
        Y_lookback_indices = Y_lookback_indices_input,
        X_lookback_indices = X_lookback_indices_input,
        inputted_transformation = mean_square_y
    )
    w_hat <- dbw_output[[1]]

    ### 4. Estimate Fixed Effects for Donor Series
    omega_star_hat_vec <- c()
    omega_star_std_err_hat_vec <- c()

    # Handle common series assumption (currently a TODO)
    if (common_series_assumption == TRUE){
        print('Common series assumption implementation pending')
    }
    else{
        # Iterate through donor series
        for (i in 2:(n+1)){
            # Create shock indicator variable
            vec_of_zeros <- rep(0, integer_shock_time_vec[i])
            vec_of_ones <- rep(1, shock_length_vec[i])
            vec_of_final_zeros <- rep(0, 20)
            post_shock_indicator <- c(vec_of_zeros, vec_of_ones, vec_of_final_zeros)
            last_shock_point <- integer_shock_time_vec[i] + shock_length_vec[i] + 20

            # Prepare covariates
            if (is.null(covariate_indices) == TRUE) {
                X_i_penultimate <- cbind(
                    Y_series_list[[i]][1:last_shock_point],
                    post_shock_indicator
                )
                X_i_final <- X_i_penultimate[,2]
            }
            else {
                X_i_subset <- covariates_series_list[[i]][1:last_shock_point,covariate_indices]
```

```
            X_i_with_indicator <- cbind(X_i_subset, post_shock_indicator)
            X_i_final <- X_i_with_indicator
        }

        # Fit GARCH model to donor series
        fitted_garch <- garchx::garchx(
            Y_series_list[[i]][1:last_shock_point],
            order = garch_order[[i]],
            xreg = X_i_final,
            backcast.values = NULL,
            control = list(
                eval.max = 100000,
                iter.max = 1500000,
                rel.tol = 1e-8
            )
        )

        # Extract fixed effects
        coef_test <- lmtest::coeftest(fitted_garch)
        extracted_fixed_effect <- coef_test[dim(coef_test)[1], 1]
        extracted_fixed_effect_std_err <- coef_test[dim(coef_test)[1], 2]

        # Store results
        omega_star_hat_vec <- c(omega_star_hat_vec, extracted_fixed_effect)
        omega_star_std_err_hat_vec <- c(omega_star_std_err_hat_vec, extracted_fixed_effect_std_err)
    }
}

### 5. Compute Linear Combination of Fixed Effects
omega_star_hat <- w_hat %*% omega_star_hat_vec

### 6. Fit GARCH to Target Series
if (is.null(covariate_indices) == TRUE){
    # GARCH without external regressors
    fitted_garch <- garchx::garchx(
        Y_series_list[[1]][1:(integer_shock_time_vec[1])],
        order = garch_order[[1]],
        xreg = NULL,
        backcast.values = NULL,
        control = list(
            eval.max = 100000,
            iter.max = 1500000,
            rel.tol = 1e-8
        )
    )

    # Generate prediction
    unadjusted_pred <- predict(fitted_garch, n.ahead = shock_length_vec[1])
}
else{
    # GARCH with external regressors
    fitted_garch <- garchx::garchx(
        Y_series_list[[1]][1:(integer_shock_time_vec[1])],
```

```
            order = garch_order,
            xreg = covariates_series_list[[1]][1:(integer_shock_time_vec[1]),covariate_indices],
            backcast.values = NULL,
            control = list(
                eval.max = 100000,
                iter.max = 1500000,
                rel.tol = 1e-8
            )
        )

        # Prepare forecast with last observed covariate values
        X_to_use_in_forecast <- covariates_series_list[[1]][integer_shock_time_vec[1],covariate_indices]
        X_replicated_for_forecast_length <- matrix(
            rep(X_to_use_in_forecast, k),
            nrow = shock_length_vec[1],
            byrow = TRUE
        )

        forecast_period <- (integer_shock_time_vec[1]):(integer_shock_time_vec[1]+shock_length_vec[1])
        mat_X_for_forecast <- cbind(
            Y_series_list[[1]][forecast_period],
            X_replicated_for_forecast_length
        )

        # Generate prediction
        unadjusted_pred <- predict(
            fitted_garch,
            n.ahead = shock_length_vec[1],
            newxreg = mat_X_for_forecast[,-1]
        )
    }

    ### 7. Adjust Predictions
    adjusted_pred <- unadjusted_pred + rep(omega_star_hat, k)
    arithmetic_mean_based_pred <- rep(mean(omega_star_hat_vec), k) + unadjusted_pred

    ### 8. Calculate Losses
    if (is.null(ground_truth_vec) == TRUE){
        QL_loss_unadjusted_pred <- NA
        QL_loss_adjusted_pred <- NA
        QL_loss_arithmetic_mean_based_pred <- NA
    }
    else {
        QL_loss_unadjusted_pred <- sum(QL_loss_function(unadjusted_pred, ground_truth_vec))
        QL_loss_adjusted_pred <- sum(QL_loss_function(adjusted_pred, ground_truth_vec))
        QL_loss_arithmetic_mean_based_pred <- sum(QL_loss_function(arithmetic_mean_based_pred, ground_t
    }

    ### 9. Prepare Output
    list_of_linear_combinations <- list(w_hat)
    list_of_forecasts <- list(
        unadjusted_pred,
        adjusted_pred,
```

```
    arithmetic_mean_based_pred
)
list_of_losses <- list(
    QL_loss_unadjusted_pred,
    QL_loss_adjusted_pred,
    QL_loss_arithmetic_mean_based_pred
)

names(list_of_forecasts) <- c(
    'unadjusted_pred',
    'adjusted_pred',
    'arithmetic_mean_based_pred'
)
names(list_of_losses) <- c(
    'unadjusted_pred',
    'adjusted_pred',
    'arithmetic_mean_based_pred'
)

output_list <- list(
    list_of_linear_combinations,
    list_of_forecasts,
    list_of_losses
)
names(output_list) <- c(
    'linear_combinations',
    'predictions',
    'loss'
)

### 10. Display Results
cat('-----------------------------------------------------------------\n',
    '-------------------SynthVolForecast Results--------------------','\n',
    '-----------------------------------------------------------------\n',
    'Donors:', n, '\n',  '\n',
    'Shock times:', shock_time_vec, '\n', '\n',
    'Lengths of shock times:', shock_length_vec, '\n', '\n',
    'Optimization Success:', dbw_output[[2]], '\n', '\n',
    'Convex combination:',w_hat,'\n', '\n',
    'Shock estimates:', omega_star_hat_vec, '\n', '\n',
    'Aggregate estimated shock effect:', omega_star_hat, '\n', '\n',
    'Unadjusted Forecast:', unadjusted_pred,'\n', '\n',
    'Adjusted Forecast:', adjusted_pred,'\n', '\n',
    'Arithmetic-Mean-Based Forecast:',arithmetic_mean_based_pred,'\n','\n',
    'Ground Truth (estimated by realized volatility):', ground_truth_vec,'\n', '\n',
    'QL Loss of unadjusted:', QL_loss_unadjusted_pred,'\n', '\n',
    'QL Loss of adjusted:', QL_loss_adjusted_pred,'\n', '\n'
)

### 11. Optional Plotting
if (plots == TRUE){
    cat('\n User has opted to produce plots.','\n')
    plot_maker_garch(
```

```
            fitted(fitted_garch),
            shock_time_labels,
            integer_shock_time_vec,
            shock_length_vec,
            unadjusted_pred,
            w_hat,
            omega_star_hat,
            omega_star_hat_vec,
            adjusted_pred,
            arithmetic_mean_based_pred,
            ground_truth_vec
        )
    }

    return(output_list)
} ### END SynthVolForecast
```

Comprehensive Summary of SynthVolForecast Function Core Purpose A sophisticated time series volatility forecasting method that:

Combines multiple time series Handles shock events dynamically Provides flexible prediction strategies

Key Methodology

Distance-Based Weighting

Compute weights for donor series Integrate multiple time series information Adaptive to series characteristics

GARCH Modeling

Handle conditional heteroskedasticity Capture complex volatility dynamics Support external regressors

Shock Effect Estimation

Quantify impact of shock events Extract fixed effects from donor series Provide multiple adjustment strategies

Unique Features

Handles both numeric and date-based inputs Supports series with/without covariates Multiple prediction methods:

Unadjusted prediction Weighted adjusted prediction Arithmetic mean adjusted prediction

Flexible Configuration

Customizable GARCH model orders Optional principal component analysis Configurable distance-based weighting

Output Components

Linear combination weights Predictions Loss calculations Optional visualization

report for SynthVolForecast Function:

The SynthVolForecast function is currently an incomplete research-stage prototype with multiple key unimplemented components, particularly the common_series_assumption section, which is entirely a placeholder. The code is riddled with #tk and #mk annotation markers, indicating an immature statistical modeling method. While the function demonstrates an innovative approach to time series volatility forecasting, it lacks comprehensive algorithm implementation, model validation processes, and parameter selection mechanisms. It currently resembles more of a conceptual modeling draft that requires further development and refinement before it can be practically applied to time series analysis.

```r
# HAR (Heterogeneous Autoregressive) function for analyzing time series data with shock events
# This function combines linear regression with difference-based weighting (dbw) optimization
HAR <- function(Y
                ,covariates_series_list
                ,shock_time_vec
                ,shock_length_vec
                ,k=1 # Parameter for k-step ahead forecasting (implementation uncertain)
                ,dbw_scale = TRUE
                ,dbw_center = TRUE
                ,dbw_indices = 1:ncol(covariates_series_list[[1]])
                ,dbw_Y_lookback = c(0)
                ,dbw_princ_comp_input = NULL
                ,covariate_indices = NULL
                ,geometric_sets = NULL # Marked as "to come" (tk)
                ,plots = TRUE
                ,shock_time_labels = NULL
                ,ground_truth_vec = NULL
                ,Y_lookback_indices_input = list(seq(1,3,1))
                ,X_lookback_indices_input = rep(list(c(1)),length(dbw_indices))
){
  # Documentation string (incomplete)

  # Variable initialization section - some issues here
  n <- length(X) - 1 # Error: X is undefined, should likely be Y_series_list
  integer_shock_time_vec <- c() # Marked for review (mk)
  integer_shock_time_vec_for_convex_hull_based_optimization <- c() # Marked for review (mk)
  omega_star_hat_vec <- c() # Vector to store shock effect estimates

  # Branch 1: Processing when Y is a dataframe
  if (is.data.frame(Y) == TRUE){
    print('Y is a dataframe')

    # Convert shock times to integer indices if they're provided as character dates
    for (i in 1:length(shock_time_vec)){
      # Code to process shock times...
    }

    # Process dates and create donor variable
    # Several undefined variables: shock_dates, RVSPY_final
    shock_dates_as_dates <- as.Date(as.Date(unlist(shock_dates)))
    shock_dates_as_dates_without_TSUS <- shock_dates_as_dates[-1]
    Y_with_donor_col <- data.frame(Y %>% mutate(donor = ifelse(row.names(RVSPY_final) %in% shock_dates_a

    # Print rows with indicator value of 1
    print('Here are the rows with 1 in the indicator variable:')
    print(Y_with_donor_col[Y_with_donor_col$donor == 1,])

    # Further data preparation for modeling
    Y_with_donor_col[Y_with_donor_col$donor == 1, "donor"] <- shock_dates_as_dates_without_TSUS
    Y_with_donor_col$donor = as.factor(Y_with_donor_col$donor)

    # Create training and forecast datasets
    training_period <- Y_with_donor_col[row.names(Y_with_donor_col) <= as.Date(shock_dates_as_dates[1])
```

```
  # Build linear model
  m1 <- lm(training_period[,1] ~. , data = training_period[,-c(1)])
  cat('We inspect the fitted linear model.')
  print(summary(m1))

  # Prepare test data and make predictions
  forecast_period <- Y_with_donor_col[row.names(Y_with_donor_col) == as.Date(shock_dates_as_dates[1])
  outcome <- forecast_period[,1]
  forecast_period_w_outcome_dropped <- forecast_period[,-c(1)]
  forecast_period_w_outcome_dropped$donor <- as.factor(forecast_period_w_outcome_dropped$donor)
  unadjusted_pred <- predict(m1, newdata = forecast_period_w_outcome_dropped)

  # Extract shock coefficients
  no_events <- length(shock_dates_as_dates) - 1
  no_coef <- length(coef(m1))
  omega_star_hat_vec <- c(omega_star_hat_vec, coef(m1)[(no_coef-no_events+1):no_coef])
}

# Branch 2: Processing when Y is not a dataframe
else{
  print('Y is not a dataframe.')

  # Convert shock times to indices
  for (i in 1:length(shock_time_vec)){
    # Similar code as in the if branch...
  }

  # Process each shock event
  for (i in 2:(n+1)){
    # Create indicator variables for post-shock periods
    vec_of_zeros <- rep(0, integer_shock_time_vec[i])
    vec_of_ones <- rep(1, shock_length_vec[i])
    post_shock_indicator <- c(vec_of_zeros, vec_of_ones)
    last_shock_point <- integer_shock_time_vec[i] + shock_length_vec[i]

    # Subset data based on whether covariate indices are provided
    if (is.null(covariate_indices) == TRUE) {
      # Code for no specific covariates...
    }
    else {
      # Code for specific covariates...
    }

    # Print data and update vector of shock effects
    print('We print the tail of the covariate df we use in the model:')
    print(tail(X_i_final))

    # Error: HAR_lm is undefined
    omega_star_hat_vec <- c(omega_star_hat_vec, HAR_lm)
  }
}

# Print shock time information
```

```r
print('We print the integers of the shock times.')
print(integer_shock_time_vec)
print('We print the booleans for the shock times.')
print(integer_shock_time_vec_for_convex_hull_based_optimization)

# Prepare data for difference-based weighting
print('Here is the Y data we will use in dbw:')
make_xts <- xts(Y[,1], order.by = as.Date(row.names(Y)))
colnames(make_xts) <- c('User_chosen_Y')
Y_list <- rep(list(make_xts), n+1)

# Calculate optimal weights using dbw function (undefined)
dbw_output <- dbw(covariates_series_list,
                  # Additional parameters for dbw function...
                  inputted_transformation = id # Error: id is undefined
)

# Extract weights and compute weighted shock effect
w_hat <- dbw_output[[1]]
omega_star_hat <- w_hat %*% omega_star_hat_vec

# Calculate different predictions
arithmetic_mean_based_pred <- mean(omega_star_hat_vec) + unadjusted_pred
adjusted_pred <- unadjusted_pred + omega_star_hat

# Calculate loss metrics using undefined QL_loss_function
if (min(Y[[1]]) < 0){
  # Code for exponential transformation...
}
else{
  # Code for direct calculation...
}

# Organize results into lists
list_of_linear_combinations <- list(w_hat)
list_of_forecasts <- list(unadjusted_pred, adjusted_pred, arithmetic_mean_based_pred)
list_of_losses <- list(QL_loss_unadjusted, QL_loss_adjusted, QL_loss_arithmetic_mean)

# Name list elements
names(list_of_forecasts) <- c('unadjusted_pred', 'adjusted_pred', 'arithmetic_mean_based_pred')
output_list <- list(list_of_linear_combinations, list_of_forecasts, list_of_losses)
names(output_list) <- c('linear_combinations', 'predictions', 'loss')

# Print detailed results
cat('------------------------------------------------------------------\n',
    '-------------------HAR Results--------------------','\n',
    # Additional output information...
)

# Generate plots if requested
if (plots == TRUE){
  cat('\n User has opted to produce plots.','\n')
  # Call to undefined plot_maker_HAR function
```

```
    plot_maker_HAR(Y_list, ...)
  }

  # Return output list
  return(output_list)
} # END HAR
```

report for HAR Function:

The HAR (Heterogeneous Autoregressive) function analyzed here represents an incomplete implementation of a time series analysis tool designed to evaluate shock effects. The code contains numerous critical issues including undefined variables (X, shock_dates, RVSPY_final), missing functions (dbw, HAR_lm, QL_loss_function), logical inconsistencies in data handling, and incomplete implementation of core features. Multiple sections are marked with "tk" (to come/to be fixed) annotations, clearly indicating the developer's awareness that substantial portions remain unfinished. In its current state, this function cannot run successfully and would require significant development work to become operational. The framework suggests an innovative approach combining linear regression with optimized weighting for financial time series analysis, but remains merely a conceptual outline rather than a functional implementation.

summary for this package up to now:

The package is currently in a research stage, with several functions and components marked as "to come" (tk) or "to be fixed" (mk). The core functions, SynthPrediction and SynthVolForecast, are designed for advanced time series forecasting, particularly in the context of financial data. However, they are incomplete and contain numerous undefined variables and functions, making them non-functional in their current state. The HAR function also exhibits similar issues, with critical components missing or poorly defined. Overall, the package requires significant development and refinement before it can be effectively utilized for practical applications in time series analysis.

This package is still in the early research and development stage. Many key functions and components are either not fully implemented (marked with "tk" or "TODO") or are in preliminary draft form. While the core functions—**SynthPrediction**, **SynthVolForecast**, and the **HAR** framework—aim to provide advanced time series forecasting under shock conditions using synthetic control principles, several issues limit their immediate applicability. Below are the main points:

1. Incomplete Testing & Data Handling

- **No End-to-End Pipeline:**
  The package lacks a complete data processing pipeline that unifies the different data formats needed for each model (e.g., time series objects for ARIMA/GARCH, regression-style data for HAR).

- **Varying Data Requirements:**
  Each model may require different types of inputs. For example, some models are designed to work with time series data (e.g., using `xts` objects) while others require regression matrices. A unified approach to data handling and transformation has not yet been established.
- **Lack of Comprehensive Testing:**
  The functions have not been fully tested on real or simulated datasets, meaning many aspects of the package remain unverified.

2. Partial Implementation

- **Optional Parameters Not Fully Realized:**
  Parameters such as different distance metrics (`normchoice`), penalty methods (`penalty_normchoice`, `penalty_lambda`), and other optional settings are either commented out or marked as future work.
- **Incomplete Branches:**
  Some branches of the code (e.g., those handling the `common_series_assumption`) are placeholders ("tk TODO") and have not been fully developed.
- **Prototype Stage:**
  The overall framework is in a conceptual phase; further refinement is necessary to ensure robust functionality.

3. Inconsistent or Missing Definitions

- **Placeholders and Debugging Marks:**
  The code contains several "tk" and "TODO" markers that indicate parts of the functions still require further development.
- **Diverse Data Processing Needs:**
  Due to the different models (ARIMA, GARCH, HAR) requiring distinct input types, the overall data processing function is not yet fully integrated or standardized.

Overall Summary

In summary, the **Postshock** package aims to combine synthetic control approaches with time series forecasting to adjust for shocks. It does so by:

- **Computing donor weights** using a distance-based weighting function (`dbw()`),
- **Estimating donor fixed effects** through individual ARIMA/GARCH model fittings,
- **Aggregating these effects** into an overall adjustment to the target series forecast, and
- **Providing visualization tools** to compare unadjusted and adjusted forecasts.

However, the package currently only has preliminary function frameworks and incomplete data processing pipelines. Major components remain untested and some optional features (e.g., handling different data types, applying HAR) are not yet fully integrated. Extensive development and thorough testing are needed before it can be effectively deployed in practice.