ArYma Labs
Decoding the past, Encoding the future

ROBYN UNDER THE HOOD – Ridge Regression in depth

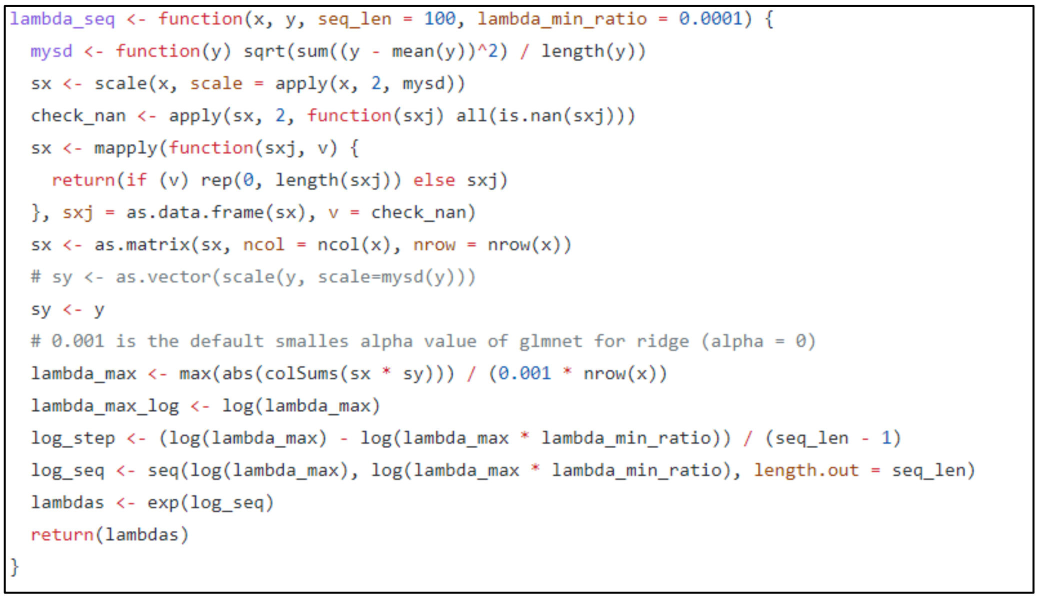
Recap:

We have been deep diving into how Robyn implements MMM. In the previous post, we had covered about how Ridge regularization is used in MMM. (ICYMI: https://www.linkedin.com/posts/ridhima-kumar7_robyn-under-the-hood-a-deep-dive-into-how-activity-6955519210609393664-QI-n?utm_source=linkedin_share&utm_medium=member_desktop_web)

As next step, we wanted to deep dive into how ridge regression is applied in the Robyn code.

1. Let's first understand how lambda is computed in the Robyn code.

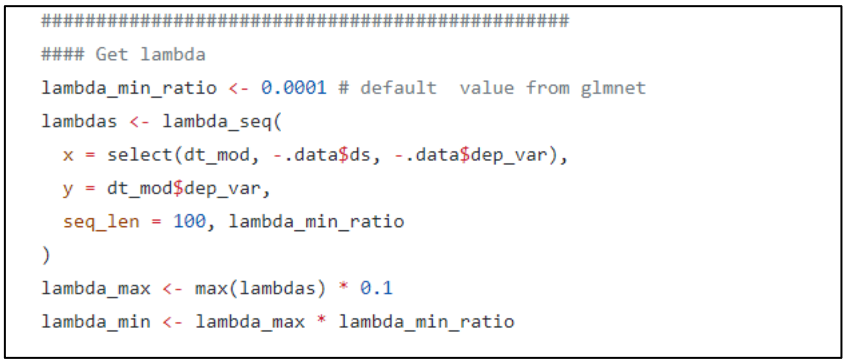
2. Firstly, the minimum and maximum values of Lambdas are computed using the "lambda_seq" function. The lambda_seq function scales all the x variables. Refer to the "lambda_seq" code below(Code Snippet 1)



```r
lambda_seq <- function(x, y, seq_len = 100, lambda_min_ratio = 0.0001) {
  mysd <- function(y) sqrt(sum((y - mean(y))^2) / length(y))
  sx <- scale(x, scale = apply(x, 2, mysd))
  check_nan <- apply(sx, 2, function(sxj) all(is.nan(sxj)))
  sx <- mapply(function(sxj, v) {
    return(if (v) rep(0, length(sxj)) else sxj)
  }, sxj = as.data.frame(sx), v = check_nan)
  sx <- as.matrix(sx, ncol = ncol(x), nrow = nrow(x))
  # sy <- as.vector(scale(y, scale=mysd(y)))
  sy <- y
  # 0.001 is the default smalles alpha value of glmnet for ridge (alpha = 0)
  lambda_max <- max(abs(colSums(sx * sy))) / (0.001 * nrow(x))
  lambda_max_log <- log(lambda_max)
  log_step <- (log(lambda_max) - log(lambda_max * lambda_min_ratio)) / (seq_len - 1)
  log_seq <- seq(log(lambda_max), log(lambda_max * lambda_min_ratio), length.out = seq_len)
  lambdas <- exp(log_seq)
  return(lambdas)
}
```

Code Snippet 1

3. The maximum and minimum values of lambdas are then computed.

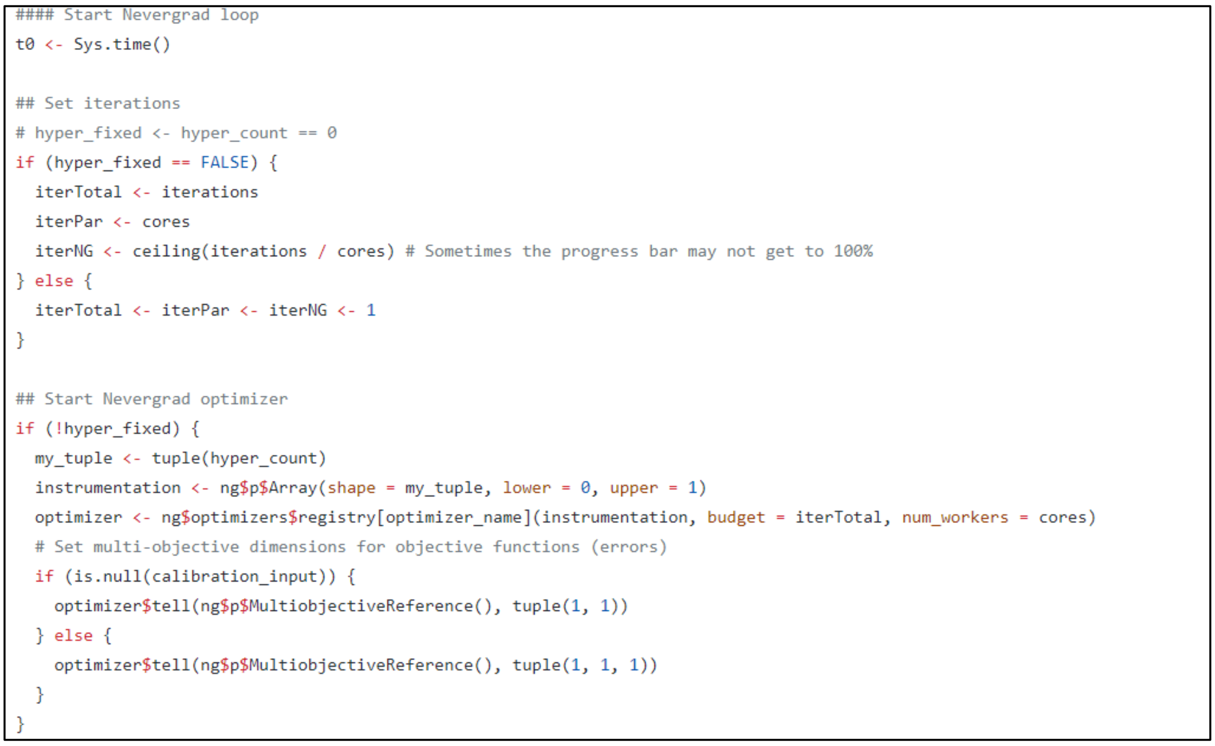


```r
##################################################
#### Get lambda
lambda_min_ratio <- 0.0001 # default  value from glmnet
lambdas <- lambda_seq(
  x = select(dt_mod, -.data$ds, -.data$dep_var),
  y = dt_mod$dep_var,
  seq_len = 100, lambda_min_ratio
)
lambda_max <- max(lambdas) * 0.1
lambda_min <- lambda_max * lambda_min_ratio
```

Code Snippet 2

4. Once the lambda min and lambda max values are computed, the nevergrad optimizer is run and it generates all the hyperparameters values. So, if geometric adstock is performed, the hyperparameters which need to be tuned are theta, alpha, gamma and lambda (lambda for ridge regression).

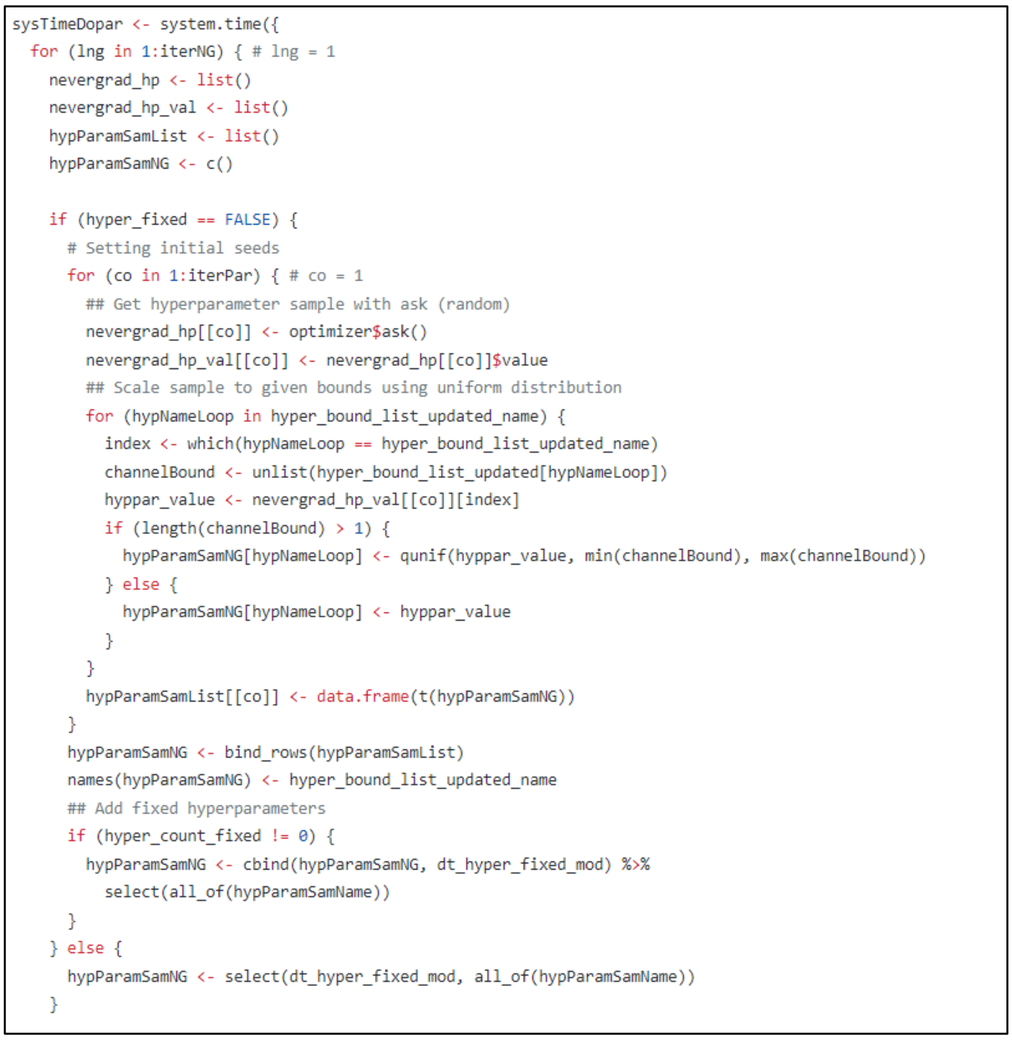
5. In the case of simulated dataset (dt_simulated_weekly) provide by Meta, there are 6 media variables: tv_S, ooh_S, print_S, facecbook_S, search_S and newspaper. Each of these variables will have 3 hyperparameters each (theta, alpha and gamma). So, total 19 hyperparameters including lambda have to be optimized. The lower and upper range of hyperparameters theta, alpha and gamma have to be prespecified before the hyperparameters are tuned. The below code snippet (Code Snippet 3) shows where nervergrad loop begins.



```
#### Start Nevergrad loop
t0 <- Sys.time()

## Set iterations
# hyper_fixed <- hyper_count == 0
if (hyper_fixed == FALSE) {
  iterTotal <- iterations
  iterPar <- cores
  iterNG <- ceiling(iterations / cores) # Sometimes the progress bar may not get to 100%
} else {
  iterTotal <- iterPar <- iterNG <- 1
}

## Start Nevergrad optimizer
if (!hyper_fixed) {
  my_tuple <- tuple(hyper_count)
  instrumentation <- ng$p$Array(shape = my_tuple, lower = 0, upper = 1)
  optimizer <- ng$optimizers$registry[optimizer_name](instrumentation, budget = iterTotal, num_workers = cores)
  # Set multi-objective dimensions for objective functions (errors)
  if (is.null(calibration_input)) {
    optimizer$tell(ng$p$MultiobjectiveReference(), tuple(1, 1))
  } else {
    optimizer$tell(ng$p$MultiobjectiveReference(), tuple(1, 1, 1))
  }
}
```

Code Snippet 3

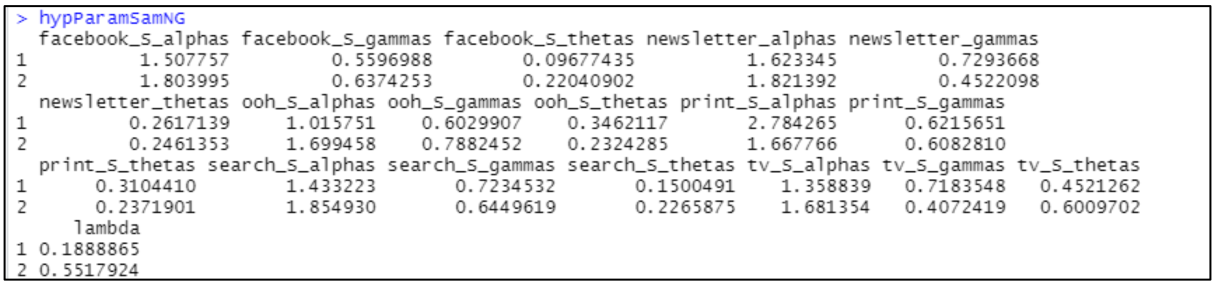
6. The below code snippet shows how hyper parameters are generated including lambda.



```
sysTimeDopar <- system.time({
  for (lng in 1:iterNG) { # lng = 1
    nevergrad_hp <- list()
    nevergrad_hp_val <- list()
    hypParamSamList <- list()
    hypParamSamNG <- c()

    if (hyper_fixed == FALSE) {
      # Setting initial seeds
      for (co in 1:iterPar) { # co = 1
        ## Get hyperparameter sample with ask (random)
        nevergrad_hp[[co]] <- optimizer$ask()
        nevergrad_hp_val[[co]] <- nevergrad_hp[[co]]$value
        ## Scale sample to given bounds using uniform distribution
        for (hypNameLoop in hyper_bound_list_updated_name) {
          index <- which(hypNameLoop == hyper_bound_list_updated_name)
          channelBound <- unlist(hyper_bound_list_updated[hypNameLoop])
          hyppar_value <- nevergrad_hp_val[[co]][index]
          if (length(channelBound) > 1) {
            hypParamSamNG[hypNameLoop] <- qunif(hyppar_value, min(channelBound), max(channelBound))
          } else {
            hypParamSamNG[hypNameLoop] <- hyppar_value
          }
        }
        hypParamSamList[[co]] <- data.frame(t(hypParamSamNG))
      }
      hypParamSamNG <- bind_rows(hypParamSamList)
      names(hypParamSamNG) <- hyper_bound_list_updated_name
      ## Add fixed hyperparameters
      if (hyper_count_fixed != 0) {
        hypParamSamNG <- cbind(hypParamSamNG, dt_hyper_fixed_mod) %>%
          select(all_of(hypParamSamName))
      }
    } else {
      hypParamSamNG <- select(dt_hyper_fixed_mod, all_of(hypParamSamName))
    }
```

Code Snippet 4

7. An object   - hypParamSamNG is created which has hyper parameters generated by Nevergrad. The below sample output has two sets of values as two iterations are performed.

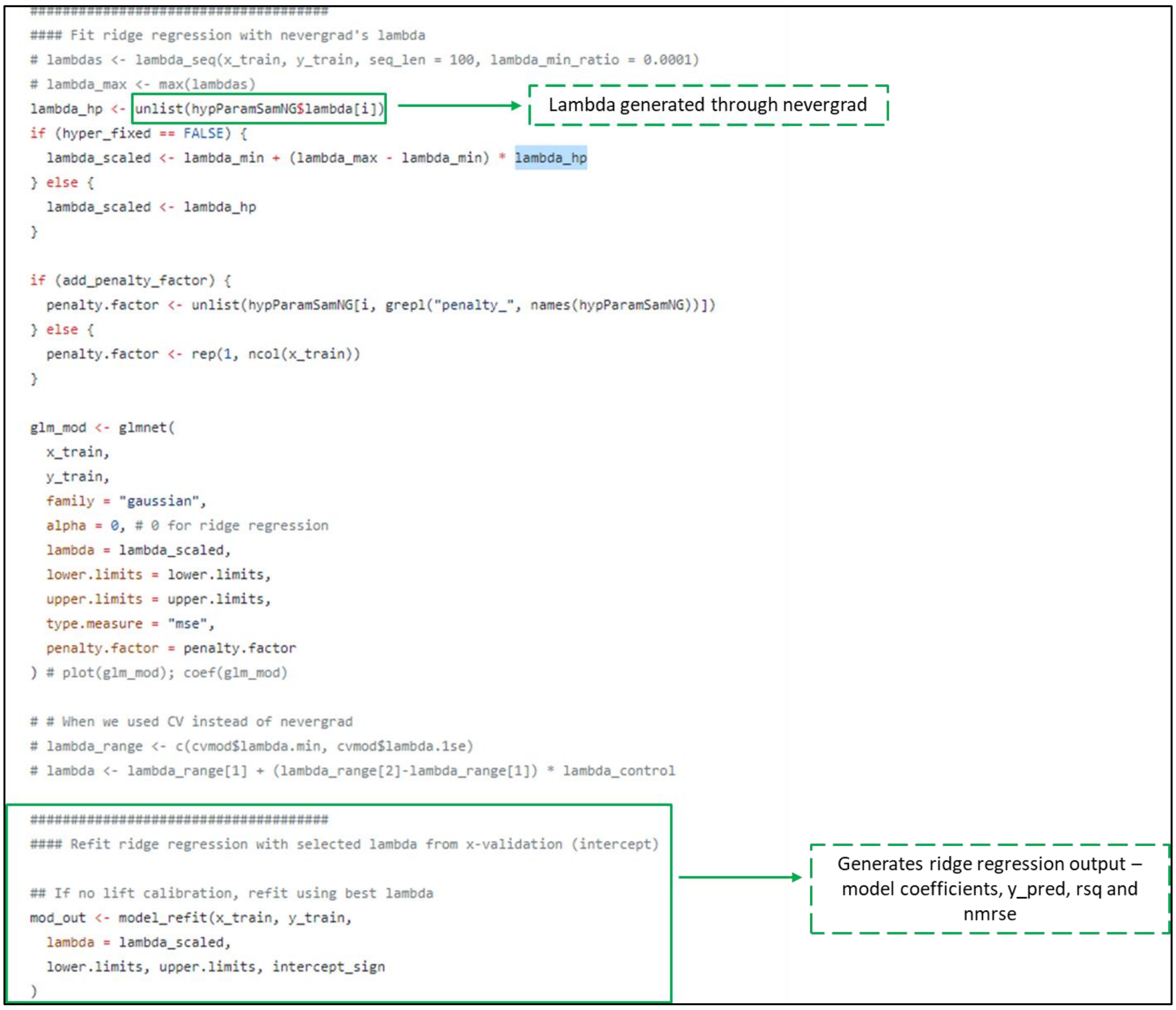


```
> hypParamSamNG
  facebook_S_alphas facebook_S_gammas facebook_S_thetas newsletter_alphas newsletter_gammas
1          1.507757         0.5596988        0.09677435          1.623345         0.7293668
2          1.803995         0.6374253        0.22040902          1.821392         0.4522098
  newsletter_thetas ooh_S_alphas ooh_S_gammas ooh_S_thetas print_S_alphas print_S_gammas
1         0.2617139     1.015751    0.6029907    0.3462117       2.784265      0.6215651
2         0.2461353     1.699458    0.7882452    0.2324285       1.667766      0.6082810
  print_S_thetas search_S_alphas search_S_gammas search_S_thetas tv_S_alphas tv_S_gammas tv_S_thetas
1      0.3104410        1.433223       0.7234532       0.1500491    1.358839   0.7183548   0.4521262
2      0.2371901        1.854930       0.6449619       0.2265875    1.681354   0.4072419   0.6009702
     lambda
1 0.1888865
2 0.5517924
```

Code Snippet 5

8. The media variables are then transformed using geometric adstock and hill function.

9. Once all the transformations are completed and the signs of the variables are specified, the nevergrad loop performs ridge regression.

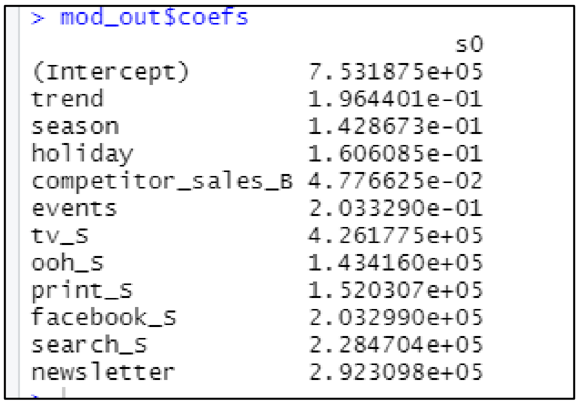


```r
######################################
#### Fit ridge regression with nevergrad's lambda
# lambdas <- lambda_seq(x_train, y_train, seq_len = 100, lambda_min_ratio = 0.0001)
# lambda_max <- max(lambdas)
lambda_hp <- unlist(hypParamSamNG$lambda[i])
if (hyper_fixed == FALSE) {
  lambda_scaled <- lambda_min + (lambda_max - lambda_min) * lambda_hp
} else {
  lambda_scaled <- lambda_hp
}

if (add_penalty_factor) {
  penalty.factor <- unlist(hypParamSamNG[i, grepl("penalty_", names(hypParamSamNG))])
} else {
  penalty.factor <- rep(1, ncol(x_train))
}

glm_mod <- glmnet(
  x_train,
  y_train,
  family = "gaussian",
  alpha = 0, # 0 for ridge regression
  lambda = lambda_scaled,
  lower.limits = lower.limits,
  upper.limits = upper.limits,
  type.measure = "mse",
  penalty.factor = penalty.factor
) # plot(glm_mod); coef(glm_mod)

# # When we used CV instead of nevergrad
# lambda_range <- c(cvmod$lambda.min, cvmod$lambda.1se)
# lambda <- lambda_range[1] + (lambda_range[2]-lambda_range[1]) * lambda_control

######################################
#### Refit ridge regression with selected lambda from x-validation (intercept)

## If no lift calibration, refit using best lambda
mod_out <- model_refit(x_train, y_train,
  lambda = lambda_scaled,
  lower.limits, upper.limits, intercept_sign
)
```

Lambda generated through nevergrad
Generates ridge regression output – model coefficients, y_pred, rsq and nmrse

Code Snippet 6

10. The mod_out from the above code snippet generates the Ridge regression output, which includes all the model coefficients and y_pred, rsq and nmrse. Refer to the model coefficients output below (Code Snippet 7)



```
> mod_out$coefs
                           s0
(Intercept)       7.531875e+05
trend             1.964401e-01
season            1.428673e-01
holiday           1.606085e-01
competitor_sales_B 4.776625e-02
events            2.033290e-01
tv_S              4.261775e+05
ooh_S             1.434160e+05
print_S           1.520307e+05
facebook_S        2.032990e+05
search_S          2.284704e+05
newsletter        2.923098e+05
```

Code Snippet 7

Resources:

1. https://github.com/facebookexperimental/Robyn/blob/main/R/R/model.R
2. https://glmnet.stanford.edu/articles/glmnet.html
3. https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/scale