Machine Learning 2ST129 26605 HT2023 Assignment 1

Anonymous Student

November 12, 2023

Contents

General Information

- Time used for reading: about 6 hours
- Time used for basic assignment: 18 hours
- Time used for extra assignment 4 hours
- Good with lab: Very practical. Now i know how to compute gradient descents.
- Things to improve in the lab: Perhaps either more specified reading instructions or less reading. For example, i did not feel that everything in the articles was relevant and I just skipped most of it. Likewise with some of the pages in the book that were mandatory.

  I also started with the assignment the previous week, which made me do many tasks not included in this. Which is my own fault but just cost me some extra hours. Anyway, in case some of the tasks are still referenced wrong then it is probably because of that.

Task 1: Basic, Stochastic, and Mini-Batch Gradient Descent

Gradient for logistic regression

Derive gradient

For this task we want to derive the gradient for $NNL(\theta, \boldsymbol{y}, \boldsymbol{X}) = -l(\theta, \boldsymbol{y}, \boldsymbol{X})$ where we have that the likelihood function can be expressed as:

$$L(\theta, \boldsymbol{y}, \boldsymbol{X}) = \prod_{i=1}^{n} p_i^{y_i}(1-p_i)^{1-yi}$$

and the log likelihood:

$$\begin{aligned} l(\theta, \mathbf{y}, \mathbf{X}) &= \sum_{i=1}^{n} y_i \log(p_i) + (1-y_i)\log(1-p_i) && (1) \\ &= \sum_{i=1}^{n} y_i \mathbf{x}_i \theta + \log(1-p_i) && (2) \\ &= \sum_{i=1}^{n} y_i \mathbf{x}_i \theta - \log(1+\exp(\mathbf{x}_i\theta)). && (3) \end{aligned}$$

The partial derivatives of the log likelihood with respect to the different parameters can be expressed as:

$$\frac{\partial l}{\partial \theta_j} = \sum_{i=1}^{n}\left(y_i x_{ij} - \frac{x_{ij}\exp(\mathbf{x}_i\theta)}{1+\exp(\mathbf{x}_i\theta)}\right)$$

Then we have that the gradient is simply when we take the partial derivatives of the log-likelihood function with respect to all parameters such that:

$$\nabla l(\theta, \mathbf{y}, \mathbf{X}) = -\left(\frac{\partial l}{\partial \theta_1}, \frac{\partial l}{\partial \theta_2}, \ldots, \frac{\partial l}{\partial \theta_p}\right)$$

Where p is the number of parameters in the vector θ. If we instead write it in matrix form, then it can be expressed as:

$$\nabla l(\theta, \mathbf{y}, \mathbf{X}) = -\mathbf{X}^T(\mathbf{y}-\mathbf{p})$$

where we have that $\boldsymbol{y}$ is a vector of the values of y_i. $\boldsymbol{X}$is the $N \times (p+1)$ design matrix of x_i values, $\boldsymbol{P}$ is the vector of fitted probabilities p_i

It is this matrix notation we will work with in practise when we compute the gradient in R.

Implement gradient in R

```
#Libraries
 library(tidyverse)
 library(xtable)
 library(uuml)
 library(gridExtra)
 library(glmnet)
 library(parallel)
```

```r
data("binary")
binary$gre_sd <- (binary$gre - mean(binary$gre)) /sd(binary$gre)
binary$gpa_sd <- (binary$gpa - mean(binary$gpa)) /sd(binary$gpa)
X <- model.matrix(admit ~ gre_sd + gpa_sd, binary)
y <- binary$admit
```

```r
#' gradient function
#' @param y  vector of observations outcomes
#' @param X Data Matrix
#' @param theta Parameters
#' @return The gradient

# Gradient function
ll_grad <- function(y, X, theta) {
  n <- length(y)
  P <- 1 / (1 + exp(-X %*% theta))
  grad <- t(X) %*% (y-P) / n
  return(grad)
}


round(t(ll_grad(y, X, theta = c(0, 0, 0))), 4)
```

```
     (Intercept) gre_sd gpa_sd
[1,]     -0.1825 0.0857 0.0829
```

```r
round(t(ll_grad(y, X, theta = c(-1, 0.5, 0.5))), 4)
```

```
     (Intercept)  gre_sd  gpa_sd
[1,]      0.0217 -0.0395 -0.0426
```

Implement Gradient Descent

1

Here we just print the values from the fitted logistic regression:

```r
theta <- c(0, 0, 0)
ll(y, X, theta)
```

```
[1] -277.2589
```

```r
glm(y ~X  -1 , family = binomial(link = "logit"))$coefficients %>%
  print()
```

```
X(Intercept)      Xgre_sd      Xgpa_sd
  -0.8097503    0.3108184    0.2872087
```

2

Now we want to implement the different gradient descent algorithms.

```r
#' Batch gradient descent
#' @param  learn_rate the step size
#' @param epochs number of iterations
```

```r
batch_gsd <- function(y, X, theta = rep(0, ncol(X)), learn_rate, epochs) {
    results <- matrix(0.0, ncol = ncol(X) + 2L, nrow = epochs)
  colnames(results) <- c("epoch", "nll", colnames(X))
  for (epoch in 1:epochs) {
    gradient <- -ll_grad(y, X, theta) #negative gradient
    theta <- theta - (learn_rate * gradient)

    results[epoch, "epoch"] <- epoch
    results[epoch, "nll"] <- ll(y, X, theta)
    results[epoch, -(1:2)] <- theta
  }
  attributes(results)$learn_rate <- learn_rate
  attributes(results)$epochs <- epochs
  return(results)
}
```

```r
stochastic_gsd <- function(y, X, theta = rep(0, ncol(X)),
                                   learn_rate, epochs, seed=1337 ){
  set.seed(seed)
  results <- matrix(0.0, ncol = ncol(X) + 2L, nrow = epochs)
  colnames(results) <- c("epoch", "nll", colnames(X))
    for (epoch in 1:epochs) {
      index_order <- sample(length(y))
      #shuffle the data by indexing in a random order

      for (i in index_order ){
    gradient <- -ll_grad(y[i], X[i,, drop=FALSE], theta)
    #dont drop "redundant" information to keep it as a matrix
   theta <- theta - (learn_rate * gradient)
      }
    results[epoch, "epoch"] <- epoch
    results[epoch, "nll"] <- ll(y, X, theta)
    results[epoch, -(1:2)] <- theta

    }
  attributes(results)$learn_rate <- learn_rate
  attributes(results)$epochs <- epochs
  return(results)
}
```

```r
minibatch_gsd <- function(y, X, theta = rep(0, ncol(X)),
                          sample_size, learn_rate, epochs,
                          batch_size, seed=1337){
  set.seed(seed)
  n <- length(y)
  num_batches <- ceiling(n/batch_size)

results <- matrix(0.0, ncol = ncol(X) + 2L, nrow = epochs)
colnames(results) <- c("epoch", "nll", colnames(X))

# Run algorithm
for(epoch in 1:epochs){
 index_order <- sample(length(y))
 #shuffle the data by indexing in a random order

### Put the algorithm code here ###
  for (i in 1:num_batches){

     batch_index <- which(findInterval(index_order,
                                       seq(1, n, by = batch_size)) == i)
     #which x and y  to subset for each iteration by
     #dividing indexes to different batches
    batch_X <- X[batch_index,]
    batch_y <- y[batch_index]

    gradient <- -ll_grad(y=batch_y, X = batch_X, theta=theta ) / batch_size
   theta <- theta - (learn_rate * gradient)
  }
 # Store epoch, nll and output results
results[epoch, "epoch"] <- epoch
results[epoch, "nll"] <- ll(y, X, theta)
results[epoch, -(1:2)] <- theta
}

attributes(results)$learn_rate <- learn_rate
attributes(results)$epochs <- epochs
return(results)
}
```

3

Now we want to try different three different values of the learning parameter η and run the algorithm for 500 epochs. To keep the report clean from unnecessary code, most of the code used for this task will be in the appendix. But the way it is structures is so that I made a plot function given the results from one of the gradient descent functions, and then i made all the previous functions, including the plot functions more generalized to multiple values of η and then put into a main function for this task. Hence only the use of the main function will be presented here.

```
plots <- Task_1.3_main(y=y,X=X, learn_rates = list(c(0.03,1,50),
                                                   c(0.0001, 0.001, 1),
                                                   c(0.05,1,15)),
                     n_epochs=500, seed=1337,
                     batch_size = 25,
                     variable_name = "gre_sd")
```

```
plot(plots$batch_GD)
```

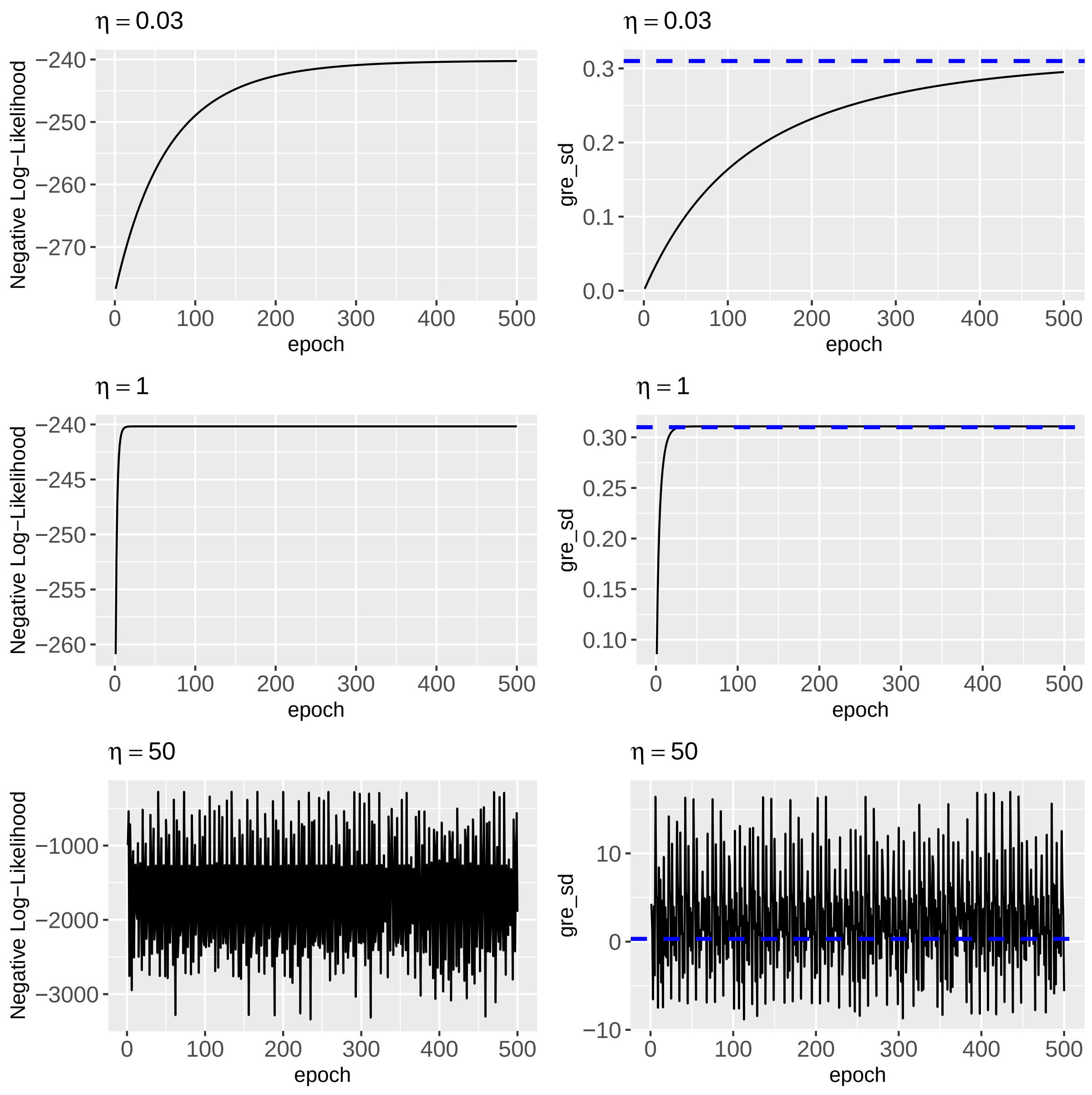

Figure 1: Results for ordinary gradient descent

```
plot(plots$stochastic_GD)
```

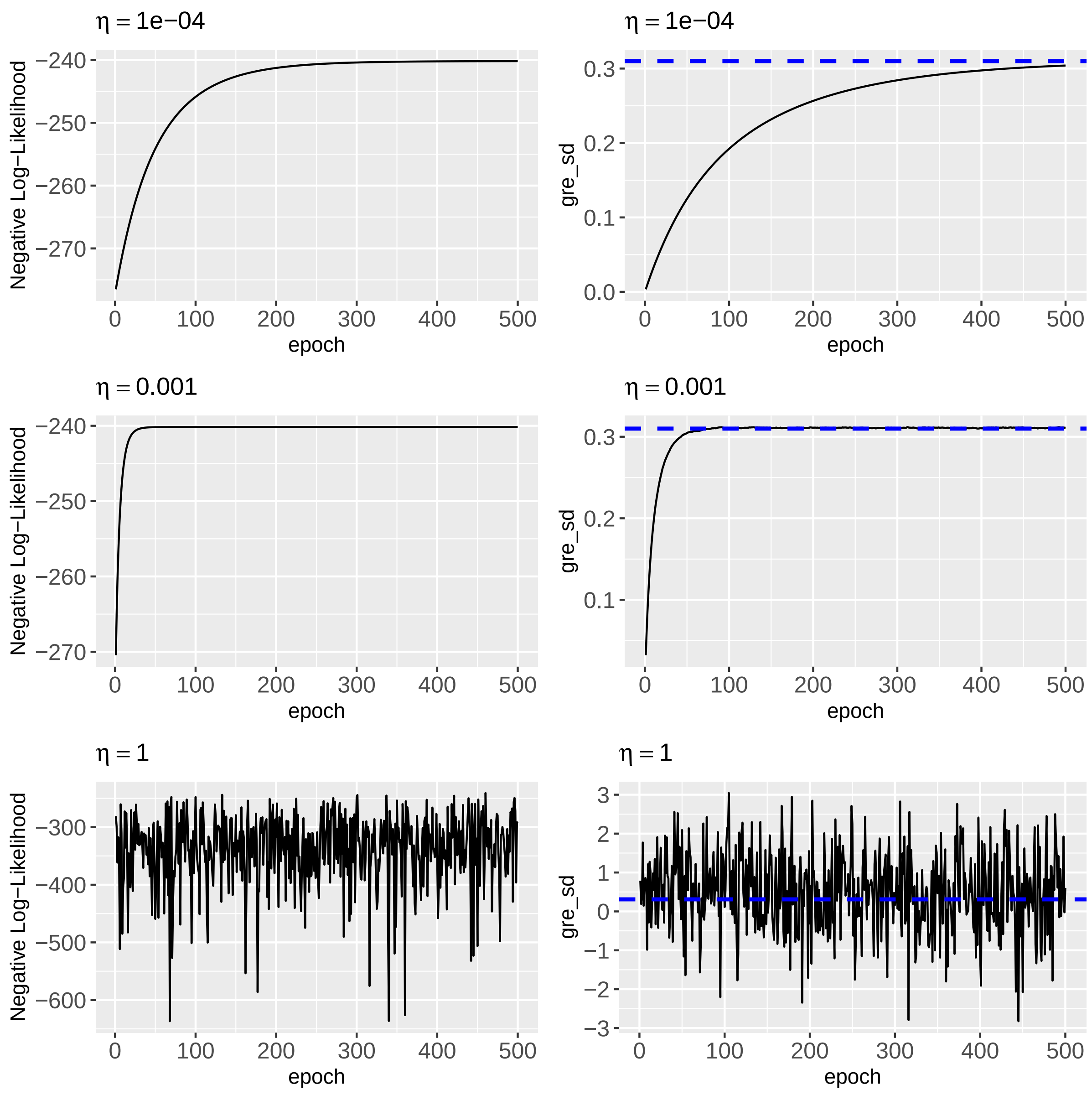

Figure 2: Results for stochastic gradient descent

```r
plot(plots$minibatch_GD)
```

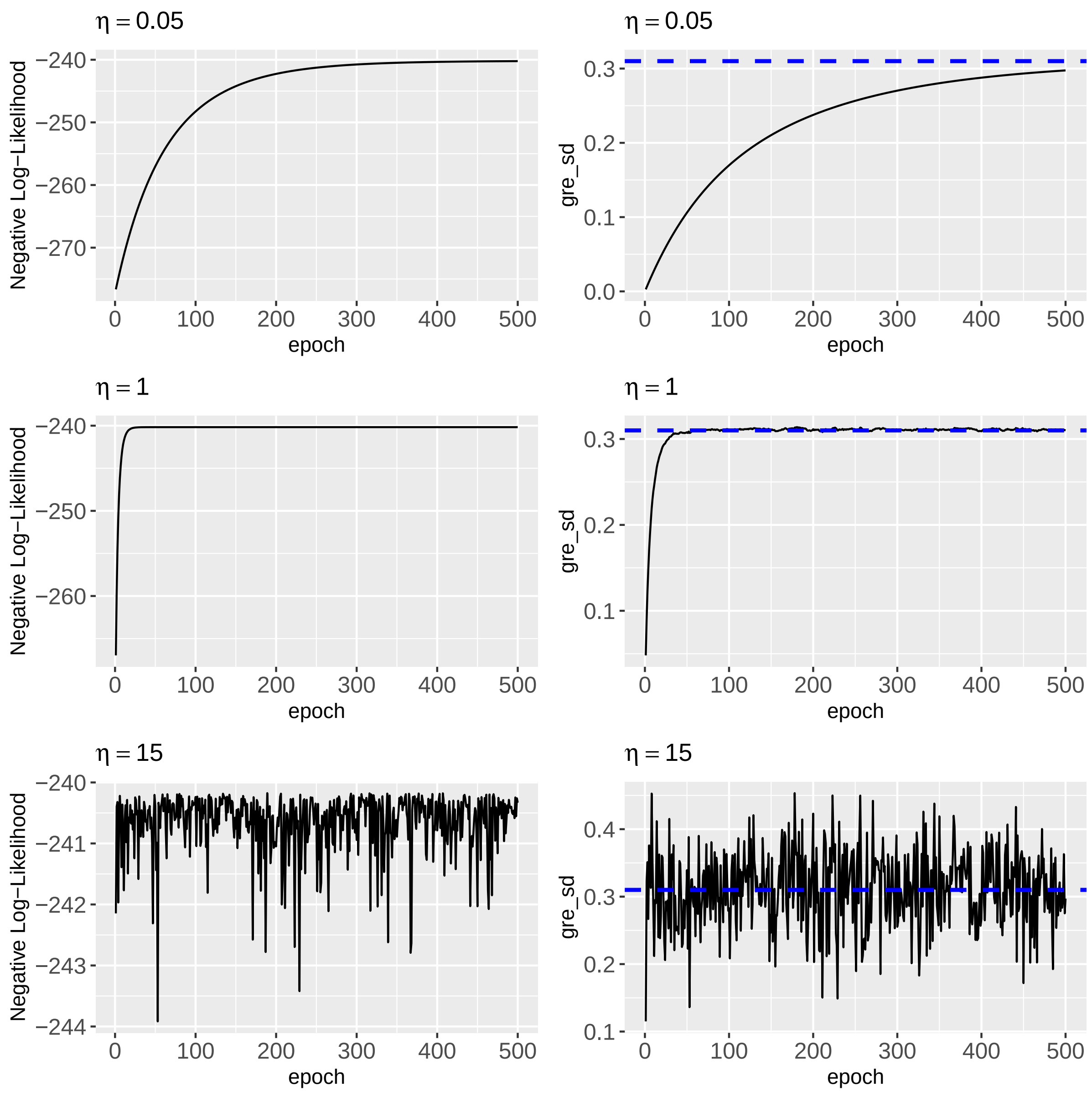

Figure 3: Results for minibatch gradient descent

Task 2: Regularized Regression

2.1 linear model

Here we try to fit a linear model to the training data

```
fit <- lm( y ~ ., data=prob2_train)
```

I wont show the full summary output due to there being many variables, but if we run the `summary()` function then we get some strange results. such as:

```
Residuals:
ALL 200 residuals are 0: no residual degrees of freedom!

Residual standard error: NaN on 0 degrees of freedom
```

```
Multiple R-squared:      1, Adjusted R-squared:    NaN
F-statistic:   NaN on 199 and 0 DF,  p-value: NA
```

as well as `Nan` for the std.error, t-values and p-values for the coefficients. This is most likely due to multicollinearity in the model, which is due to two or more variables are highly or perfectly correlated. This is perhaps not that surprising considering that there are 240 covariates.

2.2 Lasso regression

Now we want to use the `glmnet()` function to fit a linear lasso regression with $\lambda = 1$. Lasso is a regularization method which shrinks the coefficients by a penalty term λ . The `glm.net()` works by fitting the penalized maximum likelihood glm-model with the specified elastic mixing parameter α. If $\alpha = 1$, which is the value we use in this case, then its the the lasso penalty and if $\alpha = 0$ then it is the ridge penalty.

```
X <- as.matrix(prob2_train[, -241])
y <- as.matrix(prob2_train[, "y"])
X_test <- as.matrix(prob2_test[, -241])
y_test <- as.matrix(prob2_test[, "y"])
lasso_fit <- glmnet(x=X, y=y, family = "gaussian", lambda=1)

lasso_coeffs <- coef(lasso_fit)
```

The object `lasso_coeffs` contains all the coefficients of the variables in a sparse format which are from the model corresponding to the lambda value. if we just look at the first and last values it looks like this:

```
head(lasso_coeffs)

6 x 1 sparse Matrix of class "dgCMatrix"
                   s0
(Intercept) 31.492961
V1           .
V2           2.130597
V3           0.821336
V4           1.858161
V5           .

tail(lasso_coeffs)

6 x 1 sparse Matrix of class "dgCMatrix"
     s0
V235  .
V236  .
V237  .
V238  .
V239  .
V240  .
```

where the coefficients not used has a value of 0 represented by the dot. We can extract only the variables with non-zero values with this:

```
used_coefs <- lasso_coeffs[lasso_coeffs@i+1,] #@ due to S4 class
print(used_coefs)

(Intercept)          V2          V3          V4          V6          V7
```

```
 31.4929611   2.1305973   0.8213360   1.8581608  -5.2947105   0.8503707 
         V8          V9         V10 
  1.7790410   2.2202348   1.9040888 
```

```
length(used_coefs)
```

```
[1] 9
```

Hence, only 8 variables were used for this model (intercept not included).

2.3 - 2.4 10 fold cross validation

Now we want to split the data into ten folds randomly for task 2.3 and then implement a 10 fold cross-validation for task 2.4. I will just implement it in the same function here

The code for splitting the data into folds is this:

```
findInterval(cut(sample(1:n,n),breaks=nfolds),1:n)
```

which yields indexes ranging from 1 to the number of folds (10) for each observation.

I also define some helper functions to calculate the errors (I originally made them when the questions were from the previous version of the assignment when you were supposed to have both MSE and RMSE, but after it changed I figured I might as well just keep the function as it is and simply reuse it).

```
#' function to calculate MSE and RMSE
errors_func <- function(y_obs, pred){
  assertthat::assert_that(length(y_obs) == length(pred), msg="lengths differ")
  n <- length(y_obs)
  Sum_square <- sum( (y_obs - pred)^2)
  MSE <- Sum_square / n
  RMSE <- sqrt(MSE)
  res <- list("MSE"= MSE, "RMSE" = RMSE)
  return(res)
}
```

```
cross_validation <- function(X, y, n_folds, lambda, alpha=1, seed=1337){
  assertthat::are_equal(nrow(X),length(y), msg="X and y dimension mismatch")
  checkmate::assert_int(n_folds)
  set.seed(seed)

  n <- length(y)
  errors_matrix <- matrix(0, nrow= n_folds, ncol=2)
  colnames(errors_matrix) <- c("MSE", "RMSE")

  Group= findInterval(cut(sample(1:n,n),breaks=n_folds),1:n)
  #making n_folds equal groups.
  for (i in 1:n_folds){
  index <- which(Group==i)
  #index for which observations belongs to the current fold group

  X_test <- X[index,]
  X_train <- X[-index,]
  y_test <- y[index]
  y_train <- y[-index]
  model <-  glmnet(x=X_train, y=y_train, family = "gaussian",
                   lambda=lambda, alpha=alpha )
  predictions <- predict(model, s=lambda, newx=X_test, type="response")

  fold_errors <- errors_func(y_obs = y_test, pred= predictions )
  errors_matrix[i,1] <- fold_errors$MSE
  errors_matrix[i,2] <- fold_errors$RMSE


  }
  means <- apply(errors_matrix, MARGIN=2, FUN=mean)
  return(means)
}
```

Now that we have that function, to get the results for a single 10-fold validation with $\lambda = 1$ we just use:

```
results <- cross_validation(X= X,y= y, n_folds=10, lambda=1)
print(results)

     MSE     RMSE
9.986215 3.130366
```

2.5 5-fold cross-validation

Now we just use the same function again but with `n_folds=5` instead of 10.

```
results2 <- cross_validation(X= X,y= y, n_folds=5, lambda=1)
print(results2)

      MSE      RMSE
10.318704  3.172933
```

Comparing the results for the 5-folds and the 10-folds, it is pretty close. However, the ten-fold cross-validation has slightly lower RMSE.

2.6 different λ

Now to test different values of λ. Since the question is what is the best RMSE I can get, I will accept that challenge. First I create a parallel version of the cross_validation function to work for different λ-values.

```
#' @param export_vec vector of objects or functions to export to each cluster

cross_validation_parallel <- function(X, y, n_folds, lambdas, alpha=1, seed=1337, export_vec)
  n_cores <- parallel::detectCores()- 1 #saving 1 core for OS

  cl <- makeCluster(n_cores)
  clusterExport(cl, export_vec)

  results <- t(parSapply(cl, lambdas, function(lambda) {
    cross_validation(X, y, n_folds, lambda, alpha, seed)
  }))
  row_names <- t(parSapply(cl, lambdas, function(lambda){paste("lambda = ", lambda)}))
  rownames(results) <- row_names
  stopCluster(cl) #putting cores to sleep
  return(results)
}
```

Now we use the function to test it for multiple values of λ. First lets try relatively few values to see how the results looks like.

```
lambda_values <- seq(0,1, by=0.01)
lambda_cv <- cross_validation_parallel(X=X, y=y, n_folds=10, lambdas=lambda_values,
                          export_vec = c("X", "y", "cross_validation",
                                         "glmnet", "errors_func") )
```

```
plot(y=lambda_cv[,2], x=lambda_values, ylab = "RMSE")
```

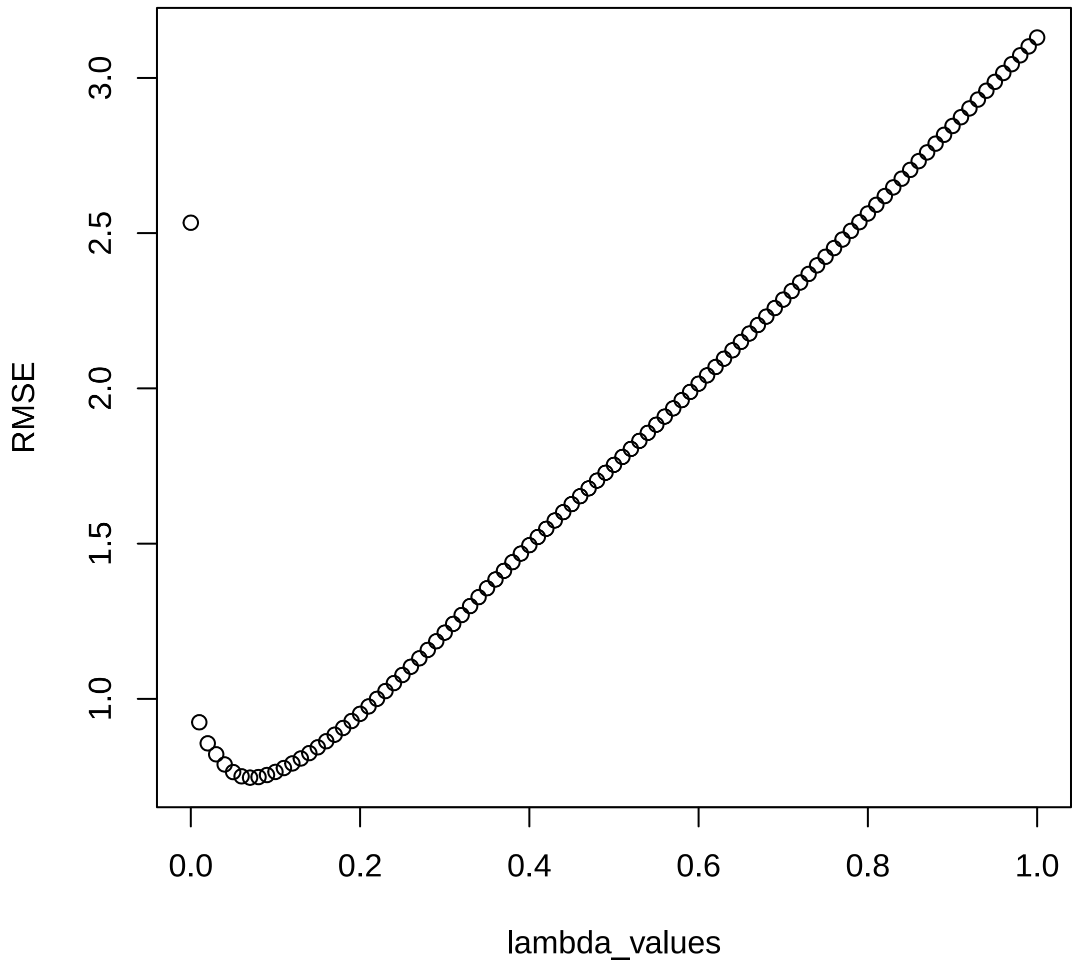

Figure 4: RMSE for different lambda values

Based on the plot, the optimal lambda values seem to be between 0-0.2 somewhere. Hence, I will focus around there.

```
#creating a sequence from 0 to 0.2 in 100 000 pieces.
new_lambdas <- seq(0,0.2, length.out = 100000)
new_lambda_cv <- cross_validation_parallel(X=X, y=y, n_folds=10, lambdas=new_lambdas,
                          export_vec = c("X", "y", "cross_validation",
                                         "glmnet", "errors_func") )

#find the minimum
which(new_lambda_cv == min(new_lambda_cv[,2]), arr.ind = TRUE)
multi_res[15,]
```

Well, I will not run this function again... but from the result I found the lambda with the least RMSE by using:

```
which(new_lambda_cv == min(new_lambda_cv[,2]), arr.ind = TRUE)
```

```
                               row col
lambda =  0.0717627176271763 35882   2
```

and then the corresponding MSE by:

```
new_lambda_cv[35882,]

 MSE      RMSE
0.5612816 0.7456794
```

Hence based on the results, the best lambda value I can get is $\lambda = 0.0717627176271763$with an RMSE of 0.7456 and MSE of 0.5613. I would probably have gotten away with simply using $\lambda = 0.07$ but now since I have this with more decimals I might as well use it. However, I do not know if it is correct or not, but each of the iterations of lambda uses the same seed, which in a way makes the differences purely due to the lambda value.

2.7 Predictions on test set

Now we fit the model according to the best lambda value and make predictions on the test data

```
best_lambda_model <- glmnet(x = X, y= y, family = "gaussian",
                            alpha=1, lambda= 0.0717627176271763)
predictions <- predict(best_lambda_model, newx = X_test, type="response")

errors_func(y=y_test, pred= predictions)

$MSE
[1] 0.650763

$RMSE
[1] 0.8066988

#just to compare that it gives the same results
uuml::rmse(y_test, predictions)

[1] 0.8066988
```

The RMSE is about 0.807 for the test set.

Task 3: Implementation of the Adam optimizer

3.1

For this task we want to implement the Adam algorithm. Since we already have code for the different gradient descents algorithms ,we can just reuse parts of that code as well as the same plot functions which have been modified to keep the values of the betas into account.

```r
Adam <- function(y, X, eta = 0.001, B1 = 0.9, B2 = 0.999,
                 epsilon = 10^-8, theta = rep(0, ncol(X)), epochs, seed=1337){
  assertthat::assert_that(length(y) == nrow(X), msg="y and X length differ")
  assertthat::assert_that(( B1 >= 0 & B1 < 1) & (B2 >=0 & B1 <1 ),
                          msg = "invalid Beta value")
  set.seed(seed)

  results <- matrix(0.0, ncol = ncol(X) + 2L, nrow = epochs)
  colnames(results) <- c("epoch", "nll", colnames(X))

  m <- 0 #initalise 1st moment vector
  v <- 0 #initalize 2nd moment vector

  for (epoch in 1:epochs) {
      index_order <- sample(length(y))
      X <- X[index_order,]
      y <- y[index_order]

      #shuffle the data by indexing in a random order
    gradient <- -ll_grad(y, X, theta) # neg.gradients w.r.t. stochastic objective at timestep

    m <- B1*m + (1-B1)*gradient # Update biased first moment estimate)
    v <- B2*v + (1-B2)*gradient^2 # Update biased second raw moment estimate)

    m_hat <- m/(1-B1^epoch) # Compute bias-corrected first moment estimate)

    v_hat <- v/(1-B2^epoch) #Compute bias-corrected second raw moment estimate)

    theta <- theta - ( eta *m_hat / ( sqrt(v_hat) + epsilon) )
    # Update parameters)

    results[epoch, "epoch"] <- epoch
    results[epoch, "nll"] <- ll(y, X, theta)
    results[epoch, -(1:2)] <- theta
  }
  attributes(results)$B1 <- B1
  attributes(results)$B2 <- B2
  attributes(results)$learn_rate<-eta
  attributes(results)$epochs <- epochs
  return(results)
}
```

3.2

```
print(Adam)
```

```
function(y, X, eta = 0.001, B1 = 0.9, B2 = 0.999,
                 epsilon = 10^-8, theta = rep(0, ncol(X)), epochs, seed=1337){
  assertthat::assert_that(length(y) == nrow(X), msg="y and X length differ")
  assertthat::assert_that(( B1 >= 0 & B1 < 1) & (B2 >=0 & B1 <1 ),
                        msg = "invalid Beta value")
  set.seed(seed)

  results <- matrix(0.0, ncol = ncol(X) + 2L, nrow = epochs)
  colnames(results) <- c("epoch", "nll", colnames(X))

  m <- 0 #initalise 1st moment vector
  v <- 0 #initalize 2nd moment vector

  for (epoch in 1:epochs) {
      index_order <- sample(length(y))
      X <- X[index_order,]
      y <- y[index_order]

      #shuffle the data by indexing in a random order
    gradient <- -ll_grad(y, X, theta) # neg.gradients w.r.t. stochastic objective at timestep

    m <- B1*m + (1-B1)*gradient # Update biased first moment estimate)
    v <- B2*v + (1-B2)*gradient^2 # Update biased second raw moment estimate)

    m_hat <- m/(1-B1^epoch) # Compute bias-corrected first moment estimate)

    v_hat <- v/(1-B2^epoch) #Compute bias-corrected second raw moment estimate)

    theta <- theta - ( eta *m_hat / ( sqrt(v_hat) + epsilon) )
    # Update parameters)

    results[epoch, "epoch"] <- epoch
    results[epoch, "nll"] <- ll(y, X, theta)
    results[epoch, -(1:2)] <- theta
  }
  attributes(results)$B1 <- B1
  attributes(results)$B2 <- B2
  attributes(results)$learn_rate<-eta
  attributes(results)$epochs <- epochs
  return(results)
}
```

3.3

First we generalize the function:

```
Adam_multi <- function(y, X, etas, B1s, B2s, epsilon, theta, epochs, seed){
  result_list <- mapply(FUN = Adam, etas, B1s, B2s,
         MoreArgs = list(y = y, X = X, epsilon = epsilon,
                         theta = theta, epochs = epochs, seed = seed),
         SIMPLIFY = FALSE)
  return(result_list)
}


task_3_main <- function(y, X, etas, B1s, B2s,
                        theta = rep(0, ncol(X)), epochs = 500,
                        epsilon= 10^-8, seed=1337, variable_name){

 res <-  Adam_multi(y=y, X=X, etas = etas, B1s = B1s, B2s = B2s,theta = theta,
             epsilon = epsilon, epochs=epochs, seed=seed)
 plots <- plot_GD_multi(res, variable_name)
 return(list(res, plots))
}
```

Now we just try some different combinations of the values for η, β_1, β_2.

```
etas_vec = c(0.001, 0.1, 1)
Beta_ones <- c(0.9, 0.85, 0.5)
Beta_twos <- c(0.999, 0.95, 0.8)
X <- model.matrix(admit ~ gre_sd + gpa_sd, binary)
y <- binary$admit

adam_results <- task_3_main(y=y, X=X, etas = etas_vec,
            B1s = Beta_ones, B2s = Beta_twos, variable_name = "gre_sd" )
```

```
plot(adam_results[[2]])
```

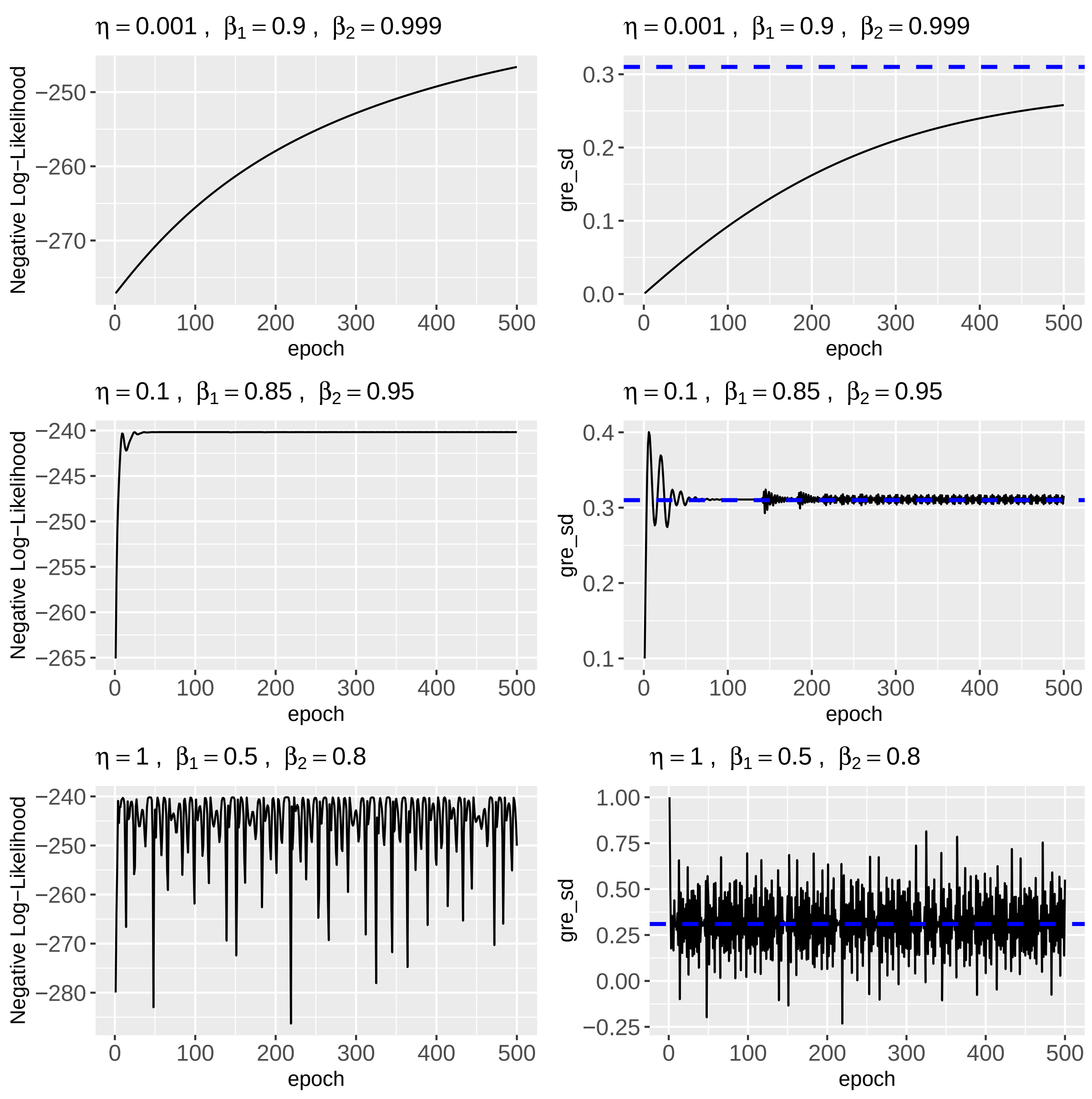

Figure 5: Results for Adam implementation

The results are seen in figure 5. The first combination is simply the recommended default settings as proposed by Kingma & Ba (2015). And as can be seen from the results, it does to to converge rather slowly as an exponential decay for this case. If we were to increase the number of epochs it looks like it would converge after around 1000-2000 epochs.

In a similar way as in task 2.3, I chose the other combination of values of η, β_1, β_2 so that it would converge faster in one case, and diverge in the other. In order to adjust for that, it would seem rather natural to change the step size or learning rate η as it is rather small. As for the hyper parameters β_1 and β_2, they control the exponential decay of the moving averages of the gradient m_t and squared gradient v_t. But i would not say that is obvious which values to choose such that it would converge faster or slower. So they were chosen by some trial and error. The second combination as can be seen in the plot, does converge rather well to the "true" value, albeit it is showing signs of slightly jumping around the "true" value, which is probably due to the changes in the hyper parameter and the effect of the moving averages. The last combination shows a scenario where it never converges.

Appendix: Additional R-code

Task 2.3

```r
#' Function to plot negative log likelihood and the results for one chosen
#' parameter
#' @param results The results from either ordinary, stochastic, or minibatch
#' gtadient descent. Also, later added support for Adam results.
#' @param  variable_name name of the parameter to plot results for
#' @return  plots for the negative log-likelihood as well as the parameter
#' of interest
#'
plot_GD <- function(results, variable_name ){
  variable_name #this is just to instantiate the object within the local
 # environment due to some weird bug later with the plot_GD_multi function
  n_epochs <- attributes(results)$epochs
  learn_rate <- attributes(results)$learn_rate
  result_df <- data.frame(results)

# Check for Adam implementation attributes (added later).
attributes_to_check <- c("B1", "B2")

if ("B1" %in% names(attributes(results)) && "B2" %in% names(attributes(results))) {
    B1 <- attributes(results)$B1
    B2 <- attributes(results)$B2
    title_text <- bquote(eta == .(learn_rate) ~
                           ", " ~ beta[1] == .(B1) ~ ", " ~ beta[2] == .(B2))

  } else {
    title_text <- bquote(eta == .(learn_rate))
  }

y_var <- ensym(variable_name) # ensym + injection with !! to inject the variable
#rather than the string provided by variable_name

   #plot negative Log-likelihood
   ll_plot <- ggplot(result_df, aes(x=epoch, y=nll)) + geom_line() +
     ylab("Negative Log-Likelihood") +
     ggtitle(title_text) +
     theme(axis.text = element_text(size = 12))
  variable_plot <-   ggplot(result_df, aes(x = epoch,
                                           y = !!y_var)) +
    geom_line() +
    geom_hline(yintercept = 0.31, col="blue",linewidth=1, linetype = "dashed") +
    ggtitle(title_text) +
    theme(axis.text = element_text(size = 12))

   plots <- grid.arrange(ll_plot, variable_plot, ncol=2)
  return(plots)
}
```

```r
#' wrapper functions to make the previous functions more
#'  generalized to multiple values of learn_rates
#' @param learn_rates a vector of different learning rates to try.
#' All other arguments are the same

batch_gsd_multi <- function(y, X, learn_rates,
                            thetas = rep(0, ncol(X)), n_epochs){
  res <- mapply(FUN = batch_gsd, learn_rate=learn_rates,
                MoreArgs = list(y=y, X=X,
                                theta = thetas, epochs = n_epochs),
                SIMPLIFY = FALSE)
return(res)
}

stochastic_gsd_multi <- function(y, X, learn_rates,
                                 thetas = rep(0, ncol(X)), n_epochs, seed=1337){
  res <- mapply(FUN = stochastic_gsd, learn_rate=learn_rates,
                MoreArgs = list(y=y, X=X,
                                theta = thetas, epochs = n_epochs,
                                seed=seed),
                SIMPLIFY = FALSE)
  return(res)
}

minibatch_gsd_multi <- function(y, X, learn_rates,
                                 thetas = rep(0, ncol(X)), n_epochs,
                                seed=1337, batch_size) {

  res <- mapply(FUN = minibatch_gsd, learn_rate=learn_rates,
                MoreArgs = list(y=y, X=X,
                                theta = thetas, epochs = n_epochs,
                                seed=seed, batch_size = batch_size),
                SIMPLIFY = FALSE)
  return(res)
  }


#' function to generalize the plot_GD function to multiple plots in the same grid

plot_GD_multi <-function(results_list, variable_name){
  extracted_plots <- lapply(results_list, FUN= plot_GD,
                            variable_name = variable_name)

  merged_plots <- do.call(grid.arrange, c(extracted_plots,
                                          nrow=length(extracted_plots)))
  #using grid.arrange on all extracted plots
  return(merged_plots)
}
```

```r
Task_3_main <- function(y, X, thetas = rep(0, ncol(X)) , learn_rates,
                        n_epochs, seed, batch_size, variable_name){
```

```r
batch_results <- batch_gsd_multi(y = y, X=X,
                                 learn_rates = learn_rates[[1]],
                                 n_epochs = 500, thetas=thetas)
print("Ordinary GD done")

stochastic_results <- stochastic_gsd_multi(y = y, X=X,
                                           learn_rates = learn_rates[[2]],
                                           n_epochs = n_epochs, seed=seed)
print("Stochastic GD done")

minibatch_results <-  minibatch_gsd_multi(y = y, X=X,
                                          learn_rates = learn_rates[[3]],
                                          n_epochs = n_epochs,
                                          batch_size=batch_size,
                                          seed=seed)
print("Minibatch GD done")
batch_plots <- plot_GD_multi(batch_results,
                             variable_name = variable_name)
stochastic_plots <- plot_GD_multi(stochastic_results,
                                  variable_name = variable_name)
minibatch_plots <- plot_GD_multi(minibatch_results,
                                 variable_name = variable_name)
list_results <- list(batch_GD =batch_plots,
                     stochastic_GD = stochastic_plots,
                     minibatch_GD = minibatch_plots)
return(list_results)
}
```