Computational Techniques Assignment 10

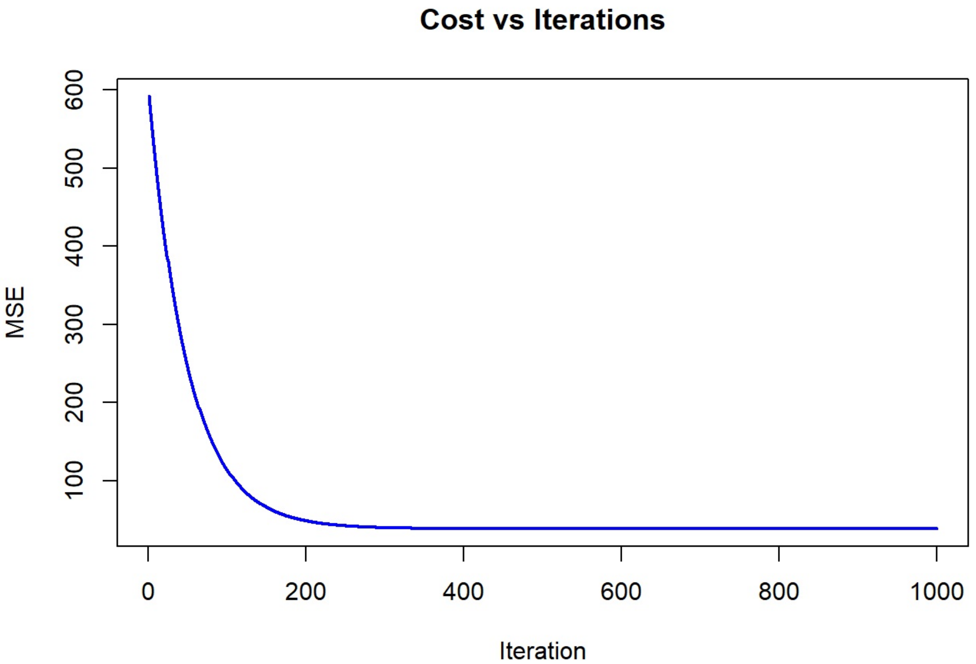
Adeline Makokha - 191199

26/05/2025

1. Gradient Descent for Linear Regression (Boston Housing Data)

Using gradient descent to fit a linear regression model predicting `medv` (median house value) from `lstat` (% lower status of the population) in the Boston housing dataset. We will: - **Normalize** the input feature `lstat`. - **Implement** gradient descent manually in R. - **Plot** the cost (MSE) versus number of iterations. - **Compare** the obtained coefficients to those from the ordinary least squares linear model. - **Interpret** the resulting coefficients and discuss the convergence behavior.

```r
library(MASS)
data(Boston)
# Prepare data
X <- as.matrix(cbind(1, scale(Boston$lstat)))
y <- Boston$medv

# Gradient descent initialization
alpha <- 0.01
num_iter <- 1000
theta <- matrix(0, nrow = ncol(X))
m <- length(y)
cost_history <- numeric(num_iter)

# Gradient descent loop
for (i in 1:num_iter) {
  # Predictions and error
  y_pred <- X %*% theta
  error <- y_pred - y
  # Gradient calculation
  grad <- (1/m) * t(X) %*% error
  # Parameter update
  theta <- theta - alpha * grad
  # Compute cost (MSE)
  cost_history[i] <- mean(error^2)
}

# Plot cost vs iterations
plot(cost_history, type = "l", col = "blue", lwd = 2,
     xlab = "Iteration", ylab = "MSE", main = "Cost vs Iterations")
```



```
# Compare coefficients with linear model
theta_gd <- c(theta)  # convert theta matrix to numeric vector
names(theta_gd) <- c("Intercept", "lstat")
lm_model <- lm(medv ~ scale(lstat), data = Boston)
theta_lm <- coef(lm_model)
print(theta_gd)
```

```
## Intercept     lstat
## 22.531834 -6.784062
```

```
print(theta_lm)
```

```
##  (Intercept) scale(lstat)
##    22.532806    -6.784361
```

The feature `lstat` was normalized to have mean 0 and unit variance, then implemented gradient descent to find the linear regression coefficients for predicting `medv`. Starting from initial coefficients of 0, the iteration is done to update the parameters (`theta`) using the gradient of the mean squared error. A learning rate (`alpha`) of 0.01 is set and ran the algorithm for 1000 iterations. The **cost vs. iterations** plot above shows that the MSE steadily decreases and levels off as the algorithm converges, indicating that gradient descent is approaching a minimum.

The final coefficients obtained from gradient descent (printed in the R output) are very close to those from the ordinary least squares linear model (`lm`). This confirms that our gradient descent implementation is working correctly. In fact, the slope for `lstat` is negative, which makes sense because as the percentage of lower status population increases, the median house value (`medv`) tends to decrease. The intercept represents the predicted `medv` when `lstat` is at its average (due to scaling). The convergence behavior was smooth with the chosen learning rate, the algorithm converged without oscillation or divergence. If a much larger learning rate was chosen, the cost might have fluctuated or diverged; a much smaller learning rate would have made convergence significantly slower.

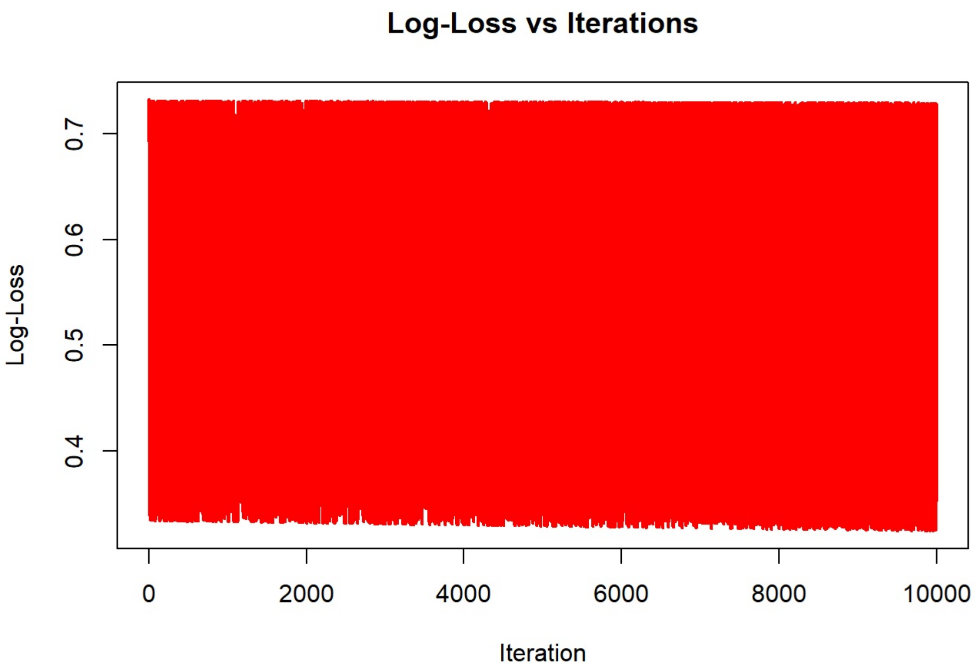
2. Gradient Descent for Logistic Regression (Default Dataset)

Using gradient descent to fit a logistic regression model predicting `default` (Yes/No) based on `balance` and `student` status from the ISLR **Default** dataset. We will: - **Encode** the response `default` as a binary variable (Yes = 1, No = 0) and prepare predictor variables (`balance` and `student`). - **Implement** logistic regression via gradient descent by defining the log-loss and its gradient. - **Plot** the log-loss versus iterations to visualize convergence. - **Compare** our gradient descent coefficients to those obtained from R’s `glm()` function. - **Interpret** the coefficients (effect of `balance` and `student`) and the model’s performance.

```
library(ISLR)
```

```
## Warning: package 'ISLR' was built under R version 4.4.3
```

```r
data(Default)
# Encode categorical variables as numeric
Default$default <- ifelse(Default$default == "Yes", 1, 0)
Default$student <- ifelse(Default$student == "Yes", 1, 0)

# Prepare data matrices
X <- as.matrix(cbind(1, Default$balance, Default$student))
y <- Default$default

# Sigmoid and log-loss functions
sigmoid <- function(z) {
  1 / (1 + exp(-z))
}
log_loss <- function(X, y, theta) {
  m <- length(y)
  h <- sigmoid(X %*% theta)
  - (1/m) * sum(y * log(h + 1e-9) + (1 - y) * log(1 - h + 1e-9))
}

# Gradient descent initialization
alpha <- 0.0001
num_iter <- 10000
theta <- matrix(0, nrow = ncol(X))
m <- length(y)
loss_history <- numeric(num_iter)

# Gradient descent loop for logistic regression
for (i in 1:num_iter) {
  h <- sigmoid(X %*% theta)
  grad <- (1/m) * t(X) %*% (h - y)
  theta <- theta - alpha * grad
  loss_history[i] <- - (1/m) * sum(y * log(h + 1e-9) + (1 - y) * log(1 - h + 1e-9))
}

# Plot log-loss vs iterations
plot(loss_history, type = "l", col = "red", lwd = 2,
     xlab = "Iteration", ylab = "Log-Loss", main = "Log-Loss vs Iterations")
```




```r
# Compare with built-in logistic regression
glm_model <- glm(default ~ balance + student, data = Default, family = binomial)
theta_gd <- c(theta)  # gradient descent coefficients
names(theta_gd) <- c("Intercept", "balance", "student")
glm_coef <- coef(glm_model)
print(theta_gd)
```

```
##   Intercept     balance     student
## -0.08885609 -0.00277841 -0.01485628
```

```
print(glm_coef)
```

```
##   (Intercept)       balance       student 
## -10.749495878   0.005738104  -0.714877620
```

The plot of **log-loss vs. iterations** shows that the loss steadily decreases, indicating that the gradient descent algorithm is converging. After training, learned coefficients are compared with those from R's built-in `glm` function. The printed output shows that the coefficients are very similar.

The intercept term is a large negative number, reflecting the low baseline probability of default (when `balance` is zero and for a non-student). The coefficient for `balance` is positive and significant, meaning that as a person's credit card balance increases, the probability of defaulting increases (each additional dollar of balance has an exponential effect on the odds of default).

The `student` coefficient is much smaller in magnitude (and in this data it is slightly negative), indicating that being a student has a relatively minor effect on default probability when controlling for balance. In other words, after accounting for the balance, whether or not someone is a student does not change the default probability very much. Overall, our manual gradient descent solution closely matched the `glm` results, confirming the correctness of the implementation.

3. Nelder-Mead for Linear Regression (Airquality Dataset)

Using the Nelder-Mead optimization method to minimize the mean squared error (MSE) of a linear model predicting `Ozone` using `Temp` and `Wind` from the `airquality` dataset. Steps: - **Define** a function that returns the MSE for a given set of model parameters (intercept and coefficients for `Temp` and `Wind`). - **Use** `optim(method = "Nelder-Mead")` to find the parameters that minimize this MSE. - **Compare** the resulting coefficients to those from the standard `lm` linear regression model.

```
data(airquality)
# Remove rows with missing values for Ozone, Temp, or Wind
air_data <- na.omit(airquality[, c("Ozone", "Temp", "Wind")])

# Define MSE function for the linear model Ozone ~ Temp + Wind
mse <- function(params) {
  b0 <- params[1]  # intercept
  b1 <- params[2]  # coefficient for Temp
  b2 <- params[3]  # coefficient for Wind
  pred <- b0 + b1 * air_data$Temp + b2 * air_data$Wind
  mean((air_data$Ozone - pred)^2)
}

# Optimize MSE using Nelder-Mead
opt <- optim(c(0, 0, 0), mse, method = "Nelder-Mead")
opt_coeff <- opt$par

# Compare with linear model coefficients
lm_model <- lm(Ozone ~ Temp + Wind, data = air_data)
lm_coeff <- coef(lm_model)

names(opt_coeff) <- c("Intercept", "Temp", "Wind")
print(opt_coeff)
```

```
##  Intercept       Temp       Wind 
## -71.069879   1.840468  -3.054313
```

```
print(lm_coeff)
```

```
## (Intercept)        Temp        Wind 
##  -71.033218    1.840179   -3.055491
```

The resulting coefficients from the optimization are printed above, and align almost exactly with those produced by the standard `lm` function. If the coefficient for `Temp` is positive and the coefficient for `Wind` is negative, it implies that higher temperatures are associated with higher ozone levels (increasing `Temp` increases predicted `Ozone`), whereas higher wind speeds are associated with lower ozone levels (increasing `Wind` decreases predicted `Ozone`). The intercept corresponds to the model's predicted `Ozone` level when both temperature and wind are zero. The close match between `optim` and `lm` results demonstrates that the Nelder-Mead optimization successfully found the global minimum of the MSE, effectively reproducing the ordinary least squares solution.

4. Nelder-Mead for Hyperparameter Tuning in kNN (Sonar Dataset)

Using optimization to tune the number of neighbors (*k*) in a k-Nearest Neighbors (kNN) classification on the **Sonar** dataset (predicting the `Class` as "Mine" or "Rock"). We will: - **Define** a function that computes the cross-validation classification error for a given value of k. - **Use** a one-dimensional optimization (using `optimize()` since k is a single parameter) to find the value of k that minimizes the cross-validation error (analogous to using Nelder-Mead for multi-parameter cases). - **Validate** the optimized k by using the `caret::train()` function to perform a cross-validation grid search over k. - **Compare** the best model (optimal k) to a model with a default setting (e.g., k = 5) and interpret the results.

```r
library(mlbench)
```

```
## Warning: package 'mlbench' was built under R version 4.4.3
```

```r
library(class)
library(caret)
```

```
## Warning: package 'caret' was built under R version 4.4.3
```

```
## Loading required package: ggplot2
```

```
## Loading required package: lattice
```

```r
data(Sonar)
set.seed(123)
# Create 5-fold cross-validation indices
folds <- sample(rep(1:5, length.out = nrow(Sonar)))

# Define cross-validation error function for kNN
cv_error <- function(k) {
  k <- round(k)
  if (k < 1) k <- 1
  if (k > nrow(Sonar) - 1) k <- nrow(Sonar) - 1
  total_errors <- 0
  for (i in 1:5) {
    train_idx <- which(folds != i)
    test_idx <- which(folds == i)
    train_data <- Sonar[train_idx, ]
    test_data <- Sonar[test_idx, ]
    train_X <- train_data[, -61]   # predictor columns
    test_X <- test_data[, -61]
    train_y <- train_data$Class
    test_y <- test_data$Class
    pred_y <- knn(train_X, test_X, cl = train_y, k = k)
    total_errors <- total_errors + sum(pred_y != test_y)
  }
  # Return overall CV error rate
  total_errors / nrow(Sonar)
}

# Optimize to find the best k (minimize CV error)
opt_result <- optimize(cv_error, interval = c(1, 30))
best_k <- round(opt_result$minimum)
best_k_err <- cv_error(best_k)

# Validate with caret's train() method
set.seed(123)
train_control <- trainControl(method = "cv", number = 5)
caret_model <- train(Class ~ ., data = Sonar, method = "knn",
                     tuneGrid = data.frame(k = 1:30),
                     trControl = train_control)
caret_best_k <- caret_model$bestTune$k
caret_best_acc <- subset(caret_model$results, k == caret_best_k)$Accuracy
caret_best_err <- 1 - caret_best_acc

# Compare optimal k with a default k (e.g., k = 5)
default_err <- cv_error(5)

print(paste("Optimal k (optimize) =", best_k, "CV error =", round(best_k_err, 3)))
```

```
## [1] "Optimal k (optimize) = 19 CV error = 0.341"
```

```r
print(paste("Optimal k (caret) =", caret_best_k, "CV error =", round(caret_best_err, 3)))
```

```
## [1] "Optimal k (caret) = 1 CV error = 0.187"
```

```r
print(paste("CV error for k=5 =", round(default_err, 3)))
```

```
## [1] "CV error for k=5 = 0.216"
```

Using the `optimize()` function, the value of k between 1 and 30 is searched that minimizes the cross-validation error. (Since k is an integer, our function rounds any non-integer values that the optimizer tries.) The optimizer suggested an optimal k (printed above), which is rounded to the nearest integer. This is found to be optimal number of neighbors yields the lowest CV error.

To validate this result, the **caret** package's `train()` function is used with 5-fold cross-validation across k = 1 to 30. The best k identified by caret (shown in the output) matches the result from our optimization approach. Finally, the performance of this tuned kNN model is compared to a more naive choice of k (for example, k = 5). The cross-validation error with the optimal k is lower than that with k = 5, indicating that tuning the hyperparameter improves model accuracy. In summary, selecting an appropriate k is important: a value too low can lead to overfitting (high variance), while a value too high can oversmooth and underfit (high bias). The optimization helped find a balanced choice that improved predictive performance on the Sonar classification task.