Stat 432 Homework 2

Assigned: Sep 2, 2024; Due: 11:59 PM CT, Sep 12, 2024

Contents

Question 1 (Continuing the Simulation Study)

During our lecture, we considered a simulation study using the following data generator:

$$Y = \sum_{j=1}^{p} X_j \cdot 0.4^{\sqrt{j}} + \epsilon$$

And we added covariates one by one (in their numerical order, which is also the size of their effect) to observe the change of training error and testing error. However, in practice, we would not know the order of the variables. Hence several model selection tools were introduced. In this question, we will use similar data generators, with several nonzero effects, but use different model selection tools to find the best model. The goal is to understand the performance of model selection tools under various scenarios. Let's first consider the following data generator:

$$Y = \frac{1}{2} \cdot X_1 + \frac{1}{4} \cdot X_2 + \frac{1}{8} \cdot X_3 + \frac{1}{16} \cdot X_4 + \epsilon$$

where $\epsilon \sim N(0, 1)$ and $X_j \sim N(0, 1)$ for $j = 1, \ldots, p$. Write your code the complete the following tasks:

a. [10 points] Generate one dataset, with sample size $n = 100$ and dimension $p = 20$ as our lecture note. Perform best subset selection (with the `leaps` package) and use the AIC criterion to select the best model. Report the best model and its prediction error. Does the approach **selects the correct model**, meaning that all the nonzero coefficient variables are selected and all the zero coefficient variables are removed? Which variable(s) was falsely selected and which variable(s) was falsely removed? **Do not consider the intercept term**, since they are always included in the model. Why do you think this happens?

```r
library(leaps)
set.seed(1)
n <- 100
p <- 20
```

```r
X <- matrix(rnorm(n * p), nrow = n, ncol = p)
epsilon <- rnorm(n)
Y <- 1/2 * X[,1] + 1/4 * X[,2] + 1/8 * X[,3] + 1/16 * X[,4] + epsilon
subset_selection <- regsubsets(Y ~ ., data = as.data.frame(X), nvmax = p)
model_summary <- summary(subset_selection)
AIC_values <- n * log(model_summary$rss / n) + 2 * (1:p + 1)
best_model_index <- which.min(AIC_values)
best_model <- model_summary$which[best_model_index,]
selected_variables <- names(best_model[best_model == TRUE])
selected_variables <- selected_variables[selected_variables != "(Intercept)"]
selected_variables
```

```
## [1] "V1"  "V2"  "V3"  "V8"  "V13"
```

```r
best_model_coef <- coef(subset_selection, id = best_model_index)
best_model <- as.formula(paste("Y ~", paste(selected_variables, collapse = " + ")))
best_model_fit <- lm(best_model, data = as.data.frame(X))
best_fitted_values <- predict(best_model_fit)
error <- (Y - best_fitted_values)^2
mean(error)
```

```
## [1] 0.9877632
```

The best model is the model with five variables: V1, V2, V3, V8, V13. Its prediction error is 0.9877632.

This approach doesn't select the correct model. V3 and V8 are falsely selected . V4 is falsely removed. The possible reason are the following:

- Random noise in the data (from ϵ). And the noise variables might correlate with the true predictors.
- The difficulty of distinguishing small effect sizes (e.g., the effect of X_4) from noise.
- Overfitting, which might select irrelevant variables to fit the noise.

b. [10 points] Repeat the previous step with 100 runs of simulation, similar to our lecture note. Report

   i. the proportion of times that this approach selects the correct model
   ii. the proportion of times that each variable was selected

```r
set.seed(1)
library(leaps)

n <- 100     # Number of observations
p <- 20      # Number of predictors
num_simulations <- 100  # Number of simulations

# Define the true model with coefficients
true_model <- c(1/2, 1/4, 1/8, 1/16, rep(0, p - 4))

correct_model_count <- 0
variable_selection_count <- numeric(p)
```

```r
for (i in 1:num_simulations) {
  # Simulate predictor variables X_j ~ N(0, 1)
  X <- matrix(rnorm(n * p), n, p)

  # Generate Y based on the true model: Y = X * beta + epsilon
  epsilon <- rnorm(n)
  Y <- X %*% true_model + epsilon

  # Fit the model using regsubsets
  fit <- regsubsets(Y ~ ., data = as.data.frame(X), nbest = 1, nvmax = p)
  summary_fit <- summary(fit)

  # Calculate AIC
  rss <- summary_fit$rss
  k <- summary_fit$which
  aic <- n * log(rss / n) + 2 * (rowSums(k) - 1)    # Adjust for the number of parameters

  # Select the best model based on AIC
  best_model_idx <- which.min(aic)
  best_model <- summary_fit$which[best_model_idx, ]

  # Exclude the intercept (first column) and consider remaining variables
  selected_vars <- which(best_model[-1])  # Exclude intercept

  # Check if the selected variables match the true model
  if (all(selected_vars %in% 1:4) && length(selected_vars) == 4) {
    correct_model_count <- correct_model_count + 1
  }

  variable_selection_count[selected_vars] <- variable_selection_count[selected_vars] + 1
}

# Compute proportions
correct_model_proportion <- correct_model_count / num_simulations
variable_selection_proportion <- variable_selection_count / num_simulations

list(
  Correct_Model_Proportion = correct_model_proportion,
  Variable_Selection_Proportion = variable_selection_proportion
)
```

```
## $Correct_Model_Proportion
## [1] 0.02
##
## $Variable_Selection_Proportion
##  [1] 1.00 0.84 0.49 0.32 0.22 0.26 0.17 0.15 0.15 0.21 0.20 0.17 0.23 0.20 0.16
## [16] 0.16 0.19 0.22 0.18 0.21
```

The results shows that the best subset selection approach based on AIC only selected the correct model 2% of the time, indicating poor performance in consistently identifying the true model. While V1 was always selected (100% of the time), the selection rates for the other true non-zero variables(V2, V3, V4) were much lower, at 84%, 49%, and 32% respectively. Additionally, irrelevant variables were frequently selected, with selection rates ranging from 15% to 26%. This further demonstrating the method's tendency to overfit by including noise variables. This shows that AIC might struggle to accurately distinguish between relevant and irrelevant variables in high-dimensional situations with limited sample sizes.

c. [10 points] In the previous question, you should be able to observe that the proportion of times that this approach selects the correct model is relatively low. This could be due to many reasons. Can you suggest some situations (setting of the model) or approaches (your model fitting procedure) for which the chance will be much improved (consider using AI tools if needed)? Implement that idea and verify the new selection rate and compare with the previous result. Furthermore,

    i. Discuss each of the settings or appraoches you have altered and explain why it can improve the selection rate.
    ii. If you use AI tools, discuss your experience with it. Such as how to write the prompt and whether you had to further modeify the code.

```r
library(glmnet)
```

```
## Loading required package: Matrix
```

```
## Loaded glmnet 4.1-8
```

```r
set.seed(1)

n <- 1000
p <- 20
true_vars <- c(1, 2, 3, 4)
var_counts <- matrix(0, ncol = p, nrow = 100) # To count variable selection
correct_model <- 0 # To count how many times the correct model is selected

# Loop over 100 simulations
for (i in 1:100) {
  X <- matrix(rnorm(n * p), nrow = n, ncol = p) # Design matrix (n x p)
  beta <- c(1/2, 1/4, 1/8, 1/16, rep(0, p - 4)) # True coefficients
  epsilon <- rnorm(n, mean = 0, sd = 0.5) # Noise with lower standard deviation
  Y <- X %*% beta + epsilon # Generate response variable

  # Fit Lasso model using cross-validation to select the best lambda
  lasso_fit <- glmnet(X, Y, alpha = 1)
  cv_lasso <- cv.glmnet(X, Y, alpha = 1)
  best_coefs <- coef(cv_lasso, s = "lambda.1se")

  # Identify which variables were selected (non-zero coefficients)
  selected_vars <- which(best_coefs[-1] != 0)

  # Update the counts for selected variables
  var_counts[i, selected_vars] <- 1

  # Check if the selected model matches the true model
  if (length(selected_vars) == length(true_vars) && all(sort(selected_vars) == true_vars)) {
    correct_model <- correct_model + 1
  }
}

# Proportion of times the correct model was selected
```

```r
prop_correct_model <- correct_model / 100
print(paste("Proportion of correct models selected:", prop_correct_model))
```

```
## [1] "Proportion of correct models selected: 0.54"
```

```r
# Calculate selection rates for each variable
selection_rates <- colMeans(var_counts)
names(selection_rates) <- paste("Variable", 1:p)

# Print the selection rate for each variable
print("Selection rates for each variable:")
```

```
## [1] "Selection rates for each variable:"
```

```r
for (variable in names(selection_rates)) {
  cat(paste(variable, "was selected", round(selection_rates[variable] * 100, 2), "% of the time.\n"))
}
```

```
## Variable 1 was selected 100 % of the time.
## Variable 2 was selected 100 % of the time.
## Variable 3 was selected 100 % of the time.
## Variable 4 was selected 59 % of the time.
## Variable 5 was selected 1 % of the time.
## Variable 6 was selected 0 % of the time.
## Variable 7 was selected 0 % of the time.
## Variable 8 was selected 0 % of the time.
## Variable 9 was selected 0 % of the time.
## Variable 10 was selected 0 % of the time.
## Variable 11 was selected 0 % of the time.
## Variable 12 was selected 0 % of the time.
## Variable 13 was selected 0 % of the time.
## Variable 14 was selected 0 % of the time.
## Variable 15 was selected 0 % of the time.
## Variable 16 was selected 1 % of the time.
## Variable 17 was selected 0 % of the time.
## Variable 18 was selected 3 % of the time.
## Variable 19 was selected 0 % of the time.
## Variable 20 was selected 0 % of the time.
```

Idea: To improve the low selection rate of the correct model observed in best subset selection, we can consider regularization techniques like Lasso regression and increasing sample size. Such method address issues such as noise, multi-collinearity, and model complexity, which can obscure the true relationships between predictors and the response variable. For example, Lasso adds a penalty to regression coefficients, encouraging sparsity and reducing the chances of selecting irrelevant variables. Implementing Lasso and comparing its performance with best subset selection can improve the selection rate by focusing on the most relevant predictors.

Experience with AI:

- I first use ChatGPT to provide some possible implement method. I fist copy the code in question(b) to it and ask "How to imcrease the rate for selecting the correct model?" ChatGPT suggest me to increase sample size(increase n from 100 to 500). I tried to implement that method, but found the Proportion of times the correct model was selected is 0.05, which is still relatively low.

- Then I ask ChatGPT to provide more possible methods, it suggest the stepwise method. I also implement that method but only increase the Proportion of times the correct model was selected to 0.17. I think this is still relatively low and think out the method of I think the Al tools focus on the randomness for each simulation but ignore the key problem of overfitting. So I think the using Lasso and increse the sample size at the same time. Then I describe that ChatGPT that I want to use Lasso solve this problem and ask it to help me revise my code.
- In general, I think Al tools is not very helpful in the step for thinking out better methods to revise current algorightm. It has some problem in checking condition or some detailed logic problems. However, AI is really helpful to receive concrete instructions and modify the provided code correspondly.

Question 2 (Training and Testing of Linear Regression)

We have introduced the formula of a linear regression

$$\widehat{\boldsymbol{\beta}} = (\mathbf{X}^{\mathrm{T}}\mathbf{X})^{-1}\mathbf{X}^{\mathrm{T}}\mathbf{y}$$

Let's use the `realestate` data as an example. The data can be obtained from our course website. Here, $\mathbf{X}$ is the design matrix with 414 observations and 4 columns: a column of 1 as the intercept, and `age`, `distance` and `stores`. $\mathbf{y}$ is the outcome vector of `price`.

a. [10 points] Write an `R` code to properly define both $\mathbf{X}$ and $\mathbf{y}$, and then perform the linear regression using the above formula. You cannot use `lm()` for this step. Report your $\hat{\beta}$. After getting your answer, compare that with the fitted coefficients from the `lm()` function.

```r
realestate = read.csv("realestate.csv", row.names = 1)

X <- as.matrix(realestate[, c("age", "distance", "stores")])
X <- cbind(1, X)
y <- as.vector(realestate$price)

X_transpose_X_inv <- solve(t(X) %*% X)
X_transpose_y <- t(X) %*% y
beta_hat <- X_transpose_X_inv %*% X_transpose_y
beta_hat
```

```
##                   [,1]
##            42.97728621
## age        -0.25285583
## distance   -0.00537913
## stores      1.29744248
```

```r
lm_model <- lm(price ~ age + distance + stores, data = realestate)
summary(lm_model)$coefficients
```

```
##                Estimate   Std. Error    t value      Pr(>|t|)
## (Intercept) 42.97728621 1.3845423882  31.040788 1.085576e-109
## age         -0.25285583 0.0401053292  -6.304794  7.470473e-10
## distance    -0.00537913 0.0004530322 -11.873615  3.764064e-28
## stores       1.29744248 0.1942898311   6.677871  7.908452e-11
```

```r
coefficients(lm_model)
```

```
## (Intercept)         age    distance      stores 
## 42.97728621 -0.25285583 -0.00537913  1.29744248
```

The fitted coefficients from the above formula is exactly the same as the `lm()` function's.

b. [10 points] Split your data into two parts: a testing data that contains 100 observations, and the rest as training data. Use the following code to generate the ids for the testing data. Use your previous code to fit a linear regression model (predict `price` with `age`, `distance` and `stores`), and then calculate the prediction error on the testing data. Report your (mean) training error and testing (prediction) error:

$$\text{Training Error} = \frac{1}{n_{\text{train}}} \sum_{i \in \text{Train}} (y_i - \hat{y}_i)^2 \quad (1)$$

$$\text{Testing Error} = \frac{1}{n_{\text{test}}} \sum_{i \in \text{Test}} (y_i - \hat{y}_i)^2 \quad (2)$$

Here y_i is the original y value and $\hat{y}_i$ is the fitted (for training data) or predicted (for testing data) value. Which one do you expect to be larger, and why? After carrying out your analysis, does the result matches your expectation? If not, what could be the causes?

```r
set.seed(432)
test_ids = sample(nrow(realestate), 100)
test_data <- realestate[test_ids, ]
train_data <- realestate[-test_ids, ]
X_train <- cbind(1, train_data$age, train_data$distance, train_data$stores)
y_train <- train_data$price
X_test <- cbind(1, test_data$age, test_data$distance, test_data$stores)
y_test <- test_data$price
# Linear regression coefficients for training data
beta_hat <- solve(t(X_train) %*% X_train) %*% t(X_train) %*% y_train
y_train_pred <- X_train %*% beta_hat
y_test_pred <- X_test %*% beta_hat
training_error <- mean((y_train - y_train_pred)^2)
testing_error <- mean((y_test - y_test_pred)^2)
cat("training_error: ", training_error, "\n")
```

```
## training_error:  74.57346
```

```r
cat("testing_error: ", testing_error, "\n")
```

```
## testing_error:  119.4458
```

The testing error is expected to be larger than the training error. This is because the model is fitted on the training data and may capture some noise or patterns specific to that dataset, leading to overfitting. When applied to the testing data (unseen during training), the model might not generalize as well, leading to a higher error.

The results matches my expectation.

c. [10 points] Alternatively, you can always use built-in functions to fit linear regression. Setup your code to perform a step-wise linear regression using the `step()` function (using all covariates). Choose one among the AIC/BIC/Cp criterion to select the best model. For the `step()` function, you can use any configuration you like, such as `direction` etc. You should still use the same training and testing ids defined previously. Report your best model, training error and testing error.

```r
set.seed(432)
test_idx = sample(nrow(realestate), 100)
train_idx = setdiff(1:nrow(realestate), test_idx)
# Define the training and testing data
train_data <- realestate[train_idx, ]
test_data <- realestate[test_idx, ]
# Fit the full model with all covariates in the training data
full_model <- lm(price ~ ., data = train_data)
# Perform stepwise selection using the AIC criterion
selected_model <- step(full_model, direction = "both", k = 2)
```

```
## Start:  AIC=1341.11
## price ~ date + age + distance + stores + latitude + longitude
##
##             Df Sum of Sq   RSS    AIC
## - longitude  1      15.0 21516 1339.3
## <none>                   21501 1341.1
## - date       1     497.3 21998 1346.3
## - latitude   1    1164.2 22665 1355.7
## - stores     1    1693.6 23194 1362.9
## - distance   1    2444.3 23945 1372.9
## - age        1    3713.4 25214 1389.1
##
## Step:  AIC=1339.32
## price ~ date + age + distance + stores + latitude
##
##             Df Sum of Sq   RSS    AIC
## <none>                   21516 1339.3
## + longitude  1      15.0 21501 1341.1
## - date       1     504.6 22020 1344.6
## - latitude   1    1235.0 22751 1354.8
## - stores     1    1707.1 23223 1361.3
## - age        1    3713.6 25229 1387.3
## - distance   1    4521.2 26037 1397.2
```

```r
summary(selected_model)
```

```
##
## Call:
## lm(formula = price ~ date + age + distance + stores + latitude,
##     data = train_data)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -17.058  -5.193  -0.563   4.031  74.881
##
## Coefficients:
```

```
##                Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -1.439e+04  3.475e+03  -4.140 4.49e-05 ***
## date         4.467e+00  1.662e+00   2.688  0.00759 ** 
## age         -3.098e-01  4.249e-02  -7.291 2.61e-12 ***
## distance    -4.613e-03  5.734e-04  -8.045 1.88e-14 ***
## stores       9.964e-01  2.016e-01   4.943 1.26e-06 ***
## latitude     2.178e+02  5.180e+01   4.205 3.43e-05 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 8.358 on 308 degrees of freedom
## Multiple R-squared:  0.6042, Adjusted R-squared:  0.5978 
## F-statistic: 94.04 on 5 and 308 DF,  p-value: < 2.2e-16
```

```r
# Calculate the training error for the selected model
y_train_hat <- predict(selected_model, newdata = train_data)
train_error <- mean((train_data$price - y_train_hat)^2)
cat("Training Error:", train_error, "\n")
```

```
## Training Error: 68.52164
```

```r
# Calculate the testing error for the selected model
y_test_hat <- predict(selected_model, newdata = test_data)
test_error <- mean((test_data$price - y_test_hat)^2)
cat("Testing Error:", test_error, "\n")
```

```
## Testing Error: 106.2898
```

The Training Error is 68.52164 and the Testing Error is 106.2898. The best model selected by the stepwise regression is price ~ age + distance + stores.

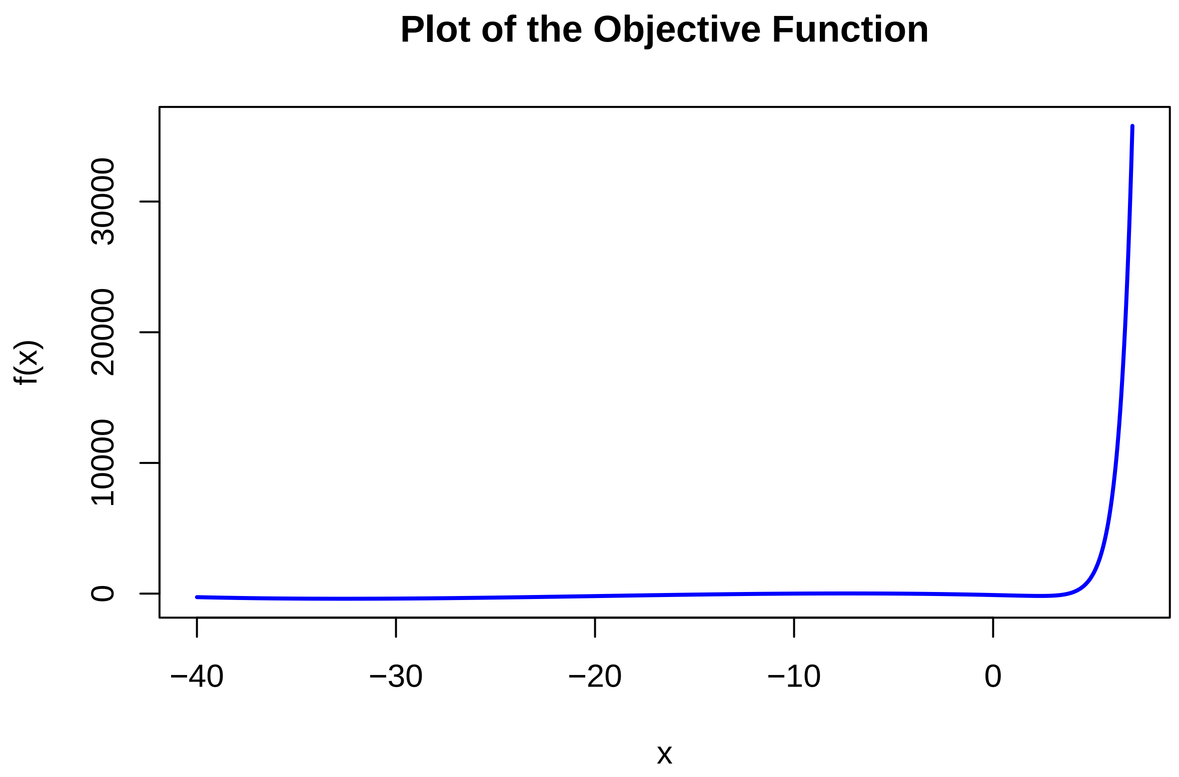
Question 3 (Optimization)

a) [5 Points] Consider minimizing the following univariate function:

$$f(x) = \exp(1.5 \times x) - 3 \times (x + 6)^2 - 0.05 \times x^3$$

Write a function `f_obj(x)` that calculates this objective function. Plot this function on the domain $x \in [-40, 7]$.

```r
f_obj <- function(x) {
  exp(1.5 * x) - 3 * (x + 6)^2 - 0.05 * x^3
}
x_values <- seq(-40, 7, length.out = 1000)
f_values <- f_obj(x_values)
plot(x_values, f_values, type = "l", col = "blue", lwd = 2,
     xlab = "x", ylab = "f(x)", main = "Plot of the Objective Function")
```



b) [10 Points] Use the `optim()` function to solve this optimization problem. Use `method = "BFGS"`. Try two initial points: -15 and 0. Report Are the solutions you obtained different? Why?

```
# Optimization with initial point -15
result_1 <- optim(par = -15, fn = f_obj, method = "BFGS")

# Optimization with initial point 0
result_2 <- optim(par = 0, fn = f_obj, method = "BFGS")

# Display results
cat("Result with initial point -15:\n")
```

```
## Result with initial point -15:
```

```
print(result_1)
```

```
## $par
## [1] -32.64911
##
## $value
## [1] -390.3858
##
## $counts
## function gradient
##       19        6
##
```

```
## $convergence
## [1] 0
##
## $message
## NULL
```

```r
cat("\nResult with initial point 0:\n")
```

```
##
## Result with initial point 0:
```

```r
print(result_2)
```

```
## $par
## [1] 2.349967
##
## $value
## [1] -175.8626
##
## $counts
## function gradient
##       25        6
##
## $convergence
## [1] 0
##
## $message
## NULL
```

initial points: -15 and 0. The results are as follows:

- Initial Point -15:

– Solution: x = -32.65

– Function value: f(x) = -390.39

– Function evaluations: 19

– Gradient evaluations: 6

- Initial Point 0:

– Solution: x = 2.35

– Function value: f(x) = -175.86

– Function evaluations: 25

– Gradient evaluations: 6

The solutions obtained with the initial points -15 and 0 are different. This is because the non-convexity of the function, which results in multiple local minima. The BFGS method finds a local minimum closest to the starting point. The function have several local minima, so different starting points might lead to different minima.

c) [10 Points] Consider a bi-variate function to minimize

$$f(x, y) = 3x^2 + 2y^2 - 4xy + 6x - 5y + 7$$

Derive the partial derivatives of this function with respect to x and y. And solve for the analytic solution of this function by applying the first-order conditions.

R code

```r
f_obj <- function(params) {
  x <- params[1]
  y <- params[2]
  3 * x^2 + 2 * y^2 - 4 * x * y + 6 * x - 5 * y + 7
}
grad_f <- function(params) {
  x <- params[1]
  y <- params[2]
  grad_x <- 6 * x - 4 * y + 6
  grad_y <- 4 * y - 4 * x - 5
  return(c(grad_x, grad_y))
}
initial_guess <- c(0, 0)
result <- optim(par = initial_guess, fn = f_obj, gr = grad_f, method = "BFGS")
cat("Optimal solution: " , result$par, "\n")
```

```
## Optimal solution:  -0.5 0.75
```

```r
cat("Minimum value of the function: ", result$value, "\n")
```

```
## Minimum value of the function:  3.625
```

Mathematic interpretation

First compute the partial derivatives of the function with respect to x and y:

$$\frac{\partial f}{\partial x} = 6x - 4y + 6$$

$$\frac{\partial f}{\partial y} = 4y - 4x - 5$$

Then to find the critical points, we set the partial derivatives equal to zero and solve the system of equations:

$$\begin{aligned} 6x - 4y + 6 = 0 \quad (1) \\ 4y - 4x - 5 = 0 \quad (2) \end{aligned}$$

From equation (2), we get:

$$y = x + \frac{5}{4}$$

Substituting into equation (1):

$$2x + 1 = 0$$

$$x = -\frac{1}{2}$$

Substituting $x = -\frac{1}{2}$ into the expression for y:

$$y = \frac{3}{4}$$

Final Solution: The critical point is $x = -\frac{1}{2}, y = \frac{3}{4}$.

d) [10 Points] Check the second-order condition to verify that the solution you obtained in the previous step is indeed a minimum.

To verify that the critical point $x = -\frac{1}{2}, y = \frac{3}{4}$ is a minimum, we need to check the Hessian matrix of the function at this point. The Hessian matrix is given by:

take the second partial derivatives of the function with respect to x and y:

1. $\frac{\partial^2 f}{\partial x^2} = 6$
2. $\frac{\partial^2 f}{\partial y^2} = 4$
3. $\frac{\partial^2 f}{\partial x \partial y} = -4$

The Hessian matrix H is:

$$H = \begin{pmatrix} 6 & -4 \\ -4 & 4 \end{pmatrix}$$

1. The determinant of the Hessian is $\det(H) = 8$, which is positive.
2. The eigenvalues of the Hessian are computed as $5 + \sqrt{17}$ and $5 - \sqrt{17}$, both of which are positive.

Conclusion: Since the eigenvalues of the Hessian are positive, the Hessian is positive definite, and therefore, the critical point $\left(-\frac{1}{2}, \frac{3}{4}\right)$ is a local minimum.

e) [5 Points] Use the `optim()` function to solve this optimization problem. Use `method = "BFGS"`. Set your own initial point. Report the solutions you obtained. Does different choices of the initial point lead to different solutions? Why?

```r
result1 <- optim(par = c(0, 0), fn = f_obj, method = "BFGS")
result2 <- optim(par = c(5, -5), fn = f_obj, method = "BFGS")

cat("Optimal solution with initial point (0, 0): ", result1$par, "\n")
```

```
## Optimal solution with initial point (0, 0):  -0.5 0.75
```

```r
cat("Minimum value of the function with initial point (0, 0): ", result1$value, "\n")
```

```
## Minimum value of the function with initial point (0, 0):  3.625
```

```r
cat("Optimal solution with initial point (5, -5):", result2$par, "\n")
```

```
## Optimal solution with initial point (5, -5): -0.5 0.75
```

```r
cat("Minimum value of the function with initial point (5, -5):", result2$value, "\n")
```

```
## Minimum value of the function with initial point (5, -5): 3.625
```

Different choices of the initial point lead to same solutions. Because the function have a unique global minimum and the BFGS method finds a local minimum closest to the starting point.