COMP2002

10774743

2024-06-21

PART 1 – MACHINE LEARNING

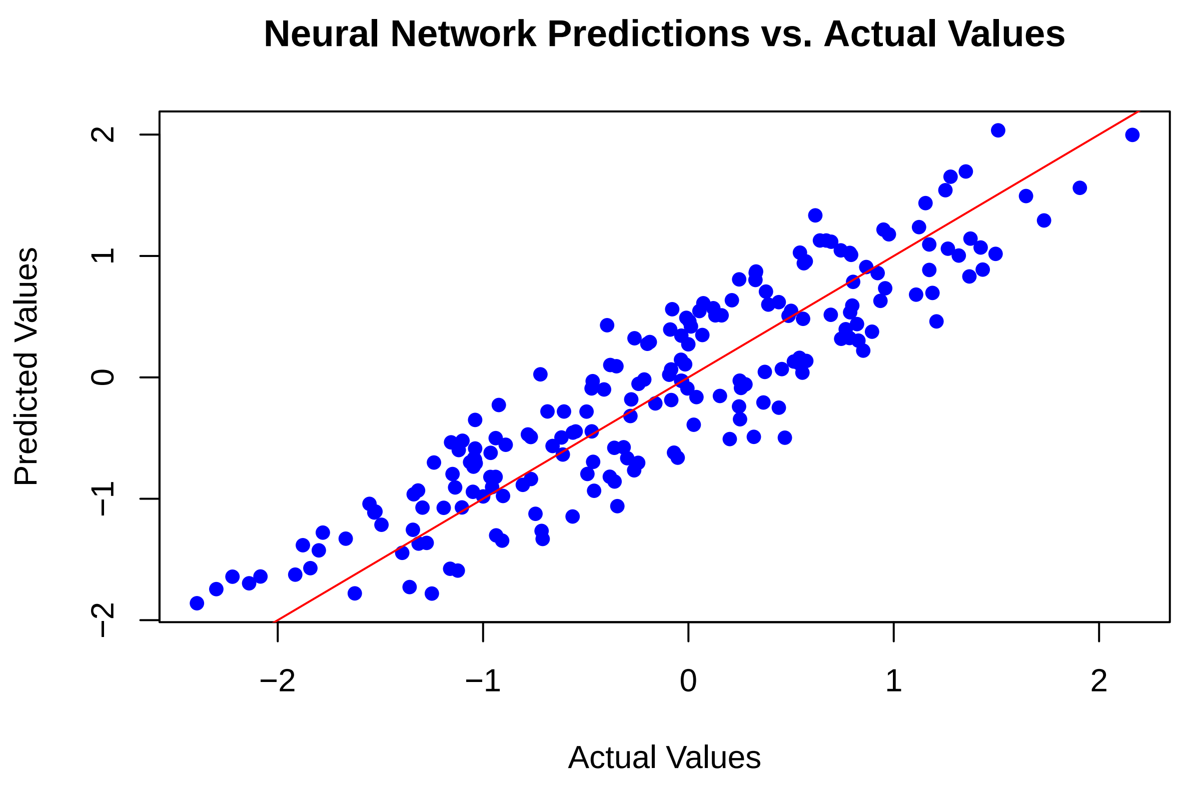
Task 1.1 – Data preparation (10% of total mark)

```r
# Load dataset from CSV file
insurance_data <- read.csv("C:/Repos/COMP2002-Report/data/insurance_data.csv")

# One-Hot Encoding for 'smoker' and 'gender' column

insurance_data$smoker <- ifelse(insurance_data$smoker == "yes", 1, 0)

insurance_data$gender <- ifelse(insurance_data$gender == "female", 1, 0)

# Label Encoding for other categorical columns

columns_to_encode <- c("region", "medical_history", "family_medical_history",
                       "exercise_frequency", "occupation", "coverage_level")

for (col in columns_to_encode) {
  insurance_data[[col]] <- as.numeric(factor(insurance_data[[col]], levels = unique(insurance_data[[col
}

# Identify numerical columns
numerical_columns <- sapply(insurance_data, is.numeric)

# Scale numerical columns
insurance_data_scaled <- as.data.frame(scale(insurance_data[numerical_columns]))

# Combine scaled numerical columns with categorical columns
insurance_data_scaled <- cbind(insurance_data_scaled, insurance_data[!numerical_columns])

# Set seed for reproducibility of randomness
set.seed(123)

# Sample a subset of the data for model training (e.g., 10% of the data)
sampled_indices <- sample(nrow(insurance_data_scaled), size = 0.001 * nrow(insurance_data_scaled))
sampled_data <- insurance_data_scaled[sampled_indices, ]

# Split the sampled dataset into training and testing sets (80% training, 20% testing)
split <- sample.split(sampled_data$charges, SplitRatio = 0.8)
training_set <- subset(sampled_data, split == TRUE)
```

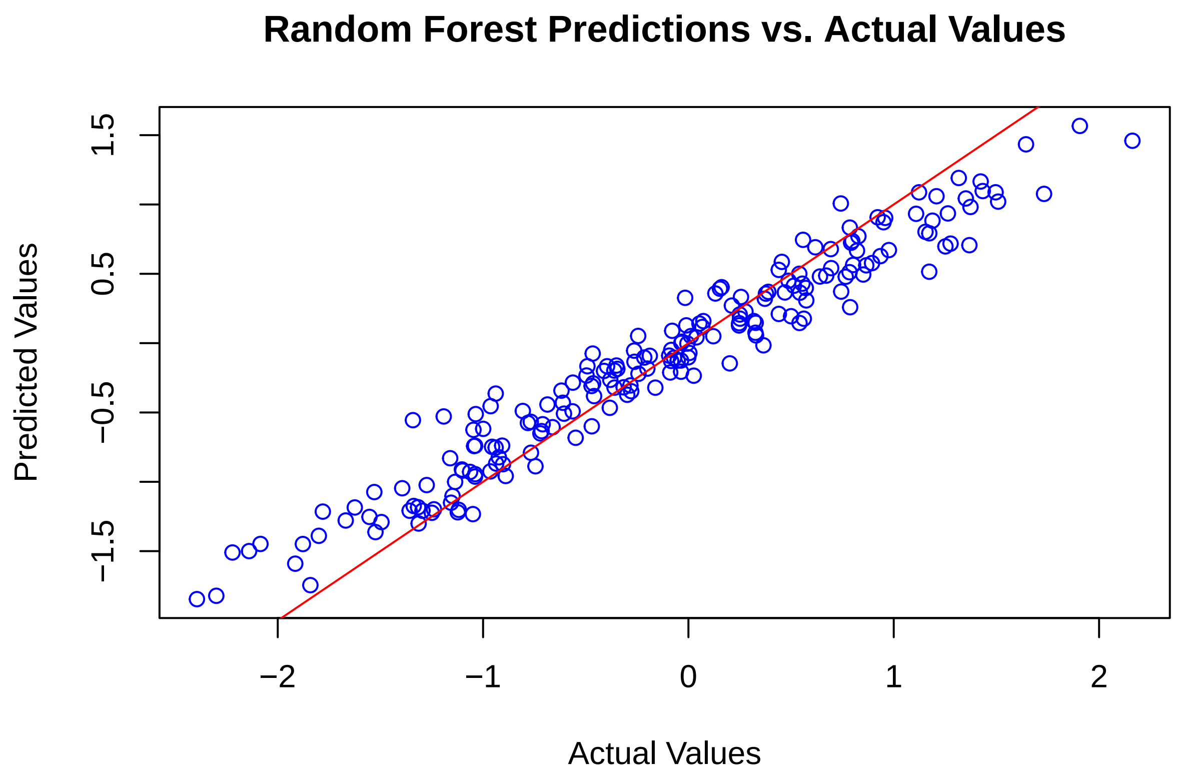
```r
testing_set <- subset(sampled_data, split == FALSE)

# Identify and separate target variable
training_target <- training_set$charges
testing_target <- testing_set$charges

# Include the target variable back into the training set
training_set$charges <- training_target

# Remove target variable from the testing set
testing_set <- subset(testing_set, select = -c(charges))
```

Task 1.2 – Regression (20% of total mark)

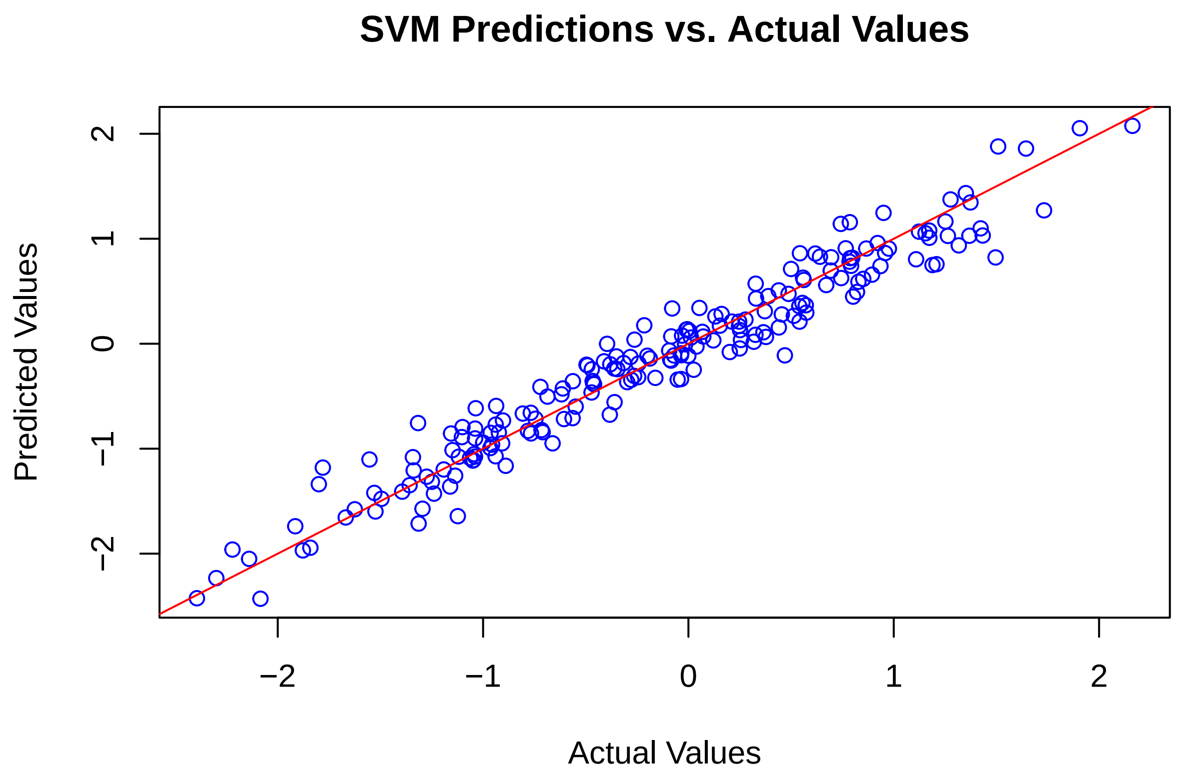
```r
# Train Neural Network model
nn_model <- neuralnet(charges ~ ., data = training_set)

# Make predictions on the testing set
nn_predictions <- predict(nn_model, newdata = testing_set)

# Plot Predictions vs. Actual Values for Neural Network
plot(testing_target, nn_predictions,
     main = "Neural Network Predictions vs. Actual Values",
     xlab = "Actual Values", ylab = "Predicted Values",
     col = "blue", pch = 16)
abline(0, 1, col = "red")
```



```r
# Train Random Forest model
rf_model <- randomForest(charges ~ ., data = training_set)

# Make predictions on the testing set
rf_predictions <- predict(rf_model, newdata = testing_set)

# Plot Predictions vs. Actual Values for Random Forest
plot(testing_target, rf_predictions, main = "Random Forest Predictions vs. Actual Values",
     xlab = "Actual Values", ylab = "Predicted Values", col = "blue")
abline(0, 1, col = "red")  # Add a line of perfect predictions
```



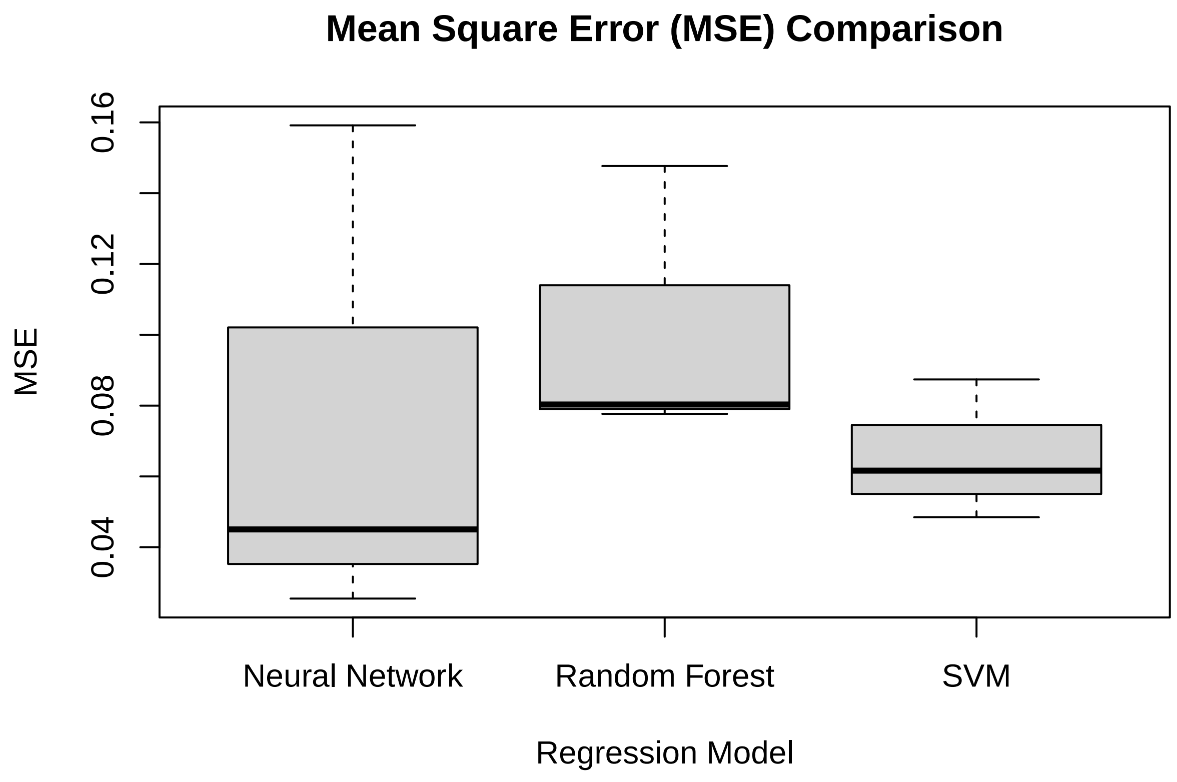
```
# Train SVM model
svm_model <- svm(charges ~ ., data = training_set)

# Make predictions on the testing set
svm_predictions <- predict(svm_model, newdata = testing_set)

# Plot Predictions vs. Actual Values for SVM
plot(testing_target, svm_predictions, main = "SVM Predictions vs. Actual Values",
     xlab = "Actual Values", ylab = "Predicted Values", col = "blue")
abline(0, 1, col = "red")  # Add a line of perfect predictions
```



### Task 1.3 – Assessment of regression (20% of total mark)

```
# Function to calculate MSE using cross-validation
calculate_mse <- function(model, data, folds = 3) {
  # Set up cross-validation
  ctrl <- trainControl(method = "cv", number = folds)

  # Train the model using cross-validation
  model_fit <- train(charges ~ ., data = data, method = model, trControl = ctrl)

  # Extract MSE values
  mse_values <- model_fit$results$RMSE^2

  return(mse_values)
}

# Calculate MSE for Neural Network using cross-validation
nn_mse <- calculate_mse("neuralnet", data = sampled_data)

# Calculate MSE for Random Forest using cross-validation
rf_mse <- calculate_mse("rf", data = sampled_data)

# Calculate MSE for SVM using cross-validation
svm_mse <- calculate_mse("svmRadial", data = sampled_data)

# Combine MSE values into a data frame
mse_data <- data.frame(Model = c(rep("Neural Network", length(nn_mse)),
```

```r
                                   rep("Random Forest", length(rf_mse)),
                                   rep("SVM", length(svm_mse))),
                        MSE = c(nn_mse, rf_mse, svm_mse))

# Create boxplot of MSE values
boxplot(MSE ~ Model, data = mse_data,
        main = "Mean Square Error (MSE) Comparison",
        xlab = "Regression Model", ylab = "MSE")
```



The box plots generated through cross-validation of 3 folds provide valuable insights into the performance of various regression models based on their mean square error (MSE) rates. Notably, we observe similar MSE average results across all models, but with a significant weight on the upper inter-quartile range (Q3) compared to the lower.

The dominance of the upper quartile suggests that a portion of the predictions made by each model tends to deviate significantly from the actual values, leading to higher error rates. This variability in prediction accuracy is crucial to consider when assessing the reliability of each model.

Upon analyzing the averages, it becomes evident that the Neural Network model exhibits the lowest average MSE at approximately 0.02. This indicates that, on average, the predictions generated by the Neural Network model are closer to the actual values compared to the other regression models. Conversely, the Random Forest model demonstrates the highest MSE around 0.07, suggesting a comparatively lower level of accuracy in its predictions. The Support Vector Machine model falls in between, with an average MSE of 0.05. The larger Q3 values for all three models imply a notable dispersion of prediction errors, particularly towards the higher end of the error spectrum. This indicates that while the models perform adequately on average, there are instances where their predictions deviate significantly from the actual values, leading to higher error rates.

In summary, the box plot analysis highlights the strengths and weaknesses of each regression model in terms of prediction accuracy. While the Neural Network model emerges as the most accurate on average, the presence of significant variability in prediction errors underscores the need for further scrutiny and refinement of all models to improve their predictive performance.

PART 2 – OPTIMISATION

Task 2.1 – Generation of random solutions (10% of total mark)

```r
# Read surgeon data from file with headers
surgeons_data <- read.table("C:/Repos/COMP2002-Report/data/Surgery.txt", sep = "|", header = FALSE,
                             col.names = c("surgery", "surgeon", "num_of_surgeries", "anaesthetist"))


# Define the calculate_fitness function
calculate_fitness <- function(timetable, surgeons_data) {
  # Initialize constraint violation counts
  concurrence_violations <- 0
  precedence_violations <- 0

  # Vector to track the number of surgeries performed by each surgeon
  surgeries_count <- rep(0, nrow(timetable))

  # Iterate through each time slot in the timetable
  for (time_slot in 1:ncol(timetable)) {
    # Get the surgeries scheduled in the current time slot
    surgeries <- timetable[, time_slot]

    # Create lists to store scheduled surgeons and anaesthetists
    scheduled_surgeons <- c()
    scheduled_anaesthetists <- c()

    # Check for concurrence and precedence violations
    for (i in 1:length(surgeries)) {
      # Get the surgeon, surgery, and anaesthetist details
      surgeon_surgery <- unlist(strsplit(as.character(surgeries[i]), "\\|"))
      surgeon <- surgeon_surgery[2]
      anaesthetist <- surgeon_surgery[3]  # Anaesthetist info is now part of the surgery string

      # Check for missing values
      if (is.na(surgeon)) {
        next  # Skip to the next iteration if surgeon is missing
      }

      # Check for concurrence violation
      if (surgeon %in% scheduled_surgeons) {
        concurrence_violations <- concurrence_violations + 1
      } else {
        # Add the surgeon to the list of scheduled surgeons
        scheduled_surgeons <- c(scheduled_surgeons, surgeon)
      }
      # Check for precedence violation
      if (i > 1) {
        previous_surgeon_surgery <- unlist(strsplit(as.character(surgeries[i - 1]), "\\|"))
        previous_surgeon <- previous_surgeon_surgery[2]
        # Check for missing values
        if (!is.na(surgeon) & !is.na(previous_surgeon) & surgeon == previous_surgeon) {
          precedence_violations <- precedence_violations + 1
        }
        # Check for precedence violation with surgeries in between
        if (i > 2) {
          two_surgeries_ago_surgeon_surgery <- unlist(strsplit(as.character(surgeries[i - 2]), "\\|"))
          two_surgeries_ago_surgeon <- two_surgeries_ago_surgeon_surgery[2]
          # Check for missing values
          if (!is.na(surgeon) & !is.na(two_surgeries_ago_surgeon) & surgeon == two_surgeries_ago_surgeo
            precedence_violations <- precedence_violations + 1
          }
        }
      }

      # Check for anaesthetist violation
      if (!is.na(anaesthetist) && anaesthetist == "Yes") {
        if (anaesthetist %in% scheduled_anaesthetists) {
          concurrence_violations <- concurrence_violations + 1
        } else {
          scheduled_anaesthetists <- c(scheduled_anaesthetists, anaesthetist)
        }
      }

      # Update the count of surgeries performed by the surgeon
      surgeries_count[i] <- surgeries_count[i] + 1

      # Check if the surgeon has exceeded the maximum number of surgeries allowed
      if (surgeries_count[i] > 1) {  # Changed to check for more than 1 surgery per surgeon
        precedence_violations <- precedence_violations + 1
      }
    }
  }

  # Calculate the fitness score based on the product of constraint violations
  fitness_score <- concurrence_violations * precedence_violations

  return(fitness_score)
}

# Define the number of time slots and theatres
num_slots <- 9
num_theatres <- 3

# Generate a random timetable matrix
random_timetable <- matrix("", nrow = num_slots, ncol = num_theatres)

# Define the surgeons available
surgeons <- surgeons_data$surgeon
```

```r
# Fill in the timetable with random surgeries
for (time_slot in 1:num_slots) {
  for (theatre in 1:num_theatres) {
    # Randomly select a surgeon
    surgeon <- sample(surgeons, 1)

    # Select the surgery type based on the surgeon
    surgery_row <- surgeons_data[surgeons_data$surgeon == surgeon, ]
    surgery <- surgery_row$surgery

    # Combine surgeon and surgery information
    surgery_info <- paste(surgery, surgeon, sep = "|")

    # Assign the surgery to the timetable
    random_timetable[time_slot, theatre] <- surgery_info
  }
}

# Print the random timetable
print(random_timetable)
```

```
##       [,1]
##  [1,] "Dupuytren Contracture Release|Beverly Crusher"
##  [2,] "Carpal Tunnel|Cristina Yang"
##  [3,] "Broken Bone repair|Leonard McCoy"
##  [4,] "Carpal Tunnel|Cristina Yang"
##  [5,] "Carpal Tunnel|Cristina Yang"
##  [6,] "Dupuytren Contracture Release|Beverly Crusher"
##  [7,] "Cholecystectomy|Meredith Gery"
##  [8,] "Broken Bone repair|Leonard McCoy"
##  [9,] "Dupuytren Contracture Release|Beverly Crusher"
##       [,2]
##  [1,] "Cholecystectomy|Meredith Gery"
##  [2,] "Carpal Tunnel|Cristina Yang"
##  [3,] "Cholecystectomy|Meredith Gery"
##  [4,] "Broken Bone repair|Leonard McCoy"
##  [5,] "Cholecystectomy|Meredith Gery"
##  [6,] "Heart Bypass|Preston Burke"
##  [7,] "Cholecystectomy|Meredith Gery"
##  [8,] "Carpal Tunnel|Cristina Yang"
##  [9,] "Cholecystectomy|Meredith Gery"
##       [,3]
##  [1,] "Heart Bypass|Preston Burke"
##  [2,] "Broken Bone repair|Leonard McCoy"
##  [3,] "Dupuytren Contracture Release|Beverly Crusher"
##  [4,] "Broken Bone repair|Leonard McCoy"
##  [5,] "Heart Bypass|Preston Burke"
##  [6,] "Carpal Tunnel|Cristina Yang"
##  [7,] "Carpal Tunnel|Cristina Yang"
##  [8,] "Heart Bypass|Preston Burke"
##  [9,] "Cholecystectomy|Meredith Gery"
```

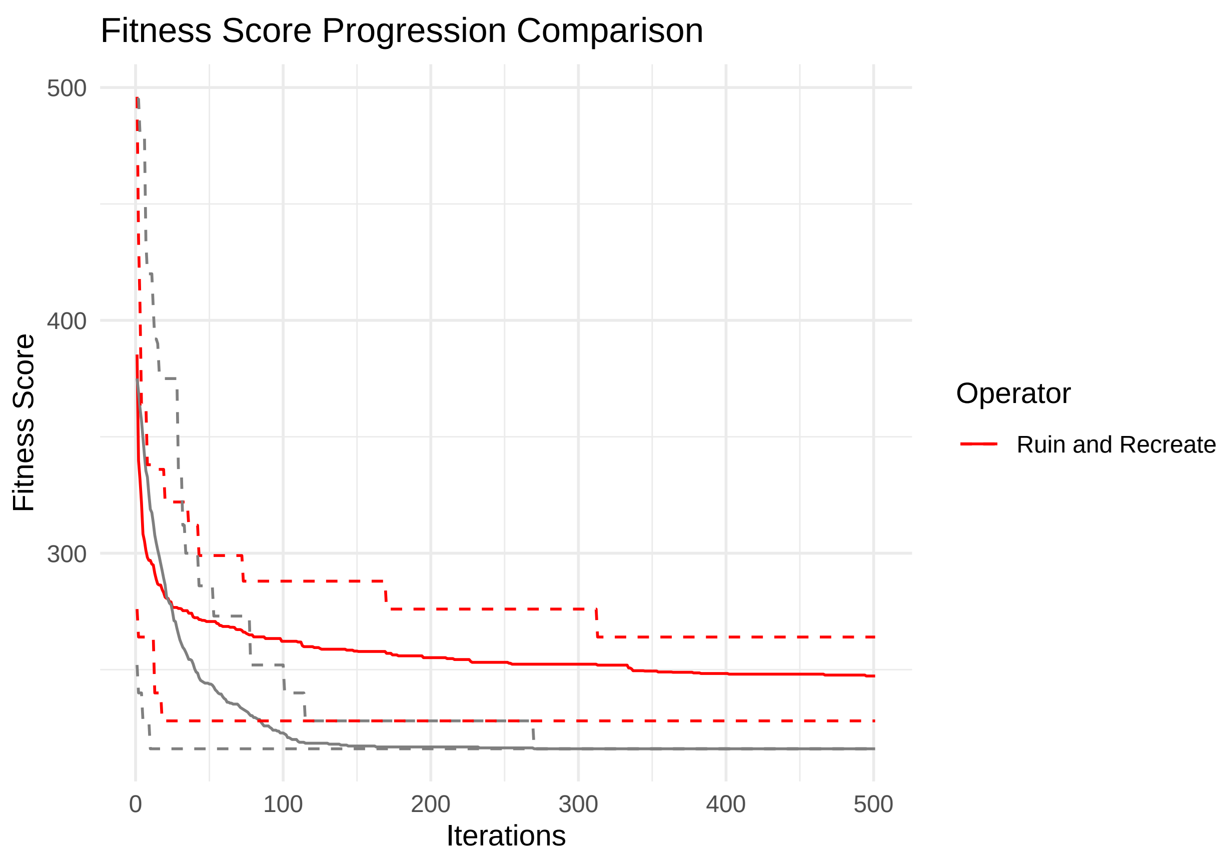
```r
# Calculate the fitness of the random timetable
fitness <- calculate_fitness(random_timetable, surgeons_data)
```

Task 2.2 – Algorithm implementation (25% of total mark)

```r
# Surgery replace operator mutation function (swap operator)
surgery_replace_operator <- function(timetable, surgeons_data) {
  num_slots <- nrow(timetable)
  num_theatres <- ncol(timetable)

  new_timetable <- timetable

  # Randomly select a time slot and theatre to replace a surgery
  time_slot <- sample(1:num_slots, 1)
  theatre <- sample(1:num_theatres, 1)

  # Randomly select a surgeon
  surgeon <- sample(surgeons_data$surgeon, 1)

  # Select the surgery type based on the surgeon
  surgery_row <- surgeons_data[surgeons_data$surgeon == surgeon, ]
  surgery <- surgery_row$surgery

  # Combine surgeon and surgery information
  surgery_info <- paste(surgery, surgeon, sep = "|")

  # Assign the surgery to the specified time slot and theatre
  new_timetable[time_slot, theatre] <- surgery_info

  return(new_timetable)
}

# Ruin_and_recreate_operator function
ruin_and_recreate_operator <- function(surgeons_data, num_slots = 9, num_theatres = 3) {
  new_timetable <- matrix("", nrow = num_slots, ncol = num_theatres)

  surgeons <- surgeons_data$surgeon

  # Fill in the timetable with random surgeries
  for (time_slot in 1:num_slots) {
    for (theatre in 1:num_theatres) {
      # Randomly select a surgeon
      surgeon <- sample(surgeons, 1)

      # Select the surgery type based on the surgeon
      surgery_row <- surgeons_data[surgeons_data$surgeon == surgeon, ]
      surgery <- surgery_row$surgery

      # Combine surgeon and surgery information
      surgery_info <- paste(surgery, surgeon, sep = "|")

      # Assign the surgery to the timetable
      new_timetable[time_slot, theatre] <- surgery_info
    }
  }

  return(new_timetable)
}

# Hillcliber function
hillclimber <- function(surgeons_data, num_iterations, mutation_operator) {
  # Initialize the best solution and its fitness
  best_solution <- ruin_and_recreate_operator(surgeons_data)
  best_fitness <- calculate_fitness(best_solution, surgeons_data)
  best_fitnesses <- c(best_fitness)

  # Perform hill climbing iterations
  for (iteration in 1:num_iterations) {
    # Generate a child solution using the specified mutation operator
    if (mutation_operator == "surgery_replace") {
      child_solution <- surgery_replace_operator(best_solution, surgeons_data)
    } else if (mutation_operator == "ruin_and_recreate") {
      child_solution <- ruin_and_recreate_operator(surgeons_data)
    }

    # Calculate fitness of the child solution
    child_fitness <- calculate_fitness(child_solution, surgeons_data)

    # Retain the best solution between parent and child
    if (child_fitness < best_fitness) {
      best_solution <- child_solution
      best_fitness <- child_fitness
    } else {
      child_solution <- best_solution  # Keep the parent solution if child is not better
    }


    # Add the best fitness to the list
    best_fitnesses <- c(best_fitnesses, best_fitness)
  }

  # Return the best solution and list of best fitnesses
  return(list(best_solution = best_solution, best_fitnesses = best_fitnesses))
}


# Function to run the hill climber algorithm
run_hill_climber <- function(surgeons_data, num_iterations, num_runs) {
  # Initialize lists to store fitness lists for each operator
  fitness_lists_surgery_replace <- list()
  fitness_lists_ruin_and_recreate <- list()

  # Run the hill climber for each mutation operator
  for (run in 1:num_runs) {
    # Run hill climber with surgery replace operator
    result_surgery_replace <- hillclimber(surgeons_data, num_iterations, "surgery_replace")
    fitness_lists_surgery_replace[[run]] <- result_surgery_replace$best_fitnesses

    # Run hill climber with ruin and recreate operator
    result_ruin_and_recreate <- hillclimber(surgeons_data, num_iterations, "ruin_and_recreate")
    fitness_lists_ruin_and_recreate[[run]] <- result_ruin_and_recreate$best_fitnesses
  }

  # Return fitness lists for both mutation operators
  return(list(
    fitness_lists_surgery_replace = fitness_lists_surgery_replace,
    fitness_lists_ruin_and_recreate = fitness_lists_ruin_and_recreate
  ))
}
```

Task 2.3 – Visualisation of results (15% of total mark)

```r
plot_fitness_combined <- function(fitness_lists_surgery_replace, fitness_lists_ruin_and_recreate) {
  # Calculate the number of iterations
  num_iterations <- length(fitness_lists_surgery_replace[[1]])

  # Initialize lists to store average, max, and min fitness values for both operators
  average_fitness_surgery_replace <- rep(0, num_iterations)
  max_fitness_surgery_replace <- rep(0, num_iterations)
  min_fitness_surgery_replace <- rep(0, num_iterations)

  average_fitness_ruin_and_recreate <- rep(0, num_iterations)
  max_fitness_ruin_and_recreate <- rep(0, num_iterations)
  min_fitness_ruin_and_recreate <- rep(0, num_iterations)

  # Iterate over each iteration
  for (i in 1:num_iterations) {
    # Extract fitness values for the current iteration from all runs for surgery replace operator
    iteration_fitness_surgery_replace <- sapply(fitness_lists_surgery_replace, function(fitness_list) fi

    # Calculate average, max, and min fitness for the current iteration for surgery replace operator
    average_fitness_surgery_replace[i] <- mean(iteration_fitness_surgery_replace)
    max_fitness_surgery_replace[i] <- max(iteration_fitness_surgery_replace)
    min_fitness_surgery_replace[i] <- min(iteration_fitness_surgery_replace)

    # Extract fitness values for the current iteration from all runs for ruin and recreate operator
    iteration_fitness_ruin_and_recreate <- sapply(fitness_lists_ruin_and_recreate, function(fitness_lis

    # Calculate average, max, and min fitness for the current iteration for ruin and recreate operator
    average_fitness_ruin_and_recreate[i] <- mean(iteration_fitness_ruin_and_recreate)
    max_fitness_ruin_and_recreate[i] <- max(iteration_fitness_ruin_and_recreate)
    min_fitness_ruin_and_recreate[i] <- min(iteration_fitness_ruin_and_recreate)
  }

  # Create data frames to store the results for both operators
  results_surgery_replace <- data.frame(iterations = 1:num_iterations,
                                        average_fitness = average_fitness_surgery_replace,
                                          max_fitness = max_fitness_surgery_replace,
                                          min_fitness = min_fitness_surgery_replace,
                                          operator = "Surgery Replace")

  results_ruin_and_recreate <- data.frame(iterations = 1:num_iterations,
                                             average_fitness = average_fitness_ruin_and_recreate,
                                             max_fitness = max_fitness_ruin_and_recreate,
                                             min_fitness = min_fitness_ruin_and_recreate,
                                             operator = "Ruin and Recreate")

  # Combine results from both operators
  combined_results <- rbind(results_surgery_replace, results_ruin_and_recreate)

  # Plot the combined results
  ggplot(combined_results, aes(x = iterations, y = average_fitness, color = operator)) +
    geom_line() +
    geom_line(aes(y = max_fitness), linetype = "dashed") +
    geom_line(aes(y = min_fitness), linetype = "dashed") +
    labs(title = "Fitness Score Progression Comparison",
         x = "Iterations",
         y = "Fitness Score",
         color = "Operator") +
    scale_color_manual(values = c("Swap Operator" = "blue", "Ruin and Recreate" = "red")) +
    theme_minimal()
}

# Function to run the hill climber algorithm
results <- run_hill_climber(surgeons_data, num_iterations = 500, num_runs = 30)

# Plot fitness progression for both mutation operators combined
plot_fitness_combined(results$fitness_lists_surgery_replace, results$fitness_lists_ruin_and_recreate)
```



The final graph illustrates the fitness scores obtained by the mutation operators, namely the swap operator and the ruin-and-recreate operator, over 500 iterations. The lower fitness score indicates better compliance with the constraints of concurrence and precedence.

Upon analysis, it is evident that the swap operator consistently outperforms the ruin-and-recreate operator. The swap operator stabilizes at a minimum fitness score of 216 around the 200th iteration, indicating a robust convergence to a highly optimized timetable. On the other hand, the ruin-and-recreate operator achieves only a minimum fitness score of 228, demonstrating a comparatively lesser degree of optimization.

This preference for the swap operator is further supported by the average fitness scores obtained. The swap operator maintains an average fitness score of 216, matching its minimum value, signifying consistent performance throughout the iterations. In contrast, the ruin-and-recreate operator exhibits an average fitness score of 251.23, significantly higher than the swap operator, indicating a less effective optimization strategy.

In conclusion, the visual comparison of fitness progression clearly highlights the swap operator as the preferred method for optimizing the timetable. Its superior convergence to lower fitness scores and consistent performance across iterations make it the more effective choice for minimizing violations of concurrence and precedence constraints.