**Name:** [**Muhammad Ahad**](mailto:muhammadahadf23@nutech.edu.pk) **Imran**

**ID: F23607034**

**PFAI ESE**

# QUESTION # 02

# Part1 & 2: R Language

## 1)

**Using mtcars dataset in R:**

* **Group data by number of gears**
* **For each group, calculate the average mpg and hp**
* **Create a grouped bar plot comparing both averages using Include code and output.**

**# Load required library**

**library(ggplot2)**

**library(dplyr)**

**# Group by gear and calculate average mpg and hp**

**mtcars\_grouped <- mtcars %>%**

**group\_by(gear) %>%**

**summarise(avg\_mpg = mean(mpg), avg\_hp = mean(hp))**

**# Reshape data for plotting**

**mtcars\_long <- tidyr::pivot\_longer(mtcars\_grouped,**

**cols = c(avg\_mpg, avg\_hp),**

**names\_to = "metric",**

**values\_to = "value")**

**# Create grouped bar plot**

**ggplot(mtcars\_long, aes(x = factor(gear), y = value, fill = metric)) +**

**geom\_bar(stat = "identity", position = position\_dodge(width = 0.8)) +**

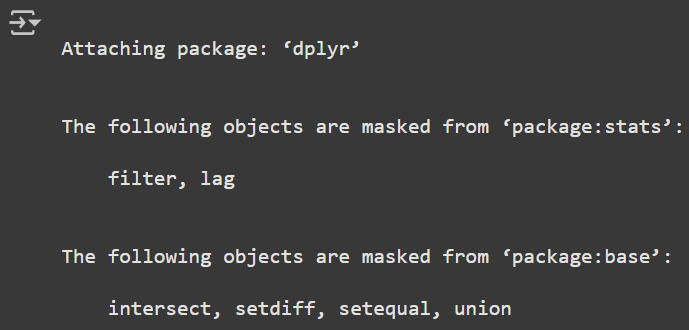
**labs(title = "Average MPG and HP by Number of Gears",**

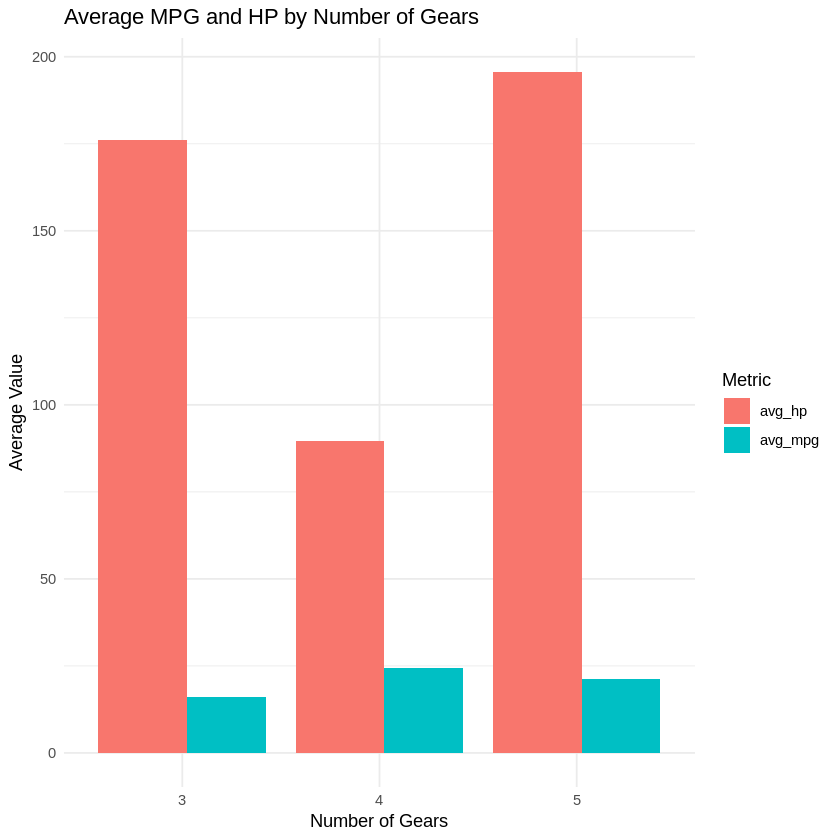
**x = "Number of Gears",**

**y = "Average Value",**

**fill = "Metric") +**

**theme\_minimal()**

****

****

## 2)

**Write a R script that:**

* **Splits mtcars dataset in train/test sets**
* **Scales the data**
* **Trains a Random Forest to predict mpg**
* **Evaluates performance using RMSE and R-squared**
* **Add comments explaining each each step**

# Load necessary libraries

install.packages("caret")

install.packages("randomForest")

library(caret)

library(randomForest)

# Set random seed for reproducibility

set.seed(123)

# Split the data into training and testing sets (80% train, 20% test)

train\_index <- createDataPartition(mtcars$mpg, p = 0.8, list = FALSE)

train\_data <- mtcars[train\_index, ]

test\_data <- mtcars[-train\_index, ]

# Separate features (X) and target variable (y) for training and testing sets

train\_x <- train\_data[, -which(names(train\_data) == "mpg")]

train\_y <- train\_data$mpg

test\_x <- test\_data[, -which(names(test\_data) == "mpg")]

test\_y <- test\_data$mpg

# Scale the numerical features using preProcess

# This is important for many machine learning algorithms, though less critical for Random Forest

# We will fit the scaler on the training data and apply it to both training and testing data

preproc\_param <- preProcess(train\_x, method = c("center", "scale"))

train\_x\_scaled <- predict(preproc\_param, train\_x)

test\_x\_scaled <- predict(preproc\_param, test\_x)

# Combine scaled features and target variable back into data frames for training and testing

train\_scaled <- cbind(train\_x\_scaled, mpg = train\_y)

test\_scaled <- cbind(test\_x\_scaled, mpg = test\_y)

# Train a Random Forest model to predict mpg

# The formula specifies mpg as the target and all other columns as predictors

# ntree specifies the number of trees in the forest

# mtry specifies the number of variables randomly sampled at each split

rf\_model <- randomForest(mpg ~ ., data = train\_scaled, ntree = 500, mtry = floor(ncol(train\_scaled)/3))

# Make predictions on the scaled test data

predictions <- predict(rf\_model, test\_scaled)

# Evaluate the model performance

# Calculate Root Mean Squared Error (RMSE)

# RMSE measures the standard deviation of the residuals (prediction errors)

rmse <- sqrt(mean((test\_scaled$mpg - predictions)^2))

cat("RMSE:", rmse, "\n")

# Calculate R-squared

# R-squared represents the proportion of the variance for a dependent variable that's explained by the independent variables in a regression model.

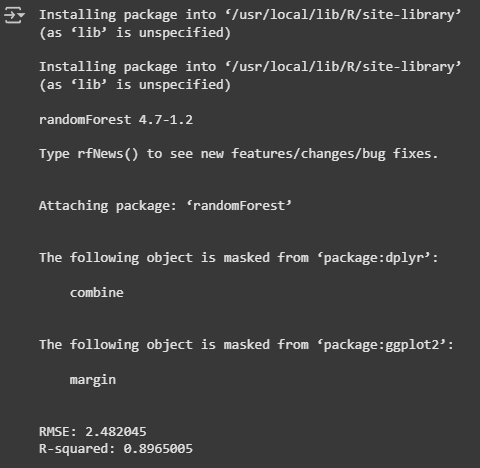
# It ranges from 0 to 1, where 1 indicates a perfect fit.

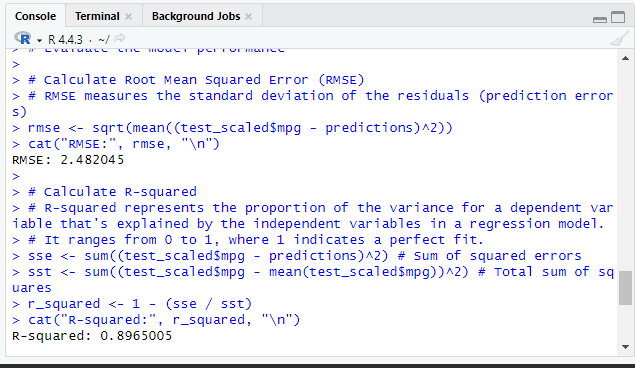
sse <- sum((test\_scaled$mpg - predictions)^2) # Sum of squared errors

sst <- sum((test\_scaled$mpg - mean(test\_scaled$mpg))^2) # Total sum of squares

r\_squared <- 1 - (sse / sst)

cat("R-squared:", r\_squared, "\n")

****

****

# Part 3: Python Language

**Given a dataset with missing values and inconsistent formatting, write a python script using the pandas library to clean the data. Your cscript should handle missing values using proper imputation techniques and standardize categorical variables for consistency.**

**Using the matplotlib and seaborn libraries, generate meaningful visualizations for a dataset containing multiple numeric variables. Create a heatmap to illustrate correlations, and use histograms or box plots to show the distribution of key feature.**

**import os**

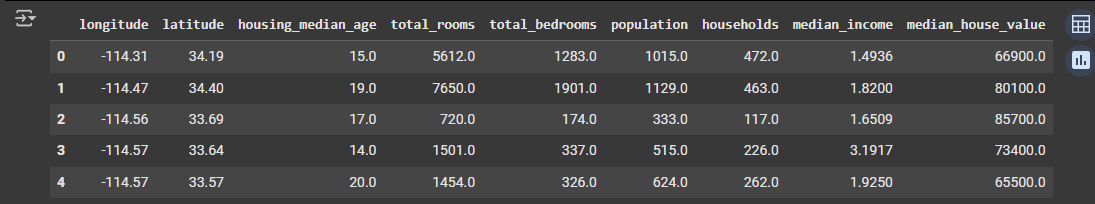
**print(os.listdir('sample\_data'))**

****

**import pandas as pd**

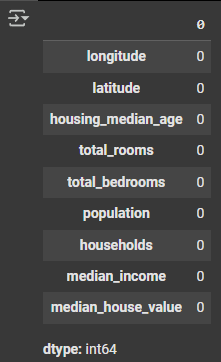
**df = pd.read\_csv('sample\_data/california\_housing\_train.csv')**

**display(df.head())**

****

## Handle missing values

**display(df.isnull().sum())**

****

## Standardize categorical variables

**categorical\_cols = df.select\_dtypes(include=['object', 'category']).columns**

**print("Categorical columns:", categorical\_cols)**

**for col in categorical\_cols:**

**print(f"\nUnique values in '{col}':")**

**display(df[col].value\_counts())**

****

## Visualize correlations

**correlation\_matrix = df.corr()**

**import seaborn as sns**

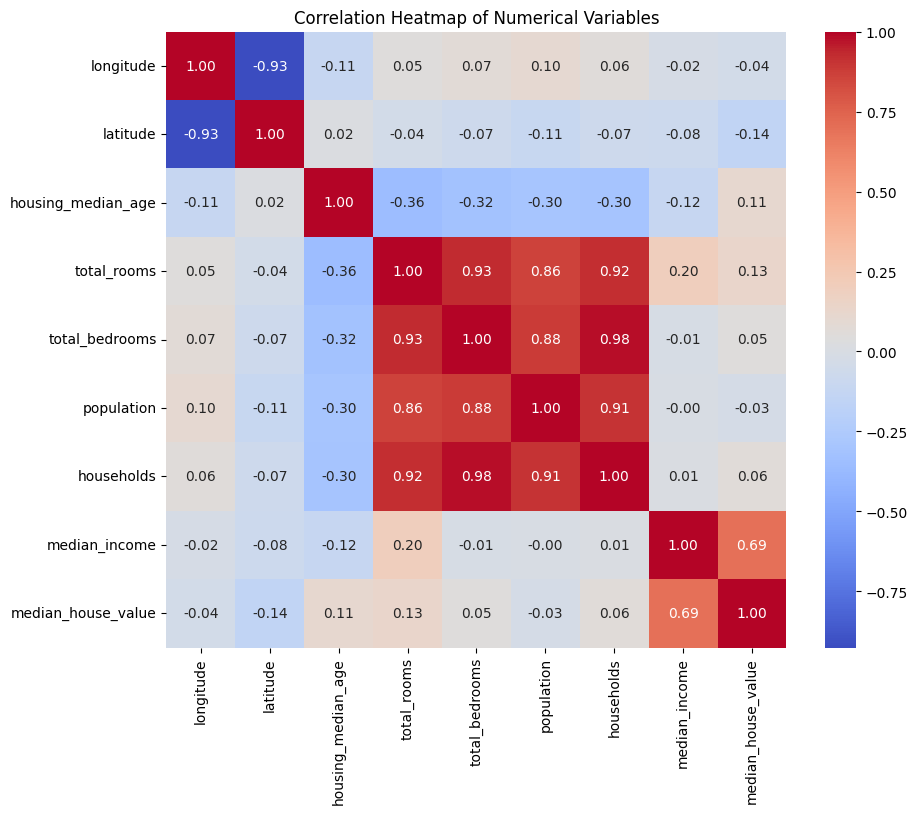
**import matplotlib.pyplot as plt**

**plt.figure(figsize=(10, 8))**

**sns.heatmap(correlation\_matrix, annot=True, cmap='coolwarm', fmt=".2f")**

**plt.title('Correlation Heatmap of Numerical Variables')**

**plt.show()**

****

## Visualize distributions

**import matplotlib.pyplot as plt**

**import seaborn as sns**

**key\_numerical\_cols = ['median\_income', 'median\_house\_value', 'housing\_median\_age', 'total\_rooms', 'population']**

**plt.figure(figsize=(15, 10))**

**for i, col in enumerate(key\_numerical\_cols):**

**plt.subplot(2, 3, i + 1)**

**sns.histplot(df[col], kde=True)**

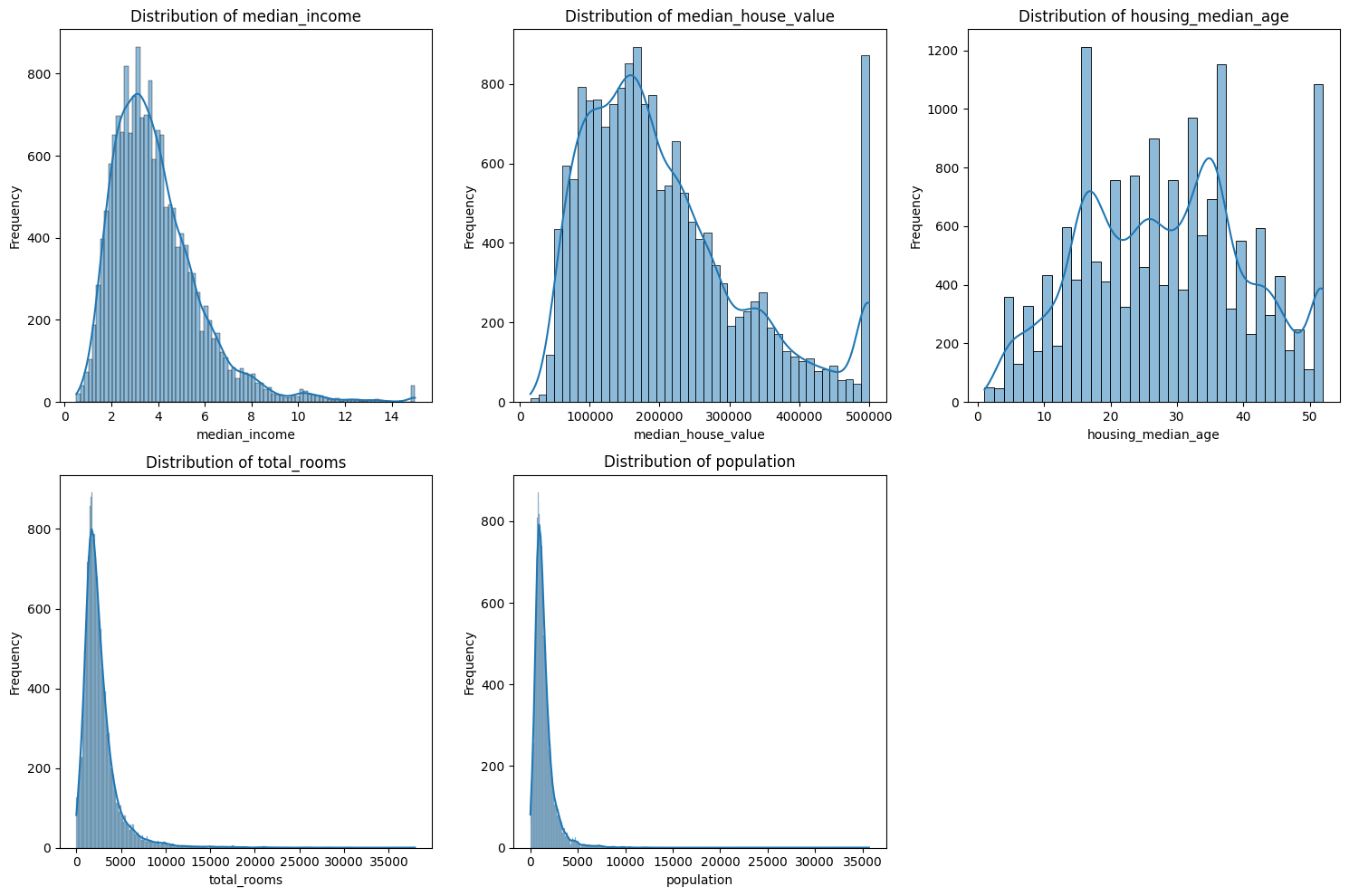
**plt.title(f'Distribution of {col}')**

**plt.xlabel(col)**

**plt.ylabel('Frequency')**

**plt.tight\_layout()**

**plt.show()**

****

# Question # 01

# Part 1: GO LANGUAGE:

## 1)

package main

import (

"fmt"

)

// BankAccount struct

type BankAccount struct {

Owner string

Balance float64

}

// Deposit method: adds amount to balance

func (b \*BankAccount) Deposit(amount float64) {

b.Balance += amount

fmt.Printf("%s deposited: $%.2f\n", b.Owner, amount)

}

// Withdraw method: subtracts if enough balance

func (b \*BankAccount) Withdraw(amount float64) {

if amount > b.Balance {

fmt.Printf(" Withdrawal of $%.2f failed: Insufficient funds!\n", amount)

} else {

b.Balance -= amount

fmt.Printf(" %s withdrew: $%.2f\n", b.Owner, amount)

}

}

// BalanceInquiry method: shows current balance

func (b BankAccount) BalanceInquiry() {

fmt.Printf(" Current balance for %s: $%.2f\n", b.Owner, b.Balance)

}

func main() {

// Create a bank account

account := BankAccount{Owner: "Ahad"}

// 3 demo transactions

account.Deposit(1000)

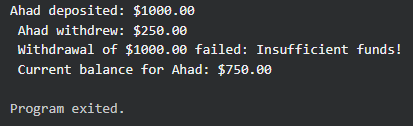
account.Withdraw(250)

account.Withdraw(1000) // This should fail

// Final balance check

account.BalanceInquiry()

}

****

## 2)

package main

import (

"net/http"

"github.com/gin-gonic/gin"

)

// Define a Student struct

type Student struct {

Name string `json:"name"`

Age int `json:"age"`

GPA float64 `json:"gpa"`

}

// Main function

func main() {

// Initialize Gin router

router := gin.Default()

// Create GET /students endpoint

router.GET("/students", func(c \*gin.Context) {

students := []Student{

{Name: "Ahad", Age: 20, GPA: 4.0},

{Name: "Sara", Age: 34, GPA: 3.8},

{Name: "Basit", Age: 45, GPA: 3.9},

}

// Return JSON with status 200

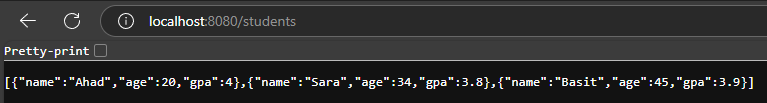
c.JSON(http.StatusOK, students)

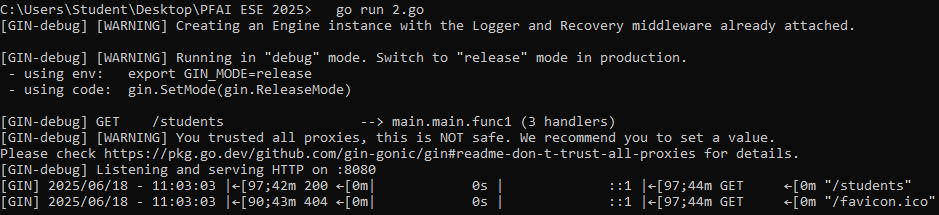
})

// Start the server on port 8080

router.Run(":8080")

}



****

## 3)

package main

import (

"fmt"

"sync"

)

// Define the worker function

func worker(id int, tasks <-chan int, wg \*sync.WaitGroup) {

defer wg.Done() // Called once when the worker finishes

for task := range tasks {

result := task \* task

fmt.Printf("Worker %d processed task %d (result: %d)\n", id, task, result)

}

}

func main() {

var wg sync.WaitGroup

tasks := make(chan int, 6) // Buffered channel with 6 tasks

// Start 3 workers

for w := 1; w <= 3; w++ {

wg.Add(1)

go worker(w, tasks, &wg)

}

// Send 6 tasks (numbers to square)

for i := 1; i <= 6; i++ {

tasks <- i

}

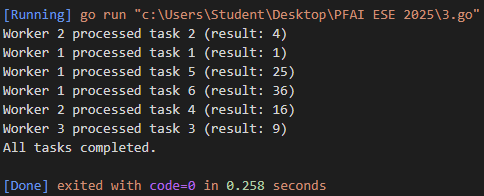
close(tasks) // Close the channel so workers stop when all tasks are read

// Wait for all workers to finish

wg.Wait()

fmt.Println("All tasks completed.")

}

****

# Part 2: JULIA

* **Create a simple neural network in julia using** [**Flux.jl**](http://flux.jl) **that learns to predict output y = 2x from inputs [1.0, 2.0 3.0]**
* **Include model definition, training, and display learned weight.**

**using Flux**

**# Define input and target data**

**X = [1.0, 2.0, 3.0] # Input data**

**y = [2.0, 4.0, 6.0] # Target outputs (y = 2x)**

**# Reshape data for Flux (each column is a sample)**

**X\_train = reshape(X, 1, :) # 1×3 matrix**

**y\_train = reshape(y, 1, :) # 1×3 matrix**

**# Define a simple linear model (single neuron, no bias for cleaner y=2x learning)**

**model = Dense(1 => 1, identity; bias=false)**

**# Define loss function that takes model as parameter**

**function loss\_fn(m)**

**return Flux.mse(m(X\_train), y\_train)**

**end**

**# Define optimizer with higher learning rate**

**opt = Flux.setup(Adam(0.1), model)**

**# Training parameters**

**epochs = 1000**

**# Training loop**

**println("Training the neural network...")**

**println("Initial weight: ", model.weight[1])**

**for epoch in 1:epochs**

**# Compute gradients - pass the loss function and model**

**grads = Flux.gradient(loss\_fn, model)**

**# Update parameters**

**Flux.update!(opt, model, grads[1])**

**# Print progress every 100 epochs**

**if epoch % 100 == 0**

**current\_loss = loss\_fn(model)**

**println("Epoch $epoch: Loss = $(round(current\_loss, digits=6)), Weight = $(round(model.weight[1], digits=4))")**

**end**

**end**

**# Display final results**

**println("\n" \* "="^50)**

**println("Training completed!")**

**println("Final learned weight: ", round(model.weight[1], digits=4))**

**println("Target weight: 2.0")**

**println("Final loss: ", round(loss\_fn(model), digits=6))**

**# Test the model**

**println("\nTesting the model:")**

**for i in 1:length(X)**

**input\_val = X[i]**

**predicted = model([input\_val])[1]**

**actual = y[i]**

**println("Input: $input\_val, Predicted: $(round(predicted, digits=4)), Actual: $actual")**

**end**

**# Verify with new data point**

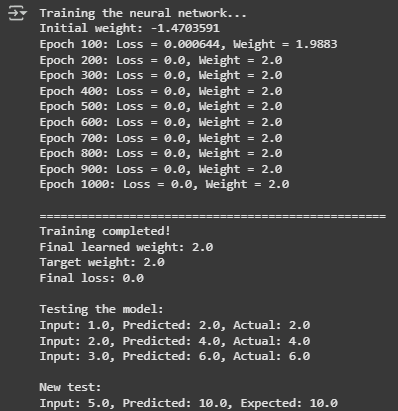
**test\_input = 5.0**

**test\_prediction = model([test\_input])[1]**

**expected\_output = 2.0 \* test\_input**

**println("\nNew test:")**

**println("Input: $test\_input, Predicted: $(round(test\_prediction, digits=4)), Expected: $expected\_output")**

****