Name: Muhammad AhadImran
ID: F23607034

PFAI ESE

QUESTION # 02

Part1 & 2: R Language

1)

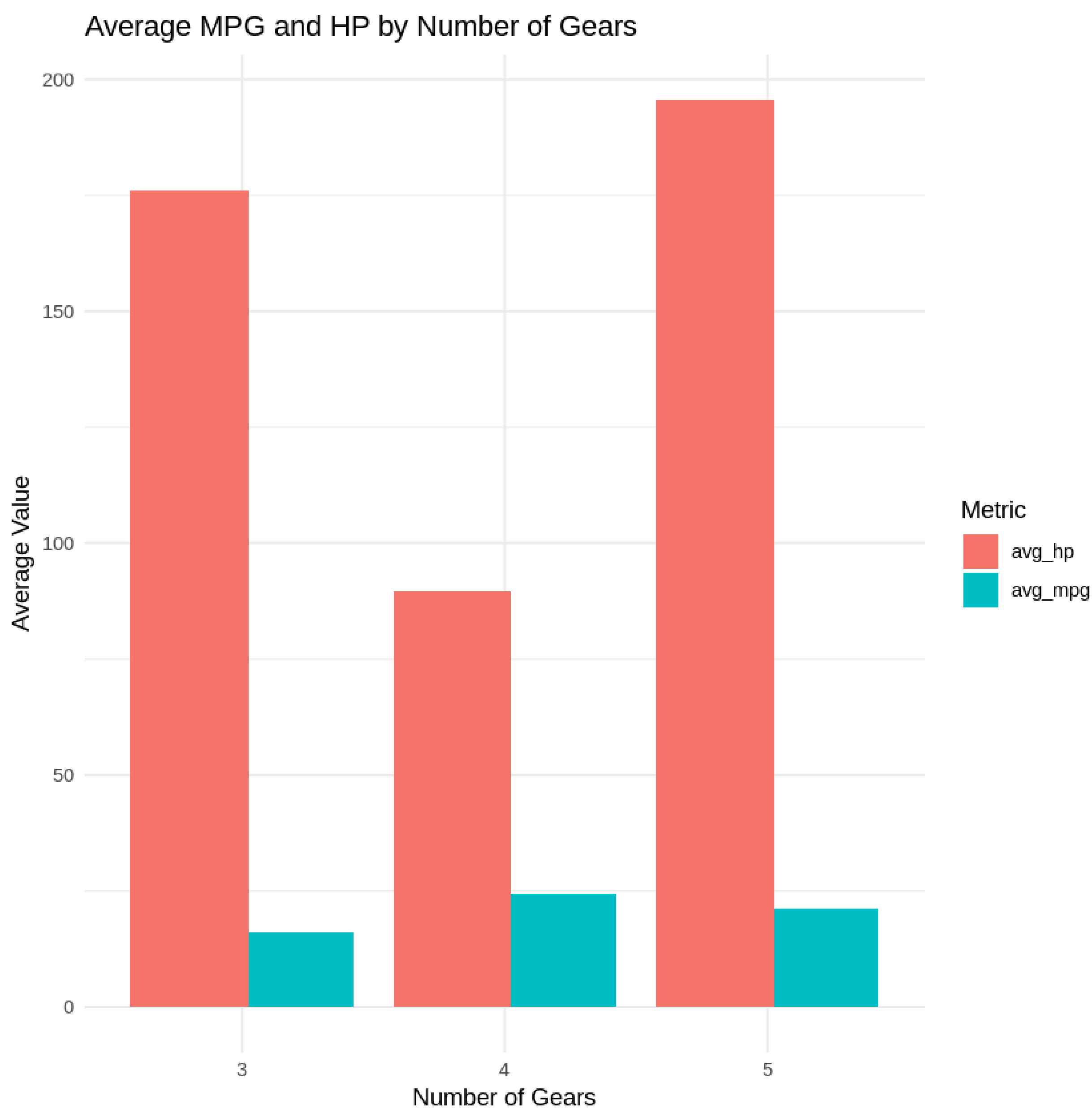
Using `mtcars` dataset in R:

- **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)) +
```

```r
labs(title = "Average MPG and HP by Number of Gears",
     x = "Number of Gears",
     y = "Average Value",
     fill = "Metric") +
theme_minimal()
```

```
Attaching package: ‘dplyr’


The following objects are masked from ‘package:stats’:

    filter, lag


The following objects are masked from ‘package:base’:

    intersect, setdiff, setequal, union
```



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**

```r
# 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")
```

```
Installing package into '/usr/local/lib/R/site-library'
(as 'lib' is unspecified)

Installing package into '/usr/local/lib/R/site-library'
(as 'lib' is unspecified)

randomForest 4.7-1.2

Type rfNews() to see new features/changes/bug fixes.


Attaching package: 'randomForest'


The following object is masked from 'package:dplyr':

    combine


The following object is masked from 'package:ggplot2':

    margin


RMSE: 2.482045
R-squared: 0.8965005
```

Console Terminal Background Jobs

R 4.4.3 · ~/

```
> # Evaluate the model performance
>
> # Calculate Root Mean Squared Error (RMSE)
> # RMSE measures the standard deviation of the residuals (prediction error
s)
> rmse <- sqrt(mean((test_scaled$mpg - predictions)^2))
> cat("RMSE:", rmse, "\n")
RMSE: 2.482045
>
> # Calculate R-squared
> # R-squared represents the proportion of the variance for a dependent var
iable that's explained by the independent variables in a regression model.
> # It ranges from 0 to 1, where 1 indicates a perfect fit.
> sse <- sum((test_scaled$mpg - predictions)^2) # Sum of squared errors
> sst <- sum((test_scaled$mpg - mean(test_scaled$mpg))^2) # Total sum of sq
uares
> r_squared <- 1 - (sse / sst)
> cat("R-squared:", r_squared, "\n")
R-squared: 0.8965005
```

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'))
```

```
['anscombe.json', 'README.md', 'california_housing_train.csv', 'mnist_test.csv', 'california_housing_test.csv', 'mnist_train_small.csv']
```

```
import pandas as pd

df = pd.read_csv('sample_data/california_housing_train.csv')
```

```
display(df.head())
```

	longitude	latitude	housing_median_age	total_rooms	total_bedrooms	population	households	median_income	median_house_value
0	-114.31	34.19	15.0	5612.0	1283.0	1015.0	472.0	1.4936	66900.0
1	-114.47	34.40	19.0	7650.0	1901.0	1129.0	463.0	1.8200	80100.0
2	-114.56	33.69	17.0	720.0	174.0	333.0	117.0	1.6509	85700.0
3	-114.57	33.64	14.0	1501.0	337.0	515.0	226.0	3.1917	73400.0
4	-114.57	33.57	20.0	1454.0	326.0	624.0	262.0	1.9250	65500.0

Handle missing values

```
display(df.isnull().sum())
```

	0
longitude	0
latitude	0
housing_median_age	0
total_rooms	0
total_bedrooms	0
population	0
households	0
median_income	0
median_house_value	0

dtype: int64

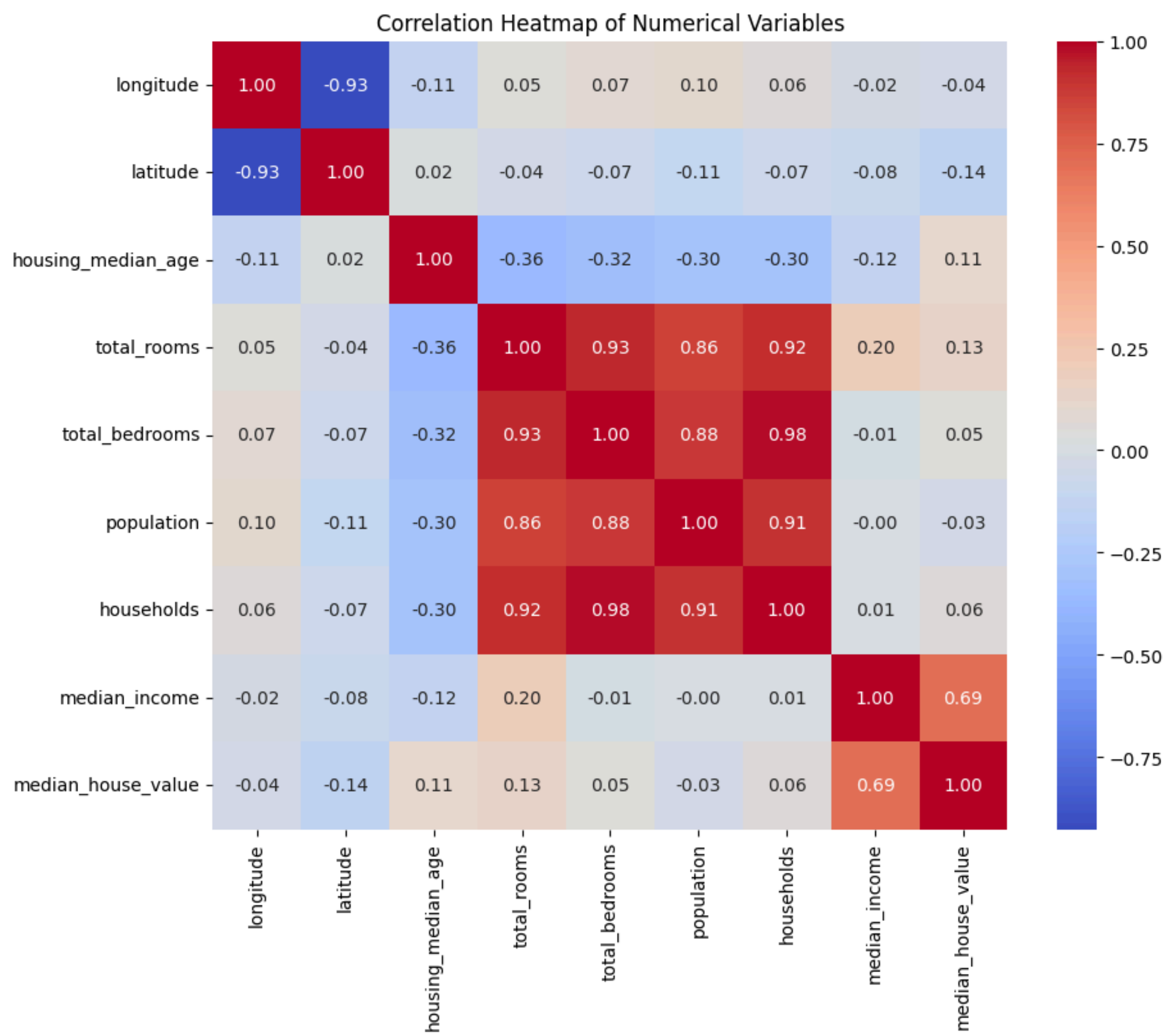
Standardize categorical variables

```
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())
```

```
Categorical columns: Index([], dtype='object')
```

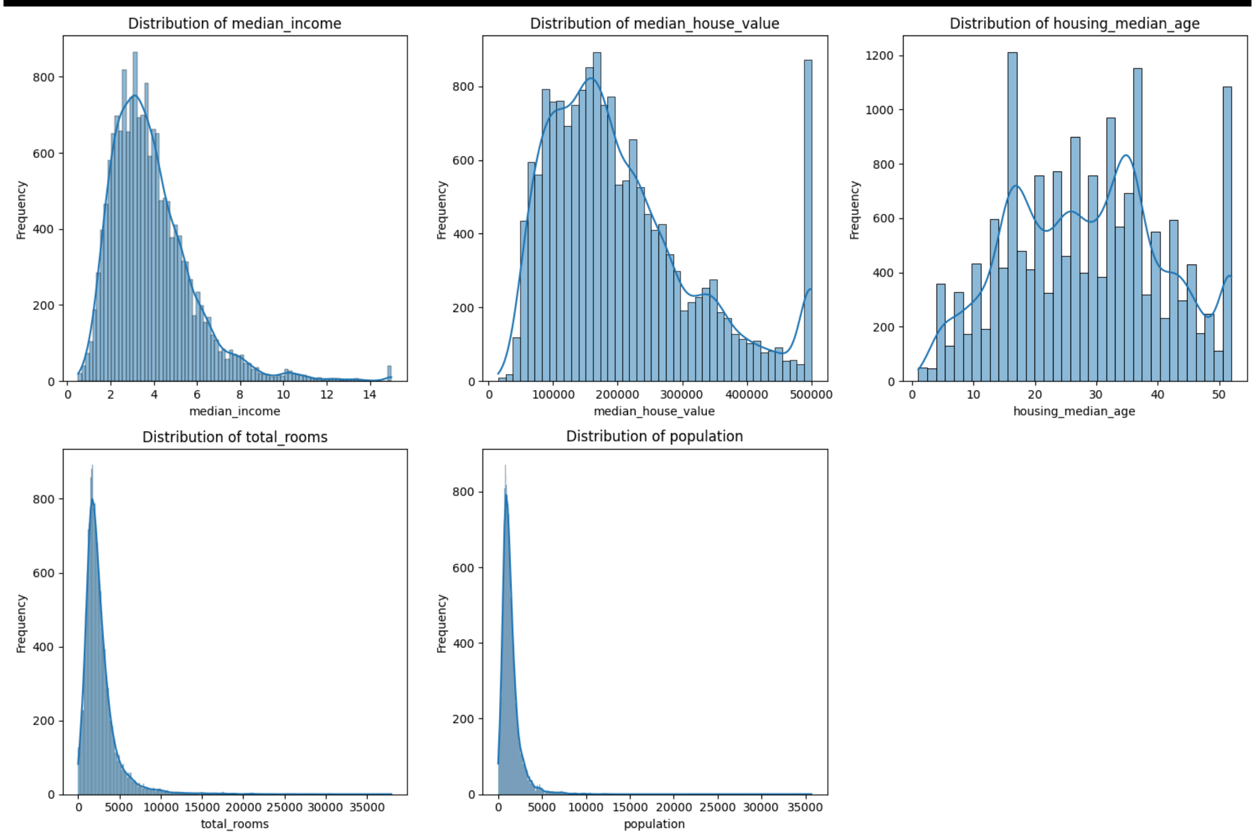
Visualize correlations

```
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

```python
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() {
```

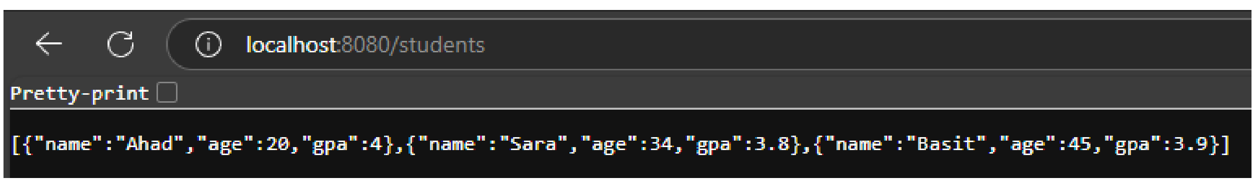
```
	fmt.Printf(" Current balance for %s: $%.2f\n", b.Owner,
b.Balance)
}

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()
}
```

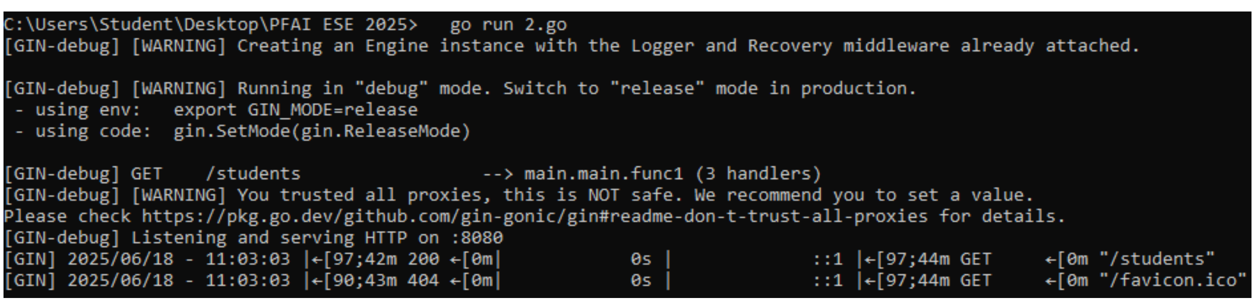
```
Ahad deposited: $1000.00
 Ahad withdrew: $250.00
 Withdrawal of $1000.00 failed: Insufficient funds!
 Current balance for Ahad: $750.00

Program exited.
```

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"`
}
```

```go
// 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)

```go
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.")
}
```

```
[Running] go run "c:\Users\Student\Desktop\PFAI ESE 2025\3.go"
Worker 2 processed task 2 (result: 4)
Worker 1 processed task 1 (result: 1)
Worker 1 processed task 5 (result: 25)
Worker 1 processed task 6 (result: 36)
Worker 2 processed task 4 (result: 16)
Worker 3 processed task 3 (result: 9)
All tasks completed.

[Done] exited with code=0 in 0.258 seconds
```

Part 2: JULIA

- **Create a simple neural network in julia using [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.**

```julia
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")
```

```
Training the neural network...
Initial weight: -1.4703591
Epoch 100: Loss = 0.000644, Weight = 1.9883
Epoch 200: Loss = 0.0, Weight = 2.0
Epoch 300: Loss = 0.0, Weight = 2.0
Epoch 400: Loss = 0.0, Weight = 2.0
Epoch 500: Loss = 0.0, Weight = 2.0
Epoch 600: Loss = 0.0, Weight = 2.0
Epoch 700: Loss = 0.0, Weight = 2.0
Epoch 800: Loss = 0.0, Weight = 2.0
Epoch 900: Loss = 0.0, Weight = 2.0
Epoch 1000: Loss = 0.0, Weight = 2.0

==================================================
Training completed!
Final learned weight: 2.0
Target weight: 2.0
Final loss: 0.0

Testing the model:
Input: 1.0, Predicted: 2.0, Actual: 2.0
Input: 2.0, Predicted: 4.0, Actual: 4.0
Input: 3.0, Predicted: 6.0, Actual: 6.0

New test:
Input: 5.0, Predicted: 10.0, Expected: 10.0
```