Exercise - 1: Basics of Go Environment Configuration

a) Go environment configuration and b) Installation

1. Download the MSI file from the official website of Go.
2. Once the MSI file is downloaded, open it to start the installer.
3. Follow the installation wizard till the end.
4. Once the Go gets installed on your system, you can check whether it is correctly installed or not. Open CMD and type the command:

    D:\>go version

[redacted] go version go1.19.3 windows/amd64

6. You are good to go if it shows the version number of the installed software.
7. Also, check if your GOPATH is set up correctly or not.
8. Go to Control Panel --> Systems and Security --> Systems --> Advanced system settings (from the left window pane) --> Environment Variable.
9. Check whether GOPATH is set or not. Here you can see GOPATH is set to %USERPROFILE%\go.

c) $GOPATH and workspace
d) Go commands

Go is a tool for managing Go source code.

Usage:

```
go <command> [arguments]
```

The commands are:

```
bug         start a bug report
build       compile packages and dependencies
clean       remove object files and cached files
doc         show documentation for package or symbol
env         print Go environment information
fix         update packages to use new APIs
```

```
fmt          gofmt (reformat) package sources
generate     generate Go files by processing source
get          add dependencies to current module and install them
install      compile and install packages and dependencies
list         list packages or modules
mod          module maintenance
work         workspace maintenance
run          compile and run Go program
telemetry    manage telemetry data and settings
test         test packages
tool         run specified go tool
version      print Go version
vet          report likely mistakes in packages
```

Use "go help <command>" for more information about a command.

e) Go development tools

Golang developer tools can help with code formatting, code completion, lining, testing, and debugging. Most of them are available as plugins for popular text editors and IDEs.

Golang developer tools for code formatting

The Go programming language has a tool called gofmt which formats Go source code. The tool can be used to format code so that it is more readable and consistent.

gofmt can also be used to fix code that does not adhere to the Go code style. It uses a simple, predictable, and human-readable format that makes it easy to read and write code.

Golang developer tools for code completion

The Go programming language has a tool called gocode that provides code completion for Go source code. It can be used to automatically complete code snippets based on the context in which they are used.

Golang developer tools for linting

Golint is a linting tool for Golang that helps lint the Go source code. It can be used to find errors in the Go source code, find code that does not adhere to the Go code style, and can find style errors and potential bugs.

Golang developer tools for testing

The Go language tool known as gotest can be used for testing Go applications. It runs unit tests and generates test coverage reports.

Golang developer tools for debugging

Delve (dlv) serves as a tool for debugging Go applications. It can be used to inspect variables, set breakpoints, and step through code.

The best Golang IDEs to be used for Go development:

1. Visual Studio Code 2. LiteIDE 3. Goland

Exercise – 2: Demonstrate CSV Handling, JSON Parsing, and SQL Database Connectivity Using Go such as

a) Read CSV file and find the maximum value in a particular column
Theory:

`os.Open`**: Opens the CSV file for reading, returning a file object and an error if unsuccessful.**
`csv.NewReader`**: Creates a CSV reader to parse the content of the file row by row.**
`reader.ReadAll`**: Reads all records from the CSV file into a 2D string slice for further processing.**
`strconv.Atoi`**: Converts a string from the CSV (like "Salary") into an integer for comparison.**
`fmt.Printf` **allows formatted output, such as printing integers or variables within a string.**

```go
package main

import (
   "encoding/csv"
   "fmt"
   "os"
   "strconv"
)

func main() {
  // Open the CSV file
   file, err := os.Open("Employee.csv")
   if err != nil {
       fmt.Println("Error: Unable to open file")
       return
   }
   defer file.Close()

   // Read the file into a CSV reader
   reader := csv.NewReader(file)

   // Read all records from the CSV
   records, err := reader.ReadAll()
   if err != nil {
       fmt.Println("Error: Unable to read CSV file")
       return
   }

   // Debugging: Print the records
   fmt.Println("CSV Records:")
   for i, record := range records {
       fmt.Printf("Row %d: %v\n", i, record)
   }

   // Check if the CSV is empty
   if len(records) == 0 {
       fmt.Println("Error: The CSV file is empty")
       return
   }

   // Create a map to associate column names with indices
   header := records[0]
   colIndex := make(map[string]int)
   for i, colName := range header {
       colIndex[colName] = i
```

```
    }

    // Access column using column name, e.g., "Salary"
    max := 0
    for _, record := range records[1:] { // Skip header row
        value, err := strconv.Atoi(record[colIndex["Salary"]])
        if err != nil {
            fmt.Println("Error: Unable to convert value to
integer")
            return
        }

        if value > max {
            max = value
        }
    }

    // Print the maximum value
    fmt.Printf("The maximum value in the column Salary is %d\n",
max)
```

b) To read iris dataset which is in csv format and handling of unexpected fields, types and manipulating CSV data.

`os.Open`: Opens the "iris.csv" file for reading, and logs a fatal error if the file cannot be accessed.

`dataframe.ReadCSV`: Reads the CSV into a DataFrame using the `gota/dataframe` package, enabling structured manipulation of the data.

`dataframe.F`: Defines a filter with a column name, comparator, and value, allowing filtering of rows based on conditions, such as species being "Iris-versicolor."

`Filter` Method: Filters the DataFrame to include only rows that match the specified filter condition.

`Select` Method: Selects specific columns from the filtered DataFrame for further analysis or display.

`Subset` Method: Extracts specific rows by index from the filtered DataFrame, useful for limiting the displayed data.

```
package main

import (
    "fmt"
    "log"
    "os"
```

```go
	"github.com/go-gota/gota/dataframe"
)

func main() {
	// Open the CSV file.
	file, err := os.Open("iris.csv")
	if err != nil {
		log.Fatal(err)
	}
	defer file.Close()

	// Read the file into a DataFrame.
	irisDF := dataframe.ReadCSV(file)

	// Create a filter for the dataframe to select rows where the
species is "Iris-versicolor".
	filter := dataframe.F{
		Colname:    "species",
		Comparator: "==",
		Comparando: "Versicolor",
	}

	// Filter the dataframe to see only the rows where the
species is "Iris-versicolor".
	versicolorDF := irisDF.Filter(filter)
	if versicolorDF.Err != nil {
		log.Fatal(versicolorDF.Err)
	}

	// Print the filtered dataframe.
	fmt.Println("Filtered DataFrame:")
	fmt.Println(versicolorDF)

	// Filter the dataframe again, but only select out the
"sepal_width" and "species" columns.
	versicolorDF =
irisDF.Filter(filter).Select([]string{"sepal_width", "species"})

	// Print the selected columns.
	fmt.Println("\nFiltered and Selected Columns:")
	fmt.Println(versicolorDF)

	// Filter and select the dataframe again, but only display
the first three results.
```

```
    versicolorDF =
irisDF.Filter(filter).Select([]string{"sepal_width",
"species"}).Subset([]int{0, 1, 2})

    // Print the subset of the dataframe.
    fmt.Println("\nFiltered, Selected, and Subset DataFrame:")
    fmt.Println(versicolorDF)
}

// go mod init read_data

// go get -u github.com/go-gota/gota

// go get github.com/go-gota/gota/dataframe@v0.12.0
```

c) Parse JSON data using Go

`json.Unmarshal`: Converts (or deserializes) JSON data from a byte slice into Go data structures, here into the `StationData` struct that holds the station information.
Struct Tags: The struct fields in `Station` use tags (e.g., `json:"station_id"`) to map JSON keys to Go struct fields, ensuring the correct association between JSON data and struct attributes.
`os.Open`: Opens the "sample.json" file for reading, and if an error occurs, the program logs it and exits.
`ioutil.ReadAll`: Reads the content of the JSON file into memory as a byte slice, which is then passed to `json.Unmarshal` for processing.

```go
package main

import (
    "encoding/json"
    "fmt"

    "log"
    "os"
)

type Station struct {
    ID                string `json:"station_id"`
    NumBikesAvailable int    `json:"num_bikes_available"`
}

type StationData struct {
```

```go
    Stations []Station `json:"stations"`
}

func main() {
    // Open the JSON file
    jsonFile, err := os.Open("sample.json")
    if err != nil {
        log.Fatal(err)
    }
    defer jsonFile.Close()

    // Read the file contents
    byteValue, _ := ioutil.ReadAll(jsonFile)

    // Unmarshal the JSON data into our stationData struct
    var stationData StationData
    json.Unmarshal(byteValue, &stationData)

    // Print the first station's data
    fmt.Printf("Station ID: %s, Bikes Available: %d\n",
stationData.Stations[0].ID,
stationData.Stations[0].NumBikesAvailable)

    // Print the number of records
    fmt.Printf("Number of records: %d\n",
len(stationData.Stations))

    // Print all stations' data
    for _, station := range stationData.Stations {
        fmt.Printf("Station ID: %s, Bikes Available: %d\n",
station.ID, station.NumBikesAvailable)
    }
}
```

d) To connect and Query SQL like databases (Postgres MySQL, SQL Lite)

To connect to a PostgreSQL database, you need to provide specific connection parameters. These parameters ensure that your application can communicate with the PostgreSQL server correctly:

- URI type: The protocol specification or application type. Both `postgresql://` and `postgres://` are valid URI schema designators.

- User credentials: The userspec is optional but typically required if you don't want to rely on defaults.
- Host: Specifies the host name and port on which PostgreSQL is running.
- Port: The TCP port of the PostgreSQL server. The default is 5432.
- Database: The PostgreSQL database name. You can enter the name of a PostgreSQL

Step 1: Set Up Your Go Environment

1. **Install Go**: If you haven't installed Go, download it from [the official website](#).
2. **Set Up a Go Workspace**:
   - Create a directory for your Go project. Mkdir project_sql
   - Cd project_sql
   - Initialize your Go module:

go mod init project_sql

To Install drivers in command line:

go get github.com/lib/pq

Step 2: Writing the Go Code

2.1 Connecting to the Database

In your `main.go` file, start by importing the necessary packages and writing the connection logic.

Download Postgresql:

sudo apt install postgresql postgresql-contrib

Start the postgresql:

sudo systemctl start postgresql

Check status of postgresql:

sudo systemctl status postgresql

The status should be active before performing any action.

Redirect to Postgresql command line for creating database and user using:

sudo -i -u postgres

Type **psql** and then **\du** for checking database and the users if not exit press q or crtrl+c

Deleting a existing database:

DROP DATABASE database_name;

Deleting a existing Role:

DROP ROLE role_name;

Then create Database using command:

CREATE DATABASE my_database;

Creating a role in the database:

CREATE USER username WITH PASSWORD '123456789';

Granting all permissions:

GRANT ALL PRIVILEGES ON DATABASE my_database TO username;

Type **\l** to check weather everything is perfect with databases and usernames if not exit press q or crtrl+c

Type exit 2 times to exit the postgresql command line after successfully creating the database and users.

PostgreSQL:

```
package main
```

```go
import (
   "database/sql"
   "fmt"
   "log"

   _ "github.com/lib/pq"
)

func main() {
   // Corrected connection string with proper single quotes
   connStr := "user=username password='abcd123' dbname=my_database
sslmode=disable"
   db, err := sql.Open("postgres", connStr)
   if err != nil {
       log.Fatal(err)
   }
   defer db.Close()

   // Test the database connection
   err = db.Ping()
   if err != nil {
       log.Fatal("Failed to connect to the database:", err)
   }
```

```
    fmt.Println("Connected to PostgreSQL!")

    //Query to create tables or for any other queries.



}
```

3.2 Creating a Table

After connecting, you can execute SQL statements to create tables.

For All Databases:

```
query := `
CREATE TABLE IF NOT EXISTS users (
	id SERIAL PRIMARY KEY,
	name TEXT NOT NULL,
	email TEXT NOT NULL UNIQUE
);`
_, err = db.Exec(query)
if err != nil {
	log.Fatal(err)
}

fmt.Println("Table created successfully!")
```

3.3 Inserting Data

You can insert data using `Exec()` as well.

For All Databases:

```
insertQuery := `INSERT INTO users (name, email) VALUES ($1, $2)`
_, err = db.Exec(insertQuery, "John Doe", "john@example.com")
if err != nil {
	log.Fatal(err)
}

fmt.Println("Data inserted successfully!")
```

3.4 Querying Data

To retrieve data, you can use `Query()` or `QueryRow()`.

For All Databases:

```
rows, err := db.Query("SELECT id, name, email FROM users")
if err != nil {
	log.Fatal(err)
}
defer rows.Close()

for rows.Next() {
	var id int
	var name, email string
	err = rows.Scan(&id, &name, &email)
	if err != nil {
	log.Fatal(err)
```

```
    }
    fmt.Printf("User: %d, %s, %s\n", id, name, email)
}
```

3.5 Updating and Deleting Data

Updating Data:

```
updateQuery := `UPDATE users SET email = $1 WHERE name = $2`
_, err = db.Exec(updateQuery, "john.doe@example.com", "John Doe")
if err != nil {
    log.Fatal(err)
}

fmt.Println("Data updated successfully!")
```

Deleting Data:

```
deleteQuery := `DELETE FROM users WHERE name = $1`
_, err = db.Exec(deleteQuery, "John Doe")
if err != nil {
    log.Fatal(err)
}

fmt.Println("Data deleted successfully!")
```

Step 4: Running the Go Program

1. **Save your `main.go` file**.
2. **Run your program**:

go run main.go

3. Demonstrate Control Statements and Data Structures in Go such as

a) Write a program that prints the numbers from 1 to 100, but for multiples of three, print “Fizz” instead of the number, and for the multiples of five, print “Buzz.” For numbers that are multiples of both three and five, print “FizzBuzz.”

Theory:

This program implements the FizzBuzz problem using a loop from 1 to 100. It checks if each number is divisible by 3 and 5 (printing "FizzBuzz"), by 3 (printing "Fizz"), or by 5 (printing "Buzz"). If none of these conditions are met, it simply prints the number.

```go
package main


import (
   "fmt"
)

func main() {
   for i := 1; i <= 100; i++ {
       if i%3 == 0 && i%5 == 0 {
           fmt.Println("FizzBuzz")
           continue
       }
       if i%3 == 0 {
           fmt.Println("Fizz")
           continue
       }
       if i%5 == 0 {
           fmt.Println("Buzz")
           continue
       }
       fmt.Println(i)
   }

}
```

b) Write a program to access the fourth element of an array or slice?

Theory:
This program demonstrates the use of arrays and slices in Go. It declares an array arr of fixed size (6) and a slice sli of dynamic size. The program accesses and prints the fourth element from both the array and slice using index notation. Arrays have a fixed length, while slices are flexible and can grow in size.

```go
package main

import (
   "fmt"
)

func main() {
   var arr = [6]int{1, 2, 3, 4, 5, 6}

   var sli = []int{11, 12, 13, 14, 15, 16}

   fmt.Println("Fourth element of array=", arr[3])
   fmt.Println("Fourth element of slice=", sli[3])
}
```

c) Write a program to perform reading, writing, deleting, emptying operations on Maps

Theory:
In Go, maps allow key-value pair lookups, where values are accessed using keys. The "comma ok" idiom (`v, ok := m["key"]`) checks whether a key exists, returning the value and a boolean indicating existence. The `delete` function removes a key-value pair from the map. Modifying map values can be done by directly assigning or incrementing the value using its key (`totalWins["Kittens"]++`). The `clear` operation empties the map by resetting all key-value pairs.

```go
package main

import "fmt"

func main() {

   totalWins := map[string]int{}

   totalWins["Orcas"] = 1

   totalWins["Lions"] = 2
```

```go
    fmt.Println(totalWins["Orcas"])
    fmt.Println(totalWins["Kittens"])
    totalWins["Kittens"]++
    fmt.Println(totalWins["Kittens"])
    totalWins["Lions"] = 3
    fmt.Println(totalWins["Lions"])

    m := map[string]int{"hello": 5, "world": 0}
    v, ok := m["hello"]
    fmt.Println(v, ok)
    v, ok = m["world"]
    fmt.Println(v, ok)
    v, ok = m["goodbye"]
    fmt.Println(v, ok)

    delete(m, "hello")
    fmt.Println(m)
    clear(m)
    fmt.Println(m, len(m))

}
```

Exercise – 4: Demonstrate Functions using GO such as

a) The simple calculator program doesn't handle one error case: division by zero. Change the function signature for the math operations to return both an int and an error. In the div function, if the divisor is 0, return errors.New("division by zero") for the error. In all other cases, return nil. Adjust the main function to check for this error

Theory

The `main` package includes error handling and function mapping for basic arithmetic operations. A map `opMap` is used to associate operators (like "+", "-", "*", "/") with their corresponding functions. The `main()` function processes a list of arithmetic expressions, converting string operands to integers using `strconv.Atoi`, and checks for errors like invalid operators or division by zero.

```go
package main

import (
   "errors"
   "fmt"
   "strconv"
)

func add(i int, j int) (int, error) { return i + j, nil }

func sub(i int, j int) (int, error) { return i - j, nil }

func mul(i int, j int) (int, error) { return i * j, nil }

func div(i int, j int) (int, error) {
   if j == 0 {
       return 0, errors.New("division by zero")
   }
   return i / j, nil
}

var opMap = map[string]func(int, int) (int, error){
   "+": add,
   "-": sub,
   "*": mul,
   "/": div,
}

func main() {
   expressions := [][]string{
       {"2", "+", "3"},
       {"2", "-", "3"},
       {"2", "*", "3"},
       {"2", "/", "3"},
       {"2", "%", "3"},
       {"two", "+", "three"},
       {"5"},
       {"10", "/", "0"},
   }


   for _, expression := range expressions {
       if len(expression) != 3 {
           fmt.Println("invalid expression:", expression)
           continue
  }
       p1, err := strconv.Atoi(expression[0])
       if err != nil {
           fmt.Println(err)
           continue
       }
       op := expression[1]
       opFunc, ok := opMap[op]
       if !ok {
```

```
            fmt.Println("unsupported operator:", op)
            continue
        }
        p2, err := strconv.Atoi(expression[2])
        if err != nil {
            fmt.Println(err)
            continue
        }
        result, err := opFunc(p1, p2)
        if err != nil {
            fmt.Println(err)
            continue
        }
        fmt.Println(result)
    }
}
```

b) Write a function with one variadic parameter that finds the greatest number in a list of numbers.

Theory

Variadic Parameters: The `Max` function accepts a variable number of integer arguments using `...int`, allowing it to handle multiple inputs flexibly.

Error Handling: If no numbers are provided, `log.Fatal` is called to log the error and terminate the program immediately.

Finding Maximum: The function iterates through the input integers, comparing each one to find the maximum value.

```
package main

import (
 "fmt"
   "log"
)

// Max finds the greatest number in a variable number of integers.
func Max(nums ...int) int {
   if len(nums) == 0 {
       log.Fatal("No numbers provided")
   }
```

```go
    max := nums[0]
    for _, num := range nums[1:] {
        if num > max {
            max = num
        }
    }
    return max
}

func main() {
    fmt.Println(Max(1, 2, 3, 4, 5))         // Output: 5
    fmt.Println(Max(10, 20, 30, 5, 15, 25)) // Output: 30
    fmt.Println(Max(7, 2, 9, 3, 8, 1, 6))   // Output: 9
    fmt.Println(Max(-1, -5, -3, -10, -2))
    fmt.Println(Max()) // Output: -1
}
```

```
Output:
5
30
9
-1
2009/11/10 23:00:00 No numbers provided
```

Exercise – 5: Demonstrate the concept of Interface and packages

a) Add a new perimeter method to the Shape interface to calculate the perimeter of a shape. Implement the method for Circle and Rectangle.

Theory

In Go, structs like `Circle` and `Rectangle` are user-defined types that group related fields together. So in the code, `Circle` and `Rectangle` are actually struct types. They implement the `Shape` interface by providing methods for `Area()` and `Perimeter()`. This allows the use of polymorphism, where a variable of the `Shape` interface can hold and manipulate values of both `Circle` and `Rectangle`.

```go
package main

import (
   "fmt"
   "math"
)

// Define the Shape interface with Area and Perimeter methods
type Shape interface {
   Area() float64
   Perimeter() float64
}

// Circle type with radius
type Circle struct {
   Radius float64
}

// Implement the Area method for Circle
func (c Circle) Area() float64 {
   return math.Pi * c.Radius * c.Radius
}

// Implement the Perimeter method for Circle
func (c Circle) Perimeter() float64 {
   return 2 * math.Pi * c.Radius
}

// Rectangle type with width and height
type Rectangle struct {
   Width, Height float64
}

// Implement the Area method for Rectangle
func (r Rectangle) Area() float64 {
   return r.Width * r.Height
}

// Implement the Perimeter method for Rectangle
func (r Rectangle) Perimeter() float64 {
   return 2 * (r.Width + r.Height)
}

func main() {
   var s Shape
   c := Circle{Radius: 5}
   r := Rectangle{Width: 3, Height: 4}

   s = c

   fmt.Printf("Area of circle: %.2f\n", s.Area())
   fmt.Printf("Perimeter of circle: %.2f\n\n", s.Perimeter())

   s = r
```

```
    fmt.Printf("Area of rectangle: %.2f\n", s.Area())
    fmt.Printf("Perimeter of rectangle: %.2f\n\n", s.Perimeter())

}
```

b) Develop a program to create and access packages.

A module is a collection of related Go packages that are versioned together. A module is defined by a `go.mod` file, which specifies the module's path and its dependencies.This creates a `go.mod` file, which will track the module's dependencies.

To create a new module, navigate to your project directory and run:

go mod init mymodule

The package name is usually the same as the directory in which the source files are stored. For instance, if your files are in a directory named `utils`, the package name should be `utils`

Dependencies Management:

- **The `go.mod` file lists all the dependencies your module requires. Dependencies are managed using commands like `go get` and are automatically downloaded and added to the `go.sum` file.**

A package is a collection of Go source files in the same directory that are compiled together. Each package serves as a separate namespace and can be imported by other packages.

The `main` package is special. It's the entry point for the executable programs in Go. When you compile a program, Go looks for a `package main` with a `func main()` to start the execution.

Step 1: Set Up Your Project Directory

1. **Open VS Code:**
    - Open Visual Studio Code.
2. **Create the Project Directory:**
    - Navigate to the location where you want to create your project directory. You can do this with the `cd` command.
    - Create a new directory for your project using the `mkdir` command.

```
mkdir myproject
cd myproject
```

Step 2: Initialize the Go Module

1. **Initialize the Module:**
   - In the terminal, run the following command to initialize a new Go module.

```
go mod init myproject
```

This command creates a `go.mod` file, which tracks your project's dependencies.

Step 3: Create the Package Directory and Files

1. **Create the Package Directory:**
   - Create a directory for your package (e.g., `mathutil`) within your project.

```
mkdir mathutil
```

2. Create the `mathutil.go` File:

- Inside the `mathutil` directory, create a new file called `mathutil.go`.

```
code mathutil/mathutil.go
```

This command will open the `mathutil.go` file in VS Code. Alternatively, you can manually create the file by right-clicking the `mathutil` directory in the VS Code Explorer pane and selecting "New File."

Write the Package Code:

- Copy and paste the following code into `mathutil/mathutil.go`:

```
// Package mathutil provides basic mathematical utilities.
package mathutil

// Add takes two integers and returns their sum.
func Add(a, b int) int {
   return a + b
}

// Subtract takes two integers and returns their difference.
func Subtract(a, b int) int {
   return a - b
}
```

Step 4: Create the Main Application File

1. **Create the `packagetest.go` File:**
   - In the root of your project (`myproject`), create a new file called `packagetest.go`.

```
code packagetest.go
```

This will open the `packagetest.go` file in VS Code.
Write the Main Program:

- Copy and paste the following code into `packagetest.go`:

```
package main

import (
    "fmt"
    "myproject/mathutil"
)

func main() {
    sum := mathutil.Add(5, 3)
    difference := mathutil.Subtract(10, 4)

    fmt.Println("Sum:", sum)
    fmt.Println("Difference:", difference)
}
```

Step 5: Run the Program

1. **Run the Program in the Terminal:**
   - Make sure you're in the project root directory (`myproject`), then run the program using the following command:

```
go run packagetest.go
```

Output:

- The terminal should display:

Sum: 8
Difference: 6

Exercise – 6: Demonstrating Concurrency in Go: Using Goroutines, Channels, and Wait Groups such as

a) Write a Go program that uses goroutines and channels to fetch several web pages simultaneously using the net/http package, and prints the URL of the biggest home page (defined as the most bytes in the response)

Theory:
A WaitGroup in Go is a synchronization mechanism that helps you wait for a collection of goroutines to complete before proceeding. It's typically used when you have multiple goroutines doing some concurrent work, and you want to ensure that all of them finish before moving on.
How WaitGroup Works:
Adding Work (wg.Add):
You inform the WaitGroup that you are adding a task (or goroutine) to wait for by calling wg.Add(n), where nis the number of tasks. This step increases an internal counter in the WaitGroup.
Marking Completion (wg.Done):
Each goroutine that completes its work must call wg.Done(). This decrements the counter in the WaitGroup.
Waiting for Completion (wg.Wait):
The main or parent goroutine calls wg.Wait(), which blocks and waits until the counter becomes zero (i.e., all the goroutines have finished their work). Once all the tasks are done, the program can proceed.

Rewriting above program using Waitgroups

```go
package main

import (
   "fmt"
   "io"
   "net/http"
   "sync"
)

type HomePageSize struct {
   URL  string
   Size int
}

func main() {
   urls := []string{
```

```go
        "http://www.apple.com",
        "http://www.amazon.com",
        "http://www.google.com",
        "http://www.microsoft.com",
    }
    results := make(chan HomePageSize, 4)
    var wg sync.WaitGroup

    for _, url := range urls {
        wg.Add(1)
        go func(url string) {
            defer wg.Done()
            res, err := http.Get(url)
            if err != nil {
                panic(err)
            }
            defer res.Body.Close()
            bs, err := io.ReadAll(res.Body)
            if err != nil {
                panic(err)
            }
            results <- HomePageSize{URL: url, Size: len(bs)}
        }(url)
    }

    wg.Wait()      // Wait for all goroutines to finish
     close(results)  //  Close  the  results  channel  after  all
goroutines complete

    var biggest HomePageSize
    for result := range results {
        if result.Size > biggest.Size {
            biggest = result
        }
    }

    fmt.Printf("The biggest home page: %s, and biggest size is %d
bytes\n", biggest.URL, biggest.Size)
}
```

Part-B Machine Learning with GO programming Exercise – 7: Develop Regression models using Go such as

a) Demonstrate how to build a linear regression model using Go.

Theory

- `csv.NewReader()`: This function creates a CSV reader to read and parse the data from the `training.csv` and `test.csv` files. It converts the data into rows that can be used for training and testing the regression model.
- `regression.Regression`: This is a linear regression model from the `github.com/sajari/regression` package. It is used to train a model that predicts a dependent variable (Sales) based on an independent variable (TV advertising spending). The `SetObserved()` method defines the dependent variable, while `SetVar()` defines the independent variable.
- `r.Train()`: This method adds training data to the regression model. In each iteration, it takes the sales value (dependent variable) and the corresponding TV advertising value (independent variable) from the training dataset and stores them as data points for training.
- `r.Run()`: This function fits the regression model to the training data by calculating the best-fit line, which minimizes prediction errors. The formula for the model is then printed.
- `r.Predict()`: This function uses the trained model to predict the dependent variable (Sales) for the test dataset based on the given TV advertising values. The predicted values are compared with actual sales values to evaluate the model's performance.
- Mean Absolute Error (MAE): The program calculates the mean absolute error (MAE) between the predicted and actual sales values from the test data. MAE measures the average magnitude of errors in the predictions, giving an indication of how well the model performs. It is printed at the end to show the model’s accuracy.

```go
package main

import (
   "encoding/csv"
   "fmt"
   "log"
   "math"
   "os"
   "strconv"
```

```
    "github.com/sajari/regression"
)

func main() {

    // Open the training dataset file.
    f, err := os.Open("training.csv")
    if err != nil {
        log.Fatal(err)
    }
    defer f.Close()

    // Create a new CSV reader reading from the opened file.
    reader := csv.NewReader(f)

    // Read in all of the CSV records
    reader.FieldsPerRecord = 4
    trainingData, err := reader.ReadAll()
    if err != nil {
        log.Fatal(err)
    }

    // In this case we are going to try and model our Sales (y)
    // by the TV feature plus an intercept.  As such, let's
create
    // the struct needed to train a model using
github.com/sajari/regression.
    var r regression.Regression
    r.SetObserved("Sales")
    r.SetVar(0, "TV")

    // Loop of records in the CSV, adding the training data to
the regression value.
    for i, record := range trainingData {

        // Skip the header.
        if i == 0 {
            continue
        }

        // Parse the Sales rogression measure, or "y".
        yVal, err := strconv.ParseFloat(record[3], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Parse the TV value.
```

```
        tvVal, err := strconv.ParseFloat(record[0], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Add these points to the regression value.
        r.Train(regression.DataPoint(yVal, []float64{tvVal}))
    }

    // Train/fit the regression model.
    r.Run()

    // Output the trained model parameters.
    fmt.Printf("\nRegression Formula:\n%v\n\n", r.Formula)

    // Open the test dataset file.
    f, err = os.Open("test.csv")
    if err != nil {
        log.Fatal(err)
    }
    defer f.Close()

    // Create a CSV reader reading from the opened file.
    reader = csv.NewReader(f)

    // Read in all of the CSV records
    reader.FieldsPerRecord = 4
    testData, err := reader.ReadAll()
    if err != nil {
        log.Fatal(err)
    }

    // Loop over the test data predicting y and evaluating the
prediction
    // with the mean absolute error.
    var mAE float64
    for i, record := range testData {

        // Skip the header.
        if i == 0 {
            continue
        }

        // Parse the observed diabetes progression measure, or
"y".
        yObserved, err := strconv.ParseFloat(record[3], 64)
        if err != nil {
```

```go
            log.Fatal(err)
        }

        // Parse the bmi value.
        tvVal, err := strconv.ParseFloat(record[0], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Predict y with our trained model.
        yPredicted, err := r.Predict([]float64{tvVal})

        // Add the to the mean absolute error.
        mAE += math.Abs(yObserved-yPredicted) /
float64(len(testData))
    }

    // Output the MAE to standard out.
    fmt.Printf("MAE = %0.2f\n\n", mAE)
}
```

b) Demonstrate how to build a multiple linear regression model using Go.

Theory

`csv.NewReader()`: **This function creates a CSV reader to read and parse data from** `training.csv` **and** `test.csv`**. It converts each row of the file into a slice of strings which are later parsed into numeric values for training and testing the regression model.**

`regression.Regression`: **This is a multiple linear regression model from the** `github.com/sajari/regression` **package. It aims to model the relationship between the dependent variable** `Sales` **and two independent variables:** `TV` **and** `Radio`**. The** `SetObserved()` **function defines the dependent variable, and** `SetVar()` **specifies the independent variables.**

`r.Train()`: **This method adds training data to the regression model. For each row of training data, it reads the values of** `Sales` **(the dependent variable),** `TV`**, and** `Radio` **(the independent variables) and trains the model using these data points.**

`r.Run()`: **This function fits the regression model by calculating the best-fit line (or plane, in this case) through the training data. It computes the regression** coefficients, which describe the relationship between the dependent and independent variables.

`r.Predict()`: After training, this method predicts `Sales` values for the test dataset using the trained model based on the given `TV` and `Radio` values. The predicted values are compared with the actual `Sales` values in the test data to evaluate the model.

Mean Absolute Error (MAE): The program calculates the mean absolute error (MAE) between the predicted and actual `Sales` values for the test data. MAE provides an average of the errors in predictions, giving a measure of how accurately the model performed.

```
package main

import (
	"encoding/csv"
	"fmt"
	"log"
	"math"
	"os"
	"strconv"

	"github.com/sajari/regression"
)

func main() {

	// Open the training dataset file.
	f, err := os.Open("training.csv")
	if err != nil {
		log.Fatal(err)
	}
	defer f.Close()

	// Create a new CSV reader reading from the opened file.
	reader := csv.NewReader(f)

	// Read in all of the CSV records
	reader.FieldsPerRecord = 4
	trainingData, err := reader.ReadAll()
	if err != nil {
```

```go
		log.Fatal(err)
	}

	// In this case we are going to try and model our Sales
	// by the TV and Radio features plus an intercept.
	var r regression.Regression
	r.SetObserved("Sales")
	r.SetVar(0, "TV")
	r.SetVar(1, "Radio")

	// Loop over the CSV records adding the training data.
	for i, record := range trainingData {

		// Skip the header.
		if i == 0 {
			continue
		}

		// Parse the Sales.
		yVal, err := strconv.ParseFloat(record[3], 64)
		if err != nil {
			log.Fatal(err)
		}

		// Parse the TV value.
		tvVal, err := strconv.ParseFloat(record[0], 64)
		if err != nil {
			log.Fatal(err)
		}

		// Parse the Radio value.
		radioVal, err := strconv.ParseFloat(record[1], 64)
		if err != nil {
			log.Fatal(err)
		}

		// Add these points to the regression value.
		r.Train(regression.DataPoint(yVal, []float64{tvVal,
radioVal}))
	}

	// Train/fit the regression model.
	r.Run()

	// Output the trained model parameters.
	fmt.Printf("\nRegression Formula:\n%v\n\n", r.Formula)
```

```go
    // Open the test dataset file.
    f, err = os.Open("test.csv")
    if err != nil {
        log.Fatal(err)
    }
    defer f.Close()

    // Create a CSV reader reading from the opened file.
    reader = csv.NewReader(f)

    // Read in all of the CSV records
    reader.FieldsPerRecord = 4
    testData, err := reader.ReadAll()
    if err != nil {
        log.Fatal(err)
    }

    // Loop over the test data predicting y and evaluating the
prediction
    // with the mean absolute error.
    var mAE float64
    for i, record := range testData {

        // Skip the header.
        if i == 0 {
            continue
        }

        // Parse the Sales.
        yObserved, err := strconv.ParseFloat(record[3], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Parse the TV value.
        tvVal, err := strconv.ParseFloat(record[0], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Parse the Radio value.
        radioVal, err := strconv.ParseFloat(record[1], 64)
        if err != nil {
            log.Fatal(err)
        }

        // Predict y with our trained model.
```

```
        yPredicted, err := r.Predict([]float64{tvVal, radioVal})

        // Add the to the mean absolute error.
        mAE += math.Abs(yObserved-yPredicted) /
float64(len(testData))
    }

    // Output the MAE to standard out.
    fmt.Printf("MAE = %0.2f\n\n", mAE)
}
```

c) Demonstrate how to build a logistic regression model using Go.

Theory

`base.LoadDataFromCSV()`: This function reads data from CSV files and converts them into matrices for training and testing logistic regression. It loads the input features (`xTrain`, `xTest`) and corresponding labels (`yTrain`, `yTest`).

`linear.NewLogistic()`: This creates a logistic regression model with parameters such as learning rate (`0.0001`) and 1000 iterations. Logistic regression is used for binary classification, where it predicts outcomes between two classes.

`model.Learn()`: This method trains the logistic regression model using the training data (`xTrain`, `yTrain`). It updates the model's parameters to reduce prediction errors.

`model.Predict()`: This function makes predictions for test data points. For each input, it returns the probability of belonging to class 1. A threshold of 0.5 is used to classify the data into either class 0 or class 1.

`evaluateAccuracy()`: This function calculates the accuracy of the model by comparing the predicted results with actual labels (`yTest`). It counts how many predictions are correct and computes the accuracy.

Accuracy Calculation: The accuracy, displayed as a percentage, indicates the proportion of correct predictions made by the model on the test data.

```go
package main

import (
	"fmt"
	"io/ioutil"

	"github.com/cdipaolo/goml/base"
	"github.com/cdipaolo/goml/linear"
)

// Run executes the logistic regression and prints test accuracy
func Run() error {
	// Load the training and test datasets
	xTrain, yTrain, err :=
base.LoadDataFromCSV("studentsTrain.csv")
	if err != nil {
		return err
	}
	xTest, yTest, err := base.LoadDataFromCSV("studentsTest.csv")
	if err != nil {
		return err
	}

	// Define logistic regression model with simple parameters
	model := linear.NewLogistic(base.BatchGA, 0.0001, 0.0, 1000,
xTrain, yTrain)
	model.Output = ioutil.Discard // Disable output during
training

	// Train the model
	err = model.Learn()
	if err != nil {
		return err
	}

	// Calculate accuracy on the test set
	accuracy := evaluateAccuracy(model, xTest, yTest)

	// Print test accuracy
	fmt.Printf("Test Accuracy: %.2f%%\n", accuracy*100)

	return nil
}

// evaluateAccuracy calculates accuracy on the test data
func evaluateAccuracy(model *linear.Logistic, xTest [][]float64,
yTest []float64) float64 {
```

```go
    correctPredictions := 0
    for i := range xTest {
        prediction, err := model.Predict(xTest[i])
        if err != nil {
            fmt.Println("Error during prediction:", err)
            continue
        }

        // Threshold is 0.5 for binary classification
        predictedClass := 0.0
        if prediction[0] >= 0.5 {
            predictedClass = 1.0
        }

        if predictedClass == yTest[i] {
            correctPredictions++
        }
    }

    // Return accuracy as the percentage of correct predictions
    return float64(correctPredictions) / float64(len(yTest))
}

func main() {
    // Run the logistic regression process
    err := Run()
    if err != nil {
        fmt.Println("Error:", err)
    }
}
```

Exercise – 8: Develop classification models using Go such as

a) Apply k-nearest neighbor classifier on iris dataset using Go

Theory:

`base.ParseCSVToInstances()`: The `ParseCSVToInstances` function from the `golearn` library is responsible for parsing the CSV file into instances that represent the data. The second argument in this function call, `true`, specifies whether the first row in the CSV contains the header information (i.e., column names).

The golearn package expects the target variable (also known as the label or class) to be the last column in the dataset by default. The remaining columns are treated as features for the model. The KNN classifier then uses this assumption to differentiate between features (inputs) and the target (output or class label) during training and evaluation.

knn.NewKnnClassifier(): This creates a k-Nearest Neighbors (k-NN) classifier. The classifier uses the Euclidean distance to find the 2 closest data points (k=2) and classify new data based on them.

evaluation.GenerateCrossFoldValidationConfusionMatrices(): This function splits the data into 5 parts (folds), trains the model on each part, and evaluates its performance to check how well it works.

evaluation.GetCrossValidatedMetric(): This function calculates the average accuracy and variance from the cross-validation results, showing how well the model performs on different parts of the data.

```go
package main

import (
   "fmt"
   "log"
   "math"

   "github.com/sjwhitworth/golearn/base"
   "github.com/sjwhitworth/golearn/evaluation"
   "github.com/sjwhitworth/golearn/knn"
)

func main() {

   // Read in the iris data set into golearn "instances".
   irisData, err := base.ParseCSVToInstances("iris_2.csv", true)
   if err != nil {
       log.Fatal(err)
   }
   // Initialize a new KNN classifier.  We will use a simple
   // Euclidean distance measure and k=2.
   knn := knn.NewKnnClassifier("euclidean", "linear", 2)
```

```
    // Use cross-fold validation to successively train and
evalute the model
    // on 5 folds of the data set.
    cv, err :=
evaluation.GenerateCrossFoldValidationConfusionMatrices(irisData
, knn, 5)
    if err != nil {
        log.Fatal(err)
    }

    // Get the mean, variance and standard deviation of the
accuracy for the
    // cross validation.
    mean, variance := evaluation.GetCrossValidatedMetric(cv,
evaluation.GetAccuracy)
    stdev := math.Sqrt(variance)

    // Output the cross metrics to standard out.
    fmt.Printf("\nAccuracy\n%.2f (+/- %.2f)\n\n", mean, stdev*2)

}
```

8. B. Build a decision tree on iris dataset using Go.

Theory:

`base.ParseCSVToInstances()`: **This function reads data from a CSV file and converts it into a structured format that GoLearn can use for building machine learning models.**

`rand.Seed()`: **This sets a fixed starting point for any random processes. It ensures that the decision tree-building process is consistent and produces the same results each time the program is run.**

`trees.NewID3DecisionTree()`: **This creates a decision tree using the ID3 algorithm. The parameter** `0.6` **means that 60% of the data is used for training the tree, while the remaining 40% is used for pruning, which simplifies the tree to avoid overfitting.**

`evaluation.GenerateCrossFoldValidationConfusionMatrices()`: **This function divides the data into 5 parts (folds). In each iteration, 4 parts are used to train the model, and the remaining part is used for testing (validation). This ensures every part of the data is used for testing exactly once.**

`evaluation.GetCrossValidatedMetric()`: **It calculates the average accuracy and variance across the cross-validation folds, providing a summary of how well the model performs.**

1. **Coss-Validation Setup**:
   - Divide the dataset into **k** folds.
2. **For Each Fold**:
   - **Train-Prune Split**: Use **k-1 folds** as training data. This training data is further split into **train** (e.g., 60%) and **pruning** (e.g., 40%) sets.
   - **Train the Tree**: Build the decision tree using the training portion (60% of the fold).
   - **Prune the Tree**: Prune the tree using the pruning set (40% of the fold).
   - **Evaluate on the Validation Fold**: Test the pruned tree on the remaining fold (the validation fold) that was not used in training or pruning.
3. **Repeat**:
   - Repeat this process **k** times, using a different fold as the validation set each time.
4. **Averaging Results**:
   - After cross-validation, you average the results of the validation fold performance across all iterations to estimate the overall model accuracy.

```
package main

import (
   "fmt"
   "log"
   "math"
   "github.com/sjwhitworth/golearn/base"
   "github.com/sjwhitworth/golearn/evaluation"
   "github.com/sjwhitworth/golearn/trees"
   "golang.org/x/exp/rand"
)

func main() {

   // Read in the iris data set into golearn "instances".
   irisData, err := base.ParseCSVToInstances("iris.csv", true)
   if err != nil {
       log.Fatal(err)
   }
   // This is to seed the random processes involved in building the
   // decision tree.
   rand.Seed(44111342)
```

```go
    // We will use the ID3 algorithm to build our decision tree.
Also, we
    // will start with a parameter of 0.6 that controls the
train-prune split.
    tree := trees.NewID3DecisionTree(0.6)
    // Use cross-fold validation to successively train and evaluate
the model
    // on 5 folds of the data set.
    cv, err :=

evaluation.GenerateCrossFoldValidationConfusionMatrices(irisData,
tree, 5)
    if err != nil {
        log.Fatal(err)
    }
    // Get the mean, variance and standard deviation of the accuracy
for the
    // cross validation.
    mean, variance := evaluation.GetCrossValidatedMetric(cv,
        evaluation.GetAccuracy)
    stdev := math.Sqrt(variance)

    // Output the cross metrics to standard out.
    fmt.Printf("\nAccuracy\n%.2f (+/- %.2f)\n\n", mean, stdev*2)
}
```

Exercise – 9: Develop Clustering models using Go such as

Demonstrate K-Means clustering method using Go

Theory

`os.Open()`: This function opens the `fleet_data.csv` file for reading. If the file cannot be opened, the program exits with an error.
`csv.NewReader()`: This function creates a reader to parse the CSV file. It sets the expected number of fields per record to 3 to ensure that each row in the file is correctly read.
`strconv.ParseFloat()`: This function converts string values from the CSV into floating-point numbers. It allows numerical data (like coordinates) to be used for k-means clustering.
`gokmeans.Node`: This represents a data point for clustering. Each point consists of two features (in this case, two float values) that are used to group the data into clusters.

`gokmeans.Train()`: This function applies the k-means clustering algorithm to the data. It divides the data into 2 clusters (`k=2`) and runs the algorithm for a maximum of 50 iterations to find the best cluster centers (centroids).
Centroids: After training, the centroids represent the center points of the two clusters, which are printed to show the result of the clustering process.

```go
package main

import (
   "encoding/csv"
   "fmt"
   "io"
   "log"
   "os"
   "strconv"

   "github.com/mash/gokmeans"
)

func main() {

   // Open the driver dataset file.
   f, err := os.Open("fleet_data.csv")
   if err != nil {
       log.Fatal(err)
   }
   defer f.Close()

   // Create a new CSV reader.
   r := csv.NewReader(f)
   r.FieldsPerRecord = 3

   // Initialize a slice of gokmeans.Node's to
   // hold our input data.
   var data []gokmeans.Node

   // Loop over the records creating our slice of
   // gokmeans.Node's.
   for {

       // Read in our record and check for errors.
       record, err := r.Read()
       if err == io.EOF {
           break
       }
       if err != nil {
           log.Fatal(err)
       }

       // Skip the header.
```

```go
        if record[0] == "Driver_ID" {
            continue
        }

        // Initialize a point.
        var point []float64

        // Fill in our point.
        for i := 1; i < 3; i++ {

            // Parse the float value.
            val, err := strconv.ParseFloat(record[i], 64)
            if err != nil {
                log.Fatal(err)
            }

            // Append this value to our point.
            point = append(point, val)
        }

        // Append our point to the data.
        data = append(data, gokmeans.Node{point[0], point[1]})
    }

    // Generate our clusters with k-means.
    success, centroids := gokmeans.Train(data, 2, 50)
    if !success {
        log.Fatal("Could not generate clusters")
    }

    // Output the centroids to stdout.
    fmt.Println("The centroids for our clusters are:")
    for _, centroid := range centroids {
        fmt.Println(centroid)
    }
}
```

Exercise – 10: Demonstrate auto regressive model using Go

Theory

1. **`dataframe.ReadCSV()`**: This function reads data from a CSV file and converts it into a Gota DataFrame, which provides structured data manipulation and analysis similar to a table, with columns and rows. In this case, it is used to store the time series data.
2. **`passengersDF.Col().Float()`**: This retrieves the values of the "log_differenced_passengers" column from the DataFrame as a slice of floating-point numbers, making them ready for numerical processing.

3. **`autoregressive()`**: This function computes an autoregressive (AR) model for the given time series data. It takes a time series and the desired lag (e.g., 2) to build a model that predicts the current value based on previous values (lags).
4. **`regression.Regression`**: This is a regression model from the `github.com/sajari/regression` package. It is used to fit a linear model where the current value of the series is predicted based on past values (lags). The regression is trained using the lagged series as independent variables.
5. **`r.Train()` and `r.Run()`**: These methods train the regression model by adding data points (current values and their lags) and then fitting the model to find the coefficients (weights) that best describe the relationship between the current value and the lagged values.
6. **Coefficients and intercept**: The output consists of the coefficients for the lagged values and the intercept term, forming an AR(2) model. The model is printed as an equation showing the relationship between the current value and lagged values.

```go
package main

import (
	"fmt"
	"log"
	"os"
	"strconv"

	"github.com/go-gota/gota/dataframe"

	"github.com/sajari/regression"
)

func main() {

	// Open the CSV file.
	passengersFile, err := os.Open("log_diff_series.csv")
	if err != nil {
		log.Fatal(err)
	}
	defer passengersFile.Close()

	// Create a dataframe from the CSV file.
	passengersDF := dataframe.ReadCSV(passengersFile)

	// Get the time and passengers as a slice of floats.
	passengers :=
passengersDF.Col("log_differenced_passengers").Float()
```

```
    // Calculate the coefficients for lag 1 and 2 and
    // our error.
    coeffs, intercept := autoregressive(passengers, 2)

    // Output the AR(2) model to stdout.
    fmt.Printf("\nlog(x(t)) - log(x(t-1)) = %0.6f + lag1*%0.6f +
lag2*%0.6f\n\n", intercept, coeffs[0], coeffs[1])
}

// autoregressive calculates an AR model for a series
// at a given order.
func autoregressive(x []float64, lag int) ([]float64, float64) {

    // Create a regresssion.Regression value needed to train
    // a model using github.com/sajari/regression.
    var r regression.Regression
    r.SetObserved("x")


    // Define the current lag and all of the intermediate lags.
    for i := 0; i < lag; i++ {
        r.SetVar(i, "x"+strconv.Itoa(i))
    }

    // Shift the series.
    xAdj := x[lag:len(x)]

    // Loop over the series creating the data set
    // for the regression.
    for i, xVal := range xAdj {

        // Loop over the intermediate lags to build up
        // our independent variables.
        laggedVariables := make([]float64, lag)

        for idx := 1; idx <= lag; idx++ {

            // Get the lagged series variables.
            laggedVariables[idx-1] = x[lag+i-idx]
        }

        // Add these points to the regression value.
        r.Train(regression.DataPoint(xVal, laggedVariables))
    }

    // Fit the regression.
    r.Run()
```

```
    // coeff hold the coefficients for our lags.
    var coeff []float64
    for i := 1; i <= lag; i++ {
        coeff = append(coeff, r.Coeff(i))
    }

    return coeff, r.Coeff(0)
}
```