**Read/Write File:**

package main

import (

"fmt"

"os"

)

func main() {

// Open the file

file, err := os.OpenFile("example.txt", os.O\_RDWR|os.O\_CREATE, 0755)

if err != nil {

fmt.Println("Error opening file:", err)

return

}

defer file.Close()

// Write to the file

newData := []byte("Hello, world!")

\_, err = file.Write(newData)

if err != nil {

fmt.Println("Error writing to file:", err)

return

}

// Read the file

data, err := os.ReadFile("example.txt")

if err != nil {

fmt.Println("Error reading file:", err)

return

} else {

fmt.Println("File content:", string(data))

}

}

**Using bufio Package:**

import (

"bufio"

"fmt"

"os"

)

func main() {

// Create or open the file

file, err := os.OpenFile("example.txt", os.O\_RDWR|os.O\_CREATE, 0644)

if err != nil {

fmt.Println("Error opening file:", err)

return

}

defer file.Close()

// Create a buffered writer

writer := bufio.NewWriter(file)

// Write data to the file

\_, err = writer.WriteString("Hello World!")

if err != nil {

fmt.Println("Error writing to file:", err)

return

}

// Flush the buffer to ensure all data is written to the file

writer.Flush()

// Reading from a file using bufio.NewScanner

readFile, err := os.Open("example.txt")

if err != nil {

fmt.Println("Error opening file:", err)

return

}

defer readFile.Close()

// Create a scanner to read from the file

scanner := bufio.NewScanner(readFile)

// Read the file line by line

for scanner.Scan() {

fmt.Println(scanner.Text())

}

// Check for scanner errors

if err := scanner.Err(); err != nil {

fmt.Println("Error scanning file:", err)

return

}

}

**Data Cleaning**

func cleanData(data [][]string) [][]string {

seen := make(map[string]struct{})

var cleanedData [][]string

for \_, row := range data {

// Merge data to create a unique identifier

identifier := fmt.Sprintf("%v", row)

// Check if the identifier has been seen before

if \_, ok := seen[identifier]; !ok {

// If not seen, add to cleaned records and mark as seen

// Skip empty rows

if len(row) == 0 || (len(row) == 1 && row[0] == "") {

continue

}

// Convert all values to lowercase and remove leading/trailing spaces

cleanedRow := make([]string, len(row))

for i, val := range row {

cleanedRow[i] = strings.TrimSpace(strings.ToLower(val))

}

//Add to cleaned records

cleanedData = append(cleanedData, cleanedRow)

//Mark as seen

seen[identifier] = struct{}{}

}

}

return cleanedData

}

**Data Transformation:**

package main

import (

"encoding/xml"

"fmt"

)

// Struct representing Person information

type Person struct {

XMLName xml.Name `xml:"person"`

Name string `xml:"name"`

Age int `xml:"age"`

Location string `xml:"location"`

}

func main() {

// Create an instance of the Person struct

person := Person{Name: "Ashok", Age: 30, Location: "New Delhi"}

// Convert struct to XML

xmlData, err := xml.MarshalIndent(person, "", " ")

if err != nil {

fmt.Println("Error marshalling XML:", err)

return

}

fmt.Println("XML representation:")

fmt.Println(string(xmlData))

// Convert XML back to struct

var newPerson Person

if err := xml.Unmarshal(xmlData, &newPerson); err != nil {

fmt.Println("Error unmarshalling XML:", err)

return

}

// Print the unmarshalled struct

fmt.Println("\nUnmarshalled struct:")

fmt.Println("Name:", newPerson.Name)

fmt.Println("Age:", newPerson.Age)

fmt.Println("Location:", newPerson.Location)

}

**Data Normalization:**

package main

import (

"fmt"

"math"

)

// MinMaxScaler performs min-max scaling on a slice of floats

func MinMaxScaler(data []float64) []float64 {

min, max := findMinMax(data)

scaled := make([]float64, len(data))

for i, val := range data {

scaled[i] = (val - min) / (max - min)

}

return scaled

}

// Function to find min and max values in a slice of floats

func findMinMax(data []float64) (float64, float64) {

min, max := math.Inf(1), math.Inf(-1)

for \_, val := range data {

if val < min {

min = val

}

if val > max {

max = val

}

}

return min, max

}

func main() {

// Original data

data := []float64{100, 550, 280, 4000, 50}

// Min-Max scaling

scaledData := MinMaxScaler(data)

// Print original and scaled data

fmt.Println("Original Data:", data) // [100 550 280 4000 50]

fmt.Println("Scaled Data:", scaledData) // [0.012658227848101266 0.12658227848101267 0.05822784810126582 1 0]

}

**EDA in Go:**

import (

"encoding/csv"

"fmt"

"log"

"os"

"sort"

"strconv"

"strings"

"gonum.org/v1/gonum/mat"

"gonum.org/v1/gonum/stat"

"gonum.org/v1/plot"

"gonum.org/v1/plot/plotter"

"gonum.org/v1/plot/vg"

)

func main() {

// Step 1: Parse the dataset

file, err := os.Open("dataset.csv")

if err != nil {

log.Fatal(err)

}

defer file.Close()

reader := csv.NewReader(file)

reader.FieldsPerRecord = -1

records, err := reader.ReadAll()

if err != nil {

log.Fatal(err)

}

// Step 2: Prepare the data for analysis

var data []float64

for \_, record := range records {

for \_, value := range record {

value = strings.TrimSpace(value)

val, err := strconv.ParseFloat(value, 64)

if err == nil {

data = append(data, val)

}

}

}

// Step 3: Descriptive statistics calculation

//stat.Quantile function requires the input data to be sorted

sort.Float64s(data)

mean := stat.Mean(data, nil)

median := stat.Quantile(0.5, stat.Empirical, data, nil)

variance := stat.Variance(data, nil)

stdDev := stat.StdDev(data, nil)

fmt.Printf("Mean: %.2f\n", mean)

fmt.Printf("Median: %.2f\n", median)

fmt.Printf("Variance: %.2f\n", variance)

fmt.Printf("Standard Deviation: %.2f\n", stdDev)

// Step 4: Data visualization (Histogram)

p := plot.New()

histogram, err := plotter.NewHist(plotter.Values(data), 16)

if err != nil {

log.Fatal(err)

}

p.Add(histogram)

p.Title.Text = "Histogram"

p.X.Label.Text = "Values"

p.Y.Label.Text = "Frequency"

err = p.Save(4\*vg.Inch, 4\*vg.Inch, "histogram.png")

if err != nil {

log.Fatal(err)

}

// Step 5: Correlation analysis (Dummy example)

// Create a dummy correlation matrix

correlationMatrix := mat.NewDense(2, 2, []float64{1.0, 0.5, 0.5, 1.0})

fmt.Println("Correlation Matrix:")

fmt.Printf("%v\n", mat.Formatted(correlationMatrix))

// Step 6: Missing values handling (Dummy example)

// Check for missing values

missingValues := 0

for \_, val := range data {

if val == 0 { // Assuming 0 indicates missing value

missingValues++

}

}

fmt.Printf("Number of missing values: %d\n", missingValues)

// Step 7: Outlier detection (Dummy example)

outliers := make([]float64, 0)

for \_, val := range data {

if val > 3\*stdDev { // Assuming anything beyond 3 standard deviations is an outlier

outliers = append(outliers, val)

}

}

fmt.Printf("Outliers: %v\n", outliers)

}

**Time Series Analysis**

package main

import (

"fmt"

"gonum.org/v1/gonum/mat"

"gonum.org/v1/gonum/optimize"

)

func main() {

// Sample time series data for the last month (assuming daily data)

// For example, you might fetch data from an API or a CSV file

x := []float64{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}

y := []float64{143.0, 143.2, 142.4, 142.6, 142.8, 143.5, 141.0, 141.4, 142.5, 142.3, 142.5, 143.0, 143.3, 143.4, 143.7, 144.0, 144.6, 144.3, 144.5, 144.6, 144.2, 144.6, 144.2, 144.5, 144.2, 144.7, 145.2, 145.0, 145.2, 144.8}

// Create the design matrix

X := mat.NewDense(len(x), 2, nil)

for i := 0; i < len(x); i++ {

X.Set(i, 0, 1) // Bias term

X.Set(i, 1, x[i]) // Feature

}

// Create the target vector

Y := mat.NewVecDense(len(y), y)

// Initial the parameters

theta := []float64{0, 0}

// Define the linear regression objective function

objective := func(theta []float64) float64 {

thetaVec := mat.NewVecDense(len(theta), theta)

var diff mat.VecDense

diff.MulVec(X, thetaVec)

diff.SubVec(&diff, Y)

return mat.Dot(&diff, &diff)

}

// Create the problem for optimization

problem := optimize.Problem{

Func: objective,

}

// Initialize optimization settings

settings := &optimize.Settings{}

// Minimize the objective function using the default BFGS method

result, err := optimize.Minimize(problem, theta, settings, nil)

if err != nil {

fmt.Println("Error:", err)

return

}

// Get the optimized parameters

thetaOpt := mat.NewVecDense(len(result.X), result.X)

// Predict the stock price for tomorrow

tomorrow := 31.0 // Assuming the next day

prediction := thetaOpt.At(0, 0) + thetaOpt.At(1, 0)\*tomorrow

fmt.Printf("Predicted stock price for tomorrow: %.2f\n", prediction)

}