**CSA4724-DEEP LEARNING FOR NUTRITION ANALYSIS**

**Prasanna.C(192124160)**

**24.06.2024(DAY 3)**

**7(b)**

**Program:**

import numpy as np

import matplotlib.pyplot as plt

def mean\_squared\_error(y\_true, y\_predicted):

    # Calculating the loss or cost

    cost = np.sum((y\_true - y\_predicted)\*\*2) / len(y\_true)

    return cost

# Gradient Descent Function

# Here iterations, learning\_rate, stopping\_threshold are hyperparameters that can be tuned

def gradient\_descent(x, y, iterations=1000, learning\_rate=0.0001, stopping\_threshold=1e-6):

    # Initializing weight, bias, learning rate and iterations

    current\_weight = 0.1

    current\_bias = 0.01

    n = float(len(x))

    costs = []

    weights = []

    previous\_cost = None

    # Estimation of optimal parameters

    for i in range(iterations):

        # Making predictions

        y\_predicted = (current\_weight \* x) + current\_bias

        # Calculating the current cost

        current\_cost = mean\_squared\_error(y, y\_predicted)

        # If the change in cost is less than or equal to stopping\_threshold we stop the gradient descent

        if previous\_cost and abs(previous\_cost - current\_cost) <= stopping\_threshold:

            break

        previous\_cost = current\_cost

        costs.append(current\_cost)

        weights.append(current\_weight)

        # Calculating the gradients

        weight\_derivative = -(2/n) \* sum(x \* (y - y\_predicted))

        bias\_derivative = -(2/n) \* sum(y - y\_predicted)

        # Updating weights and bias

        current\_weight = current\_weight - (learning\_rate \* weight\_derivative)

        current\_bias = current\_bias - (learning\_rate \* bias\_derivative)

        # Printing the parameters for each 100th iteration

        if i % 100 == 0:

            print(f"Iteration {i+1}: Cost {current\_cost}, Weight {current\_weight}, Bias {current\_bias}")

    # Visualizing the weights and cost for all iterations

    plt.figure(figsize=(8, 6))

    plt.plot(weights, costs)

    plt.scatter(weights, costs, marker='o', color='red')

    plt.title("Cost vs Weights")

    plt.ylabel("Cost")

    plt.xlabel("Weight")

    plt.show()

    return current\_weight, current\_bias

def main():

    # Data

    X = np.array([52.50234527, 63.42680403, 81.53035803, 47.47563963, 89.81320787,

           55.14218841, 52.21179669, 39.29956669, 48.10504169, 52.55001444,

           45.41973014, 54.35163488, 44.1640495 , 58.16847072, 56.72720806,

           48.95588857, 44.68719623, 60.29732685, 45.61864377, 38.81681754])

    Y = np.array([41.70700585, 78.77759598, 82.5623823 , 91.54663223, 77.23092513,

           78.21151827, 79.64197305, 59.17148932, 75.3312423 , 71.30087989,

           55.16567715, 82.47884676, 62.00892325, 75.39287043, 81.43619216,

           60.72360244, 82.89250373, 97.37989686, 48.84715332, 56.87721319])

    # Estimating weight and bias using gradient descent

    estimated\_weight, estimated\_bias = gradient\_descent(X, Y, iterations=2000)

    print(f"Estimated Weight: {estimated\_weight}\nEstimated Bias: {estimated\_bias}")

    # Making predictions using estimated parameters

    Y\_pred = estimated\_weight \* X + estimated\_bias

    # Plotting the regression line

    plt.figure(figsize=(8, 6))

    plt.scatter(X, Y, color='orange')

    plt.plot([min(X), max(X)], [min(Y\_pred), max(Y\_pred)], color='blue', markerfacecolor='red',

             markersize=10, linestyle='dashed')

    plt.xlabel("X")

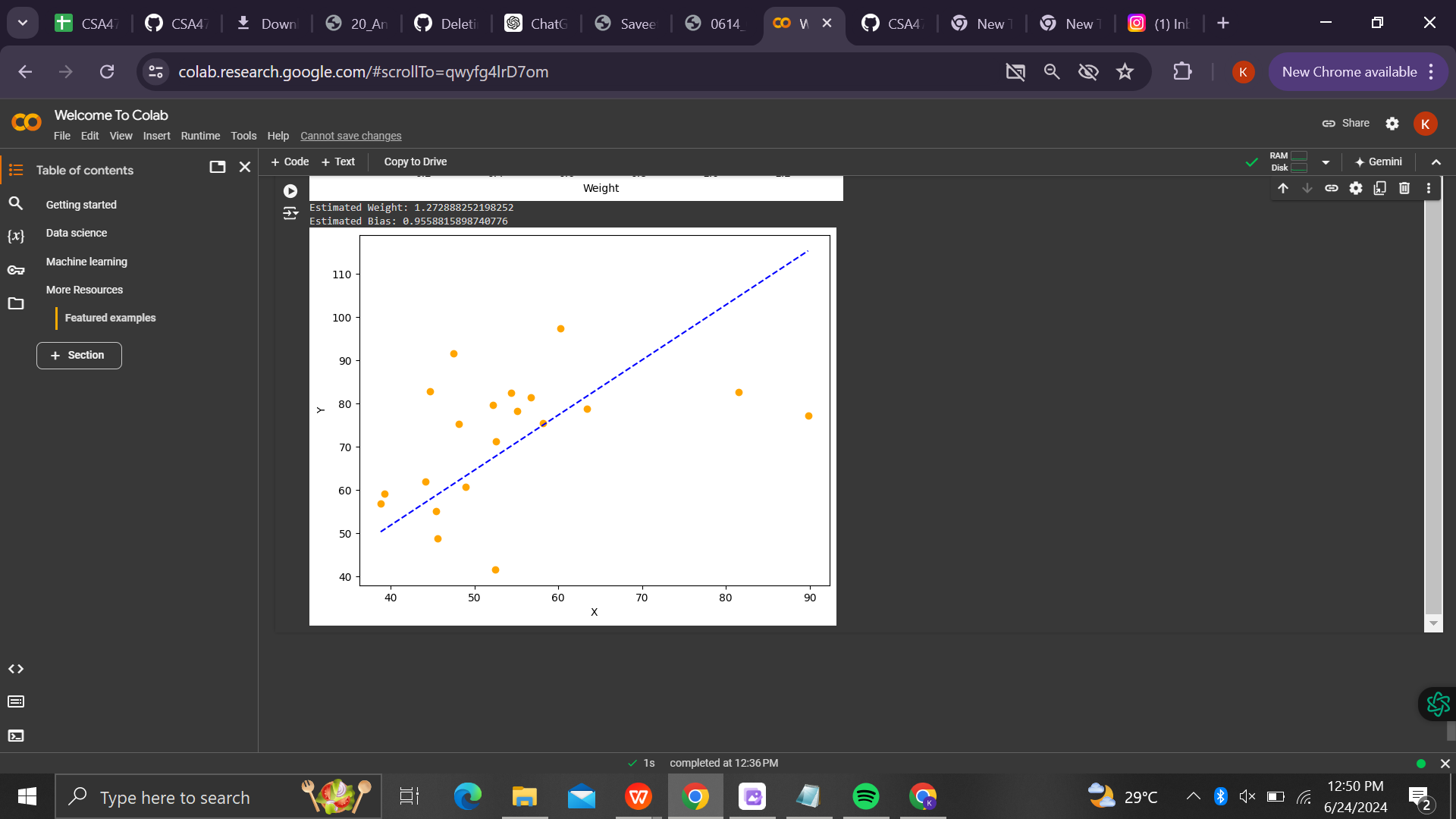
    plt.ylabel("Y")

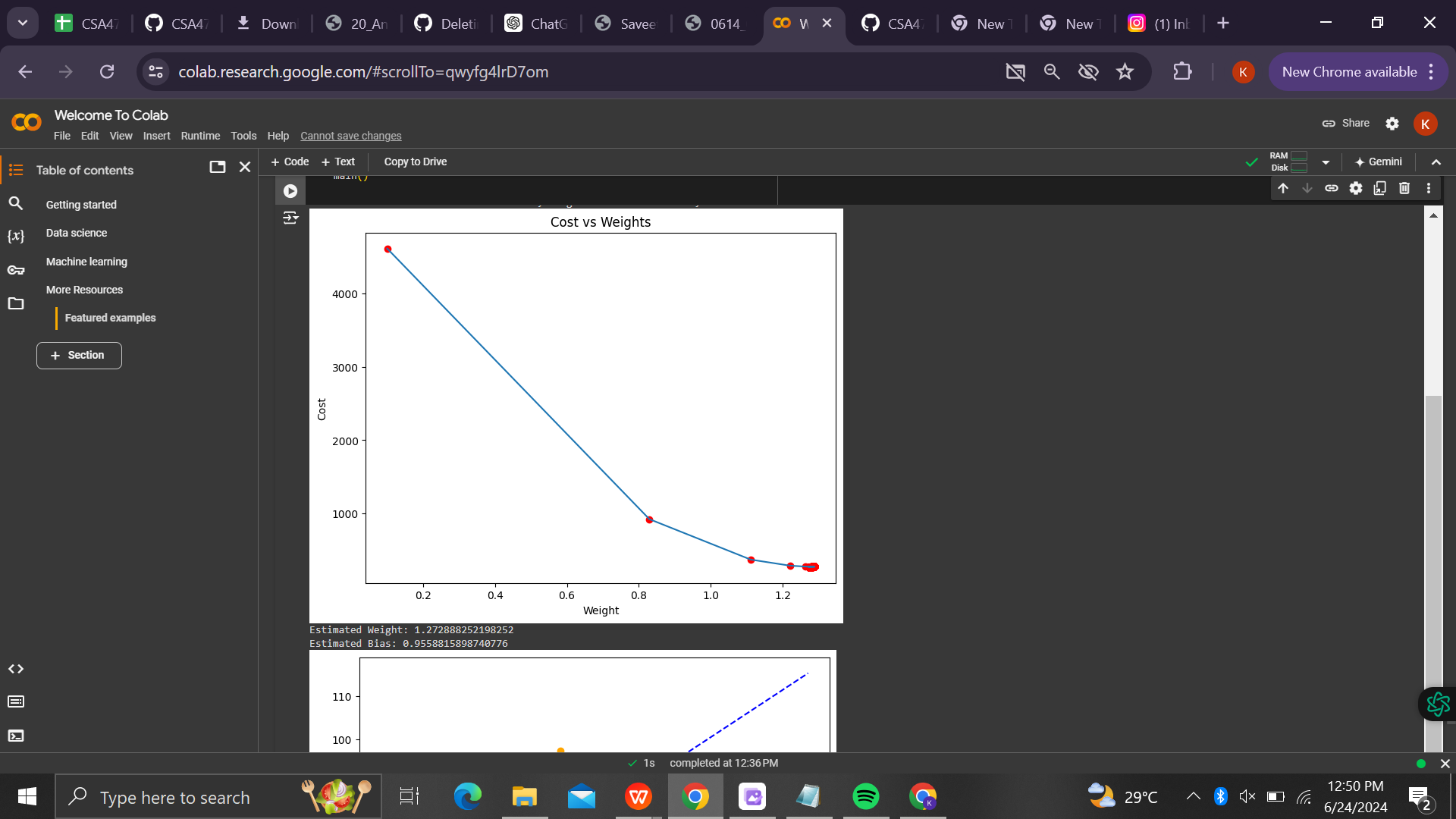
    plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

    main()

**Output:**

****

****

**7(a)**

**Program:**

import numpy as np

import matplotlib.pyplot as plt

def mean\_squared\_error(y\_true, y\_predicted):

# Calculating the loss or cost

cost = np.sum((y\_true - y\_predicted)\*\*2) / len(y\_true)

return cost

# Gradient Descent Function

# Here iterations, learning\_rate, stopping\_threshold are hyperparameters that can be tuned

def gradient\_descent(x, y, iterations=1000, learning\_rate=0.0001, stopping\_threshold=1e-6):

# Initializing weight, bias, learning rate and iterations

current\_weight = 0.1

current\_bias = 0.01

n = float(len(x))

costs = []

weights = []

previous\_cost = None

# Estimation of optimal parameters

for i in range(iterations):

# Making predictions

y\_predicted = (current\_weight \* x) + current\_bias

# Calculating the current cost

current\_cost = mean\_squared\_error(y, y\_predicted)

# If the change in cost is less than or equal to stopping\_threshold we stop the gradient descent

if previous\_cost and abs(previous\_cost - current\_cost) <= stopping\_threshold:

break

previous\_cost = current\_cost

costs.append(current\_cost)

weights.append(current\_weight)

# Calculating the gradients

weight\_derivative = -(2/n) \* sum(x \* (y - y\_predicted))

bias\_derivative = -(2/n) \* sum(y - y\_predicted)

# Updating weights and bias

current\_weight = current\_weight - (learning\_rate \* weight\_derivative)

current\_bias = current\_bias - (learning\_rate \* bias\_derivative)

# Printing the parameters for each 100th iteration

if (i + 1) % 100 == 0:

print(f"Iteration {i+1}: Cost {current\_cost}, Weight {current\_weight}, Bias {current\_bias}")

# Visualizing the weights and cost for all iterations

plt.figure(figsize=(8, 6))

plt.plot(weights, costs)

plt.scatter(weights, costs, marker='o', color='red')

plt.title("Cost vs Weights")

plt.ylabel("Cost")

plt.xlabel("Weight")

plt.show()

return current\_weight, current\_bias

def main():

# Data

X = np.array([32.50234527, 53.42680403, 61.53035803, 47.47563963, 59.81320787,

55.14218841, 52.21179669, 39.29956669, 48.10504169, 52.55001444,

45.41973014, 54.35163488, 44.1640495 , 58.16847072, 56.72720806,

48.95588857, 44.68719623, 60.29732685, 45.61864377, 38.81681754])

Y = np.array([31.70700585, 68.77759598, 62.5623823 , 71.54663223, 87.23092513,

78.21151827, 79.64197305, 59.17148932, 75.3312423 , 71.30087989,

55.16567715, 82.47884676, 62.00892325, 75.39287043, 81.43619216,

60.72360244, 82.89250373, 97.37989686, 48.84715332, 56.87721319])

# Estimating weight and bias using gradient descent

estimated\_weight, estimated\_bias = gradient\_descent(X, Y, iterations=2000)

print(f"Estimated Weight: {estimated\_weight}\nEstimated Bias: {estimated\_bias}")

# Making predictions using estimated parameters

Y\_pred = estimated\_weight \* X + estimated\_bias

# Plotting the regression line

plt.figure(figsize=(8, 6))

plt.scatter(X, Y, color='red')

plt.plot([min(X), max(X)], [min(Y\_pred), max(Y\_pred)], color='blue', markerfacecolor='red',

markersize=10, linestyle='dashed')

plt.xlabel("X")

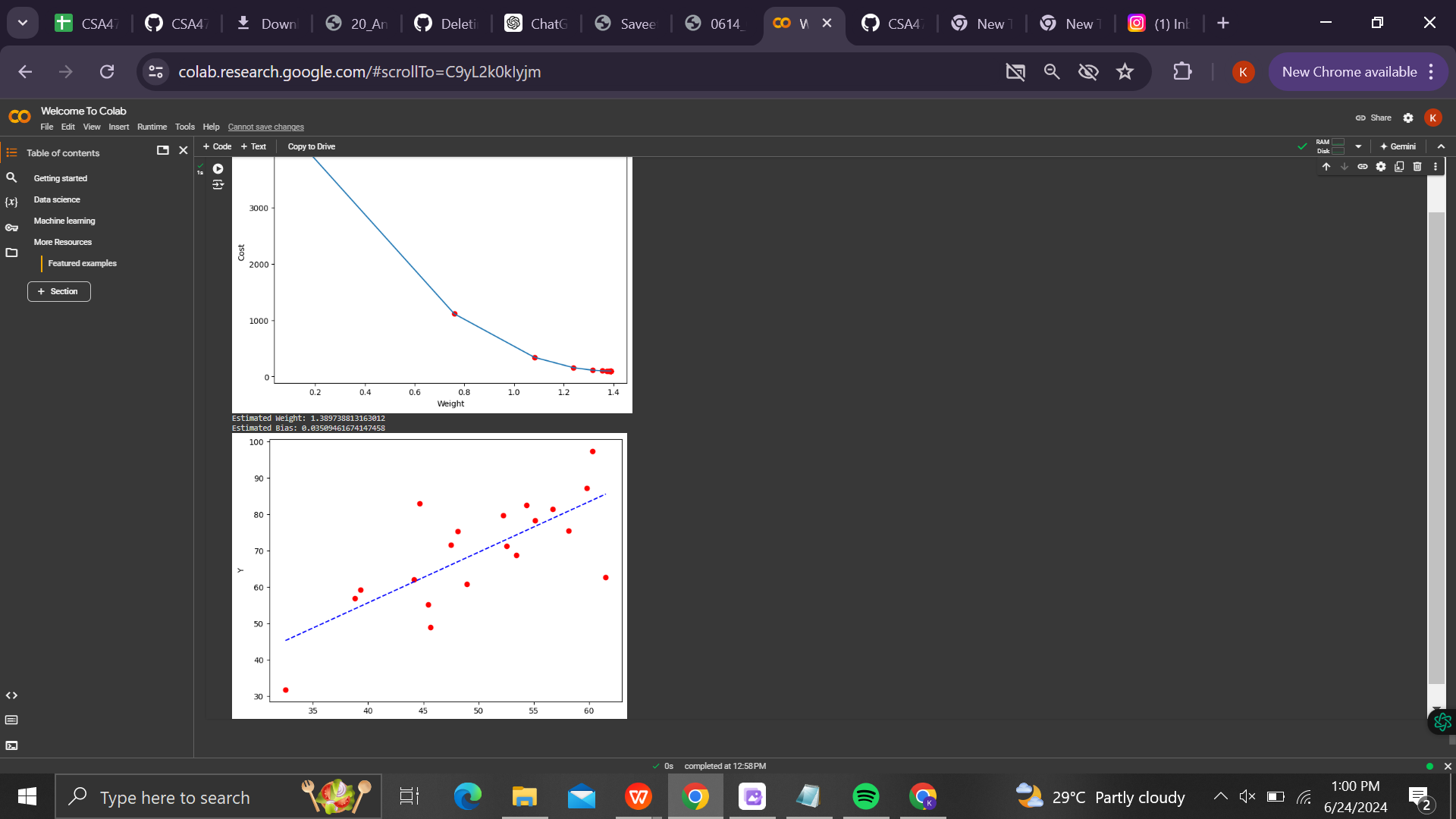
plt.ylabel("Y")

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Output:**

****

**8(a)**

**Program:**

import tensorflow as tf

from tensorflow.keras import datasets, layers, models

import matplotlib.pyplot as plt

from sklearn.metrics import classification\_report, confusion\_matrix

import seaborn as sns

import numpy as np

# Load and preprocess the dataset

(X\_train, y\_train), (X\_test, y\_test) = datasets.cifar10.load\_data()

# Normalize the images to the range [0, 1]

X\_train, X\_test = X\_train / 255.0, X\_test / 255.0

# Verify the data

class\_names = ['airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck']

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

for i in range(25):

    plt.subplot(5,5,i+1)

    plt.xticks([])

    plt.yticks([])

    plt.grid(False)

    plt.imshow(X\_train[i], cmap=plt.cm.binary)

    plt.xlabel(class\_names[y\_train[i][0]])

plt.show()

# Build the CNN model

model = models.Sequential([

    layers.Conv2D(32, (3, 3), activation='relu', input\_shape=(32, 32, 3)),

    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),

    layers.MaxPooling2D((2, 2)),

    layers.Conv2D(64, (3, 3), activation='relu'),

    layers.Flatten(),

    layers.Dense(64, activation='relu'),

    layers.Dense(10)

])

# Compile the model

model.compile(optimizer='adam',

              loss=tf.keras.losses.SparseCategoricalCrossentropy(from\_logits=True),

              metrics=['accuracy'])

# Train the model

history = model.fit(X\_train, y\_train, epochs=10,

                    validation\_data=(X\_test, y\_test))

# Evaluate the model

test\_loss, test\_acc = model.evaluate(X\_test, y\_test, verbose=2)

print(f"\nTest accuracy: {test\_acc}")

# Make predictions

y\_pred = model.predict(X\_test)

y\_pred\_classes = np.argmax(y\_pred, axis=1)

# Classification report

print("\nClassification Report:\n")

print(classification\_report(y\_test, y\_pred\_classes, target\_names=class\_names))

# Confusion matrix

conf\_matrix = confusion\_matrix(y\_test, y\_pred\_classes)

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

sns.heatmap(conf\_matrix, annot=True, fmt='d', xticklabels=class\_names, yticklabels=class\_names, cmap=plt.cm.Blues)

plt.xlabel('Predicted')

plt.ylabel('True')

plt.title('Confusion Matrix')

plt.show()

# Visualize some predictions

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

for i in range(25):

    plt.subplot(5, 5, i + 1)

    plt.xticks([])

    plt.yticks([])

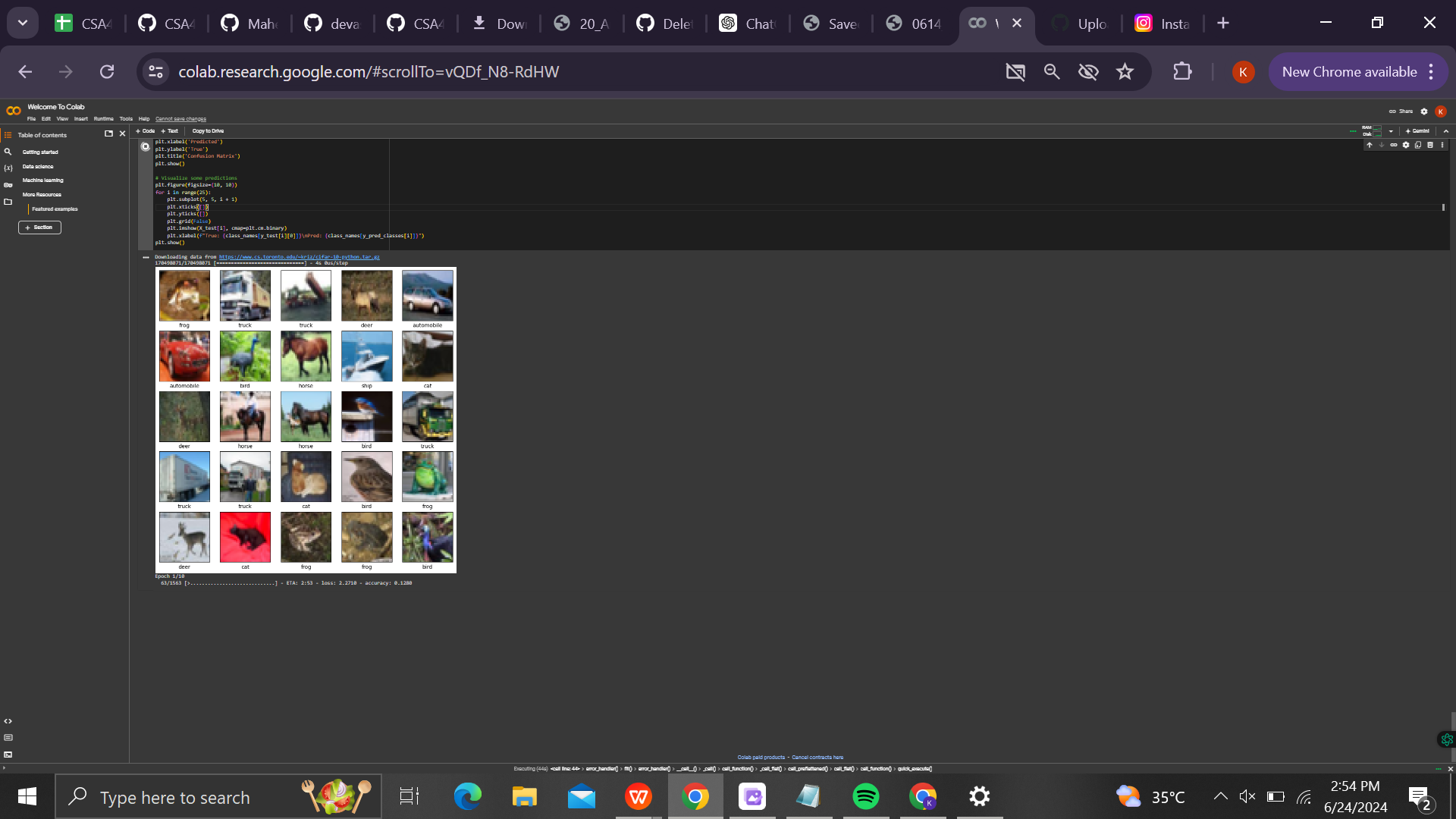
    plt.grid(False)

    plt.imshow(X\_test[i], cmap=plt.cm.binary)

    plt.xlabel(f"True: {class\_names[y\_test[i][0]]}\nPred: {class\_names[y\_pred\_classes[i]]}")

plt.show()

**Output:**

****