TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA ĐIỆN – ĐIỆN TỬ

GROUP 06

PREDICTIVE MODELING WITH LINEAR AND LOGISTIC REGRESSION STUDENT PERFORMANCE ESTIMATION AND ADVERTISEMENT CLICK PREDICTION

PROCESS 2 ASSIGNMENT REPORT ARTIFICIAL INTELLIGENCE

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2025

TỔNG LIÊN ĐOÀN LAO ĐỘNG VIỆT NAM
TRƯỜNG ĐẠI HỌC TÔN ĐỨC THẮNG
KHOA ĐIỆN – ĐIỆN TỬ

GROUP 06

PREDICTIVE MODELING WITH LINEAR AND LOGISTIC REGRESSION STUDENT PERFORMANCE ESTIMATION AND ADVERTISEMENT CLICK PREDICTION

PROCESS 2 ASSIGNMENT REPORT ARTIFICIAL INTELLIGENCE

Instructor
MSc. Thái Lâm Cường Quốc

THÀNH PHỐ HỒ CHÍ MINH, NĂM 2025

GROUP LIST

NO	Name	ID	Mission	Status	Signature
1	Bùi Hoàng Sơn	42200134	Code linear regression	20%	
2	Võ Hoàng Siêu	42200419	Code logistic regression	20%	
3	Nguyễn Văn Anh Quốc	42300357	Data analysis	20%	
4	Đinh Minh Quang	42200441	Prepare reports	20%	
5	Nguyễn Lâm Anh Quan	42200497	Prepare reports	20%	
Total				100%	

ABSTRACT

This project explores the application of regression techniques in machine learning by implementing two predictive models from scratch using Python. The first part applies Linear Regression to predict students' academic performance based on their exam scores and related features. The second part utilizes Logistic Regression to classify whether a user will click on an online advertisement given demographic and behavioral data. Both models were trained and tested on real-world style datasets, with appropriate preprocessing including normalization and train-test splitting. Evaluation metrics such as Mean Squared Error for linear regression, and Accuracy, Precision, Recall, F1-score, for logistic regression were employed to assess performance. The experimental results demonstrate that regression models can achieve high predictive accuracy when combined with effective data preprocessing and parameter optimization. This work highlights the strengths and practical applications of regression analysis in both continuous and binary prediction problems.

TABLE OF CONTENTS

TABLE OF FIGURES

TABLE OF TABLES

TABLE OF ABBREVIATIONS

ML	Machine Learning
MSE	Mean Squared Error
RMSE	Root Mean Squared Error
MAE	Mean Absolute Error
BCE	Binary Cross-Entropy (Loss function for Logistic Regression)
TP	True Positive
TN	True Negative
FP	False Positive
FN	False Negative
ACC	Accuracy
PREC	Precision
REC	Recall
F1	F1-Score (Harmonic mean of Precision and Recall)
AUC	Area Under the Curve (ROC)
ROC	Receiver Operating Characteristic curve
EDA	Exploratory Data Analysis

CHAPTER 1: INTRODUCTION

1.1 Linear regression

Linear regression is a type of supervised machine-learning algorithm that learns from the labeled datasets and maps the data points with most optimized linear functions which can be used for prediction on new datasets. It assumes that there is a linear relationship between the input and output, meaning the output changes at a constant rate as the input changes. This relationship is represented by a straight line.

For example we want to predict a student's exam score based on how many hours they studied. We observe that as students study more hours, their scores go up. In the example of predicting exam scores based on hours studied.

Independent variable (input): Hours studied because it's the factor we control or observe.

Dependent variable (output): Exam score because it depends on how many hours were studied.

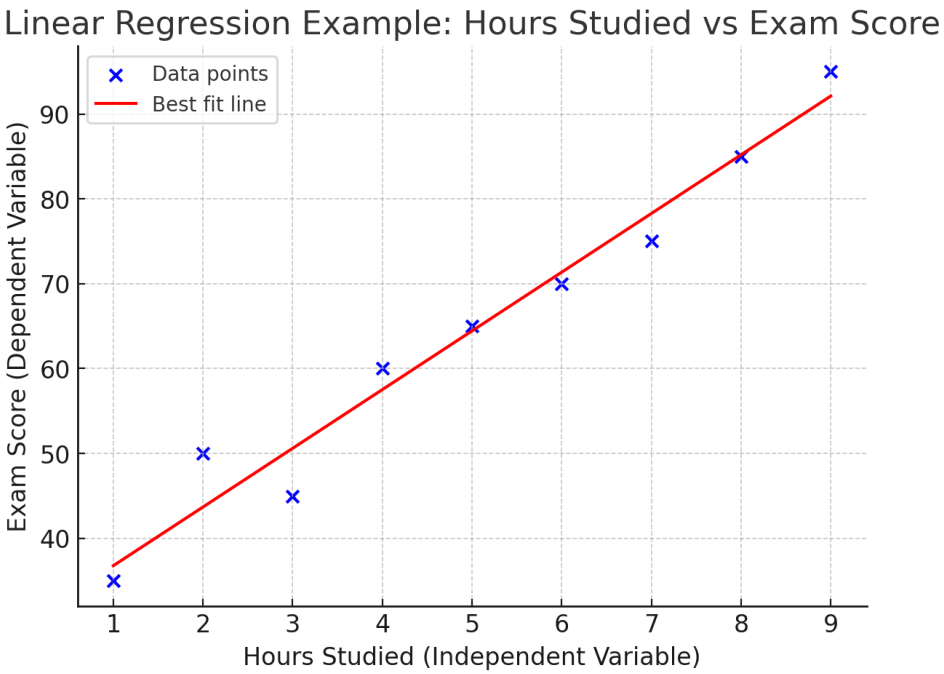

Figure 1. 1: Best fit line.

1.1.1 Best fit line in linear regression

In linear regression, the best-fit line is the straight line that most accurately represents the relationship between the independent variable (input) and the dependent variable (output). It is the line that minimizes the difference between the actual data points and the predicted values from the model.

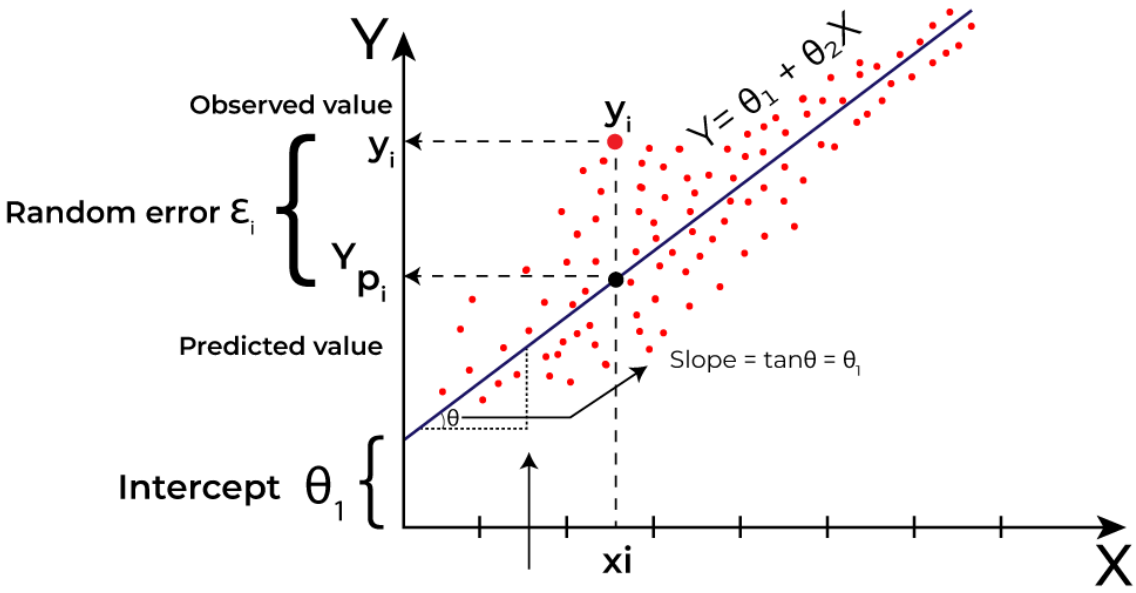

Figure 1. 2: Linear regression.

Here Y is called a dependent or target variable and X is called an independent variable also known as the predictor of Y. There are many types of functions or modules that can be used for regression. A linear function is the simplest type of function. Here, X may be a single feature or multiple features representing the problem.

For simple linear regression (with one independent variable), the best-fit line is represented by the equation.

$$y = mx + b$$

Where:

y is the predicted value (dependent variable).

x is the input (independent variable).

m is the slope of the line (how much y changes when x changes).

b is the intercept (the value of y when x = 0).

The best-fit line will be the one that optimizes the values of m (slope) and b (intercept) so that the predicted y values are as close as possible to the actual data points.

To find the best-fit line, we use a method called Least Squares. The idea behind this method is to minimize the sum of squared differences between the actual values (data points) and the predicted values from the line. These differences are called residuals.

The formula for residuals is:

$$Residual = y_i - \hat{y}_i$$

Where:

y_i is the actual observed value

$\hat{y}_i$ is the predicted value from the line for that

The least squares method minimizes the sum of the squared residuals:

$$Sumofsquarederrors(SSE) = \sum (y_i - \hat{y}_i)^2$$

This method ensures that the line best represents the data where the sum of the squared differences between the predicted values and actual values is as small as possible.

1.1.2 Types of linear regression

When there is only one independent feature it is known as Simple Linear Regression or Univariate Linear Regression and when there are more than one feature it is known as Multiple Linear Regression or Multivariate Regression.

Simple linear regression is used when we want to predict a target value (dependent variable) using only one input feature (independent variable). It assumes a straight-line relationship between the two.

Formula:

$$\hat{y} = \theta_0 + \theta_0 x$$

Where:

$\hat{y}$ is the predicted value

x is the input (independent variable)

θ_0 is the intercept (value of when x=0)

θ_1 is the slope or coefficient (how much changes with one unit of x)

Multiple linear regression involves more than one independent variable and one dependent variable. The equation for multiple linear regression is:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

Where:

$\hat{y}$ is the predicted value.

$x_1, x_2, \ldots, x_n$ are the independent variables.

$\theta_1, \theta_2, \ldots, \theta_n$ are the coefficients (weights) corresponding to each predictor.

θ_0 is the intercept.

The goal of the algorithm is to find the best Fit Line equation that can predict the values based on the independent variables.

In regression set of records are present with X and Y values and these values are used to learn a function so if you want to predict Y from an unknown X this learned function can be used. In regression we have to find the value of Y, So, a function is required that predicts continuous Y in the case of regression given X as independent features.

1.1.3 Cost function for linear regression

As we have discussed earlier about best fit line in linear regression, its not easy to get it easily in real life cases so we need to calculate errors that affects it. These errors need to be calculated to mitigate them. The difference between the predicted value $\hat{Y}$ and the true value Y and it is called cost function or the loss function.

In Linear Regression, the Mean Squared Error (MSE) cost function is employed, which calculates the average of the squared errors between the predicted values $\hat{y_i}$ and the actual values y_i. The purpose is to determine the optimal values for the intercept θ_1 and the coefficient of the input feature θ_2 providing the best-fit line for the given data points. The linear equation expressing this relationship is $\hat{y_i} = \theta_1 + \theta_1 x_i$

MSE function can be calculated as:

$$Cost\ function\ (J) = \frac{1}{n}\sum_n^i (\hat{y}_i - y_i)^2$$

Utilizing the MSE function, the iterative process of gradient descent is applied to update the values of θ1 and θ2. This ensures that the MSE value converges to the global minima, signifying the most accurate fit of the linear regression line to the dataset.

This process involves continuously adjusting the parameters θ1 and θ2 based on the gradients calculated from the MSE. The final result is a linear regression line that minimizes the overall squared differences between the predicted and actual values, providing an optimal representation of the underlying relationship in the data.

1.1.4 Gradient descent for linear regression

Gradient descent is a optimization algorithm used in linear regression to find the best fit line tohe data. It works by gradually by adjusting the line's slope and intercept to reduce the difference between actual and predicted values. This process helps the model make accurate predictions by minimizing errors step by step. In this article we will see more about Gradient Descent and its core concepts in detail.

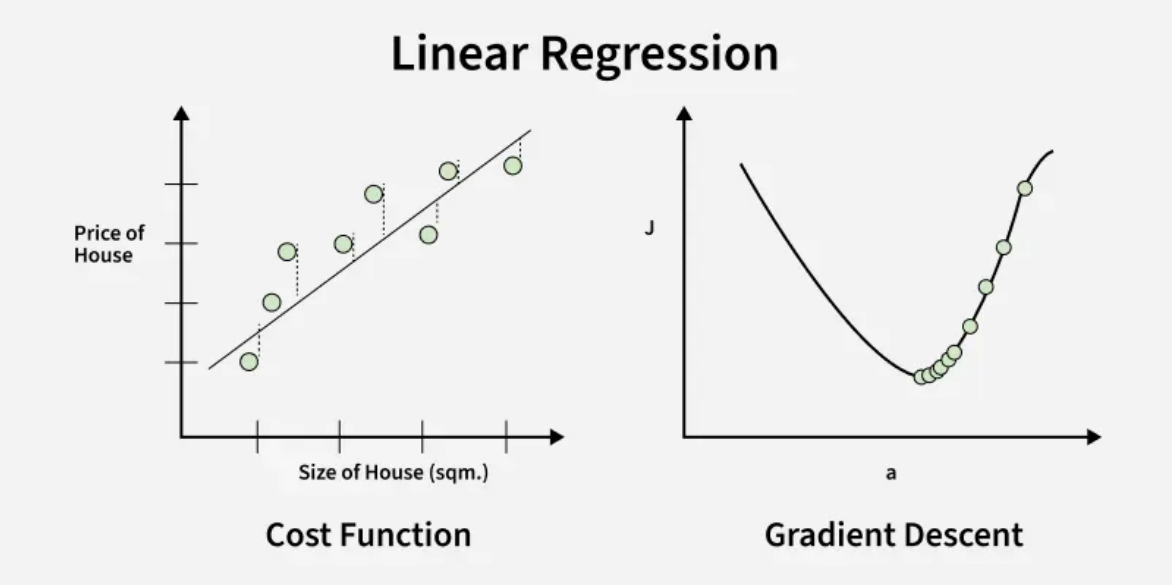

Figure 1. 3: Cost function and Gradient Descent.

Above image shows two graphs, left one plots house prices against size to show errors measured by the cost function while right one shows how gradient descent moves downhill on the cost curve to minimize error by updating parameters step by step.

Initializing Parameters: Start with random initial values for the slope (m) and intercept (b).

Calculate the Cost Function: Measure the error using the Mean Squared Error (MSE):

$$J(m,b) = \frac{1}{n}\sum_{i=1}^{n}\left(y_i - (mx_i + b)\right)^2$$

Compute the Gradient: Calculate how much the cost function changes with respect to m and b.

For slope m:

$$\frac{\partial J}{\partial m} = -\frac{2}{n}\sum_{i=1}^{n} x_i\left(y_i - (mx_i + b)\right)$$

For intercept b:

$$\frac{\partial J}{\partial b} = -\frac{2}{n}\sum_{i=1}^{n}\left(y_i - (mx_i + b)\right)$$

Update Parameters: Change *m* and *b* to reduce the error:

For slope m:

$$m = m - \alpha \cdot \frac{\partial J}{\partial m}$$

For slope b:

$$b = b - \alpha \cdot \frac{\partial J}{\partial b}$$

Here α is the learning rate that controls the size of each update.

1.1.5 Evaluation metrics for linear regression

A variety of evaluation measures can be used to determine the strength of any linear regression model. These assessment metrics often give an indication of how well the model is producing the observed outputs.

Mean Squared Error (MSE) is an evaluation metric that calculates the average of the squared differences between the actual and predicted values for all the data points. The difference is squared to ensure that negative and positive differences don't cancel each other out.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

The square root of the residuals' variance is the **Root Mean Squared Error**. It describes how well the observed data points match the expected values or the model's absolute fit to the data.

In mathematical notation, it can be expressed as:

$$RMSE = \sqrt{\frac{RSS}{n}} = \sqrt{\frac{\sum_{i=2}^{n}\left(y_i^{actual} - y_i^{\text{predicted}}\right)^2}{n}}$$

Coefficient of Determination (R-Squared) is a statistic that indicates how much variation the developed model can explain or capture. It is always in the range of 0 to 1. In general, the better the model matches the data, the greater the R-squared number.

In mathematical notation, it can be expressed as:

$$R^2 = 1 - \left(\frac{RSS}{TSS}\right)$$

1.2 Logistic regression

Logistic regression is a supervised machine learning algorithm used for classification problems. Unlike linear regression which predicts continuous values it predicts the probability that an input belongs to a specific class. It is used for binary classification where the output can be one of two possible categories such as Yes/No, True/False or 0/1. It uses sigmoid function to convert inputs into a probability value between 0 and 1. In this article, we will see the basics of logistic regression and its core concepts.

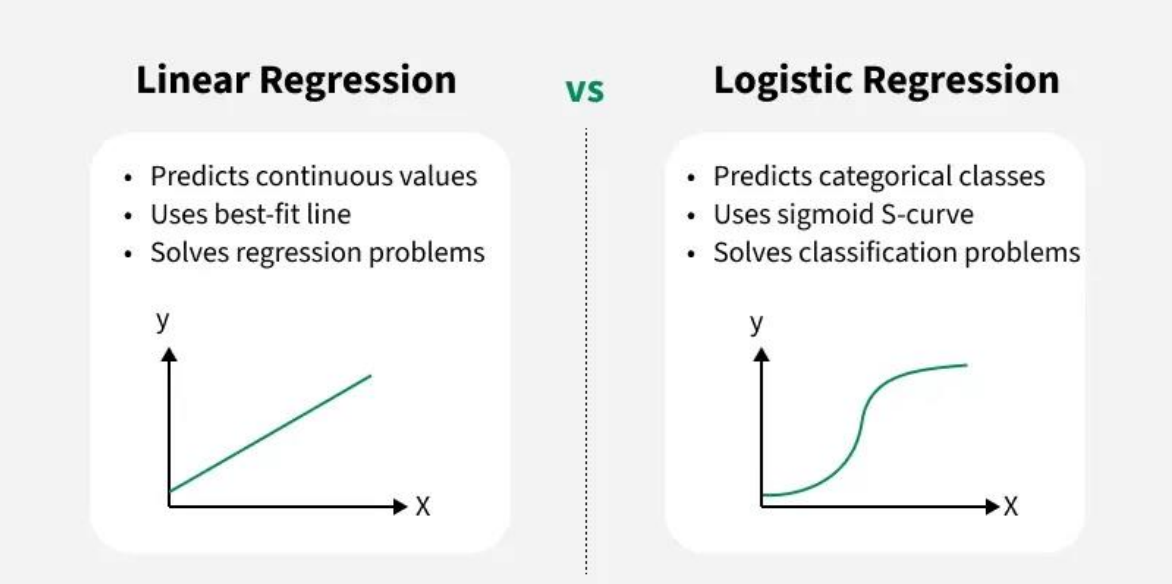

Figure 1. 4: Linear regression and Logistic regression.

1.2.1 Types of logistic regression

Logistic regression can be classified into three main types based on the nature of the dependent variable:

Binomial logistic regression: This type is used when the dependent variable has only two possible categories. Examples include Yes/No, Pass/Fail or 0/1. It is the most common form of logistic regression and is used for binary classification problems.

Multinomial logistic regression: This is used when the dependent variable has three or more possible categories that are not ordered. For example, classifying animals into categories like "cat," "dog" or "sheep." It extends the binary logistic regression to handle multiple classes.

Ordinal logistic regression: This type applies when the dependent variable has three or more categories with a natural order or ranking. Examples include ratings like "low," "medium" and "high." It takes the order of the categories into account when modeling.

1.2.2 Sigmoid function

The sigmoid function is an important part of logistic regression which is used to convert the raw output of the model into a probability value between 0 and 1.

This function takes any real number and maps it into the range 0 to 1 forming an "S" shaped curve called the sigmoid curve or logistic curve. Because probabilities must lie between 0 and 1, the sigmoid function is perfect for this purpose.

In logistic regression, use a threshold value usually 0.5 to decide the class label. If the sigmoid output is same or above the threshold, the input is classified as Class 1. If it is below the threshold, the input is classified as Class 0. This approach helps to transform continuous input values into meaningful class predictions.

1.2.3 Working principle of logistic regression

Logistic regression model transforms the linear regression function continuous value output into categorical value output using a sigmoid function which maps any real-valued set of independent variables input into a value between 0 and 1. This function is known as the logistic function.

Suppose we have input features represented as a matrix:

$$X = \begin{bmatrix} x_{11} & \dots & x_{1m} \\ x_{21} & \dots & x_{2m} \\ \vdots & \ddots & \vdots \\ x_{n1} & \dots & x_{nm} \end{bmatrix}$$

and the dependent variable is *Y* having only binary value 0 or 1.

$$Y = \begin{cases} 0 & if\ class\ 1 \\ 1 & if\ class\ 2 \end{cases}$$

then, apply the multi-linear function to the input variables X.

$$z = \left(\sum_{i=1}^{n} w_i x_1\right) + b$$

Here xi is the observation of X, wi = [w1, w2, w3, ⋯, wm] is the weights or coefficient and b is the bias term also known as intercept. Simply this can be represented as the dot product of weight and bias.

$$z = w \cdot X + b$$

At this stage, Z is a continuous value from the linear regression. Logistic regression then applies the sigmoid function to Z to convert it into a probability between 0 and 1 which can be used to predict the class.

Now we use the sigmoid function where the input will be z and we find the probability between 0 and 1 predicted y.

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

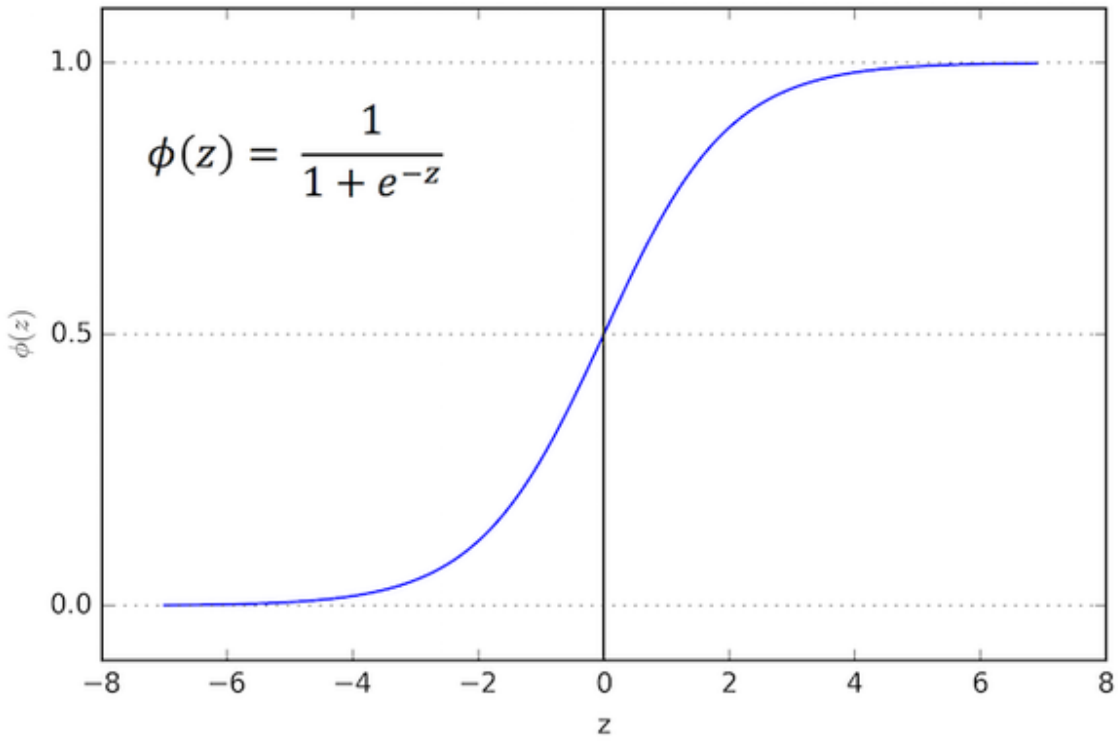

Figure 1. 5: Sigmoid function.

As shown above the sigmoid function converts the continuous variable data into the probability between 0 and 1.

1.2.4 Evaluate Logistic Regression

Evaluating the logistic regression model helps assess its performance and ensure it generalizes well to new, unseen data. The following metrics are commonly used:

Accuracy provides the proportion of correctly classified instances.

$$\text{Accuracy} = \frac{\text{True Positives} + \text{True Negatives}}{\text{Total}}$$

Precision focuses on the accuracy of positive predictions.

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}$$

Recall measures the proportion of correctly predicted positive.

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}$$

F1 score is the harmonic mean of precision and recall.

$$F1\,Score = 2 * \frac{Precision * Recall}{Precision + Recall}$$

1.3 Differences Between Linear and Logistic Regression

Logistic regression and linear regression differ in their application and output.

Linear Regression	Logistic Regression
Linear regression is used to predict the continuous dependent variable using a given set of independent variables	Logistic regression is used to predict the categorical dependent variable using a given set of independent variables
It is used for solving regression problem.	It is used for solving classification problems.
In this we predict the value of continuous variables	In this we predict values of categorical variables
In this we find best fit line.	In this we find S-Curve.
Least square estimation method is used for estimation of accuracy.	Maximum likelihood estimation method is used for Estimation of accuracy.
The output must be continuous value, such as price, age etc.	Output must be categorical value such as 0 or 1, Yes or no etc.
It required linear relationship between dependent and independent variables.	It not required linear relationship.

Table 1. 1: Linear regression and Logistic regression.

CHAPTER 2: LINEAR REGRESSION

2.1 Load dataset

The program uses pandas to load the dataset from dataset.csv

The selected features are: Hours Studied, previous Scores, sleep Hours, sample Question Papers Practiced.

The target label (Performance Index) is the value to be predicted. The total number of samples is stored in variable m.

```
# Đọc data

df = pd.read_csv("dataset.csv")
feature_names = ["Hours Studied", "Previous Scores", "Sleep Hours", "Sample Question Papers Practiced"]
X_raw = df[feature_names].values
y_raw = df["Performance Index"].values.reshape(-1, 1)
m = X_raw.shape[0]
```

Figure 2. 1: Load data.

```
Hours Studied,Previous Scores,Sleep Hours,Sample Question Papers Practiced,Performance Index
7,99,9,1,91
4,82,4,2,65
8,51,7,2,45
5,52,5,2,36
7,75,8,5,66
3,78,9,6,61
7,73,5,6,63
8,45,4,6,42
5,77,8,2,61
4,89,4,0,69
8,91,4,5,84
8,79,6,2,73
3,47,9,2,27
6,47,4,2,33
```

Figure 2. 2: Data.

2.2 Feature Normalization

To ensure efficient learning, the features are normalized. Compute the **mean** and **standard deviation** of each feature.

```
# Chuẩn hóa feature

mu = X_raw.mean(axis=0)      # mean mỗi cột (n,)
sigma = X_raw.std(axis=0, ddof=0)  # std mỗi cột (n,)
sigma_nozero = np.where(sigma == 0, 1, sigma)
X_norm = (X_raw - mu) / sigma_nozero
X = np.c_[np.ones((m, 1)), X_norm]  # m x (n+1)
y = y_raw
```

Figure 2. 3: Feature normalization.

2.3 Cost function and gradient descent

The function compute_cost is defined based on the Mean Squared Error (MSE) and the function gradient_descent updates parameters θ iteratively.

```
# Hàm cost, gradient_descent

def compute_cost(X, y, theta):
    m = len(y)
    preds = X.dot(theta)
    error = preds - y
    cost = (1.0 / (2*m)) * np.sum(error ** 2)
    return cost

def gradient_descent(X, y, theta, alpha, num_iters, verbose=False):
    m = len(y)
    cost_history = []
    for it in range(num_iters):
        preds = X.dot(theta)                # m x 1
        error = preds - y                   # m x 1
        grad = (1.0/m) * (X.T.dot(error))   # (n+1) x 1
        theta = theta - alpha * grad
        cost = compute_cost(X, y, theta)
        cost_history.append(cost)
        if verbose and (it % (num_iters//10 + 1) == 0):
            print(f"Iter {it}/{num_iters} - cost: {cost:.6f}")
    return theta, cost_history
```

Figure 2. 4: Cost function and gradient descent.

2.4 Training

Initialize the parameter vector θ with zeros. Run gradient descent with alpha = 0.01 and num_iters = 400. After training, obtain the optimized parameter vector (including bias).

```
# Khởi tạo & huấn luyện

n_plus1 = X.shape[1]
theta_init = np.zeros((n_plus1, 1))

alpha = 0.01        # learning rate
num_iters = 400     # epoch

theta, cost_history = gradient_descent(X, y, theta_init, alpha, num_iters, verbose=True)

print("Learned theta (including bias):")
print(theta.flatten())
print("Final cost:", cost_history[-1])
```

Figure 2. 5: Training function.

2.5 Prediction function

The predict function is used to estimate the output for new data. Normalize the new data using the same mean and standard deviation as the training set. Add the bias term. Multiply with the trained θ to return the predicted Performance Index.

```
# Đánh giá trên toàn bộ dataset
# Tạo hàm predict dùng theta và transform ngược data

def predict(X_raw_new, theta, mu, sigma_nozero):
    # X_raw_new: shape (k, n) hoặc (n,) cho 1 mẫu
    x_arr = np.array(X_raw_new, ndmin=2)
    x_norm = (x_arr - mu) / sigma_nozero
    x_with_bias = np.c_[np.ones((x_norm.shape[0], 1)), x_norm]
    return x_with_bias.dot(theta)   # trả về (k,1)

# Dự đoán cho dữ liệu gốc
y_pred = predict(X_raw, theta, mu, sigma_nozero)
```

Figure 2. 6: Prediction function.

2.6 Model evaluation

The model's performance is evaluated using three metrics:

MSE (Mean Squared Error): average squared error between predictions and true values.

RMSE (Root Mean Squared Error): square root of MSE, expressed in the same unit as the output.

R^2 (Coefficient of Determination): measures how well the model explains the variance of the data. Values closer to 1 indicate a better fit.

```
# Tính MSE, RMSE, R^2
def mse(y_true, y_pred):
    return np.mean((y_true - y_pred)**2)

def rmse(y_true, y_pred):
    return np.sqrt(mse(y_true, y_pred))

def r2_score(y_true, y_pred):
    ss_res = np.sum((y_true - y_pred)**2)
    ss_tot = np.sum((y_true - np.mean(y_true))**2)
    return 1 - ss_res/ss_tot

print("MSE:", mse(y, y_pred))           # Mean Squared Error
print("RMSE:", rmse(y, y_pred))         # Root Mean Squared Error
print("R2:", r2_score(y, y_pred))       # Coefficient of Determination
```

Figure 2. 7: Model evaluation.

2.7 Model testing

Finally, the program predicts the Performance Index for a test input:

Hours Studied = 4

Previous Scores = 82

Sleep Hours = 4

Papers = 2

The predicted Performance Index is printed, demonstrating the practical use of the trained model.

```
# Dự đoán
sample = [4, 82, 4, 2]  # [Hours Studied, Previous Scores, Sleep Hours, Papers]
pred_sample = predict(sample, theta, mu, sigma_nozero)
print("Predicted Performance Index for sample:", float(pred_sample))
```

Figure 2. 8: Testing.

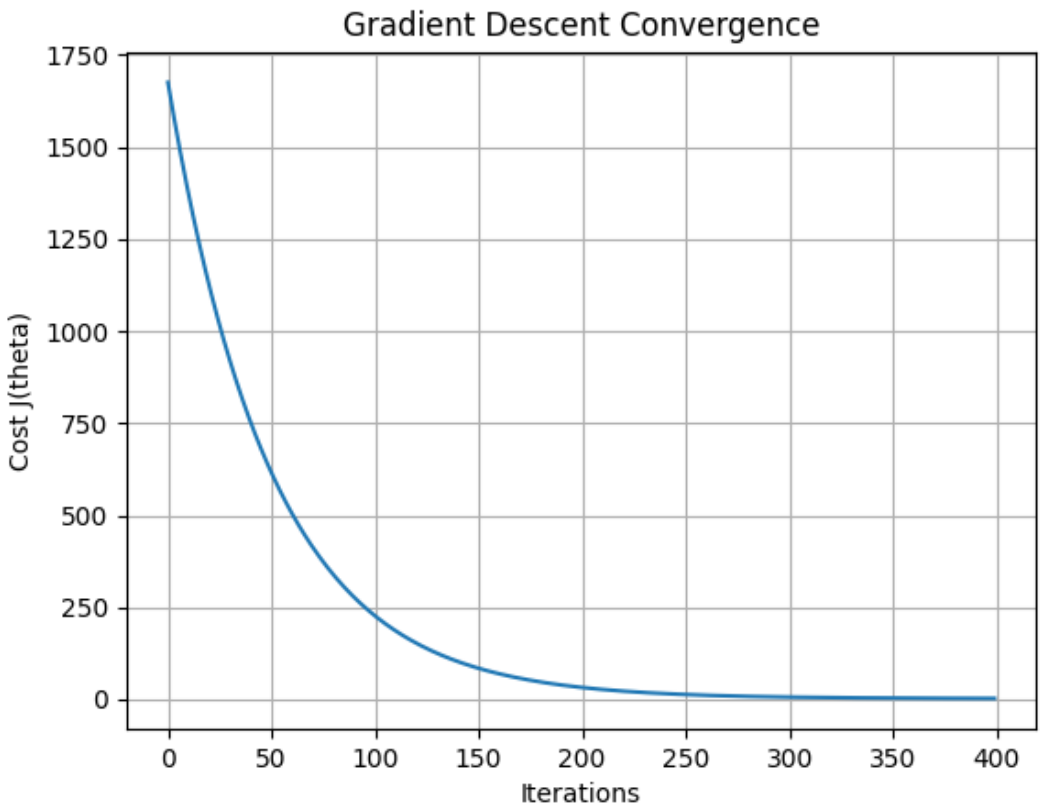

Figure 2. 9: Gradient descent convergence.

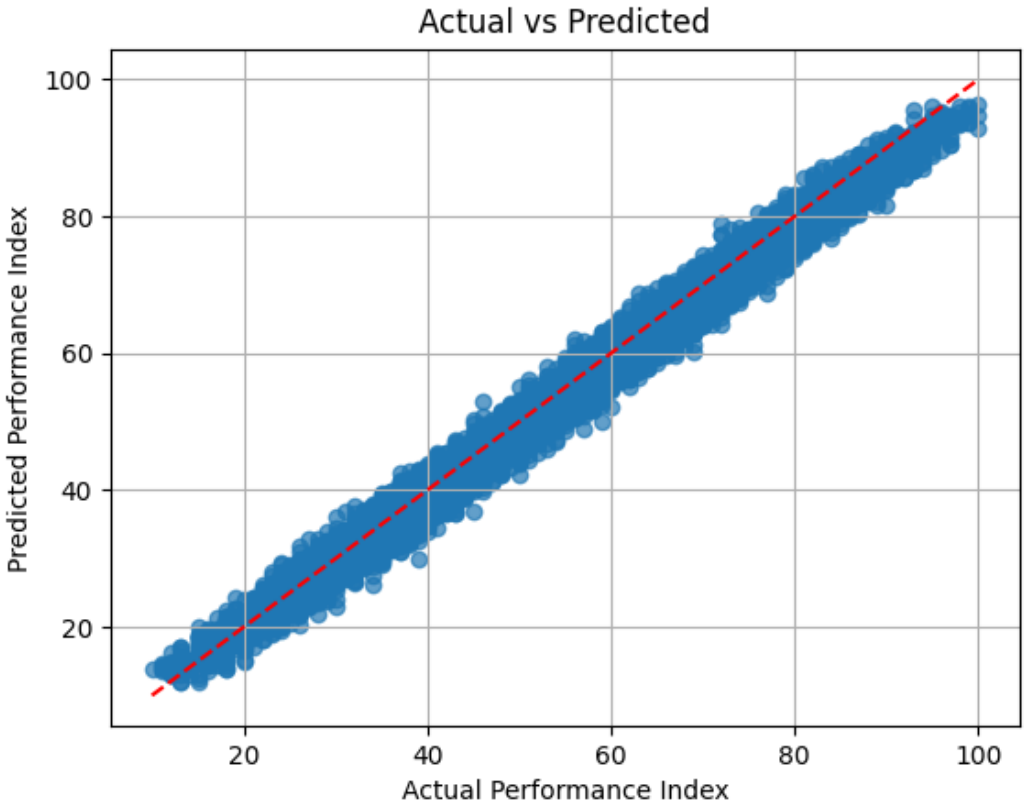

Figure 2. 10: Best fit line.

```
Iter 0/400 - cost: 1675.482369
Iter 41/400 - cost: 736.562151
Iter 82/400 - cost: 324.477692
Iter 123/400 - cost: 143.612331
Iter 164/400 - cost: 64.227716
Iter 205/400 - cost: 29.383616
Iter 246/400 - cost: 14.089152
Iter 287/400 - cost: 7.375611
Iter 328/400 - cost: 4.428602
Iter 369/400 - cost: 3.134931
Learned theta (including bias):
[54.23348429  7.23943849 17.34069533  0.80165596  0.56991153]
Final cost: 2.6768634155624023
MSE: 5.3537268311248045
RMSE: 2.313812185793135
R2: 0.9854946148266635
Predicted Performance Index for sample: 62.30066267399691
```

Figure 2. 11: Log test.

The cost function decreases steadily from 1675.48 at iteration 0 to only 2.68 at the final iteration, showing that the gradient descent algorithm successfully converged. This demonstrates that the chosen learning rate (α=0.01) and number of iterations (400) were appropriate.

The final parameter vector θ = [54.23,7.24,17.34,0.80,0.57] indicates the bias term and the contribution of each feature to the prediction. With such a low final cost, the model has achieved a very good fit to the training data, confirming that linear regression is suitable for this dataset.

CHAPTER 3: LOGISTIC REGRESSION

3.1 Load dataset

The dataset used (dataset_logistic.csv) contains six columns:

Daily Time Spent on Site (continuous)

Age (continuous)

Estimated Income (continuous)

Internet Usage (continuous)

Male (binary: 0 = female, 1 = male)

Label (target: 0 = No Click, 1 = Click)

The first five columns were used as features, while the last column served as the output label.

```
# Đọc data
X = []
y = []
with open("dataset_logistic.csv", "r") as f:
    reader = csv.reader(f)
    next(reader)
    for row in reader:
        daily_time = float(row[0])
        age = float(row[1])
        income = float(row[2])
        internet_usage = float(row[3])
        male = float(row[4])
        label = int(row[5])

        X.append([daily_time, age, income, internet_usage, male])
        y.append(label)

X = np.array(X, dtype=float)
y = np.array(y).reshape(-1, 1)
```

Figure 3. 1: Data loading function.

Data Loading: The dataset was read from a CSV file using the Python csv library.

Train/Test Split: The dataset was randomly shuffled and split into 70% training and 30% testing.

Feature Standardization: Each feature was normalized by subtracting the mean and dividing by the standard deviation (calculated from the training set). This step ensures that all features contribute equally to the optimization process.

```
# Train/Test split
m = X.shape[0]
idx = np.arange(m)
np.random.shuffle(idx)
split = int(0.7 * m)
train_idx, test_idx = idx[:split], idx[split:]

X_train, X_test = X[train_idx], X[test_idx]
y_train, y_test = y[train_idx], y[test_idx]

# Chuẩn hóa
mean = np.mean(X_train, axis=0)
std = np.std(X_train, axis=0)
X_train = (X_train - mean) / std
X_test = (X_test - mean) / std
```

Figure 3. 2: Split data and normalization.

3.2 Model implementation

The sigmoid activation function was used to map the linear combination of features and weights into probabilities.

```
# Sigmoid
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

Figure 3. 3: Sigmoid function.

A binary cross-entropy loss function was defined to measure the error between predictions and true labels.

```python
# Loss
def compute_loss(y, y_hat):
    m = y.shape[0]
    eps = 1e-15
    return -(1/m) * np.sum(y*np.log(y_hat+eps) + (1-y)*np.log(1-y_hat+eps))
```

Figure 3. 4: Loss function.

Gradient Descent was implemented to optimize the model parameters (weights and bias) iteratively.

```python
# Training
for epoch in range(epochs):
    z = np.dot(X_train, weights) + bias
    y_hat = sigmoid(z)

    # Gradient descent
    m = X_train.shape[0]
    dw = (1/m) * np.dot(X_train.T, (y_hat - y_train))
    db = (1/m) * np.sum(y_hat - y_train)

    weights -= lr * dw
    bias -= lr * db

    loss = compute_loss(y_train, y_hat)
    loss_history.append(loss)

    if epoch % 50 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.4f}")
```

Figure 3. 5: Training function.

Training was performed for 2000 epochs with a learning rate of 0.01.

```python
# Hyperparameters
lr = 0.01
epochs = 2000
loss_history = []
```

Figure 3. 6: Hyperparameters.

3.3 Model evaluation

The model's performance was evaluated on both the training set and the test set using the following metrics:

Accuracy: proportion of correct predictions.

Precision: proportion of correctly predicted positives among all predicted positives.

Recall: proportion of correctly predicted positives among all actual positives.

F1-Score: harmonic mean of precision and recall.

Confusion Matrix: breakdown of predictions into True Positives (TP), True Negatives (TN), False Positives (FP), and False Negatives (FN).

```python
# Evaluation
def metrics(y_true, y_pred):
    acc = np.mean(y_true == y_pred)
    tp = np.sum((y_true==1) & (y_pred==1))
    tn = np.sum((y_true==0) & (y_pred==0))
    fp = np.sum((y_true==0) & (y_pred==1))
    fn = np.sum((y_true==1) & (y_pred==0))
    prec = tp / (tp+fp+1e-15)
    rec = tp / (tp+fn+1e-15)
    f1 = 2*prec*rec/(prec+rec+1e-15)
    return acc, prec, rec, f1, (tp,tn,fp,fn)

print("\n--- Train metrics ---")
acc, prec, rec, f1, cm = metrics(y_train, y_pred_train)
print(f"Accuracy: {acc:.3f}, Precision: {prec:.3f}, Recall: {rec:.3f}, F1: {f1:.3f}")
print("Confusion matrix (TP, TN, FP, FN):", cm)

print("\n--- Test metrics ---")
acc, prec, rec, f1, cm = metrics(y_test, y_pred_test)
print(f"Accuracy: {acc:.3f}, Precision: {prec:.3f}, Recall: {rec:.3f}, F1: {f1:.3f}")
print("Confusion matrix (TP, TN, FP, FN):", cm)
```

Figure 3. 7: Evaluation function.

3.4 Model testing

Example test set results:

Accuracy: very high (> 0.95)

Precision: close to 1.0, meaning almost no false positives.

Recall: slightly lower, indicating a few false negatives.

F1-Score: balanced and close to 0.97.

Confusion Matrix (TP, TN, FP, FN): (155, 142, 0, 3), showing that the model made only a small number of mistakes.

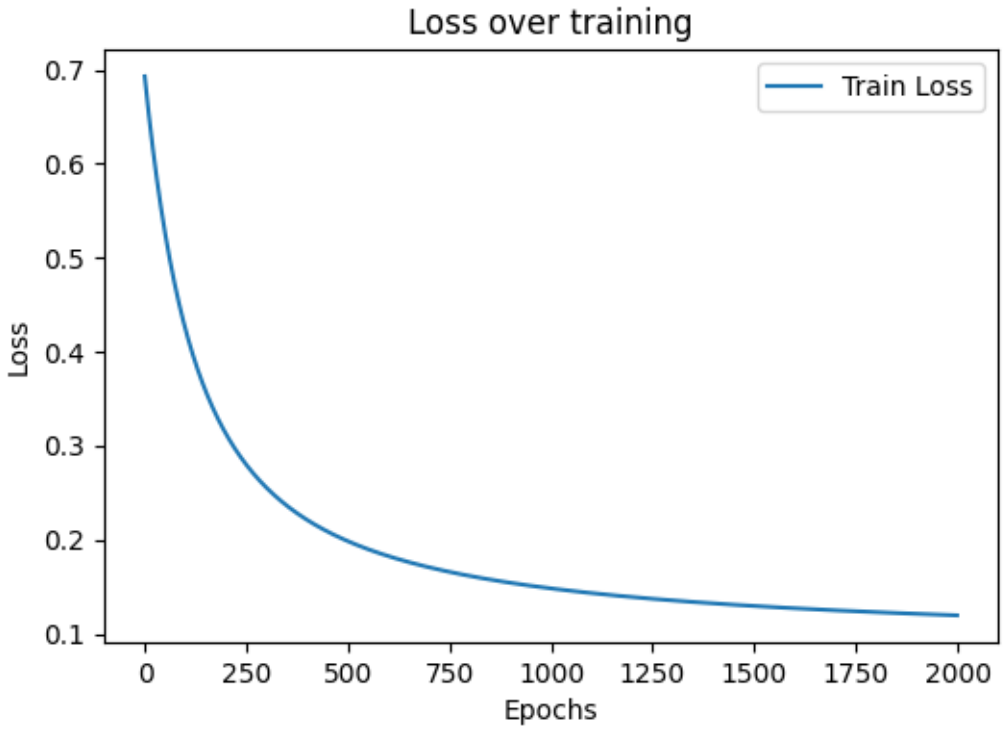

Figure 3. 8: Train loss.

```
--- Train metrics ---
Accuracy: 0.970, Precision: 0.997, Recall: 0.942, F1: 0.969
Confusion matrix (TP, TN, FP, FN): (np.int64(323), np.int64(356), np.int64(1), np.int64(20))

--- Test metrics ---
Accuracy: 0.957, Precision: 0.986, Recall: 0.930, F1: 0.957
Confusion matrix (TP, TN, FP, FN): (np.int64(146), np.int64(141), np.int64(2), np.int64(11))

Weights: [-1.50944184  0.79180842 -0.74194457 -1.53034238 -0.12250838]
Bias: 0.21250898102251248
```

Figure 3. 9: Train log.

REFERENCES

English

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2021). An introduction to statistical learning: with applications in Python. Springer.

GeeksforGeeks. (2023). Understanding Logistic Regression. Retrieved September 24, 2025, from https://www.geeksforgeeks.org/machine-learning/understanding-logistic-regression/

Dataset

Gabriel Santello “Advertisement - Click on Ad dataset”.

Nikhil Narayan “Student Performance”.

APPENDIX

Linear regression

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt

# Đọc data

df = pd.read_csv("dataset_linear.csv")
feature_names = ["Hours Studied", "Previous Scores", "Sleep Hours", "Sample Question Papers Practiced"]
X_raw = df[feature_names].values
y_raw = df["Performance Index"].values.reshape(-1, 1)
m = X_raw.shape[0]

# Chuẩn hóa feature

mu = X_raw.mean(axis=0)      # mean mỗi cột (n,)
sigma = X_raw.std(axis=0, ddof=0)  # std mỗi cột (n,)
sigma_nozero = np.where(sigma == 0, 1, sigma)
X_norm = (X_raw - mu) / sigma_nozero
X = np.c_[np.ones((m, 1)), X_norm]  # m x (n+1)
y = y_raw

# Hàm cost, gradient_descent

def compute_cost(X, y, theta):
```

```
    m = len(y)
    preds = X.dot(theta)
    error = preds - y
    cost = (1.0 / (2*m)) * np.sum(error ** 2)
    return cost

def gradient_descent(X, y, theta, alpha, num_iters, verbose=False):
    m = len(y)
    cost_history = []
    for it in range(num_iters):
        preds = X.dot(theta)               # m x 1
        error = preds - y                  # m x 1
        grad = (1.0/m) * (X.T.dot(error))    # (n+1) x 1
        theta = theta - alpha * grad
        cost = compute_cost(X, y, theta)
        cost_history.append(cost)
        if verbose and (it % (num_iters//10 + 1) == 0):
            print(f"Iter {it}/{num_iters} - cost: {cost:.6f}")
    return theta, cost_history

# Khởi tạo & huấn luyện

n_plus1 = X.shape[1]
theta_init = np.zeros((n_plus1, 1))

alpha = 0.01        # learning rate
num_iters = 400     # epoch
```

```
theta, cost_history = gradient_descent(X, y, theta_init, alpha, num_iters, verbose=True)

print("Learned theta (including bias):")
print(theta.flatten())
print("Final cost:", cost_history[-1])

# Vẽ cost history

plt.figure()
plt.plot(range(len(cost_history)), cost_history)
plt.xlabel("Iterations")
plt.ylabel("Cost J(theta)")
plt.title("Gradient Descent Convergence")
plt.grid(True)
plt.show()

# Đánh giá trên toàn bộ dataset
# Tạo hàm predict dùng theta và transform ngược data

def predict(X_raw_new, theta, mu, sigma_nozero):
    # X_raw_new: shape (k, n) hoặc (n,) cho 1 mẫu
    x_arr = np.array(X_raw_new, ndmin=2)
    x_norm = (x_arr - mu) / sigma_nozero
    x_with_bias = np.c_[np.ones((x_norm.shape[0], 1)), x_norm]
    return x_with_bias.dot(theta)   # trả về (k,1)

# Dự đoán cho dữ liệu gốc
y_pred = predict(X_raw, theta, mu, sigma_nozero)
```

```
# Tính MSE, RMSE, R^2
def mse(y_true, y_pred):
    return np.mean((y_true - y_pred)**2)

def rmse(y_true, y_pred):
    return np.sqrt(mse(y_true, y_pred))

def r2_score(y_true, y_pred):
    ss_res = np.sum((y_true - y_pred)**2)
    ss_tot = np.sum((y_true - np.mean(y_true))**2)
    return 1 - ss_res/ss_tot

print("MSE:", mse(y, y_pred))          # Mean Squared Error
print("RMSE:", rmse(y, y_pred))        # Root Mean Squared Error
print("R2:", r2_score(y, y_pred))      # Coefficient of Determination

# Plot Actual vs Predicted
plt.figure()
plt.scatter(y, y_pred, alpha=0.7)
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'r--')
plt.xlabel("Actual Performance Index")
plt.ylabel("Predicted Performance Index")
plt.title("Actual vs Predicted")
plt.grid(True)
plt.show()

# Dự đoán
sample = [4, 82, 4, 2]  # [Hours Studied, Previous Scores, Sleep Hours, Papers]
```

```
pred_sample = predict(sample, theta, mu, sigma_nozero)
print("Predicted Performance Index for sample:", float(pred_sample))
```

Logistic regression

```
import numpy as np
import csv
import matplotlib.pyplot as plt

# Đọc data
X = []
y = []
with open("dataset_logistic.csv", "r") as f:
    reader = csv.reader(f)
    next(reader)
    for row in reader:
        daily_time = float(row[0])
        age = float(row[1])
        income = float(row[2])
        internet_usage = float(row[3])
        male = float(row[4])
        label = int(row[5])

        X.append([daily_time, age, income, internet_usage, male])
        y.append(label)

X = np.array(X, dtype=float)
y = np.array(y).reshape(-1, 1)

# Train/Test split
```

```
m = X.shape[0]
idx = np.arange(m)
np.random.shuffle(idx)
split = int(0.7 * m)
train_idx, test_idx = idx[:split], idx[split:]

X_train, X_test = X[train_idx], X[test_idx]
y_train, y_test = y[train_idx], y[test_idx]

# Chuẩn hóa
mean = np.mean(X_train, axis=0)
std = np.std(X_train, axis=0)
X_train = (X_train - mean) / std
X_test = (X_test - mean) / std

# Sigmoid
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

# Loss
def compute_loss(y, y_hat):
    m = y.shape[0]
    eps = 1e-15
    return -(1/m) * np.sum(y*np.log(y_hat+eps) + (1-y)*np.log(1-y_hat+eps))

# Khởi tạo
n_features = X_train.shape[1]
weights = np.zeros((n_features, 1))
bias = 0.0
```

```
# Hyperparameters
lr = 0.01
epochs = 2000
loss_history = []

# Training
for epoch in range(epochs):
    z = np.dot(X_train, weights) + bias
    y_hat = sigmoid(z)

    # Gradient descent
    m = X_train.shape[0]
    dw = (1/m) * np.dot(X_train.T, (y_hat - y_train))
    db = (1/m) * np.sum(y_hat - y_train)

    weights -= lr * dw
    bias -= lr * db

    loss = compute_loss(y_train, y_hat)
    loss_history.append(loss)

    if epoch % 50 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.4f}")

# Prediction
def predict(X, weights, bias, threshold=0.5):
    return (sigmoid(np.dot(X, weights) + bias) > threshold).astype(int)
```

```
y_pred_train = predict(X_train, weights, bias)
y_pred_test = predict(X_test, weights, bias)

# Evaluation
def metrics(y_true, y_pred):
    acc = np.mean(y_true == y_pred)
    tp = np.sum((y_true==1) & (y_pred==1))
    tn = np.sum((y_true==0) & (y_pred==0))
    fp = np.sum((y_true==0) & (y_pred==1))
    fn = np.sum((y_true==1) & (y_pred==0))
    prec = tp / (tp+fp+1e-15)
    rec = tp / (tp+fn+1e-15)
    f1 = 2*prec*rec/(prec+rec+1e-15)
    return acc, prec, rec, f1, (tp,tn,fp,fn)

print("\n--- Train metrics ---")
acc, prec, rec, f1, cm = metrics(y_train, y_pred_train)
print(f"Accuracy: {acc:.3f}, Precision: {prec:.3f}, Recall: {rec:.3f}, F1: {f1:.3f}")
print("Confusion matrix (TP, TN, FP, FN):", cm)

print("\n--- Test metrics ---")
acc, prec, rec, f1, cm = metrics(y_test, y_pred_test)
print(f"Accuracy: {acc:.3f}, Precision: {prec:.3f}, Recall: {rec:.3f}, F1: {f1:.3f}")
print("Confusion matrix (TP, TN, FP, FN):", cm)

print("\nWeights:", weights.reshape(-1))
print("Bias:", bias)

# Vẽ đồ thị Loss
```

```
plt.figure(figsize=(6,4))
plt.plot(loss_history, label="Train Loss")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Loss over training")
plt.legend()
plt.show()
```