# Linear Regression algorithm

**Class 7**

**09/3/2025**

# Acknowledgement

**The series of the IT & Japanese language course is Supported by AOTS and OEC.**

AOTS — Ministry of Economy, Trade and Industry

OEC — Overseas Employment Corporation

# What you have Learnt Last Week

## We were focused on following points.

- Usage of control and loop flow statement
- Inspecting and Understanding Data
- Basics of creating, loading, and exploring DataFrames
- Understanding of 1D, and 2D NumPy arrays
- Array indexing and slicing
- Basics of Matplotlib’s plotting library, setting up figures, and axes
- Customize your plots with various line styles, markers, and colors
- Making data visualization more engaging and informative.
- Visualize relationships and correlations with scatter plots

# What you will Learn Today

**We will focus on following points.**

- Understand the basic concept of linear regression
- Mathematical Foundation of Linear Regression
- Types of Linear Regression: Simple vs. Multiple
- Upload code on Github
- Quiz
- Q&A Session

# Definition of Linear Regression

**Linear Regression is a statistical method used to model the relationship between a dependent variable (Y) and one or more independent variables (X).**

## [Formula]

$$Y = \beta_0 + \beta_1 X + \epsilon$$

- $Y$ = Dependent variable
- $X$ = Independent variable
- $\beta_0$ = Intercept
- $\beta_1$ = Coefficient (Slope)
- $\epsilon$ = Error term

# Importance and Applications

## Linear Regression is interpretable and helps in making decision

### [Why Linear Regression?]

- Simple and interpretable
- Helps in making predictions
- Identifies trends and relationships
- Used in predictive analytics and machine learning.
- Find the best-fitting line that minimizes errors.

### [Applications]

- Finance: Stock price prediction
- Healthcare: Disease risk estimation
- Marketing: Sales forecasting
- Education: Predicting student performance

# Assumptions of Linear Regression

## Checking assumptions using Python

**[Why Linear Regression?]**

- Linearity: The relationship between X and Y is linear.
- Independence: Observations are independent.
- Homoscedasticity: Equal variance of residuals.
- Normality: Residuals follow a normal distribution.

```
import seaborn as sns
import statsmodels.api as sm
from sklearn.linear_model import LinearRegression

# Load dataset
data = sns.load_dataset("tips")
X = data["total_bill"].values.reshape(-1, 1)
y = data["tip"].values

# Fit model
model = LinearRegression()
model.fit(X, y)

# Check residuals
residuals = y - model.predict(X)
sm.qqplot(residuals, line='s')
```

[Note]
Residuals help assess how well the model fits the data

# How Linear Regression Works

## Fitting a linear regression model in Python

**[Linear Regression Working]**

- Finding the Best Fit Line:
  - Uses the Least Squares Method to minimize the sum of squared errors.
- Gradient Descent (for large datasets):
  - Iteratively updates coefficients to minimize loss function.

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

# Sample data
X = np.array([1, 2, 3, 4, 5])
y = np.array([2, 3, 5, 6, 8])

# Fit model
model = LinearRegression()
model.fit(X.reshape(-1, 1), y)

# Plot
plt.scatter(X, y, color="blue", label="Data")
plt.plot(X, model.predict(X.reshape(-1, 1)),
color="red", label="Regression Line")
plt.legend()
plt.show()
```

# Build a Linear Regression Model

## Build a Linear Regression Model to predict house prices

### Example Dataset:

| Square Footage (sqft) | House Price ($1000s) |
|---|---|
| 800 | 150 |
| 1200 | 200 |
| 1500 | 250 |
| 1800 | 300 |
| 2200 | 350 |

# Build a Linear Regression Model

## 1. Import Required Libraries

**[Code]**

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import LinearRegression
```

[Note]

- numpy is used for handling numerical data.
- matplotlib.pyplot is used for plotting graphs.
- sklearn.linear_model provides LinearRegression for creating the regression model.

# Build a Linear Regression Model

## 2. Define the Dataset

**[Code]**

```
square_feet = np.array([800, 1200, 1500, 1800, 2200]).reshape(-1, 1)  # Independent variable (X)
house_prices = np.array([150, 200, 250, 300, 350])  # Dependent variable (Y)
```

[Note]

- square_feet: A numpy array representing the independent variable (square footage).
- house_prices: A numpy array representing the dependent variable (house prices in thousands).
- .reshape(-1, 1): Converts square_feet into a column vector for compatibility with LinearRegression.

# Build a Linear Regression Model

## 3. Create and Train the Model

**[Code]**

```
model = LinearRegression()  # Initialize the model
model.fit(square_feet, house_prices)  # Train the model using the dataset
```

[Note]

- LinearRegression(): Creates an instance of the linear regression model.
- .fit(X, y): Trains the model using the given input (X) and output (y).

# Build a Linear Regression Model

## 4. Make a Prediction

**[Code]**

```
predicted_price = model.predict(np.array([[2000]]))[0]
print(f"Predicted House Price for 2000 sqft: ${predicted_price}K")
```

[Note]

- .predict([[2000]]): Uses the trained model to predict the house price for a house with 2000 sqft.
- print(...): Displays the predicted price.

# Build a Linear Regression Model

## 5. Visualize the Results

### [Code]

```
plt.scatter(square_feet, house_prices, color="blue", label="Actual Data")  # Scatter plot of actual data
plt.plot(square_feet, model.predict(square_feet), color="red", label="Regression Line")  # Regression line
plt.scatter(2000, predicted_price, color="green", marker="*", s=150, label="Prediction (2000 sqft)")  # Predicted point
plt.xlabel("Square Footage")
plt.ylabel("House Price ($1000s)")
plt.title("Linear Regression: House Price Prediction")
plt.legend()
plt.show()
```

[Note]

- plt.scatter(): Plots the actual data points in blue.
- plt.plot(): Draws the regression line in red.
- plt.scatter(2000, predicted_price, color="green", marker="*", s=150): Highlights the predicted price for 2000 sqft.
- plt.xlabel(), plt.ylabel(), plt.title(): Labels the axes and sets the title.
- plt.legend(): Adds a legend to the plot.
- plt.show(): Displays the plot.

# Key Terminologies in Linear Regression

## Understanding these key terms is essential for grasping how Linear Regression works.

| Terminology | Definition | Example (House Price) |
|---|---|---|
| **Dependent Variable (Y)** | The variable we predict. | House Price ($1000s) |
| **Independent Variable (X)** | The variable used to make predictions. | Square Footage (sqft) |
| **Regression Line** | Best-fit line that minimizes errors. | House Price=50+0.125X\text{House Price} = 50 + 0.125XHouse Price=50+0.125X |
| **Coefficients (β1\beta_1β1)** | Slope of the line, shows X's impact on Y. | β1=0.125\beta_1 = 0.125β1=0.125 (Price increases $12.5K per 100 sqft) |
| **Intercept (β0\beta_0β0)** | Starting value of Y when X = 0. | β0=50\beta_0 = 50β0=50 ($50K base price) |

# Equation of a Straight Line

## The foundation of Linear Regression is the equation of a straight line

**y = mx + c**, where:

- y= Dependent variable (target output)
- x= Independent variable (input feature)
- m= Slope of the line
- c= Intercept

**Example:** If m=2 and c=3, then for x=5: **i.e. y=2(5)+3=13**

# Task-1

# Linear Function Representation with NumPy and Matplotlib

## [Convert these Lines into Code]

- `import numpy as np` → Import NumPy library.
- `import matplotlib.pyplot as plt` → Import Matplotlib library for plotting.
- `Define a function named linear_function with parameters x, m, and cdef linear_function(x, m=2, c=3):`
  - `return m * x + c  # Return the result of the linear equation y = mx + c`

  - `x_values = np.linspace(-10, 10, 100)` → Generate 100 evenly spaced values between -10 and 10 using NumPy.
- `y_values = linear_function(x_values)` → Compute corresponding `y` values using the `linear_function`.
- `plt.plot(x_values, y_values, label='y = 2x + 3')` → Plot the linear function with a label.
- `plt.xlabel('x')` → Label the x-axis as "x".
- `plt.ylabel('y')` → Label the y-axis as "y".
- `plt.legend()` → Display a legend for the plotted function.
- `plt.grid()` → Enable grid lines on the plot.
- `plt.show()` → Display the plot.

# Linear Function Representation with NumPy and Matplotlib

## [Answer]

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

def linear_function(x, m=2, c=3):
    return m * x + c

x_values = np.linspace(-10, 10, 100)
y_values = linear_function(x_values)

plt.plot(x_values, y_values, label='y = 2x + 3')
plt.xlabel('x')
plt.ylabel('y')
plt.legend()
plt.grid()
plt.show()
```

# Cost Function (Mean Squared Error - MSE)

**Measures how well the regression line fits the data.**

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$$

$\mathrm{MSE}$ = mean squared error

$n$ = number of data points

$Y_i$ = observed values

$\hat{Y}_i$ = predicted values

Goal: Minimize MSE to improve model accuracy

# Cost Function (Mean Squared Error - MSE)

## Measures how well the regression line fits the data.

```
def mse(y_actual, y_predicted):
    return np.mean((y_actual - y_predicted) ** 2)

y_actual = np.array([1, 2, 3, 4, 5])
y_predicted = np.array([1.1, 1.9, 3.2, 3.8, 5.1])
print("Mean Squared Error:", mse(y_actual, y_predicted))
```

Goal: Minimize MSE to improve model accuracy

# Gradient Descent for Optimization

## Iterative optimization algorithm to minimize cost function.

- Update rule for parameters: $m = m - \alpha \frac{\partial}{\partial m} MSE \; c = c - \alpha \frac{\partial}{\partial c} MSE$
- $\alpha$ = Learning rate (step size in optimization)
- Adjusts slope (m) and intercept (c) to find the optimal line.

Goal: Minimize MSE to improve model accuracy

# Gradient Descent for Optimization

**Iterative optimization algorithm to minimize cost function.**

Goal: Minimize MSE to improve model accuracy

# Gradient Descent for Optimization

## Iterative optimization algorithm to minimize cost function.

1. Gradient for slope $m$:

$$\frac{\partial J}{\partial m} = -\frac{2}{n}\sum_{i=1}^{n} X_i(y_i - (mX_i + c))$$

2. Gradient for intercept $c$:

$$\frac{\partial J}{\partial c} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - (mX_i + c))$$

**Gradient Descent Update Rule**

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

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

```
# Calculate predicted values using the current m and c
y_pred = m * X + c
# Compute the gradient of m
dm = (-2/n) * np.sum(X * (y - y_pred))
# Compute the gradient of c
dc = (-2/n) * np.sum(y - y_pred)
# Update m using the learning rate and gradient
m -= learning_rate * dm
# Update c using the learning rate and gradient
c -= learning_rate * dc
```

# Task-2

# Optimizing a Linear Model Using Gradient Descent

## [Convert these Lines into Code]

- `import numpy as np` → Import NumPy library.
- `def gradient_descent(X, y, learning_rate=0.01, iterations=1000):` → Define the `gradient_descent` function with inputs `X`, `y`, `learning_rate`, and `iterations`.
- `m, c = 0, 0` → Initialize slope `m` and intercept `c` to 0.
- `n = len(y)` → Store the number of observations in `n`.
- `for _ in range(iterations):` → Loop through the number of iterations for optimization.
- `y_pred = m * X + c` → Calculate predicted values using the current `m` and `c`.
- `dm = (-2/n) * np.sum(X * (y - y_pred))` → Compute the gradient of `m` (partial derivative with respect to `m`).
- `dc = (-2/n) * np.sum(y - y_pred)` → Compute the gradient of `c` (partial derivative with respect to `c`).
- `m -= learning_rate * dm` → Update `m` using the learning rate and gradient.
- `c -= learning_rate * dc` → Update `c` using the learning rate and gradient.
- `return m, c` → Return the optimized values of `m` and `c`.
- `X = np.array([1, 2, 3, 4, 5])` → Define `X` as a NumPy array of input values.
- `y = np.array([2, 4, 6, 8, 10])` → Define `y` as a NumPy array of actual values.
- `m, c = gradient_descent(X, y)` → Call the `gradient_descent` function to compute optimized `m` and `c`.
- `print("Optimized Slope (m):", m)` → Print the optimized value of `m`.
- `print("Optimized Intercept (c):", c)` → Print the optimized value of `c`.

# Optimizing a Linear Model Using Gradient Descent

## Iterative optimization algorithm to minimize cost function.

```
Import numpy as np
def gradient_descent(X, y, learning_rate=0.01,
iterations=1000):
    m, c = 0, 0
    n = len(y)
    for _ in range(iterations):
        y_pred = m * X + c
        dm = (-2/n) * sum(X * (y - y_pred))
        dc = (-2/n) * sum(y - y_pred)
        m -= learning_rate * dm
        c -= learning_rate * dc
    return m, c

X = np.array([1, 2, 3, 4, 5])
y = np.array([2, 4, 6, 8, 10])
m, c = gradient_descent(X, y)
print("Optimized Slope (m):", m)
print("Optimized Intercept (c):", c)
```

# Ordinary Least Squares (OLS) Method

## Analytical approach to minimize the sum of squared residuals

- Analytical approach to minimize the sum of squared residuals.
- Formula for estimating $m$ and $c$: $m = \frac{\sum(x_i - \bar{x})(y_i - \bar{y})}{\sum(x_i - \bar{x})^2}$ $c = \bar{y} - m\bar{x}$
- Finds the line that minimizes the total squared error.

# Ordinary Least Squares (OLS) Method

## Analytical approach to minimize the sum of squared residuals

```
from sklearn.linear_model import LinearRegression

X = np.array([1, 2, 3, 4, 5]).reshape(-1, 1)
y = np.array([2, 4, 6, 8, 10])
model = LinearRegression().fit(X, y)
print("Slope (m):", model.coef_[0])
print("Intercept (c):", model.intercept_)
```

# R-squared and Adjusted R-squared

1. **R-squared () measures the proportion of variance explained by the model**
2. **Adjusted R-squared accounts for the number of predictors in the model**

- **R-squared** ($R^2$) measures the proportion of variance explained by the model:
  $R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$ where:
  - $SS_{res}$ = Sum of squared residuals
  - $SS_{tot}$ = Total sum of squares
- **Adjusted R-squared** accounts for the number of predictors in the model:
  $R^2_{adj} = 1 - \left(\frac{(1-R^2)(n-1)}{n-k-1}\right)$ where:
  - $k$ = Number of predictors
  - $n$ = Sample size

# R-squared and Adjusted R-squared

1. **R-squared () measures the proportion of variance explained by the model**
2. **Adjusted R-squared accounts for the number of predictors in the model**

```
from sklearn.metrics import r2_score

y_actual = np.array([2, 4, 6, 8, 10])
y_predicted = model.predict(X)
print("R-squared:", r2_score(y_actual, y_predicted))
```

# Bias-Variance Tradeoff in Linear Regression

1. Bias: Error due to overly simplistic models (underfitting).
2. Variance: Error due to overly complex models (overfitting).
3. Tradeoff: Finding the right balance between bias and variance for better generalization.

# Quiz Section

# Quiz

## Everyone student should click on submit button before time ends otherwise MCQs will not be submitted

### [Guidelines of MCQs]

1. There are 20 MCQs
2. Time duration will be 10 minutes
3. This link will be share on 6:10pm (Pakistan time)
4. MCQs will start from 6:15pm (Pakistan time)
5. This is exact time and this will not change
6. Everyone student should click on submit button otherwise MCQs will not be submitted after time will finish
7. Every student should submit Github profile and LinkedIn post link for every class. It include in your performance

# Assignment

## Assignment should be submit before the next class

### [Assignments Requirements]

1. Create a post of today's lecture and post on LinkedIn.
2. Make sure to tag @Plus W @Pak-Japan Centre and instructors LinkedIn profile
3. Upload your code of assignment and lecture on GitHub and share your GitHub profile in respective

   your region group WhatsApp group
4. If you have any query regarding assignment, please share on your region WhatsApp group.
5. Students who already done assignment, please support other students

# Q&A Session

# ありがとうございます。
# Thank you.
شکریا

For the World with Diverse Individualities