Linear Regression

Linear Regression is a supervised learning algorithm used for predicting continuous values based on the relationship between input features. It assumes a linear relationship between the independent variable(s) (X) and the dependent variable (Y).

Types of Linear Regression

There are two main types of linear regression:

Simple Linear Regression – One independent variable (X) and one dependent variable (Y).

Multiple Linear Regression – More than one independent variable (X1, X2, X3...) predicting a single dependent variable (Y).

Equation of Linear Regression

The mathematical equation of a linear regression model is:

$$Y = mX + c$$

for multiple varriables:

$$Y = b_0 + b_1X_1 + b_2X_2 + ... + b_nX_n$$

Where:

Y = Predicted output (dependent variable) X = Input feature(s) (independent variable(s)) m (or b1, b2, etc.) = Coefficients (slopes) c (or b0) = Intercept (constant) n = Number of features

Optimization using Gradient Descent

Gradient Descent is an optimization algorithm that updates model coefficients to minimize the cost function. It updates the weights using:

$$b_j = b_j - \alpha \frac{\partial J}{\partial b_j}$$

Where:

α (alpha) = Learning rate J = Cost function

Code:

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.linear_model import LinearRegression

from sklearn.metrics import mean_squared_error


# Sample Data: Square Footage (X) vs House Price (y)

X = np.array([500, 700, 800, 1000, 1200, 1500, 1800, 2000]).reshape(-1, 1)

y = np.array([150000, 180000, 200000, 240000, 280000, 320000, 360000, 400000])


# Train-Test Split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)


# Train Model

model = LinearRegression()
```

```
model.fit(X_train, y_train)


# Predict
y_pred = model.predict(X_test)


# Evaluate
mse = mean_squared_error(y_test, y_pred)
print(f"Mean Squared Error: {mse:.2f}")
print(f"Slope (Coefficient): {model.coef_[0]:.2f}")
print(f"Intercept: {model.intercept_:.2f}")


# Visualization
plt.scatter(X, y, color='blue', label="Actual Data")  # Scatter plot of real data
plt.plot(X, model.predict(X), color='red', label="Regression Line")  # Line of best fit
plt.xlabel("Square Footage")
plt.ylabel("House Price")
plt.legend()
plt.show()
```

Covariance & Correlation in Data Analysis

Both **covariance** and **correlation** measure relationships between two variables, but they have different properties and interpretations.

covariance (How Two Variables Change Together)

Covariance measures **how two variables vary together**.

- If **one increases** while the **other also increases**, covariance is **positive**.
- If **one increases** while the **other decreases**, covariance is **negative**.

- A **zero** covariance means no relationship.

Covariance Formula

$$\mathrm{Cov}(X,Y) = \frac{\sum(X_i - \bar{X})(Y_i - \bar{Y})}{n}$$

Where:

- X,YX, YX,Y = Data variables
- X¯, Y¯ = Mean of XXX and YYY
- n = Number of observations

CODE:

```
import numpy as np

X = [10, 20, 30, 40, 50]

Y = [5, 15, 25, 35, 50]

# Calculate Covariance

cov_matrix = np.cov(X, Y)

print("Covariance Matrix:\n", cov_matrix)

print("Covariance Value:", cov_matrix[0, 1])  # Cov(X, Y)
```

output:

```
Covariance Matrix:

[[250.  125.]

 [125.  312.5]]

Covariance Value: 125.0
```

Correlation (Strength & Direction of Relationship)

Correlation **standardizes covariance**, making it easier to interpret.

- Always between **-1 and +1**.
- **+1** → Perfect positive correlation (both increase together).
- **-1** → Perfect negative correlation (one increases, the other decreases).
- **0** → No correlation.

✅ **Correlation Formula**

$$r = \frac{\mathrm{Cov}(X, Y)}{\sigma_X \sigma_Y}$$

Where:

- **σX, σY = Standard deviation of X and Y**

```
import numpy as np

# Sample Data

X = [10, 20, 30, 40, 50]

Y = [5, 15, 25, 35, 50]

# Calculate Correlation

corr_matrix = np.corrcoef(X, Y)

print("Correlation Matrix:\n", corr_matrix)

print("Correlation Value:", corr_matrix[0, 1])  # Corr(X, Y)
```

```
output:

Correlation Matrix:

[[1.  0.98]

 [0.98  1. ]]

Correlation Value: 0.98
```

Residuals & Mean Squared Error (MSE) in Regression

When evaluating a regression model, two key concepts are **residuals** and **mean squared error (MSE)**.

1. Residuals (Errors in Prediction)

Residuals measure **how far** each predicted value is from the actual value.

Residual=Actual Value−Predicted Value

🔹 **Interpretation of Residuals:**

- **Small residuals** → Good predictions
- **Large residuals** → Poor predictions
- **Randomly distributed residuals** → Model is good
- **Pattern in residuals** → Model is missing something

Code:

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.linear_model import LinearRegression

from sklearn.model_selection import train_test_split

# Sample Data (House Prices)

X = np.array([1000, 1500, 2000, 2500, 3000]).reshape(-1, 1)  # Square footage

y = np.array([200000, 250000, 320000, 400000, 450000])  # Prices

# Split Data
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
# Train Model
model = LinearRegression()
model.fit(X_train, y_train)
# Predict
y_pred = model.predict(X_test)

# Calculate Residuals
residuals = y_test - y_pred

# Plot Residuals
plt.scatter(y_test, residuals)
plt.axhline(y=0, color='r', linestyle='--')
plt.xlabel("Actual Values")
plt.ylabel("Residuals")
plt.title("Residual Plot")
plt.show()
```

2. Mean Squared Error (MSE)

MSE measures the **average squared difference** between actual and predicted values.

$$\text{MSE} = \frac{1}{n}\sum(y_{\text{actual}} - y_{\text{predicted}})^2$$

Why Squared?

- Avoids negative errors canceling positive errors.
- Gives **higher weight** to larger errors.

Code:

```python
from sklearn.metrics import mean_squared_error

mse = mean_squared_error(y_test, y_pred)

print(f"Mean Squared Error: {mse:.2f}")
```

Fine-Tuning a Linear Regression Model

Fine-tuning a model means improving its performance by optimizing hyperparameters, transforming data, or applying feature engineering techniques. Since **Linear Regression** has no hyperparameters like deep learning models, we fine-tune it by:

1. **Feature Scaling** (Standardization or Normalization)
2. **Feature Engineering** (Creating new features, removing irrelevant ones)
3. **Polynomial Features** (For non-linear relationships)
4. **Regularization** (Ridge & Lasso Regression)
5. **Removing Outliers** (Improves model accuracy)

```python
import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler, PolynomialFeatures

from sklearn.linear_model import LinearRegression, Ridge, Lasso

from sklearn.metrics import mean_squared_error, r2_score

# Sample Data: Square Footage vs House Price

data = {

    "Square_Footage": [500, 700, 800, 1000, 1200, 1500, 1800, 2000, 2200, 2500],

    "Bedrooms": [1, 2, 2, 3, 3, 4, 4, 5, 5, 6],  # Additional Feature

    "House_Price": [150000, 180000, 200000, 240000, 280000, 320000, 360000, 400000, 430000, 480000]
```

```
}

df = pd.DataFrame(data)

# Define Features & Target
X = df[["Square_Footage", "Bedrooms"]].values
y = df["House_Price"].values

# Train-Test Split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=50)

# Feature Scaling (Standardization)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Apply Polynomial Features (For Non-Linear Effects)
poly = PolynomialFeatures(degree=2)
X_train_poly = poly.fit_transform(X_train_scaled)
X_test_poly = poly.transform(X_test_scaled)

# Train Linear Model
model = LinearRegression()
model.fit(X_train_poly, y_train)

# Predictions
y_pred = model.predict(X_test_poly)
```

```
# Evaluate Models
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)


# Print Performance
print(f"Linear Regression -> MSE: {mse:.2f}, R²: {r2:.4f}")
```

Fine-Tuning Techniques Used

Feature Scaling → Standardizes data to improve model convergence.
Polynomial Features → Captures non-linear relationships.
Regularization (Ridge & Lasso) → Prevents overfitting.
Feature Engineering → Added "Bedrooms" as a new feature.
Random State Optimization → Set random_state=50 to stabilize results.