Regression

Regression is a **supervised learning technique** used to model the relationship between a dependent variable (target) and one or more independent variables (features).

Types of Regression:

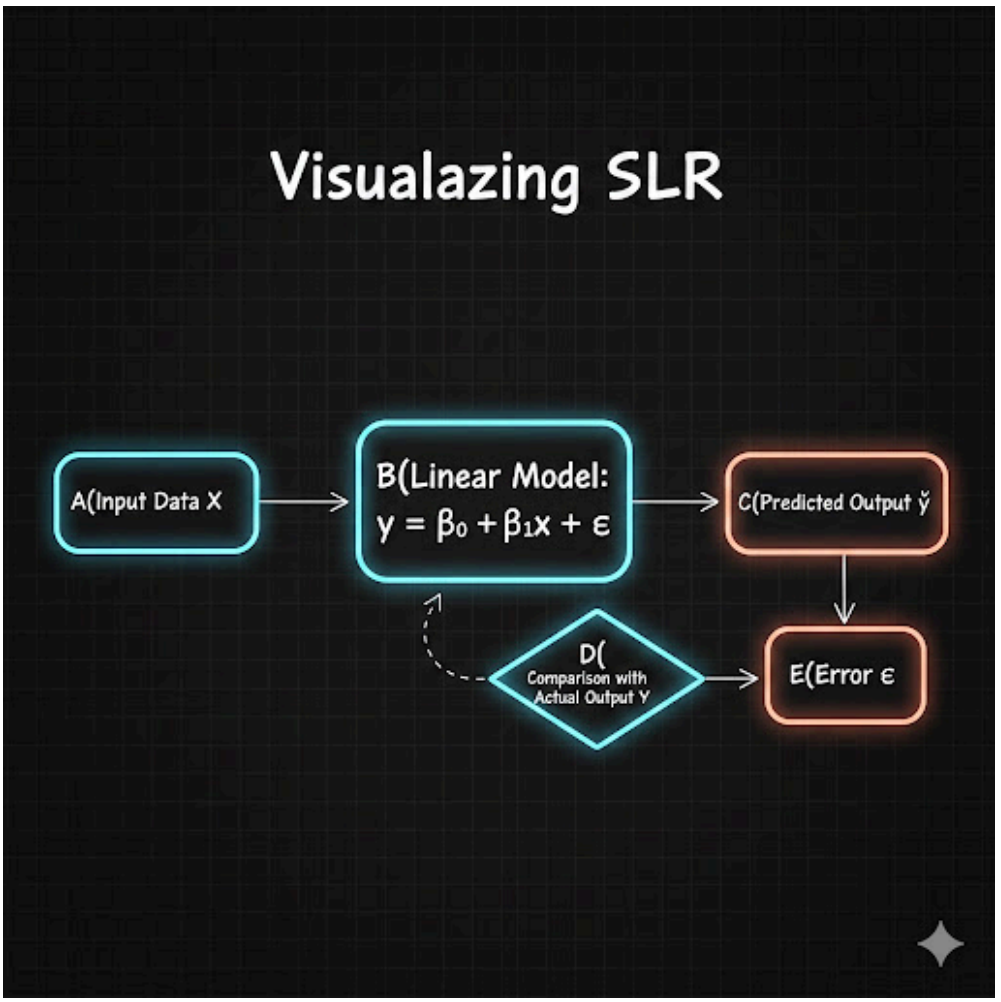
1. Simple Linear Regression: Models linear relationship between one independent variable X and one dependent variable Y.
2. Multiple Linear Regression: Models linear relationship between two or more independent variables X and one dependent variable Y.
3. Non-Linear Regression: Models non-linear, complex relationship using curves

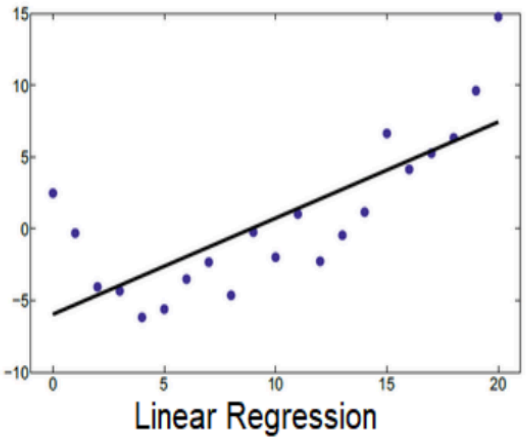
Eg : House price prediction

Consider the equation of straight line

$$y = m * x + c$$
$$y = \beta 0 + \beta 1 x$$

Visualizing SLR:





A **best fit line** is the line that **best represents the relationship between input (X) and output (Y)** by **minimizing the overall error** between actual values and predicted values.

Here y is a dependent variable and x is an independent variable.
The predicted value for a given x_i is:

$$\hat{y}_i = m * xi + c$$

The error for that point is:

$$error_i = y_i - (m * xi + c)$$

The sum we want to minimize (let's call it S) is:

$$S = \Sigma [y_i - m * xi - c]^2 \quad \#(for\ all\ i\ from\ 1\ to\ n)$$

To find the m and c that minimize S, we use calculus:

1. Take the partial derivative with respect to c and set it to 0:

   $$\partial S/\partial c = \Sigma\, 2 * [y_i - m * xi - c] * (-1) = 0$$

   This simplifies to: $\Sigma(y_i) - m * \Sigma(xi) - n * c = 0$ ---------- (1)

2. Take the partial derivative with respect to m and set it to 0:

   $$\partial S/\partial m = \Sigma\, 2 * [y_i - m * xi - c] * (-xi) = 0$$

   This simplifies to: $\Sigma(x_i * yi) - m * \Sigma(xi^2) - c * \Sigma(xi) = 0$ ---------- (2)

Now, substitute this expression for c back into Equation 2. After a bit of algebra (which involves expanding terms and simplifying), you arrive at the formula for the slope m:

$$m = [\Sigma(x_i - xmean)(yi - ymean)] / [\Sigma(xi - xmean)^2]$$

MAE – MSE – RMSE – R2

Let us take a coding

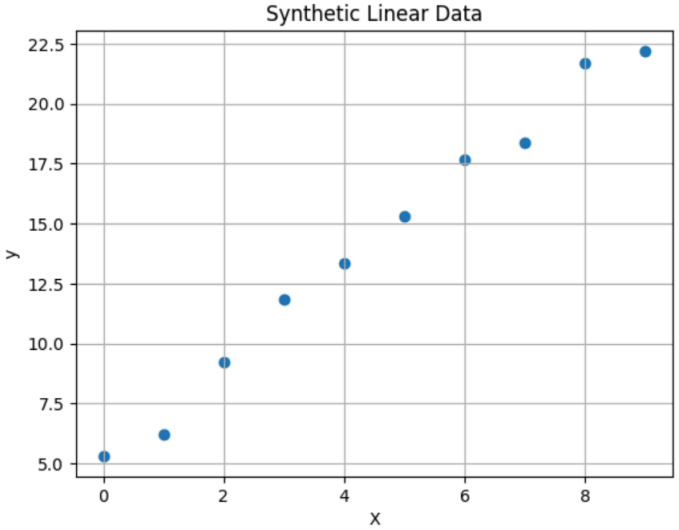
#Step-01: Generate synthetic data

```
import random
import matplotlib.pyplot as plt

# Generate data: y = 2x + 5 + some noise
# X is value for variable x which is independent variable
X = [i for i in range(10)]
```

```
# y is for value variable y which is dependent variable
y = [2*x + 5 + random.uniform(-1, 1) for x in X]
print("X =",X)
print("y =",y)

#Graph between observed value of y for X
#Keep in mind that this is not predicted output
plt.scatter(X, y)
plt.title("Synthetic Linear Data")
plt.xlabel("X")
plt.ylabel("y")
plt.grid(True)
plt.show()
```

Output

```
X = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
y = [5.643893756714421, 7.136153854035999, 8.87579476126663,
10.90575037009317, 12.830844750745847, 15.031589612216068,
17.69384606179793, 18.837674046888537, 21.011107001151377,
23.59310303127695]
```



#Step-02: Implement Linear Regression(Least Squares)

```
def mean(values):
    return sum(values) / len(values)

def linear_regression(X, y):
    x_mean = mean(X)
    y_mean = mean(y)

    print("x_mean =",x_mean)
    print("y_mean =",y_mean)

    # Compute slope (m)
    numerator = sum((X[i] - x_mean) * (y[i] - y_mean) for i in
range(len(X)))
    denominator = sum((X[i] - x_mean)**2 for i in range(len(X)))
    m = numerator / denominator

    # Compute intercept (c)
```

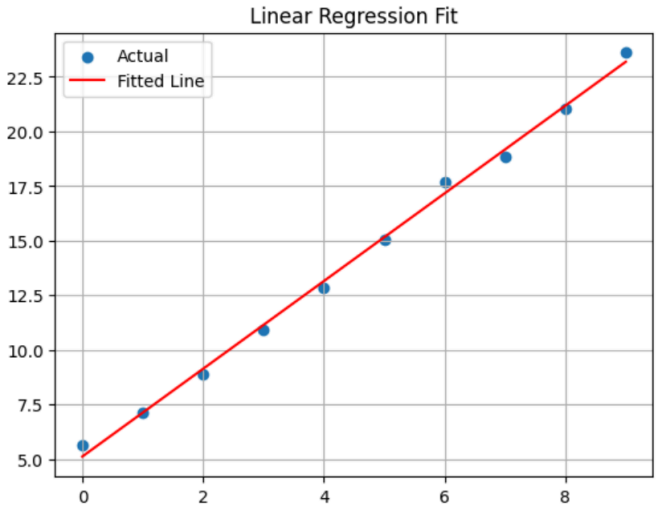
```
    c = y_mean - m * x_mean

    return m, c

m, c = linear_regression(X, y)
y_pred = [m*x + c for x in X]

print(f"Slope: {m:.2f}, Intercept: {c:.2f}")
```

Output

```
x_mean = 4.5
y_mean = 14.155975724618694
Slope: 2.01, Intercept: 5.13
```

#Step-03 Plotting

Output



Which function finds the **best-fit polynomial coefficients** for given x and y data.

np.polyfit()

Activity: Housing dataset

Import the necessary modules for data manipulation and visualization, and ensure that any non-critical warning messages are suppressed during execution

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

```
import warnings
warnings.filterwarnings('ignore')
```

Load the housing dataset from a CSV file into a Pandas DataFrame for analysis.

```
df = pd.read_csv('housing.csv')
```

Check the total number of rows and columns in the dataset to understand its size.

```
Df.shape
```

Display a summary of the DataFrame, including the column names, data types, and the presence of any null (missing) values

```
df.info()
```

Print the first five rows of the dataset to get a quick look at the actual data values.

```
df.head()
```

Generate a statistical summary (mean, standard deviation, min, max, etc.) for all the numerical columns in the DataFrame.

```
df.describe()
```

Create a scatter plot to visualize the relationship between the house area and its price to see if there is a visible correlation.

```
sns.scatterplot(df, x = df['area'], y = df['price'])
plt.show()
```

Create a side-by-side subplot containing boxplots for both 'price' and 'area' to identify potential outliers and understand the spread of the data.

```
fig, axs = plt.subplots(1, 2, figsize = (6, 3))
plt1 = sns.boxplot(df, y = df['price'], ax = axs[0])
plt1 = sns.boxplot(df, y = df['area'], ax = axs[1])
plt.tight_layout()
```

Calculate the Interquartile Range (IQR) for the 'price' column and filter out the statistical outliers to ensure they don't skew the model.

```
q1 = df['price'].quantile(0.25)
q3 = df['price'].quantile(0.75)
iqr = q3 - q1
lwb = q1 - 1.5 * iqr
uwb = q3 + 1.5 * iqr
df = df[(df['price'] >= lwb) & (df['price'] <= uwb)]
df.shape
```

Apply the IQR filtering method to the 'area' column to remove extreme values and visualize the cleaned distribution using a boxplot.

```
q1 = df['area'].quantile(0.25)
q3 = df['area'].quantile(0.75)
iqr = q3 - q1
lwb = q1 - 1.5 * iqr
uwb = q3 + 1.5 * iqr
df = df[(df['area'] >= lwb) & (df['area'] <= uwb)]
```

Check Boxplot for Price

```
sns.boxplot(df, y = df['price'])
plt.show()
```

Check Boxplot for Area

```
sns.boxplot(df, y = df['area'])
plt.show()
```

Generate a scatter plot of the cleaned data to verify the linear relationship between 'area' and 'price' after removing outliers.

```
sns.scatterplot(df, x = df['area'], y = df['price'])
plt.show()
```

Installing Scikit-learn

!pip install scikit-learn

Doc : https://scikit-learn.org/stable/user_guide.html

Split the dataset into training (70%) and testing (30%) sets, ensuring the split is reproducible by setting a random seed.

```
from sklearn.model_selection import train_test_split
df_train, df_test = train_test_split(df, test_size = 0.3, train_size = 0.7, random_state=42)
#print(df.shape, df_train.shape, df_test.shape)
print(df_train.head())
print(df_test.head())
```

Initialize a Linear Regression model, fit it using the training data, and extract the resulting slope (coefficient) and intercept.

```
from sklearn.linear_model import LinearRegression
#help(LinearRegression)
model = LinearRegression()
model.fit(df_train[['area']], df_train['price'])
print(model.coef_, model.intercept_)
```

Use the trained model to predict house prices for the 'area' values in the test dataset.

```
result = model.predict(df_test[['area']])
```

Import the Mean Squared Error metric from scikit-learn to calculate the average squared difference between the actual and predicted prices. Additionally, compute the Root Mean Squared Error (RMSE) to express the error in the same units as the house price.

```
from sklearn.metrics import mean_squared_error
mean_squared_error(df_test['price'], result)
```

Evaluation metrics

1. Introduction

After building a regression model, we must answer the question: **"How good is this model?"**
Since a model will rarely predict the exact value every time, we need metrics to quantify the **error** (the difference between the actual value Y and the predicted value y^.
There are two main categories of metrics:

1. **Error Metrics:** Measure the average distance between predictions and actuals (Lower is better).
2. **Goodness of Fit:** Measure how well the model explains the variance in the data (Higher is better).

2. Quick Summary Table

Metric	Full Name	Primary Use Case	Rule of Thumb
MAE	Mean Absolute Error	When outliers shouldn't be penalized heavily.	**Lower** is better.

MSE	Mean Squared Error	Mathematical optimization (easier to differentiate).	**Lower** is better.
RMSE	Root Mean Squared Error	The industry standard; penalizes large errors.	**Lower** is better.
$R2$	R-Squared	Checking "Goodness of Fit" (0 to 1 scale).	**Higher** is better.
Adj $R2$	Adjusted R-Squared	Comparing models with *multiple* features.	**Higher** is better.

Consider the dataset containing prices of toys-

Observation	Actual Value	Predicted Value
1	100	120
2	100	80
3	100	110
4	100	90
5	100	100

First find the error using the sum of error

Observation	Actual	Predicted	Error
1	100	120	−20
2	100	80	+20
3	100	110	−10
4	100	90	+10
5	100	100	0

So Sum of Error is

$$(-20) + 20 + (-10) + 10 + 0 = 0$$

Although the sum of errors is zero, this does not indicate that the model is perfect. The positive and negative errors cancel each other out, resulting in a net error of zero even when individual predictions are significantly inaccurate. Therefore, using the simple sum of errors as an error function is misleading and meaningless for evaluating model

Mean Square Error:

Now compute the **mean square error** for the same; here is formula

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

Observation	Actual	Predicted	Error	Squared Error
1	100	120	−20	400
2	100	80	+20	400
3	100	110	−10	100
4	100	90	+10	100
5	100	100	0	0

$$\text{MSE} = \frac{400 + 400 + 100 + 100 + 0}{5} = \frac{1000}{5} = 200$$

Advantages of MSE

A. Always non-negative, focus on large errors i.e. sensitive to outliers
B. Low MSE means low error
C. Provide smooth gradient for algorithms

To differentiate MSE, we focus on how the error changes with respect to the model's predictions. For simplicity, we differentiate the squared error for a single data point. Mean Squared Error (MSE) is just the average of these terms, and differentiation behaves the same except for a constant factor. If we look at the core for a single point:

$$Error = (y_i - \hat{y}_i)^2$$

When we take the derivative (find the slope) of this squared term with respect to the prediction:

$$\text{Slope} = 2 \times (y - \hat{y}) \times (-1) = -2(y - \hat{y})$$

This slope tells us about the direction and speed of change i.e. it tells in which way should one update the predicted output. The gradient points in the direction of steepest increase. Gradient descent updates parameters in the opposite direction of the gradient.

The second derivative tells us how the slope itself is changing. Let us differentiate the slope again with respect to predicted output.

$$\frac{d^2\text{Error}}{d\hat{y}^2} = 2$$

For the second derivative if value is

1. Positive means the curve will be U shaped (convex)
2. Negative means the curve will be ∩ shaped (concave)
3. Zero means the curve will be flat/linear

Here, since the second derivative is a positive constant (2), the error function is always convex.

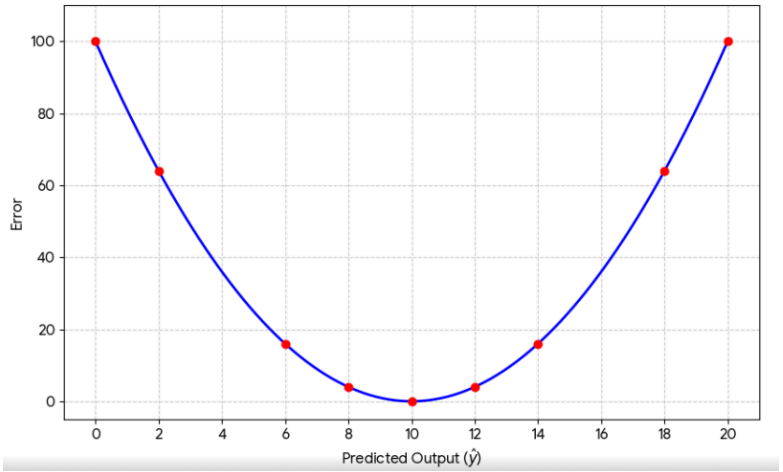
Tip: We vary the predicted value only to understand the loss function; the model itself always produces a single prediction for a given input.

Let us take a numerical example, say the actual output y is 10 and the predicted output $\hat{y}$ is [0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20].

Predicted ($\hat{y}$)	**Error (10−$\hat{y}_i$^)**	**Squared Error ($\hat{y} - \hat{y}_i$)²**
0	10	100
2	8	64
4	6	36

6	4	16
8	2	4
10	0	0
12	-2	4
14	-4	16
16	-6	36
18	-8	64
20	-10	100

Let us take predicted output of the X-axis and squared error on the Y-axis to see the shape



Where you are	**Error trend**	**Slope sign**	**Update direction**
Left of minimum	100 to 0	Negative	Move right
At minimum	0	Zero	Stop
Right of minimum	0 to 100	Positive	Move left

The MSE is a smooth parabola because its slope is continuous everywhere, so the optimizer never gets stuck at a mathematical dead-end. It can always calculate a gradient to take the next step. With respect to the prediction, the gradient magnitude is proportional to the error. If the error is large (e.g., 100), then the slope is also large (i.e., 200), and the model takes a large step toward the answer. As the error shrinks (e.g., 0.1), the slope becomes very small (0.2). The model automatically slows down, allowing it to settle precisely into the minimum without overshooting it.

For linear regression, this loss surface looks like a single, perfect bowl. This guarantees that if the model follows the negative gradient, it will eventually reach the global minimum (the best possible answer) rather than getting stuck in a shallow fake minimum. In other words, for linear regression, the MSE loss is convex, meaning it has a single global minimum and no local minima.

Squared error creates a smooth U-shaped loss curve: far from the true value the slope is steep (fast correction), near the true value the slope is flat (fine adjustment), and at perfect prediction the error and slope are both zero.

Disadvantage

Compute squared Error for both the datasets

Dataset A: Human Age (Years)

Actual	Predicted
2	7
25	20
60	65
10	15
80	75

Here is calculation for the squared error-

Actual	Predicted	Error	Squared Error
2	7	−5	25
25	20	5	25
60	65	−5	25
10	15	−5	25
80	75	5	25

The Mean square error is

$$(25+25+25+25+25)/5 = 125/5 = 25$$

Dataset B: House Price (USD)

Actual	Predicted
500000	505000
600000	595000
400000	405000
300000	295000
900000	905000

Here is calculation for the squared error-

Actual	Predicted	Error	Squared Error
500000	505000	−5000	25,000,000
600000	595000	5000	25,000,000
400000	405000	−5000	25,000,000
300000	295000	5000	25,000,000
900000	905000	−5000	25,000,000

The Mean square error is

$$(25{,}000{,}000 \times 5) / 5 = 25{,}000{,}000$$

Ask yourself, the mean square error for Age dataset is 25 but for the housing price dataset it is 25,000,000. Which model is better? The looks model is working on the age data set. Now, look at Observation 1 for both: you missed a 2-year-old's age by 5 years, and you missed a

$500,000 house by $5,000. In which scenario would your 'client' (the parent or the homebuyer) be more upset?

Now the next question, why does a higher MSE does not always mean a poorer model?

MSE is **scale dependent** i.e. you cannot compare the MSE of one problem to the MSE of a different problem because the scale of the target variable dictates the size of the error also MSE is less interpretable because of scale dependency

Root Mean Square Error:

Let us move to next cost function **RMSE (Root Mean Square Error);** here is formula for to compute the error

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

Compute the RMSE for this dataset

Observation	Actual Value	Predicted Value
1	100	120
2	100	80
3	100	110
4	100	90
5	100	100

The squared sum is

Observation	Actual	Predicted	Error	Squared Error
1	100	120	−20	400
2	100	80	+20	400
3	100	110	−10	100
4	100	90	+10	100
5	100	100	0	0

$$\text{MSE} = \frac{400 + 400 + 100 + 100 + 0}{5} = \frac{1000}{5} = 200$$

Just take the square root

$$\text{RMSE} = \sqrt{200} \approx 14.14$$

Advantages

A. The error is also available in the same unit (in rs in which the input was)
B. It is easy to understand for human
C. Can be used for the gradient descent because it is differentiable

Disadvantage

Finding the slop is messy, Say u is mean square error

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

Then, the RMSE is simply the square root of U

$$RMSE = \sqrt{U}$$

To find the derivative of RMSE with respect to a specific prediction ($\hat{y}$), we break it into two layers:

$$\frac{\partial RMSE}{\partial \hat{y}_i} = \frac{\partial \sqrt{U}}{\partial U} \cdot \frac{\partial U}{\partial \hat{y}_i}$$

This will lead to

$$\frac{\partial RMSE}{\partial \hat{y}_i} = \left(\frac{1}{2\sqrt{U}}\right) \cdot \left(\frac{-2}{n}(y_i - \hat{y}_i)\right)$$

So the final formula is

$$\text{Slope}_{\text{RMSE}} = \frac{-(y_i - \hat{y}_i)}{n \cdot RMSE}$$

Notice that RMSE is in the denominator i.e. as the model becomes perfect and the error approaches zero, you are essentially dividing by zero. The slope becomes undefined or infinite at the exact moment the model reaches the correct answer.

Tip: Use MSE to train the model (because the math is "clean" and the slope is "smooth") but report RMSE to the scholars (because the units are "intuitive").

Mean Absolute Error:

Let us move to next Error function called the **Mean Absolute Error**

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

Compute the MAE for this dataset

Observation	Actual Value	Predicted Value
1	100	120
2	100	80
3	100	110
4	100	90
5	100	100

The Mean absolute Error is

Observation	Actual	Predicted	Absolute Error

1	100	120	20
2	100	80	20
3	100	110	10
4	100	90	10
5	100	100	0

$$\mathbf{MAE} = \frac{20 + 20 + 10 + 10 + 0}{5} = \frac{60}{5} = 12$$

Advantage

A. The error is also available in the same unit (In rs, in which the input was)
B. We use MAE because it is less sensitive to outliers. Even though the 'sharp corner' makes the math harder for the computer to find the exact minimum, the resulting model is often more realistic because it doesn't let one 'crazy' data point ruin the predictions for all the 'normal' ones.
C. This is robust to outlier

Disadvantages

- Mean Absolute Error (MAE) is not differentiable at the point where the error is exactly zero. MAE is a "V" Shape y = |x|. It has a sharp, pointed corner at the origin (0,0). In calculus, to find a derivative (slope), the curve must be "smooth." At that sharp point of the "V," the direction of the line changes instantly from -1 to +1. Because there is no single, clear slope at that exact point, we say it is not differentiable there.
  Analogy: The slope is just as steep at the bottom as it is at the top. The ball will likely "jump" back and forth across the center (oscillate) because it never receives a signal to slow down.

R^2 Score:

Let us move to another cost function called **R^2 Score (coefficient of determination)**; The formula is

$$R2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}$$

Here

$SS_{\text{res}} = \sum(y_i - \hat{y}_i)^2$: The residual sum of squares (a.k.a. the sum of squared errors).

$SS_{\text{tot}} = \sum(y_i - \bar{y})^2$: The total sum of squares (total variance in the data).

y_i are the true values, $\hat{y}_i$ are the predicted values, and $\bar{y}$ is the mean of the true values. The R^2 score usually lies in the range of 0 to 1 but it can be negative. R^2 is a measure of the **goodness of fit** of a model. In regression, the R^2 coefficient of determination is a statistical measure of how well the regression predictions approximate the real data points. An R^2 of 1 indicates that the regression predictions perfectly fit the data.

Compute the R^2 score for following

Observation	Actual	Model A	Model B	Model C
1	10	12	30	50
2	20	18	30	50

3	30	33	30	50
4	40	45	30	50
5	50	52	30	50

Step 1: The Baseline Calculation (SS_{tot})

Actual Values (y): $[10, 20, 30, 40, 50]$

Mean ($\bar{y}$): $(10 + 20 + 30 + 40 + 50)/5 = \mathbf{30}$

SS_{tot} **Calculation:**

$$(10 - 30)^2 + (20 - 30)^2 + (30 - 30)^2 + (40 - 30)^2 + (50 - 30)^2$$

$$(-20)^2 + (-10)^2 + 0^2 + 10^2 + 20^2 = 400 + 100 + 0 + 100 + 400 = \mathbf{1,000}$$

Step 2: Model-A

Predictions ($\hat{y}$): $[12, 18, 33, 45, 52]$

Residuals ($y - \hat{y}$): $[-2, 2, -3, -5, -2]$

Sum of Squared Errors (SS_{res}):

$$(-2)^2 + (2)^2 + (-3)^2 + (-5)^2 + (-2)^2 = 4 + 4 + 9 + 25 + 4 = \mathbf{46}$$

R^2 **Calculation:**

$$R^2 = 1 - \frac{46}{1,000} = 1 - 0.046 = \mathbf{0.954}$$

Step 3: Model-B

Predicted Output: $[30, 30, 30, 30, 30]$

Residuals $(y - \hat{y})$**:** $[-20, -10, 0, 10, 20]$

SS_{res}**:** $(-20)^2 + (-10)^2 + 0^2 + 10^2 + 20^2 = \mathbf{1,000}$

$R2$ **Score:** $1 - (1,000/1,000) = \mathbf{0}$

Step 3: Model-C

Predicted Output ($\hat{y}$)	$[50, 50, 50, 50, 50]$
Residual ($y - \hat{y}$)	$[-40, -30, -20, -10, 0]$
SS_{res}	$(-40)^2 + (-30)^2 + (-20)^2 + (-10)^2 + 0^2 = \mathbf{3,000}$
R^2 **Score**	$1 - (3,000/1,000) = \mathbf{-2.0}$

R^2 is often interpreted as the proportion of response variation "explained" by the regressors in the model.

1. *The Model-A model explained 95.4% of the variance in the data. It is much better than simply guessing the average.*
2. *The Model-B just predicts the mean for every single house or person. An R^2 of 0 means your model is no better than a simple average. It has "zero predictive power."*
3. *If your model is so bad that its errors SS_{res} are larger than the natural variance of the data SS_{tot}, the R^2 goes negative. This model is literally "worse than guessing the average." This is what happened with Model-C*

Advantages

- It is unit-less, it measures the fraction of variance on the target captured by the model. It indicates fit on the training data but does not indicate whether the predictions are accurate on the accurate on new data
- It can be used to compare models on the same dataset

Now find R^2 score of both the datasets
The first Dataset: Human Age (Years)

Actual	Predicted
2	7
25	20
60	65
10	15
80	75

Here is calculation for the R^2 Score

Actuals: $[2, 25, 60, 10, 80]$ | **Mean:** 35.4

SS_{tot} **(Baseline Variance):** 4353.2

Errors: $[5, -5, 5, 5, -5]$ | **Squared Errors:** $[25, 25, 25, 25, 25]$

MSE: 25.0

R^2 **Score:** $1 - (125/4353.2) \approx \mathbf{0.971}$

The Second Dataset: House Price (USD)

Actual	Predicted
500000	505000
600000	595000
400000	405000
300000	295000
900000	905000

Here is calculation for the R^2 Score

Actuals: $[500k, 600k, 400k, 300k, 900k]$ | **Mean:** $540k$

SS_{tot} **(Baseline Variance):** $212,000,000,000$

Errors: $[5000, -5000, 5000, -5000, 5000]$

MSE: $25,000,000.0$

R^2 **Score:** $1 - (125,000,000/212,000,000,000) \approx \mathbf{0.999}$

Feature	Dataset A (Age)	Dataset B (Price)	The Conclusion
MSE	25	25,000,000	MSE lies: It makes the Price model look a million times worse.
R^2 Score	0.971	0.999	R2 reveals truth: The House model is actually much more precise relative to its scale.

Tip: MSE tells you the size of the mistake. R^2 tells you the significance of the mistake.

Disadvantages

- R^2 score increases when adding more predictors even if these predictors are irrelevant so high R^2 can be misleading. This is because SS_{tot} is always constant and the regression model tries to decrease the value of SS_{res} by finding some correlation with this new attribute hence the overall value of r-square increases, which can lead to a poor regression model.
  For example, if one is trying to predict the sales of a model of car from the car's gas mileage, price, and engine power, one can include probably irrelevant factors such as the first letter of the model's name or the height of the lead engineer designing the car because the R^2 will never decrease as variables are added and will likely experience an increase due to chance alone. This leads to the alternative approach of looking at the adjusted R^2.

R^2 adjusted:

Let us move to next which is **R^2 adjusted**; Here is formula

$$\bar{R}^2 = 1 - (1 - R^2)\frac{n-1}{n-p-1}$$

Here
n: Number of samples/rows in the dataset
p: number of predictors/features/ no. of independent variables

The adjusted R-squared only increases if the newly added predictor improves the model's predicting power. Adding independent and irrelevant predictors to a regression model results in a decrease of the adjusted R-squared.

Use adjusted R2 when

a) Multiple Regression: Generally speaking, adjusted r-squared is a more reliable metric because it accounts for the number of predictors, ensuring a better evaluation of the model's performance.
b) Model Comparison: Adjusted r-squared is useful when comparing models with different numbers of predictors to choose the best-performing ones.
c) To Prevent Overfitting: Adjusted r-squared helps by penalizing the inclusion of irrelevant predictors to ensure that only predictors that improve the model's performance are included.

Multiple Linear Regression

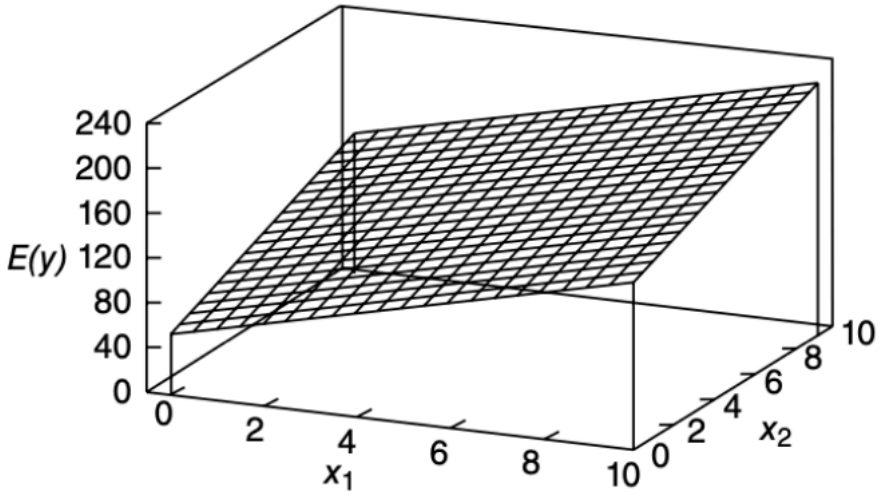
Multiple Linear Regression (MLR) is a method for estimating how several independent factors together influence a single outcome. It fits a straight-line equation to data points to reveal how each variable contributes when the others are held steady. MLR is an extension of ordinary least-squares (OLS) regression with more explanatory variables.

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_k x_k + \epsilon$$

Where-
y: response, $x_1, \ldots, x_k$: predictor
$\beta_0, \beta_1, \ldots, \beta_k$: coefficients

ϵ: error term

An example of a regression model with k = 2 predictors



The regression plane for the model E(y) = 50 + 10x_1 + 7x_2.

Letting

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} 1 & x_{11} & x_{12} & \cdots & x_{1k} \\ 1 & x_{21} & x_{22} & \cdots & x_{2k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_{n1} & x_{n2} & \cdots & x_{nk} \end{bmatrix}, \quad \boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_k \end{bmatrix}, \quad \boldsymbol{\epsilon} = \begin{bmatrix} \epsilon_1 \\ \epsilon_2 \\ \vdots \\ \epsilon_n \end{bmatrix}$$

We can rewrite the sample regression model in matrix form

$$\underbrace{\mathbf{y}}_{n\times 1} = \underbrace{\mathbf{X}}_{n\times p} \cdot \underbrace{\boldsymbol{\beta}}_{p\times 1} + \underbrace{\boldsymbol{\epsilon}}_{n\times 1}$$

Where p = k + 1 represents the number of regression parameters (note that k is the number of predictors in the model).

The LS (Least square) criterion can still be used to fit a multiple regression model

$$\hat{y} = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots + \hat{\beta}_k x_k$$

to the data as follows:

$$\min_{\hat{\boldsymbol{\beta}}} S(\hat{\boldsymbol{\beta}}) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2 = \sum_{i=1}^{n} e_i^2$$

Where for each 1 ≤ i ≤ n,

$$\hat{y}_i = \hat{\beta}_0 + \hat{\beta}_1 x_{i1} + \cdots + \hat{\beta}_k x_{ik}$$

The LS estimator of β is

$$\hat{\beta} = (\mathbf{X}'\mathbf{X})^{-1}\mathbf{X}'\mathbf{y}$$

An Example of Muliple Linear regression with R^2 score and Adjusted R^2 score

import

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.datasets import fetch_california_housing
```

```
#Load the Calfornia Housing Dataset
#help(fetch_california_housing)
```

```
housing = fetch_california_housing()
#print(type(housing))
```

```
# Create a DataFrame from the loaded data, print first 5 entries
```

```
df_house = pd.DataFrame(housing.data, columns =
housing.feature_names)
df_house.head()
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude
0	8.3252	41.0	6.984127	1.023810	322.0	2.555556	37.88	-122.23
1	8.3014	21.0	6.238137	0.971880	2401.0	2.109842	37.86	-122.22
2	7.2574	52.0	8.288136	1.073446	496.0	2.802260	37.85	-122.24
3	5.6431	52.0	5.817352	1.073059	558.0	2.547945	37.85	-122.25
4	3.8462	52.0	6.281853	1.081081	565.0	2.181467	37.85	-122.25

```
# Add the target variable to the DataFrame, print first 5 entries
```

```
df_house['Value'] = housing.target
df_house.sample(5)
```

	MedInc	HouseAge	AveRooms	AveBedrms	Population	AveOccup	Latitude	Longitude	Value
8434	4.9688	36.0	5.456853	1.000000	509.0	2.583756	33.92	-118.37	2.389
15727	3.2011	48.0	4.941463	0.946341	464.0	2.263415	37.78	-122.45	4.283
3945	6.1074	26.0	6.304094	1.007797	1597.0	3.113060	34.21	-118.62	2.586
14555	6.7120	15.0	7.844291	1.010381	1180.0	4.083045	32.96	-117.13	2.402
10105	3.6106	18.0	4.393468	1.026439	1590.0	2.472784	33.92	-117.95	1.536

 # Display the Column-name, data-type, non-null count for this dataframe

```
df_house.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20640 entries, 0 to 20639
Data columns (total 9 columns):
 #   Column      Non-Null Count  Dtype  
---  ------      --------------  -----  
 0   MedInc      20640 non-null  float64
 1   HouseAge    20640 non-null  float64
 2   AveRooms    20640 non-null  float64
 3   AveBedrms   20640 non-null  float64
 4   Population  20640 non-null  float64
 5   AveOccup    20640 non-null  float64
 6   Latitude    20640 non-null  float64
 7   Longitude   20640 non-null  float64
 8   Value       20640 non-null  float64
dtypes: float64(9)
memory usage: 1.4 MB
```

 # Divide the data into the train (0.8) and test (0.2) print the shape just for verification

```
from sklearn.model_selection import train_test_split
df_house_train, df_house_test = train_test_split(df_house,
train_size = 0.8, test_size = 0.2, random_state = 42)
print(df_house.shape, df_house_train.shape, df_house_test.shape)
```

```
(20640, 9) (16512, 9) (4128, 9)
```

 # Train the model using the input features

```
from sklearn.linear_model import LinearRegression
lr = LinearRegression()
lr = lr.fit(df_house_train.iloc[:,0:8], df_house_train.iloc[:,8])
```

 # Predictions on training data

```
y_train_pred = lr.predict(df_house_train.iloc[:,0:8])

#Compute the R2 score on training data
from sklearn.metrics import r2_score
r2_train = round(r2_score(df_house_train.iloc[:,8], y_train_pred),
4)
print(r2_train)
```

```
0.6126
```

```
#Compute the adjusted R2 score on training data
n = len(df_house_train)
p = len(df_house_train.columns) - 1
adjusted_r2_train = 1 - ((1 - r2_train) * ((n - 1) / (n - p - 1)))
adjusted_r2_train = round(adjusted_r2_train, 4)
adjusted_r2_train
```

```
0.6124
```

```
#Predict the value for test data
y_test_pred = lr.predict(df_house_test.iloc[:, 0:8])

# Compute the R2 score on testing data
r2_test = round(r2_score(df_house_test.iloc[:,8], y_test_pred), 4)
r2_test
```

```
0.5758
```

```
#Compute the adjusted R2 score on testing data
n = len(df_house_test)
p = len(df_house_test.columns) - 1
adjusted_r2_test = 1 - (1 - r2_test) * ((n - 1) / (n - p - 1))
adjusted_r2_test = round(adjusted_r2_test, 4)
adjusted_r2_test
```

```
0.5750
```

```
# Let us add 10 random columns to training data
import numpy as np

rng = np.random.default_rng(42)
for i in range(1, 11):
                                df_house_train[f'random_{i}']        =
rng.standard_normal(len(df_house_train))

df_house_train.columns

#Let us add 10 random columns to testing data
```

```
rng = np.random.default_rng(42)
for i in range(1, 11):
                                df_house_test[f'random_{i}']        =
rng.standard_normal(len(df_house_test))

df_house_test.columns
```

 # Check the shape of training and test data

```
df_house_train.shape, df_house_test.shape
```

```
((16512, 19), (4128, 19))
```

 # Change the sequence of column in both train and test

```
df_house_train = df_house_train[['MedInc', 'HouseAge', 'AveRooms',
'AveBedrms', 'Population', 'AveOccup', 'Latitude', 'Longitude',
      'random_1', 'random_2', 'random_3', 'random_4', 'random_5',
'random_5',   'random_7',   'random_8',   'random_9',   'random_10',
'Value']]
df_house_test  =  df_house_test[['MedInc',  'HouseAge',  'AveRooms',
'AveBedrms', 'Population', 'AveOccup', 'Latitude', 'Longitude',
      'random_1', 'random_2', 'random_3', 'random_4', 'random_5',
'random_5',   'random_7',   'random_8',   'random_9',   'random_10',
'Value']]
```

#Now again train the model, predict the result for train data, check the r2 score

```
lr.fit(df_house_train.iloc[:,0:18], df_house_train.iloc[:,18])
y_train_pred = lr.predict(df_house_train.iloc[:,0:18])
r2_train_again = r2_score(df_house_train.iloc[:,18], y_train_pred)
r2_train_again = round(r2_train_again, 4)
r2_train_again
```

```
0.6127
```

 # Again compute the adjusted R2 score on training data

```
n = len(df_house_train)
p = len(df_house_train.columns) - 1
adjusted_r2_train_again = 1 - ((1 - r2_train_again) * ((n - 1) /
(n - p - 1)))
adjusted_r2_train_again = round(adjusted_r2_train_again, 4)
adjusted_r2_train_again
```

```
0.6123
```

 # Again Compute the R2 score for test data

```
y_test_pred_again = lr.predict(df_house_test.iloc[:, 0:18])
r2_test_again          =          r2_score(df_house_test['Value'],
y_test_pred_again)
```

```
r2_test_again = round(r2_test_again, 4)
r2_test_again
```

0.5757

```
# Again compute the adjusted R2 score on the test data
n = len(df_house_test)
p = len(df_house_test.columns) - 1
adjusted_r2_test_again = 1 - ((1 - r2_test_again) * ((n - 1) / (n - p - 1)))
adjusted_r2_test_again = round(adjusted_r2_test_again, 4)
adjusted_r2_test_again
```

0.5738

Summarization

With 8 real features

Metric	Training	Testing
R^2	0.6126	0.5758
Adjusted R^2	0.6124	0.5750

With 18 features (10 added randomly)

Metric	Training	Testing
R^2	0.6127	0.5757
Adjusted R^2	0.6123	0.5738

In Training

- **R^2 slight increase after adding random columns** This confirms the mathematical rule that standard R^2 must increase or stay the same when you add features, even if they are random noise. The model "cheated" by finding tiny, coincidental patterns in the random numbers to fit the training data better.
- **R^2 Adjusted decreases** slightly; Unlike the standard R^2 which went up, the Adjusted R^2 went down on the training set. This is the penalty in action, the formula "realized" that the 10 new columns didn't provide enough predictive value to justify their inclusion, so it lowered the score to warn you.

In Testing

- **R^2 slightly decreases** means While the model felt more confident during training, its actual performance on unseen data dropped. This is the classic signature of Overfitting. The 10 junk columns added "noise" that confused the model's ability to generalize to new houses.
- **Adjusted R^2 dropped** much further away from the standard R^2 than it did in the 8-column model. The formula penalized you twice: first, your predictions actually got worse (lower R^2), and second, the formula increased the penalty because you used 18 features instead of 8. It effectively "double-punishes" a model that tries to hide its poor performance behind extra variables.

Note: The n/p ratio: If your test set is small (e.g., only 50 rows) and you have 18 features, your Testing Adjusted R^2 would **crash** to a very low number, even if the standard R^2 looked okay. Because you have over 4,000 rows in your test set, the penalty is "diluted." The lesson

here is: A large dataset can hide the damage done by junk features, but Adjusted R^2 will always find it.

Summary Cheat Sheet

Feature	R2	Adjusted R2
Full Name	Coefficient of Determination	Adjusted Coefficient of Determination
Behavior	**Always increases** (or stays same) when you add variables.	**Can decrease** if the new variable is useless.
Bias	Biased towards complex models (overfitting).	Biased towards simpler, effective models (parsimony).
When to use	Only for Simple Linear Regression (1 variable).	Mandatory for Multiple Linear Regression.

Assumptions of Linear Regression

Linear Regression is a powerful predictive tool, but its reliability depends on specific conditions. If these assumptions are violated, the model's predictions may be biased, and the statistical conclusions (like p-values) may be invalid. We primarily focus on the residuals (errors), defined as:

$$\mathbf{Error} = \mathbf{y}_{\mathbf{actual}} - \mathbf{y}_{\mathbf{predicted}}$$

Here is an example that we will use--

```
import numpy as np
import pandas as pd
import statsmodels.api as sm
import matplotlib.pyplot as plt
import seaborn as sns
import scipy.stats as stats
from statsmodels.stats.stattools import durbin_watson

# Generate dummy data
np.random.seed(42)
X = np.random.rand(100, 1) * 10
y = 2.5 * X + np.random.normal(0, 2, (100, 1)) # y = 2.5x + noise

# Fit the OLS model
X_with_const = sm.add_constant(X) # Add intercept column
```

```
model = sm.OLS(y, X_with_const).fit()

# Calculate Residuals
residuals = model.resid
fitted_values = model.predict(X_with_const)
```

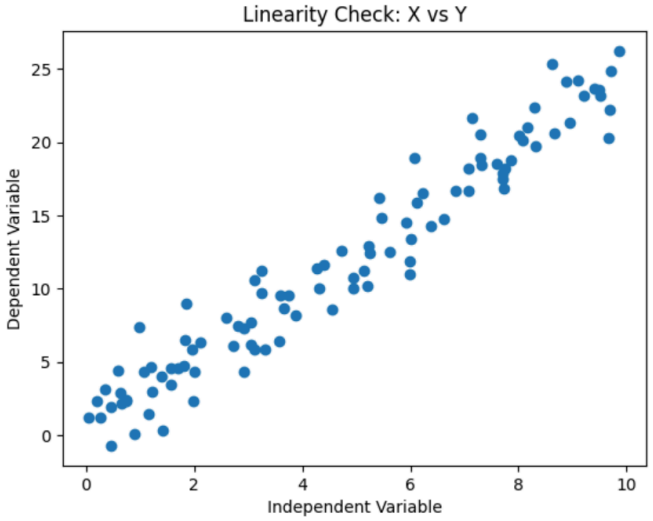
➢ **Linearity:** The first and most crucial assumption is that a linear relationship exists between the independent variable(s) and the dependent variable. This means that the change in the dependent variable is proportional to the change in the independent variable. If you were to plot the independent variable against the dependent variable, the data points should roughly form a straight line. To check:

A. Scatter Plots: The most straightforward way to visually inspect for linearity is to create scatter plots of the dependent variable against each independent variable.
B. Residual Plots: Plotting residuals against predicted values can also help. If the relationship is linear, the residuals should be randomly scattered around zero.

Consequences of violation: If the relationship is non-linear, a linear regression model will fail to capture the true underlying pattern, leading to biased estimates and poor predictive performance.

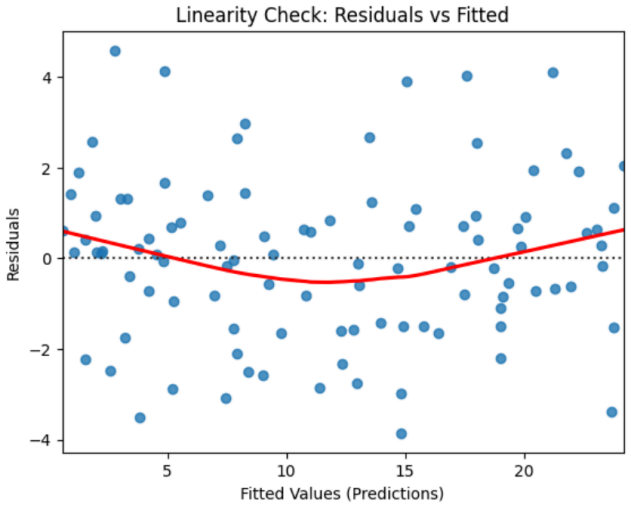
Here's an example of a scatter plot showing a linear relationship:

```
# 1. Scatter Plot (Simple check)
plt.scatter(X, y)
plt.title("Linearity Check: X vs Y")
plt.xlabel("Independent Variable")
plt.ylabel("Dependent Variable")
plt.show()
```



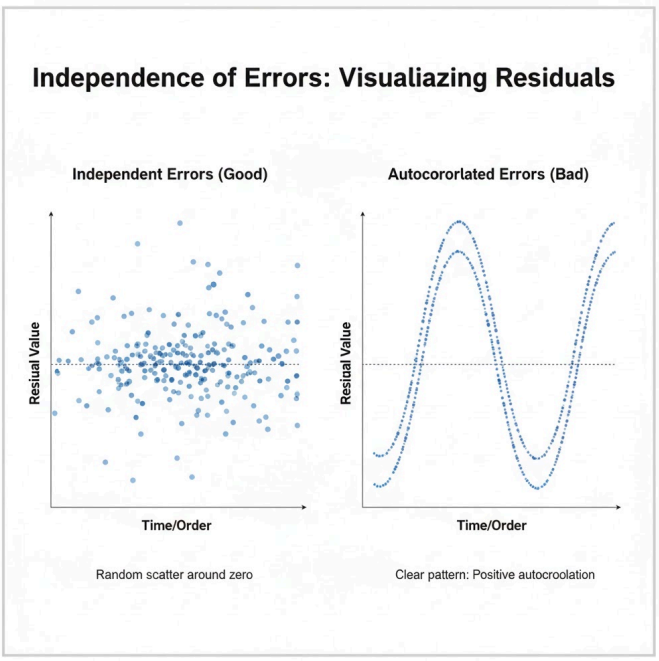

```
# 2. Residuals vs Fitted (The Pro check)
sns.residplot(x=fitted_values,    y=residuals,    lowess=True,
line_kws={'color': 'red'})
plt.title("Linearity Check: Residuals vs Fitted")
plt.xlabel("Fitted Values (Predictions)")
plt.ylabel("Residuals")
```

```
plt.show()
```



➢ **Independence of Errors (No Autocorrelation):** This assumption states that the residuals (errors) of the model should be independent of each other. In other words, the error for one observation should not be related to the error for any other observation. There should be no pattern in the errors over time or across observations. This is particularly important in time series data, where consecutive observations might be correlated. To check this we have

A. Durbin-Watson Statistic: This statistic is commonly used to detect the presence of autocorrelation. A value close to 2 suggests no autocorrelation. Values significantly less than 2 indicate positive autocorrelation, while values significantly greater than 2 indicate negative autocorrelation.
B. Residual Plots (against time or order): Plotting residuals against the order of data collection can reveal patterns.



Consequences of violation: If errors are correlated, the standard errors of the regression coefficients will be underestimated, leading to inflated t-statistics and a higher chance of incorrectly rejecting the null hypothesis (Type I error).

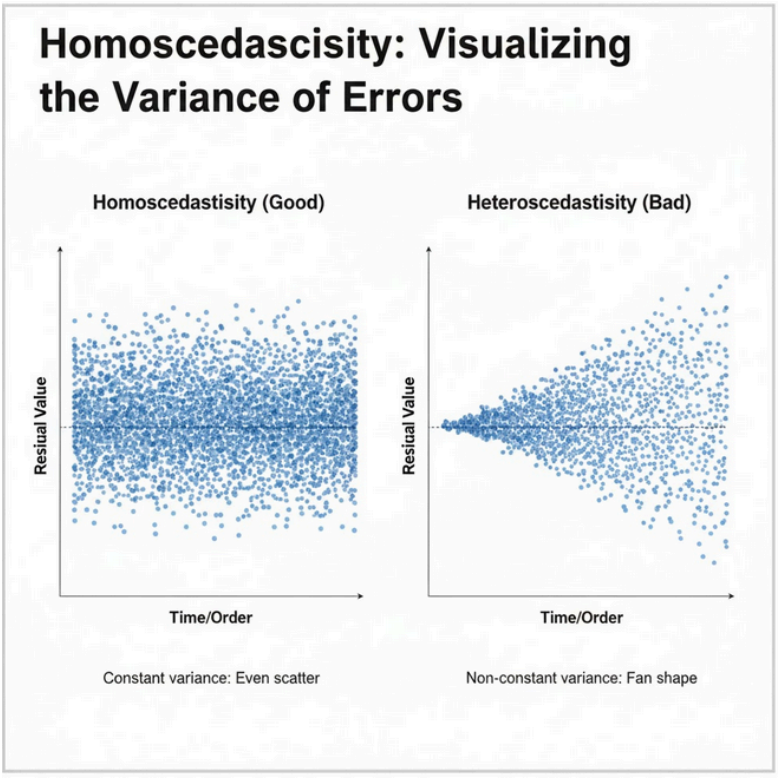
Here's an illustration of independent errors (random scatter) versus autocorrelated errors (a pattern):

```
# The input is the array of residuals
dw_score = durbin_watson(residuals)
print(f"Durbin-Watson Score: {dw_score}")
if 1.5 < dw_score < 2.5:
  print("Assumption Met: Little to no autocorrelation.")
else:
     print("Assumption Violated: Check for time-series patterns.")
```

Output:

```
Durbin-Watson Score: 2.2849976037296655
Assumption Met: Little to no autocorrelation.
```

- **Homoscedasticity** refers to the assumption that the variance of the residuals is constant across all levels of the independent variables. In simpler terms, the spread of the residuals should be roughly the same throughout the range of predicted values. The "scatter" of the data points around the regression line should be consistent, not fanning out or narrowing in. To check:
  A. Residual Plots (against predicted values): Plotting residuals against the predicted values (or against each independent variable) is the primary method. Look for a consistent band of residuals around zero. A "fan" or "cone" shape indicates heteroscedasticity.



B. Statistical Tests: **Breusch-Pagan** test and White test are formal statistical tests for homoscedasticity.

Consequences of violation (Heteroscedasticity): Heteroscedasticity does not bias the regression coefficients, but it does make them inefficient. The standard errors will be incorrect, leading to unreliable hypothesis tests and confidence intervals.

Here's a visual comparison of homoscedasticity and heteroscedasticity:

```
import statsmodels.stats.api as sms
test_stat, p_value, _, _ = sms.het_breuschpagan(residuals,
X_with_const)
print(f"Breusch-Pagan P-Value: {p_value}")
if p_value > 0.05:
          print("Assumption Met: Homoscedasticity (Equal
Variance).")
else:
    print("Assumption Violated: Heteroscedasticity present.")
```

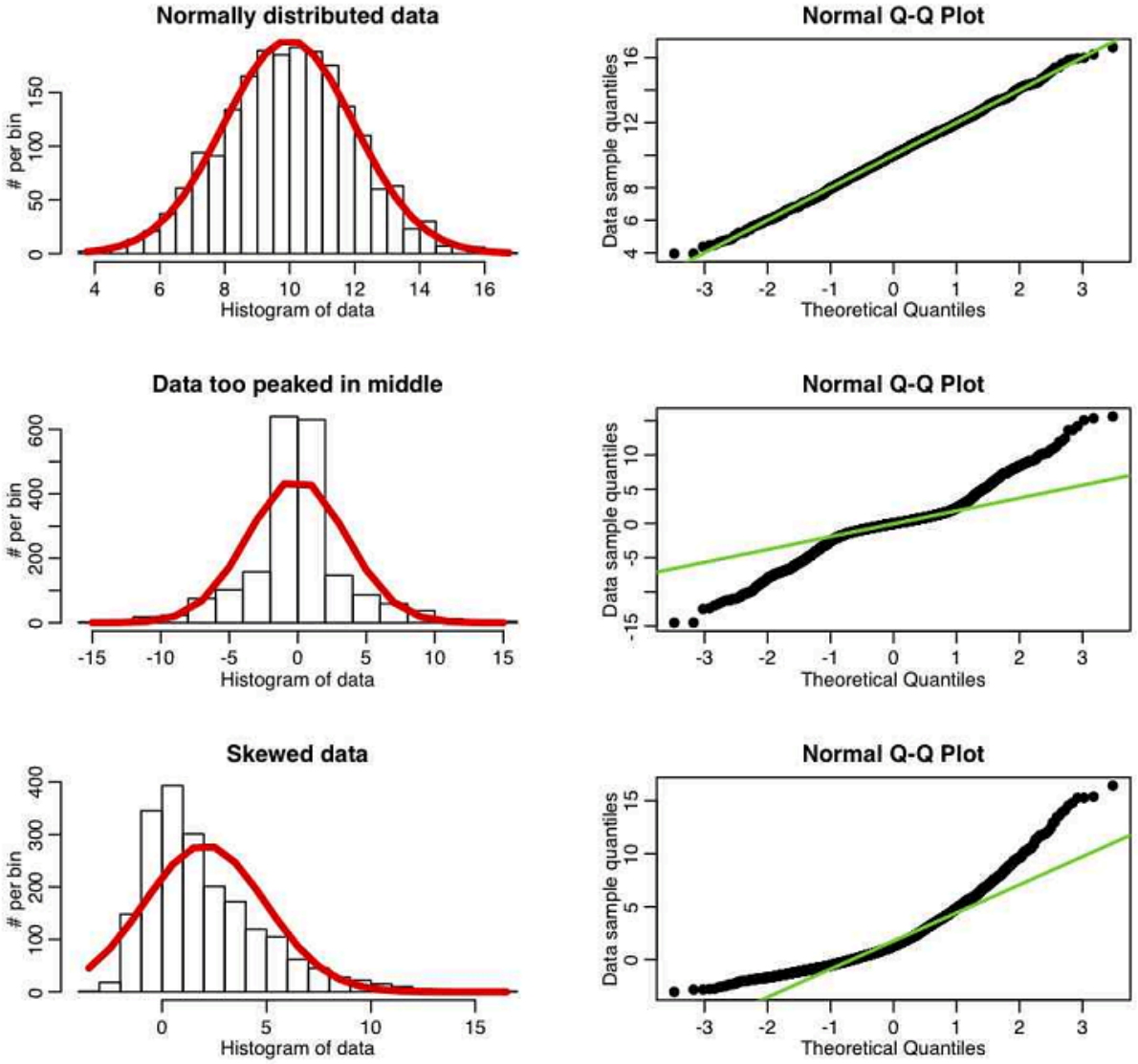
Output:

```
Breusch-Pagan P-Value: 0.906362006628118
Assumption Met: Homoscedasticity (Equal Variance).
```

➢ **Normality of Residuals (Errors):** This assumption states that the residuals (the differences between observed and predicted values) should follow a normal distribution with a mean of zero. It is a common misconception that the features (X) or the target (Y) must be normally distributed. Linear regression actually only requires the errors to be normal. This is essential for conducting hypothesis tests (like t-tests

for coefficients) and calculating confidence intervals. If your errors aren't normal, your p-values might be deceptive. To check:

A. A. Q-Q Plot (Quantile-Quantile Plot): Points should fall approximately along a straight 45-degree line.

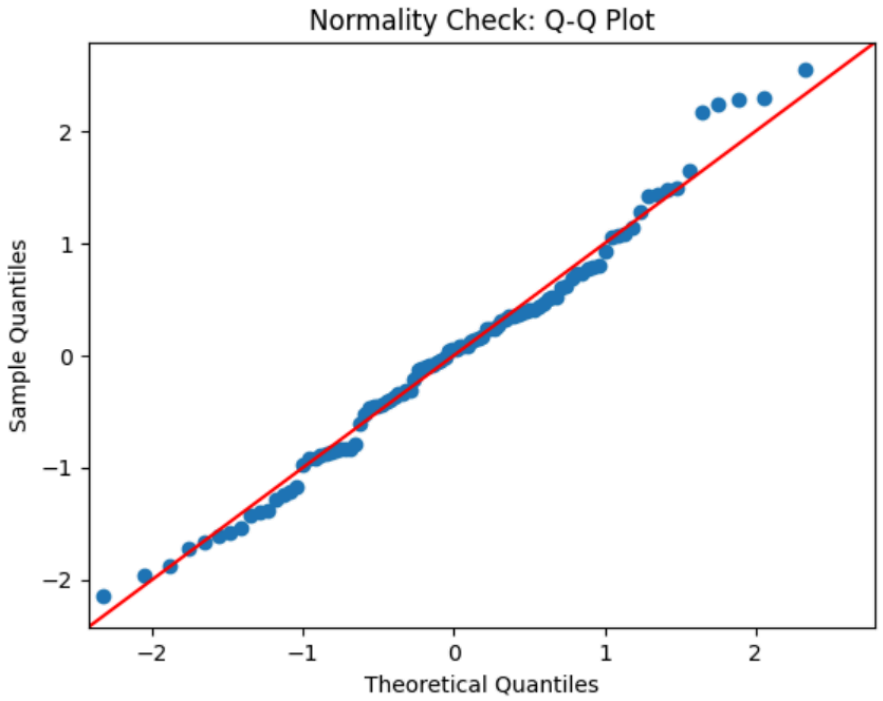


B. Histogram of Residuals: Look for a classic "bell curve" shape.
C. Shapiro-Wilk or Kolmogorov-Smirnov Tests: Statistical tests that check if a distribution deviates significantly from normality.

The "Big Data" Exception: According to the Central Limit Theorem, as your sample size becomes large (usually n > 30), the normality assumption becomes less critical because the distribution of the estimate itself will tend toward normality anyway.

An Example

```
sm.qqplot(residuals, line='45', fit=True)
plt.title("Normality Check: Q-Q Plot")
plt.show()
```




```
#Statistical: Shapiro-Wilk Test
stat, p_value = stats.shapiro(residuals)
print(f"Shapiro-Wilk P-Value: {p_value}")

if p_value > 0.05:
  print("Assumption Met: Residuals are Normal.")
else:
  print("Assumption Violated: Residuals are NOT Normal.")
```

Output

```
Shapiro-Wilk P-Value: 0.2984161405804421
Assumption Met: Residuals are Normal.
```

➢ **Multicollinearity**: It applies to Multiple Linear Regression. Independent variables should not be highly correlated with each other. e.g. If two variables are highly correlated (e.g., "Temperature in Celsius" and "Temperature in Fahrenheit"), the model can't tell which one is actually causing the change in the target. To check:

A. Variance Inflation Factor (VIF).
   VIF = 1: No correlation.
   VIF > 5 or 10: High multicollinearity that needs fixing.

```
from statsmodels.stats.outliers_influence import
variance_inflation_factor

vif_data = pd.DataFrame()
vif_data["Feature"] = ["Const", "X1"] # Replace with your
column names
```

```
vif_data["VIF"] = [variance_inflation_factor(X_with_const,
i) for i in range(X_with_const.shape[1])]

print(vif_data) # Look for VIF > 5
```

Output

```
    Feature  VIF
0   Const    3.523199
1      X1    1.000000
```

➢ **Absence of Endogeneity/Exogeneity:** Independent variables in the regression model should not be correlated with the error term.

- Endogeneity causes biased and inconsistent parameter estimates.
- Inference based on such coefficients becomes unreliable.
- Instrumental variables or additional predictors can address this issue.

The Hausman test serves as a widely used method for detecting endogeneity within regression models. The test compares the coefficients estimated from a model that assumes exogeneity with those derived from a model that takes potential endogeneity into account. If the coefficients from the two models differ significantly, it indicates the presence of potential endogeneity in the model.

```
from linearmodels.iv import IV2SLS

# 1. We need an "Instrumental Variable" (Z)
# A good instrument correlates with X but NOT with the noise
in y.
Z = X + np.random.normal(0, 1, (100, 1))

# 2. Add constants for the models
df = pd.DataFrame({'y': y.flatten(), 'X': X.flatten(), 'Z':
Z.flatten()})
df['const'] = 1

# 3. Fit the IV Model (2SLS)
# Formula: y ~ const + [X ~ Z]
iv_model = IV2SLS(df.y, df[['const']], df.X, df.Z).fit()

# 4. Perform the Wu-Hausman Test
hausman_test = iv_model.wu_hausman()
print(hausman_test)
```

Output

```
Wu-Hausman test of exogeneity
H0: All endogenous variables are exogenous
Statistic: 0.0485
P-value: 0.8261
Distributed: F(1,97)
```

Gradient Descent

Gradient descent is a generic optimization algorithm capable of finding optimal solutions to a wide range of problems such that for the problem you must define an "Error" (Cost Function) and calculate a derivative.

Application of gradient descent

In machine learning

1. Linear Regression: GD minimizes Mean Squared Error (MSE) to fit a line or plane to data for continuous predictions like house prices.
2. Logistic Regression: GD minimizes Log Loss (Binary Cross-Entropy) to establish an optimal decision boundary for binary classification tasks like spam detection.
3. Support Vector Machines (SVM): GD (specifically Sub-gradient Descent) minimizes Hinge Loss (which is not perfectly smooth) to find the maximum margin or "widest street" between data classes.

In deep learning

1. Neural Networks (Backpropagation): Backpropagation applies the Chain Rule with Gradient Descent to update millions or billions of weights, effectively reducing overall prediction error.
2. Convolutional Neural Networks (CNNs): Used for image recognition tasks like FaceID and X-ray analysis, GD optimizes the filters responsible for detecting edges, shapes, and textures.
3. Large Language Models (Transformers): GD (specifically variants like Adam) serves as the engine that trains models like Gemini and ChatGPT by optimizing their trillions of parameters for text generation and understanding.

In Real Life

1. Recommendation Systems (Netflix/Spotify): Using Matrix Factorization, GD minimizes the difference between predicted and actual ratings to "fill in the blanks" for content a user hasn't seen yet.
2. Reinforcement Learning (Game AI/Robotics): Through Policy Gradient methods, GD (or Ascent) optimizes the agent's actions to maximize rewards or minimize penalties while learning tasks like walking or playing chess.

Why do we need gradient descent over OLS

Imagine you have a dataset with 100,000 different features (like age, location, income, etc., for a house price model). To use the direct math formula, you first have to create a "summary table" (the X'X matrix) that compares every feature against every other feature. The problem is It's Too Big. A 100,000 by 100,000 table has 10 billion cells. Even if you use a standard amount of computer memory for each number, that single table would take up 80GB of RAM. Most home laptops only have 8GB or 16GB of RAM. Not only do you have to store it, but you also have to "invert" it (a very complex math operation). This process fills in even more blanks, making the data "dense" and heavy.

To overcome this, we have to use gradient descent as it doesn't need to create that massive 80GB table. It looks at small batches of data at a time. It never tries to solve the whole puzzle in one giant leap; it just takes small steps toward the

right answer. It can run on a standard computer because it only needs to remember the current "weights," not every possible interaction between features.

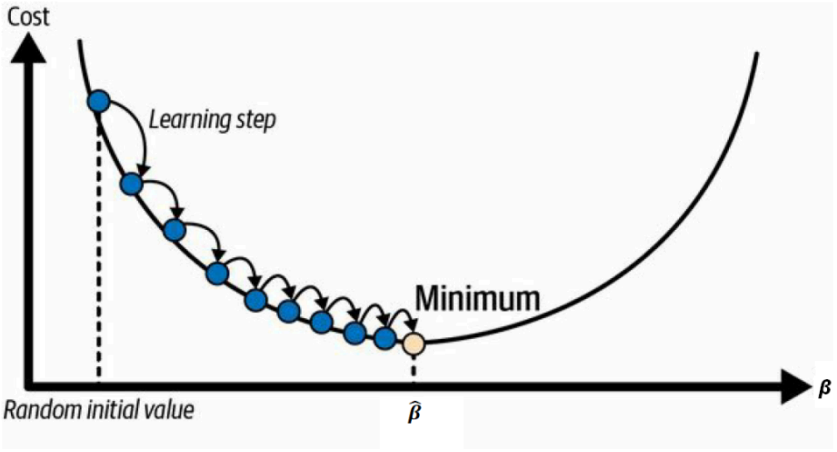
Here is a table to compare the Closed-Form (Normal Equation) and Gradient Descent

Closed-Form (Normal Equation)	**Gradient Descent**
Analytical Approach: Calculates the exact solution using a specific formula ($\beta = (X^TX)^{-1}X^Ty$).	Iterative Approach: Starts with random values and updates them repeatedly to reduce error.
One-step calculation hence Fast for small data	Scales better as the number of features increases so fast for large data
Computing the matrix inverse becomes extremely slow as features (n) grow because computation complexity for matrix inversion is High $O(n^3)$	Every iteration is computationally cheaper than a matrix inversion due to lower complexity $O(kn^2)$
No Hyper-parameters to set: i.e. learning rate or iteration.	Hyper-parameters setting requires: Requires tuning the learning rate (η) and number of iterations.
Fails if the matrix X^TX is non-invertible (singular) i.e. Invertibility	Requires a smooth, differentiable loss function to calculate gradients.
Suitable for Small datasets with few features (e.g., < 10,000).	Suitable for Large-scale machine learning and Deep Learning (Neural Networks).

The general idea of gradient descent is to tweak parameters iteratively in order to minimize a cost function. It measures the local gradient of the error function with regard to the parameter vector β, and it goes in the direction of descending gradient. Once the gradient is zero, you have reached a minimum.

Analogy: Suppose you are lost in the mountains in a dense fog, and you can only feel the slope of the ground below your feet. A good strategy to get to the bottom of the valley quickly is to go downhill in the direction of the steepest slope.

In practice, you start by filling β with random values (this is called random initialization). Then you improve it gradually, taking one baby step at a time, each step attempting to decrease the cost function (e.g., the MSE), until the algorithm converges to a minimum.



The **learning rate (η)** is a tuning parameter in an optimization algorithm that determines the step size at every iteration while moving toward a minimum of a loss function. The learning rate can determine whether a model delivers optimal performance or fails to learn during the training process.

Gradient Descent Algorithm

Step-1: Initialize Parameters, Start with random values for the parameters (β_i), the intercept (β_o), and the learning rate (η).

Simple Regression: Includes β_0 and β_1.

Multiple Regression: Includes $\beta_0, \beta_1 \ldots\ldots \beta_n$

Step-2: Compute Gradient, Calculate the partial derivative of the cost function J for each parameter:

$$\frac{\partial J}{\partial \beta_i} \quad \& \quad \frac{\partial J}{\partial \beta_0}$$

Step-3: Update Parameters, Adjust the parameters using the learning rate (η) in the opposite direction of the gradient:

$$\beta_i = \beta_i - \eta \frac{\partial J}{\partial \beta_i} \quad \& \quad \beta_0 = \beta_0 - \eta \frac{\partial J}{\partial \beta_0}$$

Step-4: Repeat, steps 2 and 3 until the model converges.

An Example

Input: House sizes (X) in square feet

Output: House prices (y) in thousands of dollars.

Model: Linear regression model $y = \beta_1 X + \beta_0$, where: β_1 is weight (slope) and β_0 is bias (intercept)

Goal: Use Gradient Descent to find the optimal values of β_1 and β_0 that minimize the Mean Squared Error (MSE) cost function.

Note: For MSE the cost function for a linear regression model happens to be a convex function, which means that if you pick any two points on the curve, the line segment joining them is never below the curve. This implies that there are no local minima, just one global minimum. It is also a continuous function with a slope that never changes abruptly. These two facts have a great consequence: gradient descent is guaranteed to approach arbitrarily close to the global minimum (if you wait long enough and if the learning rate is not too high). While the cost function has the shape of a bowl, it can be an elongated bowl if the features have very different scales.

The MSE is

$$J(\beta_1, \beta_0) = \frac{1}{n} \sum_{i=1}^{n} (y_i - (\beta_1 x_i + \beta_0))^2$$

Calculate how much the cost function changes with respect to β_1

$$\frac{\partial J}{\partial \beta_1} = -\frac{2}{n} \sum_{i=1}^{n} x_i (y_i - (\beta_1 x_i + \beta_0))$$

$$\frac{\partial J}{\partial \beta_1} = -\frac{2}{n} \sum x_i (e_i)$$

Calculate how much the cost function changes with respect to β_0

$$\frac{\partial J}{\partial \beta_0} = -\frac{2}{n}\sum_{i=1}^{n}(y_i - (\beta_1 x_i + \beta_0))$$

$$\frac{\partial J}{\partial \beta_0} = -\frac{2}{n}\sum(e_i)$$

Here is the dataset

House size (x)	House Price (y)
1	2
2	4
3	6
4	8

Solution
Step-1: Let us initialize β_1 = 0 and β_0 = 0 and η = 0.1
Step-2 & 3:
Iteration-01

Compute predicted output using formula $y = \beta_1 x + \beta_0$

y_1 = 0 x 1 + 0 = 0 y_2 = 0 x 2 + 0 = 0
y_3 = 0 x 3 + 0 = 0 y_4 = 0 x 4 + 0 = 0

e_1 = 2 - 0 = 2 e_2 = 4 - 0 = 4
e_3 = 6 - 0 = 6 e_4 = 8 - 0 = 8

Compute the Gradient

For Slope (β_1)

$$\frac{\partial J}{\partial \beta_1} = -\frac{2}{4}[(1 \times 2) + (2 \times 4) + (3 \times 6) + (4 \times 8)]$$

$$\frac{\partial J}{\partial \beta_1} = -0.5[2 + 8 + 18 + 32] = -0.5[60] = \mathbf{-30}$$

For Intercept (β_0)

$$\frac{\partial J}{\partial \beta_0} = -\frac{2}{4}[2 + 4 + 6 + 8]$$

$$\frac{\partial J}{\partial \beta_0} = -0.5[20] = \mathbf{-10}$$

Update weight and bias

$$\beta_1 = 0 - (0.1 \times -30) = \mathbf{3.0}$$
$$\beta_0 = 0 - (0.1 \times -10) = \mathbf{1.0}$$

Iteration-02

Now: $\hat{y}$ = 1.0 + 3.0x

y_1 = 3.0 x 1 + 1.0 = 4 y_2 = 3.0 x 2 +1.0 = 7.0
y_3 = 3.0 x 3 + 1.0 = 10 y_4 = 3.0 x 4 + 1.0 = 13.0

e_1 = 2 - 4 = -2 e_2 = 4 - 7 = -3

e_3 = 6 - 10 = -4 e_4 = 8 - 13 = -5

Compute the Gradient

For Slope (β_1)

$$\frac{\partial J}{\partial \beta_1} = -0.5[(1 \times -2.0) + (2 \times -3.0) + (3 \times -4.0) + (4 \times -5.0)]$$

$$\frac{\partial J}{\partial \beta_1} = -0.5[-2.0 - 6.0 - 12.0 - 20.0] = -0.5[-40.0] = \mathbf{20.0}$$

For Slope (β_o)

$$\frac{\partial J}{\partial \beta_0} = -0.5[-2.0 - 3.0 - 4.0 - 5.0]$$

$$\frac{\partial J}{\partial \beta_0} = -0.5[-14.0] = \mathbf{7.0}$$

Update weight and bias

$$\beta_1 = 3.0 - (0.1 \times 20.0) = 3.0 - 2.0 = \mathbf{1.0}$$

$$\beta_0 = 1.0 - (0.1 \times 7.0) = 1.0 - 0.7 = \mathbf{0.3}$$

Iteration-03

Now: $\hat{y}$ = 0.3+1.0x

y_1 = 1.0 x 1 + 0.3 = 1.3 y_2 = 1.0 x 2 + 0.3 = 2.3

y_3 = 1.0 x 3 + 0.3 = 3.3 y_4 = 1.0 x 4 + 0.3 = 4.3

e_1 = 2 - 1.3 = 0.7 e_2 = 4 - 2.3 = 1.7

e_3 = 6 - 3.3 = 2.7 e_4 = 8 - 4.3 = 3.7

Compute the Gradient

For Slope (β_1)

$$\frac{\partial J}{\partial \beta_1} = -0.5[(1 \times 0.7) + (2 \times 1.7) + (3 \times 2.7) + (4 \times 3.7)]$$

$$\frac{\partial J}{\partial \beta_1} = -0.5[0.7 + 3.4 + 8.1 + 14.8] = -0.5[27.0] = \mathbf{-13.5}$$

For Slope (β_o)

$$\frac{\partial J}{\partial \beta_0} = -0.5[0.7 + 1.7 + 2.7 + 3.7]$$

$$\frac{\partial J}{\partial \beta_0} = -0.5[8.8] = \mathbf{-4.4}$$

Update weight and bias

$$\beta_1 = 1.0 - (0.1 \times -13.5) = 1.0 + 1.35 = \mathbf{2.35}$$

$$\beta_0 = 0.3 - (0.1 \times -4.4) = 0.3 + 0.44 = \mathbf{0.74}$$

Continue iterating until the changes in β_1 and β_0 become very small (e.g. <0.001). After several iterations, β_1 and β_0 will converge to their optimal values. For this example, they will approach β_1 = 2, β_0 = 0 & the final model will be y = 2x

A coding Example

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

#generate 4 data observation using y = 2x
x = np.array([1,2,3,4])
y = np.array([2,4,6,8])

#find the value of slope and intercept using OLS (use numpy)
beta_closed = np.polyfit(x, y, deg = 1)

#print the value of slope and intercept (up to 4 decimal places)
print(f"slope = {beta_closed[0]:.4f}")
print(f"intercept = {beta_closed[1]:.4f}")
```

```
slope = 2.0000
intercept = 0.0000
```

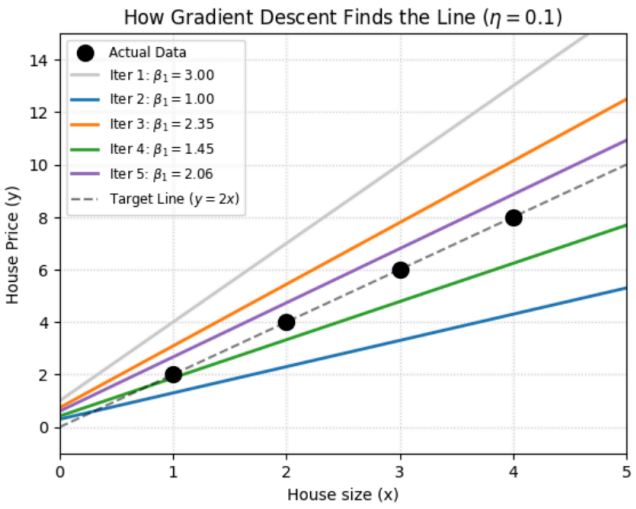
```
#Initialize slope and bias with zero
slope, intercept = 0, 0

#Initialize the learning rate and epochs
#eta, epochs = 0.1, 325 #learning rate High
#eta, epochs = 0.001, 325 #learning rate Low
eta, epochs = 0.04, 325 #learning rate Low

#History of variable to maintain graph
history = []

#print initial value of slope and intercept
print(f"slp = {slope:.4f}, intrcpt = {intercept:.4f}")

#Iterate over the number of epochs
for index in range(1, epochs + 1):
    #compute the predicted output
    y_hat = slope * x + intercept

    #find the amount of Error
    error = y - y_hat

    #compute gradient for slope
    grad_slope = -2/len(x) * np.sum(x * error)

    #compute gradient for intercept
    grad_intercept = -2/len(x) * np.sum(error)

    #update the slope
    slope = slope - eta * grad_slope
```

```
    #update the intercept
    intercept = intercept - eta * grad_intercept

    #print for first 5 iterations and for every 25th iteration
    if index <= 5 or index % 25 == 0:
        print(f"Itr = {index}, slp = {slope:.4f}, intrcpt =
{intercept:.4f}")

         #Maintain slope and intercept for first 5 iterations and
for the last iteration (for graph)
        if index <= 5:
            history.append((index, slope, intercept))
```

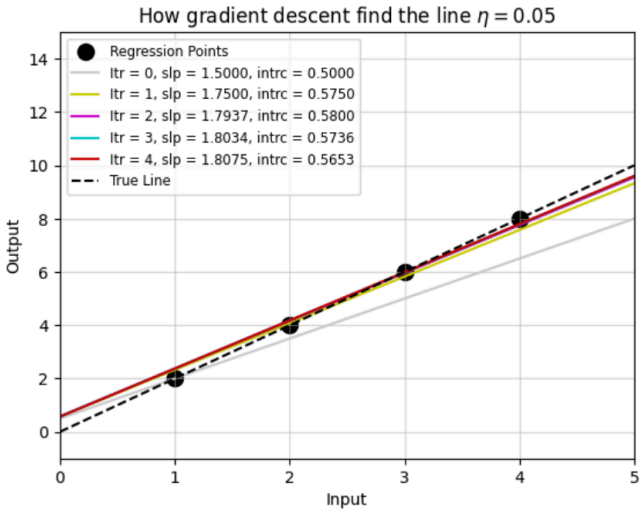
```
slp = 0.0000, intrcpt = 0.0000        Itr = 1, slp = 3.0000, intrcpt = 1.0000
Itr = 2, slp = 1.0000, intrcpt = 0.3000    Itr = 3, slp = 2.3500, intrcpt = 0.7400
Itr = 4, slp = 1.4550, intrcpt = 0.4170    Itr = 5, slp = 2.0640, intrcpt = 0.6061
Itr = 25, slp = 1.9031, intrcpt = 0.2852   Itr = 50, slp = 1.9546, intrcpt = 0.1334
Itr = 75, slp = 1.9788, intrcpt = 0.0624   Itr = 100, slp = 1.9901, intrcpt = 0.0292
Itr = 125, slp = 1.9954, intrcpt = 0.0136  Itr = 150, slp = 1.9978, intrcpt = 0.0064
Itr = 175, slp = 1.9990, intrcpt = 0.0030  Itr = 200, slp = 1.9995, intrcpt = 0.0014
Itr = 225, slp = 1.9998, intrcpt = 0.0007  Itr = 250, slp = 1.9999, intrcpt = 0.0003
Itr = 275, slp = 2.0000, intrcpt = 0.0001  Itr = 300, slp = 2.0000, intrcpt = 0.0001
Itr = 325, slp = 2.0000, intrcpt = 0.0000
```

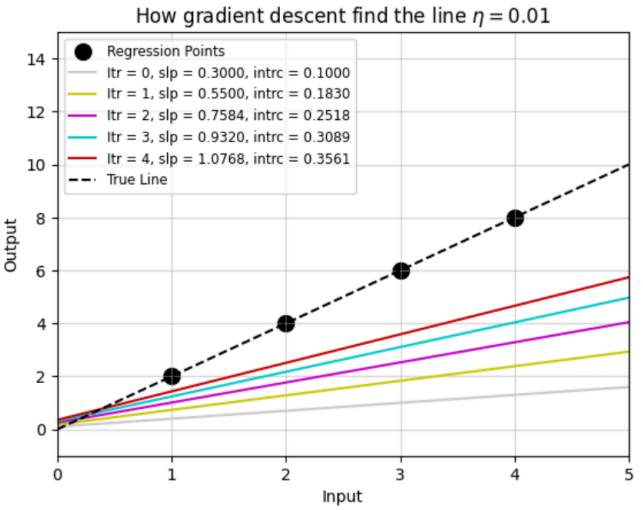
```
#Draw the scatter plot for regression points
plt.scatter(x, y, color = 'black', label = f"Regression Points", s
= 100)
# Generate predictions for each iteration across a range of x
values
x_pred = np.linspace(0, 5, 100)

#We will use 5 iterations so 5 color for every iteration
colors = ['#cccccc', '#cccc00', '#cc00cc', '#00cccc', '#cc0000']

#iterate for 5 times
for i in range(len(history)):
    #find the output according to slope and intercept
    y_pred = history[i][1] * x_pred + history[i][2]

    #draw the same on as line plot
    plt.plot(x_pred, y_pred, color = colors[i], label = f'Itr =
{i}, slp = {history[i][1]:.4f}, intrc = {history[i][2]:.4f}')

#Add target line (y = 2x) for comparison
plt.plot(x_pred, 2 * x_pred, color = 'black', linestyle = '--',
label = f"True Line")

#set the x, y label and title
plt.xlabel("Input")
```

```
plt.ylabel("Output")

#set the legend, use loc = 'upper left' and fontsize = 'small'
plt.legend(loc = 'upper left', fontsize = 'small')

#set the x-axis range from 0 to 5
plt.xlim(0, 5)

#set the y-axis range from -1 to 15
plt.ylim(-1, 15)

#turn on the grid
plt.grid(True, alpha = 0.5)

#show the graph
plt.show()
```



Let us try two different learning rates that are 0.05 and 0.01, here is the graph

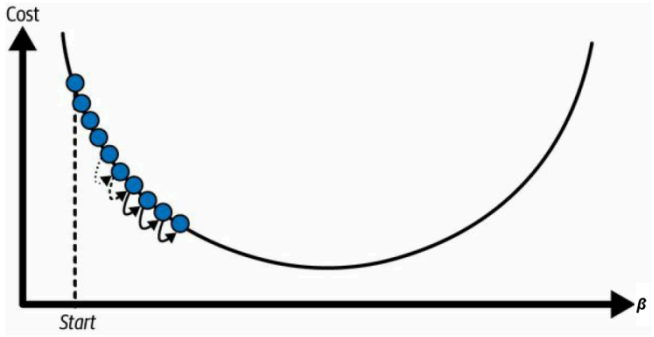


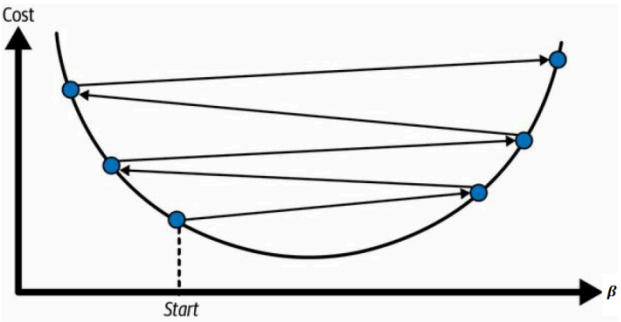


The learning rate determines how quickly or slowly the model parameters (e.g., weights β_1 and bias β_0) are updated during training. It scales the gradient of the cost function:

Learning Rate (η)	**Behavior**	**Pros**	**Cons**
Too Low	Small steps toward the minimum.	1. Stable convergence. 2. Less likely to overshoot the minimum.	1. Slow convergence (requires many iterations). 2. May get stuck in local minima or saddle points.
Optimal	Balanced steps that converge efficiently to the minimum.	1. Fast and stable convergence. 2. Efficient use of computational resources.	1. Requires tuning to find the right value

Too High	Large steps that may overshoot the minimum.	1. Faster initial progress.	1. Oscillations around the minimum. 2. Risk of divergence (moving away from the minimum)





Learning rate too Low *Learning rate too high*

Local minima, saddle point and plateau

- A local minimum is a point in the cost function where the error is lower than all nearby points but higher than the global minimum (the best possible solution). It happens when the cost function in complex models (e.g., neural networks) is often non-convex, meaning it has many "hills" and "valleys." If the model parameters (weights and biases) land in a local minimum during training, Gradient Descent or other optimization algorithms may get stuck there because the gradient is zero (or close to zero). An example of it is rolling a ball down a hilly terrain. If the ball lands in a small valley (local minimum), it won't roll further even if there's a deeper valley (global minimum) nearby.
- The saddle point is a point in the cost function where the gradient is zero, but it is neither a minimum nor a maximum (it's a flat region). It happens in high-dimensional spaces (common in neural networks), saddle points are more prevalent than local minima. The model may get stuck at a saddle point because the gradient is zero, and the optimization algorithm stops making progress. However, the second derivative (Hessian) has both positive and negative eigenvalues.
- A plateau is a region where the surface is almost perfectly flat (or has a very shallow slope) for a significant distance. Its shape is like a flat tabletop or a very gentle hill. The derivatives are not necessarily zero, but they are extremely small. This means the gradient is close to vanishing over a wide area. For a model Plateaus are often more frustrating than saddle points because the "signal" telling the model which way to go is incredibly weak, causing training to slow down to a crawl.

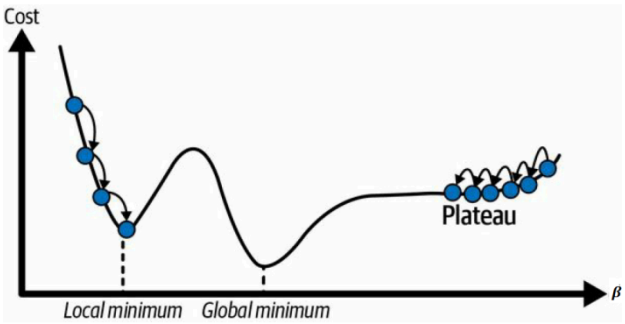
Oscillations & Divergence

- Oscillation occurs when the learning rate is too high, causing the model parameters to bounce around the minimum instead of converging smoothly. It happens when large steps cause the model to overshoot the minimum, and the next update overshoots in the

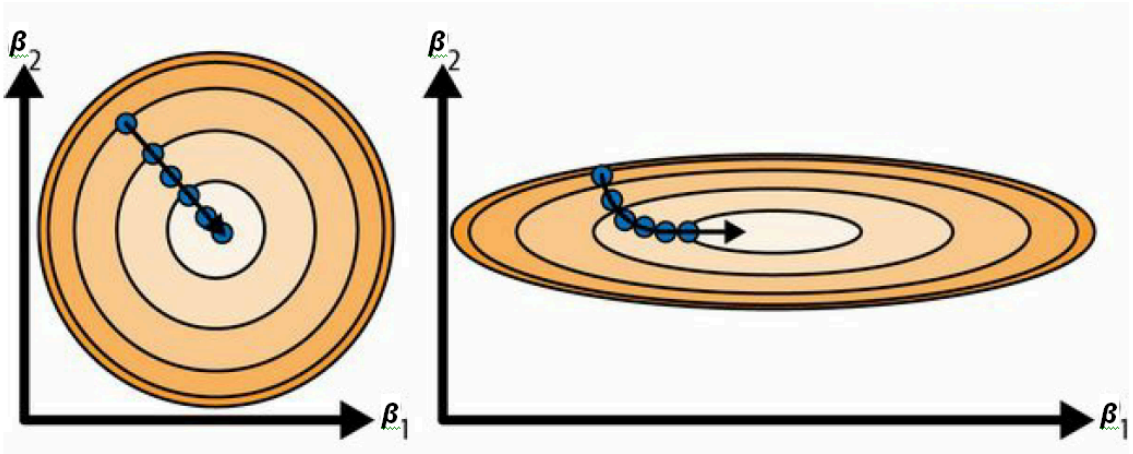
opposite direction, creating a cycle. An example of it is a ball rolling down a hill. If the ball has too much momentum (high learning rate), it will overshoot the bottom and roll up the other side, then roll back, and so on.

- Divergence occurs when the learning rate is so high that the model parameters move away from the minimum instead of converging toward it. It happens when extremely large steps cause the model to overshoot the minimum by such a large margin that the cost function increases instead of decreasing. An example of it is a ball rolling down a hill with so much momentum that it flies off the hill entirely and never returns.

Note: Not all cost functions look like nice, regular bowls. There may be holes, ridges, plateaus, and all sorts of irregular terrain, making convergence to the minimum difficult. Figure shows the two main challenges with gradient descent. If the random initialization starts the algorithm on the left, then it will converge to a local minimum, which is not as good as the global minimum. If it starts on the right, then it will take a very long time to cross the plateau. And if you stop too early, you will never reach the global minimum.



Now consider this diagram



While the cost function has the shape of a bowl, it can be an elongated bowl if the features have very different scales. Figure above shows gradient descent on a training set where features 1 and 2 have the same scale (on the left), and on a training set where feature 1 has much smaller values than feature 2 (on the right). As you can see, on the left the gradient descent algorithm goes straight toward the minimum, thereby reaching it quickly, whereas on the right it first goes in a direction almost orthogonal to the direction of the global minimum, and it ends with a long march down an almost flat valley. It will eventually reach the minimum, but it will take a long time.

This diagram also illustrates the fact that training a model means searching for a combination of model parameters that minimizes a cost function (over the training set). It is a

search in the model's parameter space. The more parameters a model has, the more dimensions this space has, and the harder the search is: searching for a needle in a 300-dimensional haystack is much trickier than in 3 dimensions. Fortunately, since the cost function is convex in the case of linear regression, the needle is simply at the bottom of the bowl.

To find a good learning rate, you can use grid search (will discuss this).

Next part is, how to set the number of epochs? If it is too low, you will still be far away from the optimal solution when the algorithm stops; but if it is too high, you will waste time while the model parameters do not change anymore. A simple solution is to set a very large number of epochs but to interrupt the algorithm when the gradient vector becomes tiny—that is, when its norm becomes smaller than a tiny number ϵ (called the tolerance)— because this happens when gradient descent has (almost) reached the minimum.

Batch Gradient descent

This is exactly what we did in the gradient descent. It processes the entire dataset to calculate the gradient for one step.

Advantages

- It moves directly towards the minimum (smooth curve).
- It finds the exact mathematical minimum (for convex problems).

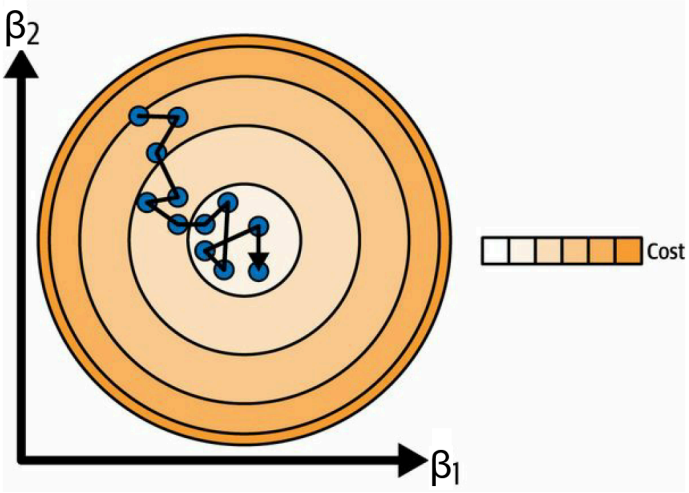
Disadvantages

- Slow: Extremely computationally expensive for large datasets.
- Memory Heavy: Requires loading all data into memory at once hence Its use is limited to small datasets.

Stochastic Gradient descent

It picks a random instance in the training set at every step and computes the gradients based only on that single instance. Obviously, working on a single instance at a time makes the algorithm much faster because it has very little data to manipulate at every iteration.

Due to its stochastic (i.e., random) nature, this algorithm is much less regular than batch gradient descent: instead of gently decreasing until it reaches the minimum, the cost function will bounce up and down, decreasing only on average. Over time it will end up very close to the minimum, but once it gets there it will continue to bounce around, never settling down. Once the algorithm stops, the final parameter values will be good, but not optimal.



Here are some key takeaways-

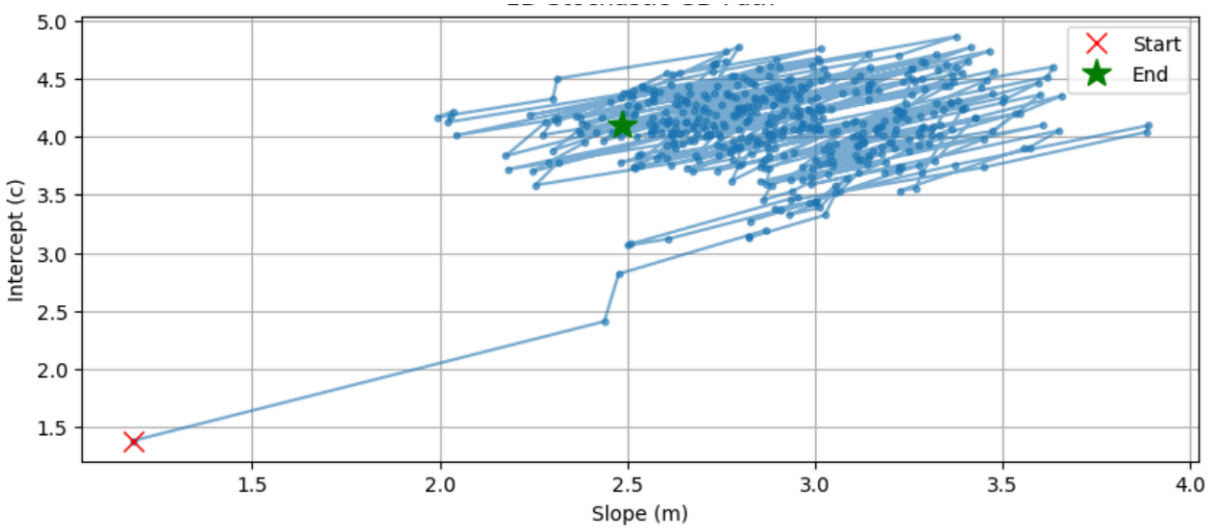
- When the cost function is very irregular (like we have seen a cost function in gradient descent that has local minima, plateau), this can actually help the algorithm jump out of local minima, so stochastic gradient descent has a better chance of finding the global minimum than batch gradient descent does. Therefore, randomness is good to escape from local optima, but bad because it means that the algorithm can never settle at the minimum. One solution to this dilemma is to gradually reduce the learning rate. The steps start out large (which helps make quick progress and escape local minima), then get smaller and smaller, allowing the algorithm to settle at the global minimum.
- It makes it possible to train on huge training sets, since only one instance needs to be in memory at each iteration.

An Example

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

np.random.seed(42)

X = 2 * np.random.rand(100)
y = 4 + 3 * X + np.random.randn(100)

# Initialize parameters as simple floats
slope, intercept = 0.0, 0.0
lr = 0.1
epochs = 5
slope_path_sgd, intercept_path_sgd = [], []

for epoch in range(epochs):
    for i in range(len(X)):
        # Pick ONE random index
        rand_idx = np.random.randint(0, len(X))
```

```
            # 2. EXTRACT SCALARS (Just a single number, not a 2D
array)
        xi = X[rand_idx]
        yi = y[rand_idx]

        # 3. CALCULATE GRADIENTS (Standard Algebra)
        y_pred = slope * xi + intercept

        dslope = -2 * xi * (yi - y_pred)
        dintercept = -2 * (yi - y_pred)

        # Update weights
        slope = slope - lr * dslope
        intercept = intercept - lr * dintercept

         # 4. STORE RESULTS (No [0][0] needed since they are simple
floats)
        slope_path_sgd.append(slope)
        intercept_path_sgd.append(intercept)

# Plotting Function
def plot_descent(slope_history, intercept_history, title):
    plt.figure(figsize=(10, 4))
    plt.title(title)
     plt.plot(slope_history, intercept_history, 'o-', markersize=3,
alpha=0.6)
    plt.xlabel("Slope (m)")
    plt.ylabel("Intercept (c)")
    plt.grid(True)
          plt.plot(slope_history[0],  intercept_history[0],  'rx',
markersize=10, label='Start')
         plt.plot(slope_history[-1],  intercept_history[-1],  'g*',
markersize=15, label='End')
    plt.legend()
    plt.show()

plot_descent(slope_path_sgd, intercept_path_sgd, "1D Stochastic GD
Path")
```

Output



The sklearn.linear_model.SGDRegressor class

The SGDRegressor is a linear regression model that finds the "best fit line" using a technique called Stochastic Gradient Descent. Some of the important parameters are as follow-

1. loss: used to define error function, the default value is 'squared_error'.
2. max_iter: It's the Maximum Attempts or Timeout.
3. tol: the Tolerance. After each pass through the data, the model checks how much the error decreased. If the improvement is smaller than tol, then the model will stop even if it hasn't reached epochs yet.
4. eta0: This is the Initial Learning Rate.
5. random_state: It locks the "randomness." If you use 42 (or any constant number), the "random" sequence of points picked will be exactly the same every time you run the script.

The fit(X, y) Method: It starts with a random slope and intercept. It then runs through your data (for max_iter epochs) and uses the Stochastic Gradient Descent math to minimize the loss. parameters are as follow-

- X: Your features (the inputs). It expects a 2D array-like shape (n_samples, n_features).
- y: The target values (the labels).

Once fit finishes, the model "saves" the best slope in an attribute called coef_ and the best intercept in intercept_.

The predict(X) Method: Once the model is trained (after calling fit), you use predict to use that knowledge on new, unseen data. parameters are as follow-

- X. your input from test data

The partial_fit(X, y) method: It performs one epoch (one pass) of semi-stochastic gradient descent on a given batch of data. Unlike fit, calling this method multiple times does not restart the learning process from scratch; it updates the existing model weights using the new information.

- X: {array-like, sparse matrix} of shape (n_samples, n_features) The subset of training data (a "chunk" or "mini-batch").
- y: ndarray of shape (n_samples,) The target values (labels) corresponding to the data in X.

fit():

Trains the model on the entire dataset at once.

- Uses all data together
- Replaces previous training (starts fresh)
- Used for normal batch learning

partial_fit():

Trains the model in small batches (incrementally).

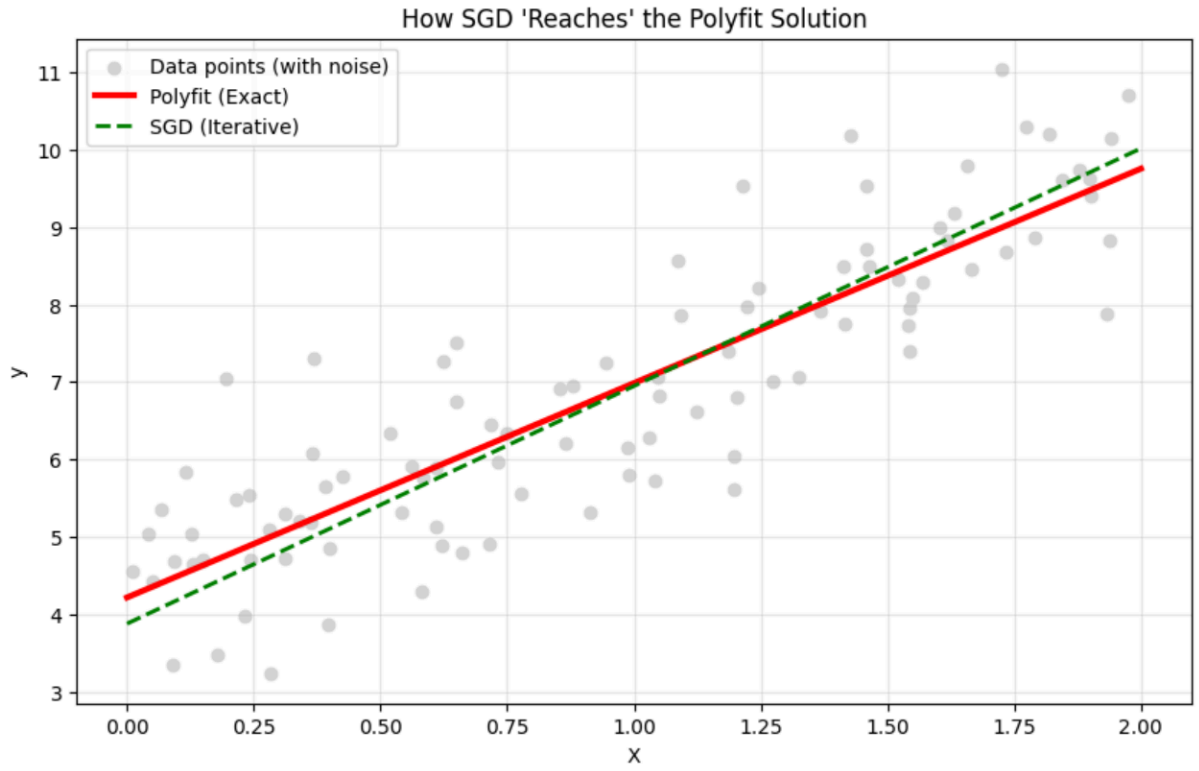
- Learns from small chunks of data
- Does NOT reset previous learning
- Updates existing weights
- Used for online learning

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

# 1. Generate the data based on your requirements
# equation: y = 4 + 3X + noise
np.random.seed(42)
X = 2 * np.random.rand(100, 1)
# X.shape
y = 4 + 3 * X + np.random.randn(100, 1)

# 2. Method 1: Polyfit (Direct Mathematical Solution)
# We flatten X and y to 1D arrays for polyfit
m_poly, c_poly = np.polyfit(X.ravel(), y.ravel(), deg=1)

# 3. Method 2: SGDRegressor (Iterative "Reaching" Solution)
# We set eta0 (learning rate) to 0.01 for a steady approach
sgd = SGDRegressor(max_iter=1000, tol=1e-3, eta0=0.01,
random_state=42)
sgd.fit(X, y.ravel())
m_sgd, c_sgd = sgd.coef_[0], sgd.intercept_[0]

# --- Print Results ---
print(f"Goal Equation: y = 4 + 3x")
print(f"Polyfit Solution: y = {c_poly:.3f} + {m_poly:.3f}x")
print(f"SGD Solution:     y = {c_sgd:.3f} + {m_sgd:.3f}x")
```

```
# --- Visualization ---
plt.figure(figsize=(10, 6))
plt.scatter(X, y, color='lightgray', label='Data points (with
noise)')

# Plotting the lines
x_line = np.array([0, 2])
plt.plot(x_line, c_poly + m_poly * x_line, 'r-', linewidth=3,
label='Polyfit (Exact)')
plt.plot(x_line, c_sgd + m_sgd * x_line, 'g--', linewidth=2,
label='SGD (Iterative)')

plt.xlabel("X")
plt.ylabel("y")
plt.title("How SGD 'Reaches' the Polyfit Solution")
plt.legend()
plt.grid(alpha=0.3)
plt.show()
```

Output

```
Goal Equation: y = 4 + 3x
Polyfit Solution: y = 4.215 + 2.770x
SGD Solution:     y = 3.877 + 3.070x
```



Values of tol = 1e-3 = $1 \times 10^{-3} = 0.001$(default)

- 1e-2 = 0.01

- 1e-4 = 0.0001
- 1e3 = 1000

Min-batch Gradient descent

Mini-batch GD computes the gradients on small random sets of instances called mini-batches.

Key takeaways

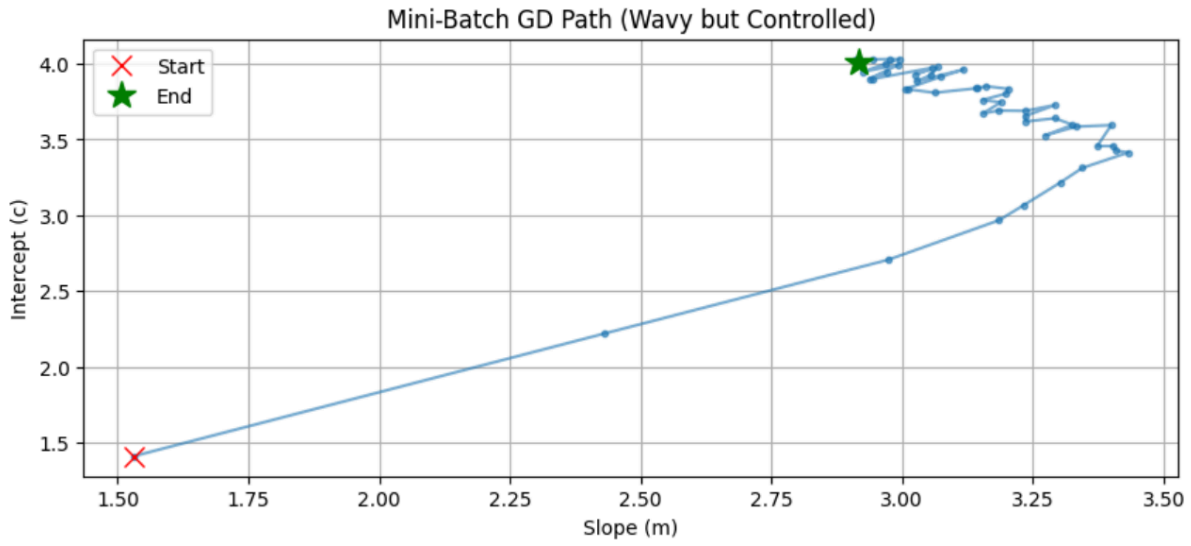
- you can get a performance boost from hardware optimization of matrix operations, especially when using GPUs.
- The algorithm's progress in parameter space is less erratic than with stochastic GD, especially with fairly large mini-batches.
- It may be harder for it to escape from local minima (in the case of problems that suffer from local minima
- Don't forget that batch GD takes a lot of time to take each step, and stochastic GD and mini-batch GD would also reach the minimum if you used a good learning schedule.

An Example

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

np.random.seed(42)

X = 2 * np.random.rand(100)
y = 4 + 3 * X + np.random.randn(100)

slope, intercept = 0.0, 0.0
lr = 0.1
batch_size = 20
epochs = 10
slope_path_mini, intercept_path_mini = [], []

for epoch in range(epochs):
  # Shuffle data first
  indices = np.random.permutation(len(X))

  X_shuffled = X[indices]
  y_shuffled = y[indices]

  for i in range(0, len(X), batch_size):
    # Slice a BATCH of data
    xi = X_shuffled[i:i+batch_size]
    yi = y_shuffled[i:i+batch_size]

    # Calculate gradient for the BATCH
    y_pred = slope * xi + c
```

```
    dslope = -2/batch_size * np.sum(xi * (yi - y_pred))
    dintercept = -2/batch_size * np.sum(yi - y_pred)

    slope = slope - lr * dslope
    intercept = intercept - lr * dintercept

    slope_path_mini.append(m)
    intercept_path_mini.append(c)

plot_descent(m_path_mini, c_path_mini, "Mini-Batch GD Path (Wavy but Controlled)")
```

Output



Feature	Batch GD	Stochastic GD (SGD)	Mini-Batch GD
Data per Step	All Data (Entire Set)	1 Sample	Batch (e.g., 32 samples)
Speed per Step	Very Slow	Very Fast	Fast
Accuracy	High (Smooth convergence)	Low (Noisy convergence)	Good (Balanced)
Memory Usage	High	Low	Moderate
Path to Minimum	Straight Line	Zig-Zag / Random Walk	Wavy Line

Tip: There is almost no difference after training: all these algorithms end up with very similar models and make predictions in exactly the same way.

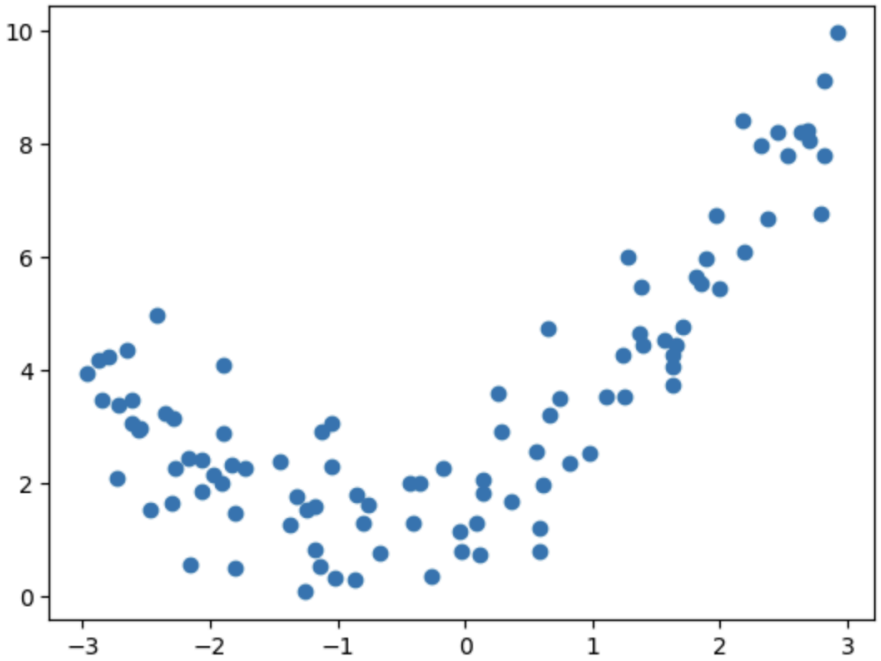
Polynomial Regression

In the field of regression analysis, It is a method used to model the relationship between an independent variable x and a dependent variable y. Instead of a straight line, this relationship is expressed as an n^{th} degree polynomial.

Consider the following code

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

np.random.seed(42)
m = 100
X = 6 * np.random.rand(m, 1) - 3
y = 0.5 * X ** 2 + X + 2 + np.random.randn(m,1)
plt.scatter(X, y)
plt.show()
```

Output



Question: can you fit a straight line that fits data points shown in this figure, answer: No, because this is a binomial equation.

Case Study: Chemical Synthesis Yield vs. Temperature

Imagine you are modeling the yield of a chemical synthesis based on the temperature at which the reaction occurs. In many laboratory settings, the relationship isn't a simple 1:1 ratio. You might encounter two specific scenarios where a linear model fails:

- Non-Uniform Improvement: The yield might increase as you turn up the heat, but the *amount* it improves changes for every degree added (e.g., a "diminishing returns" effect).
- Directional Shifts: You might observe that the yield decreases as temperature rises within a specific range, only to begin increasing again once a different temperature threshold is crossed.

Because these patterns involve a "bend" or a change in direction, a straight line (y=mx+c) is insufficient. To accurately map this curvature, we propose a quadratic model (a degree 2 polynomial) of the form:

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \epsilon$$

Where:

- y is the predicted yield.
- x is the temperature.
- β_2 (the quadratic term) is what allows the model to curve, capturing the "peak" or "valley" in the reaction's efficiency.

In general, we can model the expected value of *y* as an n^{th} degree polynomial, yielding the general polynomial regression model

$$y = \beta_0 + \beta_1 x + \beta_2 x^2 + \beta_3 x^3 + \cdots + \beta_n x^n + \varepsilon.$$

Here, y represents the dependent variable, x denotes the independent variable, and β_0, β_1, β_2, …, β_n are the coefficients to be estimated. The model's complexity is determined by the polynomial's degree, with higher degrees allowing for more flexible curve fitting. The ϵ is noise.

For m data points, this becomes a system of equations: For

$$\begin{cases} y_1 = \beta_0 + \beta_1 x_1 + \beta_2 x_1^2 + \ldots + \beta_n x_1^n \\ y_2 = \beta_0 + \beta_1 x_2 + \beta_2 x_2^2 + \ldots + \beta_n x_2^n \\ \vdots \\ y_m = \beta_0 + \beta_1 x_m + \beta_2 x_m^2 + \ldots + \beta_n x_m^n \end{cases}$$

We can represent this system as:

$$\mathrm{y} = \mathbf{X}\boldsymbol{\beta} + \epsilon$$

Where

- $\mathbf{y}$ (Response Vector - m × 1):

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_m \end{bmatrix}$$

- $\mathbf{X}$ (Design Matrix - m × (n + 1))

$$\mathbf{X} = \begin{bmatrix} 1 & x_1 & x_1^2 & \ldots & x_1^n \\ 1 & x_2 & x_2^2 & \ldots & x_2^n \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \ldots & x_m^n \end{bmatrix}$$

- $\boldsymbol{\beta}$ (Coefficient Vector - (n + 1) × 1):

$$\boldsymbol{\beta} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \vdots \\ \beta_n \end{bmatrix}$$

- ϵ (Error Vector - m × 1)

To solve for the coefficients β using the Ordinary Least Squares (OLS) method, you use the Normal Equation:

$$\hat{\boldsymbol{\beta}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

The sklearn.prerocessing.PolynomialFeature class

It is a preprocessing tool used to generate new features by creating all polynomial combinations of your input data. This class helps is

1. Feature Expansion: It transforms your existing features into a higher-dimensional space.
2. Mathematical Transformation: For an input with two features [a,b], a degree-2 transformation produces [1,a,b,a^2,ab,b^2].
3. Interaction Discovery: It captures interactions between different variables (e.g., x1 X x2) that a standard linear model might miss.

The constructor PolynomialFeatures() accepts the following parameters to control how the features are generated:

A. degree (int or tuple, default=2): Specifies the maximal degree of the polynomial features. If a tuple (min_degree,max_degree) is passed, it generates features within that specific range.
B. interaction_only (bool, default=False): If set to True, the class will only produce interaction terms (products of distinct features like a×b) and will exclude terms where a feature is multiplied by itself (like a^2 or b^2).
C. include_bias (bool, default=True): If True, the output will include a "bias" column where all polynomial powers are zero (a column of ones), which serves as the intercept term for a linear model.
D. order ({'C', 'F'}, default='C'): Determines the memory layout of the output array. 'F' (Fortran) order is generally faster to compute but may slow down certain subsequent estimators.

The fit_transform() Method

This is a convenience method that performs both fitting and transforming in a single step. It fits the transformer to the input data X to determine the output feature names and counts. It then immediately generates the new feature matrix consisting of the polynomial and interaction terms. It returns a new array (or sparse matrix) containing the transformed features. Using fit_transform() is often more efficient than calling fit() and transform() separately on the same data.

An Example

```
import numpy as np
from sklearn.preprocessing import PolynomialFeatures

X = np.array([[0, 1],
              [2, 3],
              [4, 5]])
poly = PolynomialFeatures(degree=2)
X_poly = poly.fit_transform(X)
print(X_poly)
```

Output

Bias (1)	a	b	a^2	ab	b^2
1.0	0.0	1.0	0.0	0.0	1.0
1.0	2.0	3.0	4.0	6.0	9.0
1.0	4.0	5.0	16.0	20.0	25.0

Explanation

- Bias Column (1): Because include_bias=True by default, the first column is always 1.0. This acts as the intercept (β_0) in your regression equation.
- Original Features (a,b): The next two columns are simply your original data unchanged.
- Squared Terms (a^2,b^2): These columns are the original values multiplied by themselves (e.g., 2^2=4 and 3^2=9).
- Interaction Term (ab): This is the product of the two features (2 × 3 = 6). This allows the model to learn how the two variables affect the outcome *together*.

In this example, we are going to use dataset of advertising (source: "An Introduction to Statistical Learning"), Take a look at the dataset available at link

https://raw.githubusercontent.com/selva86/datasets/master/Advertising.csv

The CSV file typically consists of the following columns:

1. Unnamed: 0: An index column ranging from 1 to 200.
2. TV: The budget spent on TV advertising (in thousands of dollars).
3. Radio: The budget spent on Radio advertising.
4. Newspaper: The budget spent on Newspaper advertising.
5. Sales (Target): The sales of the product (in thousands of units). This is the dependent variable you are trying to predict.

An Example

Load the data

```
import pandas as pd
```

```
import matplotlib.pyplot as plt
import numpy as np
data =
pd.read_csv("https://raw.githubusercontent.com/selva86/datasets/master/Advertising.csv")
data.head()
```

Basic Inspection

```
data.info()

data.describe()
```

Scatter plot

```
#Scatter plot: TV Spends vs Sales
plt.scatter(data['TV'], data['sales'])
plt.xlabel("TV Advertising Spend")
plt.ylabel("Sales")
plt.title("TV Spend vs Sales")
plt.show()
```

Train-test Split

```
from sklearn.model_selection import train_test_split

X = data[['TV']]
y = data['sales']

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

Linear Regression

Why Linear first?

Every ML workflow starts with the simplest reasonable model.

```
#Train on training data only

from sklearn.linear_model import LinearRegression
lin_model = LinearRegression()
lin_model.fit(X_train,y_train)
```

Evaluate on Test data

```
from sklearn.metrics import r2_score, root_mean_squared_error

y_test_pred_lin = lin_model.predict(X_test)

print("Linear Test R²:", r2_score(y_test, y_test_pred_lin))
print("Linear Test RMSE:", root_mean_squared_error(y_test,
y_test_pred_lin))
```

Linear fit Visualisation

```
plt.scatter(X_train, y_train, label="Train")
plt.scatter(X_test, y_test, label="Test")

X_grid = X.sort_values(by='TV')
plt.plot(X_grid, lin_model.predict(X_grid), color='red', label='Linear
Fit')

plt.legend()
plt.show()
```

Polynomial Model(Degree 2)

```
from sklearn.preprocessing import PolynomialFeatures
poly2 = PolynomialFeatures(degree=2, include_bias=False)

X_train_poly = poly2.fit_transform(X_train)
X_test_poly = poly2.transform(X_test)

poly_model = LinearRegression()
poly_model.fit(X_train_poly, y_train)
```

Final Evaluation on test data

```
y_test_pred_poly = poly_model.predict(X_test_poly)

print("Polynomial Test R²:", r2_score(y_test, y_test_pred_poly))
print("Polynomial Test RMSE:", root_mean_squared_error(y_test,
y_test_pred_poly))
```

Polynomial fit Visualisation

```
X_grid = np.linspace(X.min(), X.max(), 100).reshape(-1,1)
X_grid_poly = poly2.transform(X_grid)

plt.scatter(X_train, y_train, label="Train")
plt.scatter(X_test, y_test, label="Test")
plt.plot(X_grid, poly_model.predict(X_grid_poly), color='green',
label="Polynomial Fit")

plt.legend()
plt.show()
```

Cross-Validation for Degree Selection

```
from sklearn.model_selection import cross_val_score
import numpy as np

degrees = [1, 2, 3, 4, 5]

for d in degrees:
    poly = PolynomialFeatures(degree=d, include_bias=False)
    X_train_poly = poly.fit_transform(X_train)

    model = LinearRegression()
    cv_r2 = cross_val_score(model, X_train_poly, y_train, cv=5,
scoring='r2').mean()

    print(f"Degree {d} | CV R²: {cv_r2:.4f}")
```

Residual Analysis

```
residuals = y_test - y_test_pred_poly

plt.scatter(y_test_pred_poly, residuals)
plt.axhline(0, color='red')
plt.xlabel("Predicted Sales")
plt.ylabel("Residuals")
plt.title("Residual Plot")
plt.show()
```

If TV-only polynomial regression explains only ~67.66% variance, the model is not sufficient for the business problem.

Why the Polynomial Model Is Still Weak

- 32.34% of the variance in sales is unexplained
- The problem is not model complexity
- The problem is missing information

This is a data issue not a polynomial degree issue

Dataset: Advertising

Target: Sales

Features: TV, Radio, Newspaper

EDA

Typical observations (students should see this):

- TV vs Sales → strong, slightly curved relationship
- Radio vs Sales → strong, almost linear
- Newspaper vs Sales → weak, scattered

Train-test Split

```
from sklearn.model_selection import train_test_split

X = data[['TV', 'radio', 'newspaper']]
y = data['sales']

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

Model1: Multiple Linear Regression(All three variables)

Train Model

```
from sklearn.linear_model import LinearRegression

mlr = LinearRegression()
mlr.fit(X_train, y_train)
```

Evaluate on test data

```
from sklearn.metrics import r2_score, root_mean_squared_error

y_pred = mlr.predict(X_test)
```

```
print("MLR R²:", r2_score(y_test, y_pred))
print("MLR RMSE:", root_mean_squared_error(y_test, y_pred))
```

Observations:
$R^2 \approx 0.8994$

This is already far better than TV-only polynomial (~0.6766).

Model 2: Refined MLR(TV +Radio only)

```
X_train_tr = X_train[['TV', 'radio']]
X_test_tr = X_test[['TV', 'radio']]

mlr_tr = LinearRegression()
mlr_tr.fit(X_train_tr, y_train)

y_pred_tr = mlr_tr.predict(X_test_tr)

print("MLR (TV+Radio) R²:", r2_score(y_test, y_pred_tr))
```

Key Observation

- R^2increases slightly ~0.9005
- Model becomes simpler and more stable

✔ Newspaper can be safely removed.

Should we apply Polynomial Regression now? Yes — but selectively and logically, not blindly.

Business Logic

- TV → diminishing returns → possible non-linear effect
- Radio → near linear

Model 3: Polynomial +MLR(TV + Radio)

```
from sklearn.preprocessing import PolynomialFeatures

poly = PolynomialFeatures(degree=2, include_bias=False)

X_train_poly = poly.fit_transform(X_train_tr)
X_test_poly = poly.transform(X_test_tr)

poly_mlr = LinearRegression()
poly_mlr.fit(X_train_poly, y_train)

y_pred_poly = poly_mlr.predict(X_test_poly)

print("Polynomial MLR R²:", r2_score(y_test, y_pred_poly))
```

Observation:

R2 improves to 0.9884

Interpretation What Polynomial Added

- Captured diminishing returns of TV
- Accuracy improvement

Student task :

Onehot encoder
Label encoder
Min max scaler
Standard scaler