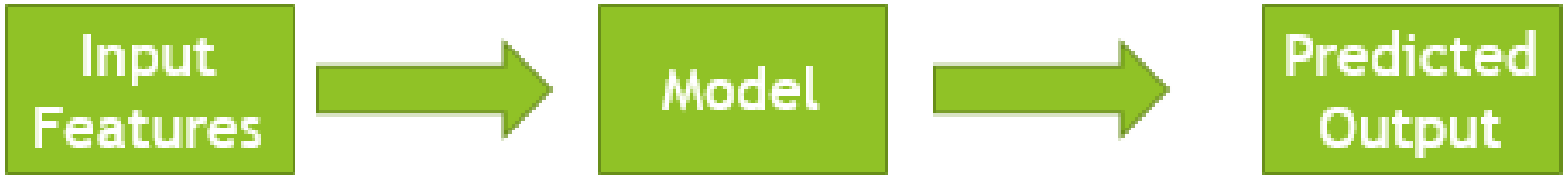
Linear Regression

The Foundation of Predictive Modeling

Introduction

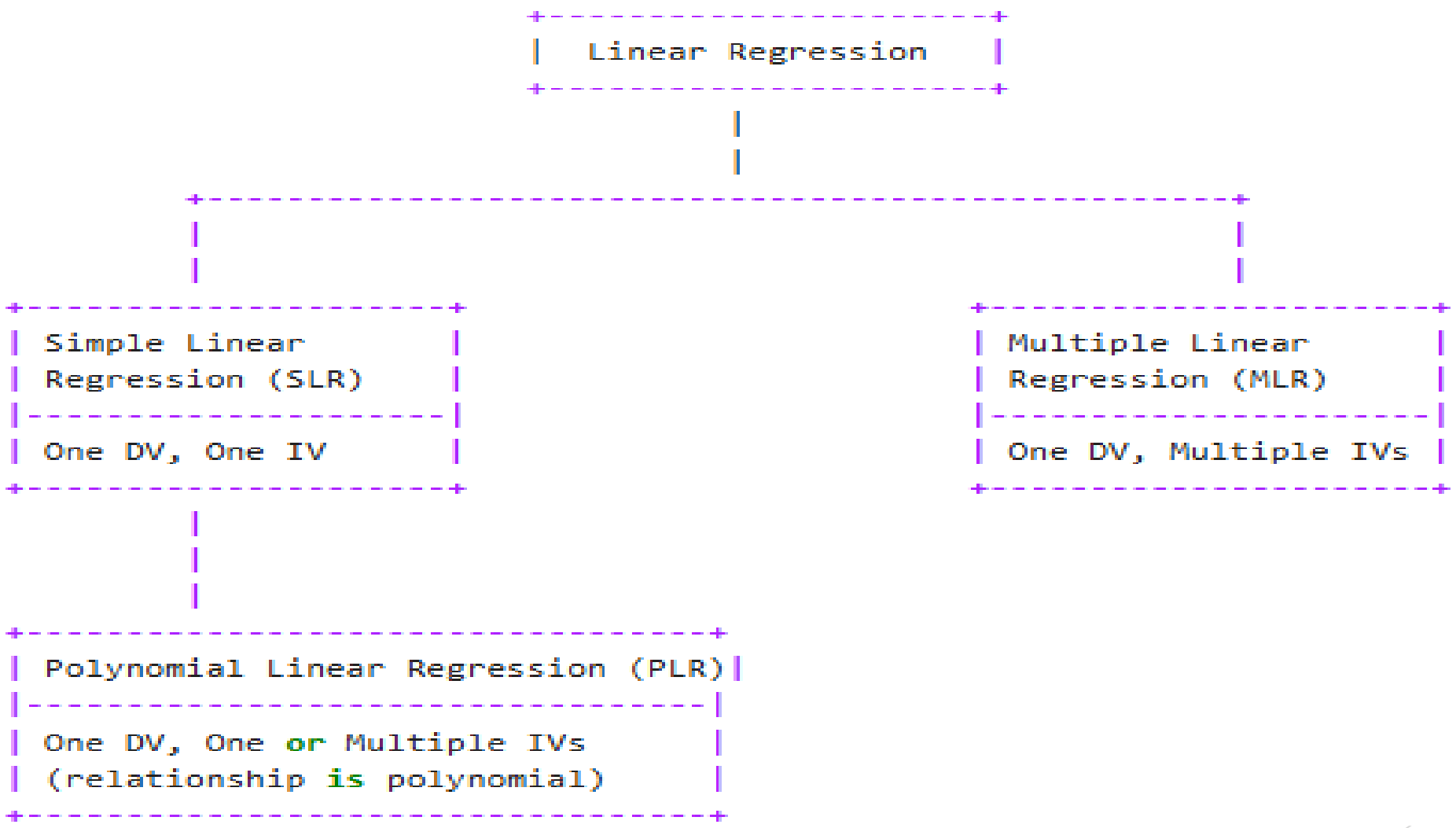
- **Linear Regression** is a supervised learning algorithm used to predict continuous outcomes by modeling the relationship between dependent and independent variables.
- **Common applications:** Predicting house prices, stock trends, sales forecasting, etc.
- **On Larger Note in LR we always looking for a best fit line.**



Types of Linear Regression:

- **Simple Linear Regression**
  - Predicts output based on a single independent variable.
  - Formula: $y = w_0 + w_1 x + \epsilon$

- **Multiple Linear Regression:**
  - Handles multiple independent variables.
  - Formula: $y = w_0 + w_1 x_1 + w_2 x_2 + \ldots + \epsilon$


Linear Regression
Simple Linear Regression (SLR)
One DV, One IV
Multiple Linear Regression (MLR)
One DV, Multiple IVs
Polynomial Linear Regression (PLR)
One DV, One or Multiple IVs
(relationship is polynomial)

Assumptions of Linear Regression:

- **Linearity** ensures the model accurately represents the data trend.
- **Normality of Residuals** validates the p-values and model behavior.
- **Homoscedasticity** maintains consistent variance in residuals.
- **No Autocorrelation** ensures independent errors.
- **No Multicollinearity** keeps predictor relationships manageable.
- **No Endogeneity** prevents biased estimates.
- **Adjusted R-Square** and **p-values** guide model evaluation and predictor significance.
- **Avoiding Overfitting/Underfitting** balances model complexity for generalization.

1. **Linearity in Data**

- **Assumption**: The relationship between the dependent variable (y) and the independent variables (X) should be linear.
- **Why it matters**: Linear regression assumes that the best-fit line (or hyperplane in higher dimensions) can describe the relationship between the variables. If the relationship is non-linear, the model won't capture the true patterns.
- **How to check**:
  - **Scatter plots**: Visualize the relationship between each independent variable and the dependent variable. If the relationship is not linear, consider using polynomial regression or transformations.
  - **Residual plot**: Check residuals (errors) against the predicted values. For linear regression, the residuals should show no pattern (i.e., they should be randomly distributed).

2. **No Heteroscedasticity**

- **Assumption**: The variance of the residuals (errors) should be constant across all levels of the independent variable(s).
- **Why it matters**: Heteroscedasticity (non-constant variance) violates the assumption and can lead to inefficient estimates and invalid statistical tests.
- **How to check**:
    - **Residual plot**: Plot residuals against fitted values. If the spread of residuals increases or decreases systematically as the predicted value increases, this suggests heteroscedasticity.

3. **No or Little Multicollinearity**

- **Assumption**: Independent variables should not be highly correlated with each other.
- **Why it matters**: If independent variables are highly correlated (multicollinearity), it becomes difficult to isolate the effect of each variable, leading to unstable coefficient estimates and inflated standard errors.
- **How to check**:
  - **Correlation matrix**: Look at the correlation coefficients between pairs of independent variables. If they are very high (e.g., > 0.8 or < -0.8), there may be multicollinearity.
  - **Variance Inflation Factor (VIF)**: A high VIF (> 10) indicates problematic multicollinearity. You can remove highly correlated predictors or use techniques like **Principal Component Analysis (PCA)** to reduce multicollinearity.

4. **No Autocorrelation**

- **Assumption**: Residuals should not be correlated with each other. This is especially important for time series data.
- **Why it matters**: If residuals are autocorrelated, it suggests that some information is missing from the model, meaning the model has failed to capture a pattern that should be accounted for.
- **How to check**:
  - **Durbin-Watson test**: This test checks for autocorrelation in residuals. A value close to 2 suggests no autocorrelation.

5. **No Endogeneity**

- **Assumption**: The independent variables should not be correlated with the error term.
- **Why it matters**: Endogeneity occurs when there's a cause-and-effect relationship between the independent variables and the error term, violating the assumption that the error term should be random. This leads to biased and inconsistent estimates of the regression coefficients.

6. **P-value Less Than 5% (for Significance)**

- **Assumption**: The individual coefficients in the regression model should have p-values below a threshold (usually 0.05) to indicate that the variable is statistically significant in explaining the variance in the dependent variable.
- **Why it matters**: A p-value greater than 0.05 indicates that the corresponding variable's coefficient is not significantly different from zero, meaning it does not contribute much to predicting the outcome. This suggests you might want to remove the variable.
- **How to check**:
    - After fitting a linear regression model, check the p-values for each coefficient. If they are higher than 0.05 (the typical threshold), you may consider removing the variable from the model.

7. **R-squared Greater Than 70% (for Model Fit)**

- **Assumption**: This is more of a practical expectation than a strict assumption. R-squared indicates how well the model explains the variance in the dependent variable.
- **Why it matters**: A high R-squared value (greater than 70%) suggests that the model fits the data well and is able to explain a large proportion of the variance. However, **R-squared alone is not enough** to conclude a good model; overfitting can artificially inflate R-squared.
- **How to check**:
  - Look at the R-squared value in your model output. If it's below 70%, it might indicate a weak fit, but you should also check the adjusted R-squared to account for the number of predictors.
  - Keep in mind that a high R-squared value doesn't imply causation or guarantee that the model will perform well on unseen data.

8. **Independence of Observations**

- **Assumption**: Observations should be independent of each other.
- **Why it matters**: If the observations are not independent (for example, in time series data or clustered data), the regression results can be biased and invalid.
- **How to check**: You should consider the structure of your data. In time series or clustered data, you might need to use methods that account for this, such as **time-series models** or **mixed-effects models**.

Additional Considerations:

- **Adjusted R-Square:**
  - Unlike R^2, the adjusted R^2 accounts for the number of predictors in the model and adjusts for the model's complexity.
  - It is used to compare models with different numbers of predictors and can provide a better measure of fit for models with multiple predictors.
- **No Overfitting/Underfitting:**
  - **Overfitting**: Occurs when the model is too complex and captures the noise in the data, leading to poor generalization to new data.
  - **Underfitting**: Occurs when the model is too simple and fails to capture the underlying trend in the data.
  - Regularization techniques, cross-validation, and model selection criteria (like AIC or BIC) can help in finding a balance to avoid overfitting or underfitting.

Journey of Regression from Stats to ML:

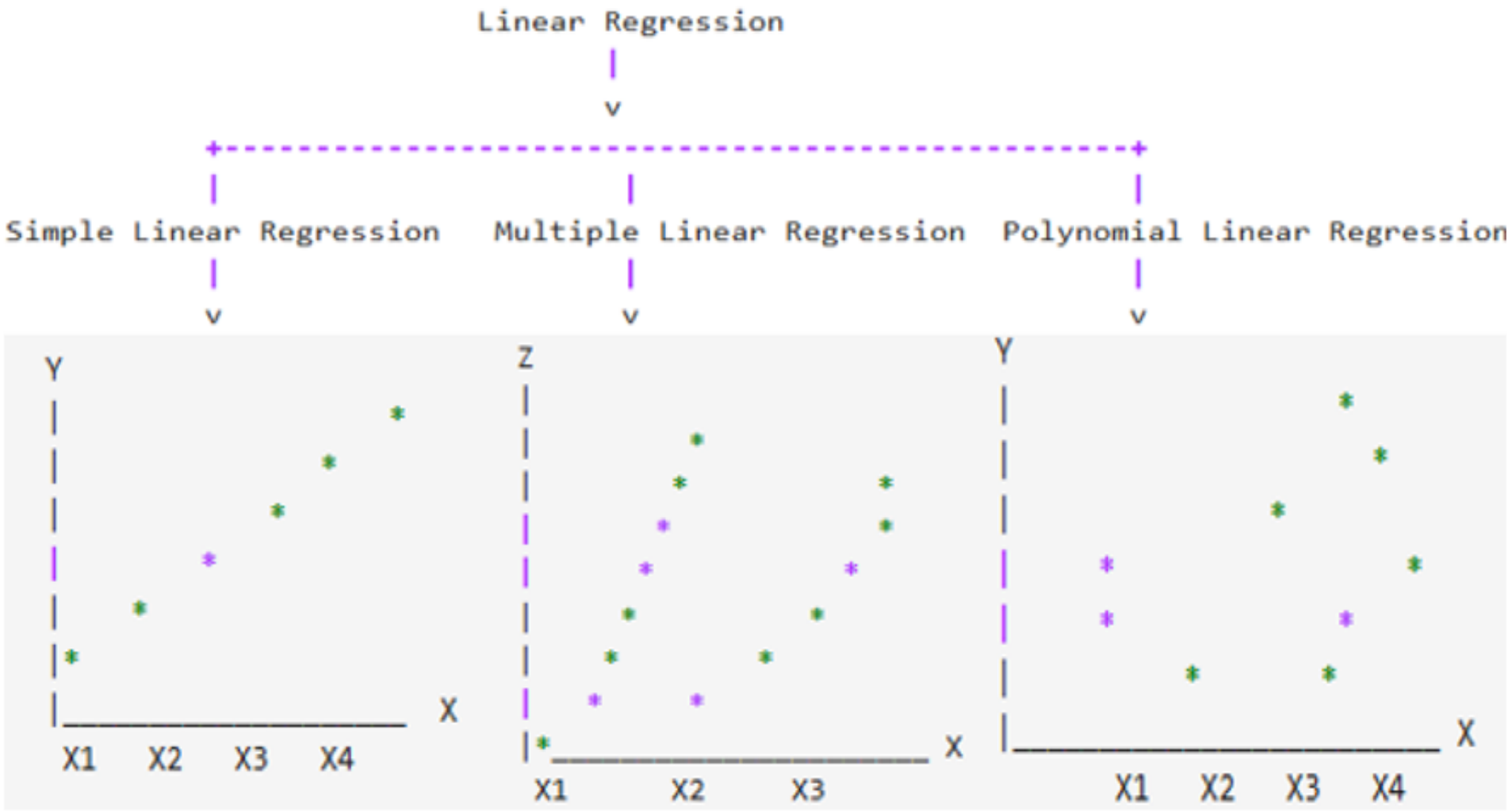
```
   Variance ------------> Co-variance ------------------> Correlation ------------> Regression
|Spread of data|   |Relationship between two variables| |Relationship and Strength|   |Quantify relationships|
```

Regression:

- Utilizes the concepts of variance and correlation to build models that predict the value of a dependent variable based on one or more independent variables.
- Provides a quantitative measure of how much the dependent variable changes with changes in the independent variables.

- **Linear Regression** models the relationship between the **response variable** (also known as the dependent, target, or result variable, denoted as y) and the **regression coefficients** (denoted as βi, wi).

- The relationship is assumed to be linear. This means the output y can be expressed as a linear combination of the input features x and the coefficients.


Linear Regression
Simple Linear Regression
Multiple Linear Regression
Polynomial Linear Regression
Y
X
X1
X2
X3
X4
Z
Y
X
X1
X2
X3
X4

General Form:

- The general form of the linear regression model is given as:

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \cdots + \beta_n x_n$$

- Here, y is the predicted output, β_0 is the intercept (constant term), and $\beta_1, \beta_2, \ldots, \beta_n$ are the coefficients for the input features $x_1, x_2, \ldots, x_n$.

➢**Regression Coefficients:**

✓These coefficients (βi, wi) are the weights assigned to each input feature x.

✓They determine how much each input feature contributes to the output y.

Geometric Intuition:

- **Objective:**
  The primary objective in simple linear regression is to determine the linear relationship between two variables—in this case, weight (f_1) and height (f_2).

- **Finding the Best Fit:**
  Imagine trying to draw a line that passes through or is closest to all the data points on the scatter plot. The line should balance itself such that the distances (vertical distances in this context) between the data points and the line are as small as possible.

- **Error Minimization:**
  The concept of the "best fit" line is quantified by minimizing the sum of squared residuals (errors). Each residual is the vertical distance between the observed height and the height predicted by the line for a given weight.

- **Slope and Intercept:**
  - The slope (w_1) tells us how steep the line is and the direction of the relationship between weight and height. If the slope is positive, height increases with weight; if negative, height decreases with weight.
  - The intercept (w_0) gives the height when the weight is zero. While this might not have a physical meaning (since weight can't be zero), it is mathematically crucial for defining the line.

- **Linear Relationship:**
  Linear regression assumes that the relationship between the variables can be described with a straight line. This assumption simplifies the problem but might not always capture more complex relationships.

Mathematical Formulation:

Given data points (x_i, y_i) where x_i represents weight and y_i represents height, the line of best fit is determined by solving:

$$y = w_1 x + w_0$$

where w_1 and w_0 are determined by minimizing the cost function:

$$\text{Cost} = \sum_{i-1}^{n} (y_i - (w_1 x_i + w_0))^2$$

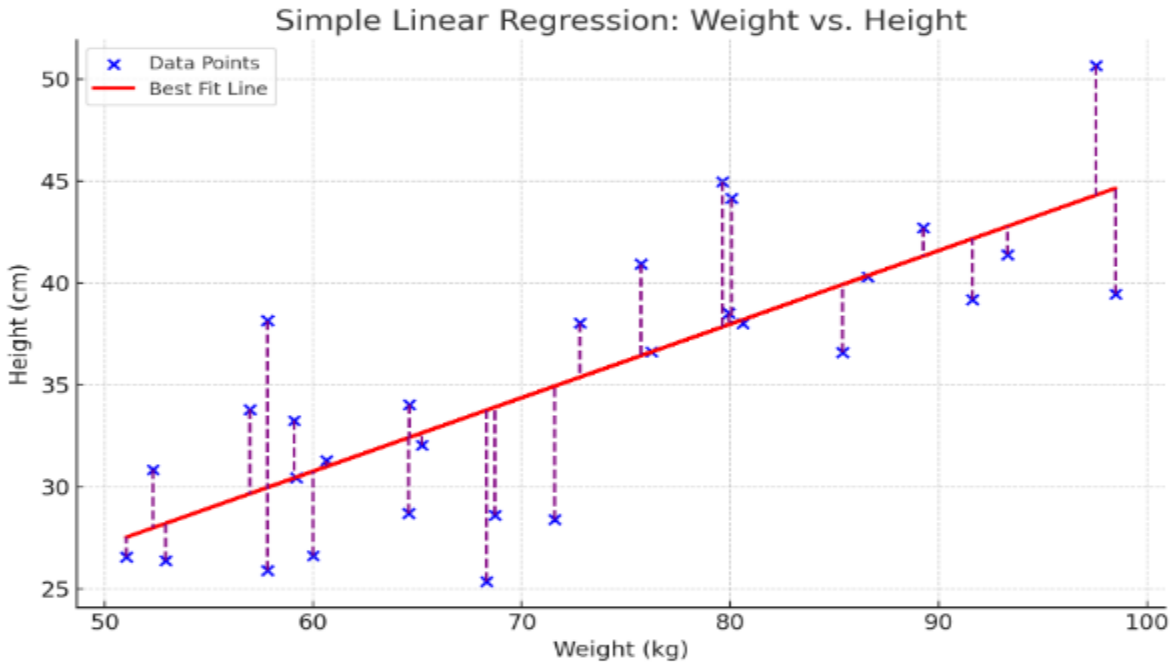
This approach helps find the optimal values for w_1 and w_0 that best describe the relationship between weight and height for the given dataset.

Practical Example:

If we have a dataset of individuals' weights and heights, we can plot these points and find the line that best predicts height from weight. Once we have the regression line, we can use it to estimate the height of someone given their weight.

Graphical Representation:

- **Scatter Plot of Data Points:** We'll plot the given data points, where each point represents a pair of weight and height values.
- **Best Fit Line:** We'll draw the line that best fits the data points according to the linear regression model. This line minimizes the sum of the squared errors between the actual data points and the line.
- **Residuals (Errors):** We'll show the vertical distances (residuals) between the data points and the regression line, highlighting the concept of minimizing these distances.



Model Parameters:

- **Slope (w1):** let it be 0.361 (approximately)
  - This means that for every unit increase in weight, the height is expected to increase by about 0.361 units.
- **Intercept (w0):** let it be 9.11 (approximately)
  - This is the height when the weight is zero. While it might not be meaningful in a practical context, it is essential for defining the regression line.

Geometric Intuition:

- The best fit line attempts to capture the linear relationship between weight and height. The slope indicates the direction and steepness of this relationship.
- Minimizing the residuals ensures that the line is as close as possible to all the data points, providing the best possible predictions.

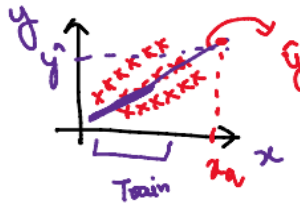
Finding a line to best fit the data points using Ordinary Least Squares (OLS) regression

Best fit line

How to create line?

m, b, x_i

C = intercept → 0



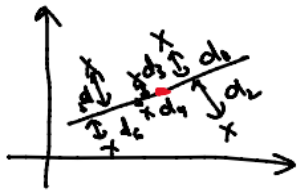
$\varepsilon = y_i - \hat{y}_i$

$\varepsilon = y_i - (mx_i + b)$

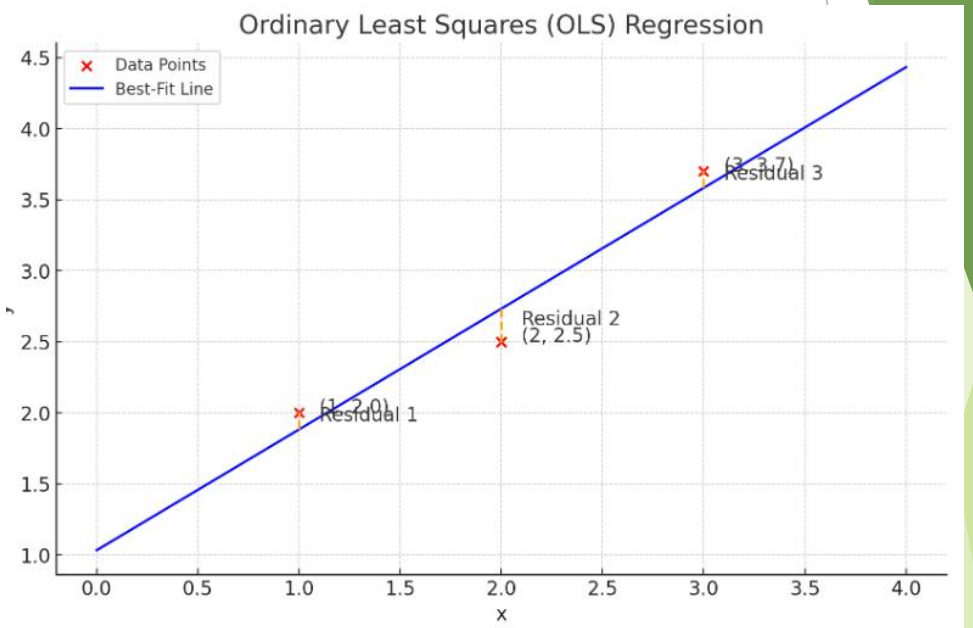
Constant: y_i, x_i

Variables: m, b

$\varepsilon_{m,b} = y_i - (mx_i + b)$



Abs deviation = $d_1 + d_3 + d_5 - d_2 - d_4 - d_6 = 0$



1. **Data Points and Line:**

   - The graph shows three data points (x_1, y_1), (x_2, y_2), and (x_3, y_3).
   - A line is fitted to these points, likely a regression line, represented in blue.

2. **Residuals:**

   - Residuals are the differences between the actual values (y) and the predicted values $(\hat{y})$ on the regression line.
   - The residual for each point is depicted as a vertical line from the point to the regression line.
     - **Residual for y_1**: This is the vertical distance between y_1 (actual) and $\hat{y}_1$ (predicted on the line).
     - **Residual for y_2**: Similarly, the distance between y_2 (actual) and $\hat{y}_2$ (predicted).
     - **Residual for y_3**: The distance between y_3 and $\hat{y}_3$ (in this case, zero because the point lies on the line).

3. **Error Notation:**

   - The error for each point is noted as $y_i - \hat{y}_i$:
     - **Error 1 (e_1)**: $y_1 - \hat{y}_1$, which is positive (+ve) because y_1 is above the line.
     - **Error 2 (e_2)**: $y_2 - \hat{y}_2$, which is negative (-ve) because y_2 is below the line.
     - **Error 3 (e_3)**: $y_3 - \hat{y}_3$, which is zero (0) because y_3 lies exactly on the line.

4. **Squared Errors:**

   - To measure the fit of the line, we square these residuals:
     - $(y_1 - \hat{y}_1)^2$: The squared residual for point 1.
     - $(y_2 - \hat{y}_2)^2$: The squared residual for point 2.
     - $(y_3 - \hat{y}_3)^2$: The squared residual for point 3.
   - Squaring is done to ensure that all errors are positive and to penalize larger deviations more heavily.

5. **Sum of Squared Errors (SSE):**

   - The sum of these squared errors gives the Sum of Squared Errors (SSE):

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

   - SSE is used to assess how well the regression line fits the data. A smaller SSE indicates a better fit.

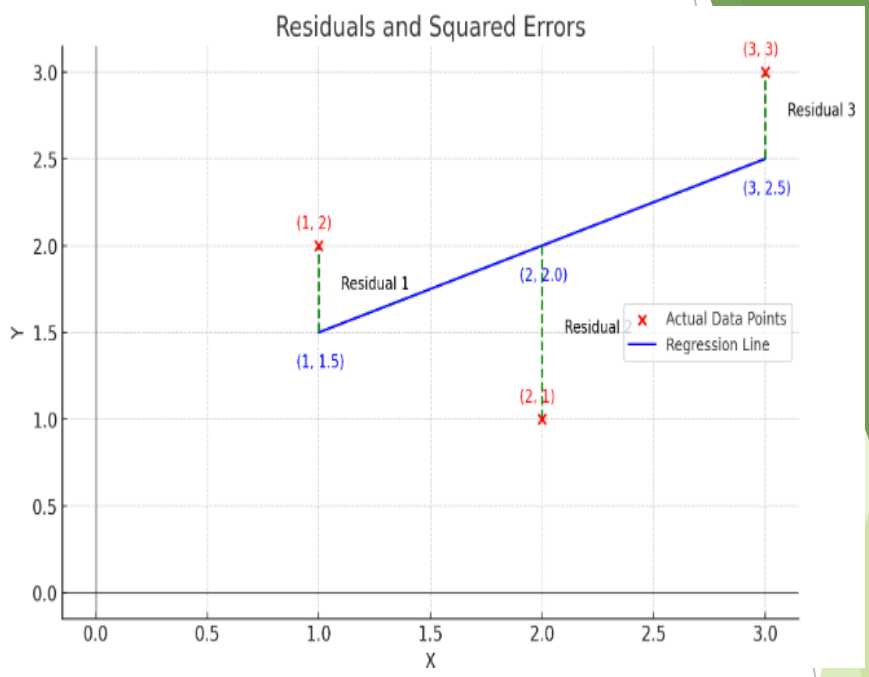
Key Points:

- **Residuals**: The differences between actual and predicted values.
- **Squared Errors**: Residuals squared to ensure positivity and penalize larger deviations.
- **SSE**: A measure of the total deviation of the actual data points from the fitted line.

Explanation of the Graph:

This graph visualizes the concepts of residuals and squared errors in a linear regression context:

1. **Data Points:**
   - The red crosses ($\times$) represent the actual data points: $(x_1, y_1) = (1, 2)$, $(x_2, y_2) = (2, 1)$, and $(x_3, y_3) = (3, 3)$.
   - These are the observed values we are trying to predict using the regression model.
2. **Regression Line:**
   - The blue line represents the regression model, providing predicted values ($\hat{y}$) based on the x values.
   - For simplicity, we assumed a linear fit with predicted points: $(1, 1.5)$, $(2, 2)$, and $(3, 2.5)$.
3. **Residuals:**
   - Green dashed vertical lines illustrate the residuals, which are the differences between the actual data points and their corresponding predicted values on the regression line.
     - **Residual 1**: The distance from $(1, 2)$ to $(1, 1.5)$.
     - **Residual 2**: The distance from $(2, 1)$ to $(2, 2)$.
     - **Residual 3**: The distance from $(3, 3)$ to $(3, 2.5)$.



Concept of residuals and squared errors in the context of linear regression.

4. **Annotations**:

   - Each residual is labeled as "Residual 1," "Residual 2," and "Residual 3" to identify the vertical distances.
   - The actual data points are labeled with their coordinates in red.
   - The predicted points on the regression line are labeled with their coordinates in blue.

5. **Sum of Squared Errors (SSE)**:

   - While not explicitly shown on the graph, the SSE is calculated by summing the squares of these residuals:

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

   - This value is used to assess how well the regression line fits the data, with lower SSE indicating a better fit.

Insights:

- **Positive Residuals**: When the actual data point is above the regression line, the residual is positive (e.g., Residual 1).
- **Negative Residuals**: When the actual data point is below the regression line, the residual is negative (e.g., Residual 2).
- **Zero Residuals**: If a data point lies exactly on the regression line, the residual is zero (e.g., Residual 3).

Key Points:

- The best-fit line is determined by minimizing the sum of the squared vertical distances between the actual data points and the predicted values on the line.
- The residuals show how far off the predictions are from the actual data points.

Why we use square of error?

- In linear regression, we often use the square of the errors, rather than just the errors themselves, to measure how well the model fits the data.
- This is called the Sum of Squared Errors (SSE).

Reasons for Using Squared Errors:

1. **Penalizing Larger Errors:**
   - Squaring the errors amplifies the impact of larger deviations. This means that points that are far from the regression line have a disproportionately larger effect on the SSE.
   - For example, an error of 2 becomes 4 when squared, while an error of 5 becomes 25. This helps ensure that large errors are given more weight and significantly influence the model fitting process.
2. **Ensuring Non-Negativity:**
   - Errors can be positive or negative. If we were to sum the errors directly, positive and negative errors could cancel each other out, potentially misleading us about the overall fit of the model.
   - By squaring the errors, all values become positive, preventing cancellation and ensuring that all errors contribute positively to the total measure of error.
3. **Mathematical Convenience:**
   - Squaring the errors leads to a smooth, differentiable function. This is essential for the optimization process used to fit the model.
   - For linear regression, the sum of squared errors has a unique minimum that can be found analytically by taking derivatives. This makes the least squares method computationally efficient and straightforward to apply.
4. **Statistical Properties:**
   - Squared errors are used in the context of normally distributed errors, which is a common assumption in regression models. The sum of squared errors closely relates to the concept of variance in statistics.
   - The least squares approach (minimizing SSE) corresponds to the maximum likelihood estimation when the errors are normally distributed.

Example:

Consider a simple example with two data points and a regression line:

- Data Points: (x_1, y_1) and (x_2, y_2)
- Predicted Values: $\hat{y}_1$ and $\hat{y}_2$
- Errors: $e_1 = y_1 - \hat{y}_1$ and $e_2 = y_2 - \hat{y}_2$

If the errors are:

- $e_1 = 2$
- $e_2 = -3$

Without squaring:

- Sum of errors = $2 + (-3) = -1$ (misleadingly small and possibly zero for some datasets)

With squaring:

- Squared errors = 4 and 9
- Sum of squared errors = $4 + 9 = 13$ (clearly shows the magnitude of deviations)

Why Not Use Absolute Errors?

Using the absolute value of the errors is an alternative approach known as the least absolute deviations (LAD) method. While this approach can be more robust to outliers, it has some drawbacks:

- **Non-Differentiability**: The absolute value function is not smooth and differentiable at zero, making it harder to use in optimization algorithms that rely on derivatives.
- **Less Sensitivity to Large Errors**: Unlike squaring, absolute values don't penalize larger deviations as heavily, which might not be desirable in some cases.

Conclusion:

Using the square of errors in linear regression helps in minimizing the overall error in a manner that is mathematically convenient, statistically grounded, and intuitively appealing. This approach ensures that larger deviations are penalized more, non-negative errors are summed, and the optimization process is simplified and efficient.

Why Not Use Absolute Errors?

Why Differentiability Matters in Optimization:

In optimization, especially for methods like gradient descent, we need to know the direction in which to move to reduce the error. This direction is given by the derivative (or gradient in higher dimensions). A smooth, differentiable function provides clear guidance on how to adjust the model parameters to reduce the error.

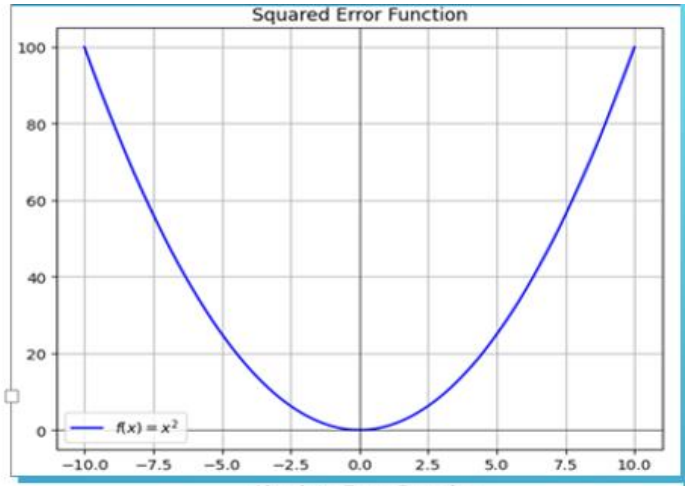
- **Smooth Functions (e.g., x^2)**: Allow for straightforward calculation of gradients, leading to efficient and effective optimization.
- **Non-Smooth Functions (e.g., $|x|$)**: Pose challenges in optimization due to undefined or abrupt changes in gradients at certain points, making it harder to find the optimal solution.

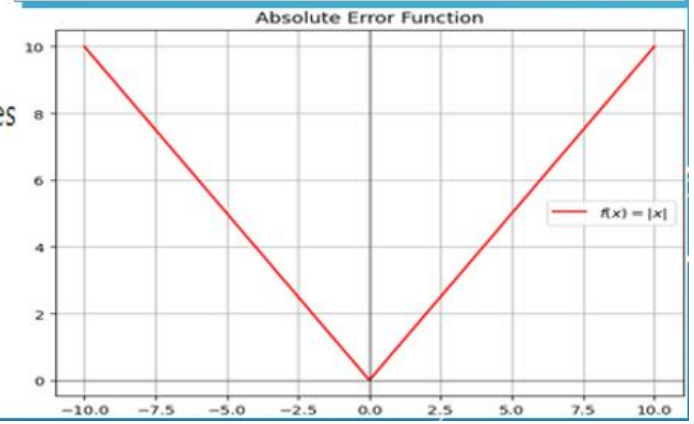
Squared Error Function $f(x) = x^2$:

- **Shape**: The U-shape of the x^2 function means it smoothly curves upwards. This smoothness implies that for any small change in x, the change in y is continuous and predictable.
- **Differentiability**: Because the curve is smooth and has no sharp points, it's differentiable everywhere. This is crucial for optimization techniques like gradient descent, which rely on calculating derivatives to find the minimum error efficiently.

Absolute Error Function $f(x) = |x|$:

- **Shape**: The V-shape indicates a sharp turn at the origin. This point is where the function changes direction abruptly.
- **Non-Differentiability**: At $x = 0$, the slope changes instantaneously from -1 to 1. This means there's no unique tangent line at $x = 0$, making the function non-differentiable at this point. Optimization algorithms that depend on derivatives can't handle this abrupt change well.





R squared: Coefficient of Determination

$$R^2 = 1 - \frac{\text{SSE}}{\text{SST}}$$

SSE (Sum of Squared Errors or Residual Sum of Squares):

- Represents the sum of the squared differences between the observed actual values and the values predicted by the model.
- **Formula:**

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

  where y_i is the actual value, $\hat{y}_i$ is the predicted value, and n is the number of observations.

SST (Total Sum of Squares):

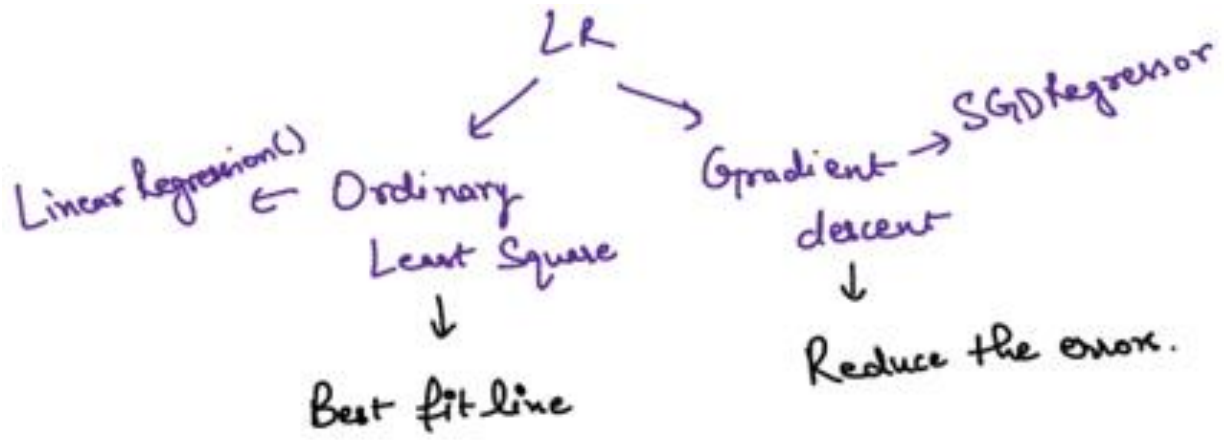
- Represents the total variance in the dependent variable. It is the sum of the squared differences between the observed actual values and the mean of those values.
- **Formula:**

  $\text{SST} = \sum_{i=1}^{n} (y_i - \bar{y})^2$

  where $\bar{y}$ is the mean of the observed values.

Given the equation $R^2 = 1 - \frac{\text{SSE}}{\text{SST}}$:

1. **SST (Total Variability)**:
    - Measures the total variability in the dependent variable.
    - Compares each data point to the mean of the dependent variable.
    - Indicates how much the actual data points deviate from the mean.
2. **SSE (Model Error)**:
    - Measures the error in the model's predictions.
    - Compares each data point to the predicted value from the model.
    - Indicates how much the actual data points deviate from the predicted values.
3. **R-squared Calculation**:
    - R^2 represents the proportion of the total variability that is explained by the model.
    - A higher R^2 means a better fit of the model to the data, indicating that more variance is explained by the independent variables.



- Ordinary Least Squares(OLS): Work on Best fit line
- Gradient Descent: Work on the concept of Reduce the error

Ordinary Least Squares (OLS):

- Directly solve the normal equation $\theta = (X^T X)^{-1} X^T y$.
- This method provides a closed-form solution and is generally used when the dataset is not too large and the design matrix X is not close to being singular.

Gradient Descent:

- Initialize parameters θ (usually starting with zeros or small random values).
- Update θ iteratively to minimize the cost function $J(\theta)$, typically the mean squared error
- This method is useful for large datasets or when dealing with high-dimensional spaces where OLS is computationally infeasible.

Example: Multiple Linear Regression

For multiple linear regression with more features, the model extends to:

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$$

In this model:

- θ_0 is the intercept (bias term).
- $\theta_1, \theta_2, \ldots, \theta_n$ are the slopes (coefficients for each feature).

Matrix Notation

In matrix form, this can be expressed as:

$$y = X\theta$$

Where:

- X is an $m \times (n + 1)$ matrix (including a column of 1s for the intercept).
- θ is a vector with $n + 1$ components (intercept and slopes).
- y is the vector of observed outcomes.

Normal Equation

The normal equation is used to find the optimal θ that minimizes the sum of squared errors:

$$\theta = (X^T X)^{-1} X^T y$$

$$\frac{\partial E}{\partial m} = \sum_{i=1}^{n} (y_i - mx_i - b)^2 = 0$$

$$\Rightarrow \sum_{i=1}^{n} (y_i - mx_i - \bar{y} + m\bar{x})^2 = 0$$

$$\Rightarrow \sum_{i=1}^{n} -(y_i - mx_i - \bar{y} + m\bar{x})(-x_i + \bar{x}) = 0$$

$$\Rightarrow \sum_{i=1}^{n} (y_i - mx_i - \bar{y} + m\bar{x})(\bar{x} - x_i) = 0$$

$$\Rightarrow \sum_{i=1}^{n} [y_i - \bar{y} - m(x_i - \bar{x})](x_i - \bar{x}) = 0$$

$$\Rightarrow \sum [(y_i - \bar{y})(\bar{x} - x_i) - m(\bar{x} - x_i)^2] = 0$$

$$\sum m(\bar{x} - x_i)^2 = \sum (y_i - \bar{y})(x_i - \bar{x})$$

slope for best fit line $\leftarrow$ $m = \dfrac{\sum (y_i - \bar{y})(\bar{x} - x_i)}{\sum (x_i - \bar{x}_i)^2}$

Take Square of it? $\rightarrow$

$E(m,b) = (y_i - \hat{y}_i)^2$

$$\frac{\partial E}{\partial b} = \frac{\partial (y_i - mx_i - b)^2}{\partial b} = 0$$

$$\Rightarrow \sum_{i=1}^{n} (y_i - mx_i - b)(-1) = 0$$

$$\Rightarrow \sum_{i=1}^{n} y_i - \sum_{i=1}^{n} mx_i - \sum_{i=1}^{n} b = 0$$

divide by n

$$\Rightarrow \sum_{i=1}^{n} \frac{y_i}{n} - \sum_{i=1}^{n} \frac{mx_i}{n} - \sum_{i=1}^{n} \frac{b}{n} = 0$$

$$\Rightarrow \bar{y} - m \sum_{i=1}^{n} \frac{x_i}{n} - \frac{nb}{n} = 0$$

$$\Rightarrow \bar{y} - m\bar{x} - b = 0$$

b $\Rightarrow$ intercept value for best fit line

$b = \bar{y} - m\bar{x}$

Math's to Find Slope and Intercept

- **Identification of significant variables:**
  - **It can be done during Exploratory Data Analysis (EDA)**
  - **As well as during model building.**

1. Correlation During EDA

During EDA, correlation analysis helps identify relationships between variables.

- **Pearson Correlation**: Measures the linear relationship between two continuous variables. Values range from -1 to 1, where values closer to 1 or -1 indicate a strong relationship, and values near 0 indicate a weak or no linear relationship.

$$\rho_{X,Y} = \frac{cov(X,Y)}{\sigma_X \sigma_Y}$$

- **Spearman's Rank Correlation**: A non-parametric measure that assesses how well the relationship between two variables can be described using a monotonic function. Suitable for ordinal data or non-linear relationships.

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2-1)}$$

Use in EDA:

- Identify pairs of variables that have a high correlation, indicating possible redundancy or multicollinearity.
- Detect potential predictor variables that have strong relationships with the target variable.
- Highlight interesting relationships and patterns in the data that may warrant further investigation.

2. Variance Inflation Factor (VIF) After Modeling

Once you start building regression models, especially multiple linear regression, checking for multicollinearity among predictors is essential. VIF quantifies how much the variance of a regression coefficient is inflated due to multicollinearity.

- **VIF Formula:**

  $$VIF_i = \frac{1}{1-R_i^2}$$

  where R_i^2 is the coefficient of determination of the regression of predictor i on all the other predictors.

Use in Modeling:

- **VIF > 10**: Indicates significant multicollinearity that could affect the stability and interpretability of the regression coefficients. This threshold is commonly used, but in some cases, a VIF > 5 can also be considered concerning.
- **VIF < 10**: Generally considered acceptable, indicating low multicollinearity.

GRADIENT DESCENT APPROACH:

Gradient Descent is a powerful optimization algorithm used to minimize the cost function in machine learning and deep learning. For Multiple Linear Regression, it iteratively updates the model's parameters (weights and biases) to find the best fit line for the given data.

Multiple Linear Regression Model

For a dataset with m samples and n features (independent variables), the model can be expressed as:

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

Where:

- $\hat{y}^{(i)}$ is the predicted value for the i-th sample.
- $x_j^{(i)}$ is the j-th feature for the i-th sample.
- θ_0 is the intercept (bias term).
- $\theta_1, \theta_2, \ldots, \theta_n$ are the coefficients (weights).

Cost Function (Mean Squared Error)

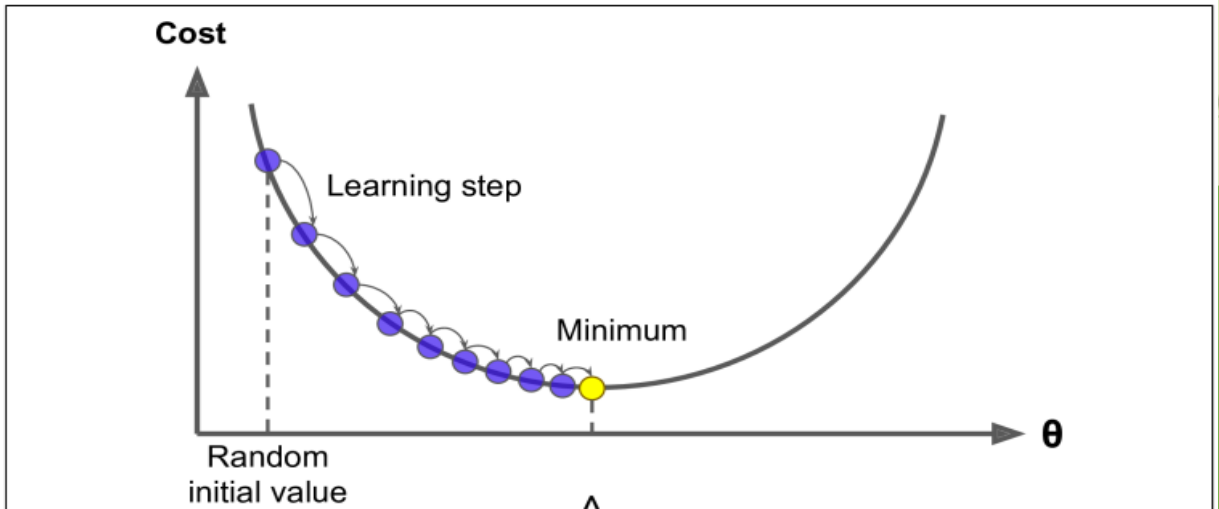
The cost function $J(\theta)$ is defined as the Mean Squared Error (MSE) between the predicted values $\hat{y}$ and the actual values y:

$$J(\theta) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2$$

What is Gradient Descent

- **Gradient Descent** is a very generic optimization algorithm capable of finding optimal solutions to a wide range of problems. The general idea of Gradient Descent is to tweak parameters iteratively in order to minimize a cost function.
- Suppose you are lost in the mountains in a dense fog; 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.
- This is exactly what Gradient Descent does:
- it measures the local gradient of the error function with regards to the parameter vector θ, and it goes in the direction of descending gradient.
- Once the gradient is zero, you have reached a minimum!
- So, you start by filling θ with random values (this is called random initialization), and 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



- An important parameter in Gradient Descent is the size of the steps, determined by the **learning rate hyperparameter**.
- If the learning rate is too small, then the algorithm will have to go through many iterations to converge, which will take a long time
- On the other hand, if the learning rate is too high, you might jump across the valley and end up on the other side, possibly even higher up than you were before.
- This might make the algorithm diverge, with larger and larger values, failing to find a good solution

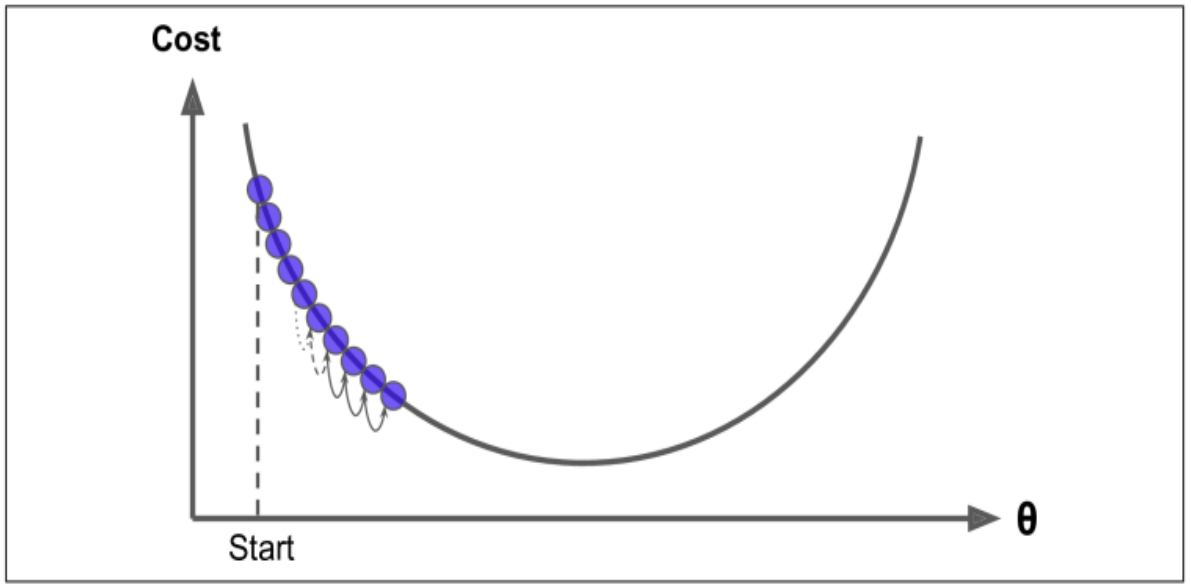


Figure 4-4. Learning rate too small

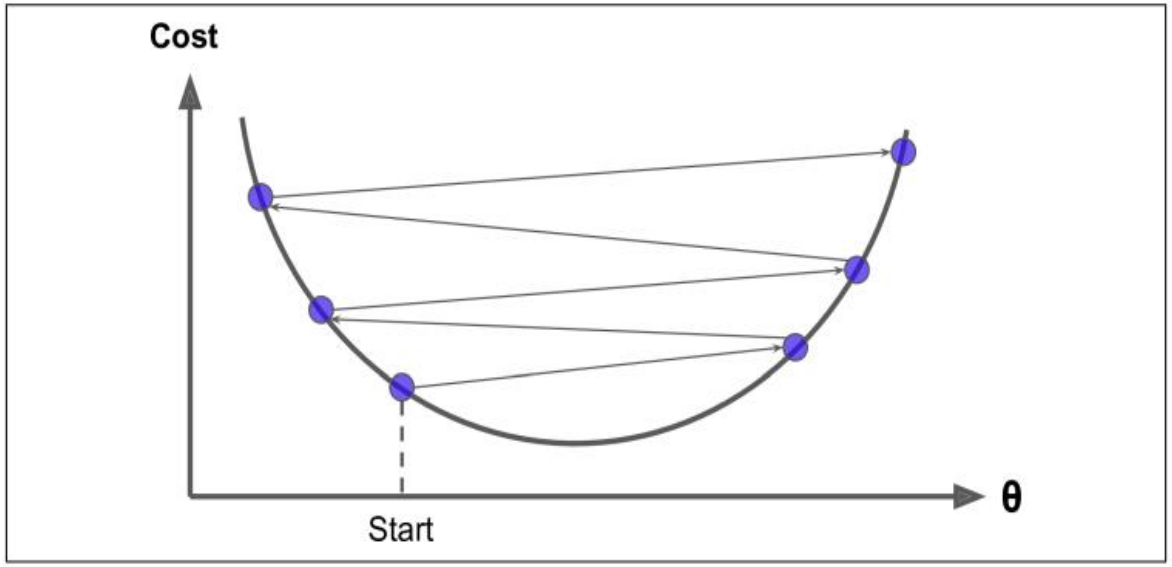


Figure 4-5. Learning rate too large

- 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.

- Fortunately, the MSE 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 never crosses 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.4

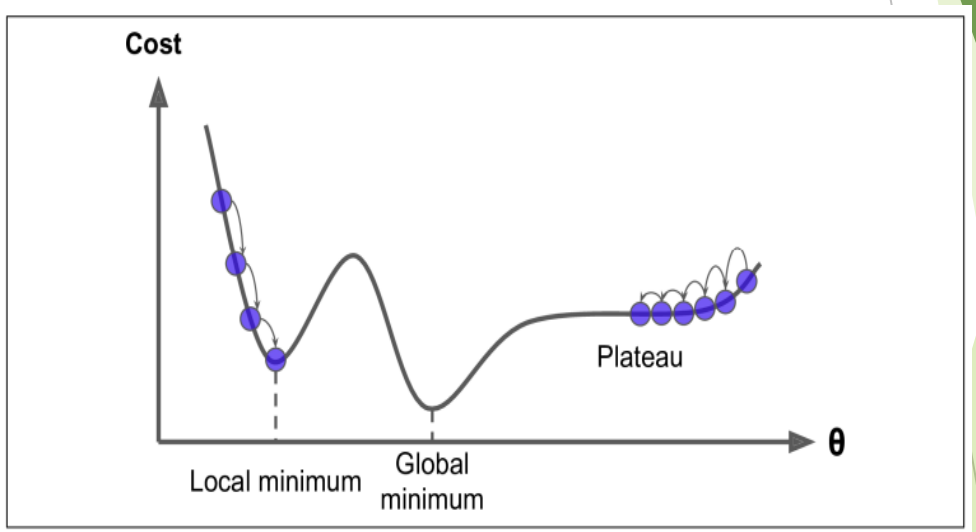
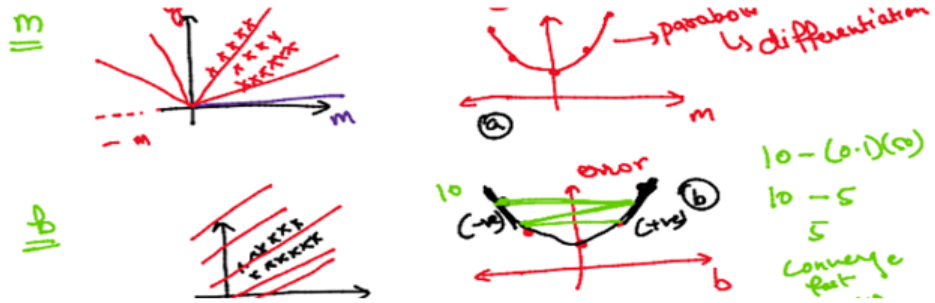


Figure 4-6. Gradient Descent pitfalls

Gradient Descent Algorithm

The gradient descent algorithm minimizes the cost function by updating the model's parameters in the direction of the steepest descent of the cost function. The update rule for each parameter θ_j is:

$$\theta_j := \theta_j - \alpha \frac{\partial J(\theta)}{\partial \theta_j}$$



Where:

- α is the learning rate.
- $\frac{\partial J(\theta)}{\partial \theta_j}$ is the partial derivative of the cost function with respect to θ_j.

Partial Derivatives

For Multiple Linear Regression, the partial derivatives of the cost function with respect to the parameters are:

$$\frac{\partial J(\theta)}{\partial \theta_j} = \frac{1}{m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)}) x_j^{(i)}$$

- Where derivative of loss or cost with weight is called slope.
- Its direction decide in which direction we need to move to reach a point where loss is minimum.
- The derivative of loss wrt ndim of vector is called gradient.
- Where ndim vector is called a tensor.
- In calculus derivative of tensor is referred as tensor.
- In machine learning, data with n number of features is represented as a tensor.

Derivative

Definition: The derivative of a function $f(x)$ at a point x is defined as the limit of the average rate of change of the function over an interval as the interval approaches zero. Mathematically, this is expressed as:

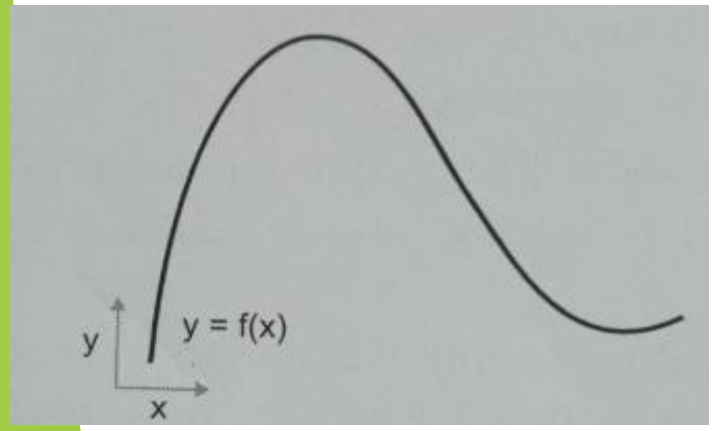
$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

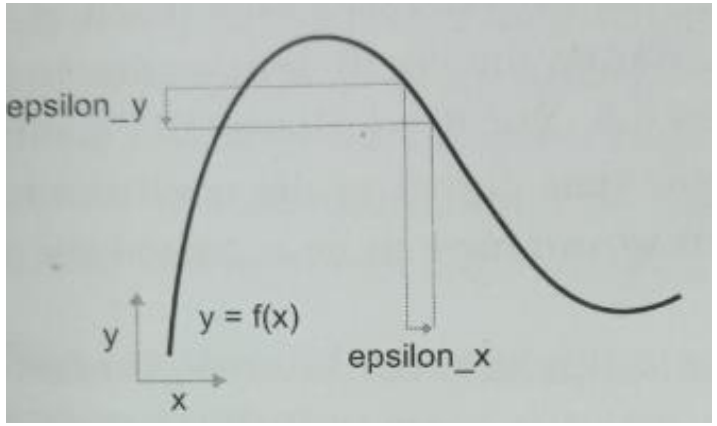
where $f'(x)$ denotes the derivative of $f(x)$ with respect to x.

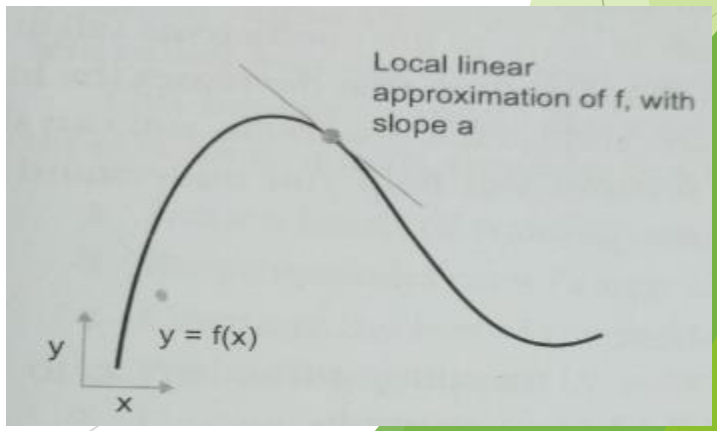
Notation: The derivative can be denoted in several ways, including:

- $f'(x)$
- $\frac{df}{dx}$
- $Df(x)$
- $\dot{f}(x)$ (in the context of time derivatives in physics)

Interpretation: The derivative $f'(x)$ represents the slope of the tangent line to the graph of the function $f(x)$ at the point x. It tells us how $f(x)$ changes as x changes.







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

m = constant

b = ?

is working slow

Step 1 → Take random value of b

Step2: $\frac{\partial L}{\partial b} = \frac{\partial}{\partial b} \sum_{i=1}^{n} (y_i - mx_i - b)^2$ $\quad b=0$

$= 2(y_i - mx_i - b)(-1)$

Step3: $b_{next} = b_{old} - \text{slope} \rightarrow \frac{\partial L}{\partial b}$

$= b_{old} - \frac{\partial L}{\partial b}$

$\text{slope} = -2(y_i - mx_i - b)$

$= -2(y_i - mx_i)$

* add learning rate α or γ → hyperparameter

$b_{next} = b_{old} - \alpha \left(\frac{\partial L}{\partial b}\right)$ $\quad [0.001, 10]$

learning rate = stepsize

Loss $\sum_{i=1}^{n} (y_i - mx_i - b)^2 = E(m, b)$ $\quad b_{next} = b_{old} - \eta(\text{slope})$

$\text{slope}_b = \frac{\partial E}{\partial b} \Rightarrow -2(y_i - mx_i - b)$

(II) $m_{next} = m_{old} - \eta(\text{slope})$

$\text{slope}_m \quad \frac{\partial E}{\partial m} = 2(y_i - mx_i - b)x_i$

Hyperparameters:

- `learning_rate`: Determines the step size for each iteration.
- `n_iterations`: The number of iterations to run the gradient descent.

Initialization:

- `theta`: Initialized to zeros. It's the vector of parameters to be learned.

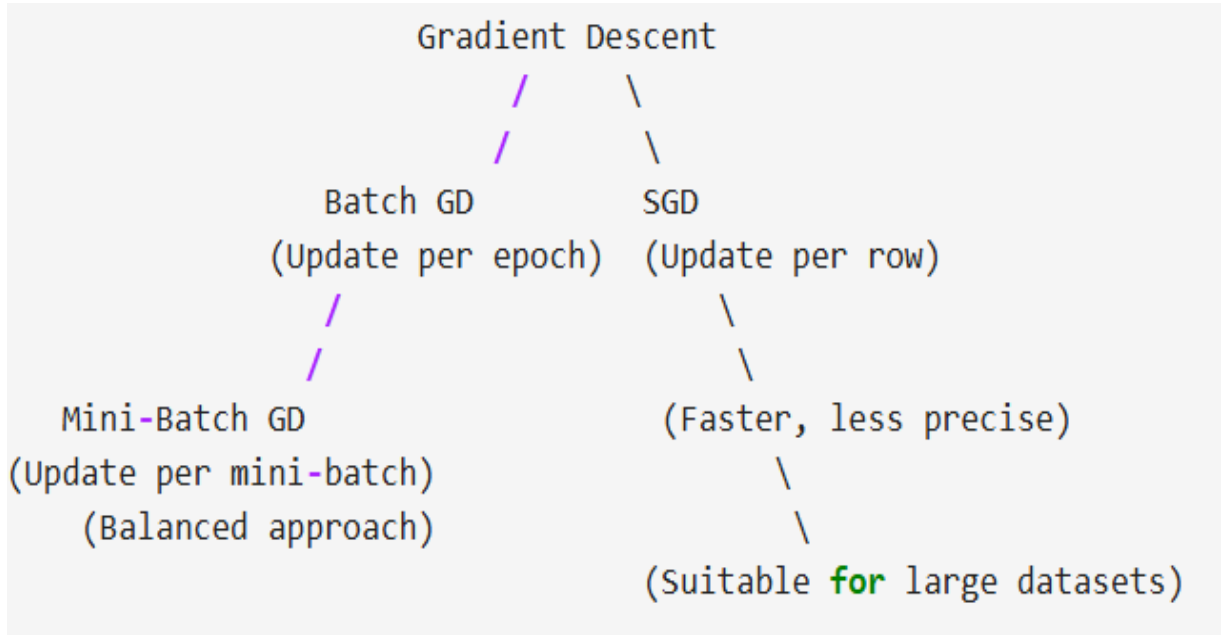
Gradient Descent Loop:

- **Predictions**: Calculate the predicted values using the current parameters.
- **Errors**: Compute the difference between predicted and actual values.
- **Gradient**: Compute the gradient of the cost function with respect to each parameter.
- **Update**: Adjust the parameters in the opposite direction of the gradient.

Key Points:

- **Learning Rate** (α): A too-large learning rate can cause the algorithm to overshoot the minimum, while a too-small learning rate can make the convergence slow.
- **Number of Iterations**: More iterations can lead to a better fit but also increase computation time. A good stopping criterion is when the change in the cost function is very small between iterations.
- **Feature Scaling**: Gradient descent converges faster if the features have similar scales. Preprocessing techniques like normalization or standardization are often applied to the feature data before training.

Gradient Descent: Types



Stochastic Gradient Descent (SGD)

1. **Stochastic Gradient Descent (SGD)**
    - **Process:**
        - Updates model parameters (e.g., weights) for each training example.
        - Performs updates row-wise (one data point at a time).
    - **Characteristics:**
        - Faster but can have less precise results due to the noise in updates.
    - **Example:**
        - For 100 iterations with 100 rows of data:
            - 1st iteration: Updates b, m using the 1st row.
            - 2nd iteration: Updates b, m using the 2nd row.
            - Continues until 100th row.
            - Repeats this process for 100 iterations, resulting in $100 \times 100 = 10,000$ updates.

Batch Gradient Descent

2. **Batch Gradient Descent**

- **Process:**
  - Updates model parameters using the entire dataset in each iteration.
- **Characteristics:**
  - More precise but slower due to computing gradients for the entire dataset.
  - Suitable for smaller datasets.
- **Example:**
  - For 100 iterations with 100 rows of data:
    - 1st iteration: Computes the gradient using all 100 rows and updates b, m.
    - 2nd iteration: Computes the gradient again using all 100 rows and updates b, m.
    - Repeats this process for 100 iterations, resulting in 100 updates.

Mini-Batch Gradient Descent

- **Process:**
  - Combines aspects of both SGD and Batch Gradient Descent.
  - Updates model parameters using a subset (mini-batch) of the dataset in each iteration.
- **Example:**
  - Uses mini-batches of, for instance, 30 rows.
  - Suitable for handling large datasets and deep learning tasks.

Linear Regression and optimization

Linear Regression Objective:

$$(\hat{w_0}, \hat{w^T}) = \arg\min_{w_0, w^T} \sum_{i=1}^{n} (y_i - \hat{y_i})^2$$

This represents the objective function for linear regression, where:

- $\hat{w_0}$ and $\hat{w^T}$ are the estimated parameters.
- y_i is the actual value.
- $\hat{y_i}$ is the predicted value.
- The goal is to find $\hat{w_0}$ and $\hat{w^T}$ that minimize the sum of squared differences (errors) between the actual and predicted values.

$$\hat{y_i} = w^T x_i + w_0$$

This is the linear model used for prediction:

- x_i is the feature vector for the i-th observation.
- w^T is the transpose of the weight vector.
- w_0 is the bias term.
- The predicted value $\hat{y_i}$ is obtained by the dot product of the weight vector and the feature vector, plus the bias term.

Linear Regression And Optimization

Optimization Problem:

The optimization problem can be restated as minimizing the sum of squared errors:

$$(\hat{w_0}, \hat{w^T}) = \arg\min_{w_0, w^T} \sum_{i=1}^{n} \left(y_i - (w^T x_i + w_0)\right)^2$$

- This formulation explicitly shows the squared error for each prediction.
- The objective is to find the parameters w_0 and w^T that minimize this sum.

- Linear regression aims to minimize the squared loss, which measures the discrepancy between the actual and predicted values.
- The squared loss function is fundamental in regression analysis for evaluating the performance of a model.

Overfitting, Under fitting, and Best Fit

- **Threshold Accuracy**:
- It's indicated that an accuracy threshold of 70-95% (or 0.7-0.95) is desired.
- This is the target range for acceptable model performance.

1. **Overfitting**:
   - **High Variance**: Overfitting occurs when a model learns the training data too well, including noise and outliers. This results in high accuracy on the training set but poor generalization to new, unseen data (test set).
   - **Example from Diagram**: A model with 90-100% accuracy on training data but only 70% accuracy on test data is overfitting. The model captures the nuances of the training data but fails to generalize.
2. **Underfitting**:
   - **High Bias**: Underfitting happens when a model is too simple to capture the underlying patterns in the data. It performs poorly on both training and test data.
   - **Example from Diagram**: A model with about 50% accuracy on both training and test data is underfitting. It's too simplistic and fails to capture the complexity of the data.
3. **Best Fit Model**:
   - **Balanced Fit**: The ideal model balances bias and variance, performing well on both training and test data.
   - **Example from Diagram**: A model with approximately 88-92% accuracy on both training and test sets represents a good fit. It captures the data patterns well without overfitting or underfitting.

Key Concepts in Regularization:

1. **Overfitting and Underfitting**:
    - **Overfitting** occurs when a model learns the noise in the training data, fitting it too closely and performing poorly on new, unseen data. It typically results from a model being too complex with too many parameters.
    - **Underfitting** happens when a model is too simple to capture the underlying pattern of the data, leading to poor performance both on the training and test data.
2. **Regularization**:
    - Regularization techniques are used to prevent overfitting by adding a penalty to the loss function, discouraging the model from fitting the training data too closely. It helps to maintain a balance between model complexity and performance.

Regularization

Types of Regularization:

1. **Lasso (L1 Regularization):**

   - **Concept**: Lasso adds a penalty equivalent to the absolute value of the magnitude of coefficients.
   - **Effect**: This can shrink some coefficients to exactly zero, effectively performing feature selection by eliminating less important features.
   - **Relation to Overfitting**: By reducing some coefficients to zero, Lasso simplifies the model, which can help mitigate overfitting.
   - **Formula**:

$$\text{Minimize} \quad \sum_{i=1}^{n} \left(y_i - (w^T x_i + w_0)\right)^2 + \lambda \sum_{j=1}^{p} |w_j|$$

   Where λ is the regularization parameter controlling the strength of the penalty, w_j are the model coefficients.

Types of Regularization

2. **Ridge (L2 Regularization):**

   - **Concept**: Ridge adds a penalty equivalent to the square of the magnitude of coefficients.
   - **Effect**: Unlike Lasso, Ridge does not shrink coefficients to zero but rather reduces them closer to zero.
   - **Relation to Overfitting**: Ridge prevents overfitting by discouraging large coefficients, thereby reducing the model's complexity.
   - **Formula**:

$$\text{Minimize} \quad \sum_{i=1}^{n} \left(y_i - (w^T x_i + w_0)\right)^2 + \lambda \sum_{j=1}^{p} w_j^2$$

   Where λ is the regularization parameter.

Types of Regularization

3. **Elastic Net (Combination of L1 and L2 Regularization):**

   - **Concept**: Elastic Net combines both L1 and L2 penalties. It adds a mix of L1 and L2 penalties to the loss function.
   - **Effect**: It can perform feature selection (like Lasso) and reduce coefficients (like Ridge), making it versatile in handling datasets with correlated features or when performing feature selection and model complexity reduction simultaneously.
   - **Relation to Overfitting**: By balancing both L1 and L2 penalties, Elastic Net provides a robust way to control overfitting and adapt to different types of data.
   - **Formula**:

$$\text{Minimize} \quad \sum_{i=1}^{n} \left(y_i - (w^T x_i + w_0)\right)^2 + \lambda_1 \sum_{j=1}^{p} |w_j| + \lambda_2 \sum_{j=1}^{p} w_j^2$$

Where λ_1 and λ_2 control the L1 and L2 regularization strengths, respectively.

Application and Interpretation:

- When choosing a regularization method, consider the nature of your data and the problem at hand. Lasso is beneficial when you expect many irrelevant features, Ridge is useful when all features are expected to be relevant, and Elastic Net is suitable for scenarios involving both correlated features and a need for feature selection.

1. **Lasso** is depicted as the first type of regularization, emphasizing how it drives coefficients to be exactly zero, which helps in feature selection and reducing overfitting by simplifying the model.

2. **Ridge** is shown as the second type, highlighting its approach to pull the coefficients closer to zero but not necessarily to zero, thus controlling the model complexity and avoiding overfitting.

3. **Elastic Net** is introduced as a combination of both L1 and L2 regularization, offering a balance that can manage both sparsity (feature selection) and coefficient shrinkage, providing a middle ground in controlling overfitting.

Application and Interpretation:

- When evaluating a linear regression model, several error metrics help determine the model's performance.
- Each serves a slightly different purpose.
- The order of accuracy typically depends on the sensitivity of the metric to outliers and the emphasis on specific error magnitudes.
- Here is a brief overview of the key error metrics, their order, and when to use them:

Evaluation of a Regression Model:

1. Mean Absolute Error (MAE):

- **Definition**: Average of the absolute differences between actual and predicted values.
  $\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$
- **Interpretation**:
  - Measures error magnitude without considering direction (sign).
  - Linear penalty for errors.
- **Accuracy**: Moderate; **robust to outliers** because it doesn't square the error.
- **When to use**: Use when you want an error measure that is simple and not sensitive to large deviations.

2. Mean Squared Error (MSE):

- **Definition**: Average of the squared differences between actual and predicted values.
  $\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$
- **Interpretation**:
  - Penalizes larger errors more than smaller ones because errors are squared.
  - Units of MSE are squared, which can be harder to interpret.
- **Accuracy**: Higher than MAE in cases with minimal outliers, as it magnifies large deviations.
- **When to use**: Use when you want to heavily penalize large errors and emphasize minimizing them.

3. Root Mean Squared Error (RMSE):

- **Definition**: Square root of the MSE.

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

- **Interpretation**:
  - Gives the error magnitude in the same units as the target variable.
  - It is sensitive to outliers, similar to MSE.
- **Accuracy**: RMSE provides a balanced measure of error when penalizing larger deviations.
- **When to use**: Use when you want a measure in the same units as the dependent variable and care about significant errors.

4. Mean Absolute Percentage Error (MAPE):

- **Definition**: Average of the absolute percentage errors.

$$\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100$$

- **Interpretation**:
  - Expresses errors as percentages, making it scale-independent.
  - Sensitive to small actual values (division by small numbers inflates the error).
- **Accuracy**: Accuracy depends on the scale of target values; not ideal if actual values are near zero.
- **When to use**: Use for comparing models across datasets or scenarios with different scales.

Comparison Order of Accuracy:

1. **RMSE** (most accurate but sensitive to outliers).
2. **MSE** (similar to RMSE but in squared units).
3. **MAE** (more robust to outliers than MSE and RMSE).
4. **MAPE** (useful for scale-independent error assessment but can be tricky for very small values).

When to Use Which Metric?

- **MAE**: When you want robustness and interpretability without emphasizing large errors.
- **MSE** or **RMSE**: When you care more about penalizing large errors significantly.
- **MAPE**: When comparing errors as percentages, especially across datasets.

In practice:

- If **outliers** are a concern: Use **MAE**.
- If you want more emphasis on large errors: Use **RMSE** or **MSE**.
- If interpretability and units matter: Prefer **RMSE**.

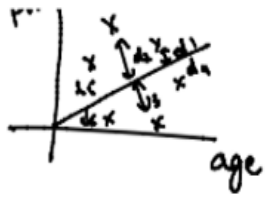
Model Evaluation Techniques

1. **Train-Test Split**:
    - Dividing the dataset into a training set to train the model and a test set to evaluate it.
    - Ensures that the model is evaluated on unseen data, providing a more realistic measure of its performance.
2. **Cross-Validation**:
    - Technique to assess how the model generalizes to an independent dataset.
    - Commonly used method is k-fold cross-validation where the dataset is divided into k subsets, and the model is trained k times, each time using a different subset as the test set and the remaining data as the training set.
    - Helps in reducing overfitting and provides a better estimate of model performance.
3. **Residual Analysis**:
    - Examines the residuals (differences between actual and predicted values) to diagnose model fit.
    - Plotting residuals can help identify patterns that suggest non-linearity, heteroscedasticity, or outliers.

Visualizing Model Performance

1. **Residual Plots:**
    - Scatter plots of residuals versus predicted values or independent variables.
    - Ideally, residuals should be randomly scattered without patterns, indicating good fit.
2. **Actual vs. Predicted Plot:**
    - Plot the actual outcomes versus predicted outcomes.
    - Ideally, points should lie on or near the 45-degree line, indicating accurate predictions.
3. **Histogram or Density Plot of Residuals:**
    - Helps assess the normality of the residuals. In linear regression, we often assume residuals are normally distributed.
4. **Q-Q Plot:**
    - Compares the quantiles of the residuals to the quantiles of a normal distribution.
    - Helps to visually check if the residuals follow a normal distribution.

Evaluations



$$MAE = \frac{1}{n}\sum_{i=1}^{n} d_i$$

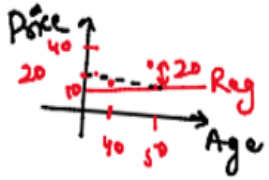
$$= \frac{1}{n}\sum_{i=1}^{n} |y_i - \hat{y}_i|$$

$\hookrightarrow 0$

Advantages

→ scale/unit are same, interpretable

→ Robust to outliers



→ **MSE** ⇒ mean squared error

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

Advantages

→ Loss fⁿ

disadvantages:

→ not in same scale/ less interpretable

→ sensitive to outliers.

⇒ **RMSE** ⇒ bring MSE to same scale

→ interpretable → ∠fⁿ

Parameters

↓ R^2 score → goodness of fit / coeff. of determination.

R^2 → subscript (R squared)

$$R^2 = 1 - \frac{SSR}{SSM}$$

Case 1: Best line ideal, SSR = 0 — Overfitting

$$R^2 = 1 - \frac{0}{SSM} = 1 - 0 = 1$$

Case 2: worst line ideal, SSR = SSM. — underfitting

$$R^2 = 1 - \frac{SSR}{SSR} = 1 - 1 = 0$$

Case 3: SSR > SSM

$$R^2 = 1 - \left(\frac{SSR}{SSM}\right) \quad \frac{SSR}{SSM} > 1$$

* R^2 = −ve

Problem with R^2 score?

As the # columns ↑, R^2 score ↑.

IQ	Study hrs	Marks

$R^2 = 0.85$

IQ	Study hrs	Outlook	Marks

$R^2 = 0.90$ or 0.95

⇒ Adjusted R^2 score:

$$\text{Adj } R^2 = 1 - \left[\frac{(1-R^2)(n-1)}{(n-1-p)}\right]$$

const.

p = # no. of ind. columns

n = # rows

R^2 = R^2 score

If $(n-1-p)$ decrease > $(1-R^2)$ decrease

Adj R^2 ↓

If $(1-R^2)$ decrease > $(n-1-p)$ decrease

Adj R^2 ↑

If you have a lot of columns, Rely Adj R^2

↓

Assignment: Create a python function

⇒ Multicollinearity ⇒ Dummy variable

↓

one column is highly correlated with another column.

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3$$

$$y = \beta_0 + \beta_1 x_1 + \beta_1 x_2^2 + \beta_3 x_3 \qquad \beta_2 = \beta_1 x_2$$

This fn is not reliable

Rectify: Drop any one of the highly correlated columns.

Detect: → Correlation Matrix → 0.75 or −0.75

→ VIF ⇒ $\frac{1}{1-R^2}$ ⇒ $[1, \infty]$

VIF > 5

LR Model → Errors & R^2 & Adj R^2

↓

VIF

↙

variance inflation factor

EDA → Correlation matrix

↓

LR Model

Variance Inflation Factor (VIF):

Purpose:

VIF is used to detect multicollinearity among independent variables in a regression model. Multicollinearity occurs when independent variables are highly correlated with each other, which can distort the estimated coefficients and weaken the statistical power of the regression analysis.

Definition:

The VIF for each predictor variable measures how much the variance of the estimated regression coefficient is inflated due to multicollinearity.

Calculation:

VIF is calculated using the formula:

$$\text{VIF}(X_j) = \frac{1}{1 - R_j^2}$$

where R_j^2 is the coefficient of determination of the regression of the j-th predictor on all the other predictors.

Interpretation:

- **VIF = 1**: No correlation between the j-th predictor and the other predictors.
- **1 < VIF ≤ 5**: Moderate correlation but not severe enough to cause concern.
- **VIF > 5**: High correlation, suggesting significant multicollinearity.
- **VIF > 10**: Very high correlation, indicating severe multicollinearity and a potential problem in the regression model.

Usage:

1. **Identify Multicollinearity**:
   - Calculate the VIF for each predictor.
   - If VIF values are high, consider removing or combining highly correlated variables.

Durbin-Watson Test:

Purpose:

The Durbin-Watson (DW) test checks for the presence of autocorrelation (serial correlation) in the residuals of a regression model. Autocorrelation can indicate that the residuals are not independent, violating one of the key assumptions of linear regression.

Definition:

The Durbin-Watson statistic is calculated using the formula:

$$DW = \frac{\sum_{t=2}^{n}(e_t - e_{t-1})^2}{\sum_{t=1}^{n} e_t^2}$$

where e_t are the residuals from the regression model at time t.

Interpretation:

- **DW ≈ 2**: No autocorrelation.
- **0 < DW < 2**: Positive autocorrelation. Residuals are positively correlated, which can indicate that successive errors are similar.
- **2 < DW < 4**: Negative autocorrelation. Residuals are negatively correlated, suggesting that errors tend to alternate signs.
- **DW < 1 or DW > 3**: Strong evidence of autocorrelation.

Usage:

1. **Detecting Autocorrelation:**
   - Calculate the DW statistic for the residuals of your regression model.
   - Compare the DW value to the critical values from the DW table to determine if there is significant autocorrelation.

Summary:

- **VIF** helps identify multicollinearity among predictors, ensuring that the model's estimates are reliable and interpretable.
- **Durbin-Watson Test** detects autocorrelation in the residuals, which is crucial for ensuring the independence of observations in the regression model.

1. **Training and Testing Data Splits**:
    - Commonly used splits for dividing the dataset into training and testing sets are 60%-40%, 65%-35%, or 70%-30%.
    - In this example, the dataset is split into 80% for training (800 observations) and 20% for testing (200 observations).
2. **Building the Model**:
    - The training set (800 observations) is used to build the regression model.
    - The model learns the relationships between the independent variables (predictors) and the dependent variable (response).
3. **Evaluating the Model**:
    - The testing set (200 observations) is used to evaluate the model's performance.
    - This helps to check how well the model generalizes to unseen data.
4. **Holdout Method**:
    - The testing set is sometimes referred to as a "holdout" set because it is held back and not used during model training.
    - After the model is built on the training data, it is validated on this holdout set to assess its predictive accuracy and robustness.

Train-Test Split

Dataset Structure

5. **Dataset Organization**:
    - The dataset is organized into columns representing the Independent Variables (Idv) and the Dependent Variable (DV).
    - Each row represents an observation in the dataset.
6. **Past Data**:
    - Historical data (past data) for both independent and dependent variables is typically used to train the model.
    - For example, if we have 1000 data points, they will be split into training and testing sets.

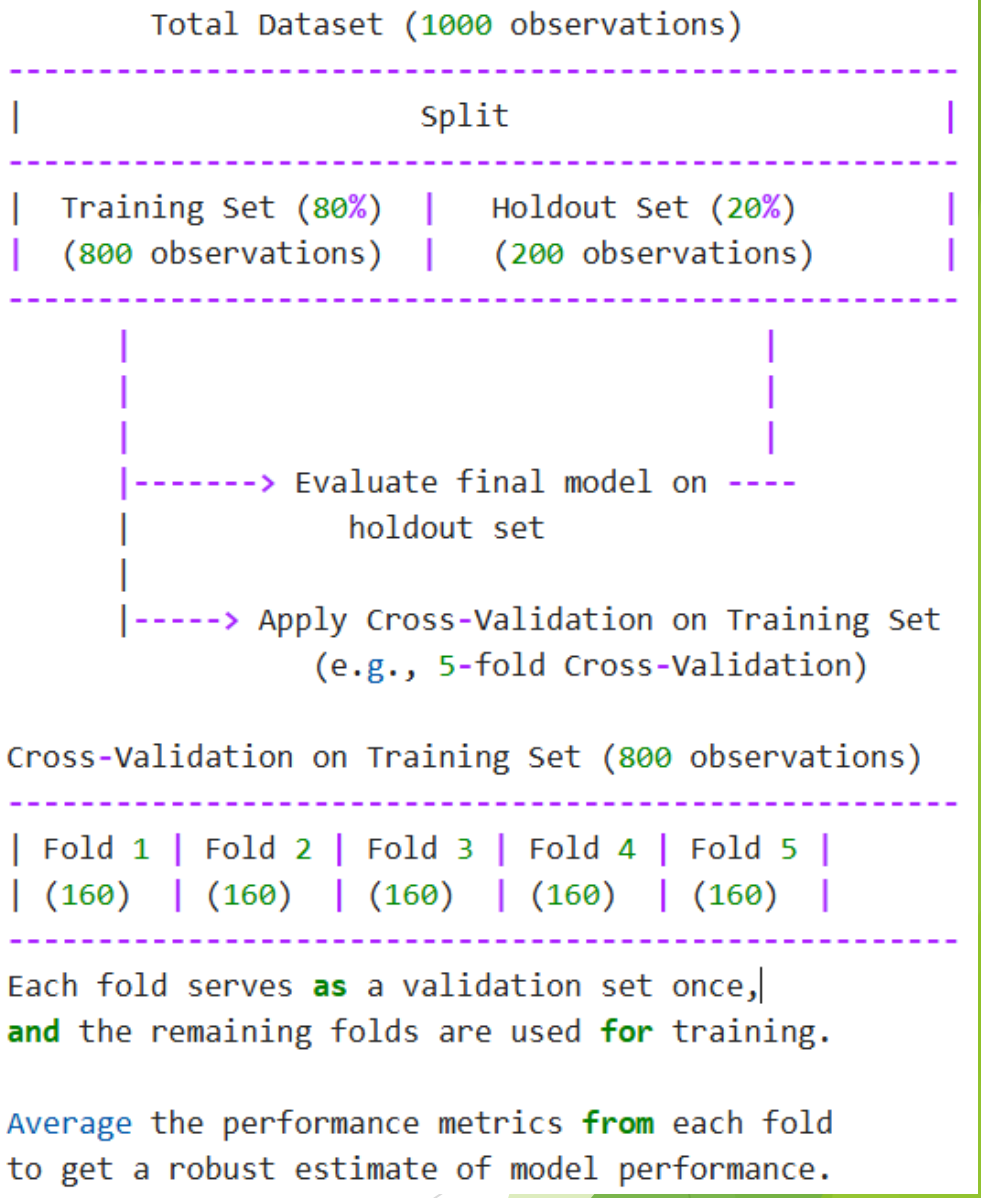
Cross-Validation

Cross-validation, particularly **k-fold cross-validation**, provides a more robust way to evaluate model performance and ensures that every observation in the dataset has a chance to be in both the training and testing set. Here's how it works:

1. **k-Fold Cross-Validation**:
   - The dataset is divided into k equal-sized folds (e.g., 5 or 10 folds).
   - The model is trained k times, each time using $k - 1$ folds for training and 1 fold for validation (testing).
   - Each fold serves as the validation set exactly once.
   - The performance metric is averaged across all k folds to get an overall performance estimate.
2. **Train-Validation Split within k-Folds**:
   - Each iteration uses a different subset (fold) of the data as the validation set.
   - The remaining $k - 1$ folds are combined to form the training set.
   - This approach mitigates the risk of the model performing well on just one specific train-test split by ensuring that every part of the dataset contributes to both training and validation.

Combining Cross-Validation with a Holdout Set

1. **Initial Holdout Set**:
   - First, split the dataset into two main parts: a training set and a final holdout (test) set.
   - The final holdout set is only used once at the end to evaluate the final model's performance.
2. **Cross-Validation on Training Set**:
   - Apply cross-validation on the training set to fine-tune the model and estimate its performance.
   - For example, if using 5-fold cross-validation, the training set is further divided into 5 folds for the cross-validation process.
3. **Final Model Training**:
   - After cross-validation, train the model on the entire training set using the best-found hyperparameters or model configurations.
4. **Final Evaluation on Holdout Set**:
   - Finally, evaluate the model on the holdout set to assess its predictive performance on unseen data.
   - This provides a realistic estimate of how the model will perform in real-world applications.



Initial Dataset Split

1. **Total Dataset**:
   - Assume you have 1000 observations.
2. **Train-Test Split**:
   - 80% (800 observations) used for training.
   - 20% (200 observations) used as a final holdout set for testing.

Cross-Validation within Training Data

3. **Cross-Validation on Training Set**:
   - The 800 training observations are split into k folds.
   - For example, with 5-fold cross-validation:
     - Each fold has 160 observations.
     - In each iteration, 640 observations are used for training, and 160 are used for validation.
   - This process repeats 5 times, with each fold serving as the validation set once.
4. **Model Building**:
   - The model is trained and validated multiple times during cross-validation.
   - Hyperparameters are tuned, and the best model configuration is identified based on cross validation performance.
5. **Final Model Training**:
   - After cross-validation, the final model is trained on all 800 training observations.
6. **Evaluation on Holdout Set**:
   - The final trained model is evaluated on the 200 holdout observations to estimate its generalization performance.

EXAMPLE

Solving for R-squared in Simple Linear Regression (SLR)

Given the data for experience (independent variable, X) and salary (dependent variable, Y), we can calculate the R^2 value which measures how well the regression line approximates the real data points.

Data Provided:

- **Experience (X)**: 2, 3, 4
- **Salary (Y)**: 10000, 13000, 14500
- **Average Salary ($\bar{Y}$)**: 12500

EXAMPLE:

SLR		
Given Data		
Experience	Salary	Avg Salary
2	10000	12500
3	13000	12500
4	14500	12500

Steps to Calculate R-squared (R^2):

1. **Fit the Regression Line:**
   - Find the slope (m) and intercept (b) for the regression line $Y = mX + b$.
2. **Calculate the Predicted Salaries ($\hat{Y}$):**
   - Use the regression equation to compute the predicted salaries for each experience value.
3. **Compute SST (Total Sum of Squares):**
   - $SST = \sum(Y_i - \bar{Y})^2$
   - This represents the total variance in the actual salaries.
4. **Compute SSE (Sum of Squared Errors):**
   - $SSE = \sum(Y_i - \hat{Y}_i)^2$
   - This represents the variance in the actual salaries that the model does not explain.
5. **Compute R-squared (R^2):**
   - $R^2 = 1 - \frac{SSE}{SST}$

Let's go through each step with the given data.

1. Fit the Regression Line

We use the least squares method to find the slope (m) and intercept (b) for the best-fit line Y $mX + b$.

Formulas for m and b:

$$m = \frac{n(\sum XY) - (\sum X)(\sum Y)}{n(\sum X^2) - (\sum X)^2}$$

$$b = \frac{(\sum Y)(\sum X^2) - (\sum X)(\sum XY)}{n(\sum X^2) - (\sum X)^2}$$

Given:

- $n = 3$
- $\sum X = 2 + 3 + 4 = 9$
- $\sum Y = 10000 + 13000 + 14500 = 37500$
- $\sum XY = (2 \times 10000) + (3 \times 13000) + (4 \times 14500) = 95500$
- $\sum X^2 = 2^2 + 3^2 + 4^2 = 29$

Let's calculate m and b:

$$m = \frac{3(95500) - (9)(37500)}{3(29) - (9)^2}$$

$$b = \frac{(37500)(29) - (9)(95500)}{3(29) - (9)^2}$$

2. Calculate Predicted Salaries ($\hat{Y}$)

Using the regression equation $Y = mX + b$, we compute the predicted salaries for each experience value.

3. Compute SST

$$SST = \sum(Y_i - \bar{Y})^2$$

Given $\bar{Y} = 12500$:

$$SST = (10000 - 12500)^2 + (13000 - 12500)^2 + (14500 - 12500)^2$$

4. Compute SSE

$$SSE = \sum(Y_i - \hat{Y}_i)^2$$

Where $\hat{Y}_i$ are the predicted salaries obtained from the regression equation.

5. Compute R-squared

$$R^2 = 1 - \frac{SSE}{SST}$$

Calculation of R-squared for the Given Data

Based on the provided data, here are the detailed steps and results:

Data:

- **Experience (X)**: 2, 3, 4
- **Salary (Y)**: 10000, 13000, 14500
- **Average Salary ($\bar{Y}$)**: 12500

Steps and Results:

1. **Fit the Regression Line:**

   Using the least squares method, we found the slope (m) and intercept (b) of the regression line $Y = mX + b$.

   - **Slope (m)**: 2250.0
   - **Intercept (b)**: 5750.0

   The regression equation is:

   $$\hat{Y} = 2250X + 5750$$

2. **Calculate the Predicted Salaries ($\hat{Y}$):**

   Using the regression equation, the predicted salaries for each experience value are:

   $$\hat{Y}_1 = 2250(2) + 5750 = 10250$$
   $$\hat{Y}_2 = 2250(3) + 5750 = 12500$$
   $$\hat{Y}_3 = 2250(4) + 5750 = 14750$$

   Thus, the predicted salaries ($\hat{Y}$) are:

   $$\hat{Y} = [10250, 12500, 14750]$$

3. **Compute SST (Total Sum of Squares):**

   The total variance in the actual salaries from the mean salary:

   $$SST = \sum(Y_i - \bar{Y})^2$$
   $$SST = (10000 - 12500)^2 + (13000 - 12500)^2 + (14500 - 12500)^2$$
   $$SST = 6250000 + 250000 + 4000000 = 10500000$$

4. **Compute SSE (Sum of Squared Errors):**

   The variance in the actual salaries that the model does not explain:

   $$SSE = \sum(Y_i - \hat{Y}_i)^2$$
   $$SSE = (10000 - 10250)^2 + (13000 - 12500)^2 + (14500 - 14750)^2$$
   $$SSE = 62500 + 250000 + 62500 = 375000$$

5. **Compute R-squared (R^2):**

   The proportion of the variance in the dependent variable that is predictable from the independent variable(s):

   $$R^2 = 1 - \frac{SSE}{SST}$$
   $$R^2 = 1 - \frac{375000}{10500000} = 0.9643$$

- **R-squared (R2): ≈0.964**
- **This R2 value indicates that approximately 96.43% of the variance in salary can be explained by the linear relationship with experience in this model.**

```python
import numpy as np
import pandas as pd
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score

# Given Data
X = np.array([2, 3, 4]).reshape(-1, 1)
Y = np.array([10000, 13000, 14500])

# Simple Linear Regression Model
model = LinearRegression()
model.fit(X, Y)
predictions = model.predict(X)

# Calculate R-squared
r2 = r2_score(Y, predictions)
print(f"R-squared for the provided data: {r2}")

# Calculate SST and SSE
Y_mean = np.mean(Y)
SST = np.sum((Y - Y_mean) ** 2)
SSE = np.sum((Y - predictions) ** 2)

print(f"SST (Total Sum of Squares): {SST}")
print(f"SSE (Sum of Squared Errors): {SSE}")
print(f"R-squared (Calculated): {1 - SSE / SST}")
```

```
R-squared for the provided data: 0.9642857142857143
SST (Total Sum of Squares): 10500000.0
SSE (Sum of Squared Errors): 375000.0
R-squared (Calculated): 0.9642857142857143
```

Introduction

The real estate market is influenced by various factors, including income levels, house age, number of rooms, number of bedrooms, and population density. Understanding how these factors affect house prices can provide valuable insights for buyers, sellers, and real estate professionals. In this project, we aim to develop a predictive model to estimate house prices based on various features in the USAHousing dataset.

Dataset Description

The USAHousing dataset contains information on various attributes related to houses in different areas. The features included in the dataset are:

- **Avg. Area Income:** The average income of residents in the area.
- **Avg. Area House Age:** The average age of houses in the area.
- **Avg. Area Number of Rooms:** The average number of rooms in houses in the area.
- **Avg. Area Number of Bedrooms:** The average number of bedrooms in houses in the area.
- **Area Population:** The population of the area.
- **Price:** The price of the house.
- **Address:** The address of the house (considered as a non-significant variable and will be excluded from the model).

Case Study: USAHOUSING PRICE PREDICTION

Objective

- The primary objective of this project is to build a robust predictive model that can accurately estimate the price of a house based on the following independent variables:

1. Avg. Area Income
2. Avg. Area House Age
3. Avg. Area Number of Rooms
4. Avg. Area Number of Bedrooms
5. Area Population

Methodology

1. **Data Preprocessing:**
    - Remove non-significant variables such as 'Address'.
    - Handle any missing or inconsistent data.
    - Normalize or scale features if necessary.
2. **Exploratory Data Analysis (EDA):**
    - Visualize the distribution of each feature.
    - Analyze the relationship between independent variables and the target variable (Price).
3. **Model Development:**
    - Split the dataset into training and testing sets.
    - Train multiple regression models including:
        - Linear Regression
        - Ordinary Least Squares (OLS)
        - Lasso Regression
        - Ridge Regression
        - Stochastic Gradient Descent (SGD) Regression
    - Evaluate the performance of each model using appropriate metrics (e.g., R-squared, Mean Squared Error).

METHODOLOGY

4. **Model Evaluation and Selection:**
    - Compare the performance of different models.
    - Select the best-performing model based on evaluation metrics.
5. **Model Deployment:**
    - Save the trained model for future predictions.
    - Create a user interface or an API for making predictions on new data.

Expected Outcome

By the end of this project, we expect to have a predictive model that can accurately estimate house prices based on the provided features. This model can be used by real estate professionals and potential buyers to make informed decisions.

Challenges and Considerations

- Ensuring the dataset is clean and free of errors.
- Selecting the right features and avoiding multicollinearity.
- Choosing the best model that generalizes well to new data.
- Interpreting the results and understanding the limitations of the model.

Conclusion

- Predicting house prices is a complex task that involves understanding various factors that influence the real estate market.
- By leveraging machine learning techniques, we aim to build a reliable model that can provide accurate price estimates and valuable insights into the housing market.

--> Import library

[1]:
```
import os # where the path is
import numpy as np # mathematical calculation
import pandas as pd # data manipulation
import matplotlib.pyplot as plt # data visuliasation
import seaborn as sns # data visuliasation
sns.set
%matplotlib inline
import warnings
warnings.filterwarnings("ignore")
```

--> Import data set

[2]:
```
# by default controlling height and width of plot
# by default controlling height and width of plot
```

[3]:
```
USAHOUSING = pd.read_csv('USA_Housing.csv')
```

--> TO VIEW THE DATA SET

[4]:
```
USAHOUSING.head()
```

[4]:

	Avg. Area Income	Avg. Area House Age	Avg. Area Number of Rooms	Avg. Area Number of Bedrooms	Area Population	Price	Address
0	79545.45857	5.682861	7.009188	4.09	23086.80050	1.059034e+06	208 Michael Ferry Apt. 674\nLaurabury, NE 3701...
1	79248.64245	6.002900	6.730821	3.09	40173.07217	1.505891e+06	188 Johnson Views Suite 079\nLake Kathleen, CA...
2	61287.06718	5.865890	8.512727	5.13	36882.15940	1.058988e+06	9127 Elizabeth Stravenue\nDanieltown, WI 06482...
3	63345.24005	7.188236	5.586729	3.26	34310.24283	1.260617e+06	USS Barnett\nFPO AP 44820
4	59982.19723	5.040555	7.839388	4.23	26354.10947	6.309435e+05	USNS Raymond\nFPO AE 09386

--->To get the name of all columns

[5]:
```
USAHOUSING.columns
```

[5]:
```
Index(['Avg. Area Income', 'Avg. Area House Age', 'Avg. Area Number of Rooms',
       'Avg. Area Number of Bedrooms', 'Area Population', 'Price', 'Address'],
      dtype='object')
```

--->There is an ambiguty in representation of name i.e. '.' this may cause problem. So we replace it with ''

--->Similary you can do for other issues in names.

[6]:
```
USAHOUSING.columns=USAHOUSING.columns.str.replace(' ','_')
```

[7]:
```
USAHOUSING.columns
```

[7]:
```
Index(['Avg._Area_Income', 'Avg._Area_House_Age', 'Avg._Area_Number_of_Rooms',
       'Avg._Area_Number_of_Bedrooms', 'Area_Population', 'Price', 'Address'],
      dtype='object')
```

[8]:
```
USAHOUSING.columns=USAHOUSING.columns.str.replace('.', '')
```

[9]:
```
USAHOUSING.columns
```

[9]:
```
Index(['Avg_Area_Income', 'Avg_Area_House_Age', 'Avg_Area_Number_of_Rooms',
       'Avg_Area_Number_of_Bedrooms', 'Area_Population', 'Price', 'Address'],
      dtype='object')
```

---> To find information about data like number of column , number of rows, number of null, type od data.

[10]:
```
USAHOUSING.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 7 columns):
 #   Column                       Non-Null Count  Dtype  
---  ------                       --------------  -----  
 0   Avg_Area_Income              4990 non-null   float64
 1   Avg_Area_House_Age           5000 non-null   float64
 2   Avg_Area_Number_of_Rooms     4995 non-null   float64
 3   Avg_Area_Number_of_Bedrooms  4994 non-null   float64
 4   Area_Population              5000 non-null   float64
 5   Price                        5000 non-null   float64
 6   Address                      5000 non-null   object 
dtypes: float64(6), object(1)
memory usage: 273.6+ KB
```

---> To find unique values of each variable use this for loop on USAHOUSING.columns

[11]:
```
for i in USAHOUSING.columns:
    print("**************************************************************",i,"**************************************************************")
    print()
    print(set(USAHOUSING[i].tolist()))
```

----> ENCODING CONCEPT:

---> LABEL ENCODING :

- Label encoding is a technique used to convert categorical data into numerical data, which is often necessary for machine learning models that can only handle numerical inputs.

- Convert Address type from object to category than into integer

- Categorical Data Type: This is a special data type in pandas which is useful for columns that have a limited number of unique values

-It helps to save memory and can make certain operations more efficient.

-The cat.codes attribute of a categorical column returns the numeric codes corresponding to each category.

-Each unique category in the 'Address' column is assigned a unique integer code.

----> ONE HOT ENCODING:

- One-hot encoding is another technique used to convert categorical data into a format that can be provided to ML algorithms.

- It improves predictions.

- One-hot encoding creates a new binary (0,1) column for each category.

- However it increases number of columns

- It is in practie that after one hot encoding we must drop one column from data set.

- Here I applied both method for 'ADDRESS'column for your information.

- But commented one hot encoding, you can check its behaviour.

[12]:
```
# LABEL ENCODING
USAHOUSING['Address'] = USAHOUSING['Address'].astype('category')
USAHOUSING['Address'] = USAHOUSING['Address'].cat.codes
```

[13]:
```
# ONE HOT ENcoding
#USAHOUSING = pd.get_dummies(USAHOUSING,columns = ['Address'],prefix='Address')
```

[14]:
```
USAHOUSING['Address']  # here you can see all values change to integer.
```

[14]:
```
0        962
1        863
2       4069
3       4794
4       4736
        ...
4995    4750
4996    4636
4997    1897
4998    4833
4999    1703
Name: Address, Length: 5000, dtype: int16
```

[15]:
```
USAHOUSING.info() # Also data type change to int earlier it was object.
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5000 entries, 0 to 4999
Data columns (total 7 columns):
 #   Column                       Non-Null Count  Dtype  
---  ------                       --------------  -----  
 0   Avg_Area_Income              4990 non-null   float64
 1   Avg_Area_House_Age           5000 non-null   float64
 2   Avg_Area_Number_of_Rooms     4995 non-null   float64
 3   Avg_Area_Number_of_Bedrooms  4994 non-null   float64
 4   Area_Population              5000 non-null   float64
 5   Price                        5000 non-null   float64
 6   Address                      5000 non-null   int16  
dtypes: float64(6), int16(1)
memory usage: 244.3 KB
```

---> ANOVA TESTING TO CHECK SIGNIFICANT VARIABLE as it gives 'P'value whose value gives us idea about the relvance of that variable.

---> Since Address have p value > 5% ,So we can drop it as it is less significant.

[16]:
```
import statsmodels.api as sm
from statsmodels.formula.api import ols
model = ols('Price ~ Address', data = USAHOUSING).fit()
anova_result = sm.stats.anova_lm(model, typ=2)
print(anova_result)
```

```
                sum_sq      df         F    PR(>F)
Address   4.729103e+10     1.0  0.379215  0.538051
Residual  6.232883e+14  4998.0       NaN       NaN
```

[17]:
```
import statsmodels.api as sm
from statsmodels.formula.api import ols
model = ols('Price ~ Avg_Area_Income', data = USAHOUSING).fit()
anova_result = sm.stats.anova_lm(model, typ=2)
print(anova_result)
```

```
                     sum_sq      df            F  PR(>F)
Avg_Area_Income  2.546601e+14     1.0  3455.920442     0.0
Residual         3.675561e+14  4988.0          NaN     NaN
```

[18]:
```
import statsmodels.api as sm
from statsmodels.formula.api import ols
model = ols('Price ~ Area_Population', data = USAHOUSING).fit()
anova_result = sm.stats.anova_lm(model, typ=2)
print(anova_result)
```

```
                     sum_sq      df            F         PR(>F)
Area_Population  1.040459e+14     1.0  1001.408749  1.736392e-200
Residual         5.192897e+14  4998.0          NaN            NaN
```

[19]:
```
import statsmodels.api as sm
from statsmodels.formula.api import ols
model = ols('Price ~ Avg_Area_House_Age', data = USAHOUSING).fit()
anova_result = sm.stats.anova_lm(model, typ=2)
print(anova_result)
```

```
                       sum_sq      df            F         PR(>F)
Avg_Area_House_Age  1.276559e+14     1.0  1287.169756  4.944750e-251
Residual            4.956797e+14  4998.0          NaN            NaN
```

----> To get statistical information like number of counts, mean value, standard deviation, minimum, maxmimum,median(50%) etc use describe

[20]:
```
USAHOUSING.describe()
```

[20]:

	Avg_Area_Income	Avg_Area_House_Age	Avg_Area_Number_of_Rooms	Avg_Area_Number_of_Bedrooms	Area_Population	Price	Address
count	4990.000000	5000.000000	4995.000000	4994.000000	5000.000000	5.000000e+03	5000.000000
mean	68584.719991	5.977222	6.987693	3.981874	36163.516039	1.232073e+06	2499.500000
std	10651.192423	0.991456	1.005938	1.234497	9925.650114	3.531176e+05	1443.520003
min	17796.631190	2.644304	3.236194	2.000000	172.610686	1.593866e+04	0.000000
25%	61481.465105	5.322283	6.299156	3.140000	29403.928700	9.975771e+05	1249.750000
50%	68797.671885	5.970429	7.002940	4.050000	36199.406690	1.232669e+06	2499.500000
75%	75779.145465	6.650808	7.665622	4.490000	42861.290770	1.471210e+06	3749.250000
max	107701.748400	9.519088	10.759588	6.500000	69621.713380	2.469066e+06	4999.000000

To find information about missing Value

[21]:
```
USAHOUSING.isnull().sum()
```

[21]:
```
Avg_Area_Income               10
Avg_Area_House_Age             0
Avg_Area_Number_of_Rooms       5
Avg_Area_Number_of_Bedrooms    6
Area_Population                0
Price                          0
Address                        0
dtype: int64
```

[22]:
```
USAHOUSING.isnull().sum()/len(USAHOUSING)*100
```

[22]:
```
Avg_Area_Income               0.20
Avg_Area_House_Age            0.00
Avg_Area_Number_of_Rooms      0.10
Avg_Area_Number_of_Bedrooms   0.12
Area_Population               0.00
Price                         0.00
Address                       0.00
dtype: float64
```

----> To remove or replace abnormal input like(/,?,.,etc.) if any , than we can do:

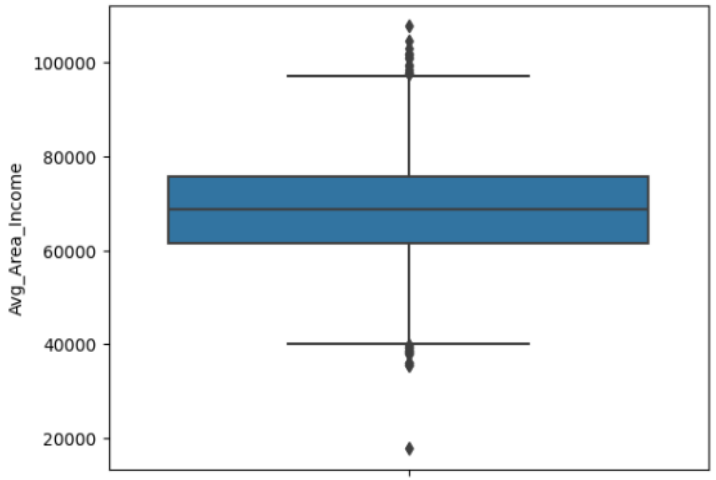
df = df.replace('?', np.nan)

df = df.apply(lambda x : x.fillna(x.median()),axis = 0)

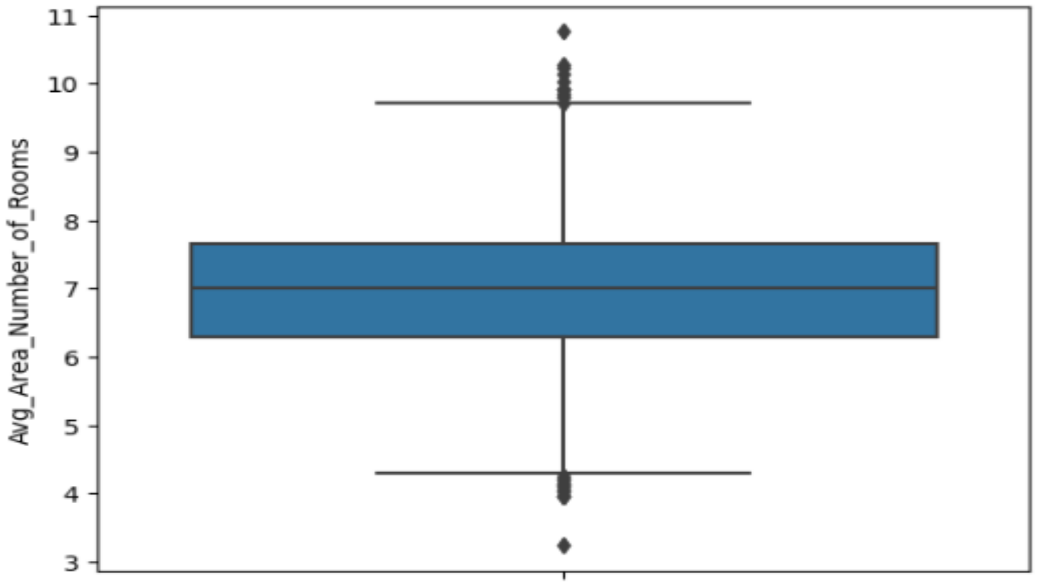
----> CHECK OUTLIER and decide what to use either mean or median to impute missing value

-If there are outliers use median else go with mean

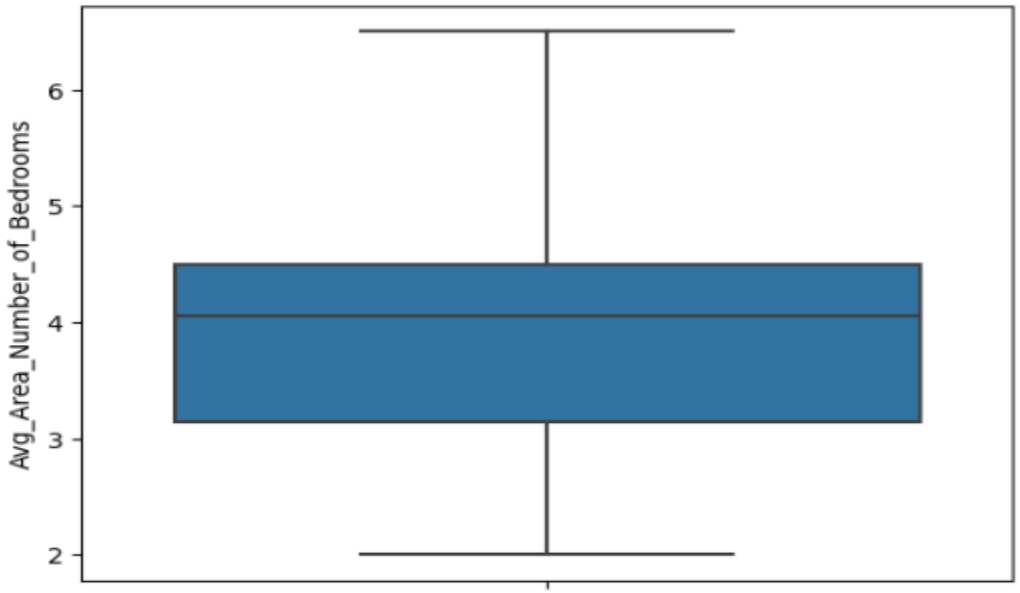
23]:
```
sns.boxplot(y = 'Avg_Area_Income' , data = USAHOUSING)
plt.show()
```



[24]:
```
sns.boxplot(y = 'Avg_Area_Number_of_Rooms' , data = USAHOUSING)
plt.show()
```



[25]:
```
sns.boxplot(y = 'Avg_Area_Number_of_Bedrooms' , data = USAHOUSING)
plt.show()
```



----> filling of missing value.

[26]:
```
USAHOUSING['Avg_Area_Income'] = USAHOUSING['Avg_Area_Income'].fillna( USAHOUSING['Avg_Area_Income'].median())
```

[27]:
```
USAHOUSING['Avg_Area_Number_of_Rooms'] = USAHOUSING['Avg_Area_Number_of_Rooms'].fillna( USAHOUSING['Avg_Area_Number_of_Rooms'].median())
```

[28]:
```
USAHOUSING['Avg_Area_Number_of_Bedrooms'] = USAHOUSING['Avg_Area_Number_of_Bedrooms'].fillna( USAHOUSING['Avg_Area_Number_of_Bedrooms'].median())
```

[29]:
```
USAHOUSING.isnull().sum()
```

[29]:
```
Avg_Area_Income               0
Avg_Area_House_Age            0
Avg_Area_Number_of_Rooms      0
Avg_Area_Number_of_Bedrooms   0
Area_Population               0
Price                         0
Address                       0
dtype: int64
```

[30]:
```
USAHOUSING.head()
```

[30]:

	Avg_Area_Income	Avg_Area_House_Age	Avg_Area_Number_of_Rooms	Avg_Area_Number_of_Bedrooms	Area_Population	Price	Address
0	79545.45857	5.682861	7.009188	4.09	23086.80050	1.059034e+06	962
1	79248.64245	6.002900	6.730821	3.09	40173.07217	1.505891e+06	863
2	61287.06718	5.865890	8.512727	5.13	36882.15940	1.058988e+06	4069
3	63345.24005	7.188236	5.586729	3.26	34310.24283	1.260617e+06	4794
4	59982.19723	5.040555	7.839388	4.23	26354.10947	6.309435e+05	4736

----> Address is non significant variable in this data so drop it.

---->If column is not at last position than we can use following code. ¶

USAHOUSING = USAHOUSING.drop(columns=['Address'] or USAHOUSING = USAHOUSING.loc[:, USAHOUSING.columns != 'Address')

[31]:
```
USAHOUSING = USAHOUSING.iloc[:,0:-1]   # -1 means last column will no be consider.
```

[32]:
```
USAHOUSING.head()
```

[32]:

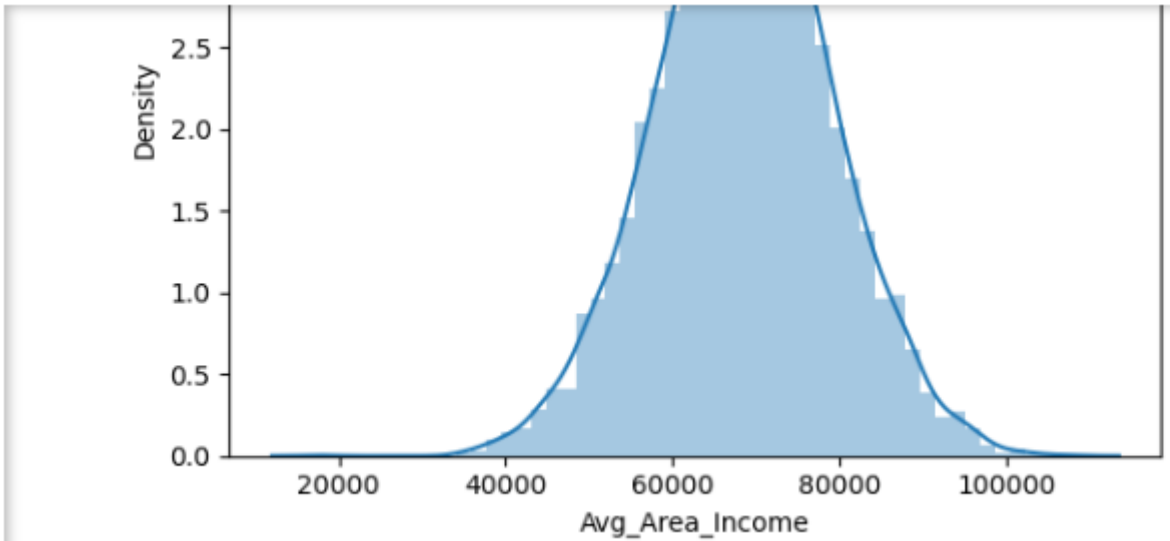
	Avg_Area_Income	Avg_Area_House_Age	Avg_Area_Number_of_Rooms	Avg_Area_Number_of_Bedrooms	Area_Population	Price
0	79545.45857	5.682861	7.009188	4.09	23086.80050	1.059034e+06
1	79248.64245	6.002900	6.730821	3.09	40173.07217	1.505891e+06
2	61287.06718	5.865890	8.512727	5.13	36882.15940	1.058988e+06
3	63345.24005	7.188236	5.586729	3.26	34310.24283	1.260617e+06
4	59982.19723	5.040555	7.839388	4.23	26354.10947	6.309435e+05

----> Distribution plots to check whether data is normally distributed or not.

----> no of bed room have non normal data.

[33]:
```
def distplots(col):
    sns.distplot(USAHOUSING[col])
    plt.show()
```

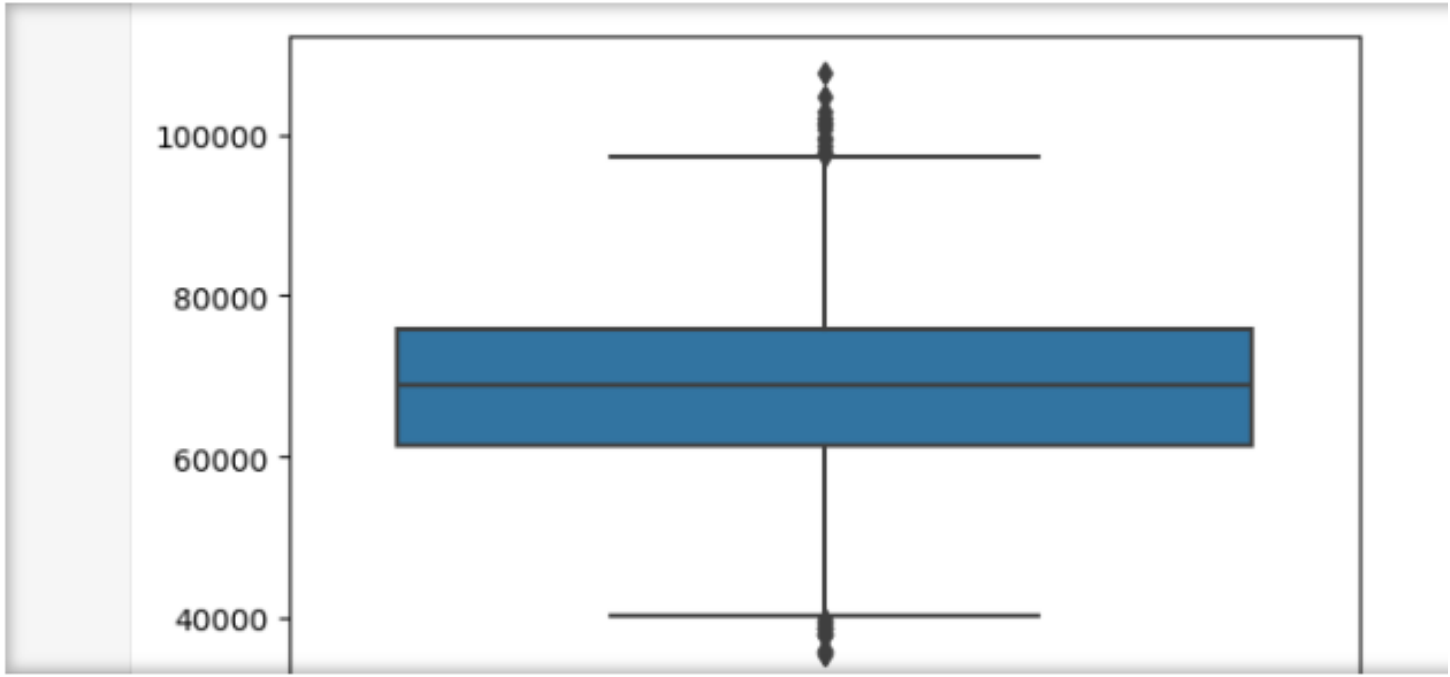
[34]:
```
for i in list(USAHOUSING.columns)[0:]:
    distplots(i)
```



----> To view the outliers we use box plot

[35]:
```
def boxplots(col):
    sns.boxplot(USAHOUSING[col])
    plt.xlabel(col)
    plt.show()
```

[36]:
```
for i in list(USAHOUSING.select_dtypes(exclude =['object']).columns)[0:]:
    boxplots(i)
```



-----> Don't touch the dependent variable for outliers

----> find Q1, ,Q3 ,IQR, Upper Limit,Lower limit to handle outliers

[37]:
```
Q1 = USAHOUSING.quantile(0.25)
Q3 = USAHOUSING.quantile(0.75)
IQR = Q3-Q1
Upper_Limit = Q3+1.5*IQR
Lower_Limit = Q1-1.5*IQR
```

[38]:
```
print('Q1')
print(Q1)
print('******'*5)
print('Q3')
print(Q3)
print('******'*5)
print('IQR')
print(IQR)
print('******'*5)
print('Upper_Limit')
print(Upper_Limit)
print('******'*5)
print('Lower_Limit')
print(Lower_Limit)
```

```
Q1
Avg_Area_Income                 61485.150192
Avg_Area_House_Age                  5.322283
Avg_Area_Number_of_Rooms            6.299692
Avg_Area_Number_of_Bedrooms         3.140000
Area_Population                 29403.928700
Price                          997577.135075
Name: 0.25, dtype: float64
******************************
Q3
Avg_Area_Income                7.576652e+04
Avg_Area_House_Age             6.650808e+00
Avg_Area_Number_of_Rooms       7.665281e+00
Avg_Area_Number_of_Bedrooms    4.490000e+00
Area_Population                4.286129e+04
Price                          1.471210e+06
Name: 0.75, dtype: float64
******************************
IQR
Avg_Area_Income                 14281.368910
Avg_Area_House_Age                  1.328525
Avg_Area_Number_of_Rooms            1.365589
Avg_Area_Number_of_Bedrooms         1.350000
Area_Population                 13457.362070
Price                          473633.069425
dtype: float64
******************************
Upper_Limit
Avg_Area_Income                9.718857e+04
Avg_Area_House_Age             8.643597e+00
Avg_Area_Number_of_Rooms       9.713664e+00
Avg_Area_Number_of_Bedrooms    6.515000e+00
Area_Population                6.304733e+04
Price                          2.181660e+06
dtype: float64
******************************
Lower_Limit
Avg_Area_Income                 40063.096827
Avg_Area_House_Age                  3.329495
Avg_Area_Number_of_Rooms            4.251308
Avg_Area_Number_of_Bedrooms         1.115000
Area_Population                  9217.885595
Price                          287127.530937
dtype: float64
```

-----> Create copy of your dataset

[39]:
```
New_USAHOUSING = USAHOUSING.copy()
```

[40]:
```
New_USAHOUSING.columns
```

[40]:
```
Index(['Avg_Area_Income', 'Avg_Area_House_Age', 'Avg_Area_Number_of_Rooms',
       'Avg_Area_Number_of_Bedrooms', 'Area_Population', 'Price'],
      dtype='object')
```

-----> USE CAPPING MEHTOD TO HANDLE OUTLIER IN

Avg. Area Income',

'Avg. Area House Age',

'Avg. Area Number of Rooms',

'Avg. Area Number of Bedrooms',

'Area Population'

[41]:
```
Income_Q1 = New_USAHOUSING['Avg_Area_Income'].quantile(0.25)
Income_Q3 = New_USAHOUSING['Avg_Area_Income'].quantile(0.75)
Income_IQR = Income_Q3 - Income_Q1
Income_Upper =Income_Q3 + Income_IQR*1.5
Income_Lower = Income_Q1 - Income_IQR*1.5
```

[42]:
```
New_USAHOUSING['Avg_Area_Income'] = np.where(New_USAHOUSING['Avg_Area_Income']>Income_Upper,Income_Upper,
                                             np.where(New_USAHOUSING['Avg_Area_Income']<Income_Lower,Income_Lower,
                                                      New_USAHOUSING['Avg_Area_Income']))
```

[43]:
```
Age_Q1 = New_USAHOUSING['Avg_Area_House_Age'].quantile(0.25)
Age_Q3 = New_USAHOUSING['Avg_Area_House_Age'].quantile(0.75)
Age_IQR = Age_Q3 - Age_Q1
Age_Upper =Age_Q3 + Age_IQR*1.5
Age_Lower = Age_Q1 - Age_IQR*1.5
```

[44]:
```
New_USAHOUSING['Avg_Area_House_Age'] = np.where(New_USAHOUSING['Avg_Area_House_Age']>Age_Upper,Age_Upper,
                                                np.where(New_USAHOUSING['Avg_Area_House_Age']<Age_Lower,Age_Lower,
                                                         New_USAHOUSING['Avg_Area_House_Age']))
```

[45]:
```
Number_of_Rooms_Q1 = New_USAHOUSING['Avg_Area_Number_of_Rooms'].quantile(0.25)
Number_of_Rooms_Q3 = New_USAHOUSING['Avg_Area_Number_of_Rooms'].quantile(0.75)
Number_of_Rooms_IQR = Number_of_Rooms_Q3 - Number_of_Rooms_Q1
Number_of_Rooms_Upper =Number_of_Rooms_Q3 + Number_of_Rooms_IQR*1.5
Number_of_Rooms_Lower = Number_of_Rooms_Q1 - Number_of_Rooms_IQR*1.5
```

[46]:
```
New_USAHOUSING['Avg_Area_Number_of_Rooms'] = np.where(New_USAHOUSING['Avg_Area_Number_of_Rooms']>Number_of_Rooms_Upper,Number_of_Rooms_Upper,
                                        np.where(New_USAHOUSING['Avg_Area_Number_of_Rooms']<Number_of_Rooms_Lower ,Number_of_Rooms_Lower ,
                                                 New_USAHOUSING['Avg_Area_Number_of_Rooms']))
```

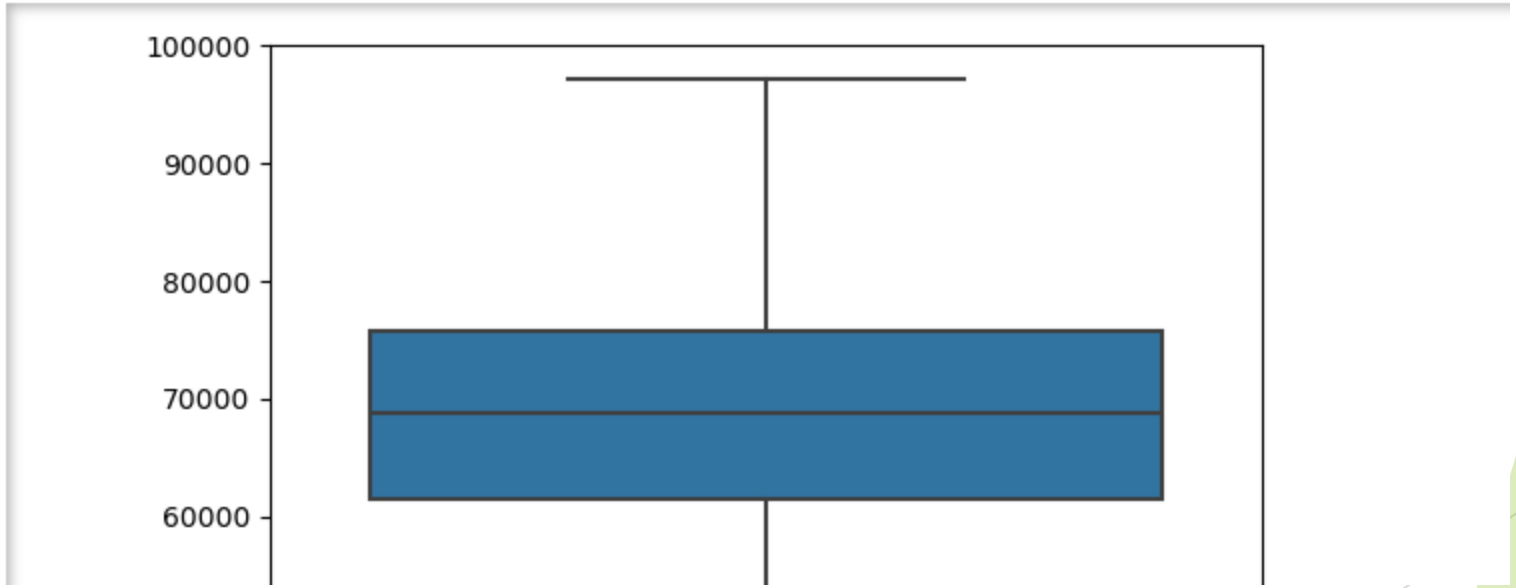
[47]:
```
Population_Q1 = New_USAHOUSING['Area_Population'].quantile(0.25)
Population_Q3 = New_USAHOUSING['Area_Population'].quantile(0.75)
Population_IQR = Population_Q3 - Population_Q1
Population_Upper =Population_Q3 + Population_IQR*1.5
Population_Lower = Population_Q1 - Population_IQR*1.5
```

[48]:
```
New_USAHOUSING['Area_Population'] = np.where(New_USAHOUSING['Area_Population']>Population_Upper,Population_Upper,
                                        np.where(New_USAHOUSING['Area_Population']<Population_Lower ,Population_Lower ,
                                                 New_USAHOUSING['Area_Population']))
```

-----> Again check the outliers

[49]:
```
def boxplots(col):
    sns.boxplot(New_USAHOUSING[col])
    plt.xlabel(col)
    plt.show()

for i in list(New_USAHOUSING.select_dtypes(exclude =['object']).columns)[0:]:
    boxplots(i)
```



---->>> FEATURE SCALING: Very important while using graident descent approach how ever give poor result with OLS approach.

----> Distance-based algorithms (e.g., KNN, SVM) perform better with scaled features because they compute distances based on feature values.

----> If the features are on different scales, the algorithm might bias towards features with larger scales.

-Feature scaling is a crucial preprocessing step in machine learning that transforms the features of your data so that they are on a similar scale.

---->This can improve the performance and convergence speed of many algorithms.

----->ONLY WITH INDEPENDENT VARIABLE

----->FIRST SPLIT DATA INTO DEPENDENT AND INDEPENDENT VARIABLE

[50]:
```
x = New_USAHOUSING.iloc[:,0:-1]
y = New_USAHOUSING['Price']
```

----->>>> Another approach to split

-----> x = New_USAHOUSING.drop('Price',axis = 1)

------> y = New_USAHOUSING[['Price']]

[51]:
```
x.head()
```

[51]:

	Avg_Area_Income	Avg_Area_House_Age	Avg_Area_Number_of_Rooms	Avg_Area_Number_of_Bedrooms	Area_Population
0	79545.45857	5.682861	7.009188	4.09	23086.80050
1	79248.64245	6.002900	6.730821	3.09	40173.07217
2	61287.06718	5.865890	8.512727	5.13	36882.15940
3	63345.24005	7.188236	5.586729	3.26	34310.24283
4	59982.19723	5.040555	7.839388	4.23	26354.10947

[52]:
```
y.head()
```

[52]:
```
0    1.059034e+06
1    1.505891e+06
2    1.058988e+06
3    1.260617e+06
4    6.309435e+05
Name: Price, dtype: float64
```

------> Understanding fit(), transform(), and fit_transform()

-----> fit(X):

This method computes the parameters (such as mean and standard deviation in the case of standard scaling) required for the transformation. For example, if you are using StandardScaler from scikit-learn, fit() calculates the mean and standard deviation of the dataset X.

-----> transform(X):

This method applies the transformation using the parameters computed by fit(). For StandardScaler, transform() uses the mean and standard deviation computed during fit() to standardize the dataset X.

-----> fit_transform(X):

This method combines the two steps. It first computes the parameters using fit(), and then applies the transformation using transform(). It is a convenient way to perform both operations in one step.

[53]:
```
from sklearn.preprocessing import StandardScaler
sc = StandardScaler()
sc_x = sc.fit_transform(x)
pd.DataFrame(sc_x)
```

[53]:

	0	1	2	3	4
0	1.036382	-0.298541	0.021620	0.087582	-1.325622
1	1.008309	0.025747	-0.256381	-0.723031	0.407049
2	-0.690457	-0.113082	1.523179	0.930619	0.073326
3	-0.495800	1.226822	-1.398967	-0.585227	-0.187484
4	-0.813869	-0.949376	0.850726	0.201068	-0.994293
...	...	...	...	...	...
4995	-0.758470	1.877474	-0.849064	-0.423104	-1.350917
4996	0.936679	1.035210	-0.410236	0.030839	-1.069131

-----> Find Correlation to check multicolinearity.

-----> if value is greater than 0.75 or 0.8, drop than column. As it is highly corelated

-----> If any of two independent variables have same value than drop any one with discussion by SME. As it leads to multicolinearit

-----> In classification problem no need to check multicolinearity.

[54]:
```
import matplotlib.pyplot as plt
import seaborn as sns

plt.figure(figsize=(20, 15))  # Set the figure size
corr = New_USAHOUSING.corr()  # Calculate the correlation matrix
sns.heatmap(corr, annot=True, cmap='coolwarm')  # Create a heatmap with annotations and a 'coolwarm' color map
plt.xticks(rotation=65)  # Rotate x-axis labels by 65 degrees
plt.show()  # Display the plot
```

-----> Variance_Inflation_Factor(VIF) is another approach to check multicolinearity other than heat map

-----> VIF- Variance Inflation Factor to check multicolinearity

------> VIF should be <= 5

------> VIF = 1/(1-R*2)

[55]:
```
variable = sc_x
variable.shape
```

[55]:
```
(5000, 5)
```

[56]:
```
from statsmodels.stats.outliers_influence import variance_inflation_factor
variable = sc_x
vif = pd.DataFrame()
vif['Variance_Inflation_Factor']=[variance_inflation_factor(variable,i) for i in range(variable.shape[1])]
vif['Features'] = x.columns
```

[57]:
```
vif
```

[57]:

	Variance_Inflation_Factor	Features
0	1.001067	Avg_Area_Income
1	1.000593	Avg_Area_House_Age
2	1.274864	Avg_Area_Number_of_Rooms
3	1.275727	Avg_Area_Number_of_Bedrooms
4	1.001144	Area_Population

----->>>> Since for any variable VIF is less than 5 so there is no multicolinearity. So one Assumption satisified.

-----> SPLIT THE DATA INTO TRAINING AND TEST for model building and prediction

[58]:
```
from sklearn.model_selection import train_test_split
x_train,x_test,y_train,y_test = train_test_split(x,y,test_size = 0.25,random_state = 101)
print(x_train.shape,x_test.shape,y_train.shape,y_test.shape)
```

```
(3750, 5) (1250, 5) (3750,) (1250,)
```

------> BUILD THE MODEL

-----> APPROACH 1 : Linear Regression

[59]:
```
from sklearn.linear_model import LinearRegression
lm = LinearRegression()
lm.fit(x_train,y_train)
```

[59]:
```
▾ LinearRegression
LinearRegression()
```

[60]:
```
print(lm.intercept_)
print('*****************')
print(lm.coef_)
```

```
-2657921.44640602
*****************
[2.17329557e+01 1.65690136e+05 1.21587070e+05 1.72620828e+03
 1.53020327e+01]
```

-----> We can also use following approach to find coefficient

[61]:
```
for idx, column in enumerate(x_train.columns):
    print("The coefficient for {} is {}".format(column, lm.coef_[idx]))
```

```
The coefficient for Avg_Area_Income is 21.732955742400346
The coefficient for Avg_Area_House_Age is 165690.1364862407
The coefficient for Avg_Area_Number_of_Rooms is 121587.06965741176
The coefficient for Avg_Area_Number_of_Bedrooms is 1726.2082822261352
The coefficient for Area_Population is 15.302032682183732
```

```
[62]: x.columns
```

```
[62]: Index(['Avg_Area_Income', 'Avg_Area_House_Age', 'Avg_Area_Number_of_Rooms',
       'Avg_Area_Number_of_Bedrooms', 'Area_Population'],
      dtype='object')
```

----->The basic linear regression equation with intercept and coeffcient will be as:(just for information)

price = interept+ slope1*Avg_Area_Income* +*slope2*Avg_Area_House_Age+ slope3*Avg_Area_Number_of_Rooms*+ *slope4*Avg_Area_Number_of_Bedrooms+ slope5*Area_Population

price = 1232873.8607364374 + 229789.16018028*Avg_Area_Income* + *63518.96517017*Avg_Area_House_Age + 121747.365563*Avg_Area_Number_of_Rooms* + *2129.51094083*Avg_Area_Number_of_Bedrooms + 150896.93828654*Area_Population

price = 1232873.8607364374 + 229789.1601802879545.4585 + *63518.96517017*5.682861+ 121747.3655637.009188 + *2129.51094083*4.09+ ## 150896.93828654*23086.80050

----->>>>> Prediction of price for tets and train

[63]:
```
y_pred_price = lm.predict(x_test)
```

[64]:
```
y_pred_price
```

[64]:
```
array([1258771.54630259,  821509.83414808, 1743523.44944712, ...,
       1117250.31198148,  718622.68490114, 1518227.07294694])
```

[65]:
```
y_pred_price_train = lm.predict(x_train)
```

[66]:
```
y_test
```

[66]:
```
1718    1.251689e+06
2511    8.730483e+05
345     1.696978e+06
2521    1.063964e+06
54      9.487883e+05
            ...
1881    1.727211e+06
2800    1.707270e+06
1216    1.167450e+06
1648    7.241217e+05
3063    1.561234e+06
Name: Price, Length: 1250, dtype: float64
```

------>>>> VALIDATION for train and test using R2 score

67]:
```
from sklearn.metrics import r2_score
r2_score(y_test,y_pred_price)
```

67]:
```
0.913624999737946
```

68]:
```
from sklearn.metrics import r2_score
r2_score(y_train,y_pred_price_train)
```

68]:
```
0.9165094280297815
```

----->>> Since R2 score for training and testing are very close so there no overfitting issue also no underfititng issue.

-APPROACH 2: OLS Method

[69]:
```
from statsmodels.regression.linear_model import OLS
import statsmodels.regression.linear_model as smf
```

[70]:
```
from statsmodels.regression.linear_model import OLS
import statsmodels.regression.linear_model as smf
reg_model = smf.OLS(endog = y_train,exog = x_train).fit()
reg_model.summary()
```

[70]:

OLS Regression Results

Dep. Variable:	Price	**R-squared (uncentered):**	0.964
Model:	OLS	**Adj. R-squared (uncentered):**	0.964
Method:	Least Squares	**F-statistic:**	2.011e+04
Date:	Sat, 13 Jul 2024	**Prob (F-statistic):**	0.00
Time:	17:02:43	**Log-Likelihood:**	-51812.
No. Observations:	3750	**AIC:**	1.036e+05
Df Residuals:	3745	**BIC:**	1.037e+05
Df Model:	5		
Covariance Type:	nonrobust		

	coef	std err	t	P>|t|	[0.025	0.975]
Avg_Area_Income	10.2091	0.314	32.560	0.000	9.594	10.824
Avg_Area_House_Age	4.916e+04	3478.628	14.131	0.000	4.23e+04	5.6e+04
Avg_Area_Number_of_Rooms	-1.09e+04	3801.792	-2.868	0.004	-1.84e+04	-3448.582
Avg_Area_Number_of_Bedrooms	5109.5344	3631.219	1.407	0.159	-2009.825	1.22e+04
Area_Population	8.5764	0.382	22.429	0.000	7.827	9.326

Omnibus:	0.212	**Durbin-Watson:**	1.997
Prob(Omnibus):	0.899	**Jarque-Bera (JB):**	0.258
Skew:	-0.000	**Prob(JB):**	0.879
Kurtosis:	2.959	**Cond. No.**	9.32e+04

------> Interesting Finding

Scaling or originality of data does not influence R2 score for linear regression

But severely affect the R2 in OLS approach.

Scaling reduce the R2 score abruptly.

In ols method we can check Adj R2 sccore

we can check P value

we can check Durbin Watson Test value to check Autocolinearity: change in variable with time

DWT < 2 = +ve autocolrelation

DWT > 2 = -ve auto corelation

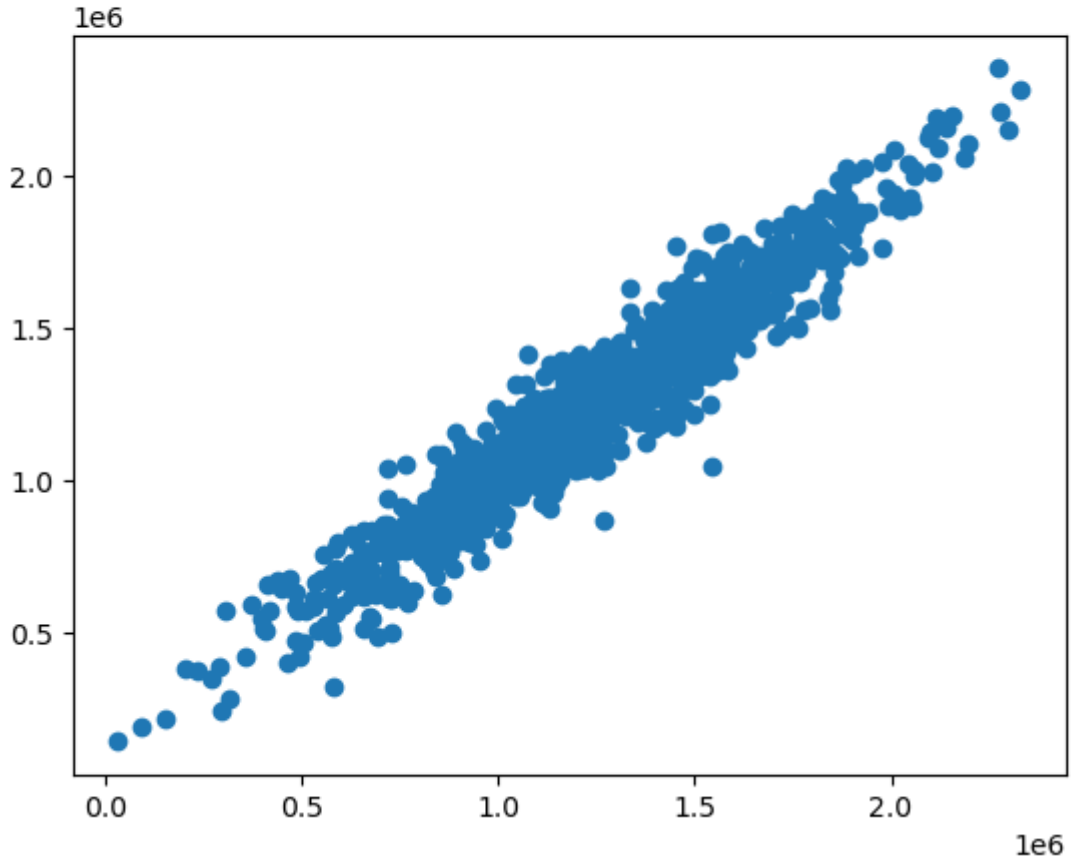
DWT = 2 = no auto corelation

If DWT lies between 1.5 to 2.5 model is acceptable else go for timeseries appraoach.

-----> > There is another assumption data should be linear. Let us check linearity using scatter plot

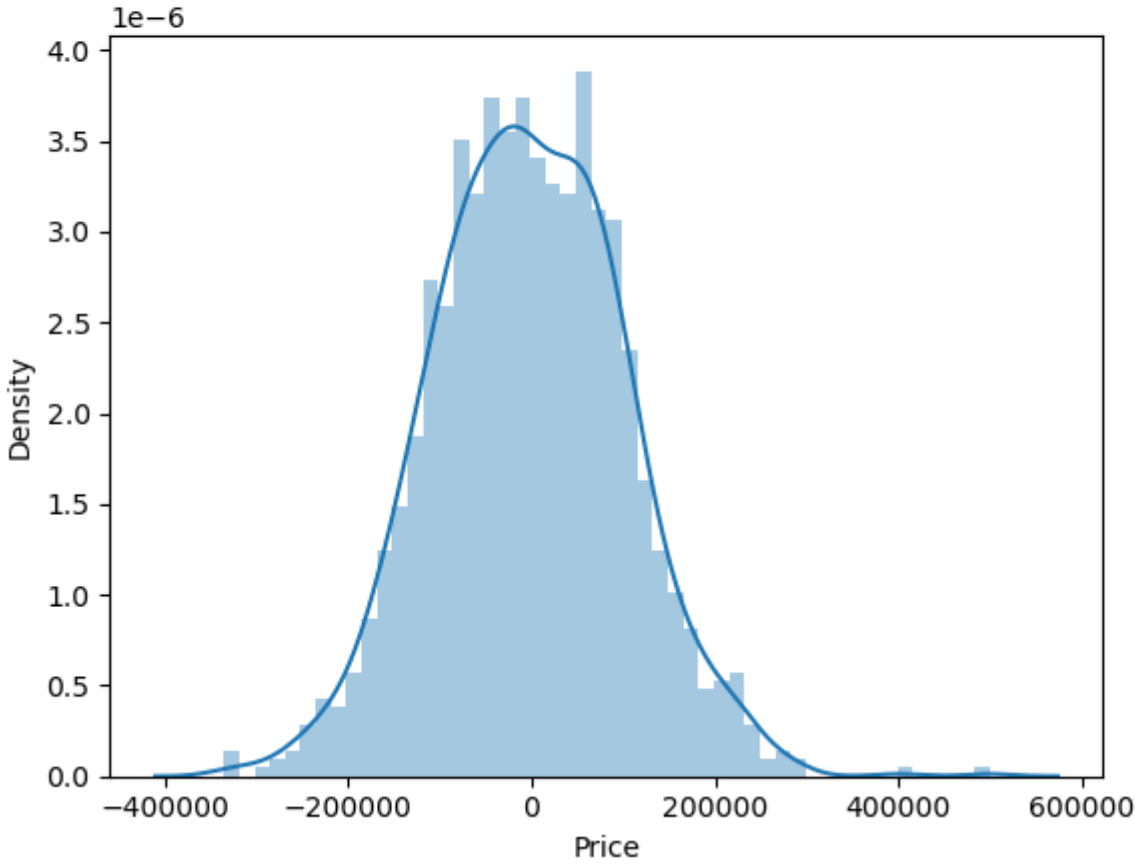
[71]:
```
plt.scatter(y_test,y_pred_price)
```

[71]:
```
<matplotlib.collections.PathCollection at 0x1f3b4f90b90>
```



---->>>There is another assumption normality of residual . Let us check normality of residual

72]:
```
sns.distplot((y_test-y_pred_price),bins = 50)
plt.show()
```



---->> Conclude the model

---> Adj R2 score = 0.96

--->All variable have p value less than 5% except the variable number of bedrooms.So drop it and do it again

--->linearity satisfied

--->Normality of Residual satisifed

---->Homosedasticity satisified

---->No autocorelation satisfied

---->No or little multicolinearity satisfied

---->No endoginity satified ¶

---->sklearn linear regrssion model is a machine learning approach

---->OLS is a statistical appraoch

APPROACH 3

----> Regularization: It is a shrinkage method the algortihm while trying to find best combination of coeffcient vlaue

----> which minimize SSE on training data by penality on higher coefficent value to reduce the error.

----> Ridge also callled L2 ; Close to zero

----> Lasso also called L1: Either zero or close to zero(Sparsity)

----> Most popular is ridge

-----> ElastiNet L1+L2 (Rarely used)

------> lasso based on absolute value which can not be differentiated.

[73]:
```
from sklearn.linear_model import LinearRegression
from sklearn.linear_model import Ridge
from sklearn.linear_model import Lasso
from sklearn.metrics import r2_score
```

[74]:
```
# PART 1 lasso
# penality preferred = 0.1
lasso = Lasso(alpha = 0.1)
lasso.fit(x_train,y_train)
print("Lasso Model :",(lasso.coef_))
```

```
Lasso Model : [2.17329556e+01 1.65690031e+05 1.21586986e+05 1.72617492e+03
 1.53020326e+01]
```

[75]:
```
lasso.intercept_
```

[75]:
```
-2657920.0945172827
```

[76]:
```
# Part2 Ridge
# penality preferred = 0.3
ridge = Ridge(alpha = 0.3)
ridge.fit(x_train,y_train)
print("ridge model",ridge.coef_)
```

```
ridge model [2.17329254e+01 1.65675916e+05 1.21574274e+05 1.73113927e+03
 1.53020563e+01]
```

[77]:
```
ridge.intercept_
```

[77]:
```
-2657765.484140858
```

[78]:
```
y_pred_train_ridge = ridge.predict(x_train)
y_pred_test_ridge = ridge.predict(x_test)
```

[79]:
```
print("Training Accuracy:", r2_score(y_train, y_pred_train_ridge))
print("Test Accuracy:", r2_score(y_test, y_pred_test_ridge))
```

```
Training Accuracy: 0.9165094255220702
Test Accuracy: 0.9136246738265379
```

[80]:
```
y_pred_train_lasso = lasso.predict(x_train)
y_pred_test_lasso = lasso.predict(x_test)
```

[81]:
```
print("Training Accuracy:", r2_score(y_train, y_pred_train_lasso))
print("Test Accuracy:", r2_score(y_test, y_pred_test_lasso))
```

```
Training Accuracy: 0.9165094280296036
Test Accuracy: 0.9136249987893023
```

Performance matrix

[82]:
```
# Mean Absolute Error
from sklearn import metrics
print("MAE", metrics.mean_absolute_error(y_test,y_pred_price))
```

```
MAE 83116.30244093585
```

[83]:
```
# Mean Absolute percent Error (MAPE)

print("MAPE:",metrics.mean_absolute_error(y_test,y_pred_price)/100)
```

```
MAPE: 831.1630244093585
```

[84]:
```
# Mean Squared Error

print("MSE:",metrics.mean_squared_error(y_test,y_pred_price))
```

```
MSE: 10794362043.077223
```

[85]:
```
# Root Mean Squared Error

print("RMSE:",np.sqrt(metrics.mean_squared_error(y_test,y_pred_price)))
```

```
RMSE: 103895.91928019706
```

Approach 4

-----> Gradient Descent : Iterate the model with concept of forward and backward propagation. for finding value of

slope and intercept and if not good, go backward to adjust them and remodel.

----> IN GD we need to use scaled data.

[86]:
```
from sklearn.model_selection import train_test_split
x_train1,x_test1,y_train,y_test = train_test_split(sc_x,y,test_size = 0.25,random_state = 101)
print(x_train1.shape,x_test1.shape,y_train.shape,y_test.shape)
```

```
(3750, 5) (1250, 5) (3750,) (1250,)
```

[87]:
```
from sklearn.linear_model import SGDRegressor
```

[88]:
```
gd_model = SGDRegressor()
gd_model.fit(x_train1,y_train)
```

[88]:
```
▾ SGDRegressor
SGDRegressor()
```

---->
SGDRegressor(alpha=0.0001,,max_iter=1000,tol=0.001,learning_rate='adaptive',eta0=0.01,,n_iter_no_change=5)

---> Key Parameters for Tuning SGDRegressor

1. alpha: Constant that multiplies the regularization term. Also known as the regularization strength; must be a positive float. Typical values: [0.0001, 0.001, 0.01, 0.1, 1, 10]
2. learning_rate: The learning rate schedule. Options include: 'constant': eta = eta0 'optimal': eta = 1.0 / (t + t0) 'invscaling': eta = eta0 / pow(t, power_t) 'adaptive': eta = eta0, as long as the training keeps decreasing. Each time n_iter_no_change consecutive epochs fail to decrease the training loss, the current learning rate is divided by 5.
3. eta0: The initial learning rate for the 'constant', 'invscaling', or 'adaptive' schedules. Typical values: [0.0001, 0.001, 0.01, 0.1, 1]
4. max_iter: The maximum number of passes over the training data (epochs). Typical values: [1000, 5000, 10000]

5.tol: The stopping criterion. If it is not None, the iterations will stop when (loss > best_loss - tol) for n_iter_no_change consecutive epochs. Typical values: [1e-3, 1e-4, 1e-5]

6. penalty: The penalty (regularization term) to be used. Options include: 'none' 'l2' 'l1'

7.n_iter_no_change: Number of iterations with no improvement to wait before early stopping. Typical values: [5, 10]

90]:
```
#learning_rate='adaptive', eta0=0.01, max_iter=10000, tol=1e-3
```

91]:
```
y_pred_gd_train = gd_model.predict(x_train1)
y_pred_gd_test = gd_model.predict(x_test1)
```

92]:
```
print("GD TRAINING ACCURACY:",r2_score(y_train,y_pred_gd_train))
```

```
GD TRAINING ACCURACY: 0.9164646801501356
```

93]:
```
print("GD TEST ACCURACY:",r2_score(y_test,y_pred_gd_test))
```

```
GD TEST ACCURACY: 0.9136619336745166
```

Model saving

[94]:
```
import joblib
```

[ ]:
```
# Save Linear Regression model
joblib.dump(lm, 'linear_regression_model.pkl')
print('lm saved as linear_regression_model.pkl')

# Save Ridge Regression model
joblib.dump(ridge, 'ridge_model.pkl')
print('ridge saved as ridge_model.pkl')

# Save Lasso Regression model
joblib.dump(lasso, 'lasso_model.pkl')
print('lasso saved as lasso_model.pkl')

# Save SGD Regressor model
joblib.dump(gd_model, 'sgd_model.pkl')
print('gd_model saved as sgd_model.pkl')
```

Load each model

Example new data for prediction:

Avg_Area_Income, Avg_Area_House_Age, Avg_Area_Number_of_Rooms, Avg_Area_Number_of_Bedrooms, Area_Population

63345.24005, 7.188236, 5.586729, 3.26, 34310.24283

[4]:
```
import joblib

# Load each model
linear_regression_model = joblib.load('C:/Users/PANKAJ SHARMA/linear_regression_model.pkl')
ridge_model = joblib.load('C:/Users/PANKAJ SHARMA/ridge_model.pkl')
lasso_model = joblib.load('C:/Users/PANKAJ SHARMA/lasso_model.pkl')
sgd_model = joblib.load('C:/Users/PANKAJ SHARMA/sgd_model.pkl')



new_data = [[63345.24005, 7.188236, 5.586729, 3.26,34310.24283]]  # Update with your features

# Make predictions with each model
linear_regression_prediction = linear_regression_model.predict(new_data)
ridge_prediction = ridge_model.predict(new_data)
lasso_prediction = lasso_model.predict(new_data)
sgd_prediction = sgd_model.predict(new_data)

# Print the predictions
print(f'Linear Regression prediction: {linear_regression_prediction}')
print(f'Ridge prediction: {ridge_prediction}')
print(f'Lasso prediction: {lasso_prediction}')
print(f'SGD prediction: {sgd_prediction}')
```

```
Linear Regression prediction: [1254212.77465137]
Ridge prediction: [1254200.70173057]
Lasso prediction: [1254212.62051748]
SGD prediction: [1.96186401e+10]
```