Linear Regression 1

Agenda

- Regression
- Use Case: Car Resale Price Prediction using Linear Regression
- Data Exploration
- Linear Regression Introduction
- Hypothesis in Linear Regression
    - Cost Function
    - Feedback
    - Gradient Decent
- Use Case Simulation
- Metrices for Evaluation Regression Models
- Stochastic Gradient Descent
- Mini Batch Gradient Descent
- Linear Regession using Scikit-Learn

Regression

What is Regression in Machine Learning?

- Regression in Machine Learning is a supervised learning technique used to predict a continuous outcome or target variable based on input features. The goal of regression is to model the relationship between the input variables (independent variables) and the output variable (dependent variable), typically by fitting a line or curve to the data. The model learns from the data and minimizes the error between the predicted and actual values.
- Types of Regression:
    1. Simple Regression
        - Used to predict a continuous dependent variable based on a single independent variable.
        - Simple linear regression should be used when there is only a single independent variable.
    2. Multiple Regression
        - Used to predict a continuous dependent variable based on multiple independent variables.

    - Multiple linear regression should be used when there are multiple independent variables.
  3. NonLinear Regression
    - Relationship between the dependent variable and independent variable(s) follows a nonlinear pattern.
    - Provides flexibility in modeling a wide range of functional forms.

- Regression Algorithms:
  - Linear Regression
  - Polynomial Regression
  - Support Vector Regression (SVR)
  - Decision Tree Regression
  - Random Forest Regression
- Regularized Linear Regression Techniques
  - Ridge Regression
  - Lasso regression

Use Case: Car Resale Price Prediction using Linear Regression

Introduction:

In this case study, we aim to predict the resale price of cars based on various features using the Linear Regression model. The data set consists of information about different cars, including attributes such as selling price, year, kilometers driven, engine capacity, max power, make, model, transmission type, and various other features.

Objective:

The primary objective is to build a predictive model that can accurately estimate the resale price of a car based on its characteristics. This model can be valuable for both buyers and sellers in the used car market, providing insights into the factors influencing resale prices.

Data Exploration:

Download Dataset From Here

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

```
# reading a CSV file named "data.csv" located in the "Data" folder on the your Go
# write your data file path
df = pd.read_csv('/content/CarPred.csv')
```

- selling_price: Integer representing the selling price of the car in lakhs.
- year: Integer representing the manufacturing year of the car
- km_driven: Integer representing the number of kilometers driven by the car.
- engine: Integer representing the engine capacity of the car.
- max_power: Integer representing the maximum power of the car.
- make: String representing the brand of the car (e.g., suzuki, maruti, honda).
- model: String representing the model of the car (e.g., swift, baleno, city).
- transmission_type: Binary variable (0 or 1) representing the transmission type of the car.
- seats_cop: Binary variable (0 or 1) representing the presence of cop seats.
- seats_family: Binary variable (0 or 1) representing the presence of family seats
- seats_large: Binary variable (0 or 1) representing the presence of large seats.
- fuel_cng, fuel_diesel, fuel_electric, fuel_lpg, fuel_patrol: Binary variables (0 or 1) representing the fuel type of the car.
- seller_dealer: Binary variable (0 or 1) representing whether the seller is a dealer.
- seller_self: Binary variable (0 or 1) representing whether the seller is an individual.

Number of Rows: 1,000 rows.

```
df.head()
```

	make	transmission_type	seats_cop	seats_family	seats_large	fuel_cng	f
0	hyundai	0	0	0	1	0	
1	toyota	0	0	0	1	0	
2	ford	1	0	0	0	0	
3	honda	1	0	0	0	0	
4	hyundai	1	0	0	1	0	

```
df.shape
```

```
(20000, 18)
```

What type of problem is this and why?

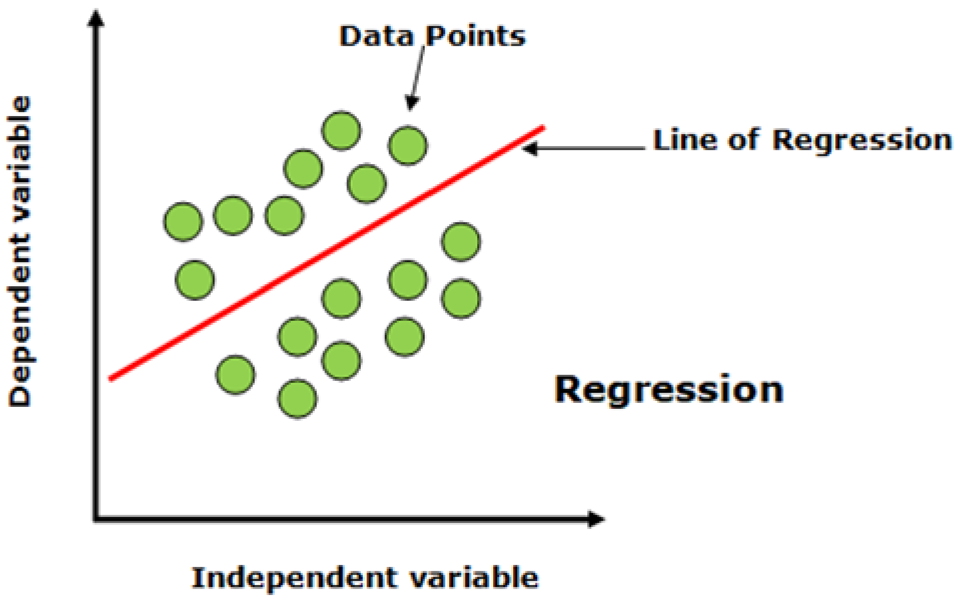
Supervised or unsupervised

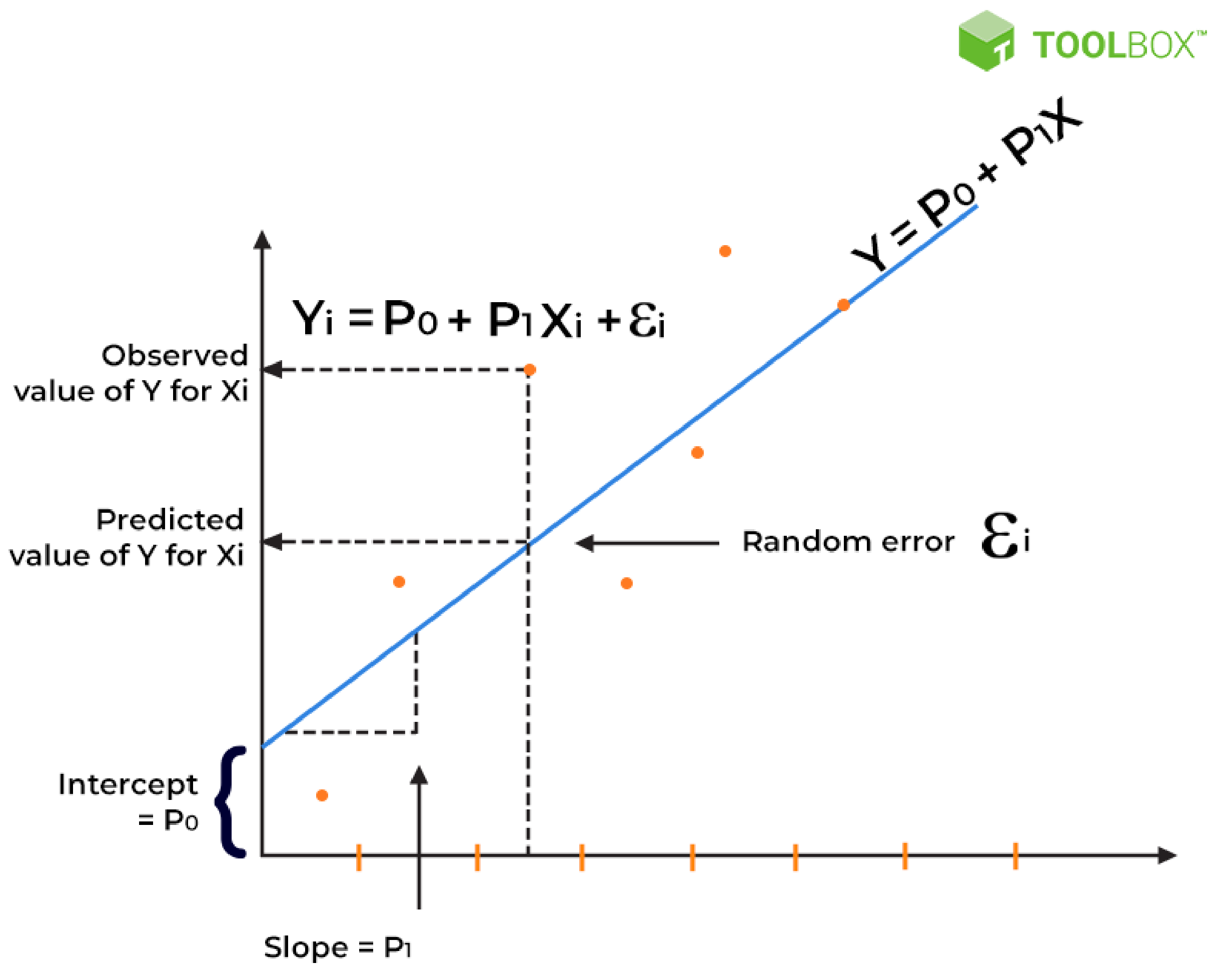
What type of objective is this and why?

Classification or Regression

Linear Regression Introduction

- Linear Regression is a statistical method used in machine learning and statistics to model the relationship between a dependent variable (also known as the target or response variable) and one or more independent variables (also known as predictors or features).
- The primary goal of linear regression is to find the best-fit line that represents the linear relationship between the variables.
- In Simple Linear Regression, there is only one independent variable, and the relationship is modeled with a straight line equation: $Y = \beta_0 + \beta_1 \cdot X$
  - Y represents the dependent variable.
  - X represents the independent variable.
  - β_0 is the intercept, the value of Y when X is zero.
  - β_1 is the slope, indicating the change in Y for a one-unit change in X.
- The goal is to estimate the coefficients β_0 and β_1 such that the sum of squared differences between the predicted values and the actual values (residuals) is minimized. Once the model is trained, it can be used to predict the dependent variable's values for new, unseen data.
- In Multiple Linear Regression, the same principles are applied, but there are multiple independent variables: $Y = \beta_0 + \beta_1 \cdot X + \beta_2 \cdot X + \ldots + \beta_n \cdot X$




TOOLBOX™
$Y = P_0 + P_1X$
$Y_i = P_0 + P_1X_i + \varepsilon_i$
Observed value of Y for Xi
Predicted value of Y for Xi
Random error ε_i
Intercept = P_0
Slope = P_1

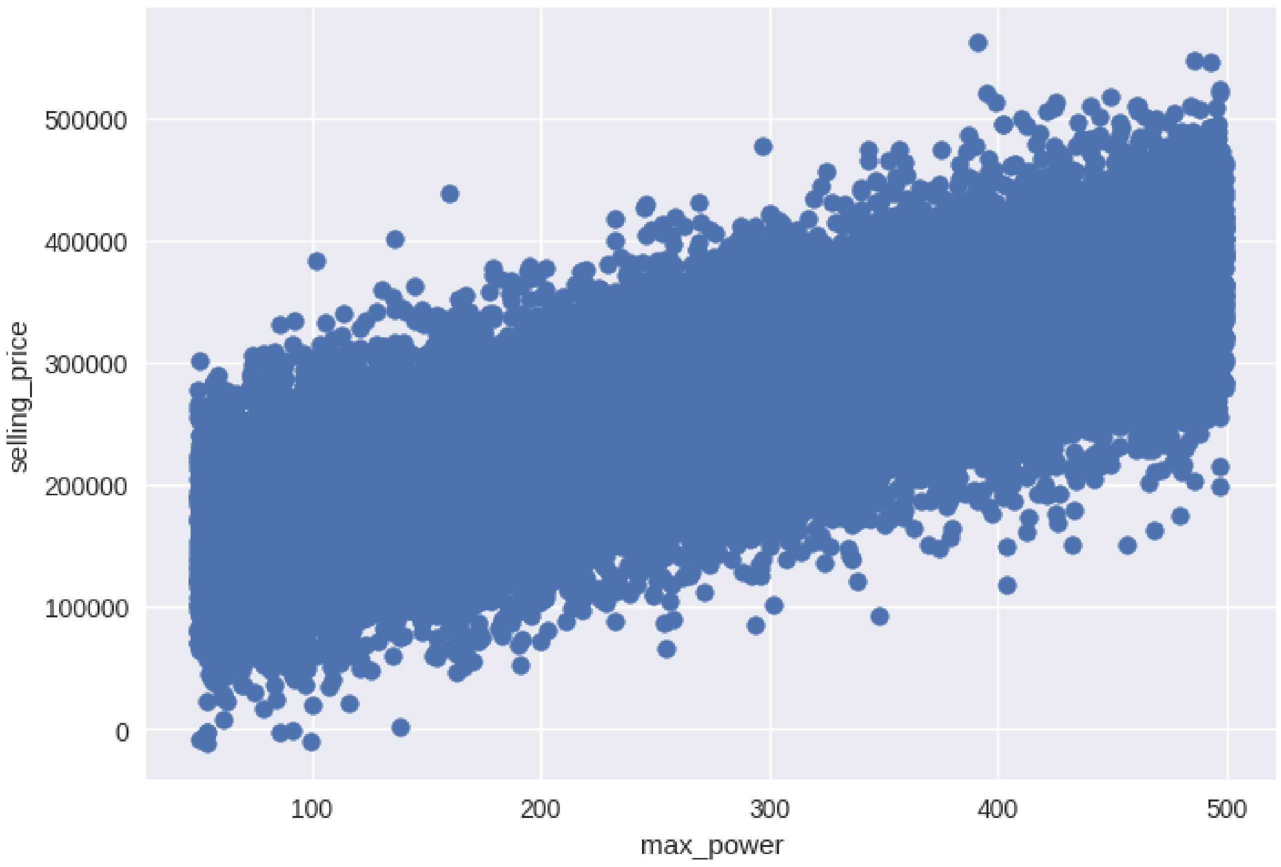
- Linear Regression is widely used for tasks such as predicting sales, stock prices, and various other real-world phenomena where understanding and quantifying the relationship between variables are essential.

The use case for linear regression in this scenario is justified by the nature of the problem and the characteristics of the data.

- The goal is to predict the resale price of cars (target variable: selling price).
- The problem involves predicting a continuous numeric variable, which aligns with the characteristics of a regression problem.

- Single Input Variable:
  - For simplicity and as a starting point, the analysis focuses on a single input variable, max power.
  - Simple Linear Regression is chosen due to having only one predictor variable initially.

```
#define x and y
x = df["max_power"].values
y = df["selling_price"].values

plt.scatter(x, y)
plt.xlabel("max_power")
plt.ylabel("selling_price")
plt.show()
```



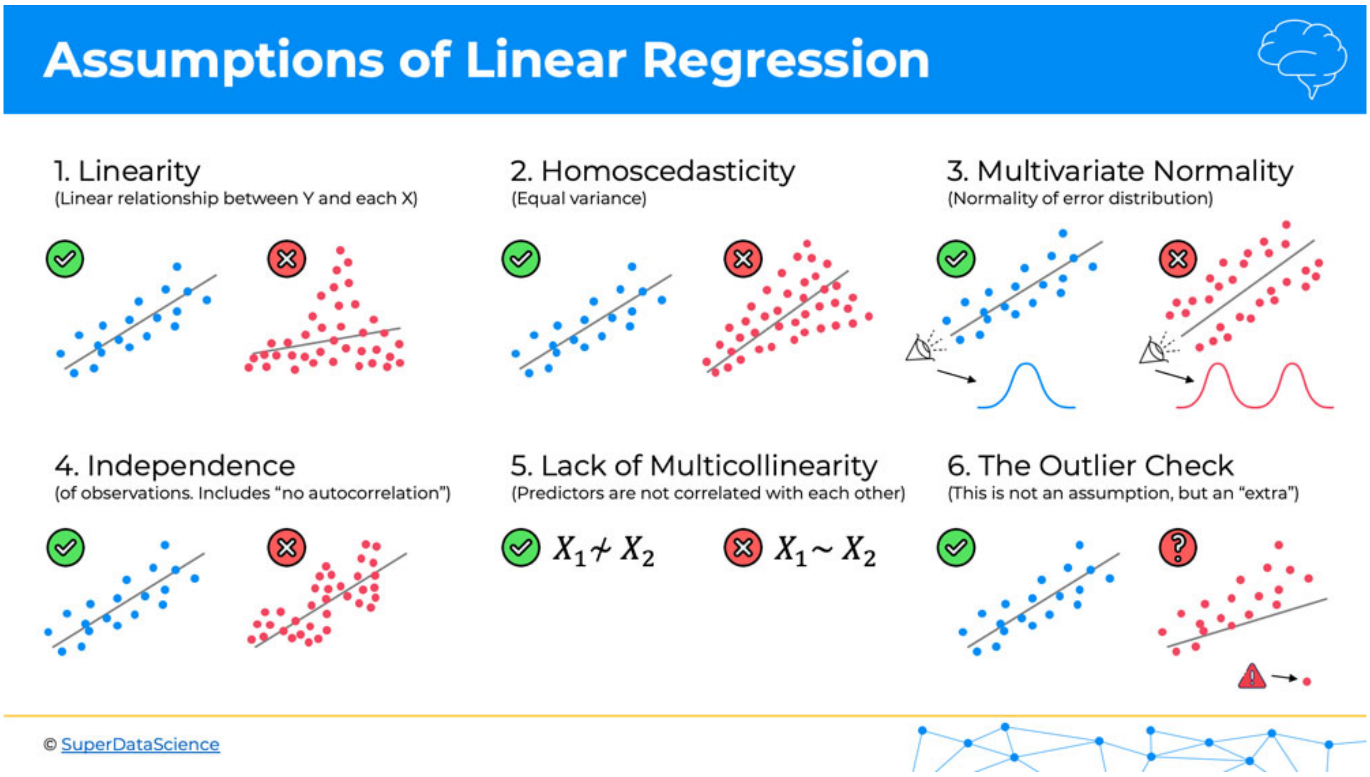
- Linear Relationship Assumption:
    - Linear regression assumes that the relationship between the input variable (max power) and the target variable (selling price) can be represented by a straight line.
    - Exploratory Data Analysis (EDA), including the examination of a scatter plot and correlation analysis, supports the assumption that there is a linear or closely linear relationship between max power and selling price.

Assumptions in Linear Regression

Linear regression is a fundamental statistical technique used to model the relationship between a dependent variable and one or more independent variables.

When performing linear regression, there are several key assumptions that must be satisfied for the model to be valid.

Here's a detailed overview of these assumptions:


Assumptions of Linear Regression
1. Linearity
(Linear relationship between Y and each X)
2. Homoscedasticity
(Equal variance)
3. Multivariate Normality
(Normality of error distribution)
4. Independence
(of observations. Includes "no autocorrelation")
5. Lack of Multicollinearity
(Predictors are not correlated with each other)
$X_1 \nsim X_2$
$X_1 \sim X_2$
6. The Outlier Check
(This is not an assumption, but an "extra")
© SuperDataScience

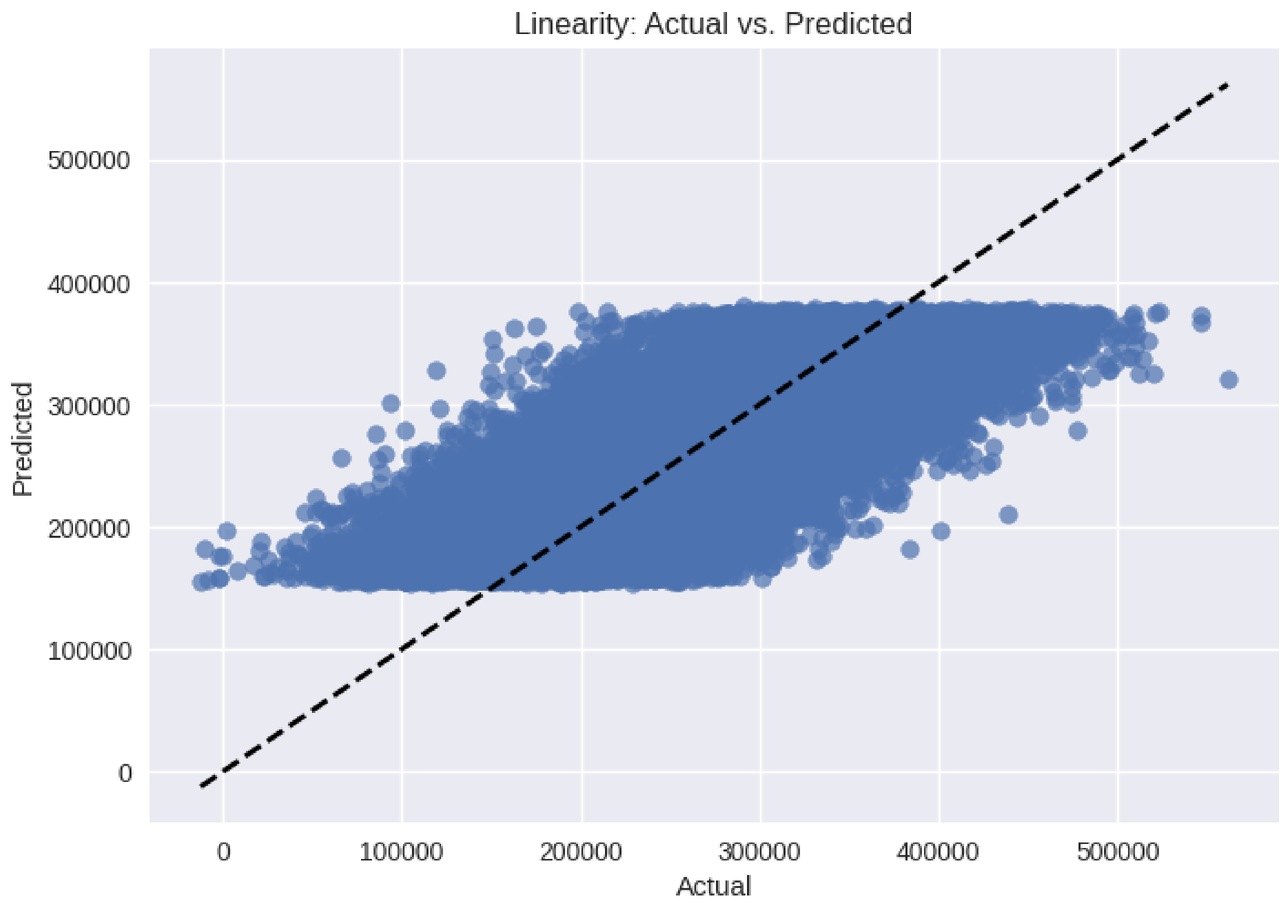
Linearity

The relationship between the independent variable(s) and the dependent variable is linear. This means that the change in the dependent variable is proportional to the change in the independent variable(s), with a constant rate of change.

Simulation for Linearity

```
# Fit linear regression model
X = df[['year', 'km_driven', 'engine', 'max_power', 'transmission_type', 'seats_c
y = df['selling_price']


import statsmodels.api as sm
# Add a constant term for intercept
X_with_const = sm.add_constant(X)


# Fit linear regression model
model = sm.OLS(y, X_with_const).fit()
```

```
# Check linearity assumption by plotting actual vs. predicted values
plt.scatter(y, model.predict(), alpha=0.7)
plt.plot([y.min(), y.max()], [y.min(), y.max()], 'k--', lw=2)
plt.xlabel('Actual')
plt.ylabel('Predicted')
plt.title('Linearity: Actual vs. Predicted')
plt.show()
```



By visually inspecting the scatter plot of actual versus predicted values, We can assess whether our linear regression model follows the linearity assumption.

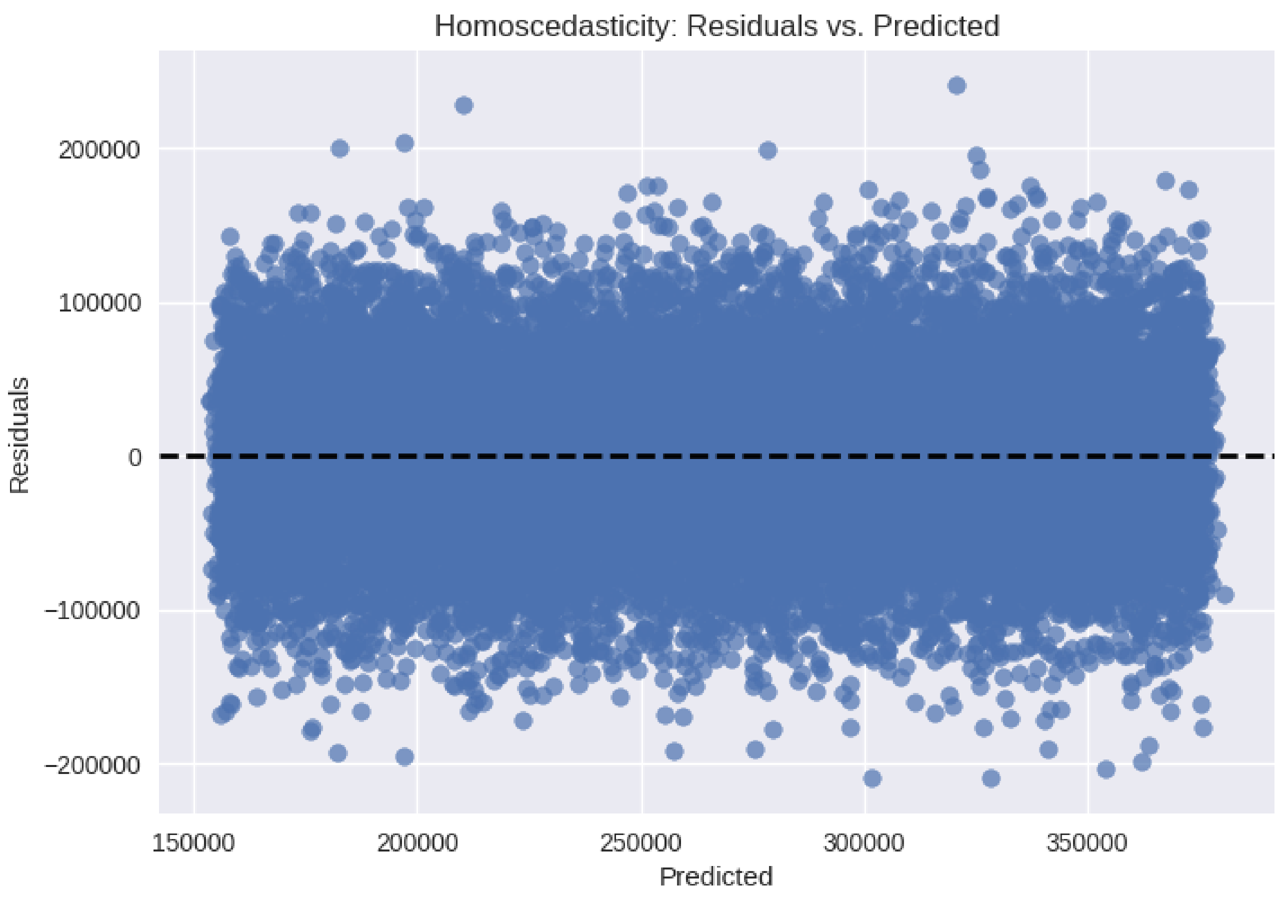
If the points form a reasonably straight line, it suggests that the assumption holds, whereas nonlinear patterns may indicate violations of the linearity assumption.

Homoscedasticity

Also known as constant variance, this assumption states that the variance of the errors is constant across all levels of the independent variables. In practical terms, this means that the spread of the residuals should remain approximately the same as the predicted values increase. A violation of this assumption results in heteroscedasticity.

Simulation for Homoscedasticity

```
# Check homoscedasticity by plotting residuals vs. predicted values
residuals = model.resid


plt.scatter(model.predict(), residuals, alpha=0.7)
plt.xlabel('Predicted')
plt.ylabel('Residuals')
plt.title('Homoscedasticity: Residuals vs. Predicted')
plt.axhline(y=0, color='k', linestyle='--', lw=2)
plt.show()
```



To assess homoscedasticity based on the plot of residuals versus predicted values, you'll need to interpret the patterns and characteristics of the scatter plot.

Here's how you can evaluate homoscedasticity using the plot:

1. Equal Spread of Residuals: Homoscedasticity implies that the spread (variance) of the residuals remains approximately constant across all levels of the predicted values. In the scatter plot, look for a consistent spread of points around the horizontal line representing zero residual (the dashed line in the plot).

2. No Clear Patterns or Trends: Ideally, the scatter plot should not exhibit any discernible pattern or trend as you move along the predicted values. There should be no systematic increase or decrease in the spread of residuals. Instead, the points should be randomly scattered around the zero residual line.
3. Consistent Residuals Across Predicted Values: Check whether the dispersion of residuals remains relatively constant across the range of predicted values. In other words, the variability of residuals should not systematically change as the predicted values increase or decrease.
4. Heteroscedasticity: If the scatter plot displays a funnel-shaped pattern or if the spread of residuals widens or narrows systematically with the predicted values, it suggests heteroscedasticity, which violates the homoscedasticity assumption.

Based on the plot generated by above simulation, you can evaluate whether your dataset follows the homoscedasticity assumption by examining these characteristics. If the scatter plot shows a consistent spread of residuals around the zero line with no clear pattern or trend, it suggests that the data satisfies the homoscedasticity assumption. However, if you observe a non-constant spread or any systematic pattern in the residuals, it may indicate potential violations of homoscedasticity, requiring further investigation and possibly model refinement.

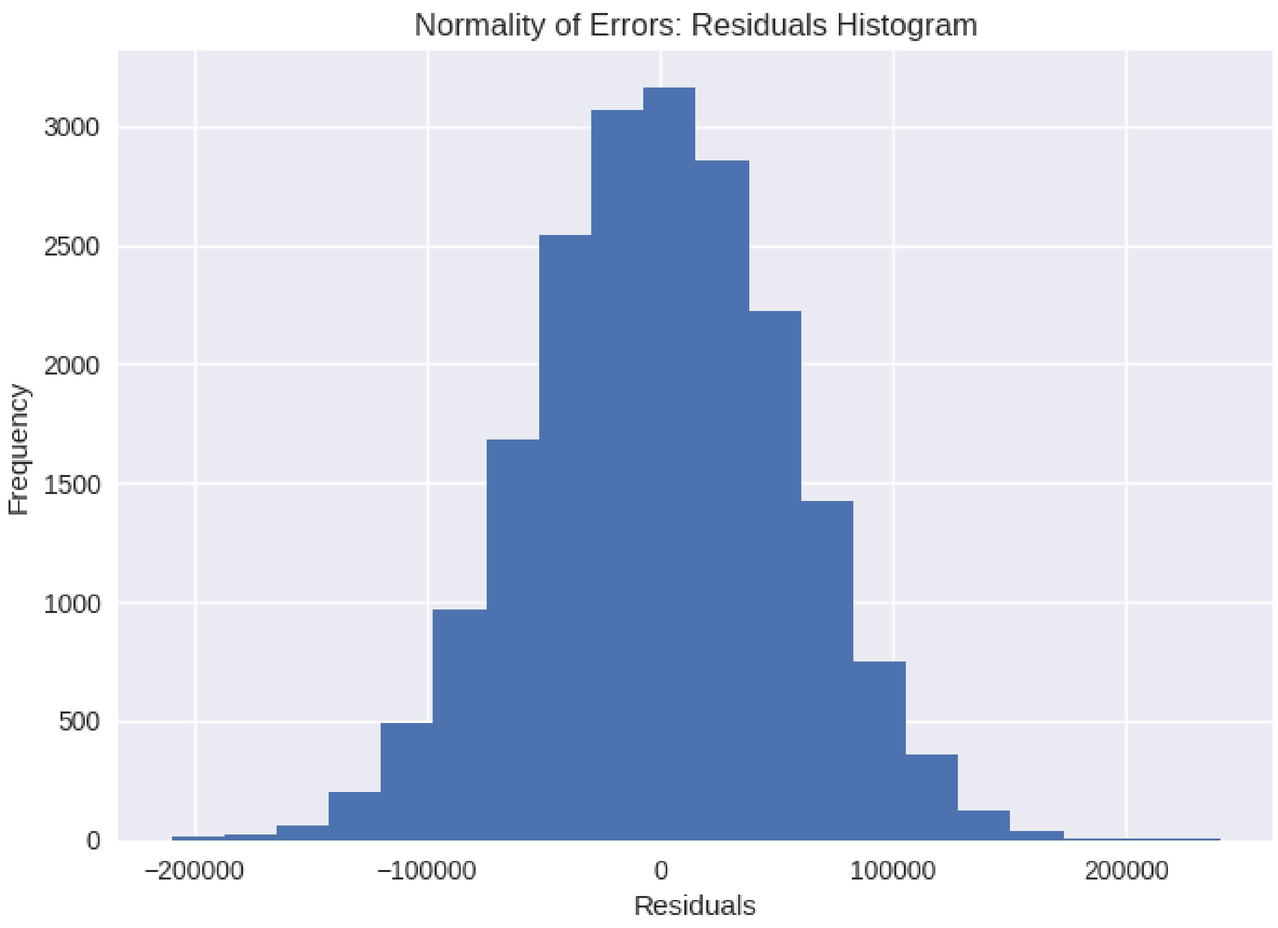
Normality of Errors

The errors (residuals) of the model are normally distributed. This means that if you were to plot the distribution of residuals, it should resemble a bell curve or Gaussian distribution. While this assumption is not strictly necessary for large sample sizes (thanks to the Central Limit Theorem), it can be important for smaller sample sizes to ensure the validity of statistical tests and confidence intervals.

Simulation for Normality of Errors

To assess the normality of errors based on the histogram of residuals, you'll need to interpret the shape and distribution of the histogram.

```
# Check normality of errors by plotting a histogram of residuals
plt.hist(residuals, bins=20)
plt.xlabel('Residuals')
plt.ylabel('Frequency')
plt.title('Normality of Errors: Residuals Histogram')
plt.show()
```



Based on the histogram generated by the simulation, you can evaluate whether your dataset follows the normality of errors assumption by examining these characteristics.

If the histogram exhibits a roughly bell-shaped curve with a single peak and comparable frequencies in each bin, it suggests that the residuals are approximately normally distributed.

However, if the histogram shows significant deviations from these patterns, it may indicate potential violations of the normality assumption, requiring further investigation and possibly transformation of the data.

No Autocorrelation

In the context of time series data or any data with a sequential ordering, this assumption states that the residuals are not correlated with each other. Autocorrelation occurs when the residuals from one observation are correlated with the residuals from nearby observations. This violates the assumption of independent errors and can lead to biased parameter estimates and incorrect statistical inference.

Simulation for No Autocorrelation

The Durbin-Watson statistic is a test used to detect the presence of autocorrelation in the residuals of a regression model.

```
# Check independence of errors using Durbin-Watson test (DW statistic)
durbin_watson = sm.stats.stattools.durbin_watson(residuals)
print("Durbin-Watson statistic:", durbin_watson)
```

```
Durbin-Watson statistic: 2.01352969275313
```

The test statistic ranges from 0 to 4, with values close to 2 indicating no significant autocorrelation.

Based on the Durbin-Watson statistic generated by the simulation, you can evaluate whether your dataset follows the assumption of no autocorrelation.

If the Durbin-Watson statistic is close to 2 (typically between 1.5 and 2.5), it suggests that there is no significant autocorrelation in the residuals, indicating that the assumption of independence of errors is met.

However, if the statistic deviates substantially from 2, it may indicate the presence of autocorrelation, requiring further investigation and possibly model adjustments.

No Perfect Multicollinearity

In multiple linear regression (where there are more than one independent variable), there should be no perfect linear relationship between the independent variables. Perfect multicollinearity occurs when one independent variable can be exactly predicted from another independent variable or a combination of other independent variables. This situation makes it impossible to estimate the unique effect of each independent variable on the dependent variable.

Simulation for No Perfect Multicollinearity

- The Variance Inflation Factor (VIF) is a statistical tool used in regression analysis to detect and measure the severity of multicollinearity.
  Multicollinearity occurs when two or more independent variables in a regression model are highly correlated with each other.
- This can lead to several problems, including:
  - Inflated variances of the regression coefficients: This makes it difficult to accurately estimate the true effect of each independent variable on the dependent variable.
  - Unreliable coefficient estimates: Even if the coefficients are statistically significant, they may not be reliable enough to draw meaningful conclusions.

- Instability of the model: Small changes in the data can lead to large changes in the estimated coefficients.

The VIF calculates how much the variance of a regression coefficient is inflated due to multicollinearity. It is calculated for each independent variable in the model. A VIF value of 1 indicates no multicollinearity, while values greater than 1 indicate some degree of multicollinearity. The higher the VIF value, the more severe the multicollinearity.

Here are some general guidelines for interpreting VIF values:

- VIF < 5: No multicollinearity issue.
- 5 ≤ VIF < 10: Moderate multicollinearity, may not be a major problem.
- VIF ≥ 10: Severe multicollinearity, investigate further and consider remedial actions.

It is important to note that these are just guidelines, and the decision of whether or not to take action to address multicollinearity will depend on the specific context of your analysis.

Here are some of the ways to address multicollinearity:

- Remove one or more of the collinear variables: This is the most common approach, but it should only be done if the variable(s) being removed are not theoretically important to the model.
- Combine collinear variables: If two or more variables are measuring the same underlying construct, they can be combined into a single variable.
- Use ridge regression or LASSO: These are regression techniques that can help to reduce the impact of multicollinearity on the coefficient estimates.

```
# Check for multicollinearity using variance inflation factor (VIF)
from statsmodels.stats.outliers_influence import variance_inflation_factor

vif_data = X_with_const.copy()
vif_data = vif_data.drop(columns='const')  # Exclude the constant term
vif = pd.DataFrame()
vif["Variable"] = vif_data.columns
vif["VIF"] = [variance_inflation_factor(vif_data.values, i) for i in range(vif_da

print(vif)
```

```
             Variable        VIF
0                year  56.648571
1           km_driven   4.007914
2              engine   7.729838
3           max_power   5.478773
4   transmission_type   1.998263
5           seats_cop   2.010321
6        seats_family   3.019145
7         seats_large   5.036540
8            fuel_cng   1.994996
9         fuel_diesel   3.047902
10      fuel_electric        inf
11        fuel_patrol  17.090605
12           fuel_lpg   9.085081
```

```
13      fuel_electric        inf
14      seller_dealer   1.995045
15        seller_self   1.977211
/usr/local/lib/python3.10/dist-packages/statsmodels/stats/outliers_influence.
  vif = 1. / (1. - r_squared_i)
```

It's important to note that while these assumptions provide a framework for building and interpreting linear regression models, real-world data often deviate from these ideal conditions. Therefore, it's crucial to assess the extent to which these assumptions are violated and consider techniques such as robust regression or transformations to address any issues. Additionally, diagnostic tests and visualizations can help evaluate the validity of these assumptions and guide model refinement.

Hypothesis in Linear Regression

The hypothesis is a mathematical representation of the relationship between the input variable (X) and the output variable (Y).

The goal is to find a linear equation that best fits the given data, allowing us to make predictions or estimates about Y based on the value of X.

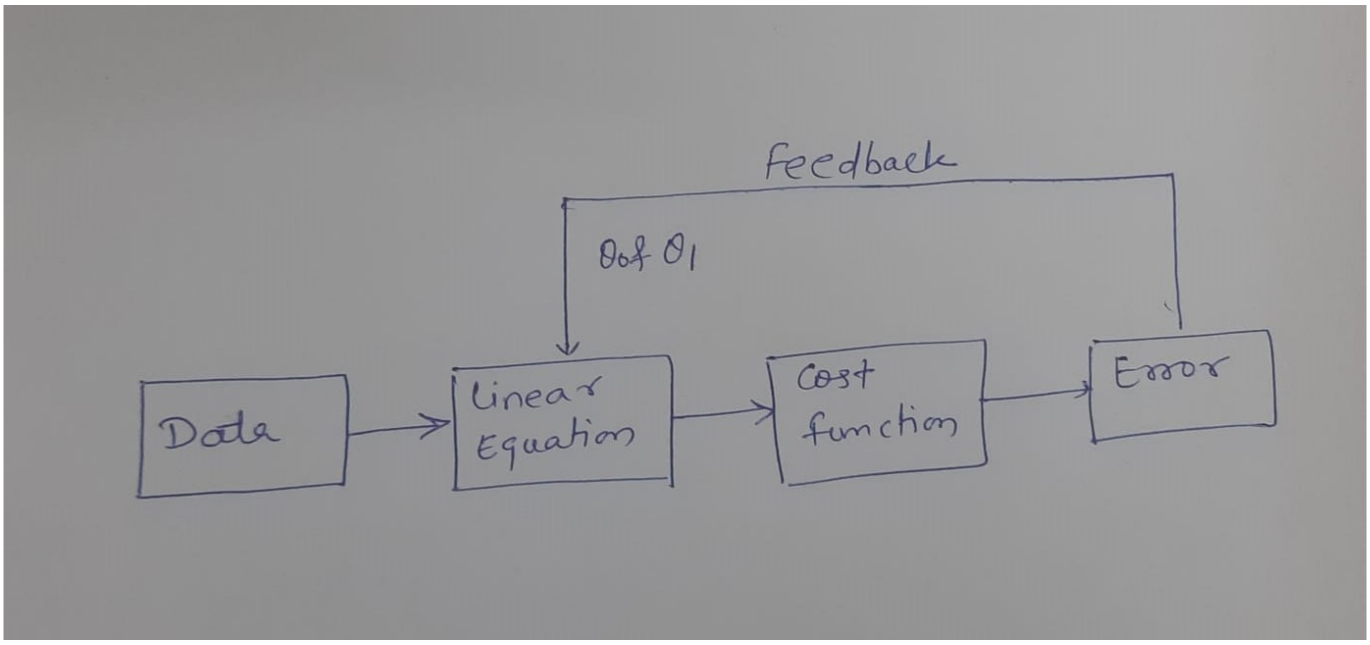
The general form of the hypothesis in linear regression is expressed as:

$Y = \Theta_0 + \Theta_1 \cdot X$

where,

- Y: This is the dependent variable or the target variable that we want to predict. In the context of the example you provided, it could be the resale price of a car.
- X: This is the independent variable or the input feature. It represents the variable for which we have data and on which we base our predictions. In your example, it might be a feature like the maximum power of a car.
- Θ_0: This is the y-intercept of the linear equation. It represents the value of Y when X is 0. In other words, it's the baseline value of Y.
- Θ_1: This is the slope of the line. It represents the change in Y for a unit change in X. In the context of your example, it indicates how much the resale price is expected to change for each unit increase in maximum power.

The linear regression model aims to find the values of Θ_0 and Θ_0 that minimize the difference between the predicted values ($Y^\wedge$) and the actual values of Y in the training data set. This process is often referred to as training the model.



Once the model is trained, you can use it to predict the values of Y for new or unseen values of X. The linear equation provides a straight-line approximation of the relationship between the input and output variables. The goal is to have a line that best fits the data points in a way that it can generalize well to make accurate predictions for new data.

Cost Function

The cost function is a measure of how well the model is performing, and it helps in quantifying the error between the predicted values and the actual values. The mean squared error (MSE) is a commonly used cost function in linear regression.

The mean squared error is calculated as the average of the squared differences between the predicted values ($Y^{\wedge}$) and the actual values (Y) in the training dataset. The formula for the mean squared error is as follows:

- **Update Rule:** $\theta_j := \theta_j - \alpha \frac{\partial J(\theta_0, \theta_1)}{\partial \theta_j}$
  $\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)$
  $\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} [(h_\theta(x_i) - y_i) \cdot x_i]$
- α: Learning rate, a small positive value (controls step size).

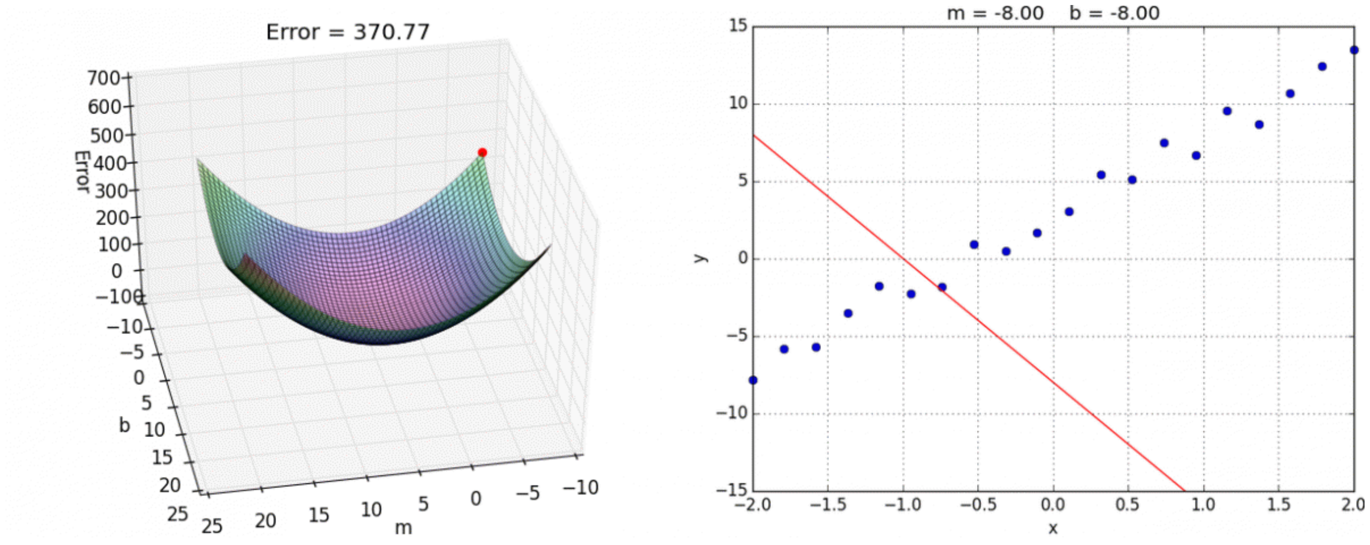
- m: This represents the number of data points in the training dataset.
- Y_{hati}: This is the predicted value of the dependent variable (Y) for the i-th data point based on the linear regression model.
- y: This is the actual observed value of the dependent variable (Y) for the i-th data point.

The goal during the training of a linear regression model is to minimize the mean squared error.

In other words, we want to find the values of the parameters (Θ_0 and Θ_1) that lead to the smallest average squared difference between predicted and actual values.

Optimization algorithms, such as gradient descent, are often used to iteratively adjust the parameters and minimize the cost function.

When working with linear regression, we aim to find the best line that fits the training data. The cost function measures the difference between the predicted values of the model and the actual target values. By minimizing this cost function, we can determine the optimal values for the model's parameters and improve its performance.



Feedback

In machine learning and linear regression, feedback refers to the process of using the information provided by the cost function to adjust the model parameters in order to minimize the error.

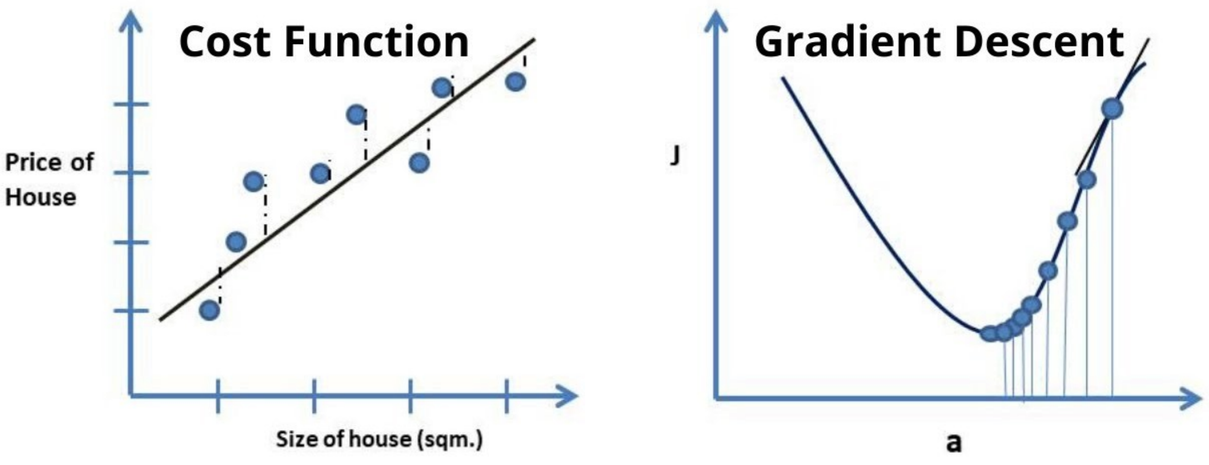
The feedback loop involves a continuous process of adjusting the model based on the information provided by the cost function. The model iteratively learns from its mistakes and refines its predictions. The feedback loop continues until the model parameters reach values that result in a minimal cost, indicating a well-performing model on the training data.

Gradient Decent

Gradient Descent is an iterative optimization algorithm used to minimize a cost function or error in the context of machine learning. It's widely employed in various optimization problems, especially in training machine learning models. The process involves adjusting parameters iteratively to reach the optimal values that result in the minimum cost or error.

Process of Gradient Descent:

1. Initialization:
    - Initialize the parameters (weights) randomly or with some predefined values.
    - Choose a learning rate (α), which determines the step size in the parameter space.
2. Compute Cost Function:
    - Evaluate the cost function (or loss function) for the current parameter values.
    - The cost function quantifies the difference between predicted values and actual values.



3. Compute Gradients:
    - Calculate the gradients of the cost function with respect to each parameter.
    - Gradients indicate the direction and magnitude of the steepest increase of the function.
4. Update Parameters:
    - Adjust the parameters in the direction opposite to the gradients to reduce the cost.
    - The update rule for each parameter θ is: θ = θ - α * (∂J/∂θ), where J is the cost function.
5. Iterate:
    - Repeat steps 2-4 until convergence or a predefined number of iterations.
    - Convergence is often determined by observing a small change in the cost function or gradients below a threshold.
6. Learning Rate:
    - The learning rate (α) is crucial. A high learning rate may cause overshooting, while a low learning rate may slow down convergence.
    - Learning rates are usually chosen through experimentation and optimization.

- **Update Rule:** $\theta_j := \theta_j - \alpha \frac{\partial J(\theta_0,\theta_1)}{\partial \theta_j}$
  $\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x_i) - y_i)$
  $\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} [(h_\theta(x_i) - y_i) \cdot x_i]$
- α: Learning rate, a small positive value (controls step size).

Gradient Descent is a foundational concept in machine learning optimization, forming the basis for more advanced optimization algorithms used in deep learning and neural networks. Understanding its principles is crucial for effectively training models and solving various optimization problems.

Defining univariate_linear_hypothesis funtion

```
# We are defining a function called univariate_linear_hypothesis.
# This function takes an input vector x and a list theta containing two values (T
# It calculates the predicted value (Y hat) using the univariate linear hypothesi

def univariate_linear_hypothesis(x, theta):
  y_hat = theta[0] + theta[1]*x
  return y_hat
```

Defining cost funtion

```
# We need to define our cost function, and in this case, we are using the mean sq
# The cost function takes input vectors X and Y, and a list theta containing two
# It computes the predicted values (Y hat) using the univariate linear hypothesis

def cost(X, Y , theta):
  m = X.shape[0] #number of training examples
  total_error = 0.0
  # Calculate the total error by summing up squared differences for each observat
  for i in range(m):
      y_hat = univariate_linear_hypothesis(X[i], theta)  # Compute Y hat using th
      total_error += (y_hat - Y[i])**2  # Square of the difference (residual) for

  # Calculate and return the mean squared error
  return total_error / m
```

Before we delve into the intricacies of gradient descent, let's take a moment to understand how it operates in practical terms.

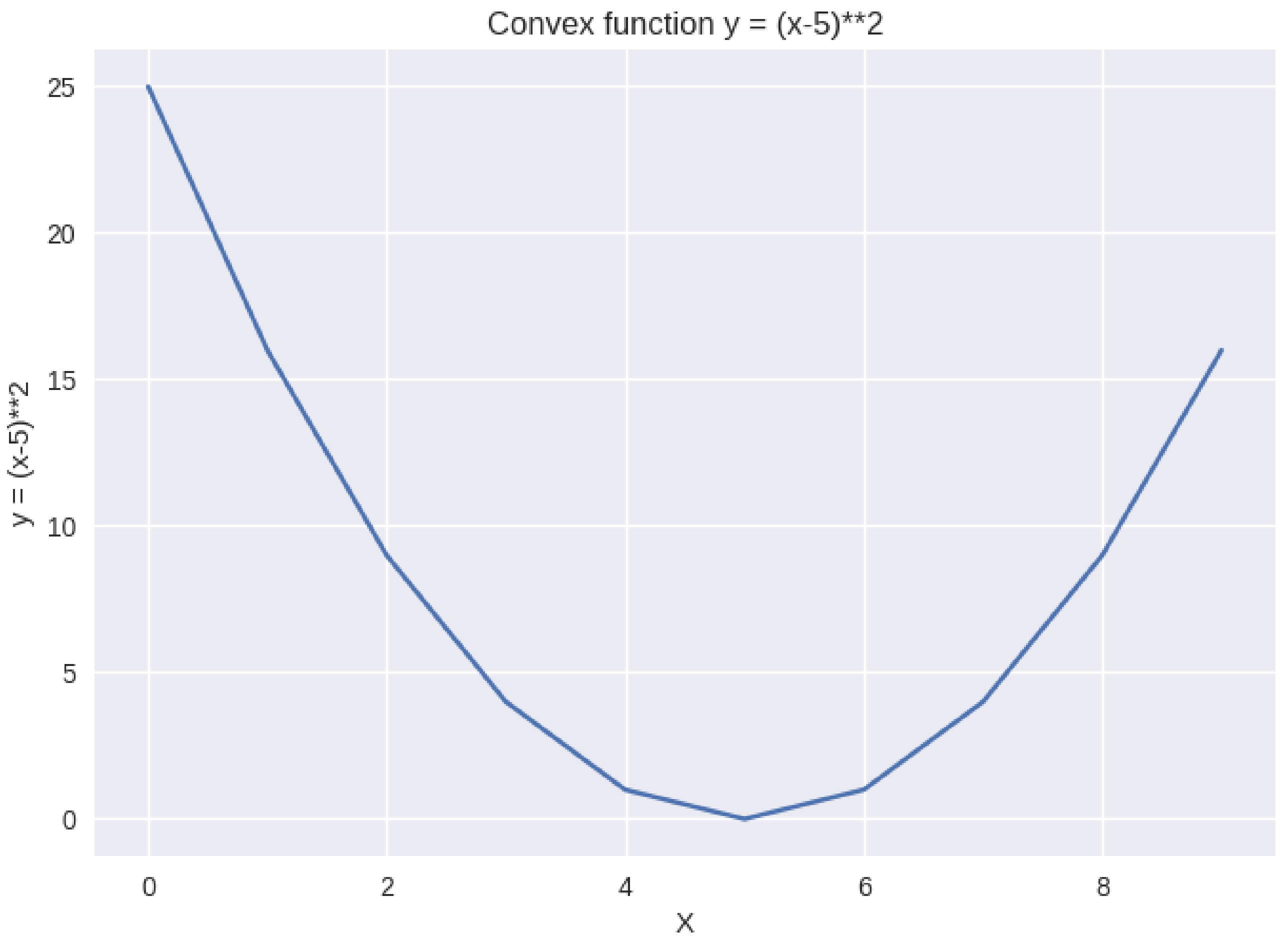
We are creating a simple convex function y = (x-5)^2 to demonstrate the concept of gradient descent. The minimum value of this function occurs at x = 5, where the cost is zero.

We will visualize this convex function using a plot and then demonstrate how the gradient descent approach can be used to find the minimum point of the function.

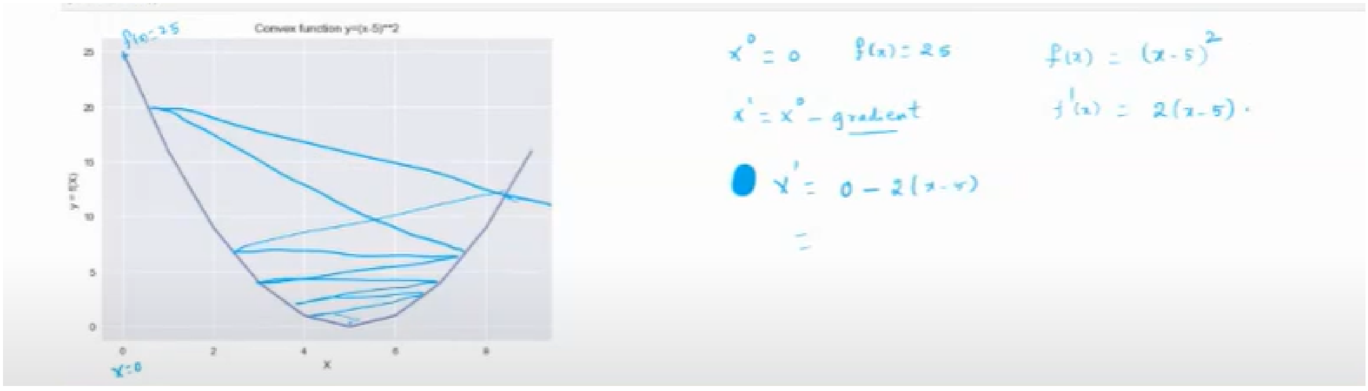
Creating simple convex funtion

```
# We are creating a simple convex function y = (x-5)^2.
# Our goal is to visualize this function and demonstrate how the gradient descent

# Create some random data for X values
X = np.arange(10)

# Define the convex function y = (x-5)^2
Y = (X-5)**2  # X = 5, cost function will give minimum (0)

# Plot the convex function
plt.style.use("seaborn")
plt.plot(X, Y)
plt.xlabel("X")
plt.ylabel("y = (x-5)**2")
plt.title("Convex function y = (x-5)**2")
plt.show()
```

```
<ipython-input-140-7d77c578fe52>:11: MatplotlibDeprecationWarning: The seaborn
  plt.style.use("seaborn")
```

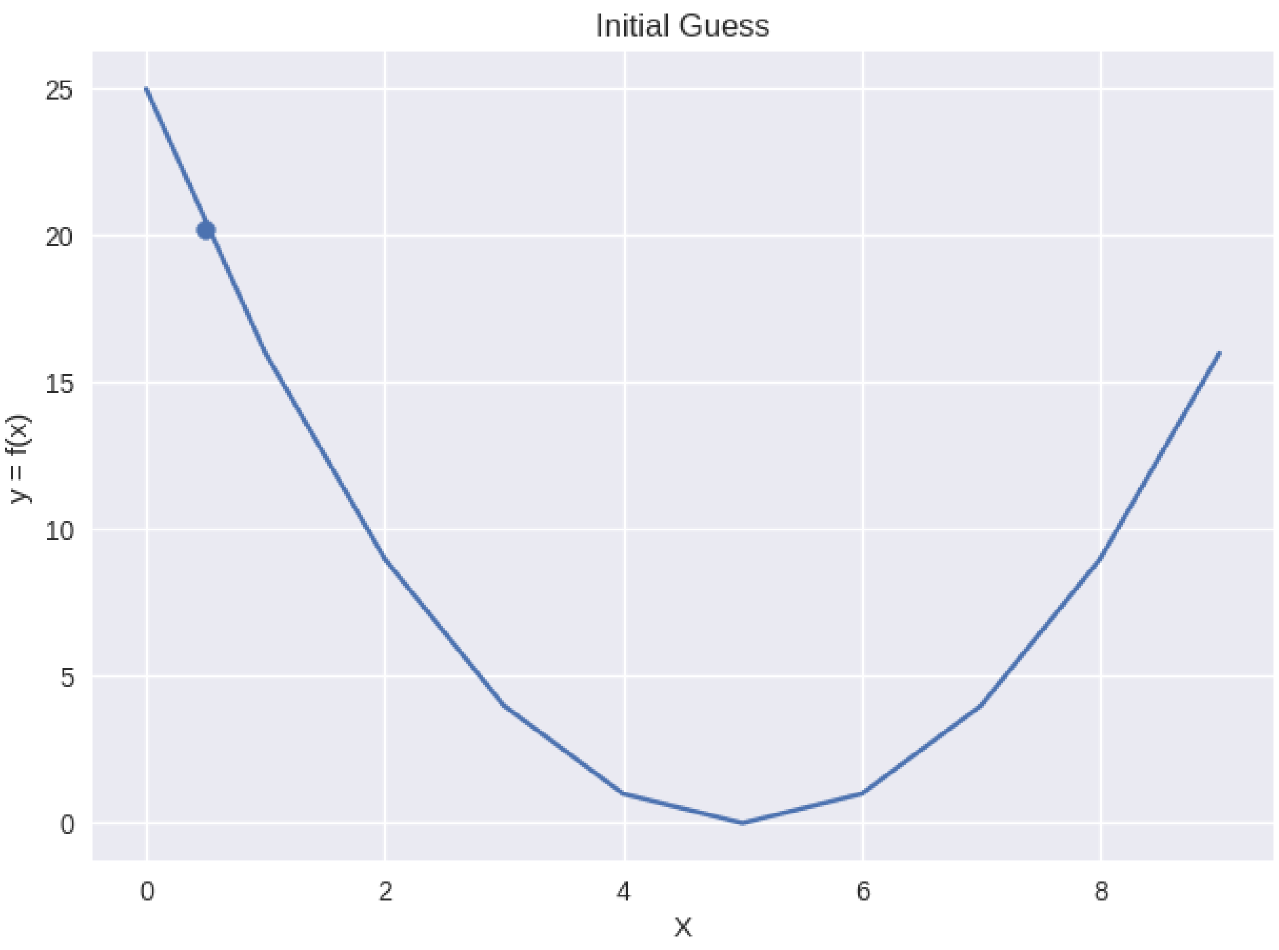


Above code creates a simple convex function and visualizes it using Matplotlib. The plot shows a smooth, U-shaped curve, and the minimum point occurs at X = 5 where the cost is zero.



- If we initialize our variable X to 0, the initial cost function value (f(x)) would be 25.
- The update rule for X (X1) using the gradient is X1 = X0 - 2 * (X0 - 5), where the gradient of the cost function f(x) = (X - 5)^2 is 2 * (X - 5).
- If we directly use the gradient without considering the learning rate, it might lead to large steps in the parameter space. This could cause the algorithm to overshoot the minimum point and oscillate or diverge.
- Especially in the case of non-smooth convex surfaces, taking large steps without a learning rate could make it challenging for the algorithm to converge to the minimum point.

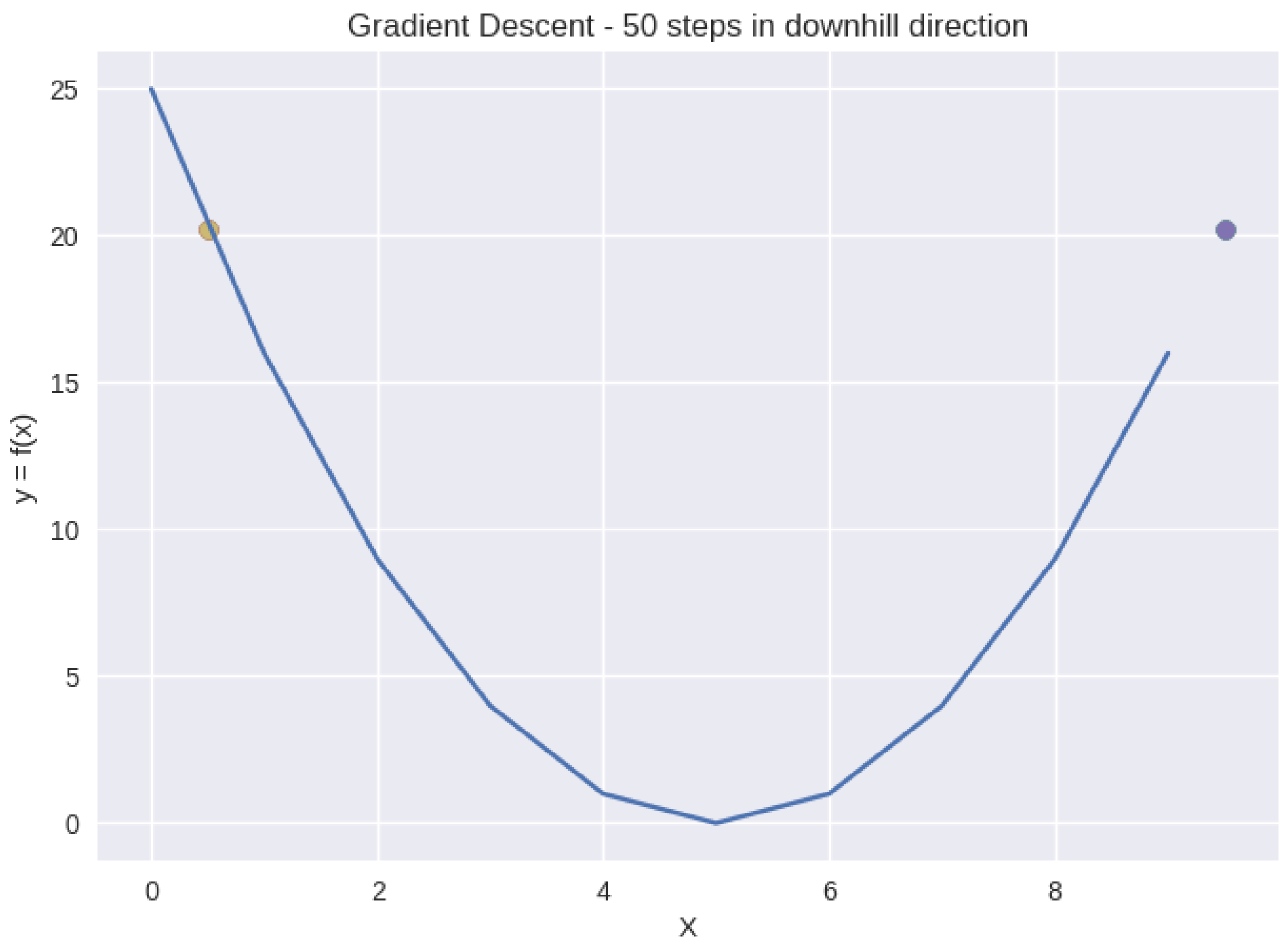
```
x = 0.5

y = (x-5)**2
plt.plot(X,Y)

plt.scatter(x,y)

plt.xlabel("X")
plt.ylabel("y = f(x)")
plt.title("Initial Guess")
plt.show()
```



This code initializes X to be 0.5 and demonstrates the effect of gradient descent on the cost function over a small number of iterations. You can observe how the algorithm updates X and the corresponding values of the cost function. Adjust the learning_rate variable to see the impact on convergence.

Gradient Descent Visualization - Without Learning Rate

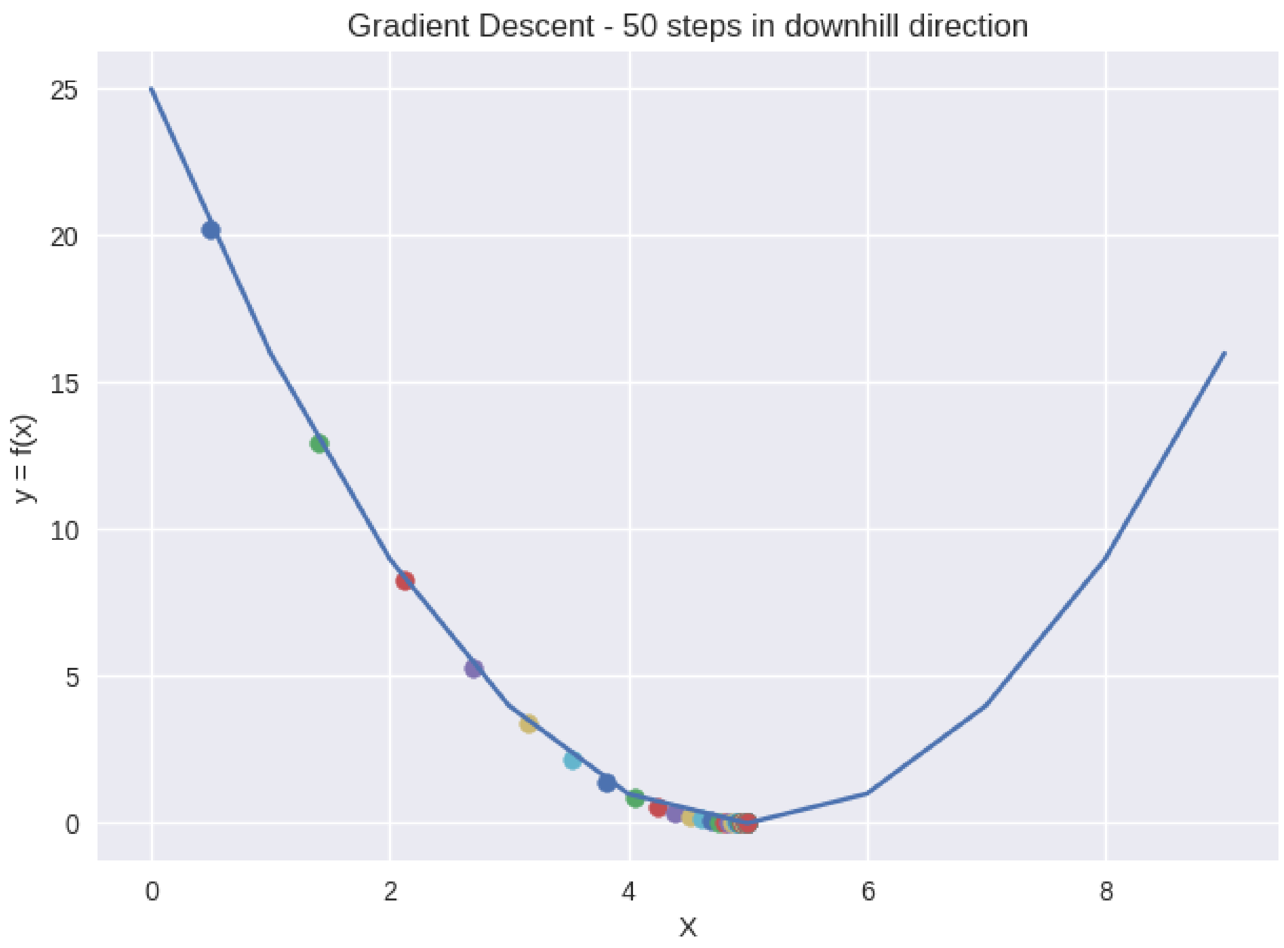
```
# In this we are initializing x to 0.5, representing the starting point on the co
# It iterates 10 times, updating x directly using the gradient of the cost functi
# The code plots the cost function and the points where each update occurs.

import time

fig = plt.figure()
ax = fig.add_subplot(111)
plt.ion()

x = 0.5

y = (x-5)**2
plt.plot(X,Y)
plt.scatter(x,y)
plt.xlabel("X")
plt.ylabel("y = f(x)")
plt.title("Gradient Descent - 50 steps in downhill direction")
# plt.show()

lr = 0.1 # not using learning rate in this part -> this is without learning rate
errors = []

for i in range(10):
  grad = 2*(x-5)
  x = x- grad
  y = (x-5)**2
  error = y - 0
  errors.append(error)
  plt.scatter(x, y)
  fig.canvas.draw()
  time.sleep(0.5)
plt.show()
```



The above code shows a rapid decrease in the cost function values but exhibits a large jump between consecutive iterations.

Gradient Descent Visualization - With Learning Rate

```
# Similar to the above code, we are initializing x to 0.5.
# It iterates 50 times, but this time updates x using the learning rate (lr) mult
# The learning rate controls the step size in the downhill direction.

import time

fig = plt.figure()
ax = fig.add_subplot(111)
plt.ion()

x = 0.5

y = (x-5)**2
plt.plot(X,Y)
plt.scatter(x,y)
plt.xlabel("X")
plt.ylabel("y = f(x)")
plt.title("Gradient Descent - 50 steps in downhill direction")
# plt.show()

lr = 0.1
errors = []

for i in range(50):
  grad = 2*(x-5)
  x = x- lr*grad
  y = (x-5)**2
  error = y - 0
  errors.append(error)
  plt.scatter(x, y)
  fig.canvas.draw()
  time.sleep(0.5)
plt.show()
```



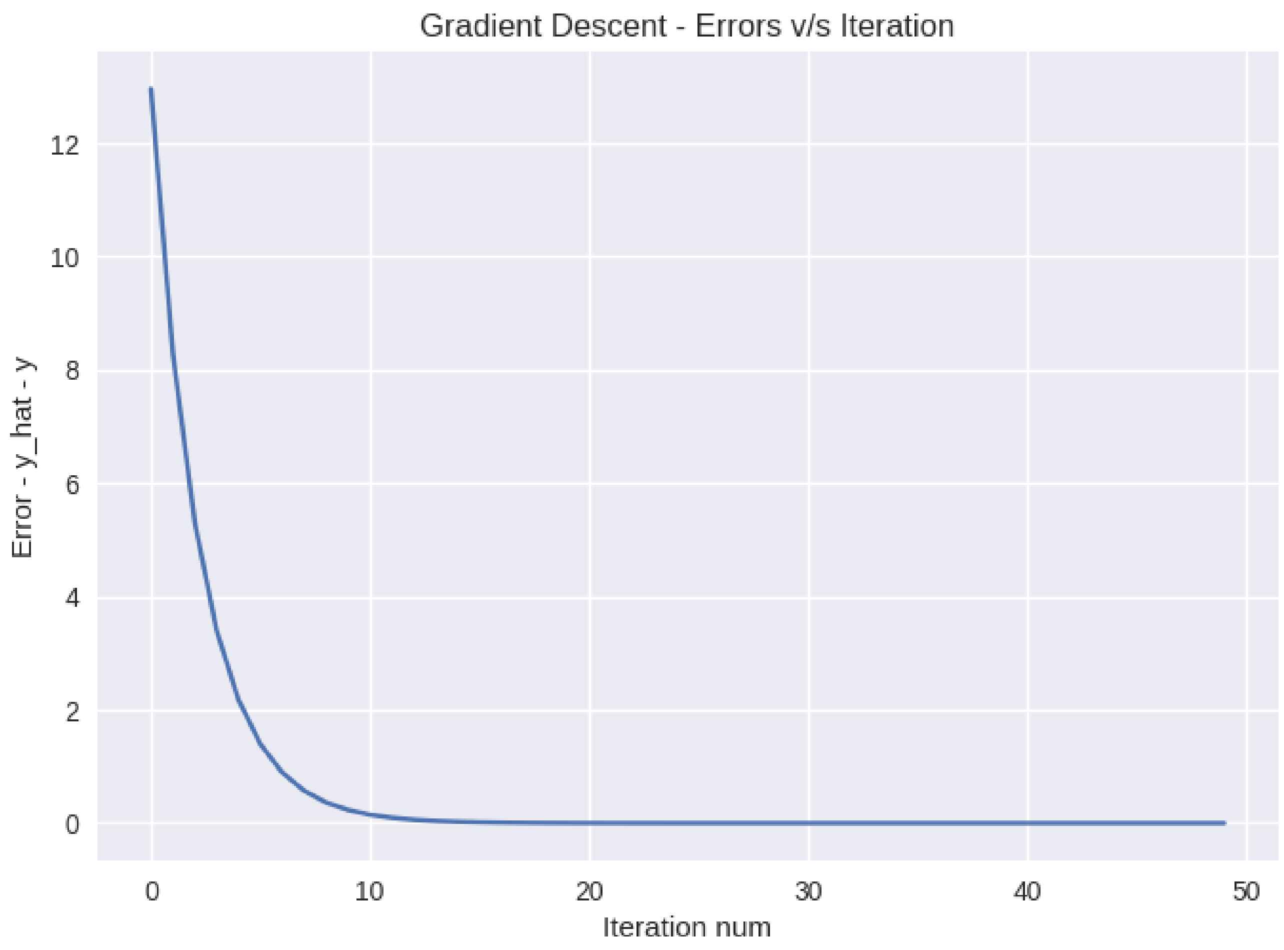
The code shows a smoother descent with smaller steps, allowing the algorithm to approach the minimum point more gradually.

The key difference is the introduction of the learning rate in the code with learning rate, which helps control the step size during each iteration.

The codew without learning rate updates x directly without considering the impact of the step size, leading to larger jumps.

The code with the learning rate results in a smoother convergence, preventing overshooting and allowing the algorithm to reach the minimum point more effectively.

```
plt.plot(errors)
plt.xlabel("Iteration num")
plt.ylabel("Error - y_hat - y")
plt.title("Gradient Descent - Errors v/s Iteration")
plt.show()
```



when to stop gradient descent iterations is crucial:

Gradient descent is an iterative optimization algorithm used to minimize a cost or loss function iteratively. Deciding when to halt these iterations is a critical aspect of the process. If the algorithm stops too early, it may not reach the optimal solution, and if it continues for too long, it might waste computational resources.

A fixed number of iterations is often set (e.g., 50):

One common approach is to predefine a fixed number of iterations. In practice, this number is a hyperparameter that can be adjusted based on experience or computational constraints. For example, setting the number of iterations to 50 means the algorithm will perform 50 updates to the model parameters.

Early stopping conditions are essential for efficiency:

While fixing the number of iterations is a straightforward strategy, it may not always be the most efficient. Early stopping conditions are introduced to dynamically halt the iterations based on the current state of the optimization. These conditions are designed to recognize when the algorithm has reached a satisfactory solution, making further iterations unnecessary.

Common early stopping conditions include:

Convergence Criteria: Monitoring the change in the cost function or model parameters. If the change falls below a predefined threshold, the algorithm can stop.

Validation Set Performance: Assessing the model's performance on a separate validation set. If the performance ceases to improve or starts degrading, the algorithm can be stopped.

Maximum Iterations: A safety net to prevent excessively long computations. If the algorithm hasn't converged by a certain predefined number of iterations, it halts to prevent prolonged execution.

Determining when to stop gradient descent involves finding a balance between computational efficiency and achieving optimal results. A fixed number of iterations provides a baseline, but incorporating early stopping conditions ensures adaptability and prevents unnecessary computations.

Use Case Simulation

Linear Regression Model Development Steps:

- Develop a univariate linear regression model to predict the selling price based on the input variable, max power.
- Define three functions: linear hypothesis, gradient computation, and error computation.
- Implement gradient descent to optimize the model parameters (Theta 0 and Theta 1).
- Stopped gradient descent after a fixed number of iterations (e.g., 50).
- Explain the importance of early stopping conditions for efficiency.

```
X = df["max_power"]
Y = df["selling_price"]

# Normalisation -> standard Normalisation
u = X.mean()
std = X.std()

X = (X-u)/std
```

Normalization, particularly standard normalization (subtracting mean and dividing by standard deviation), is often used to make computation feasible.

Dealing with smaller numbers helps in handling large differences in magnitudes between input variables, leading to better performance during gradient descent.

Linear Regression Hypothesis Function

```
def hypothesis(x, theta):
    # Linear hypothesis function: h(x) = theta[0] + theta[1]*x
    y_hat = theta[0] + theta[1]*x
    return y_hat
```

Gradient Computation for Linear Regression

```
def gradient(X, Y, theta):
    m = X.shape[0]
    grad = np.zeros((2,))  # Initializing gradient vector with zeros for Theta 0

    for i in range(m):
        x = X[i]  # Current input
        y_hat = hypothesis(x, theta)  # Computing hypothesis for the current inpu
        y = Y[i]  # Actual output

        # Updating gradients for Theta 0 and Theta 1
        grad[0] += (y_hat - y)  # Partial derivative of cost with respect to Thet
        grad[1] += (y_hat - y)*x  # Partial derivative of cost with respect to Th

    return grad/m  # Average gradient over all examples
```

grad[0]:

- The partial derivative of the cost with respect to `Theta 0` is given by: $\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})$

grad[1]:

- The partial derivative of the cost with respect to `Theta 1` is given by: $\frac{\partial J(\theta)}{\partial \theta_1} = \frac{1}{m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$

These derivatives are computed within the gradient function and are fundamental for updating the parameters during the gradient descent process. They capture the sensitivity of the cost function to changes in Theta 0 and Theta 1

Error Calculation for Linear Regression

```python
def error(X, Y, theta):
    m = X.shape[0]  # Number of examples in the dataset
    total_error = 0.0  # Initialize the total error

    for i in range(m):
        x = X[i]  # Current input
        y_hat = hypothesis(x, theta)  # Compute hypothesis for the current input
        y = Y[i]  # Actual output

        # Compute the squared error for the current example
        error_i = (y_hat - y)**2
        total_error += error_i  # Accumulate the squared error

    return (total_error/m)  # Return the mean squared error
```

Gradient Descent Optimization for Linear Regression

```python
def gradient_descent(X, Y, max_steps=100, learning_rate=0.1):
    theta = np.zeros((2,))  # Initialize weights Theta 0 and Theta 1 to zero
    error_list = []  # List to store errors during iterations
    theta_list = []  # List to store Theta values during updates

    for i in range(max_steps):
        # Compute gradient using the gradient function
        grad = gradient(X, Y, theta)

        # Compute error using the error function
        e = error(X, Y, theta)

        # Update weights Theta 0 and Theta 1 using the learning rate and gradient
        theta[0] = theta[0] - learning_rate * grad[0]
        theta[1] = theta[1] - learning_rate * grad[1]

        # Store the updated Theta values during each iteration
        theta_list.append((theta[0], theta[1]))
        # Store the error during each iteration
        error_list.append(e)

    return theta, error_list, theta_list
```

```python
theta, error_list, theta_list = gradient_descent(X, Y, max_steps=50)
print(theta)
```

```
[264899.50073976  63179.87873954]
```

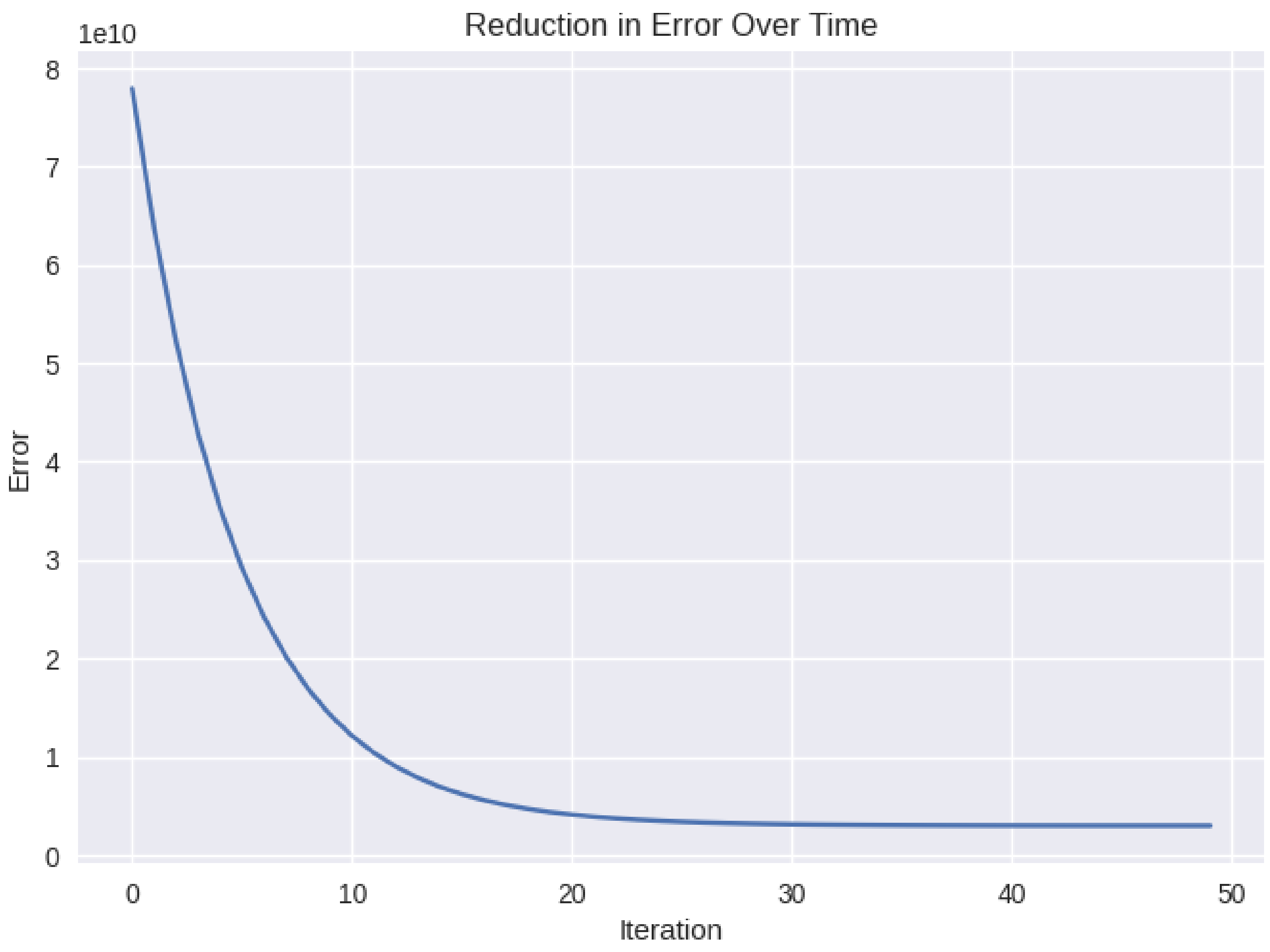
theta0 = 264899.50073976 theta1 = 63179.87873954

Y = theta0 + theta1.X

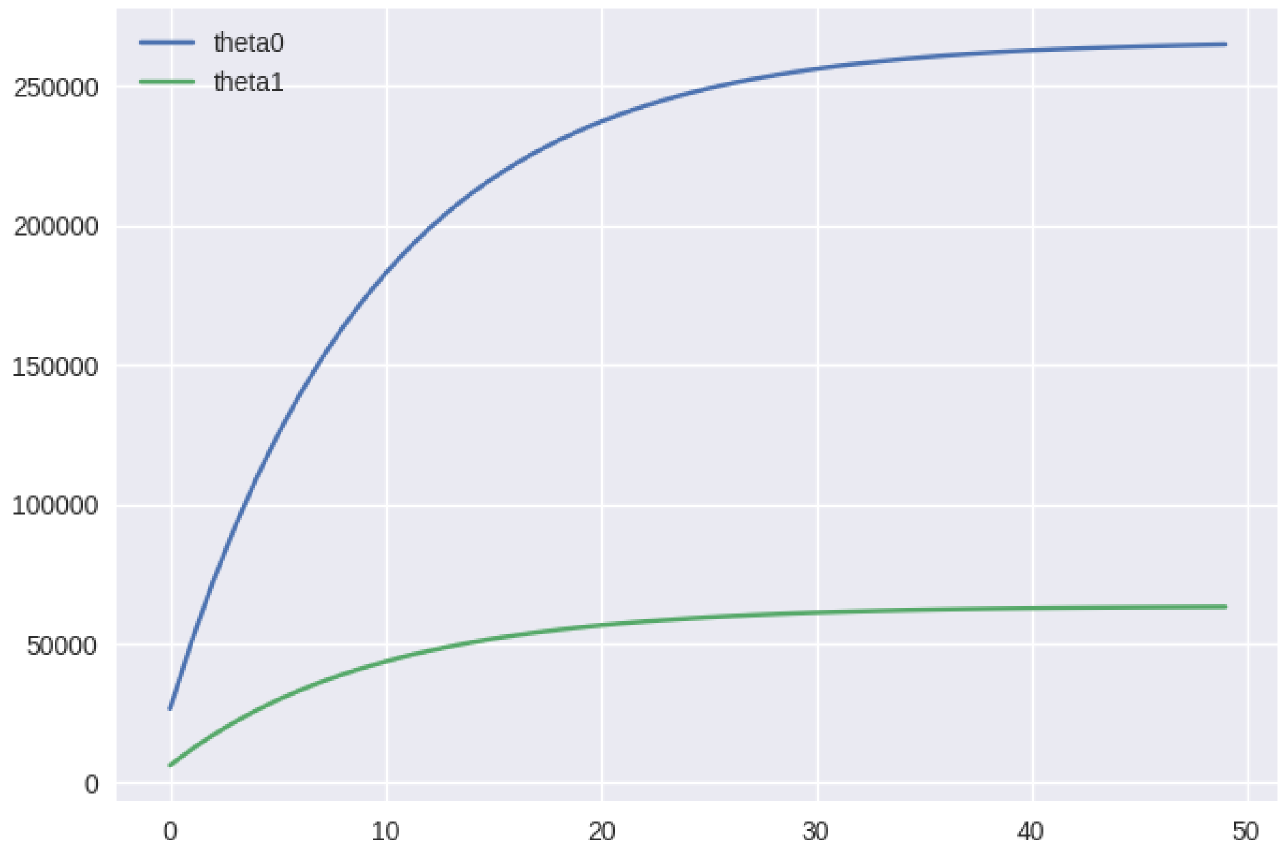
Here, X is normalised

Animate Visualization of Linear Regression Training

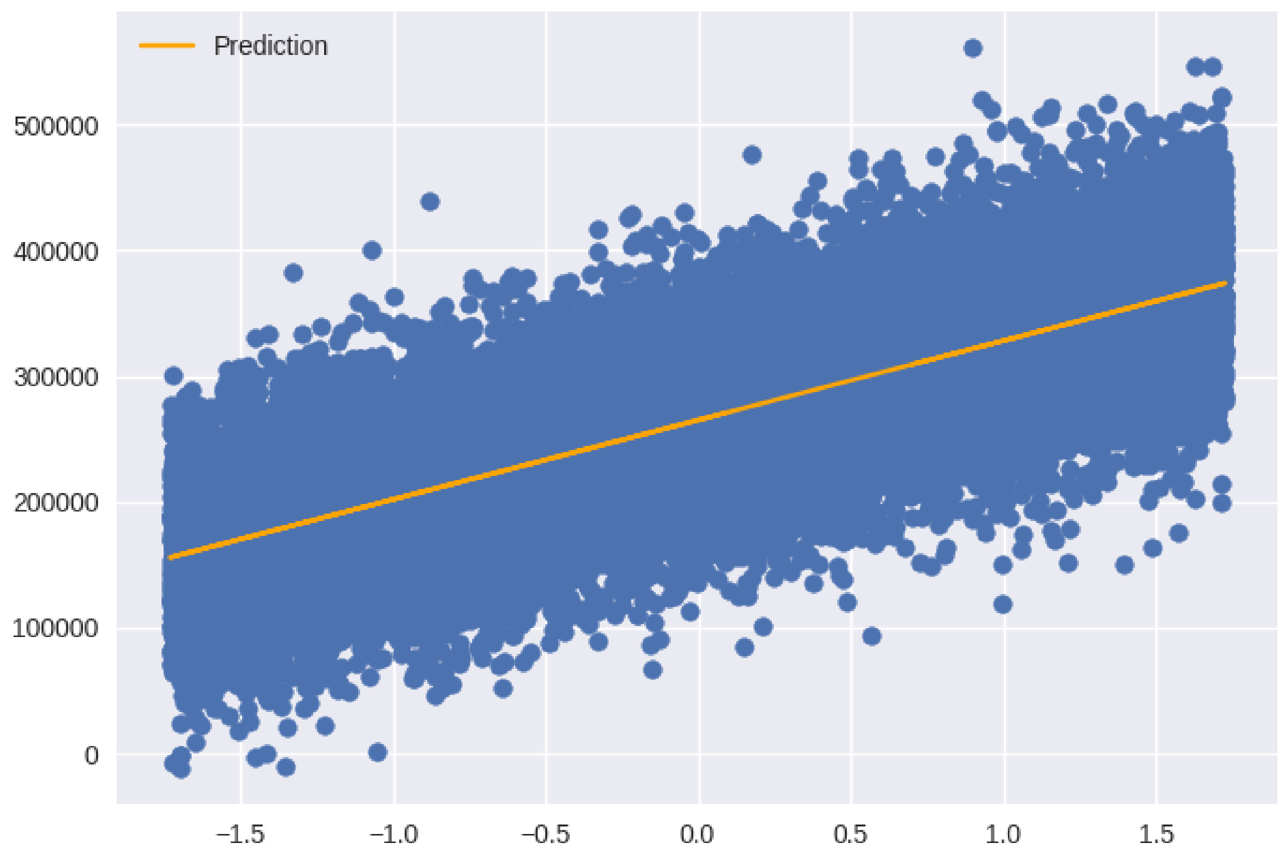
```
# Initialize a new figure for the plot
fig = plt.figure()

# Plot the error values over iterations
plt.plot(error_list)

# Set the title and axis labels for better interpretation
plt.title('Reduction in Error Over Time')
plt.xlabel('Iteration')
plt.ylabel('Error')

# Display the plot
plt.show()
```



```
# Initialize a new figure for the plot
fig = plt.figure()

# Extract Theta 0 and Theta 1 values from the 'theta_list'
theta_list = np.array(theta_list)

# Plot the changes in Theta 0 and Theta 1 over iterations
plt.plot(theta_list[:, 0], label="theta0")
plt.plot(theta_list[:, 1], label="theta1")

# Add a legend for better interpretation
plt.legend()

# Display the plot
plt.show()
```



```
# Initialize a new figure for the plot
fig = plt.figure()

# Compute predicted values using the trained model parameters
Y_hat = hypothesis(X, theta)

# Scatter plot of the original data points
plt.scatter(X, Y)

# Plot the regression line representing the model's predictions in orange
plt.plot(X, Y_hat, color='orange', label='Prediction')

# Add a legend for better interpretation
plt.legend()

# Display the plot
plt.show()
```



Metrics for Evaluating Regression Models

1. Accuracy (for Classification Problems):
    - In machine learning, classification is a type of task where the goal is to assign predefined labels or categories to input data points. Accuracy is a commonly used metric to evaluate the performance of a classification model.

- Accuracy measures how many predictions made by the model are correct out of the total number of predictions.
- The formula for accuracy is expressed as:

  Accuracy =Total Number of Predictions/Number of Correct Predictions

2. Error Metrics for Regression:
   - Mean Absolute Error (MAE): Average of absolute differences between predicted and actual values.
   - Mean Squared Error (MSE): Average of squared differences between predicted and actual values.
   - Root Mean Squared Error (RMSE): Square root of MSE, brings the error to the original scale.
   - Mean Absolute Percentage Error (MAPE): Average of absolute percentage differences between predicted and actual values.
3. R Square:
   - R Square (R^2) or Coefficient of Determination.
   - Measures the amount of variance in the dependent variable (y) explained by the independent variable (X)
   - Formula:

     R^2 = 1 - (SSR/TSS)

     SSR is the Residual Sum of Squares.

     TSS is the Total Sum of Squares.
   - R Square varies from 0 to 1.
   - Closer to 1 indicates a good model, closer to 0 is less ideal.
   - Can theoretically take values between -1 to 1, but practically between 0 to 1.
   - R Square tells how well the model explains the variability in y using the variation in X.
     - Ideal model: R^2 = 1 (perfect predictions).
     - Poor model: R^2 = 0 (no explanatory power).
     - Negative R^2 (rare) indicates an inverted relationship.
   - R square is calculated as:

     1 - Total Sum of Squares/Residual Sum of Squares

     Residual sum of squares is the sum of squared differences between actual (Y) and predicted (Y_hat) values.

     Total sum of squares involves squared differences between actual (Y) values and their mean.

R-squared Score Calculation

```
import numpy as np

def r2_score(Y, Y_hat):
    # Calculate the sum of squared differences between actual and predicted value
    num = np.sum((Y - Y_hat)**2)

    # Calculate the sum of squared differences between actual values and their me
    denom = np.sum((Y - np.mean(Y))**2)

    # Compute the R-squared score
    score = 1 - num / denom

    return score
```

```
r2_score(Y, Y_hat)
```

```
0.5723904152416218
```

3D Visualization of Cost Function

```
# Import necessary libraries
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

# Define a range of values for Theta 0 and Theta 1
T0 = np.arange(-100, 100, 1)
T1 = np.arange(-100, 100, 1)

# Create a meshgrid of Theta 0 and Theta 1 values
T0, T1 = np.meshgrid(T0, T1)
J = np.zeros(T0.shape)

# Calculate the cost (J) for each combination of Theta 0 and Theta 1
for i in range(J.shape[0]):
    for j in range(J.shape[1]):
        Y_hat = T1[i, j] * X + T0[i, j]
        J[i, j] = np.sum(((Y - Y_hat) ** 2) / Y.shape[0])

# Create a 3D plot of the cost function
fig = plt.figure()
axes = fig.add_subplot(111, projection='3d')
axes.plot_surface(T0, T1, J, cmap='rainbow')

# Label the axes for better interpretation
axes.set_xlabel("theta0")
axes.set_ylabel("theta1")
axes.set_zlabel("J(theta0, theta1)")

# Display the 3D plot
plt.show()
```

Stochastic Gradient Descent (SGD):

- Stochastic Gradient Descent is a variation of Gradient Descent that updates the weights