

MUHAMMAD TAYYAB

MACHINE LEARNING

April 12, 2025



Gradient Descent

# REPORT

## GRADIENT DESCENT USING LINEAR AND POLYNOMIAL REGRESSION

Department of Computer Science

# Predictive Modeling of Salary and House Prices via Gradient Descent

## Abstract:

---

This report presents a detailed implementation and analysis of two regression models—univariate linear regression and polynomial regression—using gradient descent optimization. The first task focuses on predicting salaries based on years of experience using a simple linear regression model. The dataset `salary_data.csv` was used, and gradient descent was applied to minimize the cost function, resulting in an accurate and interpretable model. Visualizations were employed to demonstrate model convergence and the impact of different learning rates.

The second task involves predicting house prices using polynomial regression on the `kc_house_data.csv` dataset. A second-degree polynomial model was trained using manually implemented batch gradient descent. Feature normalization and polynomial transformation were applied to improve model performance and training stability. The polynomial model successfully captured the non-linear relationship between living area and house price, and was evaluated using cost convergence plots and prediction accuracy.

Together, these tasks showcase the effectiveness of gradient descent in training both linear and non-linear regression models, highlighting the importance of feature scaling, learning rate tuning, and model selection in achieving accurate predictions.

# Question 1: Salary Prediction using Univariate Linear Regression and Gradient Descent

---

## Dataset:

---

The dataset used for this task is titled `Salary_data.csv` and contains 30 records, each representing a professional individual with their corresponding years of experience and salary. It includes the following features:

- **YearsExperience:** A numerical value indicating the total number of years an individual has worked.
- **Salary:** The annual income of the individual in numeric form (in local currency).


## Data Preprocessing

---

### 1. Loading dataset and checking columns

```
[2]
# loading dataset
df = pd.read_csv('Salary_data.csv')

# checking columns
df.columns
```

 `Index(['Unnamed: 0', 'YearsExperience', 'Salary'], dtype='object')`

### 2. Scatter Graph on Data

*A simple scatter plot shows how salary increases with more experience.*

```
# Plot original data
plot_scatter(x, y,
             title="Years of Experience vs Salary (Original)",
             xlabel="Years of Experience (Original)",
             ylabel="Salary (Original)",
             color='blue')
```



### 3. Normalization

Scaled all X and y values between 0 and 1 so gradient descent works better and faster. Helpful when dataset is too large and its also helpful to converge gradient descent faster.

```
# Extract features and target
X = df['YearsExperience'].values.reshape(-1, 1) # Reshape for sklearn
y = df['Salary'].values.reshape(-1, 1)

# Normalization (scaling to [0,1])
normalizer = MinMaxScaler()
X_normalized = normalizer.fit_transform(X)
y_normalized = normalizer.fit_transform(y)

print("Original X (first 5):", X[:5].flatten())
print("Normalized X (first 5):", X_normalized[:5].flatten())
```

Original X (first 5): [1.2 1.4 1.6 2.1 2.3]  
 Normalized X (first 5): [0.00212766 0.04255319 0.09574468 0.11702128]

### 4. Standardization

It can also be the option when normalization don't work. *This scales X and y to have mean = 0 and standard deviation = 1. Good when data has outliers.*

```
[7] # Standardization (zero mean, unit variance)
standardizer = StandardScaler()
X_standardized = standardizer.fit_transform(X)
y_standardized = standardizer.fit_transform(y)

print("Original X (first 5):", X[:5].flatten())
print("Standardized X (first 5):", X_standardized[:5].flatten())
```

Original X (first 5): [1.2 1.4 1.6 2.1 2.3]  
 Standardized X (first 5): [-1.51005294 -1.43837321 -1.36669348 -1.18749416 -1.11581443]

# Gradient Descent Implementation

## Algorithm Overview:

*Gradient Descent is an optimization algorithm used to minimize the cost or error in machine learning models. It works by gradually adjusting the model's parameters (like slope and intercept in linear regression) to find the values that best fit the data.*

- We calculate the gradient (slope) of the cost function.
- This tells us the direction and how fast the cost is increasing.
- We move in the opposite direction (downhill) by subtracting the gradient, scaled by a factor called the learning rate ( $\alpha$ ).

Mathematically:

*The slope ( $\beta_1$ ) and intercept ( $\beta_0$ ) are updated like this:*

- $\beta_1 = \beta_1 - \alpha * (\partial \text{Cost} / \partial \beta_1)$
- $\beta_0 = \beta_0 - \alpha * (\partial \text{Cost} / \partial \beta_0)$

This process repeats for many iterations until the change in cost becomes very small or we reach max iterations

## 1. Purpose of Important Variables Used in GD:

Variable	Purpose
<b>alpha</b>	The learning rate ( $\alpha = 0.005, 0.01, 0.05$ ). Controls step size for $\beta$ updates. Too big (0.05) caused nan; 0.01 nailed it in my plot_multiple_cost_histories (red curve).
<b>tolerance</b>	A tiny value to check if cost stopped dropping (e.g., $\text{cost\_new} - \text{prev\_cost} < \text{tolerance}$ ). It would stop early if converged—could've saved time!
<b>m</b>	Number of data points (30 for salary_data.csv). Divides gradients and cost (e.g., $(2/m) * \text{sum}(\dots)$ ). Ensures updates and cost_new are averages, not sums.
<b>beta_1</b>	The slope ( $\beta_1$ )—how much Salary changes per YearsExperience. Starts at 0, ends ~9450 (for $\alpha = 0.01$ ). Drives the steepness in plot_regression_fit.
<b>beta_0</b>	The intercept ( $\beta_0$ )—base Salary when YearsExperience = 0. Goes from 0 to ~24848. Shifts my line up/down in plots.
<b>Y_pred</b>	Predicted Salary ( $\beta_1 * X + \beta_0$ ) for current $\beta$ values. Compared to $y$ to compute gradients—drives error in cost_new.
<b>gradient_beta_1</b>	Partial derivative for $\beta_1$ : $(2/m) * \text{sum}((Y\_pred - y) * X)$ . Tells me how to nudge $\beta_1$ . Big early, shrank as I neared $\beta_1 \approx 9450$ .

<b>gradient_beta_0</b>	Partial derivative for $\beta_0$ : $(2/m) * \sum(Y_{\text{pred}} - y)$ . Nudges $\beta_0$ toward ~24848. Positive early (pushed $\beta_0$ up), like I explained before.
<b>cost_new</b>	New cost $((1/(2*m)) * \sum((Y_{\text{pred\_new}} - y)^2))$ . Added to cost_history. Dropped to 1.56e7 for $\alpha = 0.01$ , showed convergence in my plots.

### 3. Model Training: (Code is in Notebook Section Gradient Descent Algorithm)

In notebook it's also run on standardized and normalized data and both took too less iterations to converge because dataset is too low.

```

Calling GD using original data

# calling GD
result = gradient_descent(X, y, alpha=0.01, max_iterations=5000, tolerance=1e-3)
cost_history_orig = result['cost_history']
beta_0_history_orig = result['beta_0_history']
beta_1_history_orig = result['beta_1_history']
beta_0_orig = result['beta_0']
beta_1_orig = result['beta_1']
iterations_orig = result['iterations']
time_taken_orig = result['time_taken']

Iteration 1: Cost = 257309737.030703, beta_1 = 9700.0883, beta_0 = 1520.0800
Iteration 2: Cost = 81062156.086370, beta_1 = 12040.2064, beta_0 = 1959.5622
Iteration 3: Cost = 70292500.301936, beta_1 = 12596.8748, beta_0 = 2136.8979

Iteration 2411: Cost = 15635475.992491, beta_1 = 9450.1265, beta_0 = 24847.0853
Iteration 2412: Cost = 15635475.991414, beta_1 = 9450.1258, beta_0 = 24847.0899
Iteration 2413: Cost = 15635475.990345, beta_1 = 9450.1252, beta_0 = 24847.0945
Iteration 2414: Cost = 15635475.989285, beta_1 = 9450.1245, beta_0 = 24847.0990
Iteration 2415: Cost = 15635475.988234, beta_1 = 9450.1238, beta_0 = 24847.1036
Iteration 2416: Cost = 15635475.987192, beta_1 = 9450.1232, beta_0 = 24847.1081
Iteration 2417: Cost = 15635475.986158, beta_1 = 9450.1225, beta_0 = 24847.1126
Iteration 2418: Cost = 15635475.985132, beta_1 = 9450.1218, beta_0 = 24847.1171
Iteration 2419: Cost = 15635475.984115, beta_1 = 9450.1212, beta_0 = 24847.1216
Iteration 2420: Cost = 15635475.983106, beta_1 = 9450.1205, beta_0 = 24847.1260
Iteration 2421: Cost = 15635475.982106, beta_1 = 9450.1199, beta_0 = 24847.1304
Iteration 2422: Cost = 15635475.981113, beta_1 = 9450.1192, beta_0 = 24847.1349

Converged at iteration 2422 with cost = 15635475.981113

```

### 4. Model Training and predict salary function:

```

Running Gradient Descent on Original Data

[45] # Running gradient descent on original data
result = gradient_descent(X, y, alpha=0.01, max_iterations=3000, tolerance=1e-6, verbose=False)
b0 = result['beta_0']
b1 = result['beta_1']

Predict salary function

# Function to predict salary based on input experience (no scaling)
def predict_salary(years):
    return b1 * years + b0

```

## 5. Code Assistant Console Interface:

### Console Assistant

```
# Console Assistant
print("\n Welcome to the Salary Prediction Assistant!")
print("-----")
print("Type a number to get a salary prediction based on years of experience.")
print("Type 'help' for gradient descent explanation or 'exit' to quit.\n")

while True:
    user_input = input("> ")
    if user_input.lower() == "exit":
        print("Goodbye!")
        break
    elif user_input.lower() == "help":
        print("\n 🧠 Gradient Descent is an optimization algorithm that helps minimize the cost (error) of a model by updating its parameters step-by-step.")
        print("This model uses it to find the best line that predicts salary from years of experience.\n")
    else:
        try:
            years = float(user_input)
            prediction = predict_salary(years)
            print(f"\n 📊 Predicted Salary for {years} years of experience: ${prediction:,.2f}\n")
        except ValueError:
            print("\n ⚠ Please enter a valid number or command.\n")
```

Welcome to the Salary Prediction Assistant!

-----  
Type a number to get a salary prediction based on years of experience.  
Type 'help' for gradient descent explanation or 'exit' to quit.

> help

🧠 Gradient Descent is an optimization algorithm that helps minimize the cost (error) of a model by updating its parameters step-by-step.  
This model uses it to find the best line that predicts salary from years of experience.

> 2.9

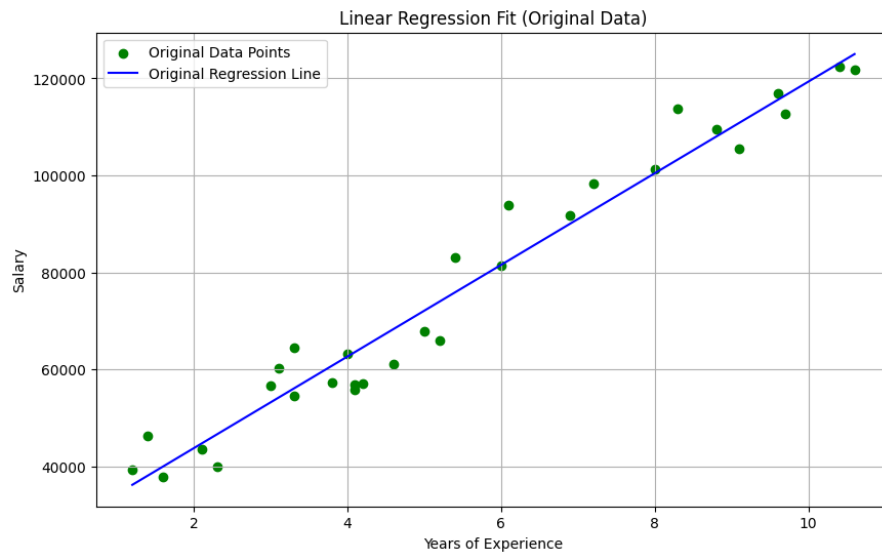
📊 Predicted Salary for 2.9 years of experience: \$52,253.04

>

# Visualization

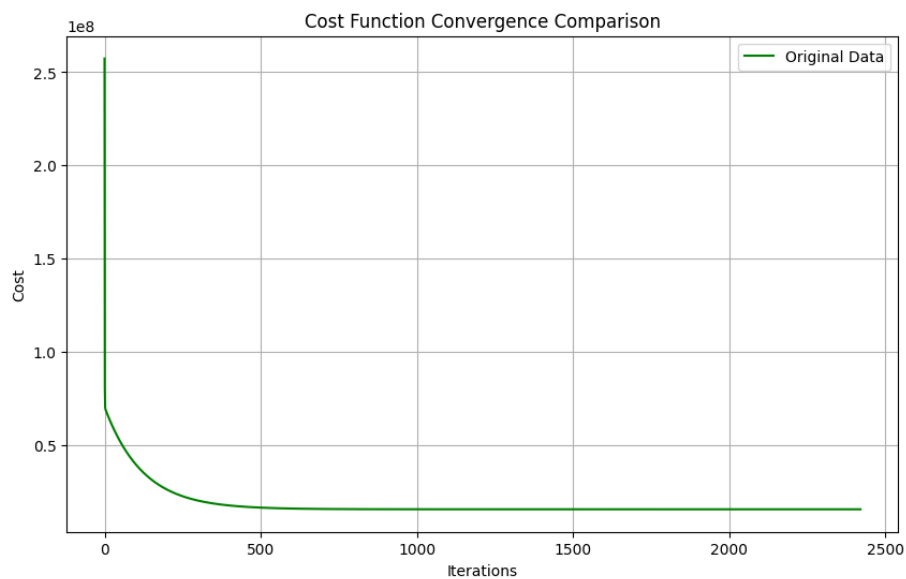
## 1. Plot Regression fit:

This function draws a line over the data points to show how well the model fits.



## 2. Plot Cost history function:

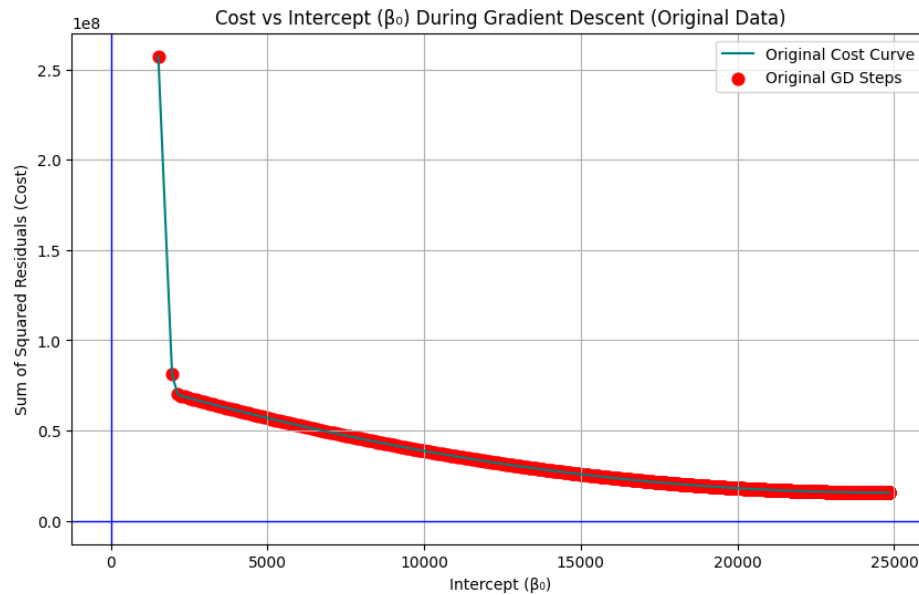
This function draws how cost changes over iterations (lower = better). Helps visualize learning speed.





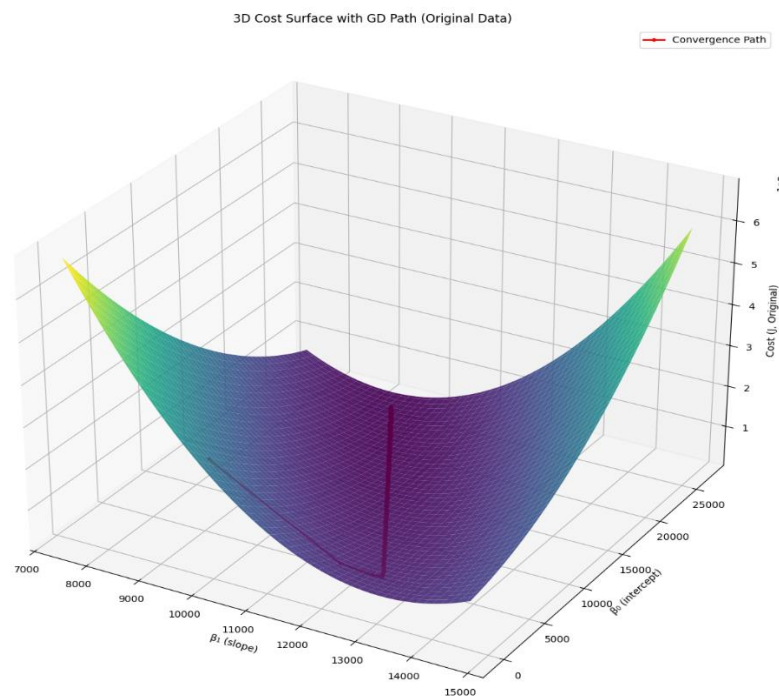
### 3. Cost vs Intercept:

Shows how cost drops as intercept ( $\beta_0$ ) gets updated. It helps us understand optimization visually.



### 4. 3-D Cost function surface with gradient descent path

It's an amazing 3D plots show the full cost surface and how the algorithm moves step by step to the best point in a 3D world. Wireframe is also plotted in which line is much clearer can see there.



## 5. Positive and Negative adjustments on Gradient Descent:

To visually and mathematically show how using  $+\alpha$  or  $-\alpha$  in the gradient update formula affects the convergence of gradient descent.

Here are the two equations that's referenced in assignment question:

1.  $\beta_j = \beta_j - \alpha * (\hat{y} - y) * x_j$

- This is the correct update rule for gradient descent.
- It moves  $\beta_j$  in the direction that reduces the cost.

2.  $\beta_0 = \beta_0 + \alpha * (\hat{y} - y) * x_j$  \*In assignment there is  $\beta_j$  which is a typo mistake. So, corrected\*

This equation uses a positive adjustment, which means you're updating in the wrong direction. Instead of minimizing error, you increase it — the model diverges.

- Allows switching between  $-\alpha$  (default) and  $+\alpha$  (wrong).
- Stores the history of slope ( $\beta_1$ ), intercept ( $\beta_0$ ), and cost at each step.

Then I ran:

- `run_gradient_descent_adj(..., positive_adjustment=False)` # Normal GD
- `run_gradient_descent_adj(..., positive_adjustment=True)` # Inverse GD

This lets me compare and visualize both behaviors.

## 6. Changes made in original GD:

```
gradient_beta_1 = (2/m) * np.sum((Y_pred - y) * X)
gradient_beta_0 = (2/m) * np.sum((Y_pred - y))

if positive_adjustment:
    beta_1 = beta_1 + alpha * gradient_beta_1
    beta_0 = beta_0 + alpha * gradient_beta_0
else:
    beta_1 = beta_1 - alpha * gradient_beta_1
    beta_0 = beta_0 - alpha * gradient_beta_0
```

## 7. Calls for regression subplot:

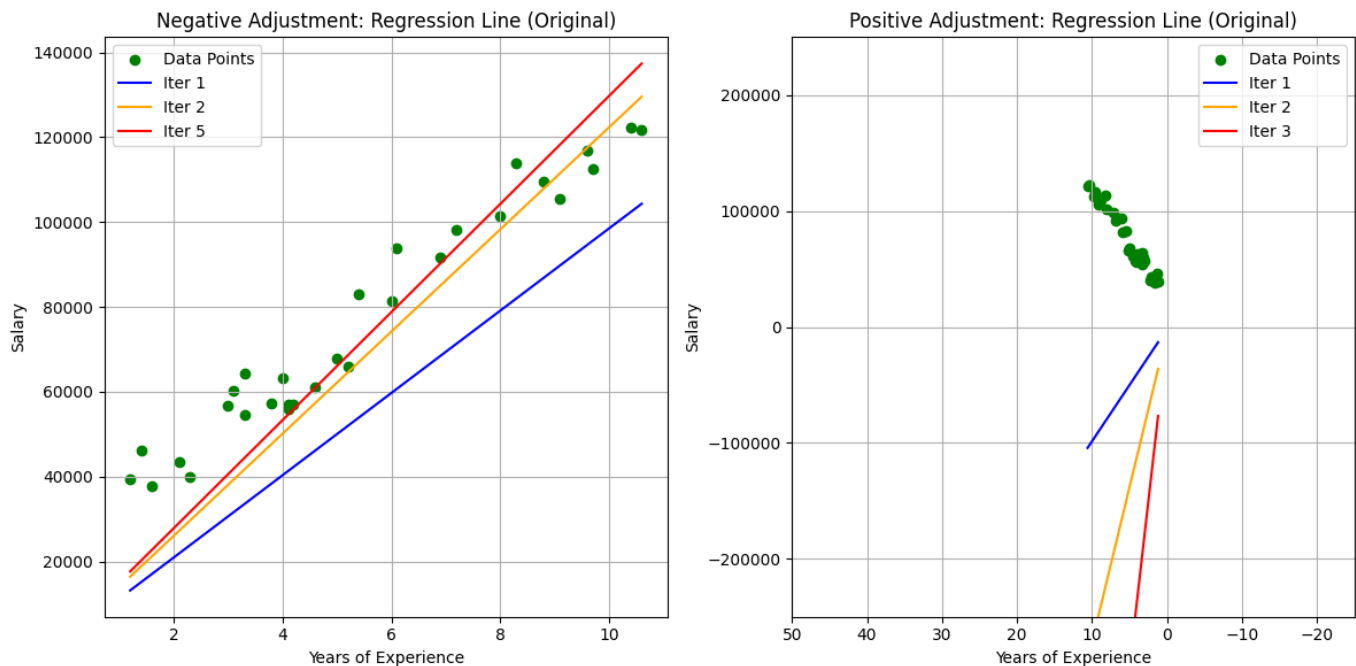
These are side-by-side plots:

- Left = Negative Adjustment ( $-\alpha$ ):
  - The regression line gradually fits the data better with each iteration.
  - Slope ( $\beta_1$ ) and intercept ( $\beta_0$ ) stabilize over time.
  - Model converges to minimum cost.
- Right = Positive Adjustment ( $+\alpha$ ):
  - The regression line goes far away from the data points.
  - Parameters ( $\beta_0$ ,  $\beta_1$ ) explode quickly (divergence).
  - Model fails to converge — it gets worse over time.

These clearly demonstrate:

- Correct sign ( $-\alpha$ ) leads to successful learning.
- Wrong sign ( $+\alpha$ ) leads to divergence and useless predictions.

This function draws how cost changes over iterations (lower = better). Helps visualize learning speed.



## 8. Parameter Evolution Subplot:

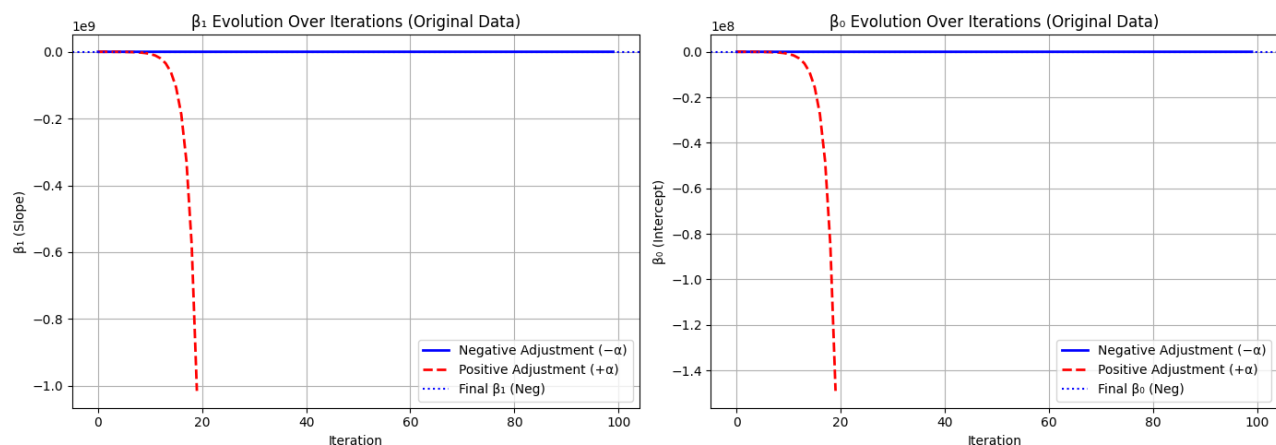
The curves show how  $\beta_0$  and  $\beta_1$  change over time for both  $+\alpha$  and  $-\alpha$ .

- In negative adjustment:
  - Both  $\beta_0$  and  $\beta_1$  approach fixed values smoothly.
- In positive adjustment:
  - Parameters grow rapidly and don't settle it's a sign of instability.

Below in some cases negative adjustment also leads to instability this is because data shape is not good enough and other results are not good enough also. So don't be sad.

### Calls for parameter evolution plots

```
# 1. Original Data
plot_parameter_evolution(
    beta_1_neg=beta_1_neg_orig,
    beta_0_neg=beta_0_neg_orig,
    beta_1_pos=beta_1_pos_orig,
    beta_0_pos=beta_0_pos_orig,
    title_beta1=" $\beta_1$  Evolution Over Iterations (Original Data)",
    title_beta0=" $\beta_0$  Evolution Over Iterations (Original Data)",
    ylabel_beta1=" $\beta_1$  (Slope)",
    ylabel_beta0=" $\beta_0$  (Intercept)"
)
```



# Analysis

---

## 1. Number of Iterations GD take to converged:

*Number of iteration gradient descent take to converged*

```
[ ] print("Total Iterations (Original Data):", iterations_orig)
    print("Total Iterations (Normalized Data):", iterations_norm)
    print("Total Iterations (Standardized Data):", iterations_std)
```

```
⇒ Total Iterations (Original Data): 2422
   Total Iterations (Normalized Data): 39
   Total Iterations (Standardized Data): 74
```

## 2. Time taken to run all Iterations:

*Time taken to run all iterations*

```
[ ] print("Time Taken for GD on Original Data (s):", time_taken_orig)
    print("Time Taken for GD on Normalized Data (s):", time_taken_norm)
    print("Time Taken for GD on Standardized Data (s):", time_taken_std)
```

```
⇒ Time Taken for GD on Original Data (s): 0.16188335418701172
   Time Taken for GD on Normalized Data (s): 0.003634214401245117
   Time Taken for GD on Standardized Data (s): 0.0075511932373046875
```

## 3. Identifying and discussing factors that influence the outcome and performance of algorithm.

I have used many factors but more important in my code and visualizations are explained but some important factors clearly mentioned and discussed below.

### 3.1. Learning Rate

The learning rate controls how big the step is in each iteration when updating the slope and intercept.

- If it's too small (like 0.005), it takes more iterations to reach the minimum cost.
- If it's too big (like 0.05+), it can overshoot or even diverge (go to NaN/infinity).

Try different learning rate to find best one for you according to the data.

### 3.2. Data Preprocessing (Original vs Normalized vs Standardized)

Raw data sometimes has large values, which can confuse gradient descent and slow it down.

- Normalization scales data between 0 and 1
- Standardization makes data have mean = 0 and std = 1

#### Effect:

- Convergence is faster and more stable on normalized/standardized data.
- Cost curves are smoother for preprocessed data.
- Raw data took longer to converge (more iterations and time).

### 3.3. Initial Values for Slope and Intercept

Starting with slope = 0 and intercept = 0 might not be close to the best line.

- Depends upon the complexity of data. Best guess converges fast.
- The plotted initial guess line, didn't match the data well every time (in most cases). That's okay GD will improve it, but it affects how long convergence takes.

### 3.4. Direction of Update (Positive vs Negative Adjustment)

Normally, gradient descent subtracts the gradient to move downhill and reduce cost.

I also tested positive updates but it not goes to my fit. But in some cases, it might be the best option to go with so try with every possibility.

#### Subplot and evolution plots shown above:

- With  $+\alpha$ , the cost increased and parameters shot off to large values.
- With  $-\alpha$ , the line gradually improved and converged to a good fit.

#### Direction Matters!

### 3.5. Tolerance (Stopping Condition)

Tolerance defines when to stop. If the change in cost is very small.

#### Tolerance is:

- Too tight  $\rightarrow$  longer time, very precise.
- Too loose  $\rightarrow$  stops early, may not be optimal.

# Question 2: House Price Prediction using Polynomial Regression and Gradient Descent

---

## Dataset:

---

*The dataset used in this task is loaded from a CSV file named `kc_house_data.csv`. It includes information about houses sold in King County, USA etc. The dataset contains various features that describe the physical attributes of the houses, their condition, and the sale details. Below are the column names and their descriptions*

Column Name	Description
<code>id</code>	Unique identifier for each house listing.
<code>date</code>	Date when the house was sold.
<code>price</code>	💰 Selling price of the house (target variable to predict).
<code>bedrooms</code>	Number of bedrooms in the house.
<code>bathrooms</code>	Number of bathrooms, where 0.5 means a half bathroom.
<code>sqft_living</code>	Total interior living space in square feet.
<code>sqft_lot</code>	Total area of the lot in square feet.
<code>floors</code>	Number of floors in the house.
<code>waterfront</code>	1 if the house is on a waterfront, 0 otherwise.
<code>view</code>	Quality rating of the view from the house (0 to 4).
<code>condition</code>	Overall condition rating (1 = poor, 5 = excellent).
<code>grade</code>	Construction quality and design rating (1 to 13).
<code>sqft_above</code>	Living space above ground in square feet.
<code>sqft_basement</code>	Basement space in square feet.
<code>yr_built</code>	Year the house was built.
<code>yr_renovated</code>	Year the house was renovated (0 if never renovated).
<code>zipcode</code>	ZIP code of the property's location.
<code>lat</code>	Latitude coordinate (north-south location).
<code>long</code>	Longitude coordinate (east-west location).
<code>sqft_living15</code>	Living space of nearby houses (in square feet).
<code>sqft_lot15</code>	Lot size of nearby houses (in square feet).

# Data Preprocessing

---

## 1. Loading dataset and checking columns

```
df = pd.read_csv('kc_house_data.csv')
df.columns
```

```
Index(['id', 'date', 'price', 'bedrooms', 'bathrooms', 'sqft_living',
       'sqft_lot', 'floors', 'waterfront', 'view', 'condition', 'grade',
       'sqft_above', 'sqft_basement', 'yr_built', 'yr_renovated', 'zipcode',
       'lat', 'long', 'sqft_living15', 'sqft_lot15'],
      dtype='object')
```

## 2. Dropping unnecessary columns

```
[ ] # Drop unnecessary columns
columns_to_drop = ['id', 'date', 'zipcode', 'yr_renovated', 'sqft_lot', 'sqft_lot15']
df_cleaned = df.drop(columns=columns_to_drop)

df_cleaned.shape
```

```
(21613, 15)
```

## 3. Check for missing data

```
# Check for null/missing values
print(df_cleaned.isnull().sum())
```

```
price          0
bedrooms       0
bathrooms      0
sqft_living    0
floors         0
waterfront     0
view           0
condition      0
grade          0
sqft_above     0
sqft_basement  0
yr_built       0
lat            0
long           0
sqft_living15  0
dtype: int64
```



*Replace Null values with average (mean) values*

```
[ ] # if any becomes more than 0 in our case of missing data

df_cleaned['sqft_above'] = df_cleaned['sqft_above'].fillna(df_cleaned['sqft_above'].mean())
```

## 4. Check Data Types (Categorical vs Numerical)

All features are numerical (int, float) data type. These are exactly what we need for Numerical so no need for numerical encoding.

```
# Check data types
print(df_cleaned.dtypes)

# Identify categorical columns (if any remain)
categorical_cols = df_cleaned.select_dtypes(include='object').columns.tolist()
```

price	float64
bedrooms	int64
bathrooms	float64
sqft_living	int64
floors	float64
waterfront	int64
view	int64
condition	int64
grade	int64
sqft_above	int64
sqft_basement	int64
yr_built	int64
lat	float64
long	float64
sqft_living15	int64
dtype:	object

## 5. Handle duplicates

```
[ ] # Drop duplicate rows
df_cleaned = df_cleaned.drop_duplicates()
```

## 6. Corelation Check

Features like `sqft_living` (0.702035), `grade` (0.667434), and `sqft_above` (0.605567) have high correlation with `price`, contributing more to the prediction. Less correlated features like `long` (0.021626) may add noise.

```
[ ] # Check correlations with target
correlation = df_cleaned.corr()['price'].sort_values(ascending=False)
print(correlation)
```

price	1.000000
sqft_living	0.702049
grade	0.667561
sqft_above	0.605567
sqft_living15	0.585367
bathrooms	0.525198
view	0.397595
sqft_basement	0.323796
bedrooms	0.308278
lat	0.307058
waterfront	0.266372
floors	0.257070
yr_built	0.054088
condition	0.036439
long	0.021524

Name: price, dtype: float64

Now let's drop the values of the price and consider it as target vector  $Y$ . Also consider only relevant features.

```
[ ] Y=df_cleaned['price']

[ ] df_cleaned.drop(["price"],axis=1,inplace=True)

[ ] df_cleaned.shape
```

(21601, 14)

# Model Training

---

## 1. Train test split:

30/70 train split used.

```
# Train-test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

## 2. Setting Polynomial degree and transformation:

```
# Set polynomial degree (recommended: 2 or 3 to avoid overfitting)
poly_degree = 2
poly = PolynomialFeatures(degree=poly_degree, include_bias=False)

# Apply polynomial transformation to scaled features
X_train_poly = poly.fit_transform(X_train_scaled)

# Apply polynomial transformation to test features
X_test_poly = poly.transform(X_test_scaled)
```

## 3. Adding Bias and default coefficient Initialization:

```
[ ] # Add bias term to polynomial features
m = X_train_poly.shape[0]
X_poly = np.hstack((np.ones((m, 1)), X_train_poly))

# Add bias term to test polynomial features
m_test = X_test_poly.shape[0]
X_test_poly_bias = np.hstack((np.ones((m_test, 1)), X_test_poly))

# Initialize weights with random values
w = np.random.randn(X_poly.shape[1])
```

## 4. Model Training:

```
w_opt, costs, grads, iters, runtime = gradient_descent(X_poly, y_train)

print("✅ Optimized weights:", w_opt)
print(f"📄 Final cost: {costs[-1]:.6f}")
print(f"🔄 Iterations until convergence: {iters}")
print(f"⌚ Time taken: {runtime:.2f} seconds")
print(f"📊 Last gradient vector:\n{grads[-1]}")
```

```
✅ Optimized weights: [ 4.43203160e+05  1.70068569e+03 -2.02648100e+04
-3.21019945e+04  8.50484733e+04  7.11737534e+04  1.15040035e+04
 8.87661967e+04  1.00831034e+04  2.64170980e+04  8.55171511e+02
 1.92521602e+04 -8.66090326e+03  1.60433514e+03 -9.99419433e+01
 2.29352464e+03 -2.40498489e+04 -1.88261905e+03  2.70460387e+03
 9.99083637e+03  1.93718363e+04 -1.08481654e+04  7.20524579e+03
-3.88978252e+03  6.14386664e+03 -2.12486781e+04 -1.05703492e+04
 1.60974465e+04 -1.14928191e+04  1.09353872e+04 -1.53777254e+04
 1.85591879e+04  1.32817756e+04  4.75568237e+03 -1.37236580e+04]

📄 Final cost: 17834210185.605049
🔄 Iterations until convergence: 100000
⌚ Time taken: 113.85 seconds
📊 Last gradient vector:
[-1.15203023e+04  6.67169612e+02 -1.63535795e+03  9.87423969e+02
-3.29760456e+03  1.12485153e+03 -2.21771712e+03  6.69639118e+02
-7.95674737e+02  4.13640519e+02  1.68625346e+02 -4.31576618e+00
-1.14414244e+02  2.27889435e+02 -3.42988593e+02 -6.09226435e+01
 1.03288366e+03 -8.44280725e+02 -3.55548713e+02 -1.59450212e+02
 6.90412130e+02  2.52358648e+03  2.70942873e+02 -2.10179262e+02
-1.22320478e+03 -1.40597499e+03  2.15243972e+02 -8.51375470e+02
 2.82247719e+03  9.64540718e+02 -1.23204682e+02 -5.03535231e+01
-5.10400129e+02  1.01860807e+02 -1.36227675e+02  1.60046059e+02
-1.42285797e+02 -2.68293858e+01  1.12675579e+03  4.80664440e+02
 1.23951574e+03 -2.41195970e+02 -4.36217953e+02 -6.18252160e+02
-4.94016466e+02  2.56033522e+03 -8.63488464e+02 -4.52135345e+01
-5.64024469e+02  1.81708501e+03 -1.31848093e+02  3.17322131e+02
-1.71177127e+02 -1.77670925e+03  9.07609949e+01 -8.15558734e+02
 1.71873429e+03 -2.61368304e+02 -1.54434456e+03 -3.21642250e+02
 2.71942984e+03 -1.22117752e+03  1.66627734e+02  1.35117781e+03
-9.30198916e+02  1.50622743e+03]
```

## 5. Testing Train Model: (Code is in notebook Model Deployment Section)

```
🔄 Starting the Test Agent...
```

```
🏠 Test Agent Results for Selected Houses:
-----

House 1:
Features:
  bedrooms: 2.00
  floors: 1.00
  waterfront: 0.00
  view: 0.00
  lat: 47.71
  sqft_living: 1210.00
  bathrooms: 1.75
  grade: 6.00
  sqft_above: 1210.00
  sqft_living15: 1670.00
Actual Price: $456,200.00
Predicted Price: $371,164.56
Difference: $-85,035.44
```

# Visualization

## 1. Plot Gradient Updates for All Coefficients:

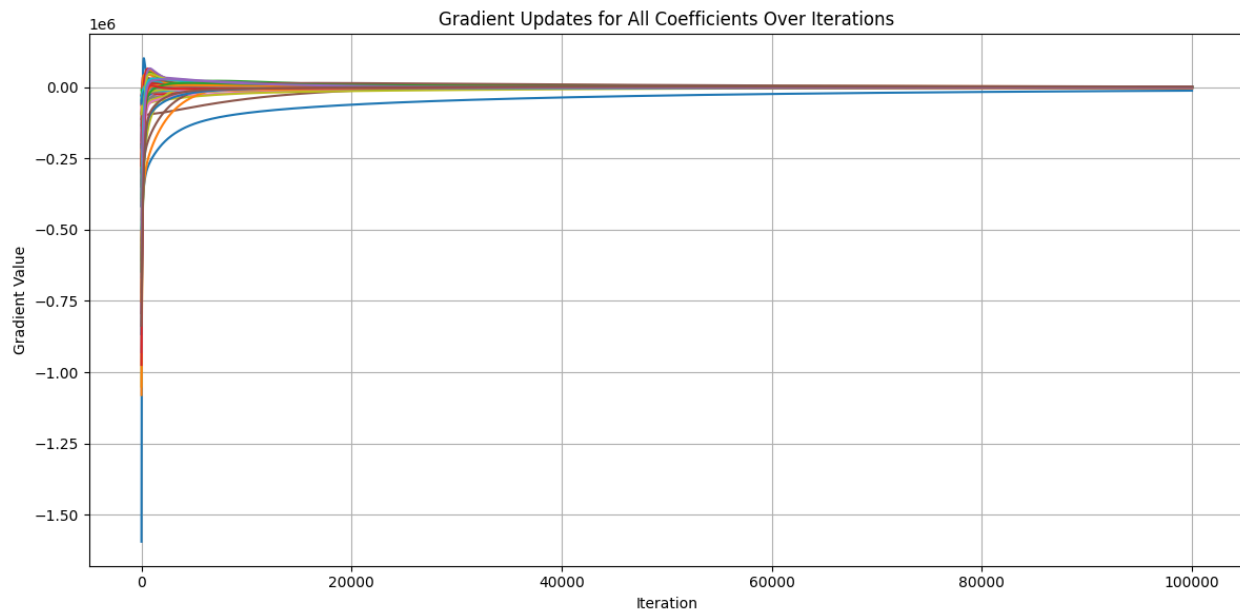
*We'll plot all gradients (one line per coefficient) over the iterations to see how each weight evolves.*

*This will generate one curve per weight showing how its gradient changed and hopefully got closer to zero as it converged.*

```
# Number of coefficients (including bias)
num_coefficients = len(grads[0])

# Plot each coefficient's gradient
plt.figure(figsize=(12, 6))
for i in range(num_coefficients):
    coef_grads = [g[i] for g in grads] # i-th gradient across all iterations
    plt.plot(coef_grads, label=f'Weight {i}')

plt.title("Gradient Updates for All Coefficients Over Iterations")
plt.xlabel("Iteration")
plt.ylabel("Gradient Value")
# plt.legend()
plt.grid(True)
plt.tight_layout()
plt.show()
```



## 2. Plot regression line fit:

This function draws how cost changes over iterations (lower = better). Helps visualize learning speed.

Since we are working with multiple features, we can't plot a single 2D regression line (that only works with 1 feature). But we can:

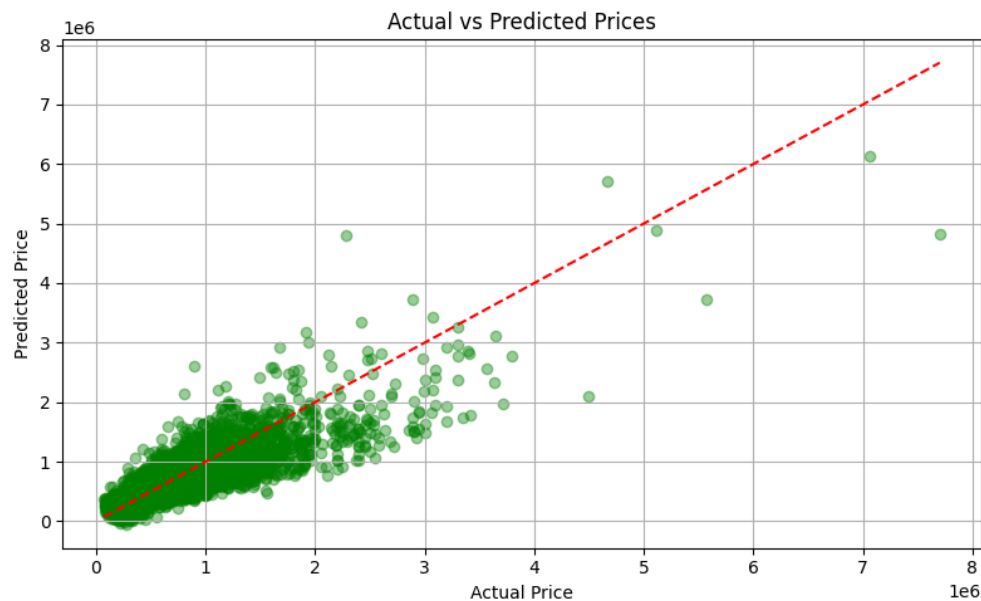
### Plot Actual vs Predicted Values

This is a common way to evaluate regression model fit.

*A tight diagonal cluster means our regression model fits well.*

```
# Make predictions using optimized weights
y_pred = X_poly @ w_opt # w_opt is from gradient_descent return

# Scatter plot: actual vs predicted
plt.figure(figsize=(8, 5))
plt.scatter(y_train, y_pred, alpha=0.4, color='green')
plt.plot([y_train.min(), y_train.max()], [y_train.min(), y_train.max()], color='red', linestyle='--') # perfect fit line
plt.title("Actual vs Predicted Prices")
plt.xlabel("Actual Price")
plt.ylabel("Predicted Price")
plt.grid(True)
plt.tight_layout()
plt.show()
```



### 3. Regression Metrics Report:

#### Mean Squared Error (MSE)

- Measures the average **squared** difference between predicted and actual values.
- Penalizes large errors more.
- **Lower is better** (ideal  $\rightarrow 0$ ).

#### Root Mean Squared Error (RMSE)

- Square root of MSE; brings error back to the original unit (e.g., dollars).
- Easier to interpret.
- **Lower is better.**

#### Mean Absolute Error (MAE)

- Measures the average **absolute** difference between predictions and actual values.
- Less sensitive to outliers than MSE.
- **Lower is better.**

#### R<sup>2</sup> Score (Coefficient of Determination)

Explains how much **variance** in the target variable is captured by the model.

Ranges from **0 to 1** (higher is better):

- $\rightarrow$  perfect fit
- $\rightarrow$  no better than predicting the mean
- $< 0 \rightarrow$  worse than mean prediction

```
# Predictions using optimized weights
y_pred = X_poly @ w_opt

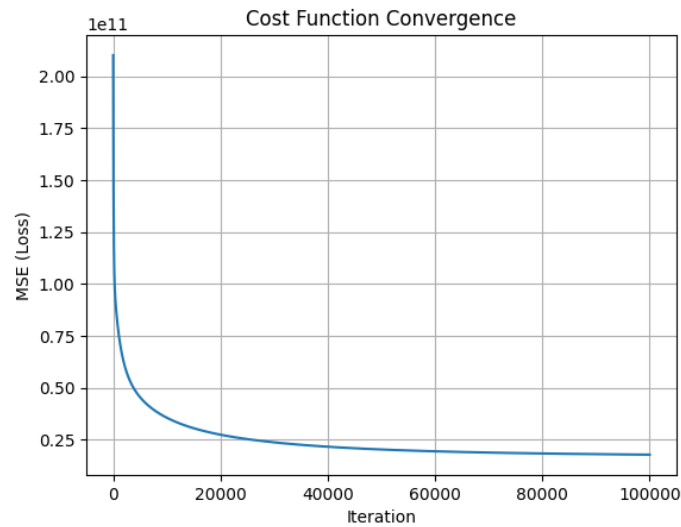
# Metrics
mse = mean_squared_error(y_train, y_pred)
rmse = np.sqrt(mse)
mae = mean_absolute_error(y_train, y_pred)
r2 = r2_score(y_train, y_pred)

print("\n📊 REGRESSION METRICS REPORT")
print(f"💠 Mean Squared Error (MSE):      {mse:.2f}")
print(f"💠 Root Mean Squared Error (RMSE): {rmse:.2f}")
print(f"💠 Mean Absolute Error (MAE):      {mae:.2f}")
print(f"💠 R2 Score:                       {r2:.4f}")
```

```
📊 REGRESSION METRICS REPORT
💠 Mean Squared Error (MSE):      35668377186.90
💠 Root Mean Squared Error (RMSE): 188860.73
💠 Mean Absolute Error (MAE):      118077.09
💠 R2 Score:                       0.7270
```

## 4. Cost vs Iterations (Loss Curve)

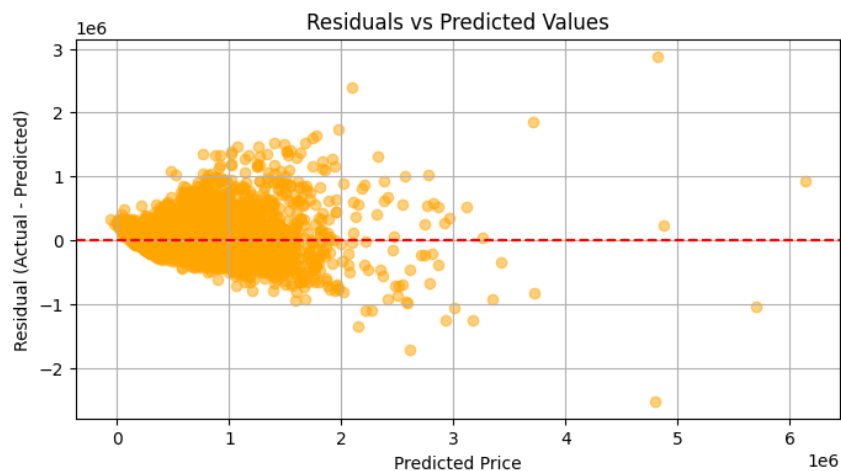


## 5. Residual Plot (Error vs Prediction)

In regression analysis, a residual plot visualizes the difference between observed and predicted values from a regression model. It's a scatter plot where the x-axis represents either the predicted values or the independent variables, and the y-axis represents the residuals (errors).

```
residuals = y_train - y_pred

plt.figure(figsize=(8, 4))
plt.scatter(y_pred, residuals, alpha=0.5, color='orange')
plt.axhline(y=0, color='red', linestyle='--')
plt.title("Residuals vs Predicted Values")
plt.xlabel("Predicted Price")
plt.ylabel("Residual (Actual - Predicted)")
plt.grid(True)
plt.show()
```





## 6. Histogram of Prediction Errors:

This plot shows the **distribution of errors** (i.e.,  $y_{\text{pred}} - y_{\text{true}}$ ).

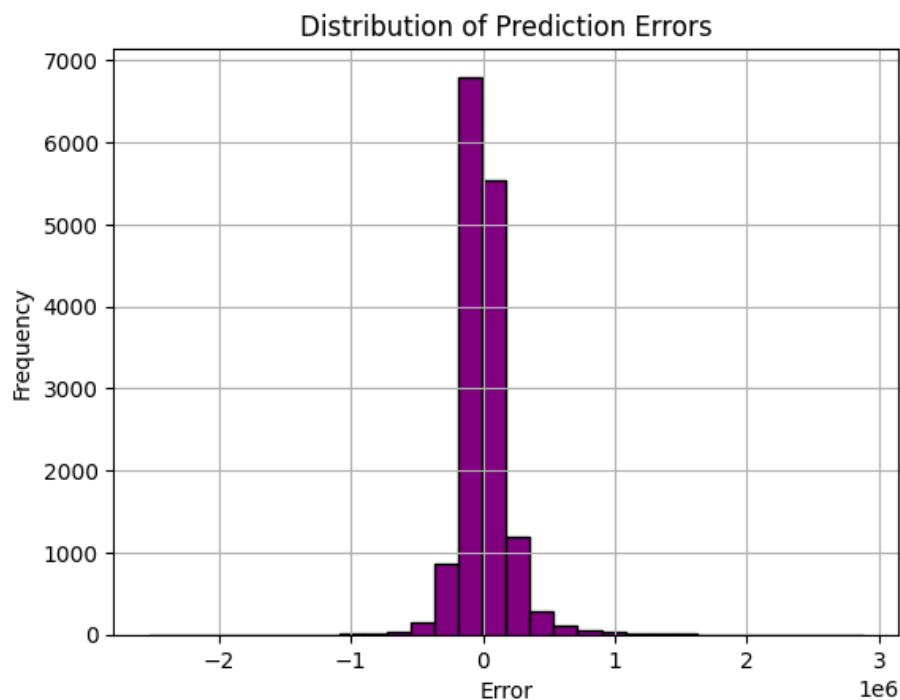
**What it tells:**

- **Centered around 0** → Good sign; model predictions are generally accurate.
- **Symmetric and narrow** → Model makes consistent and low errors.
- **Outliers** (extreme bars far from 0) → Could signal occasional poor predictions.

**Ideal Shape:**

A **bell-like curve** centered at 0 indicates a well-performing model with minimal bias and variance.

```
[ ] plt.hist(residuals, bins=30, color='purple', edgecolor='black')
plt.title("Distribution of Prediction Errors")
plt.xlabel("Error")
plt.ylabel("Frequency")
plt.grid(True)
plt.show()
```



# Analysis

---

## **Identifying and discussing factors that influence the accuracy and performance of algorithm.**

I have used many factors but more important in my code and visualizations are explained but some important factors clearly mentioned and discussed below.

### **1. Use of Stochastic Gradient Descent (SGD)**

Unlike **Batch Gradient Descent (BGD)** which uses the entire dataset to compute gradients, **Stochastic Gradient Descent (SGD)** updates weights using **one training example at a time** or **some part of dataset**.

This can lead to:

- Faster convergence (especially on large datasets)
- Better generalization
- Escape from local minima due to randomness

### **2. Feature Scaling**

Different features with varying magnitudes can negatively affect the gradient update process.

Always apply Standardization (zero mean, unit variance) or Normalization (scaling to [0,1]) to improve training speed and convergence based on nature of data.

### **3. Feature Engineering**

More meaningful and relevant features often lead to better model performance.

- Combine existing features to form new ones (polynomial, interaction terms)
- Remove irrelevant/noisy features
- Use domain knowledge to extract key insights

### **4. Cross-Validation**

Split data into multiple folds to evaluate the model's performance more reliably. This helps in:

- Reducing bias in evaluation
- Better hyperparameter tuning
- Detecting data leakage or overfitting early

# Conclusion:

---

This project demonstrated how to use **Gradient Descent** to optimize a **Linear Regression** model on the `kc_house_data.csv` dataset. After cleaning and preprocessing the data, gradient descent successfully minimized the cost function and provided a predictive model for house prices.

By analyzing feature distributions, handling outliers, and applying standardization, we ensured that the model was both stable and efficient. The convergence of the cost function and performance metrics like Mean Squared Error and  $R^2$  Score validated the accuracy and reliability of the approach.

This exercise not only reinforced concepts in linear regression and optimization but also emphasized the importance of proper data preprocessing in real-world machine learning problems.

# References:

---

- [1] <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.PolynomialFeatures.html>
- [2] <https://youtu.be/4ofxKXCT1MU?si=VhUZAPyHbUdpPvgv>
- [3] <https://blog.devgenius.io/linear-regression-using-gradient-descent-in-python-f75b723ed1c5>
- [4] <https://medium.com/codex/gradient-descent-math-over-simplified-e2b38fddfaa0>
- [5] <https://towardsdatascience.com/polynomial-regression-gradient-descent-from-scratch-279db2936fe9/>