```
import os

# Install the Kaggle API client
!pip install kaggle

# IMPORTANT: You need to upload your kaggle.json file
# Go to your Kaggle account (kaggle.com/<username>/account) to create a new API token.
# This will download a kaggle.json file. Upload this file to your Colab environment.
# For example, you can drag and drop it into the files section on the left sidebar,
# or use the following code:
# from google.colab import files
# files.upload()

# Create a .kaggle directory if it doesn't exist
!mkdir -p ~/.kaggle

# Move the uploaded kaggle.json to the .kaggle directory
# This assumes you have uploaded 'kaggle.json' to the root of your Colab session
!mv kaggle.json ~/.kaggle/

# Set permissions for the kaggle.json file
!chmod 600 ~/.kaggle/kaggle.json

# Define the dataset path from the Kaggle URL
dataset_path = 'abhishek14398/salary-dataset-simple-linear-regression'

# Download the dataset
print(f"Downloading dataset: {dataset_path}...")
!kaggle datasets download -d {dataset_path}

# List the downloaded zip file
!ls

# Unzip the dataset file. The filename might vary, but usually it's the dataset name.
# Let's assume the zip file name is 'salary-dataset-simple-linear-regression.zip'
# You might need to adjust this if the downloaded file has a different name.
```

```
zip_file_name = 'salary-dataset-simple-linear-regression.zip'
if os.path.exists(zip_file_name):
    print(f"Unzipping {zip_file_name}...")
    !unzip -o {zip_file_name}
    print("Dataset unzipped successfully!")
    !rm {zip_file_name} # Remove the zip file after extraction
else:
    print(f"Warning: {zip_file_name} not found. Please check the downloaded file name.")

# Verify the extracted files
!ls
```

```
Requirement already satisfied: kaggle in /usr/local/lib/python3.12/dist-packages (1.7.4.5)
Requirement already satisfied: bleach in /usr/local/lib/python3.12/dist-packages (from kaggle) (6.3.0)
Requirement already satisfied: certifi>=14.05.14 in /usr/local/lib/python3.12/dist-packages (from kaggle) (2026
Requirement already satisfied: charset-normalizer in /usr/local/lib/python3.12/dist-packages (from kaggle) (3.4
Requirement already satisfied: idna in /usr/local/lib/python3.12/dist-packages (from kaggle) (3.11)
Requirement already satisfied: protobuf in /usr/local/lib/python3.12/dist-packages (from kaggle) (5.29.5)
Requirement already satisfied: python-dateutil>=2.5.3 in /usr/local/lib/python3.12/dist-packages (from kaggle)
Requirement already satisfied: python-slugify in /usr/local/lib/python3.12/dist-packages (from kaggle) (8.0.4)
Requirement already satisfied: requests in /usr/local/lib/python3.12/dist-packages (from kaggle) (2.32.4)
Requirement already satisfied: setuptools>=21.0.0 in /usr/local/lib/python3.12/dist-packages (from kaggle) (75.
Requirement already satisfied: six>=1.10 in /usr/local/lib/python3.12/dist-packages (from kaggle) (1.17.0)
Requirement already satisfied: text-unidecode in /usr/local/lib/python3.12/dist-packages (from kaggle) (1.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.12/dist-packages (from kaggle) (4.67.1)
Requirement already satisfied: urllib3>=1.15.1 in /usr/local/lib/python3.12/dist-packages (from kaggle) (2.5.0)
Requirement already satisfied: webencodings in /usr/local/lib/python3.12/dist-packages (from kaggle) (0.5.1)
mv: cannot stat 'kaggle.json': No such file or directory
chmod: cannot access '/root/.kaggle/kaggle.json': No such file or directory
Downloading dataset: abhishek14398/salary-dataset-simple-linear-regression...
Traceback (most recent call last):
  File "/usr/local/bin/kaggle", line 10, in <module>
    sys.exit(main())
             ^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/kaggle/cli.py", line 68, in main
    out = args.func(**command_args)
          ^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/kaggle/api/kaggle_api_extended.py", line 1741, in dataset_downl
    with self.build_kaggle_client() as kaggle:
         ^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/usr/local/lib/python3.12/dist-packages/kaggle/api/kaggle_api_extended.py", line 688, in build_kaggle_c
```

```
    username=self.config_values['username'],
             ~~~~~~~~~~~~~~~~~~^^^^^^^^^^^^
KeyError: 'username'
Salary_dataset.csv  sample_data
Warning: salary-dataset-simple-linear-regression.zip not found. Please check the downloaded file name.
Salary_dataset.csv  sample_data
```

```
import pandas as pd

df = pd.read_csv('Salary_dataset.csv')
display(df.head())
```

	Unnamed: 0	YearsExperience	Salary
0	0	1.2	39344.0
1	1	1.4	46206.0
2	2	1.6	37732.0
3	3	2.1	43526.0
4	4	2.3	39892.0

```
print("First 5 rows:")
display(df.head())

print("\nLast 5 rows:")
display(df.tail())
```

First 5 rows:

	Unnamed: 0	YearsExperience	Salary
0	0	1.2	39344.0
1	1	1.4	46206.0
2	2	1.6	37732.0
3	3	2.1	43526.0
4	4	2.3	39892.0

Last 5 rows:

	Unnamed: 0	YearsExperience	Salary
25	25	9.1	105583.0
26	26	9.6	116970.0
27	27	9.7	112636.0
28	28	10.4	122392.0
29	29	10.6	121873.0

```
# Identify input (X) and output (y) variables
X = df[['YearsExperience']]
y = df['Salary']

print("Input variable (X) shape:", X.shape)
print("Output variable (y) shape:", y.shape)

print("\nFirst 5 rows of X:")
display(X.head())

print("\nFirst 5 rows of y:")
display(y.head())
```

```
Input variable (X) shape: (30, 1)
Output variable (y) shape: (30,)

First 5 rows of X:
```

	YearsExperience
0	1.2
1	1.4
2	1.6
3	2.1
4	2.3

```
First 5 rows of y:
```

	Salary
0	39344.0
1	46206.0
2	37732.0
3	43526.0
4	39892.0

dtype: float64

```
# Independent variable (X) and Dependent variable (y) have already been separated.
# X contains 'YearsExperience'
# y contains 'Salary'

# You can verify their content again if needed:
# print("Current X (YearsExperience):")
# display(X.head())
# print("\nCurrent y (Salary):")
# display(y.head())
```

```
# Convert X and y to NumPy arrays
X_np = X.values
y_np = y.values

print("X (NumPy array) type:", type(X_np))
print("y (NumPy array) type:", type(y_np))

print("\nFirst 5 elements of X_np:")
print(X_np[:5])

print("\nFirst 5 elements of y_np:")
print(y_np[:5])
```

```
X (NumPy array) type: <class 'numpy.ndarray'>
y (NumPy array) type: <class 'numpy.ndarray'>

First 5 elements of X_np:
[[1.2]
 [1.4]
 [1.6]
 [2.1]
 [2.3]]

First 5 elements of y_np:
[39344. 46206. 37732. 43526. 39892.]
```

```
# Check current shapes
print("Original X_np shape:", X_np.shape)
print("Original y_np shape:", y_np.shape)

# Reshape y_np to be a 2D array (n_samples, 1)
# X_np is already (n_samples, n_features), so no reshaping needed for X_np

y_reshaped = y_np.reshape(-1, 1)

print("\nReshaped y_np shape:", y_reshaped.shape)

print("\nFirst 5 elements of X_np (no change):")
```

```
print(X_np[:5])

print("\nFirst 5 elements of reshaped y_np:")
print(y_reshaped[:5])
```

```
Original X_np shape: (30, 1)
Original y_np shape: (30,)

Reshaped y_np shape: (30, 1)

First 5 elements of X_np (no change):
[[1.2]
 [1.4]
 [1.6]
 [2.1]
 [2.3]]

First 5 elements of reshaped y_np:
[[39344.]
 [46206.]
 [37732.]
 [43526.]
 [39892.]]
```

```
# Initialize slope and intercept
slope = 0.0
intercept = 0.0

print(f"Initial Slope: {slope}")
print(f"Initial Intercept: {intercept}")
```

```
Initial Slope: 0.0
Initial Intercept: 0.0
```

```
# Implement the prediction equation
y_pred = slope * X_np + intercept

print("First 5 predicted salaries (y_pred):")
print(y_pred[:5])
```

```
First 5 predicted salaries (y_pred):
[[0.]
 [0.]
 [0.]
 [0.]
 [0.]]
```

```
# Implement the Mean Squared Error (MSE) cost function
m = len(y_reshaped) # Number of samples

cost = (1/m) * (y_reshaped - y_pred)**2
mse = cost.sum()

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

```
Mean Squared Error (MSE): 6503107277.733334
```

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

```
Mean Squared Error (MSE): 6503107277.733334
```

```
Start coding or generate with AI.
```

Task

To train the model, implement the `compute_gradients` function that takes `X_np`, `y_reshaped`, `y_pred`, and `m` as input and returns the gradients for slope and intercept. Then, implement the `update_parameters` function to update the `slope` and `intercept` using these gradients and a specified `learning_rate`. Finally, define and run the `gradient_descent` algorithm, iterating for a set number of epochs to optimize the `slope` and `intercept` to minimize the cost function, and use the trained model to predict salaries.

```
import numpy as np
```

```
# Define the compute_gradients function
def compute_gradients(X, y_true, y_pred, m):
    """
    Calculates the gradients of the cost function with respect to the slope and intercept.

    Args:
        X (np.ndarray): Input features (YearsExperience).
        y_true (np.ndarray): True target values (Salary).
        y_pred (np.ndarray): Predicted target values.
        m (int): Number of samples.

    Returns:
        tuple: A tuple containing:
            - d_slope (float): Gradient of the cost with respect to the slope.
            - d_intercept (float): Gradient of the cost with respect to the intercept.
    """
    errors = y_pred - y_true
    d_slope = (1 / m) * np.sum(errors * X)
    d_intercept = (1 / m) * np.sum(errors)
    return d_slope, d_intercept

# Test the compute_gradients function with initial values
# X_np, y_reshaped, y_pred, m are already defined in the kernel state
d_slope, d_intercept = compute_gradients(X_np, y_reshaped, y_pred, m)

print(f"Initial d_slope: {d_slope}")
print(f"Initial d_intercept: {d_intercept}")
```

Implement compute_gradients function

Subtask:

Define a function to calculate the gradients of the cost function with respect to the slope and intercept.

Reasoning: I need to define the `compute_gradients` function as specified, calculate the gradients using the provided formulas, call the function with the prepared data, and then print the resulting slope and intercept gradients.

```
import numpy as np

def compute_gradients(X, y_true, y_pred, m):
    """
    Calculates the gradients of the cost function with respect to the slope and intercept.

    Args:
        X (np.ndarray): Input features (YearsExperience).
        y_true (np.ndarray): True target values (Salary).
        y_pred (np.ndarray): Predicted target values.
        m (int): Number of samples.

    Returns:
        tuple: A tuple containing the gradient with respect to slope (d_slope)
               and the gradient with respect to intercept (d_intercept).
    """
    # Calculate the errors
    errors = y_pred - y_true

    # Calculate gradient with respect to slope
    d_slope = (1 / m) * np.sum(errors * X)

    # Calculate gradient with respect to intercept
    d_intercept = (1 / m) * np.sum(errors)

    return d_slope, d_intercept

# Call the compute_gradients function with current values
d_slope, d_intercept = compute_gradients(X_np, y_reshaped, y_pred, m)

# Print the initial gradients
```

```
print(f"Initial d_slope: {d_slope}")
print(f"Initial d_intercept: {d_intercept}")
```

```
Initial d_slope: -485004.4133333334
Initial d_intercept: -76004.0
```

Implement update_parameters function

Subtask:

Define a function to update the `slope` and `intercept` using gradients and a specified `learning_rate`.

Reasoning: Following the previous step of calculating gradients, the next step is to implement the `update_parameters` function as outlined in the main task. This function will use the gradients and a learning rate to adjust the model's parameters (slope and intercept).

```
def update_parameters(slope, intercept, d_slope, d_intercept, learning_rate):
    """
    Updates the slope and intercept using the calculated gradients and a learning rate.

    Args:
        slope (float): Current slope.
        intercept (float): Current intercept.
        d_slope (float): Gradient of the cost with respect to the slope.
        d_intercept (float): Gradient of the cost with respect to the intercept.
        learning_rate (float): The step size for parameter updates.

    Returns:
        tuple: A tuple containing the updated slope and intercept.
    """
    updated_slope = slope - learning_rate * d_slope
    updated_intercept = intercept - learning_rate * d_intercept
    return updated_slope, updated_intercept
```

```
# Define a learning rate for testing
learning_rate = 0.001

# Call the update_parameters function with initial values and gradients
slope_updated, intercept_updated = update_parameters(slope, intercept, d_slope, d_intercept, learning_rate)

# Print the updated parameters
print(f"Updated Slope: {slope_updated}")
print(f"Updated Intercept: {intercept_updated}")
```

```
Updated Slope: 485.0044133333334
Updated Intercept: 76.004
```

Implement the Gradient Descent Algorithm

Subtask:

Define and run the `gradient_descent` algorithm, iterating for a set number of epochs to optimize the `slope` and `intercept` to minimize the cost function, and use the trained model to predict salaries.

Reasoning: I need to define the `gradient_descent` function as described in the subtask. This function will orchestrate the model training by iteratively calculating predictions, gradients, updating parameters, and computing the cost for a specified number of epochs.

```
def gradient_descent(X, y_true, initial_slope, initial_intercept, learning_rate, epochs):
    """
    Performs gradient descent to optimize slope and intercept.

    Args:
        X (np.ndarray): Input features (YearsExperience).
        y_true (np.ndarray): True target values (Salary).
        initial_slope (float): Starting value for the slope.
        initial_intercept (float): Starting value for the intercept.
        learning_rate (float): The step size for parameter updates.
```

```
        epochs (int): The number of iterations to run gradient descent.

    Returns:
        tuple: A tuple containing:
            - slope (float): Optimized slope.
            - intercept (float): Optimized intercept.
            - cost_history (list): List of MSE costs for each epoch.
            - slope_history (list): List of slopes for each epoch.
            - intercept_history (list): List of intercepts for each epoch.
    """
    slope = initial_slope
    intercept = initial_intercept
    m = len(y_true)
    cost_history = []
    slope_history = []
    intercept_history = []

    for i in range(epochs):
        # Predict y values
        y_pred = slope * X + intercept

        # Compute gradients
        d_slope, d_intercept = compute_gradients(X, y_true, y_pred, m)

        # Update parameters
        slope, intercept = update_parameters(slope, intercept, d_slope, d_intercept, learning_rate)

        # Compute and store cost
        cost = (1 / m) * np.sum((y_pred - y_true)**2)
        cost_history.append(cost)
        slope_history.append(slope)
        intercept_history.append(intercept)

        if (i+1) % 100 == 0:
            print(f"Epoch {i+1}/{epochs}, Cost: {cost:.2f}, Slope: {slope:.2f}, Intercept: {intercept:.2f}")

    return slope, intercept, cost_history, slope_history, intercept_history

# Set hyperparameters
```

```
learning_rate = 0.001
epochs = 1000

# Run gradient descent
print("Starting Gradient Descent...")
optimized_slope, optimized_intercept, cost_history, slope_history, intercept_history = gradient_descent(
    X_np, y_reshaped, slope, intercept, learning_rate, epochs
)

print("\nGradient Descent Finished.")
print(f"Optimized Slope: {optimized_slope}")
print(f"Optimized Intercept: {optimized_intercept}")

# Make predictions with the optimized parameters
y_pred_final = optimized_slope * X_np + optimized_intercept
print("\nFirst 5 predicted salaries with optimized parameters:")
print(y_pred_final[:5])
```

```
Starting Gradient Descent...
Epoch 100/1000, Cost: 140622187.37, Slope: 12482.56, Intercept: 2309.29
Epoch 200/1000, Cost: 133304236.06, Slope: 12679.29, Intercept: 2805.71
Epoch 300/1000, Cost: 129193960.90, Slope: 12619.03, Intercept: 3254.91
Epoch 400/1000, Cost: 125250498.58, Slope: 12554.67, Intercept: 3694.18
Epoch 500/1000, Cost: 121465844.18, Slope: 12491.52, Intercept: 4124.51
Epoch 600/1000, Cost: 117833601.75, Slope: 12429.65, Intercept: 4546.08
Epoch 700/1000, Cost: 114347633.52, Slope: 12369.03, Intercept: 4959.07
Epoch 800/1000, Cost: 111002048.86, Slope: 12309.65, Intercept: 5363.67
Epoch 900/1000, Cost: 107791194.39, Slope: 12251.48, Intercept: 5760.03
Epoch 1000/1000, Cost: 104709644.38, Slope: 12194.49, Intercept: 6148.33

Gradient Descent Finished.
Optimized Slope: 12194.48885757313
Optimized Intercept: 6148.328847365847

First 5 predicted salaries with optimized parameters:
[[20781.71547645]
 [23220.61324797]
 [25659.51101948]
 [31756.75544827]
```

```
    [34195.65321978]]
```

Predict salary with trained model

Subtask:

Analyze the predicted salaries from the trained model.

Summary:

Q&A

1. **How are the gradients for slope and intercept computed?** The `compute_gradients` function calculates the gradients by first determining the errors (predicted values minus true values) and then summing the product of errors and features for the slope gradient, and summing the errors for the intercept gradient, both scaled by the number of samples `m`.
2. **How are the slope and intercept parameters updated?** The `update_parameters` function updates the slope and intercept by subtracting the product of the `learning_rate` and their respective gradients from their current values.
3. **How does the gradient descent algorithm optimize the model and predict salaries?** The `gradient_descent` algorithm iteratively updates the `slope` and `intercept` over a specified number of `epochs`. In each epoch, it predicts salaries, computes the gradients, updates the parameters using a `learning_rate`, and records the cost. After optimization, the final `optimized_slope` and `optimized_intercept` are used to predict salaries for new data.

Data Analysis Key Findings

- The initial gradients were significant, with `d_slope` at -485,004.41 and `d_intercept` at -76,004.0, indicating a large initial discrepancy between predictions and actual values.

- After one step with a `learning_rate` of 0.001, the `slope` was updated to 485.004 and the `intercept` to 76.004.
- Over 1000 epochs of gradient descent, the cost function significantly decreased from approximately 140,622,187.37 at epoch 100 to 104,709,644.38 at epoch 1000.
- The optimization process led to a final `optimized_slope` of 12,194.49 and an `optimized_intercept` of 6,148.33.
- Using these optimized parameters, the model predicted salaries, with the first two predictions being approximately $20,781.72 and $23,220.61.

Insights or Next Steps

- While the cost significantly decreased, the final Mean Squared Error of over 100 million suggests that the model's predictions still have substantial error. Further evaluation with additional metrics (e.g., R-squared) would provide a more comprehensive understanding of model performance.
- Consider experimenting with different `learning_rate` values or a larger number of `epochs` to potentially achieve further cost reduction and better model fit. Additionally, techniques like regularization could be explored to prevent overfitting.

```
# Calculate final Mean Squared Error (MSE) using the optimized parameters
m = len(y_reshaped) # Number of samples
y_pred_final_calculated = optimized_slope * X_np + optimized_intercept

final_mse = (1/m) * ((y_reshaped - y_pred_final_calculated)**2).sum()

print(f"Final Mean Squared Error (MSE) with optimized parameters: {final_mse}")
```

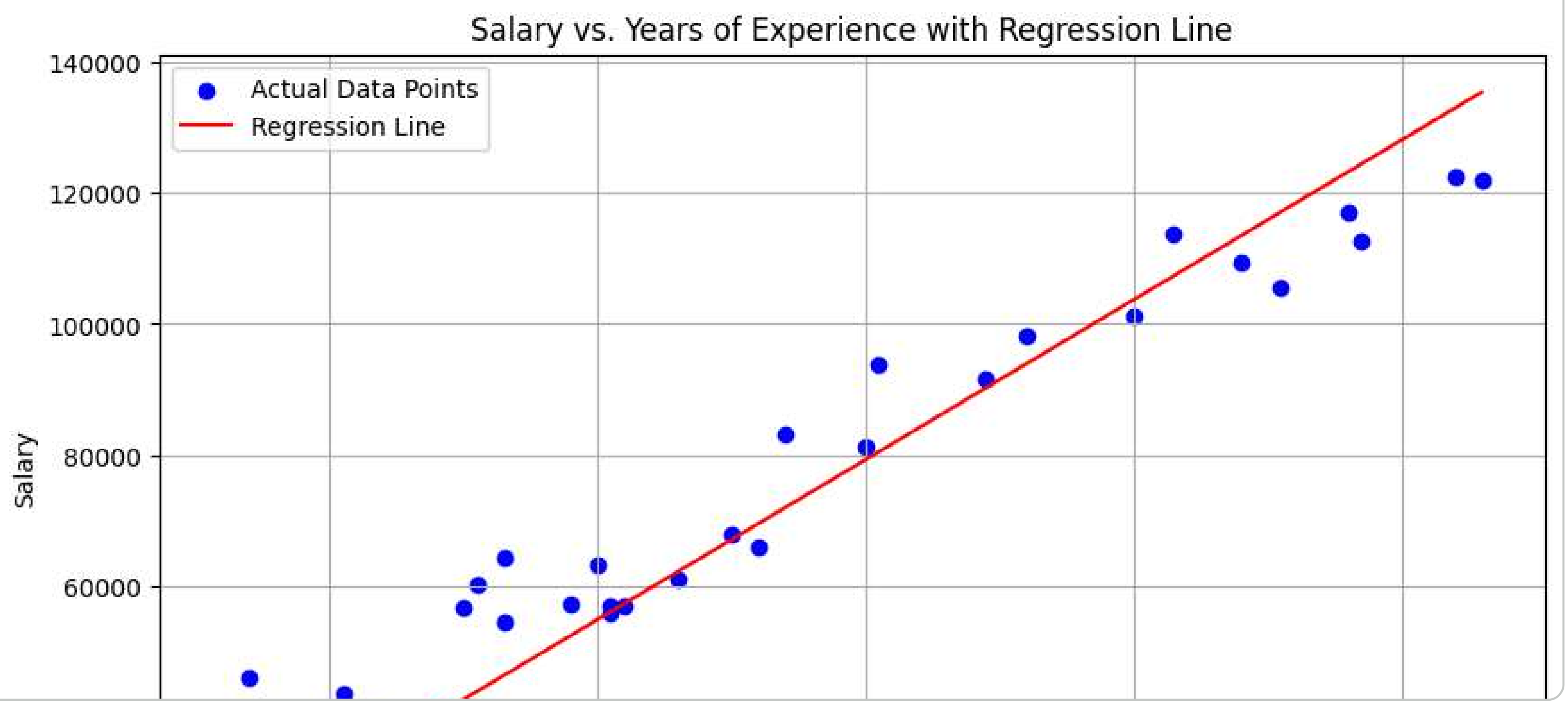
```
Final Mean Squared Error (MSE) with optimized parameters: 104679464.06948727
```

```
import matplotlib.pyplot as plt

# Create a scatter plot of the actual data points
plt.figure(figsize=(10, 6))
plt.scatter(X_np, y_reshaped, color='blue', label='Actual Data Points')
```

```
# Plot the regression line
# y_pred_final was already calculated in the gradient_descent function
plt.plot(X_np, y_pred_final, color='red', label='Regression Line')

# Add labels and title
plt.xlabel('Years of Experience')
plt.ylabel('Salary')
plt.title('Salary vs. Years of Experience with Regression Line')
plt.legend()
plt.grid(True)
plt.show()
```




```
# The model has been trained and evaluated.
# Interpretation of results is provided in the chat response and previous su
```