Simple Linear Regression Using GD	Multiple Linear Regression Using GD
Initialize coefficients: Set B_0 and B_1 to 0 **For each iteration in number of epochs:** a. Calculate the predicted values ($\hat{y}$) using the current coefficients: $\hat{y} = \widehat{B_0} + \widehat{B_1}\, x$ b. Calculate the error (difference between predicted values and actual values): $E = y - \hat{y}$ c. Calculate the gradients for B_0 and B_1: $MSE = \frac{1}{n}\Sigma(y - \hat{y})^2$ $\frac{\partial MSE}{\partial B_0} = \frac{-2}{n} * \Sigma(E)$ $\frac{\partial MSE}{\partial B_1} = \frac{-2}{n} * \Sigma(E * x)$ d. Update the coefficients using the gradients and learning rate: $B_0 = B_0 - \alpha * \frac{\partial MSE}{\partial B_0}$ $B_1 = B_1 - \alpha * \frac{\partial MSE}{\partial B_1}$	Initialize the coefficient: Set B to a zero vector **For each iteration in number of epochs:** a. Calculate the predicted values ($\hat{y}$) using the current coefficients: $\hat{y} = \hat{B}X$ b. Calculate the error (difference between predicted values and actual values): $E = y - \hat{y}$ c. Calculate the gradients for B: $\frac{\partial MSE}{\partial B} = \frac{-2}{n} * X^T E$ d. Update the coefficient using the gradient and learning rate: $B = B - \alpha * \frac{\partial MSE}{\partial B}$

Multiple Linear Regression with Gradient Descent

In [17]:
```
from sklearn.preprocessing import StandardScaler
```

In [18]:
```
# Load the dataset
df = pd.read_csv("C:\\Users\\sghoz\\Downloads\\income.csv")
df.head()
```

Out[18]:

	age	experience	income
0	25	1	30450
1	30	3	35670
2	47	2	31580
3	32	5	40130
4	43	10	47830

In [19]:
```
#Getting the number of rows and columns of the dataset
df.shape
```

Out[19]: (20, 3)

In [20]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20 entries, 0 to 19
Data columns (total 3 columns):
 #   Column      Non-Null Count  Dtype
---  ------      --------------  -----
 0   age         20 non-null     int64
 1   experience  20 non-null     int64
 2   income      20 non-null     int64
dtypes: int64(3)
memory usage: 608.0 bytes
```

In [21]:
```
df.describe()
```

Out[21]:

	age	experience	income
count	20.000000	20.000000	20.000000
mean	39.650000	6.200000	40735.500000
std	10.027725	4.124382	8439.797625
min	23.000000	1.000000	27840.000000
25%	31.500000	3.750000	35452.500000
50%	40.000000	5.000000	40190.000000
75%	47.000000	9.000000	45390.000000
max	58.000000	17.000000	63600.000000

In [22]:
```
# Extract features (X) and target (y)
X = df[['age', 'experience']].values  # Convert to NumPy array
y = df['income'].values  # Convert to NumPy array
```

In [23]:
```
# Initialize scalers
scaler_x = StandardScaler()
scaler_y = StandardScaler()

# Scale X and Y
X_scaled = scaler_x.fit_transform(X)
y_scaled = scaler_y.fit_transform(y.reshape(-1, 1)).flatten()  # Flatten to 1D
```

In [24]:
```
# Define the MultipleLinearRegression class
class MultipleLinearRegression:
    """
    A class to implement Multiple Linear Regression from scratch using Gradient Desce
nt.
    """

    def __init__(self):
        """
        Initializes the model with:
        - `weights`: The coefficients (parameters) of the model (initialized as Non
e).
        - `SSE`: Sum of Squared Errors (SSE), initialized to infinity for convergence
check.
        - `MSE`: Mean Squared Error (MSE), initialized as None.
        """

        self.weights = None  # Model parameters
        self.SSE = float('inf') # Initialize SSE to a very high value
        self.MSE = None  # Mean Squared Error

    def sum_of_squared_errors(self, y, pred):

        """
            Computes the Sum of Squared Errors (SSE) between actual and predicted val
ues.

            Parameters:
            - y (numpy array): The actual target values.
            - pred (numpy array): The predicted values from the model.

            Returns:
            - float: The sum of squared errors.
        """
        return np.sum((y - pred) ** 2)

    def fit(self, X, y, learning_rate=0.01, epochs=500, tolerance=1e-6):
        """
        Trains the model using Gradient Descent.

        Parameters:
        - X (numpy array): The feature matrix of shape (num_samples, num_features).
        - y (numpy array): The target variable of shape (num_samples,).
        - learning_rate (float): The step size for gradient descent (default = 0.01).
        - epochs (int): The maximum number of iterations for training (default = 100
0).
        - tolerance (float): The threshold for convergence; stops if SSE change is sm
aller than tolerance.

        This method updates the weights of the model to minimize the SSE.
        """
        num_samples, num_features = X.shape  # Get the number of samples and features

        # Add a bias column (intercept term) to X
        X = np.c_[np.ones((num_samples, 1)), X]

        # Initialize weights (including bias term)
        self.weights = np.zeros(num_features + 1)

        # Initial predictions before training
        pred = self.predict(X)

        for _ in range(epochs):
            # Compute gradient (partial derivative of the loss function)
            dw = np.dot(X.T, (pred - y)) * (1 / num_samples)

            # Update weights using gradient descent formula
```

```
            self.weights -= learning_rate * dw

            # Recalculate predictions with updated weights
            pred = self.predict(X)

            # Compute new loss (SSE)
            new_loss = self.sum_of_squared_errors(y, pred)

            # Check for convergence (if SSE change is smaller than tolerance, stop tr
aining)
            if abs(new_loss - self.SSE) < tolerance:
                break

            # Update SSE with new loss value
            self.SSE = new_loss

        # Compute MSE
        self.MSE = self.SSE / num_samples

    def predict(self, X):
        """
        Predicts target values based on input features using learned weights.

        Parameters:
        - X (numpy array): The feature matrix of shape (num_samples, num_features).

        Returns:
        - numpy array: Predicted values of shape (num_samples,).
        """
        if self.weights is None:
            raise ValueError("Model has not been trained yet.")

        # Add bias column if missing
        if X.shape[1] == len(self.weights) - 1:
            X = np.c_[np.ones((X.shape[0], 1)), X]

        # Compute predictions using the linear equation: y = Xw
        return np.dot(X, self.weights)

    def plot(self, X, y):
        """
        Plots the scatter plot of the data points and the regression plane.

        Parameters:
        -----------
        X : array-like
            The input feature matrix (must be 2D with exactly two features for visual
ization).
        y : array-like
            The target variable array.
        """

        if self.weights is None:
            raise ValueError("The model has not been fitted yet.")

        # Ensure X and y are numpy arrays
        X = np.array(X)
        y = np.array(y)

        # Check if we can visualize (only works with 2 features)
        if X.shape[1] != 2:
            raise ValueError("Plotting is only available for models with exactly two
independent variables.")

        # Generate predictions
        x1_range = np.linspace(X[:, 0].min(), X[:, 0].max(), 50)
        x2_range = np.linspace(X[:, 1].min(), X[:, 1].max(), 50)
        x1_grid, x2_grid = np.meshgrid(x1_range, x2_range)
```

```
        y_pred_grid = self.weights[0] + self.weights[1] * x1_grid + self.weights[2] *
x2_grid

        # Create 3D plot
        fig = plt.figure(figsize=(10, 7))
        ax = fig.add_subplot(111, projection='3d')

        # Scatter plot of actual data
        ax.scatter(X[:, 0], X[:, 1], y, color='blue', label='Data points')

        # Surface plot of predicted plane
        ax.plot_surface(x1_grid, x2_grid, y_pred_grid, color='red', alpha=0.5)

        # Labels and title
        ax.set_xlabel("Feature 1")
        ax.set_ylabel("Feature 2")
        ax.set_zlabel("Target Variable")
        ax.set_title("Multiple Linear Regression - Regression Plane")
        plt.legend()
        plt.show()
```

In [25]:

```
# Train the model
model = MultipleLinearRegression()
model.fit(X_scaled, y_scaled)
```

In [26]:

```
# Make predictions on the training set
y_pred_scaled = model.predict(X_scaled)

# Convert predictions back to the original scale
y_pred = scaler_y.inverse_transform(y_pred_scaled.reshape(-1, 1)).flatten()
```

In [27]:

```
# Add predictions to the dataframe for comparison
df['predicted_income'] = y_pred
df.head()
```

Out[27]:

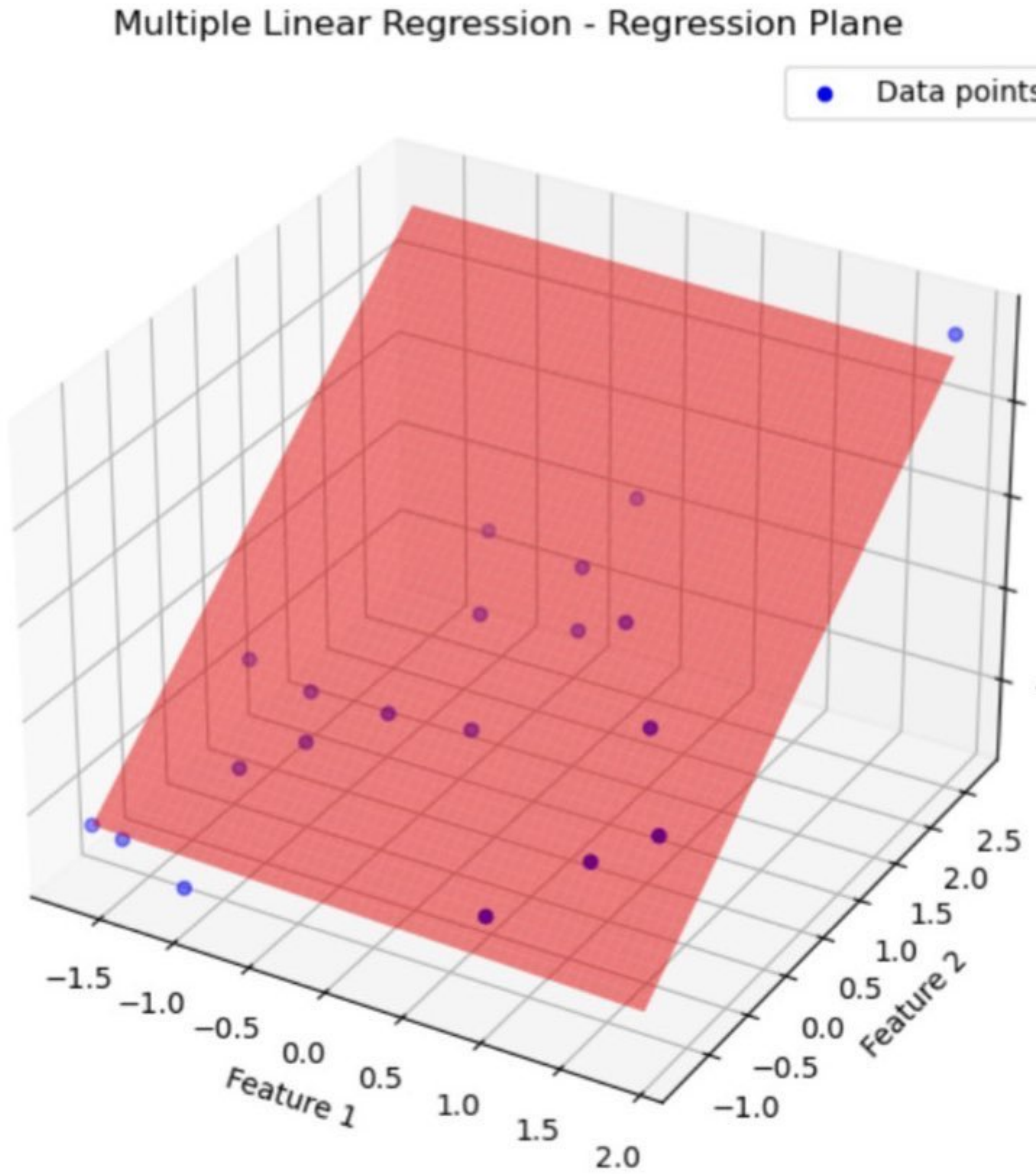
	age	experience	income	predicted_income
0	25	1	30450	30803.043973
1	30	3	35670	34640.615880
2	47	2	31580	32187.282545
3	32	5	40130	38560.425846
4	43	10	47830	48195.474645

In [28]:

```
print("Evaluation of the model")
print("------------------------")
weights = model.weights  # Extract the learned coefficients
feature_count = len(weights) - 1  # Number of independent variables

# Construct the regression equation dynamically
equation = f"y = {round(weights[0], 2)}"  # Intercept term

for i in range(1, feature_count + 1):
    equation += f" + {round(weights[i], 2)} * x{i}"

print(f"Line of best fit is: {equation}")

print(f'Mean squared error is: {round(model.MSE,3)}')
```

```
Evaluation of the model
------------------------
Line of best fit is: y = 0.0 + -0.03 * x1 + 0.97 * x2
Mean squared error is: 0.028
```

In [29]:
```
model.plot(X_scaled,y_scaled)
```



If the blue dots (actual data points) are closely clustered around the red regression plane, it means the model is predicting values close to the actual targets.

In [ ]: