Assigment 2

The project focuses on implementing multiple linear regression to predict house prices based on features like house size and the number of bedrooms. The goal is to create a model that minimizes the error between predicted and actual prices by optimizing the weights (www) and bias (bbb) of the regression equation. To achieve this, I used a vectorized approach with gradient descent, a method that iteratively updates the weights and bias by calculating the gradients of the cost function. The vectorized implementation ensures faster and more efficient computations compared to traditional looping methods. Through this project, I also generated visualizations to analyze the model’s performance, such as cost convergence, predictions vs. actual values, and residuals, providing a deeper understanding of how well the model fits the data.

I started by importing two libraries: numpy (imported as np) and matplotlib.pyplot (imported as plt). np is used for loading, storing, and manipulating data as arrays and plt is used for plotting the data (scatter plot) and visualizing the model’s predictions (line plot).

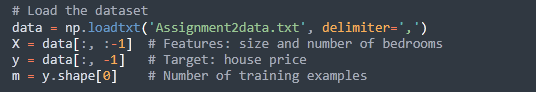


This part of the code is where we load the dataset and prepare it for use in the program. This section prepares the inputs (features) and outputs (target) for training the regression model. The np.loadtxt function reads the data from a text file named Assignment2data.txt. The file is assumed to have columns separated by commas (specified using delimiter=',').

X = data[:, :-1] selects all rows and all columns except the last one. These columns represent the features of the dataset, such as the size of the house and the number of bedrooms.

y = data[:, -1] selects the last column from the dataset, which represents the target variable (the house price we want to predict).

m = y.shape[0] calculates the total number of rows in the y array, which gives the number of training examples in the dataset.

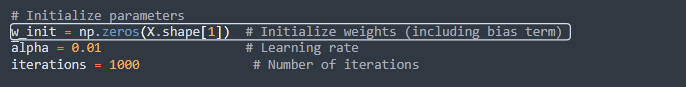


This part of the code sets up the initial values needed for training the model and defines how fast and how long the algorithm should run to learn from the data.

**w\_init = np.zeros(X.shape[1])**: This initializes the weights (including the bias term) as zeros. The size of w\_init matches the number of features in the dataset (X.shape[1]). These weights will be updated during the gradient descent process.

**alpha = 0.01**: This is the learning rate, which determines the size of the steps the algorithm takes to adjust the weights during gradient descent. A smaller value means slower training but more precise adjustments.

**iterations = 1000**: This specifies the total number of iterations (or steps) the gradient descent algorithm will run to optimize the weights.



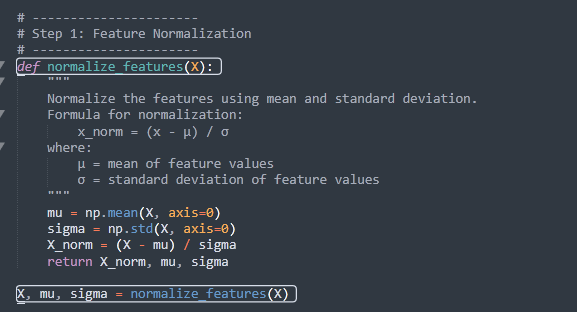
This part of the code is essential for preparing the data so that the gradient descent algorithm can work effectively. First, it normalizes all the features in the dataset to ensure they are on the same scale. This is important because features like house size, could dominate features like the number of bedrooms. If not normalized, the algorithm would place more weight on the larger values simply due to their magnitude, even if both features are equally important for predicting house prices. By applying normalization, each feature is scaled to have a mean of 0 and a standard deviation of 1, ensuring they contribute equally during the training process. This scaling also helps the gradient descent algorithm converge faster because the updates to the weights for each feature are more balanced, preventing one feature from overshadowing others.

**X**: The input feature matrix, where each column represents a feature (house size and number of bedrooms), and each row represents a training example.

**mu**: The mean value of each feature (calculated column-wise). Used to center the feature values around 0 during normalization.

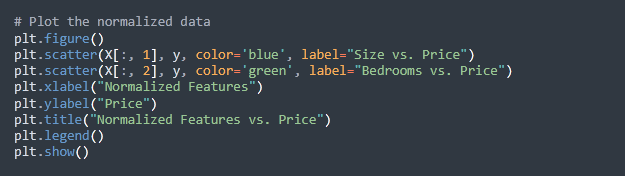
**sigma**: The standard deviation of each feature (calculated column-wise). Used to scale the feature values so they have a standard deviation of 1, ensuring consistent ranges.

**X\_norm:** The normalized version of the feature matrix X, where each feature has a mean of 0 and a standard deviation of 1. Ensures that all features contribute equally during training.

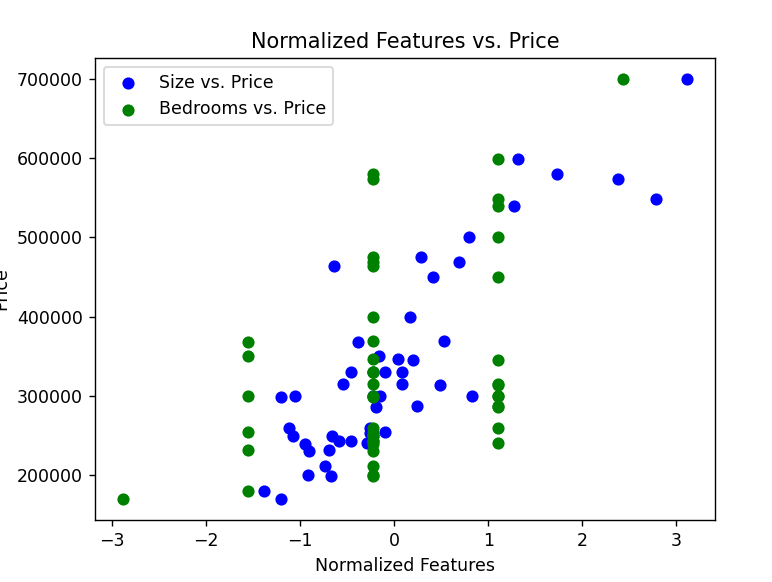


The line from bellow, ensures the model can learn this bias term during training. The bias term allows the model to shift the prediction line (or plane in higher dimensions) up or down to better fit the data. Without this column of ones, the model would assume there is no bias term, forcing the prediction line to pass through the origin (when all feature values are zero), which is unrealistic and might lead to poor predictions.





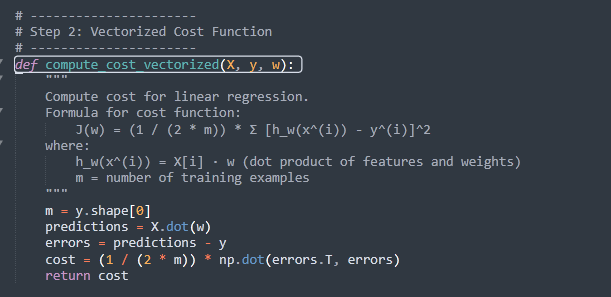
This part of the code generates a scatter plot to visualize the relationship between the normalized features and the target variable (price). The first scatter plot shows the relationship between the normalized house size (blue dots) and house price, while the second scatter plot shows the relationship between the normalized number of bedrooms (green dots) and house price. The x-axis represents the normalized feature values (centered around 0 with a standard deviation of 1), and the y-axis represents the house prices in dollars. The title and legend make it clear which points correspond to which feature. By plotting the normalized features, we can visually observe how each feature contributes to predicting house prices and identify trends or correlations that the model will leverage during training. For instance, we might notice that larger houses tend to have higher prices, which aligns with expectations. This visualization helps confirm that normalization does not distort the data's underlying patterns.



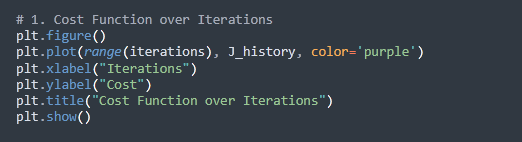
This function, **compute\_cost\_vectorized**, calculates the cost function for linear regression in a vectorized way. The cost function quantifies how well the model's predictions match the actual data by measuring the difference between predicted values and actual values. A smaller cost means the model is performing better.

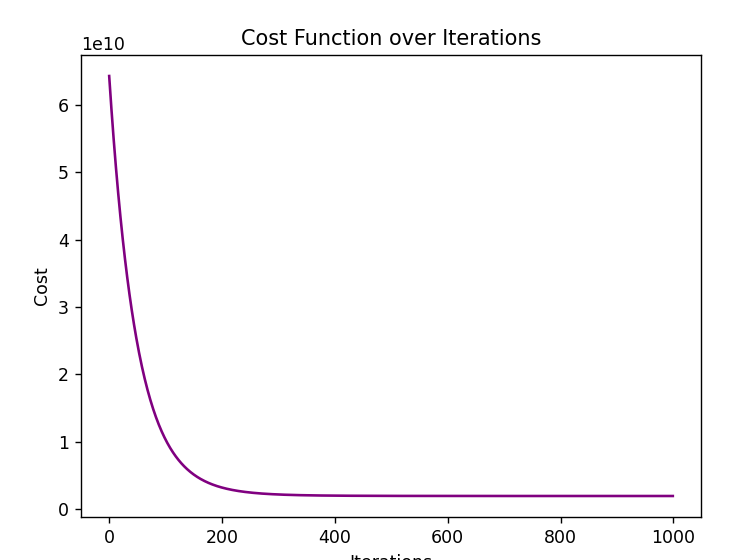
The fuction **def compute\_cost\_vectorized(X, y, w)** receive as paramaters: X ( The feature matrix ,including the bias term column), y: The target values (actual house prices)., w: The weights (including the bias term).

First, it computes the number of training examples (m) from the target variable (y). Then, it calculates the model's predictions by taking the dot product of the feature matrix (X) and the weights (w). The difference between these predictions and the actual values (y) is computed as the error. Using the error, it calculates the cost by summing the squared errors, dividing by 2m, which scales the result to represent the average squared error. Finally, it returns the cost, which quantifies the model's performance and is used to guide improvements during gradient descent.



This graph represents the behavior of the cost function as gradient descent progresses. The cost function measures the error between the model's predictions and the actual values, and the goal of gradient descent is to minimize this cost. Initially, the cost is very high, but it decreases rapidly in the first few iterations as the weights are adjusted. This steep decline shows that gradient descent is most efficient at the start, and slowing down at 400 iterations.





The function from bellow, **gradient\_descent\_vectorized,** implements the gradient descent algorithm to iteratively optimize the weights for linear regression. The function takes the feature matrix (X), target values (y), weights (w), learning rate (alpha), and the number of iterations (num\_iters) as inputs.

First, the number of training examples (m) is calculated from the size of y. An empty list J\_history is initialized to store the cost at each iteration. The main computation happens inside a loop. Within the loop, the predictions for all training examples are computed using the dot product of X and w (predictions = X.dot(w)). The prediction errors are then calculated by subtracting the actual values (y) from the predictions (errors = predictions - y).

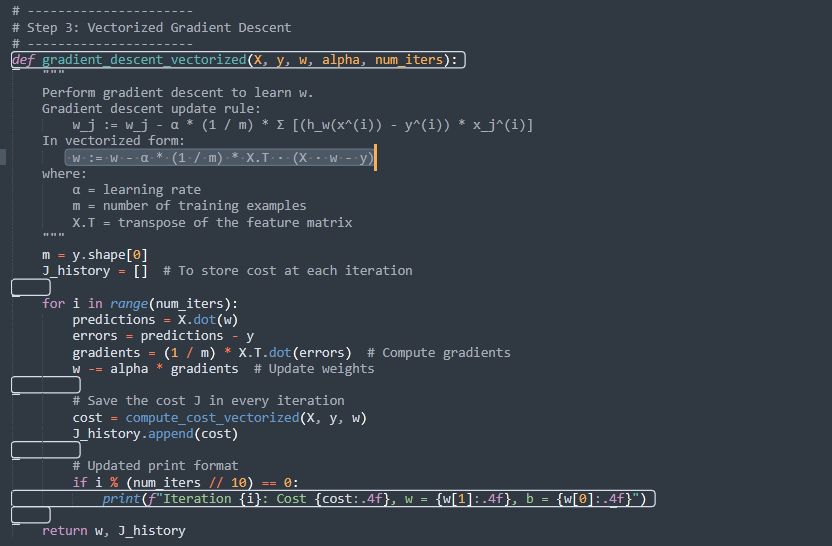
This is done in a single vectorized operation (gradients = (1 / m) \* X.T.dot(errors)), which efficiently calculates the sum of all partial derivatives. This update ensures the weights are adjusted in the direction that reduces the cost (w -= alpha \* gradients).

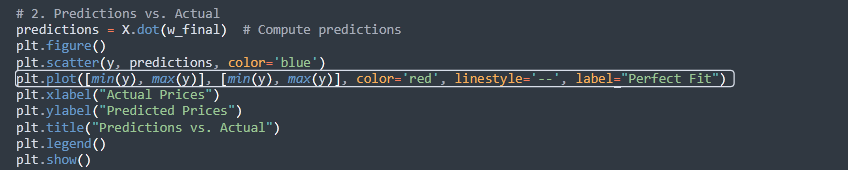
At each iteration, the cost is calculated using the compute\_cost\_vectorized function and stored in J\_history for later analysis. Every 10% of the iterations, a formatted message is printed, displaying the current iteration number, the cost, and the updated weights (print(f"Iteration {i}: Cost {cost:.4f}, w = {w[1]:.4f}, b = {w[0]:.4f}")).

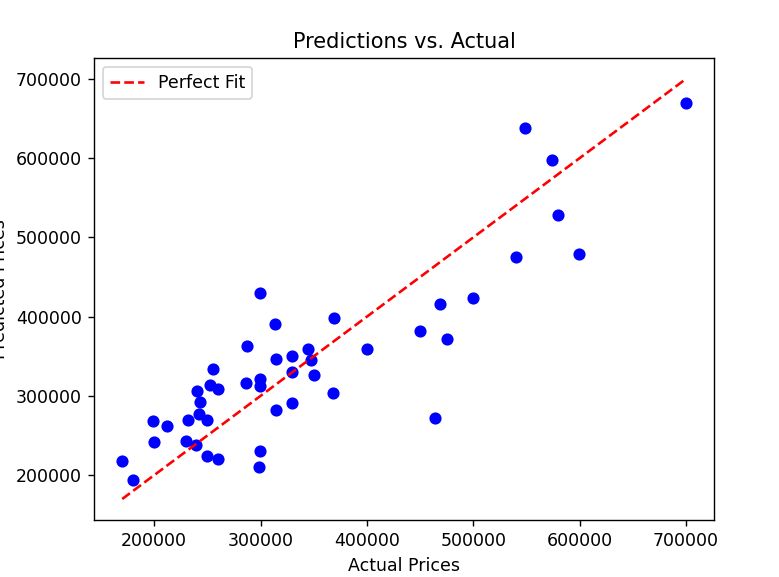
Finally, the function returns the optimized weights and the history of cost values across iterations. This implementation is efficient due to its use of vectorized operations, allowing all computations to be performed simultaneously across the dataset, making it suitable for large-scale data.

The plot shows the comparison between the actual house prices (x-axis) and the predicted house prices (y-axis) generated by the linear regression model. The blue dots represent individual data points, with each dot showing the predicted price against its corresponding actual price. The red dashed line represents a "perfect fit," where the predicted price would exactly match the actual price.

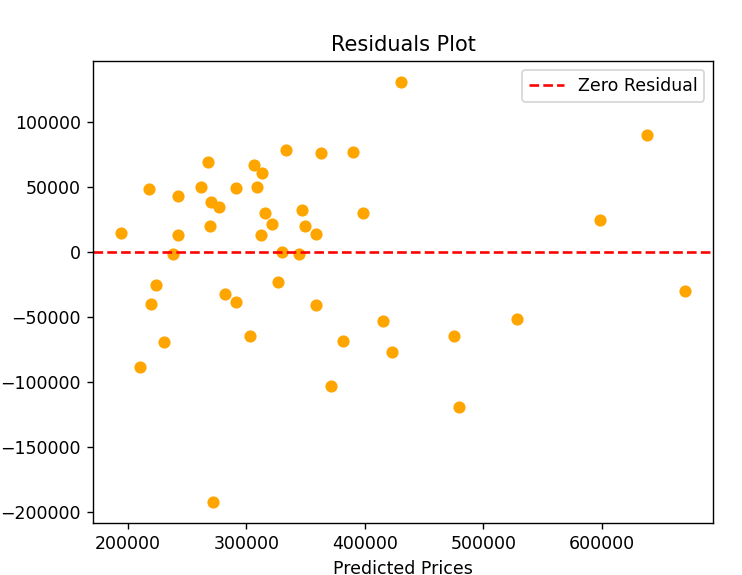
If the model were perfect, all points would lie on the red dashed line. However, some points deviate, indicating prediction errors. The closer the blue dots are to the red line, the better the model's predictions. This graph visually evaluates the model's performance and indicates that, while the model is reasonably accurate, some variation exists between predicted and actual values.





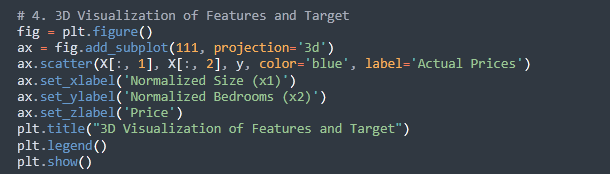


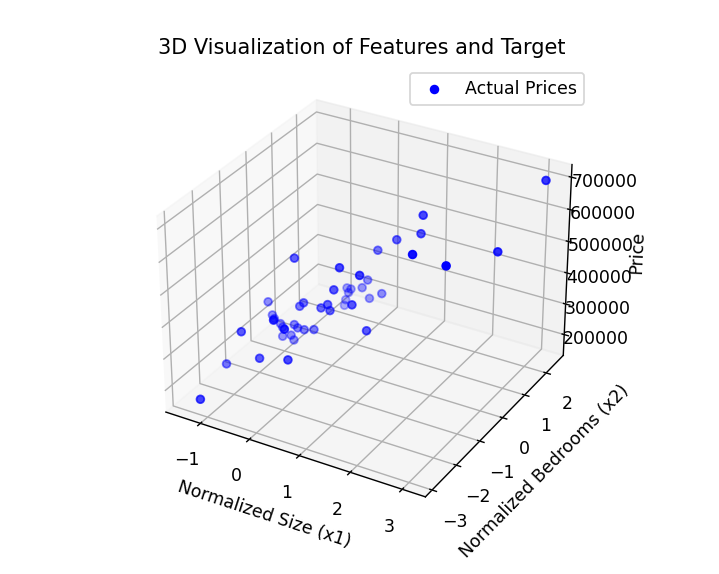
This residuals plot shows the difference between the predicted house prices and the actual prices, with the x-axis representing the predicted prices and the y-axis representing the residuals. The red dashed line at y=0 indicates perfect predictions where the model's predictions match the actual values. Points above the line show cases where the model overestimated the price, while points below the line indicate underestimation. For instance, at predicted prices around 400,000, some residuals are as high as 100,000, meaning the model overpredicted by 100,000100. Similarly, at 300,000some residuals are −100,000, showing underpredictions. Ideally, residuals should be randomly scattered around zero with no patterns, but here, we observe larger residuals for higher predicted prices, suggesting the model struggles to accurately predict expensive houses. This indicates the linear regression model captures the overall trend but could be improved, perhaps by adding more features or exploring non-linear methods.

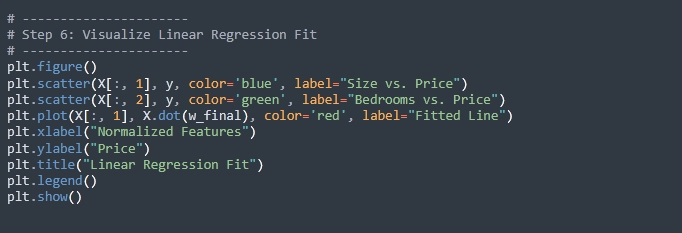


The 3D scatter plot from bellow is a graphical representation designed to explore the relationship between the two independent variables and one dependent variable (target) in three-dimensional space. It allows for a clear visualization of how the target variable is influenced by changes in the features. In this case, the x-axis represents the normalized size (x1), which correspond to the house's size. The y-axis represents normalized bedrooms (x2). Finally, the z-axis represents the target variable, the actual price of the house.

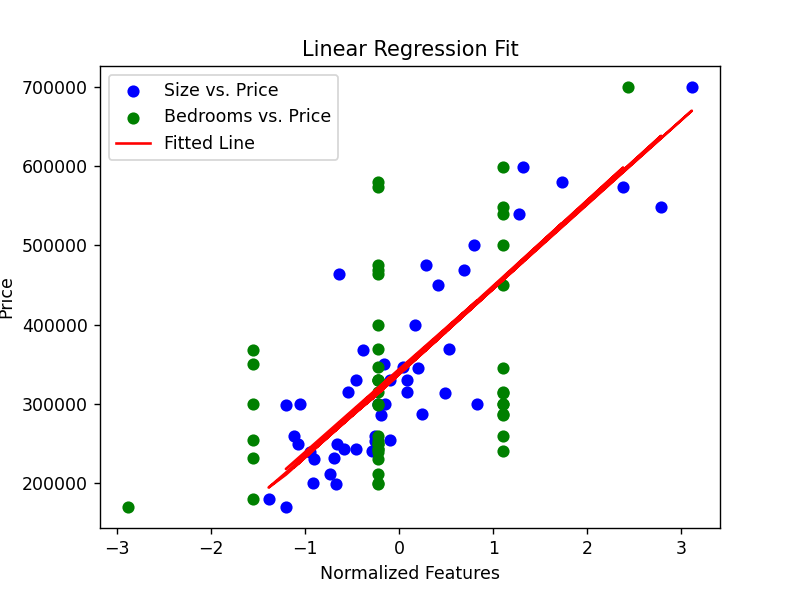
Each blue dot in the plot represents an individual data point from the dataset. The position of the dot on the x and y axes corresponds to the specific values of the normalized size and normalized bedrooms for a house. The height of the dot along the z-axis reflects the price of the house. Together, the scatter plot provides a spatial distribution of the data, making it easier to observe patterns, clusters, or outliers.





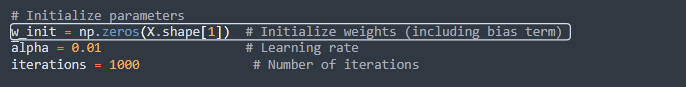


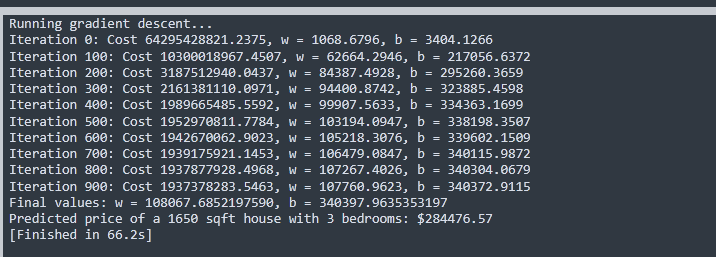
This plot visualizes the linear regression fit for the normalized features against house prices. The blue dots represent the relationship between the normalized size of the house and its price, while the green dots represent the relationship between the normalized number of bedrooms and the price. The x-axis shows the normalized feature values, while the y-axis represents the actual house prices. The red line represents the fitted regression line generated by the model using the optimized weights. This line shows the predicted trend based on the linear relationship between the features and prices. Ideally, the data points should be close to the red line, indicating the model is capturing the relationship well. In this plot, while the model captures the general trend, the scatter of points around the line suggests some variability in the predictions, especially for higher prices, indicating room for improvement.

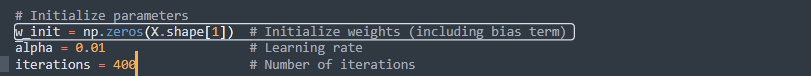


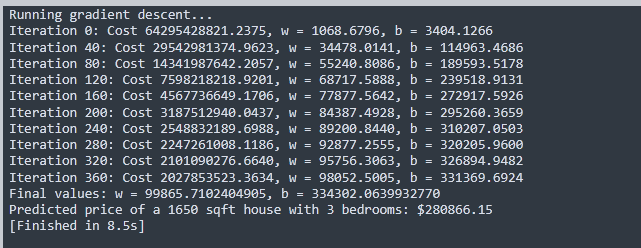
The difference between the two last graphs from above lies in their focus and interpretation. The Predictions vs. Actual graph compares the predicted house prices (y-axis) directly to the actual prices (x-axis), with the red dashed line representing a perfect match. This graph evaluates how closely the predictions align with the actual values, highlighting the model's accuracy. In contrast, the Linear Regression Fit graph shows the fitted regression line (red line) plotted against the actual data points, with normalized features (like size and bedrooms) on the x-axis and house prices on the y-axis. This graph focuses on how well the regression line represents the relationship between the features and the target variable. While the Predictions vs. Actual graph evaluates prediction performance, the Linear Regression Fit graph visualizes the model's understanding of the feature-target relationship.

Results:









After reruning the algorithm with only 400 iterations we can see that the code was completend in a much shorted time ( with a difference of around 48 seconds) with minimal changes in the results.

