Work flow of the Linear Regression model:

Step 1: Set Learning Rate & Number of Iterations; Initiate Random weight and bias value.

Step 2: Build Linear Regression Equation. (y = wx + b)

Step 3: Find the "y pred" value for given x value for the corresponding weight & bias.

Step 4: Check the loss function for these parameter values. (difference between "y pred" & "true y")

Step 5: Update the parameter values using Gradient Descent. (new weight & bias value)

Step 6: Step 3, 4, 5 are repeated till we get minimum loss function

Finally we will get the best model (best weight and bias value) as it has minimum loss function.

This Python class Linear\_Regression implements a basic linear regression model using gradient descent for optimization. Below is an explanation of each part of the class:

**1. \_\_init\_\_(self, learning\_rate, no\_of\_iterations)**

This is the constructor for the Linear\_Regression class. It initializes the learning rate (learning\_rate) and the number of iterations (no\_of\_iterations) for the gradient descent algorithm:

* learning\_rate: This is a scalar value that controls how much the model’s parameters (weights and bias) are updated during each step of gradient descent.
* no\_of\_iterations: This defines how many times the algorithm will iterate over the entire dataset to update the weights and bias.

**2. fit(self, X, Y)**

This method trains the linear regression model on the given training data (X and Y) using gradient descent. Here's a breakdown of the code:

* X: This is a matrix of input features (independent variables). It has dimensions (m, n), where m is the number of training examples and n is the number of features.
* Y: This is a vector of target values (dependent variable).
* self.m, self.n = X.shape: This line extracts the number of training examples (m) and the number of features (n) from the shape of the X matrix.
* self.w = np.zeros(self.n): The weights are initialized to zeros, with a length equal to the number of features (n).
* self.b = 0: The bias term is initialized to zero.
* self.X = X and self.Y = Y: These store the input data and target values for use in later computations.

The method then enters a loop for no\_of\_iterations, calling self.update\_weights() during each iteration. The purpose of this is to gradually adjust the weights and bias using the gradient descent algorithm.

**3. update\_weights(self)**

This method calculates and updates the weights and bias using the gradient descent algorithm:

* **Prediction**: Y\_prediction = self.predict(self.X) is the predicted output for the current weights and bias.
* **Compute Gradients**: The gradient of the loss function (Mean Squared Error) with respect to the weights and bias is computed.
  + dw = - (2 \* (self.X.T).dot(self.Y - Y\_prediction)) / self.m: This computes the gradient with respect to the weights w. The expression self.X.T.dot(self.Y - Y\_prediction) calculates the sum of the gradients for each feature. The division by self.m normalizes the gradient by the number of training examples.
  + db = - 2 \* np.sum(self.Y - Y\_prediction) / self.m: This computes the gradient with respect to the bias term b. The sum of the differences between the predicted values and the true values (Y - Y\_prediction) is calculated, and then normalized by dividing by m (the number of training examples).
* **Update Weights and Bias**: After computing the gradients (dw and db), the weights and bias are updated:
  + self.w = self.w - self.learning\_rate \* dw: The weight vector w is updated by subtracting the product of the learning rate and the gradient of the loss function with respect to w.
  + self.b = self.b - self.learning\_rate \* db: The bias term b is updated similarly.

**4. predict(self)**

This method predicts the output Y for the input features X using the current values of the weights and bias:

* return X.dot(self.w) + self.b: The predicted values are computed as the dot product of the input features (X) and the weights (self.w), plus the bias (self.b). This represents the linear regression equation y^=Xw+b\hat{y} = Xw + by^​=Xw+b.

**Key Concepts:**

* **Gradient Descent**: A common optimization algorithm used to minimize a loss function. In this case, the loss function is the Mean Squared Error (MSE), which is minimized by adjusting the weights and bias using the gradients.
* **Mean Squared Error (MSE)**: This is the loss function used to measure the difference between the actual target values (Y) and the predicted values (Y\_prediction). The goal is to minimize this difference (error).

**Example of Linear Regression:**

In a linear regression model, you are trying to find the best-fitting line through the data points. This line is defined by the weights (w) and the bias (b). The gradient descent algorithm iteratively adjusts these parameters to reduce the prediction error (MSE).

**Potential Issues:**

1. **self.predict() Method**: The predict method uses X.dot(self.w) + self.b but doesn't take X as an argument, which could cause issues if this method is called without the proper context. It would be better to pass X as an argument to predict rather than relying on self.X.
2. **Vectorization**: While the code uses NumPy's vectorized operations, there's a potential inefficiency in the calculation of dw and db, depending on the shape and size of the input data X.

**Suggestions for Improvement:**

1. **Passing X to predict**:

python

Copy code

def predict(self, X):

return X.dot(self.w) + self.b

1. **Optimizing Performance**: If needed, you could further optimize the code, especially when dealing with large datasets. This could include using more advanced optimization techniques like stochastic gradient descent or batch processing.

In summary, this code provides a basic implementation of linear regression using gradient descent, where the model iteratively adjusts its parameters (weights and bias) to minimize the error between predicted and actual target values.