INTRODUCTION

Linear regression is a statistical method used for modeling the relationship between a dependent variable (target) and one or more independent variables (features). The goal is to find a linear relationship that best predicts the target variable.

LINEAR REGRESSION MODEL

The linear regression model can be expressed as:

$$y = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n$$

Where:

- y is the predicted value.
- θ_0 is the bias (intercept).
- $\theta_1, \theta_2, \ldots, \theta_n$ are the coefficients (weights).
- $x_1, x_2, \ldots, x_n$ are the feature values.

VECTORIZED FORM

$$y = \theta^T x$$

Where:

- θ is the parameter vector $[\theta_0, \theta_1, \ldots, \theta_n]^T$.
- x is the feature vector $[x_0, x_1, \ldots, x_n]^T$ with $x_0 = 1$.

COST FUNCTION

To measure the accuracy of our linear regression model, we use a cost function. The most common cost function for linear regression is the Mean Squared Error (MSE):

$$\text{MSE}(\theta) = \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Where:

- m is the number of training examples.
- $h_\theta(x^{(i)})$ is the prediction for the i-th training example.

GRADIENT DESCENT

Gradient Descent is an optimization algorithm used to minimize the cost function by iteratively updating the model parameters. The update rule is:

$$\theta := \theta - \eta \nabla_\theta \text{MSE}(\theta)$$

Where η is the learning rate.

BATCH GRADIENT DESCENT

Computes the gradient using the entire training set:

$$\nabla_\theta \text{MSE}(\theta) = \frac{2}{m} X^T (X\theta - y)$$

STOCHASTIC GRADIENT DESCENT (SGD)

Updates the parameters using one training example at a time:

$$\theta := \theta - \eta \nabla_\theta \text{MSE}(\theta^{(i)})$$

MINI-BATCH GRADIENT DESCENT

Uses a small random subset of the training set for each update.

REGULARIZATION TECHNIQUES

Regularization techniques help to prevent overfitting by adding a penalty term to the cost function. Two common techniques are Ridge Regression and Lasso Regression.

Ridge Regression

Adds an L2 penalty to the cost function:

$$\text{Cost}(\theta) = \text{MSE}(\theta) + \alpha \sum_{j=1}^{n} \theta_j^2$$

Lasso Regression

Adds an L1 penalty to the cost function:

$$\text{Cost}(\theta) = \text{MSE}(\theta) + \alpha \sum_{j=1}^{n} |\theta_j|$$

POLYNOMIAL REGRESSION

Polynomial Regression extends linear regression by considering polynomial features of the input variables.

$$y = \theta_0 + \theta_1 x + \theta_2 x^2 + \ldots + \theta_n x^n$$

EXERCISES

1. What Linear Regression training algorithm can you use if you have a training set with millions of features?

If we have a training set with millions of features, we can use **Stochastic Gradient Descent (SGD)** or **Mini-Batch Gradient Descent**. These algorithms are more efficient than Batch Gradient Descent because they do not require loading the entire dataset into memory for each update. Instead, they update the model weights more frequently, using only a small subset of data (mini-batch) or even a single data point (stochastic). This makes them scalable and suitable for very large datasets.

2. Suppose the features in your training set have very different scales. What algorithms might suffer from this, and how? What can you do about it?

Algorithms that are sensitive to the scale of the features include:

- **Gradient Descent**: The convergence may be slow or unstable if features are on different scales, as it can lead to oscillations in the optimization path.
- **k-Nearest Neighbors (k-NN)**: Distance calculations can be dominated by features with larger scales.
- **Support Vector Machines (SVM)**: The margin calculations can be affected by feature scaling.

Solution: To address this issue, we can use **feature scaling** techniques such as:

- **Standardization (Z-score normalization)**: Rescales features to have a mean of 0 and a standard deviation of 1.
- **Min-Max scaling**: Rescales features to a range of [0, 1].

3. Can Gradient Descent get stuck in a local minimum when training a Logistic Regression model?

No, Gradient Descent will not get stuck in a local minimum when training a **Logistic Regression model**. The cost function of logistic regression (cross-entropy loss) is a convex function, meaning it has only one global minimum. Therefore, as long as the learning rate is appropriate, Gradient Descent will converge to the global minimum.

4. Do all Gradient Descent algorithms lead to the same model provided you let them run long enough?

No, not all Gradient Descent algorithms will lead to the same model even if they run for a long time. This is particularly true when:

- Different learning rates are used.
- Different convergence criteria are applied.
- Algorithms may explore different paths due to their inherent stochastic nature (in the case of SGD and Mini-Batch Gradient Descent).
- The initialization of weights can also lead to different solutions.

5. Suppose you use Batch Gradient Descent and you plot the validation error at every epoch. If you notice that the validation error consistently goes up, what is likely going on? How can you fix this?

If the validation error consistently increases, this is a sign of **overfitting**. The model is likely becoming too complex and is starting to memorize the training data instead of generalizing.

Solutions:

- **Stop training early** when the validation error starts to increase (early stopping).
- **Reduce the complexity of the model** by using simpler models or reducing the number of features.
- **Apply regularization techniques** like Ridge or Lasso to penalize large coefficients.

6. Is it a good idea to stop Mini-batch Gradient Descent immediately when the validation error goes up?

Answer: It is not always a good idea to stop Mini-batch Gradient Descent immediately when the validation error goes up. Instead, it is better to implement **early stopping**, where we monitor the validation error for several epochs. If it continues to rise for a certain number of consecutive

epochs, then stop training. This approach helps to avoid stopping too soon if the model is still learning.

7. Which Gradient Descent algorithm (among those we discussed) will reach the vicinity of the optimal solution the fastest? Which will actually converge? How can you make the others converge as well?

- **Fastest to reach the vicinity of the optimal solution: Stochastic Gradient Descent (SGD)** often reaches the vicinity of the optimal solution fastest because it updates weights after each training example, leading to quicker iterations.
- **Which will actually converge: Batch Gradient Descent** will converge to the exact solution, given sufficient time and appropriate learning rate.
- **Making others converge:** To ensure that SGD and Mini-Batch Gradient Descent converge, you can:
    - Use learning rate decay to reduce the learning rate over time.
    - Use momentum to smooth the updates.
    - Implement adaptive learning rates with algorithms like Adam or RMSprop.

8. Suppose you are using Polynomial Regression. You plot the learning curves and you notice that there is a large gap between the training error and the validation error. What is happening? What are three ways to solve this?

A large gap between the training error (low) and validation error (high) indicates **overfitting**. The model fits the training data well but does not generalize to unseen data.

Solutions:

1. **Regularization**: Apply techniques like Ridge or Lasso regression to penalize complexity.
2. **Simplify the model**: Reduce the degree of the polynomial to decrease model complexity.
3. **More training data**: Gather more training data to help the model learn better representations and generalize well.

9. Suppose you are using Ridge Regression and you notice that the training error and the validation error are almost equal and fairly high. Would you say that the model suffers from high bias or high variance? Should you increase the regularization hyperparameter α or reduce it?

This scenario indicates that the model suffers from **high bias**. The model is too simple to capture the underlying trends in the data.

Action: We should **reduce the regularization hyperparameter α\alphaα** to allow the model more flexibility and reduce bias.

10. Why would you want to use:

- **Ridge Regression instead of plain Linear Regression (i.e., without any regularization)?**
- **Lasso instead of Ridge Regression?**
- **Elastic Net instead of Lasso?**

Answer:

- **Ridge Regression** is used to handle multicollinearity and overfitting by introducing an l2 regularization term that shrinks coefficients, improving model generalization without completely eliminating any feature.
- **Lasso** is preferred over Ridge when feature selection is important because it can shrink some coefficients to zero, effectively selecting a simpler model with only the most relevant features.
- **Elastic Net** is advantageous when there are many correlated features. It combines both l1 and l2 regularization, benefiting from the properties of both Lasso and Ridge, enabling feature selection while maintaining model performance.

11. Suppose you want to classify pictures as outdoor/indoor and daytime/nighttime. Should you implement two Logistic Regression classifiers or one Softmax Regression classifier?

We should implement **one Softmax Regression classifier**. Softmax Regression is suitable for multi-class classification problems where we want to classify instances into multiple categories. In this case, the categories (outdoor/indoor and daytime/nighttime) can be treated as two classes within a multi-class framework.

12. Implement Batch Gradient Descent with early stopping for Softmax Regression (without using Scikit-Learn).

A simple implementation of Batch Gradient Descent with early stopping for Softmax Regression is mentioned below:

```
import numpy as np

# Softmax function
def softmax(z):
    exp_z = np.exp(z - np.max(z))  # Numerical stability
    return exp_z / exp_z.sum(axis=1, keepdims=True)

# Cross-entropy loss function
def cross_entropy(y_true, y_pred):
    return -np.mean(np.sum(y_true * np.log(y_pred + 1e-15), axis=1))

# Batch Gradient Descent for Softmax Regression
def softmax_regression(X, y, learning_rate=0.01, n_iterations=1000, tolerance=1e-4):
    m, n = X.shape
    num_classes = y.shape[1]
    theta = np.random.rand(n, num_classes)  # Random initialization
    prev_loss = float('inf')

    for iteration in range(n_iterations):
        # Compute predictions
        scores = X.dot(theta)
        y_pred = softmax(scores)

        # Compute the loss
        loss = cross_entropy(y, y_pred)
```

```
# Compute gradients
        gradient = X.T.dot(y_pred - y) / m
        theta -= learning_rate * gradient

        # Check for early stopping
        if abs(prev_loss - loss) < tolerance:
            print(f"Stopping early at iteration {iteration}")
            break
        prev_loss = loss

        return theta

# Example usage
if __name__ == "__main__":
    # Generate dummy data
    np.random.seed(0)
    X = np.random.rand(100, 2)  # 100 samples, 2 features
    y = np.zeros((100, 3))      # 100 samples, 3 classes
    y[np.arange(100), np.random.randint(0, 3, 100)] = 1  # Random class assignment

    # Train Softmax Regression with Batch Gradient Descent
    theta = softmax_regression(X, y)
    print("Trained parameters (theta):", theta)
```