MACHINE LEARNING ALGORITHMS

ALGORITHMS LIST :

1. Regression Algorithms

- Linear Regression
- Ridge Regression
- Lasso Regression

2. Dimensionality Reduction Algorithms

- Principal Component Analysis (PCA)
- Linear Discriminant Analysis (LDA)

3. Classification Algorithms

- Support Vector Machine (SVM)
- Decision Tree
- Random Forest

4. Clustering Algorithms

- K-Means Clustering
- Spectral Clustering
- Hierarchical Clustering

5. Deep Learning Algorithms

- Convolutional Neural Network (CNN)

1.REGRESSION ALGORITHMS

LINEAR REGRESSION

```
1. Initialize parameters:
  w = 0  // weight
  b = 0  // bias
  learning_rate = α
  max_iterations = N

2. Repeat until convergence or for max_iterations times:
  a. Compute the predicted values:
    y_pred = w * x + b

  b. Calculate the cost function (Mean Squared Error):
    J = (1 / m) * Σ((y_pred[i] - y[i])^2) for i = 1 to m

  c. Compute the gradients:
    dw = (2 / m) * Σ((y_pred[i] - y[i]) * x[i]) for i = 1 to m
    db = (2 / m) * Σ(y_pred[i] - y[i]) for i = 1 to m

  d. Update the parameters using Gradient Descent:
    w = w - α * dw
    b = b - α * db

3. End Repeat

4. Output the final parameters:
  Print "Final weight (w): ", w
  Print "Final bias (b): ", b

5. Predict new values:
  y_new = w * x_new + b
  Return y_new
```

RIDGE REGRESSION

```
1. Initialize parameters:
   w = 0  // weights
   b = 0  // bias
   learning_rate = $\alpha$
   regularization_parameter = $\lambda$
   max_iterations = N

2. Repeat until convergence or for max_iterations times:
   a. Compute the predicted values:
      y_pred = w * x + b

   b. Calculate the cost function (Regularized Mean Squared Error):
      J = (1 / m) * Σ((y_pred[i] - y[i])^2) + λ * Σ(w^2) for i = 1 to m

   c. Compute the gradients:
      dw = (2 / m) * Σ((y_pred[i] - y[i]) * x[i]) + 2 * λ * w
      db = (2 / m) * Σ(y_pred[i] - y[i])

   d. Update the parameters:
      w = w - α * dw
      b = b - α * db

3. End Repeat

4. Output the final parameters:
   Print "Final weights (w): ", w
   Print "Final bias (b): ", b

5. Predict new values:
   y_new = w * x_new + b
   Return y_new
```

LASSO REGRESSION

```
1. Initialize parameters:
  w = 0  // weights
  b = 0  // bias
  learning_rate = $\alpha$
  regularization_parameter = $\lambda$
  max_iterations = N

2. Repeat until convergence or for max_iterations times:
  a. Compute the predicted values:
    y_pred = w * x + b

  b. Calculate the cost function (Regularized Mean Squared Error):
    J = (1 / m) * Σ((y_pred[i] - y[i])^2) + λ * Σ(|w|) for i = 1 to m

  c. Compute the gradients:
    dw = (2 / m) * Σ((y_pred[i] - y[i]) * x[i])
    Adjust dw for L1 regularization:
      if w > 0: dw += λ
      if w < 0: dw -= λ
    db = (2 / m) * Σ(y_pred[i] - y[i])

  d. Update the parameters:
    w = w - α * dw
    b = b - α * db

3. End Repeat

4. Output the final parameters:
  Print "Final weights (w): ", w
  Print "Final bias (b): ", b

5. Predict new values:
  y_new = w * x_new + b
  Return y_new
```

2.**DIMENSIONALITY REDUCTION ALGORITHMS**

PRINCIPAL COMPONENT ANALYSIS (PCA)

```
1. Input:
  - Dataset X of size (m x n), where m is the number of samples and n is the number of features.

2. Preprocessing:
  a. Standardize the dataset:
    - For each feature j:
      i. Compute mean $\mu_j$ and standard deviation $\sigma_j$.
      ii. Standardize: X_standardized[i][j] = (X[i][j] - $\mu_j$) / $\sigma_j$

3. Compute the Covariance Matrix:
  - Cov_matrix = (1 / m) * (X_standardized^T * X_standardized)

4. Compute Eigenvalues and Eigenvectors:
  - Find the eigenvalues and eigenvectors of Cov_matrix.

5. Sort Eigenvalues and Eigenvectors:
  - Sort the eigenvalues in descending order.
  - Arrange eigenvectors accordingly.

6. Select Principal Components:
  - Choose the top k eigenvectors corresponding to the k largest eigenvalues.

7. Project Data:
  - Project the standardized data onto the selected eigenvectors:
    X_reduced = X_standardized * Eigenvectors_selected

8. Output:
  - Reduced dataset X_reduced of size (m x k).
```

LINEAR DISCRIMINANT ANALYSIS (LDA)

```
1. Input:
  - Dataset X of size (m x n) and class labels y of size (m).

2. Compute Class Statistics:
  - For each class c:
     a. Compute the mean vector μ_c for all features in class c.
     b. Compute the overall mean vector μ for all samples.

3. Compute Within-Class Scatter Matrix (S_W):
  - Initialize S_W = 0
  - For each class c:
     a. Compute the scatter matrix for class c:
        S_c = Σ (x_i - μ_c) * (x_i - μ_c)^T for all x_i in class c
     b. Add to S_W: S_W += S_c

4. Compute Between-Class Scatter Matrix (S_B):
  - Initialize S_B = 0
  - For each class c:
     a. Compute S_B += N_c * (μ_c - μ) * (μ_c - μ)^T
        where N_c is the number of samples in class c.

5. Solve Generalized Eigenvalue Problem:
  - Solve: S_B * w = λ * S_W * w
  - Find eigenvalues (λ) and eigenvectors (w).

6. Sort Eigenvalues and Eigenvectors:
  - Sort eigenvalues in descending order.
  - Arrange eigenvectors accordingly.

7. Select Discriminant Directions:
  - Choose the top k eigenvectors corresponding to the k largest eigenvalues.

8. Project Data:
  - Project the original data onto the selected eigenvectors:
    X_reduced = X * Eigenvectors_selected

9. Output:
  - Reduced dataset X_reduced of size (m x k).
```

3. CLASSIFICATION ALGORITHMS

SUPPORT VECTOR MACHINE (SVM)

```
1. Input:
  - Dataset X of size (m x n) and labels y of size (m), where $y \in \{+1, -1\}$.
  - Regularization parameter C (for Soft Margin SVM).
  - Learning rate $\alpha$.
  - Maximum iterations N.

2. Initialize:
  - Initialize weights w = 0 (size n).
  - Initialize bias b = 0.

3. Repeat until convergence or for N iterations:
  a. For each training example (x_i, y_i):
    i. Compute the decision function:
      $f(x_i) = (w \cdot x_i) + b$
    ii. Check the classification condition:
      If $y_i * f(x_i) >= 1$:
        - Update weights with no penalty:
          $w = w - \alpha * \lambda * w$
      Else:
        - Update weights and bias with penalty:
          $w = w - \alpha * (\lambda * w - y_i * x_i)$
          $b = b + \alpha * y_i$
    iii. $\lambda = (1 / C)$ for soft margin SVM, $\lambda = 0$ for hard margin SVM.

4. Output:
  - Final weight vector w and bias b.

5. Prediction:
  - For a new input x, predict:
    $y_{pred} = \text{sign}((w \cdot x) + b)$
```

DECISION TREE

TRAINING PHASE

```
1. Input:
  - Dataset D with features X and target variable y.
  - Stopping criteria (e.g., max depth, minimum samples per node).

2. Define:
  - Impurity measure:
    - For classification: Entropy (ID3) or Gini Index (CART).
    - For regression: Mean Squared Error (MSE).
  - Splitting criteria:
    - Choose the feature and threshold that minimize impurity or maximize information gain.

3. Function: BuildTree(D, depth)
  a. Check for stopping criteria:
     - If all examples in D belong to the same class, return a leaf node with the class label.
     - If depth exceeds the max depth or D has fewer than the minimum samples, return a leaf
node with the majority class (classification) or mean value (regression).

  b. Find the best split:
     i. For each feature x_j in X:
        - For each possible threshold t:
          - Split D into left (D_left) and right (D_right) subsets:
            D_left = {examples in D where x_j <= t}
            D_right = {examples in D where x_j > t}
          - Compute impurity for the split (weighted by subset sizes):
            Impurity = (|D_left| / |D|) * Impurity(D_left) +
                       (|D_right| / |D|) * Impurity(D_right)
     ii. Select the feature x_j and threshold t with the lowest impurity.

  c. Create a decision node:
     - Feature: x_j
     - Threshold: t

  d. Recursively build left and right subtrees:
     left_subtree = BuildTree(D_left, depth + 1)
     right_subtree = BuildTree(D_right, depth + 1)

  e. Return the decision node with:
     - Feature: x_j
     - Threshold: t
     - Left Subtree: left_subtree
     - Right Subtree: right_subtree

4. Call BuildTree(D, 0) to build the tree.
```

```
PREDICTION PHASE:

1. Input:
   - Decision tree T.
   - New input example x.

2. Traverse the tree:
   a. Start at the root node.
   b. At each decision node:
      - Check the feature and threshold:
        If x[feature] <= threshold:
          Move to the left subtree.
        Else:
          Move to the right subtree.

3. When a leaf node is reached:
   - Return the class label (classification) or mean value (regression) stored in the leaf.

4. Output:
   - Predicted value or class for x.
```

RANDOM FOREST

TRAINING PHASE

```
1. Input:
  - Dataset D with features X and target variable y.
  - Number of trees T.
  - Number of features to select for each tree f (default: sqrt(total features)).
  - Maximum tree depth, minimum samples per leaf, or other stopping criteria.

2. Initialize:
  - Forest = []  // List to store all decision trees.

3. For t = 1 to T:
  a. Bootstrap Sampling:
    - Create a bootstrap sample D_t by randomly sampling with replacement from D.

  b. Feature Subset Selection:
    - Randomly select f features from the total features.

  c. Train a Decision Tree:
    - BuildTree(D_t, selected_features) using the decision tree algorithm.
    - Use only the selected f features for splitting at each node.

  d. Add the trained tree to the Forest:
    - Forest.append(tree)

4. Output:
  - Trained random forest (Forest containing T trees).
```

PREDICTION PHASE:

```
1. Input:
  - Trained Random Forest (Forest).
  - New input example x.

2. Initialize:
  - Predictions = []  // To store predictions from all trees.

3. For each tree in the Forest:
  a. Traverse the tree to predict the value or class for x:
    - prediction = tree.predict(x)
  b. Append prediction to Predictions.

4. Aggregate Predictions:
  - For classification:
   - Final_prediction = Majority vote from Predictions.
  - For regression:
   - Final_prediction = Average of Predictions.

5. Output:
  - Final_prediction (class or value).
```

4. CLUSTERING ALGORITHMS

K-MEANS CLUSTERING

```
1. Input:
  - Dataset D with n data points {x_1, x_2, ..., x_n}.
  - Number of clusters k.
  - Maximum number of iterations max_iter.
  - Tolerance value tol for convergence.

2. Initialize:
  a. Randomly select k points from D as initial centroids {c_1, c_2, ..., c_k}.
  b. Set iteration counter iter = 0.

3. Repeat until convergence or max_iter is reached:
  a. Assign each data point to the nearest centroid:
    - For each data point x_i in D:
      Assign x_i to cluster j such that:
      j = argmin(||x_i - c_j||^2) for all j in {1, 2, ..., k}.

  b. Update centroids:
    - For each cluster j in {1, 2, ..., k}:
      Update centroid c_j = Mean of all points assigned to cluster j.

  c. Check for convergence:
    - If the change in centroids (||c_old - c_new||) is less than tol for all centroids, break.

  d. Increment iteration counter: iter = iter + 1.

4. Output:
  - Final cluster centroids {c_1, c_2, ..., c_k}.
  - Cluster assignments for all data points.
```

SPECTRAL CLUSTERING

```
1. Input:
  - Dataset D with n data points {x_1, x_2, ..., x_n}.
  - Number of clusters k.

2. Construct the Similarity Matrix:
  a. Compute a similarity matrix S of size (n x n):
    - For each pair of points (x_i, x_j):
      S[i][j] = similarity(x_i, x_j)
      (e.g., Gaussian similarity: S[i][j] = exp(-||x_i - x_j||^2 / (2σ^2))).

  b. Set diagonal elements S[i][i] = 0 (self-similarity is not considered).

3. Construct the Graph Laplacian:
  a. Compute the degree matrix D:
    - D[i][i] = Σ S[i][j] for all j.

  b. Compute the unnormalized Laplacian L:
    - L = D - S.

  c. (Optional) Normalize the Laplacian:
    - L_sym = D^(-1/2) * L * D^(-1/2) (symmetric normalization).
    - L_rw = D^(-1) * L (random walk normalization).

4. Compute Eigenvalues and Eigenvectors:
  a. Compute the first k eigenvectors of L (or L_sym/L_rw) corresponding to the k smallest
eigenvalues.

5. Form the Feature Matrix:
  - Construct a matrix U of size (n x k), where the columns are the k eigenvectors.

6. Normalize the Rows of U (Optional):
  - For each row i in U, normalize it to have unit length:
    U[i] = U[i] / ||U[i]||.

7. Apply K-Means Clustering:
  - Treat each row of U as a point in k-dimensional space.
  - Use the K-Means algorithm to cluster the rows into k clusters.

8. Output:
  - Cluster assignments for all data points in D.
```

HIERARCHICAL CLUSTERING

```
1. Input:
  - Dataset D with n data points {x_1, x_2, ..., x_n}.
  - Linkage criterion (e.g., single linkage, complete linkage, average linkage, or ward linkage).
  - Desired number of clusters k or stopping condition (e.g., a distance threshold).

2. Initialize:
  - Each data point is its own cluster. So, initially, there are n clusters.
  - Create a distance matrix to store pairwise distances between all points.
    - For each pair of points (x_i, x_j), compute the distance d(x_i, x_j).
    - Typically, Euclidean distance is used:
      $d(x_i, x_j) = ||x_i - x_j||$.

3. Repeat until there are k clusters or no further merging possible:
  a. Find the two clusters (C_i, C_j) with the smallest distance (based on the chosen linkage criterion).
     - For **single linkage**: Use the minimum pairwise distance between clusters.
     - For **complete linkage**: Use the maximum pairwise distance between clusters.
     - For **average linkage**: Use the average pairwise distance between clusters.
     - For **Ward's linkage**: Minimize the total variance of the merged clusters.

  b. Merge the two closest clusters C_i and C_j into a single cluster C_ij.
     - Remove C_i and C_j from the list of clusters.
     - Add the new merged cluster C_ij to the list of clusters.

  c. Update the distance matrix:
     - Recalculate the distances from the newly formed cluster C_ij to all other clusters.
     - If using **single linkage**, update the distance to the minimum distance between C_ij and any other cluster.
     - If using **complete linkage**, update the distance to the maximum distance between C_ij and any other cluster.
     - If using **average linkage**, update the distance to the average distance between C_ij and any other cluster.

4. Output:
  - Dendrogram: A tree structure representing the merging process of clusters.
  - Cluster assignments (if stopping at a desired number of clusters k).
```

5. DEEP LEARNING ALGORITHMS

CONVOLUTIONAL NEURAL NETWORK (CNN)

```
1. Initialize parameters:
  - Define the number of layers, filters, filter sizes, stride, padding, and activation functions.
  - Initialize weights and biases for all filters and fully connected layers.

2. Input:
  - Provide the input image or feature map (e.g., size HxWxC, where H=height, W=width, C=channels).

3. Forward Propagation:
  a. Convolutional Layer:
    - For each filter in the layer:
      i. Slide the filter over the input (using stride and padding as defined).
      ii. Perform element-wise multiplication between the filter and the input region.
      iii. Sum the results to produce a single value.
      iv. Add the bias term.
    - Generate an output feature map.

  b. Apply Activation Function:
    - Apply the chosen activation function (e.g., ReLU) to each value in the feature map.

  c. Pooling Layer (if applicable):
    - Divide the feature map into pooling regions.
    - Perform the pooling operation (e.g., max-pooling or average-pooling) within each region.
    - Reduce the spatial dimensions of the feature map.

  d. Repeat steps 3a–3c for each convolutional and pooling layer.

  e. Flattening:
    - Convert the final feature maps into a 1D vector.

  f. Fully Connected Layers:
    - Multiply the flattened vector by the weights of the fully connected layer and add biases.
    - Apply the activation function.

4. Output Layer:
  - Use the final layer to output predictions:
    - For classification tasks, apply softmax for probabilities.
    - For regression tasks, output the raw values.

5. Backpropagation:
  a. Compute Loss:
    - Calculate the loss using a suitable loss function (e.g., cross-entropy for classification, MSE for regression).
  b. Compute Gradients:
     - Use backpropagation to calculate the gradients of the loss with respect to weights and
biases for all layers.

  c. Update Parameters:
     - Update weights and biases using Gradient Descent or an optimization algorithm like Adam:
       weight = weight - learning_rate * gradient
       bias = bias - learning_rate * gradient

6. Repeat:
  - Iterate over multiple epochs and batches of the training data.

7. Prediction:
  - Use the trained CNN to make predictions on new inputs.
```