Wheat Seed Classification Prediction

Author: Ying Yiwen
Number: 12210159

Contents

1 Introduction

1.1 Wheat Seed Dataset

The Wheat Seed Dataset is a classical machine learning dataset primarily used for seed classification and recognition tasks. The dataset contains geometric attribute measurements for three different wheat seed varieties, namely Kama, Rosa and Canadian.The purpose of the dataset is to provide an experimental platform for the development and evaluation of classification algorithms, especially for applications in the field of agricultural sciences.

The Wheat Seed Dataset consists of 210 samples, each of which contains seven geometric attributes that were measured using soft x-ray technology. These attributes include area, perimeter, compactness, seed length, seed width, asymmetry coefficient and seed furrow length. These continuous real-valued attributes provide rich information for the machine learning model to distinguish between different varieties of wheat seeds.

The dataset was collected utilizing soft X-ray technology, a non-destructive inspection method that captures the internal structure of wheat seeds in detail. Through this method, researchers were able to obtain precise geometric measurements of the seeds without damaging them. This data is not only critical for seed classification, but also provides valuable information for studying the genetic and physical characteristics of wheat seeds.

Wheat Seed Dataset has a wide range of applications in the field of machine learning, especially in supervised learning tasks suitable for classification and regression analysis. It helps researchers understand the differences between different wheat varieties and can also help agricultural scientists identify and select high-quality seeds, thereby improving crop yield and quality.

1.2 Classification Method

K-Means:
K-Means is an iterative clustering algorithm designed to divide a dataset into K clusters such that points within clusters are as similar as possible and points between clusters are as different as possible. The algorithm starts by randomly selecting K data points as the initial center of mass, and then assigns each data point to the cluster represented by the nearest center of mass according to the nearest neighbor principle. Next, the algorithm recalculates the center of mass for each cluster, which is the mean of all points in that cluster. This process is repeated until the location of the center of mass no longer changes significantly or a preset number of iterations is reached.A major challenge with K-Means is choosing the appropriate value of K, which usually requires domain knowledge or is determined using methods such as the elbow rule.

PCA (Principal Component Analysis):
PCA is a dimensionality reduction technique that transforms potentially correlated variables into a set of linearly uncorrelated variables called principal components through orthogonal transformations. The first principal component captures the largest variance in the data, the second principal component captures the second largest variance and is orthogonal to the first principal component, and so on.The purpose of PCA is to reduce the dimensionality of the data while retaining as much information about the data as possible. It is useful in data preprocessing, feature extraction, and visualization, especially when working with high-dimensional datasets.

Nonlinear Autoencoder:
Nonlinear auto-encoders are deep learning models for unsupervised learning, especially feature learning and dimensionality reduction. They compress the input data into a low-dimensional representation by using an encoding process and then reconstruct the input data by a decoding process. The key to a self-encoder is its hidden layer, a layer where the learned data representation captures the most important features of the input data. By introducing a nonlinear activation function in the hidden layer, the nonlinear self-encoder is able to learn more complex data structures, making it more efficient in processing nonlinear data.

MLP (Multi-layer Perceptron):
A multilayer perceptron is a feed-forward neural network consisting of at least three layers: an input layer, one or

more hidden layers, and an output layer.MLPs are trained using an algorithm known as backpropagation, which incorporates gradient descent to optimize the network's weights.Each neuron in an MLP uses a nonlinear activation function, which allows the network to learn and simulate complex function mappings.MLPs is widely used in classification, regression, and pattern recognition tasks and is one of the foundational models in the field of deep learning.

SVM (Support Vector Machine):
Support Vector Machine is a powerful classification algorithm whose core idea is to find an optimal hyperplane in the feature space as a way of distinguishing between different classes.SVM is especially suitable for those datasets that are linearly differentiable or approximately linearly differentiable. For linearly indivisible data, SVM maps the data to a high-dimensional space by using a kernel function to find an optimal hyperplane in this space.A key advantage of SVM is its good generalization ability in high-dimensional spaces, which makes it perform well in many real-world applications.

AdaBoost (Adaptive Boosting):
AdaBoost is an integrated learning method that builds a strong classifier by combining multiple weak classifiers. During training, AdaBoost adjusts the weights of the data points according to the performance of each weak classifier, so that misclassified data points receive higher weights in subsequent iterations. In this way, subsequent weak classifiers pay more attention to the previously misclassified data points. Ultimately, AdaBoost combines the predictions of all the weak classifiers by weighted majority voting to improve the overall classification performance.AdaBoost is robust to outliers and noise and can be used in conjunction with multiple weak learning algorithms.

2 K-Means

2.1 Principle

K-Means is an unsupervised learning algorithm used to partition a dataset into k clusters. The goal is to minimize the sum of squared distances within each cluster, that is, the distance from each data point to its cluster centroid. The optimization objective of K-Means can be expressed as:

$$J = \sum_{i=1}^{k} \sum_{x_j \in S_i} ||x_j - \mu_i||^2 \tag{1}$$

where k is the number of clusters, S_i is the set of points in the i-th cluster, μ_i is the centroid of the i-th cluster, and x_j is a data point. The algorithm iteratively performs the following steps to minimize J:

- **Assignment Step:** Assign each data point x_j to the nearest centroid μ_i.
- **Update Step:** Recalculate the centroid of each cluster as the mean of all points in that cluster.

K-Means++ is an improvement over the K-Means algorithm in the initialization of centroids. The initialization steps of K-Means++ are as follows:

- Randomly select the first centroid from the dataset.
- For each remaining data point, calculate the minimum distance to the already chosen centroids.
- Select the next centroid with a probability proportional to the square of the distance from the current centroids.

The main difference between K-Means++ and K-Means is in the initialization of centroids. While K-Means randomly selects centroids, K-Means++ spreads the initial centroids more evenly, reducing the risk of getting stuck in local optima and typically leading to better clustering results and faster convergence.

Soft K-Means, also known as Fuzzy K-Means, is a variant of the K-Means algorithm that allows each data point to partially belong to multiple clusters. The optimization objective of Soft K-Means can be expressed as:

$$J = \sum_{i=1}^{k} \sum_{j=1}^{n} u_{ij}^{m} ||x_j - \mu_i||^2 \tag{2}$$

where u_{ij} is the membership degree of data point x_j belonging to cluster i, and m is a constant greater than 1 that controls the fuzziness of the clusters. Soft K-Means iteratively optimizes J through the following steps:

- **Calculate Membership Degrees:** For each data point x_j, calculate its membership degree u_{ij} for each cluster i.
- **Update Cluster Centroids:** Update the centroid of each cluster based on the membership degrees.

The main difference between Soft K-Means and K-Means is that each data point can belong to multiple clusters, rather than being exclusively assigned to one cluster. This soft assignment provides a more flexible clustering partition, especially useful when dealing with complex, multimodal data distributions.

2.2 Algorithm and Results

Algorithm 1 Fit Method of KMeans

Initialize centroids using INITIALIZE_CENTROIDS method.
for $n = 1$ to max_iters **do**
  Assign each point to the nearest centroid.
  Update centroids based on assigned points.
  Apply learning rate to smooth centroid update.
  Calculate inertia and centroid movement.
  if centroid movement $< 1e - 6$ **then**
    break
  end if
end for
Map predicted labels to true labels using MAP_LABELS method.

Algorithm 2 Initialize Centroids Method (KMeans++)

Choose the first centroid randomly.
for $i = 2$ to k **do**
  Calculate squared distances to the nearest centroid for each point.
  Create a probability distribution based on these distances.
  Select a new centroid based on the probability distribution.
end for

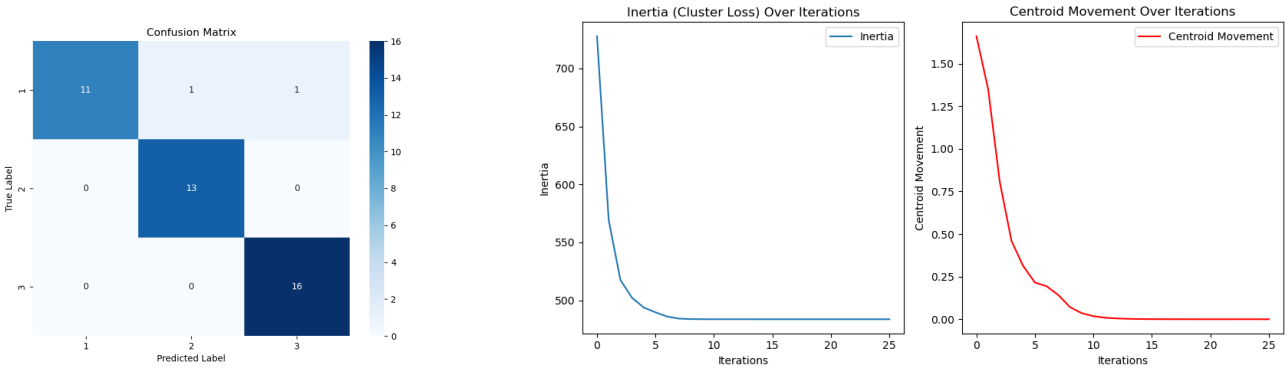
Algorithm 3 Map Labels Method

Create an empty dictionary for label mapping.
for each cluster c in 1 to k **do**
  Get true labels of points assigned to cluster c.
  Find the most common label among these true labels.
  Map cluster c to the most common label.
end for
Return the label mapping dictionary.

Running the kmeans++, we got the result:

```
kmeans = KMeans(k=3, max_iters=100)
kmeans.fit(train_features, train_labels)
kmeans.plot_metrics()
test_predictions = kmeans.predict_and_map(test_features)
accuracy_score(test_labels, test_predictions)
plot_confusion_matrix(test_labels, test_predictions)
```

Converged at iteration 25
Accuracy: 0.9523809523809523



(a) Confusion Matrix (b) Loss Trend

Fig. 1: K-Means++

Algorithm 4 Update Membership for Soft K-Means

Input: Data points X, centroids self.centroids, small value ϵ
Output: Membership values for each point
Compute the pairwise distances between X and centroids: $distances = \|X[:, \text{newaxis}] - \text{self.centroids}\|_2$
Apply a small value ϵ to avoid division by zero: $distances = \max(distances, \epsilon)$
Calculate initial membership values: $membership = 1/(distances^2)$
Normalize the membership values: $membership = membership/\sum(membership, \text{axis} = 1, \text{keepdims=True})$
return *membership*

Running the softkmeans, we got the result:

(a) Confusion Matrix (b) Loss Trend

Fig. 2: Soft K-Means

```
soft_kmeans = SoftKMeans(k=3, max_iters=100, m=1.5)
soft_kmeans.fit(train_features, train_labels)
soft_kmeans.plot_metrics()
test_predictions = soft_kmeans.predict_and_map(test_features)
accuracy_score(test_labels, test_predictions)
plot_confusion_matrix(test_labels, test_predictions)
```

Converged at iteration 30
Accuracy: 0.9047619047619048

2.3 Analyzation

Advantages of K-Means

- **Simplicity and Efficiency:** The K-Means algorithm is easy to understand and implement, and it is computationally efficient for large datasets.
- **Scalability:** The algorithm can handle large-scale datasets as it is iterative and can be parallelized.
- **Versatility:** It is applicable to a variety of datasets, especially when the data can be reasonably partitioned into spherical clusters.

Disadvantages of K-Means

- **Assumption of Cluster Shape:** K-Means assumes clusters are convex, which may not be effective for non-spherical clusters.
- **Sensitivity to the Number of Clusters:** The algorithm requires the number of clusters, k, to be specified a priori, and this choice can significantly affect the results.
- **Prone to Local Optima:** The algorithm may converge to a local optimum rather than a global optimum.

Advantages of K-Means++

- **Improved Initialization:** K-Means++ converges faster to better solutions by using a smarter initialization method.
- **Reduced Local Optima:** It is less likely to get stuck in local optima compared to the standard K-Means.

Disadvantages of K-Means++

- **Additional Computational Cost:** The initialization step adds extra computational overhead.
- **Parameter Selection:** Despite the improved initialization, the number of clusters k still needs to be specified.

Advantages of Soft K-Means

- **Fuzzy Assignment:** It allows data points to belong to multiple clusters with varying degrees of membership, which can better handle overlapping clusters.
- **Flexibility:** The concept of membership degrees provides a more flexible way to capture the uncertainty in the data.

Disadvantages of Soft K-Means

- **Computational Complexity:** Soft K-Means is more computationally intensive compared to K-Means due to the calculation of membership degrees.
- **Additional Parameter:** It introduces an additional parameter (the degree of membership, m), which requires tuning for optimal results.
- **Convergence:** It may require more iterations to converge.

3 Dimensionality

3.1 Principle

Principal Component Analysis (PCA) is a widely utilized dimensionality reduction technique that aims to simplify the complexity of datasets by identifying and extracting the most significant directions or components within the data. This process not only condenses the data into a more manageable form but also retains as much of the original information as possible.

The PCA process involves the following steps:

1. **Data Standardization:** Begin with the standardization of the data, ensuring that each feature has a mean of zero and a standard deviation of one.
2. **Covariance Matrix Calculation:** Calculate the covariance matrix of the dataset, which quantifies the degree to which two variables change together. The formula for the covariance matrix Σ is:

$$\Sigma = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \mu)(x_i - \mu)^T \tag{3}$$

   where x_i represents each data point, μ is the mean of the data, and n is the total number of data points.
3. **Eigenvalue Decomposition:** Decompose the covariance matrix into its eigenvalues and eigenvectors.
4. **Selection of Principal Components:** Select the eigenvectors corresponding to the largest eigenvalues as they represent the directions of greatest variance in the data.
5. **Data Projection:** Project the original data onto these principal components, effectively reducing the dimensionality of the data while preserving the most significant variance. The mathematical projection of the data x onto the new coordinate system defined by the matrix of eigenvectors W is:

$$y = W^T x \tag{4}$$

   where y is the data in the reduced dimensional space.

This transformation allows for a more efficient representation of the data, facilitating easier analysis and visualization.

Nonlinear Autoencoders are a type of artificial neural network used for unsupervised learning of efficient codings. They are designed to learn a compressed, distributed representation (encoding) of a set of data, typically for the purpose of dimensionality reduction or feature learning. The autoencoder consists of an encoder that maps the input to a lower-dimensional representation and a decoder that reconstructs the input from this representation.

The training process of a nonlinear autoencoder involves the following steps:

1. **Input Layer:** The input data X is fed into the autoencoder.
2. **Encoder:** The encoder maps the input X to a lower-dimensional representation h using a nonlinear function, often a sigmoid or ReLU activation function. The encoding can be represented as:

$$h = f(W_1 X + b_1) \tag{5}$$

   where W_1 and b_1 are the weights and biases of the encoder, and f is the activation function.
3. **Code Layer:** The lower-dimensional representation h is the encoded form of the input data.
4. **Decoder:** The decoder attempts to reconstruct the original input X from the encoded representation h. This is done through another set of weights W_2 and biases b_2, and can be represented as:

$$\hat{X} = g(W_2 h + b_2) \tag{6}$$

   where g is typically the same or a similar activation function as f, and $\hat{X}$ is the reconstructed input.

5. **Loss Function:** The autoencoder is trained by minimizing the difference between the original input X and the reconstructed input $\hat{X}$. This is typically done using a loss function such as mean squared error (MSE):

$$L = \frac{1}{2}\sum_{i=1}^{n}(X_i - \hat{X}_i)^2 \tag{7}$$

where L is the loss, X_i is the original input, and $\hat{X}_i$ is the reconstructed input.

6. **Backpropagation and Optimization:** The weights and biases are updated using backpropagation and an optimization algorithm such as stochastic gradient descent (SGD) to minimize the loss function.

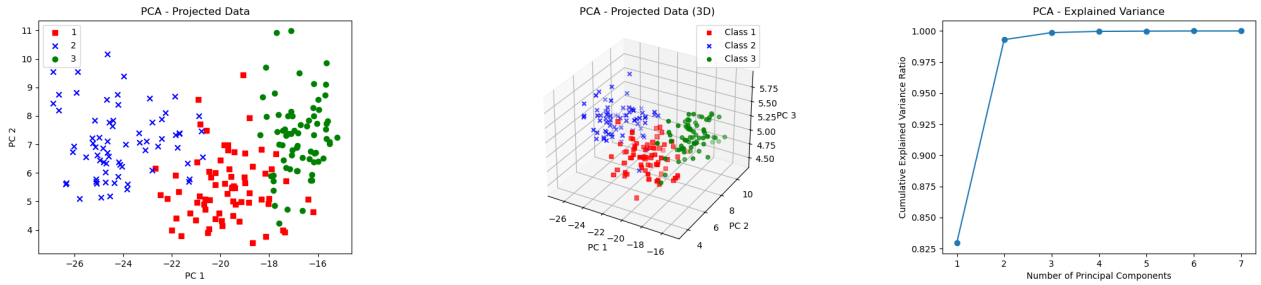
Nonlinear autoencoders are capable of learning complex representations that are not linearly separable. This makes them particularly useful for datasets with intricate structures or for tasks where the relationship between features is nonlinear.

3.2 Algorithm and Results

Algorithm 5 Fit Method for PCA

Input: Data matrix X, number of components $n\_components$
Output: Projection matrix for dimensionality reduction
Compute covariance matrix $cov\_matrix = \text{np.cov}(X^T)$
Find eigenvalues and eigenvectors: $(eigenvalues, eigenvectors) = \text{np.linalg.eig}(cov\_matrix)$
Sort eigenvectors by corresponding eigenvalues in descending order
Select top $n\_components$ eigenvectors to form projection matrix W
end for
Set model attributes:
$self.eigenvalues = eigenvalues$,
$self.eigenvectors = eigenvectors$,
$self.projection\_matrix = W$

Running the PCA, we got:



(a) Two Components (b) Three Components (c) Explained Variance

Fig. 3: PCA

```
pca = PCA(n_components=2)
pca.fit(X)
pca.visualize(X, y)
pca = PCA(n_components=3)
pca.fit(X)
pca.visualize_3d(X, y)
pca.plot_explained_variance()
```

Algorithm 6 Forward Pass in Nonlinear Autoencoder

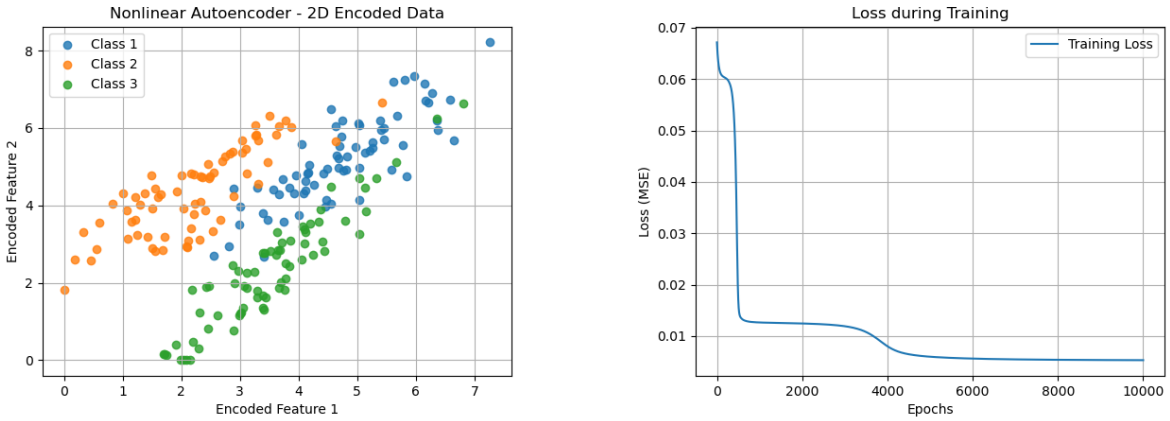
Input: Data x
Output: Reconstructed data $\hat{x}$
$activations \leftarrow [x]$
for each layer i in range(1, number_of_layers) **do**
  $x \leftarrow \text{activation}(W_i x + b_i)$ ▷ activation function varies by layer
  $activations \leftarrow activations + [x]$
end for
return x ▷ Output of the last layer

Algorithm 7 Backward Pass in Nonlinear Autoencoder

Input: Reconstructed data $\hat{x}$, true data y
Output: Updated weights and biases
$deltas \leftarrow [(\hat{x} - y) \cdot \text{sigmoid\_derivative}(\hat{x})]$ ▷ For the output layer
for each layer i from last to first **do**
  $deltas \leftarrow [np.dot(deltas[-1], W_{i+1}^T) \cdot \text{leaky\_relu\_derivative}(activations[i])] + deltas$
end for
Reverse $deltas$
for each layer i **do**
  $W_i \leftarrow W_i - \text{learning\_rate} \cdot np.dot(activations[i].T, deltas[i])$
  $b_i \leftarrow b_i - \text{learning\_rate} \cdot np.sum(deltas[i], \text{axis} = 0)$
end for

Running the nonlinear autoencoder, we got:

```
# Create and train the nonlinear autoencoder 2D
autoencoder_2d = NonlinearAutoencoder(input_dim=train_features.shape[1],
    hidden_dims=[64, 16, 2], output_dim=train_features.shape[1], learning_rate=1e
    -3, epochs=10000)
autoencoder_2d.train(train_features)
X_encoded = autoencoder_2d.encode(train_features)
autoencoder_2d.visualize(X_encoded, train_labels, n_components=2)
autoencoder_2d.plot_loss(n_components=2)
```

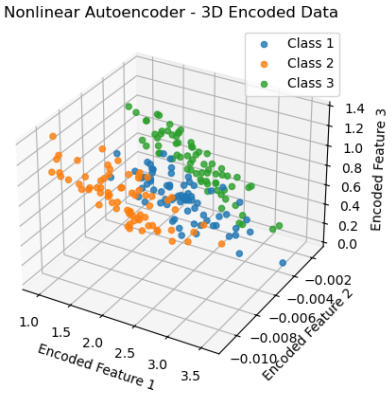


(a) Two Components Dimensionality

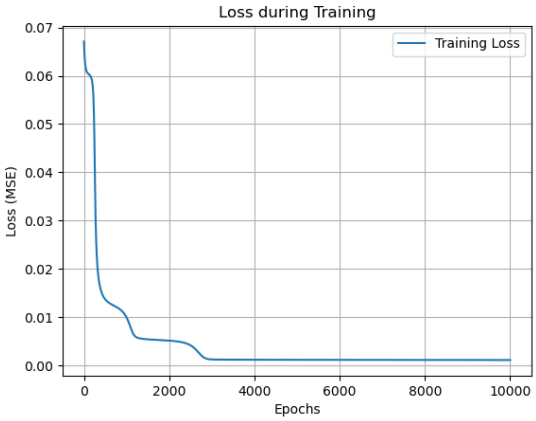
(b) Loss Function

Fig. 4: Nonlinear Autoencoder - 2 Dimensions

```
# Create and train the nonlinear autoencoder 3D
autoencoder_3d = NonlinearAutoencoder(input_dim=X.shape[1], hidden_dims=[64, 32, 8], output_dim=X.shape[1], learning_rate=1.5e-3, epochs=10000)
autoencoder_3d.train(X)
X_encoded_3d = autoencoder_3d.encode(X)
autoencoder_3d.visualize(X_encoded_3d, y, n_components=3)
autoencoder_3d.plot_loss(n_components=3)
```



(a) Three Components Dimensionality



(b) Loss Function

Fig. 5: Nonlinear Autoencoder - 3 Dimensions

3.3 Analyzation

Advantages of PCA:

- Effective in reducing the dimensions of data while retaining the most significant features.
- Capable of noise reduction, preserving the essential information.
- A type of unsupervised learning that does not require labeled data.
- Computationally simple and fast to implement.

Disadvantages of PCA:

- Assumes a linear combination of principal components, which may not be effective for non-linear data.
- The interpretation of principal components may not be as intuitive as the original features.
- Potential loss of important information, especially details that are useful for distinguishing between different classes of data.

Advantages of Nonlinear Autoencoders:

- Can capture complex non-linear relationships in data through multiple non-linear transformations.
- Versatile, suitable for tasks such as dimensionality reduction, data compression, denoising, and anomaly detection.
- Facilitates data compression and visualization, making it easier to understand and present data structures.
- Flexible and extensible, allowing for the design of various autoencoder structures to meet specific needs.

Disadvantages of Nonlinear Autoencoders:

- Risk of overfitting if the model is too complex, which can reduce generalization capabilities.
- High computational resource requirements, especially when training deep autoencoders on large datasets.
- The hidden representations are often not easily interpretable, limiting their application in some fields.
- Sensitive to data quality, as noise and outliers can affect the effectiveness of feature extraction, necessitating proper data cleaning and normalization.

4 Cluster with Dimensionality

With dimensioned data by PCA and Nonlinear Autoencoder, we can try kmeans++ and softkmeans again.

4.1 K-Means++

Using PCA, we generate two-dimensional dataset:

```
pca = PCA(n_components=2)
pca.fit(train_features)
train_pca = pca.transform(train_features)
test_pca = pca.transform(test_features)
```

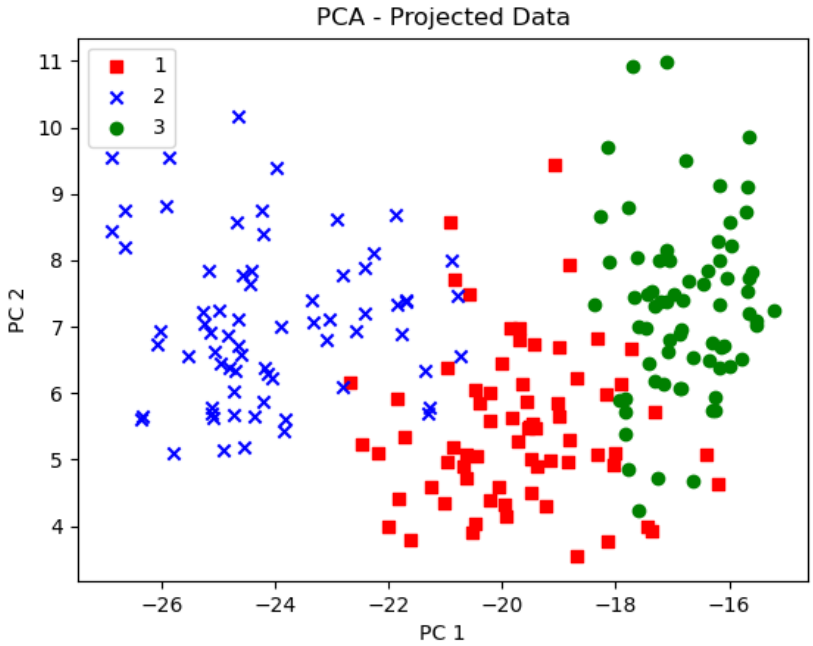
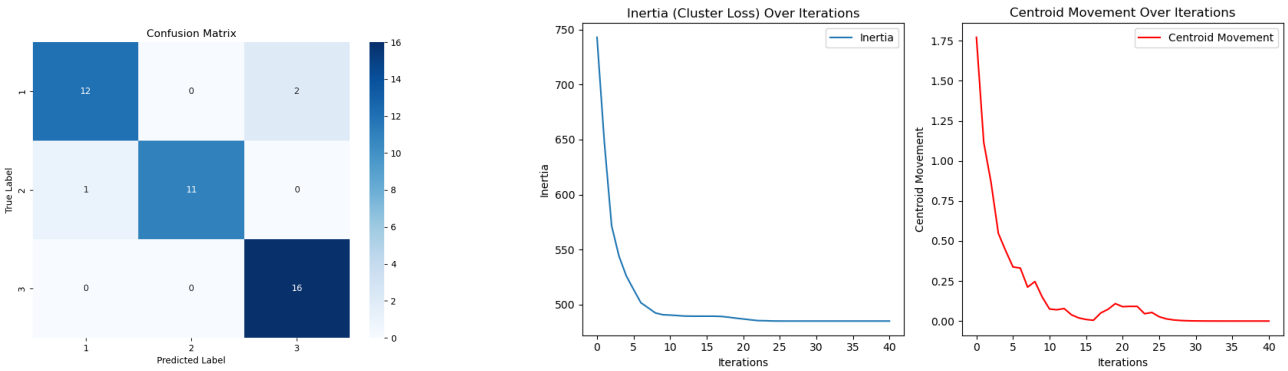


Fig. 6: PCA Result

Then use the dataset shown above, we can classify them easier.

```
kmeans = KMeans(k=3, max_iters=100)
kmeans.fit(train_pca, train_labels)
kmeans.plot_metrics()
```

```
test_predictions = kmeans.predict_and_map(test_pca)
accuracy_score(test_labels, test_predictions)
plot_confusion_matrix(test_labels, test_predictions)
```



(a) Confusion Matrix

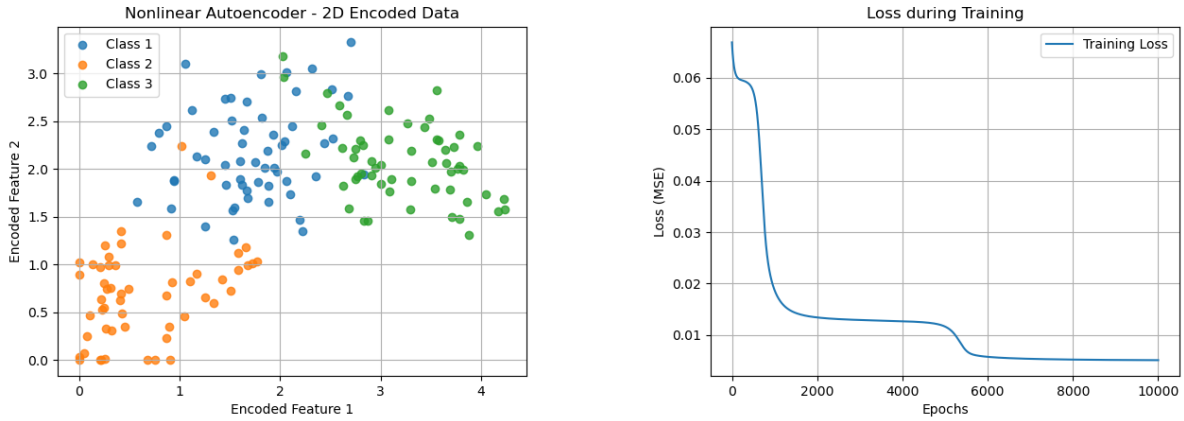
(b) Loss Trend

Fig. 7: K-Means++ and PCA

Converged at iteration 40
Accuracy: 0.9285714285714286

Using Nonlinear Autoencoder, two-dimensional dataset is also generated:

```
autoencoder_2d = NonlinearAutoencoder(input_dim=train_features.shape[1],
    hidden_dims=[64, 16, 2], output_dim=train_features.shape[1], learning_rate=1e
    -3, epochs=10000)
autoencoder_2d.train(train_features)
train_encoded = autoencoder_2d.encode(train_features)
test_encoded = autoencoder_2d.encode(test_features)
autoencoder_2d.visualize(train_encoded, train_labels, n_components=2)
autoencoder_2d.plot_loss(n_components=2)
```

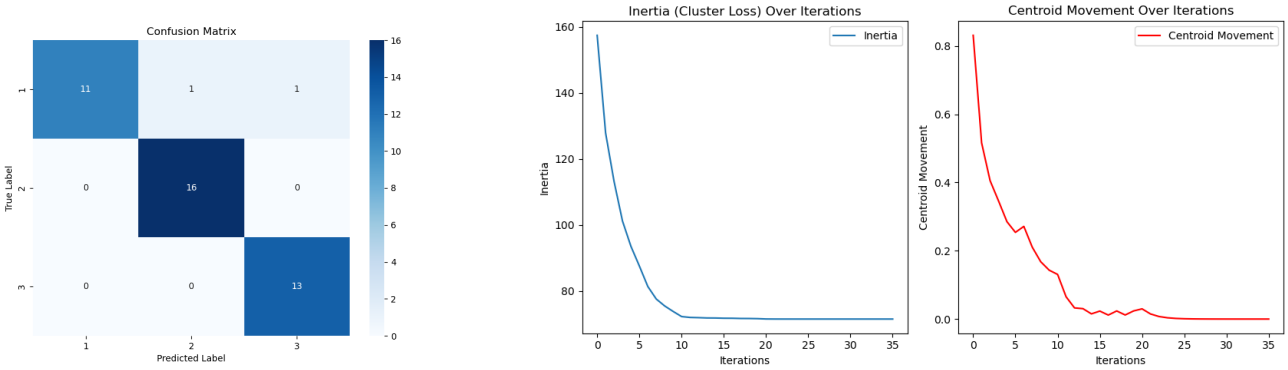


(a) Two Components Dimensionality

(b) Loss Function

Fig. 8: Nonlinear Autoencoder - 2 Dimensions

```
kmeans = KMeans(k=3, max_iters=100)
kmeans.fit(train_encoded, train_labels)
kmeans.plot_metrics()
test_predictions = kmeans.predict_and_map(test_encoded)
accuracy_score(test_labels, test_predictions)
plot_confusion_matrix(test_labels, test_predictions)
```



(a) Confusion Matrix (b) Loss Trend

Fig. 9: K-Means++ and Nonlinear Autoencoder

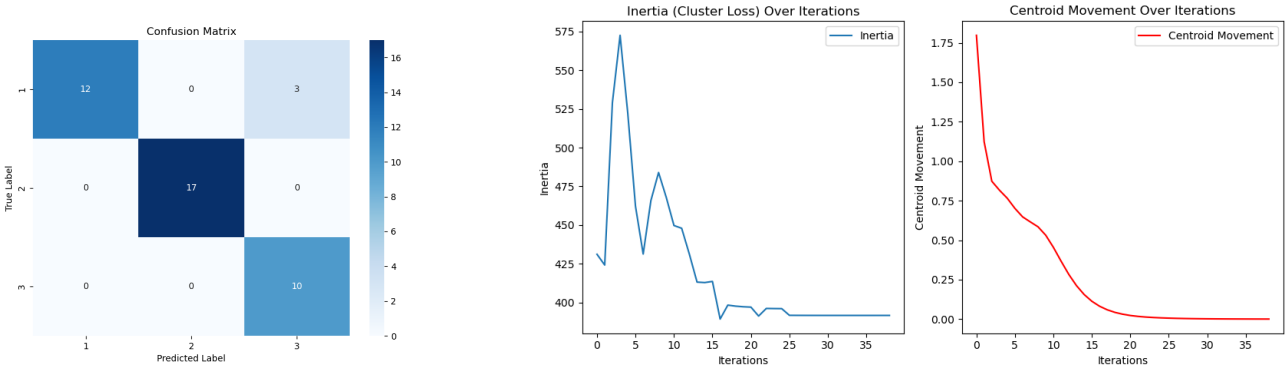
Converged at iteration 35
Accuracy: 0.9523809523809523

4.2 Soft K-Means

Using PCA, we generate two-dimensional dataset:

```
pca = PCA(n_components=2)
pca.fit(train_features)
train_pca = pca.transform(train_features)
test_pca = pca.transform(test_features)
```

Then use the dataset shown above, we can classify them easier.



(a) Confusion Matrix (b) Loss Trend

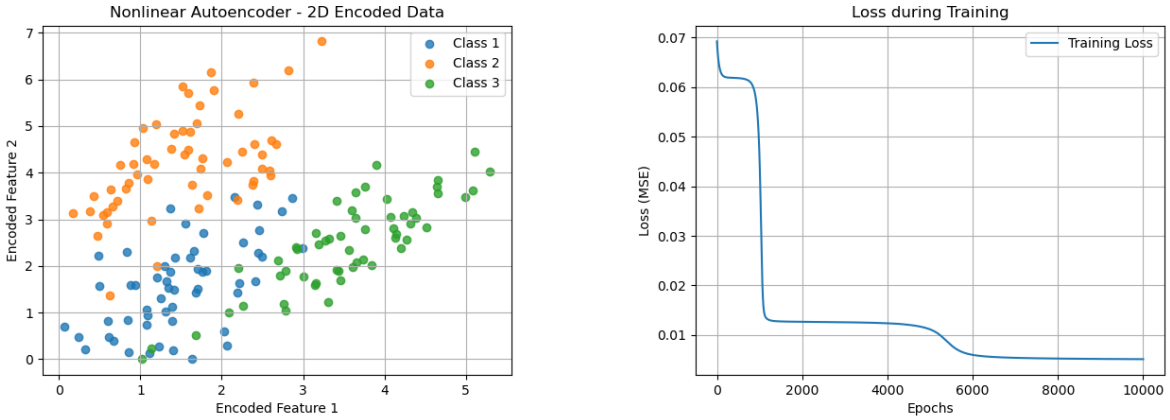
Fig. 10: Soft K-Means and PCA

```
soft_kmeans = SoftKMeans(k=3, max_iters=100, m=1.5)
soft_kmeans.fit(train_pca, train_labels)
soft_kmeans.plot_metrics()
test_predictions = soft_kmeans.predict_and_map(test_pca)
accuracy_score(test_labels, test_predictions)
plot_confusion_matrix(test_labels, test_predictions)
```

Converged at iteration 38
Accuracy: 0.9285714285714286

Using Nonlinear Autoencoder, two-dimensional dataset is also generated:

```
autoencoder_2d = NonlinearAutoencoder(input_dim=train_features.shape[1],
    hidden_dims=[64, 16, 2], output_dim=train_features.shape[1], learning_rate=1e
    -3, epochs=10000)
autoencoder_2d.train(train_features)
train_encoded = autoencoder_2d.encode(train_features)
test_encoded = autoencoder_2d.encode(test_features)
autoencoder_2d.visualize(train_encoded, train_labels, n_components=2)
autoencoder_2d.plot_loss(n_components=2)
```

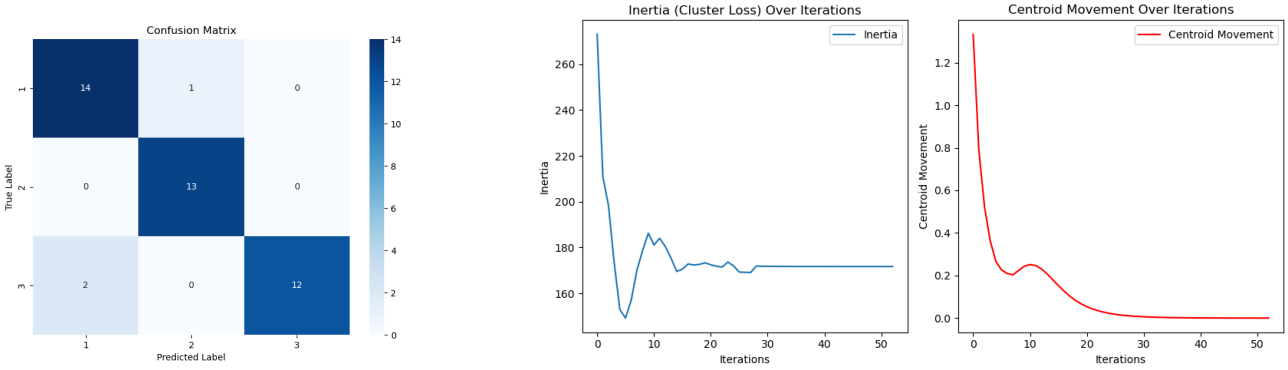


(a) Two Components Dimensionality

(b) Loss Function

Fig. 11: Nonlinear Autoencoder - 2 Dimensions

```
soft_kmeans = SoftKMeans(k=3, max_iters=100, m=1.5)
soft_kmeans.fit(train_encoded, train_labels)
soft_kmeans.plot_metrics()
test_predictions = soft_kmeans.predict_and_map(test_encoded)
accuracy_score(test_labels, test_predictions)
plot_confusion_matrix(test_labels, test_predictions)
```



(a) Confusion Matrix

(b) Loss Trend

Fig. 12: Soft K-Means and Nonlinear Autoencoder

Converged at iteration 52
Accuracy: 0.9285714285714286

4.3 Analyazation

Compared with the data without dimensionality reduction, the dimensionality reduced data is faster and occupies less memory when training, while the training results do not differ much due to the fact that the principal components have already covered most of the information, with only a slight degradation in some cases. Therefore, when classifying datasets with multiple features, PCA (for linear data) or Nonlinear Autoencoder (for nonlinear data) can be used to downsize the dataset before training and prediction.

5 Multi-Layer Perceptron

5.1 Principle

Multilayer Perceptrons (MLP) are feedforward artificial neural networks widely used in classification, regression, and pattern recognition tasks. An MLP consists of at least three layers of neurons: the input layer, one or more hidden layers, and the output layer. Each neuron (neuron) employs a nonlinear activation function, enabling the network to learn and perform complex nonlinear tasks.

1. **Network Structure and Feedforward Mechanism:** In an MLP, each neuron computes a weighted sum of its inputs and then applies an activation function. For the j-th neuron in layer l, the input is given by:

$$a_j^{(l)} = f\left(\sum_{i=1}^{n_{l-1}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}\right) \tag{8}$$

   where $a_i^{(l-1)}$ is the output from the $(l-1)$-th layer, $w_{ij}^{(l)}$ is the weight connecting neuron i in layer $(l-1)$ to neuron j in layer l, $b_j^{(l)}$ is the bias for neuron j in layer l, and f is the activation function.

2. **Activation Function:** The activation function introduces nonlinearity into the network. Common activation functions include Sigmoid, Tanh, and ReLU. For instance, the ReLU function is defined as:

$$f(x) = \max(0, x) \tag{9}$$

3. **Loss Function:** MLP uses a loss function to evaluate the discrepancy between network outputs and true values. Common loss functions include Mean Squared Error (MSE) and Cross-Entropy loss. For classification

tasks, Cross-Entropy loss can be expressed as:

$$L = -\sum_{i=1}^{C} y_i \log(\hat{y}_i) \tag{10}$$

where C is the number of classes, y_i is the true label, and $\hat{y}_i$ is the predicted probability.

4. **Training Process:** MLPs adjust weights to minimize the loss function through optimization algorithms such as gradient descent. This involves computing the gradient of the loss function with respect to each weight and updating the weights accordingly. The weight update rule is:

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \frac{\partial L}{\partial w_{ij}^{(l)}} \tag{11}$$

where η is the learning rate.

5. **Backpropagation Algorithm:** During training, MLPs use the backpropagation algorithm to compute the gradient of the loss function with respect to each weight. This is achieved by propagating the error backward from the output layer to the input layer.

The power of MLPs lies in their ability to capture and learn complex patterns and relationships in data, making them very effective for many machine learning tasks. However, MLPs also have some limitations, such as susceptibility to overfitting, the need for large amounts of data for training, and potentially high computational resource requirements.

5.2 Algorithm and Results

1. **Initialization (`__init__`):** Sets up the network architecture, including layer sizes, learning rate, and regularization parameters. Initializes weights and biases randomly.

2. **Activation (`activation`):** Applies the ReLU activation function to the input.

$$a = \max(0, z) \tag{12}$$

3. **Activation Derivative (`activation_derivative`):** Computes the derivative of the ReLU function.

$$a' = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{otherwise} \end{cases} \tag{13}$$

4. **Softmax (`softmax`):** Applies the softmax function to the output layer for classification.

$$a = \frac{\exp(z - \max(z))}{\sum \exp(z - \max(z))} \tag{14}$$

5. **Dropout (`dropout`):** Randomly sets a fraction of input units to 0 at each update during training time to prevent overfitting.

6. **Forward Pass (`forward`):** Performs a forward pass through the network, applying dropout and activation functions.

7. **Compute Loss (`compute_loss`):** Calculates the cross-entropy loss with L2 regularization.

$$L = -\frac{1}{m} \sum (y \log(\hat{y})) + \lambda \sum w^2 \tag{15}$$

8. **Backward Pass (`backward`):** Computes gradients using backpropagation and updates weights and biases.

9. **Update Parameters (`update_params`):** Updates weights and biases using the Adam optimization algorithm.

Algorithm 8 Training Process of MultiLayerPerceptron

Input: Training data (X, Y), test data (X_{test}, Y_{test}), number of iterations n_iter, batch size $batch\_size$
Output: Trained model with loss history
Initialize loss history arrays: $train\_loss\_history = []$, $test\_loss\_history = []$
One-hot encode targets: $Y\_one\_hot = \text{one\_hot\_encode}(Y, \text{layer\_sizes}[-1])$
One-hot encode test targets: $Y_{test\_one\_hot} = \text{one\_hot\_encode}(Y_{test}, \text{layer\_sizes}[-1])$
for $epoch = 1$ to n_iter **do**
  Shuffle training data indices: $indices = \text{shuffle}(\text{indices}(X))$
  for $start = 0$ to $|X|$ step $batch\_size$ **do**
    Compute batch end index: $end = \min(start + batch\_size, |X|)$
    Extract batch inputs and targets: $X\_batch = X[indices[start : end]]$, $Y\_batch = Y\_one\_hot[indices[start : end]]$
    Perform forward pass: $Z = \text{forward}(X\_batch)$
    Perform backward pass: $\text{backward}(X\_batch, Y\_batch, epoch)$
  end for
  Compute and record training loss: $train\_loss = \text{compute\_loss}(\text{forward}(X), Y\_one\_hot)$
  Compute and record test loss: $test\_loss = \text{compute\_loss}(\text{forward}(X_{test}), Y_{test\_one\_hot})$
  Append losses to history: $train\_loss\_history.append(train\_loss)$, $test\_loss\_history.append(test\_loss)$
  Decay learning rate: $lr = 0.999 \times lr$
end for
return $train\_loss\_history$, $test\_loss\_history$

10. **Training (`train`):** Trains the network using mini-batch gradient descent, with optional dropout and L2 regularization.
11. **Prediction (`predict`):** Makes predictions by passing inputs through the network and selecting the class with the highest probability.
12. **Plot Loss (`plot_loss`):** Visualizes the training and testing loss over iterations.
13. **Plot Weights and Biases (`plot_weights_and_biases`):** Visualizes the distribution of weights and biases for each layer.

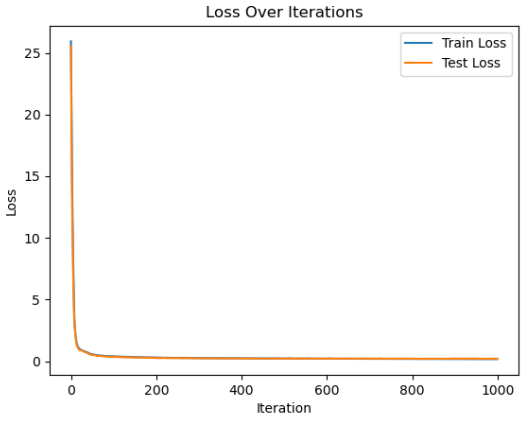
```
mlp = MultiLayerPerceptron([7, 64, 80, 32, 10], n_iter=1000, lr=3e-4, batch_size
    =32)
```

Training Record:
Epoch 1/1000, Train Loss: 25.9276, Test Loss: 25.5430
Epoch 40/1000, Train Loss: 0.7140, Test Loss: 0.6661
Epoch 80/1000, Train Loss: 0.4441, Test Loss: 0.4046
Epoch 120/1000, Train Loss: 0.3791, Test Loss: 0.3347
Epoch 160/1000, Train Loss: 0.3408, Test Loss: 0.3162
Epoch 200/1000, Train Loss: 0.3044, Test Loss: 0.2764
Epoch 240/1000, Train Loss: 0.2856, Test Loss: 0.2622
Epoch 280/1000, Train Loss: 0.2730, Test Loss: 0.2635
Epoch 320/1000, Train Loss: 0.2612, Test Loss: 0.2430
Epoch 360/1000, Train Loss: 0.2565, Test Loss: 0.2318
Epoch 400/1000, Train Loss: 0.2470, Test Loss: 0.2285
Epoch 440/1000, Train Loss: 0.2586, Test Loss: 0.2359
Epoch 480/1000, Train Loss: 0.2367, Test Loss: 0.2371
Epoch 520/1000, Train Loss: 0.2333, Test Loss: 0.2216
Epoch 560/1000, Train Loss: 0.2216, Test Loss: 0.2187
Epoch 600/1000, Train Loss: 0.2195, Test Loss: 0.2276
Epoch 640/1000, Train Loss: 0.2126, Test Loss: 0.2236

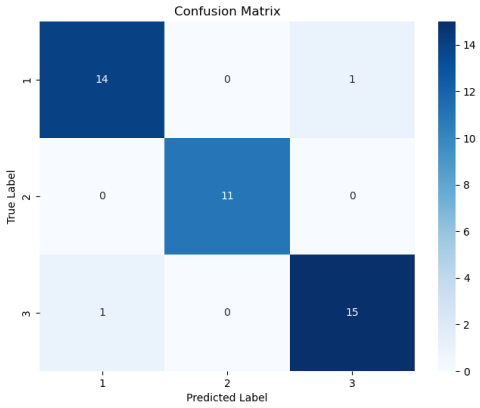
Epoch 680/1000, Train Loss: 0.2077, Test Loss: 0.2207
Epoch 720/1000, Train Loss: 0.2029, Test Loss: 0.2176
Epoch 760/1000, Train Loss: 0.1994, Test Loss: 0.2170
Epoch 800/1000, Train Loss: 0.1946, Test Loss: 0.2154
Epoch 840/1000, Train Loss: 0.1921, Test Loss: 0.2146
Epoch 880/1000, Train Loss: 0.1890, Test Loss: 0.2202
Epoch 920/1000, Train Loss: 0.1840, Test Loss: 0.2092
Epoch 960/1000, Train Loss: 0.1823, Test Loss: 0.2157
Epoch 1000/1000, Train Loss: 0.1776, Test Loss: 0.2115

Accuracy: 0.9523809523809523

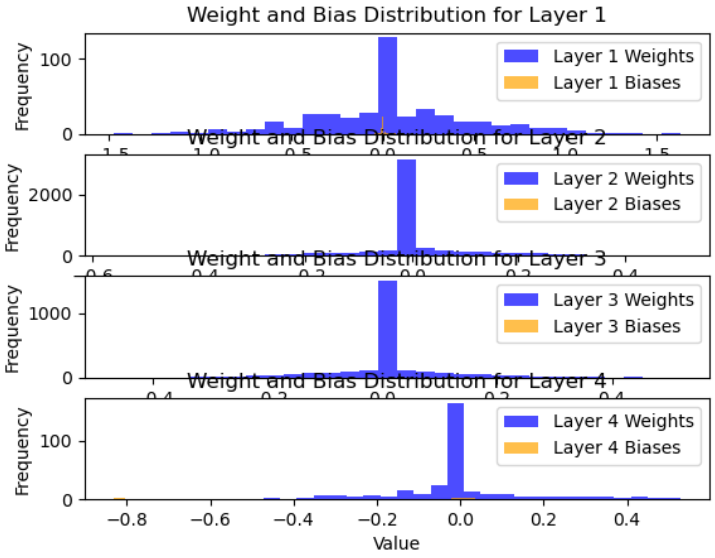
In many trials, mlp works all the time, which convince me that, mlp is still better than kmeans. (Although at the cost of time.)



(a) Loss Trend



(b) Confusion Matrix



(c) Weights and Bias Distribution

Fig. 13: Multi-Layer Perceptron

5.3 Analyzation

Advantages:

- **Strong Nonlinear Modeling:** MLPs, with their multiple hidden layers and nonlinear activation functions, can learn complex feature representations, enhancing the model's expressive power.
- **Flexibility:** They are applicable to a variety of machine learning tasks such as classification, regression, and clustering, offering robust generalization capabilities.
- **Automatic Feature Learning:** MLPs automatically extract features from raw data, reducing the need for manual intervention and potentially improving model accuracy and generalization.
- **Wide Range of Applications:** They are extensively used in image recognition, speech recognition, natural language processing, and other fields.

Disadvantages:

- **Long Training Time:** MLPs require significant computational resources and time for training due to the large number of parameters that need to be iteratively updated.
- **Overfitting:** The high model complexity of MLPs can lead to overfitting, where the model performs well on training data but poorly on unseen data. Techniques such as regularization and early stopping are necessary to mitigate this.
- **High Data Requirements:** MLPs demand high-quality and substantial amounts of data. Poor data quality or insufficient data can negatively impact model performance, necessitating careful data preparation and augmentation.
- **Lack of Interpretability:** Compared to models like decision trees, the decision-making process in MLPs is less interpretable and understandable, which can be a drawback in certain domains.

The performance of MLPs is highly dependent on the selection of hyperparameters, such as learning rate, batch size, and the number of iterations. Additionally, MLPs may not be as effective as Convolutional Neural Networks (CNNs) for data with spatial information and can be computationally intensive, requiring substantial memory and processing power during training. Despite these drawbacks, MLPs are widely used for their simplicity and broad applicability, although they may not perform as well as specialized network architectures like CNNs and Recurrent Neural Networks (RNNs) for large-scale datasets and complex tasks.

6 SVM

6.1 Principle

Support Vector Machines (SVMs) are robust supervised learning models, highly effective for classification and, with certain extensions, for regression tasks. They are particularly well-suited for scenarios where data is linearly separable in a high-dimensional space.

SVM operates on the principle of maximizing the margin between classes, which is the gap between the closest points of different classes, known as support vectors. The hyperplane that achieves this is termed the optimal separating hyperplane.

The SVM optimization problem is formulated as a convex quadratic programming problem:

$$\min_{w,b} \frac{1}{2}||w||^2 \quad \text{subject to} \quad y_i(w \cdot x_i + b) \geq 1, \quad i = 1, \ldots, n \tag{16}$$

where w represents the weight vector, b is the bias term, x_i are the feature vectors, and y_i are the corresponding class labels.

For datasets that are not linearly separable, SVM employs the kernel trick to map the input space into a higher-dimensional space where a linear separation is feasible. Common kernels include:

- Linear: $K(x_i, x_j) = x_i \cdot x_j$
- Polynomial: $K(x_i, x_j) = (\gamma x_i \cdot x_j + r)^d$
- Radial Basis Function (RBF): $K(x_i, x_j) = \exp(-\gamma ||x_i - x_j||^2)$

where γ is a parameter that defines the width of the Gaussian kernel, and r and d are parameters for the polynomial kernel.

6.2 Algorithm and Results

Algorithm 9 SVM Fit Function

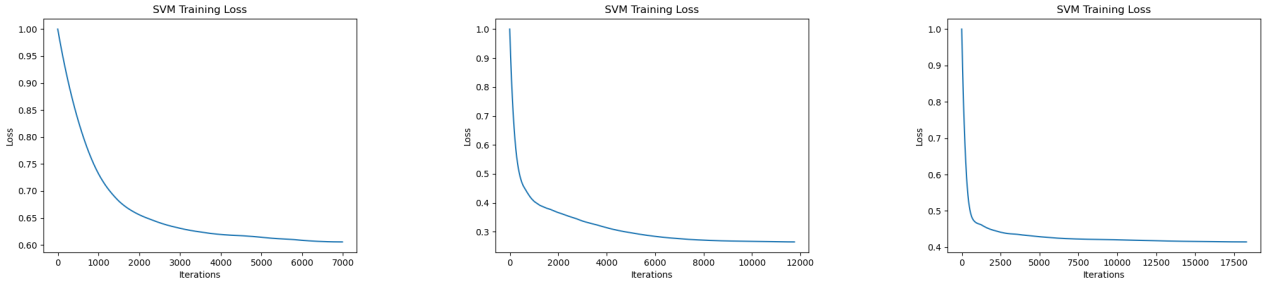
Convert labels to -1 and 1.
Normalize feature data.
Compute kernel matrix K.
Initialize Lagrange multipliers α.
for each iteration up to $n_{\text{iteration}}$ **do**
  Compute decision values.
  Update gradients.
  Update α values.
  Compute loss.
  if loss does not improve **then**
    Increment no improvement count.
  else
    Reset no improvement count.
  end if
  if no improvement count reaches early stopping patience **then**
    Stop training.
  end if
end for
Save α values and support vector mask.
Plot loss curve.

Algorithm 10 SVM Predict Function

Normalize test data.
Get support vectors and corresponding labels and α values.
Calculate bias term b.
Compute decision function scores for test samples.
Return predicted labels (sign of decision function scores).

Linear Kernal:

```
svm = SVM(kernel='linear', C=1, k=3, n_iteration=30000, lr=0.05,
    early_stopping_patience=10, tol=1e-6)
svm.fit(train_features, train_labels)
test_pred = svm.get_predictions(test_features)
accuracy_score(test_labels, test_pred)
plot_confusion_matrix(test_labels, test_pred)
```



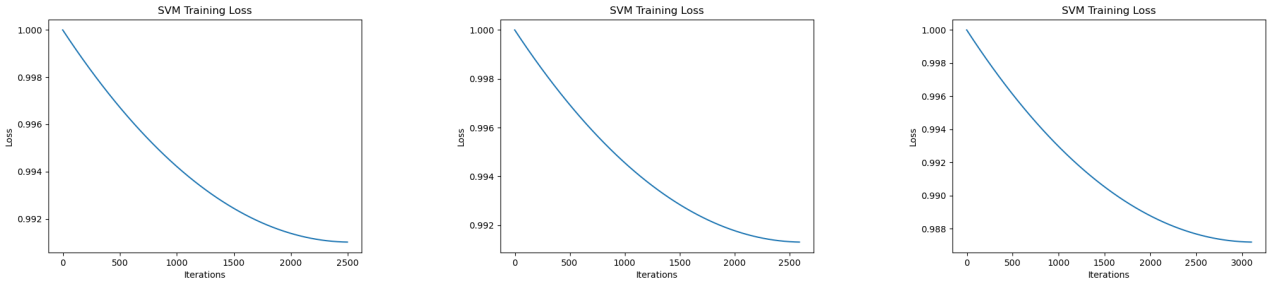
(a) First Classifier (b) Second Classifier (c) Thrid Classifier

Fig. 14: Support Vector Machine - Linear Kernal

Early stopping at iteration 6985
Early stopping at iteration 11750
Early stopping at iteration 18295
Accuracy: 0.9047619047619048

Gaussian Kernal:

```
svm = SVM(kernel='gaussian', C=0.5, k=3, n_iteration=30000, lr=2,
    early_stopping_patience=100, tol=1e-12)
svm.fit(train_features, train_labels)
test_pred = svm.get_predictions(test_features)
accuracy_score(test_labels, test_pred)
plot_confusion_matrix(test_labels, test_pred)
```

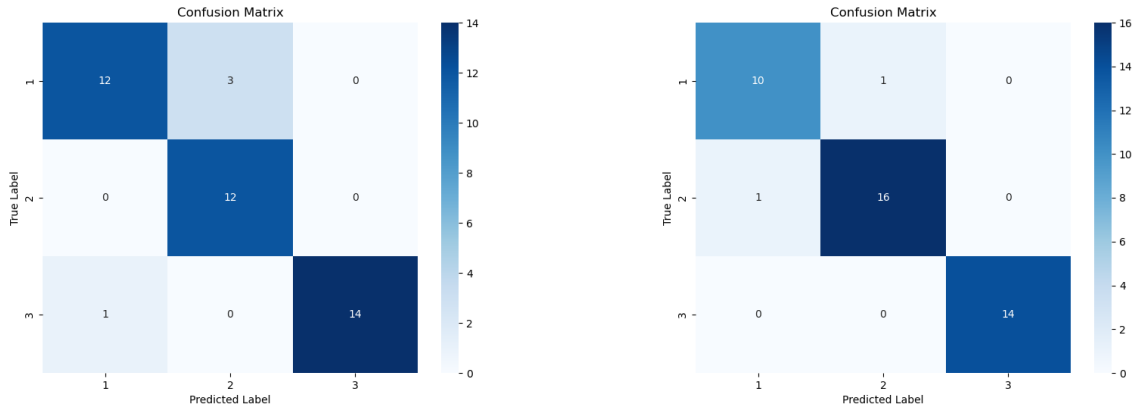


(a) First Classifier (b) Second Classifier (c) Thrid Classifier

Fig. 15: Support Vector Machine - Gaussian Kernal

Early stopping at iteration 2496
Early stopping at iteration 2590
Early stopping at iteration 3102
Accuracy: 0.9523809523809523

Cause our dataset is nonlinear, Gaussian kernal performs better and converges faster than linear kernal.



(a) Linear Kernal

(b) Gaussian Kernal

Fig. 16: Support Vector Machine - Classification Metrics

6.3 Analyzation

Advantages:

- Effective in high-dimensional spaces and in cases where the number of dimensions exceeds the number of samples.
- Memory efficiency is achieved by using only a subset of the training data (support vectors) for the decision function.
- Versatility is provided by the ability to use different kernel functions to fit various data distributions.

Disadvantages:

- Scalability issues arise with large datasets due to the computational complexity and memory requirements associated with quadratic programming.
- Performance is sensitive to parameter selection, including the regularization parameter C and the kernel parameters.
- Lacks interpretability compared to some other models, which can be a drawback in applications where understanding the decision process is crucial.

SVMs have been widely adopted across diverse fields, including bioinformatics, text categorization, and image recognition, owing to their ability to handle complex classification tasks with high accuracy. However, their application may be limited in scenarios involving extremely large datasets or those requiring real-time classification.

7 Adaboost

7.1 Principle

AdaBoost is an ensemble learning technique that constructs a strong classifier by combining multiple weak classifiers, each of which is only slightly better than random guessing. The algorithm iteratively trains weak learners, adjusting the weights of training samples to focus on instances that are misclassified by the previous weak learners.

1. Initialization of Sample Weights

Let $D_1 = \{(w_{1i}, y_i)\}_{i=1}^N$ be the initial distribution of training samples, where w_{1i} is the weight of the i-th sample and y_i is the corresponding label. Initially, all samples are assigned equal weights, i.e.,

$$w_{1i} = \frac{1}{N}$$

2. Iterative Training Process
For $t = 1, 2, \ldots, T$, where T is the total number of iterations, the following steps are performed:

- **Weak Learner Training**: Train a weak learner $h_t(x)$ on the training set D_t to minimize the weighted error:

$$\epsilon_t = \sum_{i=1}^{N} w_{ti} \mathbb{I}(h_t(x_i) \neq y_i)$$

  where $\mathbb{I}$ is the indicator function that equals 1 if the condition inside is true and 0 otherwise.

- **Weight Update**: Calculate the weight α_t of the t-th weak learner using the formula:

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \epsilon_t}{\epsilon_t} \right)$$

  This weight reflects the performance of the weak learner, with higher weights assigned to more accurate classifiers.

- **Sample Re-weighting**: Update the weights of the samples for the next iteration:

$$w_{(t+1)i} = \frac{w_{ti}}{Z_t} \exp(-\alpha_t y_i h_t(x_i))$$

  where Z_t is a normalization factor ensuring that the weights sum up to 1:

$$Z_t = \sum_{i=1}^{N} w_{ti} \exp(-\alpha_t y_i h_t(x_i))$$

3. Final Strong Classifier
The final strong classifier $H(x)$ is a weighted majority vote of the weak learners:

$$H(x) = \text{sign} \left(\sum_{t=1}^{T} \alpha_t h_t(x) \right)$$

where sign function returns the sign of the real-valued sum, effectively aggregating the predictions of the weak learners.

AdaBoost's effectiveness stems from its ability to iteratively adjust sample weights, thereby focusing on misclassified instances and enhancing the overall classification performance. However, the algorithm is sensitive to noise and outliers, as misclassified samples gain higher weights. Additionally, there is a risk of overfitting, especially when a large number of weak learners are combined.

Algorithm 11 AdaBoost Multiclass Fit Algorithm

for $class\_id \in \{1, 2, \ldots, \text{num\_classes}\}$ **do**
  $y\_binary \leftarrow \text{convert}(y == class\_id, 1, -1)$
  $\text{sample\_weights} \leftarrow \frac{1}{m}$
  for $i \in \{1, 2, \ldots, \text{n\_estimators}\}$ **do**
    $\text{stump} \leftarrow \text{train DecisionStump}(X, y\_binary, \text{sample\_weights})$
    $\text{predictions} \leftarrow \text{stump.predict}(X)$
    $\text{errors} \leftarrow (\text{predictions} \neq y\_binary)$
    $\text{weighted\_error} \leftarrow \frac{\sum(\text{sample\_weights} \cdot \text{errors})}{\sum(\text{sample\_weights})}$
    $\alpha \leftarrow 0.5 \ln\left(\frac{1 - \text{weighted\_error}}{\text{weighted\_error}}\right)$
    $\text{sample\_weights} \leftarrow \text{sample\_weights} \cdot \exp(-\alpha y\_binary \text{predictions})$
    $\text{sample\_weights} \leftarrow \frac{\text{sample\_weights}}{\sum(\text{sample\_weights})}$
    append stump, α to self.models
  end for
end for

7.2 Algorithm and Results

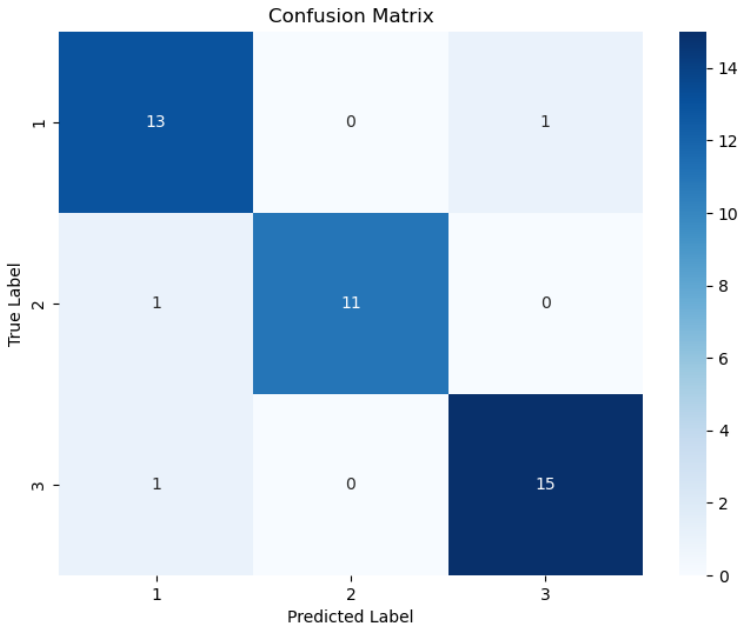


Fig. 17: Adaboost

Accuracy: 0.9285714285714286

7.3 Analyzation

Advantages:

- **High Accuracy:** AdaBoost combines multiple weak classifiers to improve classification accuracy, especially on complex datasets.

- **Flexibility:** AdaBoost can use various regression and classification models as weak learners, making it highly flexible.
- **Adaptability:** By dynamically adjusting sample and learner weights, AdaBoost is highly adaptable and performs well on various types of datasets.
- **Resilience to Overfitting:** Compared to other ensemble methods, AdaBoost is less prone to overfitting, especially with large datasets.
- **Automatic Feature Selection:** AdaBoost can automatically select effective features and ignore irrelevant or noisy features.

Disadvantages:

- **Sensitivity to Outliers:** AdaBoost increases the weights of misclassified samples, making it sensitive to noise and outliers.
- **Potentially High Computational Cost:** Due to its iterative nature, AdaBoost may require significant computational resources, especially with large-scale datasets or high-dimensional features.
- **Poor Interpretability:** While individual weak learners may be interpretable, the ensemble model becomes less so, reducing its interpretability.
- **Long Training Time:** The iterative training process of AdaBoost can lead to longer training times compared to some other algorithms.
- **Limited Parallelization:** The sequential nature of AdaBoost training, where each classifier is trained based on the previous one's outcome, limits its ability to be parallelized.

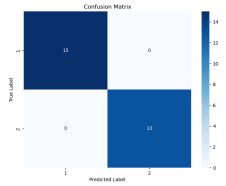
8 Binary Problem

Table 1: Accuracy for Different Models When Removing Each Class

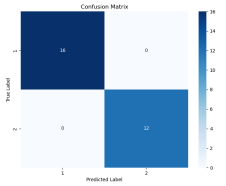
Removing Class	MLP	SVM Linear	SVM Gaussian	AdaBoost
Class 1	1.0	1.0	1.0	1.0
Class 2	0.928571	0.928571	0.928571	0.964286
Class 3	1.0	0.928571	0.964286	0.964286

- With the removal of categories 1 and 3, all models showed perfect accuracy, which could be attributed to the fact that the features of these categories are significantly different from the other categories, making the models easy to learn.
- With the removal of category 2, the accuracy of all models decreased, but AdaBoost showed the smallest decrease, which may be attributed to the fact that the AdaBoost algorithm is more adaptable when dealing with unbalanced data or when certain category features are not obvious.
- Overall, AdaBoost shows better robustness in dealing with missing categories, especially in removing category 2, and its accuracy is higher than the other models.

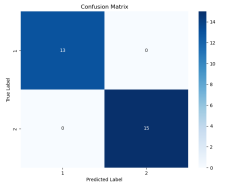
8.1 Remove Class 1



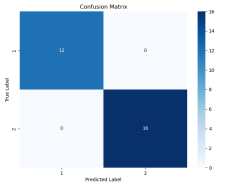
(a) AdaBoost Confusion Matrix



(b) MLP Confusion Matrix



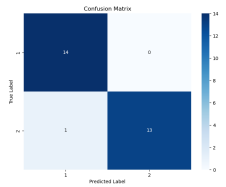
(c) SVM Linear Confusion Matrix



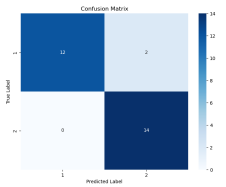
(d) SVM Gaussian Confusion Matrix

Fig. 18: Confusion Matrices for Different Algorithms

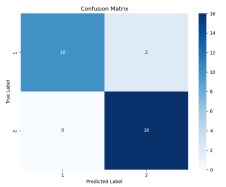
8.2 Remove Class 2



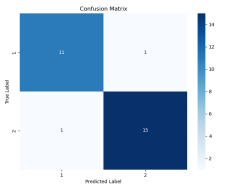
(a) AdaBoost Confusion Matrix



(b) MLP Confusion Matrix



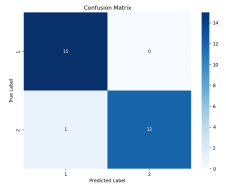
(c) SVM Linear Confusion Matrix



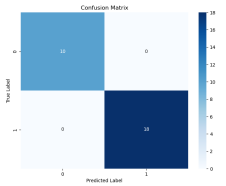
(d) SVM Gaussian Confusion Matrix

Fig. 19: Confusion Matrices for Different Algorithms

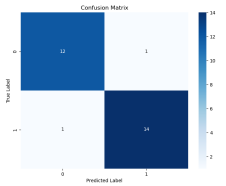
8.3 Remove Class 3



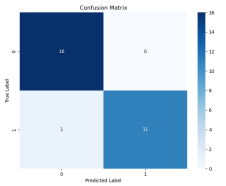
(a) AdaBoost Confusion Matrix



(b) MLP Confusion Matrix



(c) SVM Linear Confusion Matrix



(d) SVM Gaussian Confusion Matrix

Fig. 20: Confusion Matrices for Different Algorithms

9 Conclusion

The project aimed to explore and evaluate various machine learning algorithms for wheat seed classification using the Wheat Seed Dataset. The comprehensive analysis included the application of K-Means, K-Means++, Soft K-Means, Principal Component Analysis (PCA), Nonlinear Autoencoders, Multi-Layer Perceptrons (MLP), Support Vector Machines (SVM), and AdaBoost. Each method was assessed for its effectiveness in classifying the three distinct wheat seed varieties: Kama, Rosa, and Canadian. The results provide valuable insights into the performance characteristics of these algorithms and their suitability for agricultural science applications.

K-Means Clustering
The K-Means algorithm demonstrated simplicity and efficiency in partitioning the dataset into clusters. Its iterative approach allowed for the minimization of the sum of squared distances within each cluster. However, the algorithm's

sensitivity to the initial choice of centroids and the assumption of convex clusters limit its applicability in scenarios with complex data distributions. The introduction of K-Means++ improved the initialization of centroids, leading to better clustering results and faster convergence. Soft K-Means, with its fuzzy assignment of data points to multiple clusters, offered a more flexible partitioning, particularly useful for datasets with overlapping clusters.

Dimensionality Reduction
Dimensionality reduction techniques, such as PCA and Nonlinear Autoencoders, were employed to simplify the complexity of the dataset while retaining significant information. PCA was effective for linear data, reducing dimensions through the identification of principal components. Nonlinear Autoencoders excelled in capturing complex, non-linear relationships within the data, making them suitable for datasets with intricate structures. The reduced-dimensional data facilitated faster training and reduced memory usage, with minimal impact on classification accuracy.

Multi-Layer Perceptrons (MLP)
MLPs showcased their ability to learn complex function mappings through multiple hidden layers and non-linear activation functions. They were effective in classification tasks, with the capacity to automatically extract features from raw data. However, MLPs require substantial computational resources and are prone to overfitting, necessitating careful hyperparameter tuning and regularization techniques.

Support Vector Machines (SVM)
SVMs were particularly effective in maximizing the margin between classes, with the Gaussian kernel outperforming the linear kernel due to the non-linear nature of the dataset. SVMs are memory efficient and versatile, leveraging different kernel functions to accommodate various data distributions. However, their performance is sensitive to parameter selection, and they can be computationally intensive for large datasets.

AdaBoost
AdaBoost demonstrated robustness by combining multiple weak classifiers to form a strong classifier. Its adaptive nature allowed it to focus on misclassified instances, improving overall classification performance. AdaBoost was less sensitive to outliers and offered automatic feature selection. Despite its advantages, it is sensitive to noise, has potential high computational costs, and suffers from poor interpretability.

Binary Problem Analysis
The study also explored the impact of removing individual classes on the classification accuracy of various models. The removal of certain classes, particularly Class 2, affected the accuracy of all models, with AdaBoost showing the highest resilience. This suggests that AdaBoost is more adaptable in handling unbalanced data or less distinct category features.

Conclusion
In conclusion, the comparative analysis of these machine learning algorithms on the Wheat Seed Dataset has highlighted the strengths and limitations of each approach. While K-Means and its variants are straightforward and efficient for spherical clusters, PCA and Nonlinear Autoencoders are effective in reducing data dimensions, particularly for high-dimensional datasets. MLPs and SVMs are powerful in learning complex patterns, though they require careful management to avoid overfitting and are computationally demanding. AdaBoost stands out for its robustness and adaptability, especially in the presence of missing or unbalanced classes. The choice of algorithm depends on the specific characteristics of the dataset and the requirements of the classification task. For agricultural applications, these findings can guide the selection of appropriate machine learning models for seed classification, potentially aiding in the identification and selection of high-quality seeds, thereby enhancing crop yield and quality.

All the works are in https://github.com/Wendy-Ying/Wheat-Seed-Classification-Prediction.