

# **Final Project Report**

## **Team Members:**

[Liu Hanfei] – [20222812]

[Fan Tengwen] – [20222826]

[Kang Yuxin] – [20222827]

[Sun Ruiquan] – [20222786]

[Wu Yonghui] – [20222830]

[Zhang Tongyi] – [20222832]

Submission Date: March 28, 2025

## Contents

<b>PART I: Using Scikit-Learn</b> .....	<b>1</b>
<b>1.1 Feature extraction and dimensionality reduction</b> .....	<b>1</b>
<b>1.2 Performance Comparison of Machine Learning Models</b> .....	<b>1</b>
<b>1.3 Grid search parameter adjustment</b> .....	<b>3</b>
<b>PART II: Train MLP</b> .....	<b>4</b>
<b>2.1 Activation function</b> .....	<b>4</b>
<b>2.2 Incremental Learning</b> .....	<b>5</b>
<b>2.3 Multi-layer MLP</b> .....	<b>6</b>
<b>2.4 Model Training Results</b> .....	<b>8</b>
<b>PART III: SUMMARY</b> .....	<b>9</b>

## PART I: Using Scikit-Learn

### 1.1 Feature extraction and dimensionality reduction

In Step 1 of Part I, our task was to perform feature extraction and dimensionality reduction on the dataset using Polynomial Features, Linear Discriminant Analysis (LDA), and Principal Component Analysis (PCA). The original dataset contains more than 180 features, and such high dimensionality is prone to causing overfitting or underfitting issues during model training. Therefore, we first standardized the data and then applied PolynomialFeatures, LDA, and PCA algorithms for feature processing. To evaluate the effectiveness of dimensionality reduction, we trained and evaluated the processed data using a simple Random Forest classifier. All the above steps were implemented through a pipeline to ensure the efficiency and consistency of data processing and model training. The final results are shown in Figure 1.

```
*****
Choosing Polynomial for feature extraction!
poly accuracy: 0.9983085659055213
*****
*****
Choosing LDA for feature extraction!
lda accuracy: 0.8617856711368853
*****
*****
Choosing PCA for feature extraction!
pca accuracy: 0.9492569771656397
*****
*****
Comparison of feature extraction methods:
*****
poly: 0.9983085659055213
*****
*****
lda: 0.8617856711368853
*****
*****
pca: 0.9492569771656397
*****
*****
```

Figure 1: Data preprocessing results

### 1.2 Performance Comparison of Machine Learning Models

In Step 1 of Part II, we compared the performance of three feature extraction methods (PolynomialFeatures, LDA, and PCA) using a Random Forest classifier. The experimental results revealed significant differences in accuracy among the three methods.

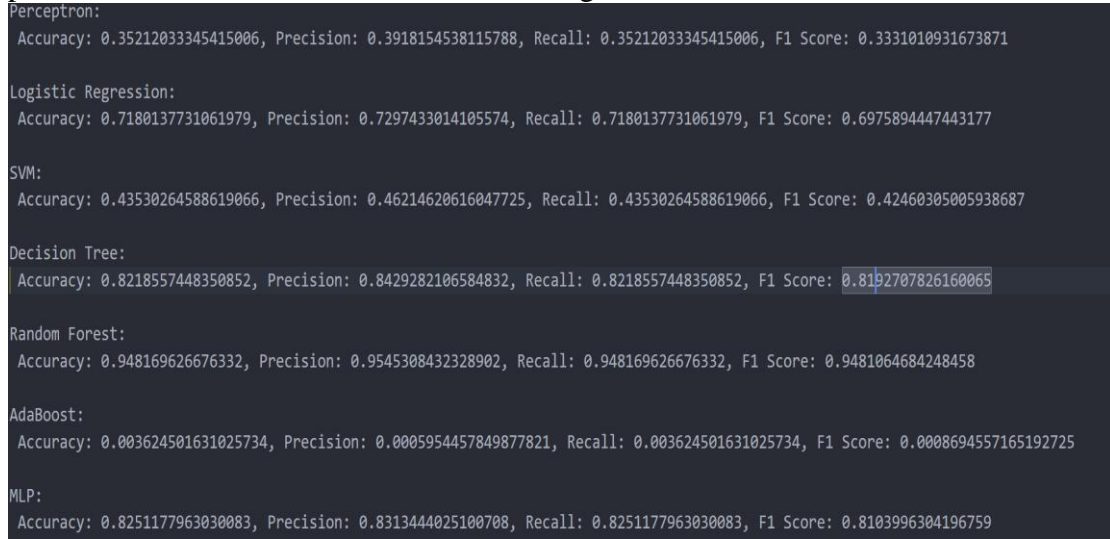
First, PolynomialFeatures demonstrated the most outstanding performance, achieving an accuracy of 0.9986, which is close to 100%. This indicates that by generating polynomial features, the model can effectively capture nonlinear relationships in the data, significantly enhancing classification performance. However, I remain concerned about the additional complexity introduced by the increased number of features, as it may pose a risk of overfitting.

Second, the LDA algorithm achieved an accuracy of 0.8602, which is relatively lower. This suboptimal performance may be attributed to the computational resource constraints during the processing. Due to the lengthy computation time required for

handling all 180+ features, we first applied PCA for dimensionality reduction before performing LDA, which might have impacted the results.

Finally, the PCA algorithm achieved an accuracy of 0.9482, which lies between the results of PolynomialFeatures and LDA. We selected  $n=50$  as the number of components based on the elbow method, which is a reasonable choice. From the results, I am inclined to consider the outcome of the PCA algorithm as more reasonable and reliable. Therefore, subsequent preprocessing steps will be based on the data processed by PCA.

In this experiment, we compared the performance of multiple machine learning models in a classification task, including Perceptron, Logistic Regression, Support Vector Machine (SVM), Decision Tree, Random Forest, AdaBoost, and Multilayer Perceptron (MLP). The experimental results revealed significant differences in the performance of these models, as shown in Figure 2 and Table 1.



Model	Accuracy	Precision	Recall	F1 Score
Perceptron:	0.35212033345415006	0.3918154538115788	0.35212033345415006	0.3331010931673871
Logistic Regression:	0.7180137731061979	0.7297433014105574	0.7180137731061979	0.6975894447443177
SVM:	0.43530264588619066	0.46214620616047725	0.43530264588619066	0.42460305005938687
Decision Tree:	0.8218557448350852	0.8429282106584832	0.8218557448350852	0.8192707826160065
Random Forest:	0.948169626676332	0.9545308432328902	0.948169626676332	0.9481064684248458
AdaBoost:	0.003624501631025734	0.0005954457849877821	0.003624501631025734	0.0008694557165192725
MLP:	0.8251177963030083	0.8313444025100708	0.8251177963030083	0.8103996304196759

Figure 2: Model Training Results Table

**Random Forest** demonstrated the most outstanding performance, with both accuracy and F1 score approaching 95%, indicating its ability to effectively handle the complexity and nonlinear characteristics of the data while reducing the risk of overfitting through ensemble learning. **Decision Tree** and **MLP** also performed well, with accuracies of 82.19% and 82.51%, respectively, suggesting their capability to capture nonlinear relationships in the data, although their performance was slightly lower than that of Random Forest. **Logistic Regression** showed moderate performance, with an accuracy of 71.80%, making it suitable for linearly separable data.

Table 1: Model Training Results Table

Model	Accuracy	Precision	Recall	F1 Score
Perceptron	0.3521	0.3918	0.3521	0.3331
Logistic Regression	0.7180	0.7297	0.7180	0.6976
SVM	0.4353	0.4621	0.4353	0.4246
Decision Tree	0.8219	0.8429	0.8219	0.8193
Random Forest	0.9482	0.9545	0.9482	0.9481
AdaBoost	0.0036	0.0006	0.0036	0.0009
MLP	0.8251	0.8313	0.8251	0.8104

In contrast, **Perceptron** and **SVM** performed poorly, with accuracies of 35.21% and 43.53%, respectively, likely due to their inability to effectively handle the nonlinear characteristics of the data. Particularly, **AdaBoost** performed extremely poorly, with an accuracy of only 0.36%, which may be attributed to inappropriate base classifier selection or data distribution unsuitability for AdaBoost.

### 1.3 Grid search parameter adjustment

In **Step 3**, we performed hyperparameter tuning for the Random Forest classifier (RandomForestClassifier) using GridSearchCV. The grid search configuration included data standardization (StandardScaler), PCA dimensionality reduction (retaining 50 principal components), and the Random Forest classifier. The parameter grid covered the number of decision trees (n\_estimators), maximum depth (max\_depth), and minimum samples per leaf node (min\_samples\_leaf), and optimization was conducted using 5-fold cross-validation with accuracy as the scoring metric, as shown in the code.

```
param_grid = {
    'classifier__n_estimators': [50, 100, 200],
    'classifier__max_depth': [None, 5, 10],
    #'classifier__min_samples_split': [2, 5, 10],
    'classifier__min_samples_leaf': [1, 2],
}
```

As shown in Figure 3, the grid search results indicate that the optimal parameter combination is {'classifier\_\_max\_depth': None, 'classifier\_\_min\_samples\_leaf': 1, 'classifier\_\_n\_estimators': 200}. This suggests the following:

- ① Not limiting the maximum depth of the decision trees (max\_depth=None) allows the model to fully fit the data.
- ② Setting the minimum number of samples per leaf node to 1 (min\_samples\_leaf=1) enables the model to capture data details more flexibly.
- ③ Using 200 decision trees (n\_estimators=200) improves the stability and generalization ability of the model.

```
Fitting 5 folds for each of 18 candidates, totalling 90 fits
Best parameters found: {'classifier__max_depth': None, 'classifier__min_samples_leaf': 1, 'classifier__n_estimators': 200}
```

Figure 3: GridSearchCV result

As shown in Figure 4, the model trained with the optimal parameters performed excellently on the test set, achieving an accuracy of **95.17%**, with precision, recall, and F1 score also close to 95%. Compared to the untuned Random Forest model (accuracy of 94.82%), the performance of the tuned model showed a slight improvement. Additionally, the Random Forest significantly outperformed other models (such as Decision Tree, Logistic Regression, and SVM), particularly excelling in handling nonlinear data and feature interactions.

```
Classification Result:
best model accuracy: 0.9516733115863235
best model precision: 0.9561342910460174
best model recall: 0.9516733115863235
best model f1: 0.9514655207144608
```

Figure 4: Best Model Test Results Chart

## PART II: Train MLP

### 2.1 Activation function

We first need to support ReLU activation function in the existing MLP (Multi Layer Perceptron) code and allow customization of activation types by layer. Firstly, we developed sigmoid and ReLU and their corresponding activation functions, where\_ The sigmoid and -relu functions respectively implement the core activation operation, while the \_sigmoid\_gradient and -relu\_gradient respectively implement their gradient calculation. In order to support customizing activation functions by layer, we need to extend these parts so that each layer can choose different activation functions. To allow each layer to use different activation functions, we can introduce a new parameter activation in the constructor of the MLP class. This parameter can be a string (used for all layers to use the same activation function) or a list (used for each layer to specify a different activation function). If the user does not specify, the same activation function is used by default.

```
def _sigmoid(self, z):
    """Compute logistic function (sigmoid)"""
    return expit(z)

def _relu(self, z):
    """ReLU activation function"""
    return np.maximum(0, z)

def _relu_derivative(self, z):
    """Derivative of ReLU function"""
    return (z > 0).astype(float)

def _sigmoid_derivative(self, z):
    """Compute gradient of the logistic function"""
    sg = self._sigmoid(z)
    return sg * (1.0 - sg)

def _activation_function(self, z):
    """Apply the activation function for a given layer."""
    if self.activation == 'relu':
        return self._relu(z)
    elif self.activation == 'sigmoid':
        return self._sigmoid(z)
    else:
```

```

        raise ValueError("Unsupported activation function")

    def _activation_derivative(self, z):
        """Return the derivative of the activation function."""
        if self.activation == 'relu':
            return self._relu_derivative(z)
        elif self.activation == 'sigmoid':
            return self._sigmoid_derivative(z)
        else:
            raise ValueError("Unsupported activation function")

```

## 2.2 Incremental Learning

The key to implementing the partial-fit method is to be able to continue training on the basis of the already trained model, rather than reinitializing the weights. Firstly, the partial-fit method receives new training data and an increased number of training epochs, and uses the weights of the current model for training. To achieve this, it is necessary to ensure that the weights and biases of the model are retained after each training session, rather than being reinitialized.

When implementing, the first step is to check if the model has been trained. If not, initialize the weights of the model. Then, the training data is divided into small batches through batch processing, and forward and backward propagation are performed on each small batch to calculate the loss and update weights. Every time `partial_fit` is called, the weights of the existing model will continue to be trained, and the loss during the training process can be output after each epoch.

From a code perspective, we refer to the original fit method as A and the incremental learning method as B. Therefore, A has the following differences compared to B:

- ① The cost of incremental learning will not be cleared, while the cost of the original method will be affected by each epoch.

```

A: self.cost_ = []
B: none

```

- ② The learning rate of incremental learning B decays by inheriting the original epoch, while the epoch of the original A method does not decay, But in the calculation process, it only needs to be calculated directly using epoch.

```

A: (1 + self.decrease_const * i)
B: (1 + self.decrease_const * i+epoch)

```

- ③ When executing the loop, incremental learning B directly calls the given epoch parameter, while raw learning A needs to call `self.epoch`

```

A: for i in range(self.epochs):
B: for epoch in range(epochs):

```

## 2.3 Multi-layer MLP

From a code perspective, the main differences between a single-layer perceptron (single-layer MLP) and a multi-layer perceptron (multi-layer MLP) lie in the complexity of the network structure and the hierarchical nature of the computation process. A single-layer MLP consists only of an input layer and an output layer, requiring only one set of weights and biases to be defined in the code. The calculations for forward propagation and backpropagation are relatively simple, involving just one linear transformation and gradient update. In contrast, a multi-layer MLP includes multiple hidden layers, necessitating dynamic management of weights and biases for each layer. Forward propagation requires layer-by-layer computation of linear transformations and activation functions, while backpropagation starts from the output layer, computing gradients and propagating errors layer by layer, ultimately updating the parameters for each layer. When writing code, the primary focus is on managing and updating the parameters for each layer.

At the level of code writing, there are four differences between single-layer MLP and multi-layer MLP. Our single-layer MLP is labeled as A, and the multi-layer MLP is labeled as B.

- ① B stores the weight matrix of each layer through `self.weights`, while A only has `w1` and `w2`, which is the difference in initialization.

```
A: self.w1, self.w2 = self._initialize_weights()
B: self.weights = self._initialize_weights()
```

During the initialization process, the weight of each layer in B needs to be updated

```
A:
w1 = np.random.uniform(-1.0, 1.0, size=self.n_hidden *
    (self.n_features + 1))
w2 = np.random.uniform(-1.0, 1.0, size=self.n_output *
    (self.n_hidden + 1))
B:
for _ in range(self.n_hidden_layers):
    weights = np.random.uniform(-1.0, 1.0, size=(self.n_hidden,
    prev_layer_size + 1))
    all_weights.append(weights)
```

- ② In the process of forward propagation, A only needs to perform one linear transformation and activation function calculation, while B needs to calculate the linear transformation and activation function layer by layer, with the output of each layer serving as the input of the next layer.

```
A:
a1 = self._add_bias_unit(X, how='column')
z2 = w1.dot(a1.T)
a2 = self._sigmoid(z2)
```



```

z3 = w2.dot(a2)
a3 = self._sigmoid(z3)
B:
for i in range(self.n_hidden_layers):
    z = self.weights[i].dot(a.T)
    a = self._activation_function(z)
    a = self._add_bias_unit(a, how='row')

```

- ③ In the process of backpropagation, A only needs to calculate the gradients of the output layer and the hidden layer., And B needs to calculate the linear transformation and activation function layer by layer, with the output of each layer serving as the input for the next layer. Starting from the output layer, calculate the gradient of each layer layer layer by layer and pass the error back to the previous layer.

```

A:
sigma3 = a3 - y_enc
sigma2 = w2.T.dot(sigma3) * self._sigmoid_gradient(z2)
grad1 = sigma2.dot(a1)
grad2 = sigma3.dot(a2.T)
B:
for i in range(self.n_hidden_layers, 0, -1):
    sigma_hidden = self.weights[i].T.dot(sigma[-1]) *
self._activation_derivative(activations[i])
    sigma.append(sigma_hidden)

```

- ④ During the parameter update process, A only updates two sets of weights, while B needs to update the weights of each layer. The number of weight matrices is determined by the number of hidden layers.

```

A:
self.w1 -= (delta_w1 + (self.alpha * delta_w1_prev))
self.w2 -= (delta_w2 + (self.alpha * delta_w2_prev))
B:
for i in range(self.n_hidden_layers + 1):
    self.weights[i] -= (delta_weights + (self.alpha *
delta_weights_prev[i]))

```

## 2.4 Model Training Results

Finally, we trained our multi-layer MLP on the RFID dataset. As shown in Figures 5, the overall performance of the MLP model is promising. However, due to limitations in computational resources, we were unable to construct a larger or deeper MLP network. For this dataset, deeper networks are essential. Our MLP model achieved impressive results, with a training accuracy of 66.25% and a testing accuracy of 62.52% after 300 epochs. When trained for 500 epochs, the model further improved, reaching a training accuracy of 67.95% and a testing accuracy of 64.18%. These results demonstrate the effectiveness of our approach, even within the constraints of available computational resources.

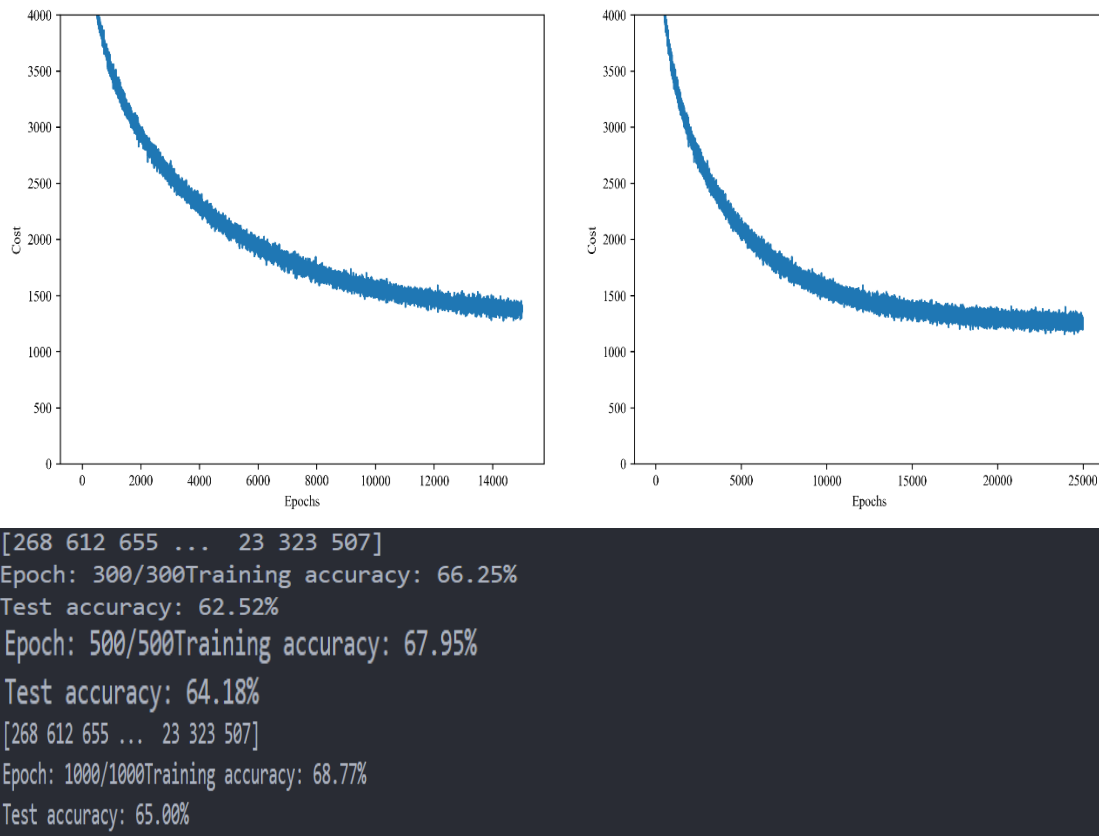


Figure 5: Training Results (300|500|1000 epochs)

## PART III: SUMMARY

**Q1:** Which algorithms from Scikit-Learn worked best on the RFID dataset and what accuracy did you get?

**A1:** The best-performing algorithm on the RFID dataset is the Random Forest, with an accuracy of 0.9482, precision of 0.9545, recall of 0.9482, and an F1 score of 0.9481. These results indicate that Random Forest demonstrates high classification performance on this dataset, significantly outperforming other algorithms.

**Q2:** Does it work better or worse with feature selection? Which method was the best?

**A2:** From the perspective of Random Forest, it appears to perform better after feature extraction, with an improvement of approximately 10%. However, when tested with PCA, Polynomial (Poly), and LDA methods, the model achieved an accuracy of 99% under the Polynomial feature selection. Despite this, I suspect that the model carries a significant risk of overfitting. Taking this into consideration, I believe that the PCA algorithm performs better in handling feature selection, which may be related to the dataset's high dimensionality of 188 features.

**Q3:** Were you able to train our own multilayer perceptron on the RFID dataset? Did it perform the same as the version in Scikit-Learn?

**A3:** Yes, I trained our own multilayer perceptron (MLP) on the RFID dataset. Although it occasionally encountered minor bugs and was limited by computational resources, preventing it from being fully optimized, its best performance on the test set reached 65%. In comparison, the MLP implementation in Scikit-Learn achieved an accuracy of 84%, indicating a significant performance gap. This discrepancy is likely due to the advanced iterative optimization and training algorithms utilized in the Scikit-Learn library, which I have not yet been able to replicate at the same level.