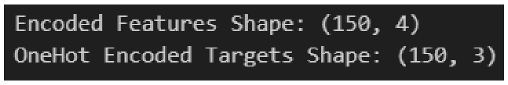
EEE8501.01-00: SPECIAL TOPICS IN PATTERN RECOGNITION
Project 2

Hari Kang

Question 1: Describe briefly the content of your data sets and provide a summary table for these data

Part (a): IRIS

The Iris dataset is used for classifying three different types of Iris flowers (Iris-setosa, Iris-versicolor, and Iris-virginica) based on the length and width of their petals and sepals (4 features). It contains a total of 150 samples, with 4 features and 3 classes.

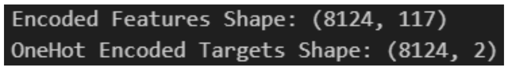


```
Encoded Features Shape: (150, 4)
OneHot Encoded Targets Shape: (150, 3)
```

Figure 1: Shape of IRIS Data

Part (b): Mushroom

The Mushroom dataset is used for classifying mushrooms as poisonous or edible based on their features. It contains a total of 8,124 samples, with 22 features and 2 classes: poisonous and edible.

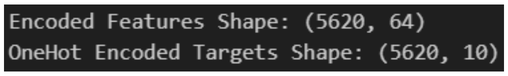


```
Encoded Features Shape: (8124, 117)
OneHot Encoded Targets Shape: (8124, 2)
```

Figure 2: Shape of MUSHROOM Data

Part (c): Optical recognition of handwritten digits

This dataset is used for classifying handwritten digits from 0 to 9 based on 64 features. It consists of a total of 5,620 samples, each with 64 features. There are 10 classes representing the digits 0 to 9.



```
Encoded Features Shape: (5620, 64)
OneHot Encoded Targets Shape: (5620, 10)
```

Figure 3: Shape of DIGITS Data

Dataset	# of samples	# of features	# of classes
IRIS	150	4	3
Mushroom	8124	22	2
Optical recognition of handwritten digits	5620	64	10

Table 1: Features of each Datasaet

Question 2: Provide details in terms of the setting and considerations for each algorithm.

Part (a): 3-layer MLP at different hidden node sizes

In this part, we train a 3-layer Multi-Layer Perceptron (MLP) with various hidden node sizes. The hidden node sizes tested are 128, 64, and 32. The ReLU activation function is used for non-linear transformation, with a learning rate of 0.001 and training epochs set to 100. The different hidden node sizes allow us to observe how the complexity of the model affects performance. The learning rate and number of epochs are chosen to balance between sufficient training and avoiding overfitting.

Settings	Value and Considerations
Different hidden node sizes	128,64,32
Activation function	ReLU
Learning Rate	1e-3
Epochs	100

Table 2: The setting and considerations for algorithm

Part (b): SVM using different kernels

This part involves training Support Vector Machines (SVM) using different kernel functions: Linear, Polynomial, and RBF (Radial Basis Function). The regularization parameter C is set to 1. For the polynomial kernel, the orders tested are 2nd and 3rd. For the RBF kernel, different γ values of 0.01, 0.1, and 1 are considered. Using different kernels allows us to explore how the kernel choice affects the SVM's performance on the given datasets. The regularization parameter C controls the trade-off between achieving a low training error and a low testing error, while the parameters for polynomial and RBF kernels adjust the kernel's complexity and sensitivity.

Settings	Value and Considerations
Kernel	Linear, Polynomial, RBF
Regularization parameter C	1
Polynomial order of Polynomial Kernel	2nd, 3rd
γ of RBF Kernel	0.01, 0.1, 1

Table 3: The setting and considerations for algorithm

Part (c): RM Model for orders 1 to 5

In this part, we use the RM (Reduced Multivariate Polynomial) model to perform polynomial regression with polynomial orders ranging from 1 to 5. Testing polynomial orders from 1 to 5 allows us to see how increasing the complexity of the model (higher-order polynomials) affects its performance. Higher-order polynomials can capture more complex patterns but also run the risk of overfitting the training data.

Settings	Value and Considerations
RM Model for orders	1 to 5

Table 4: The setting and considerations for algorithm

Question 3: Define each method : OneHot and Cross-Validation.

The OneHot function is designed to convert a given target vector Y_in into a one-hot encoded matrix Y_out for K-category problems (K > 2). The function uses the OneHotEncoder and LabelEncoder from the sklearn.preprocessing module to handle different types of input data, including pandas DataFrames and Series. The function ensures that the input vector is reshaped correctly before applying the encoding. This is useful for transforming categorical class labels into a binary matrix representation where each column corresponds to a category.

```python
from sklearn.preprocessing import OneHotEncoder, LabelEncoder
def OneHot(y_in, K):
    encoder = OneHotEncoder(categories='auto', sparse_output=False)
    y = y_in
    # pandas DataFrame turns to one hot vector
    if isinstance(y, pd.DataFrame) or isinstance(y, pd.Series):
        y = y.to_numpy()
        # OneHotEncoding

        y = y.reshape(-1, 1)
        y_out = encoder.fit_transform(y)
    # if label is string, turns it to onehot vector
    elif isinstance(y, pd.DataFrame) and all(y.dtypes == object):
        label_encoder = LabelEncoder()
        y = label_encoder.fit_transform(y)
        # OneHotEncoding
        labels_reshaped = np.array(y).reshape(-1, 1)
        y_out = encoder.fit_transform(labels_reshaped)
    #Other format turns to one hot vector
    else:
        # OneHotEncoding
        y = y.reshape(-1, 1)
        y_out = encoder.fit_transform(y)

    return y_out
```

Listing 1: OneHot in Python

The cross_validate function performs k-fold cross-validation on a given model with the input data X and target labels y. It uses the KFold class from the sklearn.model_selection module to split the data into k folds. For each fold, the model is trained on the training set and evaluated on the test set. The function collects the accuracy scores for each fold and returns the average accuracy. This method is useful for assessing the performance of a model by providing a more reliable estimate of its accuracy, as it evaluates the model on multiple subsets of the data.

```python
from sklearn.model_selection import KFold
def cross_validate(model, X, y, folds=5):
    kf = KFold(n_splits=folds)
    results = []
    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]
        model.fit(X_train, y_train)
        val = model.score(X_test, y_test) * 100
        results.append(val.round(2))
    print(results)
    ans = np.mean(results)
    return ans.round(2)
```

Listing 2: Cross-validation with KFold in Python

Question 4: Plot your average training and testing results for orders 1 to 5 for the RM model

Part (a): IRIS | Plot of average and each training and testing accuracy for orders 1 to 5

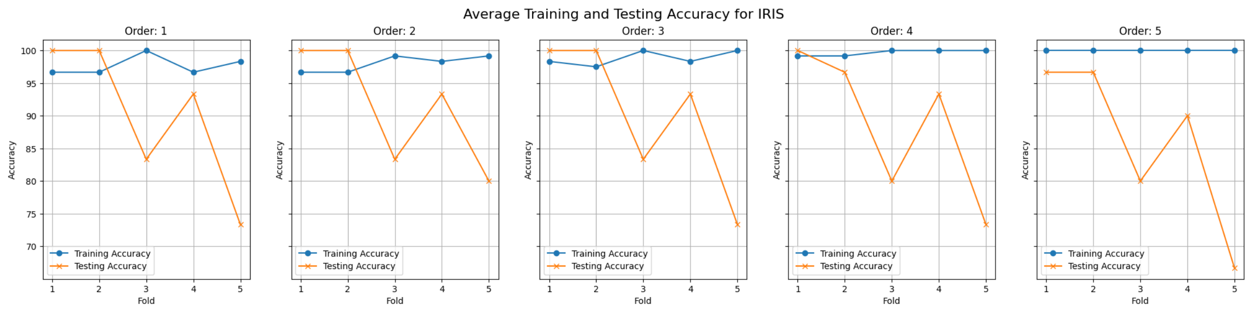


Figure 4: Training and Testing Accuracy of IRIS Data

As you can see in Fig 4, Each order of RM shows different results.

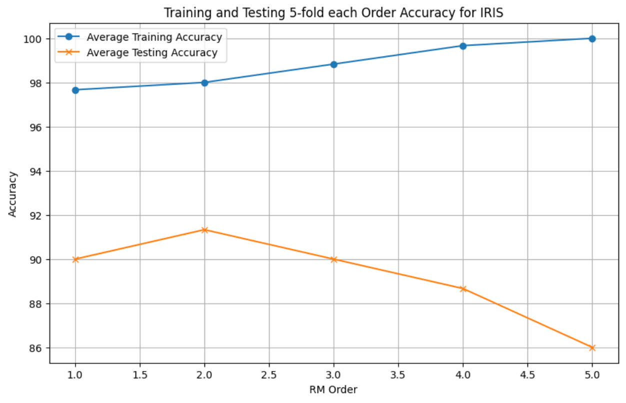


Figure 5: Average Training and Testing Accuracy of IRIS Data

Fig 5 shows the average training and testing accuracy for 5 different RM orders, each averaged over 5 folds.

Part (b): Mushroom | Plot of average and each testing results for orders 1 to 5

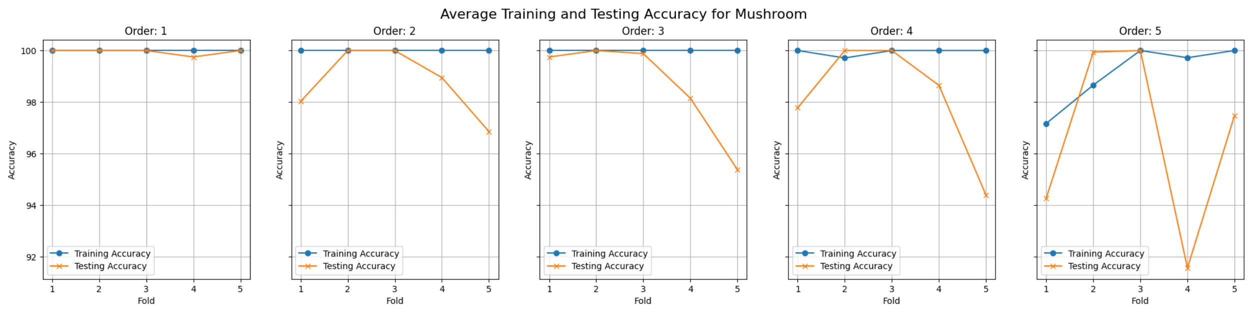


Figure 6: Training and Testing Accuracy of Mushroom Data

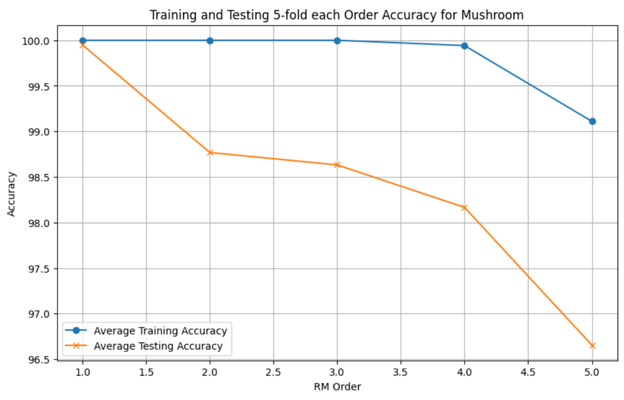


Figure 7: Average Training and Testing Accuracy of Mushroom Data

As you can see in Fig 6, Each order of RM shows different results.
Fig 7 shows the average training and testing accuracy for 5 different RM orders, each averaged over 5 folds.

Part (c): Digits | Plot of average and each testing results for orders 1 to 5

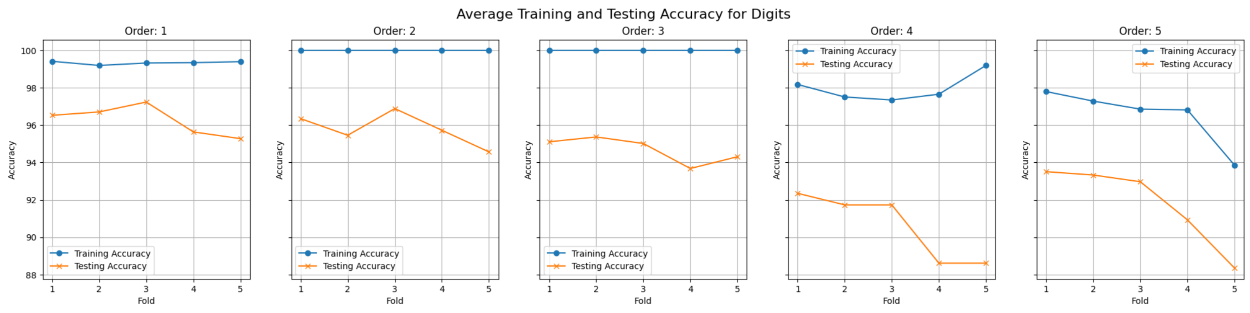


Figure 8: Training and Testing Accuracy of Digits Data

As you can see in Fig 8, Each order of RM shows different results.

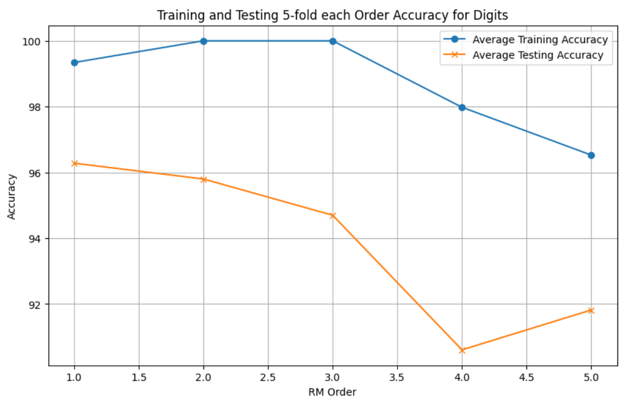


Figure 9: Average Training and Testing Accuracy of Digits Data

Fig 9 shows the average training and testing accuracy for 5 different RM orders, each averaged over 5 folds.

Question 5: Tabulate and compare all results of MLP, SVM and RM model

Part (a): Tabulate Results of MLP

Hidden Size	1 Fold	2 Fold	3 Fold	4 Fold	5 Fold	Mean accuracy
128	100	96.7	73.3	93.33	86.67	90.00
64	100	80.0	16.67	90.0	80.0	73.33
32	100	76.67	16.67	86.67	63.33	68.67

Table 5: **IRIS**: Cross-validation scores for different hidden layer sizes

Hidden Size	1 Fold	2 Fold	3 Fold	4 Fold	5 Fold	Mean accuracy
128	98.03	100.00	100.00	99.82	98.77	99.33
64	98.22	100.00	100.00	99.02	98.28	99.10
32	100.00	100.00	100.00	99.82	97.35	99.43

Table 6: **Mushroom**: Cross-validation scores for different hidden layer sizes

Hidden Size	1 Fold	2 Fold	3 Fold	4 Fold	5 Fold	Mean accuracy
128	97.33	97.42	97.07	96.00	95.47	96.65
64	96.53	97.51	97.33	95.73	95.37	96.49
32	96.80	96.26	96.35	95.11	94.58	95.82

Table 7: **DIGITS**: Cross-validation scores for different hidden layer sizes

Part (b): Tabulate Results of SVM model

Kernel	Hyper-parameter	1 fold	2 fold	3 fold	4 fold	5 fold	Mean Accuracy
linear		100.0	100.0	83.33	100.0	86.67	**94.00**
polynomial	order = 2	50.0	70.0	83.33	70.0	40.0	62.67
	order = 3	100.0	100.0	83.33	76.67	63.33	**84.67**
RBF	gamma = 0.01	100.0	83.33	0.0	80.0	0.0	52.67
	gamma = 0.1	100.0	100.0	80.0	93.33	76.67	**90.00**
	gamma = 1	100.0	96.67	76.67	93.33	83.33	**90.00**

Table 8: **IRIS**: Cross-validation scores for different Kernels

Kernel	Hyper-parameter	1 fold	2 fold	3 fold	4 fold	5 fold	Mean Accuracy
linear		100.0	100.0	100.0	100.0	100.0	**100.00**
polynomial	order = 2	100.0	100.0	100.0	**100.0**	99.14	99.83
	order = 3	100.0	100.0	100.0	98.65	99.51	99.63
RBF	gamma = 0.01	100.0	100.0	100.0	98.09	98.4	**99.30**
	gamma = 0.1	100.0	100.0	99.94	86.58	97.6	96.82
	gamma = 1	11.88	11.38	34.03	16.98	31.22	21.10

Table 9: **Mushroom**: Cross-validation scores for different Kernels

Kernel	hyper-parameter	1 fold	2 fold	3 fold	4 fold	5 fold	Mean Accuracy
linear		98.04	97.24	98.04	96.26	96.53	**97.22**
polynomial	order = 2	98.31	98.67	98.31	96.98	97.95	**98.04**
	order = 3	97.86	98.31	97.69	95.82	97.42	97.42
RBF	gamma = 0.01	98.22	98.13	98.49	97.24	97.15	**97.85**
	gamma = 0.1	96.53	95.20	95.55	93.95	92.88	94.82
	gamma = 1	10.23	11.30	12.54	13.88	13.79	12.35

Table 10: **DIGITS**: Cross-validation scores for different Kernels

RM Order	1 Fold	2 Fold	3 Fold	4 Fold	5 Fold	Mean accuracy
1	100.0	100.0	83.33	93.33	73.33	90.0
2	100.0	100.0	83.33	93.33	80.0	**91.33**
3	100.0	100.0	83.33	93.33	73.33	90.0
4	100.0	96.67	80.0	93.33	73.33	88.67
5	100.0	100.0	80.0	90.0	66.67	87.33

Table 11: **IRIS**: Cross-validation scores for different RM Orders

RM Order	1 Fold	2 Fold	3 Fold	4 Fold	5 Fold	Mean accuracy
1	100.0	100.0	100.0	99.75	100.0	**99.95**
2	98.03	100.0	100.0	98.95	96.86	98.77
3	99.02	100.0	100.0	98.71	96.98	98.94
4	97.78	100.0	100.0	98.65	97.48	98.78
5	94.28	99.94	100.0	91.57	98.28	96.81

Table 12: **Mushroom**: Cross-validation scores for different RM Orders

RM Order	1 Fold	2 Fold	3 Fold	4 Fold	5 Fold	Mean accuracy
1	96.53	96.71	97.24	95.64	95.37	**96.3**
2	95.37	94.48	94.93	93.77	92.44	94.2
3	94.84	94.93	95.11	94.13	92.97	94.4
4	67.97	69.04	62.37	60.23	66.64	65.25
5	91.1	90.84	89.5	92.08	88.88	90.48

Table 13: **DIGITS**: Cross-validation scores for different RM Orders

Part (c): Tabulate Results of RM model

Part (d): Compare all results of MLP, SVM and RM model

Let's compare by referring to the tables above.

- **Comparison with IRIS Dataset**
  - **For the IRIS Dataset, linear kernel SVM showed the highest performance with 94.00%.**
  - **MLP classifier** :The model with a hidden layer size of 128 achieves the highest mean accuracy of 90.00%, indicating that larger hidden layers significantly improve the model's performance on the IRIS dataset. As the hidden layer size decreases, the mean accuracy also decreases, with the smallest hidden layer size (32) achieving the lowest mean accuracy of 68.67%.
  - **SVM classifier** : The linear kernel achieves the highest mean accuracy of 94.00%, making it the most effective kernel for the IRIS dataset among the tested configurations. The polynomial kernel with order 3 and the RBF kernel with gamma values of 0.1 and 1 also perform well, with mean accuracies of 84.67% and 90.00%, respectively. The polynomial kernel with order 2 and the RBF kernel with gamma = 0.01 perform significantly worse.

  - **RM model** :RM Order 2 achieves the highest mean accuracy of 91.33%, indicating that this configuration is the most reliable for the IRIS dataset. Other RM orders also perform well, but their mean accuracies are slightly lower, suggesting that RM Order 2 provides a slight edge in performance.

- **Comparison with Mushroom Dataset**
  - **For the Mushroom Dataset, 1st order RM model showed the highest performance with 99.95%.**
  - **MLP classifier** : All hidden layer sizes perform exceptionally well with mean accuracies above 99%. The hidden layer size of 32 achieves the highest mean accuracy of 99.43%.
  - **SVM classifier** : The linear kernel outperforms all other kernels with a perfect mean accuracy of 100%. Polynomial and RBF kernels also show strong performances, especially with lower orders and gamma values respectively. The RBF kernel with gamma = 1 performs poorly compared to others.
  - **RM model** : RM Order 1 achieves the highest mean accuracy of 99.95%, indicating the highest reliability. Other RM orders also perform well, but their mean accuracies are slightly lower, suggesting that simpler RM orders might be more effective for this dataset.

- **Comparison with Digits Dataset**
  - **For the Digits Dataset, The polynomial kernel with order 2 showed the highest performance with 98.04%.**
  - **MLP classifier** : The model with a hidden layer size of 128 achieves the highest mean accuracy of 96.65%, suggesting that larger hidden layers provide a slight performance boost on the DIGITS dataset. The accuracy slightly decreases as the hidden layer size reduces, with the smallest hidden layer size (32) having a mean accuracy of 95.82%.
  - **SVM classifier** : The polynomial kernel with order 2 achieves the highest mean accuracy of 98.04%, making it the most effective kernel for the DIGITS dataset among the tested configurations. The linear kernel and the RBF kernel with gamma = 0.01 also perform well, with mean accuracies of 97.22% and 97.85%, respectively. The RBF kernel with gamma = 1 performs significantly worse, indicating that a high gamma value is not suitable for this dataset.
  - **RM model** : RM Order 1 achieves the highest mean accuracy of 96.3%, indicating that this configuration is the most reliable for the DIGITS dataset. Higher RM orders tend to perform worse, with RM Order 4 achieving the lowest mean accuracy of 65.25%.

Question 6: Provide a brief observation and comments on the results.

- *Brief Observation*
  - **IRIS Dataset**: The linear kernel SVM showed the highest performance, suggesting that the IRIS dataset may have a linear relationship that is well captured by the linear kernel.
  - **MUSHROOM Dataset**: For the Mushroom dataset, the 1st order RM model showed the highest performance, closely followed by the linear SVM kernel, indicating that the dataset can be effectively classified with simpler models.
  - **DIGITS Dataset**: The polynomial kernel with order 2 showed the highest performance for the Digits dataset, indicating that this dataset benefits from a higher-order polynomial relationship.

- *Comments on the results*
  - **Model Complexity**: For datasets like Mushroom, simpler models (linear SVM, RM Order 1) perform exceptionally well, suggesting that the data is linearly separable. In contrast, for datasets like Digits, more complex models (polynomial SVM) perform better, indicating non-linear relationships in the data.
  - **Overfitting**: Higher-order RM models tend to overfit, especially evident in the Digits dataset, where performance drops significantly for RM Order 4.
  - **Dataset Characteristics**: The effectiveness of different models and configurations varies significantly with the dataset, emphasizing the importance of understanding dataset characteristics when choosing and tuning models.

Question 7: Include ALL other codes in the appendix for running by TA.

```
from ucimlrepo import fetch_ucirepo
from sklearn.datasets import fetch_openml
import numpy as np
from sklearn.preprocessing import OneHotEncoder, StandardScaler, LabelEncoder
from sklearn.neural_network import MLPClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
import matplotlib.pyplot as plt
from sklearn.model_selection import KFold
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
import pandas as pd
import warnings
from sklearn.exceptions import ConvergenceWarning
def OneHot(y_in, K ):
    encoder = OneHotEncoder(categories='auto', sparse_output=False)
    y = y_in
    # pandas DataFrame to  numpy array
    if isinstance(y, pd.DataFrame) or isinstance(y, pd.Series):
        y = y.to_numpy()
        # OneHotEncoding
        y = y.reshape(-1, 1)
        y_out = encoder.fit_transform(y)
    elif isinstance(y, pd.DataFrame) and all(y.dtypes == object):
        label_encoder = LabelEncoder()
        y = label_encoder.fit_transform(y)
        labels_reshaped = np.array(y).reshape(-1, 1)
        y_out = encoder.fit_transform(labels_reshaped)
    else:
        y = y.reshape(-1, 1)
        y_out = encoder.fit_transform(y)

    return y_out
# fetch dataset
iris = fetch_ucirepo(id=53)

# data (as pandas dataframes)
iris_X = iris.data.features
iris_y = iris.data.targets
# Normalize X
scaler = StandardScaler()
iris_X = scaler.fit_transform(iris_X)
label_encoder = LabelEncoder()
int_iris_y = label_encoder.fit_transform(iris_y)
onehot_iris_y = OneHot(iris_y,3)

#               (             )
print("Encoded␣Features␣Shape:", iris_X.shape)
print("Targets␣Shape:", int_iris_y.shape)
print("OneHot␣Encoded␣Targets␣Shape:", onehot_iris_y.shape)
def encode_categorical_features(df):
    categorical_cols = df.select_dtypes(include=['object']).columns
    encoder = OneHotEncoder(sparse_output=False)

    encoded_df = pd.DataFrame(encoder.fit_transform(df[categorical_cols]))
    encoded_df.columns = encoder.get_feature_names_out(categorical_cols)

    df = df.drop(categorical_cols, axis=1)
    df = pd.concat([df, encoded_df], axis=1)

    return df
# Fetch dataset
mushroom = fetch_ucirepo(id=73)
mushroom_X = mushroom.data.features
mushroom_y = mushroom.data.targets

mushroom_X = encode_categorical_features(mushroom_X)

# Normalize X
scaler = StandardScaler()
mushroom_X = scaler.fit_transform(mushroom_X)
label_encoder = LabelEncoder()
int_mushroom_y = label_encoder.fit_transform(mushroom_y)
onehot_mushroom_y = OneHot(mushroom_y,2)


#               (             )
print("Encoded␣Features␣Shape:", mushroom_X.shape)
print("Targets␣Shape:", int_mushroom_y.shape)
print("OneHot␣Encoded␣Targets␣Shape:", onehot_mushroom_y.shape)
# fetch dataset
optical_recognition_of_handwritten_digits = fetch_ucirepo(id=80)

# data (as pandas dataframes)
optical_recognition_of_handwritten_digits_X =
    optical_recognition_of_handwritten_digits.data.features
optical_recognition_of_handwritten_digits_y =
    optical_recognition_of_handwritten_digits.data.targets

optical_recognition_of_handwritten_digits_X = encode_categorical_features(
    optical_recognition_of_handwritten_digits_X)
# Normalize X
scaler = StandardScaler()
digits_X = scaler.fit_transform(optical_recognition_of_handwritten_digits_X)
int_digits_y = optical_recognition_of_handwritten_digits_y.to_numpy().reshape(-1)
onehot_digits_y = OneHot(optical_recognition_of_handwritten_digits_y,10)

#               (             )
print("Encoded␣Features␣Shape:", digits_X.shape)
print("Targets␣Shape:", int_digits_y.shape)
print("OneHot␣Encoded␣Targets␣Shape:", onehot_digits_y.shape)
from sklearn.model_selection import KFold
def cross_validate(model, X, y, folds=5):
    kf = KFold(n_splits=folds)
    results = []
    for train_index, test_index in kf.split(X):
        X_train, X_test = X[train_index], X[test_index]
        y_train, y_test = y[train_index], y[test_index]
        model.fit(X_train, y_train)
        val = model.score(X_test, y_test) *100
        results.append(val.round(2))
    print(results)
    ans = np.mean(results)
    return ans.round(2)
```

```
datasetlist=[["Digits",digits_X,onehot_digits_y],["Mushroom",mushroom_X,
    onehot_mushroom_y],["IRIS",iris_X,onehot_iris_y]]
# Example settings for 3-layer MLP
hidden_node_sizes = [128, 64, 32]
epochs = 100  # Increase the number of epochs

# Suppress convergence warnings
warnings.filterwarnings('ignore', category=ConvergenceWarning)

# Train and evaluate models
for set_name, X, y in datasetlist:
    print(set_name)
    for hidden_nodes in hidden_node_sizes:
        model = MLPClassifier(
            hidden_layer_sizes=(hidden_nodes, hidden_nodes),
            max_iter=epochs,
            activation='relu',
            learning_rate_init=1e-3,
            random_state=42
        )
        scores = cross_validate(model, X, y)

        # Print the mean accuracy and standard deviation
        print(f"Nodes:␣{hidden_nodes},␣Score:␣{scores:.2f}␣")
datasetlist=[["Digits",digits_X,int_digits_y],["Mushroom",mushroom_X,
    int_mushroom_y],["IRIS",iris_X,int_iris_y]]
# Example settings for SVM
kernels = ['linear', 'poly', 'rbf']
degrees = [2, 3]  # Only for polynomial kernel
gammas = [0.01, 0.1, 1]  # Only for RBF kernel
# Loop through all combinations (example)
for set_name, X, y in datasetlist:
    print(set_name)
    for kernel in kernels:
        if kernel == 'poly':
            for degree in degrees:
                model = SVC(kernel=kernel, C=1, degree=degree)
                score = cross_validate(model, X, y)
                print(f"Kernel:␣{kernel},␣Degree:␣{degree},␣Score:␣{score:.2f}")
        elif kernel == 'rbf':
            for gamma in gammas:
                model = SVC(kernel=kernel, C=1, gamma=gamma)
                score = cross_validate(model, X, y)
                print(f"Kernel:␣{kernel},␣Gamma:␣{gamma},␣Score:␣{score:.2f}")
        else:
            model = SVC(kernel=kernel, C=1)
            score = cross_validate(model, X, y)
            print(f"Kernel:␣{kernel},␣Score:␣{score:.2f}")
def RM(X, order):
    # Build regressor matrix P (mxK):
    # order = desired order of approximation,
    # X = input matrix (mxl), K = number of parameters to be est.
    # m = number of data samples, l = input dimension.
    m, l = X.shape
    MM1 = []
    MM3 = []
    Msum = np.sum(X, axis=1)
    for i in range(1, order+1):
        M1 = np.zeros((m, l))
        M3 = np.zeros((m, l))
        for k in range(l):
            M1[:, k] = X[:, k]**i
            if i > 1:
                M3[:, k] = X[:, k] * Msum**(i-1)
        MM1.append(M1)
        if i > 1:
            MM3.append(M3)
    if MM3:
        P = np.concatenate([np.ones((m, 1)), np.concatenate(MM1, axis=1), np.
           concatenate(MM3, axis=1)], axis=1)
    else:
        P = np.concatenate([np.ones((m, 1)), np.concatenate(MM1, axis=1)], axis=1)
    return P

X = np.array([[1, 2], [3, 4], [5, 6]])
order = 2
P = RM(X, order)
# Example settings for RM model
orders = [1, 2, 3, 4, 5]

# Loop through all orders (example)
for set_name, X, y in datasetlist:
    print(set_name)
    for order in orders:
        P = RM(X, order)
        # Perform linear regression
        model = LogisticRegression()
        score = cross_validate(model, P, y)
        print(f"Order:␣{order},␣Score:␣{score}")
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures, StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import train_test_split, cross_val_score, KFold
datasetlist=[["Digits",digits_X,int_digits_y],["Mushroom",mushroom_X,
    int_mushroom_y],["IRIS",iris_X,int_iris_y]]

# Define a function to plot the average training results
def plot_avg_training_results(X, y, dataset_name):
    orders = range(1, 6)
    order_result1, order_result2 = [],[]
    fig, axes = plt.subplots(1, 5, figsize=(25, 5), sharey=True)

    for idx, order in enumerate(orders):
        P = RM(X, order)
        model = LogisticRegression(max_iter=1000)
        kf = KFold(n_splits=5)
        train_results, test_results = [],[]

        for train_index, test_index in kf.split(P):
            X_train, X_test = P[train_index], P[test_index]
            y_train, y_test = y[train_index], y[test_index]
            model.fit(X_train, y_train)
            train = model.score(X_train,y_train) *100
            test = model.score(X_test, y_test) *100
            train_results.append(train.round(2))
            test_results.append(test.round(2))

        train_ans = np.mean(train_results)
        test_ans = np.mean(test_results)
        order_result1.append(train_ans)
        order_result2.append(test_ans)

        axes[idx].plot(range(1, len(train_results) + 1), train_results, marker='o', label='Training␣Accuracy')
        axes[idx].plot(range(1, len(test_results) + 1), test_results, marker='x', label='Testing␣Accuracy')
        axes[idx].set_title(f'Order:␣{order}')
        axes[idx].set_xlabel('Fold')
        axes[idx].set_ylabel('Accuracy')
        axes[idx].legend()
        axes[idx].grid(True)

    fig.suptitle(f'Average␣Training␣and␣Testing␣Accuracy␣for␣{dataset_name}', fontsize=16)
    plt.show()

    plt.figure(figsize=(10, 6))
    plt.plot(orders, order_result1, marker='o', label='Average␣Training␣Accuracy')
    plt.plot(orders, order_result2, marker='x', label='Average␣Testing␣Accuracy')
    plt.title(f'Training␣and␣Testing␣5-fold␣each␣Order␣Accuracy␣for␣{dataset_name}')
    plt.xlabel('RM␣Order')
    plt.ylabel('Accuracy')
    plt.legend()
    plt.grid(True)
    plt.show()

# Plot average training results for each dataset
for set_name, X, y in datasetlist:
    print(set_name)
    plot_avg_training_results(X, y, set_name)
```

Listing 3: ALL other codes