Assignment 1

Dimensionality Reduction using PCA

COMP257(SEC. 003)

Yoo Sun Song
301091906

Questions

Question 1 [50 points]

Read the questions below carefully.

Retrieve and load the mnist_784 dataset of 70,000 instances.

```
from sklearn.datasets import fetch_openml

mnist = fetch_openml('mnist_784', version=1, as_frame=False, parser='auto')

print(mnist.data.shape)
```

Code Analysis

Load the MNIST dataset using the Scikit-learn library.

fetch the MNIST dataset with 784 features (28x28 pixels flattened to a 1D array).

The output indicates that the MNIST dataset contains 70,000 instances, each with 784 functions.

```
   ...: print(mnist.data.shape)
(70000, 784)
```

Display each digit.

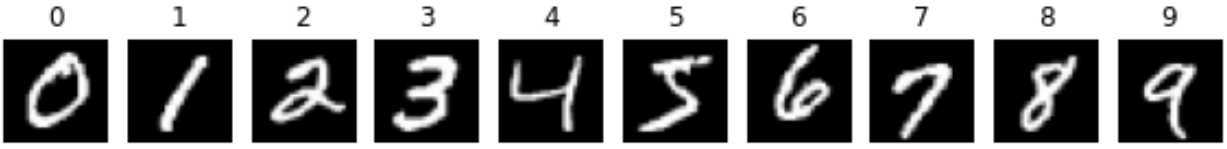
```
import matplotlib.pyplot as plt
import numpy as np
unique_labels = np.unique(y_train)
fig, axes = plt.subplots(1, 10, figsize=(10, 1))

for ax, label in zip(axes, unique_labels):
    img_data = X_train[y_train == label][0].reshape(28, 28)
    ax.imshow(img_data, cmap="gray")
    ax.axis("off")
    ax.set_title(label)
plt.show()
```

Code Analysis

1. Import Library: Import Matplotlib for plotting and NumPy for numerical operations.
2. Unique label: Identify the unique label (0-9) in the training dataset y_train.
3. Subplot: Create a 1x10 grid of subplot with size 10x1.
4. Loop: For each unique label:
    a. Find the first instance of the X_train that has that label.
    b. Reconfigure the 784-dimensional array into a 28x28 2D array.
    c. Use Matplotlib to grayscale the image.
5. Show: Displays all sub-plots.

This code essentially visualizes the first example of each unique number (0-9) in the MNIST dataset as a one-line subplot.

Each Digit



Use PCA to retrieve the 1^{th} and 2^{th}principal component and output their *explained variance ratio*.

```
from sklearn.decomposition import PCA

n_components_2 = 2

pca_2 = PCA(n_components= n_components_2)

X_pca_2 = pca_2.fit_transform(X_train)

explained_var_ratio_2 = pca_2.explained_variance_ratio_

explained_var_ratio_sum_2 = pca_2.explained_variance_ratio_.sum()

print(f"Explained Variance Ratio for 1st and 2nd components: {explained_var_ratio_2}")

print(f"Sum of Explained Variance Ratios for 1st and 2nd components: {explained_var_ratio_sum_2}")
```

Code Analysis

1. Import the PCA class from the sklearn.decomposition package.
2. Create a PCA object with **n_components=2** to keep only the first and second principal components.
3. Apply PCA to **X_train** to get the principal components, stored in **X_pca_2**
4. Calculate the explained variance ratios and their sum.

Output Explanation

1. Explained Variance Ratio for 1st and 2nd components: **[0.09704664 0.07095924].**
   The 1st principal component explains about **9.705%** of the total variance.
   The 2nd principal component explains about **7.096%** of the total variance.
2. Sum of Explained Variance Ratios for 1st and 2nd components: **0.1680058837091072**.
   Together, the 1st and 2nd principal components account for approximately **16.801**% of the total variance in the dataset.

Plot the projections of the 1^{th} and 2^{th} principal component onto a 1D hyperplane.

```
plt.figure(figsize=(12, 6))


plt.subplot(1, 2, 1)
for i in range(10):
    plt.scatter(X_pca_2[y_train == str(i), 0], [0]*len(X_pca_2[y_train == str(i), 0]), alpha=0.6, label=str(i))
plt.xlabel("1st Principal Component")
plt.title("Projection of 1st Principal Component onto 1D hyperplane")
plt.legend()


# Second subplot for the 2nd Principal Component
plt.subplot(1, 2, 2)
for i in range(10):
    plt.scatter([0]*len(X_pca_2[y_train == str(i), 1]), X_pca_2[y_train == str(i), 1], alpha=0.6, label=str(i))
plt.ylabel("2nd Principal Component")
plt.title("Projection of 2nd Principal Component onto 1D hyperplane")
plt.legend()


plt.tight_layout()
plt.show()
```

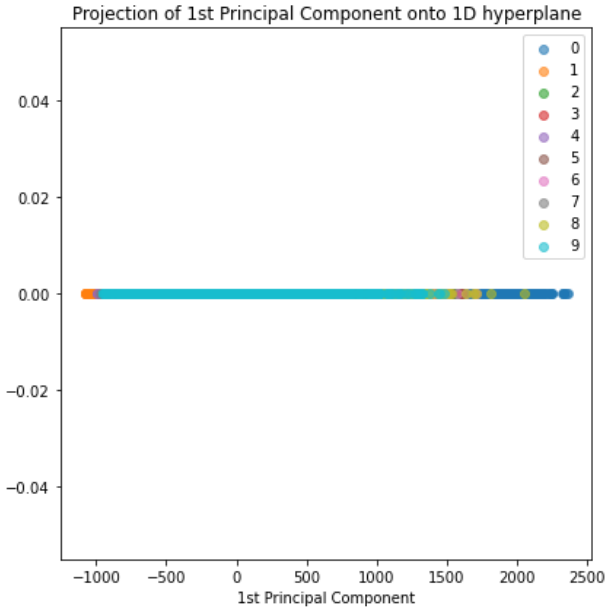
Code Analysis

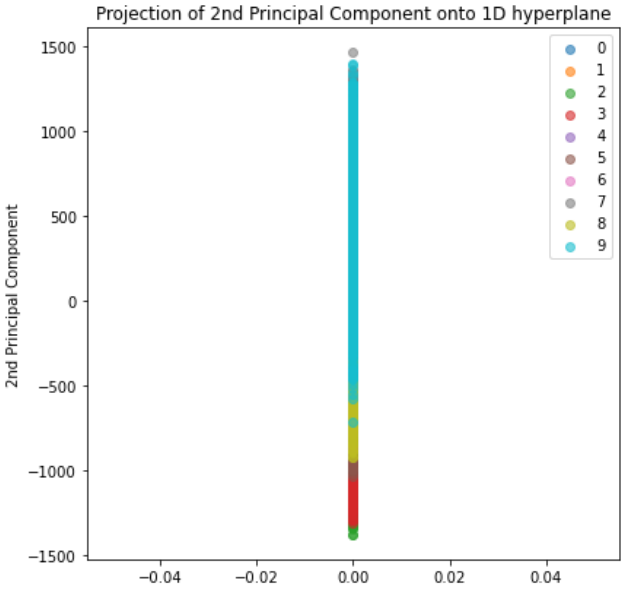
The first Subplot for the 1st Principal Component:

- plt.Subplot (1, 2, 1) sets the first subplot.
- The loop repeats each unique numeric label (0 through 9).
- The scatter function displays the 1st principal component of each numeric label along the X-axis and keeps the Y-axis to zero.

The second Subplot of the 2nd Principal Component:

- plt.Subplot (1, 2, 2) sets the second subplot.
- Another loop repeats each unique numeric label (0 to 9).
- The scatter function displays the second principal component of each digit label along the Y-axis while keeping the X-axis to zero.


Projection of 1st Principal Component onto 1D hyperplane
0.04
0.02
0.00
−0.02
−0.04
−1000
−500
0
500
1000
1500
2000
2500
1st Principal Component
0
1
2
3
4
5
6
7
8
9


Projection of 2nd Principal Component onto 1D hyperplane
1500
1000
500
0
−500
−1000
−1500
2nd Principal Component
−0.04
−0.02
0.00
0.02
0.04
0
1
2
3
4
5
6
7
8
9

Use Incremental PCA to reduce the dimensionality of the MNIST dataset down to 154 dimensions.

```
from sklearn.decomposition import IncrementalPCA

n_batches = 100

n_components_154 = 154

incremental_pca = IncrementalPCA(n_components = n_components_154)

for batch_x in np.array_split(X_train, n_batches):

    incremental_pca.partial_fit(batch_x)

X_reduced_incremental = incremental_pca.transform(X_train)
```

Code Analysis

This Python code snippet reduces the dimension of the MNIST dataset from 784 to 154 using the Principal Component Analysis in the scikit-learn library.

1. Import the IncrementalPCA Class

   This PCA variant is useful when the data set is too large for memory. It updates PCA transformations gradually, making them suitable for large datasets.

2. Initialize Variables
   a. **n_batches**: Number of batches the dataset will be split into for incremental fitting.
   b. **n_components_154**: The target number of dimensions (154) after PCA

3. Initialize IncrementalPCA Object
4. Partial Fitting

   The MNIST training data (X_train) is split into 100 batches. Incremental PCA is then "partially fitted" to each batch.

5. Dimensionality Reduction

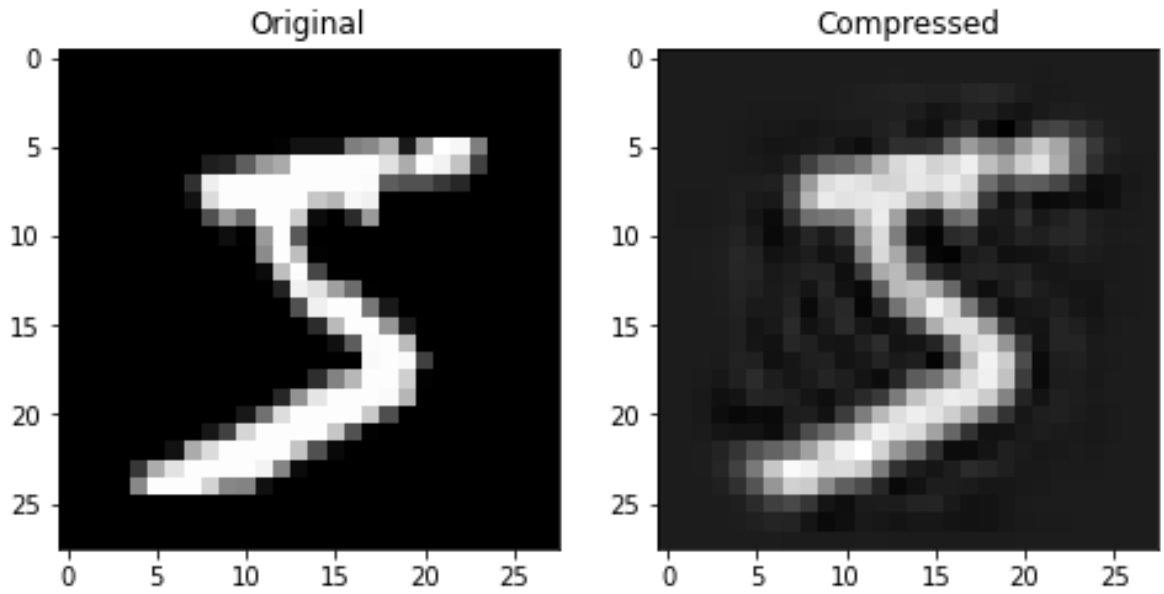
   the training data is transformed into a reduced 154-dimensional dataset (X_reduced_incremental).

   The MNIST dataset has been reduced to 154 dimensions. This is much lower than the original 784 dimensions, but maintains most of the variability of the data. This is useful for improving data visualization, storage efficiency, and potentially machine learning model performance.

Display the original and compressed digits from (5).

```
X_recovered_incremental = incremental_pca.inverse_transform(X_reduced_incremental)


index_to_show = 0


fig, axes = plt.subplots(1, 2, figsize=(8, 4))

axes[0].imshow(X_train[index_to_show].reshape(28, 28), cmap="gray")

axes[0].set_title("Original")

axes[1].imshow(X_recovered_incremental[index_to_show].reshape(28, 28), cmap="gray")

axes[1].set_title("Compressed")

plt.show()
```

Code Analysis



The original digit is clearer compared to the compressed one. This is a natural consequence of dimensionality reduction, where some level of detail is sacrificed for computational efficiency.

Clearly, the blurring observed in the compressed version of numbers is the result of information loss that occurs during dimension reduction. If you use incremental PCA to reduce data from 784 to 154 dimensions, you essentially use fewer principal components to approximate the original data.

This approximation captures the main characteristics of the data, but sacrifices some fine details, making the compressed image look blurry compared to the original. This is a balance between computational efficiency and data fidelity. Compressing data can make data processing, storage, and analysis faster and more efficient, but you have to bear quality losses.

Discuss challenges you confronted and solutions to overcoming them, if applicable.

In solving this question, I faced conceptual difficulties in visualizing and interpreting data in a 1D hyperplane. So, I tried to understand looking up various data and plotting data on the 1D hyperplane.

Question 2

Generate Swiss roll dataset.

```
from sklearn.datasets import make_swiss_roll


X, y = make_swiss_roll(n_samples=1000, noise=0.1, random_state=96)
```

Code Analysis

The code snippet uses Scikit-learn's make_**swiss_roll** function to create a Swiss roll dataset. **n_sample** is 1000, **noise** is 0.1 and **random_state** is 96. **X** contains the features, and **y** contains the labels, which can be used for various machine learning tasks. Also, **noise** parameters help simulate real data conditions, making the dataset better suited for robustness testing.

Plot the resulting generated Swiss roll dataset.

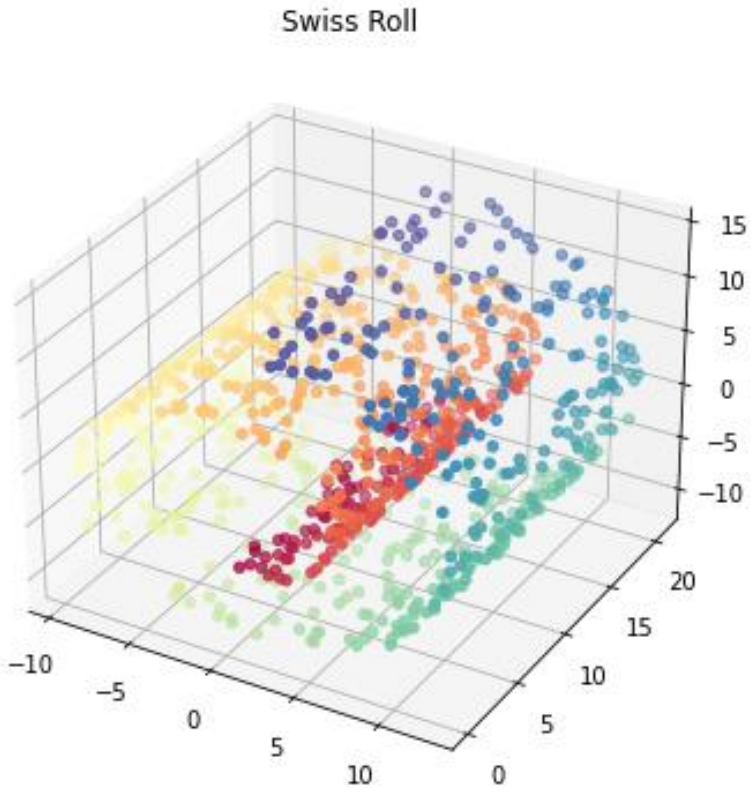
```
import matplotlib.pyplot as plt

fig = plt.figure(figsize=(8, 6))

ax = fig.add_subplot(111, projection='3d')

ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=y, cmap=plt.cm.Spectral)

plt.title('Swiss Roll')

plt.show()
```

Code Analysis

The code uses Matplotlib to visualize the 3D structure of the Swiss roll dataset.

Initialize the 8x6 inch Matplotlib Figure. Add a 3D subplot to the illustration. Then the notation '111' means the 1x1 grid and the first subplot of that grid. Use the functions stored in 'X' to create a 3D scatterplot and colour-coded the points based on the label 'y'. The color map used is 'Spectral'.

The resulting plot provides a visual validation of the Swiss roll structure to ensure that the dataset is generated as intended. Also, the 3D plot allows us to accurately represent Swiss rolls, making complexity easier to understand.



Use Kernel PCA (kPCA) with linear kernel, a RBF kernel, and a sigmoid kernel.

Plot the kPCA results of applying the linear kernel, a RBF kernel, and a sigmoid kernel from (3). Explain and compare the results.

```
from sklearn.decomposition import KernelPCA
```

What is Kernel PCA(kPCA)?

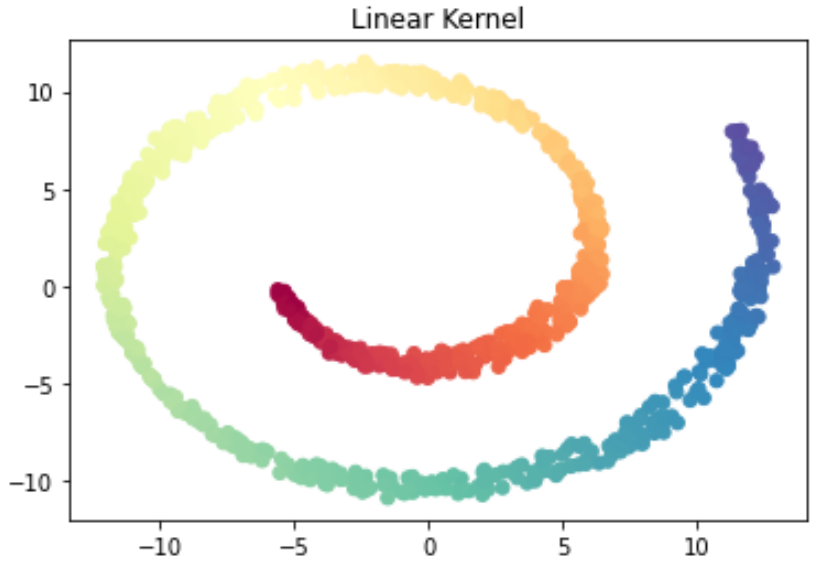
Kernel Principal Component Analysis (kPCA) is an extension of the Principal Component Analysis (PCA), which uses kernel functions that project data to a higher level before applying PCA.

Linear Kernel

```
kpca_linear_kernel = KernelPCA(kernel="linear",n_components =2)

X_kpca_linear_kernel = kpca_linear_kernel.fit_transform(X)

plt.scatter(X_kpca_linear_kernel[:, 0], X_kpca_linear_kernel[:, 1], c=y, cmap=plt.cm.Spectral)

plt.title('Linear Kernel')

plt.show()
```

Code Analysis

Applies a linear kernel, which essentially performs standard PCA. The points are mapped linearly. 'The points are mapped linearly' means that each point in the original space is transformed into a new point in the target space using a linear function. When the data is linearly separable or close to it, user the linear kernel.



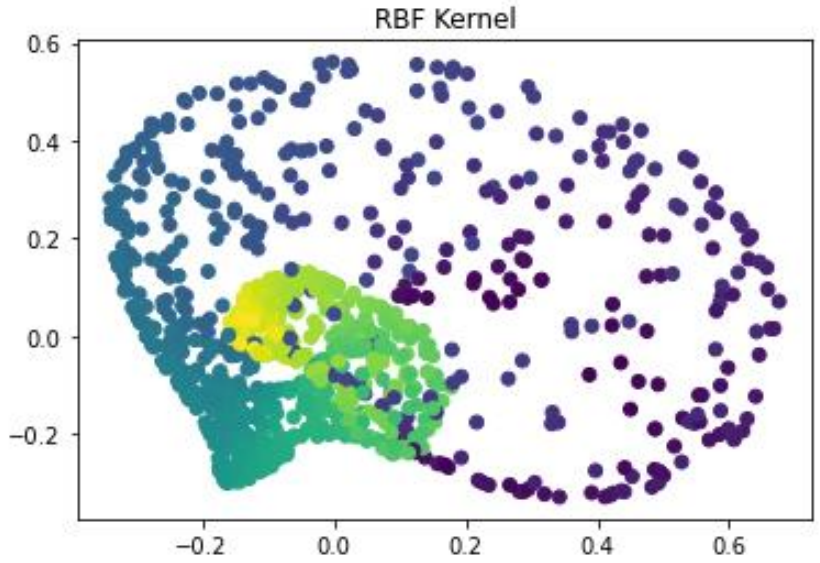
Because linear kernels project data linearly into a low-dimensional space, you'll notice that Swiss rolls can't be unfolded. In other words, linear methods are not best suited to capture the complexity of Swiss roll data.

RBF Kernel

```
kpca_rbf_kernel = KernelPCA(n_components=2, kernel='rbf',gamma=0.04)

X_kpca_rbf_kernel = kpca_rbf_kernel.fit_transform(X)

plt.scatter(X_kpca_rbf_kernel[:, 0], X_kpca_rbf_kernel[:, 1], c=y)

plt.title('RBF Kernel')

plt.show()
```

Code Analysis

Uses a Radial Basis Function (RBF) kernel, useful for non-linear separation. RBF provides a more flexible mapping, capturing complex relationships. When the data is non-linear, RBF is used. Gamma = 0.04 affects the shape of the RBF.



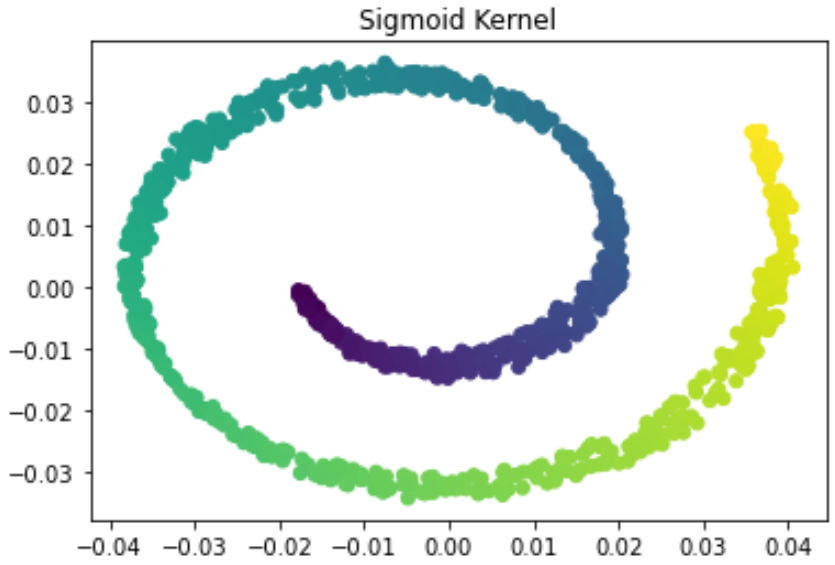
The RBF kernel can unroll Swiss rolls more successfully. Therefore the RBF kernel is effective for handling complex, non-linear data structures.

Sigmoid Kernel

```
kpca_sigmoid_kernel = KernelPCA(n_components=2, kernel='sigmoid', gamma=1e-05,coef0=0)

X_kpca_sigmoid_kernel = kpca_sigmoid_kernel.fit_transform(X)

plt.scatter(X_kpca_sigmoid_kernel[:, 0], X_kpca_sigmoid_kernel[:, 1], c=y)

plt.title('Sigmoid Kernel')

plt.show()
```

Code Analysis

Apply a sigmoid kernel that is frequently used in neural networks. The results are sensitive to the choice of gamma and coef0. Sigmoid kernels are less commonly used for kPCA but can be useful for certain types of data. Gamma=1e-05, coef0=0 are adjustable parameters.



Sigmoid kernels are more unpredictable and rely heavily on selection of parameters such as gamma and coef0. So the sigmoid kernel is not usually the first choice of kPCA due to its sensitivity to parameter tuning.

Using kPCA and a kernel of your choice, apply Logistic Regression for classification. Use *GridSearchCV* to find the best kernel and *gamma* value for kPCA in order to get the best classification accuracy at the end of the pipeline. Print out best parameters found by *GridSearchCV*.

```
import numpy as np

from sklearn.pipeline import Pipeline

from sklearn.linear_model import LogisticRegression

from sklearn.model_selection import GridSearchCV


y = np.where(y <= y.mean(), 0, 1)


clf = Pipeline([

   ('kpca', KernelPCA(n_components=2)),

   ('log_reg', LogisticRegression())

   ])


param_grid = {

   "kpca__gamma": np.linspace(0.1, 1, 10),

   "kpca__kernel": ["linear", "rbf", "sigmoid"],

   }


grid_search = GridSearchCV(clf, param_grid, scoring='accuracy', refit=True, cv=3)

grid_search.fit(X, y)


print("Best parameters: ", grid_search.best_params_)
```

Code Analysis

This code performs pipeline-based classification on the dataset. Specifically, use **Kernel Principal Component Analysis (kPCA)** for dimension reduction and **Logistic Regression** for classification. A grid of hyperparameters is defined for the kPCA. Gamma values range from 0.1 to 1 in 10 stages, and kernel types considered are "linear", "rbf", and "sigmoid". Grid search with 3-fold cross-validation optimizes accuracy by finding the optimal combination of **gamma** and **kernel** parameters for kPCA. Aslo, **y = np.where(y <= y.mean(), 0, 1)** is performing a binary transformation on the target array y.

This approach is useful for automating the hyperparametric tuning process and makes the code more maintenance and understandable. The final output provides the best hyperparameters that must be used for the kPCA to achieve the best classification accuracy.

```
    ...: print("Best parameters: ",
grid_search.best_params_)
Best parameters:  {'kpca__gamma': 0.1,
'kpca__kernel': 'rbf'}
```

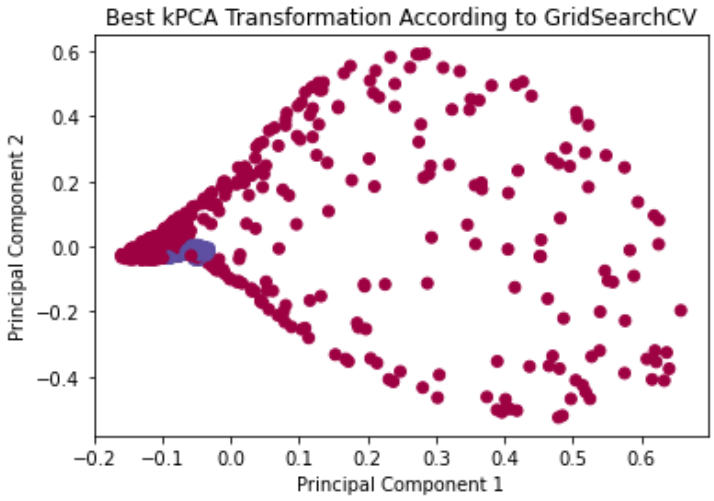
The results indicate that the best hyperparameters for kernel PCA in the pipeline are gamma values of 0.1 and **rbf** kernels.

Plot the results from using *GridSearchCV* in (5).

```
best_pipeline = grid_search.best_estimator_


X_best_transformed = best_pipeline.named_steps['kpca'].transform(X)


plt.scatter(X_best_transformed[:, 0], X_best_transformed[:, 1], c=y, cmap=plt.cm.Spectral)

plt.xlabel("Principal Component 1")

plt.ylabel("Principal Component 2")

plt.title("Best kPCA Transformation According to GridSearchCV")

plt.show()
```

Code Analysis

The code takes the best-performing kernel PCA model identified as GridSearchCV, applies it to the original dataset, and X_best_transformed the converted data. A scatterplot is then created to visualize this converted data in a two-dimensional space. The axes of the plot represent the first and second principal components, and the colors of the points represent their class labels. The purpose of the plot is to show how effectively the selected 'rbf' kernel and gamma value 0.1 were used to reduce the dimensions of the data while maintaining critical class separability in the next phase of data classification.



Discuss challenges you confronted and solutions to overcoming them, if applicable.

The most notable issue was resolving errors when using the sigmoid kernel in the kernel PCA. The error indicates that the kernel matrix may not be positive semi-formal (PSD), as indicated by the existence of significant negative eigenvalues.

This problem may result from certain hyperparameters gamma and coef0 used in the sigmoid kernel, resulting in a non-PSD kernel matrix. Hyperparameter tuning was done to resolve this issue. In other words, I experiment with various combinations of gamma and coef0 to see if the resulting kernel matrix is PSD.