```
###Dimensionally Reduction, Supervised Classification

Yusuf Abdi

Unsupervised Data Mining

06/23/2023 bold text
```

Dimensionally Reduction, Supervised Classification

Yusuf Abdi

Unsupervised Data Mining

06/23/2023 bold text

Double-click (or enter) to edit

```
# hide all warning messages
import warnings
warnings.filterwarnings('ignore')


# Importing required modules
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.pipeline import Pipeline
from mpl_toolkits.mplot3d import Axes3D
from sklearn.tree import DecisionTreeClassifier
from sklearn.decomposition import PCA, KernelPCA
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split as tts
from sklearn.datasets import fetch_openml, fetch_20newsgroups
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.metrics import accuracy_score, mean_squared_error
from sklearn.linear_model import LinearRegression, LogisticRegression
```

Data Loading

MNIST

```
# Load the MNIST dataset
mnist = fetch_openml('mnist_784', version=1)

# Separate the features (images) and labels (digits)
X_mnist = mnist['data']
y_mnist = mnist['target']
X_mnist.shape, y_mnist.shape
```

```
((70000, 784), (70000,))
```

```
# Split the data into train and test sets
train_samples = 60000  # Number of samples for training
X_mnist_train, y_mnist_train = X_mnist[:train_samples], y_mnist[:train_samples]  # First train_samples samples for training
X_mnist_test, y_mnist_test = X_mnist[train_samples:], y_mnist[train_samples:]  # Remaining samples for testing

# Print the shapes of the sets
print("Training set - Features:", X_mnist_train.shape)
print("Training set - Labels:", y_mnist_train.shape)
print("Testing set - Features:", X_mnist_test.shape)
print("Testing set - Labels:", y_mnist_test.shape)
```

```
Training set - Features: (60000, 784)
Training set - Labels: (60000,)
Testing set - Features: (10000, 784)
Testing set - Labels: (10000,)
```

Spambase

```
# Load the spambase dataset
spambase = fetch_openml('spambase', version=1)

# Separate the features and labels
X_spambase = spambase['data']
y_spambase = spambase['target']
X_spambase.shape, y_spambase.shape
```

```
((4601, 57), (4601,))
```

```
# Split the data into train and test sets
X_spambase_train, X_spambase_test, y_spambase_train, y_spambase_test = tts(X_spambase, y_spambase,
                                                                           test_size=0.2, stratify=y_spambase, random_state=8)
# Print the shapes of the sets
print("Training set - Features:", X_spambase_train.shape)
print("Training set - Labels:", y_spambase_train.shape)
print("Testing set - Features:", X_spambase_test.shape)
print("Testing set - Labels:", y_spambase_test.shape)
```

```
Training set - Features: (3680, 57)
Training set - Labels: (3680,)
Testing set - Features: (921, 57)
Testing set - Labels: (921,)
```

20NewsGroup

```
# 20NG dataset categories
categories = ['alt.atheism', 'comp.graphics', 'comp.os.ms-windows.misc', 'comp.sys.ibm.pc.hardware',
              'comp.sys.mac.hardware', 'comp.windows.x', 'misc.forsale', 'rec.autos', 'rec.motorcycles',
              'rec.sport.baseball', 'rec.sport.hockey', 'sci.crypt', 'sci.electronics', 'sci.med',
              'sci.space', 'soc.religion.christian', 'talk.politics.guns', 'talk.politics.mideast',
              'talk.politics.misc', 'talk.religion.misc']
# Load the 20NG dataset
dataset = fetch_20newsgroups(subset='all', categories=categories, shuffle=True, random_state=5)

# Convert text to numerical features using TF-IDF vectorization
vectorizer = TfidfVectorizer(max_df=0.4, min_df=50, stop_words='english', max_features=2000)
X_newsgroups = vectorizer.fit_transform(dataset.data)
y_newsgroups = dataset.target

X_newsgroups.shape, y_newsgroups.shape
```

```
((18846, 2000), (18846,))
```

```
# Split the data into train and test sets
X_newsgroups_train, X_newsgroups_test, y_newsgroups_train, y_newsgroups_test = tts(X_newsgroups, y_newsgroups,
                                                                           test_size=0.2, stratify=y_newsgroups, random_state=8)
# Print the shapes of the sets
print("Training set - Features:", X_newsgroups_train.shape)
print("Training set - Labels:", y_newsgroups_train.shape)
print("Testing set - Features:", X_newsgroups_test.shape)
print("Testing set - Labels:", y_newsgroups_test.shape)
```

```
Training set - Features: (15076, 2000)
Training set - Labels: (15076,)
Testing set - Features: (3770, 2000)
Testing set - Labels: (3770,)
```

Problem1 6 Runs of Supervised Training / Testing : 3 datasets (MNIST, Spambase, 20NG) x 2 Classification Algorithms (L2-reg Logistic Regression, Decision Trees).

MNIST

```
# Perform supervised training/testing for MNIST dataset
# Logistic Regression
lr_mnist = LogisticRegression(penalty='l2',random_state=5)
lr_mnist.fit(X_mnist_train, y_mnist_train)
```

```
LogisticRegression
LogisticRegression(random_state=5)
```

```
# Decision Tree
dt_mnist = DecisionTreeClassifier(random_state=5)
dt_mnist.fit(X_mnist_train, y_mnist_train)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(random_state=5)
```

```
y_pred_mnist_lr = lr_mnist.predict(X_mnist_test)
accuracy_mnist_lr = accuracy_score(y_mnist_test, y_pred_mnist_lr)

y_pred_mnist_dt = dt_mnist.predict(X_mnist_test)
accuracy_mnist_dt = accuracy_score(y_mnist_test, y_pred_mnist_dt)

print("MNIST Logistic Regression Accuracy:", accuracy_mnist_lr)
print("MNIST Decision Tree Accuracy:", accuracy_mnist_dt)
```

```
MNIST Logistic Regression Accuracy: 0.9255
MNIST Decision Tree Accuracy: 0.8781
```

```
top_features_lr_mnist = sorted(range(len(lr_mnist.coef_[0])), key=lambda i: abs(lr_mnist.coef_[0][i]), reverse=True)[:30]
top_features_dt_mnist = dt_mnist.feature_importances_.argsort()[::-1][:30]
print(f'Logistic Regression top 30 features: {top_features_lr_mnist}')
print(f'Decision Tree top 30 features: {top_features_dt_mnist}')
```

```
Logistic Regression top 30 features: [461, 434, 408, 379, 351, 360, 406, 323, 712, 462, 629, 714, 489, 517, 378, 407, 416,
Decision Tree top 30 features: [489 435 568 350 430 211 346 405 234 156  98 484 486 290 354 655 402 153
 155 271 101 381 658 348 267 296 297 514 300  95]
```

SpamBase

```
# Perform supervised training/testing for Spambase dataset
# Logistic Regression
lr_spambase = LogisticRegression(penalty='l2', random_state=5)
lr_spambase.fit(X_spambase_train, y_spambase_train)
```

```
LogisticRegression
LogisticRegression(random_state=5)
```

```
# Decision Tree
dt_spambase = DecisionTreeClassifier(random_state=5)
dt_spambase.fit(X_spambase_train, y_spambase_train)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(random_state=5)
```

```
# Model performance
y_pred_spambase_lr = lr_spambase.predict(X_spambase_test)
accuracy_spambase_lr = accuracy_score(y_spambase_test, y_pred_spambase_lr)

y_pred_spambase_dt = dt_spambase.predict(X_spambase_test)
accuracy_spambase_dt = accuracy_score(y_spambase_test, y_pred_spambase_dt)

print("Spambase Logistic Regression Accuracy:", accuracy_spambase_lr)
print("Spambase Decision Tree Accuracy:", accuracy_spambase_dt)
```

```
Spambase Logistic Regression Accuracy: 0.9229098805646037
Spambase Decision Tree Accuracy: 0.9044516829533116
```

```
# top 30 features for both models
top_features_lr_spambase = sorted(range(len(lr_spambase.coef_[0])), key=lambda i: abs(lr_spambase.coef_[0][i]), reverse=True)
top_features_dt_spambase = dt_spambase.feature_importances_.argsort()[::-1][:30]
```

```
print(f'Logistic Regression top 30 features: {top_features_lr_spambase}')
print(f'Decision Tree top 30 features: {top_features_dt_spambase}')
```

```
Logistic Regression top 30 features: [26, 24, 25, 45, 6, 15, 22, 41, 44, 4, 7, 16, 52, 32, 43, 34, 28, 5, 19, 48, 36, 38,
Decision Tree top 30 features: [51  6 54 24 15 52 55 45 56  7 11 26  4 18 41 10 16 20 44 49  5 27  9  2
 17 23 22 34  8 25]
```

20NewsGroup

```
# Perform supervised training/testing for NewsGroup dataset
# Logistic Regression
lr_newsgroups = LogisticRegression(penalty='l2', random_state=5)
lr_newsgroups.fit(X_newsgroups_train, y_newsgroups_train)
```

```
LogisticRegression
LogisticRegression(random_state=5)
```

```
# Decision Tree with depth = 16
dt_newsgroups_16 = DecisionTreeClassifier(random_state=5, max_depth=16)
dt_newsgroups_16.fit(X_newsgroups_train, y_newsgroups_train)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(max_depth=16, random_state=5)
```

```
# Decision Tree with depth = 64
dt_newsgroups_64 = DecisionTreeClassifier(random_state=5, max_depth=64)
dt_newsgroups_64.fit(X_newsgroups_train, y_newsgroups_train)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(max_depth=64, random_state=5)
```

```
# Report the performance
y_pred_newsgroups_lr = lr_newsgroups.predict(X_newsgroups_test)
accuracy_newsgroups_lr = accuracy_score(y_newsgroups_test, y_pred_newsgroups_lr)

y_pred_depth_16 = dt_newsgroups_16.predict(X_newsgroups_test)
accuracy_newsgroups_dt16 = accuracy_score(y_newsgroups_test, y_pred_depth_16)

y_pred_depth_64 = dt_newsgroups_64.predict(X_newsgroups_test)
accuracy_newsgroups_dt64 = accuracy_score(y_newsgroups_test, y_pred_depth_64)

print("NewsGroups Logistic Regression Accuracy:", accuracy_newsgroups_lr)
print("Decision Tree - Depth=16 Accuracy:", accuracy_newsgroups_dt16)
print("Decision Tree - Depth=64 Accuracy:", accuracy_newsgroups_dt64)
```

```
NewsGroups Logistic Regression Accuracy: 0.8297082228116711
Decision Tree - Depth=16 Accuracy: 0.3885941644562334
Decision Tree - Depth=64 Accuracy: 0.5763925729442971
```

```
# top 30 features for both models
top_features_lr_newsgroups = sorted(range(len(lr_newsgroups.coef_[0])),
                                    key=lambda i: abs(lr_newsgroups.coef_[0][i]), reverse=True)[:30]
top_features_dt_newsgroups = dt_newsgroups_16.feature_importances_.argsort()[::-1][:30]
print(f'Logistic Regression top 30 features: {top_features_lr_newsgroups}')
print(f'Decision Tree top 30 features: {top_features_dt_newsgroups}')
```

```
Logistic Regression top 30 features: [238, 821, 997, 236, 966, 965, 293, 1509, 1071, 1196, 1576, 1908, 1579, 314, 1195, 18
Decision Tree top 30 features: [ 428  611 1574 1957  888 1569  363 1686  968  268  845  821 1097 1841
  832 1771  296  236   29 1202 1141 1045  846 1159  536   64 1951  759
  914  143]
```

```
# Print the top features for each run
print("Top 30 Features for each model")
print("MNIST - Logistic Regression: ", top_features_lr_mnist)
print("MNIST - Decision Tree: ", top_features_dt_mnist)
```

```
print("Spambase - Logistic Regression: ", top_features_lr_spambase)
print("Spambase - Decision Tree: ", top_features_dt_spambase)
print("NewsGroups - Logistic Regression: ", top_features_lr_newsgroups)
print("NewsGroups - Decision Tree: ", top_features_dt_newsgroups)
```

```
Top 30 Features for each model
MNIST - Logistic Regression:  [461, 434, 408, 379, 351, 360, 406, 323, 712, 462, 629, 714, 489, 517, 378, 407, 416, 715, 7
MNIST - Decision Tree:  [489 435 568 350 430 211 346 405 234 156  98 484 486 290 354 655 402 153
 155 271 101 381 658 348 267 296 297 514 300  95]
Spambase - Logistic Regression:  [26, 24, 25, 45, 6, 15, 22, 41, 44, 4, 7, 16, 52, 32, 43, 34, 28, 5, 19, 48, 36, 38, 23,
Spambase - Decision Tree:  [51  6 54 24 15 52 55 45 56  7 11 26  4 18 41 10 16 20 44 49  5 27  9  2
 17 23 22 34  8 25]
NewsGroups - Logistic Regression:  [238, 821, 997, 236, 966, 965, 293, 1509, 1071, 1196, 1576, 1908, 1579, 314, 1195, 1857
NewsGroups - Decision Tree:  [ 428  611 1574 1957  888 1569  363 1686  968  268  845  821 1097 1841
  832 1771  296  236   29 1202 1141 1045  846 1159  536   64 1951  759
  914  143]
```

```
# Print the accuracy score for each run
print("Accuracy Score for each model")
print("MNIST - Logistic Regression: ", accuracy_mnist_lr)
print("MNIST - Decision Tree: ", accuracy_mnist_dt)
print("Spambase - Logistic Regression: ", accuracy_spambase_lr)
print("Spambase - Decision Tree: ", accuracy_spambase_dt)
print("NewsGroups - Logistic Regression: ", accuracy_newsgroups_lr)
print("NewsGroups - Decision Tree: ", accuracy_newsgroups_dt64)
```

```
Accuracy Score for each model
MNIST - Logistic Regression:  0.9255
MNIST - Decision Tree:  0.8781
Spambase - Logistic Regression:  0.9229098805646037
Spambase - Decision Tree:  0.9044516829533116
NewsGroups - Logistic Regression:  0.8297082228116711
NewsGroups - Decision Tree:  0.5763925729442971
```

Problem 2

```
'''
For MNIST   dataset,    run a   PCA-library to  get data    on  D=5 features.
'''
# Reduce dimensionality to D=5 using PCA for MNIST
pca_mnist_5 = PCA(n_components=5)
X_mnist_train_5 = pca_mnist_5.fit_transform(X_mnist_train)
X_mnist_test_5 = pca_mnist_5.transform(X_mnist_test)

X_mnist_train_5.shape, X_mnist_test_5.shape
```

```
((60000, 5), (10000, 5))
```

```
# Perform supervised training/testing with Logistic Regression on MNIST (D=5)
lr_mnist_5 = LogisticRegression(penalty='l2',random_state=5)
lr_mnist_5.fit(X_mnist_train_5, y_mnist_train)
y_pred = lr_mnist_5.predict(X_mnist_test_5)
mnist_lr_accuracy_5 = accuracy_score(y_mnist_test, y_pred)
mnist_lr_accuracy_5
```

```
0.6854
```

```
# Perform supervised training/testing with Decision Tree on MNIST (D=5)
dt_mnist_5 = DecisionTreeClassifier(random_state=5)
dt_mnist_5.fit(X_mnist_train_5, y_mnist_train)
y_pred = dt_mnist_5.predict(X_mnist_test_5)
mnist_dt_accuracy_5 = accuracy_score(y_mnist_test, y_pred)
mnist_dt_accuracy_5
```

```
0.6665
```

```
# Reduce dimensionality to D=20 using PCA for MNIST
pca_mnist_20 = PCA(n_components=20)
X_mnist_train_20 = pca_mnist_20.fit_transform(X_mnist_train)
X_mnist_test_20 = pca_mnist_20.transform(X_mnist_test)

X_mnist_train_20.shape, X_mnist_test_20.shape
```

```
((60000, 20), (10000, 20))
```

```
# Perform supervised training/testing with Logistic Regression on MNIST (D=20)
lr_mnist_20 = LogisticRegression(penalty='l2',random_state=5)
lr_mnist_20.fit(X_mnist_train_20, y_mnist_train)
y_pred = lr_mnist_20.predict(X_mnist_test_20)
mnist_lr_accuracy_20 = accuracy_score(y_mnist_test, y_pred)
mnist_lr_accuracy_20
```

```
0.8802
```

```
# Perform supervised training/testing with Decision Tree on MNIST (D=20)
dt_mnist_20 = DecisionTreeClassifier(random_state=5)
dt_mnist_20.fit(X_mnist_train_20, y_mnist_train)
y_pred = dt_mnist_20.predict(X_mnist_test_20)
mnist_dt_accuracy_20 = accuracy_score(y_mnist_test, y_pred)
mnist_dt_accuracy_20
```

```
0.8468
```

```
# compare   testing performance with    the problem 1
print('MNIST all model Accuracy comparison')
print('Logistic Regression: ', accuracy_mnist_lr)
print('Decision Tree: ', accuracy_mnist_dt)
print('Logistic Regression after applying PCA(D=5): ', mnist_lr_accuracy_5)
print('Decision Tree after applying PCA(D=5): ', mnist_dt_accuracy_5)
print('Logistic Regression after applying PCA(D=20): ', mnist_lr_accuracy_20)
print('Decision Tree after applying PCA(D=20): ', mnist_dt_accuracy_20)
```

```
MNIST all model Accuracy comparison
    Logistic Regression:  0.9255
    Decision Tree:  0.8781
    Logistic Regression after applying PCA(D=5):  0.6854
    Decision Tree after applying PCA(D=5):  0.6665
    Logistic Regression after applying PCA(D=20):  0.8802
    Decision Tree after applying PCA(D=20):  0.8468
```

```
'''
Run PCA library on  Spambase    and repeat  one of  the classification  algorithms
'''
# Reduce dimensionality to D using PCA for Spambase
min_accuracy = 0.9  # Set a minimum accuracy threshold for comparison
D = 1  # Initialize D (number of PCA dimensions)

while True:
    pca_spambase = PCA(n_components=D)
    X_spambase_train_pca = pca_spambase.fit_transform(X_spambase_train)
    X_spambase_test_pca = pca_spambase.transform(X_spambase_test)

    # Perform supervised training/testing with Logistic Regression on Spambase
    lr_spambase = LogisticRegression(penalty='l2', random_state=5)
    lr_spambase.fit(X_spambase_train_pca, y_spambase_train)
    y_pred = lr_spambase.predict(X_spambase_test_pca)
    accuracy = accuracy_score(y_spambase_test, y_pred)
    print(f'D:{D}, Accurcay: {accuracy}')

    if accuracy >= min_accuracy:
        break

    D += 1

# Print the smallest D needed to achieve a comparable test result
print("Smallest D for comparable test result:", D)
```

```
D:1, Accurcay: 0.6731813246471227
D:2, Accurcay: 0.7296416938110749
```

```
D:3, Accurcay: 0.754614549402823
D:4, Accurcay: 0.7926167209554832
D:5, Accurcay: 0.7991313789359392
D:6, Accurcay: 0.8349619978284474
D:7, Accurcay: 0.8469055374592834
D:8, Accurcay: 0.8545059717698155
D:9, Accurcay: 0.8621064060803475
D:10, Accurcay: 0.8610206297502715
D:11, Accurcay: 0.8523344191096635
D:12, Accurcay: 0.8534201954397395
D:13, Accurcay: 0.9033659066232356
Smallest D for comparable test result: 13
```

Problem 3

```
'''
Repeat  PB2 exercises   on  MNIST   (D=5    and D=20)   with    your    own PCA implementation.
'''
def pca_train(X_train, n_components):
    # Compute the covariance matrix
    covariance_matrix = np.cov(X_train.T)

    # Perform eigenvalue decomposition
    eigenvalues, eigenvectors = np.linalg.eig(covariance_matrix)

    # Sort eigenvalues and eigenvectors in descending order
    sorted_indices = np.argsort(eigenvalues)[::-1]
    sorted_eigenvalues = eigenvalues[sorted_indices]
    sorted_eigenvectors = eigenvectors[:, sorted_indices]

    # Select the top n_components eigenvectors
    components = sorted_eigenvectors[:, :n_components]

    # Project the training data onto the principal components
    X_train_projected = np.dot(X_train, components)

    return X_train_projected, components

def pca_test(X_test, components):
    # Project the testing data onto the principal components
    X_test_projected = np.dot(X_test, components)

    return X_test_projected


# Reduce dimensionality to D=5 using PCA for MNIST
X_mnist_train_5, components = pca_train(X_mnist_train, n_components=5)
X_mnist_test_5 = pca_test(X_mnist_test, components)

X_mnist_train_5.shape, X_mnist_test_5.shape
```

```
((60000, 5), (10000, 5))
```

```
# Perform supervised training/testing with Logistic Regression on MNIST (D=5)
lr_mnist_5 = LogisticRegression(penalty='l2',random_state=5)
lr_mnist_5.fit(X_mnist_train_5, y_mnist_train)
y_pred = lr_mnist_5.predict(X_mnist_test_5)
mnist_lr_accuracy_5 = accuracy_score(y_mnist_test, y_pred)
mnist_lr_accuracy_5
```

```
0.6704
```

```
# Perform supervised training/testing with Decision Tree on MNIST (D=5)
dt_mnist_5 = DecisionTreeClassifier(random_state=5)
dt_mnist_5.fit(X_mnist_train_5, y_mnist_train)
y_pred = dt_mnist_5.predict(X_mnist_test_5)
mnist_dt_accuracy_5 = accuracy_score(y_mnist_test, y_pred)
mnist_dt_accuracy_5
```

```
0.6718
```

```
# Reduce dimensionality to D=20 using PCA for MNIST
X_mnist_train_20, components = pca_train(X_mnist_train, n_components=20)
X_mnist_test_20 = pca_test(X_mnist_test, components)
```

```
X_mnist_train_20.shape, X_mnist_test_20.shape
```

```
((60000, 20), (10000, 20))
```

```
# Perform supervised training/testing with Logistic Regression on MNIST (D=20)
lr_mnist_20 = LogisticRegression(penalty='l2',random_state=5)
lr_mnist_20.fit(X_mnist_train_20, y_mnist_train)
y_pred = lr_mnist_20.predict(X_mnist_test_20)
mnist_lr_accuracy_20 = accuracy_score(y_mnist_test, y_pred)
mnist_lr_accuracy_20
```

```
0.8767
```

```
# Perform supervised training/testing with Decision Tree on MNIST (D=20)
dt_mnist_20 = DecisionTreeClassifier(random_state=5)
dt_mnist_20.fit(X_mnist_train_20, y_mnist_train)
y_pred = dt_mnist_20.predict(X_mnist_test_20)
mnist_dt_accuracy_20 = accuracy_score(y_mnist_test, y_pred)
mnist_dt_accuracy_20
```

```
0.8465
```

```
print('MNIST all model Accuracy comparison')
print('Logistic Regression: ', accuracy_mnist_lr)
print('Decision Tree: ', accuracy_mnist_dt)
print('Logistic Regression after applying Implemented PCA(D=5): ', mnist_lr_accuracy_5)
print('Decision Tree after applying Implemented PCA(D=5): ', mnist_dt_accuracy_5)
print('Logistic Regression after applying Implemented PCA(D=20): ', mnist_lr_accuracy_20)
print('Decision Tree after applying Implemented PCA(D=20): ', mnist_dt_accuracy_20)
```

```
MNIST all model Accuracy comparison
    Logistic Regression:  0.9255
    Decision Tree:  0.8781
    Logistic Regression after applying Implemented PCA(D=5):  0.6704
    Decision Tree after applying Implemented PCA(D=5):  0.6718
    Logistic Regression after applying Implemented PCA(D=20):  0.8767
    Decision Tree after applying Implemented PCA(D=20):  0.8465
```

Problem 4

```
'''
PCA for cluster visualization
'''
# Sample a subset of the data (optional)
sample_indices = np.random.choice(range(len(X_mnist)), size=5000, replace=False)
X_sample = X_mnist.values[sample_indices]
y_sample = y_mnist.values[sample_indices]

# Run KMeans clustering on the data
n_clusters = 10
kmeans = KMeans(n_clusters=n_clusters, random_state=42)
clusters = kmeans.fit_predict(X_sample)
clusters
```

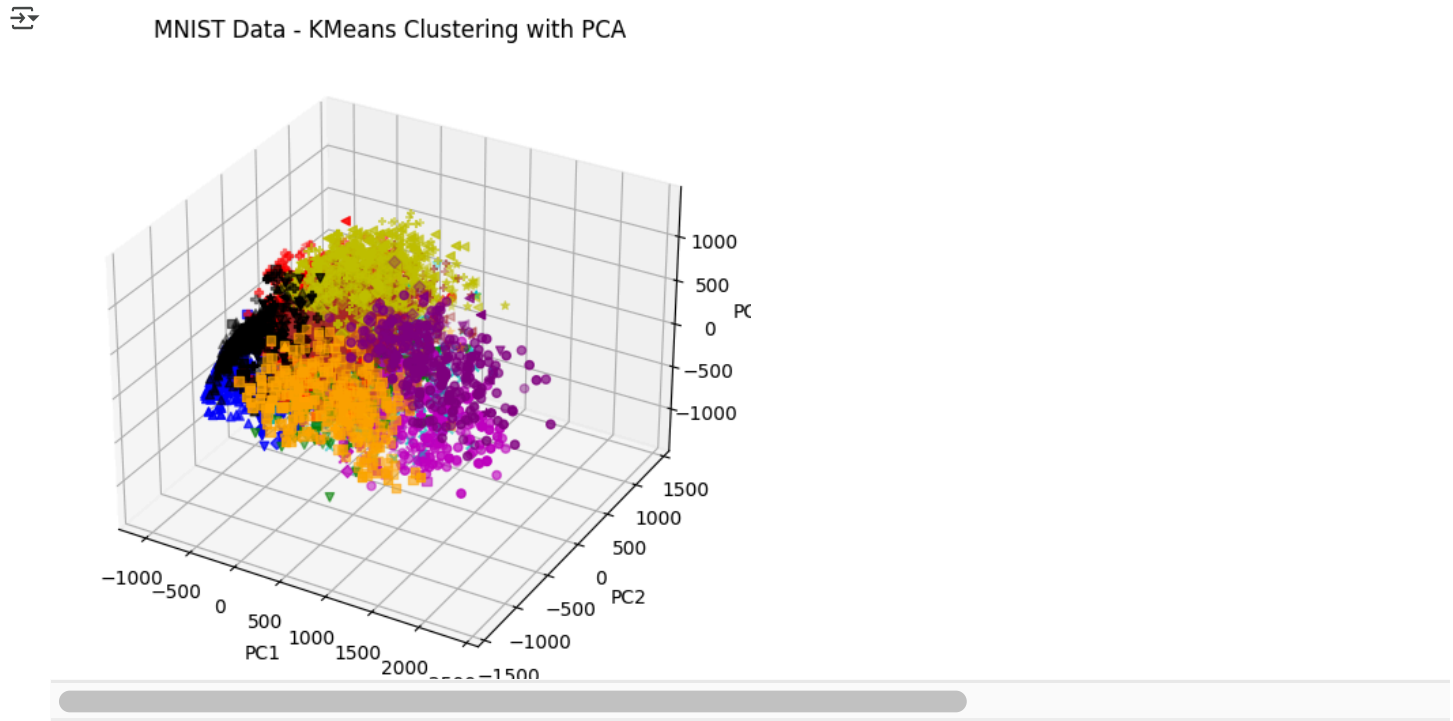
```
array([5, 7, 5, ..., 7, 9, 9], dtype=int32)
```

```
# Run PCA on the data
n_components = 3
pca = PCA(n_components=n_components)
X_pca = pca.fit_transform(X_sample)
X_pca.shape
```

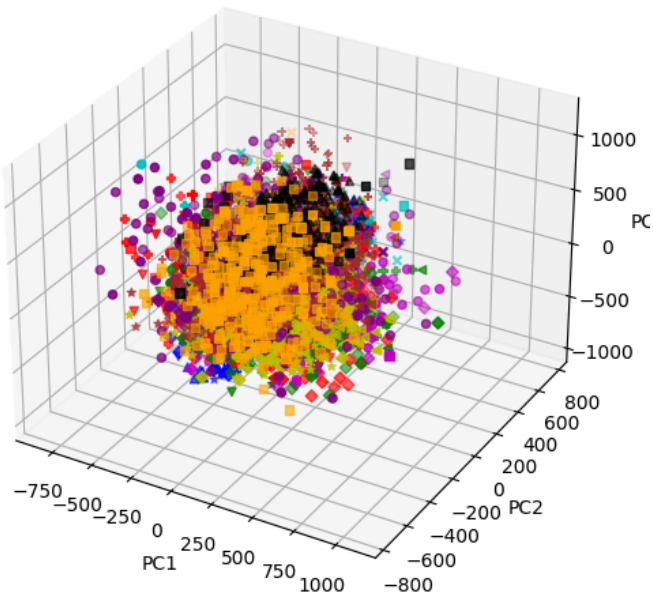
```
(5000, 3)
```

```
def plot_data(X_pca, y_sample):
  # Define shapes and colors for digit labels and cluster IDs
  shapes = ['o', '^', 's', 'P', '*', 'D', 'x', '+', 'v', '<']
  colors = ['r', 'b', 'g', 'c', 'm', 'y', 'k', 'orange', 'purple', 'brown']

  # Plot data in 3D with PCA representation
```

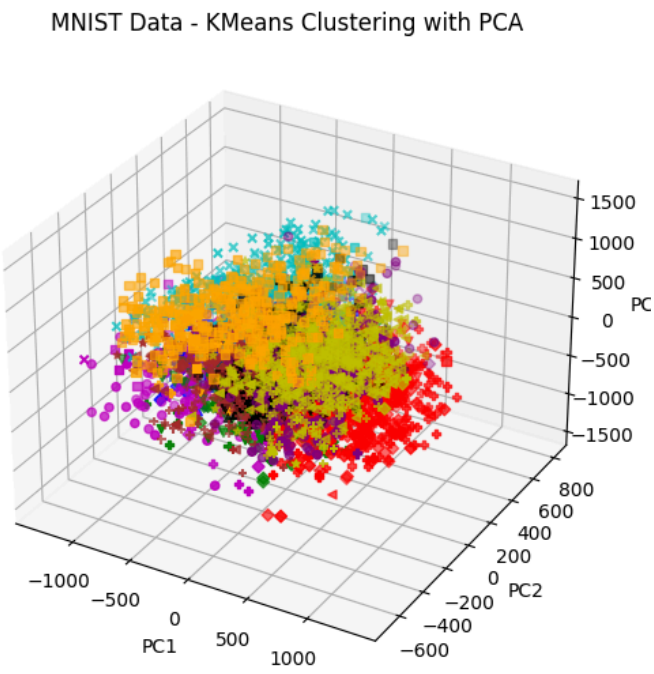
```
  fig = plt.figure(figsize=(5,10))
  ax = fig.add_subplot(111, projection='3d')

  # # Plot data points with corresponding shapes and colors
  for i in range(10):
      for j in range(10):
          ax.scatter(
              X_pca[(clusters == i) & (y_sample.astype('int') == j), 0],
              X_pca[(clusters == i) & (y_sample.astype('int') == j), 1],
              X_pca[(clusters == i) & (y_sample.astype('int') == j), 2],
              c=colors[i],
              marker=shapes[j],
          )
  ax.set_xlabel('PC1')
  ax.set_ylabel('PC2')
  ax.set_zlabel('PC3')
  ax.set_title('MNIST Data - KMeans Clustering with PCA')
  # ax.legend(bbox_to_anchor = (1.25, 0.6), loc='center right')
  plt.tight_layout()
  plt.show()
plot_data(X_pca, y_sample)
```




```
# Run PCA on the data with D = 20
pca = PCA(n_components=20)
X_pca = pca.fit_transform(X_sample)

# Select 3 random eigenvalues from top 20
random_indices = np.random.choice(range(20), size=3, replace=False)
X_pca_random = X_pca[:, random_indices]
plot_data(X_pca_random, y_sample)
print(random_indices)
```



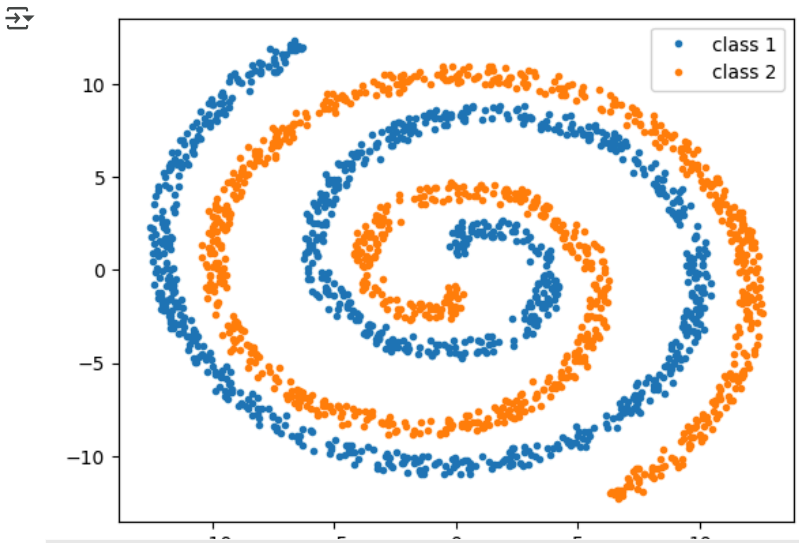

```
# Select 3 random eigenvalues from top 20
random_indices = np.random.choice(range(20), size=3, replace=False)
X_pca_random = X_pca[:, random_indices]
plot_data(X_pca_random, y_sample)
print(random_indices)
```



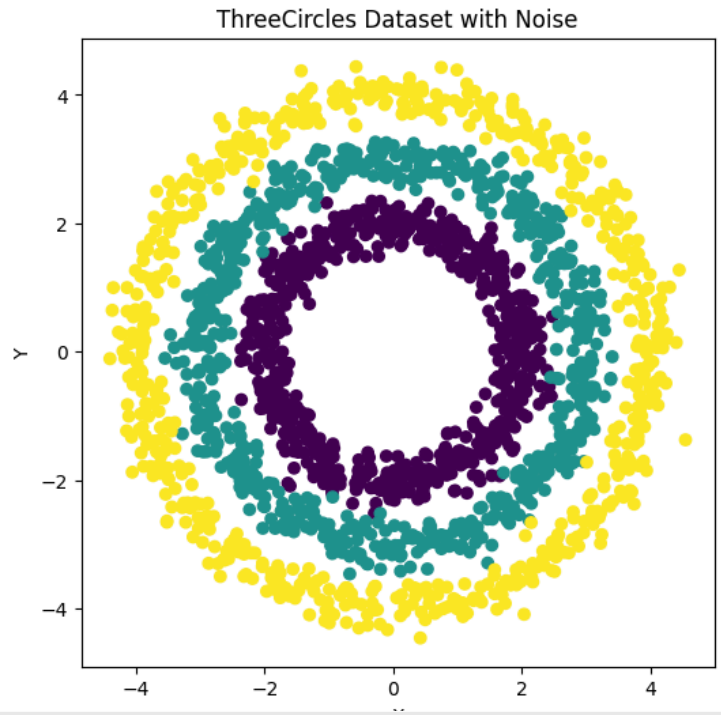
Problem 5

```
def twospirals(n_points, noise=1):
    """
     Returns the two spirals dataset.
    """
    n = np.sqrt(np.random.rand(n_points,1)) * 780 * (2*np.pi)/360
    d1x = -np.cos(n)*n + np.random.rand(n_points,1) * noise
    d1y = np.sin(n)*n + np.random.rand(n_points,1) * noise
```

```
    return (np.vstack((np.hstack((d1x,d1y)),np.hstack((-d1x,-d1y)))),
            np.hstack((np.zeros(n_points),np.ones(n_points))))

X_twospirals, y_twospirals = twospirals(1000)

plt.plot(X_twospirals[y_twospirals==0,0], X_twospirals[y_twospirals==0,1], '.', label='class 1')
plt.plot(X_twospirals[y_twospirals==1,0], X_twospirals[y_twospirals==1,1], '.', label='class 2')
plt.legend()
plt.show()
```




```
# Generate ThreeCircles dataset with noise
n_samples = 2000

# Circle 1
radius1 = 2
theta1 = np.linspace(0, 2*np.pi, n_samples//3)
x1 = radius1 * np.cos(theta1) + np.random.normal(0, 0.2, n_samples//3)
y1 = radius1 * np.sin(theta1) + np.random.normal(0, 0.2, n_samples//3)

# Circle 2
radius2 = 3
theta2 = np.linspace(0, 2*np.pi, n_samples//3)
x2 = radius2 * np.cos(theta2) + np.random.normal(0, 0.2, n_samples//3)
y2 = radius2 * np.sin(theta2) + np.random.normal(0, 0.2, n_samples//3)

# Circle 3
radius3 = 4
theta3 = np.linspace(0, 2*np.pi, n_samples//3)
x3 = radius3 * np.cos(theta3) + np.random.normal(0, 0.2, n_samples//3)
y3 = radius3 * np.sin(theta3) + np.random.normal(0, 0.2, n_samples//3)

X_threecircles = np.vstack((np.column_stack((x1, y1)), np.column_stack((x2, y2)), np.column_stack((x3, y3))))
y_threecircles = np.repeat([0, 1, 2], n_samples//3)

# Plot ThreeCircles dataset with noise
plt.figure(figsize=(6, 6))
plt.scatter(X_threecircles[:, 0], X_threecircles[:, 1], c=y_threecircles, cmap='viridis')
plt.title('ThreeCircles Dataset with Noise')
plt.xlabel('X')
plt.ylabel('Y')
plt.show()
```




```
# Linear Regression on TwoSpirals
lr_twospirals = LinearRegression()
lr_twospirals.fit(X_twospirals, y_twospirals)
y_pred_twospirals = lr_twospirals.predict(X_twospirals)
mse_twospirals = mean_squared_error(y_twospirals, y_pred_twospirals)
print("MSE for Linear Regression on TwoSpirals:", mse_twospirals)

# Linear Regression on ThreeCircles
lr_threecircles = LinearRegression()
lr_threecircles.fit(X_threecircles, y_threecircles)
y_pred_threecircles = lr_threecircles.predict(X_threecircles)
mse_threecircles = mean_squared_error(y_threecircles, y_pred_threecircles)
print("MSE for Linear Regression on ThreeCircles:", mse_threecircles)
```

```
MSE for Linear Regression on TwoSpirals: 0.23832599223644196
MSE for Linear Regression on ThreeCircles: 0.6666639060614518
```

```
# TwoSpirals dataset
kpca_twospirals = KernelPCA(n_components=10, kernel='rbf')
X_transformed_twospirals = kpca_twospirals.fit_transform(X_twospirals)

# ThreeCircles dataset
kpca_threecircles = KernelPCA(n_components=10, kernel='rbf')
X_transformed_threecircles = kpca_threecircles.fit_transform(X_threecircles)
```

```
# Linear Regression on transformed data - TwoSpirals
lr_transformed_twospirals = LinearRegression()
lr_transformed_twospirals.fit(X_transformed_twospirals, y_twospirals)
y_pred_transformed_twospirals = lr_transformed_twospirals.predict(X_transformed_twospirals)
mse_transformed_twospirals = mean_squared_error(y_twospirals, y_pred_transformed_twospirals)
print("MSE for Linear Regression on transformed TwoSpirals:", mse_transformed_twospirals)

# Linear Regression on transformed data - ThreeCircles
lr_transformed_threecircles = LinearRegression()
lr_transformed_threecircles.fit(X_transformed_threecircles, y_threecircles)
y_pred_transformed_threecircles = lr_transformed_threecircles.predict(X_transformed_threecircles)
mse_transformed_threecircles = mean_squared_error(y_threecircles, y_pred_transformed_threecircles)
print("MSE for Linear Regression on transformed ThreeCircles:", mse_transformed_threecircles)
```

```
MSE for Linear Regression on transformed TwoSpirals: 0.22564143080738894
MSE for Linear Regression on transformed ThreeCircles: 0.04130257545720563
```

```
print("MSE for Linear Regression on ThreeCircles:", mse_threecircles)
print("MSE for Linear Regression on transformed ThreeCircles:", mse_transformed_threecircles)
print("MSE for Linear Regression on TwoSpirals:", mse_twospirals)
print("MSE for Linear Regression on transformed TwoSpirals:", mse_transformed_twospirals)
```

```
MSE for Linear Regression on ThreeCircles: 0.6666639060614518
MSE for Linear Regression on transformed ThreeCircles: 0.04130257545720563
MSE for Linear Regression on TwoSpirals: 0.23832599223644196
MSE for Linear Regression on transformed TwoSpirals: 0.22564143080738894
```

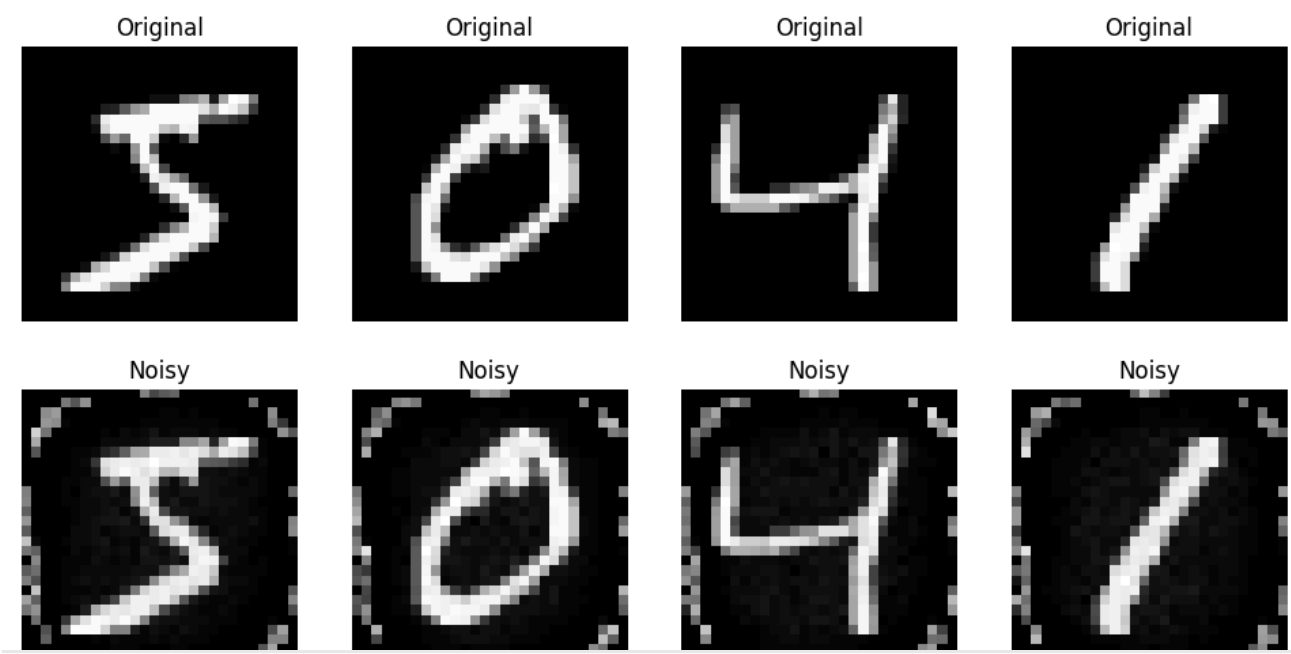
```
'''
Retrain Linear  regression  on  the transformed D-dim   data.   How large   D   needs   to  be  to  get good    performance.
'''

D = 1  # Initialize D (number of PCA dimensions)

while True:
    # TwoSpirals dataset
    kpca_twospirals = KernelPCA(n_components=D, kernel='rbf')
    X_transformed_twospirals = kpca_twospirals.fit_transform(X_twospirals)
    # ThreeCircles dataset
    kpca_threecircles = KernelPCA(n_components=D, kernel='rbf')
    X_transformed_threecircles = kpca_threecircles.fit_transform(X_threecircles)

    # Linear Regression on transformed data - TwoSpirals
    lr_transformed_twospirals = LinearRegression()
    lr_transformed_twospirals.fit(X_transformed_twospirals, y_twospirals)
    y_pred_transformed_twospirals = lr_transformed_twospirals.predict(X_transformed_twospirals)
    mse_transformed_twospirals = mean_squared_error(y_twospirals, y_pred_transformed_twospirals)
    print(f'PCA ({D})')
    print("MSE on transformed TwoSpirals:", mse_transformed_twospirals)

    # Linear Regression on transformed data - ThreeCircles
    lr_transformed_threecircles = LinearRegression()
    lr_transformed_threecircles.fit(X_transformed_threecircles, y_threecircles)
    y_pred_transformed_threecircles = lr_transformed_threecircles.predict(X_transformed_threecircles)
    mse_transformed_threecircles = mean_squared_error(y_threecircles, y_pred_transformed_threecircles)
    print("MSE on transformed ThreeCircles:", mse_transformed_threecircles)

    if D>=20:
        break

    D += 1

# Print the smallest D needed to achieve a comparable test result
print("Smallest D for comparable test result:", D)
```

```
PCA (1)
MSE on transformed TwoSpirals: 0.24082144454944382
MSE on transformed ThreeCircles: 0.6666268994977177
PCA (2)
MSE on transformed TwoSpirals: 0.24082144454944382
MSE on transformed ThreeCircles: 0.6666193112549216
PCA (3)
MSE on transformed TwoSpirals: 0.2405737675444782
MSE on transformed ThreeCircles: 0.6664452887050173
PCA (4)
MSE on transformed TwoSpirals: 0.2405737675444782
MSE on transformed ThreeCircles: 0.6663842785433244
PCA (5)
MSE on transformed TwoSpirals: 0.23937129668230056
MSE on transformed ThreeCircles: 0.6663771738969857
PCA (6)
MSE on transformed TwoSpirals: 0.23937129668230053
MSE on transformed ThreeCircles: 0.6659439663861773
PCA (7)
MSE on transformed TwoSpirals: 0.23480642588707828
MSE on transformed ThreeCircles: 0.04144942167181773
PCA (8)
MSE on transformed TwoSpirals: 0.23480642588707668
MSE on transformed ThreeCircles: 0.04144932276837453
PCA (9)
MSE on transformed TwoSpirals: 0.22564143080738916
MSE on transformed ThreeCircles: 0.041321020050705204
PCA (10)
MSE on transformed TwoSpirals: 0.22564143080738894
MSE on transformed ThreeCircles: 0.04130257545720563
PCA (11)
```

```
MSE on transformed TwoSpirals: 0.22026984533693814
MSE on transformed ThreeCircles: 0.041227693893254744
PCA (12)
MSE on transformed TwoSpirals: 0.22026984533693814
MSE on transformed ThreeCircles: 0.040808859972448694
PCA (13)
MSE on transformed TwoSpirals: 0.21972045174414415
MSE on transformed ThreeCircles: 0.04079073444554762
PCA (14)
MSE on transformed TwoSpirals: 0.20385146782841604
MSE on transformed ThreeCircles: 0.040706973967536
PCA (15)
MSE on transformed TwoSpirals: 0.20385146782841604
MSE on transformed ThreeCircles: 0.04057661968258885
PCA (16)
MSE on transformed TwoSpirals: 0.20385146782841604
MSE on transformed ThreeCircles: 0.04050175942047893
PCA (17)
MSE on transformed TwoSpirals: 0.20385146782841604
MSE on transformed ThreeCircles: 0.04049780848650503
PCA (18)
MSE on transformed TwoSpirals: 0.2036555878698734
MSE on transformed ThreeCircles: 0.04048033345140784
PCA (19)
MSE on transformed TwoSpirals: 0.17099847082383363
MSE on transformed ThreeCircles: 0.040439135042592424
PCA (20)
```

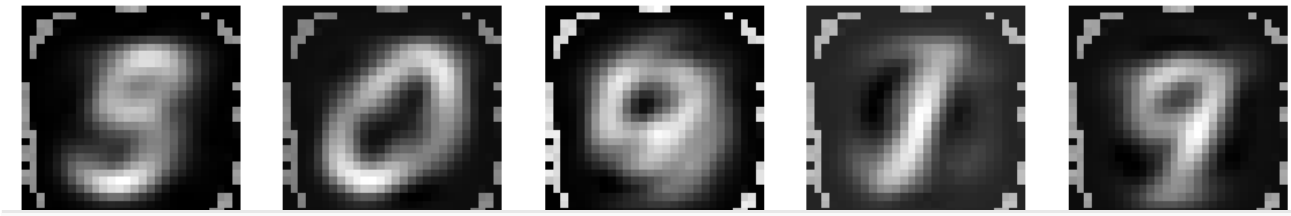
Problem 6

```
'''
Implement   Kernel  PCA on  MNIST
'''
# Add Gaussian noise to the MNIST images
np.random.seed(42)
noise_std = 0.05 * np.std(X_mnist)
X_mnist_noisy = X_mnist + np.random.normal(loc=0, scale=noise_std, size=X_mnist.shape)
# Scale the data to a range of [0, 1]
scaler = MinMaxScaler()
X_mnist_noisy = scaler.fit_transform(X_mnist_noisy[:5000])


# Plot original and noisy images
fig, axes = plt.subplots(nrows=2, ncols=4, figsize=(12, 6))
for i in range(4):
    axes[0, i].imshow(X_mnist.values[i].reshape(28, 28), cmap='gray')
    axes[0, i].axis('off')
    axes[0, i].set_title('Original')

    axes[1, i].imshow(X_mnist_noisy[i].reshape(28, 28), cmap='gray')
    axes[1, i].axis('off')
    axes[1, i].set_title('Noisy')
```




```
# Perform PCA on noisy images (D=5)
pca_dim_5 = 5
pca_5 = PCA(n_components=pca_dim_5)
X_pca_5 = pca_5.fit_transform(X_mnist_noisy)


n_images = 5
# Plot PCA-reconstructed images (D=5)
fig, ax = plt.subplots(1, n_images, figsize=(12, 6))
X_pca_5_reconstructed = pca_5.inverse_transform(X_pca_5)
for i in range(n_images):
    ax[i].imshow(X_pca_5_reconstructed[i].reshape(28, 28), cmap='gray')
    ax[i].axis('off')
print('PCA-Reconstructed Images (D=5)')
plt.show()
```

```
PCA-Reconstructed Images (D=5)
```



```
# Perform PCA on noisy images (D=20)
pca_dim_20 = 20
pca_20 = PCA(n_components=pca_dim_20)
X_pca_20 = pca_20.fit_transform(X_mnist_noisy)

# Plot PCA-reconstructed images (D=20)
fig, ax = plt.subplots(1, n_images, figsize=(12, 6))
X_pca_20_reconstructed = pca_20.inverse_transform(X_pca_20)
for i in range(n_images):
    ax[i].imshow(X_pca_20_reconstructed[i].reshape(28, 28), cmap='gray')
    ax[i].axis('off')
print('PCA-Reconstructed Images (D=20)')
plt.show()
```

PCA-Reconstructed Images (D=20)

```
# Perform Kernel-PCA on noisy images (D=5)
kpca_dim_5 = 5
kpca_5 = KernelPCA(n_components=kpca_dim_5, kernel='rbf', fit_inverse_transform =True)
X_kpca_5 = kpca_5.fit_transform(X_mnist_noisy)

# Plot Kernel-PCA-reconstructed images (D=5)
fig, ax = plt.subplots(1, 5, figsize=(12, 6))
X_kpca_5_reconstructed = kpca_5.inverse_transform(X_kpca_5)
for i in range(5):
    ax[i].imshow(X_kpca_5_reconstructed[i].reshape(28, 28), cmap='gray')
    ax[i].axis('off')
```