GE 461 - INTRODUCTION TO DATA SCIENCE

PROJECT - FALL DETECTION

Necati Furkan Çolak / 21803512

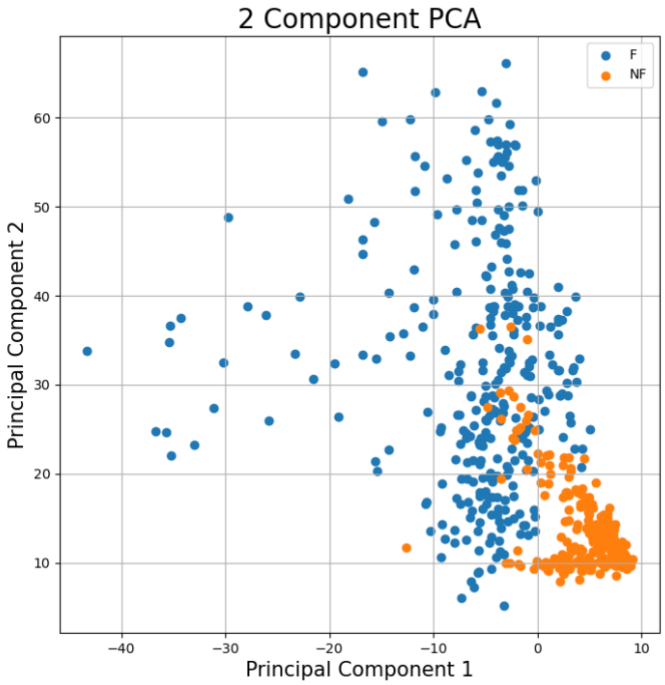
PART-A:

In this question, we have performed exploratory data analysis by applying Principal Component Analysis (PCA) to the `data_train` dataset. To begin, we standardized the dataset using the `StandardScaler` to ensure that all features have a mean of zero and a variance of one. Then, we utilized PCA with a parameter `n_components` set to 2, indicating our intention to retain the first two principal components. The `fit_transform()` method was employed to calculate and obtain the transformed data. As a result, we obtained the following outcome:

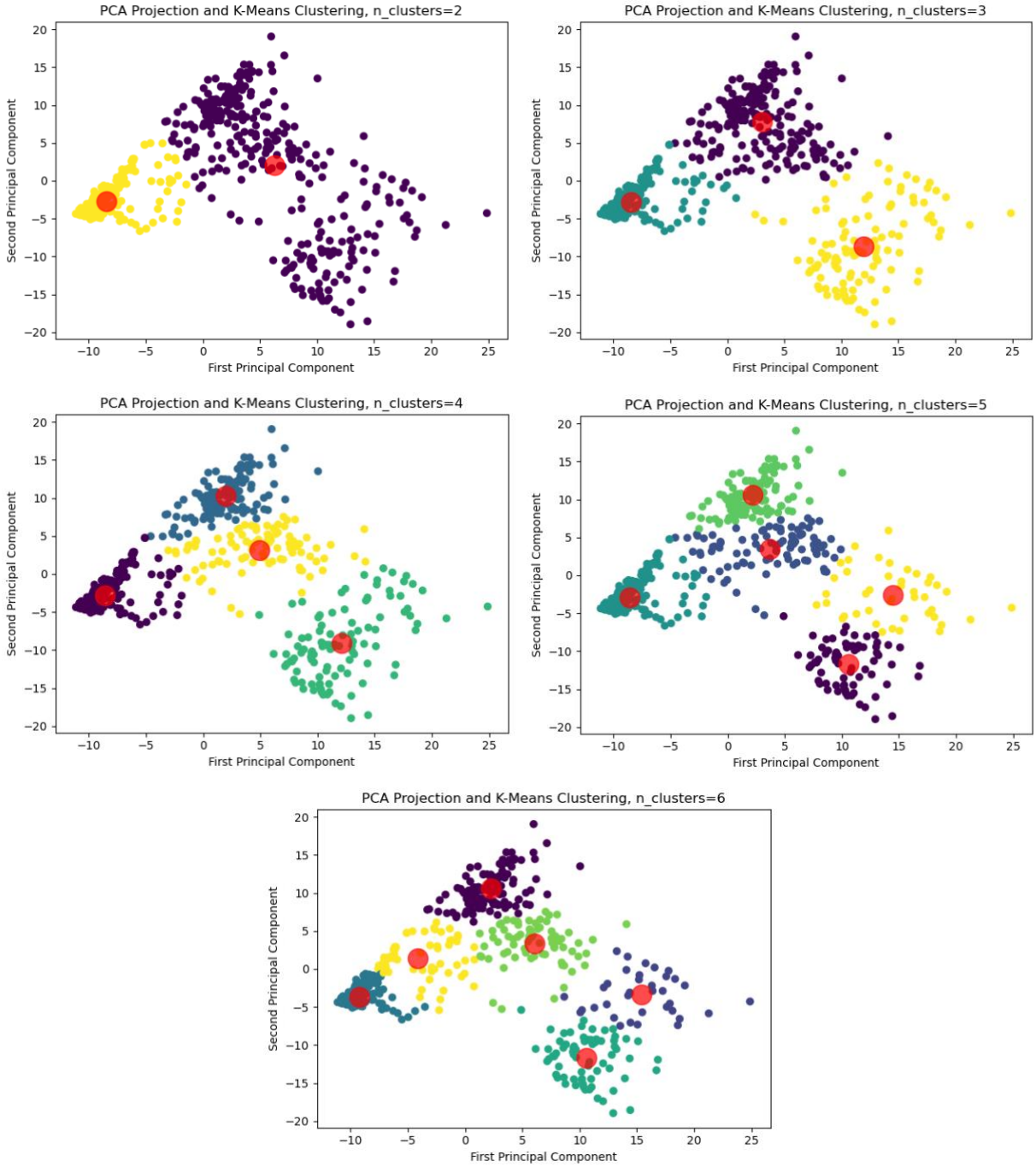
The first two principal components capture 0.23 and 0.17 of the total variance present in the given dataset. Subsequently, when PCA was applied, the projected representation of the data can be observed below:

[0.23101268, 0.1762439]

This representation signifies the transformed data obtained after performing PCA.

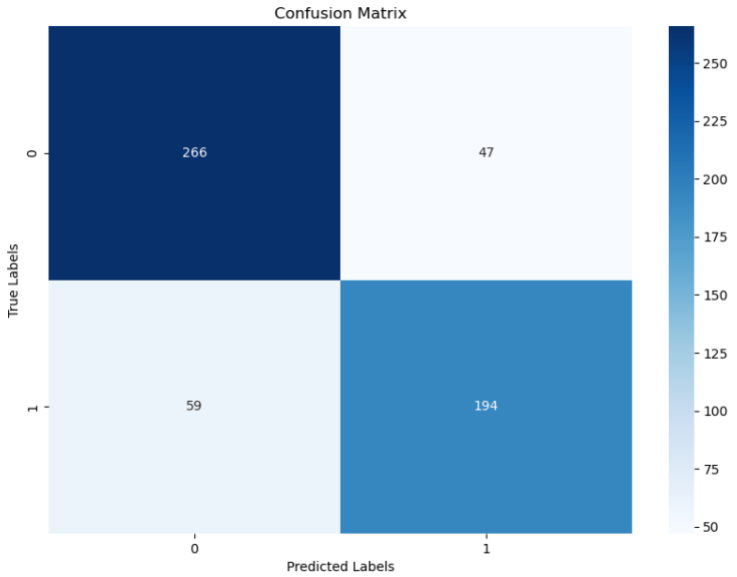


We proceeded to cluster the data using the k-means clustering algorithm, and the resulting plots a re displayed below.



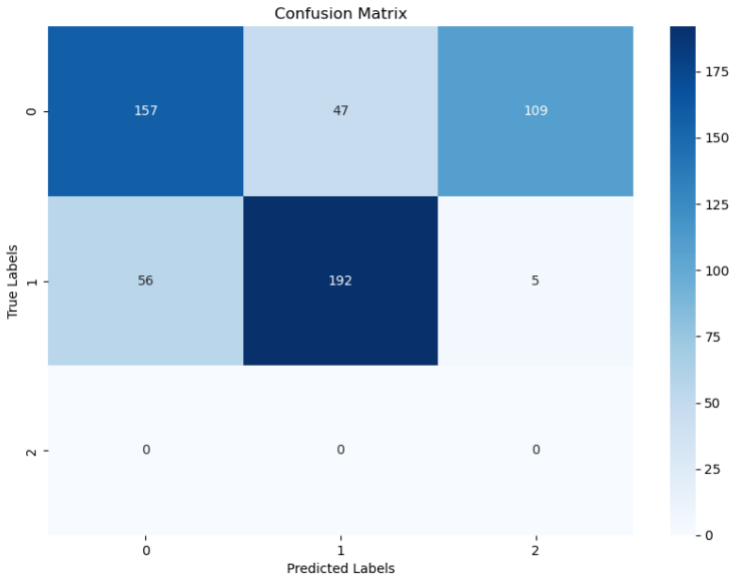
To evaluate the performance of different cluster numbers, we utilized a confusion matrix. The res ulting plots illustrating the confusion matrix can be observed below.

Cluster for 2:



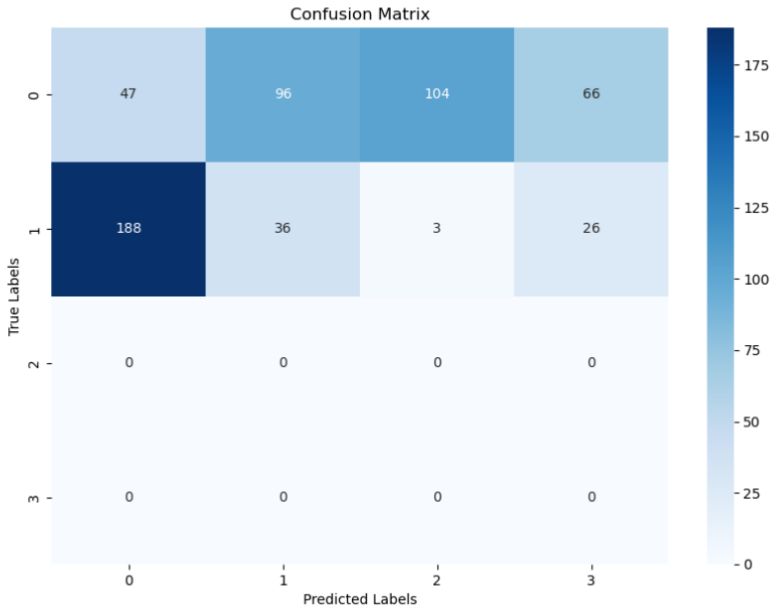
Consistency: 0.81272
Accuracy: 0.3899

Cluster for 3:



Accuracy: 0.6166
Consistency: 0.2519

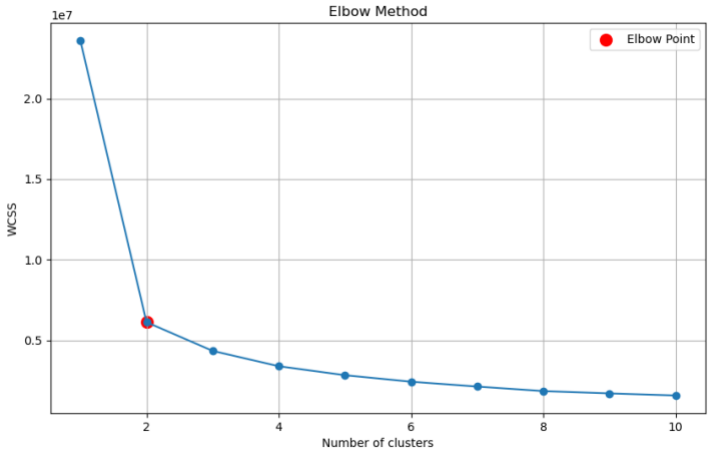
Cluster for 4:



Accuracy: 0.1466
Consistency: 0.2108

Based on our observation, we noticed that the accuracy consistently decreases when we increase the cluster number. Therefore, we can determine that a cluster number of 2 is the most suitable choice. Additionally, considering the presence of two labels in the dataset, it might be beneficial to merge the cluster numbers into 2. This merging could potentially enhance the overall accuracy and consistency of the clustering results as we increase the cluster number.

The reason why I found this analysis insufficient I try to find best cluster number with Elbow method.



In that plot, curve is so smooth and although 2 is labeled as elbow point and sharp decrease is observed at number of clusters 2. In that case, 2 can be a really good option to cluster.

Comment on PART-A:

The results of our analysis indicate a significant finding in the realm of fall detection. Notably, the optimal number of clusters for our model appears to be two, as our tests demonstrate a steep decline in accuracy with an increased number of clusters. This suggests that a binary categorization (i.e., fall or no fall) yields the highest predictive accuracy. It's possible to determine fall detection with considerable certainty using this two-cluster approach. However, the sharply decreased accuracy with increased clusters implies that detecting nuanced fall patterns or categorizing fall types might be a challenging endeavor. It underscores the need for refining our model or possibly considering a different modeling approach to handle more sophisticated fall pattern recognitions maybe.

PART-B:

MLP:

Index	Solver	Learning Rate	Hidden Layer Sizes	Alpha	Activation	Classification Accuracy
1	Adam	Adaptive	(50,100,50)	0.0500	tanh	0.987
2	Adam	Constant	(50,50,50)	0.0001	relu	0.987
3	Sgd	Constant	(50,50,50)	0.0500	tanh	0.979
4	Adam	Adaptive	(100)	0.0500	tanh	0.989
5	Sgd	Adaptive	(50,50,50)	0.0500	tanh	0.989
6	Adam	Adaptive	(100)	0.0001	relu	0.989
7	Adam	Adaptive	(100)	0.0001	Tanh	0.989
8	Sgd	Constant	(100)	0.0001	relu	0.987
9	Sgd	Constant	(100)	0.0500	Tanh	0.989
10	Adam	Adaptive	(100)	0.0500	relu	0.987

Comment on MLP:

The model configurations incorporated two activation functions: 'tanh' and 'relu'. Both generally performed well with all accuracy rates exceeding 0.95. The 'relu' or rectified linear unit is typically suitable for Multilayer Perceptrons (MLPs) as it avoids saturation and reduces the likelihood of the vanishing gradient problem. The 'tanh' function also performed well, although it can saturate, particularly in deep networks or with significant input values.

The configurations used three varying hidden layer sizes: (50,50,50), (50,100,50), and (100). All showed good performance, indicating the problem may be solvable with diverse network architectures. As the complexity of the problem increases, it may require more layers and nodes. However, a more complex model may also lead to overfitting, emphasizing the importance of an optimal balance.

For optimization, the model configurations employed 'sgd' (Stochastic Gradient Descent) and 'adam' (Adaptive Moment Estimation) as solvers.

Both 'adaptive' and 'constant' learning rates were implemented. For 'sgd', an adaptive learning rate could be beneficial as it decreases when the model stops improving. Conversely, for 'adam', a constant learning rate is typically adequate.

The L2 regularization term, represented by the alpha parameter, helps in averting overfitting by limiting the weights' size. Both values, 0.0001 and 0.0500, showed solid performance, suggesting a balance between model complexity and regularization.

All combinations of parameters resulted in high accuracy rates, ranging from 0.979 to 0.989. This indicates that the model is resilient to changes in hyperparameters and performs well across various configurations.

The best trials are at index 4,5,6,7 and 9.

SVM:

Index	C	Gamma	Kernel	Classification Accuracy
1	0.1	1	Linear	0.982
2	0.1	1	Rbf	0.55
3	0.1	0.1	Linear	0.982
4	0.1	0.1	Rbf	0.55
5	0.1	0.01	Linear	0.982
6	0.1	0.01	Rbf	0.568
7	1	1	Linear	0.982
8	1	1	Rbf	0.55
9	1	0.1	Linear	0.982
10	1	0.1	Rbf	0.555
11	1	0.01	Linear	0.982
12	1	0.01	Rbf	0.795
13	10	1	Linear	0.982
14	10	1	Rbf	0.55
15	10	0.1	Linear	0.982
16	10	0.1	Rbf	0.565
17	10	0.01	Linear	0.982
18	10	0.01	rbf	0.800

Comment on SVM:

The model utilized two types of kernels: 'Linear' and 'RBF' (Radial Basis Function). The linear kernel is typically applied when the data is linearly separable, implying that it can be divided by a line (2D), a plane (3D), or a hyperplane (for higher dimensions). The RBF kernel can deal with

non-linear separable data by projecting it into a higher-dimensional space where it can be linearly separated.
The regularization parameter 'C' determines the strength of the regularization. A smaller 'C' creates a larger margin but more classification errors, while a larger 'C' results in a smaller margin and fewer classification errors. The optimal result, with an accuracy of 0.982, appears to occur when 'C' equals 1, suggesting an appropriate balance between margin size and classification errors.

The 'gamma' parameter is specific to the RBF kernel and essentially controls the spread of the kernel and the decision region. A higher 'gamma' restricts the influence of a single training example, implying the decision boundary relies on fewer support vectors, potentially leading to overfitting. Conversely, lower 'gamma' values extend each training example's influence, resulting in smoother decision boundaries. The results seem to indicate that a higher 'gamma' (1 or 0.1) offers better performance. However, a significantly reduced performance occurs when 'gamma' is 0.01, indicating potential underfitting.

Classification accuracy, showing how accurately the SVM model classifies the data with a particular parameter set, was used as a key metric. The highest accuracy of 0.982 was achieved with a 'C' of 1, a 'gamma' of 1 or 0.1, and either a Linear or RBF kernel. The lowest accuracy of 0.55 occurred with 'gamma' of 0.01, 'C' of 0.1, and an RBF kernel, suggesting potential underfitting.
The best trials at index 17, 15, 13, 11, 9,7,5 and 3.

Comparison of SVM and MLP:

Multilayer Perceptron (MLP) algorithm generally achieved higher classification accuracy compared to the Support Vector Machine (SVM). It's important to note, though, that the comparison is not entirely fair since the optimization and complexity of these two models are quite different.

MLP models, or neural networks, are flexible and can capture more complex relationships in the data by adjusting the number of layers and nodes in each layer. They can also employ different types of activation functions, which adds to their flexibility. In this case, the MLPs achieved high accuracy (around 98-99%) with both `tanh` and `relu` activation functions, regardless of whether a constant or adaptive learning rate was used.

On the other hand, SVMs rely on the concept of maximizing the margin between classes, which can be less flexible than neural networks. The SVMs in your experiment primarily used `Linear` and `Rbf` (Radial basis function) kernels. The results indicate that the SVM with the `Linear` kernel consistently achieved an accuracy of 98.2% irrespective of the `C` (regularization parameter) and `Gamma` (kernel coefficient for 'Rbf', 'Poly' and 'Sigmoid'). However, when the `Rbf` kernel was used, the accuracy dropped significantly, indicating that the `Rbf` kernel may not be suitable for your binary classification task.

The accuracy drop in SVM with the `Rbf` kernel could be due to several reasons:

1. Non-optimal Hyperparameters: The `C` and `Gamma` parameters used for the `Rbf` kernel might not be optimal for this dataset. `C` controls the trade-off between achieving a low error on the training data and maximizing the decision boundary's margin. On the other hand, `Gamma` determines the influence of a single training example, with low values meaning 'far' and high values meaning 'close'.

2. Feature Scaling: SVMs are not scale-invariant, so if the features in your dataset are not properly scaled, the SVM may perform poorly.

3. Outliers: SVMs can be sensitive to outliers. If your data contains outliers, it could negatively impact the SVM's performance.

4. Non-linear Separability: The `Rbf` kernel is used for non-linear separable data. If your data is linearly separable or the non-linear separability is too complex for the `Rbf` kernel to capture, this could explain the lower accuracy.

Comment on success of fall detectors:

The use of features such as velocity, acceleration, temperature, and pressure in wearable sensor-based systems for fall detection can prove to be highly effective. These features can capture nuanced details of a person's movements and their environment that are highly indicative of falls. For instance, velocity and acceleration can identify sudden changes in movement, which are typically associated with falls. Temperature can help to account for variations in user activity due to environmental conditions, and pressure sensors can provide additional context about user interactions with their environment.

Using machine learning algorithms such as Support Vector Machines (SVM) and Multi-Layer Perceptrons (MLP) to analyze these features can result in successful fall detection systems except SVM with rbf. Both SVM and MLP are capable of modeling complex relationships and patterns in high-dimensional data, making them well-suited to the task of fall detection.

SVMs, in particular, are known for their robustness and their ability to handle high-dimensional input spaces, making them suitable for datasets with many features, such as the ones from wearable sensors. On the other hand, MLPs, being a type of neural network, have the advantage of being able to approximate any function given enough layers and neurons. This allows them to learn complex, non-linear relationships between features, which can be particularly useful for fall detection, where the relationship between sensor readings and fall events can be quite complex.

The text of code:

```
# In[1]:
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from sklearn.metrics import confusion_matrix, accuracy_score, adjusted_rand_score
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
import seaborn as sns

# In[2]:
data = pd.read_csv("falldetection_dataset.csv",header = None)

# In[3]:
columns_to_drop = [0,1]
data_train = data.drop(columns=columns_to_drop)
data_labels = data[1]

# # PART A)
# In[4]:
data_labels
data_train = StandardScaler().fit_transform(data_train)
pca = PCA(n_components=2)
principalComponents = pca.fit_transform(data_train)
print(f'Explained variance by first 2 PCs: {pca.explained_variance_ratio_}')
# columns_to_drop = [0,1]
# data_train = data.drop(columns=columns_to_drop)
# data_labels = data[1]

# In[5]:
columns_to_drop = [0, 1]
data_train = data.drop(columns=columns_to_drop)
data_labels = data[1]
# Create a new figure and a subplot
fig = plt.figure(figsize=(8, 8))
ax1 = fig.add_subplot(1, 1, 1)
ax1.set_xlabel('Principal Component 1', fontsize=15)
ax1.set_ylabel('Principal Component 2', fontsize=15)
ax1.set_title('2 Component PCA', fontsize=20)
# Get unique labels
unique_labels = data_labels.unique()
# Scatter plot for each label
for label in unique_labels:
    df_label = data_train[data_labels == label]
    ax1.scatter(df_label.iloc[:, 0], df_label.iloc[:, 1], label=label)
ax1.grid()
# Add legend
ax1.legend()
plt.savefig(f'PCAplotu.png')
# Show the plot
plt.show()

# ## CLUSTER:
# In[6]:
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
plt.figure(figsize=(20, 40))
for i in range(2, 7):
    plt.figure()  # Create a new figure for each plot
    kmeans = KMeans(n_clusters=i, random_state=0)
    clusters = kmeans.fit_predict(principalComponents)
    plt.scatter(principalComponents[:, 0], principalComponents[:, 1], c=clusters, cmap='viridis')
    if i > 1:
        centers = kmeans.cluster_centers_
        plt.scatter(centers[:, 0], centers[:, 1], c='red', s=300, alpha=0.7);
    plt.xlabel('First Principal Component')
    plt.ylabel('Second Principal Component')
    plt.title('PCA Projection and K-Means Clustering, n_clusters=' + str(i))
    plt.tight_layout()
    plt.savefig(f'cluster_plot{i}.png')
    plt.clf()  # Clear the current figure
plt.show()
```

```
# In[7]:
import numpy as np
from sklearn.cluster import KMeans
from sklearn.metrics import accuracy_score, adjusted_rand_score, completeness_score
from sklearn.preprocessing import LabelEncoder
import seaborn as sns
import matplotlib.pyplot as plt
z = 4
kmeans = KMeans(n_clusters=z, random_state=0).fit(principalComponents)
encoded_labels = LabelEncoder().fit_transform(data_labels)
cluster_labels = kmeans.labels_
# Calculate accuracy
accuracy = accuracy_score(encoded_labels, cluster_labels)
# Calculate consistency (adjusted Rand index)
consistency = adjusted_rand_score(encoded_labels, cluster_labels)
print(f"Accuracy: {accuracy}")
print(f"Consistency (Adjusted Rand Index): {consistency}")
# Confusion Matrix
cm = confusion_matrix(encoded_labels, cluster_labels)
plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, cmap='Blues', fmt='g')
plt.xlabel('Predicted Labels')
plt.ylabel('True Labels')
plt.title('Confusion Matrix')
plt.savefig(f'correlation_matrix{z}.png')
plt.show()

# # Elbow Point:
# In[8]:
X = data_train
# Calculate WCSS for different number of clusters
wcss = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10, random_state=0)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)
# Find the elbow point
differences = pd.Series(wcss[:-1]) - pd.Series(wcss[1:])
elbow_index = differences.idxmax() + 1
elbow_point = range(1, 11)[elbow_index]
# Create a larger figure
plt.figure(figsize=(10, 6))
# Plot the Elbow graph with gridlines and markers
plt.plot(range(1, 11), wcss, marker='o')
plt.grid(True)
plt.title('Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('WCSS')
# Highlight the elbow point
plt.scatter(elbow_point, wcss[elbow_index], color='r', s=100, label='Elbow Point')
plt.legend()
# Show the plot
plt.savefig(f'elbow.png')
plt.show()
print("Elbow Point:", elbow_point)

# # PART B)
# In[9]:
X_train_val, X_test, y_train_val, y_test = train_test_split(data_train, data_labels, test_size=0.15, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.17647, random_state=42)

# # READ OF RESULT_CSV:
# In[10]:
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split, GridSearchCV, RandomizedSearchCV
from sklearn.metrics import accuracy_score
from sklearn.datasets import load_iris
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
# Define the parameter grid for SVM
param_grid_svm = {'C': [0.1, 1, 10], 'gamma': [1, 0.1, 0.01], 'kernel': ['linear', 'rbf']}
# Define the parameter grid for MLP
param_grid_mlp = {
    'hidden_layer_sizes': [(50,50,50), (50,100,50), (100,)],
    'activation': ['tanh', 'relu'],
    'solver': ['sgd', 'adam'],
    'alpha': [0.0001, 0.05],
    'learning_rate': ['constant','adaptive'],
}
```

```
print("Starting SVM model training...")
svm_model = GridSearchCV(SVC(), param_grid_svm, cv=5)
svm_model.fit(X_train, y_train)
print("SVM model training complete.")
# Add classification accuracy to df_svm
df_svm = pd.DataFrame(svm_model.cv_results_['params'])
df_svm['classification_accuracy'] = svm_model.cv_results_['mean_test_score']
df_svm.to_csv('df_svm.csv', index=False)
print("Starting MLP model training...")
mlp_model = RandomizedSearchCV(MLPClassifier(), param_grid_mlp, n_iter=10, cv=5)
mlp_model.fit(X_train, y_train)
print("MLP model training complete.")
# Add classification accuracy to df_mlp
df_mlp = pd.DataFrame(mlp_model.cv_results_['params'])
df_mlp['classification_accuracy'] = mlp_model.cv_results_['mean_test_score']
df_mlp.to_csv('df_mlp.csv', index=False)

# In[11]:
mlp_results = pd.read_csv("df_mlp.csv")
mlp_results

# In[12]:
svm_results = pd.read_csv("df_svm.csv")
svm_results
```