GE-461

Introduction to Data Science
Spring 2024

Prepared by
Bahadır Yüzlü

April 30, 2024

1- Dataset Information and Label Adjustment

This project analyzes wearable sensor data collected from subjects performing various motor actions. The analysis focuses on distinguishing between fall (F) and non-fall (NF) activities. The dataset, stored in “falldetection_dataset.csv”, contains 566 samples. Each sample corresponds to a specific motor action and is characterized by 306 sensor-derived features, including velocity, acceleration, temperature, and pressure. To facilitate analysis and classification tasks, the original action labels were modified. The label 'F' was changed to '1' to denote a fall action, and 'NF' was changed to '0' to indicate a non-fall action.

2- PCA and K-Means Exploration

Exploratory data analysis involved reducing the 306 sensor features to two principal components (PCs) using PCA. The visualization of the PCA algorithm can be seen in Appendix B. The variance explained by PC1 and PC2 was 0.23 and 0.17, respectively. Two scaling methods, standard scaling and min-max scaling, were compared, with standard scaling (81.27%) showing better performance in terms of accuracy than min-max scaling (34.45%).

As it can be seen in the Appendix C, Different numbers of clusters (N=2 to N=6) were evaluated using K-means clustering. The optimal configuration was found at N=3, where the average silhouette score was 0.65 (indicating good clustering quality) and the average Davies-Bouldin index was 0.51 (indicating well-separated clusters).

For fall detection analysis, the clusters obtained using N=2 were examined, aligning with the two original classes in the dataset (fall and non-fall). An overlap analysis revealed an 81.27% consistency between the cluster memberships and the original fall detection labels. This high degree of alignment suggests that the clustering algorithm effectively distinguishes between fall and non-fall actions based on sensor data.

In conclusion, the results indicate promising potential for fall detection based on sensor measurements, with the clustering approach providing meaningful insights and classification of different types of activities.

3- Supervised Learning Comparison (SVM and MLP)

Data Preprocessing and Splitting

In the supervised learning stage of the project, the objective was to develop classifiers capable of accurately detecting fall actions versus non-fall actions using wearable sensor data. Two distinct models, a Support Vector Machine (SVM) classifier and a Multi-layer Perceptron (MLP) classifier, were evaluated for their performance in this task. The data was divided into separate training (60%), validation (20%), and testing (20%) sets. The training set was used to train the models. The validation set was used to fine-tune the models during the training process. Finally, the testing set was used to assess the performance of the models on unseen data. To ensure the sets were independent, a non-overlapping split was employed, meaning no data point appeared in more than one set. Additionally, standard scaling was applied to normalize the data, putting all features on a similar scale for better modeling.

Hyperparameter Tuning Methodology

A comprehensive 5-fold cross-validation strategy was employed during the parameter tuning phase. This involved testing different combinations of hyperparameters within each model and evaluating their performance based on the validation set. Grid search techniques were utilized to systematically explore the parameter space and identify optimal configurations that maximize classification accuracy. Intermediate results from each model configuration were recorded and analyzed to understand the impact of hyperparameter choices on model performance. A grid search approach was used to explore different hyperparameter settings for two machine learning models: Support Vector Machine (SVM) and Multi-layer Perceptron (MLP). Hyperparameters are essentially settings that control how the model learns from the data. By trying out various combinations of hyperparameters, the aim was to identify the configuration that yielded the best possible performance.

Tuning the SVM Model

Three key hyperparameters were focused on for the SVM model:

- **C:** This parameter controls the penalty for misclassifying data points. Lower values encourage simpler models, while higher values allow for more complex decision boundaries.
- **Gamma:** This parameter determines the influence of individual training examples on the decision boundary. Higher gamma values make the model more sensitive to specific data points.
- **Kernel:** This parameter defines the type of mathematical function used to map the data into a higher-dimensional space. Three kernel options were explored: linear, radial basis function (RBF), and sigmoid.

The findings for the SVM model were as follows:

- The performance of the RBF kernel was highly dependent on the values chosen for gamma and C. The most accurate configuration within the SVM models (0.9978) used a lower gamma and a higher C value.
- The linear kernel exhibited consistent performance across most configurations, with an accuracy of around 0.9956. Interestingly, the gamma and C parameters seemed to have less impact on the linear kernel's performance.

The validation results can be seen from Appendix D.

Tuning the MLP Model

Four hyperparameters were examined for the MLP model:

- **Hidden layer sizes:** This defines the number of neurons in each hidden layer of the neural network.
- **Activation function:** This function introduces non-linearity into the network, allowing it to capture more intricate relationships in the data. Two common activation functions were explored: ReLU and tanh.
- **Alpha:** This parameter controls the amount of regularization applied during training. Regularization helps prevent the model from overfitting to the training data.

- **Solver:** This parameter specifies the optimization algorithm used to train the neural network. Different solvers can affect how efficiently the model learns.

Here's what was discovered about the MLP model:

- All MLP configurations achieved accuracy above 0.995, moreover, some had an accuracy of 1 indicating a very stable model overall.
- There weren't any significant variations in performance observed across different hyperparameter settings. This characteristic suggests that the MLP model might be less sensitive to specific hyperparameter choices.

The results can be seen from Appendix D.

Comparing the Models and Overall Success

The best performing SVM model, utilizing a linear kernel and optimized hyperparameters, achieved an impressive accuracy of 0.982456 on the test set. Similarly, the best MLP model exhibited strong performance with an accuracy of 0.991228. It's noteworthy that both SVM and MLP classifiers surpassed the performance of a previously explored k-means clustering approach, highlighting the effectiveness of supervised learning methods tailored for fall detection tasks.

Further insights can be gathered from additional metrics beyond accuracy:

- **Precision (weighted average):** The best SVM model achieved a precision of 0.983013, indicating a high proportion of correctly identified fall and non-fall actions weighted by class support. The best MLP model demonstrated even higher precision at 0.991370.
- **Recall (weighted average):** Both models exhibited excellent recall scores, with the SVM model at 0.982456 and the MLP model at 0.991228. These scores indicate the ability of the models to correctly identify a high percentage of actual fall and non-fall actions.
- **F1-score (weighted average):** The F1-score, which balances precision and recall, was consistently high for both SVM (0.982429) and MLP (0.991222) models. This metric reflects overall model effectiveness in accurately classifying fall actions based on wearable sensor data.

In conclusion, the comparative analysis of SVM and MLP classifiers underscores their efficacy in fall detection, with the MLP model slightly outperforming SVM in terms of accuracy and precision. Both models demonstrated robust performance across multiple metrics, demonstrating their suitability for real-world applications requiring reliable and accurate identification of fall events using wearable sensors.

Remark

The provided code below needs to be adjusted before running because it is written for Google Colab. To run it, you'll need to make a path adjustment. Alternatively, you can use the following Google Colab URL to access a document that has already been run: https://colab.research.google.com/drive/1pU2bnmbO50CifS4Up019sh6VvbGpg2bu . If any changes need to be made, those who click the link will have the necessary authorization.

Appendix

A) Code Snippet

```
"""Project_4.ipynb

#Modules
"""

import numpy as np
import matplotlib.pyplot as plt
import os
from google.colab import drive
import pandas as pd
import zipfile
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.cluster import KMeans
from sklearn.metrics import silhouette_score, davies_bouldin_score
from sklearn.metrics import accuracy_score, classification_report
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier
from tabulate import tabulate

"""#Data Extraction"""

# Authenticate and mount Google Drive
drive.mount('/content/drive')

!ls "/content/drive/MyDrive/GE-461/Project 4/ge461_pw13_data.zip"

# Path to zip file inside Google Drive
zip_path = '/content/drive/MyDrive/GE-461/Project 4/ge461_pw13_data.zip'

# Specify the name of the CSV file inside the zip file
csv_filename = 'falldetection_dataset.csv'

# Extract the CSV file from the zip archive
with zipfile.ZipFile(zip_path, 'r') as zip_ref:
    zip_ref.extract(csv_filename, '/content/')

# Load the CSV file into a DataFrame
df = pd.read_csv(f'/content/{csv_filename}', header=None)

# Separate features (X) and labels (y)
X = df.iloc[:, 2:]
y = df.iloc[:, 1]

# Display the shape of X and y
print("Shape of X:", X.shape)
```

```
print("Shape of y:", y.shape)

# Display the first few rows of X and y to verify
print("\nFeatures (X):")
print(X.head())

print("\nLabels (y):")
print(y.head())

# Update y to numeric values: "F" (Fall) -> 1, "NF" (Non-Fall) -> 0
y = (y == 'F').astype(int)

"""#Part A

##PCA
"""

# Standardize the features
#scaler = MinMaxScaler()
#X_scaled = scaler.fit_transform(X)

scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)


# Perform PCA to extract top two principal components
pca = PCA(n_components=2)
X_pca = pca.fit_transform(X_scaled)

# Calculate the variance explained by each principal component
explained_variance_ratio = pca.explained_variance_ratio_
print("Variance explained by PC1 and PC2:", explained_variance_ratio)

# Visualize the PCA results with categorical labels (F: Fall, NF: Non-Fall)
plt.figure(figsize=(8, 6))
plt.scatter(X_pca[y == 1, 0], X_pca[y == 1, 1], label='Fall', color='red', alpha=0.5)
plt.scatter(X_pca[y == 0, 0], X_pca[y == 0, 1], label='Non-Fall', color='blue', alpha=0.5)
plt.title('PCA: Top Two Principal Components (Categorical Labels)')
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.legend()
plt.show()

"""##K-Means"""

# Try different numbers of clusters (N)
num_clusters = [2, 3, 4, 5, 6]

# Visualize clustering results for each number of clusters (N)
```

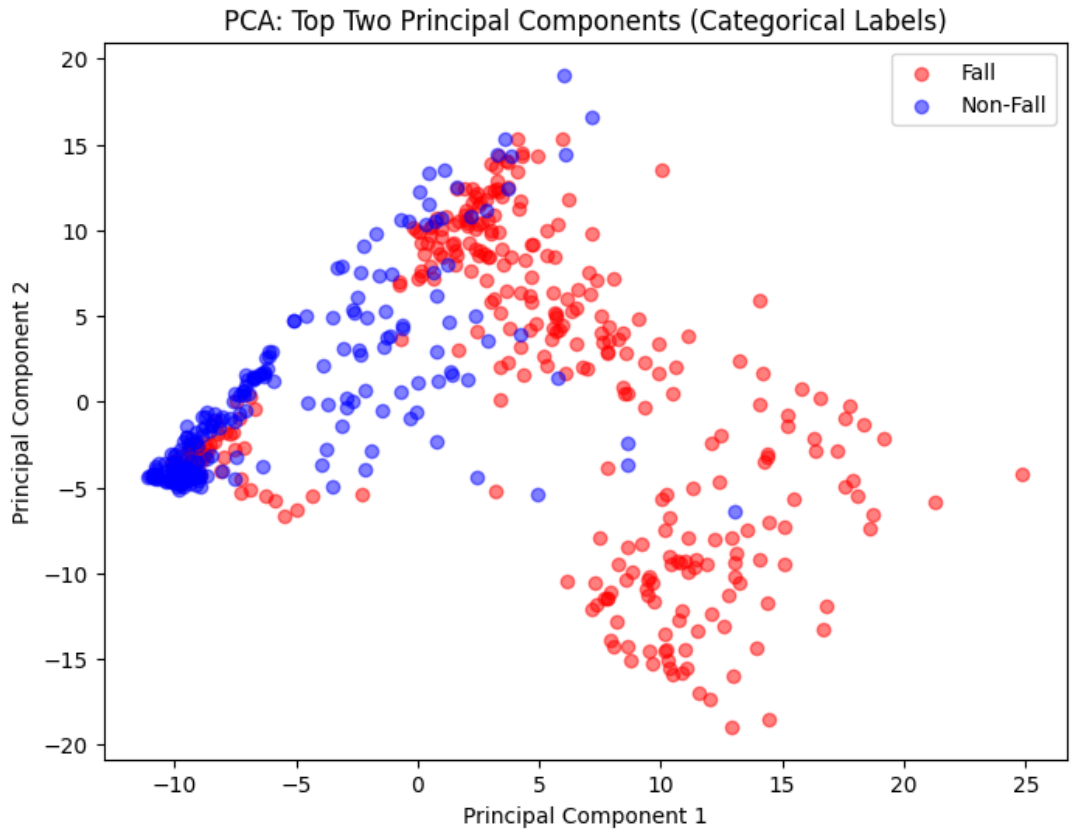
```
plt.figure(figsize=(15, 8))

for i, n in enumerate(num_clusters, start=1):
    # Initialize k-means clustering with n clusters
    kmeans = KMeans(n_clusters=n, n_init=10, random_state=42)
    cluster_labels = kmeans.fit_predict(X_pca)
    # Calculate silhouette score
    silhouette_avg = silhouette_score(X_pca, cluster_labels)
    # Calculate Davies-Bouldin Index
    davies_bouldin_avg = davies_bouldin_score(X_pca, cluster_labels)
    # Plot clusters in a subplot
    plt.subplot(2, 3, i)
    plt.scatter(X_pca[:, 0], X_pca[:, 1], c=cluster_labels, cmap='viridis', alpha=0.5)
    plt.title(f'Clusters (N={n})\nSilhouette Score: {silhouette_avg:.2f}, Davies-Bouldin Index: {davies_bouldin_avg:.2f}')
    plt.xlabel('Principal Component 1')
    plt.ylabel('Principal Component 2')
    plt.colorbar(label='Cluster')

plt.tight_layout()
plt.show()

# Initialize k-means clustering with N=2 clusters
kmeans_2 = KMeans(n_clusters=2, n_init=10, random_state=42)
cluster_labels_2 = kmeans_2.fit_predict(X_pca)

# Adjust cluster labels to match the encoding of original action labels
cluster_labels_adjusted = 1 - cluster_labels_2  # Flip labels (0 -> 1, 1 -> 0)

# Calculate accuracy (percentage overlap) between cluster memberships and original labels
accuracy_2 = accuracy_score(y, cluster_labels_adjusted)

# Calculate percentage of correct classifications
percentage_correct = accuracy_2 * 100

print(f"Percentage of Correct Classifications (N=2 Clusters): {percentage_correct:.2f}%")

"""#Part B"""

# Split the data into training (60%), validation (20%), and testing (20%) sets
X_train_val, X_test, y_train_val, y_test = train_test_split(X, y, test_size=0.20, random_state=42)
X_train, X_val, y_train, y_val = train_test_split(X_train_val, y_train_val, test_size=0.25, random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
```

```
X_val_scaled = scaler.transform(X_val)
X_test_scaled = scaler.transform(X_test)

# Define parameter grids for SVM and MLP classifiers
param_grid_svm = {'C': [0.2, 2, 20], 'gamma': [0.1,0.01, 0.001], 'kernel': ['linear', 'rbf','sigmoid'], 'class_weight':['none']}
param_grid_mlp = {'hidden_layer_sizes': [(25,), (50,), (25, 25)], 'activation': ['relu', 'tanh'], 'alpha': [0.001, 0.01,0.1], 'solver':['adam', 'sgd']}

# Initialize StratifiedKFold for 5-fold cross-validation
skf = StratifiedKFold(n_splits=5, shuffle=True, random_state=42)

# Intermediate Results Storage
svm_results = []
mlp_results = []

# Perform grid search and cross-validation for SVM classifier
best_svm_clf = None
best_svm_accuracy = 0
for C in param_grid_svm['C']:
    for gamma in param_grid_svm['gamma']:
        for kernel in param_grid_svm['kernel']:
            avg_val_accuracy = 0
            for train_index, val_index in skf.split(X_train_val, y_train_val):
                X_train_fold, X_val_fold = X_train_val.iloc[train_index], X_train_val.iloc[val_index]
                y_train_fold, y_val_fold = y_train_val.iloc[train_index], y_train_val.iloc[val_index]

                X_train_fold_scaled = scaler.fit_transform(X_train_fold)
                X_val_fold_scaled = scaler.transform(X_val_fold)

                svm_clf = SVC(C=C, gamma=gamma, kernel=kernel)
                svm_clf.fit(X_train_fold_scaled, y_train_fold)
                y_val_pred_svm = svm_clf.predict(X_val_fold_scaled)
                val_accuracy = accuracy_score(y_val_fold, y_val_pred_svm)
                avg_val_accuracy += val_accuracy / 5
            svm_results.append([C, gamma, kernel, avg_val_accuracy])
            if avg_val_accuracy > best_svm_accuracy:
                best_svm_accuracy = avg_val_accuracy
                best_svm_clf = SVC(C=C, gamma=gamma, kernel=kernel)

# Perform grid search and cross-validation for MLP classifier
best_mlp_clf = None
best_mlp_accuracy = 0

for hidden_layer_sizes in param_grid_mlp['hidden_layer_sizes']:
    for activation in param_grid_mlp['activation']:
        for alpha in param_grid_mlp['alpha']:
            avg_val_accuracy = 0
```

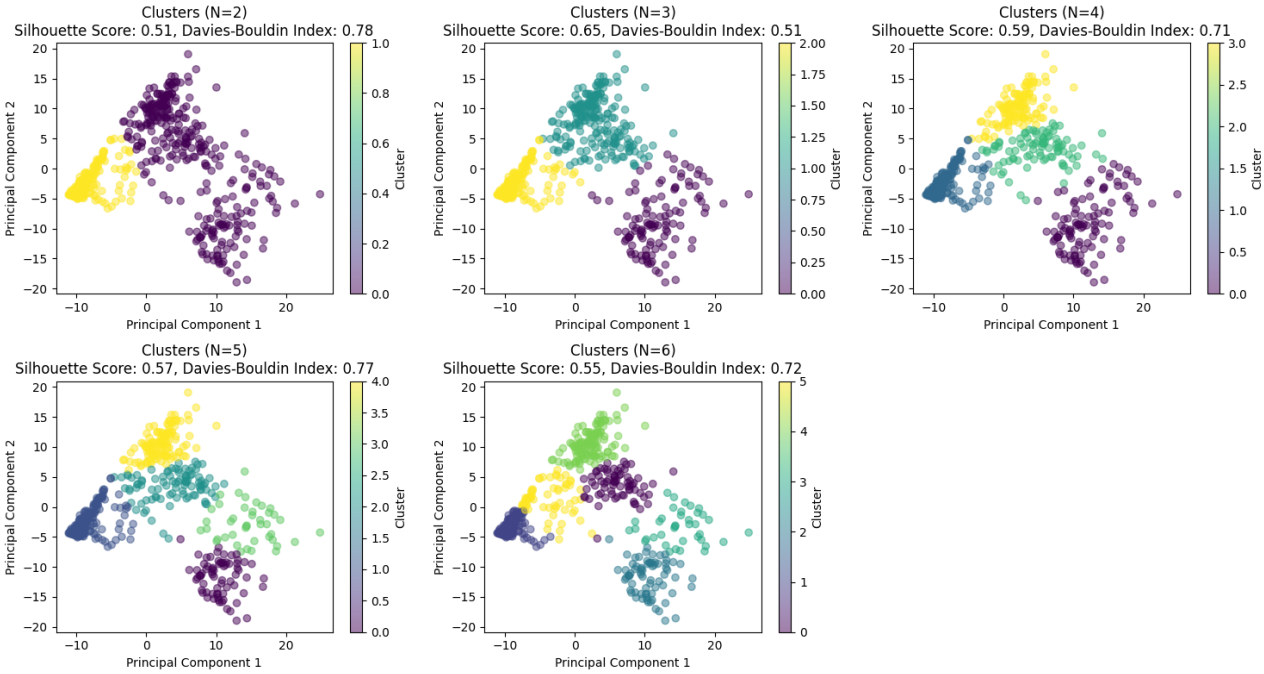
```
            for train_index, val_index in skf.split(X_train_val, y_train_val):
                X_train_fold, X_val_fold = X_train_val.iloc[train_index], X_train_val.iloc[val_index]
                y_train_fold, y_val_fold = y_train_val.iloc[train_index], y_train_val.iloc[val_index]

                X_train_fold_scaled = scaler.fit_transform(X_train_fold)
                X_val_fold_scaled = scaler.transform(X_val_fold)

                mlp_clf = MLPClassifier(hidden_layer_sizes=hidden_layer_sizes, activation=activation, alpha=alpha)
                mlp_clf.fit(X_train_fold_scaled, y_train_fold)
                y_val_pred_mlp = mlp_clf.predict(X_val_fold_scaled)
                val_accuracy = accuracy_score(y_val_fold, y_val_pred_mlp)
                avg_val_accuracy += val_accuracy / 5  # Average validation accuracy across folds
            mlp_results.append([hidden_layer_sizes, activation, alpha, avg_val_accuracy])
            if avg_val_accuracy > best_mlp_accuracy:
                best_mlp_accuracy = avg_val_accuracy
                best_mlp_clf = MLPClassifier(hidden_layer_sizes=hidden_layer_sizes, activation=activation, alpha=alpha)

# Display Results in a Table
print("\nSVM Classifier Results:")
headers_svm = ["C", "Gamma", "Kernel", "Avg. Validation Accuracy"]
print(tabulate(svm_results, headers=headers_svm, floatfmt=".4f"))

print("\nMLP Classifier Results:")
headers_mlp = ["Hidden Layer Sizes", "Activation", "Alpha", "Avg. Validation Accuracy"]
print(tabulate(mlp_results, headers=headers_mlp, floatfmt=".4f"))

# Initialize scaler for data normalization
scaler = StandardScaler()

# Evaluate the best SVM classifier on the testing set
X_train_scaled_svm = scaler.fit_transform(X_train)
best_svm_clf.fit(X_train_scaled_svm, y_train)
X_test_scaled_svm = scaler.transform(X_test)
y_test_pred_svm = best_svm_clf.predict(X_test_scaled_svm)
accuracy_test_svm = accuracy_score(y_test, y_test_pred_svm)
report_svm = classification_report(y_test, y_test_pred_svm, output_dict=True)

# Evaluate the best MLP classifier on the testing set
X_train_scaled_mlp = scaler.fit_transform(X_train)
best_mlp_clf.fit(X_train_scaled_mlp, y_train)
X_test_scaled_mlp = scaler.transform(X_test)
y_test_pred_mlp = best_mlp_clf.predict(X_test_scaled_mlp)
accuracy_test_mlp = accuracy_score(y_test, y_test_pred_mlp)
```

```
report_mlp = classification_report(y_test, y_test_pred_mlp, output_dict=True)

# Create DataFrame to summarize testing results
data = {
    'Classifier': ['Best SVM', 'Best MLP'],
    'Testing Accuracy': [accuracy_test_svm, accuracy_test_mlp],
    'Precision (weighted avg)': [report_svm['weighted avg']['precision'], report_mlp['weighted avg']['precision']],
    'Recall (weighted avg)': [report_svm['weighted avg']['recall'], report_mlp['weighted avg']['recall']],
    'F1-score (weighted avg)': [report_svm['weighted avg']['f1-score'], report_mlp['weighted avg']['f1-score']]
}

test_results_df = pd.DataFrame(data)
print("Testing Results:")
print(test_results_df)
```

B) PCA Visualization



C) K-Means Configurations



D) Hyperparameter Tuning Results

SVM Classifier Results:

C	Gamma	Kernel	Avg. Validation Accuracy
0.2000	0.1000	linear	0.9956
0.2000	0.1000	rbf	0.6106
0.2000	0.1000	sigmoid	0.7988
0.2000	0.0100	linear	0.9956
0.2000	0.0100	rbf	0.9138
0.2000	0.0100	sigmoid	0.9160
0.2000	0.0010	linear	0.9956
0.2000	0.0010	rbf	0.9425
0.2000	0.0010	sigmoid	0.9005
2.0000	0.1000	linear	0.9956
2.0000	0.1000	rbf	0.7722
2.0000	0.1000	sigmoid	0.7654
2.0000	0.0100	linear	0.9956
2.0000	0.0100	rbf	0.9867
2.0000	0.0100	sigmoid	0.8739
2.0000	0.0010	linear	0.9956
2.0000	0.0010	rbf	0.9956
2.0000	0.0010	sigmoid	0.9978
20.0000	0.1000	linear	0.9956
20.0000	0.1000	rbf	0.7722
20.0000	0.1000	sigmoid	0.7610
20.0000	0.0100	linear	0.9956
20.0000	0.0100	rbf	0.9867
20.0000	0.0100	sigmoid	0.8562
20.0000	0.0010	linear	0.9956
20.0000	0.0010	rbf	0.9978
20.0000	0.0010	sigmoid	0.9912

MLP Classifier Results:

Hidden Layer Sizes	Activation	Alpha	Avg. Validation Accuracy
(25,)	relu	0.0010	0.9978
(25,)	relu	0.0100	1.0000
(25,)	relu	0.1000	0.9978
(25,)	tanh	0.0010	0.9956
(25,)	tanh	0.0100	0.9978
(25,)	tanh	0.1000	1.0000
(50,)	relu	0.0010	0.9978
(50,)	relu	0.0100	0.9978
(50,)	relu	0.1000	0.9978
(50,)	tanh	0.0010	0.9978
(50,)	tanh	0.0100	0.9978
(50,)	tanh	0.1000	0.9978
(25, 25)	relu	0.0010	0.9956
(25, 25)	relu	0.0100	0.9978
(25, 25)	relu	0.1000	0.9978
(25, 25)	tanh	0.0010	0.9956
(25, 25)	tanh	0.0100	0.9956
(25, 25)	tanh	0.1000	0.9978

E) Test Results

Testing Results:

	Classifier	Testing Accuracy	Precision (weighted avg)	Recall (weighted avg)	F1-score (weighted avg)
0	Best SVM	0.982456	0.983013	0.982456	0.982429
1	Best MLP	0.991228	0.991370	0.991228	0.991222