Bilkent University

Department of Computer Engineering

GE 461 Introduction to Data Science

Spring 2024

Project 4

Fall Detection

Görkem Kadir Solun

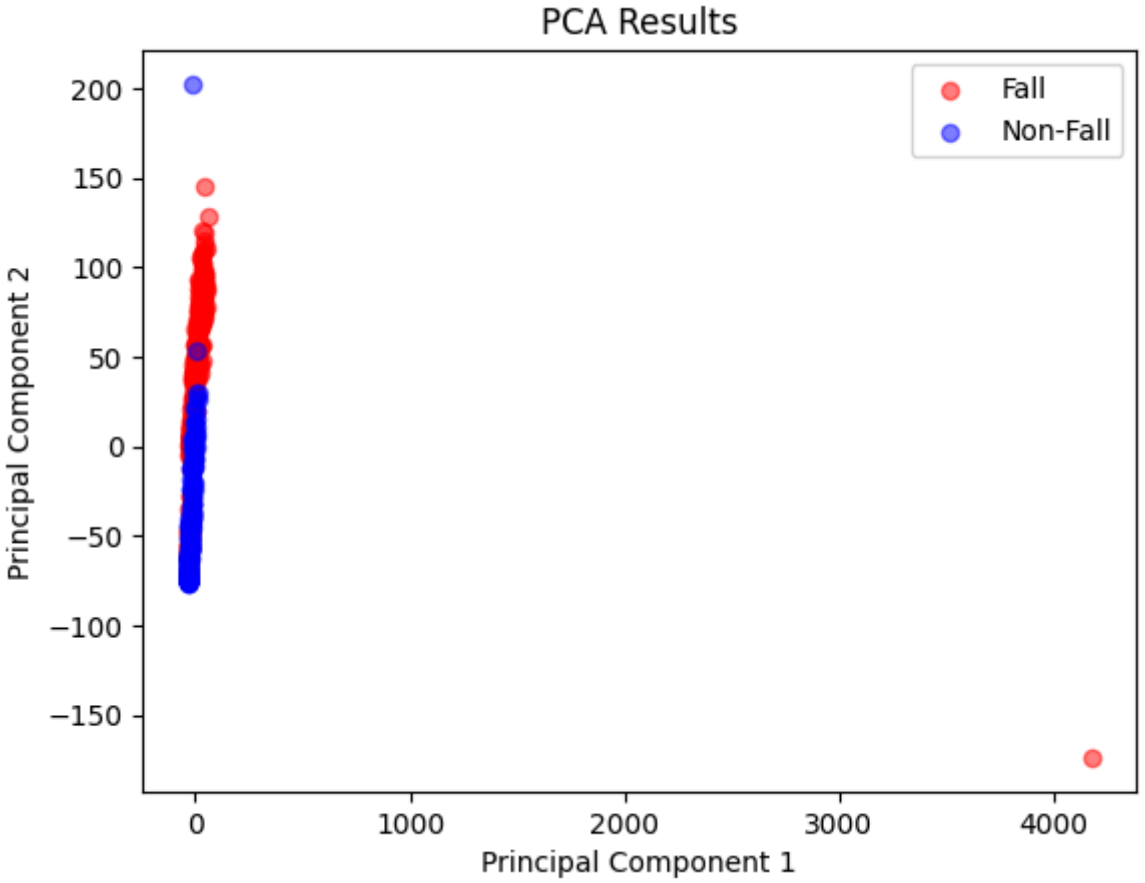
22003214

To ensure consistency in the results presented in this report, I have utilized a predefined random seed when running the tests. Please note that altering the random seed could slightly change the reported outcomes.

Part A

Getting the First PCA Result

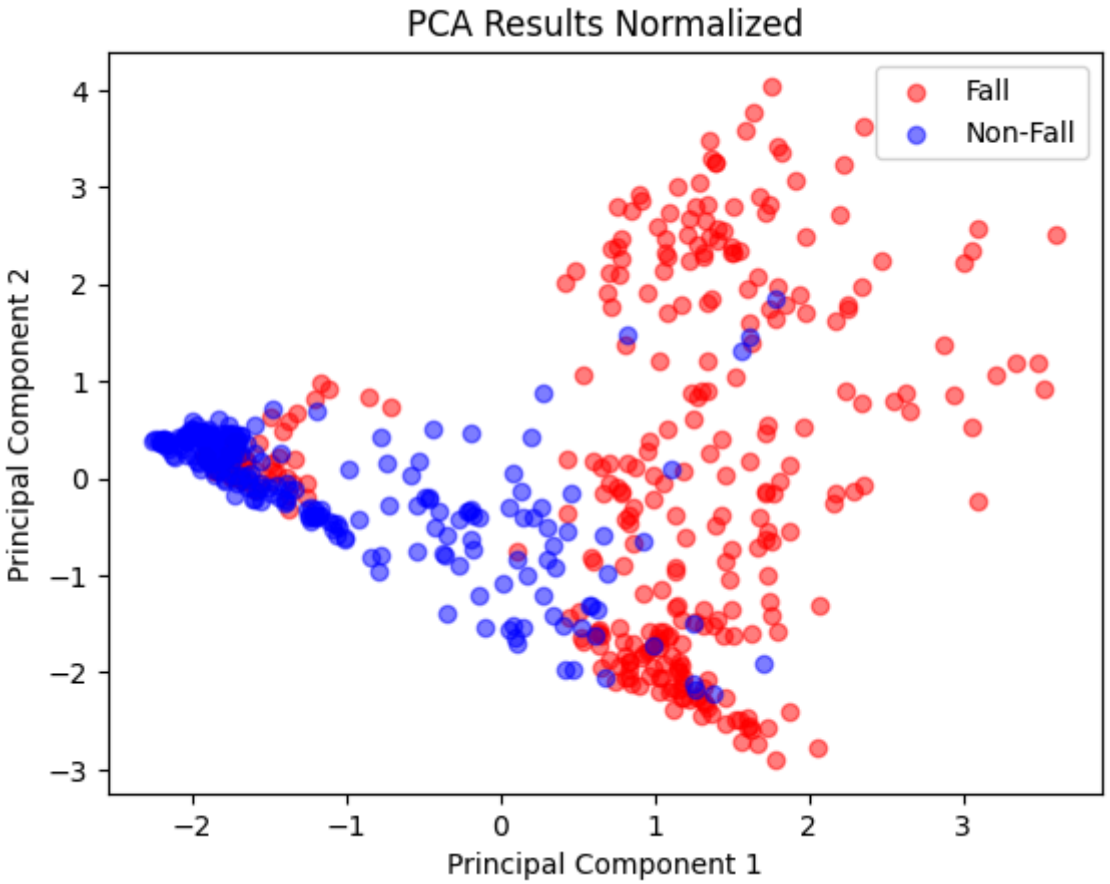
When we initially perform Principal Component Analysis (PCA) using two components on our dataset and then project the data, the resulting graph is as follows:



Applying Normalization

The analysis reveals two data points that are outliers. These outliers could skew the results, so excluding them from the dataset is essential. The variances of the first two principal components are 0.75307248 and 0.0851159, respectively. This indicates that the first principal component alone accounts for 75.3% of the variance. Combining the first two components accounts for 83.8% of the total variance, with the second component explaining 8.5%. The following steps will remove the identified outliers and apply min-max scaling to normalize the data. Since min-max scaling is sensitive to outliers, removing them beforehand ensures the effectiveness of this technique. Once these adjustments are made, the analysis

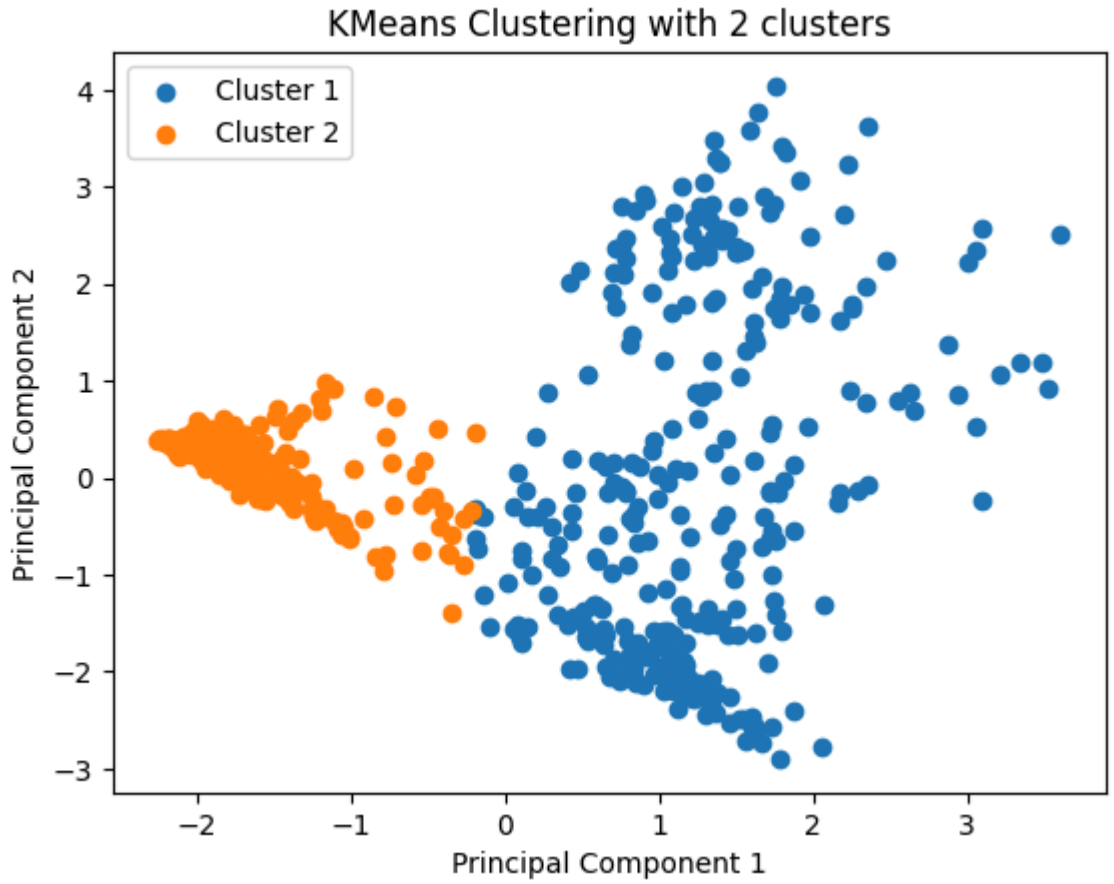
will be remade to obtain more accurate results. Following these procedures, the resulting graph is generated:



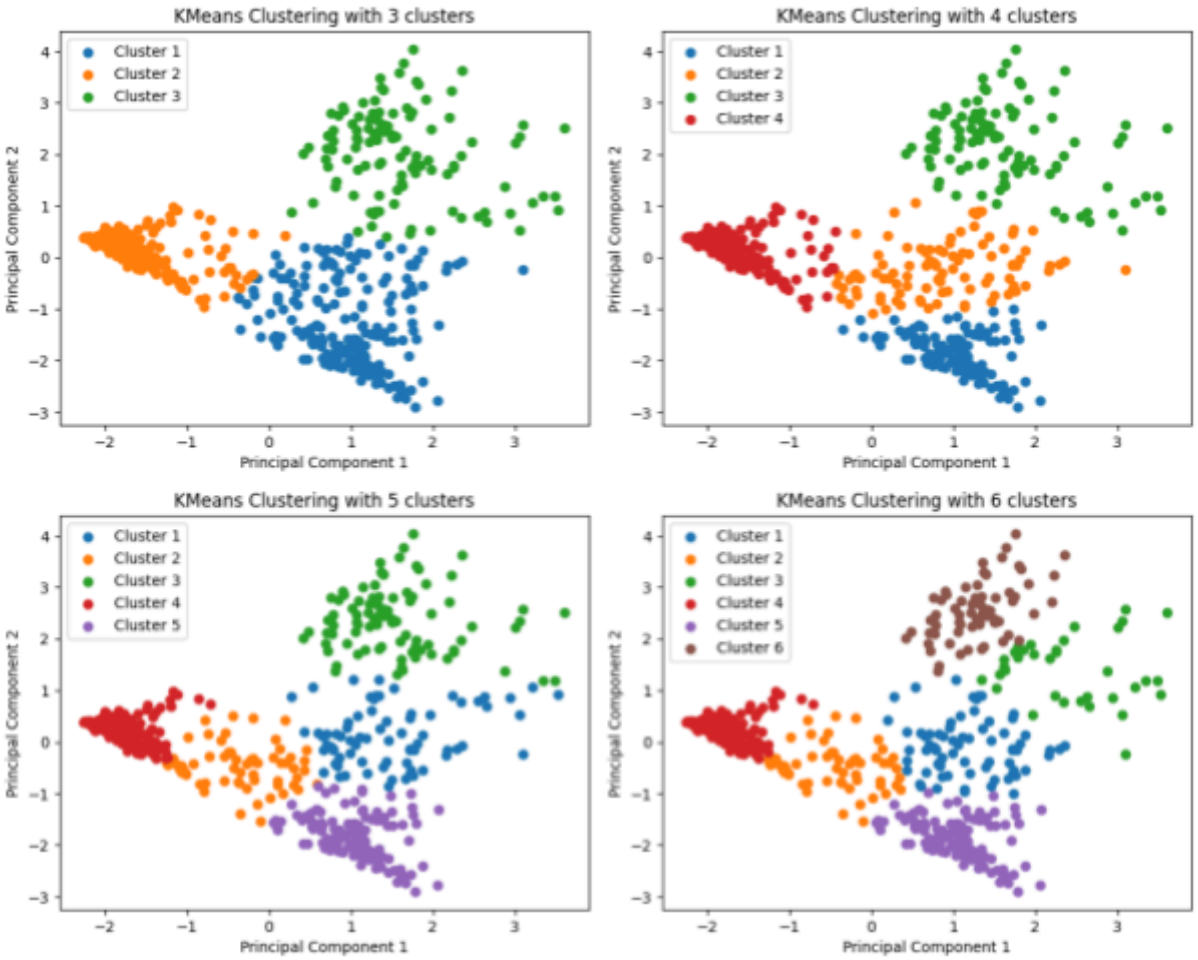
As observed, the results are now more interpretable, and the visualization more accurately reflects the data. After these adjustments, the variances explained by the first and second principal components are 26.6% and 22.0%, respectively, totaling a cumulative variance of 48.6%. Previously, the cumulative variance reached 83.8%, which appears unrealistic. Consequently, we will continue to utilize the normalized data with outliers removed for the remainder of our experiment.

Applying K-means to Normalized Data

K-means clustering was applied to the projected data. Initially, the data was clustered into two groups:



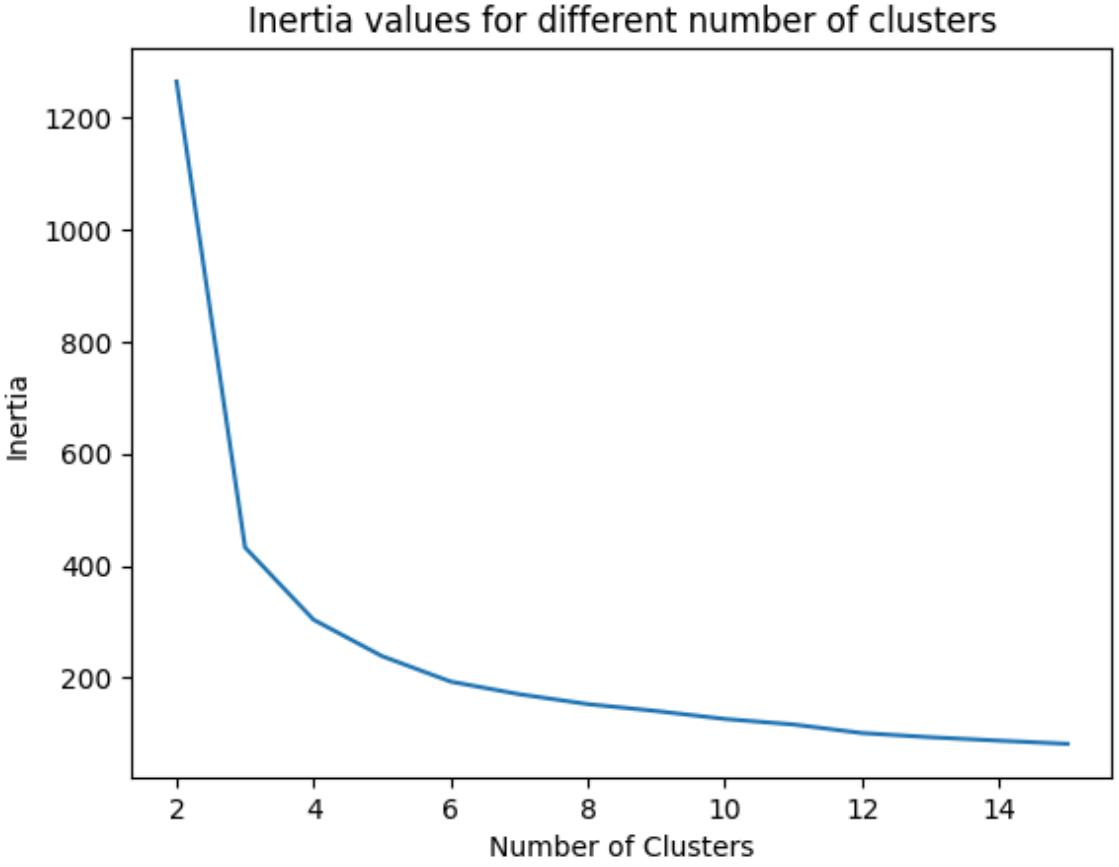
The accuracy and standard inertia metrics were applied to evaluate the clustering results. Since the cluster labels inherently lack meaning, comparing the assigned clusters with their actual values is necessary to assess their correspondence. This evaluation assumes that the labels can be optimally mapped to the ground truth to maximize accuracy. The figure illustrates two clusters: one can designate the blue data points as "Not Falling" or select the orange data points for the same label. Given that k-means clustering does not inherently assign meaningful labels, one can interpret the results as the configuration that achieves the highest accuracy, aiming to cluster most similar classes together. This method provides insights into the effectiveness of the cluster formation. For instance, applying this method to a k-means clustering with n = 2 yielded an accuracy of 81.91%. Comparing this with the normalized PCA results confirms that the clusters are visually cohesive. The 20% incorrect labeling likely arises from overlaps between "Falling" and "Not Falling" data points, presenting a challenging separation task. Nevertheless, the clustering method successfully distinguished 81.91% of the data points correctly, indicating the feasibility of fall detection. Exploring different clustering configurations might further enhance our results.

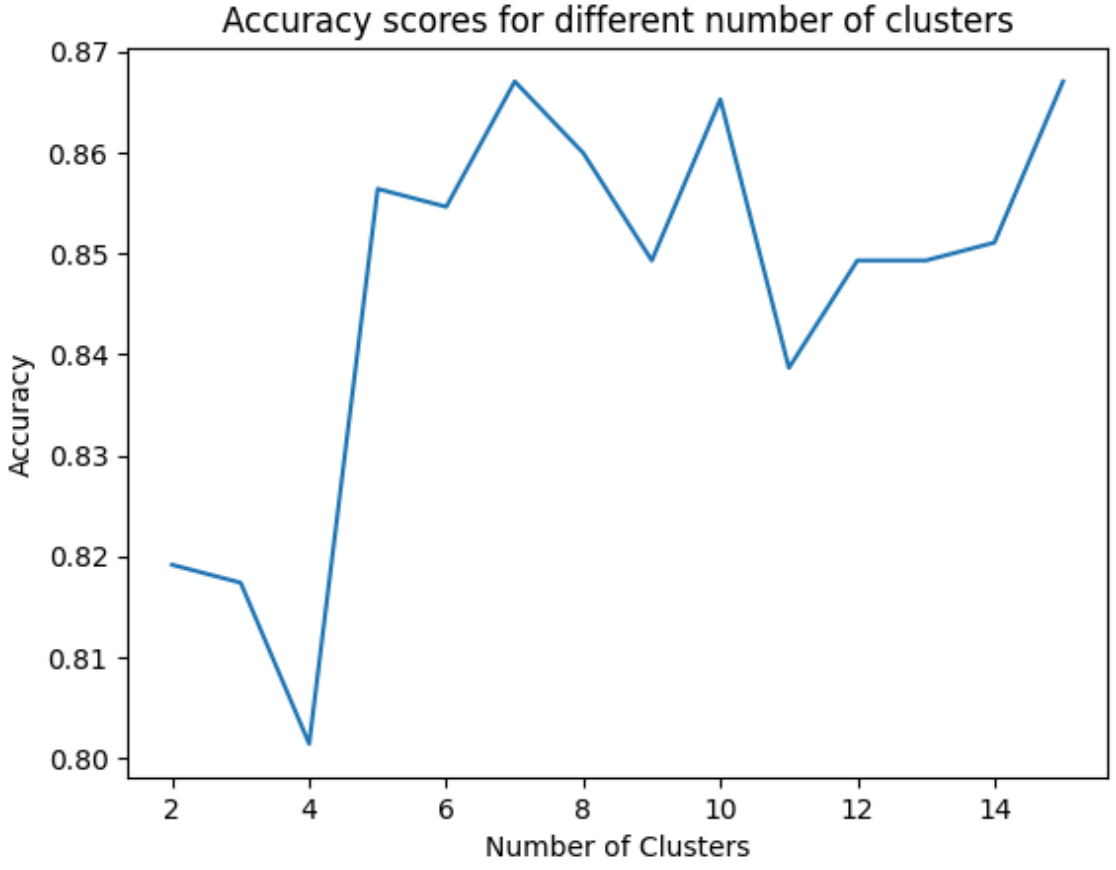


After examining the graphs and comparing them with previous findings, several observations can be made. Firstly, the accuracy of the clustering evaluation method significantly improves beyond a cluster count of five. A visual comparison between cluster counts of two and four reveals that the emergence of distinct red and orange regions accounts for this increased accuracy. Lower cluster counts failed to recognize this area as a separate entity. Despite some overlap between the ‘not falling’ and ‘falling’ data points, a distinct section predominantly composed of ‘falling’ data points is apparent in the middle region. Clusters greater than five continue to detect this specific area effectively. However, accuracy fluctuates beyond five clusters with minor fluctuations, suggesting that four clusters may represent the optimal count. Additionally, this observation underscores the separability of the data points, reinforcing the potential for accurate fall detection with the appropriate configuration.

Similarly, inertia values fall after some point as inertia is calculated by summing up the squared distances between each data point and its nearest cluster center. This value serves as a metric to evaluate the quality of the cluster assignment in the K-means algorithm. Minimizing the inertia is the objective of the K-means algorithm during the iterative process of finding the optimal cluster centers. Thus, we see a drop as we increase the number of clusters.

These can be visualized with the following graphs.


Inertia values for different number of clusters
Inertia
1200
1000
800
600
400
200
2
4
6
8
10
12
14
Number of Clusters


Accuracy scores for different number of clusters
Accuracy
0.87
0.86
0.85
0.84
0.83
0.82
0.81
0.80
2
4
6
8
10
12
14
Number of Clusters

Part B

Defining Tested SVM Parameters

The normalized data is separated (70% training, 15% validation, and 15% testing), and the scikit-learn implementations of SVC (an SVM type) and MLP are used. The following parameters are tested:

```
param_grid_poly = {
    "C": [0.001, 0.01, 0.1, 1, 10, 100],
    "degree": [2, 3, 4, 5],
    "gamma": ["scale", "auto"],
    "kernel": ["poly"],
}

param_grid = {
    "C": [0.001, 0.01, 0.1, 1, 10, 100],
    "gamma": ["scale", "auto"],
    "kernel": ["rbf", "linear", "sigmoid"],
}
```

The documentation for scikit-learn specifies that the ‘degree’ parameter exclusively impacts the polynomial (poly) kernel. Consequently, calculations for the poly kernel and others are handled separately in the implementation. This approach prevents unnecessary computations across various kernels with differing degrees. The ‘C’ parameter is a regularization parameter, with its value inversely proportional to the regularization strength. Additionally, the ‘gamma’ parameter modifies the kernel coefficient for the chosen kernels.

Defining Tested MLP Parameters

```
    "hidden_layer_sizes": [(2), (2, 2), (4), (4, 4),  (8), (8, 8),],
    "activation": ["identity", "logistic", "tanh", "relu"],
    "solver": ["lbfgs", "adam"],
    "alpha": [0.0001, 0.001, 0.01, 0.1],
    "learning_rate": ["constant", "invscaling", "adaptive"],
```

For the solver, three renowned methods that were recommended for use are tested. ‘Sgd’ was also tested, but it was removed due to computational performance issues, and its results were not good. The alpha value is crucial as it determines the strength of the L2 regularization term. Several activation functions are chosen, including ‘identity’, ‘tanh’, ‘logistic’, and ‘ReLU’, with ‘ReLU’ being the default and most popular. However, the logistic function is also evaluated to assess its performance. The size and number of hidden layers are critical factors; hence, various layer configurations are experimented with.

Testing Parameters

Initially, models are trained using 70.8% of the normalized data. A 6-fold cross-validation method was selected for validating the dataset. This involves dividing the training dataset into six parts. Each model is then trained in five parts, validated in the sixth part (14.1%), and repeated six times. The ‘GridSearchCV’ function from the scikit-learn library facilitated hyperparameter tuning through cross-validation. This function explores all possible parameter combinations based on the specified settings. It then adjusts the parameters under the outcomes from the 6-fold cross-validation. After the tuning, the function evaluates and reports the accuracy for each parameter combination. This procedure was implemented for the Support Vector Machine (SVM) and the Multi-layer Perceptron (MLP).

Best SVMs

Score Percentage (Validation)	Parameters
100	'C': 1, 'gamma': 'scale', 'kernel': 'rbf'
100	'C': 1, 'gamma': 'scale', 'kernel': 'linear'
100	'C': 1, 'gamma': 'auto', 'kernel': 'linear'

The parameters 'C': 1, 'gamma': 'scale', 'kernel': 'rbf' are picked as the best result.

Best MLPs

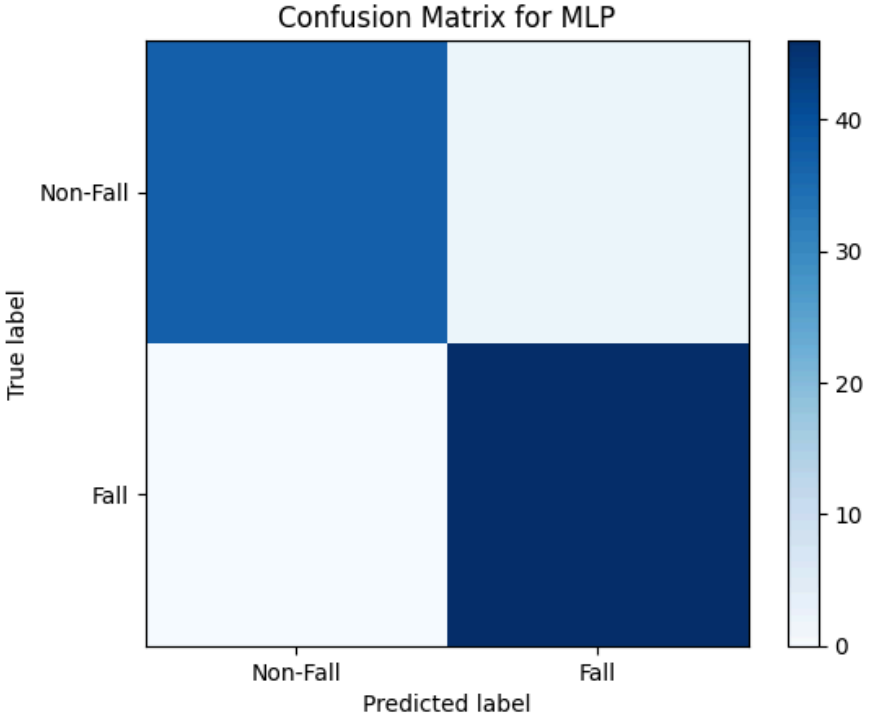
Score Percentage (Validation)	Parameters
100	'activation': 'identity', 'alpha': 0.0001, 'hidden_layer_sizes': 2, 'learning_rate': 'constant', 'solver': 'lbfgs'
100	'activation': 'identity', 'alpha': 0.0001, 'hidden_layer_sizes': 2, 'learning_rate': 'invscaling', 'solver': 'lbfgs'
100	'activation': 'identity', 'alpha': 0.0001, 'hidden_layer_sizes': 2, 'learning_rate': 'adaptive', 'solver': 'lbfgs'

The parameters 'activation': 'identity', 'alpha': 0.0001, 'hidden_layer_sizes': 2, 'learning_rate': 'constant', 'solver': 'lbfgs' are the best results.

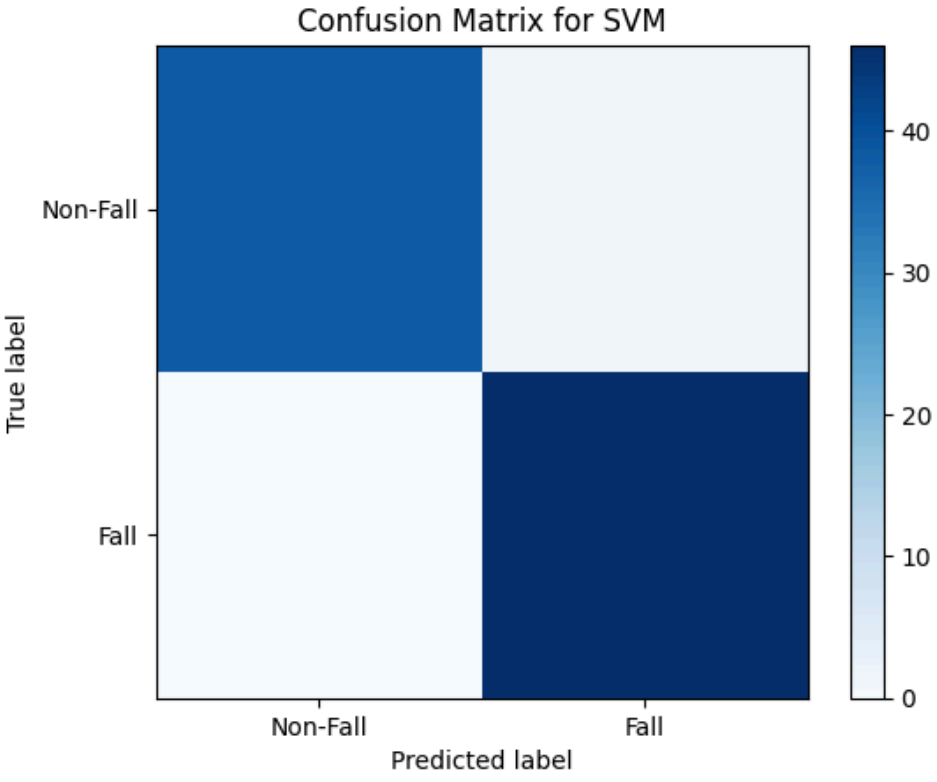
Results and Discussion

Test dataset results of the best models:

Accuracy on the test set for MLP: 97.6470588235294%



Accuracy on the test set for SVM: 98.82352941176471%



The SVM model achieved a perfect accuracy of 98.8%, while the MLP model obtained 97.6% accuracy. Although both models performed commendably, the SVM model was the best by a narrow margin. This slight disparity could be attributed to several factors. Initially, we opted for an 'rbf' kernel for the SVM, known for its capability to create non-linear classification boundaries. Similarly, the MLP, as a neural network, inherently captures the non-linearities of the data structure. Our successful hyperparameter tuning contributed to these impressive outcomes.

Additionally, the limited size of the dataset might have masked the complexity of a more extensive dataset, potentially making the task appear simpler than it is. Normalizing the data also helped minimize the impact of outliers, thus simplifying the classification task for both models. The performance of the MLP could be due to its learning of the underlying data structure, which is influenced by its architecture and optimization process. In comparing configurations, 575 out of 472 for the MLP achieved over 95% accuracy, while only 32 out of 84 SVC configurations reached this threshold, highlighting MLP's reliability in fall detection applications with the proper parameters.

In summary, it is demonstrated that with data preprocessing, careful model selection, and precise hyperparameter tuning, fall detection can be predicted with high confidence, approximately 97%. This success underscores the efficacy of devices in providing critical data for fall prediction. Given the structure of the problem, utilizing these algorithms could enable reliable fall detection in practical applications, affirming both methodologies in monitoring falls.

Appendix

```
# Görkem Kadir Solun - 22003214

### Notes

# Changing the random seeds affects the output.

# You may need to update the data path.

# Finding the best MLP may take up to 10 minutes.

# It may give warnings about the convergence of the MLP model.

import os
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
import pandas as pd
```

```
from sklearn.decomposition import PCA
from sklearn.preprocessing import MinMaxScaler
from sklearn.neural_network import MLPClassifier
from sklearn.cluster import KMeans
from sklearn.model_selection import GridSearchCV
from sklearn.svm import SVC
from sklearn.metrics import confusion_matrix, accuracy_score
from sklearn.utils._testing import ignore_warnings
from sklearn.exceptions import ConvergenceWarning
import random

# Set random seed
random_state = 92
random.seed(random_state)
np.random.seed(random_state)

# make pandas do not truncate
pd.set_option("display.max_colwidth", None)
pd.set_option("display.max_rows", None)

# Load the data
# NOTE: This data may need to be configured to the correct path
directory = os.getcwd()
data_path = os.path.join(directory,
"data\\falldetection_dataset.csv")

# Read the data with the first row as the data instead of the
header
raw_data = pd.read_csv(data_path, header=None)

# Get the labels at the second column
data_labels = raw_data.iloc[:, 1]
data_unique_labels = data_labels.unique()
# Get the rest of the data by removing the first two columns
data_features = raw_data.drop(raw_data.columns[[0, 1]], axis=1)

print("Data shape: ", data_features.shape)
print("Labels shape: ", data_labels.shape)
print("Unique labels: ", data_unique_labels)

""" print("Data: ", data)
print("Labels: ", labels) """
```

```
# Apply PCA to the data to reduce the dimensionality of the data
to 2
pca = PCA(n_components=2)
principal_components = pca.fit_transform(data_features)
explained_variance_ratios = pca.explained_variance_ratio_
eigenvalues = pca.explained_variance_
total_explained_variance_percentage =
sum(explained_variance_ratios) * 100

print("Principal Components shape: ", principal_components.shape)
print("Explained Variance Ratios: ", explained_variance_ratios)
print("Eigenvalues: ", eigenvalues)
print("Total Explained Variance Percentage: ",
total_explained_variance_percentage)


# Function to plot the PCA results with the labels
def plot_pca_results(principal_components, labels, title):
    # Plot the PCA results
    plt.figure()
    # First scatter the fall points
    plt.scatter(
        principal_components[labels == "F", 0],
        principal_components[labels == "F", 1],
        c="r",
        label="Fall",
        alpha=0.5,
    )
    # Then scatter the Non-fall points
    plt.scatter(
        principal_components[labels == "NF", 0],
        principal_components[labels == "NF", 1],
        c="b",
        label="Non-Fall",
        alpha=0.5,
    )
    plt.xlabel("Principal Component 1")
    plt.ylabel("Principal Component 2")
    plt.title(title)
    plt.legend()
    plt.show()
```

```
# Plot the PCA results
plot_pca_results(principal_components, data_labels, "PCA Results")

# Remove the first maximum outlier from the first principal
component

max_index = np.argmax(principal_components[:, 0])
principal_components_no_outliers = np.delete(principal_components,
max_index, axis=0)
labels_no_outlier = np.delete(data_labels, max_index, axis=0)
data_features_no_outlier = np.delete(data_features, max_index,
axis=0)

# Remove the first maximum outlier from the second principal
component
max_index = np.argmax(principal_components_no_outliers[:, 1])
principal_components_no_outliers = np.delete(
    principal_components_no_outliers, max_index, axis=0
)
labels_no_outlier = np.delete(labels_no_outlier, max_index,
axis=0)
data_features_no_outlier = np.delete(data_features_no_outlier,
max_index, axis=0)

print(
    "Principal Components without outliers shape: ",
    principal_components_no_outliers.shape,
)
print("Labels without outliers shape: ", labels_no_outlier.shape)
print("Data without outliers shape: ",
data_features_no_outlier.shape)

# Plot the previous PCA results without the outliers
# NOTE: The outliers are removed from the data and labels, so we
need to calculate the PCA again
# to get the principal components without the outliers, this is
just for visualization purposes
plot_pca_results(
    principal_components_no_outliers,
    labels_no_outlier,
```

```
    "Previous PCA Results without Outliers (without updated PCA)",
)

# Applay PCA to data withou outliers
pca_no_outliers = PCA(n_components=2)
principal_components_no_outliers = pca_no_outliers.fit_transform(
    data_features_no_outlier
)
explained_variance_ratios_no_outliers =
pca_no_outliers.explained_variance_ratio_
eigenvalues_no_outliers = pca_no_outliers.explained_variance_
total_explained_variance_percentage_no_outliers = (
    sum(explained_variance_ratios_no_outliers) * 100
)

print(
    "Principal Components without outliers shape: ",
    principal_components_no_outliers.shape,
)
print(
    "Explained Variance Ratios without outliers: ",
    explained_variance_ratios_no_outliers,
)
print("Eigenvalues without outliers: ", eigenvalues_no_outliers)
print(
    "Total Explained Variance Percentage without outliers: ",
    total_explained_variance_percentage_no_outliers,
)

plot_pca_results(
    principal_components_no_outliers, labels_no_outlier, "PCA
Results without Outliers"
)

# Scale the data with the MinMaxScaler to normalize the data
min_max_scaler = MinMaxScaler()
data_features_scaled =
min_max_scaler.fit_transform(data_features_no_outlier)
# Transform the labels to binary, F=1, NF=0 for the binary
classification in the future
data_labels_binary = np.where(labels_no_outlier == "F", 1, 0)
```

```
# Apply PCA again to data without outliers
pca_normalized = PCA(n_components=2)
principal_components_normalized =
pca_normalized.fit_transform(data_features_scaled)
explained_variance_ratios_normalized =
pca_normalized.explained_variance_ratio_
eigenvalues_normalized = pca_normalized.explained_variance_
total_explained_variance_percentage_normalized = (
    sum(explained_variance_ratios_normalized) * 100
)

print("Principal Components normalized shape: ",
principal_components_normalized.shape)
print("Explained Variance Ratios normalized: ",
explained_variance_ratios_normalized)
print("Eigenvalues normalized: ", eigenvalues_normalized)
print(
    "Total Explained Variance Percentage normalized: ",
    total_explained_variance_percentage_normalized,
)

plot_pca_results(
    principal_components_normalized, labels_no_outlier, "PCA
Results Normalized"
)


# Plotting function to plot the KMeans clustering results for
different number of clusters
def plot_kmeans_results(principal_components, labels, n_clusters):
    plt.figure()
    for i in range(n_clusters):
        plt.scatter(
            principal_components[labels == i, 0],
            principal_components[labels == i, 1],
            label="Cluster " + str(i + 1),
        )
    plt.xlabel("Principal Component 1")
    plt.ylabel("Principal Component 2")
    plt.title("KMeans Clustering with " + str(n_clusters) + "
clusters")
    plt.legend()
```

```
    plt.show()


# Map the cluster labels of the k-means into 1 and 0's according
to the majority label of that cluster.
# If a cluster had more Fall than Non Fall, that label of the
cluster will be mapped to Fall.
def map_clusters_predicted_values(labels, predictions,
cluster_count):
    new_predictions = np.zeros(len(labels))
    for i in range(cluster_count):
        cluster_indices = np.where(predictions == i)
        fall_count = np.sum(labels[cluster_indices] == 1)
        non_fall_count = np.sum(labels[cluster_indices] == 0)
        if fall_count > non_fall_count:
            new_predictions[cluster_indices] = 1

    return new_predictions


# Calculate the accuracy of the clustering results
def calculate_cluster_accuracy(labels, predictions,
cluster_count):
    correct_predictions = map_clusters_predicted_values(
        labels, predictions, cluster_count
    )
    return accuracy_score(labels, correct_predictions)


# Inertia values for different number of clusters
inertia_values = []
# Accuracy scores for different number of clusters
accuracy_scores = []

# Apply KMeans clustering to the normalized data withe different
number of clusters
for n_clusters in range(2, 16):
    kmeans = KMeans(n_clusters=n_clusters, random_state=0)
    prediction =
kmeans.fit_predict(principal_components_normalized)

    labels = kmeans.labels_
```

```
    centroids = kmeans.cluster_centers_

    # Calculate the inertia value of the clustering
    inertia_values.append(kmeans.inertia_)

    # Calculate the accuracy of the clustering
    accuracy_scores.append(
        calculate_cluster_accuracy(data_labels_binary, labels,
n_clusters)
    )

    # Plot the KMeans clustering results for different number of
clusters
    # Also, calculate the accuracy of the clustering results
    if n_clusters <= 7:
        plot_kmeans_results(principal_components_normalized,
labels, n_clusters)
        print(
            "Accuracy of the clustering with ",
            n_clusters,
            " clusters: ",
            accuracy_scores[-1],
        )

# Plot the inertia values for different number of clusters
plt.figure()
plt.plot(range(2, 16), inertia_values)
plt.xlabel("Number of Clusters")
plt.ylabel("Inertia")
plt.title("Inertia values for different number of clusters")
plt.show()

# Plot the accuracy scores for different number of clusters
plt.figure()
plt.plot(range(2, 16), accuracy_scores)
plt.xlabel("Number of Clusters")
plt.ylabel("Accuracy")
plt.title("Accuracy scores for different number of clusters")
plt.show()

# Apply SVM to the normalized data
```

```
# First split the normalized data into training, validation and
testing sets
# 85% training, 15% testing
# NOTE: Validation set is automatically created by GridSearchCV
with cv=6 as 14.1% of the data
data_train, data_test, labels_train, labels_test =
train_test_split(
    data_features_scaled, data_labels_binary, test_size=0.15,
random_state=random_state
)

# Find the best hyperparameters for the SVM model
# Use GridSearchCV to find the best hyperparameters for the SVM
model
# Create two dictionaries for the hyperparameters to search for
# One is for polynomial kernel and the other is for rbf, linear,
and sigmoid kernels
param_grid_poly = {
    "C": [0.001, 0.01, 0.1, 1, 10, 100],
    "degree": [2, 3, 4, 5],
    "gamma": ["scale", "auto"],
    "kernel": ["poly"],
}

param_grid = {
    "C": [0.001, 0.01, 0.1, 1, 10, 100],
    "gamma": ["scale", "auto"],
    "kernel": ["rbf", "linear", "sigmoid"],
}

# Search for the best hyperparameters for the polynomial kernel
svm_poly = SVC()
svm_poly_search = GridSearchCV(svm_poly, param_grid_poly, cv=6)
svm_poly_search.fit(data_train, labels_train)
grid_results_poly = svm_poly_search.cv_results_

# Search for the best hyperparameters for the rbf, linear, and
sigmoid kernels
svm = SVC()
svm_search = GridSearchCV(svm, param_grid, cv=6)
svm_search.fit(data_train, labels_train)
grid_results = svm_search.cv_results_
```

```
# Combine the results of the polynomial kernel and the rbf,
linear, and sigmoid kernels
# and sort them by the mean_test_score

# Combine the results
results = []
for i in range(len(grid_results["mean_test_score"])):
    results.append(
        {
            "mean_test_score": grid_results["mean_test_score"][i],
            "params": grid_results["params"][i],
        }
    )

for i in range(len(grid_results_poly["mean_test_score"])):
    results.append(
        {
            "mean_test_score":
grid_results_poly["mean_test_score"][i],
            "params": grid_results_poly["params"][i],
        }
    )

# Sort the results by the mean_test_score
results = sorted(results, key=lambda x: x["mean_test_score"],
reverse=True)

# Show the hyperparameters in the dataframe nicely
results_df = pd.DataFrame(results)
print("Results: ", results_df)

# Get the best hyperparameters
best_hyperparameters = svm_search.best_params_
print("Best Hyperparameters: ", best_hyperparameters)

# Train the SVM model with the best hyperparameters
svm_best = SVC(**best_hyperparameters)
svm_best.fit(data_train, labels_train)

# Get the accuracy of the model on the test set in percentage
accuracy_test = svm_best.score(data_test, labels_test)
```

```
print("Accuracy on the test set for SVM: ", accuracy_test * 100,
"%")

# Get the confusion matrix of the model on the test set
labels_pred = svm_best.predict(data_test)
conf_matrix = confusion_matrix(labels_test, labels_pred)
print("Confusion Matrixfor SVM: ", conf_matrix)

# Plot the confusion matrix
plt.figure()
plt.imshow(conf_matrix, interpolation="nearest",
cmap=plt.cm.Blues)
plt.title("Confusion Matrix for SVM")
plt.colorbar()
tick_marks = np.arange(2)
plt.xticks(tick_marks, ["Non-Fall", "Fall"])
plt.yticks(tick_marks, ["Non-Fall", "Fall"])
plt.ylabel("True label")
plt.xlabel("Predicted label")
plt.show()

# Apply MLP to the normalized data

# Find the best hyperparameters for the MLP model
# Use GridSearchCV to find the best hyperparameters for the MLP
model

# Create a dictionary for the hyperparameters to search for
param_grid_mlp = {
    "hidden_layer_sizes": [
        (2),
        (2, 2),
        (4),
        (4, 4),
        (8),
        (8, 8),
    ],
    "activation": ["identity", "logistic", "tanh", "relu"],
    "solver": ["lbfgs", "adam"],
    "alpha": [0.0001, 0.001, 0.01, 0.1],
    "learning_rate": ["constant", "invscaling", "adaptive"],
}
```

```
# Function to apply grid search for the MLP model without warnings
@ignore_warnings(category=ConvergenceWarning)
def apply_grid_search(mlp, param_grid, data_train, labels_train):
    mlp_search = GridSearchCV(mlp, param_grid, cv=6,
scoring="accuracy")
    mlp_search.fit(data_train, labels_train)
    return mlp_search


# Search for the best hyperparameters for the MLP model
mlp = MLPClassifier()
mlp_search = apply_grid_search(mlp, param_grid_mlp, data_train,
labels_train)
grid_results = mlp_search.cv_results_

# Combine the results
results = []
for i in range(len(grid_results["mean_test_score"])):
    results.append(
        {
            "mean_test_score": grid_results["mean_test_score"][i],
            "params": grid_results["params"][i],
        }
    )

# Sort the results by the mean_test_score
results = sorted(results, key=lambda x: x["mean_test_score"],
reverse=True)

# Show the hyperparameters in the dataframe nicely
results_df = pd.DataFrame(results)
print("Results: ", results_df)

# Get the best hyperparameters
best_hyperparameters_mlp = mlp_search.best_params_
print("Best Hyperparameters for MLP: ", best_hyperparameters_mlp)

# Train the MLP model with the best hyperparameters
mlp_best = MLPClassifier(**best_hyperparameters_mlp)
mlp_best.fit(data_train, labels_train)
```

```
# Get the accuracy of the model on the test set in percentage
accuracy_test_mlp = mlp_best.score(data_test, labels_test)
print("Accuracy on the test set for MLP: ", accuracy_test_mlp *
100, "%")

# Get the confusion matrix of the model on the test set
labels_pred_mlp = mlp_best.predict(data_test)
conf_matrix_mlp = confusion_matrix(labels_test, labels_pred_mlp)
print("Confusion Matrix for MLP: ", conf_matrix_mlp)

# Plot the confusion matrix
plt.figure()
plt.imshow(conf_matrix_mlp, interpolation="nearest",
cmap=plt.cm.Blues)
plt.title("Confusion Matrix for MLP")
plt.colorbar()
tick_marks = np.arange(2)
plt.xticks(tick_marks, ["Non-Fall", "Fall"])
plt.yticks(tick_marks, ["Non-Fall", "Fall"])
plt.ylabel("True label")
plt.xlabel("Predicted label")
plt.show()
```