Bilkent University

GE461 Introduction to Data Science

Project 5 Report

Kerem Şahin

21901724

PART A)

When we first apply PCA to our dataset and project it, we get the following graphs:

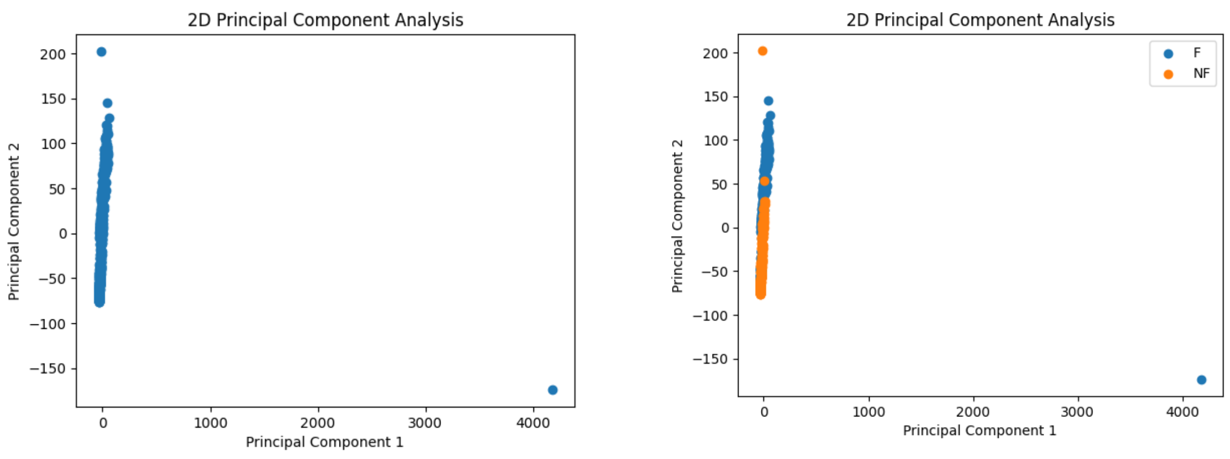
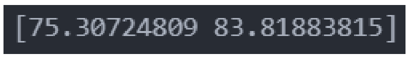


Fig. 1. Data PCA Projected Visualization with and without labels



```
[75.30724809 83.81883815]
```

Fig. 2. Cumulative Sum of Explained Variance by Principal Components

We can see that in Fig. 1., there are two data points that are outliers. These outliers may affect our analysis, therefore it would be appropriate to remove these two points from our dataset. In Fig. 2., we can see the cumulative sum of the variances for the first two components. The array means that the first principal component explains 75% of the variance by itself, and the combination of the first and second principal component explain 83% of the total variance (implies that the second principal component explains 8% of the variance). However, since we have detected outliers and we have not applied normalization, we must rerun our test in order to get a more realistic result. MinMaxScaling is sensitive to outliers, but once we remove them, we can use it safely. Therefore, when we remove the outliers and apply MinMaxScaler to our dataset, we will get the following graphs:

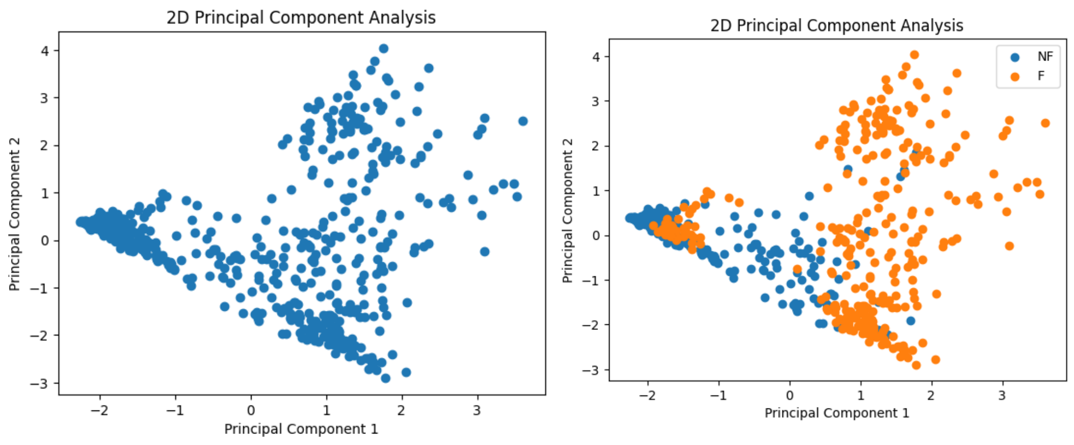
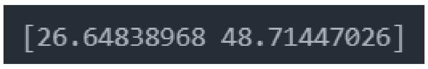


Fig. 3. Normalized Data PCA Projected Visualization with and without labels



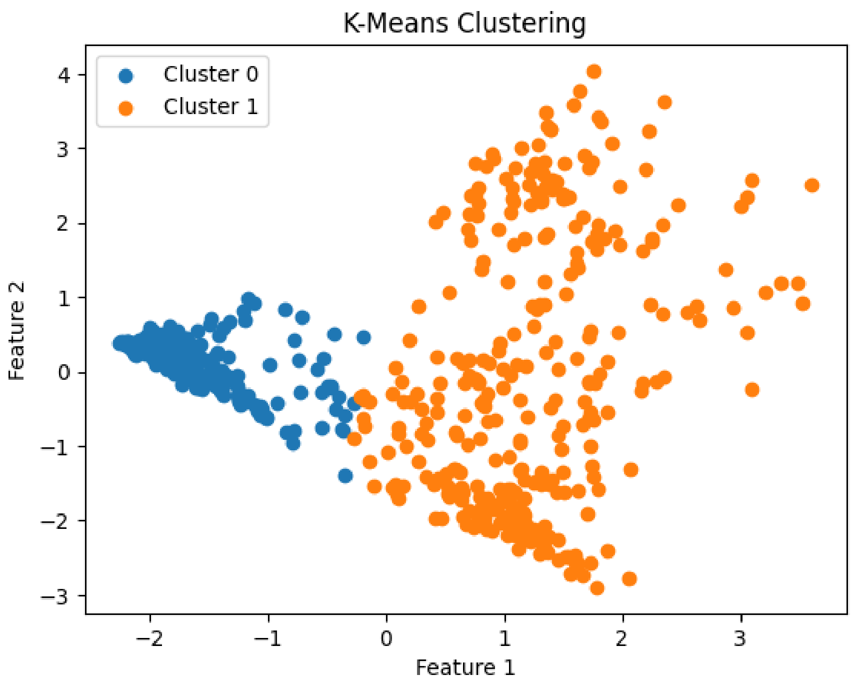
```
[26.64838968 48.71447026]
```

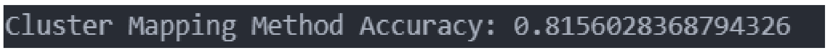
Fig. 4. Cumulative Sum of Explained Variance by Normalized Principal Components

As we can see, we get a more interpretable result and the visualization represents the data better. After this, the variances explained by the principal components are 26% and 22%, respectively

(48 – 26 = 22). The first result gave a cumulative variance of 80%, which also seems unrealistic. Therefore, we will keep on with the normalized and outlier deleted results throughout our experiment.

After that, K-means clustering was applied on the projected data. First, clustering with number of clusters = 2 will be given:

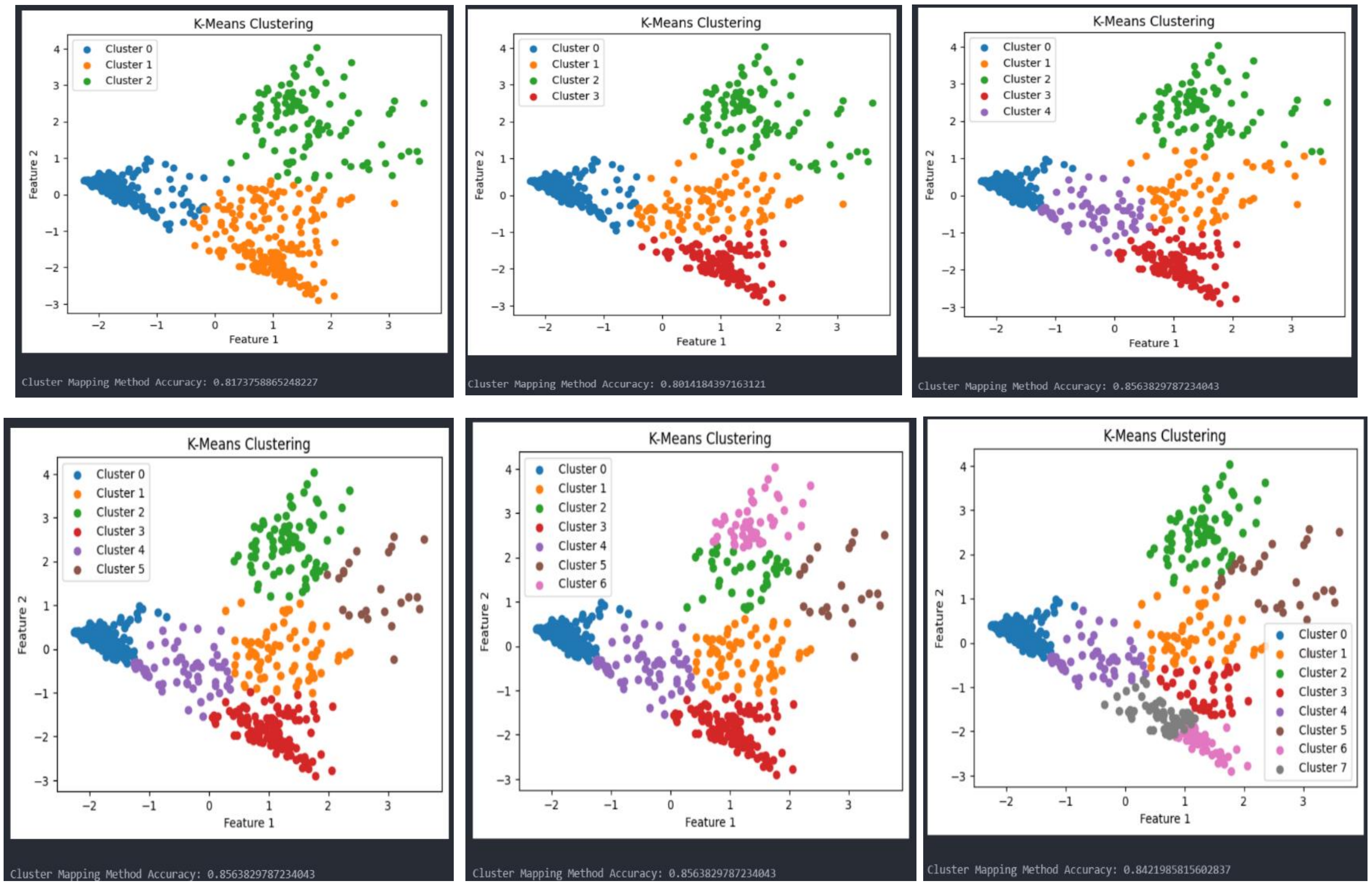




```
Cluster Mapping Method Accuracy: 0.8156028368794326
```

Fig. 5. K-Means clustering and its Accuracy with 2 Clusters

The method of accuracy is as follows. Since the cluster labels do not have any innate meaning, we can only look at whether the given clusters and their actual corresponding values are matching or not. In order to achieve this, we have made an assumption that we can always map the labelings to the ground truth label which gives the maximum accuracy. The logic can be better explained from Fig. 5. visually. In the figure, there are two clusters. We could have either chosen the blue data points to represent Not Falling, or we could have chosen orange as Not Falling. Since they labels gathered from the k-means does not have any meaning by itself, we can always interperate it as the clustering which achieves the highest accuracy, because the only purpose is to cluster majority of the same classes together. By using this method, we will be able to understand how well the clusters are formed. When this method was applied to k-means cluster with n = 2, we have gotten a 81.56% accuracy. When we compare Fig. 5. With Fig. 3., we can also observe that the clusters are visually accurate as well. The 20% not correct labelings are most likely coming from the data points where the Falling and Not Falling data points are overlapping each other (in Fig. 3.). It would be a hard task to separate it, but still the clustering method was able to differentiate 81.56% of the datapoints correctly, which indicates that fall detection is possible. We can try different clusterings to see if our clusterings improve. The ranges of clusters that are tried are in the range between 3 to 8:



```
Cluster Mapping Method Accuracy: 0.8173758865248227
Cluster Mapping Method Accuracy: 0.8014184397163121
Cluster Mapping Method Accuracy: 0.8563829787234043
Cluster Mapping Method Accuracy: 0.8563829787234043
Cluster Mapping Method Accuracy: 0.8563829787234043
Cluster Mapping Method Accuracy: 0.8421985815602837
```

Fig. 6. Different Settings of K-Means Clustering with Num of Clusters from 3 to 8 (Along with accuracies below them)

If you zoom belove the graphs, you will see the accuracy results for different clusterings. When we look at the graphs and compare them with previous findings, we can have the following observations. First of all, after the cluster count of 5, we can see that our clustering accuracy evaluation method has increased. Visually, when we compare Fig. 3 with the Fig. 6's third figure (cluster count = 5), we can see that the newly created purple region is the reason for the increase in accuracy. Lower cluster counts were not able to detect that region as a separate entity. However, we can see from Fig. 3 that despite the overlappings of NF and F datapoint, the middle region has a separate section that consists mostly of Falling datapoints. Clusters higher than 5 are able to detect this region. After clusterings of 5, the accuracies do not change much. Therefore, for this method a clustering of 5 is the optimal value. On top of that, we can again give this as a proof of the seperability of the datapoints, which indicates that falling detection is possible with the right setting.

PART B)

SVC Configuration

The scikit implementations SVC (an SVM type) and MLP were used. For SVC, important parameters that have been tuned are as follows:

```
kernel:('linear', 'rbf', 'sigmoid, poly'),
C:[0.001,0.01,0.1,1,10],
gamma:('scale', 'auto'),
degree: [0,1,2,3,4,5,6]}
```

As a side note, as specified in the docs of scikit, the degree parameter only affects the poly kernel. Therefore, in the code, the calculations have been separated for poly and other kernels in order to not redundantly calculate different kernels with different degrees. The C parameter is the regularization parameter and it is inverse with the strength of the regularization. Gamma adjusts the kernel coefficient for our selected kernels.

MLP Configuration

For the MLP, the following important parameters have been tuned:

```
solver : ('lbfgs', 'adam'),
alpha:[0.001,0.01,0.1,1,10],
activation : ('logistic','relu'),
hidden_layer_sizes: [(64),(4,4),(8,8),(16,16),(32,32)]
```

For solver, two of the famous methods were tested which were also recommended to use. Alpha value is going to determine how powerful the term for L2 regularization will be. Our activation functions are selected to be logistic and relu. The default and the most famous one is relu. Since we know that relu is better than tanh for most cases, it was not selected for testing. However, we also looked at the logistic to see how it performs. The hidden layer size is important as well as the number of hidden layers. Therefore, different layer configurations were selected.

Train & Validation & Test Procedure

First of all, since we have too many features and only 500 samples, PCA was appropriate to apply. After tests, it turns out that 10 principal components were able to represent 80% of the variance in the data. This will both reduce the unnecessary complexity and it will also make the calculations faster. Then, the dataset has first been split as 70% train dataset and 30% test dataset. For **validation** dataset, a 3-fold cross-validation strategy was chosen, which means that our training dataset will be split into 3 sections and will be trained on 2 different splits, and will be cross validated on the 3rd split, which will go on for 3 times. In order to make this cross validation hyperparameter tuning, "GridSearchCV" library of scikit learn was used. The method exhaustively searchs all possible combinations and tunes the parameters according to the 3-fold cross validation strategy. Then, the method returns the accuracy of each possible combination. This has been applied to both SVC and MLP. The results are as follows:

SVC Hyperparameter Tuning Results:

Accuracy	C	degree	gamma	kernel
0.997455	1.000	0	scale	rbf
0.994930	1.000	0	auto	rbf
0.984791	1.000	0	auto	linear
0.984791	1.000	0	scale	linear
0.966960	0.100	0	auto	linear
0.966960	0.100	0	scale	linear
0.959326	1.000	3	scale	poly

Accuracy	C	degree	gamma	kernel
0.933958	1.000	0	auto	sigmoid
0.911057	1.000	2	scale	poly
0.906084	1.000	0	scale	sigmoid
0.900937	1.000	3	auto	poly
0.895906	0.100	3	scale	poly
0.885670	0.100	0	scale	rbf
0.880619	0.100	0	auto	rbf
0.875549	0.010	0	scale	linear
0.875549	0.010	0	auto	linear
0.870499	1.000	2	auto	poly
0.867974	0.100	3	auto	poly
0.867954	0.100	0	auto	sigmoid
0.867935	1.000	5	scale	poly
0.865410	0.100	0	scale	sigmoid
0.865371	1.000	4	scale	poly
0.860340	0.100	2	auto	poly
0.840003	0.100	2	scale	poly
0.829921	1.000	4	auto	poly
0.812129	1.000	6	scale	poly
0.807040	1.000	5	auto	poly
0.779108	0.100	4	scale	poly
0.779089	0.100	5	scale	poly
0.756303	0.100	5	auto	poly
0.754202	0.100	4	auto	poly
0.733422	1.000	6	auto	poly
0.718193	0.100	6	scale	poly
0.588827	0.100	6	auto	poly
0.573579	0.010	6	scale	poly
0.573579	0.010	5	scale	poly
0.571035	0.010	4	scale	poly
0.560895	0.010	3	scale	poly
0.558370	0.001	4	auto	poly
0.558370	0.001	6	auto	poly
0.558370	0.001	6	scale	poly
0.558370	0.001	5	auto	poly
0.558370	0.001	5	scale	poly
0.558370	0.001	3	auto	poly

Accuracy	C	degree	gamma	kernel
0.558370	0.001	4	scale	poly
0.558370	0.010	2	auto	poly
0.558370	0.001	3	scale	poly
0.558370	0.001	2	auto	poly
0.558370	0.010	5	auto	poly
0.558370	0.010	2	scale	poly
0.558370	0.001	2	scale	poly
0.558370	0.010	3	auto	poly
0.558370	0.010	4	auto	poly
0.558370	0.010	0	auto	sigmoid
0.558370	0.010	0	auto	rbf
0.558370	0.001	0	scale	rbf
0.558370	0.001	0	scale	sigmoid
0.558370	0.001	0	auto	linear
0.558370	0.001	0	auto	rbf
0.558370	0.001	0	auto	sigmoid
0.558370	0.010	0	scale	rbf
0.558370	0.010	0	scale	sigmoid
0.558370	0.001	0	scale	linear
0.558370	0.010	6	auto	poly

Table 1. Hyperparameter Tuning Results of SVC

MLP Hyperparameter Tuning Results:

Accuracy	activation	alpha	hidden_layer_sizes	solver
0.994930	relu	1.000	(32, 32)	adam
0.994930	relu	0.100	(32, 32)	adam
0.994930	relu	0.010	(32, 32)	adam
0.994930	relu	1.000	(32, 32)	lbfgs
0.994930	relu	0.001	64	adam
0.994930	relu	1.000	(16, 16)	lbfgs
0.994930	relu	0.010	64	lbfgs
0.994930	relu	0.100	(32, 32)	lbfgs
0.994911	relu	0.001	64	lbfgs
0.992405	relu	0.001	(16, 16)	adam
0.992386	logistic	0.010	(32, 32)	lbfgs
0.992386	logistic	0.100	(4, 4)	lbfgs

Accuracy	activation	alpha	hidden_layer_sizes	solver
0.992386	relu	0.100	(16, 16)	adam
0.992386	logistic	0.100	64	lbfgs
0.992386	relu	0.100	(16, 16)	lbfgs
0.992386	logistic	0.001	(8, 8)	lbfgs
0.992386	relu	1.000	(8, 8)	lbfgs
0.992386	logistic	0.010	(16, 16)	lbfgs
0.992386	relu	1.000	(4, 4)	lbfgs
0.992386	logistic	0.100	(8, 8)	lbfgs
0.992386	relu	0.010	(8, 8)	lbfgs
0.992386	relu	0.001	(8, 8)	lbfgs
0.992386	relu	0.001	(32, 32)	lbfgs
0.992386	relu	0.010	(16, 16)	adam
0.992386	relu	0.010	(4, 4)	adam
0.992386	relu	0.010	(4, 4)	lbfgs
0.992386	relu	0.001	(16, 16)	lbfgs
0.992386	relu	0.100	64	lbfgs
0.992386	logistic	0.100	(32, 32)	lbfgs
0.992386	relu	0.001	(32, 32)	adam
0.992386	logistic	0.100	(16, 16)	lbfgs
0.992386	relu	0.010	(16, 16)	lbfgs
0.992386	relu	0.010	(8, 8)	adam
0.989880	relu	1.000	(8, 8)	adam
0.989860	relu	0.100	(4, 4)	lbfgs
0.989860	relu	0.010	64	adam
0.989860	relu	0.100	64	adam
0.989860	relu	1.000	64	lbfgs
0.989841	logistic	0.010	(8, 8)	lbfgs
0.989841	relu	0.100	(8, 8)	lbfgs
0.989841	logistic	0.010	(4, 4)	lbfgs
0.989841	logistic	0.010	64	lbfgs
0.987335	relu	1.000	(4, 4)	adam
0.987316	logistic	0.001	(32, 32)	lbfgs
0.987316	relu	0.100	(8, 8)	adam
0.987316	relu	0.001	(8, 8)	adam
0.987316	relu	1.000	(16, 16)	adam
0.987297	relu	0.010	(32, 32)	lbfgs
0.987297	logistic	0.001	(16, 16)	lbfgs

Accuracy	activation	alpha	hidden_layer_sizes	solver
0.987277	relu	0.100	(4, 4)	adam
0.984791	relu	1.000	64	adam
0.984791	logistic	1.000	(4, 4)	lbfgs
0.982265	logistic	1.000	(8, 8)	lbfgs
0.982246	logistic	0.100	(32, 32)	adam
0.982246	logistic	0.001	(4, 4)	adam
0.982208	logistic	0.001	64	lbfgs
0.982208	relu	0.001	(4, 4)	lbfgs
0.979721	logistic	1.000	(32, 32)	lbfgs
0.979721	logistic	1.000	(16, 16)	lbfgs
0.979682	logistic	0.001	(4, 4)	lbfgs
0.977176	logistic	0.100	(16, 16)	adam
0.977176	logistic	0.100	64	adam
0.977176	logistic	0.010	64	adam
0.977176	logistic	0.100	(8, 8)	adam
0.977157	logistic	0.010	(4, 4)	adam
0.977157	logistic	0.001	(8, 8)	adam
0.977138	logistic	0.001	(32, 32)	adam
0.974632	logistic	0.001	64	adam
0.974632	logistic	0.010	(8, 8)	adam
0.974632	logistic	0.001	(16, 16)	adam
0.972087	logistic	1.000	64	lbfgs
0.972087	logistic	0.010	(16, 16)	adam
0.972087	logistic	0.010	(32, 32)	adam
0.954256	logistic	1.000	64	adam
0.954256	logistic	1.000	(8, 8)	adam
0.954237	logistic	1.000	(16, 16)	adam
0.951712	logistic	1.000	(32, 32)	adam
0.842297	logistic	0.100	(4, 4)	adam
0.840813	relu	0.001	(4, 4)	adam
0.695775	logistic	1.000	(4, 4)	adam

Table 2. Hyperparameter Tuning Results of MLP

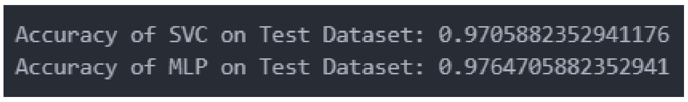
Selection of Model

After we gathered our results, we can choose the best models for both MLP and SVC according to the accuracies gathered from hyperparameter tuning. For SVC, the best model is **C = 1**, **gamma = scale** and **the kernel is an rbf**. For MLP, there are several models with the same top accuracy score. The

activation function will be selected as relu as all of the top scoring ones are relu. For **the solver, lbfgs** will be used because in the documentation, it is stated that lbfgs can achieve better and faster results for small datasets. **Alpha value will be selected as 0.1**. And lastly, **a hidden layer with size (32,32) will be chosen** because with two hidden layers, we will be able to get better classifications as the model will be able to detect complex relations as well.

Test Results & Discussion

Now let's see the prediction accuracies on the test dataset for the two selected models:



```
Accuracy of SVC on Test Dataset: 0.9705882352941176
Accuracy of MLP on Test Dataset: 0.9764705882352941
```

Fig. 7. Accuracy of Final Models on Test Set

As it can be seen, the MLP model is obtaining the better accuracy result with 97.64%, whereas SVC gathered 97.05%. Even though both of them achieved good results, MLP is the winner, but the difference is very slight. There might be several reasons for this behaviour. First of all, we selected an rbf kernel for the SVC and rbf kernel is also capable of forming non-linear boundaries for classification. MLP is already a neural network which is capable of capturing the non linearities of the underlying structure. Therefore, as a result of our hyperparameter tuning's success, both of them achieved very good results. On top of that, the dataset is very limited and therefore by chance, we may have not seen a complex dataset, but in reality the task may be harder. Other than that, since we applied PCA, we were able to reduce the noise that may have been caused by 300 features because it is relatively high when compared to the sample size. This situation probably made the task easier for the classifiers. The slightly better outcome of the MLP model may be due to several reasons. First of all, it may have been by pure chance as their accuracy levels are very similar and you may get different results on different runs. Secondly, MLP is a neural network and therefore it may have been able to learn the underlying structure better because of its structure and optimization process. Lastly, when we look at all different models, we see that out of 80 configurations, 77 of them had an accuracy above 95% for MLP. However, for SVC, out of 64 models, only 7 of them were able to get an accuracy above 95%. This means that MLP is more reliable when it comes to detecting fall, and it is advised to be used in real life applications over SVC.

In conclusion, we were able to predict fall with a high confidence (~97%), which can be considered as a success. Therefore, we can say that fall detection is possible with careful data preprocessing, model selection and hyperparameter tuning, we can predict falling with a high confidence. This also means that the wearable devices are providing enough information on the required features for predicting fall. The problem at hand has a structure that is seperable, so if a company tries to use these algorithms, they will be able to make reliable detections with these mechanisms. Therefore, we can say that both the wearable device and our methods can be used for predicting fall detection.

APPENDIX: CODE

!README!: In the code, I am using the tabulate library and you need to install it in order for the code to run. Secondly, due to the behavior of python, if a figure is displayed, you have to close the figure in order for the code to continue. Also, the dataset has to be in the same directory as the file. Lastly, some of the outputs for dataframes are sometimes shrinked in the terminal, you can run one more time if the terminal shrinks. The code is not copyable from pdf because it disturbs indentation, therefore I am also adding a google drive link for the code: https://drive.google.com/file/d/1RMPSrXBCT5LpcoR1-AxRoI2QyvcC3tiS/view?usp=sharing

```
# -*- coding: utf-8 -*-
"""fall_detect.ipynb

Transformed to .py from .ipynb by collab
Kerem Şahin

PLEASE INSTALL TABULAR LIBRARY
Imports
"""

import pandas as pd
import numpy as np
import os
from sklearn.preprocessing import RobustScaler
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.decomposition import PCA
from sklearn.cluster import KMeans
from sklearn import metrics
from matplotlib import pyplot as plt
from sklearn.metrics import adjusted_rand_score
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn.neural_network import MLPClassifier as MLP
from sklearn.model_selection import GridSearchCV
from tabulate import tabulate# YOU NEED TABULATE LIBRARY

"""Function Definitions"""

#method is defined to plot clusters of different cluster sizes
def plot_clusters(X, labels):
    unique_labels = np.unique(labels)
    for label in unique_labels:
        cluster_points = X[labels == label]
        plt.scatter(cluster_points[:, 0], cluster_points[:, 1],
label=f'Cluster {label}')
```

```python
    plt.xlabel('Feature 1')
    plt.ylabel('Feature 2')
    plt.title('K-Means Clustering')
    plt.legend()
    plt.show()

#method is defined to plot principal components
def plot_pca(X, labels):
    unique_labels = np.unique(labels)
    for label in unique_labels:
        cluster_points = X[labels == label]
        if label == 1 or label == 'F':
            plt.scatter(cluster_points[:, 0], cluster_points[:, 1], label='F')
        else:
            plt.scatter(cluster_points[:, 0], cluster_points[:, 1],
label='NF')
    plt.xlabel('Principal Component 1')
    plt.ylabel('Principal Component 2')
    plt.title('2D Principal Component Analysis')
    plt.legend()
    plt.show()

#this emthod maps the clustering labels of the kmeans into 1 and 0's according
to th 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 cluster_prediction_mapping(labels,predictions,cluster_num):
    mapped_predictions = predictions
    for i in range(cluster_num):
        indexes = np.where(predictions == i)[0]#indexes of the ith cluster
members
        F_count = np.count_nonzero(labels[indexes] == 1)
        NF_count = np.count_nonzero(labels[indexes] == 0)
        if F_count > NF_count:
            mapped_predictions[indexes] = 1
        else:
            mapped_predictions[indexes] = 0

    return mapped_predictions

#Returns the accuracy according to the cluster mapping methodology
def cluster_accuracy(labels,predictions,cluster_num):
    mapped_predictions =
cluster_prediction_mapping(labels,predictions,cluster_num)
    return metrics.accuracy_score(labels, mapped_predictions)

"""Data Read & Preprocessing"""

cur_dir = os.path.dirname(os.path.abspath(__file__))
path = os.path.join(cur_dir,"falldetection_dataset.csv")
```

```
data_df = pd.read_csv(path, header=None)
features_df = data_df.drop([0,1], axis=1)
labels_df = data_df.iloc[:,1]
features = features_df.values
labels = labels_df.values

"""PART A"""

#Use PCA as explained. Also, it can be used to detect outliers so remove them.
pca = PCA(n_components=2)
principal_components = pca.fit_transform(features)

explained_variance_ratio = pca.explained_variance_ratio_
eig_vals = pca.explained_variance_
sorted_eigenvalues = sorted(eig_vals, reverse=True)
cumulative_var_ratio = np.cumsum(explained_variance_ratio)*100
print(cumulative_var_ratio)

plt.scatter(principal_components[:, 0], principal_components[:, 1])
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('2D Principal Component Analysis')
plt.show()

plot_pca(principal_components,labels)

outlier_sample_idx1 = np.argmax(principal_components[:,0])#selects the outlier
in column 0
outlier_sample_idx2 = np.argmax(principal_components[:,1])#selects the outlier
in column 1

features_no_outlier = np.delete(features, outlier_sample_idx1, axis=0)
features_no_outlier = np.delete(features_no_outlier, outlier_sample_idx2,
axis=0)
labels_no_outlier = np.delete(labels, outlier_sample_idx1, axis=0)
labels_no_outlier = np.delete(labels_no_outlier, outlier_sample_idx2, axis=0)

#Scale the data with the removed outliers
scaler = MinMaxScaler()
features_scaled = scaler.fit_transform(features_no_outlier)#features are
scaled
labels_mapped = np.where(labels_no_outlier == 'F', 1, 0)#labels are mapped to
1 and 0

pca2 = PCA(n_components=2)
pc = pca2.fit_transform(features_scaled)
explained_variance_ratio = pca2.explained_variance_ratio_
eig_vals = pca2.explained_variance_
sorted_eigenvalues = sorted(eig_vals, reverse=True)
```

```
cumulative_var_ratio = np.cumsum(explained_variance_ratio)*100
print(cumulative_var_ratio)

plt.scatter(pc[:, 0], pc[:, 1])
plt.xlabel('Principal Component 1')
plt.ylabel('Principal Component 2')
plt.title('2D Principal Component Analysis')
plt.show()

plot_pca(pc, labels_mapped)

#kmeans with 2 clusters
cluster_num = 2
kmeans = KMeans(n_clusters=cluster_num, random_state=22, n_init='auto')
cluster_predictions = kmeans.fit_predict(pc)
cluster_mapping_score = cluster_accuracy(labels_mapped, cluster_predictions,
cluster_num=cluster_num)
print(f"Cluster Mapping Method Accuracy: {cluster_mapping_score}")
plot_clusters(pc, cluster_predictions)

#Try out different k numbers for K-Means
for i in range(2,8):
    print(f"K-Means with {i + 1} Clusters")
    cluster_num = i + 1
    kmeans = KMeans(n_clusters=cluster_num, random_state=22, n_init='auto')
    cluster_predictions = kmeans.fit_predict(pc)
    plot_clusters(pc, cluster_predictions)
    cluster_mapping_score = cluster_accuracy(labels_mapped,
cluster_predictions, cluster_num=cluster_num)
    print(f"Cluster Mapping Method Accuracy: {cluster_mapping_score}\n")

"""PART B"""

pca2 = PCA(n_components=10, random_state=510)
features_projected = pca2.fit_transform(features_scaled)
X_train, X_test, y_train, y_test = train_test_split(features_projected,
labels_mapped, test_size=0.3, random_state=777)

svc = SVC()
svc_pol = SVC()

parameters = {'kernel':('linear', 'rbf', 'sigmoid'), 'C':[1e-3,1e-2,1e-1,1e0],
'gamma':('scale', 'auto'), 'degree':[0]}
parameters_pol = {'kernel':['poly'], 'C':[1e-3,1e-2,1e-1,1e0],
'gamma':('scale', 'auto'), 'degree': [2,3,4,5,6]}

#cross validation
clf = GridSearchCV(svc, parameters, cv=3) #Grid Search
clf.fit(X_train, y_train)
```

```python
results = clf.cv_results_
sorted_indices = np.argsort(-results['mean_test_score'])

#cross validation
clf_pol = GridSearchCV(svc_pol, parameters_pol, cv=3) #Grid Search
clf_pol.fit(X_train, y_train)
results_pol = clf_pol.cv_results_
sorted_indices_pol = np.argsort(-results_pol['mean_test_score'])
table_data = []

# Print the accuracy values and parameters in descending order
for index in sorted_indices:
    mean_score = results['mean_test_score'][index]
    params = results['params'][index]
    row = [mean_score]
    for key in sorted(params.keys()):
        row.append(params[key])
    table_data.append(row)

# Print the accuracy values and parameters in descending order
for index in sorted_indices_pol:
    mean_score = results_pol['mean_test_score'][index]
    params = results_pol['params'][index]
    row = [mean_score]
    for key in sorted(params.keys()):
        row.append(params[key])
    table_data.append(row)

# Define the table headers
headers = ["Accuracy"] + sorted(results['params'][0].keys())

# Generate the table in Markdown format
table = tabulate(table_data, headers, tablefmt="pipe")

# Sort the table data based on accuracy in descending order
table_data.sort(reverse=True, key=lambda x: x[0])

# Create a DataFrame from the table data
df = pd.DataFrame(table_data, columns=headers)

# Sort the DataFrame based on accuracy in descending order
df = df.sort_values(by='Accuracy', ascending=False)

print(df)

# to convert df to html table
table_html = df.to_html(index=False)

""" Write the HTML table to a file, use for creating the table
```

```python
with open('table.html', 'w') as file:
    file.write(table_html)"""

mlp = MLP(max_iter=100000)

parameters_mlp = {'solver' : ('lbfgs', 'adam'), 'alpha':[1e-3,1e-2,1e-1,1e0],
'activation' : ('logistic','relu'), 'hidden_layer_sizes':
[(64),(4,4),(8,8),(16,16),(32,32)]}

#cross validation
search = GridSearchCV(mlp, parameters_mlp, cv=3) #Grid Search
search.fit(X_train, y_train)
results_mlp = search.cv_results_
sorted_indices_mlp = np.argsort(-results_mlp['mean_test_score'])
table_data_mlp = []

# Print the accuracy values and parameters in descending order
for index in sorted_indices_mlp:
    mean_score = results_mlp['mean_test_score'][index]
    params = results_mlp['params'][index]
    row = [mean_score]
    for key in sorted(params.keys()):
        row.append(params[key])
    table_data_mlp.append(row)

# Define the table headers and generate table
headers_mlp = ["Accuracy"] + sorted(results_mlp['params'][0].keys())
table_mlp = tabulate(table_data_mlp, headers_mlp, tablefmt="pipe")

# Sorting table based on descending accuracy
table_data_mlp.sort(reverse=True, key=lambda x: x[0])

# Create dataframe for better visualization
df_mlp = pd.DataFrame(table_data_mlp, columns=headers_mlp)

df_mlp = df_mlp.sort_values(by='Accuracy', ascending=False)

print(df_mlp)

# to convert df to html table
table_html_mlp = df_mlp.to_html(index=False)

#Write the HTML table to a file
"""with open('table_mlp.html', 'w') as file:
    file.write(table_html_mlp)"""

"""TEST"""

svc_final = SVC(C=1,gamma='scale',kernel='rbf')
```

```
mlp_final =
MLP(activation='relu',alpha=0.1,hidden_layer_sizes=(32,32),solver='lbfgs',
max_iter=100000)
svc_final = svc_final.fit(X_train,y_train)
mlp_final = mlp_final.fit(X_train,y_train)
svc_predictions = svc_final.predict(X_test)
mlp_predictions = mlp_final.predict(X_test)
print(f"Accuracy of SVC on Test Dataset:
{metrics.accuracy_score(y_test,svc_predictions)}")
print(f"Accuracy of MLP on Test Dataset:
{metrics.accuracy_score(y_test,mlp_predictions)}")
```