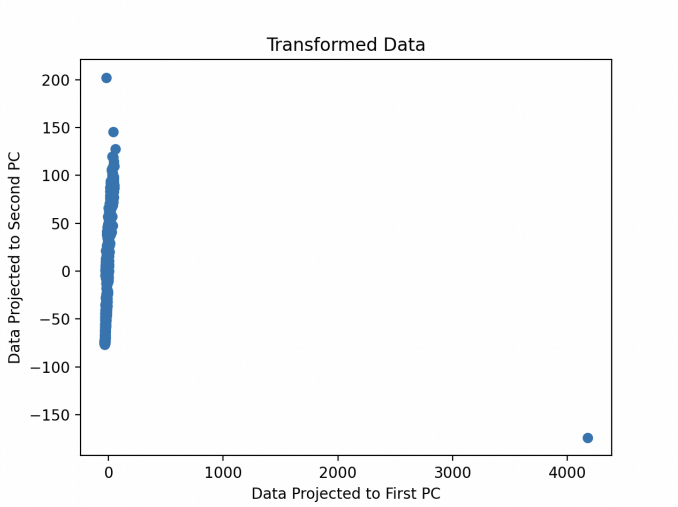
Bilkent University - GE461 Introduction to Data Science - Project 4 Fall Detection - Report

Kaan Tek - 21901946

Part A

Before doing any preprocessing such as looking for missing values, scaling, examining outliers, I first tried to apply PCA and map the data onto the first two principal components (PC). However, as can be seen from the below figure, the visual results were not meaningful, and the algorithm stated that the first PC explained %75 of the variance just by itself, and a total of %83 with both the first and second PC combined, which again is a questionable statistic.

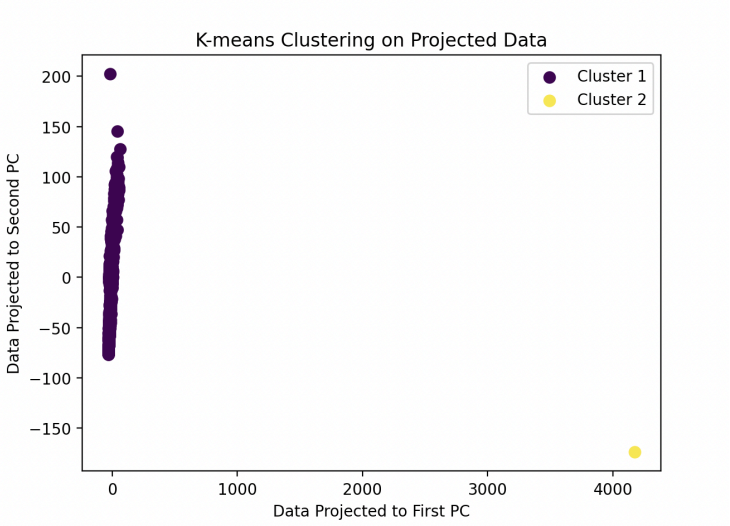



```
Cumulative variance explained before any preprocessing
[75.30724809 83.81883815]
```

Still, I wanted to apply an initial 2-means clustering in order to further examine the results. In this project, since we are dealing with only 2 labels and more than 2 clusters, and the clusters itself do not infer which label they refer to, I used the Majority Voting algorithm to assign labels to clusters and evaluate the algorithms.

Majority voting is a method used to assign a class label to data points based on the most common label within the cluster, improving the interpretability of unsupervised learning methods such as clustering. In this approach, each cluster formed by the algorithm is analyzed to determine which class label (Fall or Non-Fall') appears most frequently among the points within that cluster. This dominant label is then assigned to all points in the cluster for evaluating the clustering's accuracy. Then the accuracy is calculated by comparing these assigned labels to the true labels of the data points.

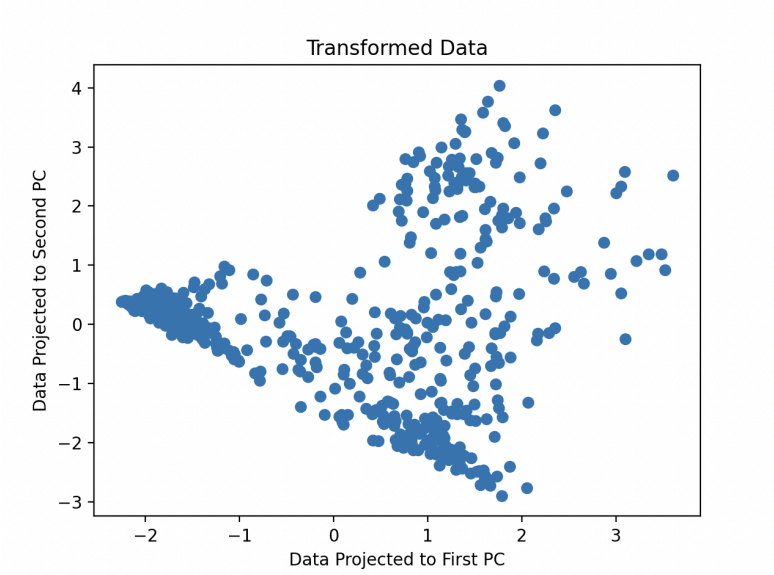
With these approach, running the 2-means clustering algorithm on the the unprepossessed data gave the below results:




```
Accuracy of the k-means before preprocessing (2 clusters): 0.5530
```

As can be seen, the results are not satisfactory, yielding only 0.55 accuracy. Therefore, I decided to apply some preprocessing on our data.

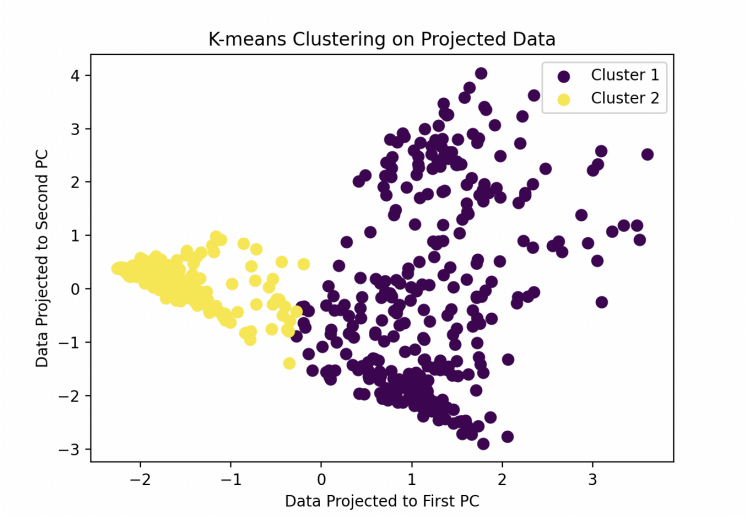
One might notice upon looking at the above graphs, there are 2 outliers in our data, after the projection onto the first two PCs. One being on the top left and the other on the bottom right. I first removed those outliers, since there were only 2. Then, upon further examination on our data, I noticed that the scale of the features were far different from each other. Therefore, I applied min-max scaling, in order to prevent the bias resulting from different scales, and allowing gradient descent a faster convergence. Upon completing preprocessing, I re-run the PCA algorithm to project our data onto first two components:




```
Cumulative variance explained after preprocessing
[26.64838968 48.71447026]
```

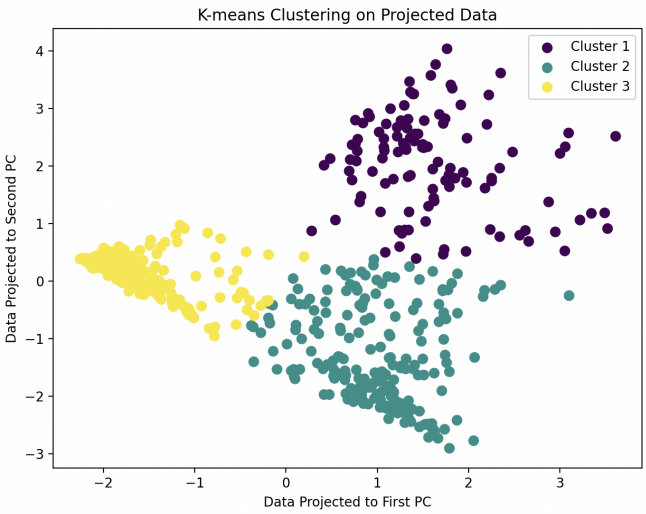
As can be seen, we now obtained a much better visualization with a more reasonable explained variance. Now the first PC explains %26.6 of the total variance, and a total of %48.8 of the variance with first and the second PC combined.

Now re-running our k-means clustering algorithm with 2 clusters, we obtain the below results:

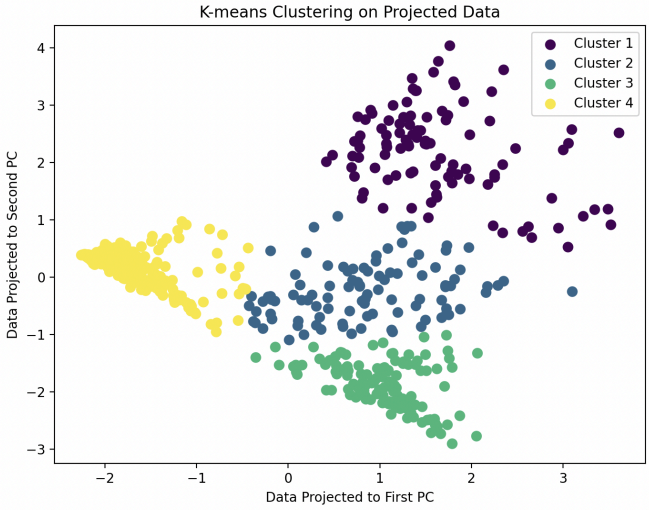



```
Accuracy of the k-means after preprocessing (2 clusters): 0.8156
```

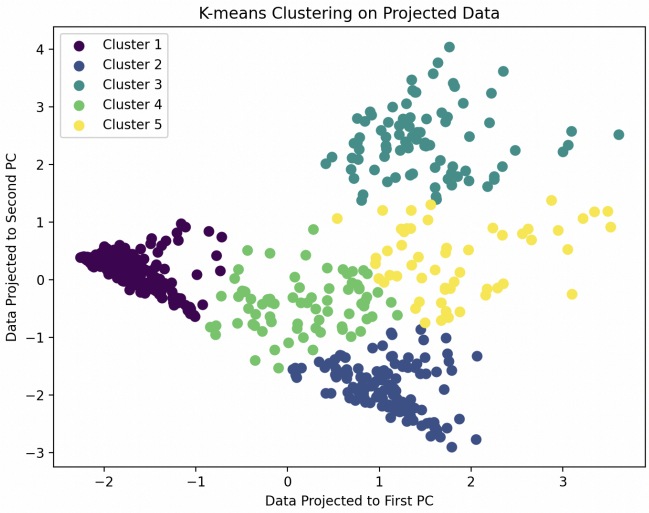
This indicates that only by using 2 clusters after carefully preprocessing our data, we are able to achieve 0.82 accuracy, from which can be stated that Fall Detection is possible to a great degree. Then I re-ran our k-means clustering algorithm for different number of clusters (values of *k=3,4,5,6,7,8,9,10*):



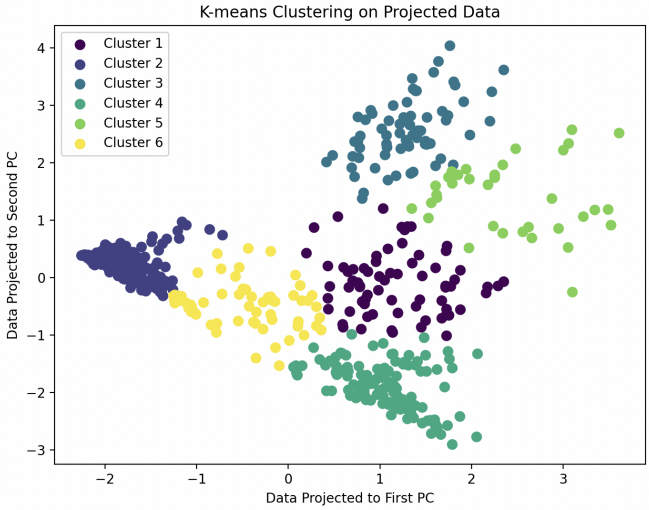

```
Accuracy of the k-means with 3 Clusters: 0.8174
```



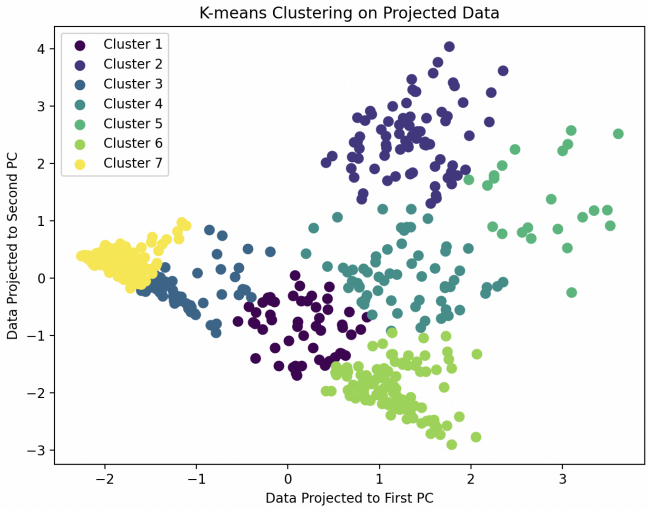

```
Accuracy of the k-means with 4 Clusters: 0.8014
```



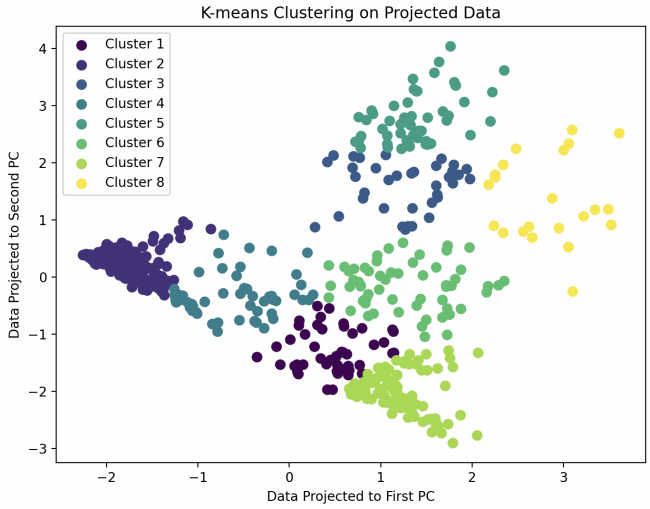

```
Accuracy of the k-means with 5 Clusters: 0.8280
```



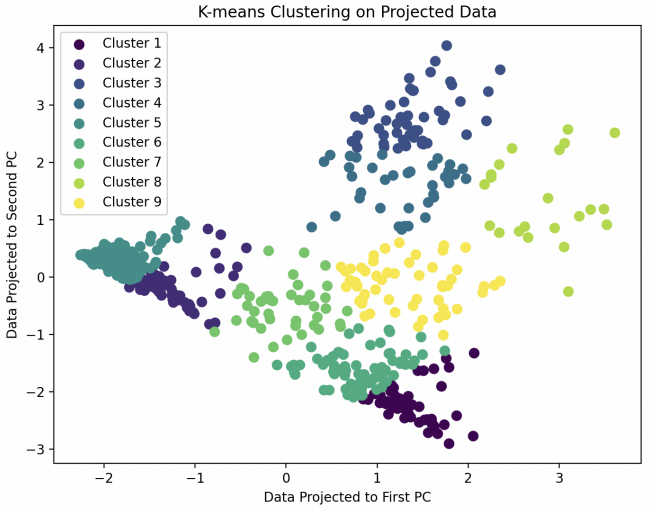

```
Accuracy of the k-means with 6 Clusters: 0.8546
```



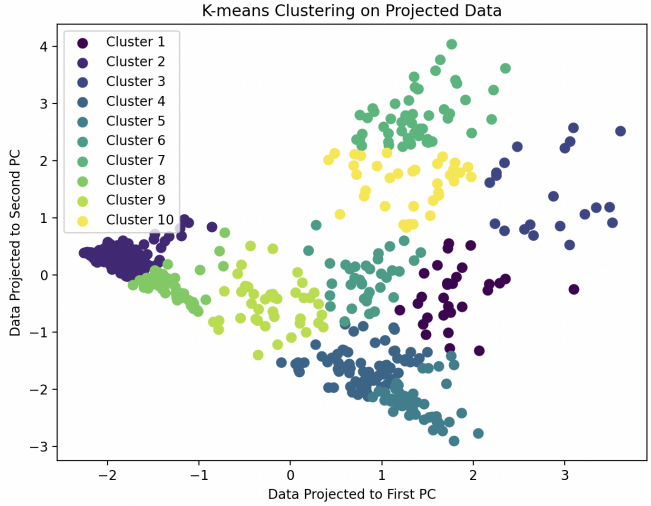

```
Accuracy of the k-means with 7 Clusters: 0.8688
```




```
Accuracy of the k-means with 8 Clusters: 0.8511
```




```
Accuracy of the k-means with 9 Clusters: 0.8528
```




```
Accuracy of the k-means with 10 Clusters: 0.8528
```

From the above results, it can be inferred that no matter the number of clusters, we can be confident in detecting Falls with over an %80 percent accuracy. In order to obtain the best results, one should consider using at least 6 clusters, because from that point we see an increase in the accuracy up to %85 accuracy. The reason behind this might be that starting from k=6, we observe new little separate clusters at the very center of our data points, whereas this part consisted of only one cluster for lower k values. Most likely, that region consists of overlapping F and NF values, and for higher k values, we are able to

detect that separation. Specifically, we obtained the best accuracy of %86.88 when using 7 clusters, and hence, it can be said that 7 is the optimal value for our hyperparameter k.

Part B

Before diving into implementation of SVM and MLP models, I applied PCA to the dataset, because of our high feature count and limited sample size of 500. After my experiments, I observed that using only the first 19 PCs was enough for capturing %90 of the total variance, which not only reduced complexity but also enhanced computational efficiency. The dataset was then split into 70% for training and 30% for testing. For validation, I used a 3-fold cross-validation method by using the GridSearchCV of scikit-learn, meaning the training set was divided into three parts, with two parts used for training and the third for validation, repeated three times. Using GridSearchCV helped to easily apply this cross validation procedure, while exhaustively testing all combinations of the hyperparameters.

Part B.1: SVM

During the training & validation process of my SVM model, I experimented with the combinations of the below hyperparameters:

```
     params = {'C': [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1], 'kernel':
('linear', 'sigmoid', 'rbf'), 'gamma': ('scale', 'auto'), 'degree': [0]}
     poly_params = {'C': [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1],
'kernel': ['poly'],  'gamma':('scale', 'auto'), 'degree': [1,2,3,4,5]}
```

First of all, the reason I have introduced a new variable **`poly_params`** is because the degree parameter impacts only the polynomial (poly) kernel, as noted in the scikit-learn documentation. Consequently, I created this variable to separately compute the poly kernel from other kernels, ensuring that different degrees are not redundantly computed for kernels where the degree parameter is irrelevant.

- The regularization parameter C was tested for values 0.001, 0.005, 0.01, 0.05, 0.1, 0.5, and 1
- Three different kernel types were tested: linear, sigmoid, rbf, poly

- Two gamma values were tested for adjusting the kernel coefficients: scale and auto
- Degrees 1, 2, 3, 4, and 5 were tested, specifically for the polynomial kernel.

As a result, the below accuracies were obtained for different hyperparameter combinations:

```
Accuracy   | C      | degree    | gamma      | kernel
-------------------------------------------------------------------
0.99747475 | C: 0.5 | degree: 0 | gamma: auto | kernel: linear
0.99747475 | C: 1 | degree: 0 | gamma: auto | kernel: linear
0.99747475 | C: 1 | degree: 0 | gamma: scale | kernel: linear
0.99747475 | C: 0.5 | degree: 0 | gamma: scale | kernel: linear
0.99493022 | C: 1 | degree: 0 | gamma: scale | kernel: rbf
0.99491094 | C: 1 | degree: 0 | gamma: auto | kernel: rbf
0.99238569 | C: 0.1 | degree: 0 | gamma: auto | kernel: linear
0.99238569 | C: 0.5 | degree: 0 | gamma: scale | kernel: rbf
0.99238569 | C: 0.1 | degree: 0 | gamma: scale | kernel: linear
0.99238569 | C: 1 | degree: 1 | gamma: auto | kernel: poly
0.99238569 | C: 1 | degree: 1 | gamma: scale | kernel: poly
0.99238569 | C: 0.5 | degree: 1 | gamma: scale | kernel: poly
0.98986044 | C: 0.05 | degree: 0 | gamma: auto | kernel: linear
0.98986044 | C: 0.05 | degree: 0 | gamma: scale | kernel: linear
0.98984116 | C: 0.5 | degree: 0 | gamma: auto | kernel: rbf
0.98731591 | C: 1 | degree: 0 | gamma: auto | kernel: sigmoid
0.97968232 | C: 1 | degree: 3 | gamma: scale | kernel: poly
0.97206801 | C: 0.5 | degree: 3 | gamma: scale | kernel: poly
0.96703678 | C: 0.5 | degree: 0 | gamma: scale | kernel: sigmoid
0.96703678 | C: 1 | degree: 2 | gamma: scale | kernel: poly
0.96701750 | C: 1 | degree: 0 | gamma: scale | kernel: sigmoid
0.95689722 | C: 0.5 | degree: 2 | gamma: scale | kernel: poly
0.95431413 | C: 0.5 | degree: 1 | gamma: auto | kernel: poly
0.95176960 | C: 0.5 | degree: 0 | gamma: auto | kernel: sigmoid
0.92645925 | C: 1 | degree: 4 | gamma: scale | kernel: poly
0.91626186 | C: 0.1 | degree: 0 | gamma: scale | kernel: rbf
0.90116817 | C: 1 | degree: 5 | gamma: scale | kernel: poly
0.90111034 | C: 0.5 | degree: 2 | gamma: auto | kernel: poly
0.90101396 | C: 0.1 | degree: 3 | gamma: scale | kernel: poly
0.89848870 | C: 1 | degree: 3 | gamma: auto | kernel: poly
0.89087439 | C: 0.1 | degree: 1 | gamma: scale | kernel: poly
0.89085512 | C: 0.05 | degree: 3 | gamma: scale | kernel: poly
0.88326008 | C: 0.1 | degree: 0 | gamma: scale | kernel: sigmoid
0.87822885 | C: 0.5 | degree: 4 | gamma: scale | kernel: poly
0.87819030 | C: 0.01 | degree: 0 | gamma: scale | kernel: linear
0.87819030 | C: 0.01 | degree: 0 | gamma: auto | kernel: linear
```

0.87059527 | C: 1 | degree: 2 | gamma: auto | kernel: poly
0.87057599 | C: 0.05 | degree: 1 | gamma: scale | kernel: poly
0.86560259 | C: 0.5 | degree: 5 | gamma: scale | kernel: poly
0.86556404 | C: 0.1 | degree: 2 | gamma: scale | kernel: poly
0.86296168 | C: 0.005 | degree: 0 | gamma: auto | kernel: linear
0.86296168 | C: 0.005 | degree: 0 | gamma: scale | kernel: linear
0.86296168 | C: 0.1 | degree: 0 | gamma: auto | kernel: sigmoid
0.86296168 | C: 0.1 | degree: 1 | gamma: auto | kernel: poly
0.86294240 | C: 0.1 | degree: 0 | gamma: auto | kernel: rbf
0.86039787 | C: 0.05 | degree: 0 | gamma: scale | kernel: sigmoid
0.83516462 | C: 0.1 | degree: 4 | gamma: scale | kernel: poly
0.83504896 | C: 0.05 | degree: 0 | gamma: scale | kernel: rbf
0.82999846 | C: 0.05 | degree: 0 | gamma: auto | kernel: rbf
0.82743465 | C: 0.05 | degree: 1 | gamma: auto | kernel: poly
0.82489012 | C: 0.05 | degree: 0 | gamma: auto | kernel: sigmoid
0.80430257 | C: 0.05 | degree: 5 | gamma: scale | kernel: poly
0.79204256 | C: 0.1 | degree: 5 | gamma: scale | kernel: poly
0.72578842 | C: 0.5 | degree: 3 | gamma: auto | kernel: poly
0.65978487 | C: 0.05 | degree: 4 | gamma: scale | kernel: poly
0.62934690 | C: 0.05 | degree: 2 | gamma: scale | kernel: poly
0.59135246 | C: 1 | degree: 4 | gamma: auto | kernel: poly
0.58882720 | C: 0.01 | degree: 0 | gamma: auto | kernel: sigmoid
0.58882720 | C: 0.01 | degree: 0 | gamma: scale | kernel: rbf
0.58882720 | C: 0.01 | degree: 0 | gamma: scale | kernel: sigmoid
0.58882720 | C: 0.01 | degree: 0 | gamma: auto | kernel: rbf
0.58882720 | C: 0.005 | degree: 0 | gamma: auto | kernel: rbf
0.58882720 | C: 0.005 | degree: 0 | gamma: auto | kernel: sigmoid
0.58882720 | C: 0.005 | degree: 0 | gamma: scale | kernel: rbf
0.58882720 | C: 0.005 | degree: 0 | gamma: scale | kernel: sigmoid
0.58882720 | C: 0.001 | degree: 0 | gamma: auto | kernel: rbf
0.58882720 | C: 0.001 | degree: 0 | gamma: auto | kernel: sigmoid
0.58882720 | C: 0.001 | degree: 0 | gamma: auto | kernel: linear
0.58882720 | C: 0.001 | degree: 0 | gamma: scale | kernel: rbf
0.58882720 | C: 0.001 | degree: 0 | gamma: scale | kernel: sigmoid
0.58882720 | C: 0.001 | degree: 0 | gamma: scale | kernel: linear
0.58882720 | C: 0.005 | degree: 5 | gamma: scale | kernel: poly
0.58882720 | C: 0.005 | degree: 3 | gamma: scale | kernel: poly
0.58882720 | C: 0.005 | degree: 3 | gamma: auto | kernel: poly
0.58882720 | C: 0.005 | degree: 4 | gamma: scale | kernel: poly
0.58882720 | C: 0.005 | degree: 2 | gamma: scale | kernel: poly
0.58882720 | C: 0.005 | degree: 1 | gamma: auto | kernel: poly
0.58882720 | C: 0.005 | degree: 4 | gamma: auto | kernel: poly
0.58882720 | C: 0.005 | degree: 2 | gamma: auto | kernel: poly
0.58882720 | C: 0.001 | degree: 3 | gamma: auto | kernel: poly
0.58882720 | C: 0.005 | degree: 1 | gamma: scale | kernel: poly
0.58882720 | C: 0.001 | degree: 5 | gamma: auto | kernel: poly
0.58882720 | C: 0.001 | degree: 5 | gamma: scale | kernel: poly

```
0.58882720 | C: 0.001 | degree: 4 | gamma: auto | kernel: poly
0.58882720 | C: 0.001 | degree: 4 | gamma: scale | kernel: poly
0.58882720 | C: 0.01 | degree: 1 | gamma: scale | kernel: poly
0.58882720 | C: 0.001 | degree: 3 | gamma: scale | kernel: poly
0.58882720 | C: 0.001 | degree: 2 | gamma: auto | kernel: poly
0.58882720 | C: 0.001 | degree: 2 | gamma: scale | kernel: poly
0.58882720 | C: 0.001 | degree: 1 | gamma: auto | kernel: poly
0.58882720 | C: 0.005 | degree: 5 | gamma: auto | kernel: poly
0.58882720 | C: 1 | degree: 5 | gamma: auto | kernel: poly
0.58882720 | C: 0.01 | degree: 1 | gamma: auto | kernel: poly
0.58882720 | C: 0.01 | degree: 2 | gamma: scale | kernel: poly
0.58882720 | C: 0.5 | degree: 5 | gamma: auto | kernel: poly
0.58882720 | C: 0.5 | degree: 4 | gamma: auto | kernel: poly
0.58882720 | C: 0.1 | degree: 5 | gamma: auto | kernel: poly
0.58882720 | C: 0.1 | degree: 4 | gamma: auto | kernel: poly
0.58882720 | C: 0.1 | degree: 3 | gamma: auto | kernel: poly
0.58882720 | C: 0.1 | degree: 2 | gamma: auto | kernel: poly
0.58882720 | C: 0.05 | degree: 5 | gamma: auto | kernel: poly
0.58882720 | C: 0.05 | degree: 4 | gamma: auto | kernel: poly
0.58882720 | C: 0.05 | degree: 3 | gamma: auto | kernel: poly
0.58882720 | C: 0.05 | degree: 2 | gamma: auto | kernel: poly
0.58882720 | C: 0.01 | degree: 5 | gamma: auto | kernel: poly
0.58882720 | C: 0.01 | degree: 5 | gamma: scale | kernel: poly
0.58882720 | C: 0.01 | degree: 4 | gamma: auto | kernel: poly
0.58882720 | C: 0.01 | degree: 4 | gamma: scale | kernel: poly
0.58882720 | C: 0.01 | degree: 3 | gamma: auto | kernel: poly
0.58882720 | C: 0.01 | degree: 3 | gamma: scale | kernel: poly
0.58882720 | C: 0.01 | degree: 2 | gamma: auto | kernel: poly
0.58882720 | C: 0.001 | degree: 1 | gamma: scale | kernel: poly
```

From the above results, we can infer that for a few of the combinations we obtain a very high accuracy of `0.99747475`, while for most of the combinations we can confidently get over accuracy `0.90`. From the above results, I chose the SVM with the configurations that have *C=0.5, degree=0, gamma='auto'*, and *kernel='linear'* as the best performing model, and re-trained it on the training data, and then tested on the test data, in order to obtain the final performance:

```
Best SVM configuration: C=0.5, degree=0, gamma='auto', kernel='linear'

Final Accuracy of the best SVM model on Test Dataset: 1.0
```

As can be seen, we obtained an accuracy of %100 with this configuration, which is an exceptional accomplishment.

Part B.2: MLP

During the training & validation process of my MLP model, I experimented with the combinations of the below hyperparameters:

```
    mlp_params = {'hidden_layer_sizes':
[(2,2),(4,4),(8,8),(16,16),(32,32),(64,64)], 'solver' : ('sgd', 'adam'),
'alpha':[0.001, 0.005, 0.01, 0.05, 0.1], 'activation' :
('logistic','relu')}
```

- Tested with 5 different values of alpha hyperparameter, which is used in the L2 regularization: 0.001, 0.005, 0.01, 0.05, and 0.1.
- Different numbers of hidden layer sizes were tested to evaluate the model complexity: (2,2), (4,4), (8,8), (16,16), (32,32), (64,64).
- Adam and SGD solvers were tested, which are used for efficiently updating the model weights.
- ReLu and Logistic activations were used to introduce non-linearity into our model.

As a result, the below accuracies were obtained for different hyperparameter combinations:

```
Accuracy   | activation       | alpha     | hidden_layer_sizes       | solver
-------------------------------------------------------------------------------
1.00000000 | activation: logistic | alpha: 0.01 | hidden_layer_sizes: (16, 16) | solver: adam
1.00000000 | activation: relu | alpha: 0.1 | hidden_layer_sizes: (8, 8) | solver: adam
1.00000000 | activation: logistic | alpha: 0.005 | hidden_layer_sizes: (2, 2) | solver: adam
0.99747475 | activation: relu | alpha: 0.1 | hidden_layer_sizes: (64, 64) | solver: adam
0.99747475 | activation: logistic | alpha: 0.05 | hidden_layer_sizes: (64, 64) | solver: adam
0.99747475 | activation: relu | alpha: 0.01 | hidden_layer_sizes: (4, 4) | solver: adam
0.99747475 | activation: logistic | alpha: 0.01 | hidden_layer_sizes: (64, 64) | solver: adam
0.99747475 | activation: logistic | alpha: 0.01 | hidden_layer_sizes: (8, 8) | solver: adam
0.99747475 | activation: relu | alpha: 0.005 | hidden_layer_sizes: (64, 64) | solver: adam
0.99747475 | activation: relu | alpha: 0.001 | hidden_layer_sizes: (8, 8) | solver: adam
0.99747475 | activation: logistic | alpha: 0.005 | hidden_layer_sizes: (64, 64) | solver: adam
0.99747475 | activation: logistic | alpha: 0.005 | hidden_layer_sizes: (32, 32) | solver: adam
0.99747475 | activation: logistic | alpha: 0.005 | hidden_layer_sizes: (16, 16) | solver: adam
0.99747475 | activation: relu | alpha: 0.05 | hidden_layer_sizes: (8, 8) | solver: adam
0.99747475 | activation: relu | alpha: 0.01 | hidden_layer_sizes: (2, 2) | solver: adam
```

0.99747475 | activation: logistic | alpha: 0.005 | hidden_layer_sizes: (8, 8) | solver: adam
0.99747475 | activation: logistic | alpha: 0.01 | hidden_layer_sizes: (32, 32) | solver: adam
0.99747475 | activation: relu | alpha: 0.001 | hidden_layer_sizes: (32, 32) | solver: adam
0.99747475 | activation: logistic | alpha: 0.005 | hidden_layer_sizes: (4, 4) | solver: adam
0.99747475 | activation: logistic | alpha: 0.001 | hidden_layer_sizes: (8, 8) | solver: adam
0.99747475 | activation: relu | alpha: 0.05 | hidden_layer_sizes: (64, 64) | solver: adam
0.99747475 | activation: relu | alpha: 0.1 | hidden_layer_sizes: (2, 2) | solver: adam
0.99747475 | activation: logistic | alpha: 0.001 | hidden_layer_sizes: (64, 64) | solver: adam
0.99747475 | activation: logistic | alpha: 0.001 | hidden_layer_sizes: (16, 16) | solver: adam
0.99747475 | activation: logistic | alpha: 0.001 | hidden_layer_sizes: (32, 32) | solver: adam
0.99745547 | activation: relu | alpha: 0.1 | hidden_layer_sizes: (16, 16) | solver: adam
0.99494949 | activation: relu | alpha: 0.005 | hidden_layer_sizes: (16, 16) | solver: adam
0.99493022 | activation: logistic | alpha: 0.05 | hidden_layer_sizes: (32, 32) | solver: adam
0.99493022 | activation: relu | alpha: 0.001 | hidden_layer_sizes: (64, 64) | solver: adam
0.99493022 | activation: logistic | alpha: 0.05 | hidden_layer_sizes: (2, 2) | solver: adam
0.99493022 | activation: logistic | alpha: 0.1 | hidden_layer_sizes: (8, 8) | solver: adam
0.99493022 | activation: logistic | alpha: 0.05 | hidden_layer_sizes: (8, 8) | solver: adam
0.99493022 | activation: logistic | alpha: 0.05 | hidden_layer_sizes: (16, 16) | solver: adam
0.99493022 | activation: relu | alpha: 0.005 | hidden_layer_sizes: (32, 32) | solver: adam
0.99493022 | activation: logistic | alpha: 0.01 | hidden_layer_sizes: (4, 4) | solver: adam
0.99493022 | activation: relu | alpha: 0.01 | hidden_layer_sizes: (8, 8) | solver: adam
0.99493022 | activation: relu | alpha: 0.01 | hidden_layer_sizes: (64, 64) | solver: adam
0.99493022 | activation: relu | alpha: 0.1 | hidden_layer_sizes: (4, 4) | solver: adam
0.99493022 | activation: relu | alpha: 0.05 | hidden_layer_sizes: (32, 32) | solver: adam
0.99493022 | activation: relu | alpha: 0.05 | hidden_layer_sizes: (32, 32) | solver: sgd
0.99493022 | activation: relu | alpha: 0.001 | hidden_layer_sizes: (4, 4) | solver: adam
0.99493022 | activation: relu | alpha: 0.01 | hidden_layer_sizes: (64, 64) | solver: sgd
0.99493022 | activation: relu | alpha: 0.01 | hidden_layer_sizes: (32, 32) | solver: adam
0.99491094 | activation: relu | alpha: 0.05 | hidden_layer_sizes: (2, 2) | solver: sgd
0.99240497 | activation: relu | alpha: 0.05 | hidden_layer_sizes: (8, 8) | solver: sgd
0.99240497 | activation: relu | alpha: 0.01 | hidden_layer_sizes: (16, 16) | solver: adam
0.99240497 | activation: relu | alpha: 0.005 | hidden_layer_sizes: (32, 32) | solver: sgd
0.99240497 | activation: relu | alpha: 0.1 | hidden_layer_sizes: (8, 8) | solver: sgd
0.99240497 | activation: relu | alpha: 0.001 | hidden_layer_sizes: (16, 16) | solver: adam
0.99240497 | activation: relu | alpha: 0.05 | hidden_layer_sizes: (16, 16) | solver: adam
0.99238569 | activation: logistic | alpha: 0.05 | hidden_layer_sizes: (4, 4) | solver: adam
0.99238569 | activation: logistic | alpha: 0.1 | hidden_layer_sizes: (4, 4) | solver: adam
0.99238569 | activation: logistic | alpha: 0.1 | hidden_layer_sizes: (16, 16) | solver: adam
0.99238569 | activation: logistic | alpha: 0.1 | hidden_layer_sizes: (32, 32) | solver: adam
0.99238569 | activation: logistic | alpha: 0.1 | hidden_layer_sizes: (64, 64) | solver: adam
0.99238569 | activation: relu | alpha: 0.05 | hidden_layer_sizes: (4, 4) | solver: adam
0.99238569 | activation: relu | alpha: 0.1 | hidden_layer_sizes: (32, 32) | solver: sgd
0.99238569 | activation: relu | alpha: 0.1 | hidden_layer_sizes: (32, 32) | solver: adam
0.99238569 | activation: relu | alpha: 0.01 | hidden_layer_sizes: (32, 32) | solver: sgd
0.99238569 | activation: relu | alpha: 0.001 | hidden_layer_sizes: (64, 64) | solver: sgd
0.98987971 | activation: relu | alpha: 0.1 | hidden_layer_sizes: (16, 16) | solver: sgd
0.98986044 | activation: relu | alpha: 0.05 | hidden_layer_sizes: (64, 64) | solver: sgd
0.98986044 | activation: relu | alpha: 0.005 | hidden_layer_sizes: (8, 8) | solver: adam
0.98986044 | activation: relu | alpha: 0.005 | hidden_layer_sizes: (64, 64) | solver: sgd
0.98986044 | activation: relu | alpha: 0.001 | hidden_layer_sizes: (32, 32) | solver: sgd
0.98986044 | activation: relu | alpha: 0.05 | hidden_layer_sizes: (4, 4) | solver: sgd
0.98984116 | activation: relu | alpha: 0.01 | hidden_layer_sizes: (16, 16) | solver: sgd
0.98984116 | activation: relu | alpha: 0.005 | hidden_layer_sizes: (16, 16) | solver: sgd
0.98735446 | activation: relu | alpha: 0.005 | hidden_layer_sizes: (4, 4) | solver: adam
0.98733518 | activation: relu | alpha: 0.01 | hidden_layer_sizes: (8, 8) | solver: sgd
0.98733518 | activation: relu | alpha: 0.05 | hidden_layer_sizes: (2, 2) | solver: adam
0.98733518 | activation: relu | alpha: 0.005 | hidden_layer_sizes: (8, 8) | solver: sgd
0.98731591 | activation: relu | alpha: 0.1 | hidden_layer_sizes: (4, 4) | solver: sgd

```
0.98731591 | activation: relu | alpha: 0.001 | hidden_layer_sizes: (16, 16) | solver: sgd
0.98731591 | activation: relu | alpha: 0.05 | hidden_layer_sizes: (16, 16) | solver: sgd
0.98729663 | activation: relu | alpha: 0.001 | hidden_layer_sizes: (4, 4) | solver: sgd
0.98479065 | activation: relu | alpha: 0.1 | hidden_layer_sizes: (64, 64) | solver: sgd
0.98477138 | activation: relu | alpha: 0.01 | hidden_layer_sizes: (4, 4) | solver: sgd
0.98477138 | activation: relu | alpha: 0.001 | hidden_layer_sizes: (8, 8) | solver: sgd
0.97968232 | activation: relu | alpha: 0.005 | hidden_layer_sizes: (4, 4) | solver: sgd
0.97201018 | activation: relu | alpha: 0.01 | hidden_layer_sizes: (2, 2) | solver: sgd
0.96439587 | activation: relu | alpha: 0.1 | hidden_layer_sizes: (2, 2) | solver: sgd
0.94912869 | activation: relu | alpha: 0.005 | hidden_layer_sizes: (2, 2) | solver: sgd
0.86363636 | activation: logistic | alpha: 0.01 | hidden_layer_sizes: (2, 2) | solver: adam
0.86259542 | activation: logistic | alpha: 0.001 | hidden_layer_sizes: (4, 4) | solver: adam
0.86007017 | activation: logistic | alpha: 0.001 | hidden_layer_sizes: (2, 2) | solver: adam
0.85500039 | activation: relu | alpha: 0.001 | hidden_layer_sizes: (2, 2) | solver: adam
0.84740535 | activation: relu | alpha: 0.005 | hidden_layer_sizes: (2, 2) | solver: adam
0.82292390 | activation: relu | alpha: 0.001 | hidden_layer_sizes: (2, 2) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.001 | hidden_layer_sizes: (4, 4) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.005 | hidden_layer_sizes: (16, 16) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.005 | hidden_layer_sizes: (32, 32) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.005 | hidden_layer_sizes: (8, 8) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.001 | hidden_layer_sizes: (32, 32) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.001 | hidden_layer_sizes: (8, 8) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.001 | hidden_layer_sizes: (16, 16) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.005 | hidden_layer_sizes: (4, 4) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.005 | hidden_layer_sizes: (2, 2) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.001 | hidden_layer_sizes: (64, 64) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.01 | hidden_layer_sizes: (32, 32) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.005 | hidden_layer_sizes: (64, 64) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.01 | hidden_layer_sizes: (2, 2) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.1 | hidden_layer_sizes: (64, 64) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.1 | hidden_layer_sizes: (32, 32) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.1 | hidden_layer_sizes: (16, 16) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.1 | hidden_layer_sizes: (8, 8) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.1 | hidden_layer_sizes: (4, 4) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.1 | hidden_layer_sizes: (2, 2) | solver: adam
0.58882720 | activation: logistic | alpha: 0.1 | hidden_layer_sizes: (2, 2) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.05 | hidden_layer_sizes: (64, 64) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.05 | hidden_layer_sizes: (32, 32) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.05 | hidden_layer_sizes: (16, 16) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.05 | hidden_layer_sizes: (8, 8) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.05 | hidden_layer_sizes: (4, 4) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.05 | hidden_layer_sizes: (2, 2) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.01 | hidden_layer_sizes: (64, 64) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.01 | hidden_layer_sizes: (16, 16) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.01 | hidden_layer_sizes: (8, 8) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.01 | hidden_layer_sizes: (4, 4) | solver: sgd
0.58882720 | activation: logistic | alpha: 0.001 | hidden_layer_sizes: (2, 2) | solver: sgd
```

From the above results, we can infer that for a few of the combinations we obtain a very %100 accuracy, while for most of the combinations we can confidently get over accuracy `0.98`. From the above results, I chose the MLP with the configurations that have *activation=logistic, alpha=0.01, hidden_layer_sizes=(16,16)* , and *solver=adam* as the best performing model, and re-trained it on the training data, and then tested on the test data, in order to obtain the final performance:

```
Best MLP configuration: activation=logistic, alpha=0.01, hidden_layer_sizes=(16, 16), solver=adam
Final Accuracy of the best MLP model on Test Dataset: 0.9882352941176471
```

As can be seen, we obtained an accuracy of near %99 with this configuration of MLP, which again is a pretty strong result.

Conclusion

We can confidently say that we have obtained pretty good results for both our SVM and MLP models. For most of the hyperparameter configurations for both of the models, we can easily obtain an accuracy of over %90. For the MLP, more than half of the models that were configured with different hyperparameters were able to yield at least %98 accuracy. For the SVM, almost one third of the tested models were able to yield at least %90 accuracy. Only by looking at these statistics, it is safe to say that both of these models can be reliably used for Fall detection.

Even though the MLP models displayed an overall better performance, the chosen best model of SVM yielded a %100 accuracy on the test dataset, whereas the chosen best model of MLP yielded a near %99 for this dataset. Since the difference is very little, the fact that SVM performed better could simply be a result of luck, or the way data is arranged. Because of the overall better performance of the MLP models, for a different, maybe a more complex dataset, MLP could be the first option.

It should also be noted that, before running our models, we applied PCA and then only used the first 19 PCs. If we wanted to use more, we most likely would have obtained even better accuracies, but in return, we might have dealt with longer training times, especially for the MLP models.

To conclude, we achieved a fall prediction accuracy of over 98%, indicating our algorithms were successful after applying the necessary data preprocessing and finding the correct combinations of hyperparameters. It can be said that the necessary companies can confidently use this wearable-sensor data for fall detection, but one suggestion of mine would be to enlarge this dataset at hand, as having only 566 samples could still be misleading.

Appendix

Below is the full python code used in this project. Note that in the first part of the project, due to using matplotlib library, plots will appear as pop-up windows in some sections. In order for code to keep running, the user should close those pop-up windows. Also, make sure the dataset is in the same directory of the python file, in order to be able to successfully read the data.

```
# imports

import os
import pandas as pd
import numpy as np
import warnings

from os import path
from sklearn.model_selection import train_test_split
from sklearn import metrics

from sklearn.cluster import KMeans
from sklearn.decomposition import PCA
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import GridSearchCV
from matplotlib import pyplot as plt
from itertools import product
from sklearn.metrics import accuracy_score
from sklearn.neural_network import MLPClassifier
from sklearn.svm import SVC

NEW_LINE = "\n"
SEPERATOR = "\n----------------------------------\n"
warnings.filterwarnings('ignore')

'''
```

```
    DATA READ

    '''

    cur_dir = os.path.dirname(os.path.abspath(__file__))
    path = os.path.join(cur_dir,"falldetection_dataset.csv")
    df = pd.read_csv(path, header=None)

    #print(df.head())
    #print(SEPERATOR)



    X_df = df.drop([0,1], axis=1)
    y_df = df.iloc[:,1]
    #print(X_df.head())
    #print(NEW_LINE)
    #print(y_df.head())
    #print(SEPERATOR)

    X = X_df.to_numpy() # X is 2D numpy array, each item is row / motor
action
    y = y_df.to_numpy() # Y is a 1D numpy array, each item is the label
(F/NF)



    ''' PART A Functions '''

    def plot_clusters(pcs, y_kmeans, num_clusters=2, figsize=(8, 6)):

        plt.figure(figsize=figsize)
        colors = plt.cm.viridis(np.linspace(0, 1, num_clusters))  #
Generate a color map for the clusters

        for i in range(num_clusters):
```

```
            plt.scatter(pcs[y_kmeans == i, 0], pcs[y_kmeans == i, 1],
s=50, c=[colors[i]], label=f'Cluster {i+1}')

        plt.xlabel('Data Projected to First PC')
        plt.ylabel('Data Projected to Second PC')
        plt.title('K-means Clustering on Projected Data')
        plt.legend()
        plt.show()

    def calculate_majority_vote_accuracy(y, y_kmeans, num_clusters=2):
        label_map = {'F': 0, 'NF': 1}
        y_numeric = np.array([label_map[label] for label in y])

        # map cluster labels to the most frequent true label in each
cluster
        cluster_labels = np.zeros_like(y_kmeans)
        for cluster in range(num_clusters):
            # find all items in this cluster
            mask = (y_kmeans == cluster)
            if np.sum(mask) == 0:  # If no points in the cluster,
continue
                continue

            # count the occurrences of each class label in this cluster
            # return counts of unique elements sorted by element value
            labels, counts = np.unique(y_numeric[mask],
return_counts=True)
            majority_label = labels[np.argmax(counts)]  # The label with
the highest count

            # assign this majority label to all members of this cluster
            cluster_labels[mask] = majority_label

        # calculate and return the accuracy
        return accuracy_score(y_numeric, cluster_labels)
```

```
    '''

    PART A

    '''

    print("\n\n----------- PART A -----------\n")

    # use PCA and extract the top 2 PCs
    pca = PCA(n_components=2)
    pcs = pca.fit_transform(X)
    #print(pcs)

    explained_variance_ratio = pca.explained_variance_ratio_
    eig_vals = pca.explained_variance_
    eig_vals_sorted = sorted(eig_vals, reverse=True)
    cumulative_variance = np.cumsum(explained_variance_ratio)*100
    print("Cumulative variance explained before any preprocessing")
    print(cumulative_variance) # first PCA explains %75.3, first and
second %83.82

    plt.scatter(pcs[:, 0], pcs[:, 1])
    plt.xlabel('Data Projected to First PC')
    plt.ylabel('Data Projected to Second PC')
    plt.title('Transformed Data')
    plt.show() # not interpretable

    # let's try to run K-mean with this setting

    # Run K-means clustering
    num_clusters = 2
    kmeans = KMeans(n_clusters=num_clusters, random_state=42)
    y_kmeans = kmeans.fit_predict(pcs)

    accuracy = calculate_majority_vote_accuracy(y, y_kmeans)
```

```
    print(f'Accuracy of the k-means before preprocessing (2 clusters):
{accuracy:.4f}')
    print(NEW_LINE)

    # Plot the clusters
    plot_clusters(pcs, y_kmeans)

    # remove outliers
    out_1= np.argmax(pcs[:,0])
    out_2= np.argmax(pcs[:,1])

    X_new = np.delete(X, out_1, axis=0)
    X_new = np.delete(X_new, out_2, axis=0)
    y_new = np.delete(y, out_1, axis=0)
    y_new = np.delete(y_new, out_2, axis=0)

    # scale the new data
    scaler = MinMaxScaler()
    X_scaled = scaler.fit_transform(X_new)

    pca_new = PCA(n_components=2)
    pcs_new = pca_new.fit_transform(X_scaled)

    explained_variance_ratio = pca_new.explained_variance_ratio_
    eig_vals = pca_new.explained_variance_
    eig_vals_sorted = sorted(eig_vals, reverse=True)
    cumulative_var_ratio = np.cumsum(explained_variance_ratio)*100
    print("Cumulative variance explained after preprocessing")
    print(cumulative_var_ratio)

    plt.scatter(pcs_new[:, 0], pcs_new[:, 1])
    plt.xlabel('Data Projected to First PC')
    plt.ylabel('Data Projected to Second PC')
    plt.title('Transformed Data')
    plt.show()
```

```
    # Now re-run K-means clustering
    num_clusters = 2
    kmeans = KMeans(n_clusters=num_clusters, random_state=42)
    y_kmeans = kmeans.fit_predict(pcs_new)

    accuracy = calculate_majority_vote_accuracy(y_new, y_kmeans)
    print(f'Accuracy of the k-means after preprocessing (2 clusters):
{accuracy:.4f}')

    # plot the clusters
    plot_clusters(pcs_new, y_kmeans)

    # now we are ready to test with different k-values
    for i in range(3,11):
        print(NEW_LINE)
        print(f"K-Means with {i} Clusters")
        kmeans = KMeans(n_clusters=i, random_state=123)
        y_kmeans = kmeans.fit_predict(pcs_new)
        accuracy = calculate_majority_vote_accuracy(y_new, y_kmeans, i)
        print(f'Accuracy of the k-means with {i} Clusters:
{accuracy:.4f}')
        plot_clusters(pcs_new, y_kmeans, num_clusters=i)



    print("\n\n----------- PART B -----------\n")

    ''' PART B Functions '''

    # implements cross validation with Grid Search
    def cross_validator(estimator, params):
        clf = GridSearchCV(estimator, params, cv=3)
        clf.fit(X_train, y_train)
        return clf.cv_results_

    # print results
```

```
    def process_grid_search_results(report, results, mlp=False):
        display = []
        mlp_best = []
        idx = -1
        for i in results:
            idx += 1
            acc = report['mean_test_score'][i]
            params = report['params'][i]
            data = [f"{acc:.8f}"]
            for key in sorted(params.keys()):
                data.append(f"{key}: {params[key]}")
                if mlp and idx == 0:
                    mlp_best.append(params[key])
            display.append(' | '.join(data))
        if mlp:
            return display, mlp_best
        return display

    '''

    PART B

    '''

    print("\n------- B.1) SVM -------\n")

    # first apply PCA to reduce the number of features that are to be
used
    pca = PCA(n_components=19)
    pcs = pca.fit_transform(X_scaled)
    print('Varince explained with first 19 PCs: %' +
str(np.sum(pca.explained_variance_ratio_)*100))
    print(NEW_LINE)

    # map labels to binary nd perform the split
    y_binary = np.where(y_new == 'F', 1, 0)
```

```
    X_train, X_test, y_train, y_test = train_test_split(pcs, y_binary,
test_size=0.3, random_state=123)

    svm = SVC()
    svm_poly = SVC()

    params = {'C': [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1], 'kernel':
('linear', 'sigmoid', 'rbf'), 'gamma': ('scale', 'auto'), 'degree': [0]}
    poly_params = {'C': [0.001, 0.005, 0.01, 0.05, 0.1, 0.5, 1],
'kernel': ['poly'],  'gamma':('scale', 'auto'), 'degree': [1,2,3,4,5]}

    # perform 3-fold cross validation using Grid Search
    cv_report = cross_validator(svm, params)
    results_descending = np.argsort(cv_report['mean_test_score'])[::-1]

    poly_cv_report = cross_validator(svm_poly, poly_params)
    poly_results_descending=
np.argsort(poly_cv_report['mean_test_score'])[::-1]

    results = process_grid_search_results(cv_report, results_descending)
    results += process_grid_search_results(poly_cv_report,
poly_results_descending)

    # print results
    column_names = ["Accuracy"] + [key for key in
sorted(cv_report['params'][0].keys())]
    header_line = column_names[0] + "    | " + column_names[1] + "      |
" + column_names[2]
    header_line += "      | " + column_names[3] + "       | " +
column_names[4]
    print(header_line)
    print('-' * len(header_line))

    # print each row
    for row in sorted(results, reverse=True, key=lambda x:
float(x.split('|')[0].strip())):
```

```
        print(row)


    print("\n---- SVM BEST MODEL ----\n")

    svm_best = SVC(C=0.5,gamma='auto',kernel='linear',degree=0)
    svm_best = svm_best.fit(X_train, y_train)
    svm_result = svm_best.predict(X_test)
    print("Best SVM configuration: C=0.5, degree=0, gamma='auto',
kernel='linear'" + "\n")
    print(f"Final Accuracy of the best SVM model on Test Dataset:
{metrics.accuracy_score(y_test, svm_result)}")


    print("\n------- B.2) MLP -------\n")

    mlp = MLPClassifier(max_iter=100000)
    mlp_params = {'hidden_layer_sizes':
[(2,2),(4,4),(8,8),(16,16),(32,32),(64,64)], 'solver' : ('sgd', 'adam'),
'alpha':[0.001, 0.005, 0.01, 0.05, 0.1], 'activation' :
('logistic','relu')}

    mlp_cv_report = cross_validator(mlp, mlp_params)
    mlp_results_descending =
np.argsort(mlp_cv_report['mean_test_score'])[::-1]

    mlp_results, mlp_best_config =
process_grid_search_results(mlp_cv_report, mlp_results_descending, True)

    # print results
    column_names = ["Accuracy"] + [key for key in
sorted(mlp_cv_report['params'][0].keys())]
    header_line = column_names[0] + "   | " + column_names[1] + "
| " + column_names[2]
    header_line += "     | " + column_names[3] + "       | " +
column_names[4]
```

```
      print(header_line)
      print('-' * len(header_line))

      # print each row
      for row in sorted(mlp_results, reverse=True, key=lambda x:
float(x.split('|')[0].strip())):
         print(row)



      print("\n---- MLP BEST MODEL ----\n")
      print("Best MLP configuration: activation=" + mlp_best_config[0] +
", alpha=" + str(mlp_best_config[1])
           + ", hidden_layer_sizes=" + str(mlp_best_config[2]) + ",
solver=" + mlp_best_config[3] + "\n")

      mlp_best = MLPClassifier(activation=mlp_best_config[0],
alpha=mlp_best_config[1], hidden_layer_sizes=mlp_best_config[2],
solver=mlp_best_config[3], max_iter=100000)
      mlp_best = mlp_best.fit(X_train, y_train)
      mlp_result = mlp_best.predict(X_test)
      print(f"Final Accuracy of the best MLP model on Test Dataset:
{metrics.accuracy_score(y_test, mlp_result)}")
      print(NEW_LINE)
```