

Human Activity Recognition

Today smartphones are equipped with variety of sensors to enhance the user experience, 2 such sensors are accelerometer which measures acceleration and gyroscope which measures angular velocity. In this project we use the data provided by these sensors to predict the human posture and the activity the user is performing.

Data Information: 30 volunteers within an age bracket of 19-48 years were chosen and each person performed six activities (walking, walking-upstairs, walking-downstairs, standing, sitting, laying) while wearing the smartphone. Using the embedded sensors, 3-axial linear acceleration and 3-axial angular velocity were recorded at 50Hz which was then manually labelled. The dataset was randomly split in 70:30 ratio to obtain training and test datasets.

From the recorded signals, domain experts engineered a feature vector of size 561 using time and frequency domain variables and various metrics like mean, max, min etc.

Dataset link: https://drive.google.com/drive/folders/1Whk9g_S00RnIis_U3JXbl1rAJa2suV2?usp=sharing

Problem Statement: Using the 561 feature data, the goal is to predict one of the six activities tha a smartphone user is performing.

Loading Data:

```
#loading the dataset
train_df=pd.read_csv('train.csv')
test_df=pd.read_csv('test.csv')
train_df.head()
```

	tBodyAcc- mean()-X	tBodyAcc- mean()-Y	tBodyAcc- mean()-Z	tBodyAcc- std()-X	tBodyAcc- std()-Y	tBodyAcc- std()-Z	tBodyAcc- mad()-X	tBodyAcc- mad()-Y	tBodyAcc- mad()-Z	...	fBodyGyroJerkMag- kurtosis()	angle(tBodyAccMean,gravity)	angle(tBodyAccJerkMean,gravityMean)	angle(tBodyGyroMean,gr
0	0.288585	-0.028294	-0.132905	-0.995279	-0.983111	-0.913526	-0.995112	-0.983185	-0.923527	-0.934724	...	-0.710304	-0.112754	0.030400
1	0.278419	-0.016411	-0.123520	-0.998245	-0.975300	-0.960322	-0.998807	-0.974914	-0.957686	-0.943068	...	-0.861499	0.053477	-0.007435
2	0.279653	-0.019467	-0.113462	-0.995380	-0.967187	-0.978944	-0.996520	-0.963668	-0.977469	-0.938692	...	-0.760104	-0.118559	0.177899
3	0.279174	-0.026201	-0.123283	-0.996091	-0.983403	-0.990675	-0.997099	-0.982750	-0.989302	-0.938692	...	-0.482845	-0.036788	-0.012892
4	0.276629	-0.016570	-0.115362	-0.998139	-0.980817	-0.990482	-0.998321	-0.979672	-0.990441	-0.942469	...	-0.699205	0.123320	0.122542

5 rows x 563 columns

All values are between -1 and 1, so there are no outliers

```
[4] > MI
```

```
#addition info about the data, no need to normalise or feature scale as data is between -1 and 1
train_df.describe()
```

	tBodyAcc- mean()-X	tBodyAcc- mean()-Y	tBodyAcc- mean()-Z	tBodyAcc- std()-X	tBodyAcc- std()-Y	tBodyAcc- std()-Z	tBodyAcc- mad()-X	tBodyAcc- mad()-Y	tBodyAcc- mad()-Z	tBodyAcc- max()-X	...	fBodyGyroJerkMag- skewness()	fBodyGyroJerkMag- kurtosis()	angle(tBodyAccMean,gravity)	angle(tBodyAc
count	7352.000000	7352.000000	7352.000000	7352.000000	7352.000000	7352.000000	7352.000000	7352.000000	7352.000000	7352.000000	...	7352.000000	7352.000000	7352.000000	
mean	0.274488	-0.017695	-0.109141	-0.605438	-0.510938	-0.604754	-0.630512	-0.526907	-0.606150	-0.468604	...	-0.307009	-0.625294	0.008684	
std	0.070261	0.040811	0.056635	0.448734	0.502645	0.418687	0.424073	0.485942	0.414122	0.544547	...	0.321011	0.307584	0.336787	
min	-1.000000	-1.000000	-1.000000	-1.000000	-0.999873	-1.000000	-1.000000	-1.000000	-1.000000	-1.000000	...	-0.995357	-0.999765	-0.976580	
25%	0.262975	-0.024863	-0.120993	-0.992754	-0.978129	-0.980233	-0.993591	-0.978162	-0.980251	-0.936219	...	-0.542602	-0.845573	-0.121527	
50%	0.277193	-0.017219	-0.108676	-0.946196	-0.851897	-0.859365	-0.950709	-0.857328	-0.857143	-0.881637	...	-0.343685	-0.711692	0.009509	
75%	0.288461	-0.010783	-0.097794	-0.242813	-0.034231	-0.262415	-0.292680	-0.066701	-0.265671	-0.017129	...	-0.126979	-0.503878	0.150865	
max	1.000000	1.000000	1.000000	1.000000	0.916238	1.000000	1.000000	0.967664	1.000000	1.000000	...	0.989538	0.956045	1.000000	

8 rows x 562 columns

Checking for redundancies and null data

```
> MI
```

```
#check for redundancies and null values
print("Total Duplicates Train: {} \n".format(sum(train_df.duplicated())))
print("Total Duplicates in Test: {} \n".format(sum(test_df.duplicated())))
print("Total Null values in Train: {} \n".format(train_df.isnull().values.sum()))
print("Total Null values in Test: {} \n".format(test_df.isnull().values.sum()))
```

Total Duplicates Train: 0

Total Duplicates in Test: 0

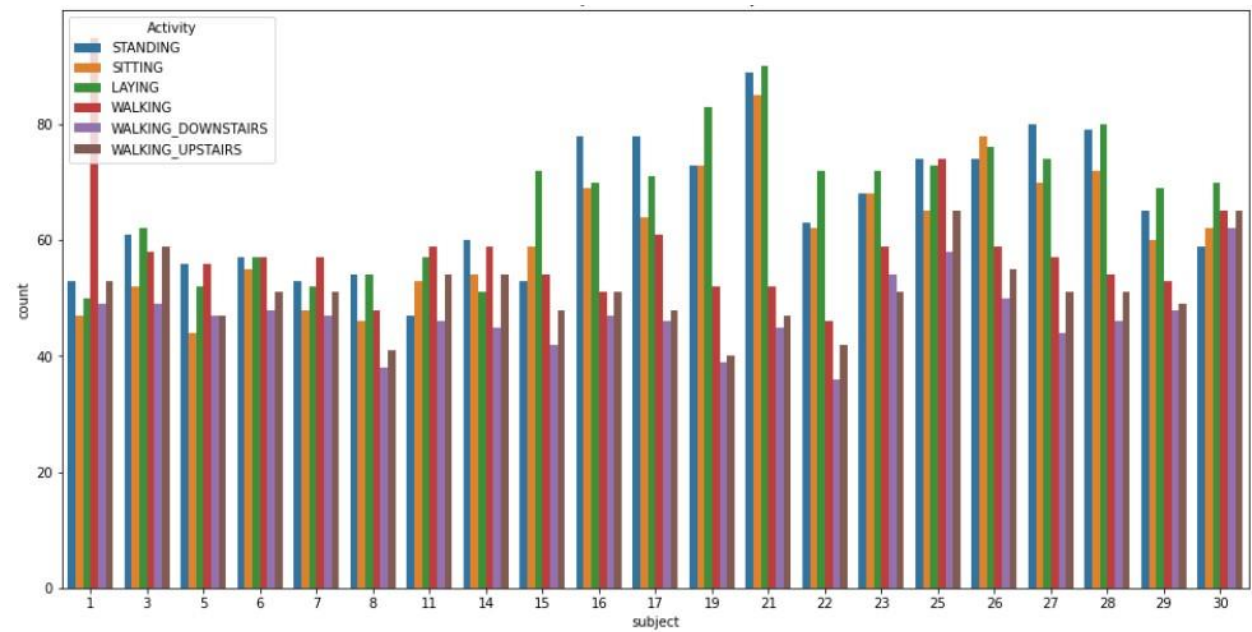
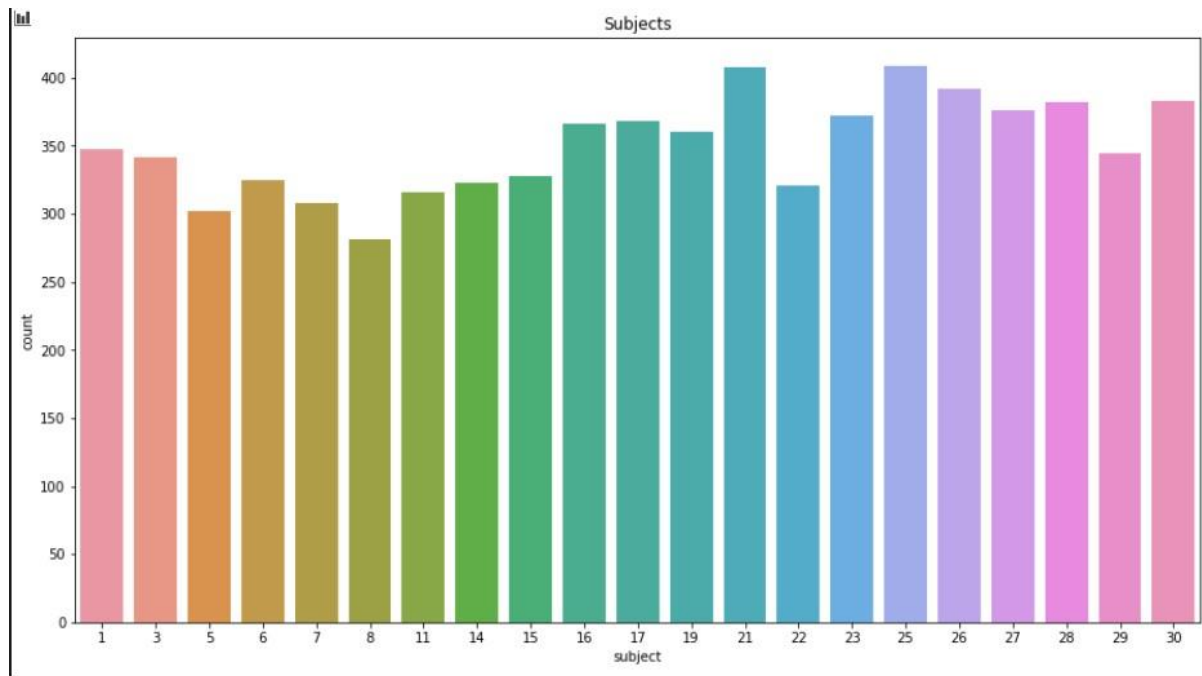
Total Null values in Train: 0

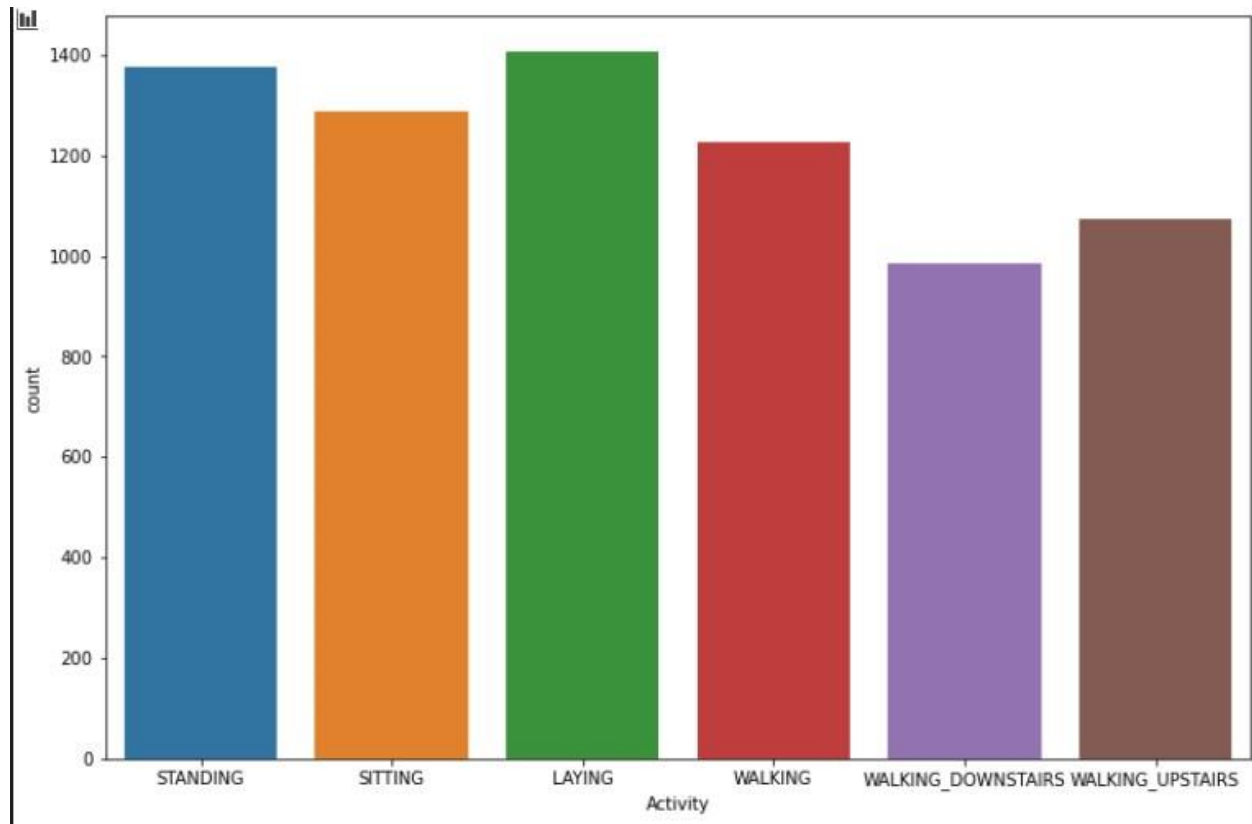
Total Null values in Test: 0

Checking for imbalance in data

```
#checking count of exampmples
train_df['Activity'].value_counts()
```

```
LAYING          1407
STANDING        1374
SITTING         1286
WALKING         1226
WALKING_UPSTAIRS 1073
WALKING_DOWNSTAIRS 986
Name: Activity, dtype: int64
```





The data is almost well balanced and all classes have more or less the same count and instances.

ML Models:

As this is a multi-class classification problem, various classification algorithms were used and evaluated.

Logistic Regression:

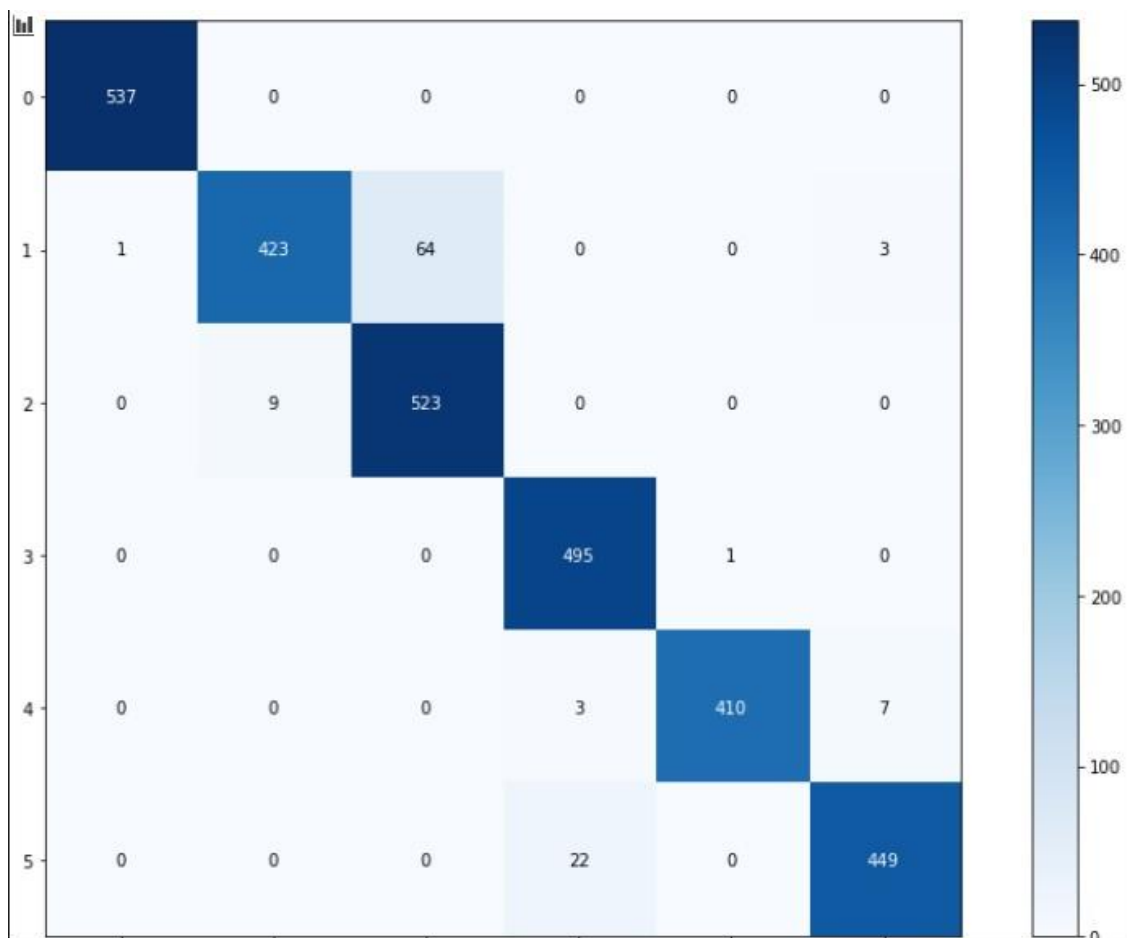
```
#logistic regression
parameters = {'C':np.arange(10,61,10),'penalty':['l2','l1']}
lr_classifier = LogisticRegression(multi_class='ovr')
lr_classifier_rs = RandomizedSearchCV(lr_classifier, param_distributions = parameters, cv = 5, random_state = 42)
lr_classifier_rs.fit(X_train, y_train)
y_pred = lr_classifier_rs.predict(X_test)
```

```
lr_accuracy = accuracy_score(y_true = y_test, y_pred = y_pred)
print("Accuracy using Logisitc Regression:", lr_accuracy)
print(classification_report(y_test,y_pred))
```

Accuracy using Logisitc Regression: 0.9626739056667798

	precision	recall	f1-score	support
LAYING	1.00	1.00	1.00	537
SITTING	0.98	0.86	0.92	491
STANDING	0.89	0.98	0.93	532
WALKING	0.95	1.00	0.97	496
WALKING_DOWNSTAIRS	1.00	0.98	0.99	420
WALKING_UPSTAIRS	0.98	0.95	0.97	471
accuracy			0.96	2947
macro avg	0.97	0.96	0.96	2947
weighted avg	0.96	0.96	0.96	2947

Confusion matrix:



```
## getting best random search attributes

get_best_randomsearch_results(lr_classifier_rs)
Best estimator: LogisticRegression(C=30, multi_class='ovr')
Best set of parameters: {'penalty': 'l2', 'C': 30}
Best score: 0.9401606570568404
```

Analysis:

Logistic regression model uses the default lbfgs solver and we iterate it with different set of parameters with accuracy as the evaluating score. The best set of parameters returned are C=30 and l2 penalty. We get an accuracy of 94.02% and through the confusion matrix we can check which class gets falsely labelled. We can see that there a slight misclassification and few sitting classes were labelled as standing, which decreases the recall of sitting class but still produces a good f1 score of 0.92. Laying class was perfectly classified and has the prefect f1 score followed closely by walking-downstairs class.

Support Vector Machine:

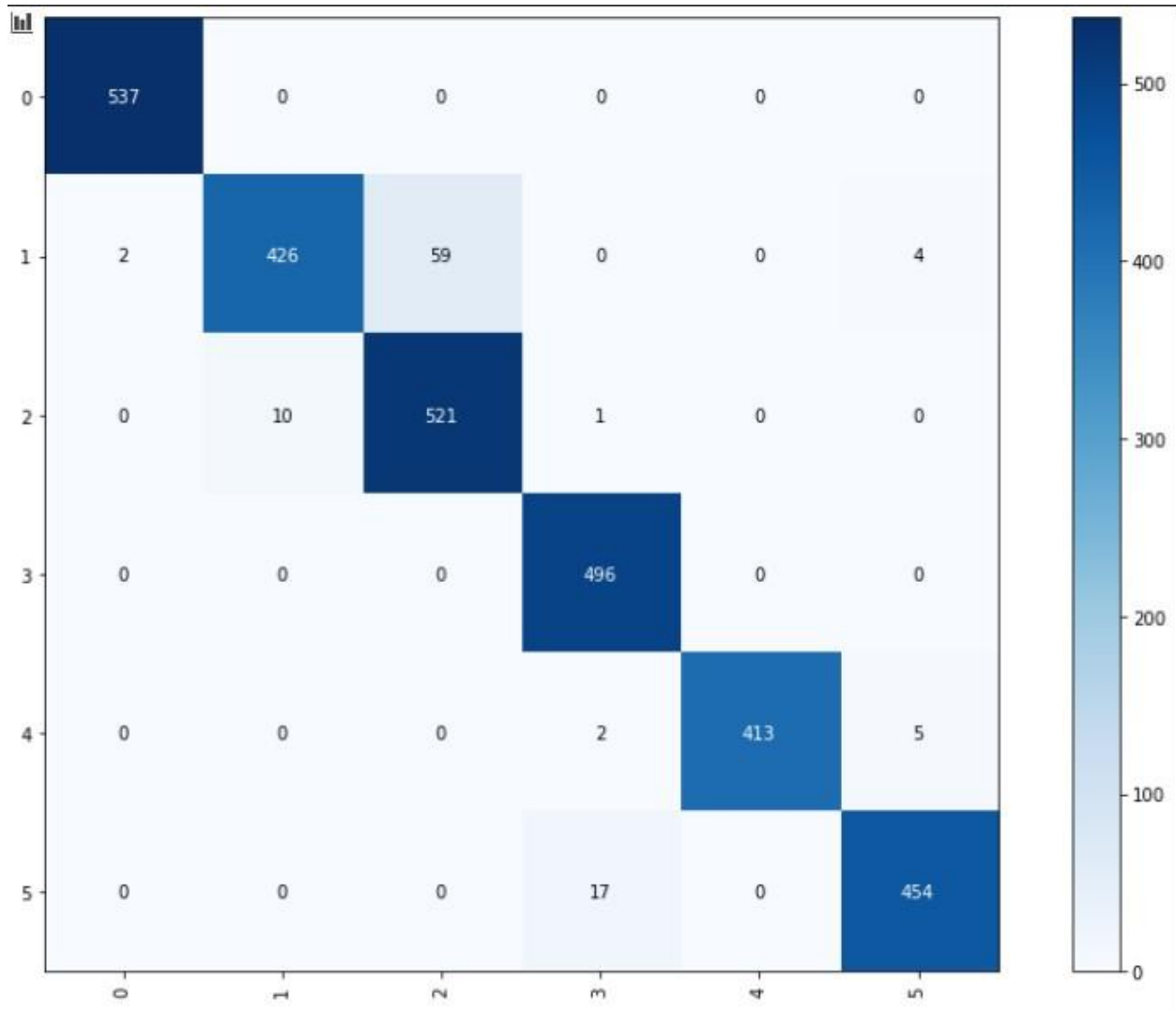
```
#SVM
parameters = {'C': np.arange(1,12,2)}
lr_svm = LinearSVC(tol = 0.00005)
lr_svm_rs = RandomizedSearchCV(lr_svm, param_distributions = parameters, random_state = 42)
lr_svm_rs.fit(X_train, y_train)
y_pred = lr_svm_rs.predict(X_test)
```

```
lr_svm_accuracy = accuracy_score(y_true = y_test, y_pred = y_pred)
print("Accuracy using Linear SVM:", lr_svm_accuracy)
print(classification_report(y_test,y_pred))
```

Accuracy using Linear SVM: 0.9653885307091958

	precision	recall	f1-score	support
LAYING	1.00	1.00	1.00	537
SITTING	0.98	0.86	0.92	491
STANDING	0.90	0.98	0.94	532
WALKING	0.96	1.00	0.98	496
WALKING_DOWNSTAIRS	1.00	0.98	0.99	420
WALKING_UPSTAIRS	0.98	0.96	0.97	471
accuracy			0.97	2947
macro avg	0.97	0.97	0.97	2947
weighted avg	0.97	0.97	0.97	2947

Confusion matrix:



```
▶ ML  
## getting best random search attributes  
get_best_randomsearch_results(lr_svm_rs)  
  
Best estimator: LinearSVC(C=1, tol=5e-05)  
Best set of parameters: {'C': 1}  
Best score: 0.942336047947391
```

Analysis:

Here we have used SVM with a liner kernel which gives slightly better result than logistic regression. On iterating the model with various parameters, the best parameters were found to be $C=1$ and tol value = $5e-05$. It gives an accuracy of 96.54% and through confusion matrix we see the misclassified labels.

Similar to logistic regression sitting class was mislabeled as standing for some instances. This number is slightly less than that of logistic regression and gives similar recall and f1 scores for the class. Here too laying was perfectly classified followed by walking-downstairs class.

Decision Tree:

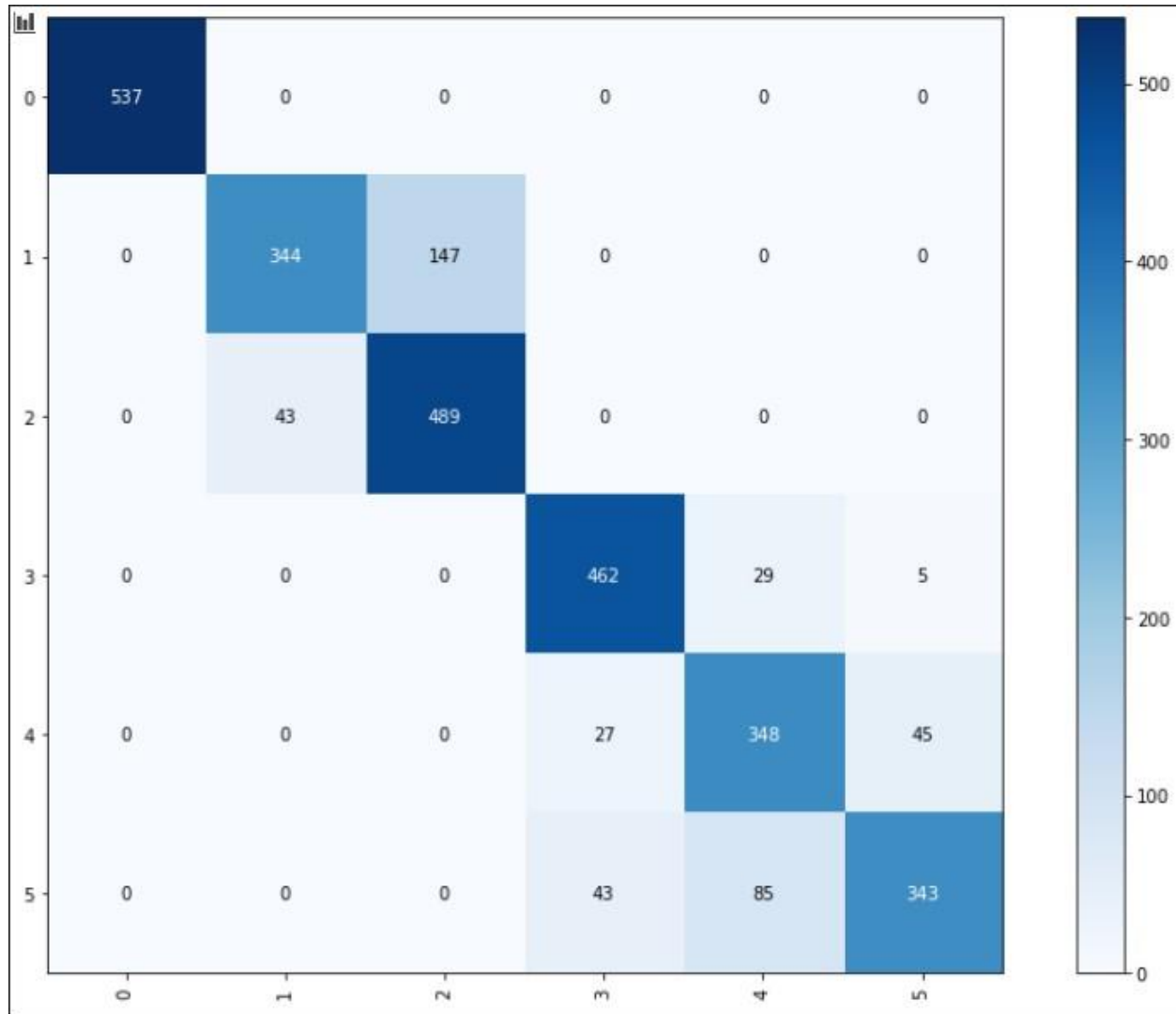
```
▶ ML
#Decision tree
parameters = {'max_depth':np.arange(2,10,2)}
dt_classifier = DecisionTreeClassifier()
dt_classifier_rs = RandomizedSearchCV(dt_classifier,param_distributions=parameters,random_state = 42)
dt_classifier_rs.fit(X_train, y_train)
y_pred = dt_classifier_rs.predict(X_test)
```

```
dt_accuracy = accuracy_score(y_true = y_test, y_pred = y_pred)
print("Accuracy using Decision tree:", dt_accuracy)
print(classification_report(y_test,y_pred))
```

```
Accuracy using Decision tree: 0.8690193417034272
```

	precision	recall	f1-score	support
LAYING	1.00	1.00	1.00	537
SITTING	0.86	0.76	0.81	491
STANDING	0.80	0.89	0.84	532
WALKING	0.84	0.94	0.89	496
WALKING_DOWNSTAIRS	0.87	0.82	0.85	420
WALKING_UPSTAIRS	0.85	0.77	0.81	471
accuracy			0.87	2947
macro avg	0.87	0.86	0.87	2947
weighted avg	0.87	0.87	0.87	2947

Confusion matrix:



```
## getting best estimators
get_best_randomsearch_results(dt_classifier_rs)

Best estimator: DecisionTreeClassifier(max_depth=6)
Best set of parameters: {'max_depth': 6}
Best score: 0.8483418656381655
```

Analysis:

We use the vanilla decision tree classifier and iterate it with different maximum depths values which gives the optimal depth to be 6. This model gives an accuracy of 86.90%. Upon observing the confusion matrix for the model we see various misclassifications. Here too, the largest mislabeling has been the sitting-standing pair and a significant amount of walking-upstairs has been misclassified as walkingdownstairs which reduces recall and f1 score for both. Here too, laying has been perfectly

classifies with 1.0 f1 score which is followed by walking. Another thing to note is that the difference between the f1 scores of these classes is comparatively higher than previous models.

Random Forest:

```
#Random Forest
params = {'n_estimators': np.arange(20,101,10), 'max_depth':np.arange(2,16,2)}
rf_classifier = RandomForestClassifier()
rf_classifier_rs = RandomizedSearchCV(rf_classifier, param_distributions=params,random_state = 42)
rf_fit=rf_classifier_rs.fit(X_train, y_train)
y_pred = rf_classifier_rs.predict(X_test)
```

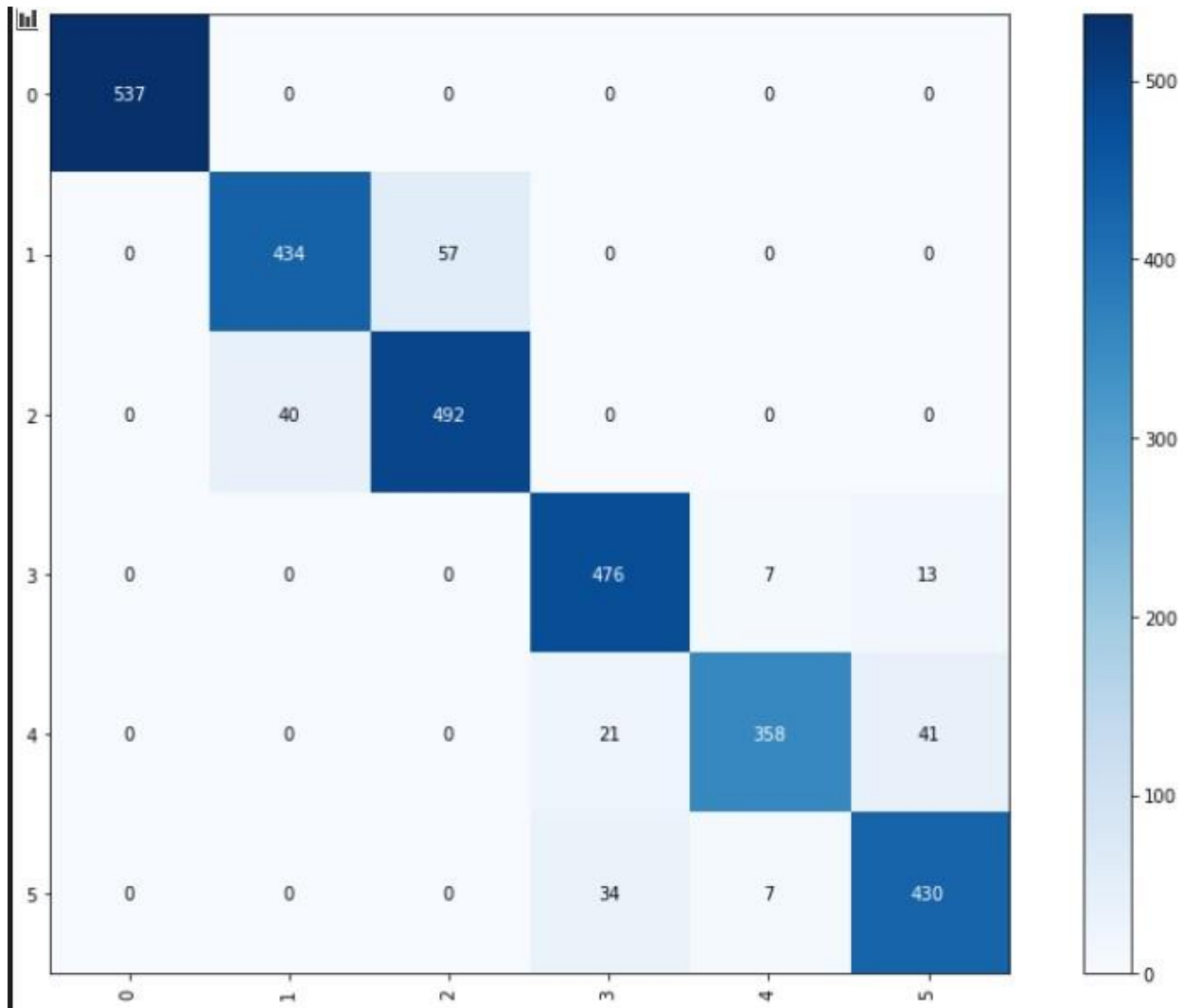
```
rf_accuracy = accuracy_score(y_test, y_pred)
print("Accuracy using Random Forest:", rf_accuracy)
print(classification_report(y_test,y_pred))
```

```
Accuracy using Random Forest: 0.9297590770274856
              precision    recall  f1-score   support

    LAYING              1.00      1.00      1.00        537
    SITTING              0.92      0.89      0.91        491
    STANDING              0.90      0.93      0.92        532
    WALKING              0.90      0.97      0.94        496
WALKING_DOWNSTAIRS      0.96      0.86      0.91        420
WALKING_UPSTAIRS        0.90      0.91      0.90        471

   accuracy                   0.93        2947
  macro avg              0.93      0.93      0.93        2947
weighted avg              0.93      0.93      0.93        2947
```

Confusion matrix:



```
## getting best estimators
```

```
get_best_randomsearch_results(rf_classifier_rs)
```

```
Best estimator: RandomForestClassifier(max_depth=14, n_estimators=80)
```

```
Best set of parameters: {'n_estimators': 80, 'max_depth': 14}
```

```
Best score: 0.9197534187026271
```

Analysis:

Random Forest after iterating over a set of parameters gives the best max_depth as 14 and n_estimators 80. It gives an accuracy of 92.97% which is greater than decision tree model. Upon observing the confusion matrix we see that it's clearly an improvement over decision tree but there's still some confusion between the sitting/standing pair and also a slight misclassification where walkingupstairs was

misclassified as walking. This also perfectly classifies laying and has the best f1 score for it followed by walking.

Linear Regression:

Since this dataset is essentially for multi-class classification, linear regression can't be used to achieve the given task, but we see that there are a large no of features and some of them might be linearly dependent. So, from the random forest model we take out the most important features according to variance and try to fit a linear model in between a pair of these. First, we see the relation between features.

```
rf_class = RandomForestClassifier()
rf_class.fit(X_train,y_train)
model_vars0 = pd.DataFrame(
    {'variable':X_train.columns,
     'importance':rf_class.feature_importances_})

model_vars0.sort_values(by='importance',
                        ascending=False,
                        inplace=True)
```

```
n_used = 4

cols_model = [col for col in model_vars0.variable[:n_used].values] + [model_vars0.variable[6]]
```

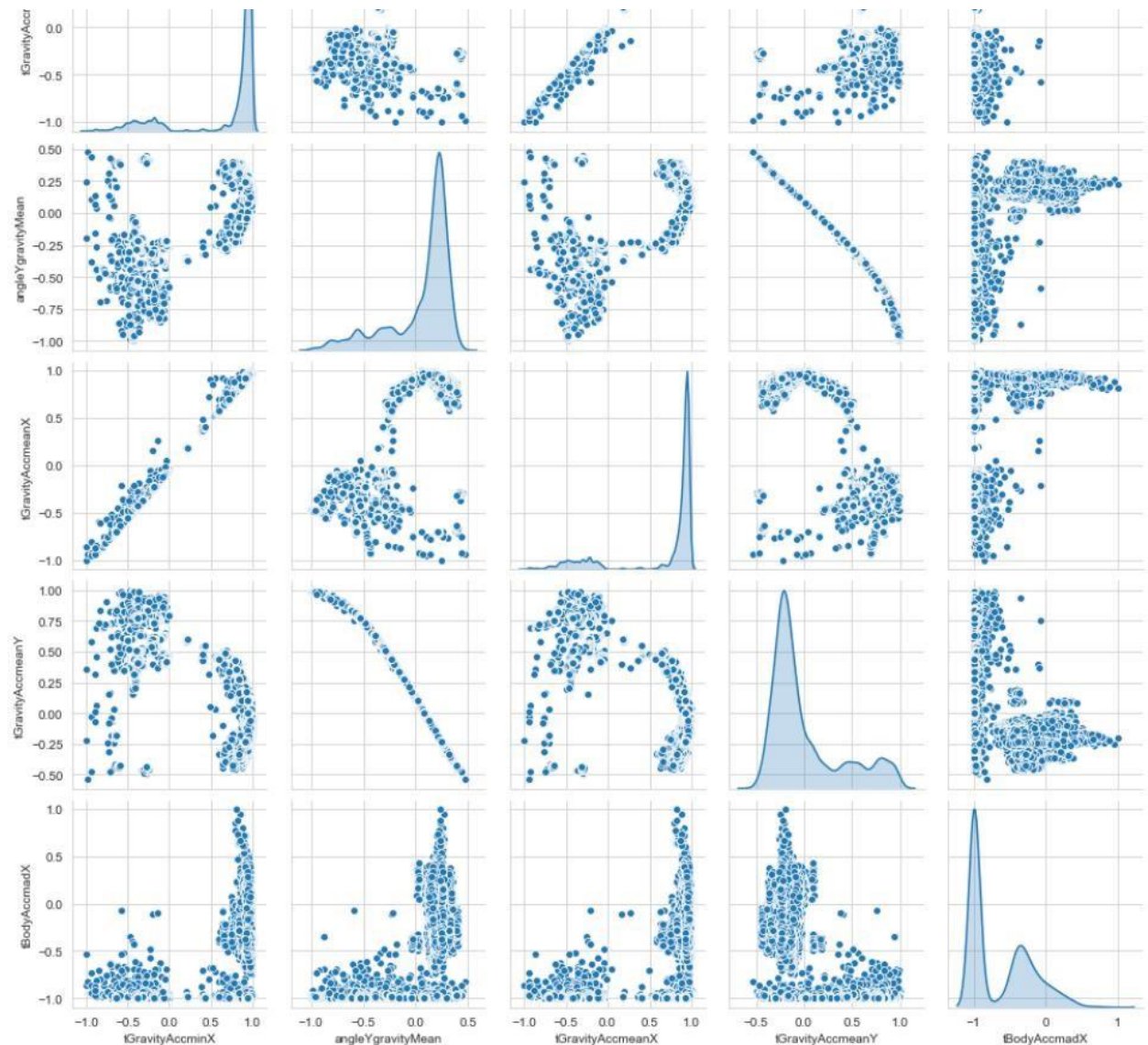
▶ ML

```
X = train_df[cols_model]
Y = train_df.Activity
```

▶ ML

```
plt.figure(figsize=(6,6))
sns.set_style('whitegrid')

sns.pairplot(data=train_df[[col for col in X.columns]],
             palette='Set2',
             diag_kind='kde')
```



We observe that the features tGravityAccmeanY and angleYgravityMean are highly correlated, so let's fit a linear model in them. Since there are a large number of instances, we randomly select 500 for training set and 200 for testing set.

```
import random
#we see that tGravityAccmeanY and angleYgravityMean are linearly related so let's do regression on these
linX_tr = train_df['tGravityAccmeanY']
liny_tr = train_df['angleYgravityMean']
linX_tst = test_df['tGravityAccmeanY']
liny_tst = test_df['angleYgravityMean']
#this set is too big so randomly select 500 training examples and 200 test examples
rand_list_tr = random.sample(range(0,7352),500)
rand_list_tst = random.sample(range(0,2947),200)
linX_tr = linX_tr[rand_list_tr].to_numpy()
liny_tr = liny_tr[rand_list_tr].to_numpy()
linX_tst = linX_tst[rand_list_tst].to_numpy()
liny_tst = liny_tst[rand_list_tst].to_numpy()
```

```
linX_tr = linX_tr[:,np.newaxis]  
linX_tst = linX_tst[:,np.newaxis]
```

▶  M4


```
from sklearn.linear_model import LinearRegression  
lin_regress = LinearRegression()  
lin_regress.fit(linX_tr, liny_tr) #training the algorithm
```

LinearRegression()

▶  M4

```
#To retrieve the intercept:  
print(lin_regress.intercept_)  
  
#For retrieving the slope:  
print(lin_regress.coef_)
```

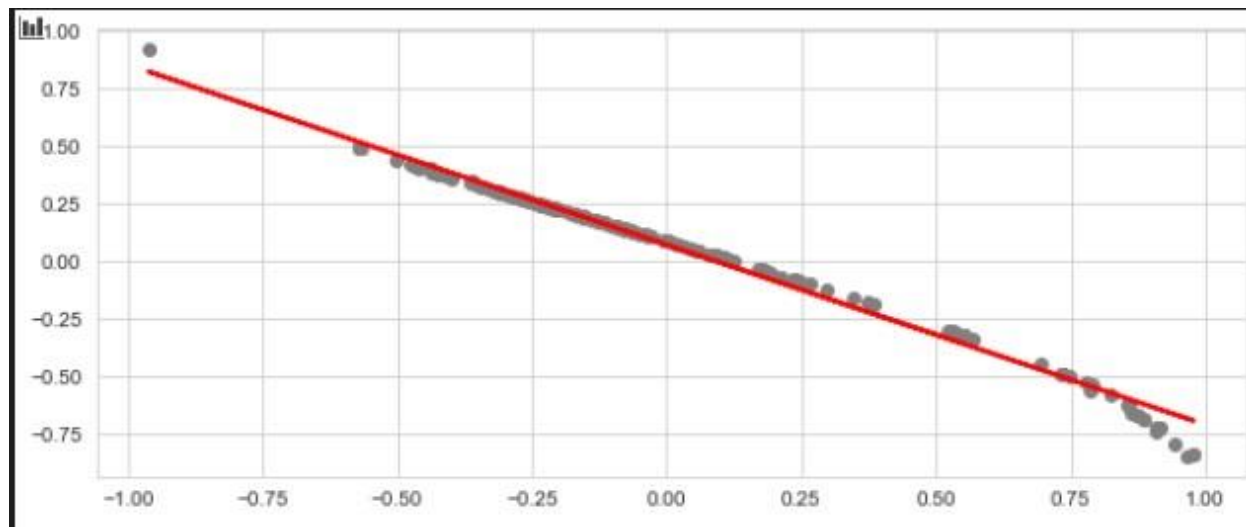
0.06861740980976902
[-0.78284484]

▶  M4

```
y_pred = lin_regress.predict(linX_tst)
```

▶  M4

```
plt.scatter(linX_tst, liny_tst, color='gray')  
plt.plot(linX_tst, y_pred, color='red', linewidth=2)  
plt.show()
```




```

from sklearn import metrics
print('Mean Absolute Error:', metrics.mean_absolute_error(liny_tst, y_pred))
print('Mean Squared Error:', metrics.mean_squared_error(liny_tst, y_pred))
print('Root Mean Squared Error:', np.sqrt(metrics.mean_squared_error(liny_tst, y_pred)))

```

```

Mean Absolute Error: 0.020935208998939164
Mean Squared Error: 0.00105693584122261
Root Mean Squared Error: 0.032510549691178864

```

Analysis:

As expected, fitting a liner model on these features gave great results and we can almost fit a perfect straight line. Both the absolute error and mean squared error are negligible. This results gives the intuition that performing PCA on this dataset would give promising results and we can significantly reduce the dimensions or no of features which would prevent overfitting and further improve results of all our classifiers.

Artificial Neural Network:

For the neural net model we first encode the activities into integers and then employ one-hot encoding to produce output labels.

```

##neural net
##creating an encoded label
activity_tr = train_df['Activity'] #for training set
label_name_tr = activity_tr.map({"WALKING":0, "WALKING_UPSTAIRS":1, "WALKING_DOWNSTAIRS":2, "SITTING":3, "STANDING":4, "LAYING":5})
activity_tst = test_df['Activity'] #for test data
label_name_tst = activity_tst.map({"WALKING":0, "WALKING_UPSTAIRS":1, "WALKING_DOWNSTAIRS":2, "SITTING":3, "STANDING":4, "LAYING":5})

from keras.utils import to_categorical
y_train = to_categorical(label_name_tr )
y_test = to_categorical(label_name_tst )

```

After that we create out neural architecture and train it.

```

model = Sequential()

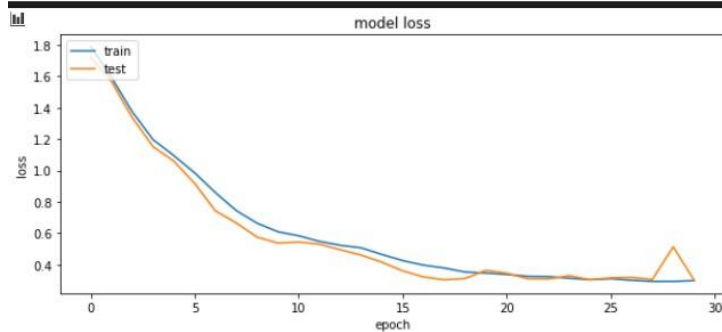
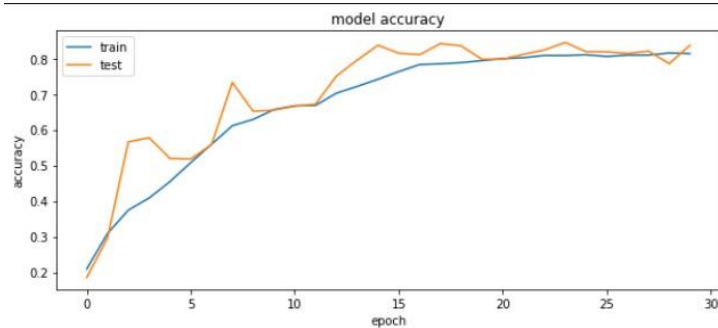
model.add(Dense(64, input_dim=X_train.shape[1] , activation='relu'))
model.add(Dense(64, activation='relu'))
model.add(BatchNormalization())
model.add(Dense(128, activation='relu'))
model.add(Dense(196, activation='relu'))
model.add(Dense(32, activation='relu'))
model.add(Dense(6, activation='sigmoid'))

model.compile(optimizer = Adam(lr = 0.0005),loss='categorical_crossentropy', metrics=['accuracy'])
print(model.summary())

```

Model: "sequential"

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 64)	35968
dense_1 (Dense)	(None, 64)	4160
batch_normalization (Batch Normalization)	(None, 64)	256
dense_2 (Dense)	(None, 128)	8320
dense_3 (Dense)	(None, 196)	25284
dense_4 (Dense)	(None, 32)	6304
dense_5 (Dense)	(None, 6)	198
Total params: 80,490		
Trainable params: 80,362		
Non-trainable params: 128		



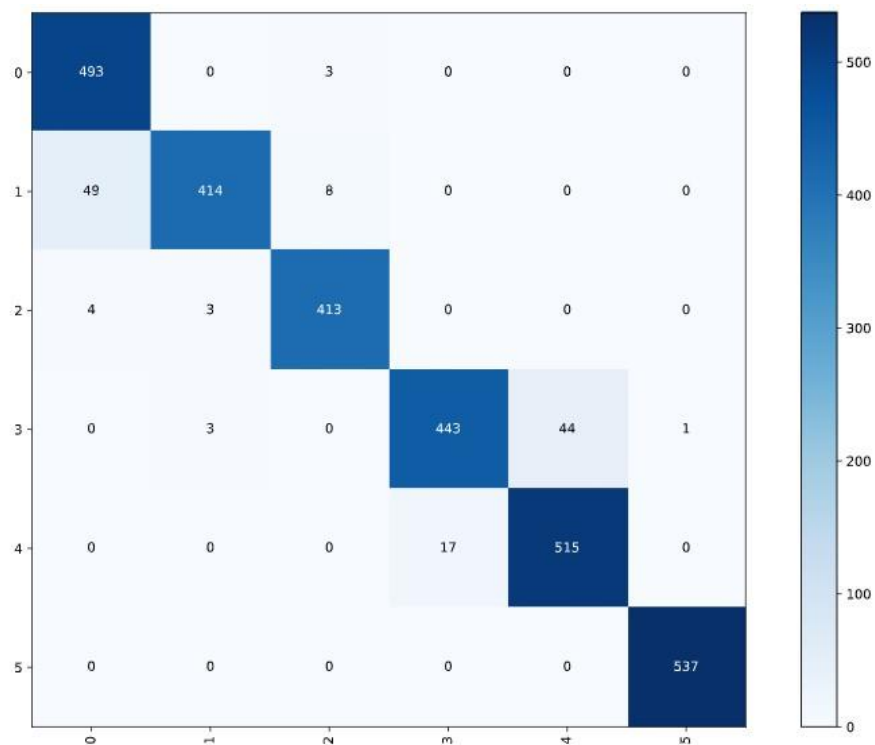

```
nn_accuracy = accuracy_score(y_true, pred)
print("Accuracy using neural network:", nn_accuracy)
```

Accuracy using neural network: 0.9467254835425857

```
cm = confusion_matrix(y_true, pred)
plot_confusion_matrix(cm, np.unique(pred))
print(classification_report(y_true, pred))
```

	precision	recall	f1-score	support
0	0.90	0.99	0.95	496
1	0.99	0.88	0.93	471
2	0.97	0.98	0.98	420
3	0.96	0.90	0.93	491
4	0.92	0.97	0.94	532
5	1.00	1.00	1.00	537
accuracy			0.96	2947
macro avg	0.96	0.95	0.95	2947
weighted avg	0.96	0.96	0.95	2947

Confusion matrix:



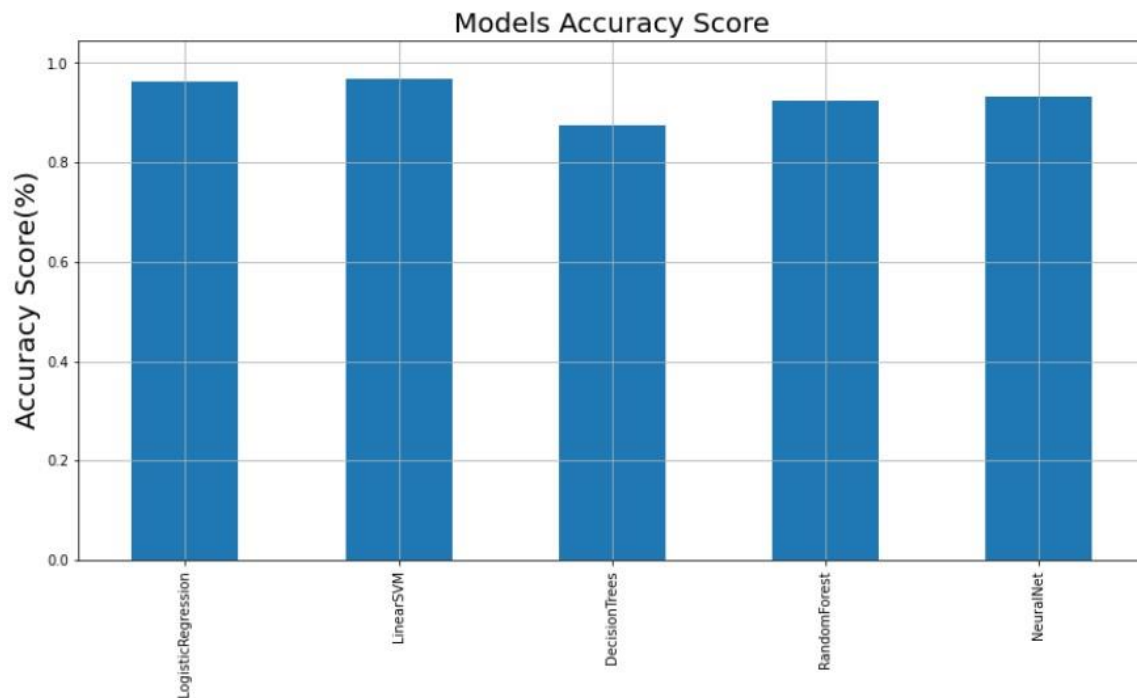
Analysis:

We use a simple network architecture with 5 hidden layers and a output layer with 6 nodes. We also use dropout regularization, batch normalization and callbacks to further elevate the performance. The model is compiled with categorical-crossentropy loss function and adam optimizer. The model is trained on 30 epochs with batch size of 256. The model gives an accuracy of 94.67%. From the confusion matrix, we see that there are some misclassifications. Walking-upstairs was mislabeled as walking few times and sitting mislabeled as standing like usual. This also gives the perfect f1 score for laying down followed by walking downstairs.

Comparative Analysis:

Accuracy of each model was stored in a dataframe and plotted to compare the results.

```
ax = model_score.plot(x = "Model", y = "Score", kind = "bar", figsize = (14, 7), legend = False)
plt.title("Models Accuracy Score", fontsize = 20)
plt.xlabel("")
plt.margins(x = 0, y = 0.08)
plt.ylabel("Accuracy Score(%)", fontsize = 20)
plt.grid(visible = True)
for i in ax.patches:
    ax.text(x = i.get_x()+0.05, y = i.get_height()+1, s = str(i.get_height())+"%", fontsize = 16, color = "#232b2b")
```



From the accuracies data we see that SVM with linear kernel gives the best result. Logistic regression is a very close second, artificial neural network at third followed by random forest. Decision tree gives the worst accuracy of all. Overall all the models gives similar results and are of practical use in classifying

human activity recognition data. From the classification reports too, we see the same trend. The slight ambiguity between the sitting-standing pair is present in every model but certain models produces other pair ambiguity too though not in a significant amount. Except decision tree, all models give acceptable f1 scores. Using PCA and dimension reduction might help to further enhance their performance and tuning the hyperparameters for the neural network can give increased accuracy too.

Unsupervised Learning:

For unsupervised learning part, we apply k-means to the given data set and see what kind of results we get.

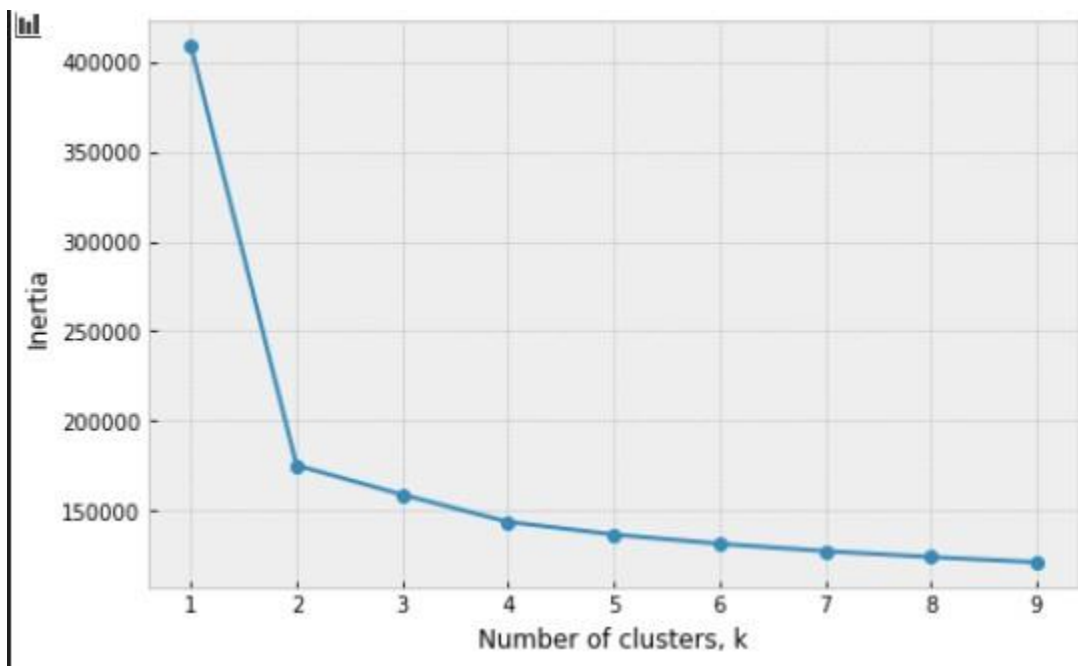
```
#a function to run kmeans
def k_means(n_clust, data_frame, true_labels):
    k_means = KMeans(n_clusters = n_clust, random_state=123, n_init=30)
    k_means.fit(data_frame)
    c_labels = k_means.labels_
    df = pd.DataFrame({'clust_label': c_labels, 'orig_label': true_labels.tolist()})
    ct = pd.crosstab(df['clust_label'], df['orig_label'])
    y_clust = k_means.predict(data_frame)
    display(ct)
```

We use the knee method to get the optimal value of k (no of clusters).

```
#check the optimal k value
ks = range(1, 10)
inertias = []

for k in ks:
    model = KMeans(n_clusters=k)
    model.fit(trdata)
    inertias.append(model.inertia_)

plt.figure(figsize=(8,5))
plt.style.use('bmh')
plt.plot(ks, inertias, '-o')
plt.xlabel('Number of clusters, k')
plt.ylabel('Inertia')
plt.xticks(ks)
plt.show()
```



```
k_means(n_clust=2, data_frame=trdata, true_labels=labels)
```

orig_label	LAYING	SITTING	STANDING	WALKING	WALKING_DOWNSTAIRS	WALKING_UPSTAIRS
0	1396	1285	1374	0	0	0
1	11	1	0	1226	986	1073

```
##ckeck k means with 6 clusters (original no of classes)
k_means(n_clust=6, data_frame=trdata, true_labels=labels)
```

orig_label	LAYING	SITTING	STANDING	WALKING	WALKING_DOWNSTAIRS	WALKING_UPSTAIRS
0	0	0	0	85	223	49
1	0	924	947	0	0	0
2	10	1	0	595	134	810
3	0	0	0	546	629	214
4	158	311	427	0	0	0
5	1239	50	0	0	0	0

As using 2 clusters gives better result with change the labels to create 2 categories of movement and nomovement.

```
k_means(n_clust=2, data_frame=trdata, true_labels=labels_binary)
```

orig_label	0	1
clust_label		
0	4055	0
1	12	3285

Analysis:

By applying the k-means algorithm to the dataset we try to categorize the features into classes. We discard the given classes and try to create our own classes. By the graph we see that the knee is at 2 so, 2 clusters can be identified with maximum accuracy. By performing k-means with 2 clusters we see that it clumps together data with similar movement information or activities with requires motion and those which don't get clumped. By applying k-means with 6 classes we don't get any meaningful result sp finally we re-structure the data set labels and create 2 classes only. As expected by apply k-means now we get optimal results with very little misclassification with precision=1, recall = 0.99 anf f1 score=0.99