Human Activity Recognition

Description

These days Smartphones have become an integral part of our life. We cannot assume our life without a mobile phone. Since, the advent of Smartphones, a revolution has been created in the mobile communication industry. Smartphones are not just restricted for calling these days. Infact, they are more often used for entertainment purpose.

Smartphone manufacturing companies load Smartphones with various sensors to enhance the user experinece. Two of the such sensors are **Accelerometer** and **Gyroscope**. **Accelerometer** measures acceleration while **Gyroscope** measures angular velocity.

Here, we will try to use the data provided by accelerometer and gyroscope of Smartphone to classify the activity which a Smartphone user is performing.

Why this is Useful?

These days, in addition to Smartphones, we are also using Smart-Watches like Fitbit or Apple-Watch, which help us to track our health. They monitor our each activity throughout the day check how many calories we have burnt. How many hours have we slept. However, in addition to Accelerometer and Gyroscope, they also use Heart-Rate data to monitor our activity. Since, we only have Smartphone data so just by using Accelerometer and Gyroscope data we will monitor the activity of a person. This software can then be converted into an App which can be downloaded in Smartphone. Hence, a person who has Smartphone can monitor his/her health using this App

Information about Data

How Data is recorded

The experiments have been carried out with a group of 30 volunteers within an age bracket of 19-48 years. Each person performed six activities **(WALKING, WALKING-UPSTAIRS, WALKING-DOWNSTAIRS, SITTING-DOWN, STANDING-UP, LAYING-DOWN)** wearing a smartphone (Samsung Galaxy S II) on the waist. Using its embedded accelerometer and gyroscope, we captured 3-axial linear acceleration and 3-axial angular velocity at a constant rate of 50Hz. The experiments have been video-recorded to label the data manually. The obtained dataset has been randomly partitioned into two sets, where 70% of the volunteers was selected for generating the training data and 30% the test data. 5.2. Features

1. These sensor signals are pre-processed by applying noise filters and then sampled in fixed-width windows(sliding windows) of 2.56 seconds each with 50% overlap. i.e., each window has 128 readings. A 128 size vector is created from each window.
2. From Each window or to be more precise, from each 128 readings domain experts from signal processing have engineered feature vector of size 561 by calculating variables from the time and frequency domain. In our dataset, each data-point represents a window with different readings.
3. 561 features are stored in the file "Features.docx". Check it out.
4. Check out 561 features here.(In your blog give here the link of the docx file of features which you upload on github).
5. The acceleration signal was separated into Body and Gravity acceleration signals(tBodyAcc-XYZ and tGravityAcc-XYZ) using some low pass filter with corner frequency of 0.3Hz.
6. After that, the body linear acceleration and angular velocity were derived in time to obtain jerk signals (tBodyAccJerk-XYZ and tBodyGyroJerk-XYZ).
7. The magnitude of these 3-dimensional signals were calculated using the Euclidian norm. This magnitudes are represented as features with names like tBodyAccMag, tGravityAccMag, tBodyAccJerkMag, tBodyGyroMag and tBodyGyroJerkMag.
8. Finally, We've got frequency domain signals from some of the available signals by applying a FFT (Fast Fourier Transform). These signals obtained were labelled with prefix 'f' just like original signals with prefix 't'. These signals are labelled as fBodyAcc-XYZ, fBodyGyroMag etc.

These are the signals that we got so far.

- tBodyAcc-XYZ
- tGravityAcc-XYZ
- tBodyAccJerk-XYZ
- tBodyGyro-XYZ
- tBodyGyroJerk-XYZ
- tBodyAccMag
- tGravityAccMag

- tBodyAccJerkMag
- tBodyGyroMag
- tBodyGyroJerkMag
- fBodyAcc-XYZ
- fBodyAccJerk-XYZ
- fBodyGyro-XYZ
- fBodyAccMag
- fBodyAccJerkMag
- fBodyGyroMag
- fBodyGyroJerkMag

9 We can estimate some set of variables from the above signals. i.e., We will estimate the following properties on each and every signal that we recorded so far.

- mean(): Mean value
- std(): Standard deviation
- mad(): Median absolute deviation
- max(): Largest value in array
- min(): Smallest value in array
- sma(): Signal magnitude area
- energy(): Energy measure. Sum of the squares divided by the number of values.
- iqr(): Inter-quartile range
- entropy(): Signal entropy
- arCoeff(): Auto-regression coefficients with Burg order equal to 4
- correlation(): correlation coefficient between two signals
- maxInds(): index of the frequency component with largest magnitude
- meanFreq(): Weighted average of the frequency components to obtain a mean frequency
- skewness(): skewness of the frequency domain signal
- kurtosis(): kurtosis of the frequency domain signal
- bandsEnergy(): Energy of a frequency interval within the 64 bins of the FFT of each window.
- angle(): Angle between to vectors.

10 We can obtain some other vectors by taking the average of signals in a single window sample. These are used on the angle() variable.

- gravityMean
- tBodyAccMean
- tBodyAccJerkMean
- tBodyGyroMean
- tBodyGyroJerkMean

Data Source

Data is downloaded from following source:
https://archive.ics.uci.edu/ml/datasets/human+activity+recognition+using+smartphones

Quick Overview of Dataset

Accelerometer and Gyroscope readings are taken from 30 volunteers(referred as subjects) while performing the following 6 Activities.

These activites are encoded as follows:
WALKING-- 1
WALKING_UPSTAIRS-- 2
WALKING_DOWNSTAIRS-- 3
SITTING-- 4
STANDING-- 5
LYING-- 6

- Readings are divided into a window of 2.56 seconds with 50% overlapping.
- Accelerometer readings are divided into gravity acceleration and body acceleration readings, which has x, y and z components each.
- Gyroscope readings are the measure of angular velocities which has x, y and z components.
- Jerk signals are calculated for Body-Acceleration readings.

- Fourier Transforms are made on the above time readings to obtain frequency readings.
- Now, on all the base signal readings., mean, max, mad, sma, arcoefficient, energy-bands, entropy etc., are calculated for each window.
- Extra features are calculated by taking the average of signals in a single window sample. These are used on the angle() variable.
- Finally, we got feature vector of 561 features and these features are given in the dataset.
- Each window of readings is a data-point of 561 features.

Y-Encoded Labels

WALKING-- 1
WALKING_UPSTAIRS-- 2
WALKING_DOWNSTAIRS-- 3
SITTING-- 4
STANDING-- 5
LYING-- 6

Business Problem

Work-flow is as follows:

1. Domain experts from the field of Signal Processing collects the data from Accelerometer and Gyroscope of Smartphone.
2. They break up the data in the time window of 2.56 seconds with 50% overlapping i.e., 128 reading
3. They engineered 561 features from each time window of 2.56 seconds.

By using either human engineered 561 feature data or raw features of 128 reading, our goal is to predict one of the six activities that a Smartphone user is performing at that 2.56 Seconds time window.

Problem Statement

By using either human engineered 561 feature data or raw features of 128 reading, our goal is to predict one of the six activities that a Smartphone user is performing at that 2.56 Seconds time window.

Objective and Constraints

1. No Low latency requirement.
2. Errors are not much costly.

ML Problem Formulation

All of the Accelerometer and Gyroscope are tri-axial, means that they measure acceleration and angular-velocity respectively in all the three axis namely X-axis, Y-axis and Z-axis. So, we have in total six time-series data. Given this six time-series data, we want to predict six activities namely **Walking** or **Walking-Upstairs** or **Walking-Downstairs** or **Lying-Down** or **Standing-Up** or **Sitting-Down**.

At the outset, this is a multi-class classification problem.

Performance Metric

1. We will use Accuracy as one of the metric.
2. We will also use Confusion-Matrix to check that in which two activities our model is confused and predicting incorrect activity. For example, between Standing-Up and Sitting-Down. Between Walking-Upstairs and Walking-Downstairs.

Data

All the data is present in 'UCI_HAR_dataset/' folder in present working directory.
Feature names are present in 'UCI_HAR_dataset/features.txt'

Train Data
'UCI_HAR_dataset/train/X_train.txt'

'UCI_HAR_dataset/train/X_train.txt'
'UCI_HAR_dataset/train/subject_train.txt'
'UCI_HAR_dataset/train/y_train.txt'
Test Data
'UCI_HAR_dataset/test/X_test.txt'
'UCI_HAR_dataset/test/subject_test.txt'
'UCI_HAR_dataset/test/y_test.txt'

Data-Points Distribution

- 30 test-subjects data is randomly split to 70%(21) train and 30%(7) test data.
- Each data-point corresponds one of the 6 Activities.

Plan of Action

- We will apply classical Machine Learning models on these 561 sized domain expert engineered features.
- As we know that LSTM works well on time-series data, so we have decided that we will apply LSTM of Recurrent Neural Networks on 128 sized raw readings that we obtained from accelerometer and gyroscope signals.

In [1]:

```
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE
import warnings
from datetime import datetime
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import confusion_matrix
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.svm import LinearSVC
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import GradientBoostingClassifier
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers.core import Dense, Dropout
from hyperopt import Trials, STATUS_OK, tpe
from hyperas import optim
from hyperas.distributions import choice, uniform
warnings.simplefilter("ignore")
```

```
C:\Users\GauravP\Anaconda3\lib\site-packages\h5py\__init__.py:36: FutureWarning: Conversion of the
second argument of issubdtype from `float` to `np.floating` is deprecated. In future, it will be
treated as `np.float64 == np.dtype(float).type`.
  from ._conv import register_converters as _register_converters
Using TensorFlow backend.
```

Extracting Features

In [18]:

```
features = list()
with open("../Data/Features.txt") as f:
    for line in f:
        features.append(line.split()[1])
```

Reading train Data

In [64]:

```
train_df = pd.read_csv("../Data/train/X_train.txt", delim_whitespace = True, names = features)
```

```
train_df["subject_id"] = pd.read_csv("../Data/train/subject_train.txt", header = None, squeeze = Tr
ue) #squeeze = True will
#return data in pandas series format

train_df["activity"] = pd.read_csv("../Data/train/y_train.txt", header = None, squeeze = True)

activity = pd.read_csv("../Data/train/y_train.txt", header = None, squeeze = True)

#mapping activity to activity name
label_name = activity.map({1: "WALKING", 2:"WALKING_UPSTAIRS", 3:"WALKING_DOWNSTAIRS", 4:"SITTING",
5:"STANDING", 6:"LYING"})

train_df["activity_name"] = label_name

train_df.head()
```

Out[64]:

	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	...	angle(tBody
0	0.288585	-0.020294	-0.132905	-0.995279	-0.983111	-0.913526	-0.995112	-0.983185	-0.923527	-0.934724	...	
1	0.278419	-0.016411	-0.123520	-0.998245	-0.975300	-0.960322	-0.998807	-0.974914	-0.957686	-0.943068	...	
2	0.279653	-0.019467	-0.113462	-0.995380	-0.967187	-0.978944	-0.996520	-0.963668	-0.977469	-0.938692	...	
3	0.279174	-0.026201	-0.123283	-0.996091	-0.983403	-0.990675	-0.997099	-0.982750	-0.989302	-0.938692	...	
4	0.276629	-0.016570	-0.115362	-0.998139	-0.980817	-0.990482	-0.998321	-0.979672	-0.990441	-0.942469	...	

5 rows × 564 columns

In [66]:

```
print("Size of Train data = {}".format(train_df.shape))
```

```
Size of Train data = (7352, 564)
```

1. Reading Test Data

In [67]:

```
test_df = pd.read_csv("../Data/test/X_test.txt", delim_whitespace = True, names = features)

test_df["subject_id"] = pd.read_csv("../Data/test/subject_test.txt", header = None, squeeze = True)
#squeeze = True will
#return data in pandas series format

test_df["activity"] = pd.read_csv("../Data/test/y_test.txt", header = None, squeeze = True)

activity = pd.read_csv("../Data/test/y_test.txt", header = None, squeeze = True)

#mapping activity to activity name
label_name = activity.map({1: "WALKING", 2:"WALKING_UPSTAIRS", 3:"WALKING_DOWNSTAIRS", 4:"SITTING",
5:"STANDING", 6:"LYING"})

test_df["activity_name"] = label_name

test_df.head()
```

Out[67]:

	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	...	angle(tBody
0	0.257178	-0.023285	-0.014654	-0.938404	-0.920091	-0.667683	-0.952501	-0.925249	-0.674302	-0.894088	...	
1	0.286027	-0.013163	-0.119083	-0.975415	-0.967458	-0.944958	-0.986799	-0.968401	-0.945823	-0.894088	...	
2	0.275485	-0.026050	-0.118152	-0.993819	-0.969926	-0.962748	-0.994403	-0.970735	-0.963483	-0.939260	...	
3	0.270298	-0.032614	-0.117520	-0.994743	-0.973268	-0.967091	-0.995274	-0.974471	-0.968897	-0.938610	...	
4	0.274833	-0.027848	-0.129527	-0.993852	-0.967445	-0.978295	-0.994111	-0.965953	-0.977346	-0.938610		

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	...	angle(tBody

5 rows × 564 columns

In [68]:

```
print("Size of Test data = {}".format(test_df.shape))
```

```
Size of Test data = (2947, 564)
```

2. Data Cleaning

In [72]:

```
# Checking for nan values
print("Number of NaN values in train data is "+str(train_df.isnull().sum().sum()))
print("Number of NaN values in test data is "+str(test_df.isnull().sum().sum()))
```

```
Number of NaN values in train data is 0
Number of NaN values in test data is 0
```

In [74]:

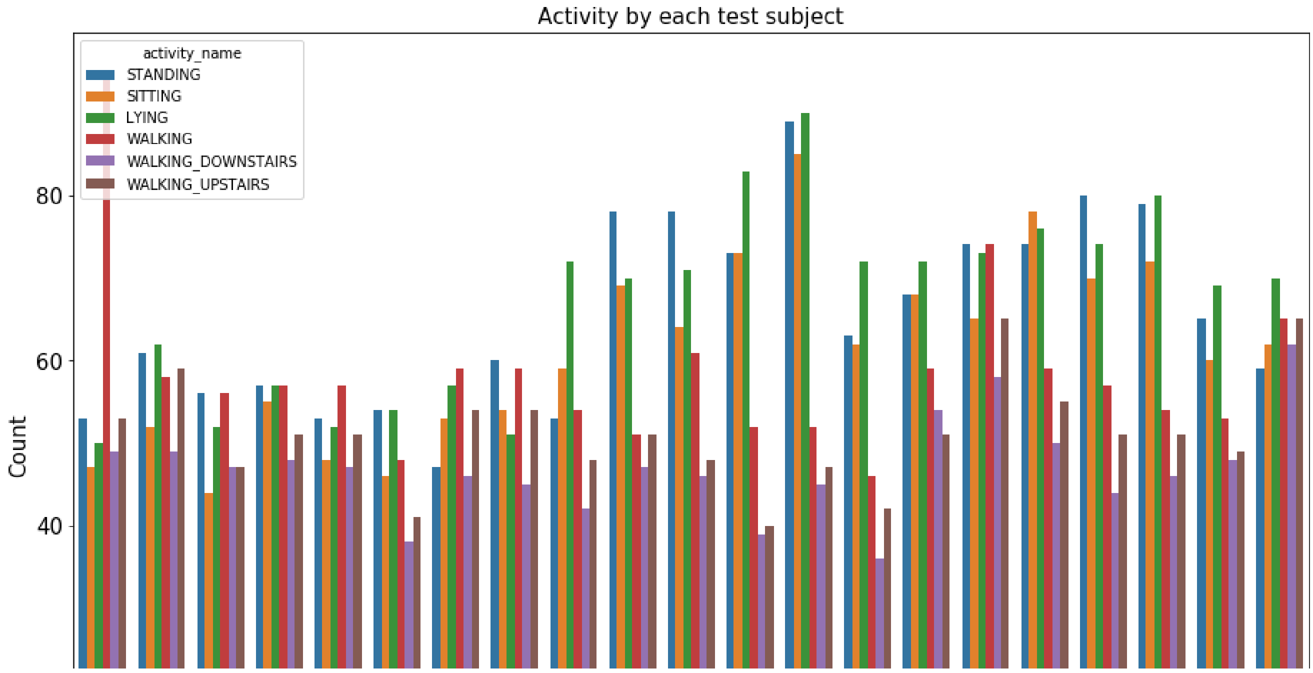
```
# Checking for duplicate values
print("Number of duplicate values in train data is "+str(sum(train_df.duplicated())))
print("Number of duplicate values in test data is "+str(sum(test_df.duplicated())))
```

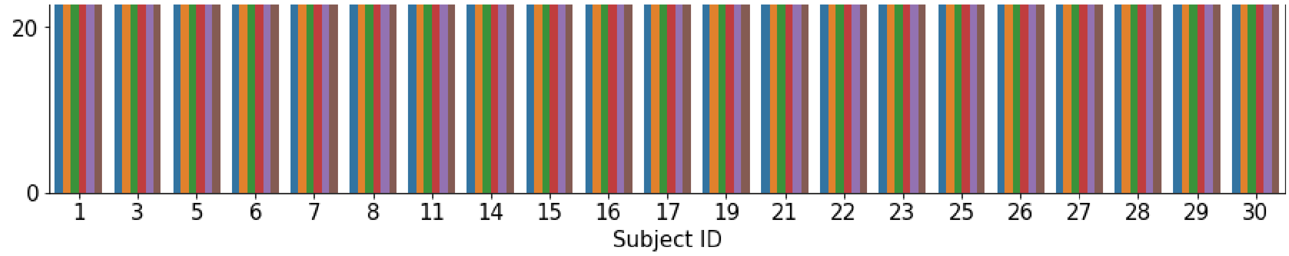
```
Number of duplicate values in train data is 0
Number of duplicate values in test data is 0
```

3. Checking for imbalance in data

In [304]:

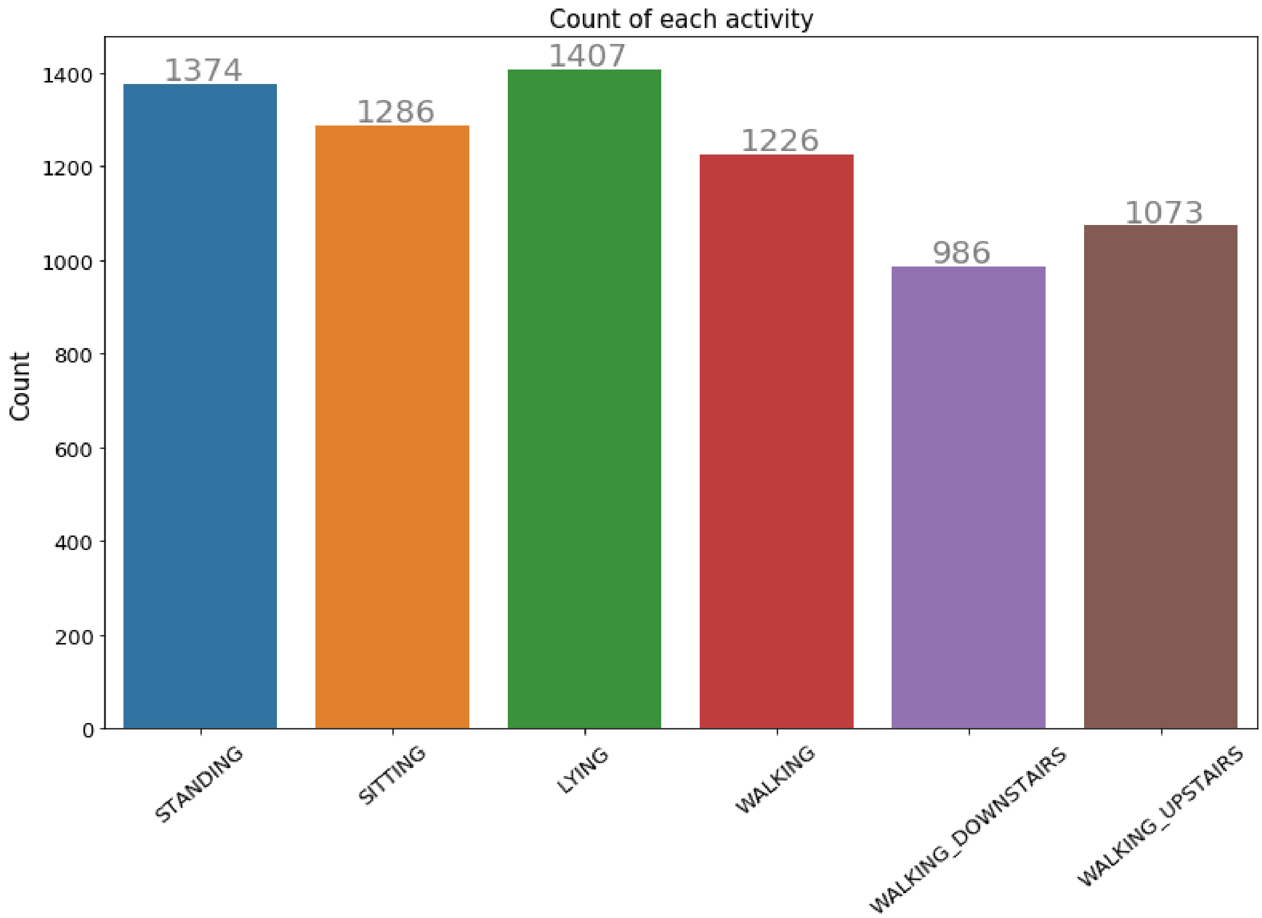
```
fig = plt.figure(figsize = (12, 8))
ax = fig.add_axes([0,0,1,1])
ax.set_title("Activity by each test subject", fontsize = 15)
plt.tick_params(labelsize = 15)
sns.countplot(x = "subject_id", hue = "activity_name", data = train_df)
plt.xlabel("Subject ID", fontsize = 15)
plt.ylabel("Count", fontsize = 15)
plt.show()
```





In [302]:

```
fig = plt.figure(figsize = (10, 6))
ax = fig.add_axes([0,0,1,1])
ax.set_title("Count of each activity", fontsize = 15)
plt.tick_params(labelsize = 10)
sns.countplot(x = "activity_name", data = train_df)
for i in ax.patches:
    ax.text(x = i.get_x() + 0.2, y = i.get_height()+10, s = str(i.get_height()), fontsize = 20, colo
r = "grey")
plt.xlabel("")
plt.ylabel("Count", fontsize = 15)
plt.tick_params(labelsize = 13)
plt.xticks(rotation = 40)
plt.show()
```



Observation
From the above two plots, we can infer that our classes are almost balanced.

4. Changing Feature Name

In [117]:

```
columns = train_df.columns
```

In [144]:

```
columns = columns.str.replace("[()]", '')
columns = columns.str.replace("-", '')
columns = columns.str.replace(",", '')
#here, columns is of type pandas index. By writing "columns.str" we have changed its type to
#pandas string. Pandas string has method called replace which we have used here.

train_df.columns = columns
test_df.columns = columns
```

In [145]:

```
train_df.columns
```

Out[145]:

```
Index(['tBodyAccmeanX', 'tBodyAccmeanY', 'tBodyAccmeanZ', 'tBodyAccstdX',
       'tBodyAccstdY', 'tBodyAccstdZ', 'tBodyAccmadX', 'tBodyAccmadY',
       'tBodyAccmadZ', 'tBodyAccmaxX',
       ...
       'angletBodyAccMeangravity', 'angletBodyAccJerkMeangravityMean',
       'angletBodyGyroMeangravityMean', 'angletBodyGyroJerkMeangravityMean',
       'angleXgravityMean', 'angleYgravityMean', 'angleZgravityMean',
       'subject_id', 'activity', 'activity_name'],
      dtype='object', length=564)
```

In [147]:

```
train_df.head()
```

Out[147]:

	tBodyAccmeanX	tBodyAccmeanY	tBodyAccmeanZ	tBodyAccstdX	tBodyAccstdY	tBodyAccstdZ	tBodyAccmadX	tBodyAccmadY	t
0	0.288585	-0.020294	-0.132905	-0.995279	-0.983111	-0.913526	-0.995112	-0.983185	
1	0.278419	-0.016411	-0.123520	-0.998245	-0.975300	-0.960322	-0.998807	-0.974914	
2	0.279653	-0.019467	-0.113462	-0.995380	-0.967187	-0.978944	-0.996520	-0.963668	
3	0.279174	-0.026201	-0.123283	-0.996091	-0.983403	-0.990675	-0.997099	-0.982750	
4	0.276629	-0.016570	-0.115362	-0.998139	-0.980817	-0.990482	-0.998321	-0.979672	

5 rows × 564 columns

In [148]:

```
test_df.head()
```

Out[148]:

	tBodyAccmeanX	tBodyAccmeanY	tBodyAccmeanZ	tBodyAccstdX	tBodyAccstdY	tBodyAccstdZ	tBodyAccmadX	tBodyAccmadY	t
0	0.257178	-0.023285	-0.014654	-0.938404	-0.920091	-0.667683	-0.952501	-0.925249	
1	0.286027	-0.013163	-0.119083	-0.975415	-0.967458	-0.944958	-0.986799	-0.968401	
2	0.275485	-0.026050	-0.118152	-0.993819	-0.969926	-0.962748	-0.994403	-0.970735	
3	0.270298	-0.032614	-0.117520	-0.994743	-0.973268	-0.967091	-0.995274	-0.974471	
4	0.274833	-0.027848	-0.129527	-0.993852	-0.967445	-0.978295	-0.994111	-0.965953	

5 rows × 564 columns

5. Saving Dataframe for future use

In [149]:

```
train_df.to_csv("../Data/train/train_df.csv", index = False)
test_df.to_csv("../Data/test/test_df.csv", index = False)
```

In [3]:

```
train_df = pd.read_csv("../Data/train/train_df.csv")
test_df = pd.read_csv("../Data/test/test_df.csv")
```

6. Exploratory Data Analysis

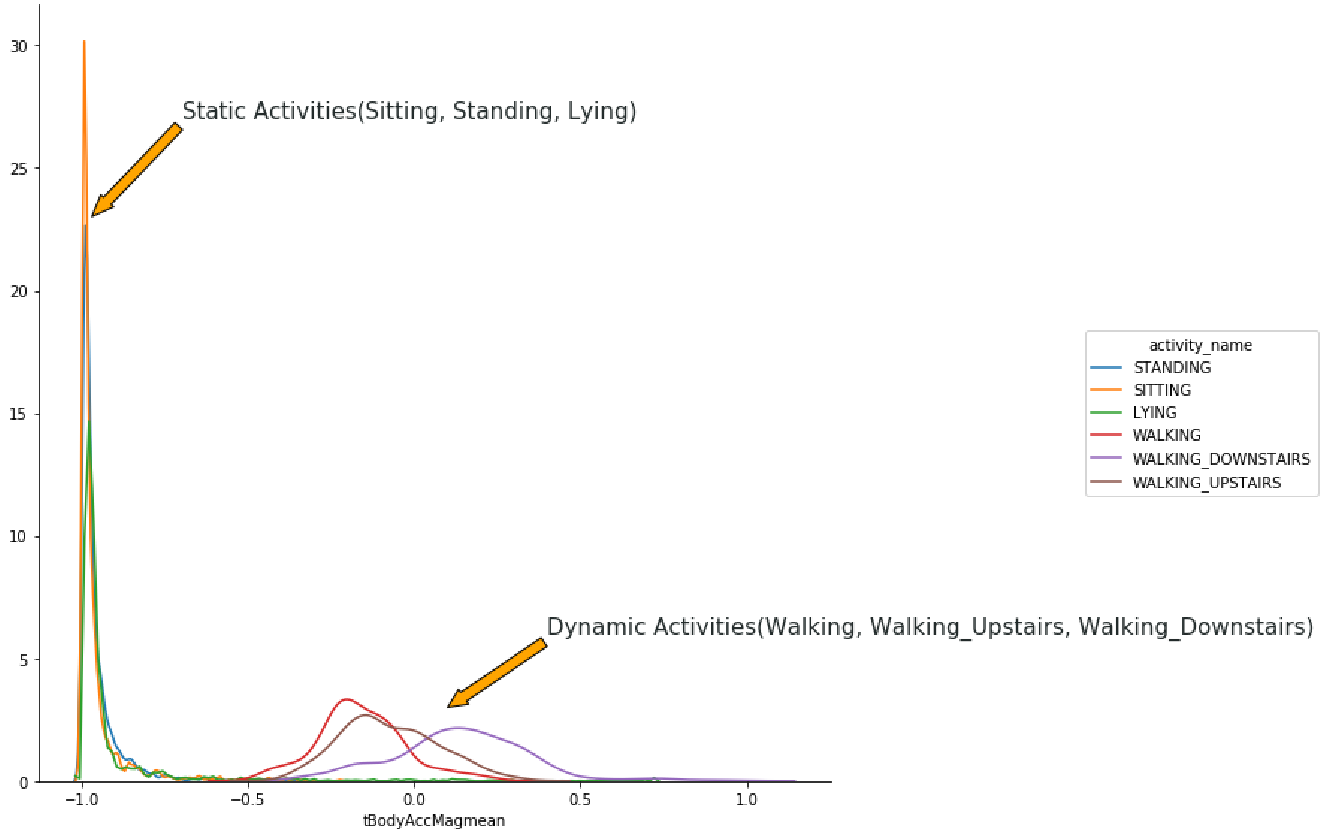
Feature information from domain knowledge

1. **Static:** We have three types static features where test subject is in rest:
    - Sitting
    - Standing
    - Lying
2. **Dynamic:** We have three types of dynamic features where test subject is in motion:
    - Walking
    - Walking_Downstairs
    - Walking_Upstairs

Magnitude of Body Accelerator Mean Matters

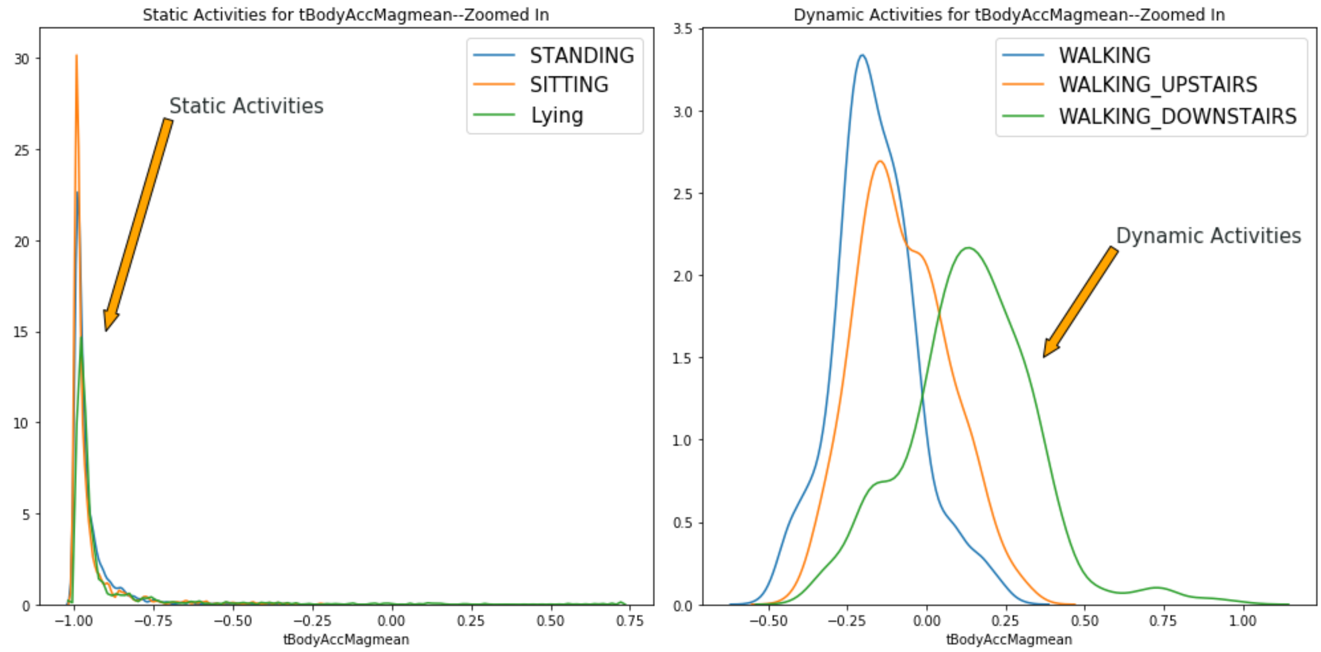
In [199]:

```
facetgrid = sns.FacetGrid(data = train_df, hue = "activity_name", size = 8)
facetgrid.map(sns.distplot, "tBodyAccMagmean", hist = False).add_legend()
plt.annotate('Static Activities(Sitting, Standing, Lying)', xy=(-0.97, 23), xytext=(-0.7, 27),
            arrowprops=dict(facecolor='orange', width = 7, headlength = 15), size = 15, color =
"#232b2b")
plt.annotate('Dynamic Activities(Walking, Walking_Upstairs, Walking_Downstairs)', xy=(0.1, 3),
xytext=(0.4, 6),
            arrowprops=dict(facecolor='orange', width = 7, headlength = 13), size = 15, color =
"#232b2b")
plt.show()
```



In [250]:

```
#let's plot "tBodyAccMagmean" for both static and dynamic activites separately to analysis them in
more detail
df_standing = train_df[train_df["activity_name"] == "STANDING"]
df_sitting = train_df[train_df["activity_name"] == "SITTING"]
df_lying = train_df[train_df["activity_name"] == "LYING"]
df_walking = train_df[train_df["activity_name"] == "WALKING"]
df_walking_upstairs = train_df[train_df["activity_name"] == "WALKING_UPSTAIRS"]
df_walking_downstairs = train_df[train_df["activity_name"] == "WALKING_DOWNSTAIRS"]

fig, axes = plt.subplots(nrows = 1, ncols = 2, figsize = (14, 7))

axes[0].set_title("Static Activities for tBodyAccMagmean--Zoomed In")
sns.distplot(df_standing["tBodyAccMagmean"], hist = False, label = "STANDING", ax = axes[0])
sns.distplot(df_sitting["tBodyAccMagmean"], hist = False, label = "SITTING", ax = axes[0])
sns.distplot(df_lying["tBodyAccMagmean"], hist = False, label = "Lying", ax = axes[0])
axes[0].legend(fontsize = 15)
axes[0].annotate('Static Activities', xy=(-0.90, 15), xytext=(-0.7, 27),
            arrowprops=dict(facecolor='orange', width = 7, headlength = 15), size = 15, color =
"#232b2b")

axes[1].set_title("Dynamic Activities for tBodyAccMagmean--Zoomed In")
sns.distplot(df_walking["tBodyAccMagmean"], hist = False, label = "WALKING", ax = axes[1])
sns.distplot(df_walking_upstairs["tBodyAccMagmean"], hist = False, label = "WALKING_UPSTAIRS", ax =
axes[1])
sns.distplot(df_walking_downstairs["tBodyAccMagmean"], hist = False, label = "WALKING_DOWNSTAIRS",
ax = axes[1])
axes[1].legend(fontsize = 15)
axes[1].annotate('Dynamic Activities', xy=(0.37, 1.5), xytext=(0.60, 2.2),
            arrowprops=dict(facecolor='orange', width = 7, headlength = 13), size = 15, color =
"#232b2b")

plt.tight_layout()
plt.show()
```



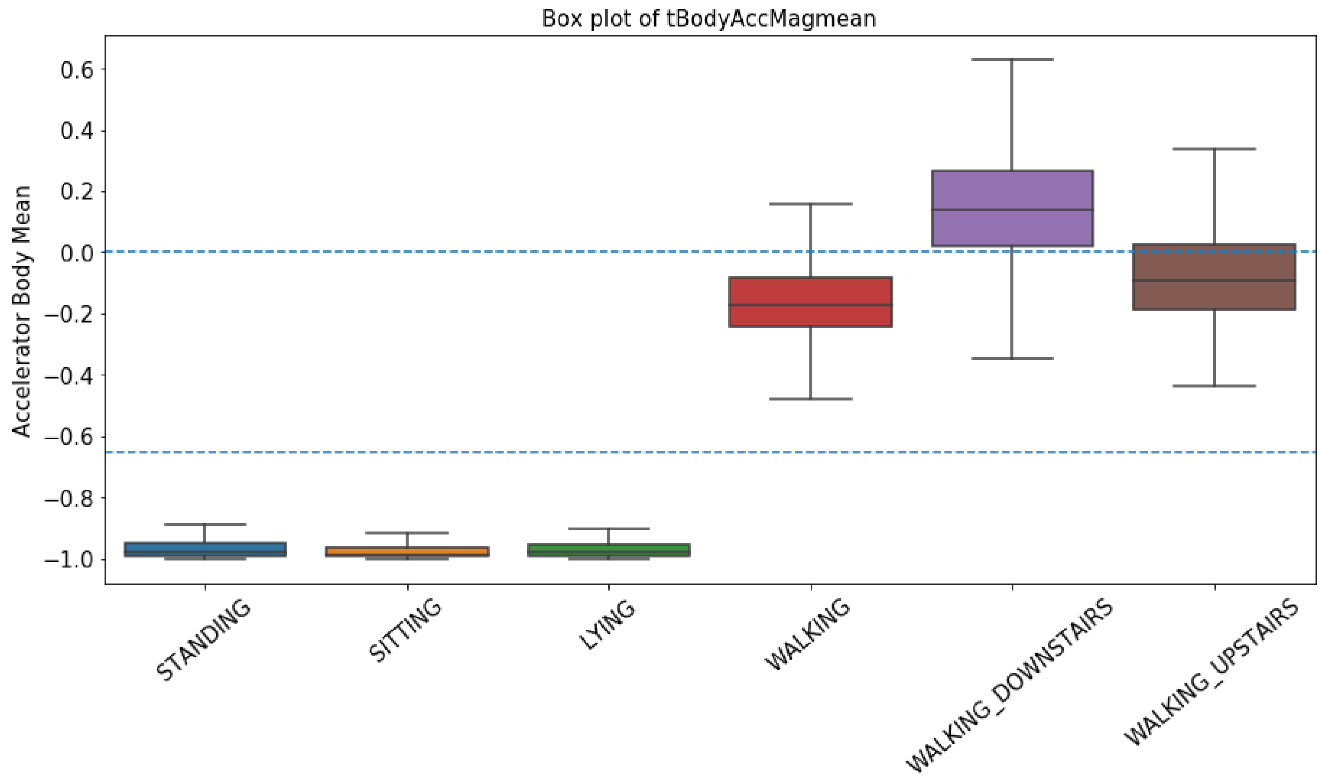
Observation

From above two plots we can clearly observe that how well "tBodyAccMagmean"--which is the magnitude of the mean of body acceleration in time-domain meaured by accelerometer--is able to separate static activity from dynamic activity. This shows that features are very carefully engineered by domian experts.

In [296]:

```
plt.figure(figsize = (15, 7))
sns.boxplot(x = "activity_name", y = "tBodyAccMagmean", showfliers = False, data = train_df)
plt.axhline(y = -0.65, linestyle = "--")
plt.axhline(y = 0, linestyle = "--")
plt.title("Box plot of tBodyAccMagmean", fontsize = 15)
plt.ylabel("Accelerator Body Mean", fontsize = 15)
plt.xlabel("Activity Name", fontsize = 15)
```

```
plt.xlabel("")
plt.tick_params(labelsize = 15)
plt.xticks(rotation = 40)
plt.show()
```



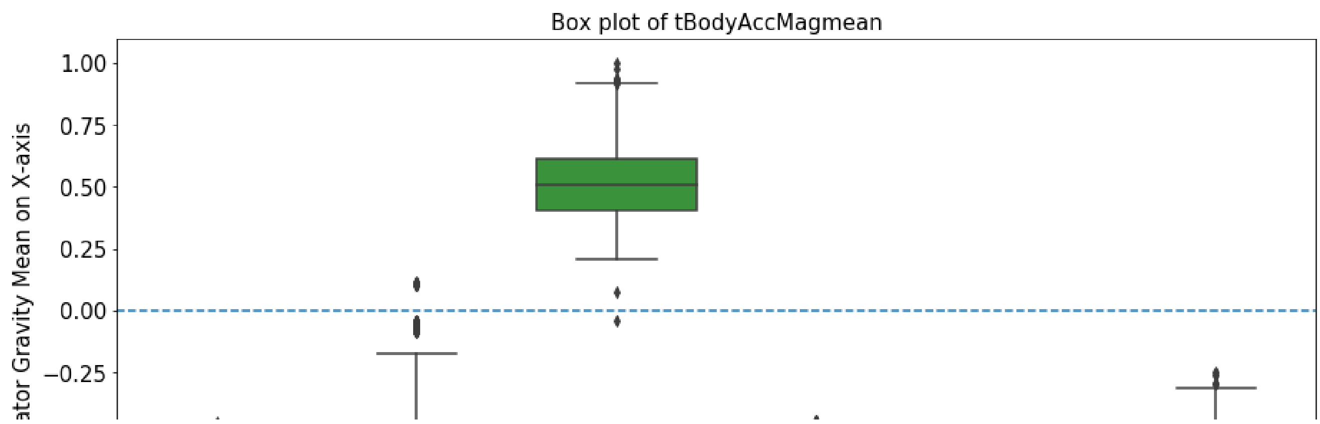
Observations:

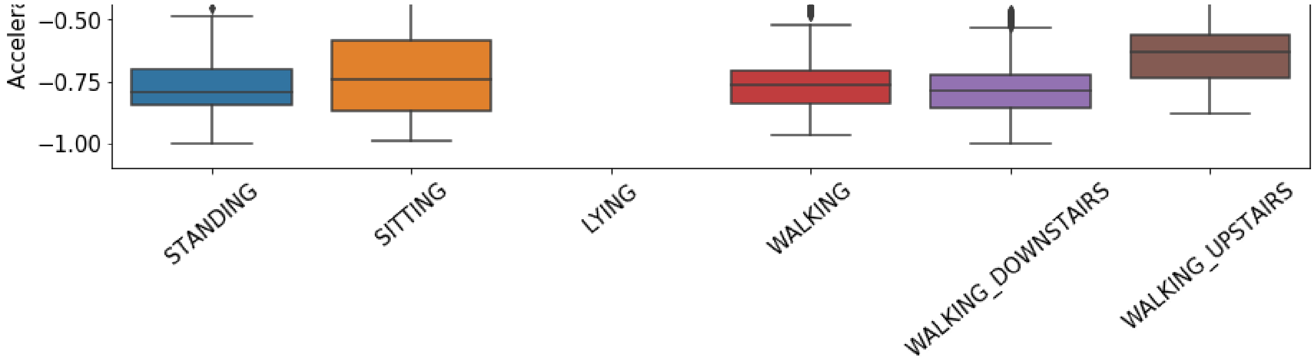
- If tAccMean is < -0.8 then the Activities are either Standing or Sitting or Laying.
- If tAccMean is > -0.6 then the Activities are either Walking or WalkingDownstairs or WalkingUpstairs.
- If tAccMean > 0.0 then the Activity is WalkingDownstairs.
- We can classify 75% the Acitivity labels with some errors.

Accelerator Gravity Mean on X-axis can be quite important

In [299]:

```
plt.figure(figsize = (15, 7))
sns.boxplot(x = "activity_name", y = "angleXgravityMean", showfliers = True, data = train_df)
plt.axhline(y = 0, linestyle = "--")
plt.title("Box plot of tBodyAccMagmean", fontsize = 15)
plt.ylabel("Accelerator Gravity Mean on X-axis", fontsize = 15)
plt.xlabel("")
plt.tick_params(labelsize = 15)
plt.xticks(rotation = 40)
plt.show()
```





Observation

- If Acc Gravity Mean > 0, we can infer that the activity will most likely be **Lying**.
- If Acc Gravity Mean < 0, we can infer that the activity can be anything but **Lying**.

7. Applying T-SNE on Data

In [75]:

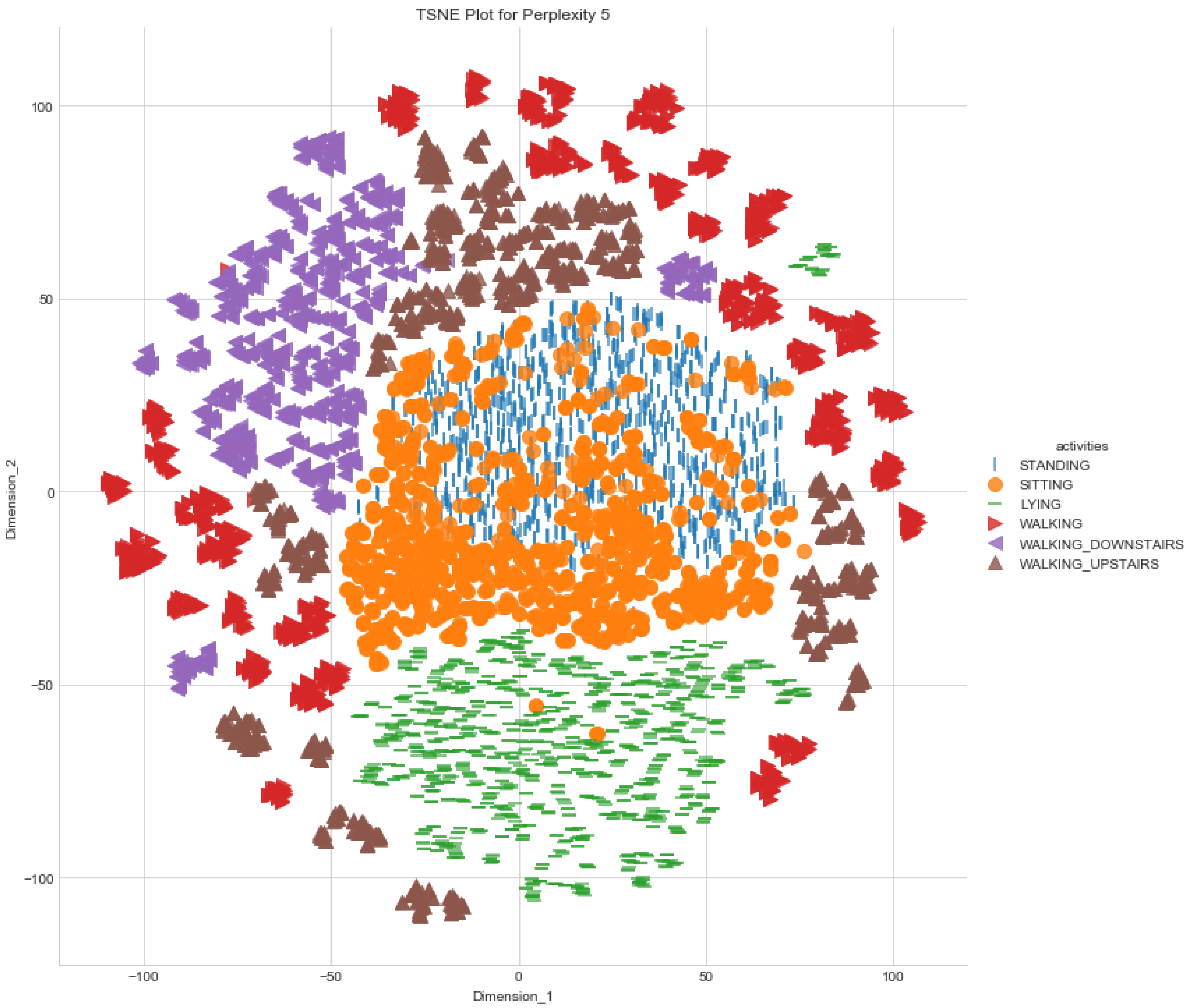
```
def plt_tsne(perplexity, train_df):
    data = train_df.drop(["subject_id", "activity", "activity_name"], axis = 1)
    data_label = train_df["activity_name"]
    applying_tsne = TSNE(n_components = 2, perplexity = perplexity, n_iter = 1000, verbose = 2)
    reduced_dim = applying_tsne.fit_transform(data)
    d = {'Dimension_1': applying_tsne.embedding_[:,0], 'Dimension_2': applying_tsne.embedding_[:,1]
, "activities":data_label}
    df = pd.DataFrame(data = d)
    print("Done...")
    print("Plotting TSNE Visualization...")
    sns.set_style('whitegrid')
    sns.lmplot("Dimension_1", "Dimension_2", df, hue = 'activities', markers=['|','o','_', ">", "<"
, "^"], fit_reg = False, size = 10, scatter_kws={'s':100})
    plt.title("TSNE Plot for Perplexity "+str(perplexity))
    plt.show()
```

In [76]:

```
perplexities = [5, 10, 20, 40, 100]
for perplexity in perplexities:
    plt_tsne(perplexity, train_df)
```

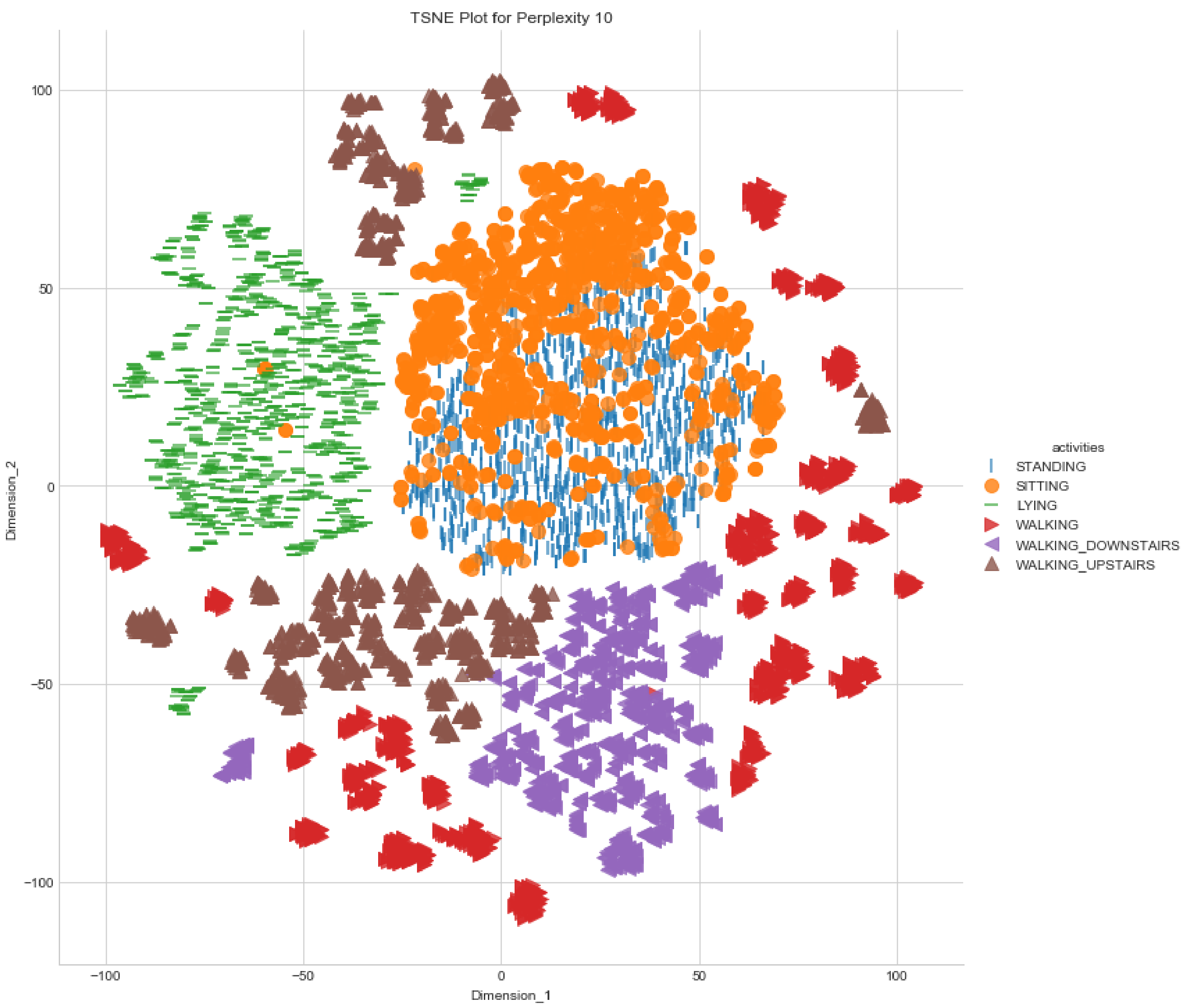
```
[t-SNE] Computing 16 nearest neighbors...
[t-SNE] Indexed 7352 samples in 0.335s...
[t-SNE] Computed neighbors for 7352 samples in 43.932s...
[t-SNE] Computed conditional probabilities for sample 1000 / 7352
[t-SNE] Computed conditional probabilities for sample 2000 / 7352
[t-SNE] Computed conditional probabilities for sample 3000 / 7352
[t-SNE] Computed conditional probabilities for sample 4000 / 7352
[t-SNE] Computed conditional probabilities for sample 5000 / 7352
[t-SNE] Computed conditional probabilities for sample 6000 / 7352
[t-SNE] Computed conditional probabilities for sample 7000 / 7352
[t-SNE] Computed conditional probabilities for sample 7352 / 7352
[t-SNE] Mean sigma: 0.961265
[t-SNE] Computed conditional probabilities in 0.073s
[t-SNE] Iteration 50: error = 113.8233261, gradient norm = 0.0235335 (50 iterations in 14.634s)
[t-SNE] Iteration 100: error = 97.6684570, gradient norm = 0.0148992 (50 iterations in 8.922s)
[t-SNE] Iteration 150: error = 93.1876678, gradient norm = 0.0094125 (50 iterations in 7.373s)
[t-SNE] Iteration 200: error = 91.2166061, gradient norm = 0.0067544 (50 iterations in 6.872s)
[t-SNE] Iteration 250: error = 90.0454941, gradient norm = 0.0046577 (50 iterations in 6.981s)
[t-SNE] KL divergence after 250 iterations with early exaggeration: 90.045494
[t-SNE] Iteration 300: error = 3.5713027, gradient norm = 0.0014567 (50 iterations in 7.488s)
[t-SNE] Iteration 350: error = 2.8163037, gradient norm = 0.0007607 (50 iterations in 6.976s)
[t-SNE] Iteration 400: error = 2.4362845, gradient norm = 0.0005298 (50 iterations in 6.660s)
[t-SNE] Iteration 450: error = 2.2200058, gradient norm = 0.0004020 (50 iterations in 7.274s)
[t-SNE] Iteration 500: error = 2.0754416, gradient norm = 0.0003333 (50 iterations in 7.076s)
[t-SNE] Iteration 550: error = 1.9702364, gradient norm = 0.0002839 (50 iterations in 6.718s)
[t-SNE] Iteration 600: error = 1.8892900, gradient norm = 0.0002465 (50 iterations in 7.395s)
```

```
[t-SNE] Iteration 650: error = 1.8242882, gradient norm = 0.0002178 (50 iterations in 7.038s)
[t-SNE] Iteration 700: error = 1.7706470, gradient norm = 0.0001978 (50 iterations in 6.820s)
[t-SNE] Iteration 750: error = 1.7253084, gradient norm = 0.0001825 (50 iterations in 6.719s)
[t-SNE] Iteration 800: error = 1.6863036, gradient norm = 0.0001652 (50 iterations in 6.794s)
[t-SNE] Iteration 850: error = 1.6524775, gradient norm = 0.0001523 (50 iterations in 6.793s)
[t-SNE] Iteration 900: error = 1.6227095, gradient norm = 0.0001437 (50 iterations in 6.841s)
[t-SNE] Iteration 950: error = 1.5959746, gradient norm = 0.0001343 (50 iterations in 6.751s)
[t-SNE] Iteration 1000: error = 1.5721576, gradient norm = 0.0001280 (50 iterations in 7.715s)
[t-SNE] Error after 1000 iterations: 1.572158
Done...
Plotting TSNE Visualization...
```




```
[t-SNE] Computing 31 nearest neighbors...
[t-SNE] Indexed 7352 samples in 0.400s...
[t-SNE] Computed neighbors for 7352 samples in 43.159s...
[t-SNE] Computed conditional probabilities for sample 1000 / 7352
[t-SNE] Computed conditional probabilities for sample 2000 / 7352
[t-SNE] Computed conditional probabilities for sample 3000 / 7352
[t-SNE] Computed conditional probabilities for sample 4000 / 7352
[t-SNE] Computed conditional probabilities for sample 5000 / 7352
[t-SNE] Computed conditional probabilities for sample 6000 / 7352
[t-SNE] Computed conditional probabilities for sample 7000 / 7352
[t-SNE] Computed conditional probabilities for sample 7352 / 7352
[t-SNE] Mean sigma: 1.133828
[t-SNE] Computed conditional probabilities in 0.163s
[t-SNE] Iteration 50: error = 105.7820053, gradient norm = 0.0174431 (50 iterations in 13.429s)
[t-SNE] Iteration 100: error = 90.8498993, gradient norm = 0.0124366 (50 iterations in 9.540s)
[t-SNE] Iteration 150: error = 87.5110779, gradient norm = 0.0073947 (50 iterations in 8.205s)
[t-SNE] Iteration 200: error = 86.1822968, gradient norm = 0.0053608 (50 iterations in 7.826s)
[t-SNE] Iteration 250: error = 85.4495468, gradient norm = 0.0037724 (50 iterations in 7.975s)
[t-SNE] KL divergence after 250 iterations with early exaggeration: 85.449547
[t-SNE] Iteration 300: error = 3.1341319, gradient norm = 0.0013910 (50 iterations in 7.986s)
[t-SNE] Iteration 350: error = 2.4909160, gradient norm = 0.0006464 (50 iterations in 7.810s)
[t-SNE] Iteration 400: error = 2.1722710, gradient norm = 0.0004236 (50 iterations in 7.778s)
```

```
[t-SNE] Iteration 450: error = 1.9877188, gradient norm = 0.0003173 (50 iterations in 7.876s)
[t-SNE] Iteration 500: error = 1.8698498, gradient norm = 0.0002527 (50 iterations in 7.995s)
[t-SNE] Iteration 550: error = 1.7864486, gradient norm = 0.0002117 (50 iterations in 8.088s)
[t-SNE] Iteration 600: error = 1.7234244, gradient norm = 0.0001810 (50 iterations in 8.063s)
[t-SNE] Iteration 650: error = 1.6743083, gradient norm = 0.0001619 (50 iterations in 8.131s)
[t-SNE] Iteration 700: error = 1.6350037, gradient norm = 0.0001427 (50 iterations in 8.610s)
[t-SNE] Iteration 750: error = 1.6023960, gradient norm = 0.0001304 (50 iterations in 7.868s)
[t-SNE] Iteration 800: error = 1.5749978, gradient norm = 0.0001206 (50 iterations in 8.296s)
[t-SNE] Iteration 850: error = 1.5515244, gradient norm = 0.0001114 (50 iterations in 8.187s)
[t-SNE] Iteration 900: error = 1.5317587, gradient norm = 0.0001023 (50 iterations in 8.231s)
[t-SNE] Iteration 950: error = 1.5143646, gradient norm = 0.0000989 (50 iterations in 7.952s)
[t-SNE] Iteration 1000: error = 1.4989291, gradient norm = 0.0000920 (50 iterations in 7.828s)
[t-SNE] Error after 1000 iterations: 1.498929
Done...
Plotting TSNE Visualization...
```

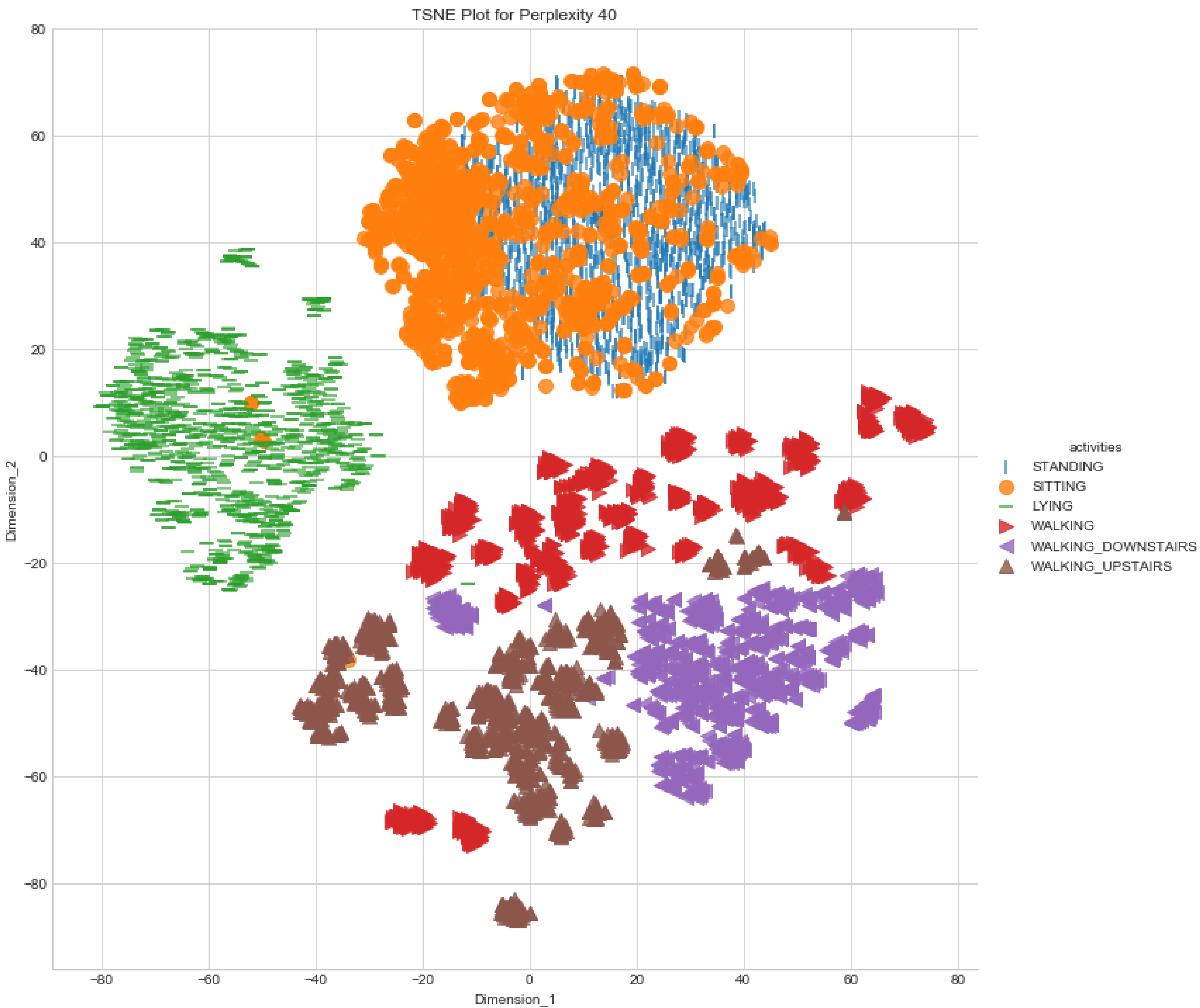



```
[t-SNE] Computing 61 nearest neighbors...
[t-SNE] Indexed 7352 samples in 0.336s...
[t-SNE] Computed neighbors for 7352 samples in 43.906s...
[t-SNE] Computed conditional probabilities for sample 1000 / 7352
[t-SNE] Computed conditional probabilities for sample 2000 / 7352
[t-SNE] Computed conditional probabilities for sample 3000 / 7352
[t-SNE] Computed conditional probabilities for sample 4000 / 7352
[t-SNE] Computed conditional probabilities for sample 5000 / 7352
[t-SNE] Computed conditional probabilities for sample 6000 / 7352
[t-SNE] Computed conditional probabilities for sample 7000 / 7352
[t-SNE] Computed conditional probabilities for sample 7352 / 7352
[t-SNE] Mean sigma: 1.274335
[t-SNE] Computed conditional probabilities in 0.287s
[t-SNE] Iteration 50: error = 97.7753448, gradient norm = 0.0145347 (50 iterations in 19.519s)
[t-SNE] Iteration 100: error = 84.2433472, gradient norm = 0.0088132 (50 iterations in 11.848s)
[t-SNE] Iteration 150: error = 82.0076218, gradient norm = 0.0035071 (50 iterations in 10.412s)
[t-SNE] Iteration 200: error = 81.1837006, gradient norm = 0.0022608 (50 iterations in 10.294s)
```

```
[t-SNE] Iteration 250: error = 80.7715073, gradient norm = 0.0020507 (50 iterations in 9.802s)
[t-SNE] KL divergence after 250 iterations with early exaggeration: 80.771507
[t-SNE] Iteration 300: error = 2.7096515, gradient norm = 0.0013108 (50 iterations in 9.852s)
[t-SNE] Iteration 350: error = 2.1729641, gradient norm = 0.0005774 (50 iterations in 9.417s)
[t-SNE] Iteration 400: error = 1.9221689, gradient norm = 0.0003486 (50 iterations in 9.773s)
[t-SNE] Iteration 450: error = 1.7748548, gradient norm = 0.0002490 (50 iterations in 9.754s)
[t-SNE] Iteration 500: error = 1.6807389, gradient norm = 0.0001933 (50 iterations in 9.629s)
[t-SNE] Iteration 550: error = 1.6163493, gradient norm = 0.0001588 (50 iterations in 9.849s)
[t-SNE] Iteration 600: error = 1.5696250, gradient norm = 0.0001362 (50 iterations in 9.797s)
[t-SNE] Iteration 650: error = 1.5341796, gradient norm = 0.0001188 (50 iterations in 9.705s)
[t-SNE] Iteration 700: error = 1.5064334, gradient norm = 0.0001088 (50 iterations in 9.812s)
[t-SNE] Iteration 750: error = 1.4845377, gradient norm = 0.0000992 (50 iterations in 10.047s)
[t-SNE] Iteration 800: error = 1.4666576, gradient norm = 0.0000895 (50 iterations in 9.794s)
[t-SNE] Iteration 850: error = 1.4516509, gradient norm = 0.0000843 (50 iterations in 9.835s)
[t-SNE] Iteration 900: error = 1.4388338, gradient norm = 0.0000795 (50 iterations in 9.798s)
[t-SNE] Iteration 950: error = 1.4279175, gradient norm = 0.0000735 (50 iterations in 10.697s)
[t-SNE] Iteration 1000: error = 1.4185984, gradient norm = 0.0000725 (50 iterations in 10.158s)
[t-SNE] Error after 1000 iterations: 1.418598
Done...
Plotting TSNE Visualization...
```


```
[t-SNE] Computing 121 nearest neighbors...
[t-SNE] Indexed 7352 samples in 0.496s...
[t-SNE] Computed neighbors for 7352 samples in 46.739s...
[t-SNE] Computed conditional probabilities for sample 1000 / 7352
[t-SNE] Computed conditional probabilities for sample 2000 / 7352
[t-SNE] Computed conditional probabilities for sample 3000 / 7352
[t-SNE] Computed conditional probabilities for sample 4000 / 7352
[t-SNE] Computed conditional probabilities for sample 5000 / 7352
[t-SNE] Computed conditional probabilities for sample 6000 / 7352
[t-SNE] Computed conditional probabilities for sample 7000 / 7352
[t-SNE] Computed conditional probabilities for sample 7352 / 7352
[t-SNE] Mean sigma: 1.399086
[t-SNE] Computed conditional probabilities in 0.532s
```

```
[t-SNE] Iteration 50: error = 88.6822128, gradient norm = 0.0260302 (50 iterations in 26.060s)
[t-SNE] Iteration 100: error = 77.6090622, gradient norm = 0.0048039 (50 iterations in 17.536s)
[t-SNE] Iteration 150: error = 76.4387817, gradient norm = 0.0038548 (50 iterations in 15.788s)
[t-SNE] Iteration 200: error = 76.0391006, gradient norm = 0.0016221 (50 iterations in 15.902s)
[t-SNE] Iteration 250: error = 75.8269119, gradient norm = 0.0013776 (50 iterations in 15.330s)
[t-SNE] KL divergence after 250 iterations with early exaggeration: 75.826912
[t-SNE] Iteration 300: error = 2.2853305, gradient norm = 0.0012208 (50 iterations in 14.434s)
[t-SNE] Iteration 350: error = 1.8533092, gradient norm = 0.0005086 (50 iterations in 15.058s)
[t-SNE] Iteration 400: error = 1.6659527, gradient norm = 0.0002964 (50 iterations in 14.569s)
[t-SNE] Iteration 450: error = 1.5599132, gradient norm = 0.0002017 (50 iterations in 13.650s)
[t-SNE] Iteration 500: error = 1.4917234, gradient norm = 0.0001502 (50 iterations in 14.235s)
[t-SNE] Iteration 550: error = 1.4452350, gradient norm = 0.0001227 (50 iterations in 14.392s)
[t-SNE] Iteration 600: error = 1.4121413, gradient norm = 0.0001023 (50 iterations in 14.041s)
[t-SNE] Iteration 650: error = 1.3877604, gradient norm = 0.0000891 (50 iterations in 13.686s)
[t-SNE] Iteration 700: error = 1.3694947, gradient norm = 0.0000828 (50 iterations in 13.621s)
[t-SNE] Iteration 750: error = 1.3561211, gradient norm = 0.0000758 (50 iterations in 13.897s)
[t-SNE] Iteration 800: error = 1.3460970, gradient norm = 0.0000728 (50 iterations in 14.451s)
[t-SNE] Iteration 850: error = 1.3382318, gradient norm = 0.0000689 (50 iterations in 13.671s)
[t-SNE] Iteration 900: error = 1.3320208, gradient norm = 0.0000656 (50 iterations in 14.103s)
[t-SNE] Iteration 950: error = 1.3267668, gradient norm = 0.0000636 (50 iterations in 14.214s)
[t-SNE] Iteration 1000: error = 1.3224055, gradient norm = 0.0000612 (50 iterations in 13.662s)
[t-SNE] Error after 1000 iterations: 1.322405
Done...
Plotting TSNE Visualization...
```




```
[t-SNE] Computing 301 nearest neighbors...
[t-SNE] Indexed 7352 samples in 0.417s...
[t-SNE] Computed neighbors for 7352 samples in 47.684s...
[t-SNE] Computed conditional probabilities for sample 1000 / 7352
[t-SNE] Computed conditional probabilities for sample 2000 / 7352
[t-SNE] Computed conditional probabilities for sample 3000 / 7352
[t-SNE] Computed conditional probabilities for sample 4000 / 7352
[t-SNE] Computed conditional probabilities for sample 5000 / 7352
[t-SNE] Computed conditional probabilities for sample 6000 / 7352
```

```
[t-SNE] Computed conditional probabilities for sample 7000 / 7352
[t-SNE] Computed conditional probabilities for sample 7352 / 7352
[t-SNE] Mean sigma: 1.559265
[t-SNE] Computed conditional probabilities in 1.366s
[t-SNE] Iteration 50: error = 77.9275742, gradient norm = 0.0171849 (50 iterations in 30.204s)
[t-SNE] Iteration 100: error = 68.2980347, gradient norm = 0.0049000 (50 iterations in 27.590s)
[t-SNE] Iteration 150: error = 67.7081375, gradient norm = 0.0018278 (50 iterations in 25.309s)
[t-SNE] Iteration 200: error = 67.5039749, gradient norm = 0.0012888 (50 iterations in 25.034s)
[t-SNE] Iteration 250: error = 67.3914261, gradient norm = 0.0010411 (50 iterations in 25.121s)
[t-SNE] KL divergence after 250 iterations with early exaggeration: 67.391426
[t-SNE] Iteration 300: error = 1.7983552, gradient norm = 0.0011949 (50 iterations in 26.464s)
[t-SNE] Iteration 350: error = 1.4792659, gradient norm = 0.0004503 (50 iterations in 26.253s)
[t-SNE] Iteration 400: error = 1.3532579, gradient norm = 0.0002466 (50 iterations in 26.401s)
[t-SNE] Iteration 450: error = 1.2853377, gradient norm = 0.0001623 (50 iterations in 25.243s)
[t-SNE] Iteration 500: error = 1.2440071, gradient norm = 0.0001169 (50 iterations in 25.218s)
[t-SNE] Iteration 550: error = 1.2169261, gradient norm = 0.0000916 (50 iterations in 25.201s)
[t-SNE] Iteration 600: error = 1.1973919, gradient norm = 0.0000779 (50 iterations in 25.182s)
[t-SNE] Iteration 650: error = 1.1837749, gradient norm = 0.0000652 (50 iterations in 25.648s)
[t-SNE] Iteration 700: error = 1.1736444, gradient norm = 0.0000581 (50 iterations in 25.783s)
[t-SNE] Iteration 750: error = 1.1661189, gradient norm = 0.0000535 (50 iterations in 26.009s)
[t-SNE] Iteration 800: error = 1.1605114, gradient norm = 0.0000497 (50 iterations in 26.155s)
[t-SNE] Iteration 850: error = 1.1565733, gradient norm = 0.0000466 (50 iterations in 26.159s)
[t-SNE] Iteration 900: error = 1.1532556, gradient norm = 0.0000440 (50 iterations in 27.499s)
[t-SNE] Iteration 950: error = 1.1506367, gradient norm = 0.0000423 (50 iterations in 25.935s)
[t-SNE] Iteration 1000: error = 1.1484059, gradient norm = 0.0000399 (50 iterations in 26.400s)
[t-SNE] Error after 1000 iterations: 1.148406
Done...
Plotting TSNE Visualization...
```

Observation

From above TSNE plots, we can observe that except **STANDING** and **SITTING**, all other activities are separated fairly well.

8. Machine Learning Models

In [82]:

```
x_train = train_df.drop(["subject_id", "activity", "activity_name"], axis = 1)
y_train = train_df["activity"]

x_test = test_df.drop(["subject_id", "activity", "activity_name"], axis = 1)
y_test = test_df["activity"]

x_train.shape, y_train.shape, x_test.shape, y_test.shape
```

Out[82]:

```
((7352, 561), (7352,), (2947, 561), (2947,))
```

In [166]:

```
table = pd.DataFrame(columns = ["Model", "Accuracy(%)"])
def keeping_record(model_name, accuracy):
    global table
    table = table.append(pd.DataFrame([[model_name, accuracy]], columns = ["Model", "Accuracy(%)"])
)
    table.reset_index(drop = True, inplace = True)
```

In [2]:

```
def print_confusionMatrix(Y_TestLabels, PredictedLabels):
    confusionMatx = confusion_matrix(Y_TestLabels, PredictedLabels)

    precision = confusionMatx/confusionMatx.sum(axis = 0)

    recall = (confusionMatx.T/confusionMatx.sum(axis = 1)).T

    sns.set(font_scale=1.5)

    # confusionMatx = [[1, 2],
    #                  [3, 4]]
    # confusionMatx.T = [[1, 3],
    #                    [2, 4]]
    # confusionMatx.sum(axis = 1)  axis=0 corresponds to columns and axis=1 corresponds to rows in
two diamensional array
    # confusionMatx.sum(axix =1) = [[3, 7]]
    # (confusionMatx.T)/(confusionMatx.sum(axis=1)) = [[1/3, 3/7]
    #                                                  [2/3, 4/7]]

    # (confusionMatx.T)/(confusionMatx.sum(axis=1)).T = [[1/3, 2/3]
    #                                                    [3/7, 4/7]]
    # sum of row elements = 1

    labels = ["WALKING", "WALKING_UPSTAIRS", "WALKING_DOWNSTAIRS", "SITTING", "STANDING", "LYING"]

    plt.figure(figsize=(16,7))
    sns.heatmap(confusionMatx, cmap = "Blues", annot = True, fmt = ".1f", xticklabels=labels, ytick
labels=labels)
    plt.title("Confusion Matrix", fontsize = 30)
    plt.xlabel('Predicted Class', fontsize = 20)
    plt.ylabel('Original Class', fontsize = 20)
    plt.tick_params(labelsize = 15)
    plt.xticks(rotation = 90)
    plt.show()

    print("-"*125)

    plt.figure(figsize=(16,7))
    sns.heatmap(precision, cmap = "Blues", annot = True, fmt = ".2f", xticklabels=labels,
yticklabels=labels)
    plt.title("Precision Matrix", fontsize = 30)
    plt.xlabel('Predicted Class', fontsize = 20)
    plt.ylabel('Original Class', fontsize = 20)
    plt.tick_params(labelsize = 15)
    plt.xticks(rotation = 90)
    plt.show()
```

```
    print("-"*125)

    plt.figure(figsize=(16,7))
    sns.heatmap(recall, cmap = "Blues", annot = True, fmt = ".2f", xticklabels=labels, yticklabels=
labels)
    plt.title("Recall Matrix", fontsize = 30)
    plt.xlabel('Predicted Class', fontsize = 20)
    plt.ylabel('Original Class', fontsize = 20)
    plt.tick_params(labelsize = 15)
    plt.xticks(rotation = 90)
    plt.show()
```

In [160]:

```
def apply_model(cross_val, x_train, y_train, x_test, y_test, model_name):
    start = datetime.now()
    cross_val.fit(x_train, y_train)
    predicted_points = cross_val.predict(x_test)

    print("Total time taken for tuning hyperparameter and making prediction by the model is
(HH:MM:SS): {}\n".format(datetime.now() - start))
    accuracy = np.round(accuracy_score(y_test, predicted_points)*100, 2)

    print('---------------------')
    print('|      Accuracy      |')
    print('---------------------')
    print(str(accuracy)+"%\n")

    print('---------------------------')
    print('|      Best Estimator      |')
    print('---------------------------')
    print("{}\n".format(cross_val.best_estimator_))

    print('----------------------------------')
    print('|      Best Hyper-Parameters      |')
    print('----------------------------------')
    print(cross_val.best_params_)

    keeping_record(model_name, accuracy)

    print("\n\n")

    print_confusionMatrix(y_test, predicted_points)
```

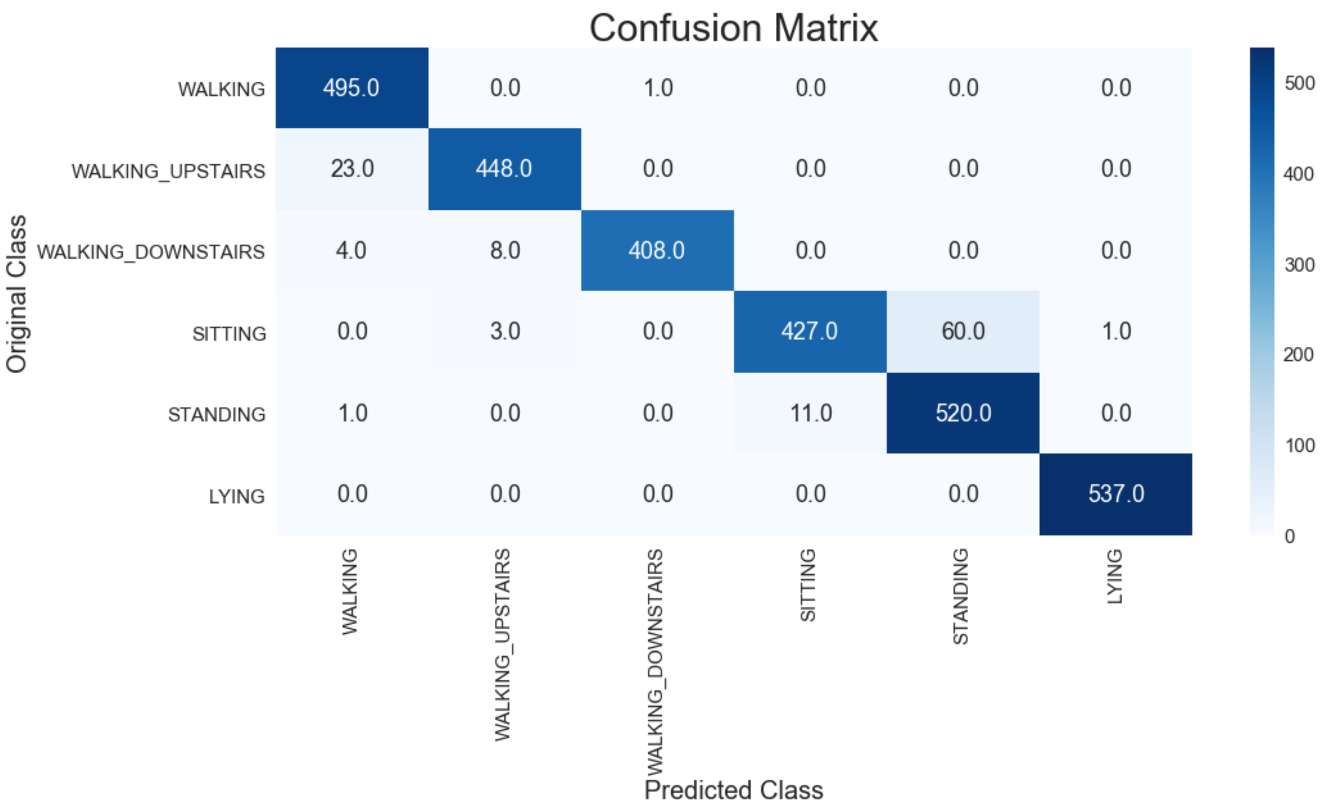
8.1 Logistic Regression

In [167]:

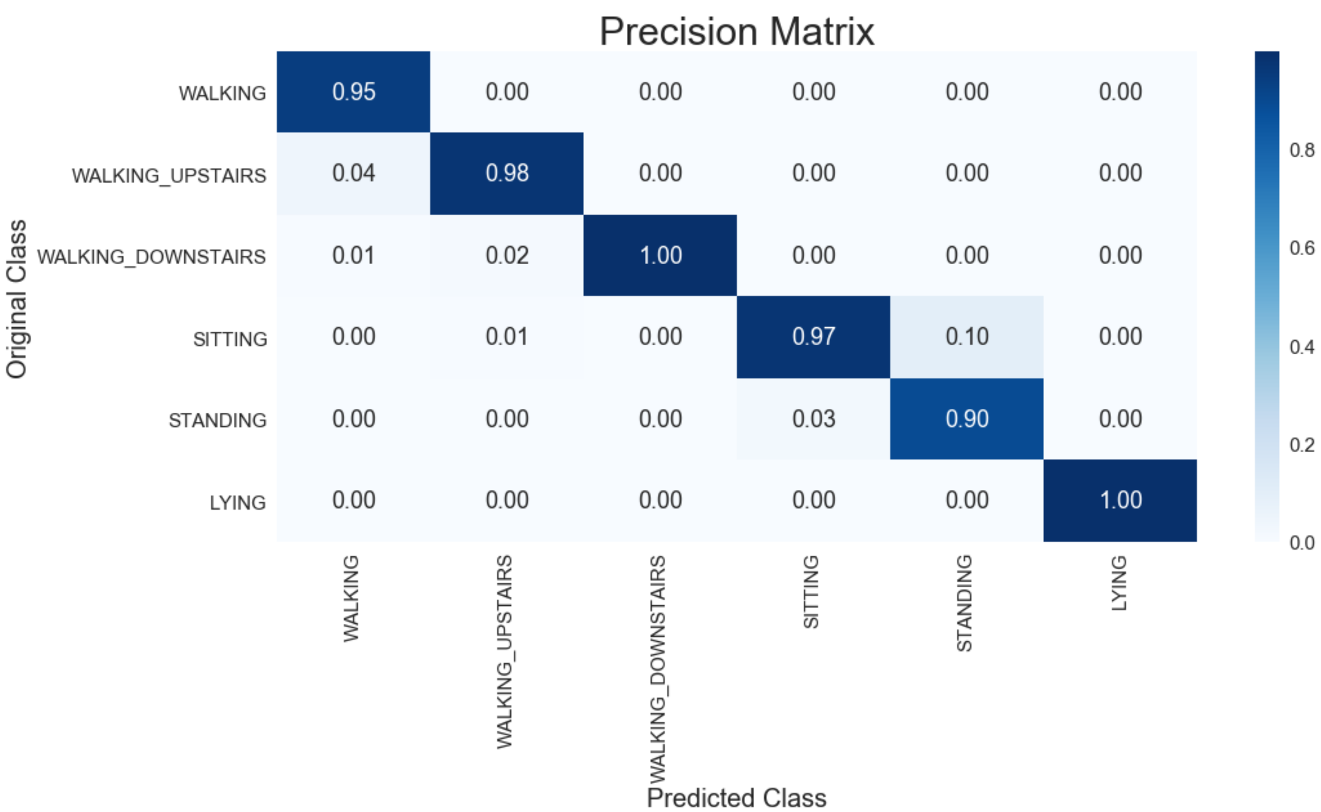
```
parameters = {"C": [0.001, 0.01, 0.1, 1, 10**1, 10**2, 10**3], "penalty": ["l1", "l2"]}
clf = LogisticRegression(multi_class = "ovr")
cross_val = GridSearchCV(clf, parameters, cv=3)
apply_model(cross_val, x_train, y_train, x_test, y_test, "Logistic Regression")
```

```
Total time taken for tuning hyperparameter and making prediction by the model is (HH:MM:SS): 0:03:
40.871923

---------------------
|      Accuracy      |
---------------------
96.2%

---------------------------
|      Best Estimator      |
---------------------------
LogisticRegression(C=10, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='ovr', n_jobs=1,
          penalty='l2', random_state=None, solver='liblinear', tol=0.0001,
          verbose=0, warm_start=False)

----------------------------------
|      Best Hyper-Parameters      |
----------------------------------
{'C': 10, 'penalty': 'l2'}
```

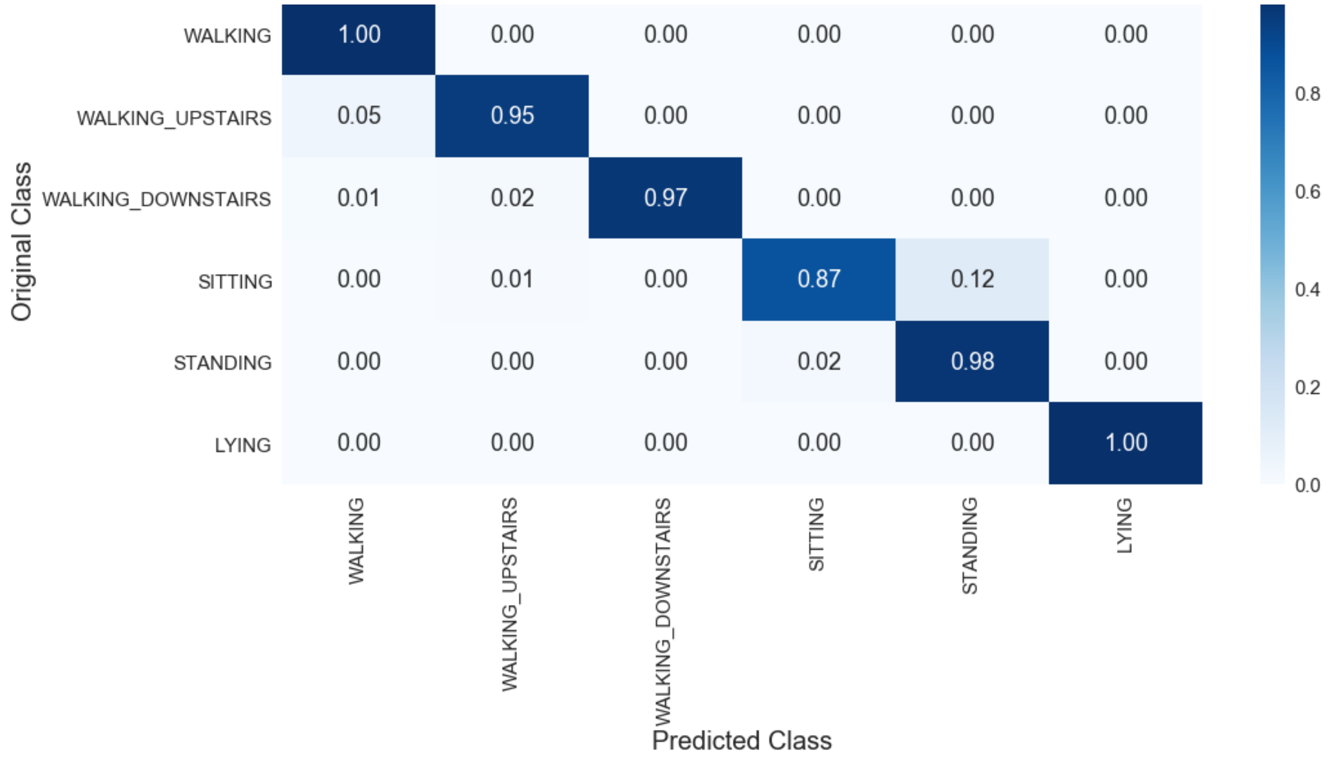
```
{'C': 10, 'penalty': 'l2'}
```




```
----------------------------------------------------------------------------------------------------
----------------------
```




```
----------------------------------------------------------------------------------------------------
----------------------
```

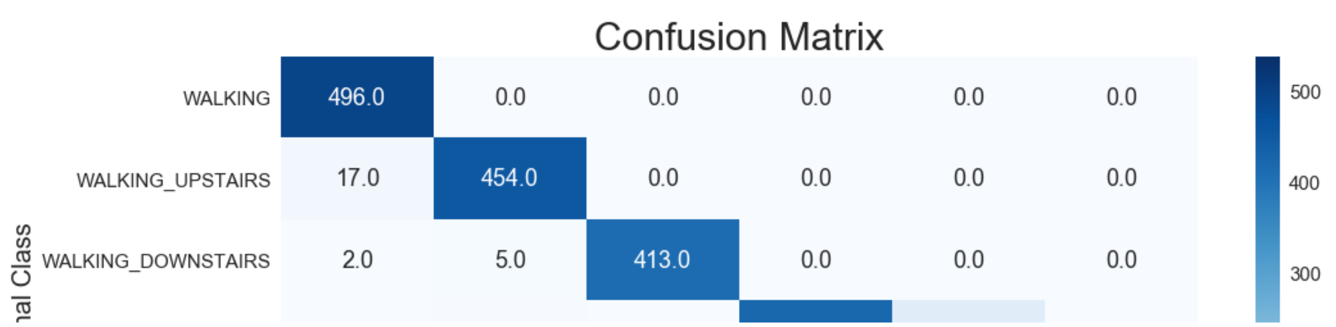


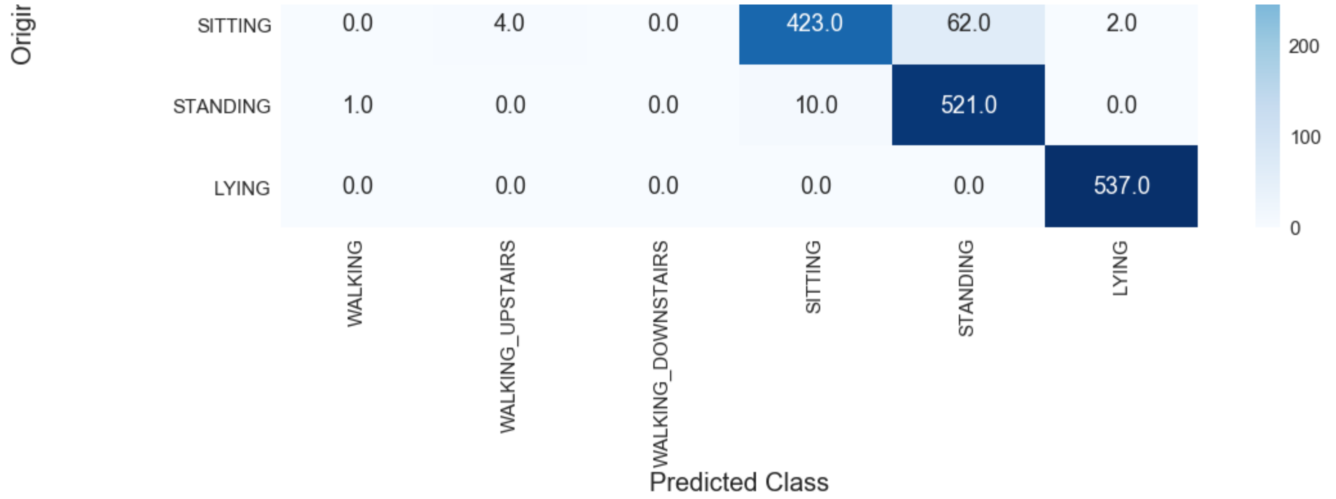
8.2 Linear SVM

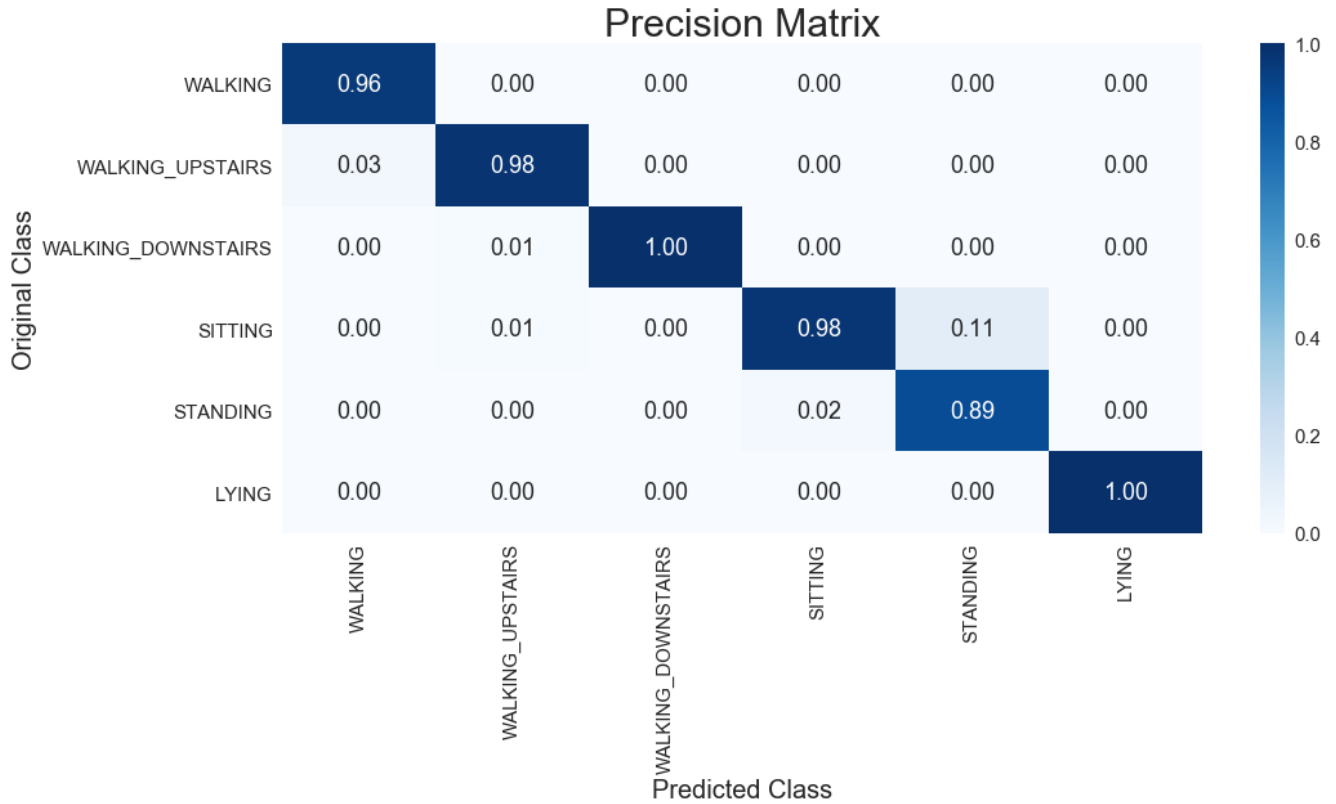
In [168]:

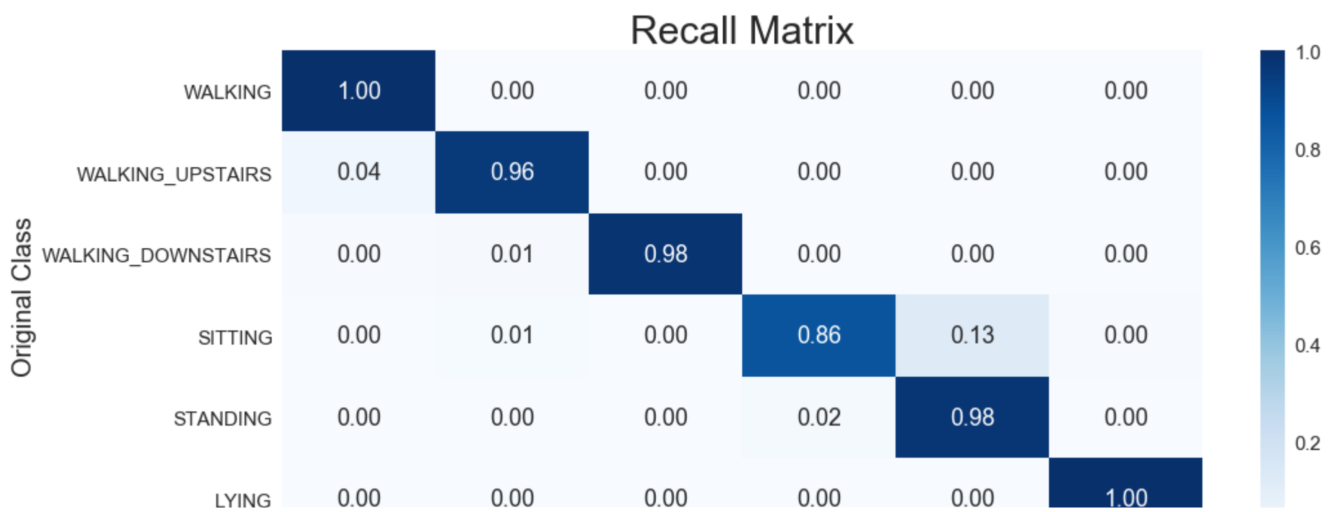
```
parameters = {"C": [0.001, 0.01, 0.1, 1, 10**1, 10**2, 10**3]}
clf = LinearSVC()
cross_val = GridSearchCV(clf, parameters, cv=3)
apply_model(cross_val, x_train, y_train, x_test, y_test, "Linear SVM")
```

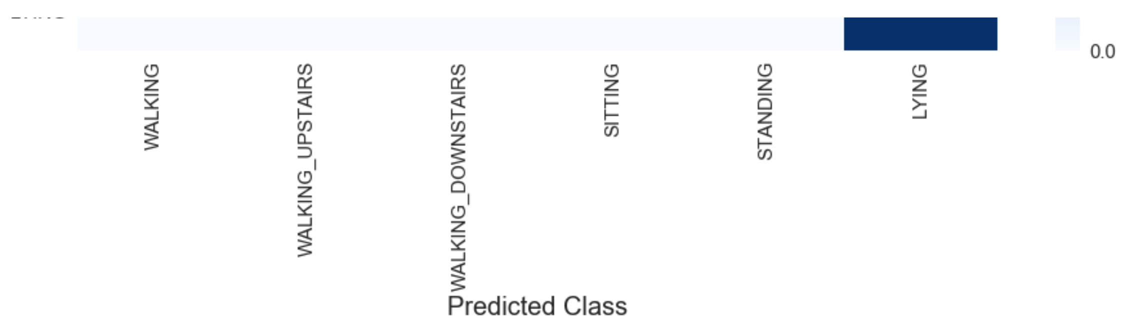
```
Total time taken for tuning hyperparameter and making prediction by the model is (HH:MM:SS): 0:01:
07.034103

---------------------
|      Accuracy      |
---------------------
96.5%

---------------------------
|      Best Estimator      |
---------------------------
LinearSVC(C=1, class_weight=None, dual=True, fit_intercept=True,
     intercept_scaling=1, loss='squared_hinge', max_iter=1000,
     multi_class='ovr', penalty='l2', random_state=None, tol=0.0001,
     verbose=0)

----------------------------------
|      Best Hyper-Parameters      |
----------------------------------
{'C': 1}
```




Origir
SITTING 0.0 4.0 0.0 423.0 62.0 2.0
STANDING 1.0 0.0 0.0 10.0 521.0 0.0
LYING 0.0 0.0 0.0 0.0 0.0 537.0
WALKING
WALKING_UPSTAIRS
WALKING_DOWNSTAIRS
SITTING
STANDING
LYING
Predicted Class
200
100
0


Precision Matrix
Original Class
WALKING 0.96 0.00 0.00 0.00 0.00 0.00
WALKING_UPSTAIRS 0.03 0.98 0.00 0.00 0.00 0.00
WALKING_DOWNSTAIRS 0.00 0.01 1.00 0.00 0.00 0.00
SITTING 0.00 0.01 0.00 0.98 0.11 0.00
STANDING 0.00 0.00 0.00 0.02 0.89 0.00
LYING 0.00 0.00 0.00 0.00 0.00 1.00
WALKING
WALKING_UPSTAIRS
WALKING_DOWNSTAIRS
SITTING
STANDING
LYING
Predicted Class
1.0
0.8
0.6
0.4
0.2
0.0


Recall Matrix
Original Class
WALKING 1.00 0.00 0.00 0.00 0.00 0.00
WALKING_UPSTAIRS 0.04 0.96 0.00 0.00 0.00 0.00
WALKING_DOWNSTAIRS 0.00 0.01 0.98 0.00 0.00 0.00
SITTING 0.00 0.01 0.00 0.86 0.13 0.00
STANDING 0.00 0.00 0.00 0.02 0.98 0.00
LYING 0.00 0.00 0.00 0.00 0.00 1.00
1.0
0.8
0.6
0.4
0.2

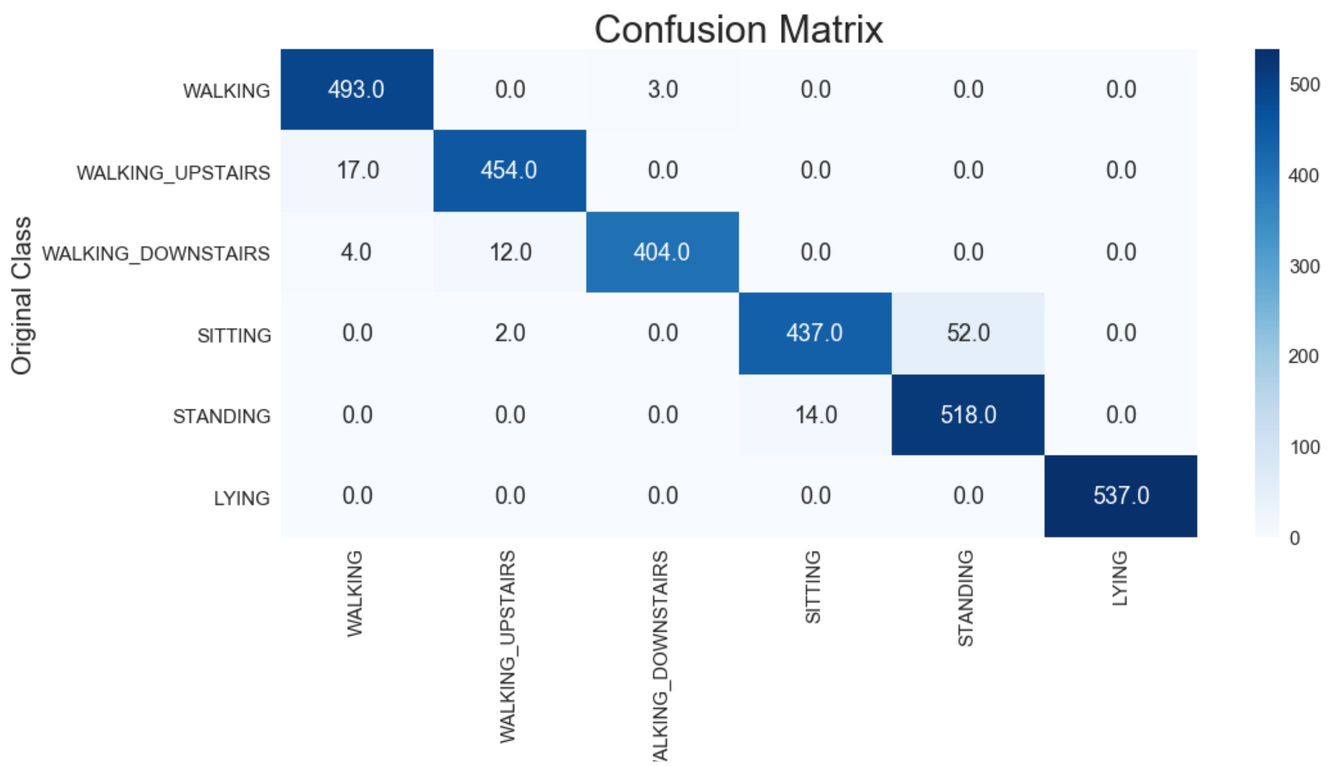


8.3 RBF SVM

In [170]:

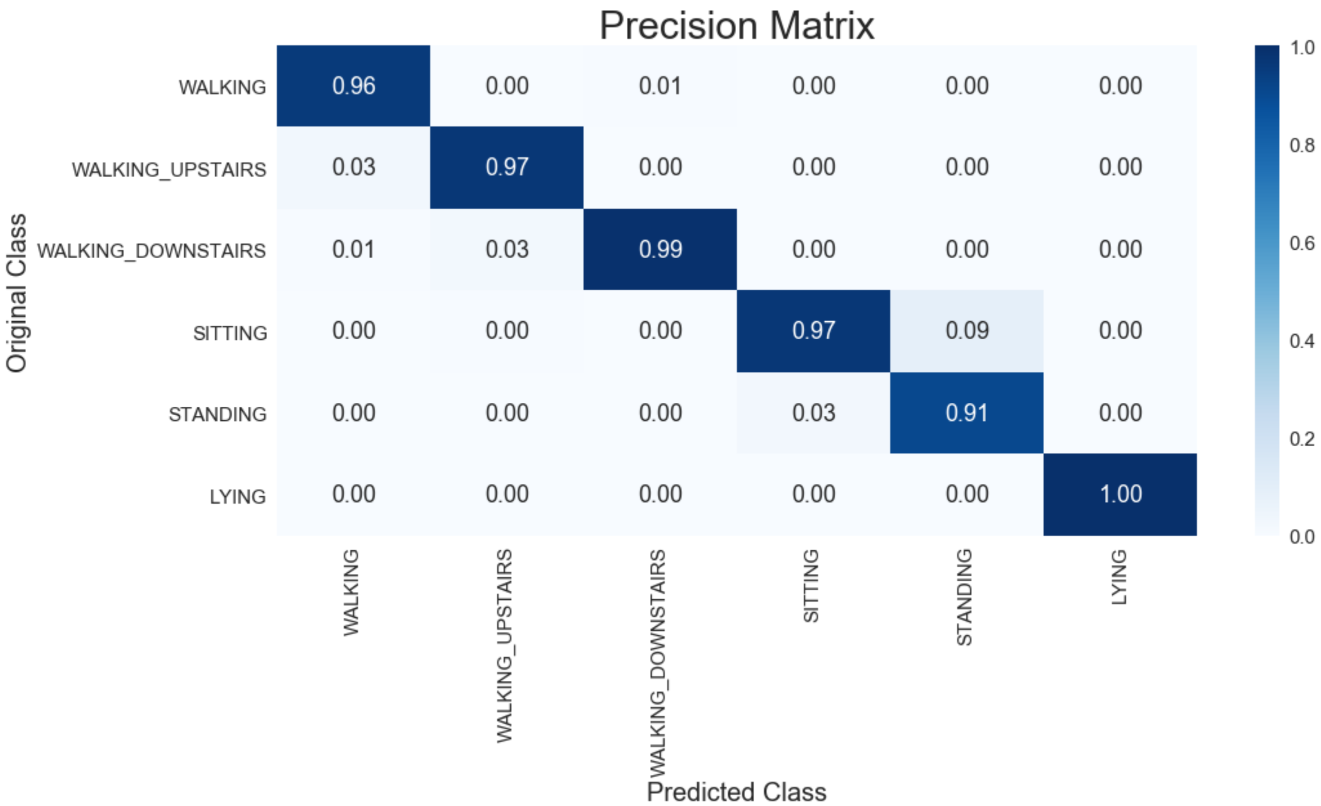
```
parameters = {"C": [0.001, 0.01, 0.1, 1, 10**1, 10**2, 10**3]}
clf = SVC()
cross_val = GridSearchCV(clf, parameters, cv=3)
apply_model(cross_val, x_train, y_train, x_test, y_test, "RBF SVM")
```

```
Total time taken for tuning hyperparameter and making prediction by the model is (HH:MM:SS): 0:08:
52.090489

---------------------
|      Accuracy      |
---------------------
96.47%

---------------------------
|      Best Estimator      |
---------------------------
SVC(C=100, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma='auto', kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)

----------------------------------
|      Best Hyper-Parameters      |
----------------------------------
{'C': 100}
```

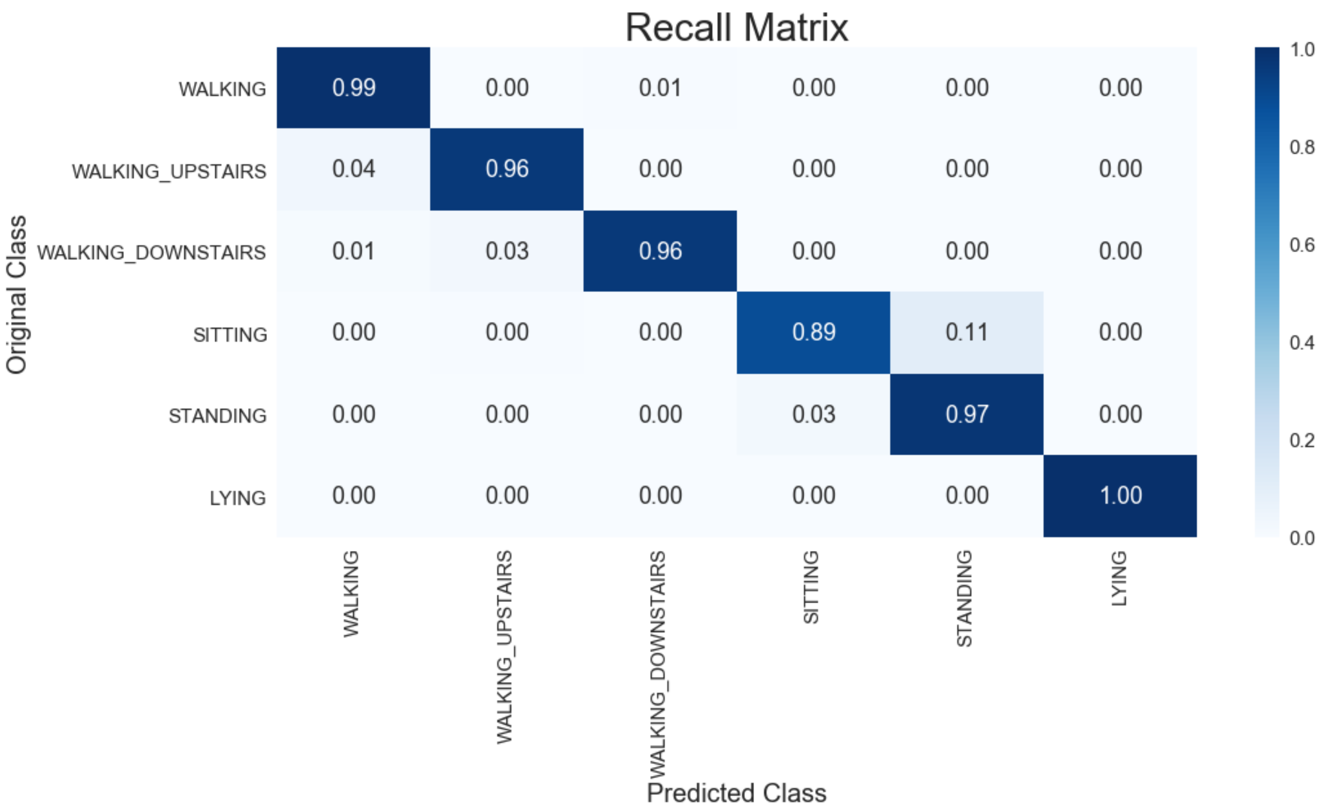


W
Predicted Class

```
-----------------------------------------------------------------------------------------------------------------------
```




```
-----------------------------------------------------------------------------------------------------------------------
```

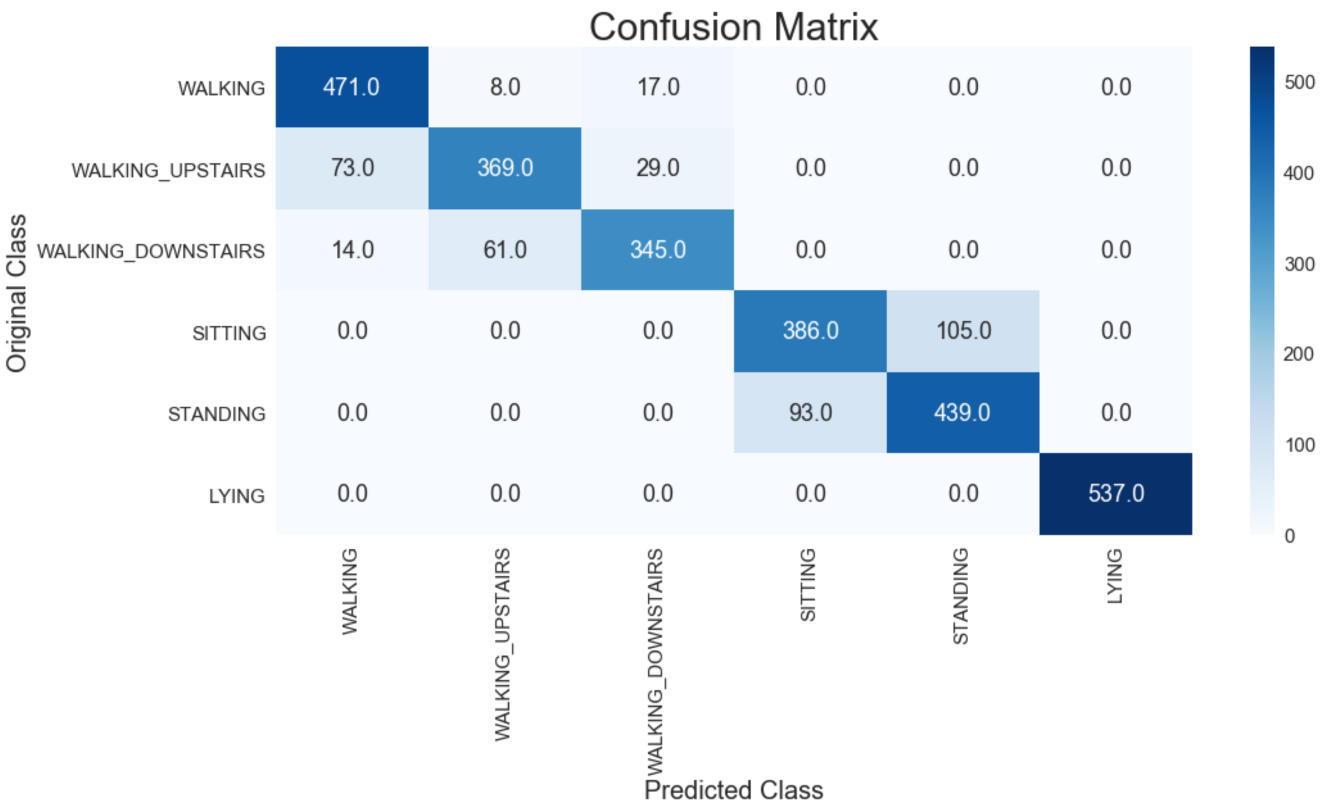


8.4 Decision Trees

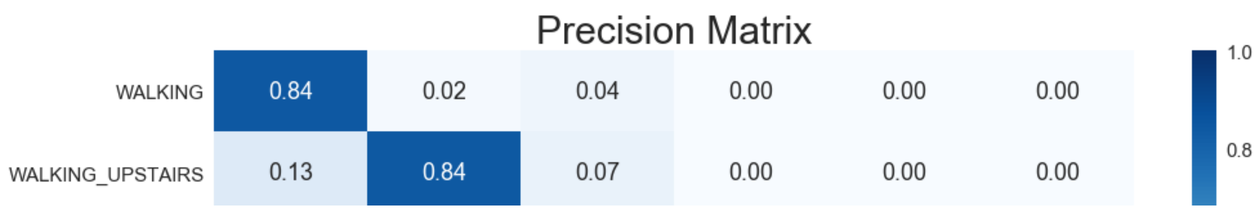
In [171]:

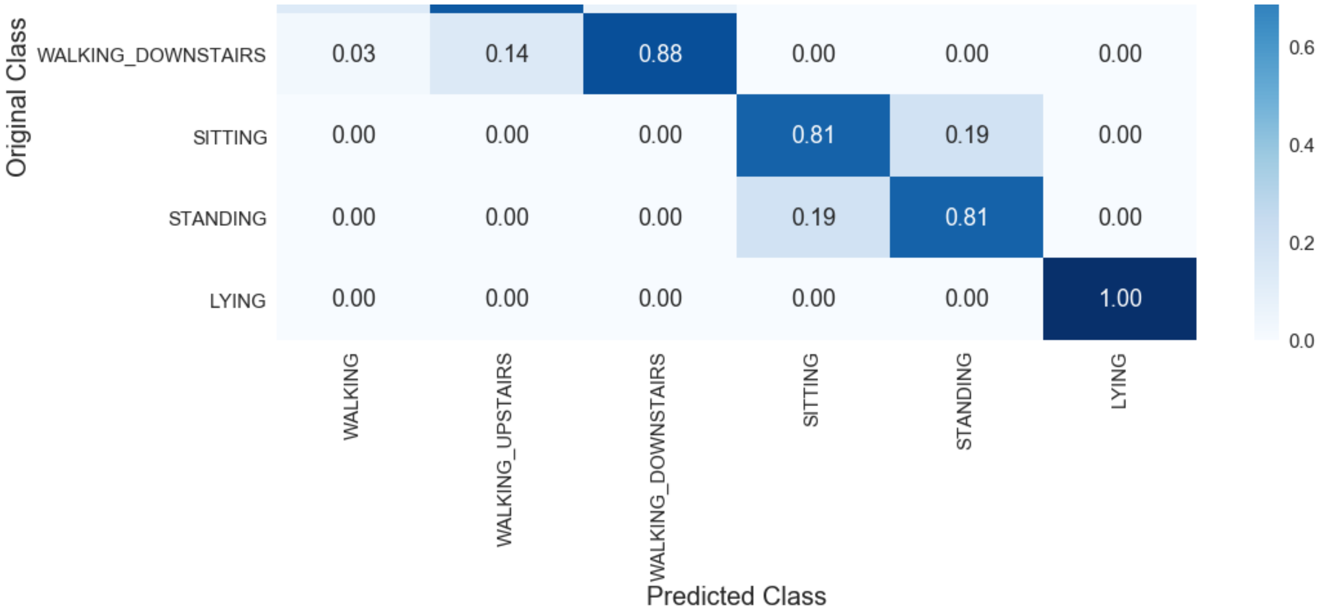
```
parameters = {"max_depth": [2, 3, 4, 5, 6, 7, 8]}
clf = DecisionTreeClassifier()
cross_val = GridSearchCV(clf, parameters, cv=3)
apply_model(cross_val, x_train, y_train, x_test, y_test, "Decision Trees")
```

```
Total time taken for tuning hyperparameter and making prediction by the model is (HH:MM:SS): 0:00:
36.169150

---------------------
|      Accuracy      |
---------------------
86.43%

---------------------------
|      Best Estimator      |
---------------------------
DecisionTreeClassifier(class_weight=None, criterion='gini', max_depth=7,
            max_features=None, max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, presort=False, random_state=None,
            splitter='best')

----------------------------------
|      Best Hyper-Parameters      |
----------------------------------
{'max_depth': 7}
```

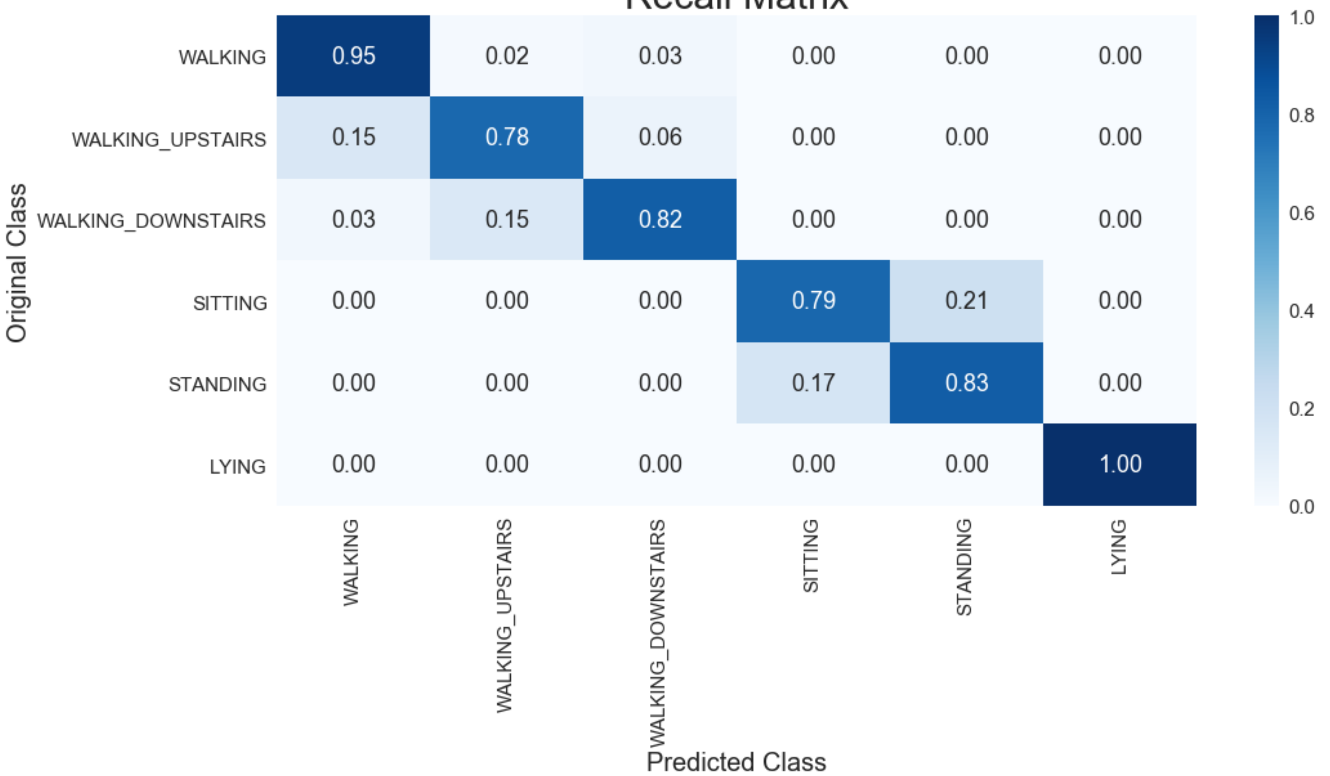



```
--------------------------------------------------------------------------------------------------------
----------------------
```






```
-----------------------------------------------------------------------------------------------------
----------------------
```



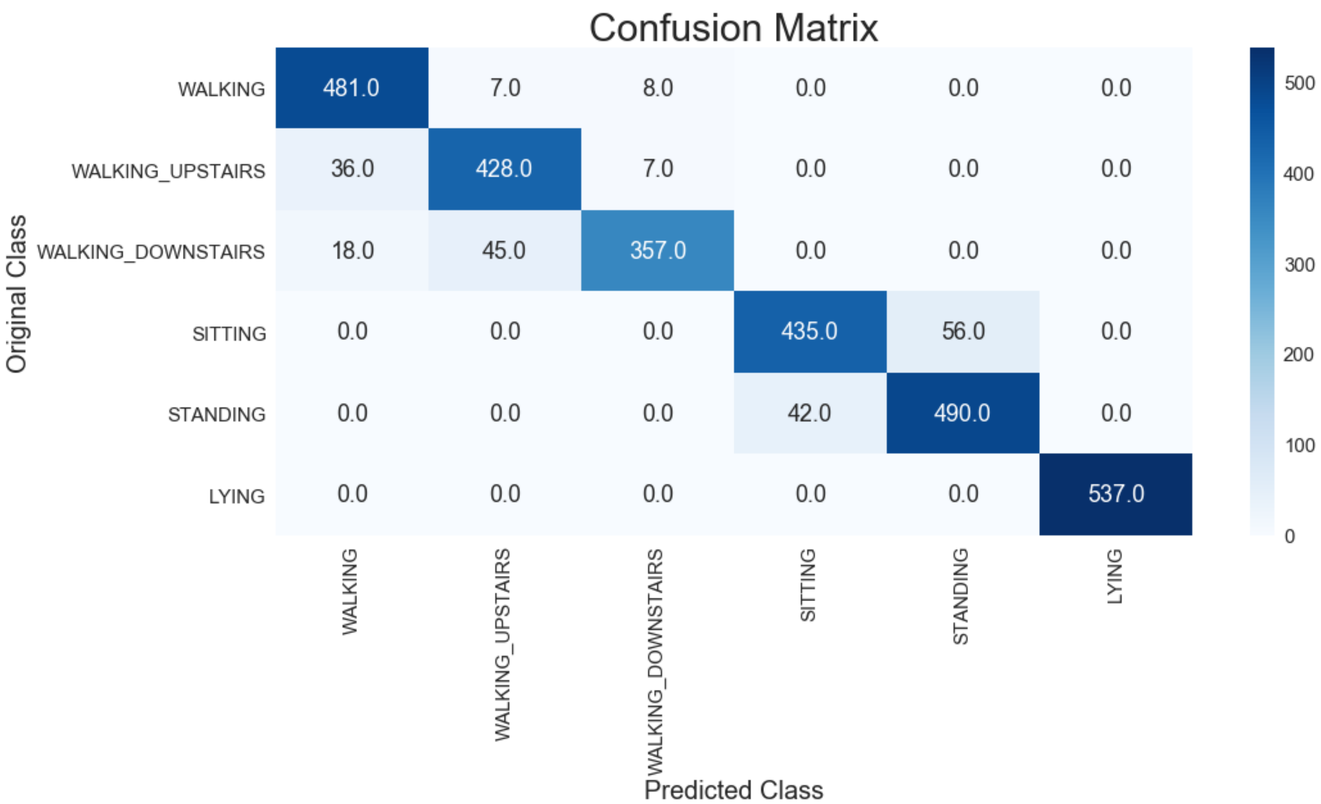
8.5 Random Forest

In [172]:

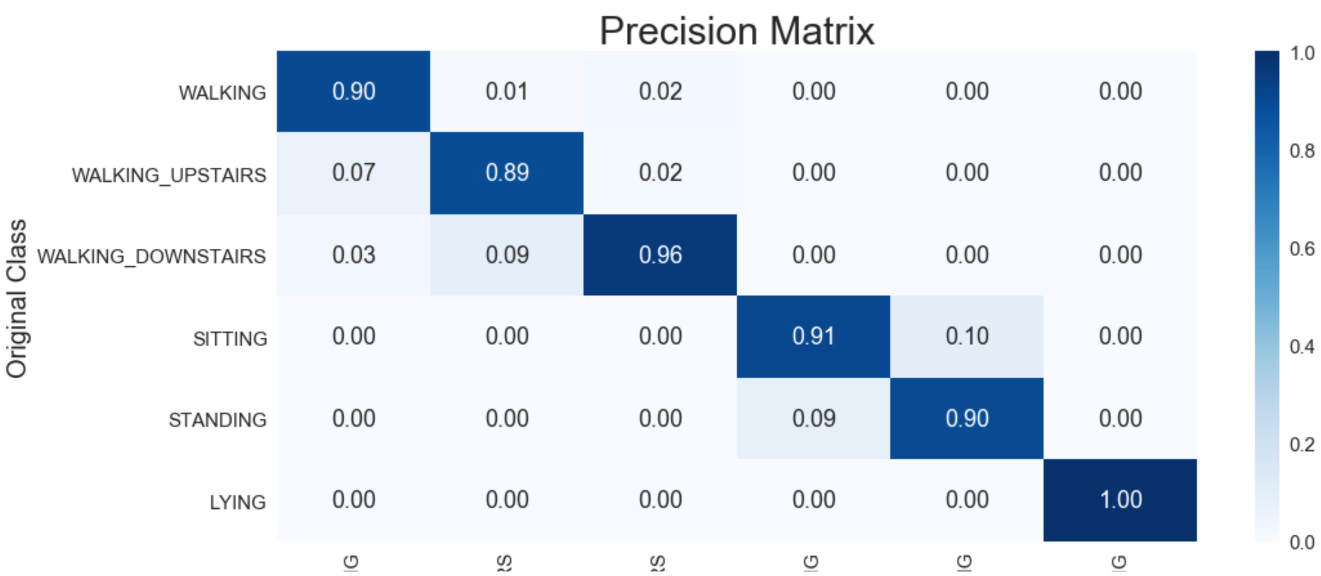
```
parameters = {"n_estimators": [50, 100, 200, 400, 800]}
clf = RandomForestClassifier()
cross_val = GridSearchCV(clf, parameters, cv=3)
apply_model(cross_val, x_train, y_train, x_test, y_test, "Random Forest")
```

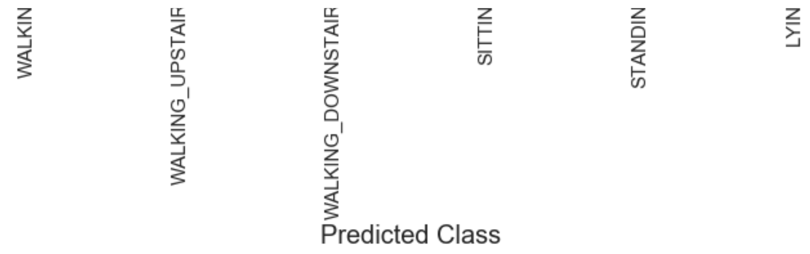
```
Total time taken for tuning hyperparameter and making prediction by the model is (HH:MM:SS): 0:06:
41.823309

---------------------
|      Accuracy      |
---------------------
```

```
92.57%

---------------------------
|      Best Estimator     |
---------------------------
RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=200, n_jobs=1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)

----------------------------------
|      Best Hyper-Parameters      |
----------------------------------
{'n_estimators': 200}
```

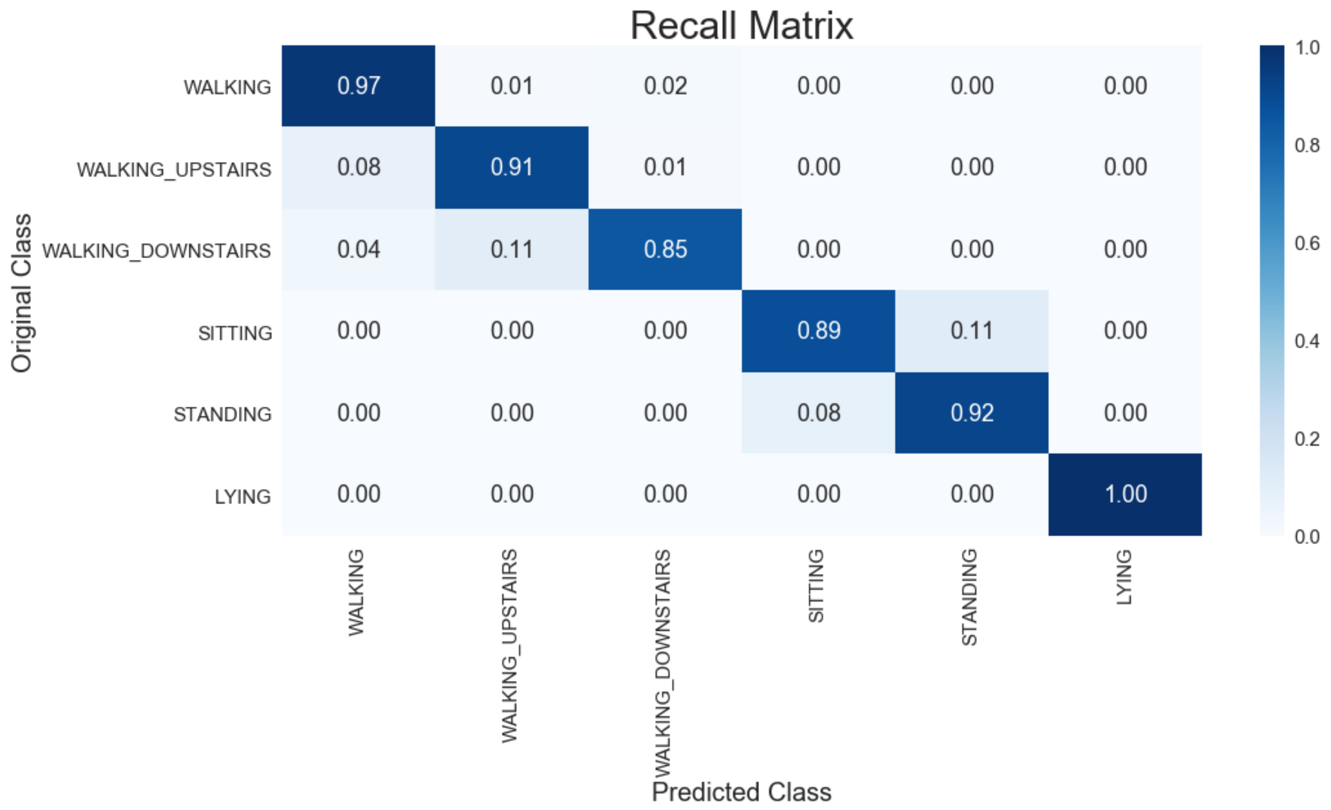



```
--------------------------------------------------------------------------------------------------
----------------------
```






```
--------------------------------------------------------------------------------------------------------------
```



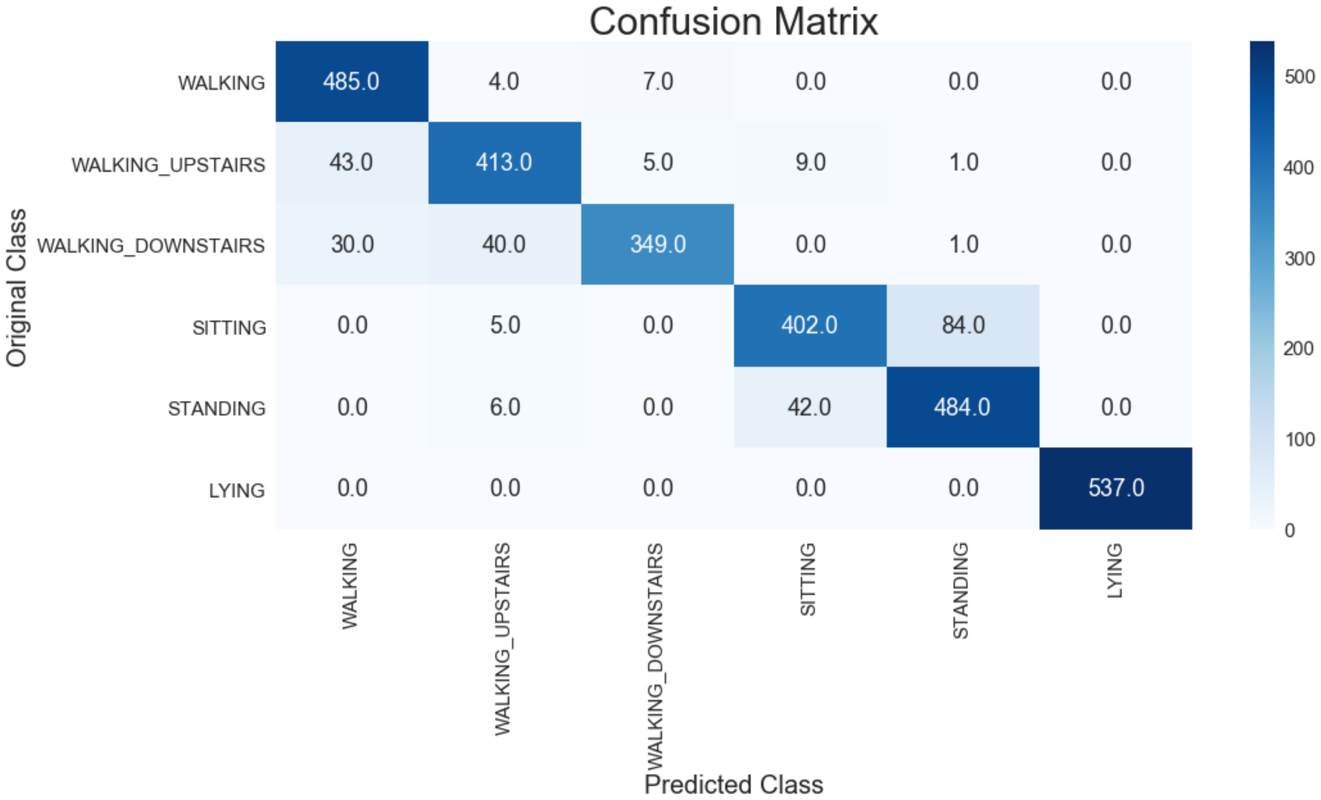
8.6 Gradient Boosted Decision Trees

In [175]:

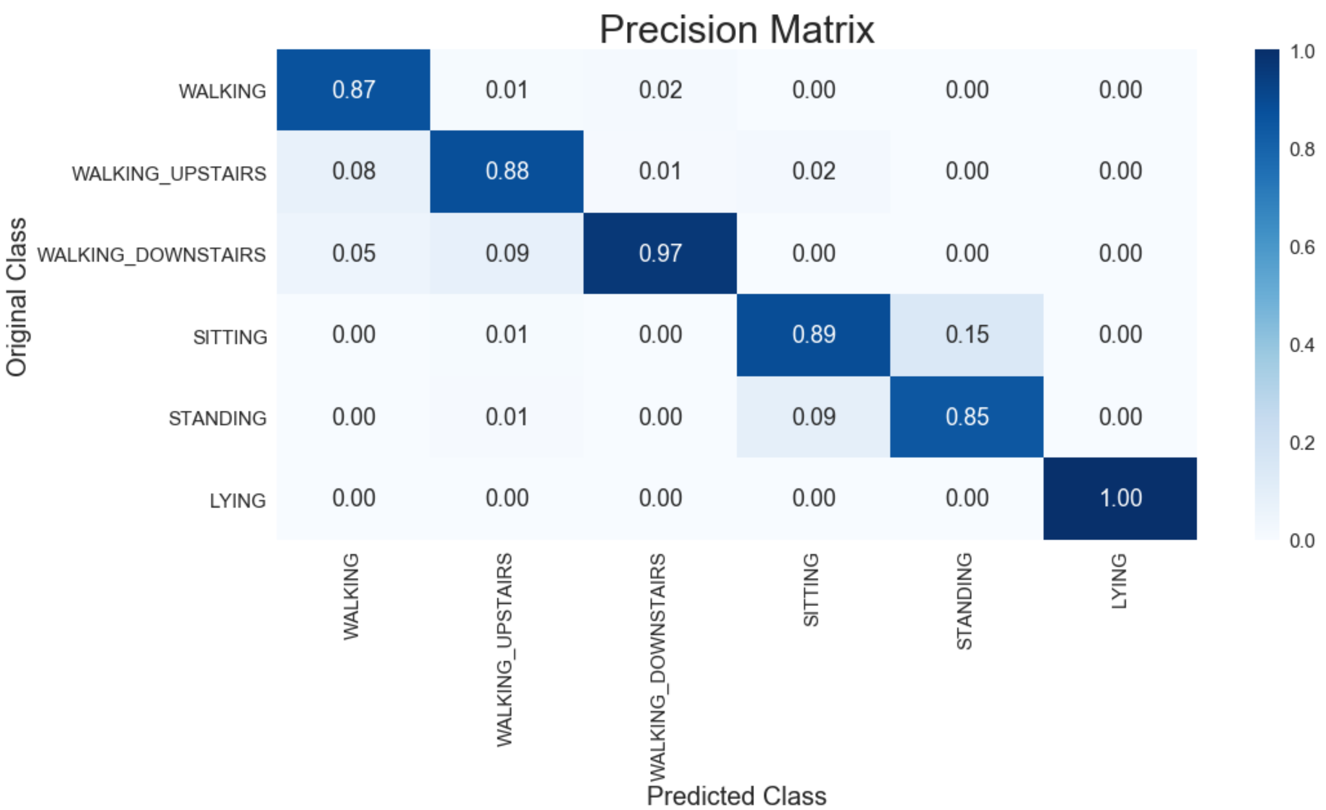
```
parameters = {"n_estimators": [50, 100], "max_depth":[1, 3]}
clf = GradientBoostingClassifier()
cross_val = GridSearchCV(clf, parameters, cv=3)
apply_model(cross_val, x_train, y_train, x_test, y_test, "Gradient Boosted DT")
```

```
Total time taken for tuning hyperparameter and making prediction by the model is (HH:MM:SS): 0:15:
06.368304

---------------------
|      Accuracy      |
---------------------
90.6%

---------------------------
|      Best Estimator      |
---------------------------
GradientBoostingClassifier(criterion='friedman_mse', init=None,
              learning_rate=0.1, loss='deviance', max_depth=1,
              max_features=None, max_leaf_nodes=None,
              min_impurity_decrease=0.0, min_impurity_split=None,
              min_samples_leaf=1, min_samples_split=2,
              min_weight_fraction_leaf=0.0, n_estimators=100,
              presort='auto', random_state=None, subsample=1.0, verbose=0,
              warm_start=False)
```

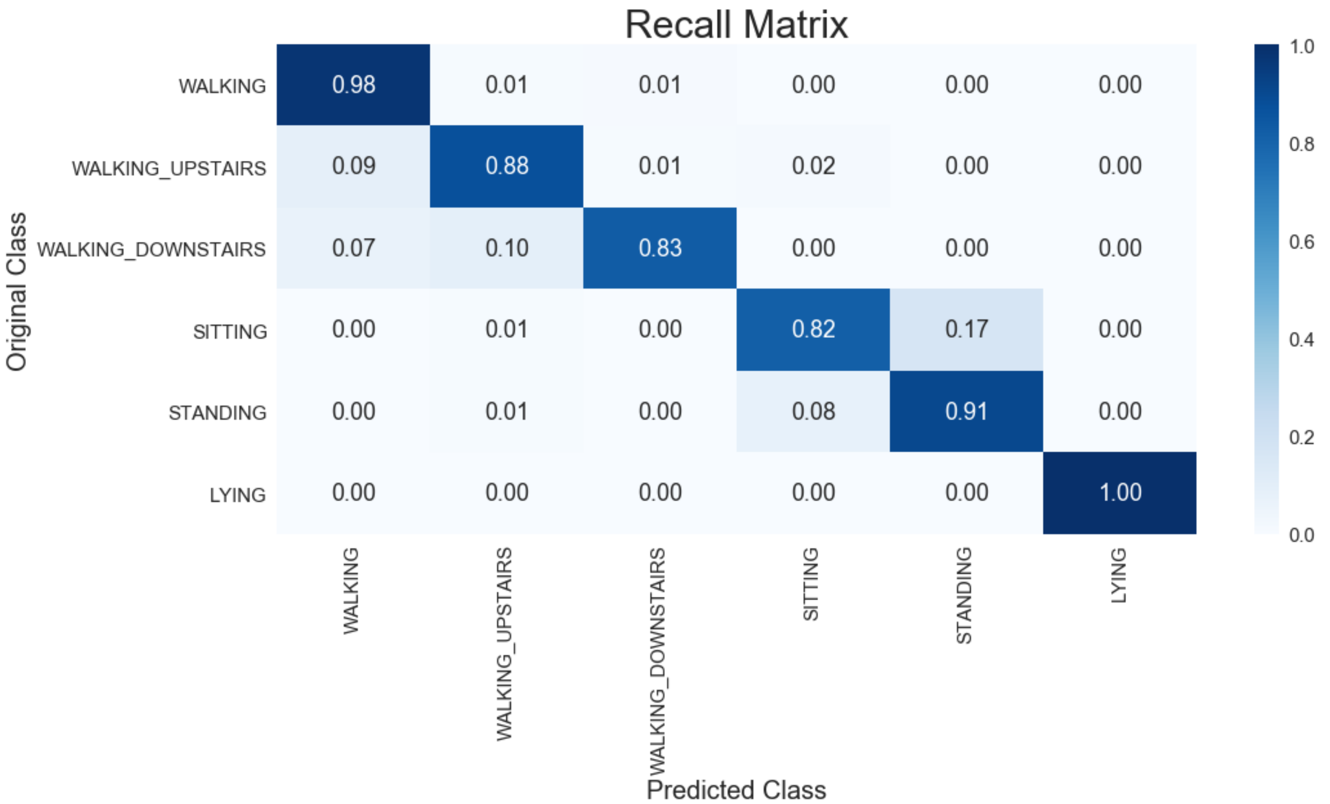
```
------------------------------------
|      Best Hyper-Parameters       |
------------------------------------
{'max_depth': 1, 'n_estimators': 100}
```




```
--------------------------------------------------------------------------------------------------------------------
--------------------
```



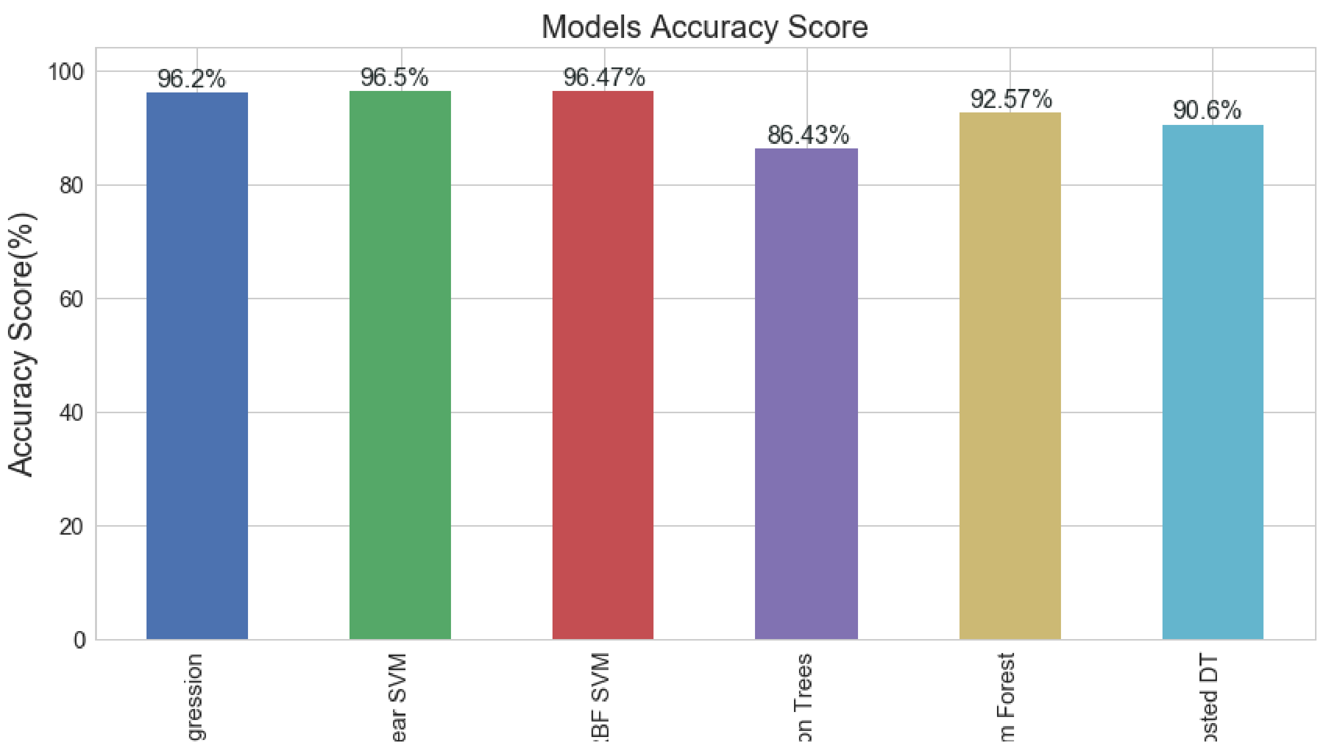

```
--------------------------------------------------------------------------------------------------------------------
--------------------
```



9. Model Comparison

In [230]:

```
ax = table.plot(x = "Model", y = "Accuracy(%)", 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, co
lor = "#232b2b")
```



In [192]:

```
table
```

Out[192]:

	Model	Accuracy(%)
0	Logistic Regression	96.20
1	Linear SVM	96.50
2	RBF SVM	96.47
3	Decision Trees	86.43
4	Random Forest	92.57
5	Gradient Boosted DT	90.60

Comments

- Models: Logistic Regression, rbf SVM and Linear SVM give accuracy above 96%.
- In real world, having domain knowledge is one of the most important aspects of machine learning Modelling. Here, we got pretty good accuracy of above 96%. This is very much due to the fact that features are very well engineered by domain experts in signal processing.
- In a nutshell, feature engineering is one of the most important aspect of machine learning.

10. Applying Deep Learning Model: LSTM

Here in LSTM, we will use 128 sized raw readings that we obtained from accelerometer and gyroscope signals.

10.1 Reading Data

In [2]:

```
all_signals_list = ["body_acc_x_", "body_acc_y_", "body_acc_z_", "body_gyro_x_", "body_gyro_y_", "b
ody_gyro_z_",
                   "total_acc_x_", "total_acc_y_", "total_acc_z_"]
```

In [3]:

```
def reading_data(filename):
    return pd.read_csv(filename, delim_whitespace = True, header = None)
```

In [4]:

```
def total_signal_matrix(trainOrTest):
    complete_data = []
    for signal in all_signals_list:
        complete_data.append(reading_data("../Data/"+ trainOrTest +"/Inertial Signals/"+ signal + t
rainOrTest +".txt").as_matrix())
    return np.transpose(complete_data, (1, 2, 0))
```

In [6]:

```
def load_labels(subset):
    filename = "../Data/"+subset+"/y_"+subset+".txt"
    y = reading_data(filename)
    return pd.get_dummies(y[0]).as_matrix()
# here, get_dummies takes pandas series as input and returns its one-hot encoded vector of each el
ement in a series.
```

In [7]:

```
def load_full_data():
    x_train = total_signal_matrix("train")
    y_train = load_labels("train")
    x_test = total_signal_matrix("test")
    y_test = load_labels("test")
    return x_train, y_train, x_test, y_test
```

In [11]:

```
x_train, y_train, x_test, y_test = load_full_data()
x_train.shape, y_train.shape, x_test.shape, y_test.shape
```

Out[11]:

```
((7352, 128, 9), (7352, 6), (2947, 128, 9), (2947, 6))
```

In [9]:

```
#saving data for loading it later in hyperas for hyper-parameter tuning
np.save("../Data/train", x_train)
np.save("../Data/train_label", y_train)
np.save("../Data/test", x_test)
np.save("../Data/test_label", y_test)
```

In [3]:

```
def data():
    x_train = np.load("../Data/train.npy")
    y_train = np.load("../Data/train_label.npy")
    x_test = np.load("../Data/test.npy")
    y_test = np.load("../Data/test_label.npy")
    return x_train, y_train, x_test, y_test
```

In [10]:

```
#this function will return number of classes
def count_unique_classes(y_train):
    return len(set([tuple(a) for a in y_train]))
```

10.2 Hyper-Parameter Tuning with Hyperas and Applying LSTM with best Hyper-Parameters

In [ ]:

```
# Refer documentation of hyperas here: https://github.com/maxpumperla/hyperas
```

In [4]:

```
def create_model(x_train, y_train, x_test, y_test):

    epochs = 8
    batch_size = 32
    timesteps = x_train.shape[1]
    input_dim = len(x_train[0][0])
    n_classes = 6

    model = Sequential()

    model.add(LSTM(64, return_sequences = True, input_shape = (timesteps, input_dim)))
    model.add(Dropout({{uniform(0, 1)}}))

    model.add(LSTM({{choice([32, 16])}}))
    model.add(Dropout({{uniform(0, 1)}}))

    model.add(Dense(n_classes, activation='sigmoid'))
```

```
    print(model.summary())

    model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='rmsprop')

    result = model.fit(x_train, y_train, batch_size = batch_size, epochs=epochs, verbose=2,
validation_split=0.01)

    validation_acc = np.amax(result.history['val_acc'])

    print('Best validation acc of epoch:', validation_acc)

    return {'loss': -validation_acc, 'status': STATUS_OK, 'model': model}
```

In [5]:

```
best_run, best_model = optim.minimize(model=create_model, data=data, algo=tpe.suggest, max_evals=4,
trials=Trials(), notebook_name = "HumanActivityRecognition")
x_train, y_train, x_test, y_test = data()

score = best_model.evaluate(x_test, y_test)

print('---------------------')
print('|      Accuracy      |')
print('---------------------')
acc = np.round((score[1]*100), 2)
print(str(acc)+"%\n")

print('----------------------------------')
print('|      Best Hyper-Parameters      |')
print('----------------------------------')
print(best_run)
print("\n\n")

true_labels = [np.argmax(i)+1 for i in y_test]
predicted_probs = best_model.predict(x_test)
predicted_labels = [np.argmax(i)+1 for i in predicted_probs]
print_confusionMatrix(true_labels, predicted_labels)
```

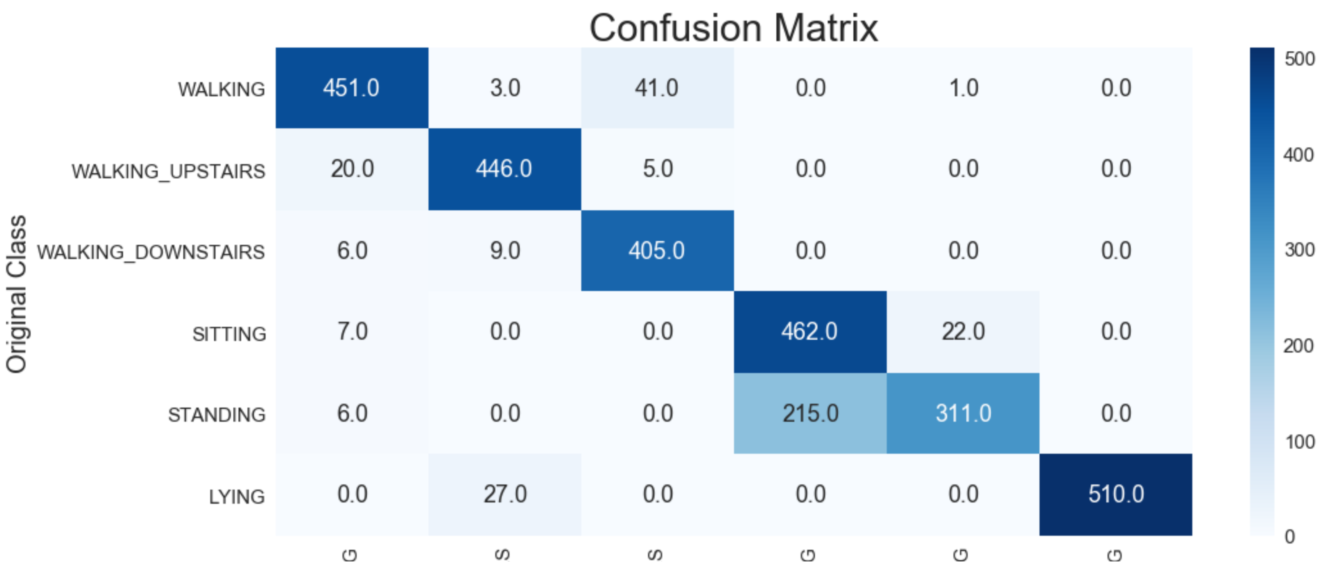
```
>>> Imports:
#coding=utf-8

try:
    import numpy as np
except:
    pass

try:
    import pandas as pd
except:
    pass

try:
    import seaborn as sns
except:
    pass

try:
    import matplotlib.pyplot as plt
except:
    pass

try:
    from sklearn.manifold import TSNE
except:
    pass

try:
    import warnings
except:
    pass

try:
    from datetime import datetime
except:
```

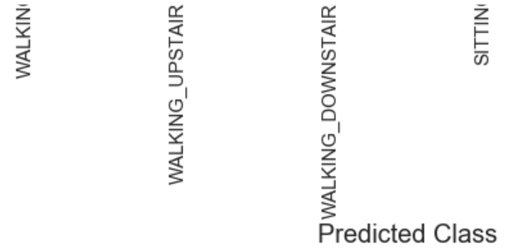
```
    pass

try:
    from sklearn.model_selection import GridSearchCV
except:
    pass

try:
    from sklearn.metrics import confusion_matrix
except:
    pass

try:
    from sklearn.metrics import accuracy_score
except:
    pass

try:
    from sklearn.linear_model import LogisticRegression
except:
    pass

try:
    from sklearn.svm import LinearSVC
except:
    pass

try:
    from sklearn.svm import SVC
except:
    pass

try:
    from sklearn.tree import DecisionTreeClassifier
except:
    pass

try:
    from sklearn.ensemble import RandomForestClassifier
except:
    pass

try:
    from sklearn.ensemble import GradientBoostingClassifier
except:
    pass

try:
    from keras.models import Sequential
except:
    pass

try:
    from keras.layers import LSTM
except:
    pass

try:
    from keras.layers.core import Dense, Dropout
except:
    pass

try:
    from hyperopt import Trials, STATUS_OK, tpe
except:
    pass

try:
    from hyperas import optim
except:
    pass

try:
    from hyperas.distributions import choice, uniform
except:
    pass
```

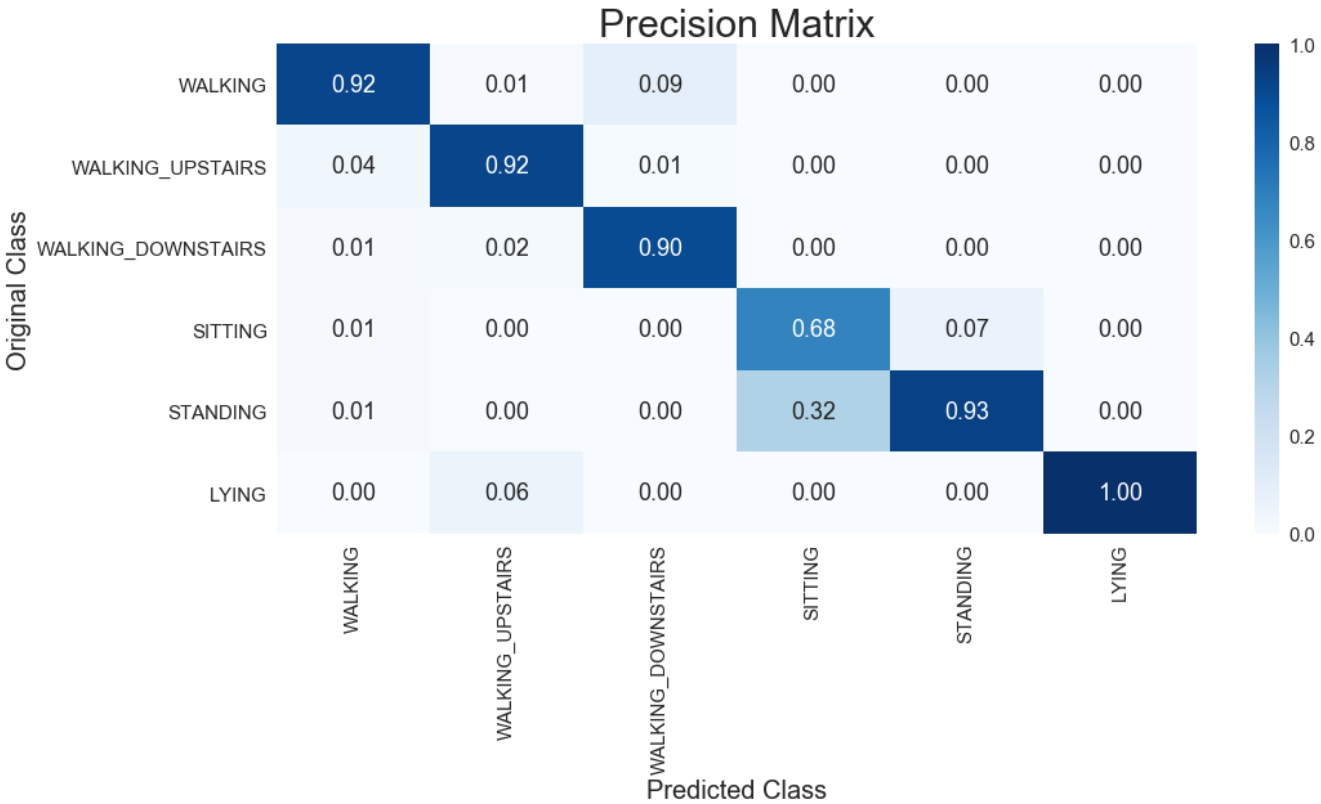
```
>>> Hyperas search space:

def get_space():
    return {
        'Dropout': hp.uniform('Dropout', 0, 1),
        'LSTM': hp.choice('LSTM', [32, 16]),
        'Dropout_1': hp.uniform('Dropout_1', 0, 1),
    }

>>> Data
  1:
  2: x_train = np.load("../Data/train.npy")
  3: y_train = np.load("../Data/train_label.npy")
  4: x_test = np.load("../Data/test.npy")
  5: y_test = np.load("../Data/test_label.npy")
  6:
  7:
  8:
>>> Resulting replaced keras model:

  1: def keras_fmin_fnct(space):
  2:
  3:
  4:     epochs = 8
  5:     batch_size = 32
  6:     timesteps = x_train.shape[1]
  7:     input_dim = len(x_train[0][0])
  8:     n_classes = 6
  9:
 10:     model = Sequential()
 11:
 12:     model.add(LSTM(64, return_sequences = True, input_shape = (timesteps, input_dim)))
 13:     model.add(Dropout(space['Dropout']))
 14:
 15:     model.add(LSTM(space['LSTM']))
 16:     model.add(Dropout(space['Dropout_1']))
 17:
 18:     model.add(Dense(n_classes, activation='sigmoid'))
 19:
 20:     print(model.summary())
 21:
 22:     model.compile(loss='categorical_crossentropy', metrics=['accuracy'], optimizer='rmsprop')
 23:
 24:     result = model.fit(x_train, y_train, batch_size = batch_size, epochs=epochs, verbose=2, v
alidation_split=0.01)
 25:
 26:     validation_acc = np.amax(result.history['val_acc'])
 27:
 28:     print('Best validation acc of epoch:', validation_acc)
 29:
 30:     return {'loss': -validation_acc, 'status': STATUS_OK, 'model': model}
 31:
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_1 (LSTM)                (None, 128, 64)           18944
_________________________________________________________________
dropout_1 (Dropout)          (None, 128, 64)           0
_________________________________________________________________
lstm_2 (LSTM)                (None, 32)                12416
_________________________________________________________________
dropout_2 (Dropout)          (None, 32)                0
_________________________________________________________________
dense_1 (Dense)              (None, 6)                 198
=================================================================
Total params: 31,558
Trainable params: 31,558
Non-trainable params: 0
_________________________________________________________________
None
Train on 7278 samples, validate on 74 samples
Epoch 1/8
 - 34s - loss: 1.2287 - acc: 0.5000 - val_loss: 1.3144 - val_acc: 0.6081
Epoch 2/8
 - 33s - loss: 0.8247 - acc: 0.6652 - val_loss: 1.0918 - val_acc: 0.3919
Epoch 3/8
 - 33s - loss: 0.6093 - acc: 0.7725 - val_loss: 0.6550 - val_acc: 0.7838
```

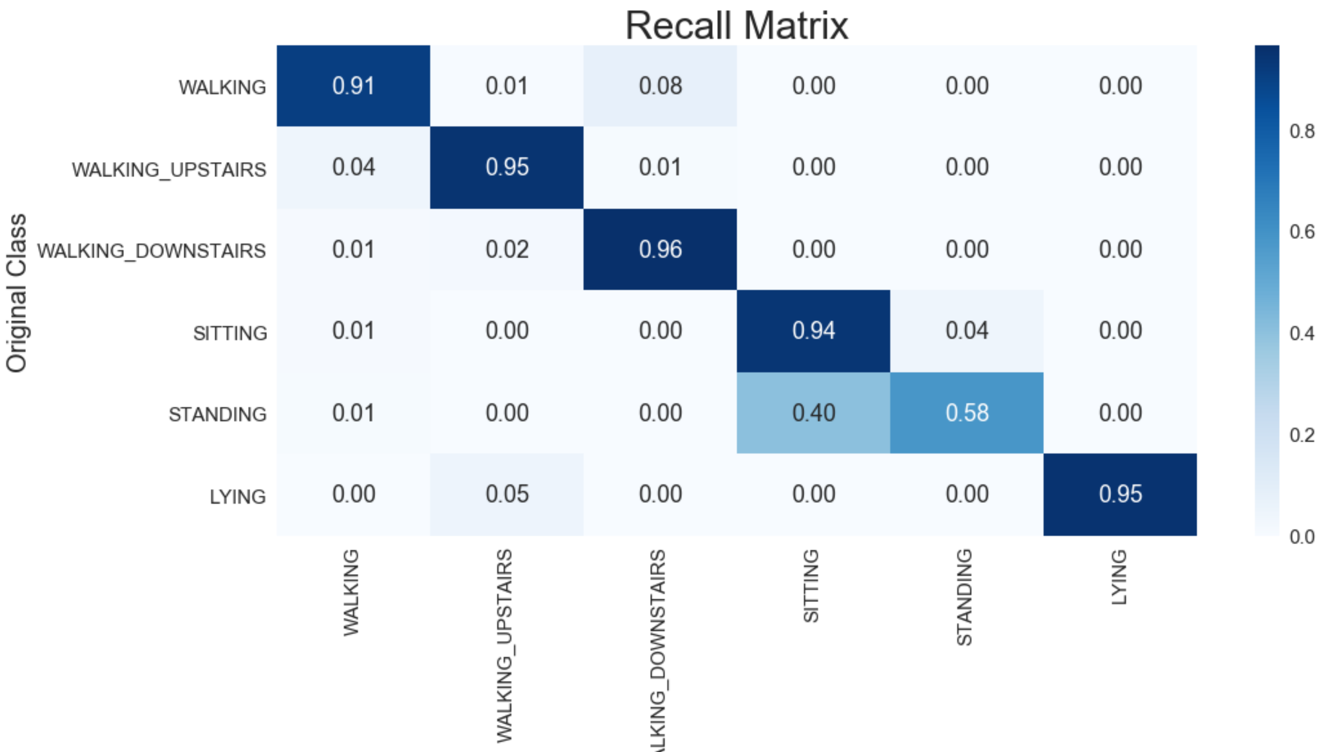
```
Epoch 4/8
 - 32s - loss: 0.5195 - acc: 0.8248 - val_loss: 0.2623 - val_acc: 0.9595
Epoch 5/8
 - 32s - loss: 0.3589 - acc: 0.8897 - val_loss: 0.1033 - val_acc: 0.9865
Epoch 6/8
 - 32s - loss: 0.2909 - acc: 0.9122 - val_loss: 0.0374 - val_acc: 1.0000
Epoch 7/8
 - 33s - loss: 0.2232 - acc: 0.9279 - val_loss: 0.0189 - val_acc: 1.0000
Epoch 8/8
 - 32s - loss: 0.1957 - acc: 0.9321 - val_loss: 0.0107 - val_acc: 1.0000
Best validation acc of epoch: 1.0
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
lstm_3 (LSTM)                (None, 128, 64)           18944     
_________________________________________________________________
dropout_3 (Dropout)          (None, 128, 64)           0         
_________________________________________________________________
lstm_4 (LSTM)                (None, 32)                12416     
_________________________________________________________________
dropout_4 (Dropout)          (None, 32)                0         
_________________________________________________________________
dense_2 (Dense)              (None, 6)                 198       
=================================================================
Total params: 31,558
Trainable params: 31,558
Non-trainable params: 0
_________________________________________________________________
None
Train on 7278 samples, validate on 74 samples
Epoch 1/8
 - 33s - loss: 1.1814 - acc: 0.5302 - val_loss: 1.2119 - val_acc: 0.3919
Epoch 2/8
 - 32s - loss: 0.8180 - acc: 0.6412 - val_loss: 1.0903 - val_acc: 0.4595
Epoch 3/8
 - 32s - loss: 0.6648 - acc: 0.7259 - val_loss: 0.9529 - val_acc: 0.5946
Epoch 4/8
 - 32s - loss: 0.5555 - acc: 0.7815 - val_loss: 0.6713 - val_acc: 0.6486
Epoch 5/8
 - 32s - loss: 0.4661 - acc: 0.8013 - val_loss: 0.5429 - val_acc: 0.7973
Epoch 6/8
 - 32s - loss: 0.3942 - acc: 0.8630 - val_loss: 0.1906 - val_acc: 0.9865
Epoch 7/8
 - 32s - loss: 0.3038 - acc: 0.9092 - val_loss: 0.0446 - val_acc: 1.0000
Epoch 8/8
 - 32s - loss: 0.2122 - acc: 0.9332 - val_loss: 0.0165 - val_acc: 1.0000
Best validation acc of epoch: 1.0
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
lstm_5 (LSTM)                (None, 128, 64)           18944     
_________________________________________________________________
dropout_5 (Dropout)          (None, 128, 64)           0         
_________________________________________________________________
lstm_6 (LSTM)                (None, 16)                5184      
_________________________________________________________________
dropout_6 (Dropout)          (None, 16)                0         
_________________________________________________________________
dense_3 (Dense)              (None, 6)                 102       
=================================================================
Total params: 24,230
Trainable params: 24,230
Non-trainable params: 0
_________________________________________________________________
None
Train on 7278 samples, validate on 74 samples
Epoch 1/8
 - 33s - loss: 1.4913 - acc: 0.3894 - val_loss: 1.3277 - val_acc: 0.3108
Epoch 2/8
 - 31s - loss: 1.2572 - acc: 0.4783 - val_loss: 1.2003 - val_acc: 0.2432
Epoch 3/8
 - 31s - loss: 1.1256 - acc: 0.5187 - val_loss: 1.1413 - val_acc: 0.2162
Epoch 4/8
 - 31s - loss: 1.0899 - acc: 0.5185 - val_loss: 1.1427 - val_acc: 0.2162
Epoch 5/8
 - 31s - loss: 1.0622 - acc: 0.5235 - val_loss: 1.1488 - val_acc: 0.2162
Epoch 6/8
```

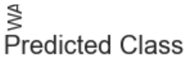
```
 - 31s - loss: 0.9769 - acc: 0.5596 - val_loss: 1.1349 - val_acc: 0.2162
Epoch 7/8
 - 31s - loss: 0.9549 - acc: 0.5595 - val_loss: 1.1195 - val_acc: 0.2162
Epoch 8/8
 - 31s - loss: 0.9556 - acc: 0.5595 - val_loss: 1.1188 - val_acc: 0.2162
Best validation acc of epoch: 0.3108108124217471
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
lstm_7 (LSTM)                (None, 128, 64)           18944
_________________________________________________________________
dropout_7 (Dropout)          (None, 128, 64)           0
_________________________________________________________________
lstm_8 (LSTM)                (None, 16)                5184
_________________________________________________________________
dropout_8 (Dropout)          (None, 16)                0
_________________________________________________________________
dense_4 (Dense)              (None, 6)                 102
=================================================================
Total params: 24,230
Trainable params: 24,230
Non-trainable params: 0
_________________________________________________________________
None
Train on 7278 samples, validate on 74 samples
Epoch 1/8
 - 33s - loss: 1.6284 - acc: 0.2837 - val_loss: 1.6123 - val_acc: 0.1081
Epoch 2/8
 - 31s - loss: 1.5063 - acc: 0.3362 - val_loss: 1.4299 - val_acc: 0.3243
Epoch 3/8
 - 31s - loss: 1.4426 - acc: 0.3530 - val_loss: 1.3619 - val_acc: 0.2162
Epoch 4/8
 - 31s - loss: 1.3952 - acc: 0.3581 - val_loss: 1.3201 - val_acc: 0.2162
Epoch 5/8
 - 31s - loss: 1.3783 - acc: 0.3611 - val_loss: 1.2938 - val_acc: 0.2162
Epoch 6/8
 - 31s - loss: 1.3465 - acc: 0.3608 - val_loss: 1.2727 - val_acc: 0.2162
Epoch 7/8
 - 31s - loss: 1.3236 - acc: 0.3707 - val_loss: 1.2493 - val_acc: 0.2162
Epoch 8/8
 - 31s - loss: 1.3339 - acc: 0.3659 - val_loss: 1.2239 - val_acc: 0.2162
Best validation acc of epoch: 0.32432432432432434
2947/2947 [==============================] - 3s 1ms/step
---------------------
|      Accuracy      |
---------------------
87.72%

----------------------------------
|      Best Hyper-Parameters      |
----------------------------------
{'Dropout': 0.692539034315719, 'Dropout_1': 0.21280043312755825, 'LSTM': 0}
```




WALKIN
WALKING_UPSTAIR
WALKING_DOWNSTAIR
SITTIN
STANDIN
LYIN
Predicted Class


Precision Matrix
Original Class
WALKING 0.92 0.01 0.09 0.00 0.00 0.00
WALKING_UPSTAIRS 0.04 0.92 0.01 0.00 0.00 0.00
WALKING_DOWNSTAIRS 0.01 0.02 0.90 0.00 0.00 0.00
SITTING 0.01 0.00 0.00 0.68 0.07 0.00
STANDING 0.01 0.00 0.00 0.32 0.93 0.00
LYING 0.00 0.06 0.00 0.00 0.00 1.00
WALKING
WALKING_UPSTAIRS
WALKING_DOWNSTAIRS
SITTING
STANDING
LYING
Predicted Class
1.0
0.8
0.6
0.4
0.2
0.0


Recall Matrix
Original Class
WALKING 0.91 0.01 0.08 0.00 0.00 0.00
WALKING_UPSTAIRS 0.04 0.95 0.01 0.00 0.00 0.00
WALKING_DOWNSTAIRS 0.01 0.02 0.96 0.00 0.00 0.00
SITTING 0.01 0.00 0.00 0.94 0.04 0.00
STANDING 0.01 0.00 0.00 0.40 0.58 0.00
LYING 0.00 0.05 0.00 0.00 0.00 0.95
WALKING
WALKING_UPSTAIRS
LKING_DOWNSTAIRS
SITTING
STANDING
LYING
0.8
0.6
0.4
0.2
0.0



Final Comments

- By Simple two layered LSTM, we got a good accuracy of 87.72%. In short, DeeP Learning help us to built models even when we don't have domain expert engineered features.
- LSTM model can be further improved by running it for more epochs and more evaluations while tuning hyper-parameter.