Gym Exercises Recognition for Embedded Systems

Adnane LAMNAOUAR

2023 / 2024

Import Libraries :

In [8]:
```
import os
import numpy as np
import pandas as pd
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, InputLayer
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.metrics import accuracy_score, classification_report
#from sklearn.ensemble import RandomForestrClassifier
from sklearn.ensemble import *

from sklearn.svm import SVC
from joblib import dump
from scipy.stats import skew, kurtosis
from numpy.fft import *
import time
import matplotlib.pyplot as plt
```

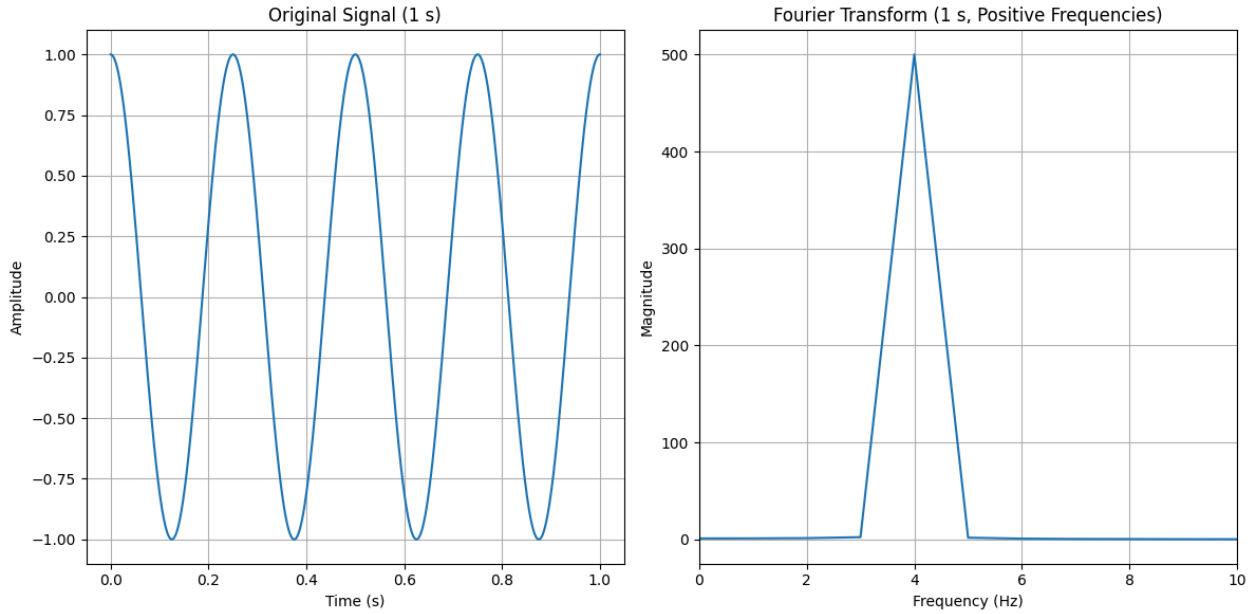
Showing the importance of sample size :

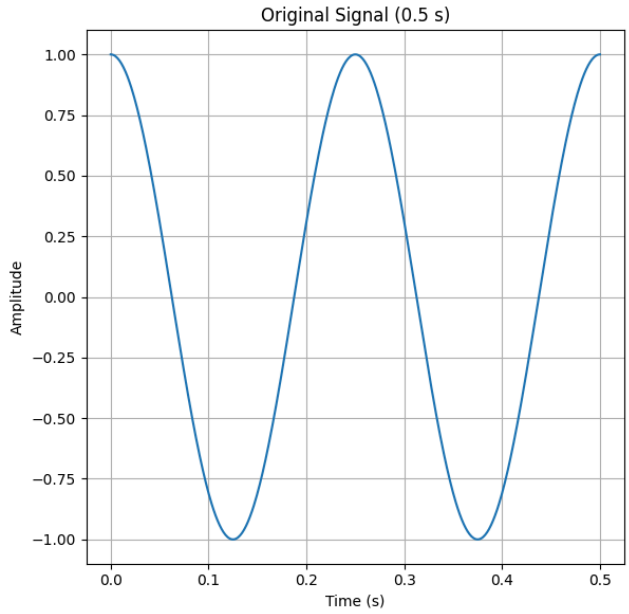
Plotting the Fourier transform of a periodic function defined as a cosine with a frequency of 4 Hz. Show the result under different time restrictions:

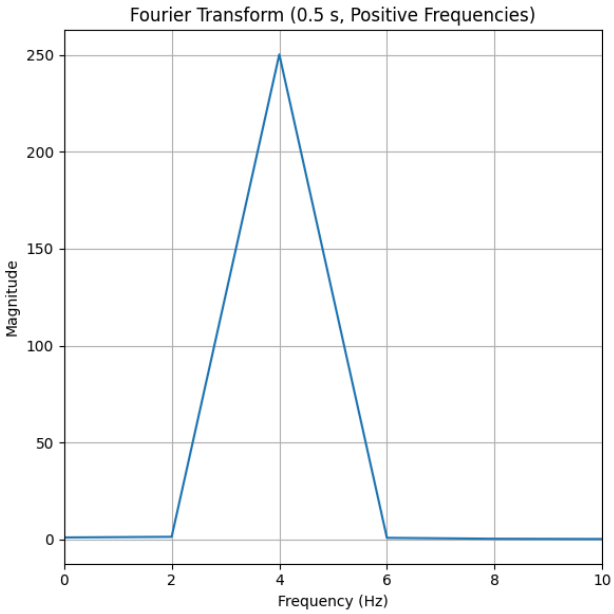
- 4 periods [0,1],
- 2 periods [0,0.5],
- 1 period [0,0.25],
- Half a period [0,0.125].

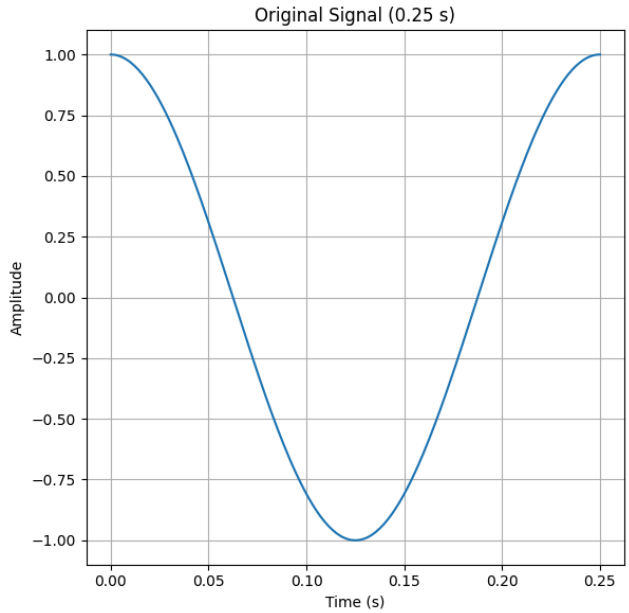
We display only the positive frequencies for each case and verify if the peak remains at 4 Hz.

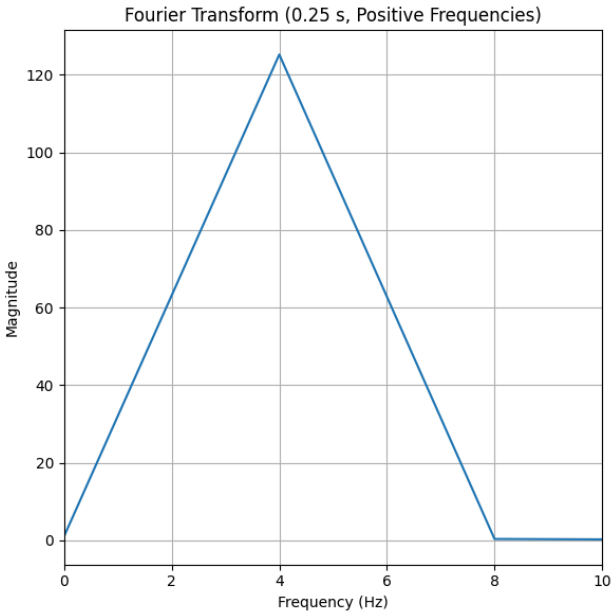
In [9]:
```
# Define a function to display signals and their Fourier transforms (positive frequencies only)
def plot_signal_and_positive_fft(frequency, duration, sampling_rate=1000):
    t = np.linspace(0, duration, int(duration * sampling_rate))  # Time in seconds
    signal = np.cos(2 * np.pi * frequency * t)  # Cosine function with the given frequency

    # Perform the Fourier Transform
    N = len(signal)
    T = 1 / sampling_rate  # Sampling interval
    yf = fft(signal)
    xf = fftfreq(N, T)

    # Keep only positive frequencies
    positive_xf = xf[:N // 2]
    positive_yf = np.abs(yf[:N // 2])

    # Plot the original signal and its Fourier Transform
    plt.figure(figsize=(12, 6))

    # Original signal
    plt.subplot(1, 2, 1)
    plt.plot(t, signal)
    plt.title(f"Original Signal ({duration} s)")
    plt.xlabel("Time (s)")
    plt.ylabel("Amplitude")
    plt.grid()

    # Magnitude of the Fourier Transform (positive part)
    plt.subplot(1, 2, 2)
    plt.plot(positive_xf, positive_yf)
    plt.title(f"Fourier Transform ({duration} s, Positive Frequencies)")
    plt.xlabel("Frequency (Hz)")
    plt.ylabel("Magnitude")
    plt.grid()
    plt.xlim(0, 10)  # Limit the display to relevant frequencies

    plt.tight_layout()
    plt.show()

# Test for each case
frequency = 4  # Frequency of 4 Hz
durations = [1, 0.5, 0.25, 0.125]  # Durations corresponding to 4 periods, 2 periods, 1 period, and 0.5 period

for duration in durations:
    plot_signal_and_positive_fft(frequency, duration)
```

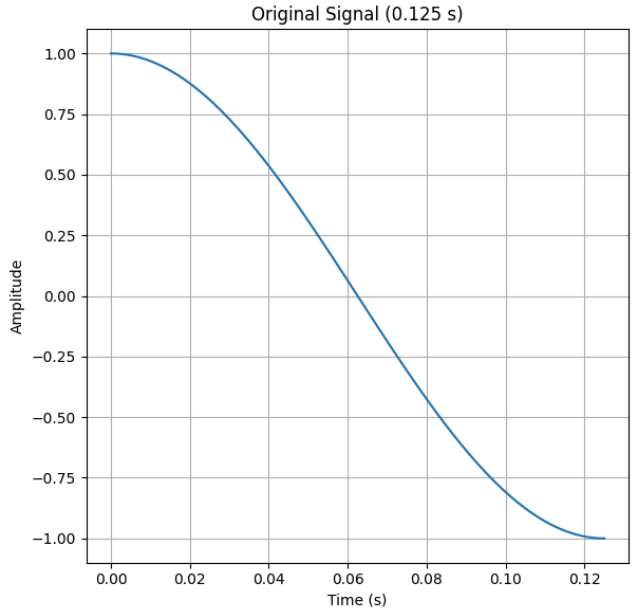


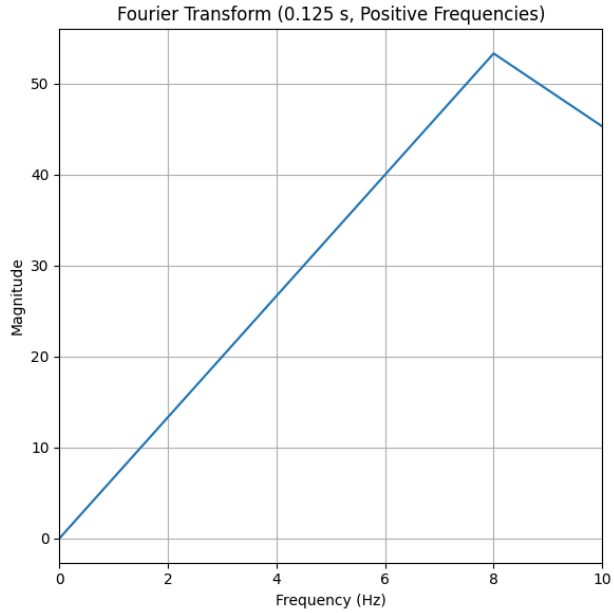












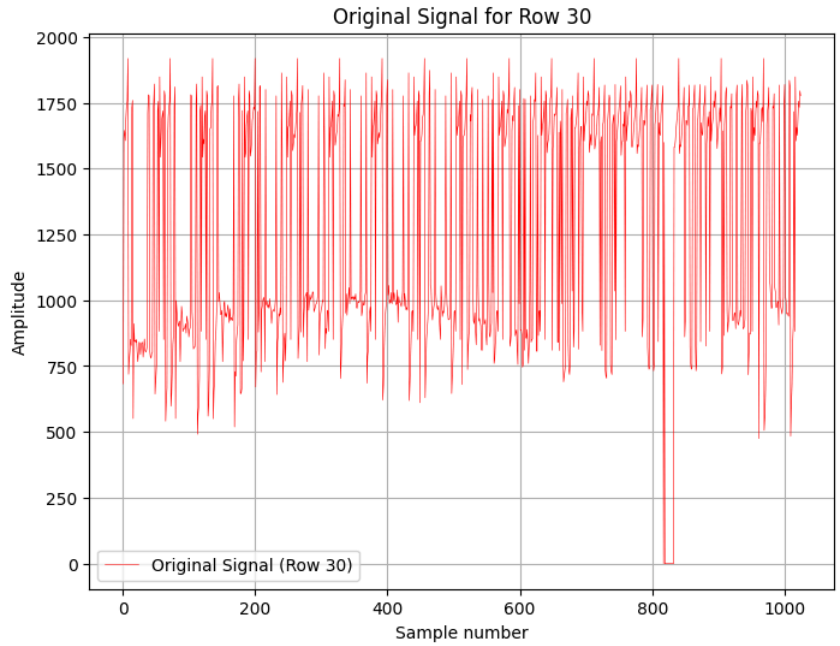
- **So, we need to take samples that cover at least one complete period of the signal. In our case, since the signal is not very smooth (with only 16 values per period), we need to cover slightly more than one period of the signal.**
- **The dataset we are loading next was collected according to the criteria mentioned above.**

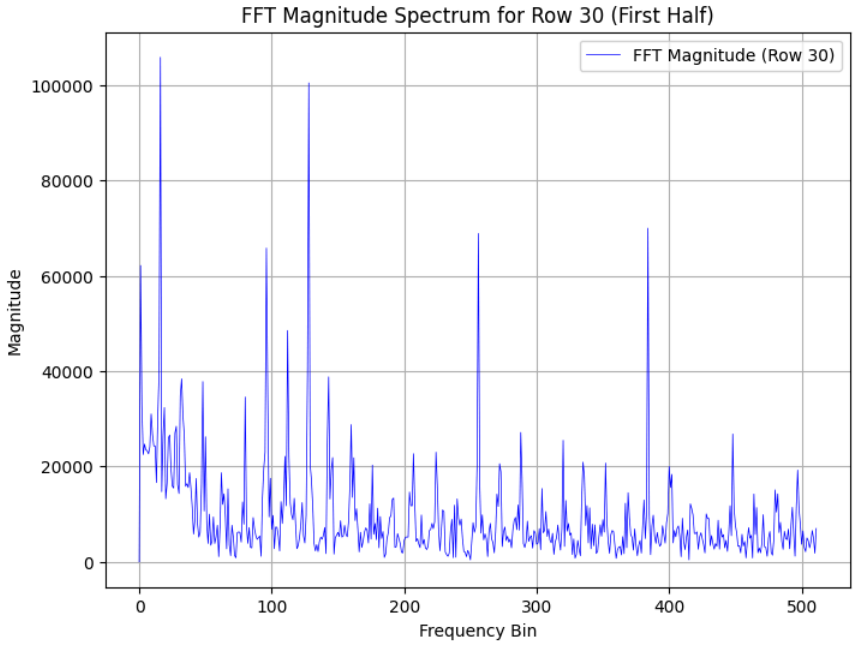
Load datasets :

In [10]:
```
standing_train = pd.read_csv('16_ready/standing_train.csv', delimiter=' ', header=None)
standing_test = pd.read_csv('16_ready/standing_test.csv', delimiter=' ', header=None)
squat_train = pd.read_csv('16_ready/squat_train.csv', delimiter=' ', header=None)
squat_test = pd.read_csv('16_ready/squat_test.csv', delimiter=' ', header=None)
shoulders_train = pd.read_csv('16_ready/shoulders_train.csv', delimiter=' ', header=None)
shoulders_test = pd.read_csv('16_ready/shoulders_test.csv', delimiter=' ', header=None)
nobody_train = pd.read_csv('16_ready/nobody_train.csv', delimiter=' ', header=None)
nobody_test = pd.read_csv('16_ready/nobody_test.csv', delimiter=' ', header=None)
arm_streching_train = pd.read_csv('16_ready/arm_streching_train.csv', delimiter=' ', header=None)
arm_streching_test = pd.read_csv('16_ready/arm_streching_test.csv', delimiter=' ', header=None)
```

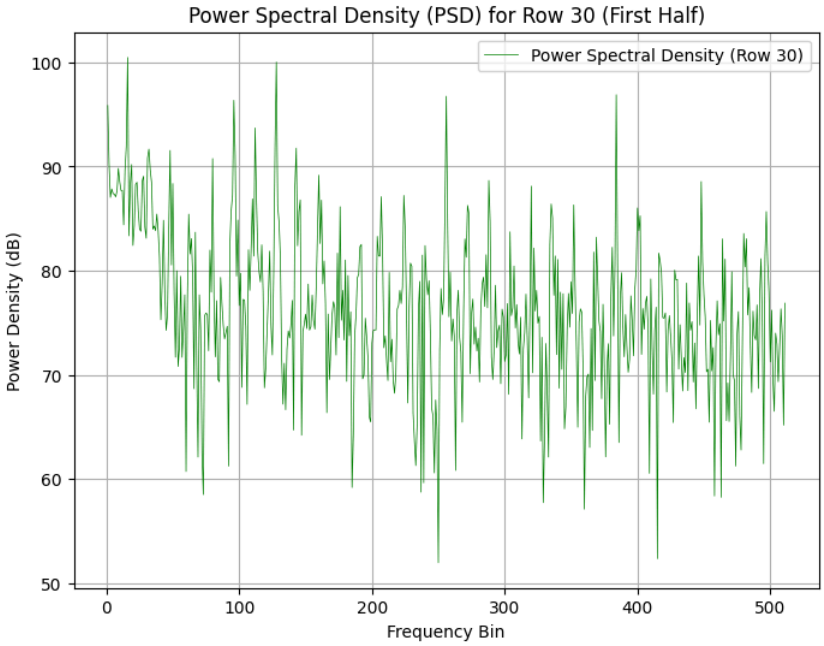
Data Visualisation :

In [13]:
```
# Apply FFT to the whole dataframe (row-wise FFT)
df1 = squat_train

# Copy the original dataframe to preserve the non-centered version for plotting
df_non_centered = df1.copy()

# Remove the mean from each row (centering) before applying FFT
df_centered = df1.sub(df1.mean(axis=1), axis=0)
fft_results = df_centered.apply(np.fft.fft, axis=1)

# Compute FFT magnitudes
fft_magnitudes = fft_results.apply(np.abs)

# Compute Power Spectral Density (PSD)
psd_results = fft_magnitudes.apply(np.square)

# Function to plot original signal, FFT magnitudes, and PSD for a specific row index
def plot_real_fft_psd(index):
    x = np.arange(1, df1.shape[1] + 1)  # Assuming 1024 points in the signal

    # Get original (non-centered) signal, FFT magnitudes, and PSD
    original_signal = df_non_centered.iloc[index]  # Non-centered signal for time-domain plotting

    # Plot only the first half of the FFT results
    signal_magnitudes = fft_magnitudes.iloc[index][:len(original_signal)//2]
    signal_psd = psd_results.iloc[index][:len(original_signal)//2]
    freq_bins = np.arange(0, len(original_signal)//2)  # First half of frequency bins

    # Plot original (non-centered) signal (real values)
    plt.figure(figsize=(8, 6))
    plt.plot(x, original_signal, label=f'Original Signal (Row {index})', color='red', linewidth=0.4)
    plt.title(f'Original Signal for Row {index}')
    plt.xlabel('Sample number')
    plt.ylabel('Amplitude')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Plot FFT Magnitude Spectrum (first half only)
    plt.figure(figsize=(8, 6))
    plt.plot(freq_bins, signal_magnitudes, label=f'FFT Magnitude (Row {index})', color='blue', linewidth=0.5)
    plt.title(f'FFT Magnitude Spectrum for Row {index} (First Half)')
    plt.xlabel('Frequency Bin')
    plt.ylabel('Magnitude')
    plt.legend()
    plt.grid(True)
    plt.show()

    # Plot PSD with logarithmic scale (first half only)
    plt.figure(figsize=(8, 6))
    plt.plot(freq_bins, 10 * np.log10(signal_psd), label=f'Power Spectral Density (Row {index})', color='green', linewidth=0.5)
    plt.title(f'Power Spectral Density (PSD) for Row {index} (First Half)')
    plt.xlabel('Frequency Bin')
    plt.ylabel('Power Density (dB)')
    plt.legend()
    plt.grid(True)
    plt.show()

# Example: To plot for a specific index (e.g., row 30)
index = 30
plot_real_fft_psd(index)
```






```
C:\Users\HP Zbook\AppData\Local\Temp\ipykernel_21228\2943849464.py:51: RuntimeWarning: divide by zero encountered in log10
  plt.plot(freq_bins, 10 * np.log10(signal_psd), label=f'Power Spectral Density (Row {index})', color='green', linewidth=0.5)
```



Add labels to each dataset :

In [14]:
```
# Add labels to each dataset :
standing_train['label'] = 'standing'
squat_train['label'] = 'squat'
shoulders_train['label'] = 'shoulders'
nobody_train['label'] = 'nobody'
arm_streching_train['label'] = 'arm_streching'

standing_test['label'] = 'standing'
squat_test['label'] = 'squat'
shoulders_test['label'] = 'shoulders'
nobody_test['label'] = 'nobody'
arm_streching_test['label'] = 'arm_streching'

# Concatenate all train and test data
train_data = pd.concat([standing_train,
                        squat_train,
                        shoulders_train,
                        nobody_train,
                       # arm_streching_train
                       ])
test_data = pd.concat([standing_test,
                       squat_test,
                       shoulders_test,
                       nobody_test,
                      # arm_streching_test
                      ])
print(arm_streching_train.info(),'\n')
print(train_data.info())

# Separate labels and features
X_train = train_data.drop(columns=['label'])
y_train = train_data['label']

X_test = test_data.drop(columns=['label'])
y_test = test_data['label']

# Ensure the dataframe contains only numeric data
def ensure_numeric(df):
    df_numeric = df.apply(pd.to_numeric, errors='coerce')  # Coerce non-numeric to NaN
    df_numeric = df_numeric.fillna(0)  # Fill NaNs with 0 for simplicity
    return df_numeric
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 140 entries, 0 to 139
Columns: 1025 entries, 0 to label
dtypes: float64(1024), object(1)
memory usage: 1.1+ MB
None

<class 'pandas.core.frame.DataFrame'>
Int64Index: 562 entries, 0 to 139
Columns: 1025 entries, 0 to label
dtypes: float64(1024), object(1)
memory usage: 4.4+ MB
None
```

Feature Extraction :

In [15]:
```
from scipy.stats import skew, kurtosis
from numpy.fft import fft

# Function to extract rich features from the raw data
def extract_rich_features(df):
    df = ensure_numeric(df)  # Ensure data is numeric
    features = pd.DataFrame()

    # Time-domain features
    features['mean'] = df.mean(axis=1)
    features['std'] = df.std(axis=1)
    #features['min'] = df.min(axis=1)
    #features['max'] = df.max(axis=1)
    features['skew'] = df.apply(lambda row: skew(row), axis=1)
    features['kurtosis'] = df.apply(lambda row: kurtosis(row), axis=1)

    # Root Mean Square (RMS) giving the energy of the signal
    features['rms'] = df.apply(lambda row: np.sqrt(np.mean(row**2)), axis=1)

    # Remove the mean from each row (centering the signal)
    df_centered = df.sub(df.mean(axis=1), axis=0)

    # Frequency-domain features (FFT) - considering only the first half of the FFT values
    fft_vals = df_centered.apply(lambda row: np.abs(fft(row.values))[:len(row)//2], axis=1)  # Take only the first half of FFT

    # Compute FFT-based features from the first half
    features['fft_mean'] = fft_vals.apply(np.mean)
    #features['fft_max'] = fft_vals.apply(np.max)
    features['fft_std'] = fft_vals.apply(np.std)
    features['fft_skew'] = fft_vals.apply(lambda x: skew(x))

    # Adding four most dominant frequencies from the first half of the FFT
    for i in range(4):
        features[f'dom_freq_{i+1}'] = fft_vals.apply(lambda x: np.argsort(x)[-i-1])  # Index of the ith most dominant frequency

    return features
```

In [16]:
```
# Convert NumPy arrays to DataFrame for feature extraction
X_train_df = pd.DataFrame(X_train)
X_test_df = pd.DataFrame(X_test)
```

In [17]:
```
# Feature extraction
start_time = time.time()
X_train_rich_features = extract_rich_features(X_train_df)
X_test_rich_features = extract_rich_features(X_test_df)
computation_time = time.time()- start_time
print(f"computation_time: {computation_time * 1000:.4f} milliseconds")
```

```
computation_time: 3207.2802 milliseconds
```

Scaling Data :

In [18]:
```
# Feature scaling for PCA, MLP and SVM
scaler = StandardScaler()
X_train_pca = scaler.fit_transform(X_train_rich_features)
X_test_pca = scaler.transform(X_test_rich_features)
```

Run only if you want to select Relevant features

Note : Applying PCA didn't give good results. If you runned this cell by mistake, please rerun the previous cell and skip this cell.

In [19]:
```
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler

# Apply PCA to reduce dimensionality (adjust 'n_components' as needed)
pca = PCA(n_components=0.97)  # Retain 98% of the variance
X_train_pca = pca.fit_transform(X_train_pca)  # Fit PCA on the training set
X_test_pca = pca.transform(X_test_pca)  # Apply PCA to the test set

# Print the amount of variance explained by each principal component
print("Explained variance ratio by PCA:", pca.explained_variance_ratio_)
print("Number of components selected:", pca.n_components_)
print(f"X_train_pca shape: {X_train_pca.shape}")
```

```
Explained variance ratio by PCA: [0.53370309 0.19252288 0.10146822 0.05852944 0.04547288 0.02466131
 0.01889449]
Number of components selected: 7
X_train_pca shape: (562, 7)
```

Encode labels to integers :

In [20]:
```
# Encode the labels (y_train, y_test) to integers
label_encoder = LabelEncoder()
y_train_encoded = label_encoder.fit_transform(y_train)
y_test_encoded = label_encoder.transform(y_test)
print(f"y_train_encoded shape: {y_train_encoded.shape}")
```

```
y_train_encoded shape: (562,)
```

MLP Model :

In [21]:
```
# ---------------- TensorFlow MLP Model ----------------
mlp_model = Sequential([
    InputLayer(input_shape=(X_train_pca.shape[1],)),  # Input shape matches the number of features
    Dense(80, activation='relu'),
    Dense(len(label_encoder.classes_), activation='softmax')  # Adjust output based on number of classes
])

mlp_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
mlp_model.fit(X_train_pca, y_train_encoded, epochs=30, verbose=0)

# Evaluate the TensorFlow MLP before quantization
start_time = time.time()
mlp_loss, mlp_accuracy = mlp_model.evaluate(X_test_pca, y_test_encoded, verbose=0)
inference_time = time.time() - start_time
print(f"MLP Accuracy: {mlp_accuracy * 100:.6f}%")
y_pred_mlp = np.argmax(mlp_model.predict(X_test_pca), axis=-1)
print("MLP Classification Report:\n", classification_report(y_test_encoded, y_pred_mlp, target_names=label_encoder.classes_))
print(f"Inference Time (Before Quantization): {inference_time * 1000:.4f} milliseconds")

# Save and calculate the size of the TensorFlow model
mlp_model.save('mlp_model.h5')
mlp_model_size = os.path.getsize('mlp_model.h5') / 1024  # Size in KB
print(f"MLP Model Size (Before Quantization): {mlp_model_size:.2f} KB \n")
```

```
MLP Accuracy: 92.045456%
3/3 [==============================] - 0s 4ms/step
MLP Classification Report:
               precision    recall  f1-score   support

      nobody       1.00      1.00      1.00        20
   shoulders       0.79      0.95      0.86        20
       squat       0.95      0.95      0.95        22
    standing       0.95      0.81      0.88        26

    accuracy                           0.92        88
   macro avg       0.93      0.93      0.92        88
weighted avg       0.93      0.92      0.92        88

Inference Time (Before Quantization): 392.5388 milliseconds
MLP Model Size (Before Quantization): 35.82 KB
```

Model Quantization :

In [22]:
```
# ---------------- Model Quantization ----------------
# Convert the trained TensorFlow model to TensorFlow Lite model and apply quantization
converter = tf.lite.TFLiteConverter.from_keras_model(mlp_model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]  # Apply 8-bit quantization
quantized_model = converter.convert()

# Save the quantized model
with open('mlp_model_quantized.tflite', 'wb') as f:
    f.write(quantized_model)

# Load and run inference on the quantized model using TensorFlow Lite interpreter
interpreter = tf.lite.Interpreter(model_content=quantized_model)
interpreter.allocate_tensors()

# Get input and output tensors
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

print("Input shape expected by the model:", input_details[0]['shape'])  # Should print [1, 10]

# Test the quantized model on the test set (one sample at a time)
start_time = time.time()
predictions = []

for i in range(X_test_pca.shape[0]):
    # Correctly shape the input to match the model's input dimensions (1 sample, 10 features)
    input_data = np.expand_dims(X_test_pca[i], axis=0).astype(np.float32)
    interpreter.set_tensor(input_details[0]['index'], input_data)
    interpreter.invoke()
    output_data = interpreter.get_tensor(output_details[0]['index'])
    predictions.append(np.argmax(output_data))

inference_time_quant = time.time() - start_time

# Get accuracy after quantization
accuracy_quant = accuracy_score(y_test_encoded, predictions)

# Quantized model size (memory footprint)
quantized_model_size = os.path.getsize('mlp_model_quantized.tflite') / 1024  # Size in KB

# Output Post-Quantization metrics
print("\nPost-Quantization Report:")
print(f"Accuracy: {accuracy_quant * 100:.6f}%")
print(f"Inference Time: {inference_time_quant * 1000:.4f} milliseconds")
print(f"Memory Footprint (model size): {quantized_model_size:.2f} KB")

# Emphasize improvements
speedup_factor = inference_time / inference_time_quant
memory_reduction = mlp_model_size / quantized_model_size

print(f"\nPerformance Improvements:")
print(f"Inference Speedup: {speedup_factor:.2f}x faster")
print(f"Memory Reduction: {memory_reduction:.2f}x smaller","\n")
```

```
WARNING:absl:Found untraced functions such as _update_step_xla while saving (showing 1 of 1). These functions will not be directly calla
ble after loading.

INFO:tensorflow:Assets written to: C:\Users\HPZBOO~1\AppData\Local\Temp\tmprrma2305\assets

INFO:tensorflow:Assets written to: C:\Users\HPZBOO~1\AppData\Local\Temp\tmprrma2305\assets

Input shape expected by the model: [1 7]

Post-Quantization Report:
Accuracy: 92.045455%
Inference Time: 5.9810 milliseconds
Memory Footprint (model size): 5.41 KB

Performance Improvements:
Inference Speedup: 65.63x faster
Memory Reduction: 6.62x smaller
```

Random Forest Model :

In [23]:
```
# ---------------- Random Forest Model ----------------

rf_model = RandomForestClassifier(n_estimators=20)  # Reduced number of trees for lightweight
rf_model.fit(X_train_pca, y_train_encoded)

# Evaluate Random Forest on the PCA-transformed test data
start_time = time.time()
y_pred_rf = rf_model.predict(X_test_pca)
inference_time = time.time() - start_time
rf_accuracy = accuracy_score(y_test_encoded, y_pred_rf)
print(f"Random Forest Accuracy: {rf_accuracy * 100:.4f}%")  # Accuracy as percentage with 4 decimal precision
print("RF Classification Report:\n", classification_report(y_test_encoded, y_pred_rf, target_names=label_encoder.classes_))

# Save and calculate the RF model size
rf_model_filename = 'rf_model.pkl'
dump(rf_model, rf_model_filename)
rf_model_size = os.path.getsize(rf_model_filename) / 1024  # Size in KB
print(f"RF Model Size: {rf_model_size:.2f} KB")
print(f"Inference Time: {inference_time * 1000:.4f} milliseconds")
```

```
Random Forest Accuracy: 90.9091%
RF Classification Report:
               precision    recall  f1-score   support

      nobody       1.00      1.00      1.00        20
   shoulders       0.77      0.85      0.81        20
       squat       0.88      1.00      0.94        22
    standing       1.00      0.81      0.89        26

    accuracy                           0.91        88
   macro avg       0.91      0.91      0.91        88
weighted avg       0.92      0.91      0.91        88

RF Model Size: 245.32 KB
Inference Time: 4.8754 milliseconds
```

SVM Model :

In [24]:
```
# ---------------- SVM Model ----------------
svm_model = SVC(kernel='linear', class_weight='balanced')
svm_model.fit(X_train_pca, y_train_encoded)

# Evaluate SVM
start_time = time.time()
y_pred_svm = svm_model.predict(X_test_pca)
inference_time = time.time() - start_time
svm_accuracy = accuracy_score(y_test_encoded, y_pred_svm)
print(f"SVM Accuracy: {svm_accuracy * 100:.4f}%")  # Accuracy as percentage with 4 decimal precision
print("SVM Classification Report:\n", classification_report(y_test_encoded, y_pred_svm, target_names=label_encoder.classes_))

# Save and calculate SVM model size
svm_model_filename = 'svm_model.pkl'
dump(svm_model, svm_model_filename)
svm_model_size = os.path.getsize(svm_model_filename) / 1024  # Size in KB
print(f"SVM Model Size: {svm_model_size:.2f} KB")
print(f"Inference Time: {inference_time * 1000:.4f} milliseconds")
```

```
SVM Accuracy: 89.7727%
SVM Classification Report:
               precision    recall  f1-score   support

      nobody       1.00      1.00      1.00        20
   shoulders       0.72      0.90      0.80        20
       squat       0.95      0.95      0.95        22
    standing       0.95      0.77      0.85        26

    accuracy                           0.90        88
   macro avg       0.91      0.91      0.90        88
weighted avg       0.91      0.90      0.90        88

SVM Model Size: 28.01 KB
Inference Time: 1.4179 milliseconds
```

In [ ]:
```
```