

Wearable-Driven Exercise Recognition

Why This Project ?

As we move further into the era of automation, the fitness industry still has significant scope for improvement in workout monitoring and tracking. Most individuals with regular gym experience rely on applications to log their workouts, such as the number of exercises, sets, and repetitions. However, this process is largely manual—users must select the exercise type and enter repetitions themselves, which can be time-consuming and error-prone.

To address this limitation, this project aims to automate workout tracking by designing a machine learning model that can detect and classify exercises using motion data captured from a fitness band worn on the wrist. By analyzing sensor data, the model can identify the performed exercise without requiring manual input from the user.

Currently, the model is trained to recognize barbell exercises, and future extensions may include additional workout types. This document focuses on the technical aspects of the project, including data processing, model design, and evaluation.

All libraries and their use-cases:

```
import pandas as pd # Used for all DataFrame-related operations
from glob import glob # Used to extract all CSV files from the MetaMotion folder
import matplotlib.pyplot as plt # Used for creating plots and visualizations
import matplotlib as mpl # Used for configuring global Matplotlib settings
import math # Used for basic mathematical operations required in statistical formulas.
```

```
import scipy # Used for probability and statistical functions.
from sklearn.neighbors import LocalOutlierFactor # Used for distance-based,
multivariate outlier detection.
from DataTransformation import LowPassFilter, PrincipalComponentAnalysis
# Noise reduction and dimensionality reduction
from TemporalAbstraction import NumericalAbstraction # Temporal (sliding-w
indow) feature extraction
from FrequencyAbstraction import FourierTransformation # Frequency-domain
feature extraction (FFT)
from sklearn.cluster import KMeans # Unsupervised clustering of motion patt
erns
```

Flow:

1. Make Dataset:

```
file = glob("MetaMotion/*.csv")
len(file)
```

- Here, we create a list containing the paths of all CSV files present in the `MetaMotion` directory.
- We can train machine learning model on one file only but as of now we have separate files for separate performer and for different sets and different weights so our first task is to merge all files into one file.

A machine learning model can be trained on a single file; however, in this dataset the data is distributed across multiple files. Each file corresponds to a different participant, exercise set, and weight. Therefore, the first step of this project is to merge all individual files into a single consolidated dataset that can be used for model training.

As a preliminary step, we extract metadata such as **participant**, **label**, and **category** from the filename and append this information as additional columns to the data.

```

acc_df = pd.DataFrame()
gyro_df = pd.DataFrame()

acc_set = 1
gyro_set = 1

for f in file:
    participant = f.split('-')[0].replace(datapath,'')
    label = f.split('-')[1]
    category = f.split('-')[2].rstrip("123").rstrip("_MetaWear_2019")

    df = pd.read_csv(f)
    df["participant"] = participant
    df["label"] = label
    df["category"] = category

    if "Accelerometer" in f:
        df["set"] = acc_set
        acc_set += 1
        acc_df = pd.concat([acc_df,df])

    elif "Gyroscope" in f:
        df["set"] = gyro_set
        gyro_set += 1
        gyro_df = pd.concat([gyro_df,df])

```

- In this step, the sensor timestamps stored as epoch time (in milliseconds) are converted into datetime format and set as the index of the accelerometer and gyroscope DataFrames. This transformation enables time-series operations such as resampling and alignment between sensors. After setting the datetime index, redundant timestamp-related columns are removed to avoid duplication and keep the dataset clean for further processing and model training.

```

acc_df.info()
pd.to_datetime(df["epoch (ms)"], unit="ms")

```

```
acc_df.index = pd.to_datetime(acc_df["epoch (ms)"], unit="ms")
gyro_df.index = pd.to_datetime(gyro_df["epoch (ms)"], unit="ms")
```

```
del acc_df["epoch (ms)"]
del acc_df["time (01:00)"]
del acc_df["elapsed (s)"]
```

```
del gyro_df["epoch (ms)"]
del gyro_df["time (01:00)"]
del gyro_df["elapsed (s)"]
```

epoch (ms)	x-axis (g)	y-axis (g)	z-axis (g)	participant	label	category	set
2019-01-11 15:42:43.566	-0.136	0.986	-0.053	B	ohp	heavy	1
2019-01-11 15:42:43.646	-0.143	0.977	-0.053	B	ohp	heavy	1
2019-01-11 15:42:43.726	-0.187	0.935	0.039	B	ohp	heavy	1
2019-01-11 15:42:43.806	-0.152	0.958	-0.075	B	ohp	heavy	1
2019-01-11 15:42:43.886	-0.143	0.944	0.010	B	ohp	heavy	1
...
2019-01-14 13:50:00.195	-0.085	0.911	-0.057	A	ohp	heavy	94
2019-01-14 13:50:00.275	-0.070	0.926	-0.102	A	ohp	heavy	94
2019-01-14 13:50:00.355	-0.082	0.955	-0.128	A	ohp	heavy	94
2019-01-14 13:50:00.435	-0.091	0.985	-0.119	A	ohp	heavy	94
2019-01-14 13:50:00.515	-0.084	0.988	-0.121	A	ohp	heavy	94

epoch (ms)	x-axis (deg/s)	y-axis (deg/s)	z-axis (deg/s)	participant	label	category	set
2019-01-15 19:09:07.087	4.451	-1.159	-0.061	A	squat	heavy	1
2019-01-15 19:09:07.127	3.902	-1.951	-1.220	A	squat	heavy	1
2019-01-15 19:09:07.167	-1.159	-1.585	-2.866	A	squat	heavy	1
2019-01-15 19:09:07.207	-5.732	-1.646	1.037	A	squat	heavy	1
2019-01-15 19:09:07.247	2.195	-4.024	0.061	A	squat	heavy	1
...
2019-01-18 16:46:23.293	7.988	-3.049	-0.549	D	squat	medium	93
2019-01-18 16:46:23.333	8.537	-2.622	0.061	D	squat	medium	93
2019-01-18 16:46:23.373	5.305	-1.951	0.061	D	squat	medium	93
2019-01-18 16:46:23.413	0.793	-1.951	0.305	D	squat	medium	93
2019-01-18 16:46:23.453	-3.598	-0.061	0.061	D	squat	medium	93

47218 rows x 7 columns

- The primary reason for converting the `epoch (ms)` column into a readable datetime format is that the other two time-related fields represent the same information in less useful forms. The epoch timestamp records time in milliseconds since 1 January 1970, providing the high precision required for sensor-based data. In contrast, the `time (01:00)` column does not reliably capture timestamps at millisecond resolution, making it unsuitable for precise time-series operations. The `elapsed (s)` column represents relative time since the start of recording, which does not add meaningful information for sensor alignment

or model training. Therefore, epoch time is retained and converted to a datetime index, while the remaining time-related columns are removed.

```
data_merged = pd.concat([acc_df.iloc[:,3],gyro_df],axis=1)
```

- In this step, accelerometer and gyroscope signals are merged using their datetime index to align measurements from both sensors. Since the sensors operate at different sampling rates, the combined dataset is resampled to a common frequency of 200 ms using mean aggregation for sensor values and forward propagation for metadata. This process produces a synchronized and clean time-series dataset suitable for machine learning.

```
# Accelerometer: 12.500HZ
# Gyroscope: 25.000Hz
sampling = {
    "acc_x": "mean",
    "acc_y": "mean",
    "acc_z": "mean",
    "gyr_x": "mean",
    "gyr_y": "mean",
    "gyr_z": "mean",
    "participant": "last",
    "label": "last",
    "category": "last",
    "set": "last",
}
data_merged[: 1000].resample(rule="200ms").apply(sampling)
# Split by day
days = [g for n, g in data_merged.groupby(pd.Grouper(freq="D"))]
data_resampled = pd.concat([df.resample(rule="200ms").apply(sampling).dropna() for df in days])
data_resampled["set"] = data_resampled["set"].astype("int")
data_resampled.info()
```

- Since the accelerometer and gyroscope operate at different sampling rates, the merged dataset is resampled into fixed 200 ms time windows. Sensor values within each window are averaged to align both signals in time and reduce noise, producing a uniform time-series suitable for machine learning.

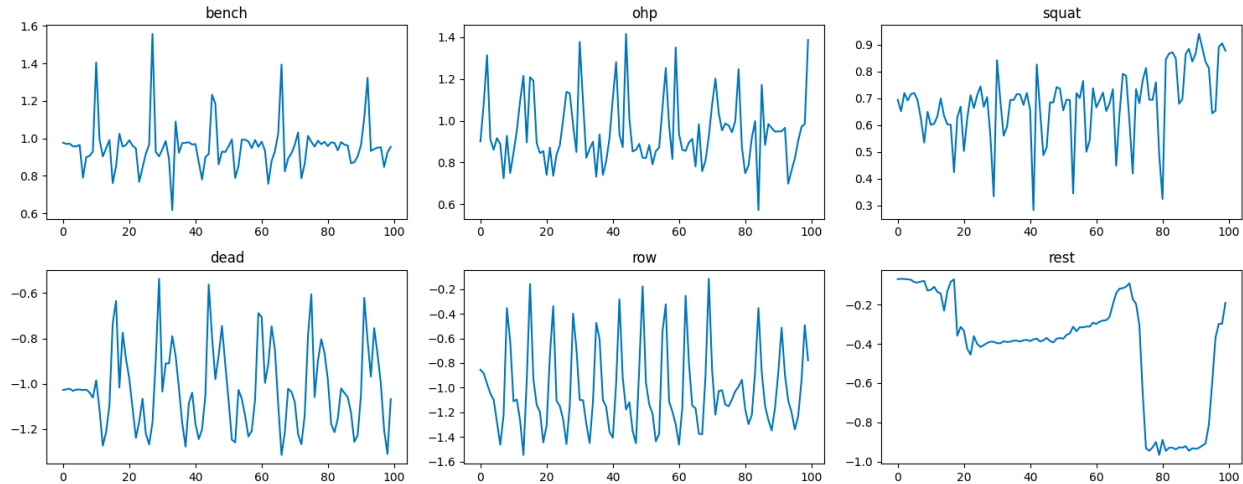
```
data_resampled.to_pickle("data_resampled.pkl")
```

- `.pkl` stands for **pickle**.
- In CSV format, data types are not preserved. For example, when a datetime column is stored in a `.csv` file, it is converted into a string and must be explicitly typecast back to a datetime object after loading. In contrast, a pickle (`.pkl`) file stores the Python object as it is, preserving data types and the DataFrame structure. This makes pickle especially useful for small to medium-sized datasets during experimentation and rapid iteration.

2. Visualisation:

```
for label in df["label"].unique():
    subset = df[df["label"] == label]
    fig, ax = plt.subplots()
    plt.plot(subset["acc_y"].reset_index(drop=True), label=label)
    plt.legend()
    plt.show()
```

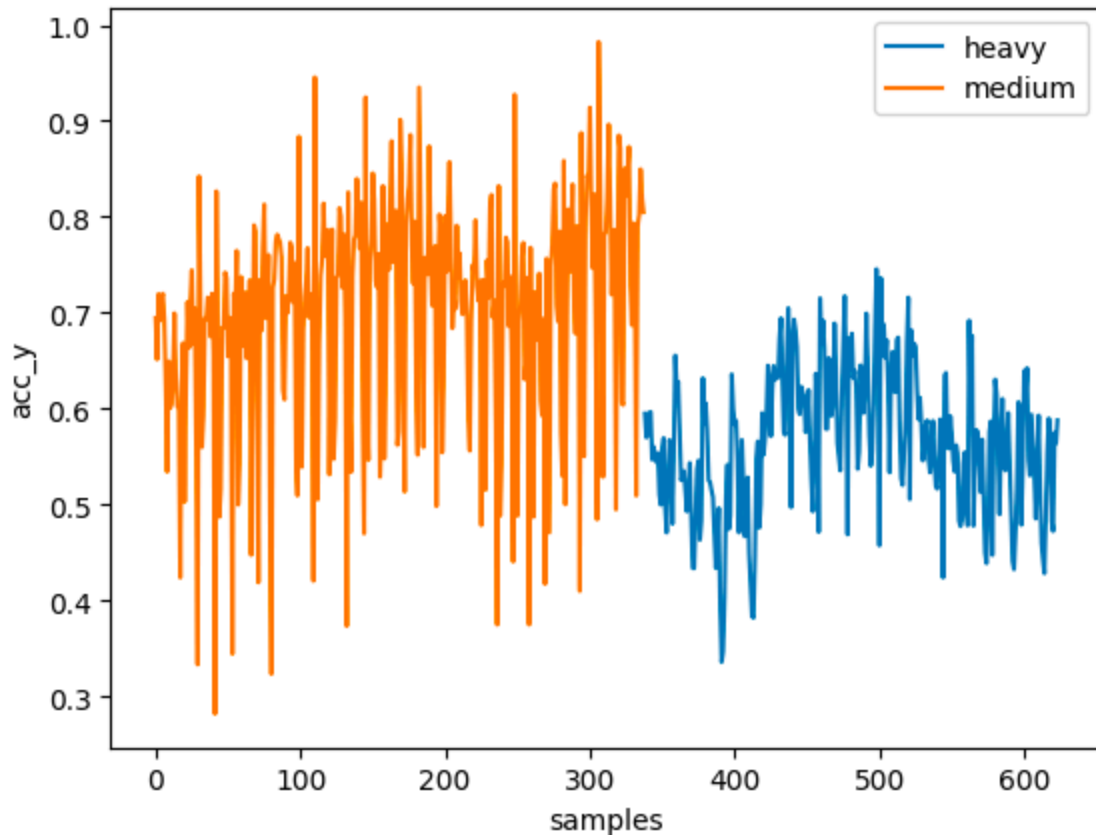
- Here we are plotting `acc_y` for all labels.



- Visual inspection of the accelerometer Y-axis signals reveals distinct motion patterns for each exercise. Pushing movements such as bench press and overhead press show sharp vertical spikes, while pulling movements like rows and deadlifts exhibit strong oscillatory patterns. Squats display smoother, full-body motion, and rest periods are characterised by low-variance signals. These clear differences indicate that the sensor data contains sufficient discriminative information for exercise classification.

```
category_df = df.query("label == 'squat'").query("participant == 'A'").reset_index(drop=True)
```

```
fig, ax = plt.subplots()
category_df.groupby(["category"])["acc_y"].plot()
ax.set_ylabel("acc_y")
ax.set_xlabel("samples")
plt.legend()
plt.show()
```



- For the squat exercise performed by participant A, the **medium** category shows higher variability and larger fluctuations in `acc_y`, indicating more dynamic or faster movements. In contrast, the **heavy** category exhibits lower variability with a more stable acceleration pattern, suggesting slower, more controlled motion under higher load. This difference indicates that movement intensity and control vary clearly with weight category, making `acc_y` useful for distinguishing exercise intensity.

3. Outliers Detection:

- Due to natural variations in human movement, sensor recordings may include unintended motions that appear as outliers. Identifying and removing these outliers helps improve data quality and model robustness.

```
df = pd.read_pickle("data_resampled.pkl")
outlier_columns = list(df.columns[:6])
```


- First, we load the processed dataset that was generated in the *dataset creation* (make dataset) component of the pipeline.

Method 1: IQR:

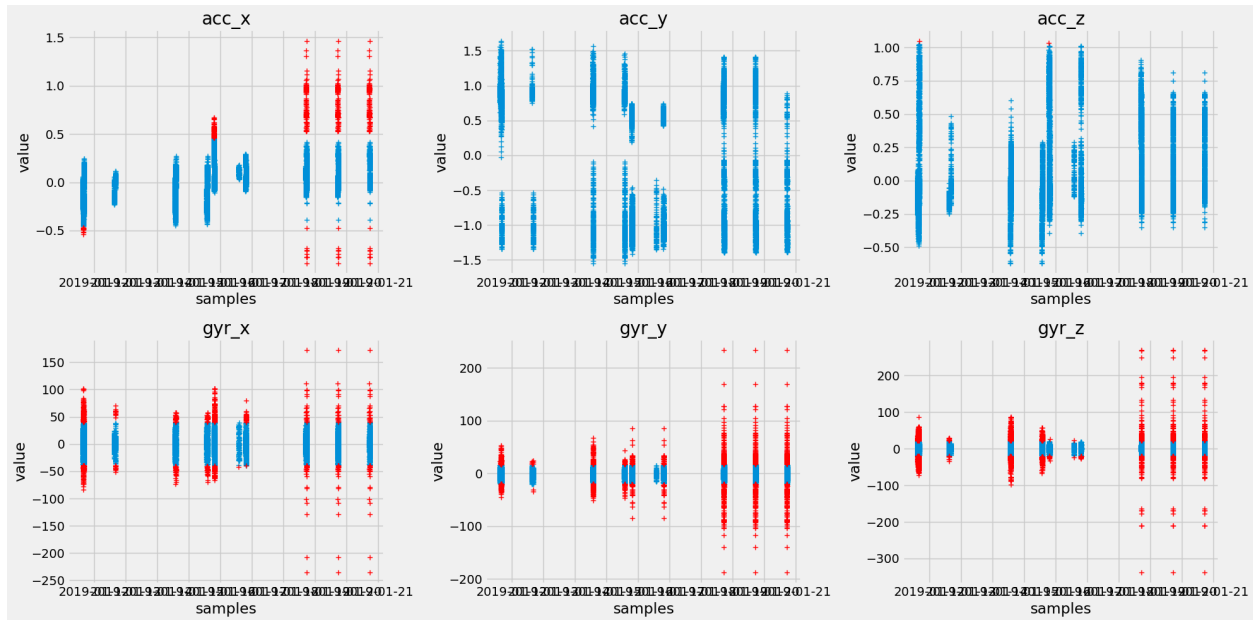
```
df[outlier_columns[:3] + ["label"]].boxplot(by="label", figsize=(20, 10), layout=(1,3))  
df[outlier_columns[3:] + ["label"]].boxplot(by="label", figsize=(20, 10), layout=(1,3))
```

- Here we are doing outliers detection using box plot.

A **box plot** is a statistical visualization technique used to summarize the distribution of a dataset and identify potential outliers. It represents the data using:

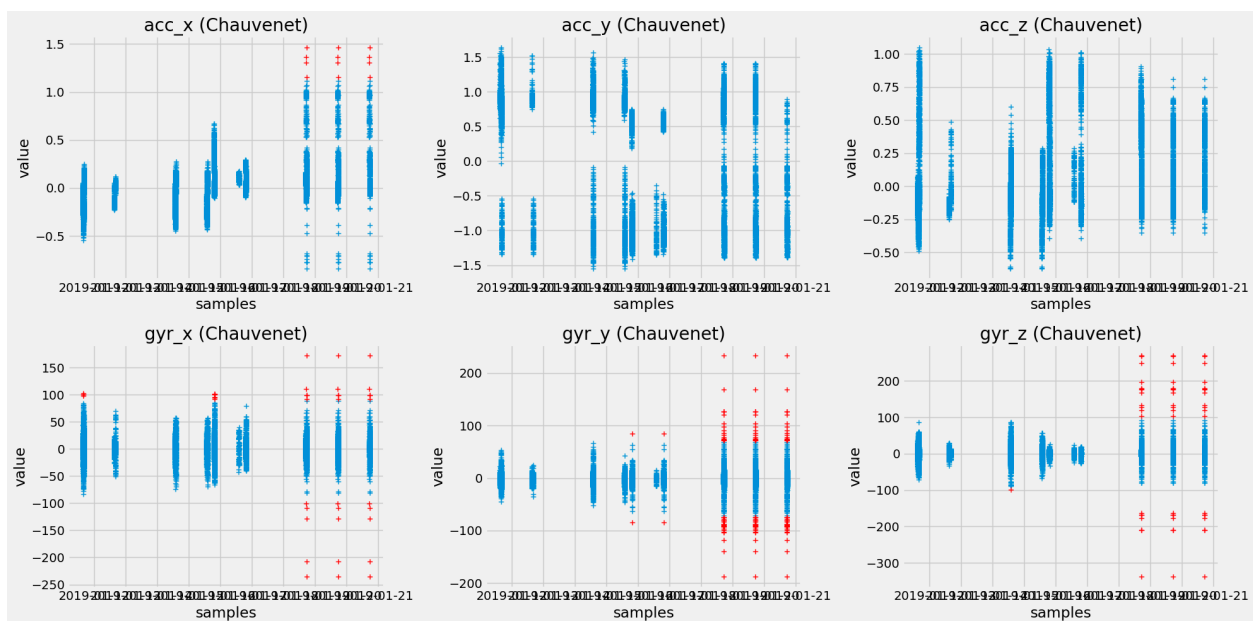
- **Median**
- **First quartile (Q1)**
- **Third quartile (Q3)**
- **Interquartile range (IQR)**

```
IQR = Q3 - Q1  
Q1 - 1.5 X IQR   or   Q3 + 1.5 X IQR
```



Method 2: Chauvenet Method

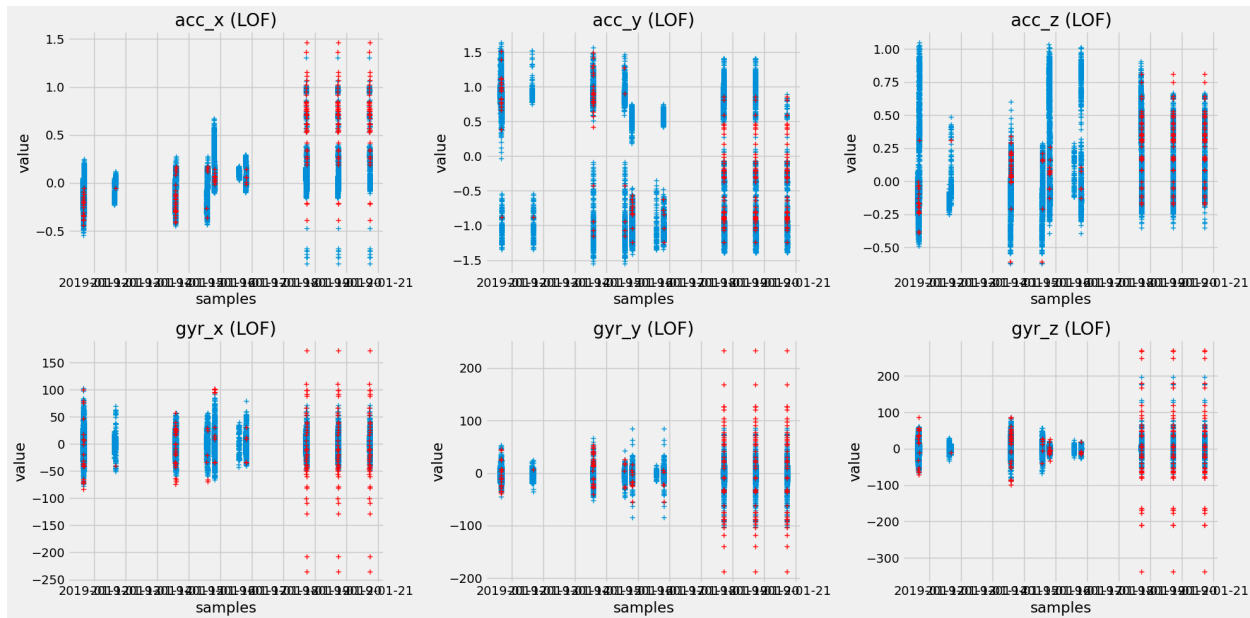
- **Chauvenet's criterion** is a **statistical, probability-based method** for detecting outliers under the assumption that the data follows a **normal (Gaussian) distribution**.
- Chauvenet removes statistically improbable sensor readings without over-filtering valid motion signals.



- Now we can notice that there are less and valid outliers.

Method 3: LOF

- **Local Outlier Factor (LOF)** is a density-based outlier detection algorithm.



- After analysing all we can notice that in all 3 methods Chauvenet is best method. It is manual method, we will select method which will detect outliers and reasonable amount.

```
outliers_removed_df.to_pickle("outliers_removed_chauvenet.pkl")
```

- Finally we will remove that and store other values in the pkl file.

4. Feature Engineering:

To calculate the **duration of each exercise set** and analyze how long different exercise categories take on average.

```
# Compute duration of a single set (example)
duration = df[df["set"]==1].index[-1] - df[df["set"]==1].index[0]
duration.seconds
```

```

# Compute duration for every set
for s in df["set"].unique():
    start = df[df["set"]==s].index[0]
    stop = df[df["set"]==s].index[-1]
    duration = stop - start
    # Store duration back into the dataframe
    df.loc[(df["set"]==s), "duration"] = duration.seconds

# Analyze duration by exercise category
duration_df = df.groupby(["category"])[["duration"]].mean()

# Normalize duration by repetitions
duration_df.iloc[0] / 5
duration_df.iloc[1] / 10

```

- Here we compute the duration of each exercise set using timestamps and analyze how set length varies across exercise categories to understand tempo and intensity differences.

```

df_lowpass = df.copy()
LowPass = LowPassFilter()

fs = 1000 / 200
# Frequencies above 1 Hz are treated as noise
# Human exercise movements typically occur below 1 Hz
cutoff = 1

df_lowpass = LowPass.low_pass_filter(df_lowpass, "acc_y", fs , cutoff)

subset = df_lowpass[df_lowpass["set"]==45]
print(subset["label"][0])

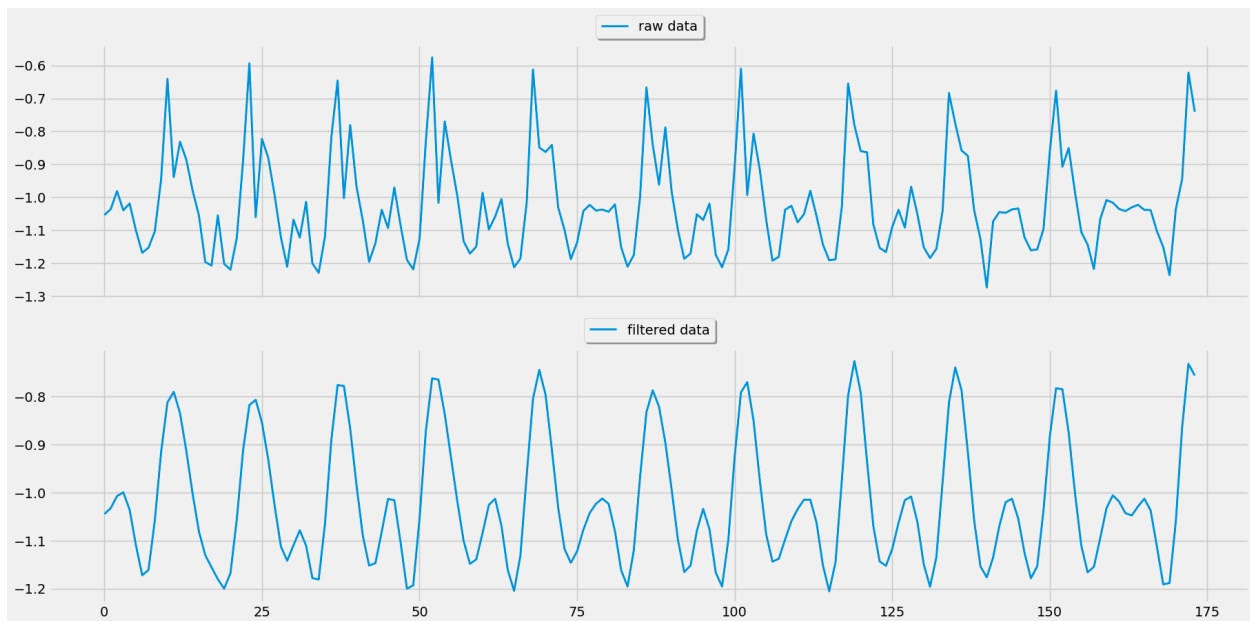
fig, ax = plt.subplots(nrows=2, sharex=True, figsize=(20, 10))
ax[0].plot(subset["acc_y"].reset_index(drop=True), label = "raw data")
ax[1].plot(subset["acc_y_lowpass"].reset_index(drop=True), label = "filtered d

```

```
ata")
ax[0].legend(loc="upper center", bbox_to_anchor=(0.5, 1.15), ncol=3, fancybox=True, shadow=True)
ax[1].legend(loc="upper center", bbox_to_anchor=(0.5, 1.15), ncol=3, fancybox=True, shadow=True)
```

```
for col in predicted_columns:
    df_lowpass = LowPass.low_pass_filter(df_lowpass, col, fs, cutoff, order=5)
    df_lowpass[col] = df_lowpass[col + "_lowpass"]
    del df_lowpass[col + "_lowpass"]
```

- To **remove high-frequency noise** from the sensor signals using a low-pass filter, while preserving the meaningful human motion patterns.



- High-frequency noise does not represent actual motion
- Can mislead feature extraction and ML models
- Temporal features (mean, std) become smoother
- Frequency features become more meaningful

PCA:

- To **reduce the dimensionality** of the sensor data by transforming correlated sensor axes into a smaller set of informative, uncorrelated features.

```
df_pca = df_lowpass.copy()
PCA = PrincipalComponentAnalysis()

# Applies PCA using all sensor axes
# Computes how much variance each principal component explains
pc_values = PCA.determine_pc_explained_variance(df_pca, predicted_columns)

plt.figure(figsize=(10, 10))
plt.plot(range(1, len(predicted_columns) + 1), pc_values)
plt.xlabel("principle component number")
plt.ylabel("explained variance")
plt.show()

df_pca = PCA.apply_pca(df_pca, predicted_columns, 3)

subset = df_pca[df_pca["set"]==35]
subset[["pca_1", "pca_2", "pca_3"]].plot()
```

```
df_squared = df_pca.copy()
acc_r = df_squared["acc_x"]**2 + df_squared["acc_y"]**2 + df_squared["acc_z"]**2
gyr_r = df_squared["gyr_x"]**2 + df_squared["gyr_y"]**2 + df_squared["gyr_z"]**2

df_squared["gyr_r"] = np.sqrt(gyr_r)
df_squared["acc_r"] = np.sqrt(acc_r)

subset = df_squared[df_squared["set"]==14]
subset[["acc_r", "gyr_r"]].plot(subplots=True)
```

Feature	Captures
PCA	Dominant movement directions
acc_r	Overall movement intensity

```

df_temporal = df_squared.copy()
Numbs = NumericalAbstraction()

predicted_columns = predicted_columns + ["acc_r", "gyr_r"]

ws = int(1000 / 200)

for col in predicted_columns:
    df_temporal = Numbs.abstract_numerical(df_temporal,[col], ws, "mean")
    df_temporal = Numbs.abstract_numerical(df_temporal,[col], ws, "std")

df_temporal_list = []
for s in df_temporal["set"].unique():
    subset = df_temporal[df_temporal["set"]==s].copy()
    for col in predicted_columns:
        subset = Numbs.abstract_numerical(subset,[col], ws, "mean")
        subset = Numbs.abstract_numerical(subset,[col], ws, "std")
    df_temporal_list.append(subset)

df_temporal = pd.concat(df_temporal_list)
df_temporal.info()

subset[["acc_y", "acc_y_temp_mean_ws_5", "acc_y_temp_std_ws_5"]].plot()
subset[["gyr_y", "gyr_y_temp_mean_ws_5", "gyr_y_temp_std_ws_5"]].plot()

```

- ML models struggle with millisecond-level noise
- Temporal summaries capture **behavior**, not spikes
- To extract **frequency-domain features** from sensor data in order to capture **exercise rhythm, repetition speed, and motion regularity**, which are not visible in the time domain.

```

df_freq = df_temporal.copy().reset_index()
FreqAbs = FourierTransformation()

fs = int(1000 / 200)
ws = int(2800 / 200)

df_freq = FreqAbs.abstract_frequency(df_freq, ["acc_y"], ws, fs)

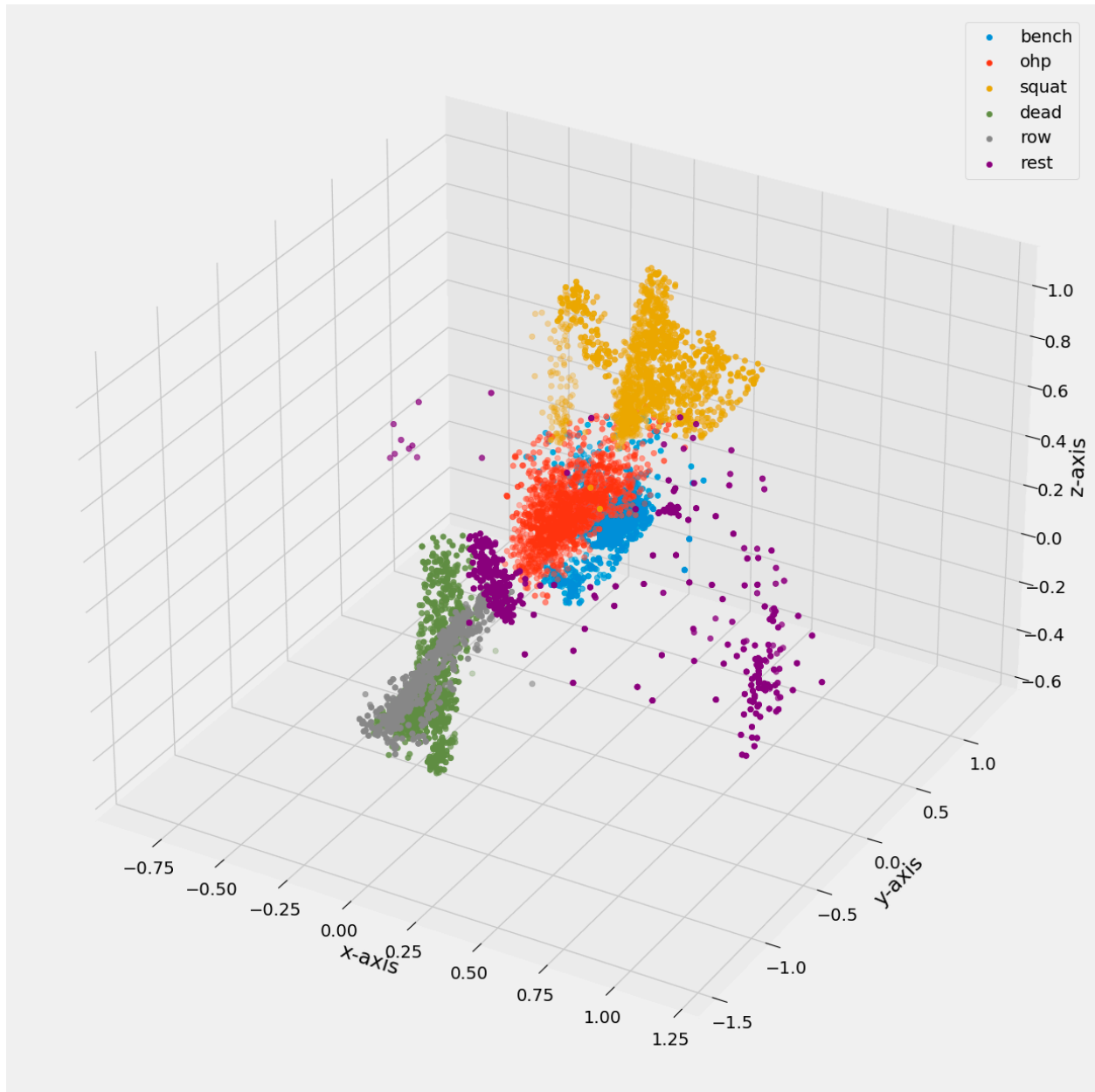
subset = df_freq[df_freq["set"] == 15]
subset[["acc_y"]].plot()

subset[[
    "acc_y_max_freq",
    "acc_y_freq_weighted",
    "acc_y_pse",
    "acc_y_freq_1.429_Hz_ws_14",
    "acc_y_freq_2.5_Hz_ws_14",
]].plot()

df_freq_list = []
for s in df_freq["set"].unique():
    print(f"Applying Fourier transformation to set {s}")
    subset = df_freq[df_freq["set"] == s].reset_index(drop=True).copy()
    df_freq_list.append(subset)

df_freq = pd.concat(df_freq_list).set_index("epoch (ms)", drop=True)

```

cluster for different exercise.