ELEC 292

Project Report

Group Number 65
Vikran Vigneswaran: 20418569, 22HR38@queensu.ca
Ray Chen: 20418768, 22SD60@queensu.ca
Jack Marcinow: 20418837, 22RQD@queensu.ca

May 4, 2025

1. Data Collection:

The project at hand required for the members of the group to collect motion data while jumping and while walking. In order to achieve this, we used the Phyphox app on our mobile devices which allowed us to use the built in accelerometers to record the acceleration along the x, y and z axes and this is shown in *figure 1* below. Once recorded, the data was exported in CSV format via email to allow for further processing.

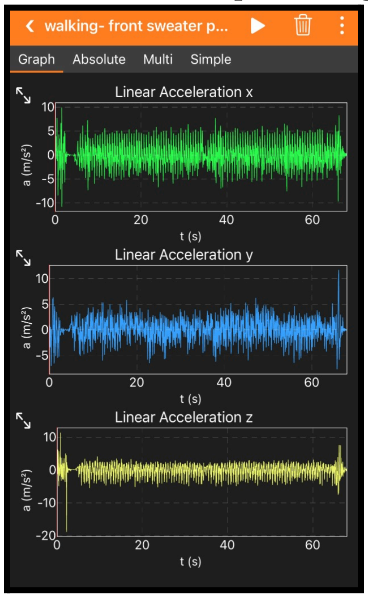


Figure 1: Phyphox data collection app

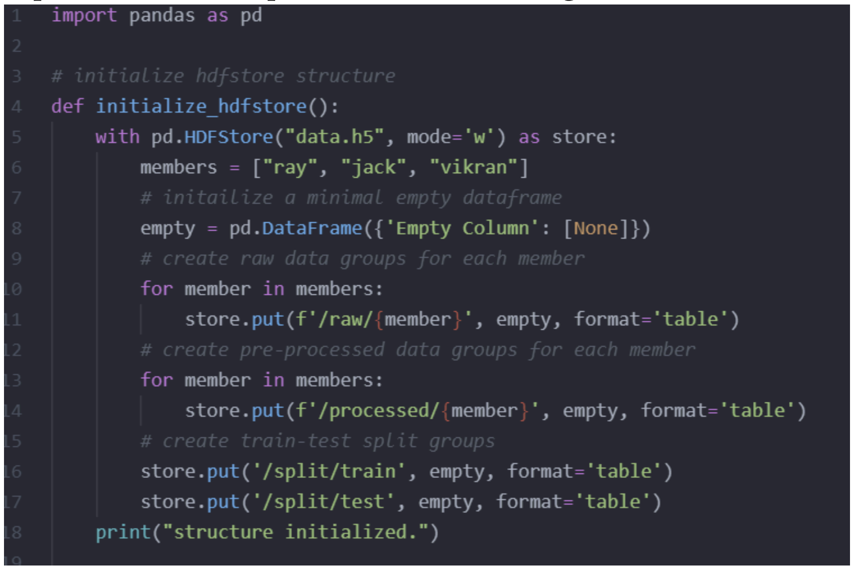
Each group member was required to collect their own sets of data in order to ensure that the data collected was diverse. To make for a balanced dataset between the group members, we all recorded around five to six minutes of data while maintaining an even duration of walking and jumping data. Data variability was also increased by conducting the walking and jumping data in multiple sets where each time the phone was held in a different position which included left and right hands, left and right pockets, sweater pockets and back pockets. Ray also further introduced variability by including changes of elevation by walking and jumping up and down stairs. Each set was labeled either “jumping” or “walking” along with where the phone was held for that specific set.

Some challenges arose throughout the data collection process with one example being finding a good area to conduct the trials so that there is nothing that may alter and change the output. We overcame this challenge by planning out where we would be collecting the data in order to remove obstacles in our way to ensure a good read of the data. For example when collecting the data for walking, a route was figured out of where in the house we would be walking. We then moved objects like chairs and tables out of the way to ensure a smooth collection process and attempting to avoid bumping into something and negatively affecting the data.

Overall, the data collection process was carefully designed designed to collect the most authentic data possible so that we can create a proper dataset suitable for the project at hand.

2. Data Storage:

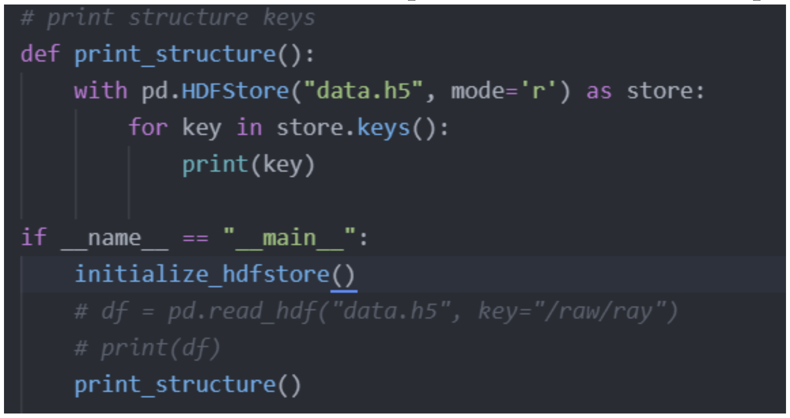
For this project, when it came to data storage, the HDF5 file format was chosen because it allowed an organized and efficient way to store the data from the accelerometer. This was done using “pandas.HDFStore” in python to build a structure that held everything ranging from the raw collected data, to the processed data, and to the final testing/training sets in the same place. To create this **initialize_hdfstore**() function was called to create a layout for the file “data.h5”. In this file each team member got their own allocated space in the file under “/raw/{member}” for the raw data, and “/processed/{member}” for the processed data. There were also specific sections added for the training data under “/split/train”, and “/split/test” for the testing data.



```python
import pandas as pd

# initialize hdfstore structure
def initialize_hdfstore():
    with pd.HDFStore("data.h5", mode='w') as store:
        members = ["ray", "jack", "vikran"]
        # initailize a minimal empty dataframe
        empty = pd.DataFrame({'Empty Column': [None]})
        # create raw data groups for each member
        for member in members:
            store.put(f'/raw/{member}', empty, format='table')
        # create pre-processed data groups for each member
        for member in members:
            store.put(f'/processed/{member}', empty, format='table')
        # create train-test split groups
        store.put('/split/train', empty, format='table')
        store.put('/split/test', empty, format='table')
    print("structure initialized.")
```

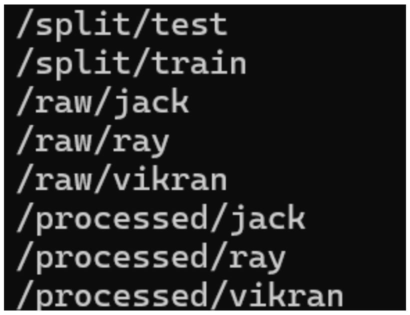
Figure 2 : Initialization of storage format

The **print_structure**() function was called to ensure the structure was created properly, the function would print all the keys contained in the file and would confirm that the data, whether that be raw, processed, and train/test splits were in the correct spot.

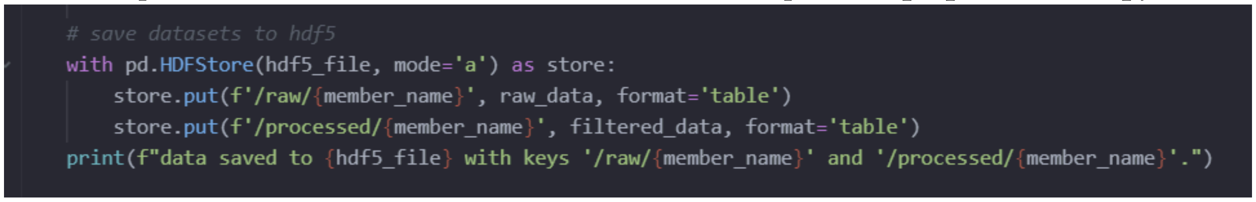


```python
# print structure keys
def print_structure():
    with pd.HDFStore("data.h5", mode='r') as store:
        for key in store.keys():
            print(key)

if __name__ == "__main__":
    initialize_hdfstore()
    # df = pd.read_hdf("data.h5", key="/raw/ray")
    # print(df)
    print_structure()
```

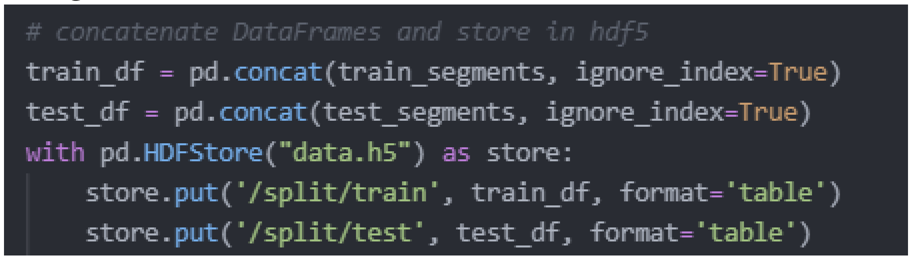
Figure 3 : Function to visualize file structure


```
/split/test
/split/train
/raw/jack
/raw/ray
/raw/vikran
/processed/jack
/processed/ray
/processed/vikran
```
Figure 4: Output of print function to show file structure

Once all the data was collected, the raw accelerometer readings were saved to each individual's raw data section. After this the data was cleaned by smoothing it out and reducing the method which was achieved by a moving average filter, and then was saved under the processed section, the function that was called upon was **preprocess_data.py**.


```
# save datasets to hdf5
with pd.HDFStore(hdf5_file, mode='a') as store:
    store.put(f'/raw/{member_name}', raw_data, format='table')
    store.put(f'/processed/{member_name}', filtered_data, format='table')
print(f"data saved to {hdf5_file} with keys '/raw/{member_name}' and '/processed/{member_name}'.")
```
Figure 5 : Store data after preprocessing according to structure

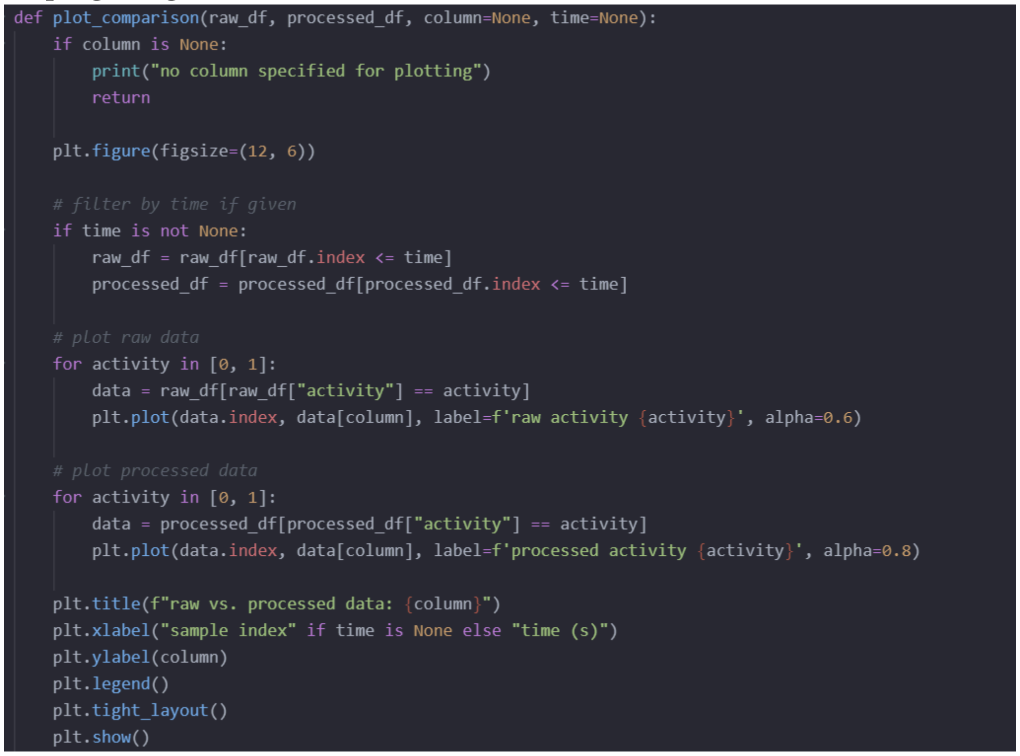
Once preprocessing was done, the data was then separated into 5 second windows and then was randomly split into 90% for training and 10% for testing. Then these two sets were combined into two dataframes, **train_df** and **test_df**, and saved into the correct part of the HDF5 file. This step ensured that the machine learning model was trained on one set of data and was tested on a completely separate set of data which would prevent overlap or data leakage.


```
# concatenate DataFrames and store in hdf5
train_df = pd.concat(train_segments, ignore_index=True)
test_df = pd.concat(test_segments, ignore_index=True)
with pd.HDFStore("data.h5") as store:
    store.put('/split/train', train_df, format='table')
    store.put('/split/test', test_df, format='table')
```
Figure 6: Function to store segmented data

Altogether, having stored all the data in a HDF5 file allowed to keep the flow of data clean and consistent, which made it easier to work with it. It also made it easier to access original and process data for all parts of the workflow.

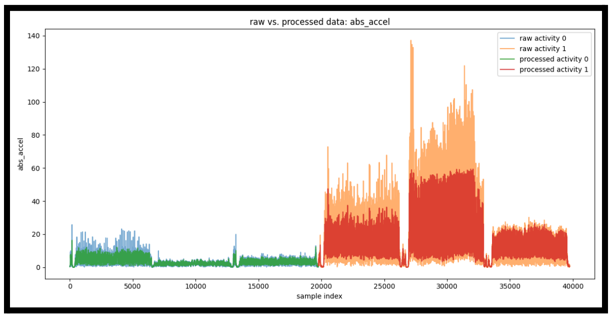
3. Visualization:

The function below named **plot_comparison** was designed to visually compare the raw and pre-processed data for walking and jumping. It intakes raw and filtered data along with which column to plot and the option to add a time limit (for better visualization). If no column is given then the function exits and tells the user that there is no column. If a time value is provided, then the data is filtered to only include data from within the first **time** seconds. The function generates a line plot using Matplotlib, which shows both the raw and processed data for walking and jumping along with labels, axis titles, a legend, and a plot title. This function helps visualize how the filtering process helps reduce noise while also keeping the general data.



```python
def plot_comparison(raw_df, processed_df, column=None, time=None):
    if column is None:
        print("no column specified for plotting")
        return

    plt.figure(figsize=(12, 6))

    # filter by time if given
    if time is not None:
        raw_df = raw_df[raw_df.index <= time]
        processed_df = processed_df[processed_df.index <= time]

    # plot raw data
    for activity in [0, 1]:
        data = raw_df[raw_df["activity"] == activity]
        plt.plot(data.index, data[column], label=f'raw activity {activity}', alpha=0.6)

    # plot processed data
    for activity in [0, 1]:
        data = processed_df[processed_df["activity"] == activity]
        plt.plot(data.index, data[column], label=f'processed activity {activity}', alpha=0.8)

    plt.title(f"raw vs. processed data: {column}")
    plt.xlabel("sample index" if time is None else "time (s)")
    plt.ylabel(column)
    plt.legend()
    plt.tight_layout()
    plt.show()
```

Figure 7 : Function to show comparison plot



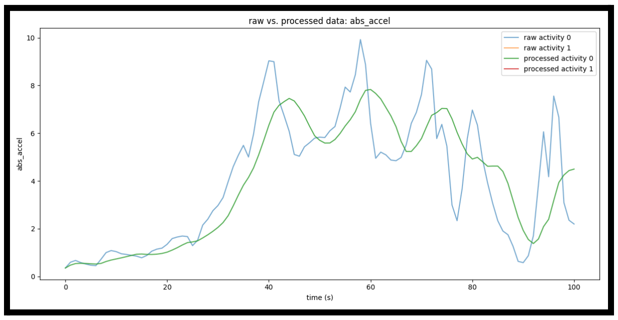
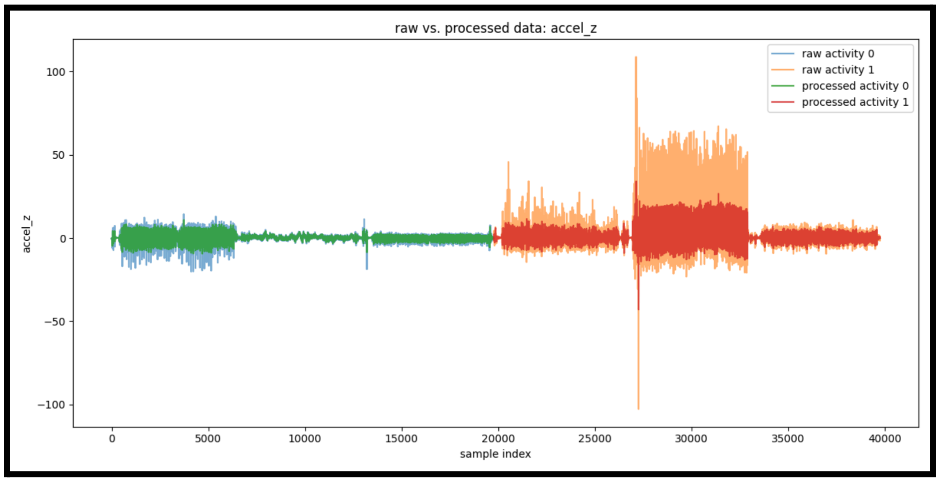
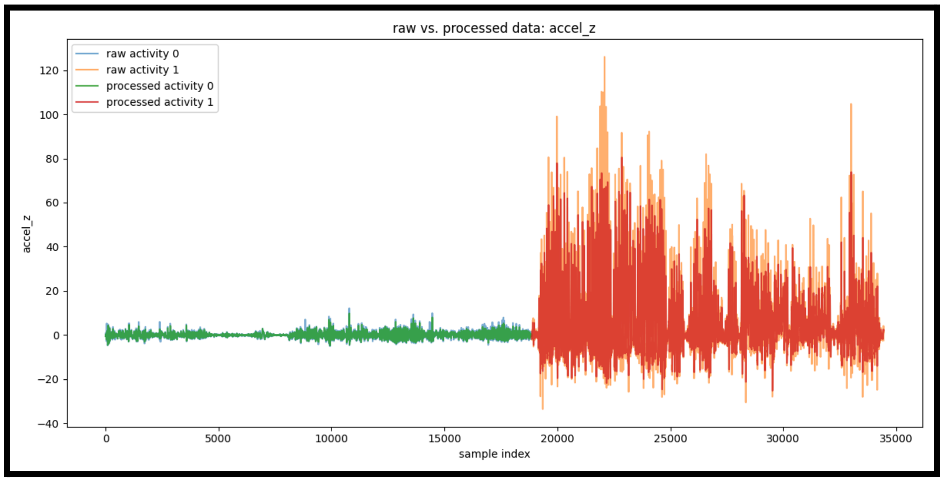


Figure 8 & 9 : Jack's abs_accel function output with and without specified time frame

In order to better understand the collected data and seek out issues within it, we implemented multiple different visualization techniques. We first had to plot the raw versus the pre-processed acceleration signals in order to see the impact of filtering. These plots were made separately for each group member's data across the z-axis and can be seen below in figures 9, 10, and 11. To distinguish between both types of activities in the visualization process, we gave activity labels to the data segments where "0" represented walking and "1" represented jumping. The labels were used again to annotate the plots to make it easier to see and compare the data. The initial visualizations show how noisy the raw data was especially during unusual phone movement which proves the need for pre-processing.





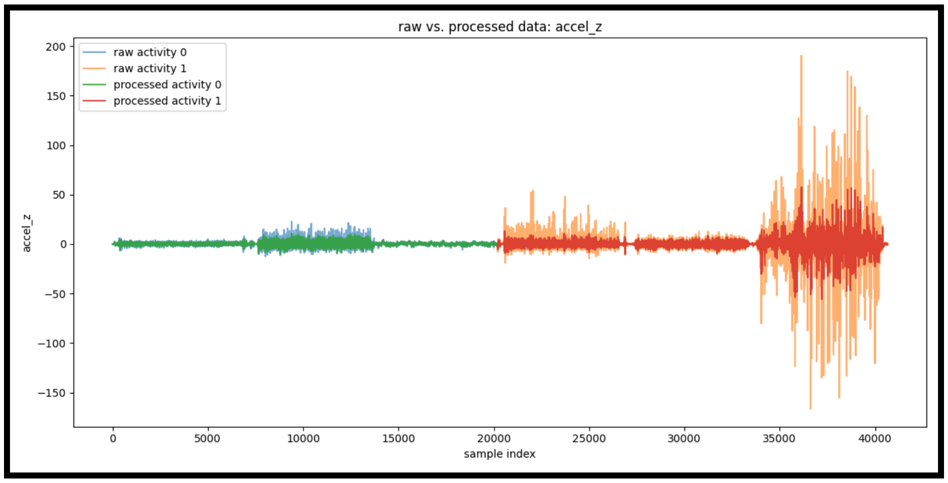
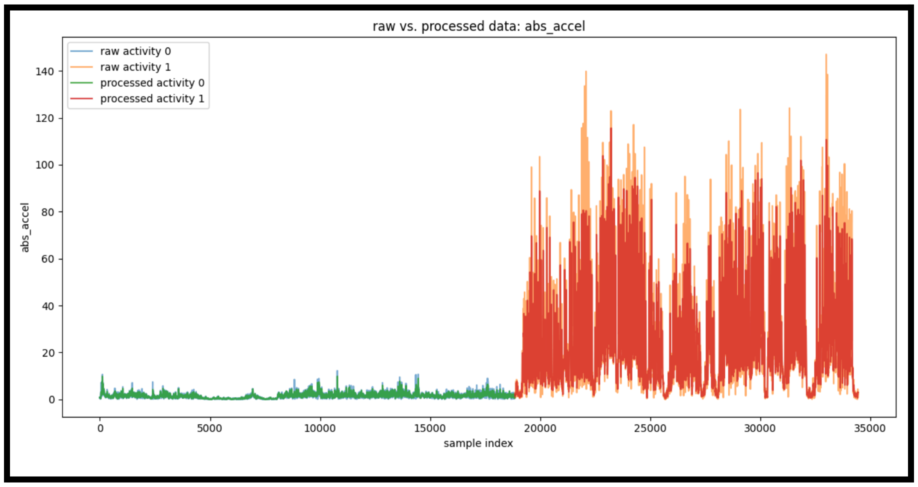


Figure 10, 11, &12 : Plot showing Jack, Ray, and Vikran raw vs processed data for acceleration in the z plane

Along with the z-axis plots, we also visualized the absolute acceleration across all three axes, which gave a more complete view of the overall motion data. This served useful in differentiating between walking and jumping, as jumping mostly showed sharper peaks along with higher variation, where walking was often a more smoother wave with a lower intensity.



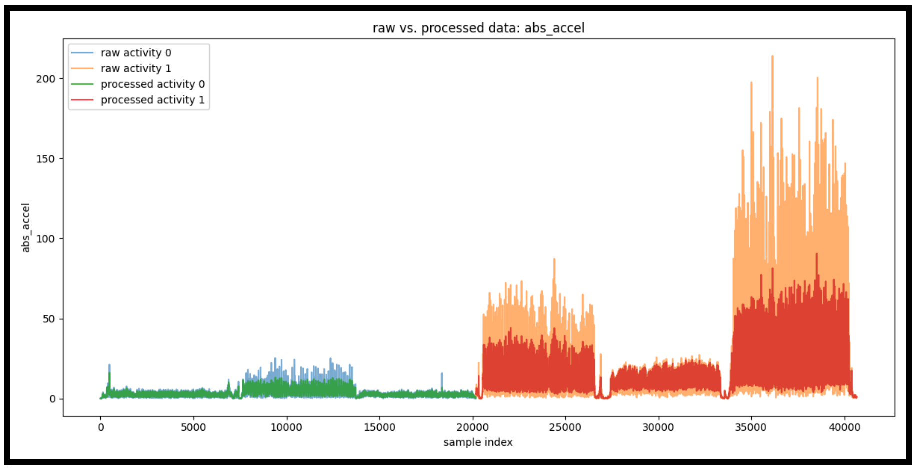
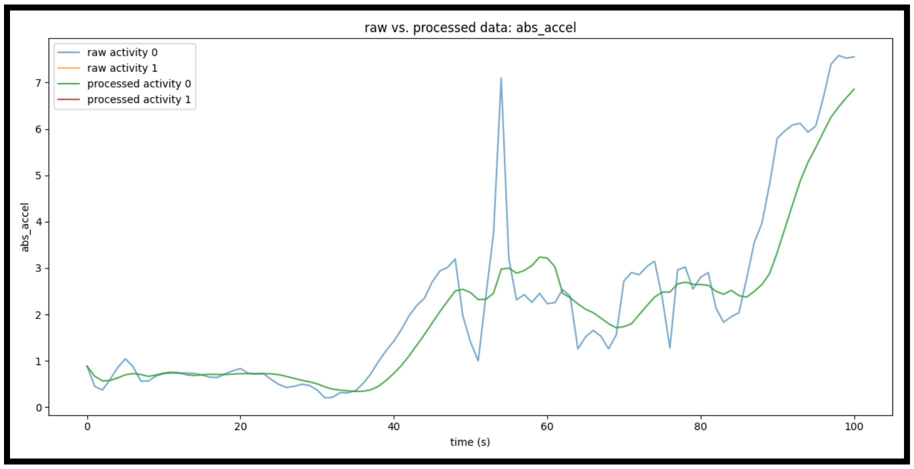


Figure 13 & 14: Plot showing Ray, and Vikran raw vs processed data for absolute acceleration

To check the effectiveness of the smoothing filter that was applied during the pre-processing stage, we plotted smaller time windows to help visualize the filters behaviour. The plots show how noise was significantly reduced while preserving the general signal which confirms that the window size of 8 samples was the correct choice as it provides both noise reduction and signal retention.



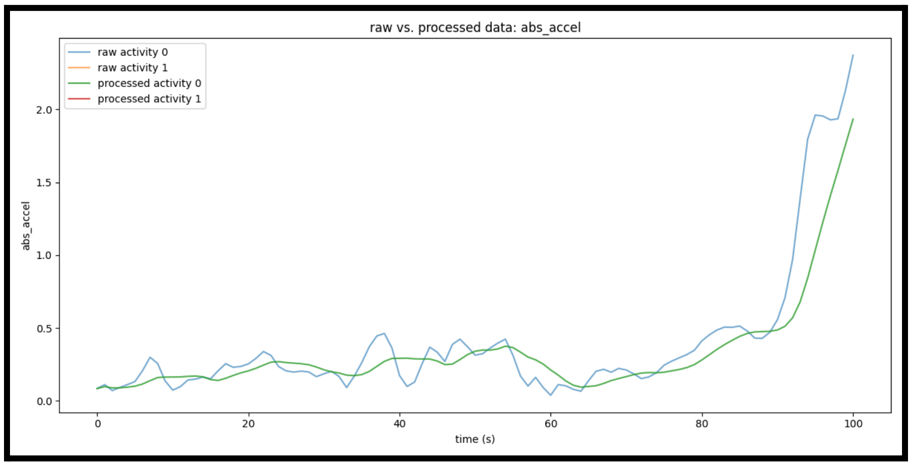
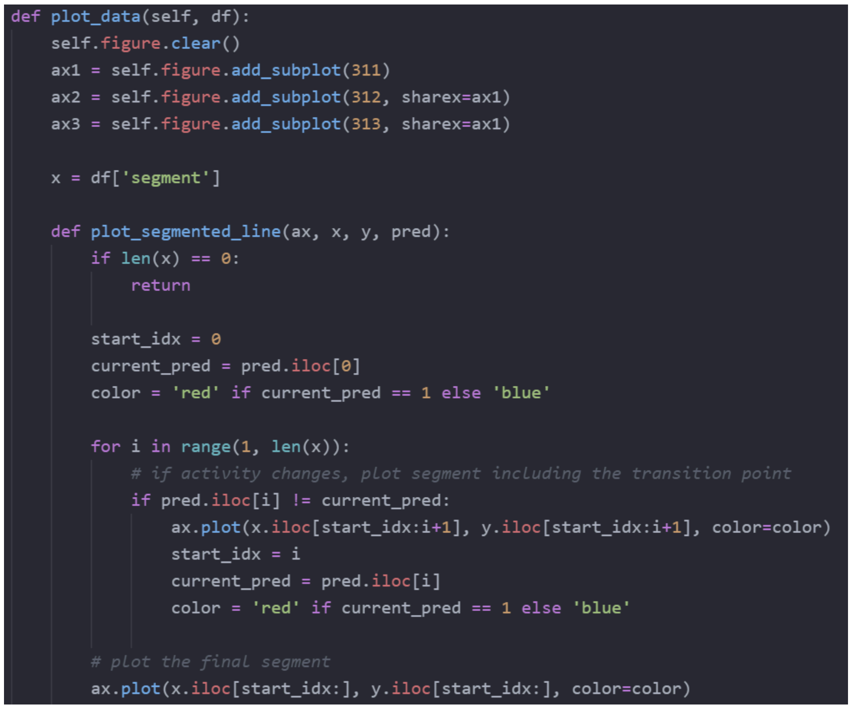
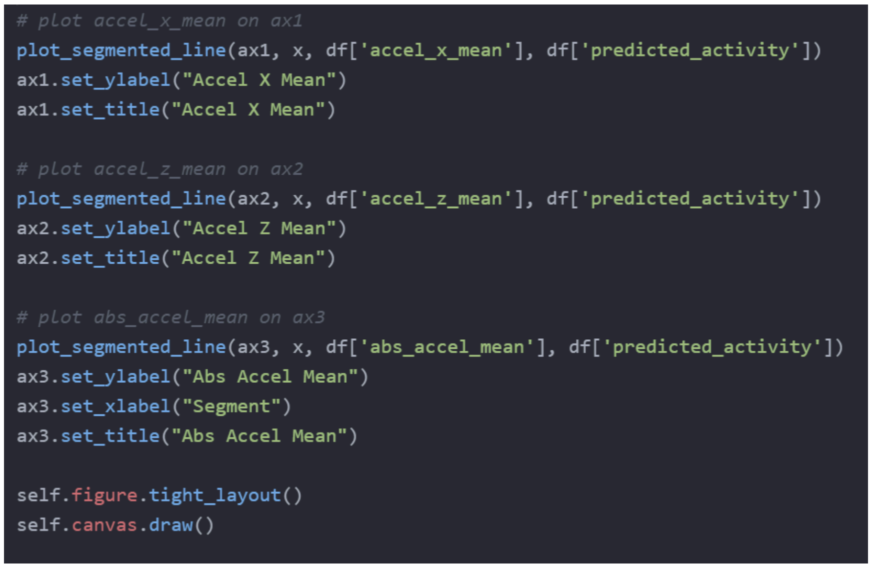


Figure 15, 16 : Plot showing Ray, and Vikran raw vs processed data for absolute in the z plane

To enhance user experience and to provide better feedback, we integrated a visualization feature using PyQtCanvas. The function below named “plot_data” plots segmented and classified walking versus jumping data using colour coded graphs in the GUI. It visualizes the mean x and z acceleration along with the mean absolute acceleration which helps users understand activity over a period of time. Below is the function that performs the visualization as well as an example output of a plot.



```python
def plot_data(self, df):
    self.figure.clear()
    ax1 = self.figure.add_subplot(311)
    ax2 = self.figure.add_subplot(312, sharex=ax1)
    ax3 = self.figure.add_subplot(313, sharex=ax1)

    x = df['segment']

    def plot_segmented_line(ax, x, y, pred):
        if len(x) == 0:
            return

        start_idx = 0
        current_pred = pred.iloc[0]
        color = 'red' if current_pred == 1 else 'blue'

        for i in range(1, len(x)):
            # if activity changes, plot segment including the transition point
            if pred.iloc[i] != current_pred:
                ax.plot(x.iloc[start_idx:i+1], y.iloc[start_idx:i+1], color=color)
                start_idx = i
                current_pred = pred.iloc[i]
                color = 'red' if current_pred == 1 else 'blue'

        # plot the final segment
        ax.plot(x.iloc[start_idx:], y.iloc[start_idx:], color=color)
```



```python
# plot accel_x_mean on ax1
plot_segmented_line(ax1, x, df['accel_x_mean'], df['predicted_activity'])
ax1.set_ylabel("Accel X Mean")
ax1.set_title("Accel X Mean")

# plot accel_z_mean on ax2
plot_segmented_line(ax2, x, df['accel_z_mean'], df['predicted_activity'])
ax2.set_ylabel("Accel Z Mean")
ax2.set_title("Accel Z Mean")

# plot abs_accel_mean on ax3
plot_segmented_line(ax3, x, df['abs_accel_mean'], df['predicted_activity'])
ax3.set_ylabel("Abs Accel Mean")
ax3.set_xlabel("Segment")
ax3.set_title("Abs Accel Mean")

self.figure.tight_layout()
self.canvas.draw()
```

Figure 17 & 18: Visualization functions for use in UI

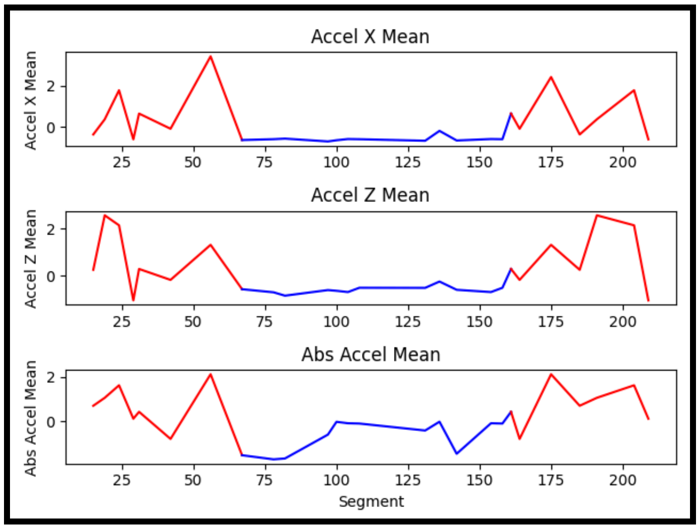
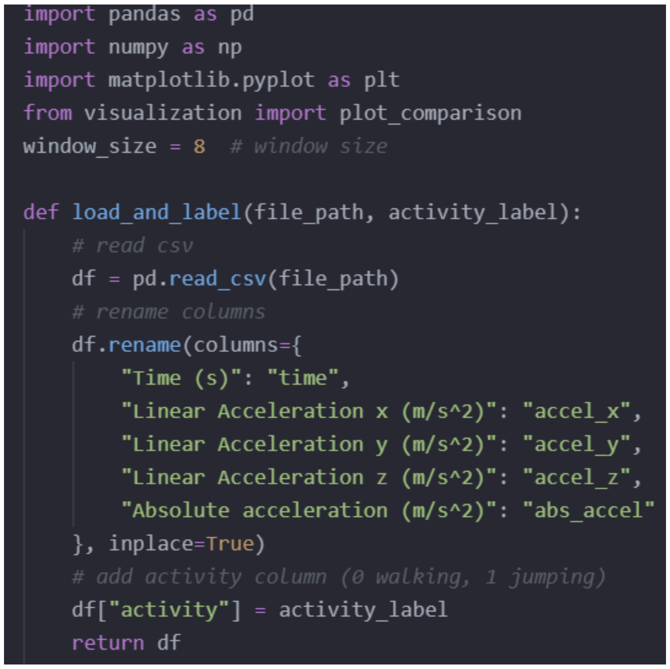


Figure 19: Output of plot_data function in UI

From the visualizations, it is clear to us that the signal clarity varied depending on the phone placement and movement. If we were to redo the data collection, we would aim to reduce the amount of sharp turns to attempt to create a smoother collection process. Overall, the visualization process proved to be a very essential step in validating the quality of the data collected.

4. Preprocessing:

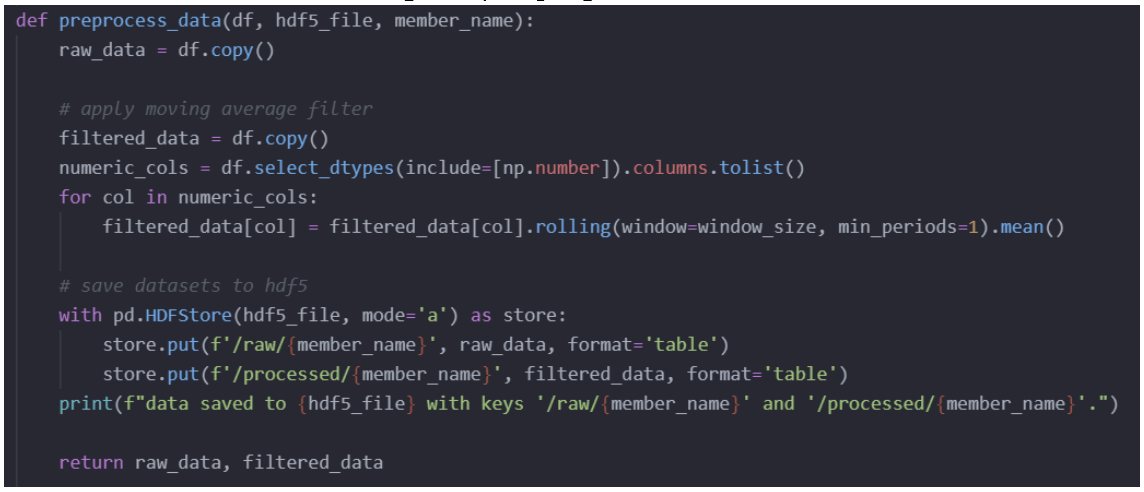
In the preprocessing stage of our project, one of the main focuses was to clean up the raw data that was collected, which would make it easier to use for machine learning. Since smartphones were used to collect the data there is a lot of naturally occurring noise that gets included into the data such as hand movements, phone shakes, or the different ways the phone was positioned. This was dealt with accordingly using a moving average filter which adequately smoothed out the signals and was able to reduce the random fluctuations that occurred in the data.



```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from visualization import plot_comparison
window_size = 8  # window size

def load_and_label(file_path, activity_label):
    # read csv
    df = pd.read_csv(file_path)
    # rename columns
    df.rename(columns={
        "Time (s)": "time",
        "Linear Acceleration x (m/s^2)": "accel_x",
        "Linear Acceleration y (m/s^2)": "accel_y",
        "Linear Acceleration z (m/s^2)": "accel_z",
        "Absolute acceleration (m/s^2)": "abs_accel"
    }, inplace=True)
    # add activity column (0 walking, 1 jumping)
    df["activity"] = activity_label
    return df
```

Figure 20: Code for loading data, and renaming columns

To continue processing the data a function named **preprocess_data()** was called, this function would go through each numeric column in the dataset, through the acceleration values for x, y, z and applied the moving average using the method from pandas **rolling().mean()**. Testing out a couple of different window sizes led to the conclusion that a window size of 8 worked best because it adequately removed a lot of the noise but was able to keep the patterns in motion. When smaller window sizes were tested it did not smooth the signal enough and larger window sizes flattened the signal which made it much harder to see the difference between walking and jumping.



```python
def preprocess_data(df, hdf5_file, member_name):
    raw_data = df.copy()

    # apply moving average filter
    filtered_data = df.copy()
    numeric_cols = df.select_dtypes(include=[np.number]).columns.tolist()
    for col in numeric_cols:
        filtered_data[col] = filtered_data[col].rolling(window=window_size, min_periods=1).mean()

    # save datasets to hdf5
    with pd.HDFStore(hdf5_file, mode='a') as store:
        store.put(f'/raw/{member_name}', raw_data, format='table')
        store.put(f'/processed/{member_name}', filtered_data, format='table')
    print(f"data saved to {hdf5_file} with keys '/raw/{member_name}' and '/processed/{member_name}'.")

    return raw_data, filtered_data
```

Figure 21: Function to process data with moving average filter, window size of 8, store accordingly in file storage format

This difference could clearly be seen after applying the moving average filter, an example of this is when looking at a before and after filtering plot of Vikran's walking data, it is shown that the raw signal is riddled with sharp spikes and much more jagged movements. Meanwhile the processed signal is much smoother and much easier to interpret. It is great as it makes the data more reliable for the following steps such as feature extraction and then training the model.

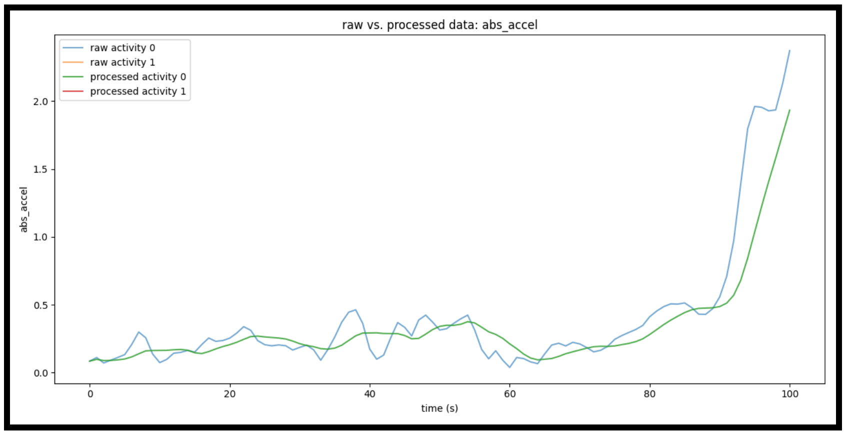
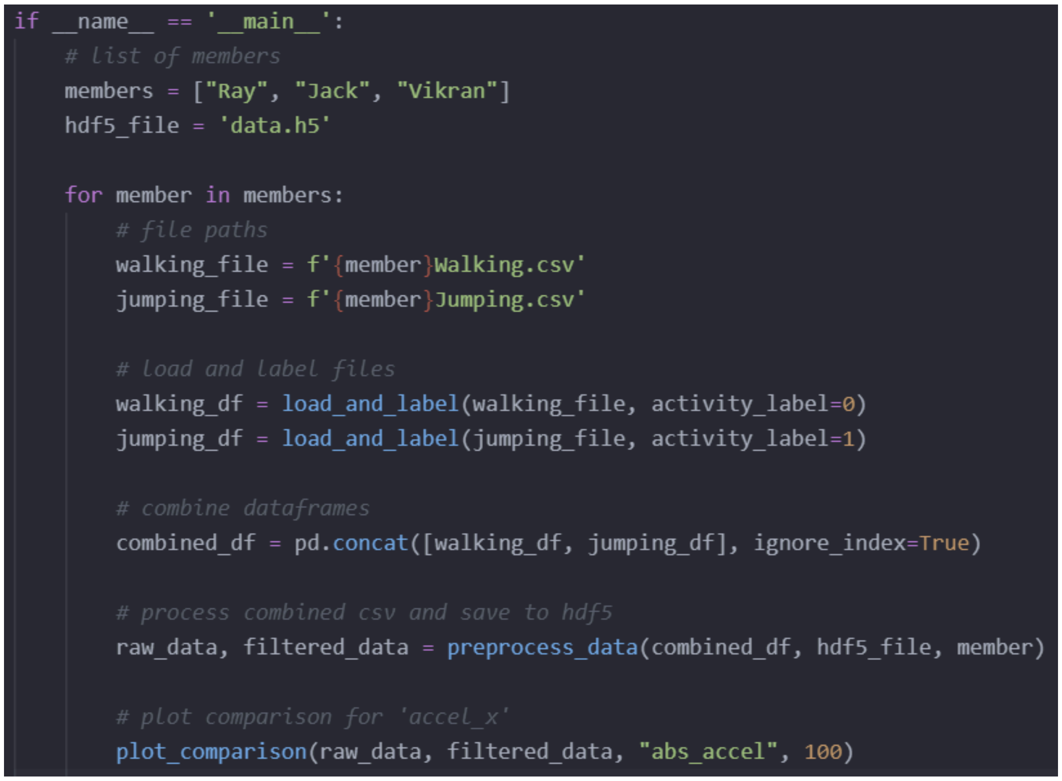


Figure 22: Effects of moving average filter on Vikran's walking data with filter

After the data was filtered, both the raw and processed version were saved in the HDF5 file. Each team member's data was saved under their own specific path "/raw/{member}" for the original data, and "/processed/{member}" for the processed data. This made it much easier to compare the original and processed signal if it is needed.

Overall, the preprocessing step was greatly needed as it allowed the noise affecting the signal to be cleaned and made sure the signals fed into the model could accurately represent walking and jumping patterns. This will help improve the consistency and accuracy of the model making it a much needed step.

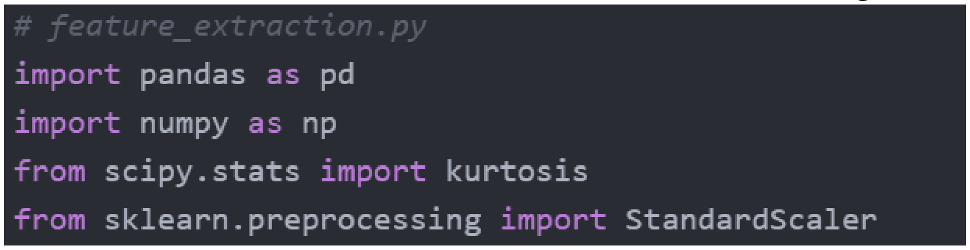


```python
if __name__ == '__main__':
    # list of members
    members = ["Ray", "Jack", "Vikran"]
    hdf5_file = 'data.h5'

    for member in members:
        # file paths
        walking_file = f'{member}Walking.csv'
        jumping_file = f'{member}Jumping.csv'

        # load and label files
        walking_df = load_and_label(walking_file, activity_label=0)
        jumping_df = load_and_label(jumping_file, activity_label=1)

        # combine dataframes
        combined_df = pd.concat([walking_df, jumping_df], ignore_index=True)

        # process combined csv and save to hdf5
        raw_data, filtered_data = preprocess_data(combined_df, hdf5_file, member)

        # plot comparison for 'accel_x'
        plot_comparison(raw_data, filtered_data, "abs_accel", 100)
```

Figure 23 : Main function using the functions mentioned above

5. Feature Extraction & normalization:

For feature extraction, we utilized the following imports; Kurtosis was one of the features we extracted, and the standard scaler was utilized during normalization of the data.



```python
# feature_extraction.py
import pandas as pd
import numpy as np
from scipy.stats import kurtosis
from sklearn.preprocessing import StandardScaler
```

Figure 24: Imports for feature extraction

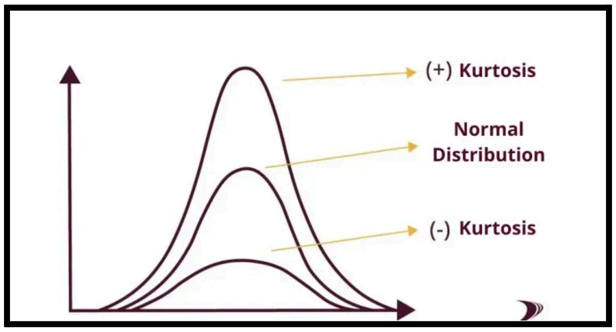
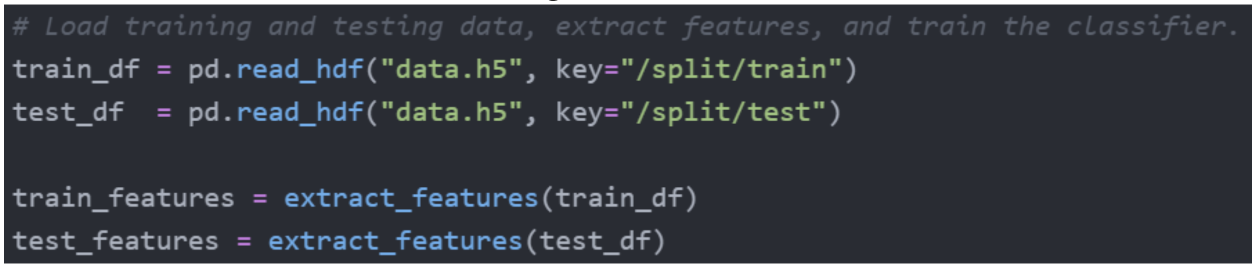


Figure 25: Kurtosis of a graph

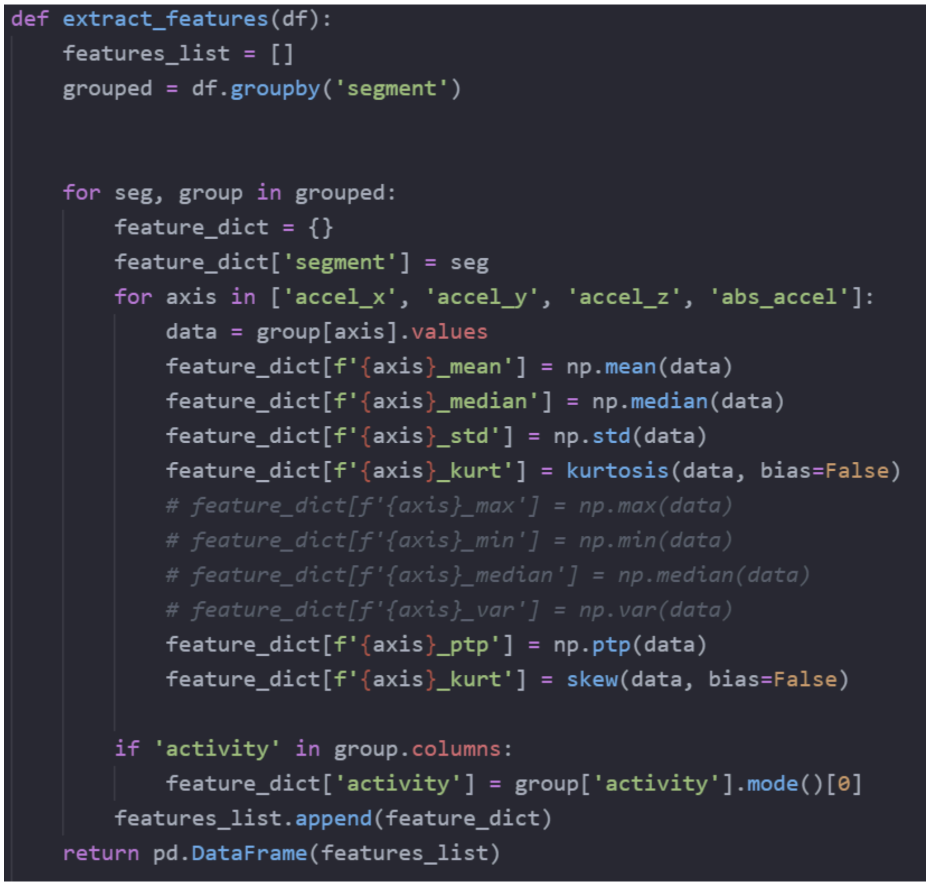
We import training and test data frames from the hdf5 file, and run the **extract_features** function test and training datasets.



```
# Load training and testing data, extract features, and train the classifier.
train_df = pd.read_hdf("data.h5", key="/split/train")
test_df  = pd.read_hdf("data.h5", key="/split/test")

train_features = extract_features(train_df)
test_features = extract_features(test_df)
```

Figure 26: code for importing and using feature extraction function

The **extract_features** function selects a minimum of 10 features; after testing and fitting the model, we found the most efficient set of features to exclude max, min, median, and variance (commented out below).



```
def extract_features(df):
    features_list = []
    grouped = df.groupby('segment')


    for seg, group in grouped:
        feature_dict = {}
        feature_dict['segment'] = seg
        for axis in ['accel_x', 'accel_y', 'accel_z', 'abs_accel']:
            data = group[axis].values
            feature_dict[f'{axis}_mean'] = np.mean(data)
            feature_dict[f'{axis}_median'] = np.median(data)
            feature_dict[f'{axis}_std'] = np.std(data)
            feature_dict[f'{axis}_kurt'] = kurtosis(data, bias=False)
            # feature_dict[f'{axis}_max'] = np.max(data)
            # feature_dict[f'{axis}_min'] = np.min(data)
            # feature_dict[f'{axis}_median'] = np.median(data)
            # feature_dict[f'{axis}_var'] = np.var(data)
            feature_dict[f'{axis}_ptp'] = np.ptp(data)
            feature_dict[f'{axis}_kurt'] = skew(data, bias=False)

        if 'activity' in group.columns:
            feature_dict['activity'] = group['activity'].mode()[0]
        features_list.append(feature_dict)
    return pd.DataFrame(features_list)
```

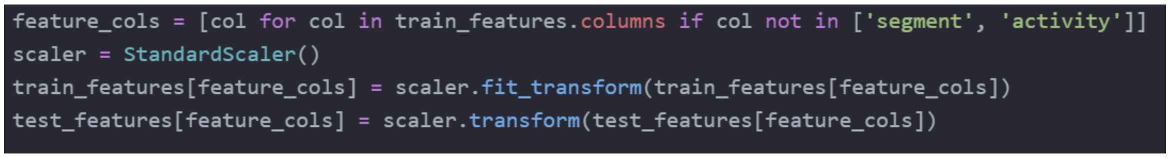
Figure 27: all features extracted, without (max, min, median, and variance)

The following features were extracted from the processed accelerometer data to capture various patterns:

Feature	Description	Reasoning
mean	The average value of the data	Represents the general tendency of the dataset and for understanding overall behavior.
median	The middle value when data is sorted	Represent the general average of the dataset, while avoiding outlier skews.
std (Standard Deviation)	Measures dispersion of data from the mean	Gives an understanding of the variability of the dataset. A higher value means more spread.
kurt (Kurtosis)	Measures the tailedness of the distribution	Identifies whether the data has higher or low peaks compared to a normal distribution.
max	Maximum value in the dataset	For detecting peaks and extreme values.
min	Minimum value in the dataset	For understanding the lowest point of the values.
var (Variance)	The average of squared deviations from the mean	Measures dispersion like standard deviation but without taking the square root.
ptp (Peak-to-Peak)	Range of data (max - min)	Useful for identifying the range of values in a dataset.
skew (Skewness)	Measures asymmetry in data distribution	Helps determine whether the data is symmetrically distributed or biased towards one side.

Table 1: List of features extracted

After features are extracted, the dataset of features are normalized using StandardScaler from sklearn.preprocessing. First, the relevant features are identified by excluding the 'segment' and 'activity' columns, as these should not be normalized. Then, I apply StandardScaler to standardize the numerical features in the training set, ensuring they have a mean of 0 and a standard deviation of 1. I fit the scaler on the training data and then apply the same transformation to the test set.



```
feature_cols = [col for col in train_features.columns if col not in ['segment', 'activity']]
scaler = StandardScaler()
train_features[feature_cols] = scaler.fit_transform(train_features[feature_cols])
test_features[feature_cols] = scaler.transform(test_features[feature_cols])
```

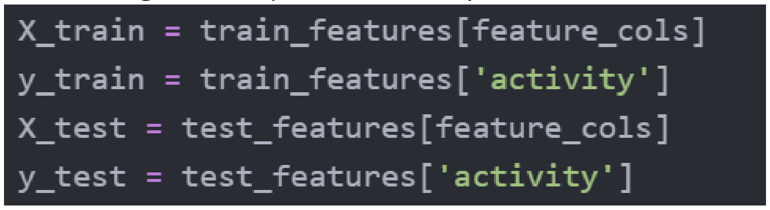
Figure 28: Normalization of data with standard scaler

With the **train_features** and **test_features** dataset, we take out target and feature columns so the data is ready for training the model, as we will see in the next section.

6. Training and testing the classifier:

In order to train the logistic regression model, the extracted features were split accordingly:

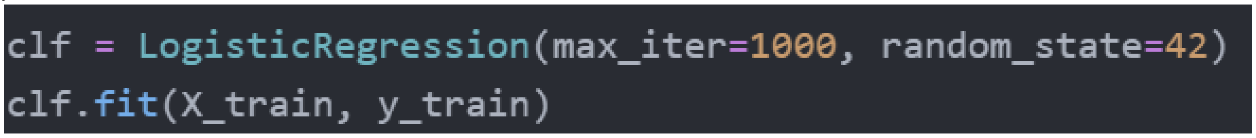
- Training data features extracted into **X_train**
- Target variable column 'activity' was extracted into **y_train**
- Testing data was by extracting the same features into **X_test**
- Target activity column into **y_test**



```
X_train = train_features[feature_cols]
y_train = train_features['activity']
X_test = test_features[feature_cols]
y_test = test_features['activity']
```

Figure 29: split data for model training and testing

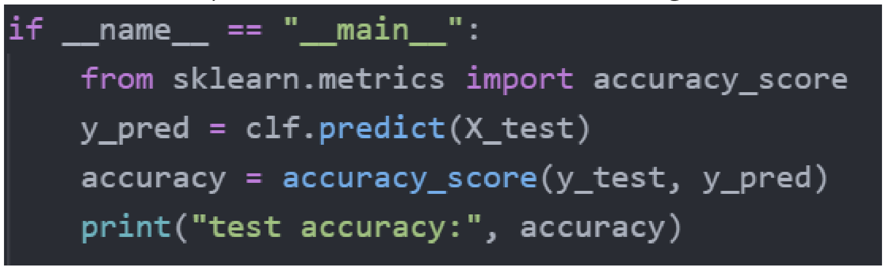
The logistic regression classifier was instantiated using scikit-learn's LogisticRegression class with parameters **max_iter**=1000 and **random_state**=42. The parameter **max_iter** was set to 1000 to ensure convergence of the model, ensuring the model's weights converge. The parameter **random_state** was chosen to be 42 to reproduce the results during testing. The model was then trained with the training data (**X_train** and **y_train**)



```
clf = LogisticRegression(max_iter=1000, random_state=42)
clf.fit(X_train, y_train)
```

Figure 30: Training the model

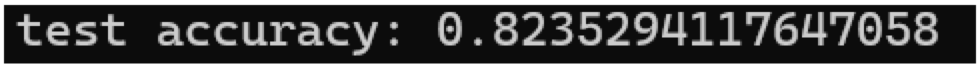
Predictions were generated with the test features (**X_test**). Accuracy of the logistic regression model was then tested by comparing the predicted labels (**y_pred**) against the true labels (**y_test**) using the accuracy score metric. After testing various combinations of features, we found the most accurate combination of features to exclude (max, min, var, median) which you can see commented out during feature selection.



```
if __name__ == "__main__":
    from sklearn.metrics import accuracy_score
    y_pred = clf.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred)
    print("test accuracy:", accuracy)
```

Figure 31: Model testing and displaying accuracy

The final model achieved an accuracy of 0.82, indicating a strong and reliable performance on the test data.



```
test accuracy: 0.8235294117647058
```

Figure 32: Accuracy output

7. Model deployment:

The desktop application was developed using Python and PyQt5 to create an intuitive, user-friendly interface that integrates the trained model for activity analysis. Upon launching, the app displays an empty graphical user interface (GUI) featuring a simple "Upload CSV" button, a label indicating "No file selected," and an empty plotting area.

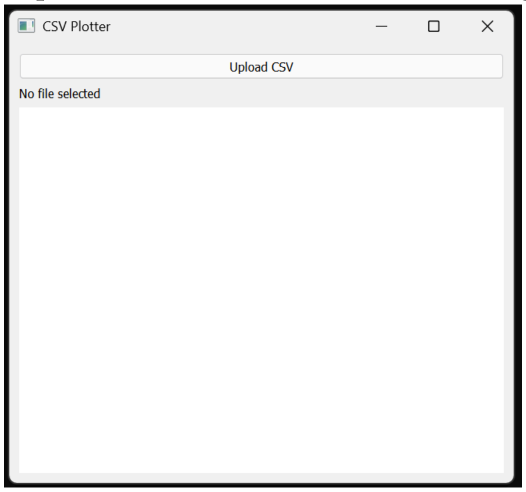


Figure 33: Empty UI layout

This design was intentionally chosen to provide a clean and simple UI that processes, analyzes, and labels the data automatically. Once a CSV file is uploaded, the app calls the **load_csv** function which encapsulates all preprocessing and prediction functions from previous sections.

```
def load_csv(self):
    file_path, _ = QFileDialog.getOpenFileName(self, "Open CSV", "", "CSV Files (*.csv)")
    if file_path:
        self.csv_path.setText(file_path)
```

To process the data, the following imports are utilized:

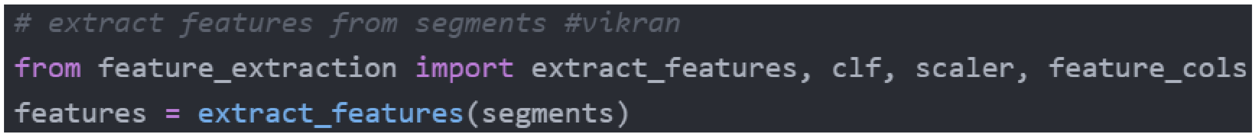
- Loading and relabeling columns (from preprocessing)

```
from process_data import load_and_label
df = load_and_label(file_path)
```

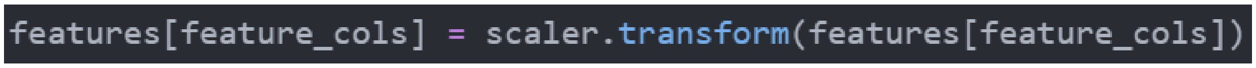
- Segmenting the data into segments of 5 second windows (from preprocessing and storage)

```
# segment data
from storage import segment_signal #jack
segments = pd.concat(segment_signal(df))
print("CSV loaded:\n", segments)
```

- Extracting features and scaling the data (from feature extraction)

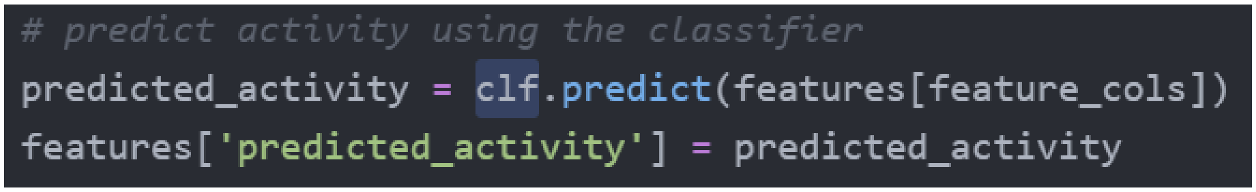


```python
# extract features from segments #vikran
from feature_extraction import extract_features, clf, scaler, feature_cols
features = extract_features(segments)
```



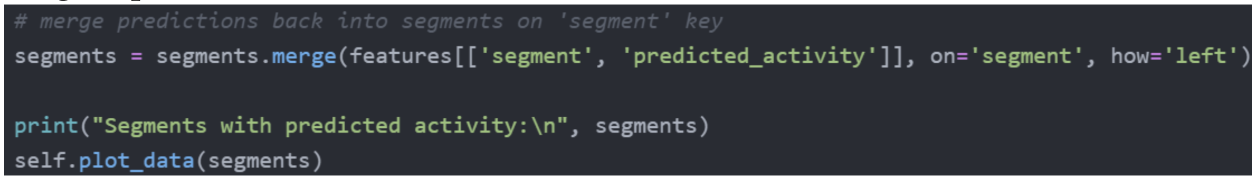
```python
features[feature_cols] = scaler.transform(features[feature_cols])
```

- Finally predicting activities using the imported linear regression model (clf model was trained in **feature_extraction** file)



```python
# predict activity using the classifier
predicted_activity = clf.predict(features[feature_cols])
features['predicted_activity'] = predicted_activity
```

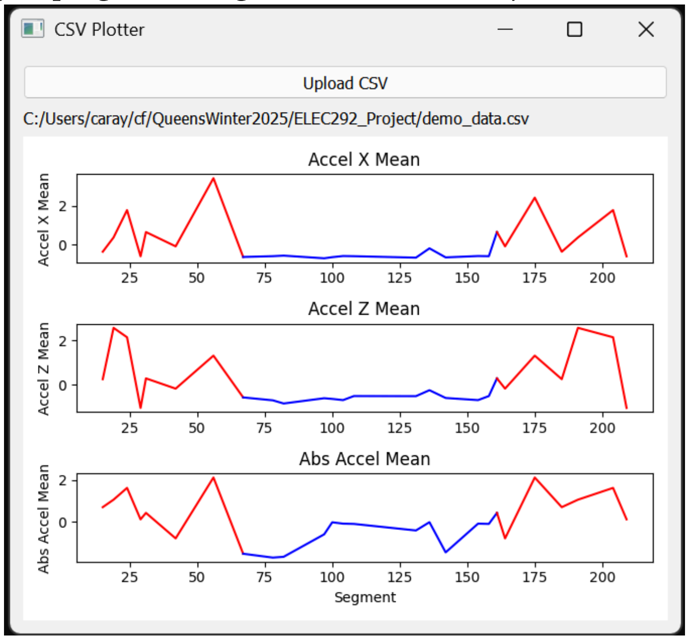
- The resulting predictions are merged back into the segmented data and visualized using the **plot_data** function (from visualization)



```python
# merge predictions back into segments on 'segment' key
segments = segments.merge(features[['segment', 'predicted_activity']], on='segment', how='left')

print("Segments with predicted activity:\n", segments)
self.plot_data(segments)
```

Figures 34-40: functions to process and segment data on upload

This automatically populates the UI with color-coded graphs (red for walking and blue for jumping) ensuring that users can easily visualize the model's output.



CSV Plotter

Upload CSV

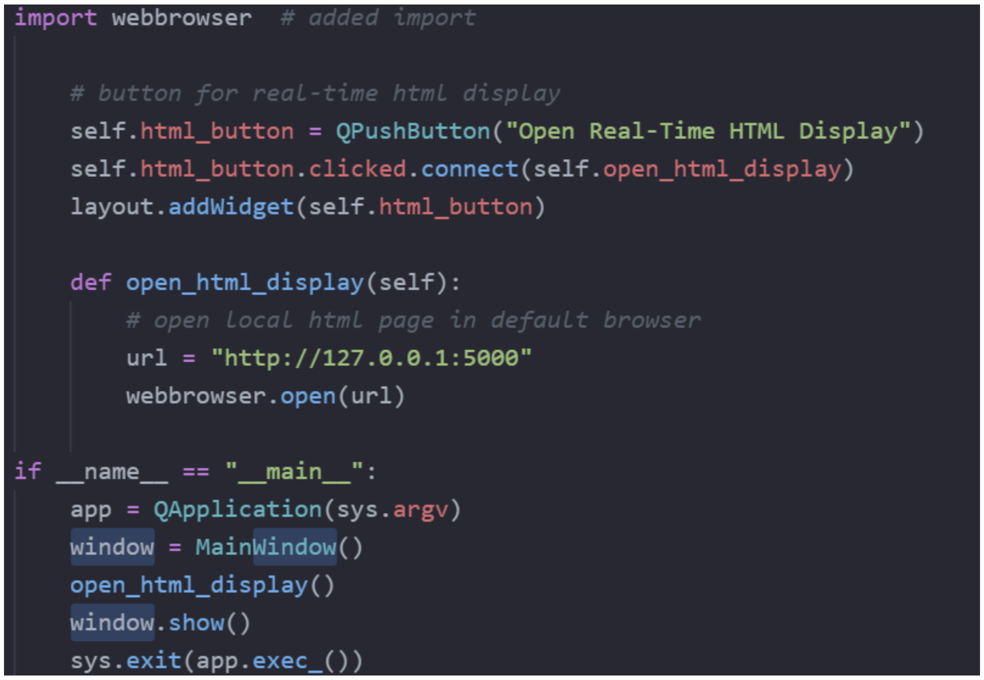
C:/Users/caray/cf/QueensWinter2025/ELEC292_Project/demo_data.csv

Figure 41: Populated and labeled UI

This design and clear visual feedback were key considerations in enhancing both functionality and user experience.

8. Bonus:

For the bonus component of our project, we set out to extend the desktop application's functionality to classify activities from a smartphone in real-time. The envisioned workflow was that the smartphone, using the 'Enable remote access' option of the Phyphox app, would stream its accelerometer data to our desktop application. In turn, the app would pre-process the incoming data and run it through our trained classifier to immediately display the predicted activity (e.g., 'walking'). This approach required substantial changes to our pre-processing and classification pipeline, as well as modifications to the GUI to support dynamic, real-time updates.



```
import webbrowser  # added import

    # button for real-time html display
    self.html_button = QPushButton("Open Real-Time HTML Display")
    self.html_button.clicked.connect(self.open_html_display)
    layout.addWidget(self.html_button)

    def open_html_display(self):
        # open local html page in default browser
        url = "http://127.0.0.1:5000"
        webbrowser.open(url)

if __name__ == "__main__":
    app = QApplication(sys.argv)
    window = MainWindow()
    open_html_display()
    window.show()
    sys.exit(app.exec_())
```

Figure 42: App modifications for real time data processing

In an effort to integrate real-time data visualization alongside our offline CSV-based processing, we added a new button within the PyQt application to launch a local HTML page in the user's default browser. By importing the webbrowser module and creating an open_html_display function, we aimed to bridge the gap between static file loading and dynamic data feeds. Although our current setup still relies on an external server or script (e.g., Flask) to collect and serve live accelerometer data, this step demonstrates how the app would connect to a real-time web interface.

9. Participation Report

Code	**Ray**	**Jack**	**Vikran**
Data Collection	X	X	X
Data Storage & Pre-processing			X
Data Visualization		X	
Feature Extraction & Normalization	X	X	X
Training & Testing the Classifier	X		
Desktop App Development	X		

Report	**Ray**	**Jack**	**Vikran**
Data Collection		X	
Data Storage			X
Data Visualization		X	
Pre-processing			X
Feature Extraction & Normalization	X	X	X
Training & Testing the Classifier	X		
Desktop App Development	X		
Revisions	X	X	X
Video Recording	X	X	X
Video Editing		X	
Bonus	X	X	X