ELEC 390 Final Project

Course: ELEC 390

Project Report

Group 49

Daniel Sa, 20154580, 18vds1@queensu.ca

Bradley Stephen, 20207843, 19bbs2@queensu.ca

Josh Allen, 20216289, 19jwda@queensu.ca

April 14th, 2023

Data Collection

The study utilized the "Phypox" mobile application to acquire walking and jumping data. To ensure data diversity, the phone was placed in both the right and left pockets for each trial. The process was as follows; The phypox app was loaded up on a smartphone and the acceleration without gravity option was selected. Once selected the play button was pressed to commence the collection of data and the smartphone was placed in the left pocket of the user. The user then walked in a straight line attempting to avoid and sudden movements, obstacles, and elevated surfaces (Stairs). After roughly 2 minutes the smartphone was removed from the user's pocket and the accelerator was paused. The data was then saved as “Walking Left”, and the same process was repeated for walking with the smartphone in the right pocket. The Same process was used when collecting jumping data however the user would ensure to have minimal horizontal movement. Each member collected data for more than five minutes, resulting in three distinct subsets.

During the data collection phase, the research team encountered various challenges. One significant issue pertained to the variability in the surfaces used for walking data collection. The team came across numerous elevated surfaces such as stairs, which may have inadvertently contributed to the "jumping" effect in the data. Furthermore, the phone exhibited considerable pocket movement, which was independent of the participant's motion, thereby potentially compromising data accuracy.

The exact hardware used during this process can be seen using the metadata that was exported along with the acceleration data from Phypox, these ca be seen in Table 1, Table 2, and Table 3. It should be noted that Android devices provided much more metadata compared to Apple devices. With this in mind, ‘null’ values for Android metadata has been removed to save on space.

Table 1: Metadata for Josh.

property	value
version	1.1.11
build	10011
fileFormat	1.15
deviceModel	iPhone10,4
deviceBrand	,"Apple"
deviceBoard	
deviceManufacturer	
deviceBaseOS	
deviceCodename	
deviceRelease	16.3.1
depthFrontSensor	0
depthFrontResolution	
depthFrontRate	
depthBackSensor	0
depthBackResolution	

depthBackRate	

Table 2: Metadata for Bradley.

property	value
version	1.1.11
build	10011
fileFormat	1.15
deviceModel	iPhone12,1
deviceBrand	,"Apple"
deviceBoard	
deviceManufacturer	
deviceBaseOS	
deviceCodename	
deviceRelease	16.2
depthFrontSensor	1
depthFrontResolution	
depthFrontRate	
depthBackSensor	0
depthBackResolution	
depthBackRate	

Table 3: Metadata for Daniel.

property	Value
version	,"1.1.11"
build	,"1011102"
fileFormat	,"1.15"
deviceModel	,"SM-G781W"
deviceBrand	,"samsung"
deviceBoard	,"kona"
deviceManufacturer	,"samsung"
deviceBaseOS	,"samsung/r8qcsx/r8q:13/TP1A.220624.014/G781WVLU7GVK6:user/release-keys"
deviceCodename	,"REL"
deviceRelease	,"13"
depthFrontSensor	,"0"
depthBackSensor	,"0"
depthBackRate	
accelerometer Name	,"LSM6DSO Accelerometer"
accelerometer Vendor	,"STMicro"

accelerometer Range	,"78.4532"
accelerometer Resolution	,"0.0023928226"
accelerometer MinDelay	,"2404"
accelerometer MaxDelay	,"1000000"
accelerometer Power	,"0.17"
accelerometer Version	,"15932"
linear_acceleration Name	,"linear_acceleration"
linear_acceleration Vendor	,"qualcomm"
linear_acceleration Range	,"156.99008"
linear_acceleration Resolution	,"0.01"
linear_acceleration MinDelay	,"5000"
linear_acceleration MaxDelay	,"200000"
linear_acceleration Power	,"0.515"
linear_acceleration Version	,"1"
gravity Name	,"gravity Non-wakeup"
gravity Vendor	,"qualcomm"
gravity Range	,"156.99008"
gravity Resolution	,"0.01"
gravity MinDelay	,"5000"
gravity MaxDelay	,"200000"
gravity Power	,"0.515"
gravity Version	,"1"
gyroscope Name	,"LSM6DSO Gyroscope"
gyroscope Vendor	,"STMicro"
gyroscope Range	,"17.452517"
gyroscope Resolution	,"6.1084726E-4"
gyroscope MinDelay	,"2404"
gyroscope MaxDelay	,"1000000"
gyroscope Power	,"0.55"
gyroscope Version	,"15932"
magnetic_field Name	,"AK09918 Magnetometer"
magnetic_field Vendor	,"akm"
magnetic_field Range	,"4912.0503"
magnetic_field Resolution	,"0.15"
magnetic_field MinDelay	,"10000"
magnetic_field MaxDelay	,"1000000"
magnetic_field Power	,"1.1"
magnetic_field Version	,"146953"
light Name	,"stk_stk31610 Ambient Light Sensor Non-wakeup"

light Vendor	,"sensortek"
light Range	,"4095.9001"
light Resolution	,"0.54"
light MinDelay	,"0"
light MaxDelay	,"0"
light Power	,"0.09"
light Version	,"1000"
proximity Name	,"Ear Hover Proximity Sensor (ProToS)"
proximity Vendor	,"Samsung Electronics."
proximity Range	,"5.0"
proximity Resolution	,"1.0"
proximity MinDelay	,"0"
proximity MaxDelay	,"0"
proximity Power	,"0.75"
proximity Version	,"1"
attitude Name	,"Rotation Vector Non-wakeup"
attitude Vendor	,"qualcomm"
attitude Range	,"1.0"
attitude Resolution	,"0.01"
attitude MinDelay	,"5000"
attitude MaxDelay	,"200000"
attitude Power	,"1.415"
attitude Version	,"1"

Data Storing

Each subset of data from each group member was segregated into four folders, named "Walking Right," "Walking Left," "Jump Left," and "Jump Right.". Each of these folders contained the acceleration data, along with its metadata in the form of CVS files. These CVS files were uploaded to a group GitHub repository under an enclosed folder named “Data”. Afterwards, an h5 file is used to structure the data as specified in the project instructions. Each team member is given their own folder, which holds the original data used with an added column “WalkingJumping” which specifies the action perform in each dataset (to be used when training the model).

The python program then imports this data, applies preprocessing, divide the data into 5 second windows, concatenate all the jumping and walking windows together, shuffle the data around, and the split the data into training and testing sets with a 90:10 ratio. Once the training and testing sets are created, they are then stored into the h5 file in their own respective directories along with their labels. The full layout of this h5 can be seen in Figure 1 and can also be found in hdf5_groups.h5.

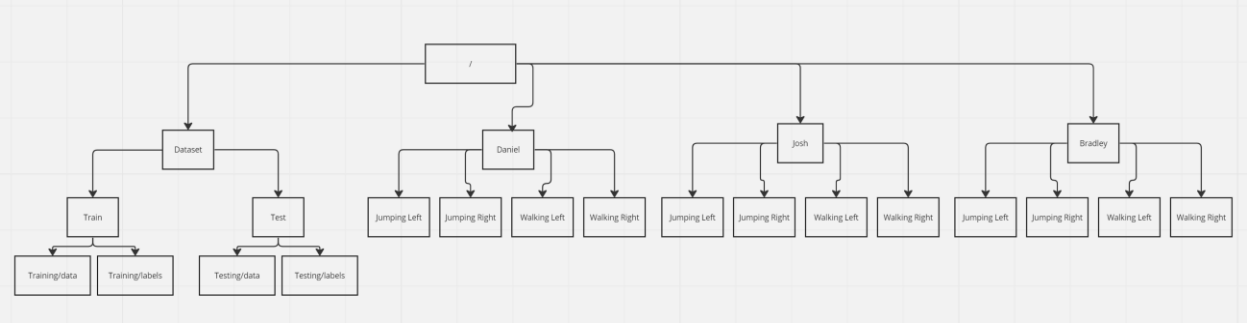

Figure 1: Structure of the h5 file that holds the needed data.

Visualization

Although many versions of each plot were created for various data collections, a single image for each visualization technique is displayed as seen in Figure 2 to describe the general case. The analysis of acceleration data for jumping revealed several correlations between acceleration values in different directions during various phases of the jump. During the upward phase of the jump, positive correlations were observed between the acceleration values in the Z direction and the X and Y directions. These correlations reflect the coordinated upward movement of the body. Similarly, during the downward phase of the jump, similar positive correlations were observed between the acceleration values in the Z direction and the X and Y directions, reflecting the coordinated downward movement of the body.

During the landing phase of the jump, negative correlations were observed between the acceleration values in the Z direction and the X and Y directions. These correlations reflect the deceleration of the body as it lands on the ground. Finally, during the rest phase between jumps, weak or no correlations were observed between the acceleration values in different directions. This reflects the relatively stable position of the body during this phase.

By identifying these correlations, it is possible to select the most important features for distinguishing between jumping and other activities, such as walking. This information can then be used to develop effective machine learning models for classification purposes. Furthermore, the observed correlations can provide valuable insights into the movement patterns and dynamics of the body during jumping. This information can be used to identify areas for improvement in training and technique.

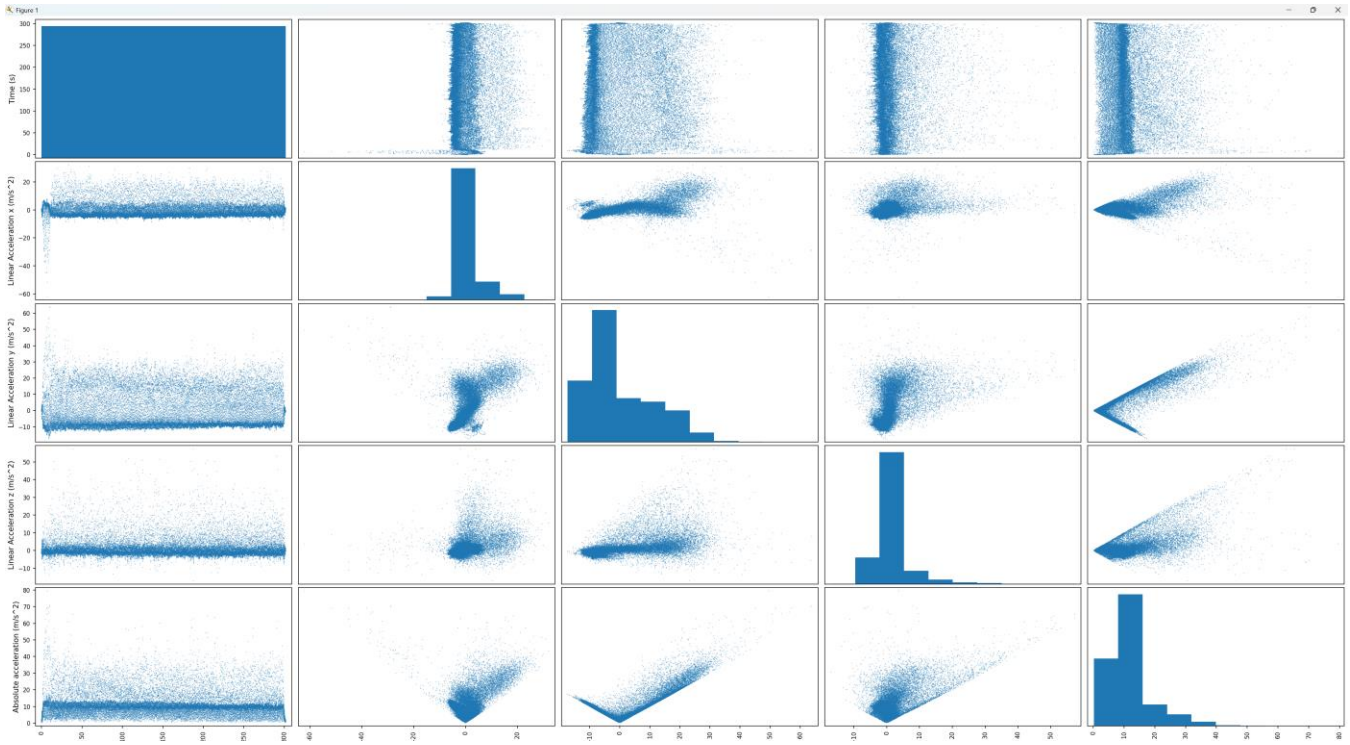

Figure 2: Scatter Matrix Plot for visualization of the raw CSV jumping data.

A standard line chart of normalized x-acceleration as seen in Figure 3 over time for walking data visualization reveals a repeating pattern of positive and negative values. The cyclic nature of the walking movement involves the repetitive sequence of lifting a foot, swinging it forward, and placing it back on the ground. As a result, this movement pattern generates a periodic oscillation of acceleration values in the x-direction.

In this visualization, we can observe a correlation between the cyclical pattern of the x-acceleration values and time. Positive values correspond to the acceleration during the forward movement of the leg, while negative values correspond to the deceleration during the backward movement of the leg. By analyzing this pattern, we can identify the frequency of the walking cycle and the magnitude of the acceleration values, which can be used to distinguish walking from other activities such as running or jumping.

As seen In Figure 3, Figure 4, Figure 5 the standard line charts for x, y, and z axis appear the same, it may indicate a uniform movement pattern in all directions. Such uniformity suggests that the body moves in a linear or steady pattern without significant changes in direction or acceleration.

For instance, when data is collected from a person who is sitting in a chair, the acceleration values tend to remain constant in all directions, which results in similar line charts for the x, y, and z axes. Similarly, when data is collected while the person is standing still, the acceleration values may fluctuate slightly due to postural changes or minor movements but remain relatively constant in all directions.

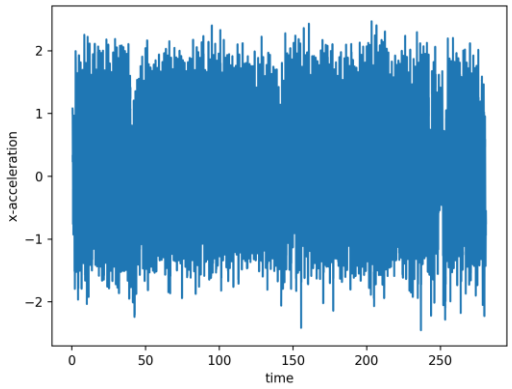

Figure 3: Standard line chart of normalized x-acceleration over time for walking data

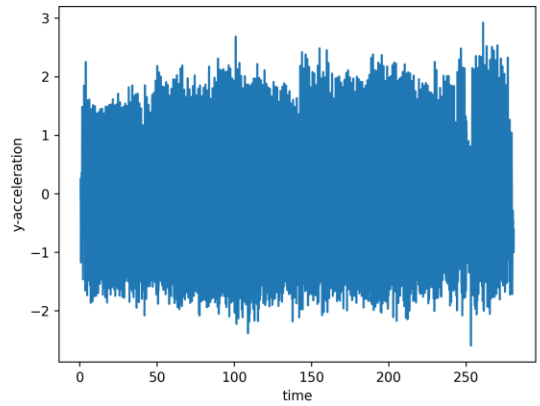

Figure 4: Standard line chart of normalized y-acceleration over time for walking data

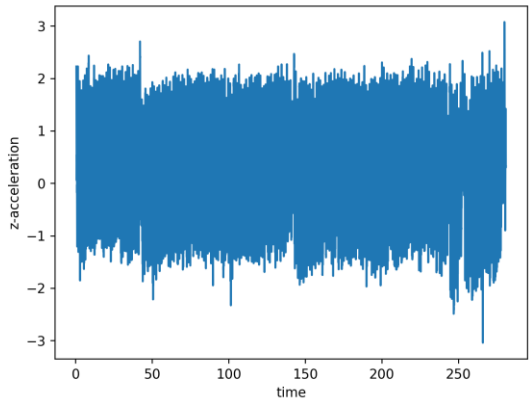

Figure 5: Standard line chart of normalized z-acceleration over time for walking data

Preprocessing

The team decided to take the approach of applying preprocessing to all datasets independently before splitting the data into 5 second windows, concatenating them together, and shuffling the windows to be split into training and testing sets. For every data frame passed into the function "SmoothNormalize", a moving average filter would be applied, and then the data would be normalized as specified in the project instructions.

For the moving average filter, the team played around with several different window sizes, originally starting at 5 and eventually getting to a final value of 50. With smaller window sizes, the team found that the accuracy of the model was significantly worse, with values at around the 50-60% range, as seen in Figure 6. With a window size of 50, the team saw great improvement with the model's performance, obtaining accuracies in the range of 85-95%, as seen in Figure 7.

During the preprocessing phase, the team was careful to only apply the moving average filter and normalization to the acceleration data. Columns "Time" and "WalkingJumping" were temporarily removed, and then added back in afterwards. This process itself had to be carefully done, as when applying the moving average filter, the first few rows of data would become "NaN", It was decided that these values would be dropped, instead of filled in by some algorithm. With this data being shortened, the two columns that were removed would have to be adjusted in size before being added back into the data frame. The python code for the entire preprocessing section can be seen in Figure 8.

```
Accuracy: 54.76%
Recall:  0.29411764705882354
AUC:  0.5270588235294118
F1 Score:  0.3448275862068966
Prediction Output:
[1. 0. 0. 0. 0. 1. 0. 0. 1. 1. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 1. 0. 0.
 0. 0. 0. 0. 0. 0. 0. 1. 0. 1. 1. 0. 0. 1. 0. 0. 1. 1.]
```

Figure 6: Final results when using a window size of 5 for the moving average filter.

```
Accuracy: 87.80%
Recall:  0.8181818181818182
AUC:  0.8875598086124401
F1 Score:  0.8780487804878049
Prediction Output:
[1. 1. 0. 0. 0. 1. 0. 0. 1. 1. 1. 1. 1. 0. 1. 0. 0. 1. 0. 0. 0. 0. 0. 0.
 1. 0. 0. 0. 1. 1. 1. 1. 1. 0. 1. 0. 1. 0. 0. 0. 1.]
```

Figure 7: Final results when using a window size of 50 for the moving average filter.

```python
def SmoothNormalize(dataframe):
    otherColumns = dataframe[['Time (s)', 'WalkingJumping']]
    data = dataframe[['Linear Acceleration x (m/s^2)', 'Linear Acceleration y (m/s^2)', 'Linear Acceleration z (m/s^2)', 'Absolute acceleration
    (m/s^2)']]
    data_smoothed = data.rolling(window=50).mean().dropna() #moving average filter
    data_normalized = scaler.fit_transform(data_smoothed) #normalize
    df_smoothed_normalized = pd.DataFrame(data=data_normalized, columns=data.columns) #add time and WalkingJumping column back in
    df_smoothed_normalized[['Time (s)', 'WalkingJumping']] = otherColumns[len(data) - len(data_smoothed):] #add time and WalkingJumping column
    back in
    return df_smoothed_normalized

jump1Smooth = SmoothNormalize(G1Data1).dropna()
jump2Smooth = SmoothNormalize(G1Data2).dropna()
jump3Smooth = SmoothNormalize(G1Data3).dropna()
jump4Smooth = SmoothNormalize(G1Data4).dropna()
jump5Smooth = SmoothNormalize(G1Data5).dropna()
jump6Smooth = SmoothNormalize(G1Data6).dropna()
walk1Smooth = SmoothNormalize(G2Data1).dropna()
walk2Smooth = SmoothNormalize(G2Data2).dropna()
walk3Smooth = SmoothNormalize(G2Data3).dropna()
walk4Smooth = SmoothNormalize(G2Data4).dropna()
walk5Smooth = SmoothNormalize(G2Data5).dropna()
walk6Smooth = SmoothNormalize(G2Data6).dropna()
```

Figure 8: Python code for preprocessing.

Feature extraction

The features extracted were Minimum, maximum, range, mean, median, variance, skewness, standard deviation, and kurtosis for each x, y, z, and absolute acceleration data columns and were used to train the classifier.

- Minimum and maximum: These features represent the lowest and highest values of acceleration in the data. For walking, the acceleration values will generally be lower compared to running. Therefore, if the minimum and maximum values are relatively low, it may indicate that the person is walking.

- Range: The range is the difference between the maximum and minimum values. It can give an idea of the variability in the acceleration data. In general, running involves larger variations in acceleration compared to walking, so a higher range could indicate running.

- Mean and median: These features represent the central tendency of the data. Walking and running have different acceleration patterns, so their mean and median values will be different. The mean is sensitive to outliers, while the median is not. Therefore, if there are any extreme values in the acceleration data, the median may be a better measure of central tendency.

- Variance: This feature measures the spread of the data around the mean. Running involves larger variations in acceleration compared to walking, so a higher variance could indicate running.

- Skewness: This feature measures the degree of asymmetry in the data distribution. Running and walking have different acceleration patterns, so their distributions may be

skewed in different ways. For example, running may have a positive ske , while walking may have a negative.

- Standard deviation (SD): In the context of acceleration data, the SD of the acceleration values can be a useful feature to distinguish between walking and jumping. Walking involves relatively stable acceleration values with less variation, resulting in a lower SD compared to jumping. Jumping, on the other hand, involves sudden changes in acceleration values, resulting in a higher SD compared to walking. By using SD as a feature in a training model, the model can learn to distinguish between the relatively stable acceleration values of walking and the sudden changes in acceleration values of jumping. This can improve the accuracy of the model in classifying the two activities.

- Root mean square (RMS): RMS acceleration represents the overall magnitude of acceleration. Running involves higher overall acceleration compared to walking, so a higher RMS acceleration could indicate running.

- Kurtosis: Walking typically produces a distribution of acceleration values that is more bell-shaped or Gaussian, with a relatively low kurtosis. Jumping, on the other hand, produces a distribution of acceleration values that is more peaked or "heavy-tailed", with a higher kurtosis. By using kurtosis as a feature in a training model, the model can learn to distinguish between the bell-shaped distribution of acceleration values of walking and the peaked distribution of acceleration values of jumping. This can improve the accuracy of the model in classifying the two activities.

The chosen features were extracted from each window data frame, and were then all stored into a separate data frame which would then be used for the classifier. This process was done for both the training and testing sets of data. Seen in Figure 9 is the python code showing each of the features extracted.

```python
def extract_features(window):      You, yesterday • Model runn
    features = []
    x_acc = window["Linear Acceleration x (m/s^2)"]
    y_acc = window["Linear Acceleration y (m/s^2)"]
    z_acc = window["Linear Acceleration z (m/s^2)"]
    abs_acc = window["Absolute acceleration (m/s^2)"]
    features.append(np.min(x_acc))
    features.append(np.min(y_acc))
    features.append(np.min(z_acc))
    features.append(np.max(x_acc))
    features.append(np.max(y_acc))
    features.append(np.max(z_acc))
    features.append(np.max(x_acc)-np.min(x_acc))
    features.append(np.max(y_acc)-np.min(y_acc))
    features.append(np.max(z_acc)-np.min(z_acc))
    features.append(np.mean(x_acc))
    features.append(np.mean(y_acc))
    features.append(np.mean(z_acc))
    features.append(np.median(x_acc))
    features.append(np.median(y_acc))
    features.append(np.median(z_acc))
    features.append(np.var(x_acc))
    features.append(np.var(y_acc))
    features.append(np.var(z_acc))
    features.append(skew(x_acc))
    features.append(skew(y_acc))
    features.append(skew(z_acc))
    features.append(np.std(x_acc))
    features.append(np.std(y_acc))
    features.append(np.std(z_acc))
    features.append(np.sqrt(np.mean(np.square(x_acc))))
    features.append(np.sqrt(np.mean(np.square(y_acc))))
    features.append(np.sqrt(np.mean(np.square(z_acc))))
    features.append(kurtosis(x_acc))
    features.append(kurtosis(y_acc))
    features.append(kurtosis(z_acc))
    #absolute acceleration features (was added in afterwards)
```

Figure 9: Selected features shown in the python code, for the x, y, and z acceleration data columns.

Training the Classifier

Seen in Figure 10 is the complete python code for training the classifier once the features have been extracted from both the training and testing sets. The team decided that the classifier would use both a standard scalar and a logistic regression with a max iteration value of 10000. The classifier is then created and trained using the training set along with its labels. As mentioned earlier, the classifier uses the features that were extracted from the earlier phase. Once training is completed, a prediction is made using the testing set, which then allows the performance of the model to be determined. For the model to be used in the GUI desktop app, it is dumped to a joblib file to be used later. A sample test run was used to show the confusion matrix, seen in Figure 11, ROC graph, seen in Figure 12, learning curve, seen in Figure 13, and the accuracy of the model, including other stats such as recall, AUC, and F1, seen in Figure 14.

```python
#Classifier
# Train the logistic regression model
clf = make_pipeline(scaler, l_reg)
clf.fit(X_train_features, Y_train_labels)

# Predict the labels for the test set
y_pred = clf.predict(X_test_features)
y_clf_prob = clf.predict_proba(X_test_features)

# Compute the accuracy of the model
accuracy = accuracy_score(Y_test_labels, y_pred)
#print('y_pred: ', y_pred)
#print('y_clf_prob: ', y_clf_prob)
print("Accuracy: {:.2f}%".format(accuracy * 100))
recall = recall_score(Y_test_labels, y_pred)
print('Recall: ', recall)

cm = confusion_matrix(Y_test_labels, y_pred)
cm_display = ConfusionMatrixDisplay(cm).plot()
plt.show()

filename = 'ignore.joblib'
joblib.dump(clf, filename)

fpr, tpr, _ = roc_curve(Y_test_labels, y_clf_prob[:, 1], pos_label=clf.classes_[1])
roc_display = RocCurveDisplay(fpr=fpr, tpr=tpr).plot()
plt.show()

auc = roc_auc_score(Y_test_labels, y_clf_prob[:, 1])
print('AUC: ', auc)

f1score = f1_score(Y_test_labels, y_pred)
print('F1 Score: ', f1score)

print("Prediction Output:")
print(y_pred)
```

Figure 10: Python code for training the classifier.

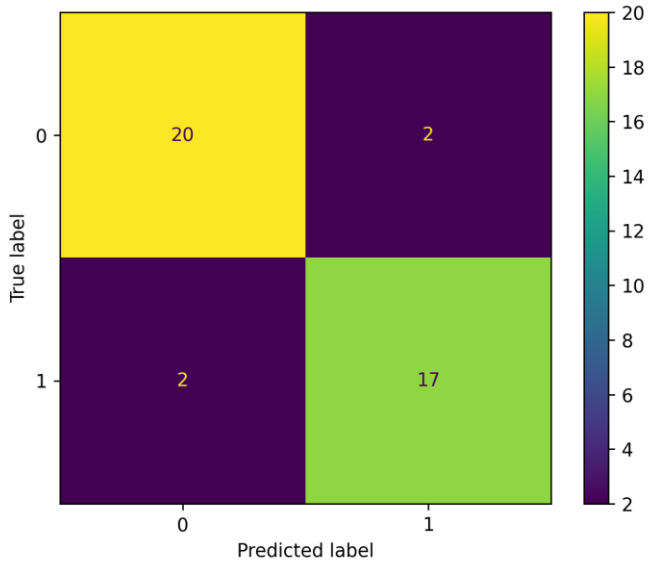

Figure 11: Confusion matrix for the test run.

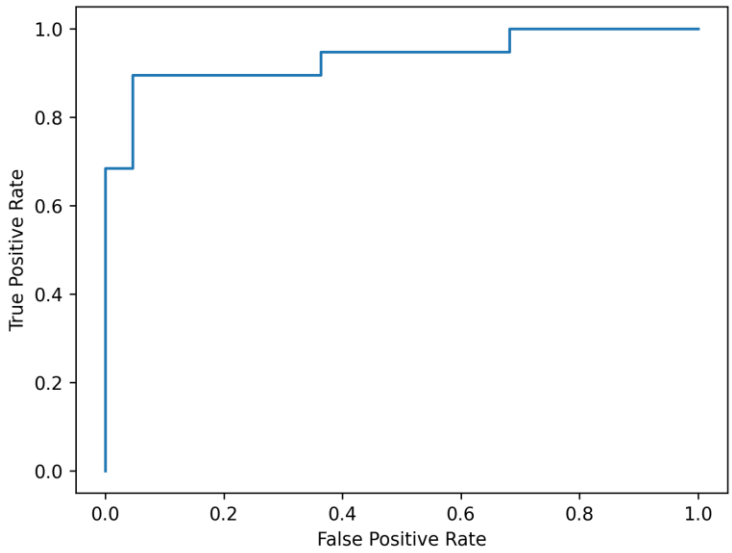

Figure 12: ROC graph for the test run.

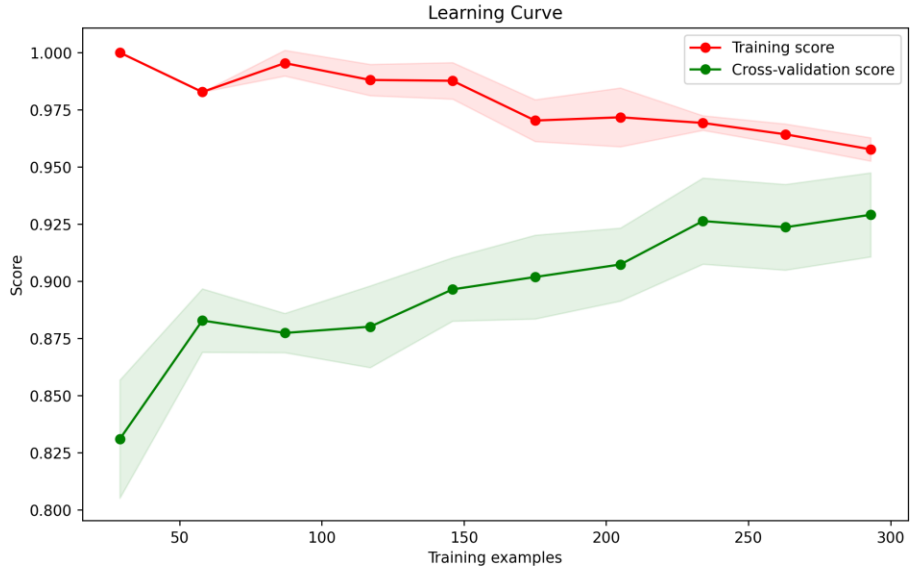

Figure 13: Learning Curve for the test run.

```
[Running] python -u "c:\Users\danie\Desktop\AllCode\ELEC390\390-Course-Project\training.py"
Accuracy: 90.24%
Recall:  0.8947368421052632
AUC:  0.9354066985645932
F1 Score:  0.8947368421052632
Prediction Output:
[1. 0. 0. 0. 1. 1. 1. 1. 0. 0. 0. 1. 1. 0. 0. 1. 0. 0. 0. 1. 1. 1. 0. 0.
 0. 0. 1. 0. 0. 1. 1. 0. 0. 0. 1. 1. 0. 0. 1. 1. 1.]
```

Figure 14: Accuracy and other stats for the test run.

Model Deployment

To implement the graphical user interface (GUI) using Python, the TKinter library was used. Although there exist a multitude of Python GUI libraries, such as PyQT, TKinter was ultimately chosen due to its collection of customizable widgets, facilitating quick prototyping with minimal user-interface design knowledge. To get more modern styling, a folder of assets were added to the project that includes button styles, background textures, and helpful icons.

The first step in creating the GUI was developing a skeleton UI that consisted of the proper buttons, labels, and background, but lacked any functionality such as onclick functions for the buttons. Next, the functionality was added one step at a time, such as the addition of the file-explorer button to properly load the user's CSV files from their hard drive. Furthermore, a matplotlib figure was embedded into the app and would display after a certain button was clicked.

Once the model was finished training and the group was satisfied with the accuracy, the model was dumped into a '.joblib' file so it could be used within other programs to continue classifying new data. The GUI python file then was modified to import the '.joblib' file as well as take in the user's CSV data. Moreover, all of the steps to preprocess the training data needed to be implemented in the GUI to preprocess the user's input data. This allows the model to receive input that is most similar to the data that it was trained and tested on, allowing for the most accurate prediction results.

Once the CSV data is imported from the user's computer, preprocessed, and classified using the model, the output of the predictions are added into a column of a new CSV file, which is output to the user's computer. This file is identical to the input CSV except for having one additional column that displays a 0 if the model predicted walking or a 1 if jumping was detected. The user also has the option to plot the walking/jumping vs time plot in the desktop app to get a more intuitive sense of the prediction results.

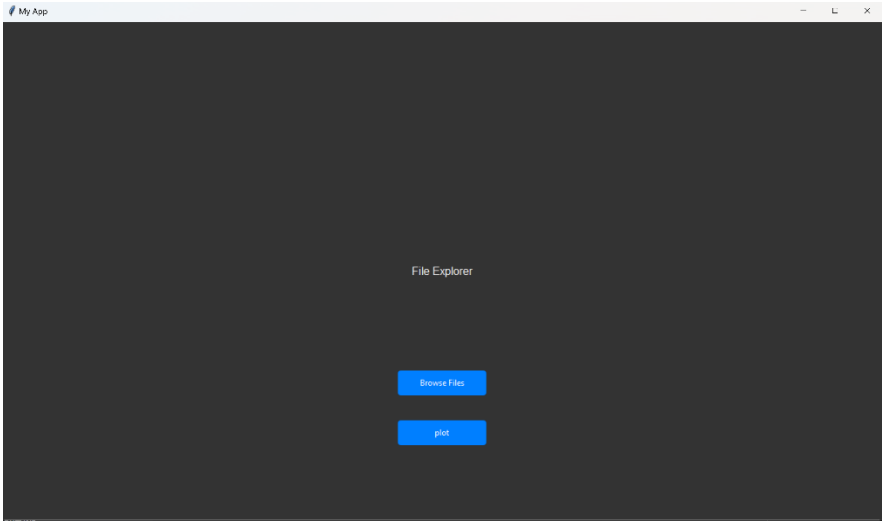

Figure 15: GUI format when first booted up.

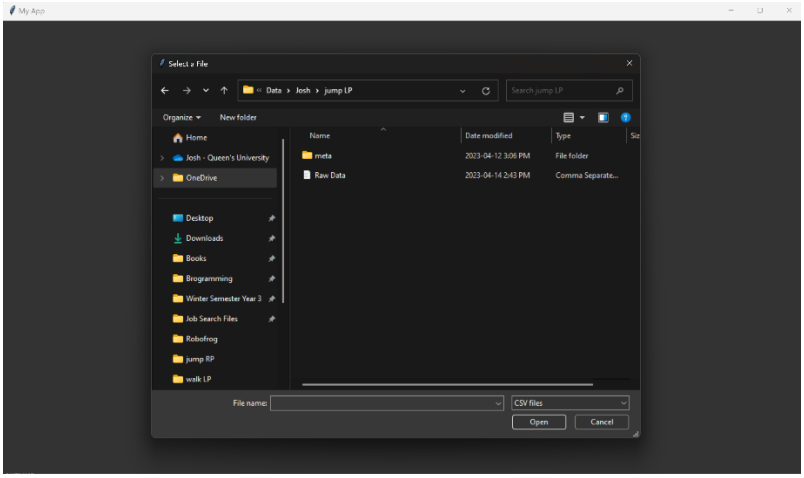

Figure 16: GUI instructing the user to select a csv file.

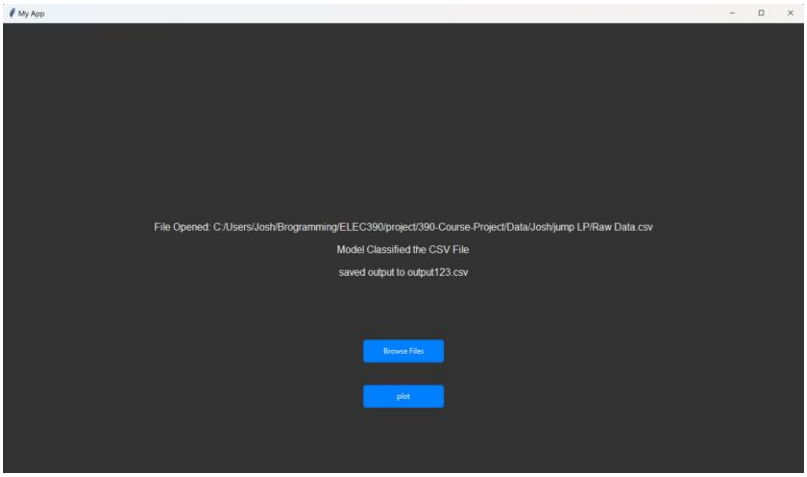

Figure 17: GUI loading in the csv file.

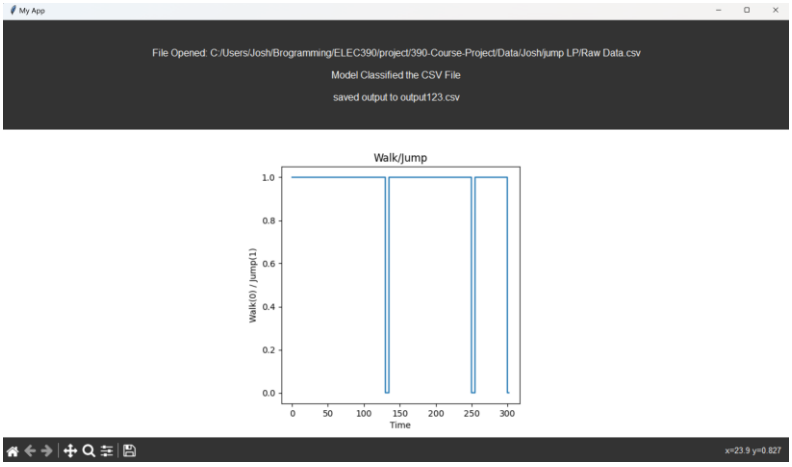

Figure 18: GUI showing a plot of Walking/Jumping vs Time.

Bonus Live Implementation

In order to approach the live classification feature, the remote access was enabled within Phyphox, as suggested in the project instructions.

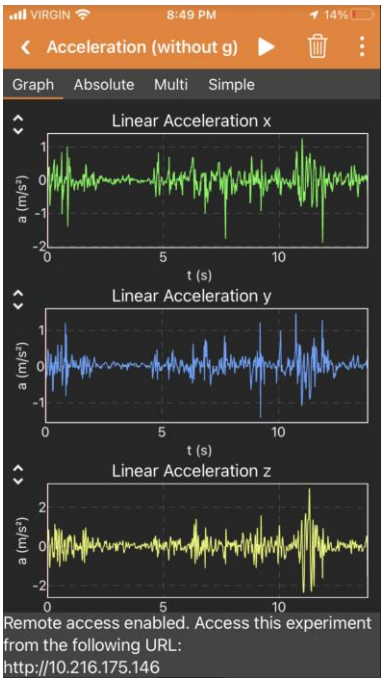

Figure 19: Screenshot of the remotely accessed Phypox data.

After typing in the url from the photo, the remote webpage was opened. Although it was suggested to use selenium and beautiful soup to web scrape the data, the graphs on the webpage were canvas html elements that would not likely allow their data to be easily dumped. Instead, it was discovered that to retrieve the data to update the webpage, the website was making http get requests many times per second to the http address shown in the figure. Using this, the data could be directly extracted without having to use webscraping as a middleman. Instead, the data could be retrieved from the REST API in the same way that the website was retrieving it. When using the browser to inspect the results of a get request to the address "http://192.168.0.102/get?accX=13.19988608334097|acc_time&acc_time=13.19988608334097&accY=13.19988608334097|acc_time&accZ=13.19988608334097|acc_time", the following

json was returned.

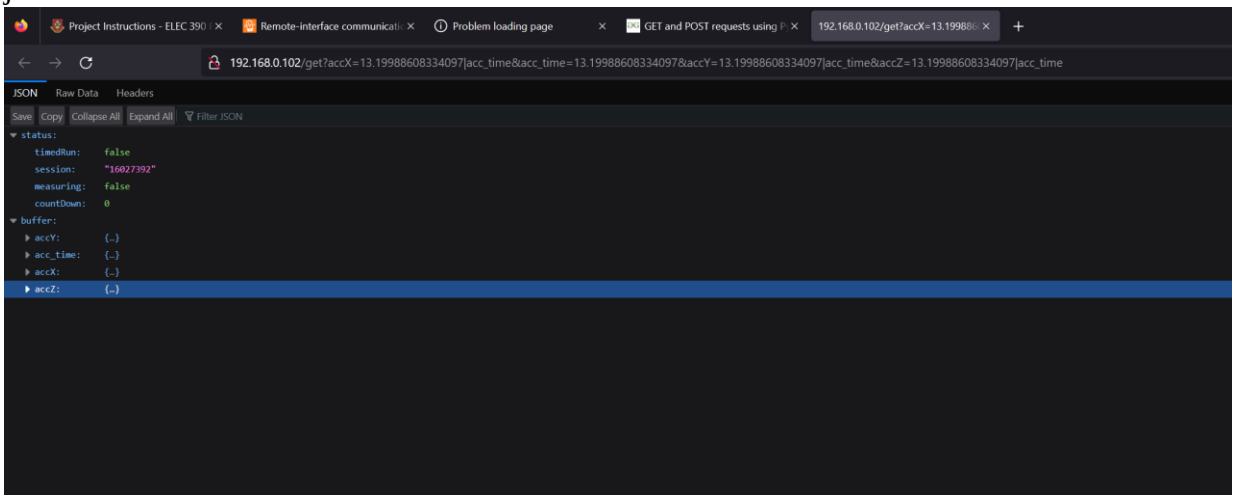

Figure 20: Output JSON from an API get request to the given URL

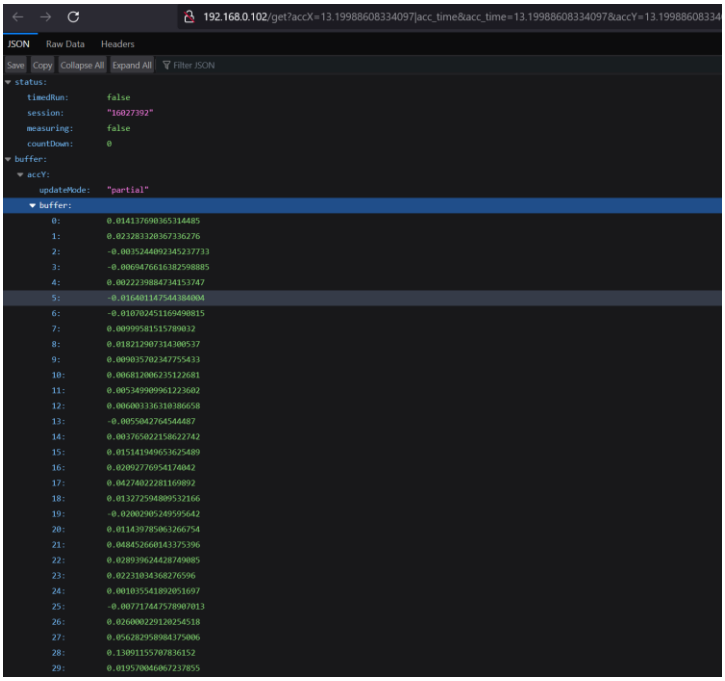

Figure 21: Expanded buffer contents of the JSON response

In the URL, there are parameters to specify the values that should be returned. Here, it was specified to get the x, y, and z acceleration values since a certain time value. This way, every time the API is pinged, all of the values can be retrieved since the last get request and returned in the buffer array so that there are no gaps within the data.

Now, this same output must be retrieved programmatically through a python script. This is a fairly trivial task and was done with the following code:

```python
import requests
from contextlib import redirect_stdout

URL = "http://192.168.0.102/get?accX=314.776183833339|acc_time&acc_time=314.7861088333302&accY=314.776183833339|acc_time&accZ=314.7861088333302|acc_ti
response = requests.get(url=URL)
print(response)
data = response.json()

with open('out.txt', 'w') as f:
    with redirect_stdout(f):
        print(data)
```

Figure 22: Python script to trigger an http GET request and store the results in a text file

This outputs the results of the query to an output text file. The contents of the file match the structure of the browser query results.

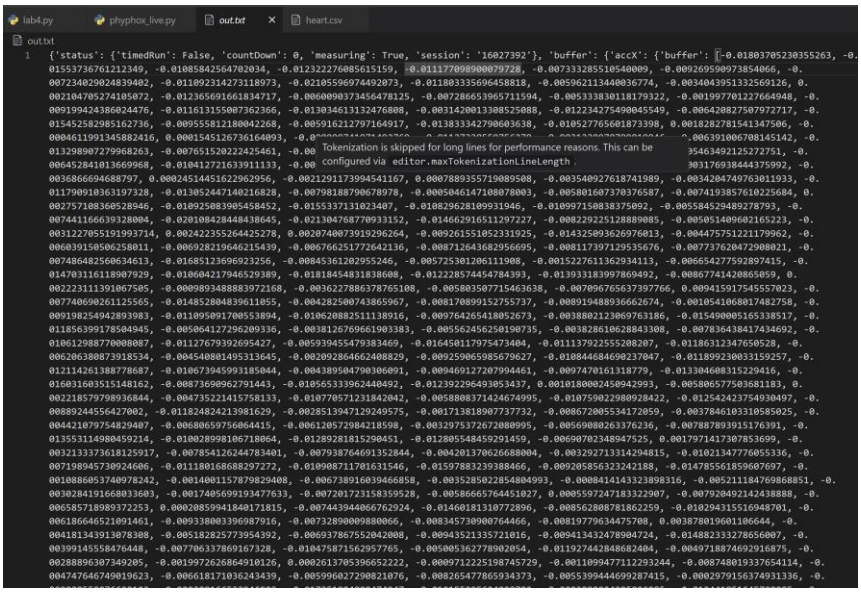

Figure 23: The output text file of the Python request script

Although the entire live implementation was not finished due to time constraints, it would not have taken many changes in order to complete. First, the GUI would need some UI to allow the user to type in their given remote access http address. The model would not need to be altered, but the processing of data would need to be modified. Here, the live data would need to be stored. Next the preprocessing would need to take place. Since the live stream of data is constant, the data would need to utilize a sliding window to only process chunks of 5 seconds at a time. Finally, the output prediction of the model for every 5 second window would need to be displayed in the GUI to have a label of either ‘walking’ or ‘jumping’ so the user can get real-time feedback of the results.

Participation Report

A breakdown of the work completed by each member can be seen below:

Josh: Report writing, Data collection, GUI creation, Visualization.

Daniel: Report writing, Data collection, Preprocessing, Feature extraction, and creating the classifier.

Bradley: Report writing, Data collection, Data storing, test and training set creation.

Overall, the work was split evenly, with each member doing 1/3 of the total project.