ELEC 390
Project Report

Group Number: 77

Date: April 14th, 2023

Ainsley Taylor – 20210012 - 19act7@queensu.ca

Ryan Baker – 20223116 – 19rb42@queensu.ca

Matthew Mamelak - 20216737 - 19mjm17@queensu.ca

Table of Contents

Data Collection 1
Data Storing 1
Visualization 2
Preprocessing 9
Feature Extraction 10
Training the Classifier 11
Model Deployment 13
Justification of Design Choices: 13
Code Walkthrough: 14
Real-Time Activity Classification (BONUS) 17
Changes to the Desktop Application 17
Real-Time Data Transfer to PC 19
Using Selenium to Access HTML Code 19
Using Selenium to Change the Webpage View 19
Scraping the Phyphox Webpage Using Beautiful Soup 19
Using the Acceleration Data to Predict Activity 20
Changes to the Trained Classifier 20
Participation Report 21
References 22

Data Collection

To collect the data, we used a mobile app called Phyphox. This app works for both iOS and Android, allowing each member of our group to use it on their device. This meant that we were able to get multiple different sources of data to use for testing and training. Phyphox works by recording the accelerometer data in x, y, and z directions. It was decided to use linear acceleration without gravity so that the sensor reports no acceleration when the device is not moving. The app permits users to have a five-second delay before starting data collection so that they can place the device in the location they desire before starting the data collection. The app also allowed the data recording to end at a specified time, which was used by all group members. These features helped ensure that the entirety of the data recorded was of the intended activity, rather than recording the device being placed in the required position or stopping the recording, which reduces the incorrect data that is collected.

Each member of our group independently collected data using four distinct locations for the device: hand, back pocket, front pocket, and jacket pocket. We performed the data collection for both jumping and walking activities. Each data collection session lasted one minute per person. The data was outputted using a comma-delimited .csv file, which was then transferred and labelled with the correct action and position. We then labelled the data based on whether it was data corresponding to someone walking or jumping. In the CSV file generated by Phyphox, we added a "target" column, which contained a 0 for walking and a 1 for jumping.

A challenge encountered during data collection was the inconsistent alignment of the phone when placed in jacket pockets compared to back and front pockets or when held in hand. This is because our jacket pockets were looser, so the device was able to move sideways in the pockets. As a result, we re-collected data in one instance where it significantly deviated from the rest. However, we decided to leave the rest of the data so that our app could accurately recognize whether an individual was walking or jumping, regardless of the phone's orientation or location in a jacket pocket.

Data Storing

To store the collected data for use within the classifier and desktop app, an HDF5 file was created to allow for quick and simple access to the data. The original data was to be stored within the directory name corresponding to each group member. To do this, the eight CSVs collected by each group member were combined into one file, which was then read into a Pandas Data Frame and saved to the HDF5, with the key set to the name of the group member who collected the data.

Training the classifier requires all data to be combined, shuffled, and split into training and testing segments. The data was shuffled so the classifier was trained on samples of all collected data. The testing set is representative of the data used in training and the data that must be classified later. Without shuffling the data, the classifier may be trained on a larger quantity of walking data compared to jumping, which can cause inaccurate results. The shuffling of data was achieved by splitting the data into segments of 500 rows, which corresponds to 5 seconds at the

100 Hz accelerometer rate used by Phyphox on the iPhone 12 and iPhone 13 [1]. The segments were then shuffled using np.random.shuffle() and recombined into one Data Frame to be split into the train and test datasets.

Dividing the data into training and test sets is essential for developing an effective model and testing its accuracy. The training data is used to train the classifier, so most of the data is allocated for this purpose to improve its performance. In general, using a larger amount of high-quality data for training leads to a more accurate classifier. The testing data is separate from the training data and is used to provide the accuracy of the classifier on new data. This test data must be labelled so that it can be compared to the prediction made by the classifier. The *train_test_split()* function from sklearn was used to split the data into 90% training and 10% testing, which was then appended to the existing HDF5 file. The keys used for this data in the HDF5 were *'Dataset/Train'* and *'Dataset/Test'* to create the desired directory structure.

Visualization

Below are the plotted graphs for the acceleration in the x, y, and z axes for both jumping and walking in Figures 1 to 8. As well, a graph combining x, y, and z acceleration for jumping and walking is shown in Figure 9 and Figure 10. Figures 11 and 12 show a close-up of the acceleration in the x, y, and z axes for better viewing.

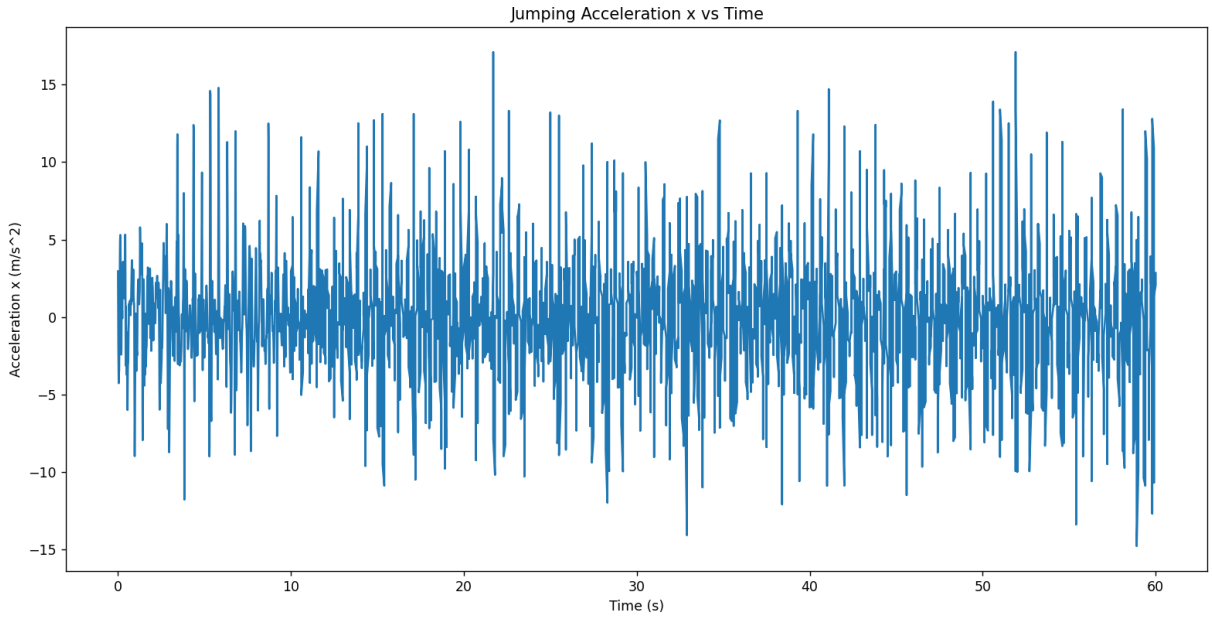

Figure 1: Acceleration in x-axis vs time for jumping.

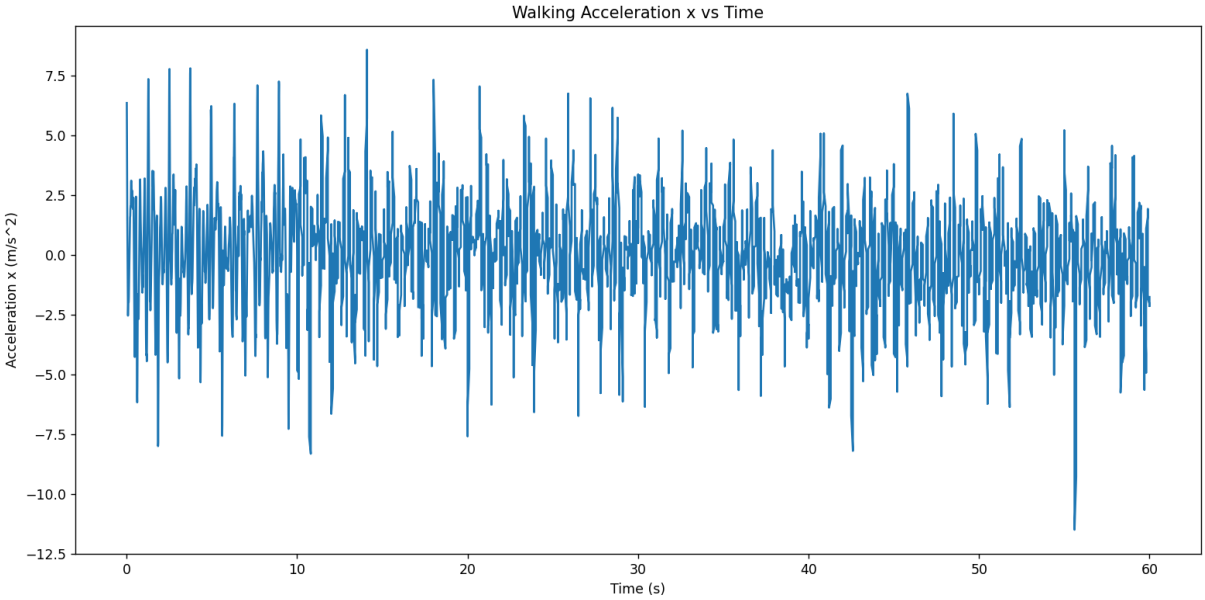

Figure 2: Acceleration in x-axis vs time for walking.

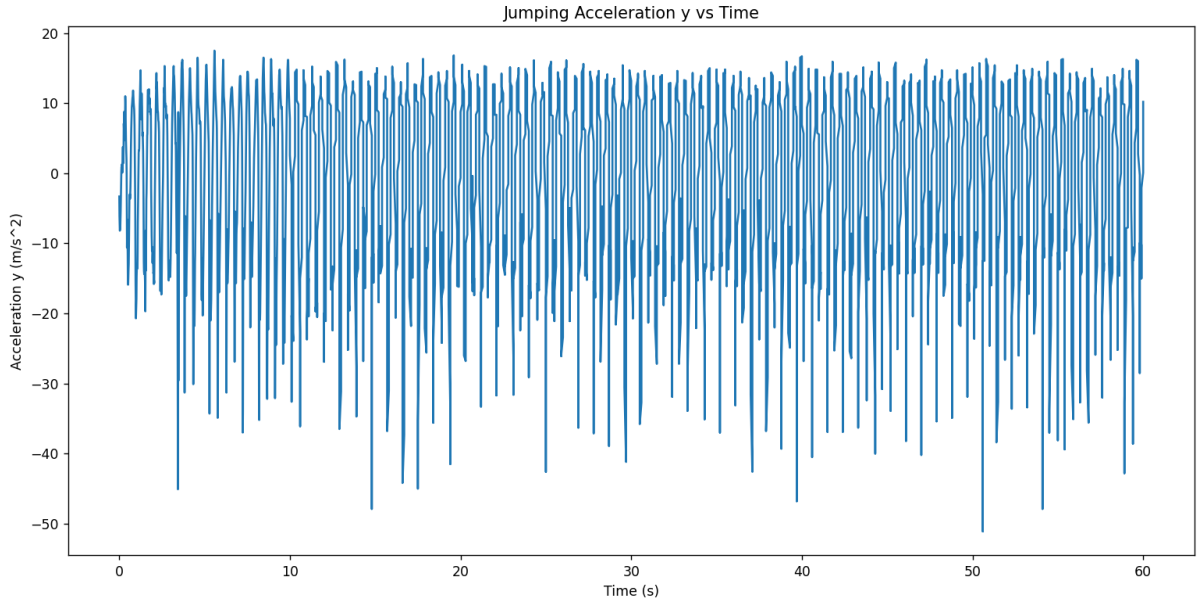

Figure 3: Acceleration in y-axis vs time for jumping.

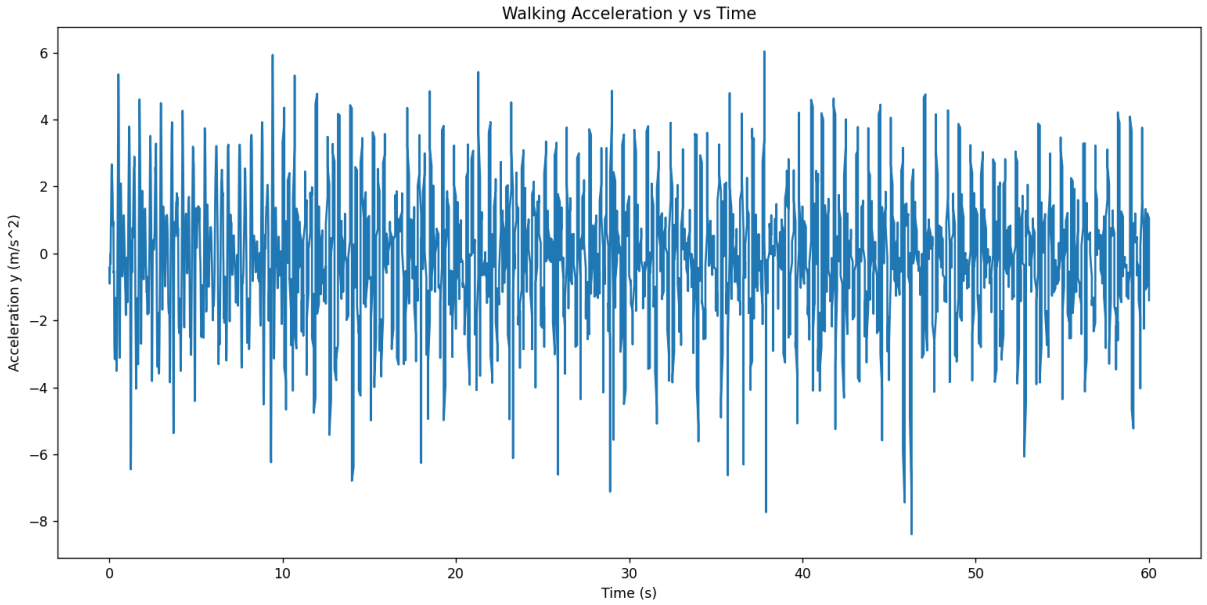

Figure 4: Acceleration in y-axis vs time for walking.

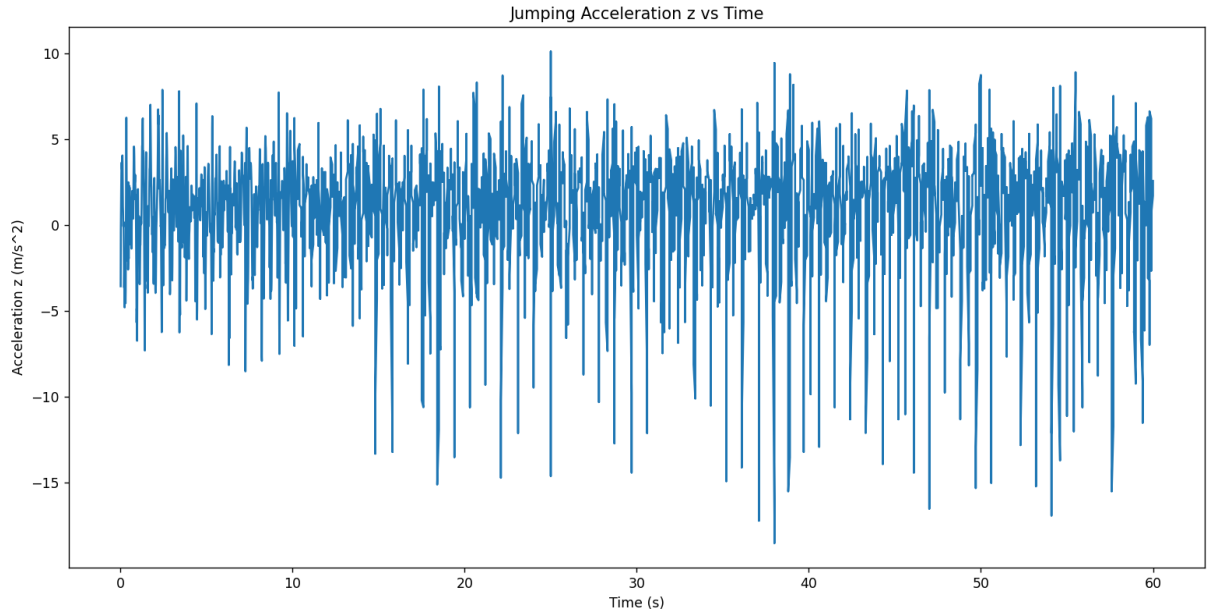

Figure 5: Acceleration in z-axis vs time for jumping.

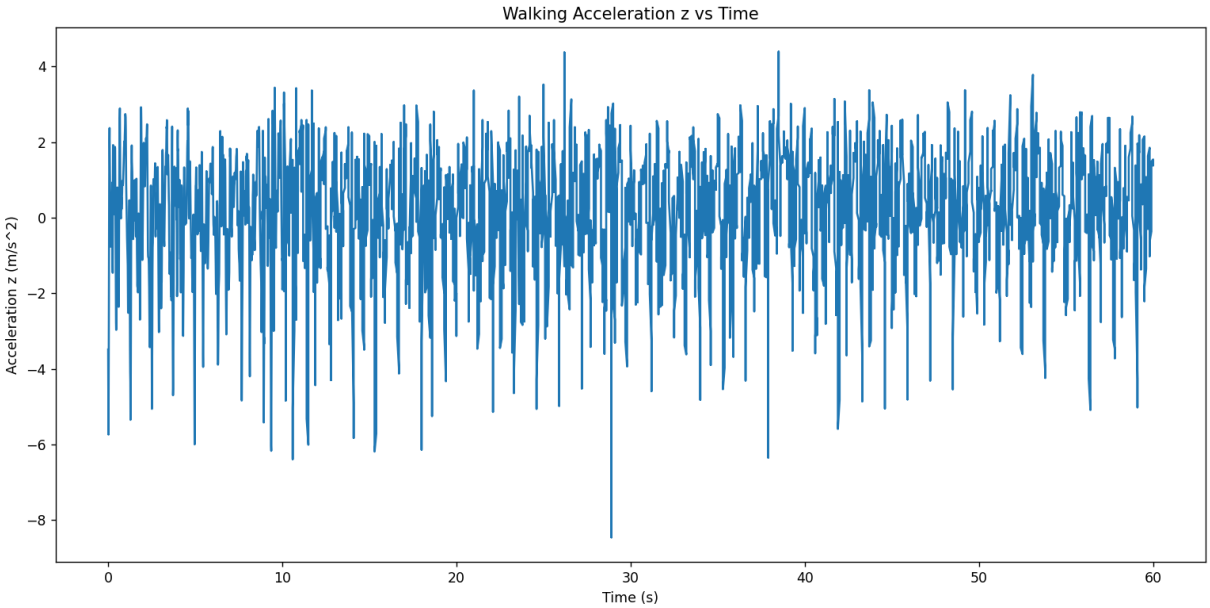

Figure 6: Acceleration in z-axis vs time for walking.

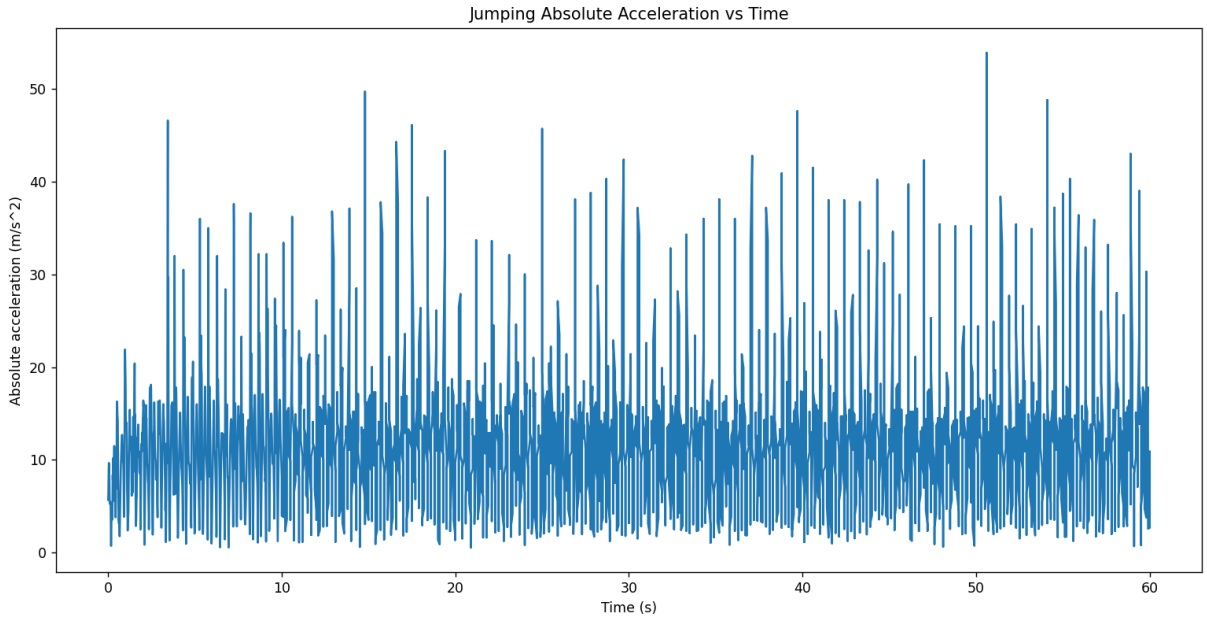

Figure 7: Absolute acceleration vs time for jumping.

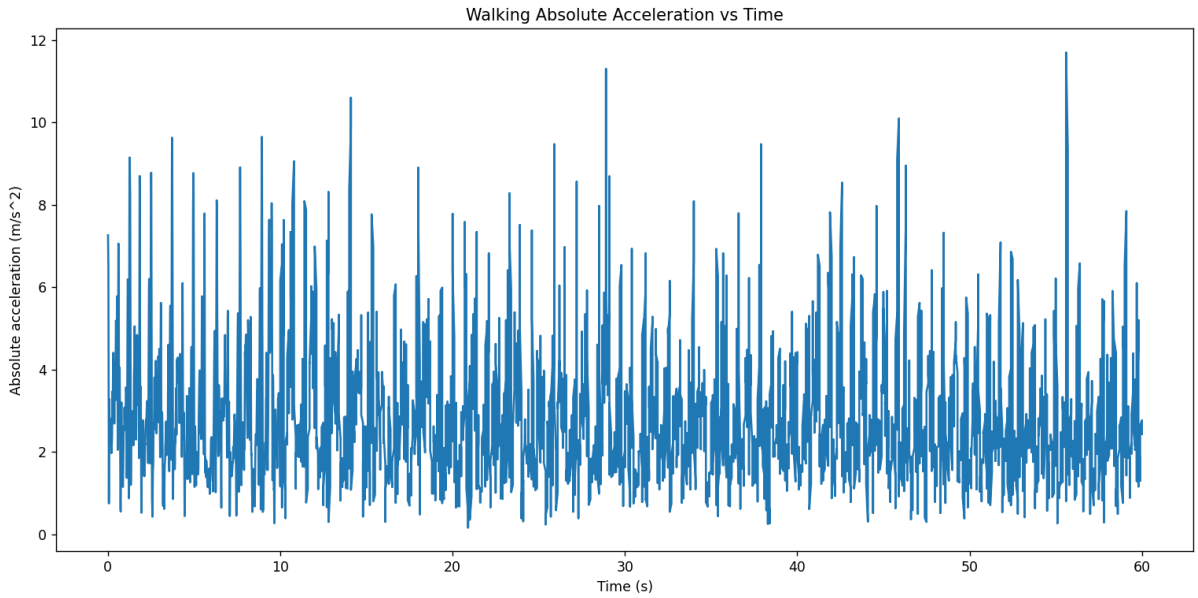

Figure 8: Absolute acceleration vs time for walking.

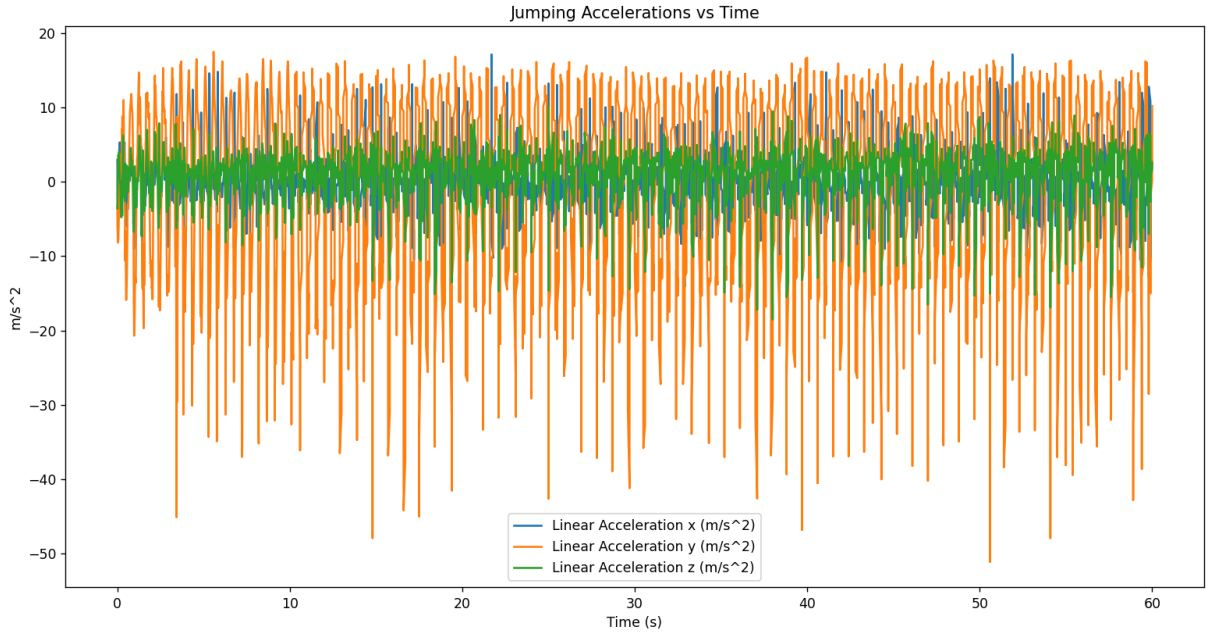

Figure 9: Accelerations vs time for jumping.

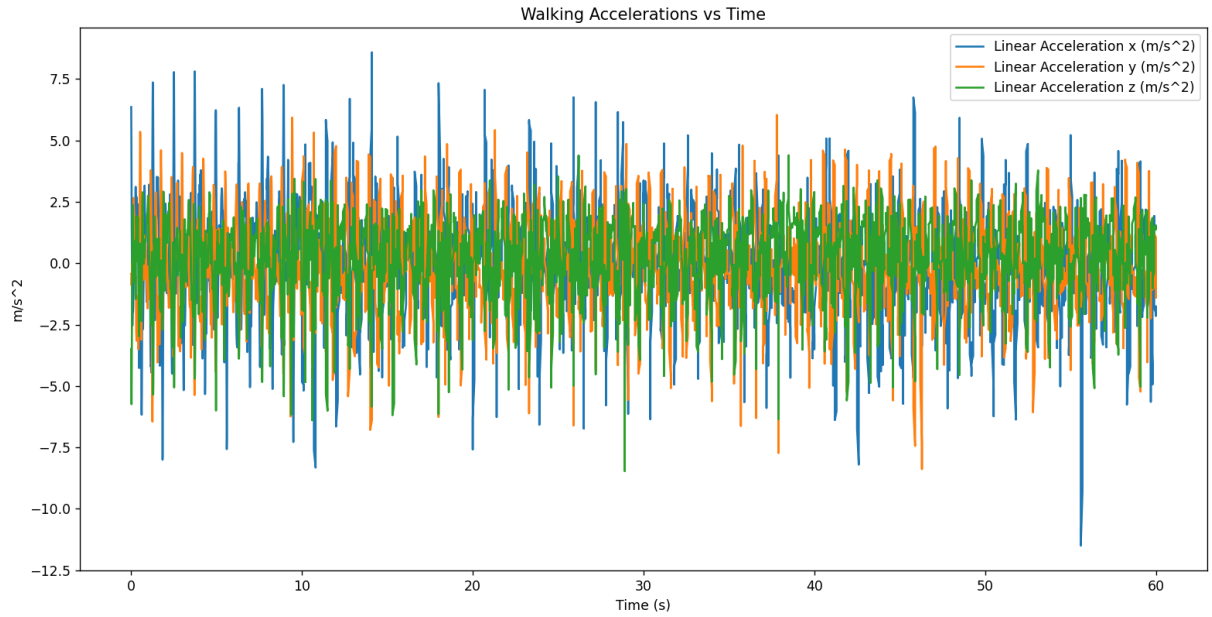

Figure 10: Accelerations vs time for walking.

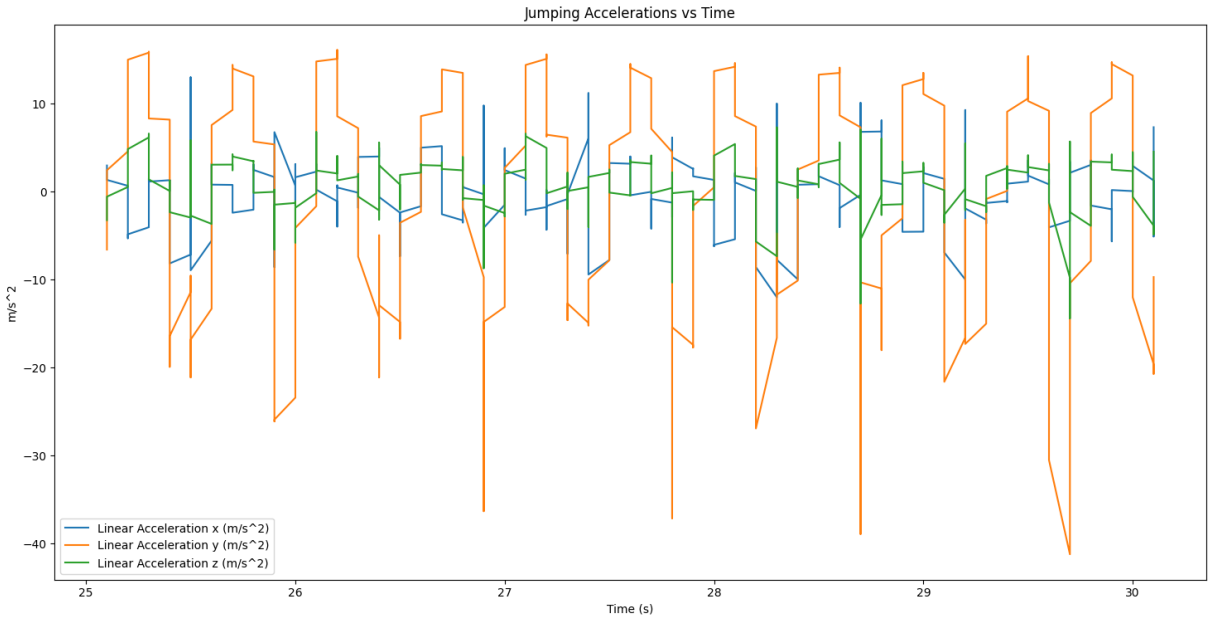

Figure 11: Close-up snippet of acceleration vs time for jumping.

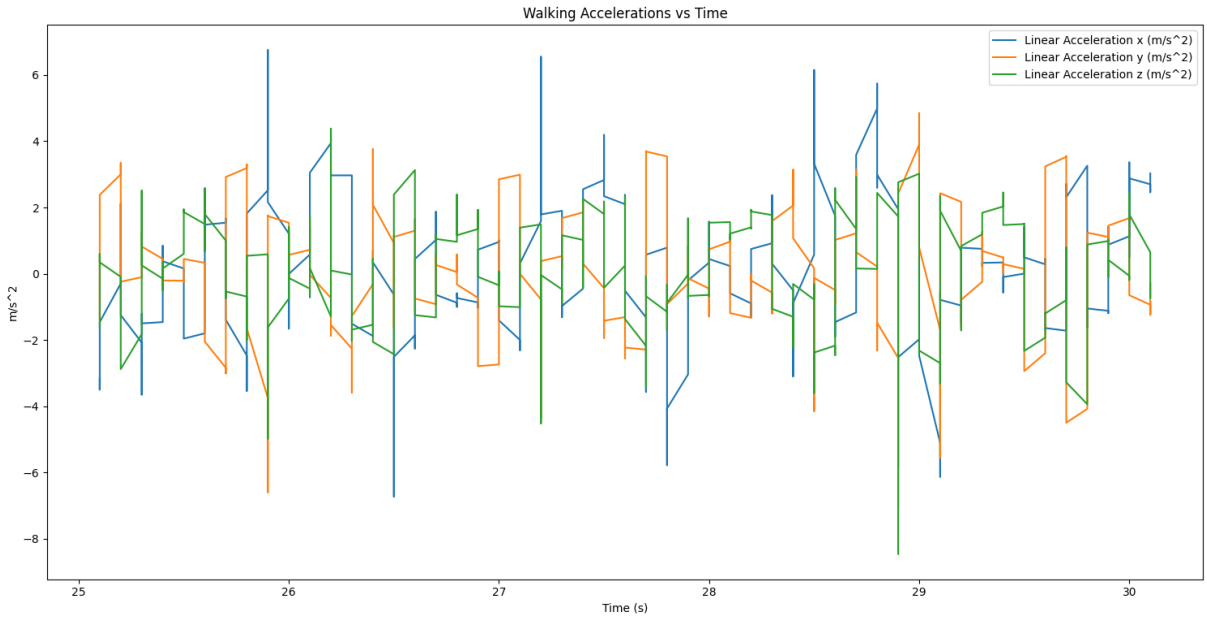

Figure 12: Close-up snippet of acceleration vs time for walking.

The plots reveal that the acceleration in the x, y, and z for walking is significantly lower than that of jumping. The absolute acceleration of jumping goes to around 50m/s^2, whereas the absolute acceleration for walking only reaches 12m/s^2. This trend is consistent across the individual axes' plots.

Another important observation is the larger acceleration in the y-axis for jumping compared to walking. This is expected, as the y-axis undergoes rapid changes during a vertical jump. The acceleration is also especially large in the negative direction. This is because we did not include gravity in our data collection. This means that gravity would be pulling the person down faster, causing a larger negative acceleration.

Furthermore, Table 1 provides metadata about the dataset and sensors. The metadata contains information about the device and how the data was collected on each different device, including accelerometer rate and standard deviation [1]. The table also includes the height of the person collecting the data, as different heights may affect certain accelerations.

Table 1: Metadata about dataset and sensors.

Name	**Device**	**Accelerometer Rate**	**Standard Deviation**	**Height**
Ainsley	iPhone 13	100.0 Hz	0.019 m/s^2	158 cm
Ryan	iPhone 13	100.0 Hz	0.019 m/s^2	183 cm
Matthew	iPhone 12	100.0 Hz	0.020 m/s^2	192 cm

As observed in some of the plots, especially in Figure 2 and Figure 6, there are outliers where the acceleration becomes larger compared to the rest of the dataset. This is likely due to the walking data being collected without following a straight line, and instead, walking around without a

clear path. To improve the data collection process, we would ensure that the walking data is collected by walking in a straight line or using a treadmill, more closely resembling typical walking patterns.

Preprocessing

Preprocessing data before it is used to train a classifier is vital to attaining an accurate model. Processing the data allows for the handling of missing data, normalization of data, and noise reduction, all of which could reduce the model's performance if the data is left in its original state. For this project, a moving average was implemented to reduce noise in the data. The moving average sets the value of the data to the average of all the data within the window, where the window size can be altered to find the best balance between noise reduction and loss of features.

To find the optimal window size for the classifier, the model was trained using a range of window sizes between 1 and 30 which allowed the best window size to be determined by comparing the resulting accuracy. As shown in Figure 13, the optimal window size was found to be 3, which allowed for sufficient noise reduction while still retaining the features of each activity.

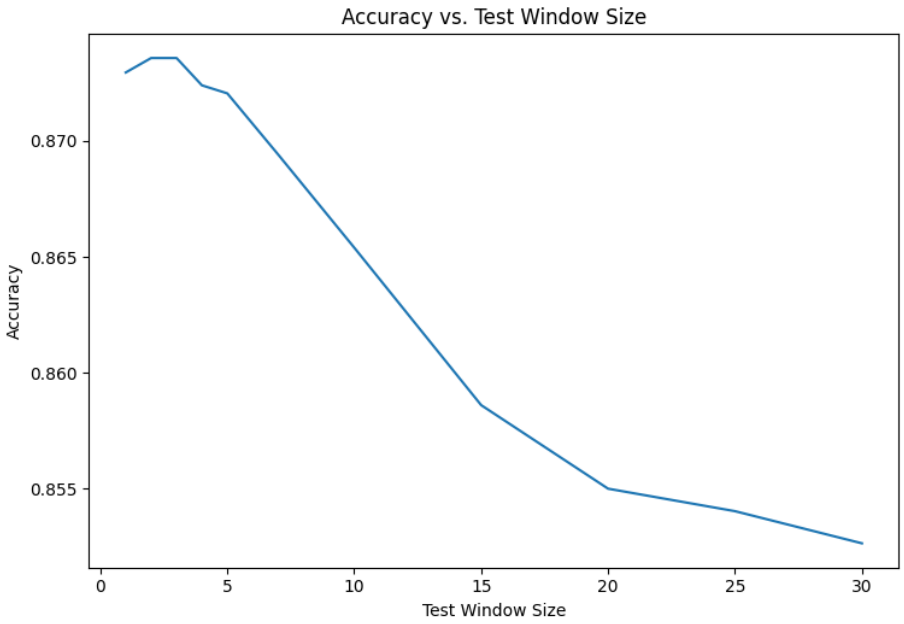

Figure 13: Accuracy of a model and window size used for noise reduction.

The larger window sizes resulted in lower accuracy. This can likely be explained by the larger window sizes reducing the large spikes in acceleration that occur when jumping. With a lower maximum, that feature of jumping can be lost, which makes the classifier less accurate. Normalization of the data was completed right before the model was trained.

Feature Extraction

The features extracted from the raw data are listed below.

1. Max: The maximum value in the dataset represents the peak amplitude of the signal across all axes (x, y, and z). This is important when differentiating walking and jumping activities as jumps may have higher peak values because of an increased force exerted on the ground. For example, the maximum value of acceleration in the y-axis may be significantly higher during a jump than during walking. Furthermore, the Max value can help with recognizing outliers, as it captures the extreme values in the dataset. Outliers can impact the performance of machine learning models, so it is essential to identify and address them during the preprocessing stages [2].

2. Min: The minimum value in the dataset provides information about the lowest amplitude of the signal across all axes. This is very similar to the max value, as it helps distinguish the two activities. Walking may have a smaller minimum value compared to jumps, especially in the vertical (y-axis) component. Another reason why extracting the minimum value is important is for normalization and scaling. This is a crucial preprocessing step in machine learning algorithms, as it helps ensure that all features contribute equally to the model [2].

3. Range: The range (the difference between the maximum and minimum values) is useful for identifying the overall variability of the signal across all axes. Jumps may present a larger range due to the rapid changes in acceleration during the activity, particularly in the vertical component (y-axis). The range also helps with feature comparison. When examining the range of each axis, it is easier to identify which axis experiences more variability during the activities. This information helps with determining the relative importance of each axis under certain conditions.

4. Mean: The mean value provides a value for the central tendency of the dataset for each axis. It can be used to identify any systematic differences in the average force exerted while walking and jumping. For instance, the mean value of acceleration in the x-axis may differ between walking and jumping due to the varying levels of forward motion during each activity. The mean is also important for baseline correction. The mean can be used to correct any baseline drifts or offsets present in the data. The mean can also be used to help identify trends in the data by highlighting the average behaviour of a signal over time [2].

5. Median: The median value represents the middle value of the dataset for each axis when the dataset is ordered by magnitude. The median is less sensitive to outliers and can provide a closer measure of central tendency, which can help differentiate between walking and jumping. The median is a non-parametric statistic, meaning it does not rely on assumptions about the underlying distribution of the data. This is helpful when analyzing data that may not follow a normal distribution, which is often in real-world examples. Using the median as a feature can help build models that are more robust to deviations from normality.

6. Variance: The variance measures the dispersion of the dataset around the mean for each axis. Variance provides insights into the stability of the signal. A lower variance indicates that the values in the dataset are more tightly clustered around the mean, while a larger variance suggests more scattered values. Variance can also be used as an estimate of the noise present in the data.

High Variance values may indicate higher levels of noise, thus meaning it is more challenging to identify the underlying patterns in the acceleration data [3].

7. Skewness: Skewness is a measure of the asymmetry of the dataset distribution for each axis. The skewness can help identify any significant differences in the distribution of acceleration values between walking and jumping. For example, a jump may have a more positively skewed distribution in the y-axis due to the sudden increase in acceleration during the take-off phase. Skewness also provides insights into the tail behaviour of the dataset. A positive skewness indicates that the right tail (higher values) is longer, while a negative skewness indicates that the left tail (lower values) is longer [4].

8. Kurtosis: Kurtosis measures the "tailedness" of the dataset distribution for each axis. It can help detect any differences in the shape of the distribution between walking and jumping activities. Kurtosis is useful when detecting extreme values or outliers in the data. A higher kurtosis value indicates that the distribution has heavier tails and a more peaked centre, which implies that the data may contain extreme values [5].

9. Standard Deviation: The standard deviation is a measure of the dispersion of the dataset for each axis. Having the standard deviation can be helpful during certain preprocessing steps such as feature scaling and normalization. Since it is possible to scale the features to have the same variance, this can help improve the models that are sensitive to the scale of input features, such as logistic regression.

10. IQR: The interquartile range (IQR) represents the range between the first quartile (25th percentile) and the third quartile (75th percentile) of the dataset. The IQR is a measure of dispersion that is less sensitive to outliers. It is easier to identify which axis experiences more variability during the activities while allowing for more outliers and extreme values in the dataset [6].

The extracted features aim to capture essential characteristics of the raw data that can help differentiate between walking and jumping activities. In the feature extraction process, the ScipPy library's 'scipy.stats' module was used to compute the IQR, Kurtosis, and Skewness, and the NumPy library was used to calculate the remaining features, such as Max, Min, Range, Mean, Median, Variance, and Standard Deviation.

Training the Classifier

Our logistic regression model was built using the scikit-learn library in Python. The following elements below summarize the steps we took to accomplish building our classifier:

1. Data normalization: We defined a StandardScaler() to normalize the input data, making sure that all features contribute to the model equally.
2. Classifier initialization: We defined the logistic regression classifier with a maximum iteration of 10,000 to ensure convergence.
3. Pipeline Creation: We created a pipeline that combined the StandardScaler() and the logistic regression classifier to streamline the preprocessing and training process.

4. Training Model: We trained the classifier on the train_data and train_labels datasets using the `clf.fit()` function.
5. Learning curve visualization: We plotted the learning curve using the LearningCurveDisplay.from_estimator() function. This shows the model's performance as a function of the number of training examples.
6. Saving the model: We saved the trained model using the pickle.dump() function for future use. This way in our app deployment, the model does not need to be re-trained each time a new .csv is inputted.
7. Prediction & probability estimations: We obtained the predicted labels (y_pred) and class probabilities (y_clf_prob) for the test dataset (X_test) using the `clf.predict()` and `clf.predict_proba()` functions.
8. Model evaluation: We calculated the accuracy, recall, F1 score, and AUC (Area Under the ROC Curve) of the model using sklearn's metrics functions (accuracy_score(), recall_score(), f1_score(), and roc_auc_score()).
9. Confusion matrix and ROC curve visualization: We plotted the confusion matrix and ROC curve using the ConfusionMatrixDisplay() and RocCurveDisplay() functions to visualize the classifier's performance.

Through training the classifier and comparing the predicted outputs to the labels in the test dataset, the accuracy, recall, F1 score, and AUC can be calculated, and the ROC curve generated. When running the test dataset, the model provided the results shown in Table 2.

Table 2: Results of model evaluation.

	Testing Dataset	Training Dataset
Accuracy	0.8735791516495702	0.8770804291402562
Recall	0.793625	0.8100072288399284
F1 Score	0.874397465913786	0.8659364290034273
AUC	0.9428610570939638	0.9457875572754912

The accuracy and recall for the training set are slightly higher than that of the testing dataset. This is to be expected since the classifier has already seen the data and was trained on its patterns, so it is more likely to pick up the patterns in the data again. The testing dataset is still very close to the results in the training dataset, having a slightly higher F1 score and AUC.

Along with the numerical values, it is also possible to create plots to view the learning curve, ROC curve, and confusion matrix. The results for the testing set are shown in Figure 14 and the training set in Figure 15.

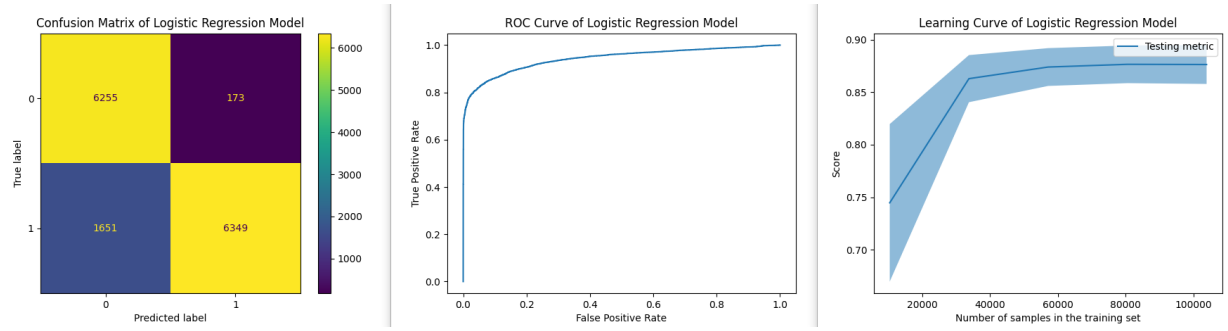

Figure 14: Confusion matrix, ROC curve, and learning curve on testing set.

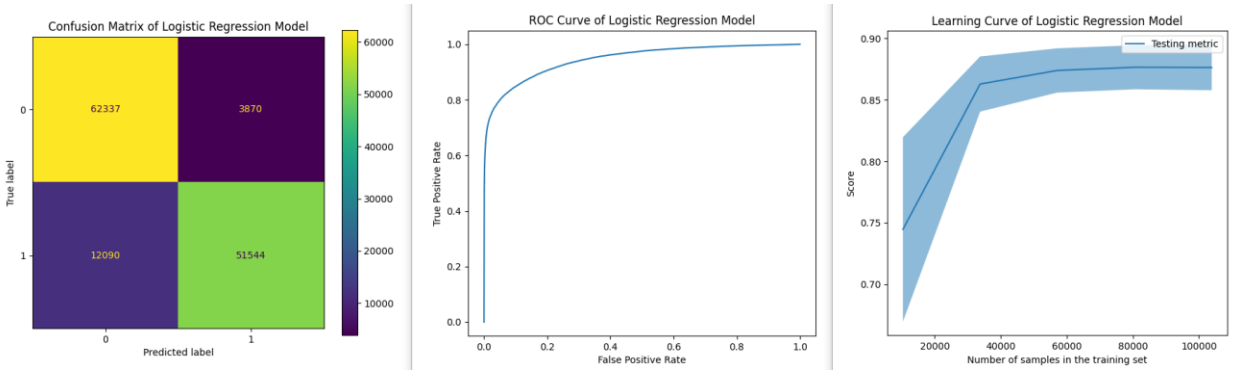

Figure 15: Confusion matrix, ROC curve, and learning curve on training set.

As seen in the confusion matrix, the classifier has a higher rate of false negatives compared to false positives. This may be a result of jumping having some of the same characteristics as walking for short periods. Given that walking is represented by the 0 value, this makes sense since the model would be more likely to misclassify jumping as walking than walking as jumping. To avoid having to retrain the model each time the application is run, the model was saved to the base directory using the pickle library in write binary mode [7].

Model Deployment

In this project, we implemented a desktop application that uses a logistic regression model in Python. The app classifies activities such as walking and jumping based on data from a CSV file generated through Phyphox. We used the PyQt library to create a graphical user interface (GUI) that allows users to open a CSV file, where our application will then classify the activity and display the results in a plot.

The GUI for this desktop app is simple and easy to use. It consists of a main window containing a title label, an "Open CSV File" button, a scatter plot to display the results, a help button and a user activity box which displays whether the user is jumping or walking. The title label and open button are styled with bold fonts and padding for easy readability and a presentable UI.

Justification of Design Choices:

1. PyQt: PyQt is a Python library used for creating GUI applications. It has a wide range of functionalities, making it a great choice for designing our app's interface.

2. Simple layout: To make the application user-friendly, we chose a simple layout with a title label, an open button, a help button, a user activity box, and a scatter plot. This allows users to easily understand the app's purpose and navigate through the interface.
3. Window size: Setting the window size to 2/3 of the screen width and height allows the app to be accessible and readable on various screen sizes across all devices.
4. Style: We used bold fonts, colouring and padding to make the text and buttons visually appealing and easy to read and use.

Code Walkthrough:

1. Define the `classify_preds` function that takes a data frame of predictions and classifies them into walking and jumping activities using a moving average and masking. The function returns a data Frame with the mapped classifications. This helps to remove the noise on the resulting graph when there are miss classifications.
2. Create the `HelpWindow` class, a QDialog subclass that displays help information for the application. The window contains a QLabel with instructions and a Close button.
3. Create the `MainWindow` class. This class has the following methods:
    - `__init__`: Initialize the main window and calls the `init_ui` method.
    - `init_ui`: Sets up the user interface, including a title label, Open CSV File and Help buttons, a user activity box, and a matplotlib FigureCanvas for displaying the graph. The QVBoxLayout and QHBoxLayout are used to arrange the widgets.
    - `open_file`: Opens a file dialog that allows the user to select a CSV file for processing.
    - `process_csv`: Reads the selected CSV file, preprocesses the data and applies the pre-trained model to generate predictions. It then classifies predictions into walking and jumping activities based on a moving average window, saves the output to a file, and calls the `plot` method to display to plot onto the GUI.
    - `plot`: Creates a prediction vs. time graph using the matplotlib FigureCanvas widget.
    - `show_help`: Opens the `HelpWindow` dialog to display help information.
4. The `if __name__ == '__main__':` is used to run the application. It initializes a QApplication object, creates an instance of the `MainWindow` class, and starts the application event loop.

When the user opens a CSV file, the application loads the previously saved model, processes the data, generates predictions, classifies activities, and updates the user activity box with the relevant information. To help reduce the noise in the resulting plot, the moving average of the data is taken with a window size of 100. If the result from the moving average is less than 0.5, the data is classified as walking. Otherwise, it is classified as jumping. It also displays the prediction vs. time graph in the main window. The output CSV is created in the /Prediction_Data folder, which contains the raw accelerometer data as generated by Phyphox and the corresponding activity label. The Help button opens the `HelpWindow` dialog, which provides instructions for using the application. We have included screenshots of the entire GUI below:

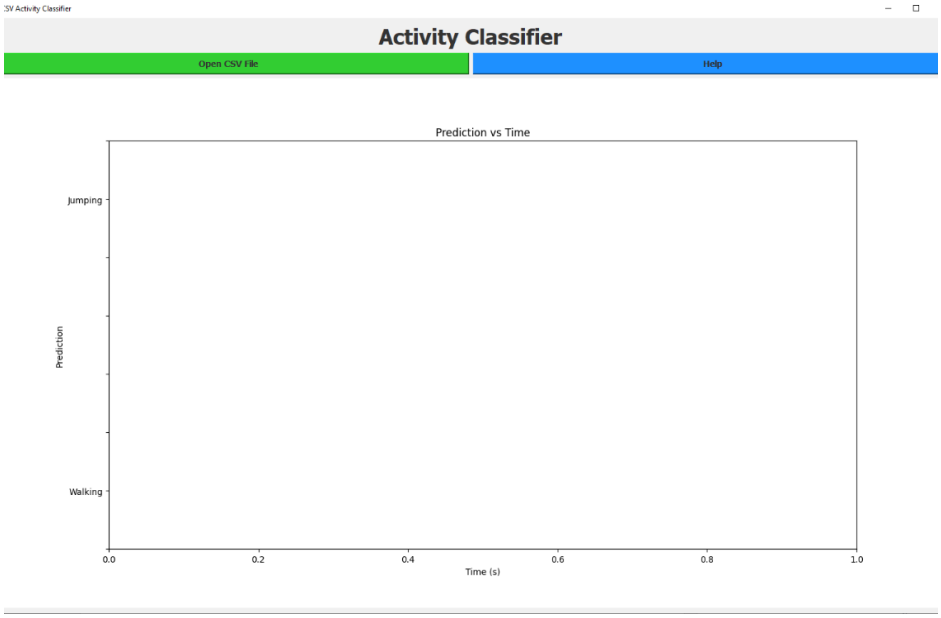

Figure 16: GUI on start-up.

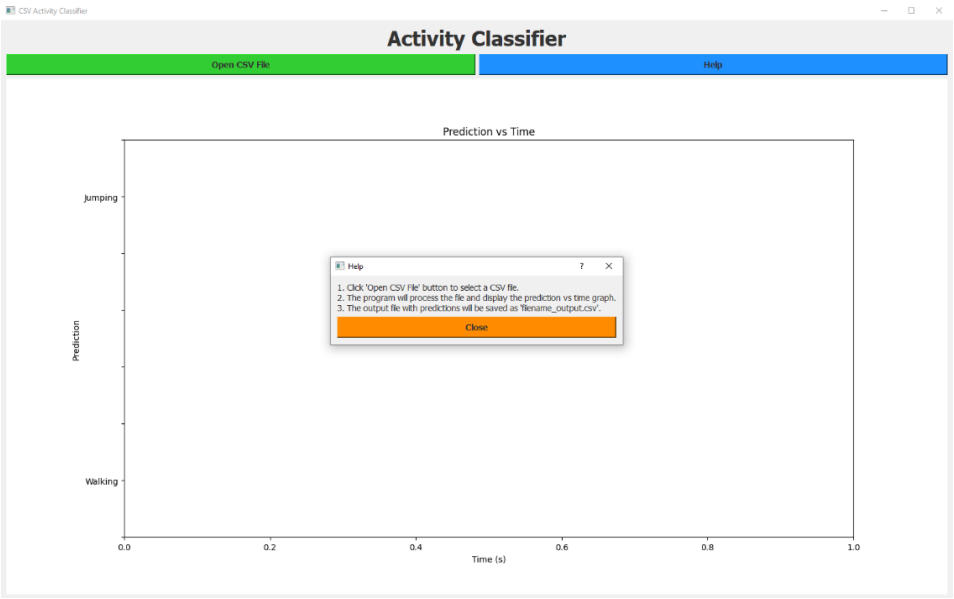

Figure 17: GUI Help Button.

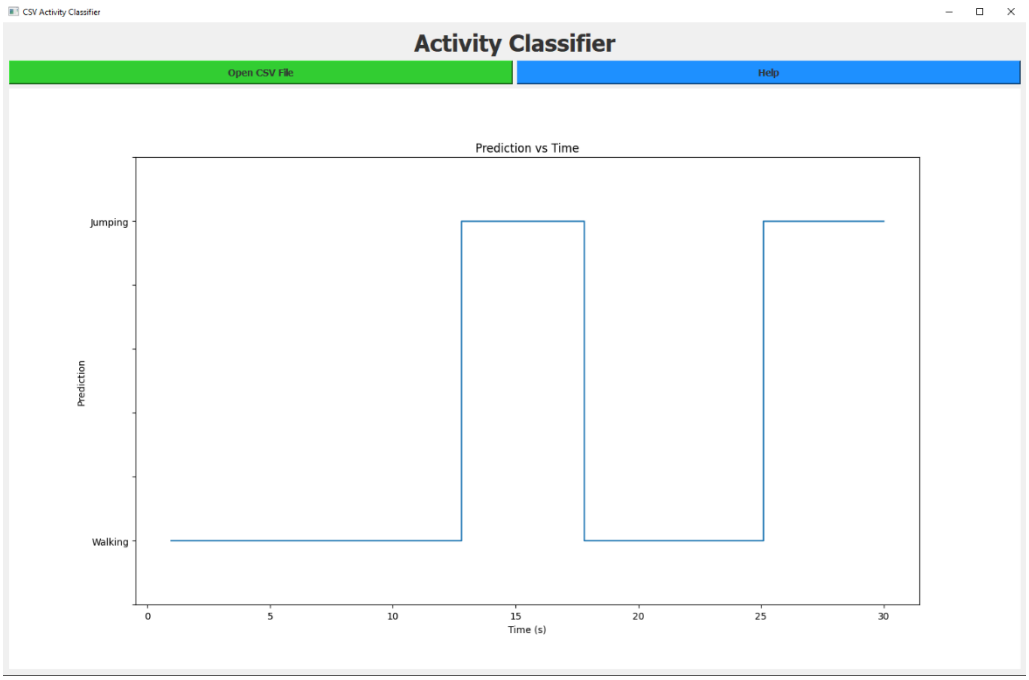

Figure 18: GUI once data has been loaded and processed.

If the user reloads another CSV for classification, the current plot will be cleared and the new plot generated. The app will continue to function as expected as new CSVs of data are analyzed and plotted without any crashes during final testing.

We tested the system by feeding it various CSV files containing walking and jumping motion data. The model was exposed to diverse datasets to ensure its robustness and accuracy in identifying the activities. After recording the input data, we compared the model's output with the duration of the activity types to ensure that the system correctly classified the motion data and performed as intended.

Real-Time Activity Classification (BONUS)

The primary objective of the desktop application was to enable real-time activity classification. This required that data be transferred from the mobile device to the desktop PC in order to display the correct classification. The original app served as a foundation, with the added capability of real-time classification.

Changes to the Desktop Application

Adding support for the real-time classification involved some changes to the original desktop app. One notable change was the addition of a third button alongside the existing Open CSV and Help buttons. When clicked, this new button launched a user interface similar to the main menu, facilitating real-time activity classification. Figure 19 shows the main menu of the updated UI with the “Launch Real Time Classifier” button.

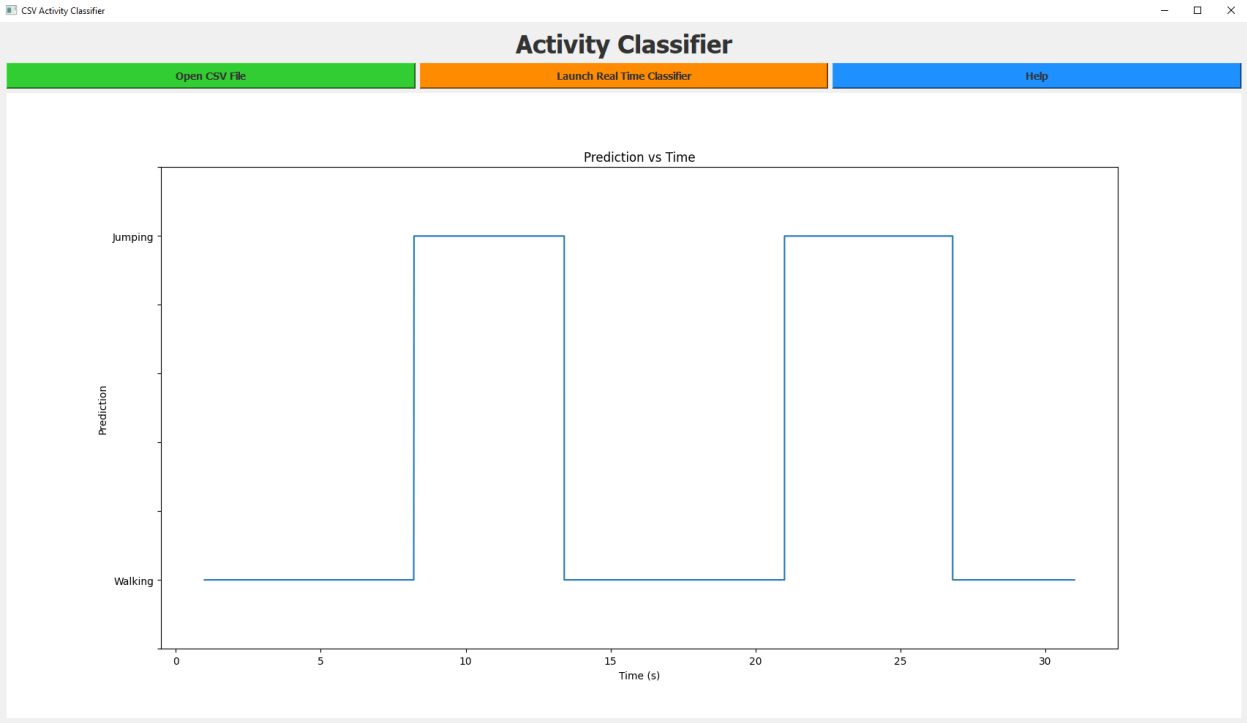

Figure 19: Screenshot of the new desktop application with the additional button "Launch Real Time Classifier"

Clicking the “Launch Real Time Classifier” button brings the user to a new interface shown in Figure 20. In this window, the user is prompted to enter the URL provided for Phyphox remote access, which connects the application to the remote access webpage integrated with the Phyphox app. The user can then press the "Connect" button to establish the connection. The application is designed to work without requiring the Phyphox remote access page to be pre-loaded in a browser. If the app fails to connect, the on-screen text updates to inform the user of the error and prompts them to retry.

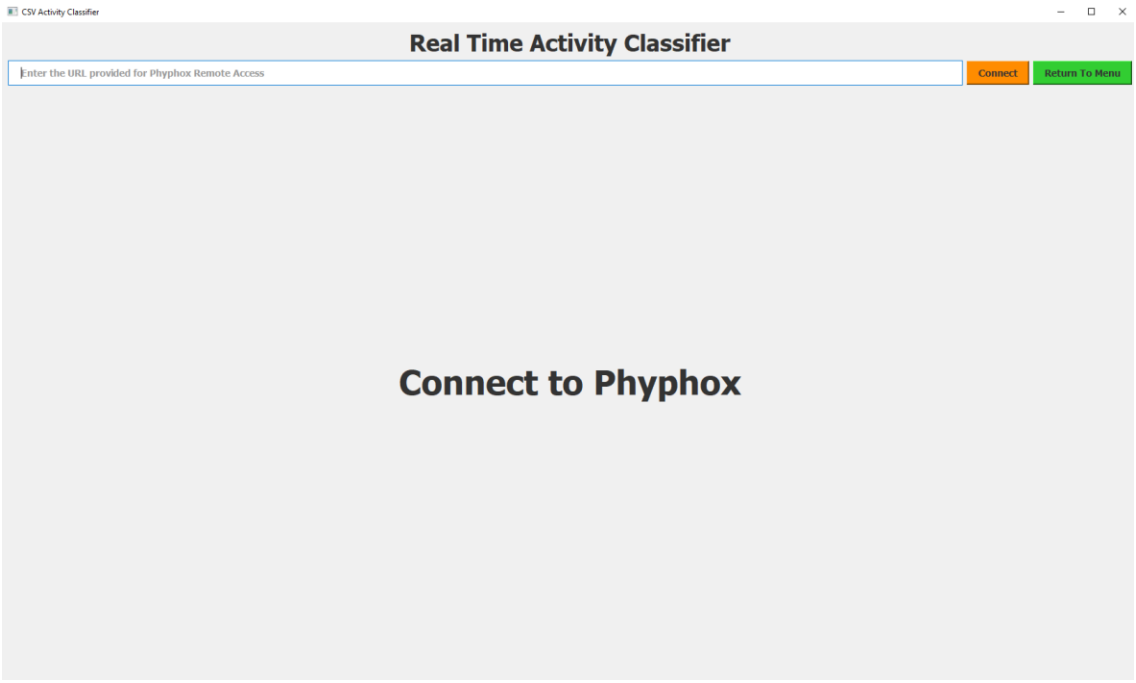

Figure 20: Screenshot of the Real Time Activity Classifier window when first launched

Once a valid URL is provided to the application, the classifier will start, and the predicted activity is displayed on the screen as shown in Figure 21.

Figure 21: The application outputs the current predicted activity in real-time

Navigating through the new application is simple and intuitive based on the descriptions shown on the buttons. Users can seamlessly switch between the main menu, CSV input for classification, and the real-time classifier without any issues, as the app has demonstrated stability during testing. Significant work was done on the backend of the app to ensure the correct functionality of all buttons and to eliminate any crashes or errors. Flags were used within the MainWindow class to execute specific lines only, when necessary, thus avoiding potential crashes.

Real-Time Data Transfer to PC

Implementing the real-time activity classifier required that the data produced by the accelerometer be transferred wirelessly to a PC running the desktop app. This was accomplished by using Phyphox's built-in remote access feature. When activated, this feature outputs the app data to a webpage at the URL provided within the app.

Using Selenium to Access HTML Code

The Phyphox remote access webpage is written in Javascript and contains multiple tabs to view the data including graphs for the x, y, and z linear acceleration, a graph for absolute acceleration, a multi-graph combining all four acceleration values, and a simple tab that only displays the numerical values of the accelerations. The Selenium library in Python was required to convert the Javascript source code to the HTML markup code needed to extract the data from the webpage. To do this, a selenium webdriver, based on the Chrome browser, was used to pull the HTML code of the webpage [8]. The "—headless" option was used to prevent a Chrome window from opening when creating the webdriver.

Using Selenium to Change the Webpage View

The default view of the Phyphox webpage when opened is the graphs of x, y, and z linear acceleration. Unfortunately, this page does not include numerical values of the acceleration, however, as shown in Figure 22, the "Simple" tab has these values as plain text. Selenium was used to change the current display by virtually clicking on the "Simple "Tab" to bring up the numerical values. The webdriver was used to locate the Simple button by XPath, and the Selenium click() function alters the webpage as though the user clicked the button with their mouse [9]. It is then possible to pull the page source code in HTML using driver.page_source, which contains the required values in plain text format.

Scraping the Phyphox Webpage Using Beautiful Soup

Since the desired values are now visible in plain text on the webpage, it is possible to use the Beautiful Soup library to filter the HTML code to pull the acceleration values. As shown in Figure 22, the acceleration values are contained within the div class "valueElement adjustableColor", inside the span "valueNumber".

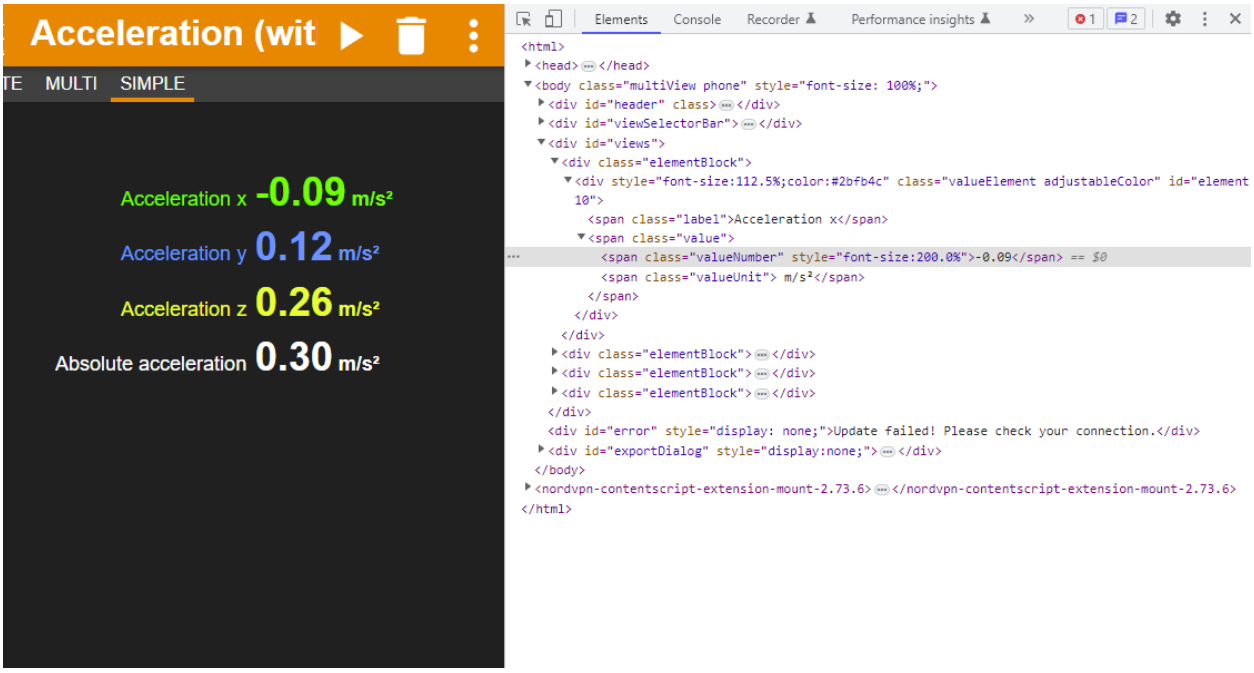

Figure 22: Screenshot of the Phyphox webpage, showing the Acceleration x value in the HTML code

The values were pulled using the find_all() function in Beautiful Soup using the parameters specified above.

Using the Acceleration Data to Predict Activity

Once the acceleration was successfully retrieved from the webpage, it can then be passed to the classifier for classification. To reduce false readings, the data was collected into a buffer and take an average of the results before sending it to the classifier. One challenge we encountered was the need to repeatedly pull data from the webpage for real-time classification. This was an issue since continually scraping the data from the webpage would use all resources allocated to the program, causing the GUI to become unresponsive. To solve this issue, a QThread was created to run the intensive task of scraping the acceleration data from the webpage, while the main thread ran the GUI [10]. The thread can then send a signal back to the main thread containing the data for the current classification, where it is classified using an average of the data. An average closer to 0 indicates walking, while an average closer to 1 signifies jumping.

Changes to the Trained Classifier

No changes were required to the classifier to allow for real-time classification. This allows the same model to be used for both the original and updated applications. When passing the data to the model for classification in real-time, it was ensured that the data frame column labels matched those that were used for training the model.

Participation Report

Section	Contributing Members
Data Collection	Ainsley, Ryan, Matthew
Data Storing	Ryan
Visualization	Ainsley
Preprocessing	Ryan
Feature Extraction	Matthew
Training the Classifier	Ryan, Matthew
Model Deployment	Ryan, Matthew
Bonus	Ryan
Report Formatting	Ainsley, Ryan

References

[1] Phyphox, "Phyphox Sensor Database," 12 April 2023. [Online]. Available: https://phyphox.org/sensordb/. [Accessed 12 April 2023].

[2] "W3Schools," [Online]. Available: https://www.w3schools.com/python/python_ml_getting_started.asp.

[3] s. k. gajawada, "Towards Data Science," 19 October 2019. [Online]. Available: https://towardsdatascience.com/anova-for-feature-selection-in-machine-learning-d9305e228476.

[4] A. Sharma, "Understanding Skewness in Data," Analytics Vidhya , 6 July 202. [Online]. Available: https://www.analyticsvidhya.com/blog/2020/07/what-is-skewness-statistics/.

[5] V. Rai, "Skewness And Kurtosis In Machine Learning," Medium, 27 January 2021. [Online]. Available: https://vivekrai1011.medium.com/skewness-and-kurtosis-in-machine-learning-c19f79e2d7a5.

[6] J. Frost, " Interquartile Range (IQR): How to Find and Use It," Statistics By Jim, [Online]. Available: https://statisticsbyjim.com/basics/interquartile-range/.

[7] J. Brownlee, "Save and Load Machine Learning Models in Python with scikit-learn," machinelearningmastery.com, 28 August 2020. [Online]. Available: https://machinelearningmastery.com/save-load-machine-learning-models-python-scikit-learn/. [Accessed 12 April 2023].

[8] L. Kwong, "How to scrape JavaScript webpages using Selenium in Python," 4 Feb 2022. [Online]. Available: https://lynn-kwong.medium.com/how-to-scrape-javascript-webpages-using-selenium-in-python-21d56731bb1f. [Accessed 13 April 2023].

[9] Geeks for Geeks, "How to click a button on webpage using selenium ?," 26 Nov 2020. [Online]. Available: https://www.geeksforgeeks.org/how-to-click-a-button-on-webpage-using-selenium/. [Accessed 13 April 2023].

[10] L. P. Ramos, "Use PyQt's QThread to Prevent Freezing GUIs," RealPython, [Online]. Available: https://realpython.com/python-pyqt-qthread/. [Accessed 13 April 2023].