Sit-To-Stand recognition using machine learning and wearable sensors

Name:
Theodore Bedos

Student number:
179330418
Degree program:
Integrated Mechanical and Electrical Engineering

Document:
MEng Final Project Report

Academic Year:
2022/2023

Supervisor name:
Dr. Uriel Martinez Hernandez

Assessor name:
Dr. Tom Fletcher

Word count:
12716

Abstract

This report investigates the effectiveness of various machine learning models in detecting Sit-to-Stand transitions using different combination of wearable sensor, window size and data format. Among all models, the RF model performed the best due to its robustness against overfitting and excellent adaptability. The optimal combination of parameter was to use raw data from all sensors combined and a 0.05s window of analysis. SVM performed better with larger window and the longer method to train by far. Both MLP and CNN showed the strongest potential for further optimization through hyperparameters tuning. KNN had less variation in its accuracy across all combinations compared to other methods, however, its best performance was slightly below other methods. Interestingly, raw data led to higher accuracy when combined with gyroscope data, or when used with all sensors, across all models, while feature extraction proved more beneficial with accelerometer and EMG data. This could be attributed to the more complex, time-dependent information captured by gyroscopes that can get oversimplified during feature extraction. The study also found that smaller window sizes surprisingly did not decreased performance, and even exhibited the best accuracy when using raw data for all models except for SVM, potentially due to the increased quantity of training data. However, feature extraction did not seem to be well-suited for small window sizes. Again, due to the tendency to oversimplify the data. This report also highlights some potential limitations related to the database used and suggests future research directions, including more rigorous hyperparameter tuning, experimental feature selection, and further testing with Stand-to-Sit transitions.

Acknowledgments

I would like to extend my sincere gratitude to my supervisor, Dr. Uriel Martinez Hernandez, who has been incredibly flexible and approachable throughout this project. Your guidance and friendly conduct made this project more enriching and manageable.

A special mention to my parents, to who I owe my achievement as my journey here would not have been possible without them.

I also want to express my appreciation to Grace Maxwell. Your constant support and reassurance have been greatly beneficial during this process.

To all of you, thank you for your part in this journey. Your contributions have not gone unnoticed, and I am truly grateful.

Table of Contents

1 - Introduction

1.1 - Motivations

The act of transitioning from a seated to a standing position, known as the Sit-to-Stand, and transitioning from a standing position to a seated position, known as Stand-to-Sit, are fundamental parts of daily human activities, as they are necessary to begin and end most activities. However, as people age or if they suffer from certain medical conditions, this seemingly simple task can become challenging. Nearly 35% of adults aged 65 years and older experience difficulty or inability to perform these transitions, and this percentage increases with advancing age [1]. Certain medical conditions can exacerbate this difficulty. For example, more than 60% of U.S. adults with arthritis report being unable to perform sit-to-stand transitions due to their condition, and people with Parkinson's disease often struggle with these transitions due to bradykinesia and muscle rigidity [2][3].

The difficulty with sit-to-stand and stand-to-sit transitions could have a significant negative impact on an individual's quality of life and independence. A study found that these difficulties were associated with lower physical function and quality of life among older adults [4]. These challenges also have profound repercussions on mental health, with research showing that elderly individuals with impaired mobility, including difficulty with these transitions, have a significantly higher risk of depression [5]. Moreover, these struggles with sit-to-stand transitions can precipitate serious physical health complications. For older adults, the difficulty with these transitions is a primary factor contributing to falls [6][7][8]. Alarmingly, they are three times more likely to experience falls than those without such difficulties [9], to various physical health issues. Indeed, the World Health Organization reports that falls are the leading cause of injury-related deaths among the elderly, accounting for an estimated 684000 fatalities globally each year [10]. Such falls are not merely fatal but can also cause severe injuries, like fractures. According to the National Osteoporosis Foundation, falls are responsible for more than 95% of the over 300,000 hip fractures that result in hospital admissions in the U.S. annually [11]. Moreover, a significant portion of falls result in a situation where an elderly person is unable to get up from the floor, often for at least an hour. A study in the Journal of the American Geriatrics Society found that 47% of falls in older adults resulted in such situation which can be associated with serious injuries, fear of falling, and loss of independence [12]. The dramatic consequences of falls among the elderly, therefore, underscore the importance of addressing and managing sit-to-stand transition difficulties.

In order to address this important problem, promising solutions have started to emerge based on the detection of sit-to-stand transitions using supervised machine learning. Supervised machine learning being a branch of Artificial Intelligence that involves training a model on a dataset of classified examples to enable it to make accurate predictions when confronted with new, unseen data of the same type. The solutions based on this technology can be broadly classified into two types: preventive and assistive.

Preventive solutions focus on early detection of physical ability deterioration. An accurate detection and analysis of Sit-To-Stand transition can provide valuable information on an individual's health condition and the progression of certain diseases. For instance, research has shown that these transitions can be particularly revealing for conditions like Parkinson's disease and other neuromuscular disorders [13]. This involves consistent monitoring of sit-to-stand performance using non-intrusive sensors like force plates and applying Machine Learning algorithms to detect subtle changes indicating a potential risk of falls. For instance, Wu et al. [14] integrated a force plate into an intelligent walking stick, Šantić et al. [15] incorporated them into shoes, and Arcelus and colleagues [16] placed them under matrices. These methods operate on the entirety of the data available, focusing on detcting the precise moment of position change. This enables the extraction of valuable information about each phase of the transition, including aspects such as duration, speed, and stability. As such,

they are not constrained by time, allowing for a comprehensive and accurate assessment of the individual's physical capacity.

On the other hand, assistive solutions aim to enhance the mobility of individuals already struggling with these transitions. This could be achieved through intelligent mobility aids, such as exoskeletons. These devices are equipped with wearable sensors that supply data to the Machine Learning model. The model, in turn, predicts the current position of the user and provide the necessary support to complete the transition safely and effectively. These devices, such as the HAL-3 exoskeleton [17], have been effectively utilized in rehabilitation settings, allowing patients to regain mobility earlier in their recovery process. However, these tools are bulky and restricted to controlled medical environments. Nonetheless, recent advancements in actuator technology and material sciences have given rise to more lightweight and flexible exoskeleton solutions [18]. In the near future, we may witness the development of more accessible options specifically tailored for older adults. For that to happen, progress need to be made on the software side to develop a solution offering an optimal balance between detection speed and accuracy. In contrast to preventive applications, these assistive devices necessitate real-time detection. Any delay in response can not only cause discomfort, but also potentially disrupt the user's natural rhythm of movement. Conversely, insufficient accuracy might result in serious injury, particularly if the user is forced to follow a sudden and unexpected movement. The performance of Machine Learning software depends on various factors, including the quality and volume of the database used for training, the type and number of sensors, the amount of data used for a classification (window), the methods employed for data cleaning and preparation (pre-processing), the data format (feature extraction), the type of Machine Learning algorithm, and the specific hyperparameters of each algorithm. Despite the growing body of research in this field, there remains a lack of consensus regarding the optimal combination of these factors for recognizing Sit-To-Stand transitions effectively.

1.2 – Aims and Report Structure

The aim of this paper is to offer a performance comparison of the combination of different Machine Learning models, wearable sensors, window size and data format (see Table 1) for the real-time detection of sit-to-stand transitions. By improving the understanding of the effect of these elements on the performance of Sit-To-Stand detection software, this report hopes to contribute to the development of intelligent mobility aids that can help people with mobility issue to regain independency. With this objective in mind, the hardware, software and database resources used in this project where select with the intent to facilitate reproducibility and understanding of this work.

The report is organized as follows, the 'Related work' provides thorough understanding of the fundamentals of machine learning, the specifics of each algorithm, and how they can be applied to time-series data from wearable sensors. This is followed by 'Resources', which describe the hardware, software, and databases used in the project. The 'Method' section describes the pipeline of the project. This consists of data visualization to understand the relationship between actions and patterns in the data, re-labelling to correct inconsistency in the data, feature extraction to convert the data in a more efficient format, models implementation to describe the implementation of the different Machine Learning model and their hyperparameters, and finally validation to obtain the results. The 'Results' section presents the performance of the models for the different combination of parameters. This is followed by a 'Discussion' of the results, limitations and potential future works. In the 'Conclusion' section, we summarise the key findings of the study and their implications.

Model	Sensors	Window size	Data format
SVM	Accelerometer	0.05	Raw
MLP	Gyroscope	0.075	Features
CNN	IMU (Acc & Gyro)	0.1	
KNN	EMG	0.15	
RF	All	0.2	
		0.3	
		0.5	

Table 1: List of all parameters to be investigated

1.3 - Theory Background:

1.3.1 - Wearable sensors

Intelligent mobility aids integrate embedded sensors to monitor ongoing movements. These include IMUs which include accelerometers and gyroscopes and provide acceleration and angular velocity data. Their compact size, light weight, and affordability make them a popular choice for such applications. However, IMU signals can be compromised by noise due to electrical interference and sensor bias. Gyroscopes in particular are prone to drift errors, where small measurement errors accumulate over time. These issues may necessitate advanced filtering techniques to ensure data accuracy.

Electromyography sensors (EMGs), on the other hand, measure the electrical activity of muscles during contraction. This can potentially offer early detection of a user's movement intent before it is reflected in the IMU data, allowing the intelligent mobility aid to provide support or even substitute for leg action. The intricate patterns of muscle activation and the numerous physiological and biomechanical factors influencing them provide personalized insights, which are beneficial for preventive applications. However, this complexity makes it challenging to establish consistent baseline values across different sessions or individuals, and in creating a highly adaptive model. Like IMUs, EMG data can also be subject to noise.

1.3.2 - Machine Learning Algorithms

a) SVM

SVM is a widely used supervised learning model in machine learning. Its principle consists in finding hyperplanes in a multidimensional space that separates different classes of data with the largest possible margin. The data points that are closest to the hyperplane, known as support vectors, determine this margin. SVMs are effective in high-dimensional spaces and are resistant to overfitting. SVM are known to perform optimally with smaller datasets. However, the training process can be time-consuming and may demand consequent computational resources compared to other models. In the context of sit-to-stand transition detection, SVM has demonstrated its effectiveness.

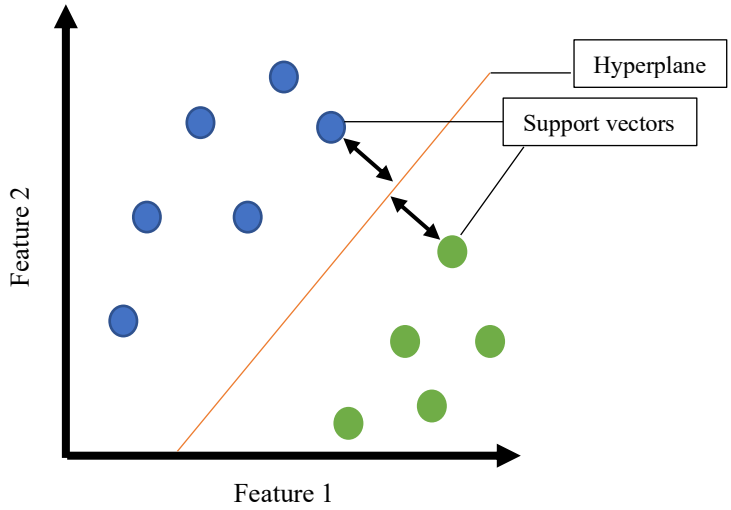

Figure 1: SVM diagram with data points as circles of different colors for different class.

b) MLP

MLP are a form of ANN that consists of layers of interconnected nodes also called neurons. The MLP is structured with an input layer, one or more hidden layers, and an output layer. The size of the input and output layers are respectively determined by the dimensions of the training data and the number of classes to be identified. The number of hidden layers and their nodes is determined by the complexity of the dataset, too many may lead to overfitting, where the model over adapts to the training data and performs poorly on unseen data. On the other hand, too few neurons can result in underfitting, where the model fails to capture the relationships in the data.

Every neuron in a layer is connected to all neurons in the adjacent layers, each has a specific weight assigned to it. Through a process called Gradient Descent, these weights are incrementally adjusted to minimize the “loss function” that is related to the difference between the model's outputs and the known correct outputs. The speed at which the weights are adjusted is dictated by a hyperparameter known as the “learning rate”. If set too high, the learning process may overshoot the optimal weight configuration; if set too low, the optimal weights may never be reached. Additionally, hidden and output layers have an activation function (e.g., ReLU, tanh, sigmoid) that determine if a neuron should be activated or not, and allows the model to create non-linear relationship between the input and output.

The capacity of MLPs to learn from and adapt to complex datasets contributes to their popularity. However, the multitude of hyperparameters inherent to MLPs presents a challenge when it comes to finding the most effective configuration. Consequently, it takes considerable efforts to the optimize these parameters for each application.

MLPs have been utilized effectively in sit-to-stand detection. One such example can be found in the work of abul et al. (XXXX), where an MLP was used to recognize Sit-to-Stand transition. The MLP model was trained with data from IMU sensors and demonstrated impressive accuracy with no false positive, highlighting the potential of MLPs to be implemented on intelligent mobility aids.

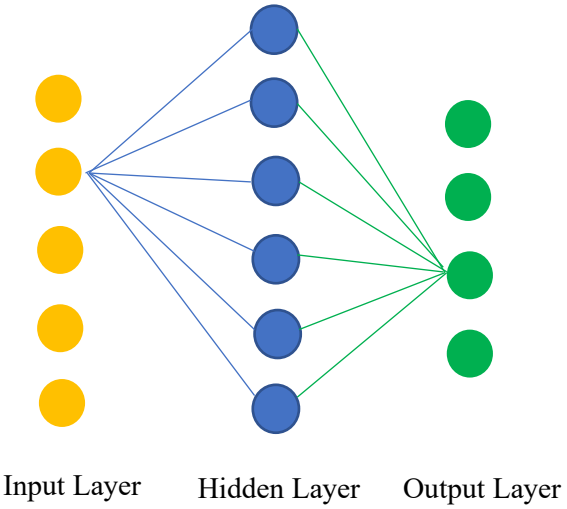

Figure 2: MLP diagram, with 3 layers

c) CNN

CNN are a type of ANN designed to process data with a grid-like format, making them particularly well-suited for image recognition. However, the data of time-series can also be organised as a grid where each row is a window and each column a feature or data sample of the window. As for MLP, CNN are composed of multiple layers of neurons, but they differentiate as the first layer is in fact a 1D kernel convoluting along the column as shown in figure X. The kernel extract features from each row, on which is applied a second layer called pooling layer to select certain features depending on the type of pooling layer. Multiple layers of convolutional and pooling layer can be applied to the grid depending on the dimensionality of the data. After each time the dimensionality of the data being reduced. The extracted features are then passed through a hidden layer similar to the one in MLP and output layer which number of neurons is the number of class in the classification problem. By recognizing patterns in local input patches and gradually assembling them into higher-level features, CNNs can effectively handle complex datasets.

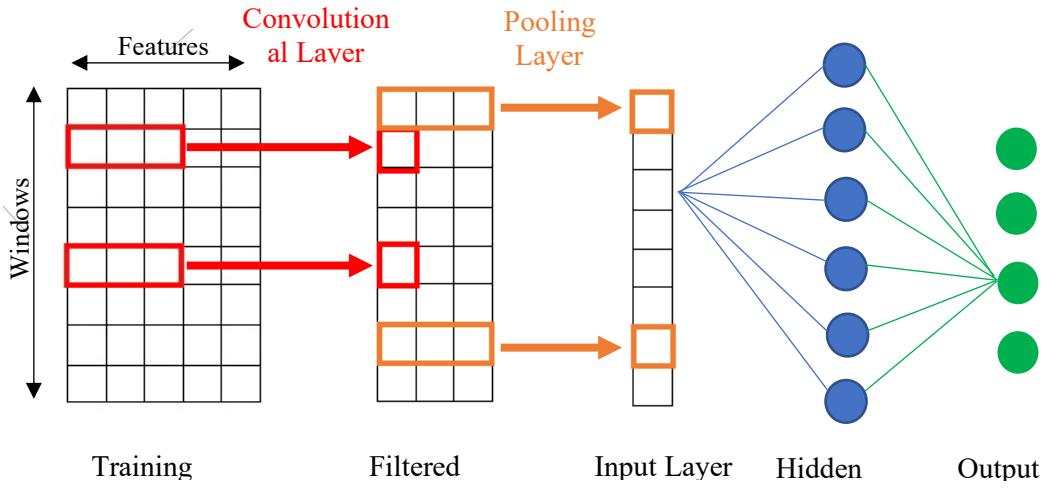

Figure 3: CNN applied to time series diagram.

d) KNN

KNN predict the class of new unseen window based on the classes of its nearest neighbors in the training dataset. The 'k' number of neighbors considered is the key hyperparameter of KNN model. The influence of the neighbors on the prediction is inversely proportional to their distance with the window being analysed. The distance of the neighbor can have more or less impact on the prediction depending on the distance metric used (e.g., Euclidean, Manhattan, Chebyshev, Minkowski).
When applied to sit-to-stand detection, KNN can be quite effective. For instance, in a study conducted by Gjoreski et al. (2016), a KNN algorithm was employed to classify different activities, including sit-to-stand transitions, using data from wearable sensors. The study found that KNN, in combination with feature extraction, can accurately detect sit-to-stand transitions.

e) RF

RF operates by building multiple decision trees during training, for each element of the training data, each tree predicts a class, and the most represented class is the output of the model. By combining the predictions of several trees compared to one, generalizability and robustness are improved. RF main strength is its resistance to overfitting. This due to a process called bagging which consist of training individual trees using different data sample. This way trees do not share the same floss and adapt better to knew unseen data.
Data samples are allocated to the training of each tree through bagging. Each tree consists of decision nodes that splits the training samples in branches leading to deeper nodes. Nodes that don't split and so end a branch are called leaf. The hyperparameters of RF are the maximum number of trees, maximum depth of a tree (number of nodes on a branch), minimum number of samples to split a node and minimum number of samples to create a leaf. A higher value for the two first parameter may increase performance but also the risk of overfitting. While a higher value for the two last increase adaptation but also the risk of underfitting.

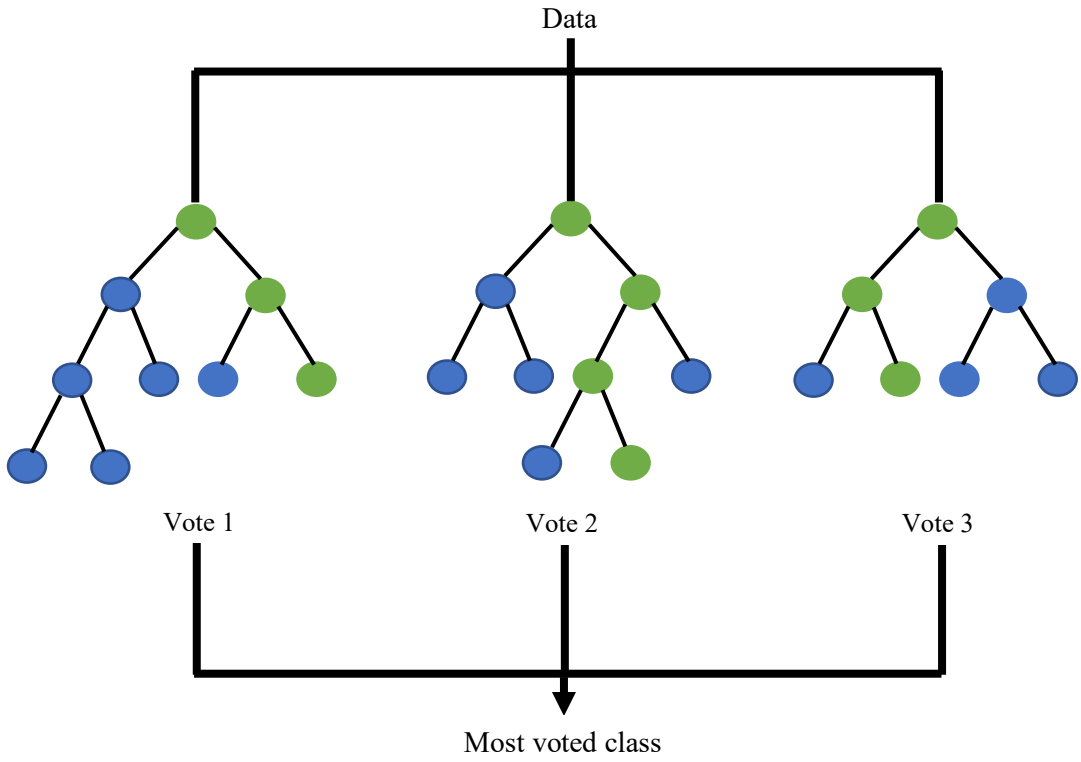

Figure 4: RF diagram with split nodes as dotes, green when voted for, blue when not voted for.

2 – Resources

The following section describe the resources necessary to reproduce the project.

2.1 - Hardware

The hardware resources necessary for this project were determined by the computational requirements of the machine learning algorithms and the size of the database used. The project's aim was to test various combinations of sensor arrangements, pre-processing methods, and machine learning algorithms to identify the most appropriate approach for detecting sitting and standing activities. By employing basic implementations of each method and a reasonably sized database, it was possible to gauge the performance of each combination. Consequently, a relatively high-quality personal computer, such as a MacBook Pro (Retina, 13-inch, Early 2015) was sufficient for training. The computer's specifications are provided in Table 2.
In addition, an external storage disk was required to store the sensor data and trained models. Utilizing an external disk ensured efficient data management and prevented strain on the laptop's internal storage. Storing data on an external disk also offered added security in case of computer failure.
The hardware setup was cost-effective and easily accessible, allowing the project to be reproduced without the need for specialized equipment or additional expenses. However, if more recent devices or specialized equipment are available, their use is recommended, as they can significantly accelerate the training process and enable experimentation with smaller windows. The properties of the external storage disk are detailed in Table 3.

Model	Retina, 13-inch, Early 2015
Operating System	macOS Monterey v12.1
Processor	2,7 GHz Intel Core i5 dual core
RAM	8 Go 1867 MHz DDR3
Graphic Card	Intel Iris Graphics 6100 1536 Mo

Table 2: Laptop specification

Model	UnionSine External Hard Drive
Digital Storage Capacity	1 TB
Connection Type	USB

Table 3: Hard drive specification

2.2 - Software

Python3 was chosen as the programming language for this project over MATLAB, another popular option known for its built-in functions for pre-processing, machine learning, and visualization. The decision to use Python was motivated by the goal of advancing Sit-to-Stand software development while allowing individuals to build upon this work with greater ease.
As an open-source language, Python is free to use, whereas the cost associated with MATLAB could be a limitation for those interested in continuing this work. Python can support multiple libraries, such as TensorFlow, Keras, and sklearn, specifically tailored for machine learning and data processing tasks, making it an ideal choice for implementing and comparing different machine learning algorithms. Moreover, Python benefits from a large community, resulting in plenty of support, and continuous updates. Python's readability and user-friendly nature make it more accessible to those new

to programming or with limited experience in MATLAB. Taking these factors into account, Python was deemed a more suitable choice for developing Sit-to-Stand detection software.

Anaconda was selected as the preferred platform for this project. It comes with pre-installed and up-to-date packages that offer valuable libraries, such as NumPy, matplotlib and csv. Although it is not strictly necessary to use Anaconda for the project, as each library can be installed separately, it provides a convenient way to manage these resources. The libraries utilized in the project are outlined in Table 5. Some of the libraries required for the project, such as TensorFlow (which includes Keras and Sklearn) and Joblib, were not pre-installed in Anaconda. Installing TensorFlow proved to be somewhat challenging, necessitating the creation of a separate environment within Anaconda rather than using the default base environment. Consequently, the base environment was duplicated to establish a new environment where TensorFlow could be installed.

Visual Studio Code, which is available through Anaconda, was employed as the Integrated Development Environment (IDE) for this project. However, the choice of IDE does not impact the project's outcome, as it mainly serves as a personal preference for the developer.

Coding Language	Python3
Environment	Anaconda
IDE	Visual Studio Code

Table 4: Software specification

Library	Version	Description
TensorFlow	2.10.0	Contain machine learning libraries, necessary for installing keras
Keras	2.10.0	Deep learning library for MLP and CNN
Sklearn	1.2.1	Machine learning library for SVM, KNN, RF
Numpy	1.23.5	To facilitate array processing
JobLib	1.1.1	To save and load SVM, KNN and RF models
csv	0.0.13	To facilitate reading in .csv file
MatplotLib	3.7.0	To display graphical results

Table 5: Libraries specification

2.3 - Database

Data collection was not necessary for this project, as the data was sourced from a public database called "Benchmark datasets for bilateral lower limb neuromechanical signals during unassisted locomotion in able-bodied individuals"[19]. This database contains recordings from 10 able-bodied individuals aged 23 to 29, measuring between 160 and 193 cm and weighing between 54 and 95 kg. The participants' genders are not specified, but the height range suggests a mix of men and women. The detailed weight and height of each candidate is shown in Appendix A.

Each subject was fitted with bilateral EMG sensors on seven distinct muscles (14 in total): tibialis anterior (TA), medial gastrocnemius (MG), soleus (SOL), vastus lateralis (VL), rectus femoris (RF), biceps femoris (BF), and semitendinosus (ST). Bilateral IMU sensors, which include separate accelerometer and gyroscope signals, were placed on the shanks and thighs, while one was attached to the waist (5 in total). Goniometers were positioned bilaterally on the knees and ankles. The setup is illustrated in Figure 5. Data was collected at a frequency of 500 Hz.

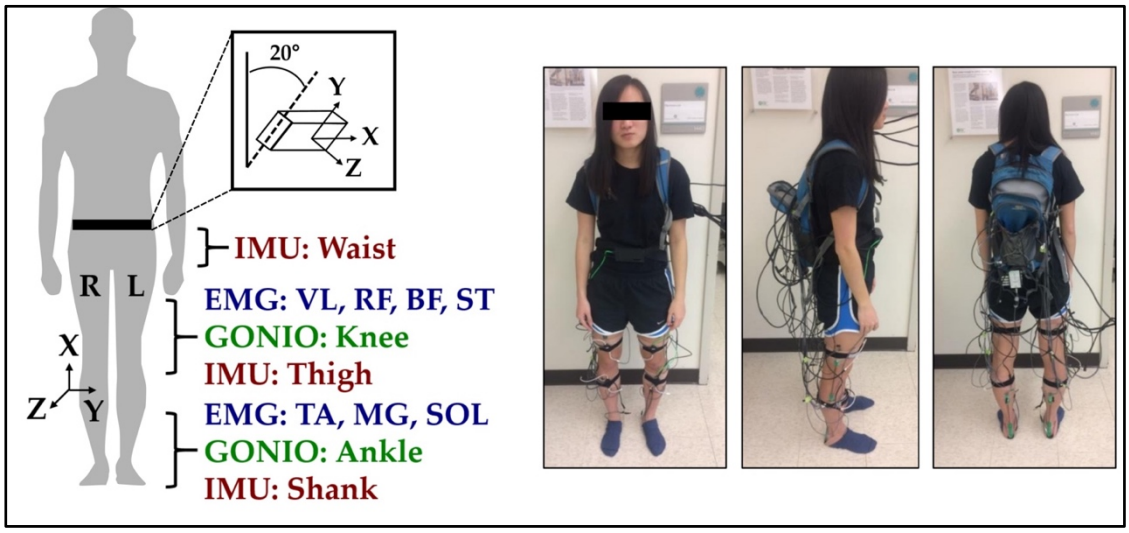

Figure 5: Sensor's disposition and orientation

Subjects began and ended each trial in a sitting position. In between, they were required to stand, navigate a circuit featuring ramp and stair ascent and descent, and then sit down again. Each individual completed 50 circuits. Each circuit is saved in a different .csv file. For each circuit, the data is available as "row" and "processed". The row data is the data without modification. In the post-processed data circuits that have been affected by pauses or trips were removed. The EMG signals were respectively high-pass and low-pass filtered with a sixth-order Butterworth at 20 Hz and 350 Hz, and notch-filtered with a sixth-order Butterworth (6 Hz width) at 60, 180, and 300 Hz to attenuate motion artifact and ambient interference. The goniometer and IMU signals were low-pass filtered with a sixth-order Butterworth at 10 and 25 Hz, respectively. Each sample of the data is assigned a label describing the current state according to Table 6.

Label	0	1	2	3	4	5	6
Phase	Sitting	Level ground walking	Ramp ascent	Ramp descent	Stair ascent	Stair descent	Standing

Table 6: Database gate labelling

3 - Method

In this section, we present the methodological approach employed in this project. The process consists of several stages, from data collection to state estimation using machine learning models. Together they form the project pipeline shown in Figure 6. The pipeline does not describe the flow of a single program but rather the different stages necessary to complete the project.

Figure 6: Project pipeline flow diagram

The two first steps of the pipeline can be grouped as pre-processing. It consists in refining and preparing the data, addressing any issues identified in the database. The labelling of the database only considers two states, sitting and standing. However, to be able to produce a smooth and adapted support, the most important phase to detect is the transition phase. Ideally, the transition phase should be segmented in multiple sub-phases. Therefore, the pre-processing is split into visualisation to understand the relationship in the data and relabelling to implement pattern recognition algorithm and relabel the data based on the patterns identified in visualisation. Following is Feature Extraction which allow to considerably reduce the size of the training data and transform it into a format more suitable for classification. This step is optional, and the models will also be tested using raw data. Then, the implementation stage consists in implementing all machine learning models and testing them with a range of hyperparameters to determine the default hyperparameters for the testing phase. Finally, all combination of Machine Learning method, sensors, window size and data format are tested with cross validation and saved. Each of these stages will be discussed in detail in the subsequent sections, providing a comprehensive understanding of the pipeline and the rationale behind each step.

3.1 - Visualisation

Data visualization is the initial stage of the pipeline. The objective of this stage is to familiarize ourselves with the patterns, trends, and relationships within the dataset. Recognizing these specific patterns enables us to implement methods for consistently segmenting the data into the different phases of the activity (see section 3.2). Furthermore, visual inspection of the data allows us to identify potential outliers, inconsistencies, or anomalies that may confuse our classification model, and subsequently apply appropriate cleaning methods to rectify these issues.

The visualisation was performed using the algorithm called *visualistion.py* (available in Appendix A). It allows to collect data from the database and graphically represent it. The data was displayed against time in seconds by multiplying the sample number by the sampling frequency (500Hz). The vertical blue lines on the graphs correspond to the change in label according to the database.

The first concerns with the database is the lack of explanation regarding how labels for standing and sitting phases are assigned. Thus, the first step of visualisation should be to understand which exact position of Sit-to- Stand and Stand-to-Sit triggers the default labelling.

As we can see in the vertical waist accelerometer data (Waist_Ay) shown in Figure 7, the beginning of standing according to the default label does not correspond to the beginning of the trunk forward tilt as we can see that the accelerometer data start changing before the peak reaches its top. This is confirmed by the sagittal plane waist gyroscope (Waist_Gx) shown in picture 8 and the quadriceps EMG (Right_VL) shown in picture 9, as both starts changing before the peak. Therefore, we can deduce that the default labelling correspond to the instant were the candidate lifts up off the chair. This may mean that the data was labelled manually based on recording of the candidates as this event is easy to detect visually.

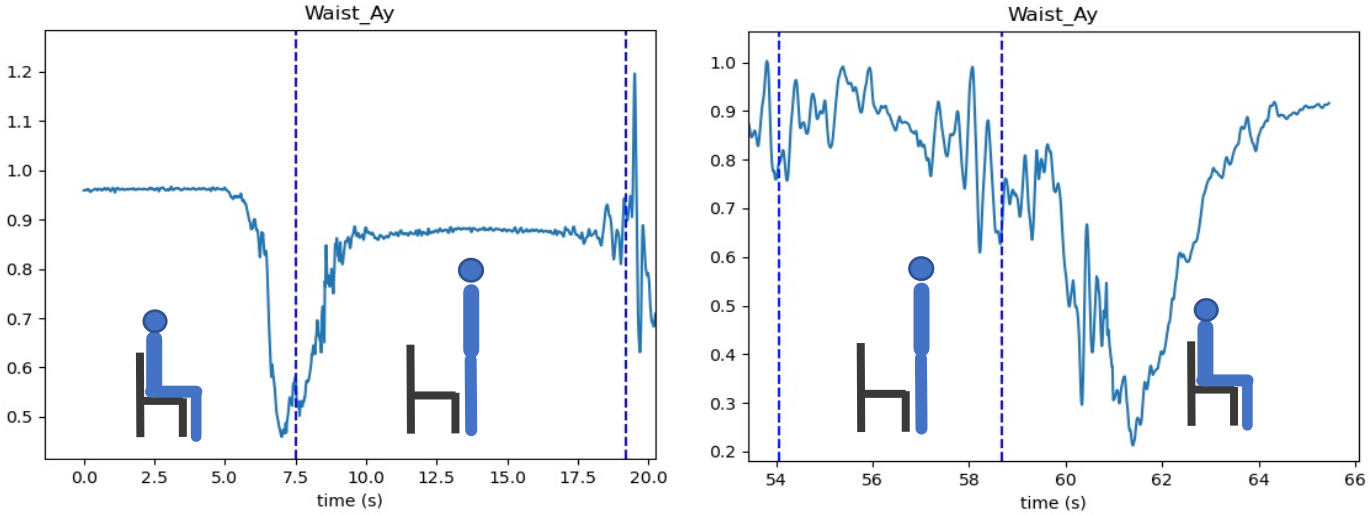

Figure 8: Vertical waist acceleration from accelerometer for Sit-to-Stand (left) and Stand-to-Sit (right). Segmented using default segmentation (blue dashed line)

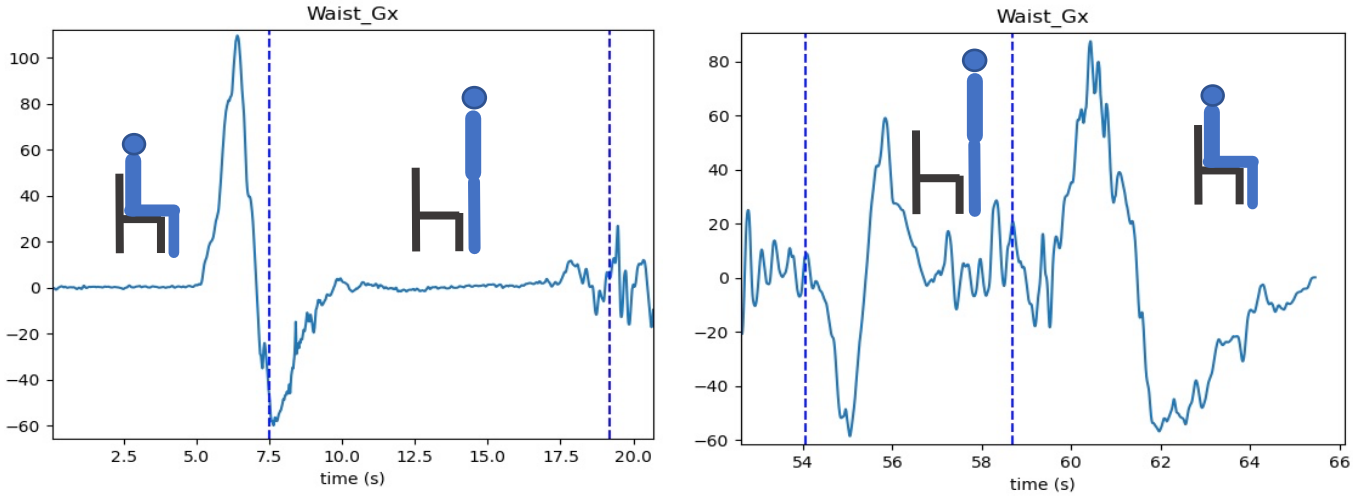

Figure 7: Sagital plane waist angular speed from gyroscope for Sit-to-Stand (left) and Stand-to-Sit (right). Segmented using default segmentation (blue dashed line)

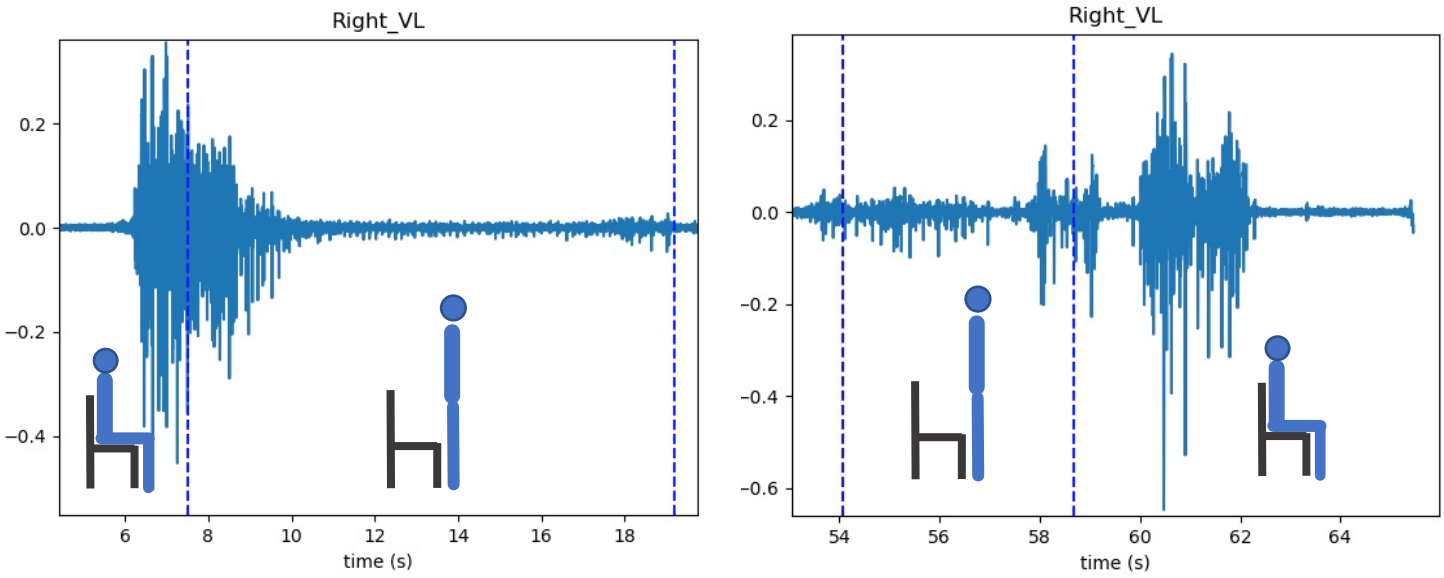

Figure 9: Right quadriceps signal from EMG for Sit-to-Stand (left) and Stand-to-Sit (right). Segmented using default segmentation (blue dashed line)

As the default labelling can't be used reliably to segment the data, the second step consist in finding the sensors with the most distinct patterns in order to relabel the data as precisely as possible. A good sensor must also exhibit patterns that allow to segment the transition phase in different sub-phases. The Waist_Ay and Waist_Gx shown in figure 7 and 8 present strong and distinct patterns. As Waist_Ay and Waist_Gx are attached to the trunk, they could be used to split the transition in leaning backward and leaning forward. However, it was found that the patterns were subject to too much variation across individuals to serve as reference for labelling.
As we can see in figure 9, the quadriceps starts activating before the body lift off the chair. The oscillation of the quadriceps signal could be used to determine the start and end of transition. However, it would not be possible to detect sub-phases in the transition. Another problem is the high amplitude variation across individuals due to different muscle mass and strength. Finally, the quadriceps might have stopped contracting that the body is still moving and inversely. Thus, it is unsure which level of oscillation correspond to the precise end of the movement.

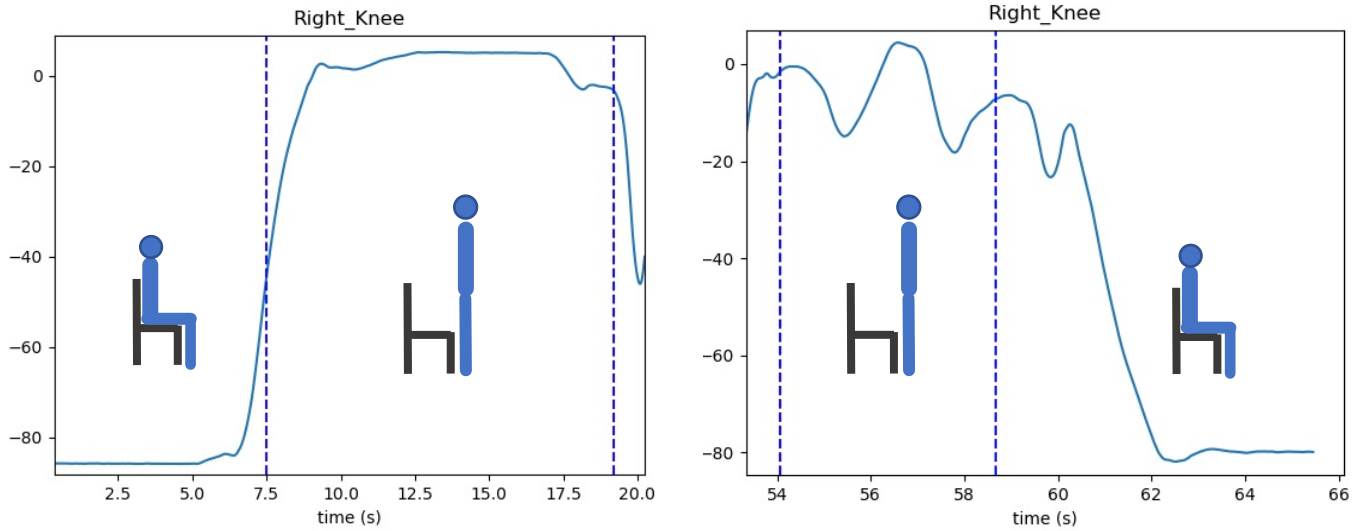

Figure 10: Right knee angle from goniometer for Sit-to-Stand (left) and Stand-to-Sit (right). Segmented using default segmentation (blue dashed line)

The goniometer sensors were not considered in the group of sensors used to train the model. However, the goniometer sensors, especially the knee ones have a clear signal, and distinct patterns subject to

low variations across individuals that can be used to relabel the database. The main slop in figure 10, correspond to the knee extension and can serve to determine the transition very accurately. However, it would not be possible to segment the transition phase in sub-phases as it doesn't translate the trunk motion.

The previous observations outline the need for a signal based on the trunk motion to be able to segment the transition. Therefore, the hip angle was calculated according to equation 1, 2 and 3 derived from R. Williamson and B.J.Andrews [19] work to calculate the knee angle [ref]. Figure 11 is used to derive the hip angle equation.
First, the initial hip and thigh tilt are calculated according to equation 1:

$$\theta_n(0) = \sum_{k=0}^{49} \tan^{-1}\left(a_{y,n}(k)/\left(a_{x,n}(k)\right)\right)/50 \quad (1)$$

Where:

$\theta_n(0)$: Initial tilt
a_y: Y-axis acceleration
a_x: X-axis acceleration
n: link

Second, the hip and thigh tilt are calculated by digitally integrating the sagittal plane gyroscope for the waist and thigh according to equation 2:

$$\theta_n(k+1) = (\omega_n(k+1) - \omega_n(0)) * dt + \theta_n(k) \quad (2)$$

Where:

θ_n: Tilt
ω_n: Angular speed
dt: time step

The hip angle is then obtained by subtracting the knee tilt to the hip tilt according to equation 3:

$$\phi_n = \theta_{n+1} - \theta_n \quad (3)$$

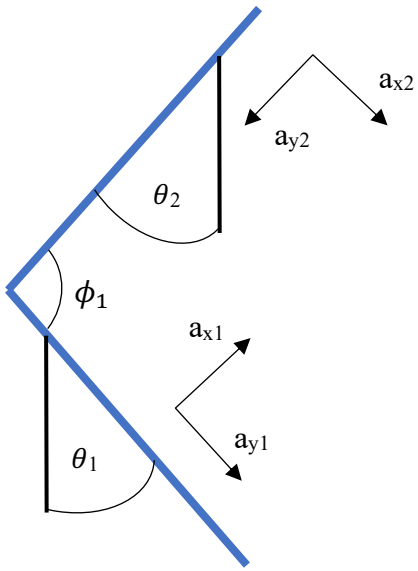

Figure 11: Hip angle and hip tilt digram

The resulting hip angle and hip tilt signals are shown in figure 12. There are two hip angle signals as the angle is formed with the thighs which are dissociated during walking.
The hip angle and hip tilt signal allows to split the data in 6 different phases as illustrated in figure 11. The first phase is the sitting phase, it ends when the trunk starts leaning forward, which translates to the start of the initial peak. The second phase is referred to in this report as Trans1 or first part of Sit-to-Stand transition, it ends when the trunk stops leaning forward and the knees start extending. It translates to the top of the initial peak. The third phase is referred to as Trans2 or second part of Sit-to-Stand transition, and it ends when then hip angle stops extending. It translates to the end of the slope after the initial peak. The fourth phase is the standing phase, it ends when the individual starts walking for Sit-to-Stand or when the individual starts leaning forward for Stand-to-Sit. This translates to the right and left hip signal splitting for Sit-to-Stand and to the start of the final peak in Stand-to-Sit. The fifth phase is referred to as Trans3 or first part of Stand-to-Sit transition, it ends when the individual stops bending is hip and knee to lower down. It translates to the top of the final peak. The sixth phase is referred to as Trans4 or second part of Stand-to-Sit transition and it ends when the individual stops leaning backward after being in contact with the seat. It translates to the end of the final peak. This segmentation is conserved through the report and used for relabelling.

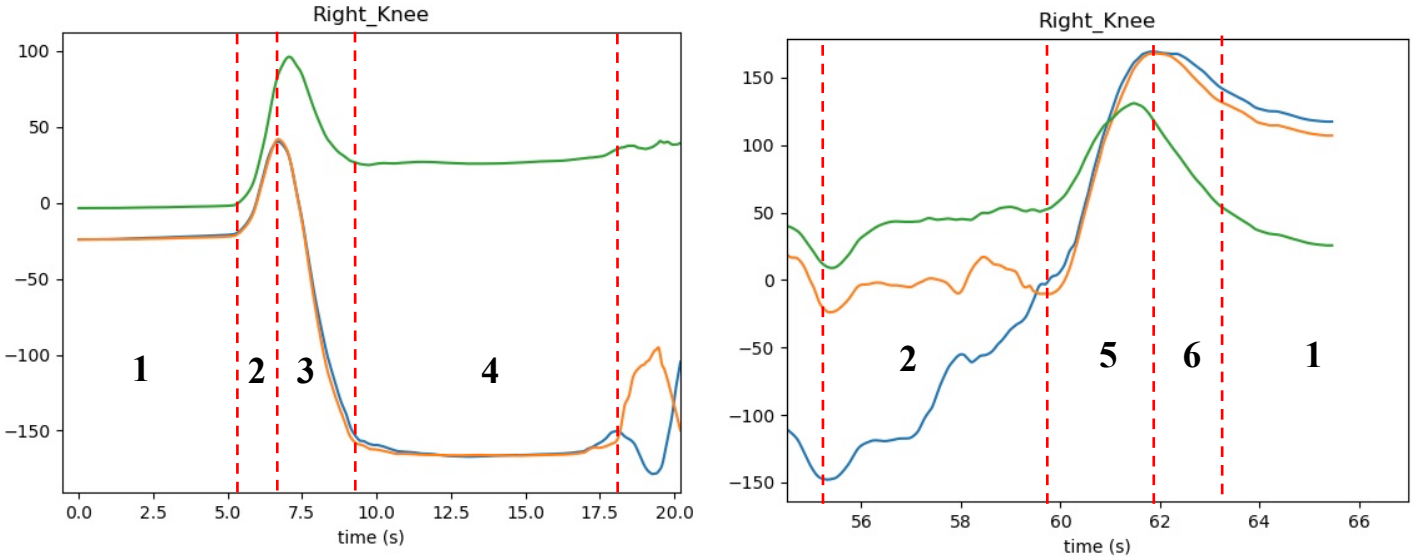

Figure 12: Hip tilt (green), right hip angle (blue) and left hip angle (orange) signal during Sit-to-Stand (left) and Stand-to-Sit (right). Segmented using new segmentation (red dashed line). Using circuit 6 of candidate 189.

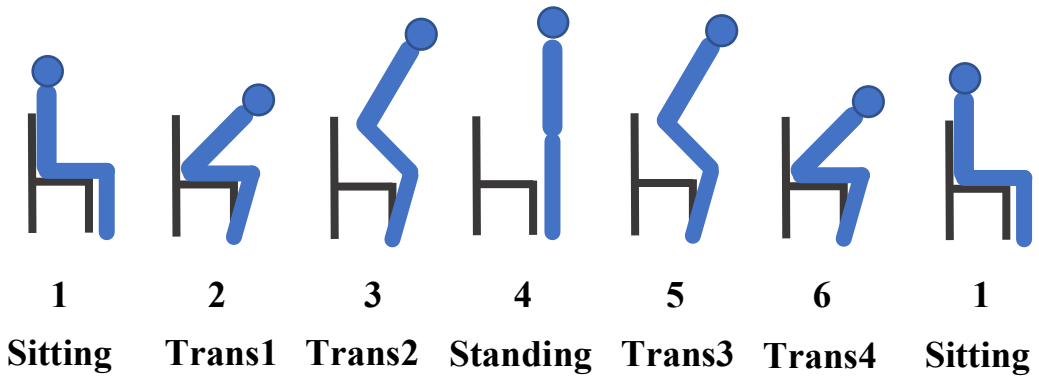

Figure 13: New segmentation for Sit-to-Stand and Stand-to-Sit

3.2 - Re-Labelling

In the previous section, we observed that the hip angle and hip tilt signals exhibit distinct patterns with minimal variations across different individuals. These patterns enable precise detection of the beginning and ending of each phase. The objective of this stage is to implement methods for identifying these patterns.

The pattern detection algorithms are incorporated into a program called *relabel.py*. This program iterates through every .csv file containing the data, relabels the data, eliminates actions that are not relevant to the project (LW, SA, SD, RA, RD) and saves the Sit-to-Stand and Stand-to-Sit data in separate files. This separation will be beneficial later on, simplifying the process of training and testing the models for each specific activity.

3.2.1 - Labelling algorithm:

a) Sit-to-Stand

2nd part of transition start: It corresponds to the end of the forward tilt of the upper body. At this point, the hip angle is the smallest and we can observe a peak on the hip angle signal. The index of the peak is obtained by finding in the right and left hip signal the index of the maximum value from the start of the signal to the end of standing according to the original labeling (second blue vertical line). The biggest of the two indexes is selected as the reference index. Choosing the largest index allows to include both right and left signal's peaks in the detection of the *1st part of transition start* and exclude the flat parts of the signal's peak to facilitate the detection of the *standing start.*
1st part of transition start: From the start of the signal to the *2nd part of transition start* index, it is the first index where the gradients of the right and left hip signals go above a certain threshold. Both signals don't have to be above the threshold at the same time. The gradient is calculated as it follows:

$$gradient = \frac{S(i+\Delta)-S(i)}{\Delta} \quad (4)$$

Where:

- S() : signal's value
- i : current index
- Δ : distance in sample
- S() : signal's value

Increasing the value of the threshold allows the algorithm to avoid triggering the pattern recognition for peaks that are not steep enough. This can happen if the data is constantly increasing or slightly oscillating during the sitting phase. A too-high threshold may result in not detecting the main peak. Another factor to adjust to avoid triggering the pattern recognition on noise is Δ. As peaks created by noise are short and sudden, increasing Δ, allows to not be calculating the gradient between the bottom and the top of small the peak. For this pattern, the threshold was set to 0.05 and Δ to 100 samples.

Standing start: From the *2nd part of transition start* index, it is the first index where the gradient of the right and left hip angle goes below a certain threshold. Both signals don't have to be below the threshold at the same time The gradient is calculated using equation 4. For this pattern, the threshold was set to 0.01 and Δ to 50 samples.

Standing end: It is detected using the difference between the right and left hip angle signals. If the difference between the hip signal starts increasing, it indicates that the candidate starts walking. However, even when standing still the difference is subject to variation that can trigger the pattern recognition. To avoid this small variation, the solution consists in finding the index where the difference is 1.5 times the difference at the *Standing start* index. However, for certain circuit where the candidate starts directly to walk such as shown in Figure 14, the initial difference is already important, thus, the difference may never increase by as much as half. To respond to this problem, if the difference goes below 1/10 time the initial difference, the end of standing index is the index where the difference goes above 1.2 times the initial difference. Inversely, if the initial difference is very small, a very small variation will trigger the recognition pattern. Thus, when the initial difference is below 25degree the index is the index where the difference goes above 25 degrees.

b) Stand-to-Sit

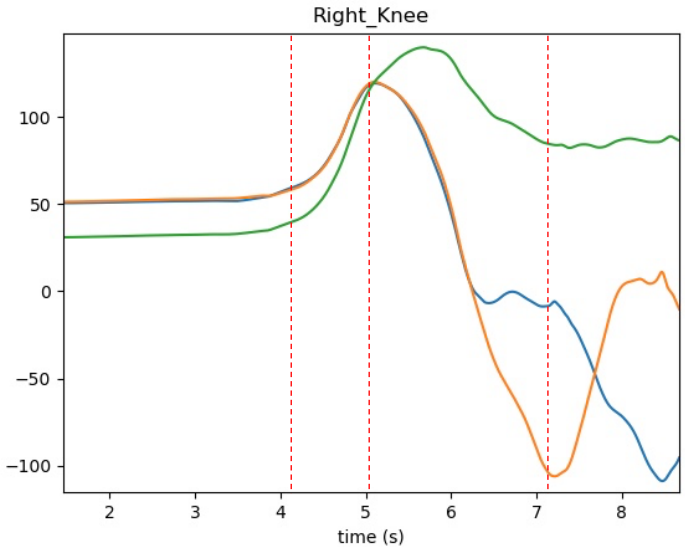

Figure 14: Circuit 6 of candidate 185 goes directly into walking

The hip angle and hip tilt signals were obtained by integrating over time the waist and thigh gyroscope signals. The integration process is sensitive to even the slightest inaccuracies in the gyroscope measurements. These inaccuracies can be caused by various factors, such as sensor noise, temperature changes, manufacturing imperfections, or electronic interference. As these small errors accumulate over time, they cause the estimated orientation to drift away from the true orientation, resulting in the so-called drift. Because Stand-to-Sit happens at the end of the circuit, the drift was significantly more important than for Sit-to-Stand. Therefore, extra conditions had to be implemented in the pattern recognition algorithms and the accuracy of the labelling was diminished.

1st part of transition end: Similar to the *2nd part of transition start* for Sit-To-Stand, the pattern that characterizes this stage of the Stand-To-Sit activity is the index at which the maximum value is reached. However, contrary to Sit-To-Stand the signal is a lot more affected by noise and constant errors. As we can see in Figure 15, the constant error in the gyroscope makes the signal increase despite being in a static sitting position. Thus, the maximum value of the signal is not the desired peak. The solution consists in first using the function *signal.peak()* from the *scipy* library to find the index of every major peak. Then, use the knee angle signal from the goniometer to find the index when the knee angle stops changing. This corresponds to when the knee signal reaches 95% of the range between the minimum and maximum value in the part of the signal between the start of standing (original label) and the end of the signal. Finally, the peak selected in the hip angle signal is the peak which index is the closest to the index found in the knee angle signal. As the knee angle from the goniometer is a reliable indicator of standing, as shown in figure 10.

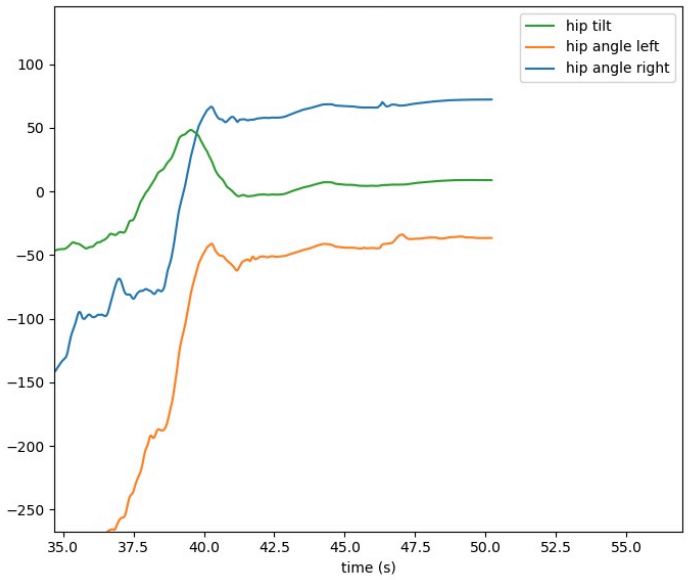

Figure 15: Hip angle and hip tilt signal increasing in sitting phase despite the candidate being static

Sitting start: Sitting should occur when the hip angle signal return to constant. However, due to accumulated error in the gyroscope signal, the signal may decrease while the candidate is in a sitting position. Thus, if the gradient threshold method is used with a low threshold, the gradient may never go below the threshold. If the gradient is set too high, the pattern recognition may be triggered to soon and not include the whole second part of Stand-to-Sit transition. The solution found is to adapt the threshold. If the minimum value from the *1st part of transition end* to the end of the signal is part of the 100 last samples, it means that the signal is constantly decreasing and therefore a higher threshold is used. A high threshold of 0.03 and a low threshold of 0.01 were used. The distance, Δ, was set to 150 samples to avoid noisy peaks that are bigger and occurs more often than for Sit-To-Stand. To reduce the risk of triggering the algorithm on a noisy peak, the right and left hip angle signal as well as the hip tilt signal must have gone below the threshold (not necessarily at the time).

Standing end: As mentioned previously, it is unsure whether small steps to turn and sit are included in the standing part of Stand-To-Sit. Consequently, the standing part which should be flat is often noisy as we can see in figure 12. Therefore, it is very difficult for a pattern recognition algorithm to not be triggered if we start from the beginning of the standing phase. Instead, we'd rather go backward from the *1st part of transition end* index which was clearly redefined earlier. The pattern recognition is triggered when the gradient of 2 signals out of the right and left hip angle signal and the hip tilt signal goes below the threshold. Depending on the subject the action of sitting can be unsmooth, thus it is common to observe some stops through the motion. These stops can trigger the pattern recognition algorithm. The solution was to find for each signal the range between the minimum value of the standing phase (as pre-labelled) and the value at the *End of transition phase 1* index, and only allow the gradient to be checked against the threshold if the value of the signal was below 30% of the range. Because the bottom of the slop can be a very sharp negative peak, the gradient threshold was set to 0.08 and Δ to 50 samples to make sure to detect it.

Standing start: Because the start of standing isn't clearly defined in the experiment, no pattern could be identified with enough consistency through the circuits and candidates to re-label the data with certainty. It was decided to conserve the original labeling.

3.2.2 - Data Cleaning

Despite spending effort visualising the data and testing the pattern recognition algorithms to try to comprehend for as many inconsistencies in the data as possible, *relabel.py* was interrupted multiple time while looping through the files due to singularities that went undetected during the visualisation phase. Some singularities could be corrected, other could not and add to be removed from the data base. Here is a list of the most recurrent errors and how they have been mitigated:

- **Gyroscope sensors inverted**: As we can see in figure 17, the peak in the hip angle data that is characteristic to the *2nd part of transition start* in Sit-To-Stand is missing. This is due to the thigh gyroscope signal being inverted as we can see in figure 16. This is probably due to an incorrect set up of the sensor. This set up error was spotted on multiple candidates, among which candidate 185 and 189, surprisingly not on all of their circuits. Such error was detected by calculating the integral of the thigh gyroscope signal from the start to the end of standing index (original index). If the integral is negative, the signal was corrected by inverting it.

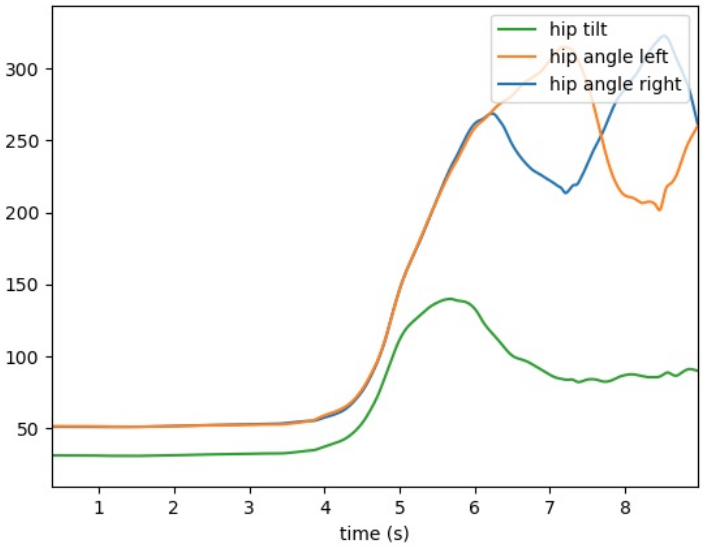

Figure 17: Hip angle and hip tilt error due to inverted gyroscope signal. Circuit 6 of candidate AB189

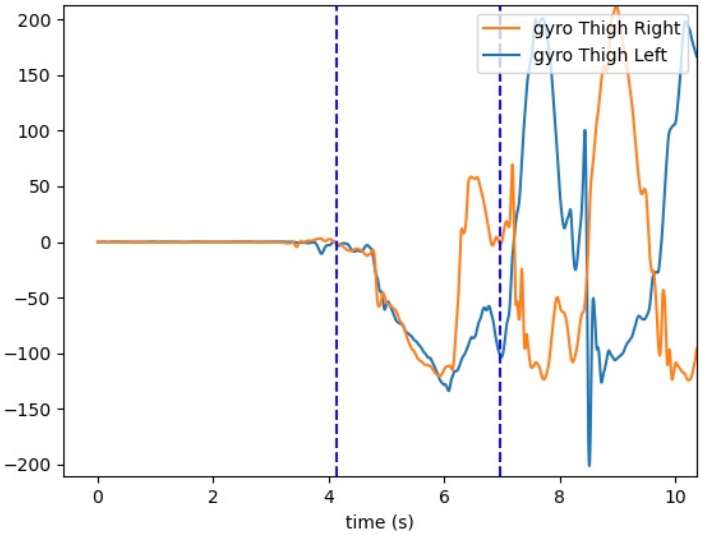

Figure 16: Inverted left and right thigh gyroscope due to set up error. Circuit 6 of candidate AB189

- **Incorrect labelling**: In candidate 1 of subject 185 sitting was labelled as 6 instead of 0 and standing as 1 instead of 6. In circuit 7 of candidate 190 sitting was labelled as 1 instead of 0. Because the algorithm was looking for a change in label between 0 to 6 or 6 to 1 to detect the index of the change of position, this led to an error as these transitions never occurred. It was resolved by looking for the first 2 changes in the labels starting from the beginning and the end of the signal regardless of the labels value.

- **Extreme drift in gyroscope:** As we can see in Figure 18, the hip angle signal displays the usual characteristics of Sit-To-Stand at the beginning. However, the longer it lasts, the less we can recognise a pattern. The fact that the Sit-To-Stand patterns are present at the beginning of the signals shows that the error doesn't originate from the incorrect positioning of the gyroscopes. This is due to abnormal drift of the gyroscope signal. The pattern recognition algorithm could not be adapted to operate with such signals, thus, these signals had to be removed from the database.

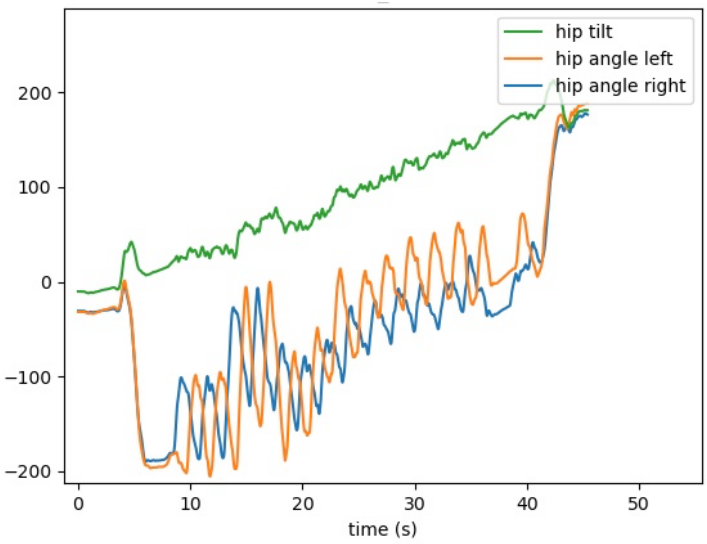

Figure 18: Extreme drift in gyroscope. Circuit 30 of candidate 190

3.3 - Feature Extraction

The next stage of the pipeline can be skipped to use raw data. It consists in splitting the data into windows and computing different statistical metrics based on the data contained in these windows. These metrics are called features of the data and help transform the raw data into a more concise, informative, and meaningful representation. Given the number of samples comprised in the window is superior to the number of features, it allows to reduce the dimensionality of the problem which consequently offers many benefits. Firstly, by working with a lower-dimensional representation of the data, computational complexity is reduced, making it faster and more efficient to process, analyse, and train machine learning models. This reduction in size and complexity becomes particularly important when dealing with large datasets or real-time applications, where computational resources may be limited. Secondly, feature extraction can help improve the performance of machine learning algorithms by highlighting the most relevant aspects of the data while reducing the impact of noise or irrelevant information. Ultimately, focusing on the most important features can lead to better generalization and robustness in the resulting models and consequently enhance their predictive capabilities.

Due to the significant number of parameters of interest, such as the activity, the machine learning method, the group of sensors, the window size, and the type of data (features or raw), testing each combination of parameters is already pushing the limits of the time available for this project. As a result, adding features to the list of parameters is not feasible. Consequently, a fixed number of features were selected based on existing literature [ref, ref]. The implemented features are as follows:

- Mean
- Maximum
- Minimum
- Standard deviation
- Variance
- Root Mean Square
- Skweness
- Kurtosis
- Interquartil Range

- Mean absolute deviation
- Peak to Peak
- Energy

To further reduce the dimensionality, sensors that measure movement out of the sagittal plan and so whose relevance is minor for the analysis of Sit-To-Stand were not considered. These are the Z-axis accelerometers for the thighs and shanks, the X-axis accelerometer for the waist, the X and Y-axis gyroscopes for the thighs and shanks and the Y and Z-axis gyroscopes for the waist. Figure 5 shows the axis for each group of sensor.

For every window, the above-mentioned features are extracted for every sensor. For instance, if the sensor selected is the accelerometer, which includes 10 sensors, there will be 130 features (13 x 10) per window.

As extracting features from candidates' circuits can be relatively long, especially with small windows, the features were saved in order to not have to extract them before training a model with *train.py*. For each candidate, all windows from his 50 circuits are saved in a .csv file. There is a file for each combination of activity, group of sensors and window size. This will facilitate training the models for different activities. At the end of each line of the .csv file the window is assigned a label. The label assigned is the one that was the most represented by the samples in the window.

3.4 - Model Implementation

3.4.1 - PCA

The PCA model was implemented in *train.py* using the function *PCA()* from the *sklearn.decomposition* library. The *PCA()* function can be set in two different ways, if the parameter value is superior to 1, the parameter value is the number of components of the PCA model and the information carried by the component will variate. Inversely, if it is in the range 0 to 1, the parameter value is the percentage of information carried by the components and the number of components will variate. For similar reasons than the number of features, it was decided to not experiment with the parameter of the PCA model. The PCA parameter was set to 0.99.

After loading the desired windows for the model that we want to train, the data is split between training and testing. The training data is used to fit the PCA model, then the training and testing data are transformed using the fitted PCA model.

3.4.2 - Normalization

Normalization was implemented using the function *MinMaxScaler()* from the *sklearn.preprocessing* library. *MinMaxScaler()* scales the data to fit the range 0 to 1 while preserving the shape of the original distribution. The minimum and maximum values of the data used to fit the model respectively become 0 and 1. The data is transformed using the following equation:

$$normalised\ value = \frac{value - min}{\max - min} \tag{5}$$

The normalisation model was fitted using the training data that has been transformed by the PCA model. Then the training and testing data (PCA transformed) are transformed using the fitted normalisation model.

3.4.3 - ML model:

The five models that were implemented are SVM, MLP, CNN, KNN and RF. As for the number of feature and PCA components, it was not possible to add the multiple parameters of each model to the list of parameters to test. Thus, the parameter of each model will be fixed while evaluating the other parameters. However, the parameters of each model can't be selected randomly as a selection that would not be adapted to the activity may affect the performance a model by such extant that it would lead us to have a wrong idea of the suitability of the model applied to Sit-To-Stand detection. Therefore, it was necessary to carry on a primal optimisation of the parameter. For each model, the combination of different parameters was investigated while using a fix 0.2s window and accelerometers sensors on the Sit-To-Stand data. The model was trained and tested using 10-fold cross validation, the parameter combination with the highest average accuracy over the 10-fold was selected as the default parameters of the model.
The following section describe the implementation of the 5 ML methods and the parameters selected:

a) SVM

```
# Create the SVM model
svm_model = SVC(kernel='rbf', C=1, gamma='scale', cache_size=1000, probability=True)

# Train the model
svm_model.fit(X_train_pca_norm, Y_train)
```

Figure 19: SVM model implementation

The SVM model is implemented using the *SVC()* function from the *sklearn.svm* library. The model is instantiated with a 'rbf' kernel. The model is configured with a regularization parameter C set to 1, the gamma parameter set to 'scale', a cache size of 1000, and the probability parameter set to True for obtaining probability estimates. The model is trained by fitting the model with the training data (X_train_pca_norm) and the corresponding labels (Y_train).

The combinations of parameters in table 7 were tested:

svm_C	svm_gamma
0.1	**'scale'**
1	'auto'
10	0.1
	1
	10

Table 7: SVM default parameters in bold

b) MLP

```
    # Create a MLP model
    model = Sequential()    #Define the type of model
    model.add(Input(len(X_train_list[0])))      # Input layer
    model.add(Dense(32, activation = 'relu'))       # Hidden layer
    model.add(Dense(nb_class, activation = 'softmax'))      # Output layer
    model.compile(loss='categorical_crossentropy', optimizer=Adam(learning_rate=0.01),
metrics=['accuracy'])        # Create the model
    model.fit(X_train_list, Y_train, epochs = 10, batch_size = 15)      # Train the model
```

Figure 20: MLP model implementation

The MLP model as shown in figure X is implemented using the Keras library. *Sequential()* from *keras.model* is used to instantiate the model. It creates an object with easy-to-use function to create a neural network architecture by simply adding one layer at a time with *model.add()*, where each layer is connected to the previous one in a sequential manner. *Input* and *Dense* from keras.layers are used to define the type of layer. *Input* for the first layer, *Dense* for layer which neurons are connected to a previous layer, this are called fully connected layer.

The first layer added is the input layer, using the *Input* function and specifying the number of neurons to be the number of component out of the PCA model. Next, we add a hidden layer with 32 neurons using the Dense function and the ReLU activation function. The last layer is the output layer, its number of neurons is the number of classes in the classification problem. In this layer, we use the softmax activation function to generate probabilities for each class.

The model is compiled using the categorical cross-entropy loss function, which is suitable for multi-class classification problems, and the Adam optimizer with a learning rate of 0.01 to adjust the model's weights during training. We also specify the 'accuracy' metric to monitor the performance of our model.

Finally, the model is trained using the *fit()* function, with the PCA transformed and normalised training data (X_train_pca_norm) and the corresponding label (Y_train). The number of epochs is set to 10 and using a batch size of 15. The batch size determines how many samples are used in each update of the model's weights during training.

The combinations of parameters in table 8 were tested:

units (hidden layer)	activation (hidden layer)	learning_rate
16	**ReLU**	0.001
32	tanh	**0.01**
64		0.1

Table 8: MLP default parameters in bold

c) CNN

```
   # Create a CNN model
    model = Sequential() #Define the type of model
    model.add(Conv1D(filters=64, kernel_size=3, activation='relu', input_shape=input_shape)) #
Input layer
    model.add(MaxPooling1D(pool_size=2)) # Pooling layer
    model.add(Flatten()) # Flatten the output of the convolutional layer
    model.add(Dense(units = 32, activation='relu')) # Hidden layer
    model.add(Dropout(0.5)) # Drop out layer
    model.add(Dense(nb_class, activation='softmax'))  # Output layer
    model.compile(optimizer=Adam(learning_rate=0.001), loss='categorical_crossentropy',
metrics=['accuracy'])   # Compile the model
    model.fit(X_train_pca_norm, Y_train, epochs=10, batch_size=32, verbose=1)    # Train the
model
```

Figure 21: CNN model implementation

Similar to the MLP model, the CNN model is implemented using the Keras library and instantiated using the *Sequential()* class.

The first layer in the model is a 1D convolutional layer consisting of 64 filters, a kernel size of 3, and a ReLU activation function. The input shape is set to correspond with the number of components out of the PCA model. Following that, a max-pooling layer with a pool size of 2 is added to reduce the spatial dimensions of the input data while preserving essential features. The 'kernel_size' and 'pool_size' were limited by the number of components from the PCA model. They were respectively set to 3 and 2 to accommodate most of the training data. In the few case were the number of components was inferior to 4 the PCA model parameter was set to 4 instead of 0.99 to force having a minimum of 4 components. After the max-pooling layer, the Flatten function is employed to convert the data into a 1D vector, which serves as input for the subsequent fully connected layers. Next, a dense layer with 32 neurons and a ReLU activation function is added, followed by a dropout layer with a rate of 0.5 to mitigate overfitting.

The output layer, compilation, and training implementation share similarities with the MLP model, with the exception of the learning rate being set to 0.001 and the batch size to 32.

This time the number of neuron and activation function of the hidden layer were not included in the parameters to investigate as they had already been tested for MLP which is relatively similar. Instead, the number of filters in the convolutional layer was tested. The combinations of the parameters in table 9 were tested:

filters	learning_rate
16	**0.001**
32	0.01
64	0.1

Table 9: CNN default parameters in bold

d) KNN

```
    # Create the KNN model
    model = KNeighborsClassifier(n_neighbors=200 , metric='euclidean')

    # Train the model
    model.fit(X_train_pca_norm, Y_train)
```

Figure 22: KNN model implementation

The KNN model is implemented using the *KNeighborsClassifier()* function from the *sklearn.neighbors* library. The model is instantiated by specifing the number of neighbors and the distance metric. They were respectively set to 200 and 'euclidean', after testing different parameters as shown in table X. The model is simply fitted with the training data (X_train_pca_norm) and the corresponding labels (Y_train).

The combinations of the parameters in table 10 were tested:

n_neighbors	metric
5	**Euclidian**
10	Manhattan
20	Chebyshev
40	Minkowski
80	
100	
150	
200	
300	

Table 10: KNN default parameters

e) RF

```
    # Create the Random Forest model
    model = RandomForestClassifier(n_estimators=200, max_depth=40, min_samples_split=5,
min_samples_leaf=1, random_state=42)

    # Train the model
    model.fit(X_train_pca_norm, Y_train)
```

Table 11: RF model implementation

The Ranndom Forest model is implemented using the *RandomForestClassifier()* function from the *sklearn.ensemble* library. The model is set up with 200 decision trees, a maximum depth of 20 for each tree, a minimum of 5 samples required to split an internal node, and a minimum of 5 samples needed at each leaf node. The random state is set to 42 to ensure reproducibility of the results. To train the model, the fit method is called on the RandomForestClassifier object with the input training data (X_train_pca_norm) and the corresponding labels (Y_train).

The combinations of the parameters in table 12 were tested:

n_estimators	max_depth	min_samples_split	min_samples_leaf
y	5	2	**1**
100	10	**5**	3
200	20	10	6
400	**40**	20	10

Table 12: RF default parameters

3.4.4 - Performance Evaluation

```
start_time = time.time()                      # Start the timer
Y_predict = model.predict(X_test_pca_norm)    # Predict classes
time_all = time.time() - start_time           # Stop the timer
time_one = time_all/len(X_test_pca_norm)      # time for one prediction
```

Figure 23: Performance evaluation implementation

Evaluating the model is done using the *predict()* function, providing the test data (X_test_pca_norm). The trained model output an array of length equivalent to the test data, where each element is the predicted class for the corresponding window in the training data.

The average duration of an individual prediction is obtained by timing the duration of the *predict()* function and dividing it by the length of the testing data as shown in figure 23.

The predicted labels are then compared to the test labels to evaluate the performance of the model. For the MLP and CNN model, the predicted labels are encoded as one hot vector of length equal to the number of class and where each element of the vector is a percentage of the probability to be the class. Each element needs to be rounded to 0 or 1 first and then the vector converted to a decimal value before being compared to the test labels.

The *keras.metrics* library is used to evaluate the performance of the model. The accuracy, confusion matrix and classification report are obtained using respectively the *accuracy_score()*, *confusion_matrix()* and *classification_report()* fucntions as shown in figure 24.

```
# Evaluate the model
accuracy = metrics.accuracy_score(Y_test, Y_predict)
conf_matrix = metrics.confusion_matrix(Y_test, Y_predict)
report = metrics.classification_report(Y_test, Y_predict)
```

Figure 24: Performance metrics implementation

3.5 - Testing Procedure

The procedure consists in training and testing each model for each combination of the parameters shown in table 1. To do so the following procedure is executed.

First, all windows of the 50 circuits of each candidate are loaded from the files corresponding to the current combination of parameters into an array. Each element of the array containing all window of a candidate. The corresponding label are similarly loaded in another array. Then, the 10 elements arrays containing the data and the labels are split the same way into training and testing to perform a 10-fold cross validation where each fold is a candidate data. This way the testing data is completely new to the model and we can evaluate the adaptability of the model. Once split, the training and testing data are shuffled to avoid overfitting the model to one candidate if all its window were fed one after the other. The labels are shuffled the exact same way. The training and testing data are passed through the PCA and normalisation functions as described in Section 3.4.1 and 3.4.2. The result is an array where each row is a window and each column a normalised PCA component. Then, the model is created using the default model parameters described in Section 3.4.5 and it is trained using the normalised component training array and corresponding training labels. Given the normalised components testing array, the trained model outputs an array containing the predicted label of each window. The predicted labels are compared to the true testing labels and the performance metrics are calculated as explained in Section 3.4.4. If the accuracy of the model is the best accuracy obtained at this point the ML model, PCA model, normalisation model, testing data and number of the candidate whose data is used for testing are saved to be able to reproduce the performance or the best model. The model is then trained and tested using the 9 other folds. The average accuracy, average time per prediction and cumulated confusion matrix over the 10-fold, as well as the worst and best fold accuracy and confusion matrix are saved and will be used to assess the performance of the parameter combination.
The process is repeated for each model and combination of parameter in table 1.

4 – Results

The following section presents the accuracy of each model for all combinations of window size, data format, and sensor groups. Due to practical constraints, it's not feasible to detail the performance of every combination in this report. Therefore, the settings offering the best balance between accuracy and speed for each model have been selected for further exploration. It's important to note that the selection of extracted features was fixed and was based on literature rather than experimental evaluation. This means that the accuracy results for feature extraction, might not fully reflect the model's potential performance. Thus, the best performance using features is detailed independently of the comparison with raw data, providing valuable insight for future potential future work employing a more meticulous feature selection strategy.

The average accuracy (in percentage) and computational time (in seconds) over the 10-fold, the best and worth accuracy (in percentage), the number of the candidates that obtain the best and worst accuracy, and the computational time (in seconds) for the best accuracy, of each combination of the parameters in table X are listed in Table X of Appendix Y.

4.1 - SVM

SVM		Sensors									
		Features					Raw				
		Acc	Gyro	IMU	EMG	All	Acc	Gyro	IMU	EMG	All
Window	0.1	84,61	64,874	64,874	68,42	69,93	75,148	85,582	85,62	67,165	95,784
	0.15	84,673	62,089	62,092	67,073	73,455	75,99	85,593	85,571	67,054	95,795
	0.2	83,246	54,917	54,923	65,108	68,998	76,544	85,583	85,563	66,444	95,675
	0.3	83,325	52,989	52,989	64,547	66,21	79,348	86,317	86,254	65,173	96,038
	0.5	83,245	57,885	57,885	65,043	58,727	87,112	86,882	86,819	65,317	95,686

Table 13: Heat map of the average accuracy over 10-folds of SVM model for each combination of sensors, data format and window size. Green being the best accuracy and red the worst.

Table 13 shows that the highest accuracy for the SVM model was 96.038% and was achieved using raw data from all sensors combined and a 0.3s window. Interestingly, the smaller window of 0.1s had an accuracy that was only 0.254% lower. Hence, the performance of these two window sizes is further compared below to find if the marginal increase in accuracy justifies the longer delay.

As shown in Tables 14, both windows demonstrate comparable accuracy across all phases. The most challenging phase for both windows was the second part of the transition, which was often misclassified as standing. Both windows have similar consistency, with the 0.3s window having a slight edge, as its variation between the best and worst performance was only 4.44% compared to 7.71% for the 0.1s window. The major difference was in terms of computation time per sample. The 0.1s window took an average of 0.406ms to classify the samples, compared to 0.187ms for the 0.3s window. However, the combined window size and computational time for the 0.05s window was still approximately three times less than for the 0.3s window. Thus, it's reasonable to conclude that the minor accuracy improvement offered by the larger window does not justify the significantly increased delay.

SVM/raw/all/0.1		True			
		Sit	Trans1	Trans2	Stand
Predicted	Sit	99,35	0,63	0,03	0,00
	Trans1	5,06	92,66	1,78	0,50
	Trans2	0,01	0,88	90,74	8,37
	Stand	0,98	0,26	4,38	94,38

SVM/raw/all/0.3		True			
		Sit	Trans1	Trans2	Stand
Predicted	Sit	99,44	0,54	0,02	0,00
	Trans1	4,17	93,10	1,95	0,78
	Trans2	0,00	1,40	90,64	7,96
	Stand	1,02	0,20	3,61	95,17

Table 14: Cumulated confusion matrix (in %) over 10-folds for SVM with all sensors combined, raw data and 0.1s window (left) and 0.3s (right).

Avg Accuracy (%)	Avg Time (ms)	Best Accuracy (%)	Best Candidate	Worst Accuracy (%)	Worst Candidate	Best Candidate Time (ms)
95,78	0,460	98,00	6	92,29	4	0,494

Table 15: Average, best and worst performance over 10-folds for SVM with all sensors combined, raw data and 0.1s window.

Avg Accuracy (%)	Avg Time (ms)	Best Accuracy (%)	Best Candidate	Worst Accuracy (%)	Worst Candidate	Best Candidate Time (ms)
96,04	0,187	97,98	0	93,54	8	0,191

Table 16: Average, best and worst performance over 10-folds for SVM with all sensors combined, raw data and 0.3s window.

When using features, the most accurate results were obtained using accelerometer data. The accuracy using a 0.1s window was only 0.063% less than the best accuracy for this setup, 84.673%, which was obtained for the 0.15s window. Consequently, only the performance of the 0.1s window is further examined below.

Table 17 shows that the SVM model, using feature-extracted data, encountered the most difficulties when identifying the initial phase of transition, often identifying it as sitting. As seen in Table 18, the average computational time per sample was 1.178ms, ranking it among the slowest setups in terms of computational speed. With a difference of 14.88% between the best and worst accuracy, the model demonstrated a lack of consistency and potential difficulties in adapting to different individuals.

SVM/Features/ acc/0.1		True			
		Sit	Trans1	Trans2	Stand
Predicted	Sit	91,43	8,47	0,10	0,00
	Trans1	32,81	60,05	6,49	0,65
	Trans2	0,00	3,21	78,71	18,08
	Stand	0,95	0,33	9,90	88,82

Table 17: Cumulated confusion matrix (in %) over 10-folds for SVM with accelerometer sensors, extracted features and 0.1s window.

Avg Accuracy (%)	Avg Time (ms)	Best Accuracy (%)	Best Candidate	Worst Accuracy (%)	Worst Candidate	Best Candidate Time (ms)
84,61	1,178	91,11	7	76,23	8	1,300

Table 18: Average, best and worst performance over 10-folds for SVM with all sensors combined, features extracted and 0.1s window.

4.2 – MLP

MLP		Sensors									
		Features					Raw				
		Acc	Gyro	IMU	EMG	All	Acc	Gyro	IMU	EMG	All
Window	0.05	84,445	50,59	50,878	62,506	71,334	71,18	83,388	84,178	46,983	94,452
	0.075	84,299	49,37	49,64	61,892	70,601	73,507	84,921	80,432	47,06	95,042
	0.1	83,965	48,707	55,468	64,018	72,464	73,515	82,336	83,251	47,12	94,824
	0.15	84,033	54,4	53,306	62,433	74,407	75,413	81,107	81,854	47,294	94,779
	0.2	83,393	49,367	49,308	63,343	63,331	73,403	81,062	77,217	47,373	94,227
	0.3	83,632	51,699	51,556	61,812	60,882	79,87	80,402	79,33	47,655	94,395
	0.5	84,511	52,064	51,457	61,842	52,384	79,67	77,252	79,193	48,011	93,082

Table 19: Heat map of the average accuracy over 10-folds of MLP model for each combination of sensors, data format and window size. Green being the best accuracy and red the worst.

Table 19 shows that the highest accuracy for the MLP model was 95.042% and was achieved using raw data from all sensors combined and a 0.075s window. Interestingly, the smaller window of 0.05s had an accuracy that was only 0.59% lower. Hence, the performance of these two window sizes is further compared below to find if the marginal increase in accuracy justifies the longer delay.

Tables 20 reveal a comparable performance for both the 0.075s and 0.05s windows in detecting each phase, with a slight edge to the 0.075s window in all instances. The second part of the transition phase proves to be the most challenging for this model, often resulting in confusion with the standing phase. Both windows demonstrate similar computation time per sample, as shown in Tables 21 and 22. However, the 0.05s window shows less consistency in performance, reaching a best accuracy that is comparable to the 0.075s window, but dropping to as low as 85.85%, versus 92.05% for the 0.075s window. This suggests that the 0.075s window offers greater adaptability and potential for consistent and safe performance across all individuals. Therefore, we can conclude that the slightly enhanced performance of the 0.075s window justifies its increased delay inherent to the window size.

MLP/raw/all/0.05		True			
		Sit	Trans1	Trans2	Stand
Predicted	Sit	98,77	1,16	0,03	0,04
	Trans1	5,11	92,30	1,72	0,87
	Trans2	0,09	1,91	87,25	10,75
	Stand	1,13	0,22	5,58	93,07

MLP/raw/all/0.075		True			
		Sit	Trans1	Trans2	Stand
Predicted	Sit	98,73	1,21	0,02	0,05
	Trans1	4,55	92,94	1,86	0,64
	Trans2	0,00	1,28	89,45	9,27
	Stand	1,04	0,71	4,81	93,44

Table 20: Cumulated confusion matrix (in %) over 10-folds for MLP with all sensors combined, raw data and 0.05s window (left) and 0.075s (right).

Avg Accuracy (%)	Avg Time (ms)	Best Accuracy (%)	Best Candidate	Worst Accuracy (%)	Worst Candidate	Best Candidate Time (ms)
94,45	0,122	97,61	0	85,85	1	0,091

Table 21: Average, best and worst performance over 10-folds for MLP with all sensors combined, raw data and 0.075s window.

Avg Accuracy (%)	Avg Time (ms)	Best Accuracy (%)	Best Candidate	Worst Accuracy (%)	Worst Candidate	Best Candidate Time (ms)
95,04	0,152	97,19	6	92,08	4	0,107

Table 22: Average, best and worst performance over 10-folds for MLP with all sensors combined, raw data and 0.075s window.

When using features, the most accurate results were obtained using accelerometer data. The accuracy using a 0.05s window was only 0.066% less than the best accuracy for this setup, 84.511%, which was obtained for the largest window. Consequently, only the performance of the 0.05s window is further examined below.

Table 23 reveals that the MLP model, when applied to feature-extracted data, considerably struggled with the classification of the first transition phase, more than half of the time misidentifying it as the sitting phase. As indicated by Table 24, feature extraction did not compromise time performance compared to using raw data. The variability between the best and worst results was a moderate 10.6%, suggesting a reasonable level of adaptability compared to all configurations, and an average performance when compared with the best configurations discussed in this section.

MLP/Features/acc/0.05		True			
		Sit	Trans1	Trans2	Stand
Predicted	Sit	98,75	1,15	0,10	0,00
	Trans1	51,83	43,55	4,10	0,51
	Trans2	0,01	4,15	74,43	21,41
	Stand	1,04	0,20	14,65	84,11

Table 23: Cumulated confusion matrix (in %) over 10-folds for MLP with accelerometer sensors, extracted features and 0.05s window.

Avg Accuracy (%)	Avg Time (ms)	Best Accuracy (%)	Best Candidate	Worst Accuracy (%)	Worst Candidate	Best Candidate Time (ms)
84,45	0,153	89,87	2	79,27	8	0,146

Table 24: Average, best and worst performance over 10-folds for MLP with all sensors combined, features extracted and 0.05s window.

4.3 - CNN

CNN		Sensors									
		Features					Raw				
		Acc	Gyro	IMU	EMG	All	Acc	Gyro	IMU	EMG	All
Window	0.05	82,424	71,967	72,256	63,076	73,885	69,714	82,67	82,393	68,016	95,22
	0.075	83,761	64,062	64,817	58,791	71,839	74,961	81,892	82,083	69,419	94,941
	0.1	83,889	52,612	52,821	62,649	70,744	71,561	82,987	83,609	65,878	94,374
	0.15	82,975	50,795	50,749	59,227	71,426	67,724	83,658	83,781	65,943	94,029
	0.2	82,547	49,148	49,007	62,336	68,095	73,409	83,59	82,42	68,016	93,779
	0.3	81,932	49,66	49,991	61,035	50,289	72,102	82,401	81,038	66,422	94,425
	0.5	79,936	50,365	50,3	61,802	50,367	75,577	79,893	81,334	61,713	93,141

Table 25: Heat map of the average accuracy over 10-folds of CNN model for each combination of sensors, data format and window size. Green being the best accuracy and red the worst.

Table 25 shows that the highest accuracy for the CNN model was 95.22% and was achieved using raw data from all sensors combined and a 0.05s window. The detailed performance of this set-up is shown below.

Table 26 shows that the CNN model, had excellent performance with all phases. As indicated by Table 27, the average computational time per sample of 0.075ms was also excellent. The difference between the highest and lowest accuracy over the 10-fold was only 5.84% which makes this set-up one of the most consistent and demonstrate a high capacity to adapt to all individuals.

CNN/raw/ all/0.05		True			
		Sit	Trans1	Trans2	Stand
Predicted	Sit	99,03	0,89	0,05	0,03
	Trans1	5,81	92,08	1,94	0,17
	Trans2	0,00	1,46	90,94	7,60
	Stand	1,11	0,57	5,98	92,34

Table 26: Cumulated confusion matrix (in %) over 10-folds for CNN with all sensors combined, raw data and 0.05s window.

Avg Accuracy (%)	Avg Time (ms)	Best Accuracy (%)	Best Candidate	Worst Accuracy (%)	Worst Candidate	Best Candidate Time (ms)
95,22	0,075	97,91	6	92,07	4	0,077

Table 27: Average, best and worst performance over 10-folds for CNN with all sensors combined, raw data and 0.05s window.

When using features, the most accurate results were obtained using accelerometer data. The best accuracy, 83.339%, was obtained with the 0.1s window. The difference with the smallest window was only 1.465% less. Hence, the performance of these two window sizes is further compared below to find if the marginal increase in accuracy justifies the longer delay.

As shown by Tables 28, the larger window size significantly improved the detection of sitting and standing phases. However, it significantly underperformed in identifying the first part of the transition phase. Both window sizes misclassified the initial transition phase as sitting most of the time. Tables 29 and 30 indicate that the smaller window size offered performed prediction slightly faster, but still significantly slower than when utilizing raw data. There was a 20.17% difference between the best and worst accuracy for the 0.05s window, and a 14.21% difference for the 0.1s window. These figures are above average across all configurations, but below averages across the best set-up. Although the 0.1s window demonstrated better adaptability and slightly higher average accuracy, these were insufficient to balance its doubled delay compared to the 0.05s window, which demonstrated a superior potential for detecting transition phases.

CNN/Features/acc/0.05		True			
		Sit	Trans1	Trans2	Stand
Predicted	Sit	94,32	4,70	0,77	0,20
	Trans1	48,52	43,29	7,61	0,58
	Trans2	0,02	2,19	78,43	19,36
	Stand	1,02	0,13	17,94	80,91

CNN/Features/acc/0.1		True			
		Sit	Trans1	Trans2	Stand
Predicted	Sit	98,00	1,41	0,10	0,49
	Trans1	59,79	32,88	6,51	0,82
	Trans2	0,00	2,78	77,26	19,96
	Stand	1,03	0,22	12,03	86,72

Table 28: Cumulated confusion matrix (in %) over 10-folds for CNN with accelerometer sensors, extracted features and 0.05s window (left) and 0.075s window (right).

Avg Accuracy (%)	Avg Time (ms)	Best Accuracy (%)	Best Candidate	Worst Accuracy (%)	Worst Candidate	Best Candidate Time (ms)
82,42	0,203	88,40	7	68,23	6	0,192

Table 29: Average, best and worst performance over 10-folds for CNN with all sensors combined, features extracted and 0.05s window.

Avg Accuracy (%)	Avg Time (ms)	Best Accuracy (%)	Best Candidate	Worst Accuracy (%)	Worst Candidate	Best Candidate Time (ms)
83,89	0,238	90,79	6	76,58	8	0,248

Table 30: Average, best and worst performance over 10-folds for MLP with all sensors combined, features extracted and 0.1s window.

4.4 - KNN

KNN		Sensors									
		Features					Raw				
		Acc	Gyro	IMU	EMG	All	Acc	Gyro	IMU	EMG	All
Window	0.05	82,881	81,851	81,85	75,961	74,607	75,563	88,257	88,263	54,521	94,556
	0.075	84,639	81,518	81,518	74,614	73,869	75,692	88,056	88,067	50,845	94,546
	0.1	84,028	80,755	80,758	71,963	79,663	77,027	87,499	87,564	49,075	94,197
	0.15	84,507	80,732	80,732	73,442	77,875	77,974	86,767	86,798	48,034	93,659
	0.2	84,621	78,247	78,247	71,44	73,23	78,575	86,313	86,311	47,759	93,266
	0.3	84,586	77,277	77,277	70,756	79,297	80,653	85,112	84,955	47,682	93,011
	0.5	84,872	72,411	72,411	71,379	79,174	79,824	82,006	82,033	48,023	91,781

Table 31:Heat map of the average accuracy over 10-folds of KNN model for each combination of sensors, data format and window size. Green being the best accuracy and red the worst.

Table 31 shows that the highest accuracy for the KNN model was 94.556% and was achieved using raw data from all sensors combined and a 0.05s window. The detailed performance of this set-up is shown below.

Table 32 shows that the KNN model performed exceptionally well at detecting the sitting phase with not a single misclassification for the second part of transition or standing. However, performances were a bit low for other phases compared to the best set-ups with less than 90% for both transition phase. As indicated by Table 33, the average computational time per sample was 0.317ms which is slower than the other best set-ups. The difference between the highest and lowest accuracy over the 10-fold was only 8.05% which a demonstrate a good consistency and capacity to adapt to all individual.

KNN/raw/ all/0.05		True			
		Sit	Trans1	Trans2	Stand
Predicted	Sit	99,27	0,73	0,00	0,00
	Trans1	7,51	89,55	2,88	0,07
	Trans2	0,13	0,67	88,93	10,28
	Stand	1,67	0,28	6,11	91,94

Table 32: Cumulated confusion matrix (in %) over 10-folds for KNN with all sensors combined, raw data and 0.05s window.

Avg Accuracy (%)	Avg Time (ms)	Best Accuracy (%)	Best Candidate	Worst Accuracy (%)	Worst Candidate	Best Candidate Time (ms)
94,56	0,317	98,15	6	90,10	4	0,217

Table 33: Average, best and worst performance over 10-folds for KNN with all sensors combined, raw data and 0.05s window.

When using features, the most accurate results were obtained using accelerometer data. The best accuracy, 84.639%, was obtained with the 0.075s window. The difference with the smallest window was only 1.758% less. Hence, the performance of these two window sizes is further compared below to find if the marginal increase in accuracy justifies the longer delay.

As illustrated by Tables 34, both window sizes show similar results for each phase detection. Both windows performed poorly when classifying the first part of transition classified it most time as sitting. According to Tables 35 and 36, the difference between the best and worst outcomes was 17.76% for the 0.05s window and 10.6% for the 0.075s window, indicating the smaller window's inferior adaptability. The average computation time for the 0.05s window was more than three time superior to the 0.075s. This significant difference between two relatively close window sizes may be due to the hardware performance at the time of computation. Nevertheless, even if the computation time was disregarded, the increased accuracy of the 0.075s window and its better accuracy outweigh its small increased delay due to the window size. Therefore, the 0.075s window should be considered as the more optimal choice.

KNN/Features/acc/0.05		True			
		Sit	Trans1	Trans2	Stand
Predicted	Sit	94,18	4,94	0,10	0,78
	Trans1	57,58	34,27	7,32	0,84
	Trans2	0,30	2,48	80,99	16,24
	Stand	1,15	0,10	12,94	85,81

KNN/Features/acc/0.075		True			
		Sit	Trans1	Trans2	Stand
Predicted	Sit	96,75	3,15	0,10	0,00
	Trans1	58,44	34,44	6,59	0,53
	Trans2	0,36	2,43	80,90	16,30
	Stand	1,18	0,07	10,30	88,45

Table 34: Cumulated confusion matrix (in %) over 10-folds for KNN with accelerometer sensors, extracted features and 0.05s window (left) and 0.075s window (right).

Avg Accuracy (%)	Avg Time (ms)	Best Accuracy (%)	Best Candidate	Worst Accuracy (%)	Worst Candidate	Best Candidate Time (ms)
82,88	0,510	88,75	7	70,99	8	0,392

Table 35: Average, best and worst performance over 10-folds for KNN with all sensors combined, features extracted and 0.05s window.

Avg Accuracy (%)	Avg Time (ms)	Best Accuracy (%)	Best Candidate	Worst Accuracy (%)	Worst Candidate	Best Candidate Time (ms)
84,45	0,153	89,87	2	79,27	8	0,146

Table 36: Average, best and worst performance over 10-folds for KNN with all sensors combined, features extracted and 0.075s window.

4.5 – RF

RF		Sensors									
		Features					Raw				
		Acc	Gyro	IMU	EMG	All	Acc	Gyro	IMU	EMG	All
Window	0.05	69,695	66,946	66,942	75,722	87,306	56,235	88,892	88,919	75,728	95,599
	0.075	69,155	61,202	61,203	76,082	86,95	55,854	89,089	89,072	74,893	95,53
	0.1	71,716	82,099	82,059	75,796	90,999	53,611	89,278	89,312	74,447	95,226
	0.15	71,429	82,959	82,959	75,989	90,648	56,491	89,94	90,101	72,707	94,616
	0.2	73,554	79,655	79,665	75,839	89,171	63,275	90,292	90,305	71,698	95,086
	0.3	73,663	78,724	78,724	75,922	88,098	72,081	90,916	90,807	70,127	95,919
	0.5	73,547	79,22	79,195	76,065	86,131	82,07	91,44	91,339	67,906	95,679

Table 37: Heat map of the average accuracy over 10-folds of RF model for each combination of sensors, data format and window size. Green being the best accuracy and red the worst.

Table 37 shows that the highest accuracy for the RF model was 95.919% and was achieved using raw data from all sensors combined and a 0.3s window. Interestingly, the smaller window of 0.05s had an accuracy that was only 0.32% lower. Hence, the performance of these two window sizes is further compared below to find if the marginal increase in accuracy justifies the longer delay.

Tables 38 demonstrate marginal differences in the performance of the 0.3s and 0.05s windows when it comes to detecting each phase. The second part of the transition phase was the most challenging for this model, often misclassifying it as the standing phase. Both windows exhibit excellent computational time per sample, as indicated in Tables 39 and 40. Furthermore, both windows maintain a similar level of consistency, with less than a 5.5% difference between their best and worst performance. This consistency places the RF model among the most reliable models. The only discernible difference between the two windows is the slightly faster computation time of the 0.05s window, although the difference is relatively minor in comparison to the window size. Thus, the minor improvement in performance achieved with the 0.3s window doesn't outweigh the significant increase in delay due to the larger window size.

RF/raw/ all/0.05		True			
		Sit	Trans1	Trans2	Stand
Predicted	Sit	98,80	1,08	0,04	0,07
	Trans1	4,84	93,29	1,41	0,46
	Trans2	0,00	1,19	91,21	7,60
	Stand	1,15	0,38	4,59	93,88

RF/raw/ all/0.3		True			
		Sit	Trans1	Trans2	Stand
Predicted	Sit	98,76	1,16	0,08	0,00
	Trans1	3,52	93,29	2,73	0,46
	Trans2	0,00	1,27	92,47	6,26
	Stand	1,02	0,51	4,52	93,95

Table 38: Cumulated confusion matrix (in %) over 10-folds for RF with all sensors combined, raw data and 0.05s window (left) and 0.3s window (right).

Avg Accuracy (%)	Avg Time (ms)	Best Accuracy (%)	Best Candidate	Worst Accuracy (%)	Worst Candidate	Best Candidate Time (ms)
95,60	0,029	98,05	6	92,90	4	0,024

Table 39: Average, best and worst performance over 10-folds for RF with all sensors combined, raw data and 0.05s window.

Avg Accuracy (%)	Avg Time (ms)	Best Accuracy (%)	Best Candidate	Worst Accuracy (%)	Worst Candidate	Best Candidate Time (ms)
95,92	0,046	98,21	6	92,87	8	0,048

Table 40: Average, best and worst performance over 10-folds for RF with all sensors combined, raw data and 0.3s window.

When using features, the most accurate results were obtained using all sensors data combined. The best accuracy, 90.999%, was obtained with the 0.1s window. The difference with the smallest window was only 3.693% less. Hence, the performance of these two window sizes is further compared below to find if the marginal increase in accuracy justifies the longer delay.

As demonstrated in Tables 41, the 0.1s window shows comparable accuracy to the 0.05s window for the sitting phase, but perform significantly better for all other phases, particularly for the first part of the transition phase. Both windows have excellent computational times per sample, with the 0.05s window having a slight advantage, as illustrated in Tables 42 and 43. However, this difference is negligible in comparison to the window sizes. The 0.05s window had poor consistency, with a variation of 22.5% between its best and worst accuracies, compared to 7.98% for the 0.1s window. This suggests that the 0.1s window offers better adaptability and the potential for more consistent and safe performance across different individuals. Therefore, it's reasonable to conclude that the superior performance of the 0.1s window justifies the increased delay due to the larger window size.

RF/Features/ all/0.05		True			
		Sit	Trans1	Trans2	Stand
Predicted	Sit	98,71	1,16	0,10	0,03
	Trans1	15,71	62,68	18,02	3,59
	Trans2	3,15	3,04	79,70	14,12
	Stand	3,10	3,09	11,75	82,05

RF/Features/ all/0.1		True			
		Sit	Trans1	Trans2	Stand
Predicted	Sit	98,00	1,96	0,03	0,02
	Trans1	6,40	82,16	8,68	2,76
	Trans2	0,00	2,69	83,33	13,98
	Stand	0,98	4,27	7,07	87,68

Table 41: Cumulated confusion matrix (in %) over 10-folds for RF with accelerometer sensors, extracted features and 0.05s window (left) and 0.1s (right).

Avg Accuracy (%)	Avg Time (ms)	Best Accuracy (%)	Best Candidate	Worst Accuracy (%)	Worst Candidate	Best Candidate Time (ms)
87,31	0,029	92,96	5	70,46	1	0,028

Table 42: Average, best and worst performance over 10-folds for RF with all sensors combined, features extracted and 0.05s window.

Avg Accuracy (%)	Avg Time (ms)	Best Accuracy (%)	Best Candidate	Worst Accuracy (%)	Worst Candidate	Best Candidate Time (ms)
91,00	0,040	93,94	6	85,96	2	0,039

Table 43: Average, best and worst performance over 10-folds for RF with all sensors combined, features extracted and 0.1s window.

5 - Discussion

5.1 – Format & sensors

The different models were trained and tested using two formats for the data, features or raw.
As mentioned previously, the selection of extracted features was fixed and was based on literature rather than experimental evaluation. This means that the accuracy results for feature extraction, might not fully reflect the model's potential performance.

Extracting features resulted in poor accuracy when classifying the first part of transition for all Machine Learning method except for RF which performance remained above average but still below its performance for other phases. 60.05% for SVM, 43.55% for MLP, 43.29% for CNN, 34.44% for KNN and 82.16% for RF. The phase that it was the most misidentified as for all methods was the sitting phase. This may be due to the short duration of the first part of transition and the small amplitude of it characteristic peak (see figure 12) that happens right after the sitting phase. The first part of transition could be considered as a noise peak in the sitting phase. As one of the advantages of extracting feature is to reduce the effect of noise on the data, the extracted features of the first part of transition may be similar to the one of the sitting phases. Raw data on the other hand preserve more insights and therefore may perform better with short phase.

As we can see in Table A, B, C, D, and E, for every model the raw data had higher accuracy when used in combination of the gyroscopes, IMUs and all sensors, while the extracted feature performed better with accelerometers and EMG. This is possibly because gyroscope data, which measures rotational motion, tends to be more complex and potentially holds a higher degree of intricate, time-dependent information that may get lost or oversimplified during the feature extraction process. Utilizing raw data allows the machine learning model to process the full complexity of the gyroscope data, and when combined with accelerometer and EMG data, it can capitalize on the complementary nature of these sensors to capture even more information. The accelerometer and EMG signals are by nature noisier than the gyroscope signal. Thus, these sensors benefit from the filtering that result from feature extraction.

5.2 - Window

The initial assumption about the window size would have been that smaller window would result in lower accuracy as they would fail to gather enough information to differentiate phases. Surprisingly, for the top performing set-up the window size had very little effect on the performance. More surprisingly the best accuracy when using raw data was obtained with one of the two smallest windows for all methods except SVM. This may be due to the fact that small windows resulted in more training data, and so may explain why SVM performance was down with small window as they are known to perform better with smaller dataset. For further testing it would be interesting to test with even smaller windows as it may be that the smallest window use was not small enough to decrease the performance of the data. Thus, it should not be expected that even smaller window would increase much further accuracy.
These observations did not translate to the best model with feature extracted. This confirms the observation made in Section 5.1 that feature extraction may not be adapted to small window and small phases. Small windows have a smaller spectrum of values and therefore the few valuable information may get lost by oversimplifying it.

5.3 – Models

The top performance across all models was achieved by the RF model. With raw data and features extracted when combined with all sensors. Consistency for both set ups was the highest across all top performing set-up with the same data format, suggesting a strong potential for consistent high performance across all kind of individuals. This excellent adaptability is due to the fact that RF aggregates predictions from multiple decision trees to make the final decision. Each tree being trained with different parts of the training data. This approach minimizes the risk of overfitting, a common problem in machine learning models, and leads to better generalization on unseen data.

SVM were found to be the model that took the longer to train. The training time was inversely proportional to the window size to the point that it was not possible to experiment with 0.05s and 0.075s window with the hardware available. Smaller window sizes did not seem to increase SVM performance, thus we can conclude that it is not worth continuing this work for experimenting with smaller window. It would be highly recommended to use better performing device than the one used in this report if SVM were to be selected for further optimisation.

MLP and CNN exhibited very good accuracy results when using raw data in combination with all sensors. Both methods have a very strong potential for hyperparameter optimisation. Given the small range of hyperparameters values that were tested and the already good performances of the models, we can assume that MLP and CNN have the most potentials to be the top-performing model.

5.4 - Limitations

When reviewing the performance of the different models it is important to mention some concerns about the quality of the database.

- The specifics of the support on which the individuals sit are not provided, the cushioning of the support could impact the transition phase duration. Therefore, the train model may not be adaptable to new unseen data using a different support. And the pattern detected in Section 3.2 may not be efficient for classifying data using or not using arms help.

- It is unclear whether the participants were allowed to use their arms for assistance when standing up or sitting down. Using arms affect grandly the sequence of the movement and the activation of the muscles. Similarly, to the type of seat problem, this may affect the model adaptability and the efficacity of the relabelling method presented.

- The instructions for waiting after standing before starting to walk are not well-defined. Some candidate started walking right after standing and some don’t. Despite, the relabelling, data labelled as standing probably contains small part of walking phases. While this variability could be advantageous for diversifying training data and preparing for different scenarios, provided the data is labelled correctly. It may instead confuse the model if labelling is too inconsistent.

- The unclear procedure for sitting. Since participants must walk between standing and sitting, they will face the chair when returning to it, necessitating a half or quarter turn to sit. It is ambiguous whether the small steps needed for turning are considered part of standing or walking. Combined with the absence of a mandatory pause between the end of walking and sitting, it becomes challenging to determine the beginning of the standing phase before sitting with certainty.

5.6 – Future works

Future work would consist in three axes:

- All combination should be tested on Stand-to-Sit data separately as well as combined with Sit-to-Stand data. Work has been done to relabel and extract Stand-to-Sit data as explained in this report and it was ready to be tested. However, testing through all combination is a very long process and it was not possible to complete it in the limit of this project.
  Once trained, it would be interesting to compare the performance of a software using a model trained with Sit-to-Stand and Stand-to-Sit compared to a software switching between a Sit-to-Stand model and Stand-to-Sit model depending on if the current state is sitting or standing.

- Now that we know the optimal window size, sensors and format combination for each method, these parameters should be kept fixed and used to experiment more rigorously with each method hyperparameters. Especially with CNN and MLP.

- This work should be reproduced using a different feature selection for each sensor. The feature selection method should be experimental rather than based on literature.

6 - Conclusions

In conclusion, this study has effectively demonstrated the potential of employing machine learning algorithms to detect Sit-To-Stand transitions using data collected from wearable sensors. Our analysis suggests that the Random Forest model outperformed the other evaluated algorithms, SVM, MLP, CNN, and KNN, in terms of accuracy, consistency, and computational speed, making it the most suitable for this application. Moreover, we discovered that the type of sensor data used and whether it was raw or feature-extracted, significantly impacted the model's performance. The top performance for all methods was achieved using raw data with all sensors combined.
Machine learning models using extracted features performed better with data from EMG and accelerometers, while those utilizing raw data had superior results with gyroscope, IMU, and, IMU and EMG combined data. Future work should involve a more rigorous and model-specific feature selection approach, as this could significantly improve the performance of models utilizing feature-extracted data.

7 - References

[1] - Nishimura, T., Imai, A., Fujimoto, M., Kurihara, T., Kagawa, K., Nagata, T. and Sanada, K., 2020. Adverse effects of the coexistence of locomotive syndrome and sarcopenia on the walking ability and performance of activities of daily living in Japanese elderly females: a cross-sectional study [Online]. *Journal of Physical Therapy Science* [Online], 32(3), pp.227–232. Available from: https://doi.org/10.1589/jpts.32.227 [Accessed 14 December 2021].

[2] - CDC, 2019. Arthritis-Related Statistics [Online]. Centers for Disease Control and Prevention. Available from: https://www.cdc.gov/arthritis/data_statistics/arthritis-related-stats.htm.

[3] - Parkinson's Foundation, 2019. Statistics [Online]. Parkinson's Foundation. Available from: https://www.parkinson.org/Understanding-Parkinsons/Statistics.

[4] - Takai, Y., Ohta, M., Akagi, R., Kanehisa, H., Kawakami, Y., Fukunaga, T., 2009. Sit-to-stand test to evaluate knee extensor muscle size and strength in the elderly: a novel approach.

[5] - Kim, J. (2018). Longitudinal association between physical function changes and depressive symptoms in older adults. Journal of Aging and Health, 31(6), 1027–1045.

[6] - WGM, J.W., HBJ, B.H. and HJ, S., 2002. Determinants of the Sit-to-Stand Movement: A Review [Online]. Physical Therapy [Online]. Available from: https://doi.org/10.1093/ptj/82.9.866.

[7] - Rapp, K., Becker, C., Cameron, I.D., König, H.-H. and Büchele, G., 2012. Epidemiology of Falls in Residential Aged Care: Analysis of More Than 70,000 Falls From Residents of Bavarian Nursing Homes [Online]. Journal of the American Medical Directors Association [Online], 13(2), pp.187.e1–187.e6. Available from: https://doi.org/10.1016/j.jamda.2011.06.011.

[8] - Nyberg, L. and Gustafson, Y., 1995. Patient Falls in Stroke Rehabilitation [Online]. Stroke [Online], 26(5), pp.838–842. Available from: https://doi.org/10.1161/01.str.26.5.838.

[9] - Lord, S.R., Murray, S.M., Chapman, K., Munro, B. and Tiedemann, A., 2002. Sit-to-stand performance depends on sensation, speed, balance, and psychological status in addition to strength in older people [Online]. The Journals of Gerontology. Series A, Biological Sciences and Medical Sciences [Online], 57(8), pp.M539-543. Available from: https://doi.org/10.1093/gerona/57.8.m539.

[10] - World Health Organization, 2021. Falls [Online]. World Health Organization. Available from: https://www.who.int/news-room/fact-sheets/detail/falls.

[11] - National Osteoporosis Foundation, n.d. Osteoporosis Fast Facts [Online]. Available from: https://www.bonehealthandosteoporosis.org/wp-content/uploads/2015/12/Osteoporosis-Fast-Facts.pdf.

[12] - Inability to get up after falling, subsequent time on floor, and summoning help: prospective cohort study in people over 90

[13] - Mancini, M., Horak, F.B., Zampieri, C., Carlson-Kuhta, P., Nutt, J.G. and Chiari, L., 2011. Trunk accelerometry reveals postural instability in untreated Parkinson's disease [Online]. Parkinsonism & Related Disorders [Online], 17(7), pp.557–562. Available from: https://doi.org/10.1016/j.parkreldis.2011.05.010 [Accessed 21 March 2020].

[14] - Wu, W., Au, L., Jordan, B., Stathopoulos, T., Batalin, M., Kaiser, W., Vahdatpour, A., Sarrafzadeh, M., Fang, M. and Chodosh, J., 2008. The SmartCane System: An Assistive Device for Geriatrics [Online]. Proceedings of the 3rd International ICST Conference on Body Area Networks [Online]. Available from: https://doi.org/10.4108/icst.bodynets2008.2944 [Accessed 18 March 2020].

[15] - Ante Šantić, Bilas, V. and Igor Lacković, 1997. A system for measuring forces in the legs and crutches from ambulatory patients [Online]. Available from: https://doi.org/10.1109/iembs.1997.758704 [Accessed 17 May 2023].

[16] - Arcelus, A., Veledar, I., Goubran, R., Knoefel, F., Sveistrup, H. and Bilodeau, M., 2011. Measurements of Sit-to-Stand Timing and Symmetry From Bed Pressure Sensors [Online]. IEEE Transactions on Instrumentation and Measurement [Online], 60(5), pp.1732–1740. Available from: https://doi.org/10.1109/tim.2010.2089171 [Accessed 26 January 2022].

[17] - Kawamoto, H., Lee, S., Sadao Kanbe and Yoshiyuki Sankai, 2003. Power assist method for HAL-3 using EMG-based feedback controller [Online]. Systems, Man and Cybernetics [Online]. Available from: https://doi.org/10.1109/icsmc.2003.1244649 [Accessed 22 April 2023].

[18] - Di Natali, C., Poliero, T., Sposito, M., Graf, E., Bauer, C., Pauli, C., Bottenberg, E., De Eyto, A., O'Sullivan, L., Hidalgo, A.F., Scherly, D., Stadler, K.S., Caldwell, D.G. and Ortiz, J., 2019. Design and Evaluation of a Soft Assistive Lower Limb Exoskeleton [Online]. Robotica [Online], 37(12), pp.2014–2034. Available from: https://doi.org/10.1017/s0263574719000067.

[19] - Hu, B., Rouse, E. and Hargrove, L., 2018. Benchmark Datasets for Bilateral Lower-Limb Neuromechanical Signals from Wearable Sensors during Unassisted Locomotion in Able-Bodied Individuals [Online]. Frontiers in Robotics and AI [Online], 5. Available from: https://doi.org/10.3389/frobt.2018.00014 [Accessed 14 January 2021].

[20]- Early Detection of the Initiation of Sit-to-Stand Posture Transitions Using Orthosis-Mounted Sensors
Abul Doulah 1 ID , Xiangrong Shen 2 and Edward Sazonov 1,* ID

[21] - Detecting absolute human knee angle and angular velocity using accelerometers and rate gyroscopes R. Williamson I B.J. Andrews 2

8 – Appendices

	A	B	C	D
1	Subject	Age (years)	Height (cm)	Weight (kg)
2	AB156	26	193	77
3	AB185	28	181	75
4	AB186	24	163	54
5	AB188	25	185	87
6	AB189	24	178	66
7	AB190	23	160	54
8	AB191	26	163	54
9	AB192	23	170	75
10	AB193	27	185	95
11	AB194	29	160	61

Table 44: Candidates' specifications

9.1 – Visualisation.py

Visualistion.py

```
import csv
import os
import numpy as np
import math
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import MinMaxScaler

from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn import metrics

import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense, Input
from keras.optimizers import Adam

from scipy.signal import butter, lfilter, find_peaks

import pywt




def Main(filename):

    data = []

    #Open .csv file for read
    with open(filename +'.csv', 'r') as csvfile:
        reader = csv.reader(csvfile)

        # Create objects for each header
        # Assign the name of each header to the objects
        for row in reader:
            for i in range(len(row)): data.append([row[i]])
            break

        # Assign the value of each column to a different object
        for row in reader:
            for i in range(len(data)):
                if row[i] != "":
                    data[i].append(float(row[i]))

    return data
```

```
def processed(data):

    #Find number of columns
    i = 0
    for col in data:
        if col[0] == "Mode":
            break
        i = i + 1

    #Add headers
    dataPost = []
    for col in range(i+1):
        dataPost.append([data[col][0]])
    dataPost.append(['Step'])
    dataPost.append(['Changes'])

    #Add data
    n=0
    for row in range(1,len(data[i])):

        #Add data corresponding to sit or stand
        if int(data[i][row]) == 0 or int(data[i][row]) == 6:
            for col in range(i+1):
                if col == i:
                    dataPost[col].append(int(data[col][row]))
                else:
                    dataPost[col].append(data[col][row])
            dataPost[i+1].append(row)  #Add step number

            if (row > 1) and (data[i][row] != data[i][row-1]):  #If there
is a change in position
                dataPost[i+2].append(1)                         #Mark
change as 1

                if row - dataPost[i+1][n] >100:                 #If the
last standing position was more than 100 steps ago,
                    dataPost[i+2][n] = 1                        #It means
it that it the start of standing to sitting
            else:
                dataPost[i+2].append(0)                         #Add 0 if
there is no change
            n = n+1

    return dataPost


def MaxPeakToPeak(signal):
    """
    Finds when the magnitude between the positive and negative peaks in a
noisy oscillating signal
    goes below a certain threshold. Returns the index of the data point
where the threshold is crossed.
    """
    # Find the indices of the positive and negative peaks in the signal
    pos_peak_indices, _ = find_peaks(np.array(signal))
```

```
    neg_peak_indices, _ = find_peaks(-np.array(signal))

    # Calculate the magnitudes between each positive and negative peak
    magnitudes = []
    for pos_index in pos_peak_indices:
        for neg_index in neg_peak_indices:
            if neg_index > pos_index:
                magnitudes.append(signal[pos_index] - signal[neg_index])
                break

    max_mag = 0
    # Iterate through the magnitudes and find the index where the
threshold is crossed
    for i, magnitude in enumerate(magnitudes):
        if magnitude > max_mag:
            max_mag = magnitude
            max_mag_index = i

    #return index of max peak to peak
    return pos_peak_indices[max_mag_index]


def MedianPeakToPeak(signal):
    """
    Finds when the magnitude between the positive and negative peaks in a
noisy oscillating signal
    goes below a certain threshold. Returns the index of the data point
where the threshold is crossed.
    """
    # Find the indices of the positive and negative peaks in the signal
    pos_peak_indices, _ = find_peaks(np.array(signal))
    neg_peak_indices, _ = find_peaks(-np.array(signal))

    # Calculate the magnitudes between each positive and negative peak
    magnitudes = []
    for pos_index in pos_peak_indices:
        for neg_index in neg_peak_indices:
            if neg_index > pos_index:
                magnitudes.append(signal[pos_index] - signal[neg_index])
                break

    """
    sum = 0
    # Sum of the magnitude
    for i, magnitude in enumerate(magnitudes):
        sum = sum + magnitude

    avg = sum / i
    """
    #find median of peak to peak magnitude
    median = np.median(magnitudes)

    #return average peak to peak magnitude
    return median
```

```
def PeakToPeakTreshold(signal, threshold):
    """
    Finds when the magnitude between the positive and negative peaks in a
noisy oscillating signal
    goes below a certain threshold. Returns the index of the data point
where the threshold is crossed.
    """
    # Find the indices of the positive and negative peaks in the signal
    pos_peak_indices, _ = find_peaks(np.array(signal))
    neg_peak_indices, _ = find_peaks(-np.array(signal))

    # Calculate the magnitudes between each positive and negative peak
    magnitudes = []
    for pos_index in pos_peak_indices:
        for neg_index in neg_peak_indices:
            if neg_index > pos_index:
                magnitudes.append(signal[pos_index] - signal[neg_index])
                break

    # Iterate through the magnitudes and find the index where the
threshold is crossed
    for i, magnitude in enumerate(magnitudes):
        """
        sum = 0
        for n in range(1,5):
            if i+ n > len(magnitudes)-1:
                break
            sum = sum + magnitudes[i+n]
        """
        median = np.median(magnitudes[i:i+10])
        if median < threshold*20:
            return pos_peak_indices[i]

    # If the threshold is never crossed, return None
    return None

def EndMaxPeak(signal, threshold):
    """
    Finds where the signal crosses a certain threshold that is the median
valuenof the sitting phase.
    Returns the index of the data point where the threshold is crossed.
    """
    # Find the indices of the positive and negative peaks in the signal
    pos_peak_indices, _ = find_peaks(np.array(signal))

    peak = 0
    # Iterate through the peaks and find the maximum peak
    for pos_index in pos_peak_indices:
        if signal[pos_index] > peak:
            peak = signal[pos_index]
            peak_index = pos_index
```

```
    # Iterate through the signal after the peak and find the index where
the threshold is crossed
    for i, value in enumerate(signal[peak_index:]):
        if value < threshold:
            return i + peak_index

    # If the threshold is never crossed, return None
    return None


def StartTransition(signal_R, signal_L, init_start, init_end):

    median_R = np.median(signal_R[:init_start])
    median_L = np.median(signal_L[:init_start])

    max_R_idx = np.argmax(signal_R[:init_end])
    max_L_idx = np.argmax(signal_L[:init_end])

    range_R = abs(signal_R[max_R_idx] - median_R)
    range_L = abs(signal_L[max_L_idx] - median_L)

    for i in range(len(signal_R[:init_end])-1):
        if signal_R[i] > (median_R + range_R*0.02) and signal_L[i] >
(median_L + range_L*0.02):
            return i



def EndTransition(signal):

    peak_index = np.argmax(signal)
    range = abs(signal[0] - signal[peak_index])

    for i, value in enumerate(signal):
        if value > (signal[0] + range*0.90):
            return i

    return 0


def EndTransition2(signal_R, signal_L):

    peak_indices_R, _ = find_peaks(np.array(signal_R))
    peak_indices_L, _ = find_peaks(np.array(signal_L))

    check_R =0
    for _,idx in enumerate(peak_indices_R):
        if signal_R[idx] > -50:
            peak_R = idx
            check_R = 1
            break

    if check_R ==0:
        peak_R = np.argmax(signal_R)
```

```
    check_L =0
    for _,idx in enumerate(peak_indices_L):
        if signal_L[idx] > -50:
            peak_L = idx
            check_L = 1
            break
    if check_L ==0:
        peak_L = np.argmax(signal_L)


    idx_end_trans = min(peak_R, peak_L)

    if idx_end_trans == peak_R:
        test = np.where(peak_indices_R ==peak_R)[0][0]
        idx_end_standing = peak_indices_R[test+1]
    else:
        test = np.where(peak_indices_L ==peak_L)[0][0]
        idx_end_standing = peak_indices_L[test+1]


    return idx_end_trans, idx_end_standing




def StartSitting(signal):

    min_index = np.argmin(signal)
    median = np.median(signal[min_index:])

    max_index = np.argmax(signal)

    test = signal[max_index]
    test2 = signal[min_index]
    range = abs(signal[max_index] - median)

    for i, value in enumerate(signal[max_index:]):
        if value < (signal[max_index] - range*0.95):
            return i + max_index

    return None


def butter_lowpass(cutoff, fs, order=5):
    nyq = 0.5 * fs
    normal_cutoff = cutoff / nyq
    b, a = butter(order, normal_cutoff, btype='low', analog=False)
    return b, a

def butter_lowpass_filter(data, cutoff, fs, order=5):
    b, a = butter_lowpass(cutoff, fs, order=order)
    y = lfilter(b, a, data)
    return y
```

```
def HipAngle(acc_WZ, acc_WY, gyro_WX,acc_TZ,acc_TX, gyro_TY, init_end):

    dt = 1/500
    init_0 = 250
    new_0 = 245

    gyro_WX = np.array(gyro_WX) * 2 * np.pi / 360
    gyro_TY = np.array(gyro_TY) * 2 * np.pi / 360


    max_gyro_WX = np.argmax(gyro_WX[:init_end])

    """     max_TY = abs(np.max(gyro_TY[:init_end]- gyro_TY[0]))
    min_TY = abs(np.min(gyro_TY[:init_end]- gyro_TY[0]))
    if max_TY < min_TY:
        gyro_TY = -gyro_TY
        print("chocolat") """

    integral = np.sum(gyro_TY[:init_end]) - gyro_TY[0]*init_end
    if integral < 0:
        gyro_TY = -gyro_TY
        print("chocolat")


    static_W= []
    static_T = []

    gyro_W0 = 0
    gyro_T0 = 0
    tilt_W0 =0
    tilt_T0 = 0
    for i in range(50):
        gyro_W0 = gyro_W0 + gyro_WX[i]/init_0
        gyro_T0 = gyro_T0 + gyro_TY[i]/init_0
        tilt_W0 = tilt_W0 + np.arctan2(acc_WZ[i],acc_WY[i])/init_0
        tilt_T0 = tilt_T0 + np.arctan2(acc_TZ[i],acc_TX[i])/init_0

    tilt_W = np.zeros(len(gyro_WX)-1)
    tilt_T = np.zeros(len(gyro_WX)-1)
    angle_hip = np.zeros(len(gyro_WX)-1)
    for i in range(len(gyro_WX)-1):

        if i < init_0:
            tilt_W[i] = tilt_W0
            tilt_T[i] = tilt_T0
            angle_hip[i] = tilt_W0 - tilt_T0
        else:

            tiltW_deg = tilt_W[i-new_0:i]*180/np.pi
            tiltT_deg = tilt_T[i-new_0:i]*180/np.pi

            #variance_W = math.sqrt((np.sum((tiltW_deg**2)*dt) -
np.sum(tiltW_deg*dt)**2)/new_0)
```

```
            #variance_T = math.sqrt((np.sum((tiltT_deg**2)*dt) -
np.sum(tiltT_deg*dt)**2)/new_0)

            variance_W = np.sum((tiltW_deg -
np.sum(tiltW_deg)/new_0)**2)/new_0
            variance_T = np.sum((tiltT_deg -
np.sum(tiltT_deg)/new_0)**2)/new_0

            #variance_W = math.sqrt((np.sum((tilt_W[i-new_0:i]**2)*dt) -
np.sum(tilt_W[i-new_0:i]*dt)**2)/new_0)
            #variance_T = math.sqrt((np.sum((tilt_T[i-new_0:i]**2)*dt) -
np.sum(tilt_T[i-new_0:i]*dt)**2)/new_0)

            #variance_W = np.sum((tilt_W[i-new_0:i] - np.sum(tilt_W[i-
new_0:i])/new_0)**2)/new_0
            #variance_T = np.sum((tilt_T[i-new_0:i] - np.sum(tilt_T[i-
new_0:i])/new_0)**2)/new_0

            """
            if variance_W < 0.1:
                gyro_W0 = np.sum(gyro_WX[i-new_0:i])/new_0
                #if i> 4000:
                    #print("W ", i)
                    #print("gyro_W0 ", gyro_W0)
                static_W.append(i)
            if variance_T < 0.1:
                gyro_T0 = np.sum(gyro_TY[i-new_0:i])/new_0
                #if i> 4000:
                    #print("T ", i)
                    #print("gyro_T0 ", gyro_T0)
                static_T.append(i)
            """

            tilt_W[i] = tilt_W[i-1] + (gyro_WX[i] - gyro_W0 ) * dt
            tilt_T[i] = tilt_T[i-1] + (gyro_TY[i] - gyro_T0) * dt
            angle_hip[i] = tilt_W[i] - tilt_T[i]


        angle_hip[i] = angle_hip[i] * 360 / (2 * np.pi)

    tilt_W = tilt_W * 360 / (2 * np.pi)

    sin_angle_hip = np.sin(angle_hip*2*math.pi/360)


    """
    # Define the wavelet type and level of decomposition
    wavelet = 'db1'
    level = 2

    # Perform the DWT
    coeffs = pywt.wavedec(sin_angle_hip, wavelet, level=level)

    # Filter the signal by thresholding the detail coefficients
    threshold = 2  # Set the threshold value
```

```
    new_coeffs = [coeffs[0]] + [pywt.threshold(detail_coeff, threshold)
for detail_coeff in coeffs[1:]]

    # Reconstruct the signal
    filtered_signal = pywt.waverec(new_coeffs, wavelet)
    """

    return angle_hip, tilt_W  #filtered_signal[1:]    #sin_angle_hip
# filtered_signal[1:]


def SiStTransHipAngle(signal_R, signal_L, init_start, init_end,
knee_start, hip_tilt_signal):


    max_R_idx = np.argmax(signal_R[:init_end])
    max_L_idx = np.argmax(signal_L[:init_end])

    # Index where the trunk end leaning forward
    # Do it by finding the max angle forward
    max_idx = max(max_R_idx, max_L_idx)

    #Find the index where the trunk start leaning forward
    # Do it by finding when the hip angle is 5% of the range between the
static angle and the max angle forward

    delta = 100
    treshold = 0.05

    check_R = 0
    check_L = 0

    for i in range(max_idx):

        gradient_R = abs(signal_R[i+delta] - signal_R[i])/(delta)
        gradient_L = abs(signal_L[i+delta] - signal_L[i])/(delta)

        if gradient_R > treshold:
            check_R = 1
        if gradient_L > treshold:
            check_L = 1

        if check_R == 1 and check_L == 1:
            break

    start_trans = i


    # Find the index where the trunk stops leaning backward
    # Do it by finding when the slop decrease

    delta = 50
    treshold = 0.01

    check_tilt = 0
```

```
    check_R = 0
    check_L = 0

    max_hip_tilt_idx= np.argmax(hip_tilt_signal[:init_end])

    for i in range(max_hip_tilt_idx,init_end*2):

        gradient_tilt = abs(hip_tilt_signal[i+delta] -
hip_tilt_signal[i])/(delta)
        gradient_R = abs(signal_R[i+delta] - signal_R[i])/(delta)
        gradient_L = abs(signal_L[i+delta] - signal_L[i])/(delta)

        if gradient_tilt < treshold:
            check_tilt = 1
        if gradient_R < treshold:
            check_R = 1
        if gradient_L < treshold:
            check_L = 1

        if (check_R == 1 and check_tilt == 1) or (check_L == 1 and
check_tilt == 1):
            break

    end_trans = i


    diff_start = abs(signal_R[end_trans] - signal_L[end_trans])

    for i in range(end_trans, end_trans+ 5000):
        diff_end = abs(signal_R[i] - signal_L[i])
        if diff_end > diff_start*2:
            break

    end_stand = i


    return start_trans, max_idx, end_trans, end_stand



def StSiTransHipAngle(signal_R, signal_L, init_start, init_end, knee_end,
hip_tilt_signal):

    peak_indices_R, _ = find_peaks(np.array(signal_R[init_start:]),
prominence=5)
    peak_indices_L, _ = find_peaks(np.array(signal_L[init_start:]),
prominence=5)
    peak_indices_tilt, _ =
find_peaks(np.array(hip_tilt_signal[init_start:]))

    diff = [abs(init_start + value - knee_end) for _, value in
enumerate(peak_indices_R)]
    max_R_idx = peak_indices_R[diff.index(min(diff))]
    diff = [abs(init_start + value - knee_end) for _, value in
enumerate(peak_indices_L)]
```

```
    max_L_idx = peak_indices_L[diff.index(min(diff))]
    diff = [abs(init_start + value - knee_end) for _, value in
enumerate(peak_indices_tilt)]
    max_tilt_idx = peak_indices_tilt[diff.index(min(diff))]

    max_idx = max(max_R_idx, max_L_idx) + init_start

    """
    for i in range(len(signal_R[max_idx:])-1):
        if signal_R[i] < (median_R + range_R*0.02) and signal_L[i] <
(median_L + range_L*0.02):
            break

    end_trans = i + max_idx
    """

    # Find the index where the trunk stops leaning backward
    # Do it by finding when the slop decrease

    min_R_end = np.argmin(signal_R[max_idx:])
    min_L_end = np.argmin(signal_L[max_idx:])
    min_tilt_end = np.argmin(hip_tilt_signal[max_idx:])

    check = 0
    if min_R_end > len(signal_R[max_idx:]) - 100:
        check = check +1
    if min_L_end > len(signal_R[max_idx:])- 100:
        check = check +1
    if min_tilt_end > len(signal_R[max_idx:])- 100:
        check = check +1

    delta = 150

    if check >= 2:
        treshold = 0.05
    else:
        treshold = 0.01

    check_tilt = 0
    check_R = 0
    check_L = 0

    for i in range(max_idx + 100, len(signal_R)-1):

        gradient_tilt = abs(hip_tilt_signal[i+delta] -
hip_tilt_signal[i])/(delta)
        gradient_hip_R = abs(signal_R[i+delta] - signal_R[i])/(delta)
        gradient_hip_L = abs(signal_L[i+delta] - signal_L[i])/(delta)

        if gradient_tilt < treshold:
            check_tilt = 1
        if gradient_hip_R < treshold:
            check_R = 1
        if gradient_hip_L < treshold:
            check_L = 1
```

```
        if check_tilt == 1 and check_R == 1 and check_L == 1:
            break

    end_trans = i


    # Find the index where the trunk start leaning forward
    # Do it by finding when the hip tilt slop decrease.
    # Starts from the end of the trunk leaning forward and goes backward

    max_hip_tilt = np.argmax(hip_tilt_signal[init_start:]) +init_start -
1
    max_hip_tilt = max_tilt_idx + init_start

    delta = 50
    treshold_grad = 0.05
    treshold_range = 0.3

    check_tilt = 0
    check_R = 0
    check_L = 0

    min_tilt = np.min(hip_tilt_signal[init_start:max_hip_tilt])
    min_R = np.min(signal_R[init_start:max_R_idx+ init_start])
    min_L = np.min(signal_L[init_start:max_L_idx+ init_start])

    range_tilt = abs(tilt_hip[max_hip_tilt] - min_tilt)
    range_R = abs(signal_R[max_R_idx+ init_start] - min_R)
    range_L = abs(signal_L[max_L_idx+ init_start] - min_L)

    for i in range(max_hip_tilt - init_start):

        n = max_hip_tilt - i
        gradient_tilt = abs(tilt_hip[n-delta] - tilt_hip[n])/(delta)
        gradient_hip_R = abs(signal_R[n-delta] - signal_R[n])/(delta)
        gradient_hip_L = abs(signal_L[n-delta] - signal_L[n])/(delta)

        if gradient_tilt < treshold_grad and tilt_hip[n] < (min_tilt +
range_tilt*treshold_range):
            check_tilt = 1
        if gradient_hip_R < treshold_grad and signal_R[n] < (min_R +
range_R*treshold_range):
            check_R = 1
        if gradient_hip_L < treshold_grad and signal_L[n] < (min_L +
range_L*treshold_range):
            check_L = 1

        if check_R +check_L + check_tilt >= 2:
            break

    start_trans = n

    return start_trans, max_idx, end_trans
```

```
#-------------------Main-------------------#


col_end_trans = 35
col_end_trans_gyro = 29
col_plot_data = 45
sensor = "accelerometer"  #accelerometer, gyroscope, IMU, EMG, all
SiSt = 1
StSi = 0
multi = 0


thisFolderParent = os.getcwd()

candidate = ["156","185", "186", "188", "189", "190", "191", "192",
"193", "194"]
c = 5

# Initialise the lists
Sit = []
Stand = []
all_windows = []

# Loop through all the circuits of each candidate
for i in range(35,36):
    print("Circuit " + str(i))

    hardrive_path = "/Volumes/UnionSine/IMEE 2022/Semester2/FYP/FYPcode"

    if i<10:
        filename  = hardrive_path + "/Source/AB" + candidate[c] +
"/Processed/AB" + candidate[c] + "_Circuit_00" + str(i) + "_post"
    else:
        filename  =  hardrive_path + "/Source/AB" + candidate[c] +
"/Processed/AB" + candidate[c] + "_Circuit_0" + str(i) + "_post"

    #filename  = thisFolderParent +
"/Data/AB156/Processed/AB156_Circuit_002_post"

    allData = Main(filename)

    STprocessed = processed(allData)

    nbCol = len(STprocessed)
    nbRow = len(STprocessed[nbCol-3])

    indexChange = [STprocessed[nbCol-2][idx] for idx, value in
enumerate(STprocessed[nbCol-1]) if value == 1]
    #for i in range(len(indexChange)): print(STprocessed[nbCol -
2][indexChange[i]])
```

```
    # separate the data into Sit to Stand and Stand to Sit
    SiSt_data = []
    StSi_data = []
    for col in range(nbCol-2):
        #Add header
        SiSt_data.append([STprocessed[col][0]])
        StSi_data.append([STprocessed[col][0]])

        #Add data

        # Normalise every measurement to start at 0 (Not good results)
        # if STprocessed[col][1] > 0:
        #     for row in range(1,nbRow):
        #         if row < indexChange[1]:
        #             SiSt_data[col].append(STprocessed[col][row] -
STprocessed[col][1])
        #         else:
        #             StSi_data[col].append(STprocessed[col][row] -
STprocessed[col][1])
        # else:
        #     for row in range(1,nbRow):
        #         if row < indexChange[1]:
        #             SiSt_data[col].append(STprocessed[col][row]+
STprocessed[col][1])
        #         else:
        #             StSi_data[col].append(STprocessed[col][row] +
STprocessed[col][1])

        for row in range(1,nbRow):
            if row < indexChange[1]:
                SiSt_data[col].append(STprocessed[col][row])
            else:
                StSi_data[col].append(STprocessed[col][row])


fs = 500  # Sampling frequency
cutoff = 50  # Desired cutoff frequency in Hz
order = 6  # Filter order

filtered_SiSt = butter_lowpass_filter(SiSt_data[col_end_trans][1:],
cutoff, fs, order)



median_pk2pk = MedianPeakToPeak(filtered_SiSt[0:indexChange[0]])
max_pk2pk_index = MaxPeakToPeak(filtered_SiSt)
Threshold_Index = PeakToPeakTreshold(filtered_SiSt[max_pk2pk_index:],
median_pk2pk) + max_pk2pk_index
print("EMG index: " + str(Threshold_Index))


median_pk2pk_gyro = MedianPeakToPeak(SiSt_data[col_end_trans_gyro][1:])
```

```
Threshold_Index_gyro = EndMaxPeak(SiSt_data[col_end_trans_gyro][1:],
median_pk2pk_gyro)
print("Gyro index: " + str(Threshold_Index_gyro))

idx_start_trans_SiSt = StartTransition(SiSt_data[45][1:indexChange[1]],
SiSt_data[47][1:indexChange[1]], indexChange[0], indexChange[1])


index_end_trans_SiSt = EndTransition(SiSt_data[45][indexChange[0]:]) +
indexChange[0]
index_end_trans_SiSt2, idx_end_standing = EndTransition2(allData[45][1:],
allData[47][1:])

print("gonio index: " +str(index_end_trans_SiSt))

index_end_trans_StSi = StartSitting(StSi_data[45][indexChange[3]-
indexChange[2]:]) + indexChange[3]

#print(indexChange[3])


hip_angle_R, tilt_hip = HipAngle(allData[26][1:], allData[25][1:],
allData[29][1:], allData[8][1:], allData[6][1:], allData[9][1:],
indexChange[1]) #Right allData[8][1:], allData[6][1:], allData[9]
hip_angle_L, _ = HipAngle(allData[26][1:], allData[25][1:],
allData[29][1:], allData[20][1:], allData[18][1:], allData[21][1:],
indexChange[1])
# Left allData[20][1:], allData[18][1:], allData[21][1:]

"""
SiSt_sit_end, SiSt_lean_end, SiSt_stand_start, SiSt_stand_end =
SiStTransHipAngle(hip_angle_R[1:],hip_angle_L[1:], indexChange[0],
indexChange[1], idx_start_trans_SiSt, tilt_hip)

StSi_stand_end, StSi_lean_end, StSi_sit_start  =
StSiTransHipAngle(hip_angle_R[1:],hip_angle_L[1:], indexChange[2],
indexChange[3], index_end_trans_StSi, tilt_hip)
"""

# Find the indices of the positive and negative peaks in the signal
#pos_peak_indices, _ = find_peaks(np.array(filtered_SiSt))
#neg_peak_indices, _ = find_peaks(-np.array(filtered_SiSt))


#----------------------Plot the data----------------------#


SiStData = allData[col_plot_data][1:len(allData[col_plot_data])-1]

"""
SiStData = allData[35][1:len(allData[col_plot_data])-1]
SiStData2 = allData[42][1:len(allData[col_plot_data])-1]

SiStData3 = allData[45][1:len(allData[col_plot_data])-1]
SiStData4 = allData[47][1:len(allData[col_plot_data])-1]
```

```
for i in range(len(SiStData3)):
    test = SiStData3[i]
    SiStData3[i] = SiStData3[i]/100
    SiStData4[i] = SiStData4[i]/100
"""

#SiStData = filtered_SiSt


if multi == 1:

    SiStDataY = allData[col_plot_data+1][1:len(allData[col_plot_data])-1]
    SiStDataZ = allData[col_plot_data+2][1:len(allData[col_plot_data])-1]

freq = 500
SiStTime = [i/freq for i in range(len(SiStData))]




# create a new figure
fig, ax = plt.subplots()

#ax.plot(SiStTime, SiStData)



""" # plot the data
ax.plot(SiStTime, SiStData, label= allData[35][0])
ax.plot(SiStTime, SiStData2, label= allData[42][0])

ax.plot(SiStTime, SiStData3, label= allData[45][0])
ax.plot(SiStTime, SiStData4, label= allData[47][0]) """


ax.plot(SiStTime, hip_angle_R, label= "hip angle right")
ax.plot(SiStTime, hip_angle_L, label= "hip angle left")
ax.plot(SiStTime, tilt_hip, label= "hip tilt")


#ax.plot(SiStTime, allData[29][1:len(allData[col_plot_data])-1], label=
"gyroWX")
#ax.plot(SiStTime, allData[21][1:len(allData[col_plot_data])-1], label=
"gyro Thigh Left")
#ax.plot(SiStTime, allData[9][1:len(allData[col_plot_data])-1], label=
"gyro Thigh Right")


"""
# Add a cross at each peak index
for index in staticW:
    plt.plot(SiStTime[index], hip_angle_R[index], 'rx')
for index in staticT:
```

```
    plt.plot(SiStTime[index], hip_angle_R[index], 'ro')
"""


"""
# Add a cross at each peak index
for index in peak_indices_R:
    plt.plot(SiStTime[index], hip_angle_R[index], 'rx')
for index in peak_indices_L:
    plt.plot(SiStTime[index], hip_angle_L[index], 'ro')
"""

if multi == 1:
    ax.plot(SiStTime, SiStDataY, label= allData[col_plot_data+1][0])
    ax.plot(SiStTime, SiStDataZ, label= allData[col_plot_data+2][0])

#Plot change of state with a vertical dash line
#plt.axvline(x = indexChange[0]/freq, color = 'b', linestyle = 'dashed',
label = 'start standing')
#plt.axvline(x = indexChange[1]/freq, color = 'b', linestyle = 'dashed',
label = 'start walking')
#plt.axvline(x = indexChange[2]/freq, color = 'b', linestyle = 'dashed',
label = 'end walking')
#plt.axvline(x = indexChange[3]/freq, color = 'b', linestyle = 'dashed',
label = 'start sitting')
#plt.axvline(x = SiStTime[idx_start_trans_SiSt], color = 'r', label =
'new start transition')
#plt.axvline(x = SiStTime[idx_end_standing], color = 'r', linestyle =
'dashed', label = 'new end standing')

"""
plt.axvline(x = indexChange[0]/freq, color = 'b', linestyle = 'dashed')
plt.axvline(x = indexChange[1]/freq, color = 'b', linestyle = 'dashed')
plt.axvline(x = indexChange[2]/freq, color = 'b', linestyle = 'dashed')
plt.axvline(x = indexChange[3]/freq, color = 'b', linestyle = 'dashed')
"""

"""
plt.axvline(x = SiStTime[SiSt_sit_end], color = 'y', linestyle =
'dashed', label = 'SiSt start transition hip')
plt.axvline(x = SiStTime[SiSt_lean_end], color = 'r', linestyle =
'dashed', label = 'SiSt end leaning forward')
plt.axvline(x = SiStTime[SiSt_stand_start], color = 'g', linestyle =
'dashed', label = ' SiSt end transiton')
plt.axvline(x = SiStTime[SiSt_stand_end], color = 'b', label = ' SiSt end
standing')
"""

#plt.axvline(x = SiStTime[StSi_stand_end], color = 'y', linestyle =
'dashed', label = 'StSi end standing')
#plt.axvline(x = SiStTime[StSi_lean_end], color = 'r', linestyle =
'dashed', label = 'StSi end leaning forward')
#plt.axvline(x = SiStTime[StSi_sit_start], color = 'g', linestyle =
'dashed', label = 'StSi start sitting')
```

```
#plt.axvline(x = SiStTime[idx_start_trans_SiSt], color = 'r', label =
'knee band')
idx_start_trans_SiSt
#plt.axvline(x = 2500/500, color = 'g', linestyle = 'dashed', label =
'test')


#plt.axvline(x = SiStTime[max_pk2pk_index], color = 'r', linestyle =
'dashed', label = 'max peak to peak')
#plt.axvline(x = SiStTime[Threshold_Index], color = 'g', linestyle =
'dashed', label = 'end transition')
#plt.axvline(x = SiStTime[Threshold_Index_gyro], color = 'r', linestyle =
'dashed', label = 'end transition gyro')
#plt.axvline(x = SiStTime[index_end_trans_SiSt], color = 'y', linestyle =
'dashed', label = 'end transition gonio')
#plt.axvline(x = SiStTime[index_end_trans_SiSt2], color = 'r', linestyle
= 'dashed', label = 'end transition gonio')

#plt.axvline(x = SiStTime[index_end_trans_StSi], color = 'g', linestyle =
'dashed', label = 'end transition gonio')


handles, labels = ax.get_legend_handles_labels()

# reverse the order
ax.legend(handles[::-1], labels[::-1], loc='upper right')

# set the x and y axis labels
ax.set_xlabel('time (s)')

# set the title of the plot
ax.set_title(allData[col_plot_data][0])

# display the plot
plt.show()


#plt.plot(time1, plot1)
#plt.yticks(labels)
#plt.show

print("done")
```

8.2 – Relabel.py

Relabel.py

```
import csv
import os
import numpy as np
import math
```

```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import MinMaxScaler

from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn import metrics

import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense, Input
from keras.optimizers import Adam

from scipy.signal import butter, lfilter, find_peaks

import pywt



def readData(filename):

    data = []

    #Open .csv file for read
    with open(filename +'.csv', 'r') as csvfile:
        reader = csv.reader(csvfile)

        # Create objects for each header
        # Assign the name of each header to the objects
        for row in reader:
            for i in range(len(row)): data.append([row[i]])
            break

        # Assign the value of each column to a different object
        for row in reader:
            for i in range(len(data)):
                if row[i] != "":
                    data[i].append(float(row[i]))

    return data




def StartTransition(signal_R, signal_L, init_start, init_end):

    median_R = np.median(signal_R[:init_start])
    median_L = np.median(signal_L[:init_start])

    max_R_idx = np.argmax(signal_R[:init_end])
    max_L_idx = np.argmax(signal_L[:init_end])

    range_R = abs(signal_R[max_R_idx] - median_R)
    range_L = abs(signal_L[max_L_idx] - median_L)
```

```
    for i in range(len(signal_R[:init_end])-1):
        if signal_R[i] > (median_R + range_R*0.02) and signal_L[i] >
(median_L + range_L*0.02):
            return i



def StartSitting(signal, init_start):

    min_index = np.argmin(signal[init_start:]) + init_start
    median = np.median(signal[min_index:])
    max_index = np.argmax(signal[init_start:]) + init_start

    range = abs(signal[max_index] - median)

    for i, value in enumerate(signal[max_index:]):
        if value < (signal[max_index] - range*0.95):
            break

    return i + max_index



def HipAngle(acc_WZ, acc_WY, gyro_WX,acc_TZ,acc_TX, gyro_TY, init_end):

    dt = 1/500
    init_0 = 250
    new_0 = 245

    gyro_WX = np.array(gyro_WX) * 2 * np.pi / 360
    gyro_TY = np.array(gyro_TY) * 2 * np.pi / 360


    integral = np.sum(gyro_TY[:init_end]) - gyro_TY[0]*init_end
    if integral < 0:
        gyro_TY = -gyro_TY
        print("inverted gyro thigh")

    gyro_W0 = 0
    gyro_T0 = 0
    tilt_W0 =0
    tilt_T0 = 0
    for i in range(50):
        gyro_W0 = gyro_W0 + gyro_WX[i]/init_0
        gyro_T0 = gyro_T0 + gyro_TY[i]/init_0
        tilt_W0 = tilt_W0 + np.arctan2(acc_WZ[i],acc_WY[i])/init_0
        tilt_T0 = tilt_T0 + np.arctan2(acc_TZ[i],acc_TX[i])/init_0

    tilt_W = np.zeros(len(gyro_WX)-1)
    tilt_T = np.zeros(len(gyro_WX)-1)
    angle_hip = np.zeros(len(gyro_WX)-1)
    for i in range(len(gyro_WX)-1):

        if i < init_0:
```

```
            tilt_W[i] = tilt_W0
            tilt_T[i] = tilt_T0
            angle_hip[i] = tilt_W0 - tilt_T0
        else:

            tiltW_deg = tilt_W[i-new_0:i]*180/np.pi
            tiltT_deg = tilt_T[i-new_0:i]*180/np.pi

            #variance_W = math.sqrt((np.sum((tiltW_deg**2)*dt) -
np.sum(tiltW_deg*dt)**2)/new_0)
            #variance_T = math.sqrt((np.sum((tiltT_deg**2)*dt) -
np.sum(tiltT_deg*dt)**2)/new_0)

            variance_W = np.sum((tiltW_deg -
np.sum(tiltW_deg)/new_0)**2)/new_0
            variance_T = np.sum((tiltT_deg -
np.sum(tiltT_deg)/new_0)**2)/new_0

            #variance_W = math.sqrt((np.sum((tilt_W[i-new_0:i]**2)*dt) -
np.sum(tilt_W[i-new_0:i]*dt)**2)/new_0)
            #variance_T = math.sqrt((np.sum((tilt_T[i-new_0:i]**2)*dt) -
np.sum(tilt_T[i-new_0:i]*dt)**2)/new_0)

            #variance_W = np.sum((tilt_W[i-new_0:i] - np.sum(tilt_W[i-
new_0:i])/new_0)**2)/new_0
            #variance_T = np.sum((tilt_T[i-new_0:i] - np.sum(tilt_T[i-
new_0:i])/new_0)**2)/new_0


            tilt_W[i] = tilt_W[i-1] + (gyro_WX[i] - gyro_W0 ) * dt
            tilt_T[i] = tilt_T[i-1] + (gyro_TY[i] - gyro_T0) * dt
            angle_hip[i] = tilt_W[i] - tilt_T[i]


        angle_hip[i] = angle_hip[i] * 360 / (2 * np.pi)

    tilt_W = tilt_W * 360 / (2 * np.pi)

    return angle_hip, tilt_W



def SiStTransHipAngle(signal_R, signal_L, init_start, init_end,
knee_start, hip_tilt_signal):


    max_R_idx = np.argmax(signal_R[:init_end])
    max_L_idx = np.argmax(signal_L[:init_end])

    # Index where the trunk end leaning forward
    # Do it by finding the max angle forward
    max_idx = max(max_R_idx, max_L_idx)

    #Find the index where the trunk start leaning forward
```

```
    # Do it by finding when the hip angle is 5% of the range between the
static angle and the max angle forward

    delta = 100
    treshold = 0.05

    check_R = 0
    check_L = 0

    for i in range(max_idx):

        gradient_R = abs(signal_R[i+delta] - signal_R[i])/(delta)
        gradient_L = abs(signal_L[i+delta] - signal_L[i])/(delta)

        if gradient_R > treshold:
            check_R = 1
        if gradient_L > treshold:
            check_L = 1

        if check_R == 1 and check_L == 1:
            break

    start_trans = i


    # Find the index where the trunk stops leaning backward
    # Do it by finding when the slop decrease

    delta = 50
    treshold = 0.01

    check_tilt = 0
    check_R = 0
    check_L = 0

    max_hip_tilt_idx= np.argmax(hip_tilt_signal[:init_end])

    for i in range(max_hip_tilt_idx,init_end*2):

        gradient_tilt = abs(hip_tilt_signal[i+delta] -
hip_tilt_signal[i])/(delta)
        gradient_R = abs(signal_R[i+delta] - signal_R[i])/(delta)
        gradient_L = abs(signal_L[i+delta] - signal_L[i])/(delta)

        if gradient_tilt < treshold:
            check_tilt = 1
        if gradient_R < treshold:
            check_R = 1
        if gradient_L < treshold:
            check_L = 1

        if (check_R == 1 and check_tilt == 1) or (check_L == 1 and
check_tilt == 1):
            break
```

```
    end_trans = i


    diff_start = abs(signal_R[end_trans] - signal_L[end_trans])
    check_diff = 0
    for i in range(end_trans, end_trans+ 5000):
        diff_end = abs(signal_R[i] - signal_L[i])
        if diff_start < 25:
            if diff_end > 25:
                break
        else:
            if diff_end > diff_start*1.5:
                break
            if diff_end < diff_start/10 and check_diff == 0:
                check_diff = 1
            if check_diff == 1 and diff_end > diff_start*1.2:
                break

    end_stand = i


    return start_trans, max_idx, end_trans, end_stand



def StSiTransHipAngle(signal_R, signal_L, init_start, init_end, knee_end,
hip_tilt_signal):

    peak_indices_R, _ = find_peaks(np.array(signal_R[init_start:]),
prominence=5)
    peak_indices_L, _ = find_peaks(np.array(signal_L[init_start:]),
prominence=5)
    peak_indices_tilt, _ =
find_peaks(np.array(hip_tilt_signal[init_start:]))

    diff = [abs(init_start + value - knee_end) for _, value in
enumerate(peak_indices_R)]
    max_R_idx = peak_indices_R[diff.index(min(diff))]
    diff = [abs(init_start + value - knee_end) for _, value in
enumerate(peak_indices_L)]
    max_L_idx = peak_indices_L[diff.index(min(diff))]
    diff = [abs(init_start + value - knee_end) for _, value in
enumerate(peak_indices_tilt)]
    max_tilt_idx = peak_indices_tilt[diff.index(min(diff))]

    max_idx = max(max_R_idx, max_L_idx) + init_start

    """
    for i in range(len(signal_R[max_idx:])-1):
        if signal_R[i] < (median_R + range_R*0.02) and signal_L[i] <
(median_L + range_L*0.02):
            break

    end_trans = i + max_idx
    """
```

```
    # Find the index where the trunk stops leaning backward
    # Do it by finding when the slop decrease

    min_R_end = np.argmin(signal_R[max_idx:])
    min_L_end = np.argmin(signal_L[max_idx:])
    min_tilt_end = np.argmin(hip_tilt_signal[max_idx:])

    check = 0
    if min_R_end > len(signal_R[max_idx:]) - 100:
        check = check +1
    if min_L_end > len(signal_R[max_idx:])- 100:
        check = check +1
    if min_tilt_end > len(signal_R[max_idx:])- 100:
        check = check +1

    delta = 150

    if check >= 2:
        treshold = 0.03
    else:
        treshold = 0.01

    check_tilt = 0
    check_R = 0
    check_L = 0

    for i in range(max_idx + 100, len(signal_R)-1):

        if i + delta >= len(signal_R)-2:
            i = len(signal_R)-2
            break

        gradient_tilt = abs(hip_tilt_signal[i+delta] -
hip_tilt_signal[i])/(delta)
        gradient_hip_R = abs(signal_R[i+delta] - signal_R[i])/(delta)
        gradient_hip_L = abs(signal_L[i+delta] - signal_L[i])/(delta)

        if gradient_tilt < treshold:
            check_tilt = 1
        if gradient_hip_R < treshold:
            check_R = 1
        if gradient_hip_L < treshold:
            check_L = 1

        if check_tilt == 1 and check_R == 1 and check_L == 1:
            break

    end_trans = i


    # Find the index where the trunk start leaning forward
    # Do it by finding when the hip tilt slop decrease.
    # Starts from the end of the trunk leaning forward and goes backward
```

```
    max_hip_tilt = np.argmax(hip_tilt_signal[init_start:]) +init_start -
1
    max_hip_tilt = max_tilt_idx + init_start

    delta = 50
    treshold_grad = 0.008
    treshold_range = 0.3

    check_tilt = 0
    check_R = 0
    check_L = 0

    min_tilt = np.min(hip_tilt_signal[init_start:max_hip_tilt])
    min_R = np.min(signal_R[init_start:max_R_idx+ init_start])
    min_L = np.min(signal_L[init_start:max_L_idx+ init_start])

    range_tilt = abs(tilt_hip[max_hip_tilt] - min_tilt)
    range_R = abs(signal_R[max_R_idx+ init_start] - min_R)
    range_L = abs(signal_L[max_L_idx+ init_start] - min_L)

    for i in range(max_hip_tilt - init_start + 1000):

        n = max_hip_tilt - i
        gradient_tilt = abs(tilt_hip[n-delta] - tilt_hip[n])/(delta)
        gradient_hip_R = abs(signal_R[n-delta] - signal_R[n])/(delta)
        gradient_hip_L = abs(signal_L[n-delta] - signal_L[n])/(delta)

        if gradient_tilt < treshold_grad and tilt_hip[n] < (min_tilt +
range_tilt*treshold_range):
            check_tilt = 1
        if gradient_hip_R < treshold_grad and signal_R[n] < (min_R +
range_R*treshold_range):
            check_R = 1
        if gradient_hip_L < treshold_grad and signal_L[n] < (min_L +
range_L*treshold_range):
            check_L = 1

        if check_R +check_L + check_tilt >= 2:
            break

    start_trans = n

    return start_trans, max_idx, end_trans




#------------------Main------------------#


col_end_trans = 35
col_end_trans_gyro = 29
col_plot_data = 35
```

```
sensor = "accelerometer"  #accelerometer, gyroscope, IMU, EMG, all
SiSt = 1
StSi = 0
multi = 0


thisFolderParent = os.getcwd()

candidate = ["156","185", "186", "188", "189", "190", "191", "192",
"193", "194"]
#c = 6

# Initialise the lists
Sit = []
Stand = []
all_windows = []

for c in range(0,len(candidate)):
    print("Candidate " + candidate[c])

    # Loop through all the circuits of each candidate
    for circ in range(1 ,51):
        print("Circuit " + str(circ))
        if circ<10:
            filename  = thisFolderParent + "/Data/Source/AB" +
candidate[c] + "/Processed/AB" + candidate[c] + "_Circuit_00" + str(circ)
+ "_post"
        else:
            filename  = thisFolderParent + "/Data/Source/AB" +
candidate[c] + "/Processed/AB" + candidate[c] + "_Circuit_0" + str(circ)
+ "_post"

        #filename  = thisFolderParent +
"/Data/AB156/Processed/AB156_Circuit_002_post"

        if os.path.exists(filename + ".csv"):

            allData = readData(filename)

            for i in range(len(allData)):
                if allData[i][0] == "Mode":
                    m = i
                    break

            indexChange = []
            prev_label = int(allData[m][1])
            for i in range(2,len(allData[m])):
                current_label = int(allData[m][i])
                if current_label != prev_label:
                    indexChange.append(i)
                    if len(indexChange) >= 2:
                        break
                prev_label = current_label

            prev_label = int(allData[m][len(allData[m])-1])
```

```
            for i in range(2,len(allData[m])):
                n = len(allData[m])-i
                current_label = int(allData[m][n])
                if current_label != prev_label:
                    indexChange.append(n)
                    if len(indexChange) >= 4:
                        break
                prev_label = current_label

            idx4 = indexChange[3]
            idx3 = indexChange[2]
            indexChange[2] = idx4
            indexChange[3] = idx3



            idx_start_trans_SiSt =
StartTransition(allData[45][1:indexChange[1]],
allData[47][1:indexChange[1]], indexChange[0], indexChange[1])
            index_end_trans_StSi = StartSitting(allData[45],
indexChange[2])


            hip_angle_R, tilt_hip = HipAngle(allData[26][1:],
allData[25][1:], allData[29][1:], allData[8][1:], allData[6][1:],
allData[9][1:], indexChange[1])
            hip_angle_L, _ = HipAngle(allData[26][1:], allData[25][1:],
allData[29][1:], allData[20][1:], allData[18][1:], allData[21][1:],
indexChange[1])


            SiSt_sit_end, SiSt_lean_end, SiSt_stand_start, SiSt_stand_end
= SiStTransHipAngle(hip_angle_R[1:],hip_angle_L[1:], indexChange[0],
indexChange[1], idx_start_trans_SiSt, tilt_hip)

            StSi_stand_end, StSi_lean_end, StSi_sit_start  =
StSiTransHipAngle(hip_angle_R[1:],hip_angle_L[1:], indexChange[2],
indexChange[3], index_end_trans_StSi, tilt_hip)


            freq = 500
            SiStTime = [i/freq for i in range(len(allData[0][1:])-1)]

            """
            # create a new figure
            fig, ax = plt.subplots()

            ax.plot(SiStTime, hip_angle_R, label= "hip angle right")
            ax.plot(SiStTime, hip_angle_L, label= "hip angle left")
            ax.plot(SiStTime, tilt_hip, label= "hip tilt")

            plt.axvline(x = SiStTime[SiSt_sit_end], color = 'y',
linestyle = 'dashed', label = 'SiSt start transition hip')
            plt.axvline(x = SiStTime[SiSt_lean_end], color = 'r',
linestyle = 'dashed', label = 'SiSt end leaning forward')
```

```
            plt.axvline(x = SiStTime[SiSt_stand_start], color = 'g',
linestyle = 'dashed', label = ' SiSt end transiton')
            plt.axvline(x = indexChange[1]/freq, color = 'b', linestyle =
'dashed', label = 'start walking')
            plt.axvline(x = SiStTime[SiSt_stand_end], color = 'b', label
= ' SiSt end standing')

            plt.axvline(x = indexChange[2]/freq, color = 'b', linestyle =
'dashed', label = 'end walking')
            plt.axvline(x = SiStTime[StSi_stand_end], color = 'y',
linestyle = 'dashed', label = 'StSi end standing')
            plt.axvline(x = SiStTime[StSi_lean_end], color = 'r',
linestyle = 'dashed', label = 'StSi end leaning forward')
            plt.axvline(x = SiStTime[StSi_sit_start], color = 'g',
linestyle = 'dashed', label = 'StSi start sitting')

            plt.show()
            """


            #-------------------Write output file-------------------#
            if circ<10:
                out_file_name  = thisFolderParent + "/Data/New/AB" +
candidate[c] + "/AB" + candidate[c] + "_Circuit_00" + str(circ) + "_SiSt"
            else:
                out_file_name  = thisFolderParent + "/Data/New/AB" +
candidate[c] + "/AB" + candidate[c] + "_Circuit_0" + str(circ) + "_SiSt"

            file = open(out_file_name +'.csv', 'w')
            file.truncate()

            for i in range(m):
                file.write(allData[i][0] + ",")
            file.write("right_hip_angle,")
            file.write("left_hip_angle,")
            file.write("hip_tilt,")
            file.write("mode")
            file.write("\n")

            for n in range(1,SiSt_stand_end):
                for i in range(m):
                    file.write(str(allData[i][n]) + ",")
                file.write(str(hip_angle_R[n]) + ",")
                file.write(str(hip_angle_L[n]) + ",")
                file.write(str(tilt_hip[n]) + ",")
                if n < SiSt_sit_end:
                    file.write("1\n")
                elif n < SiSt_lean_end:
                    file.write("121\n")
                elif n < SiSt_stand_start:
                    file.write("122\n")
                else:
                    file.write("2\n")

            file.close()
```

```
            if circ<10:
                out_file_name  = thisFolderParent + "/Data/New/AB" +
candidate[c] + "/AB" + candidate[c] + "_Circuit_00" + str(circ) + "_StSi"
            else:
                out_file_name  = thisFolderParent + "/Data/New/AB" +
candidate[c] + "/AB" + candidate[c] + "_Circuit_0" + str(circ) + "_StSi"

            file = open(out_file_name +'.csv', 'w')

            for i in range(m):
                file.write(allData[i][0] + ",")
            file.write("right_hip_angle,")
            file.write("left_hip_angle,")
            file.write("hip_tilt,")
            file.write("mode")
            file.write("\n")

            for n in range(indexChange[2], len(allData[0])-2):
                for i in range(m):
                    file.write(str(allData[i][n]) + ",")
                file.write(str(hip_angle_R[n]) + ",")
                file.write(str(hip_angle_L[n]) + ",")
                file.write(str(tilt_hip[n]) + ",")
                if n > StSi_sit_start:
                    file.write("1\n")
                elif n > StSi_lean_end:
                    file.write("212\n")
                elif n > StSi_stand_end:
                    file.write("211\n")
                else:
                    file.write("2\n")


            file.close()
```

8.3 – ExtractFeatures.py

ExtractFeatures.py

```
import csv
import os
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.preprocessing import MinMaxScaler

from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn import metrics

import tensorflow as tf
```

```
from keras.models import Sequential
from keras.layers import Dense, Input
from keras.optimizers import Adam



def Main(filename):

    data = []

    #Open .csv file for read
    with open(filename +'.csv', 'r') as csvfile:
        reader = csv.reader(csvfile)

        # Create objects for each header
        # Assign the name of each header to the objects
        for row in reader:
            for i in range(len(row)):
                data.append([row[i]])
                if row[i] == "mode":
                    break
            break

        # Assign the value of each column to a different object
        x = 1
        for row in reader:
            for i in range(len(data)):
                if row[i] != "":
                    data[i].append(float(row[i]))

    return data


def featureExtraction(data, window_time, freq, mode, sensor, activity):

    #Indicate which columns to use for each sensor
    if sensor == "accelerometer":
        cols = [0,2,6,8,12,14,18,20,25,26]
    elif sensor == "gyroscope":
        cols = [3,9,15,21,29]
    elif sensor == "IMU":
        cols = [*[0,2,6,8,12,14,18,20,25,26],*[3,9,15,21,29]]
    elif sensor == "EMG":
        cols = [*range(30,44)]
    elif sensor == "all":
        cols =
[*[0,2,6,8,12,14,18,20,25,26],*[3,9,15,21,29],*range(30,44),
*range(52,55)]

    window_size = int(window_time * freq)

    #Create a list of dictionaries
    w = 0
    windows = [{}]
```

```
    #Create a dictionary for each sensor (X,Y,Z and Positions) for the
first window
    for col in cols:
        windows[w].update({data[col][0]:{"data" : np.zeros(window_size,
dtype = float)} })
    if mode == "train":
        windows[w].update({"label" : np.zeros(window_size) })

    #Loop through every window
    n = 0
    for i in range(1, len(data[0])):
        j = i-1-(w * window_size)

        #Initialize a new window
        if j >= window_size:
            w = w + 1
            windows.append({})
            for col in cols:
                windows[w].update({data[col][0]:{"data" :
np.zeros(window_size, dtype = float)} })
            if mode == "train":
                windows[w].update({"label" : np.zeros(window_size)})
            j = 0

        #Add data to the current window
        if j < window_size:
            for col in cols:
                windows[w][data[col][0]]["data"][j] = data[col][i]
            if mode == "train":
                windows[w]["label"][j] = data[len(data)-1][i]

    windows.pop()
    # The last window is not full of data, so most of it is 0


    #Calculate features for each window
    for i in range(len(windows)):
        for col in cols:
            windows[i][data[col][0]]["mean"] =
np.mean(windows[i][data[col][0]]["data"])         #Mean
            windows[i][data[col][0]]["max"] =
np.max(windows[i][data[col][0]]["data"])           #Maximum
            windows[i][data[col][0]]["min"] =
np.min(windows[i][data[col][0]]["data"])           #Minimum
            windows[i][data[col][0]]["std"] =
np.std(windows[i][data[col][0]]["data"])           #Standard deviation
            windows[i][data[col][0]]["var"] =
np.var(windows[i][data[col][0]]["data"])           #Variance
            windows[i][data[col][0]]["rms"] =
np.sqrt(np.mean(windows[i][data[col][0]]["data"]**2))
#Root mean square
            windows[i][data[col][0]]["skew"] =
np.mean((windows[i][data[col][0]]["data"] -
np.mean(windows[i][data[col][0]]["data"]))**3)           #Skewness
```

```
            windows[i][data[col][0]]["kurt"] =
np.mean((windows[i][data[col][0]]["data"] -
np.mean(windows[i][data[col][0]]["data"]))**4)           #Kurtosis
            windows[i][data[col][0]]["iqr"] =
np.subtract(*np.percentile(windows[i][data[col][0]]["data"], [75, 25]))
#Interquartile range
            windows[i][data[col][0]]["mad"] =
np.mean(np.absolute(windows[i][data[col][0]]["data"] -
np.mean(windows[i][data[col][0]]["data"])))    #Mean absolute deviation
            windows[i][data[col][0]]["ptp"] =
np.ptp(windows[i][data[col][0]]["data"])
#Peak to peak
            windows[i][data[col][0]]["energy"] =
np.sum(windows[i][data[col][0]]["data"]**2)
#Energy
            #windows[i][data[col][0]]["arCoeff"] =
np.polyfit(np.arange(0,window_size), windows[i][data[col][0]]["data"], 1)
#Linear regression
            #windows[i][data[col][0]]["arCoeff"] =
windows[i][data[col][0]]["arCoeff"][0]
#Slope of the linear regression


        #Give a label to the window
        if mode ==  "train":
            sit = 0
            trans1 = 0
            trans2 = 0
            stand = 0
            if activity == "SiSt":
                for n in range(window_size):
                    if int(windows[i]["label"][n]) == 1:
                        sit = sit + 1
                    elif int(windows[i]["label"][n]) == 2:
                        stand = stand + 1
                    elif int(windows[i]["label"][n]) == 121:
                        trans1 = trans1 + 1
                    elif int(windows[i]["label"][n]) == 122:
                        trans2 = trans2 + 1

                if max(sit, stand, trans1, trans2) == sit:
                    windows[i]["label"] = 1
                elif max(sit, stand, trans1, trans2) == stand:
                    windows[i]["label"] = 2
                elif max(sit, stand, trans1, trans2) == trans1:
                    windows[i]["label"] = 121
                elif max(sit, stand, trans1, trans2) == trans2:
                    windows[i]["label"] = 122

            elif activity == "StSi":
                for n in range(window_size):
                    if int(windows[i]["label"][n]) == 1:
                        sit = sit + 1
                    elif int(windows[i]["label"][n]) == 2:
                        stand = stand + 1
```

```
                    elif int(windows[i]["label"][n]) == 211:
                        trans1 = trans1 + 1
                    elif int(windows[i]["label"][n]) == 212:
                        trans2 = trans2 + 1

                if max(sit, stand, trans1, trans2) == sit:
                    windows[i]["label"] = 1
                elif max(sit, stand, trans1, trans2) == stand:
                    windows[i]["label"] = 2
                elif max(sit, stand, trans1, trans2) == trans1:
                    windows[i]["label"] = 211
                elif max(sit, stand, trans1, trans2) == trans2:
                    windows[i]["label"] = 212

    return windows, len(cols)



def idxLabelChange(Data):

    for i in range(len(Data)):
        if Data[i][0] == "mode":
            m = i
            break

    ChangeIdxs = []
    prev_label = int(Data[m][1])
    for i in range(2,len(Data[m])):
        current_label = int(Data[m][i])
        if current_label != prev_label:
            ChangeIdxs.append(i)
            if len(ChangeIdxs) >= 3:
                break
        prev_label = current_label

    return ChangeIdxs




#-------------------Main-------------------#


col = 0
sensor = "gyroscope"  #accelerometer, gyroscope, IMU, EMG, all
SiSt = 0
multi = 1
window_len = 0.025 #in seconds
feature_or_data = "Data"  #Features


thisFolderParent = os.getcwd()
```

```
candidate = ["156","185", "186", "188", "189", "190", "191", "192",
"193", "194"]

first_candidate = 0
last_candidate = 9

first_circuit = 1
last_circuit = 50


all_sensor = ["accelerometer", "gyroscope", "IMU", "EMG", "all"]


for sensor in all_sensor:

    all_window_len = [0.05, 0.075, 0.1, 0.15, 0.2, 0.3, 0.5]

    for window_len in all_window_len:

        print("Sensor: " + sensor)
        print("Window length: " + str(window_len) + "s")

        for act in range(0,2):
            if act ==  1:
                activity = "SiSt"
            elif act == 0:
                activity = "StSi"

            for c in range(first_candidate, last_candidate+1):
                print("Candidate " + candidate[c])
                all_windows = []

                # Loop through all the circuits of each candidate
                for circ in range(first_circuit, last_circuit+1):
                    #print("Circuit " + str(circ))
                    """
                    if act == 1:
                        if circ<10:
                            filename  = thisFolderParent + "/Data/New/AB"
+ candidate[c] + "/AB" + candidate[c] + "_Circuit_00" + str(circ) +
"_SiSt"
                        else:
                            filename  = thisFolderParent + "/Data/New/AB"
+ candidate[c] + "/AB" + candidate[c] + "_Circuit_0" + str(circ) +
"_SiSt"
                    else:
                        if circ<10:
                            filename  = thisFolderParent + "/Data/New/AB"
+ candidate[c] + "/AB" + candidate[c] + "_Circuit_00" + str(circ) +
"_StSi"
                        else:
                            filename  = thisFolderParent + "/Data/New/AB"
+ candidate[c] + "/AB" + candidate[c] + "_Circuit_0" + str(circ) +
"_StSi"
```

```
                    """
                    hardrive_path = "/Volumes/UnionSine/IMEE
2022/Semester2/FYP/FYPcode"

                    if act == 1:
                        if circ<10:
                            filename  =  hardrive_path + "/New/AB" +
candidate[c] + "/AB" + candidate[c] + "_Circuit_00" + str(circ) + "_SiSt"
                        else:
                            filename  =  hardrive_path + "/New/AB" +
candidate[c] + "/AB" + candidate[c] + "_Circuit_0" + str(circ) + "_SiSt"
                    else:
                        if circ<10:
                            filename  =  hardrive_path + "/New/AB" +
candidate[c] + "/AB" + candidate[c] + "_Circuit_00" + str(circ) + "_StSi"
                        else:
                            filename  =  hardrive_path + "/New/AB" +
candidate[c] + "/AB" + candidate[c] + "_Circuit_0" + str(circ) + "_StSi"


                    if os.path.exists(filename + ".csv"):

                        allData = Main(filename)

                        label_change = idxLabelChange(allData)

                        windows, nb_sensors = featureExtraction(allData,
window_len, 500, "train", sensor, activity)


                        for n in range(len(windows)):
                            all_windows.append(windows[n])


                if feature_or_data == "Features":
                    #Transform dictionnary in array
                    features_only = []
                    labels = []
                    for i in range(len(all_windows)):
                        features_only.append([])
                        test = all_windows[i]
                        for j, header in enumerate(all_windows[i]):
                            if  header != "label":
                                for n, featur in
enumerate(all_windows[i][header]):
                                    if featur != "data":
                                        test2 =
all_windows[i][header][featur]

features_only[i].append(all_windows[i][header][featur])
                            else:
                                labels.append(all_windows[i]["label"])

                    headers = []
                    for j, header in enumerate(all_windows[0]):
```

```
                        if  header != "label":
                            for n, featur in
enumerate(all_windows[i][header]):
                                if featur != "data":
                                    test2 =
all_windows[i][header][featur]
                                    headers.append(header + "_" + featur)

                    nb_features = int(len(headers)/nb_sensors)

                else:
                    #Transform dictionnary in array
                    data_only = []
                    labels = []
                    for i in range(len(all_windows)):
                        data_only.append([])
                        test = all_windows[i]
                        for j, header in enumerate(all_windows[i]):
                            if  header != "label":
                                for n, featur in
enumerate(all_windows[i][header]):
                                    if featur == "data":
                                        test2 =
all_windows[i][header][featur]
                                        for d in
range(len(all_windows[i][header][featur])):

data_only[i].append(all_windows[i][header][featur][d])
                            else:
                                labels.append(all_windows[i]["label"])

                    headers = []
                    for j, header in enumerate(all_windows[0]):
                        if  header != "label":
                            for h in range(int(window_len*500)):
                                headers.append(header)

                    nb_features = int(int(window_len*500)*nb_sensors)


                #-------------------Write output file-------------------#


                if act == 1:

                    if feature_or_data == "Features":
                        end_name  = "AB" + candidate[c] + "_SiSt_" +
str(window_len) +"s_" + str(nb_features) + "f_" + sensor
                    else:
                        end_name  = "AB" + candidate[c] + "_SiSt_" +
str(window_len) +"s_data_" + sensor

                    if first_circuit<10:
```

```
                        #out_file_name  = thisFolderParent +
"/Data/Window/"+ feature_or_data+"/SiSt/AB" + candidate[c] + "/" +
end_name

                        out_file_name  = hardrive_path + "/Window/"+
feature_or_data+"/SiSt/AB" + candidate[c] + "/" + end_name

                    file = open(out_file_name +'.csv', 'w')
                    file.truncate()

                    for i in range(len(headers)):
                        file.write(headers[i] + ",")
                    file.write("label\n")

                    if feature_or_data == "Features":
                        for n in range(len(features_only)):
                            for i in range(len(headers)):
                                file.write(str(features_only[n][i]) +
","")
                            file.write(str(labels[n]) + "\n")

                    else:
                        for n in range(len(data_only)):
                            for i in range(len(headers)):
                                file.write(str(data_only[n][i]) + ",")
                            file.write(str(labels[n]) + "\n")

                    file.close()

                elif act == 0:

                    if feature_or_data == "Features":
                        end_name  = "AB" + candidate[c] + "_StSi_" +
str(window_len) +"s_" + str(nb_features) + "f_" + sensor
                    else:
                        end_name  = "AB" + candidate[c] + "_StSi_" +
str(window_len) +"s_data_" + sensor

                    if first_circuit<10:

                        #out_file_name  = thisFolderParent +
"/Data/Window/"+ feature_or_data+"/StSi/AB" + candidate[c] + "/" +
end_name

                        out_file_name  = hardrive_path + "/Window/"+
feature_or_data+"/StSi/AB" + candidate[c] + "/" + end_name

                    file = open(out_file_name +'.csv', 'w')

                    file = open(out_file_name +'.csv', 'w')
                    file.truncate()

                    for i in range(len(headers)):
                        file.write(headers[i] + ",")
                    file.write("label\n")
```

```
                    if feature_or_data == "Features":
                        for n in range(len(features_only)):
                            for i in range(len(headers)):
                                file.write(str(features_only[n][i]) +
","")
                            file.write(str(labels[n]) + "\n")

                    else:
                        for n in range(len(data_only)):
                            for i in range(len(headers)):
                                file.write(str(data_only[n][i]) + ",")
                            file.write(str(labels[n]) + "\n")

                    file.close()
```

8.4 – train.py

train.py

```
import csv
import os
import numpy as np
import matplotlib.pyplot as plt

# For PCA
from sklearn.decomposition import PCA
from sklearn.preprocessing import MinMaxScaler

# For SVM
from sklearn.model_selection import train_test_split
from sklearn.svm import SVC
from sklearn import metrics
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
import pickle

# For MLP
import tensorflow as tf
from keras.models import Sequential
from keras.layers import Dense, Input
from keras.optimizers import Adam

import numpy as np
from sklearn.model_selection import RandomizedSearchCV
from scikeras.wrappers import KerasClassifier


# For CNN
from keras.layers import Conv1D, MaxPooling1D, Flatten, Dense, Dropout
```

```
# For KNN
from sklearn.neighbors import KNeighborsClassifier

# For RF
from sklearn.ensemble import RandomForestClassifier


# For cross validation
from sklearn.model_selection import KFold

# General
from keras.utils import to_categorical
import time
from joblib import dump, load
import random



# Save the trained model to a file
def save_model(model, filename):
    with open(filename, 'wb') as file:
        pickle.dump(model, file)

# Load a trained model from a file
def load_model(filename):
    with open(filename, 'rb') as file:
        return pickle.load(file)



def readData(filename):

    data = []
    labels = []

    #Open .csv file for read
    with open(filename +'.csv', 'r') as csvfile:
        reader = csv.reader(csvfile)

        # Create objects for each header
        # Assign the name of each header to the objects
        for row in reader:
            for i in range(len(row)):
                if row[i] == "label":
                    label_col = i
                    break
            break

        # Assign the value of each column to a different object
        for r,row in enumerate(reader):
            data.append([])
            for i in range(label_col+1):
                if row[i] != "":
                    if i == label_col :
```

```
                        labels.append(int(row[i]))
                    else:
                        data[r].append(float(row[i]))

    return data, labels


def reLabel(label, action):
    new_label = 0
    if action == 0:
        if label == 1:
            new_label = 0
        elif label == 2:
            new_label = 1
        elif label == 211:
            new_label = 2
        elif label == 212:
            new_label = 3
        else:
            new_label = label
    else:
        if label == 1:
            new_label = 0
        elif label == 2:
            new_label = 1
        elif label == 121:
            new_label = 2
        elif label == 122:
            new_label = 3
        elif label == 211:
            new_label = 4
        elif label == 212:
            new_label = 5
        else:
            new_label = label

    return new_label



# Function to create the MLP model
def create_mlp_model(len_input, hidden_nodes, activation):
    model = Sequential()
    model.add(Input(len_input))
    model.add(Dense(hidden_nodes, activation=activation))
    model.add(Dense(nb_class, activation = 'softmax'))

    model.compile(loss='categorical_crossentropy',
optimizer=Adam(learning_rate=0.01), metrics=['accuracy'])


    return model
```

```
#-------------------Main-------------------#


action =  0  # 0 for StSi, 1 for SiSt, 2 for both
window_len = 0.5 #s
window_type = "Data"  #Data, Features
nb_features = 13
sensor = "EMG"  #accelerometer, gyroscope, IMU, EMG, all
method = "SVM"   #SVM, RF, KNN, CNN, DT, LR, NB, MLP
new = 1       # 1 foor new model, 0 for loading an old model
validation_method = "candidate"  # candidate, circuit



if action ==  0:
    a = 0
    b = 1
    activity = "StSi"
    nb_class = 4
elif action == 1:
    a = 1
    b = 2
    activity = "SiSt"
    nb_class = 4
elif action == 2:
    a = 0
    b = 2
    activity = "both"
    nb_class = 6

thisFolderParent = os.getcwd()

candidates = ["156","185", "186", "188", "189", "190", "191", "192",
"193", "194"]

first_candidate = 0
last_candidate = 9

methods = ["KNN","CNN", "MLP"]

for method in methods:
    #all_sensor = ["accelerometer", "gyroscope", "IMU", "EMG", "all"]
    all_sensor = ["all"]


    for sensor in all_sensor:


        all_window_len = [0.05, 0.075, 0.1, 0.15, 0.2, 0.3, 0.5]
        if method == "SVM":
            all_window_len = [0.1, 0.15, 0.2, 0.3, 0.5]

        for window_len in all_window_len:
```

```
            print("method: " + method)
            print("Sensor: " + sensor)
            print("Window length: " + str(window_len) + "s")

            # Initialise the lists
            all_data = []
            all_labels = []


            for c in range(first_candidate, last_candidate+1):
                #print("Candidate " + candidate[c])

                if action == 2:
                    both_data = []
                    both_label = []

                for act in range(a, b):
                    if act ==  1:
                        act_feature = "SiSt"
                    elif act == 0:
                        act_feature = "StSi"

                    hardrive_path = "/Volumes/UnionSine/IMEE
2022/Semester2/FYP/FYPcode"
                    #filename  = hardrive_path + "/Window/"+ window_type
+"/"+ act_feature +"/AB" + candidates[c] + "/AB" + candidates[c] + "_" +
act_feature + "_" + str(window_len) +"s_" + str(nb_features) + "f_" +
sensor
                    if window_type == "Features":
                        filename  = hardrive_path + "/Window/"+
window_type +"/"+ act_feature +"/AB" + candidates[c] + "/AB" +
candidates[c] + "_" + act_feature + "_" + str(window_len) +"s_" +
str(nb_features) + "f_" + sensor
                    else:
                        filename  = hardrive_path + "/Window/"+
window_type +"/StSi/AB" + candidates[c] + "/AB" + candidates[c] + "_" +
act_feature + "_" + str(window_len) +"s_data_" + sensor


                    data, labels = readData(filename)

                    if validation_method == "none":
                        for i in range(len(data)):
                            all_data.append(data[i])

                            new_label = reLabel(labels[i], action)
                            all_labels.append(new_label)
                    else:
                        if action == 2:
                            for i in range(len(data)):
                                both_data.append(data[i])
                                labels[i] = reLabel(labels[i], action)
                                both_label.append(labels[i])
                        else:
```

```
                            all_data.append(data)

                            for i in range(len(labels)):
                                labels[i] = reLabel(labels[i], action)

                            all_labels.append(labels)

                    if action == 2:
                        all_data.append(both_data)
                        all_labels.append(both_label)


                accuracy_max = 0
                accuracy_min = 100
                tot_accuracy = 0
                conf_matrix_tot = np.zeros((nb_class,nb_class))
                time_one_tot = 0
                nb_validations = 1
                for val in range(nb_validations):

                    # Configure KFold cross-validation
                    nb_fold = 10 # The number of folds for cross-validation
                    kf = KFold(n_splits=nb_fold, shuffle=True,
random_state=42)

                    fold = 0
                    for train_index, test_index in kf.split(all_data):
                        fold = fold + 1
                        print("Fold " + str(fold))

                        if validation_method == "none":

                            X_train, X_test = all_data[train_index],
all_data[test_index]
                            Y_train, Y_test = all_labels[train_index],
all_labels[test_index]

                        else:

                            X_train = []
                            X_test = []
                            Y_train = []
                            Y_test = []
                            for candidate in train_index:
                                for w in range(len(all_data[candidate])):
                                    X_train.append(all_data[candidate][w])
                                    Y_train.append(all_labels[candidate][w])

                            for candidate in test_index:
                                for w in range(len(all_data[candidate])):
                                    X_test.append(all_data[candidate][w])
                                    Y_test.append(all_labels[candidate][w])

                        rng = random.Random(42)
                        rng.shuffle(X_train)
```

```
                    rng = random.Random(42)
                    rng.shuffle(Y_train)


                    # Perform PCA
                    pca = PCA(n_components=0.99)       #define the number
of components
                    pca.fit(X_train)                #Find the componet for
our data
                    X_train_pca = pca.transform(X_train)
                    X_test_pca = pca.transform(X_test)

                    if method == "CNN":
                        if X_train_pca.shape[1] < 4:
                            pca = PCA(n_components=4)       #define the
number of components
                            pca.fit(X_train)               #Find the
componet for our data
                            X_train_pca = pca.transform(X_train)
                            X_test_pca = pca.transform(X_test)

                    #----------------------plot PCA variance ------------
----------#
                    # Determine amount of variance explained by
components
                    #print("Total Variance Explained: ",
np.sum(pca.explained_variance_ratio_))

                    """
                    # Plot the explained variance
                    plt.plot(pca.explained_variance_ratio_)
                    plt.title('Variance Explained by Extracted
Componenents')
                    plt.ylabel('Variance')
                    plt.xlabel('Principal Components')
                    plt.show()
                    """

                    # Normalise the data sets
                    min_max_scaler = MinMaxScaler()
                    min_max_scaler.fit(X_train_pca)    #Normalise the
components
                    X_train_pca_norm =
min_max_scaler.transform(X_train_pca)
                    X_test_pca_norm =
min_max_scaler.transform(X_test_pca)



                    print("X_train length: ", len(X_train))

                    if method == "SVM":

                        """
```

```
                        # Create a pipeline to train the SVM model using
the RBF kernel
                        svm_pipeline = Pipeline([
                            ('svm', SVC(kernel='rbf', C=1, gamma='scale',
cache_size=1000, probability=True))
                        ])

                        # Perform grid search for hyperparameter tuning
                        param_grid = {
                            'svm__C': [0.1, 1, 10],
                            'svm__gamma': ['scale', 'auto', 0.1, 1, 10]
                        }

                        grid_search = GridSearchCV(svm_pipeline,
param_grid=param_grid, cv=3, n_jobs=-1, verbose=3)
                        grid_search.fit(X_train_pca_norm, Y_train)



                        # Train the SVM model with the best parameters
                        model = grid_search.best_estimator_

                        """

                        # Create the SVM model
                        model = SVC(kernel='rbf', C=1, gamma='scale',
cache_size=1000, probability=True)

                        # Train the model
                        model.fit(X_train_pca_norm, Y_train)



                    elif method == "MLP":

                        X_train_list = []
                        for i in range(len(X_train_pca_norm)):

X_train_list.append(X_train_pca_norm[i].tolist())


                        Y_train = tf.keras.utils.to_categorical(Y_train,
nb_class)
                        Y_train = Y_train.astype(int)
                        Y_test = tf.keras.utils.to_categorical(Y_test,
nb_class)
                        Y_test = Y_test.astype(int)
                        Y_train = Y_train.tolist()
                        Y_test = Y_test.tolist()

                        #model=create_mlp_model(len(X_train_list[0]), 16,
'relu')

                        model = Sequential()    #Define the type of model
```

```
                        model.add(Input(shape = len(X_train_list[0])))
# Input layer
                        model.add(Dense(units = 32, activation = 'relu'))
# Hidden layer
                        model.add(Dense(units = nb_class, activation =
'softmax'))      # Output layer
                        model.compile(loss='categorical_crossentropy',
optimizer=Adam(learning_rate=0.01), metrics=['accuracy'])        # Create
the model
                        model.fit(X_train_list, Y_train, epochs = 10,
batch_size = 15)      # Train the model




                    elif method == "CNN":

                        Y_train = tf.keras.utils.to_categorical(Y_train,
nb_class)
                        Y_train = Y_train.astype(int)
                        Y_test = tf.keras.utils.to_categorical(Y_test,
nb_class)
                        Y_test = Y_test.astype(int)

                        # Reshape the input for the CNN (batch_size,
num_features, 1)
                        X_train_pca_norm =
X_train_pca_norm.reshape(X_train_pca_norm.shape[0],
X_train_pca_norm.shape[1], 1)
                        X_test_pca_norm =
X_test_pca_norm.reshape(X_test_pca_norm.shape[0],
X_test_pca_norm.shape[1], 1)


                        # Reshape the input for the CNN (batch_size,
window_size, num_features)
                        test = X_train_pca_norm.shape[0]
                        test1 = X_train_pca_norm.shape[1]

                        input_shape = ( X_train_pca_norm.shape[1],
X_train_pca_norm.shape[2])

                        # Create a CNN model
                        model = Sequential()       #Define the type of
model
                        model.add(Conv1D(filters=64, kernel_size=3,
activation='relu', input_shape=input_shape))    # Input layer
                        model.add(MaxPooling1D(pool_size=2))        #
Pooling layer
                        model.add(Flatten())        # Flatten the output
of the convolutional layer
                        model.add(Dense(units = 32, activation='relu'))
                        model.add(Dropout(0.5))
                        model.add(Dense(nb_class, activation='softmax'))
```

```
model.compile(optimizer=Adam(learning_rate=0.001),
loss='categorical_crossentropy', metrics=['accuracy'])    # Compile the
model
                        model.fit(X_train_pca_norm, Y_train, epochs=10,
batch_size=32, verbose=1)     # Train the model



                    elif method == "KNN":

                        # Create the KNN model
                        model = KNeighborsClassifier(n_neighbors=200 ,
metric='euclidean')

                        # Train the model
                        model.fit(X_train_pca_norm, Y_train)



                    elif method == "RF":

                        # Create the Random Forest model
                        model = RandomForestClassifier(n_estimators=200,
max_depth=40, min_samples_split=5, min_samples_leaf=1, random_state=42)

                        # Train the model
                        model.fit(X_train_pca_norm, Y_train)



                    start_time = time.time()                        # Start
the timer
                    Y_predict = model.predict(X_test_pca_norm)    #
Predict classes
                    time_all = time.time() - start_time             # Stop
the timer
                    time_one = time_all/len(X_test_pca_norm)      # time
for one prediction


                    if method == "CNN" or method == "MLP":

                        # Convert the softmax probabilities into one-hot
encoded vectors
                        y_pred = (Y_predict ==
Y_predict.max(axis=1)[:,None]).astype(int)
                        Y_pred_dec = []
                        Y_test_dec = []
                        for j in range(len(y_pred)):
                            Y_pred_dec.append(np.argmax(y_pred[j]))
                            Y_test_dec.append(np.argmax(Y_test[j]))
                        Y_predict = Y_pred_dec
```

```
                        Y_test = Y_test_dec


                    # Evaluate the model
                    accuracy = metrics.accuracy_score(Y_test, Y_predict)
                    conf_matrix = metrics.confusion_matrix(Y_test,
Y_predict)
                    report = metrics.classification_report(Y_test,
Y_predict)


                    print("test candiudate: ", test_index)
                    print("Accuracy = {}".format(np.round(accuracy,4)))
                    #print("Confusion Matrix:")
                    #print(conf_matrix)
                    #print("Classification Report:")
                    #print(report)




                    tot_accuracy = tot_accuracy + accuracy
                    conf_matrix_tot = conf_matrix_tot + conf_matrix
                    time_one_tot = time_one_tot + time_one


                    if accuracy < accuracy_min:
                        accuracy_min = accuracy
                        worst_test_index = test_index


                    if accuracy > accuracy_max:

                        accuracy_max = accuracy
                        best_time = time_one
                        conf_matrix_save = conf_matrix
                        report_save = report
                        best_test_index = test_index

                        X_test_save = X_test_pca_norm
                        Y_test_save = Y_test


                        if window_type == "Features":
                            model_name = activity + "_" + method + "_" +
sensor + "_" + str(nb_features) + "f_"+ str(window_len) + "s"
                        else:
                            model_name = activity + "_" + method + "_" +
sensor + "_data_"+ str(window_len) + "s"

                        window_folder = str(window_len).replace(".","_")
```

```
                        test_features_name = thisFolderParent +
"/Models/" + method + "/" + window_folder + "/" + window_type +
"/test_features_" + model_name + ".npy"
                        np.save(test_features_name, X_test_save)

                        test_labels_name = thisFolderParent + "/Models/"
+ method  + "/" + window_folder + "/" + window_type + "/test_labels_" +
model_name + ".npy"
                        np.save(test_labels_name, Y_test_save)

                        # Save the trained PCA model
                        pca_model_name = thisFolderParent + "/Models/" +
method  + "/" + window_folder + "/" + window_type + "/PCA_" + model_name
+ ".joblib"
                        dump(pca, pca_model_name)

                        # Save the trained normalization model
                        norm_model_name = thisFolderParent + "/Models/" +
method  + "/" + window_folder + "/" + window_type + "/Norm_" + model_name
+ ".joblib"
                        dump(min_max_scaler, norm_model_name)

                        model_save = thisFolderParent + "/Models/" +
method + "/" + window_folder + "/" + window_type + "/" + method + "_" +
model_name + "f_" + sensor

                        if method == "SVM" or method == "KNN" or method
== "RF":
                            # Save the model
                            save_model(model, model_save)

                        else:
                            # Save the model
                            model.save(model_save + '.h5')



            avg_accuracy = tot_accuracy / (nb_fold * nb_validations)

            avg_time_one = time_one_tot / (nb_fold * nb_validations)

            # Normalize the confusion matrix by row
            row_sums = conf_matrix_tot.sum(axis=1, keepdims=True)
            normalized_conf_matrix = conf_matrix_tot / row_sums

            # Convert to percentage
            percentage_conf_matrix = normalized_conf_matrix * 100


            if window_type == "Features":
                model_name = activity + "_" + str(window_len) +"s_" +
str(nb_features) + "f_" + sensor
            else:
                model_name = activity + "_" + str(window_len) +"s_" +
"data_" + sensor
```

```
            result_file = thisFolderParent + "/Models/" + method + "/" +
window_folder + "/" + window_type + "/result_" + model_name + ".csv"


            with open(result_file, 'w', newline='') as file:
                file.truncate()

                file.write("Average Accuracy\n")
                file.write(str(np.round(avg_accuracy*
100,4)).replace(".",",")+ "\n")
                file.write("Average Time\n")
                file.write(str(format(np.round(avg_time_one*
1000,10))).replace(".",",")+ "\n")
                file.write("Total Confusion Matrix:\n")
                for row in conf_matrix_tot:
                    file.write(str(row) + "\n")
                file.write("Total Confusion Matrix in percentage:\n")
                for row in percentage_conf_matrix:
                    file.write(str(row) + "\n")

                file.write("Best Accuracy\n")
                file.write(str(format(np.round(accuracy_max *
100,4))).replace(".",",")+ "\n")
                file.write("Best test index\n")
                file.write(str(*best_test_index)+ "\n")
                file.write("Best time\n")
                file.write(str(format(np.round(best_time*
1000,10))).replace(".",",")+ "\n")
                file.write("Worst Accuracy\n")
                file.write(str(format(np.round(accuracy_min*
100,4))).replace(".",",")+ "\n")
                file.write("Worst test index\n")
                file.write(str(*worst_test_index)+ "\n")
                file.write("Best Confusion Matrix:\n")
                for row in conf_matrix_save:
                    file.write(str(row) + "\n")



            print("\n FINAL: \n\n")
            print("Best Confusion Matrix:")
            print(conf_matrix_save)
            print("Best Classification Report:")
            print(report_save)
            print("Best Accuracy = {}".format(np.round(accuracy_max,4)))
            print("Average Accuracy =
{}".format(np.round(avg_accuracy,4)))
            print("Best time = {}".format(np.round(best_time,10)))
            print("Best test index = {}".format(*best_test_index))

            print("Worst Accuracy = {}".format(np.round(accuracy_min,4)))
            print("Worst test index = {}".format(*worst_test_index))
```