A SYSTEM FOR MONITORING GYM EQUIPMENTS WITH MBIENTLAB METAMOTION SENSORS

by

Zhiyun Lu

Signature Work Product, in partial fulfillment of the
Duke Kunshan University Undergraduate Degree Program

March 23, 2023

Signature Work Program
Duke Kunshan University

APPROVALS

Mentor: Kai Huang, Division of Natural and Applied Sciences

Marcia B. France, Dean of Undergraduate Studies

CONTENTS

ABSTRACT

In this project, we built a system for monitoring the fix-resistance machines in the school gym using MetaMotion sensors from Mbientlab following frameworks and practices from cloud manufacturing. We end up with three different systems relying on either FTP or InfluxDB. The system is capable of monitoring if the gym equipment is occupied by checking the moving standard deviation of the acceleration reading of the accelerometer. The acceleration from using the equipment during workout sessions appears in pulses and is at least detectable compared to background noise. The moving standard deviation greatly smoothes the result and creates recognizable traits for identifying periods in which the gym equipment is being used.

在这个项目中，我们遵循云制造的框架, 使用来自 Mbientlab 的 MetaMotion 传感器构建用于监控学校体育馆中的固定器械的系统。我们最终得到了三套不同的方案，分别依赖于 FTP 或 InfluxDB。该系统能够通过检查加速度计的加速度读数的移动标准偏差来监测体育馆设备是否被人占用。访客在锻炼过程中使用设备产生的加速度呈脉冲状，且与背景噪音有明显区分。通过移动标准差处理数据，其曲线被极大地平滑化了，产生了易于识别体育馆设备是否正在使用的特征。

Acknowledgments

We would like to thank SRS for providing funding for the project. We would like to thank DKU Collective Dynamic Labs for providing the equipment to make the project possible. We would like to thank Prof Kai Huang specially for providing guidance on this project.

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

In this project, we created a system for monitoring the use of pieces of equipment in the DKU gym using sensors from Mbientlab based on the process and principles of cloud manufacturing. On the industry level, the project adds to existing research and application of cloud manufacturing (CMfg).

In a typical cloud manufacturing process, five layers are involved: resource, perception, network, service, and application [20]. The resource layer encompasses various manufacturing hardware, computational resources, intellectual resources, and other necessary components [20]. The perception layer is responsible for sensing and providing information about the resources in the resource layer, consisting of various sensing devices and adapters [20]. The network layer involves the use of various carrier networks to transmit resources or information about them safely and efficiently [20]. The service layer encapsulates the resources into CMfg services for end-users and provides CMfg operational services to operators managing the system's components [20]. Finally, the application layer utilizes these services throughout the product lifecycle[20].

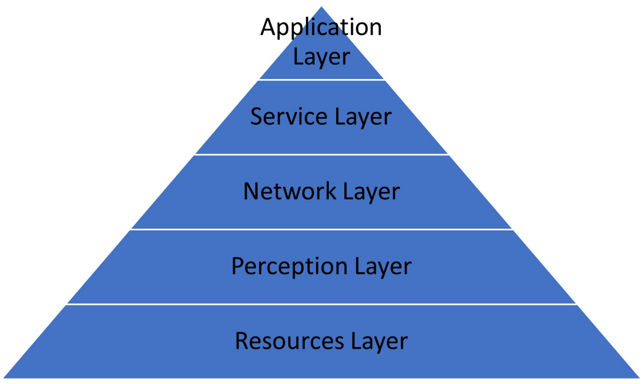

Figure 1.1: Layers of a CMfg system

Our project aims to apply this framework to monitoring the gym equipment in our school by seeing them as the resource layer. Using the Mbientlab sensors and a Raspberry Pi in a manner similar to how they are used in a drop tower experiment from the Collective Dynamics Lab[18] as the perception layer.

According to the latest literary review on CMfg [21], we found that while various architectures have been theorized, there are few practical use cases of CMfg. We believe that our project can contribute to filling that gap.

Chapter 2

MATERIAL AND METHODS

2.1 Hardware Selection and Preparation

The hardware components of the project mainly consist of four parts: Sensors that directly interface with gym equipment and/or the environment, a data hub that controls the sensors and sends the data to the server, a server that stores the data, and the user computer that queries and process the data from the server. The former two hardware components are more specialized therefore we will introduce them in detail.

For the sensors, we started by using Mbientlab's MMC sensors[14] that were available in our lab from the drop tower experiment for granular drag[18]. Later, as Mbientlab's sales of older model stopped, we switched to their latest model then MetaMotionS[15]. This had consequences both positive and negative to the project.

Figure 2.1: The PCB of the MetaMotionC[14]

On the positive side, first and foremost MetaMotionS has a more advanced onboard accelerator and gyroscope. The sensor on board MetaMotionC is BMI160 while MetaMotionS contains a BMI270 onboard[14][15]. The latter has a better acceleration performance and a lower power consumption[2]. Furthermore, BMI160 is no longer available through Bosch while BMI270 still is, meaning MetaMotionS would be easier to repair if the acceleration sensor breaks down.

Other than the onboard acceleration sensor, the new MetaMotionS has a 512MB onboard storage compared to MetaMotionC's much smaller 8MB memory[14][15]. This would be useful

Figure 2.2: MetaMotionC with the outer case[14]

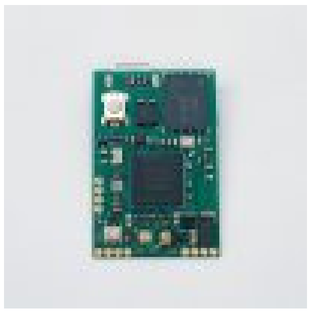

Figure 2.3: The PCB of the MetaMotionS[15]

Figure 2.4: MetaMotionS with the outer case[15]

Feature	**BMI160**	**BMI270**
Package Dimensions	2.5x3.0x0.8mm^3	2.5x3.0x0.8mm^3
Temperature Range	-40...+85°C	-40...+85°C
Supply Voltage VDDIO	1.2...3.6V	1.2...3.6V
Supply Voltage VDD	1.71...3.6V	1.7...3.6V
Current Consumption	850µA	685µA
Sensitivity (Acc.)	±2g: 16384 LSB/g	±2g: 16384 LSB/g
Sensitivity (Gyro.)	±125°/s: 262.4 LSB/°/s	±125°/s: 262.1 LSB/dps
Zero-g/Zero-rate Offset	±40mg (Acc.)	±20mg (Acc.)
Noise Density	180µg/√Hz (Acc.)	160µg/√Hz (Acc.)

Table 2.1: Comparison of BMI160 and BMI270 Sensors. Some specs are missing from either sensor therefore ignored[4][3]

for our project if we chose to utilize the logging mode of the sensors, which we sadly didn't. In the logging mode, the sensor can record data, store it in the on-board memory, and send the data to the datahub at the end of the logging session. We will elaborate on the logging and streaming mode of the sensors when we discuss the design choices for the software. In logging mode, the MetaMotionS is also capable of recording data at 800Hz compared to MetaMotionC's 400Hz. This likely is a result of the update to the memory since the two sensors have the same Bluetooth system.

MetaMotionS is also better supported than MetaMotionC. Firstly, not only is the MetaMotionC no longer produced, but all subsequent products from the company starting from MetaMotionR utilize the R-type case design that is shared by the MetaMotionS and supports the same accessories which are incompatible with MetaMotionC[10]. Secondly, MetaMotionS has its Bluetooth range with and without casing tested and listed in the specs (5m -10m and 10m - 30m respectively)[15] while the MetaMontionC doesn't have these specs listed.

On the negative side, to begin with, the MetaMotionS has its PCB hot glued to the casing and the glue also connects the two halves of the casing, making the casing difficult to open and the PCB almost impossible to remove. On the other hand, MetaMotionC has a casing that is easy to open and its PCB can be easily removed. The process of opening the case of MetaMotionS and removing its PCB might even cause harm to its electronics. This led to our team discarding the plan of designing a new case for mounting the sensors.

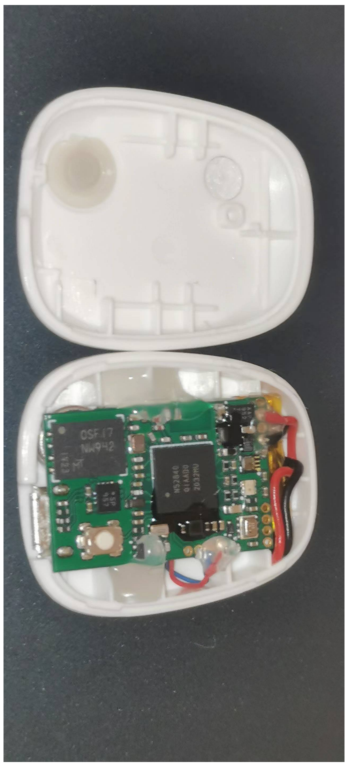

Figure 2.5: Hot glue inside a MetaMotionS, note how there are markings from the lid of the casing on the glue.

Another problem is that MetaMotionS and all models that come after MetaMotionC that have R-type casings have lithium batteries[10]. More specifically, the MetaMotionS is powered by a 100mAh rechargeable lithium-ion 3.7V battery[15] while the MetaMotionC is powered by CR2032 batteries[14]. The latter typically have a nominal voltage of 3V and a 235mAh capacity[16]. While for many applications switching to rechargeable lithium batteries is an upgrade, for our specific case the resulting decrease in battery life and the need for long recharges when the battery runs out is not ideal.

Finally, the MetaMotionS is much more expensive than the MetaMotionC while there aren't too many significant updates. The former sensor retails at $136.99[15] with casing while the latter retails at $86.99 with casing[14]. There is, however, not much updating to the onboard electronics other than the acceleration and gyroscope sensor, the memory, and the battery. The update of the acceleration sensor, while beneficial, isn't particularly important for our project to justify the price rise.

Feature	**MetaMotionC**
Processor	ARM Cortex M4F (nRF52 SOC from Nordic)
Bluetooth Version	Bluetooth Low Energy
Max Raw Data Logging Rate	400Hz
Max Raw Data Streaming Rate	100Hz
Memory	8MB NOR FLASH
Power Source	CR2032 coin-cell battery
Motion Sensors	- 3-axis Accelerometer: BMI160 - 3-axis Gyroscope: BMI160 - 3-axis Magnetometer: BMM150 - 9-axis Sensor Fusion: Bosch
Other Electronics	- Temperature Sensor: BMP280 - Light Sensor: LTR-329ALS - Altimeter/Barometer/Pressure Sensor: BMP280 - LED, GPIOs, push button switch

Table 2.2: MetaMotionC onboard sensors and electronics[14]

Feature	**MetaMotionS**
Processor	ARM Cortex M4F (nRF52 SOC from Nordic)
Bluetooth Version	Bluetooth Low Energy
Max Raw Data Logging Rate	800Hz
Max Raw Data Streaming Rate	100Hz
Memory	512MB NAND FLASH
Power Source	100mAH lithium-ion 3.7V battery
Motion Sensors	- 3-axis Accelerometer: BMI270 - 3-axis Gyroscope: BMI270 - 3-axis Magnetometer: BMM150 - 9-axis Sensor Fusion: Bosch
Other Electronics	- Temperature Sensor: BMP280 - Light Sensor: LTR-329ALS - Altimeter/Barometer/Pressure Sensor: BMP280 - LED, GPIOs, push button switch

Table 2.3: MetaMotionS onboard sensors and electronics[15]

Overall, switching over to the new MetaMotionS doesn't necessarily benefit the system much from the capabilities of the sensor or the price, but is the more reliable choice for future-proofing the system as the company no longer supports MetaMotionC. Luckily, this change doesn't have much effect on the software part as both sensors support the same library and don't require distinction in code.

The data hub is connected to the sensors through Bluetooth. While Mbientlab did once retail a plug-n-play data hub Metahub Gateway with software from the company preinstalled[11], it is no longer available for purchase like the MetaMotionC. Unlike the MetaMotionC, we didn't have spare Metahubs so we started with a Raspberry Pi 4[17] with an Argon ONE v2 third-party casing[1] as the data hub.

Figure 2.6: Metahub Gateway[11]

Figure 2.7: Raspberry Pi 4 Model B[17]

Figure 2.8: Argon ONE v2[1]

The Raspberry Pi was chosen as the data hub for two reasons. Firstly, the official data hub Metahub is a modified Raspberry Pi 3, therefore we believe having a Raspberry Pi as the datahub should achieve similar effects. Second, the official Python API MetaWear by Mbientlab runs on Linux Ubuntu Desktop in their tutorial[13], and the Raspberry Pi 4 was the only model then to support Ubuntu Desktop. Having the GUI for Ubuntu available also greatly simplified the development process. This not only allows for quicker and easier testing, but since a device's IP address changes when connected to DKU wifi, using the GUI is more reliable than SSH into the Raspberry Pi. We can also look up the IP address through the GUI before needing to SSH into the Raspberry Pi.

The third-party casing Argon ONE v2 was used instead of the official casing for three reasons. First, it provides two HDMI ports instead of the Raspberry Pi's single Micro HDMI. HDMI cables are cheaper and more common than Micro HDMI. Secondly, the Argon ONE v2 casing is much sturdier than the official plastic casing, making it more suitable for an environment like the gym. Last but not least, the third-party casing has a better cooling system than the official casing. This helps the Ubuntu Desktop run more smoothly on the Pi as running the Ubuntu Desktop generates a lot of excessive heat.

One problem with installing the software required for the data hub by ourselves is that we require a VPN for installing almost any package required and for updating the Ubuntu system we installed to a stable version. Most commercial VPN software that operates in China lacks support for Linux systems. While the Cisco Anyconnect Client we use for connecting to the Duke VPN has support for Ubuntu, it doesn't support Linux systems running on ARM64 devices like the Raspberry Pi. We used an open-source command line VPN client openconnect to connect to the Duke VPN.

For the rest of the installation, following the tutorial from the official tutorial page[13] gives us decent results. However, another problem that wasn't mentioned by the tutorial was that the dependency of the MetaWear library needs to be manually installed in the order of Warble C, PyWarble, and MetaWear, or else the packages cannot be installed.

2.2 Programming

For the server, first we used the Box as a File Transfer Protocol *FTP* server, later we used InfluxDB OSS to host a time series database on a laptop. We will compare and elaborate in the software section. For the final part of the hardware, I used the same computer for SSH into the Raspberry Pi and later hosted the InfluxDB server. However, accessing the server from remote is possible as long as you have the IP address and the token. I used this method to access the InfluxDB server despite the server being hosted locally to show that accessing it from remote is possible.

Our programming primarily runs on the Raspberry Pi, part of it runs on the computer that queries and processes the data. The latter will be elaborated upon in the results section.

For the programming on the Raspberry Pi, we chose the Python version of the MetaWear APIs. We made this design choice because this project traces its roots back to a SRS project in which we worked as a team and Python was the language that everyone on the team knew. Python was used also because we thought would assist the system's integration between different parts of the system.

The design of the programming went through 3 iterations as we switched from FTP to InfuxDB. We will focus on the first version and cover the changes made in the following versions.

During the initial development, as we chose the Python MetaWear API, several problems arose from choosing this version of the MetaWear API. This version of the API wasn't as well-maintained or well-documented as we thought.

The first problem came when we were experimenting with the official tutorial[13]. When experimenting with disconnecting, the tutorial asked us to use MetaWear.on_disconnect to listen to the event that is the completion of the disconnecting and set it as the callback function to run upon successful disconnect. However, upon experimenting, we discovered that the MetaWear class no longer has an instance variable or event on_disconnect for setting disconnect callback functions, instead, it has a boolean instance variable MetaWear.is_connected indicating whether the device is connected. Note that disconnecting requires some time to complete so MetaWear.is_connected will remain True after calling device. disconnect for a while so newer official examples that use this variable usually either have a sleep before printing the disconnecting message or use a custom event.

Noticing the inaccuracy in the tutorial, we switched to primarily relying on the official example code GitHub repository [9]. The example code there, however, wasn't without its own problems. Some of the example programs don't really run for reasons that can't be solved at the Python level. Most notably, the program for scanning nearby MetaMotion Sensors and connecting to them, scan_connect.py doesn't run. The error comes from functions in mbientlab.warble and according to error messages the MetaWear library has issues interacting with Warble. Therefore for all programs involving the MetaMotion Sensors, one needs to connect to the sensors based on their MAC address instead of automating the connection by scanning for the sensors.

The other source we relied on was the documentation for the C++ APIs[12]. The code in the GitHub repository doesn't cover all the functionalities MetaWear is capable of. We need detailed documentation to learn how to achieve capabilities such as adjusting the sensors' sampling frequency and threshold. However, we can't find a link to such detailed documentation for the Python version on the official site. According to the Python version tutorial[13], the Python APIs are a thin wrapper around the C++ APIs and we should refer to the documents for the C++ APIs.

Some other problems with MetaWear itself might have arisen from the fact that the Python APIs are a thin wrapper around the C++ APIs and the developers of MetaWear Python might have not done the garbage collection correctly. Firstly, every time one uses MetaWear.connect to connect to a MetaMotion sensor, there is a random chance of the program crashing due to a buffer overflow error. This signals a memory leak that can not be resolved at the Python level, and the crash can't be caught as an exception at the Python level.

Another problem is that when using the logging mode of the sensor. The sensor has two modes for recording data, streaming and logging. In streaming mode, the sensor sends data directly to the hub through a callback function whenever a new data point is produced. In logging mode, the sensor stores the data in its memory and sends the data at the end of the logging session. Logging mode permits a higher sampling frequency, has a more stable interval between data points, and doesn't require a stable connection during the session. The problem is that when trying to reset the logger and start a new logging session after downloading leads to crashes with an error message related to the pointer unless you disconnect and reconnect to the sensor.

While logging mode does provide data with better quality and is one of the features that makes use of improvements in the MetaMotionS, we chose streaming mode for four reasons. Firstly, the two previous problems combined with how logging mode works make it difficult to utilize it for long-term monitoring since disconnecting and reconnecting triggers the crash. Secondly, the downloading process of the logging mode takes a while and increases the gaps between sessions. Thirdly, since the acceleration data we want to capture likely won't include signals with a frequency higher than 50Hz since we are not tracking sound, 100Hz would likely be enough for our purpose. Finally, the data hub can't receive the logged data until the end of each session for logging mode. While for the first version, this doesn't matter, streaming mode proved to be superior when we tried to have the data hub upload in real-time in the third version, in which we require the constant flow of data.

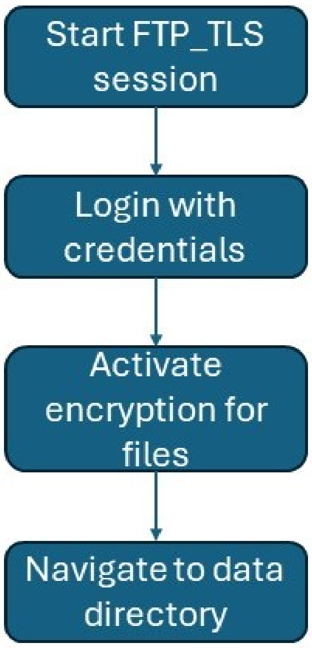

Figure 2.9: Flow chart of start and configure FTPS session, FTPS refers to FTP Safe

For the first version of the program, we were using the Box as an FTP server. We need to activate the FTP capability on the Duke Box site first. We use the Python standard library ftplib to connect to the server. Due to the server's security requirements, we need to start a session that

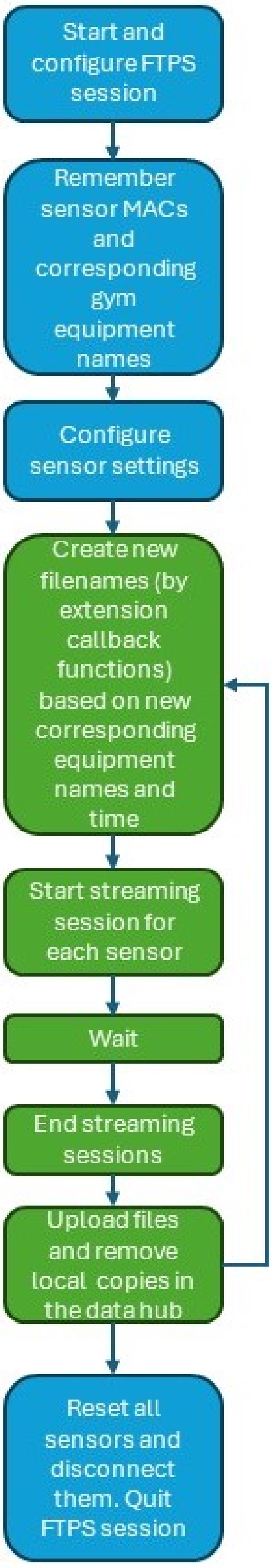
Start and configure FTPS session
Remember sensor MACs and corresponding gym equipment names
Configure sensor settings
Create new filenames (by extension callback functions) based on new corresponding equipment names and time
Start streaming session for each sensor
Wait
End streaming sessions
Upload files and remove local copies in the data hub
Reset all sensors and disconnect them. Quit FTPS session

Figure 2.10: Flow chart of program version 1

is TLS encrypted with the ftplib.FTP_TLS object and call FTP_TLS.prot_p to ensure encryption is applied to not only user credentials but the files as well.

We store the sensors' information and their corresponding gym equipment as a list with the following form : [['d6:3e:9f:34:9e:1a','Alex'],['d3:fd:fb:2c:15:02','Bob']]. The reason why each sensor is associated with a name is so that we can continue the record for one gym equipment even after we have placed a different sensor on it after the old one breaks down.

For the step of configuring the sensors, we are setting the configurations that don't change throughout the working cycles. Namely the Bluetooth configuration parameters, the sampling frequency of the accelerometer, and the range of the accelerometer.

At the start of each working cycle, we create new filenames based on the name of the equipment and the current time. A callback function is also defined for each sensor to write into their corresponding file at the time. The filename consists of three parts: name of the equipment, time in "%Y%m%d-%H%M%S" form, and "acc.data" ending to tell the acceleration data apart from other potential readings we want to include such as the gyroscope data. We included the time in the filename for two reasons. First, the time stamp makes sure each filename is unique. Second, while each data point has its time stamp, having a timestamp for each file makes querying for entries more efficient time-complexity-wise. A resulting filename would look something like "Squat20230601-151551acc.data".

The callback function reads the data being sent back. Each entry contains an epoch in the form of UNIX time with ms precision that can be accessed by data.contents.epoch and a value that can be read by the parse_value function provided by MetaWear. In the case of acceleration, the value is the acceleration along three axes measured in g up to 3 digits after decimal in the form of {x : (-)X.XXX, y : (-)Y.YYY, z : (-)Z.ZZZ}. As a result, a typical we write into a file would look like "1685610994859, x : -0.018, y : -0.024, z : -1.004". The callback function utilizes file handles in append mode.

To start the streaming session for each sensor, we follow a fixed routine of retriveing the corresponding signal object through libmetawear.mbl_mw_acc_get_acceleration_data_signal(), subscribe to the signal through libmetawear.mbl_mw_datasignal_subscribe() with the callback function corresponding to the sensor as one of the arguments. After that, enable acceleration sampling and start the accelerometer. During the session, just use the sleep() function from the Python time library and wait a while. To end the streaming session, we need to undo the steps for starting the session step by step.

The last step in the working cycle involves uploading and removing the files. When we created the filenames for each sensor, we also stored the filenames and file handles. We need to first close the append file handles that were used for writing into the files and then upload and remove the local file with the file name. For uploading, we need to create a new file handle in read-binary mode and close it after uploading. For uploading, we are using the FTP_TLS.storbinary(). In our case, it requires a cmd argument in the form of "STOR SomeFileName" and the new file handle. The removal is done by using os.remove("filename").

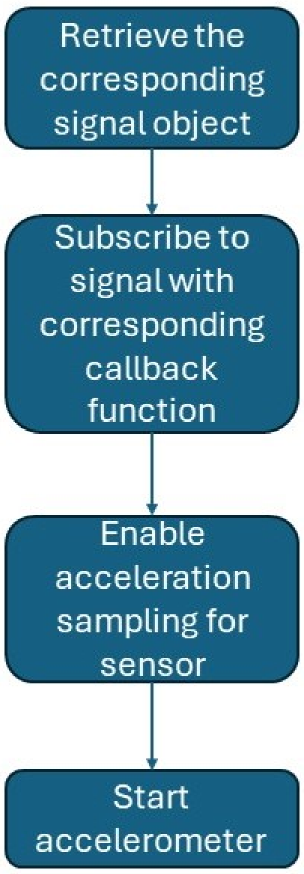
Retrieve the corresponding signal object
Subscribe to signal with corresponding callback function
Enable acceleration sampling for sensor
Start accelerometer

Figure 2.11: Flow chart of starting accelerometer streaming session for one sensor

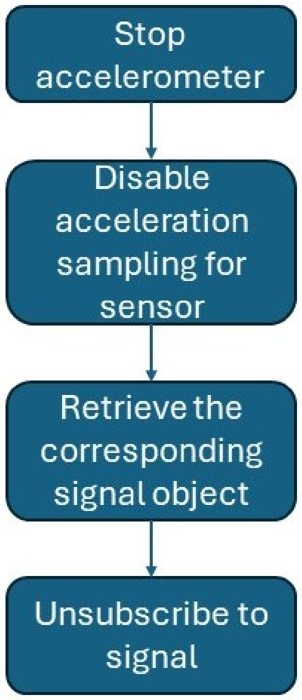
Stop accelerometer
Disable acceleration sampling for sensor
Retrieve the corresponding signal object
Unsubscribe to signal

Figure 2.12: Flow chart of ending accelerometer streaming session for one sensor

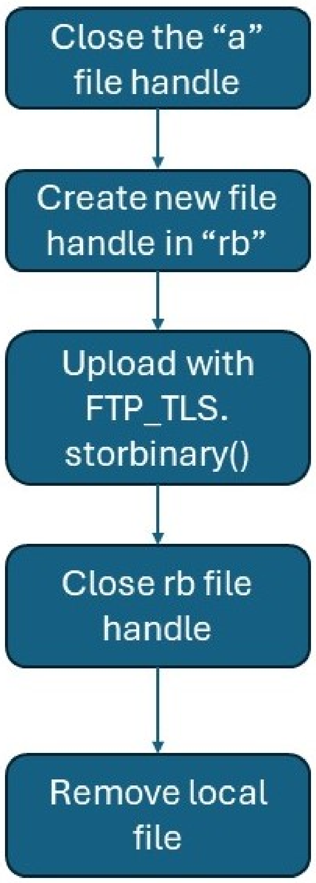
Close the "a" file handle
Create new file handle in "rb"
Upload with FTP_TLS. storbinary()
Close rb file handle
Remove local file

Figure 2.13: Flow chart of uploading and removing local files in one working cycle for each sensor

This ends the working cycle. The steps belonging working cycle are repeated again and again and each step except for wait is applied to every sensor through a loop. The wait step controls the time we monitor for that working cycle and is shared by the sensors. After ending the final working cycle, we disconnect and reset the sensors. Missing the last step due to interruptions caused by the battery running out or bad connections doesn't affect the system. Such interruptions will cause the system to stop running but won't lead to any data loss. Any data that isn't uploaded can be found locally. The system supports connecting to multiple sensors and tests show that without dongles one data hub can handle around 5 sensors.

Starting from version 2, we switched from FTP to a time series database InfluxDB to take its inherent capabilities for querying, processing and visualizing time series data[8]. InfluxDB allows future integration of readings from other kinds of sensors. To enable better testing we established an InfluxDB v2.7 instance on a laptop with InfluxDB OSS. The commercially available cloud servers have v3 client libraries that can't be accessed for free. In reality, since it is difficult to keep the laptop on and unoccupied for prolonged periods with a fixed IP under school WIFI, this change is more of a proof of concept than what we actually used for collecting data.

Version 2 keeps the basic framework of version one, except that when uploading, the files are parsed into InfluxDB line protocols. Given the number of entries we need to upload each time, we need to set up a batch processing API[7] so that the system doesn't attempt to re-establish connection with the time series database for each entry. The batch upload API only works with Observables from the ReactiveX for Python library(RxPy)[19]. The line protocol contains ”measurement” to specify which test run the entries belong to, two tags in ”tags” ”location” and ”equipment” specifying the gym and the piece of gym equipment, ”fields” with content ”x” ”y” ”z” recording the acceleration along the 3 axes and ”time” specifying the UNIX time of the entry.

One of the key capabilities of InfluxDB is to monitor and do simple processing of the data in real time. However, the structure of version 2 can not take advantage of this since it still uploads periodically like version 1. Therefore in version 3, we are trying to make the uploading real time. This proves to be a much more difficult task than expected.

The difficulty comes from integrating 2 different concurrent systems that are different from native Python concurrency. In the MetaWear library, callback functions are converted to FnVoid_VoidP_DataP objects which receive and process data from the sensors concurrently with each other and the main thread. The callback functions cannot produce return values. Using indfluxDB requires working with Observables from RxPy, which has a different way of achieving concurrency[19]. These two concurrency systems are then different from the native Python threads and there is no effective way for managing the thread-like processes from the two systems.

We can not find a real reliable solution to this so we have to proceed with less coordination between the threads. We have the callback functions write to a Queue and define a function that either dequeues the last element of the queue when the queue isn't empty or outputs the final element output before the queue goes empty. For the Observable, we follow the example given by the InfluxDB User Guide on using InfluxDB with IOT[6] and produce periodically while setting the interval smaller than the reciprocal of the combined frequency of the sensors. However because we cannot manage when the threads join or wait for each other, this version suffers from errors related to race condition.

Chapter 3

RESULTS

3.1 Potential Setup

The system that we completed is pretty flexible, but it works best with fixed-resistance machines. Given the sensor's sampling rate, it isn't ideal for capturing the vibrations from gym bikes and rowing machines. While the sensors could potentially work with free weights, we didn't adopt this setup since a protruding sensor from a bell could potentially cause dangers to gym goers or be damaged by curious people. The ideal way of setting up a sensor is by sticking it to the weight of fixed-resistance machines, usually with an adhesive like a double-sided tape. All sensors need to be within 5m -10m of their data hub.

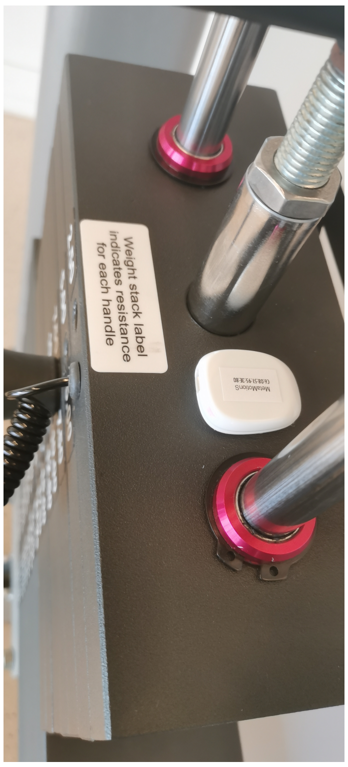

Figure 3.1: Close up of sensor fixed to a fixed-resistance machine

The data hub should be plugged in at nearby power outlets. Sockets are relatively abundant in gyms as they are needed to power certain types of gym equipment like treadmills. One should also provide extra sockets through an extension cord to prevent people from unplugging the data hub to charge their cell phones. The entire setup needs to have signs to prevent people from touching it.

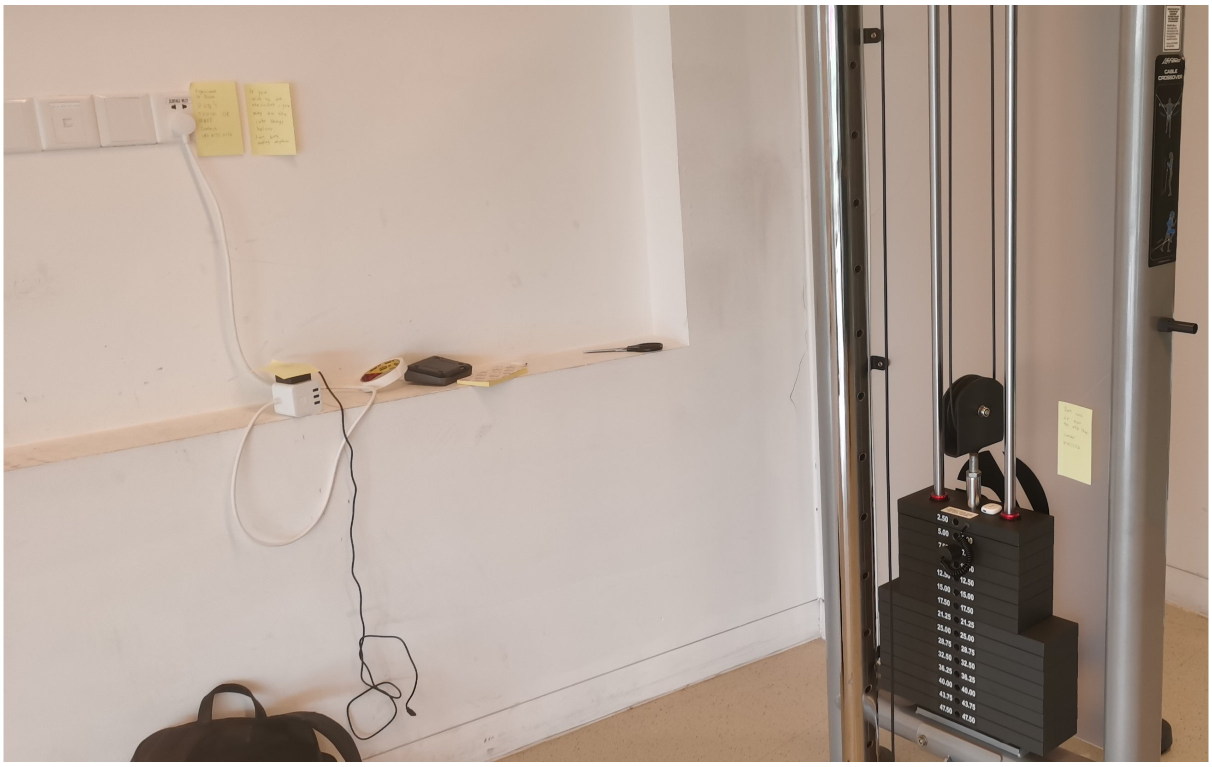

Figure 3.2: An example for setting up the sensor

3.2 Data Retrieval

Depending on the version of the program used for data collection, the data can be queried and processed in different ways. For version 1, one can always download all the files. In the meantime, we have written some useful functions for retrieving files by specified criteria. For example, the most recent file for a sensor. This is thanks to the fact that through FTP_TLS.nlst(), we can retrieve a list of filenames under a directory sorted based on names. This makes it easy to search for the name of the file we need and retrieve it. Once we have the file name, we can either retrieve the content as lines of text or retrieve the content as binary and write it to a local file. Either way, all subsequent processing and graphing happens on the computer that queries the data.

For version 2 and version 3, the data is stored in instances of the time-series database InfluxDB. Unlike version 1, having a copy of the database locally doesn't automatically grant you access to all the entries all at once and data retrieval almost always needs to be done by querying the database with its client APIs. Unlike version 1, thanks to InfluxDB, much data processing and analysis can be done on the server that hosts the database.

3.3 Collected Data and Analysis

For our monitoring sessions, we set up the system in the way described in detail above in 3.1. Since our sensors are fixed on the weights of the fixed-resistance machines, we only need to care about the acceleration on the vertical axis. Figure 3.3 is an hour of reading from the sensors plotted, this hour is selected because it contains readings from someone's workout.

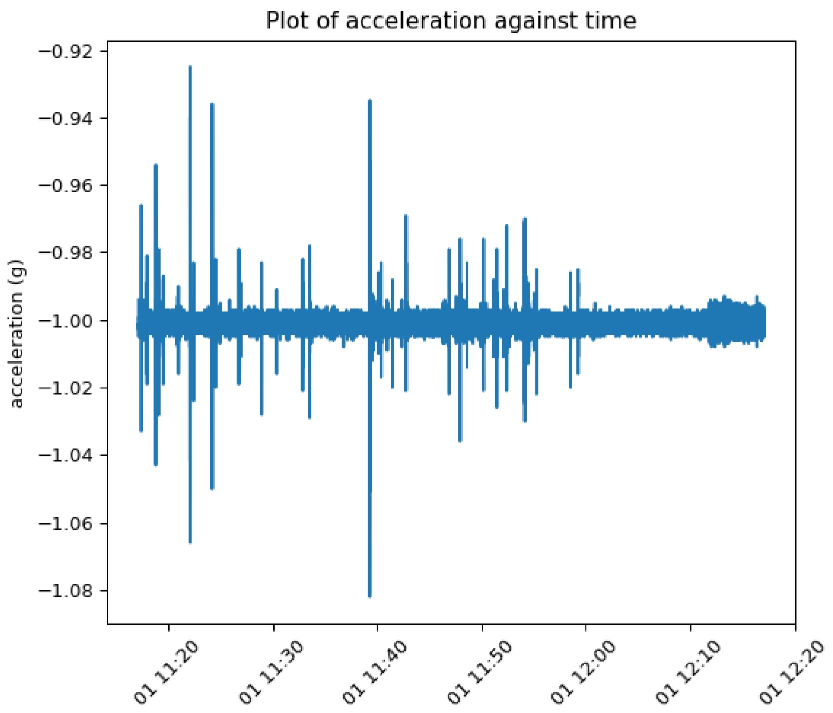

Figure 3.3: Workout Session captured by sensor

As we can see in the figure, at rest the reading fluctuates around -1 g in a noisy fashion. Whenever the weight is moved in a workout, there are pulse-like spikes in both positive and negative directions.

One of the ways to make the data useful is by taking the moving standard deviation and using the result to figure out when the equipment is occupied. The moving standard deviation removes the effect of the orientation of the sensor, smoothes the data, and can be done in real-time by the InfluxDB.

By changing the window size, we can decide how much time without movement means the fixed-resistance machine is vacant. According to a 2009 article, there should be 5 mins of rest between sets during weight training[5]. Therefore, we set our window size as 5 minutes.

As we can see here a problem with analyzing period by period like version 1. The first 5 minutes of each working cycle don't have a meaningful Moving standard deviation. We need to either painstakingly prepend data from the previous working cycle or crop it out.

After that, we can determine whether or not the fixed-resistance machine is vacant using either

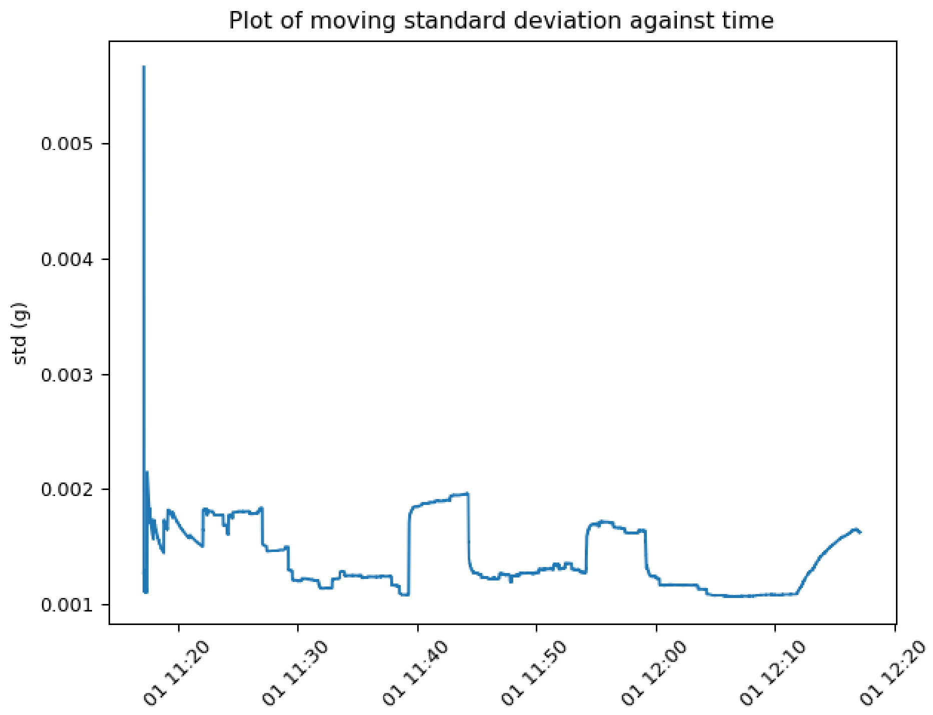

Figure 3.4: Moving standard deviation of Figure 3.3 with window size 5 mins

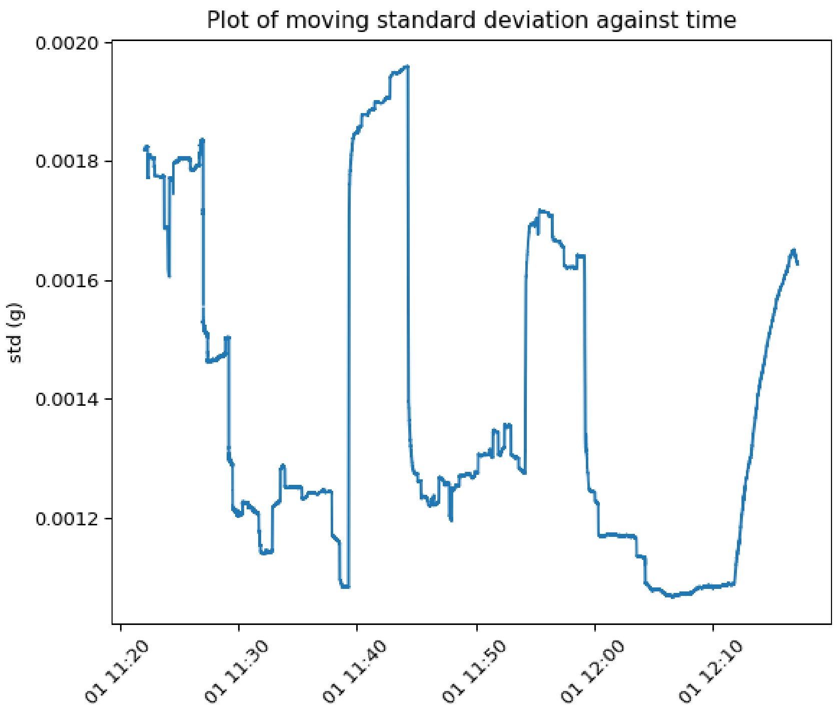

Figure 3.5: Figure 3.4 cropped

an adaptive or fixed threshold.

This cropping process isn’t required for version 2 or version 3 as they are based on InfluxDB and it can do moving standard deviation continuously for incoming data. It can also apply the threshold in real time.

Chapter 4

DISCUSSION

Looking at the system we built, we can see that it lacks the application layer. This is the limitation of building it for monitoring the school gym. Another bottleneck is that the system can not really achieve problem-free long-term monitoring due to the incompatibility between certain systems we decided to use.

In our original vision of the project, we sought to incorporate machine learning and determine whether the piece of gym equipment is occupied by semi-supervised training by cutting the record into periods and with labels coming from periods we know for sure it is occupied. However, as we progress and start using tools such as InfluxDB, it turns out that simple traditional methods such as moving standard deviation are much more suitable for this relatively simple task.

In the future, the project can improve in several ways. Firstly, as we already have a good framework for monitoring sensors based on InfluxDB, we can integrate other projects underway such as a project to measure the indoor AQI. Secondly, the Mbientlab sensors can be replaced or complemented with other sensors like a radar that is being worked on by other teams from our lab. Last but not least, we can work on resolving the issues with monitoring in real-time by migrating to other languages such as C++.

Chapter 5

CONCLUSIONS

In this project, we have built a system for monitoring the equipment in the gym with MetaMotion sensors from Mbientlab. We have built three versions each with their strength and flaw. This first version is based on FTP, runs most stably, and enables access to all history. The second version utilizes InfluxDB. This provides easier analysis and monitoring compared to the first version but harder to access historical records. The third version builds upon that and attempts to achieve truly real-time uploading and analysis. It ends up sacrificing the stability of the system.

Looking into the processing of the data, we can see that the workout sessions are characterized by short bursts of acceleration that can be easily missed by the sensor if its sampling frequency is low, suggesting that a system that directly measures displacement might be a more effective practice

REFERENCES

[1] Argon40. *Argon ONE V2 Case for Raspberry Pi 4*. https://argon40.com/products/argon-one-v2-case-for-raspberry-pi-4. Accessed: Mar. 05, 2024.

[2] *BMI270 vs BMI160*. https://community.bosch-sensortec.com/t5/MEMS-sensors-forum/BMI270-vs-BMI160/td-p/8619. Accessed: Mar. 05, 2024. community.bosch-sensortec.com, July 2019.

[3] Bosch. *IMU:BMI160*. https://www.bosch-sensortec.com/products/motion-sensors/imus/bmi160/. Accessed: Mar. 05, 2024.

[4] Bosch. *IMU:BMI270*. https://www.bosch-sensortec.com/products/motion-sensors/imus/bmi270/. Accessed: Mar. 05, 2024.

[5] Belmiro Freitas De Salles et al. "Rest Interval between Sets in Strength Training:" en. In: *Sports Medicine* 39.9 (Sept. 2009), pp. 765–777. ISSN: 0112-1642. DOI: 10.2165/11315230-000000000-00000.

[6] InfluxDB. *Efficiency write data from IOT sensor*. https://influxdb-client.readthedocs.io/en/latest/usage.html#efficiency-write-data-from-iot-sensor. Accessed: Mar. 13, 2024.

[7] InfluxDB. *How to efficiently import large dataset?* https://influxdb-client.readthedocs.io/en/latest/usage.html#how-to-efficiently-import-large-dataset. Accessed: Mar. 13, 2024.

[8] InfluxDB. *InfluxDB OSS v2*. https://docs.influxdata.com/influxdb/v2/. Accessed: Mar. 13, 2024.

[9] Laura Kassovic and Matt Baker. *MetaWear SDK for Python by MBIENTLAB*. https://github.com/mbientlab/MetaWear-SDK-Python. Accessed: Mar. 11, 2024.

[10] Mbientlab. *Hidden Page for Past Mbientlab Products*. https://mbientlab.com/product-category/uncategorized/. Accessed: Mar. 05, 2024.

[11] Mbientlab. *Metahub Gateway*. https://mbientlab.com/store/metahub/. Accessed: Mar. 05, 2024.

[12] Mbientlab. *MetaWear C++ APIs*. https://mbientlab.com/documents/metawear/cpp/latest/. Accessed: Mar. 13, 2024.

[13] Mbientlab. *MetaWear Python APIs*. https://mbientlab.com/tutorials/PyLinux.html. Accessed: Mar. 05, 2024.

[14] Mbientlab. *MMC –MetaMotionC –MBIENTLAB*. https://mbientlab.com/store/metamotionc/. Accessed: Mar. 04, 2024.

[15] Mbientlab. *MMS –MetaMotionS –MBIENTLAB*. https://mbientlab.com/store/metamotions/. Accessed: Mar. 05, 2024.

[16] Microbattery. *Everything You Need To Know About The CR2032 Battery*. https://www.microbattery.com/blog/post/battery-bios:-everything-you-need-to-know-about-the-cr2032-battery/. Accessed: Mar. 05, 2024. www.microbattery.com.

[17] Raspberry Pi. *Raspberry Pi 4 Model-B*. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/. Accessed: Mar. 05, 2024.

[18] Tivadar Pongó and Jinchen Zhao. *Drop tower / projectile impact experiments*. https://sites.duke.edu/dukecdl_internal/projects-soft-matter/drop-tower-experiment/. DKU CDL Internal Site, May 2023.

[19] ReactiveX. *Documents for ReactiveX for Python*. https://rxpy.readthedocs.io/en/latest/index.html. Accessed: Mar. 13, 2024.

[20] Fei Tao et al. "IoT-Based Intelligent Perception and Access of Manufacturing Resource Toward Cloud Manufacturing". In: *IEEE Transactions on Industrial Informatics* 10.2 (May 2014), pp. 1547–1557. ISSN: 1551-3203, 1941-0050. DOI: 10.1109/TII.2014.2306397.

[21] "Theory, supporting technology and application analysis of cloud manufacturing: a systematic and comprehensive literature review". In: 120 (). ISSN: 0263-5577. DOI: 10.1108/IMDS-10-2019-0570.

Appendix A

ADDITIONAL MATERIAL

This template can be viewed on Overleaf at https://www.overleaf.com/read/hxjcgtkhjqcd. If you have an Overleaf account (either free or paid) you can copy this template to start a new Overleaf project. If you do not want an Overleaf account you can install TeX on your computer and download the template files from Overleaf.

Here are all three versions and the data from version 1. I can not migrate the database onto Box, so we don't have data from versions 2 and 3 here. https://duke.box.com/s/66vr7c9sy12kc zw76or803n9gh8i2zmi