Portfolio Summary

Brandon Rodriguez
School of Computing and Augmented Intelligence
Arizona State University
Tempe, Arizona
blrodri4@asu.edu

In order to complete my graduate program and receive a Master of Computer Science degree, my program requires a culminating experience project in the form of a portfolio. The portfolio contains reports from two classes regarding the projects in the class, as long as the classes were at least 35% project based. In addition, the professors could add requirements for what the report must consist of or if additional work must be done to include the course in the portfolio. Professor Yann-Hang Lee did not have any requirements; thus, I was able to use any project from the course for my portfolio. Professor Ayan Banerjee required that all projects in the course be used in the portfolio, as the projects were dependent on the projects that came before.

The two classes I chose to include were CSE 522 Real-Time Embedded Systems and CSE 572 Data Mining, which I both took in the first full semester of my master's program in the Spring of 2022. The Real-Time Embedded systems course involved learning how to implement software on an embedded device equipped with the Zephyr real-time operating system and utilize this real-time operating system to also handle scheduling and other real-time events. The data mining course encompassed advanced data mining techniques such as preprocessing, supervised learning approaches, and unsupervised learning approaches.

I had mostly taken courses pertaining to more standard software development, such server-side API development, website development, basic database management, as well as, of course, computer science foundational concepts. In addition, I have completed internships with Georgia Tech and Amazon that reinforced that learning. Therefore, I felt these courses had more to offer in terms of expanding my knowledge; these two courses allowed me to step out of my comfort zone into the fields of embedded software and data analysis.

For the RTOS course, the project included in the report is the most involved and largest project of the class which was developing an application with three shell commands on an NXP MIMXRT1050-EVKB board that controls two external devices: a single-bulb RGB LED and a MAX7219 LED display, which is essentially a small 8x8 panel with 64 red circular LEDs. The portfolio report for this project describes all the different activities that went into the development of this embedded application and all that I was able to learn from this process.

The Data Mining report in the portfolio outlines the three projects that were undertaken as part of the course. Collectively, the three projects operated on same type of data set that captured serial glucose-related data of a person wearing a monitoring device and other data tracked manually such as when meals were eaten. The first project was a straightforward, but fundamental preprocessing task, and classification and clustering models were derived from the data sets and tested/validated as a part of the remaining two projects. The portfolio report includes a description of my solution of all these projects, the results (since these projects were graded based of the margin of error of the results the model produced when fed a set of data), and what I learned from the projects and how I would improve upon them based on the knowledge and skills I have gained since.

Compared to the training I have undergone both academically and professionally, these projects have challenged me more in terms of dealing with real-world issues, such as there not being a "correct" solution in the Data Mining project series or dealing with compatibility issues between systems in the Real-time Embedded Systems main project. The class was informed by the professor in the Data Mining course that we were not expected to arrive at any particular "solution" as one did not technically exist; but we instead instructed to get as close as possible using the techniques learned in class in whatever variation we so choose. The Real-time Embedded Systems course dealt with a variety of different systems that clearly were not part of the course material to be explained to the class. Thus, the class was instructed to rely on manuals of different systems, ie. the MAX7219 LED display, and the documentation/code base of the Zephyr operating system, rather than the course materials that would only get you so far. These circumstances encouraged me to take complete ownership in these projects as there was not one single path the projects were designed to be guiding the participants of the class towards in either course.

In the topic of real-world problems, it is often that requirements or deadlines change, and you are forced to adjust, which is exactly what happened in the Data Mining course. Due to higher up scheduling issues, the course was shortened significantly and was moved entirely online. By far the most challenging part was the windows of time to perform each project shrinking in response, since as mentioned before the projects were dependent on the previous projects.

Another element that encouraged me to take full ownership of these projects other than the loose reigns was the individual nature of these projects. Not to neglect the importance of collaboration, as it is usually essential and encouraged in the workplace rather than being penalized as in most academic settings; however, responsibility is usually centralized. In the work environments I have been exposed to in the beginnings of my career, projects are assigned to individual engineers, which is a fairly standard practice in work environments that have Agile approach to development. Of course, guidance and enlisting others for assistance is common, but the success or failure of the project lies squarely on the shoulders of the assignee of the project.

A part of my graduate school application for this program required a statement on why I wanted to pursue this graduate degree. I stated that I wanted to further prepare myself before being fully immersed in the workforce. I believe these two courses provided great growth opportunities that were representative of the final stage of my academic career, and my portfolio highlights this.

Real-time Embedded Systems Report

Brandon Rodriguez
School of Computing and Augmented Intelligence
Arizona State University
Tempe, Arizona
blrodri4@asu.edu

ABSTRACT

In this report, I describe the background and process for implementing the second project of the CSE 522 Real-time Embedded system course, and the results, namely insight knowledge and skills gained, from completing this project successfully.

1. INTRODUCTION

In Spring 2022, I took the CSE 522 Real-time Embedded Systems course instructed by Professor Yann-Hang Lee. The course required the use of an NXP MIMXRT1050-EVKB board as the embedded system with Zephyr v2.6.0 as the real-time operating system (RTOS) on the board. The two topics in the title, real-time and embedded, were central to all the projects. However, each project had its focus on one more than the other. For example, the first and last projects had more to do with scheduling, so their focus was on the real-time aspect of the course. The middle projects involved the creation/utilization of drivers for devices, and therefore, was focused on the embedded aspect of the course. The second project was the most intensive of them all and is the one I have chosen for the report. This project is more based on the embedded side through the creation of the driver, but it does involve scheduling to implement the blinking functionality required for the finally shell command later described in detail in the report.

The second project consisted of creating three shell commands for the board to run on the board. Although the console that the shell commands run on is hosted by the board, its interface is displayed by the laptop connected to the board since there is no display panel for the board. I used an extension on VS Code to access this UART shell and run the commands that are flashed to the board.

The first shell command controls the RGB values of an LED light with the number for each of the colors being the duty cycle percentage, which gives intensity control to each color. An example shell command would be "p2 rgb 20 40 60". Intensity control is not directly possible as each light can only be on or off using GPIO (General Purpose Input/Output). However, using PWM (Pulse Width Modulation) devices already on the board, which essentially turn on and off the lights at a specified frequency, the intensity of the lights can be simulated by its duty cycle. On a power signal level, Pulse Width Modulation "is a way of digitally encoding analog signal levels" while also reducing power and effects of noise [1]. On top of the obvious dimming effect, simulating the intensity of the lights allows virtually every color on the spectrum to be created by this LED device by modifying the parameters of the shell command.

The next two shell commands interact with a MAX 7219 LED panel. The panel is an 8x8 display of red LEDs. The first command has an index value specifying a starting row followed by hex values specifying how the LEDs of each row should light up. An example command would look like "p2 ledm 3 0xff 0x00 0xf0", which would turn on all the lights in the fourth row, off all the lights in the fifth row, and on half the lights in the sixth row (Note: The rows start at index 0). The last shell command turns on and off blinking mode for the display which turns off all the LEDs on the display and then restores the display to its previous state every second. "p2 ledb 1" is an example command that turns blinking mode on, and if the 1 is replaced with 0, blinking mode is then turned off.

Implementing the last two commands were more challenging than the first. Unlike the first command which interacted with the PWM devices already configured on the board, the last two commands required a driver to be created for the MAX 7219 display and the device type had to be defined in the devicetree for the driver to use.

2. Explanation of the Solution

2.1 Preparing the Board

The MIMXRT1050-EVKB board, by default, does not have SPI devices connected. Therefore, the hardware on the board must be modified. The way this was to be done was by soldering the corresponding connections on the underside of the board. With the board being as thin as it is and the connection pads being only about a millimeter or two in size, this proved to be a fairly difficult task. The first attempt fried one of the connections resulting in the board being essentially useless for the project. Fortunately, a replacement board was provided, and the second attempt at soldering these connections was successful.

2.2 RGB LED

Creating the shell commands themselves was relatively straightforward using Zephyr's library to create the shell commands and register their corresponding functions. To create a command with the format specified like "p2 rgb 20 40 60", a "p2" shell command had to be created with a registered subcommand for the "rgb" portion with a corresponding function to process the arguments following. The same Zephyr library function was to create the all the subcommands for the project. Implementing the functions to executes the commands as desired by the description was not as straightforward.

In order to configure the hardware on the board, Zephyr uses a hierarchical device tree data structure in the form of dts files. Each node in the device tree has a compatibility property that corresponds to a binding YAML file that describes the structure of nodes of that type. When an application is built to be flashed to the board, a device type can be specified, which in this case was "mimxrt1050_evk". This allows Zephyr to determine which dts files to include by default for that application. By extension, this gives the application developer access to some devices already on the board such as the GPIO, PWM, and SPI devices. Of which, some are disabled by default on the device tree.

If further configurations are needed from what is already in the default device tree files, an overlay file can be created. The overlay file modifies the built device tree at compilation time. It was used for the first shell command to enable the FLEXPWM modules that exist in the default MIMXRT1050 device tree, because, by default, they are not enabled, and organize them for the macros in the main program to access them. The overlay was also used in this project to configure the MAX7219 device instance which will be discussed in further detail later.

To then use the devices defined and configured in the device tree in the application, the Zephyr function, device_get_binding, retrieves the data structure for the device. The instructions of the project forbade the use of hardcoded string names as parameters for the function to reference the device. Rather, we were instructed to use the Zephyr device tree macros to get these names, which is considered a best practice. Thus, it made it more readable and proper to use the already defined “pwm-leds” binding in the overlay to specify which LEDs correspond to which FLEXPWM configurations. Once the LED devices were configured and available to use, a roadblock was discovered with the configuration that prevented the devices from operating correctly.

Zephyr’s device tree for the NXP MIMXRT1050-EVKB board came with 4 FLEXPWM modules. Each of the modules had 4 submodules, and each submodule had two channels. The project’s instructions specified that the red and green LEDs would utilize the two channels of the third submodule of the first module, and the blue LED would utilize the second channel of the first submodule of the first module. The PWM driver for Zephyr’s library has a configuration setting that has submodules taking a full cycle to reload (the PWM pin set function is specified a period and pulse width). Thus, calling the pin set function back-to-back for the red and green LEDs resulted in unexpected behavior, namely one appearing and the other not and subsequent commands not being reflected in the LED device. The solutions for this problem other than using different submodules, which was not feasible due to the instructions being fixed, was to either wait a cycle between the pin set calls or to change the configuration to an immediate reload. I chose the second option. Although waiting for a period would not have had a visible effect due to how small the period was 0.02 seconds (or 20 milliseconds), I still believed it was more proper than sleeping the thread for a cycle. After connecting the LEDs to the appropriate pins on the board and configuring the PWM modules associated with them, the first shell command successfully gives intensity control to the lights.

2.3 MAX7219 Display

To lay the groundwork for the next two shell commands, I had to create a driver for MAX 7219 LED display utilizing the SPI (Serial Peripheral Interface) module included on the board. This SPI module enables the communication between the display and the board with clock and data lines. The clock line indicates to the client, in this case the display, when to read bits from the data line which gets fed data from host, in this case the board [2]. SPI also comes equipped with a chip select wire, which is needed if multiple SPI clients are in the system and therefore irrelevant in this project.

In order to create the driver that utilizes SPI, a C file with the driver code must be created in the display driver directory, a Kconfig file must be created in the same directory to define configuration options for driver and also add a dependency for SPI, and modifications must be made on the main Kconfig and CMakeLists.txt files already in the display drivers directory to allow the configuration to become available when the display configuration is enabled and to link the driver code to the newly created configuration definition.

Devices in Zephyr generally have a defined structure in which a device consists of a name, a configuration, an API, and corresponding data, all as pointers [3]. On top of needing to stay within these confines for driver implementation, the higher-level driver for displays functionally acts as a struct interface for this device creation since all the display drivers use the same struct for the APIs, so the MAX7219 display driver APIs must be in accordance with the common display interface. The project specifications only required 3 API functions to be implemented out of the display interface function, and thus all other functions required by the interface but not for the project were made to simply returned the ENOTSUP (134) error code to indicate that the function was not supported for the MAX7219 display device according to my implementation of the driver. The API functions that made the most sense to implement were the write, display_on, and display_off functions. Of the three functions, the first would be utilized for the second shell subcommand that specified the lighting pattern on the display, and the other two will be used for the last shell subcommand that turns on and off blinking mode on the display. However, before implementing these functions, one must first understand the protocol for communicating to the MAX7219 display via the SPI bus.

The MAX7219 display receives data “in 16-bit packets” and “is shifted into internal 16-bit shift register with each rising edge of CLK” according to the MAX7219/MAX7221 manual [4]. Not only is this important to understand for sending data from the board to the display, but it is also important for configuring the SPI in the driver initializer. This indicates a word set of 16, and it indicates the clock polarity is not reversed and the clock phase is on the rising edge. In Zephyr SPI driver implementation, these clock configurations are defaults, therefore, the only configuration that must be set is the word size. The format for the 16-bit instructions is as follows: bits 15 – 12 are unused, bits 11 – 8 specify the mode or “digit” which in this case is the row on the display, and the remaining bits are specific to what bits 11 – 8 are [4]. For example, if bits 11 – 8 are 0x3 (the code for the third row) and the remaining bits are 0xFF, all the lights in the third row would be turned on. Another example is if bits 11 – 8 are 0xC (the code for shutdown mode), only the last bit is relevant, and it controls whether the display is shutdown or normal mode.

Now understanding the protocols for the communication between the board and the MAX7219 display device via SPI bus, the APIs in the driver can now be implemented, and the device driver initializer can be created. The write function uses three parameters of the 5 parameters provided, x, a buffer, and a pointer to the device data structure instance, to construct the instruction for the MAX7219 display. The buffer description and y parameter are unused. The x parameter specifies the row on the display and the buffer contains the bits specifying which lights in the row light up. The display_on and display_off functions were straightforward as the display_on function simply transmitted the normal mode instruction to the board, and the display_off function transmitted the shutdown mode instruction.

The initialization process comes in two parts: the initializer function that configures the display and the initializer macro that extracts the data from the device tree of all instances of the driver’s compatibility identifier. The initializer function transmits startup configuration instructions to the display that should be done before any other instructions and initializes all LEDs of each row to be off. These configurations define the decode mode of the display, the intensity, the scan limit, and display mode. The decode mode is set to none, the scan limit is set to allow all rows to be used, and the display mode is set to normal mode instead of test mode, in which all the LEDs on the display are turned on. The intensity is set to about halfway due to electrical current limitations of the USB connection to the MIMXRT1050-EVKB board that also provides the power to the display. The initializer macro function uses built-in Zephyr macros to access the device tree where the MAX7219 device is defined including the SPI device being used, the SPI frequency, and dimensions of the display, calls the first initializer function, and assigns the aforementioned data as well as the APIs to the object in the application when the device binding is requested.

Now with the creation of the driver, the remaining 2 shell commands in the application can now be fully implemented. The "p2 ledm" command uses the write API from the driver to write the user-defined bytes in the form of bits from the row number specified as mentioned in the beginning. The "p2 ledb" command requires the creation of another thread to simulate the blinking display and uses the display_on and display_off functions from the device driver API. Essentially, the thread will turn the display on, sleep for a second, turn the display off, sleep for a second, and repeat while the blinking condition is true. This thread had to have a higher priority than the main thread, otherwise it wouldn't run as the main thread will not yield to the blinking thread. When the "p2 ledb" shell command is run with the 1 argument, a new thread is created (if one does not already exist), and the global blinking Boolean variable is set to true. When the shell command is run with the 0 argument, the global blinking Boolean variable is set to false, and the thread will terminate itself if it is running.

3. Description of the Results

The result of the project was an application with three fully functioning shell commands that can be called with the UART shell that interact with a single RGB LED bulb and MAX7219 display. The application files could be placed in any Zephyr workspace to run; however, the driver and its associated CMakeLists and Kconfig additions/modifications had to be delivered in the form of a patch file since the driver changes are higher up in the Zephyr tree than the application directory.

The following image demonstrates how through using the PWM modules to control the duty cycle percentage for each color in the RGB LED, intensity control and by extension color control can be simulated by the first command. The command used to display the yellow light was "p2 rgb 100 15 0". The command used to display the purple light was "p2 rgb 70 10 95". The weakness of the red light compared to the green and blue lights on the device required the percentages for the color to be slightly different than what the actual rgb percentages are.

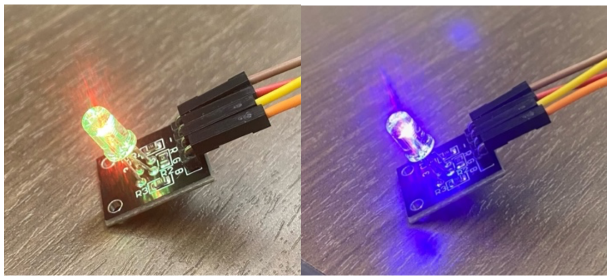

Figure 1. Images of yellow and purple light from RGB LED

The following image illustrates how the second command grants the user the ability to interact with the display. The specific command for this display configuration was "p2 ledm 0 0x30 0x78 0x7C 0x4E 0x4E 0x7C 0x78 0x30".

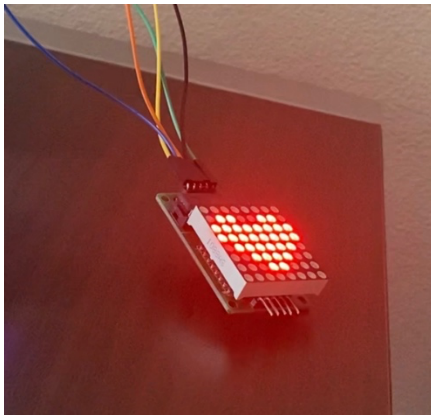

Figure 2. Image of MAX7219 display

I had not worked this closely to hardware components before this project. I was able to gain insight into the process of how central systems interact with external devices through drivers that are specific to the device and the protocols the device uses to communicate. It also taught me, quite painfully might I add, how much more expensive mistakes are the closer they are to the hardware. An issue in the driver will plague all the applications and dependent drivers that use it. Even further upstream, a broken component in the hardware could render the entire system useless. As mentioned in section 2.1, a mistake in the initial step caused the connections to burn off and required another board entirely.

I also learned how to handle working with significantly less resources than I am used to. Embedded systems is a specialized field in meaning the resources available are small relative to standard application software development. Hence, whenever I encountered any issues and went online to try to troubleshoot, I found next to nothing much of the time.

Although not explicitly taught, but rather by example, one of the most important concepts I learned from Professor Lee was relying on the source code of a system rather than just the documentation. I had rarely looked beyond the documentation and into the source code of any library or framework that I had used due to the higher-level abstraction of application software development than embedded software development; the importance was always placed on how use it rather than how it worked, especially when, after running into problems, it is simple to find someone online that has also run into the same problem. Without that luxury, it was essential to dive into the source code of Zephyr for this project to figure out why the program was erroring or behaving in an unintended manner.

REFERENCES

[1] Michael Barr. 2001. *Pulse Width Modulation*. Embedded Systems Programming (2001), 103-104. https://barrgroup.com/Embedded-Systems/How-To/PWM-Pulse-Width-Modulation

[2] Frederic Leens. 2009. *An introduction to I2C and SPI protocols*. IEEE Instrumentation & Measurement Magazine 12, 1 (2009), 8–13. DOI=https://doi.org/10.1109/MIM.2009.4762946

[3] Zephyr Project members and individual contributors. 2022. *Device Driver Model*. Zephyr Project Documentation. Zephyr Project. https://docs.zephyrproject.org/2.6.0/reference/drivers/index.html

[4] 2021. *Serially Interfaced, 8-Digit LED Display Drivers*. MAX7219/MAX7221. Maxim Integrated Products, Inc. https://datasheets.maximintegrated.com/en/ds/MAX7219-MAX7221.pdf

Data Mining Report

Brandon Rodriguez
School of Computing and Augmented Intelligence
Arizona State University
Tempe, Arizona
blrodri4@asu.edu

ABSTRACT

In this report, I describe the background and process for completing the three projects in the CSE 572 Data Mining course. Also covered in the quantified results of the three projects as well as the improved proficiency I was able to acquire through the duration of the project and even after, since rarely are any learning models perfect, that still leaves room for improvement.

1. INTRODUCTION

The report consists of the three projects from the CSE 572 Data Mining course. The projects involved the processing of data coming from an artificial pancreas medical control system, for which data from two sources were used. The first was the continuous glucose monitor (CGM) part of the system that collects blood glucose levels every five minutes among other metrics that were not as relevant to the projects. The second was the insulin pump, which contains data obtained manually from the user like estimated carbohydrate intake and the mode among other data that was not particularly relevant to any of the projects in the report.

The data mining tasks of the projects involve handling missing data and extraction of properties from the data, training and testing a classification model using the data set, and clustering points from the data set and inspecting the results of the clusters. In a way, the projects build off the previous. For example, all the projects deal with imperfect data, because it is realistic data coming from sensors and user-supplied entries. Therefore, the techniques used in the first project to handle missing data were also used in the following projects. As with any learning operations, one of the foundational elements that determines the success of the outcome is feature extraction. The feature extraction process that occurred for classification in project 2 was utilized as well in project 3, with some slight modifications to improve upon the results.

All the projects were conducted using the same Python 2.7 environment. The projects also used the same Python libraries to handle the data processing and learning algorithms. The mentioned libraries used were Pandas, SciPy, Sklearn, and Numpy to name a few. For the projects the given data to process was in the form of csv/excel files, and likewise, the results produced by the Python scripts were also in csv files that were automatically graded by Gradescope, which was the submission platform used for the course.

2. Explanation of the Solution

2.1 Project 1: Preprocessing

The objective of the first project was to obtain 6 metrics based off the glucose sensor (mg/dL) for daytime, nighttime, and the entire day for the two different modes (auto and manual). The 6 metrics were the average percentage of time in hyperglycemia, average percentage of time in critical hyperglycemia, average percentage of time in the normal range, average percentage of time in normal range secondary, average percentage of time in hypoglycemia, and average percentage of time in critical hypoglycemia. Thus, there were 18 metrics per mode, and 36 metrics to extract from the data altogether.

I used the Pandas library in Python to extract the data from csv files and then process and analyze it programmatically. The Pandas library is an essential data analysis library for Python that puts Python on par with more statistically oriented languages like R [1]. The library facilitated all the data handling operations of the projects in general, but mainly for the first project since the machine learning operations in the subsequent projects was outside of the scope of Pandas.

Although the glucose sensor was supposed to collect data every 5 minutes, it was not perfect, and I even discovered an instance where the date/times were out of order. Upon extraction from the csv file, I made the date and time columns a single datetime column that would serve as the index of each row, which allowed me to sort the rows by index so the rows would be in order by date. I then located the instance where auto mode was switched on in the insulin data spreadsheet to make the distinction in the CGM data for which data belong to which mode. From there, I subdivided the data into more easily manageable segments that would be iterated through. The first split was by the mode and then each day within each of the modes would be processed. Within each day is the three time periods for which the metrics are to be extracted, and the extraction process looks the same for all three time periods.

The most significant part of this project is not the metric extraction itself from the divisions and subdivisions mentioned above, but how missing/erroneous data is handled. My handling of this problem consisted of a three-part operation. The first was to re-index the data points for each given time period. Since the data each data point was not exactly 5 minutes apart, re-indexing would move the points to their nearest 5-minute increment, which in turn handles any duplicate data points and further highlights portions of the data that are missing by assigning any 5-minute increments without nearby data as empty. This also standardizes all time periods of the same type to have the same number of data points. The next part was to discard the time periods that fall in the bottom 10% in terms of number of data points missing from that respective time period. There would be too much filling in in the next step that would interfere too much with the integrity of the data if these segments were allowed to contribute to the final calculations. The final step was to interpolate the missing data points in the remaining segments. I used the default interpolation method in Pandas which was linear interpolation. Essentially, this method draws a line between the surrounding, known points of the missing point(s) and fills them in based on that line.

From there with the data all cleaned up without any missing points or duplicates, the calculation was fairly straightforward. Once the metrics were extracted from the resulting segments for each time period and mode, they were written into a csv file where the results would be evaluated automatically on Gradescope.

2.2 Project 2: Classification

For the second project, the goal was to create a classification model that, given a series of glucose levels (CGM data) over a span of time, determines if the person has eaten a meal or not in that time. In order to train such a model, the procedure was to extract the actual mealtimes from the insulin data, which was manually recorded by the user, then extract the 30 minutes before and 2 hours after each mealtime from the CGM data as meal data and extract every 2-hour segment where a meal was not eaten (that does not overlap with any meal data segments) as no-meal data. The reason the mealtime ranges include the 30 minutes prior is to handle any user error when recording the meal intake time. After extracting these time series segments with their corresponding labels, the next step was to extract features from this data to convert the series of timestamped glucose levels to features that could be used to train the model. Finally, the last step was to fine-tune and evaluate the performance of the model on the test set.

To extract the mealtime and non-mealtime ranges and handle any erroneous data, the same strategy was used from project 1 with slight modification. The re-indexing remained the same, but the changes came from the threshold to discard segments from training based on the number of missing data points in a segment and a limit on the interpolation. Rather than be the bottom ten percent that get thrown out, any segment with more than 7 missing points was thrown out. Instead of interpolating all points of remaining segments after the last step was completed, consecutive points that were missing were not interpolated and discarded. Therefore, interpolation was only done when one missing point was surrounded by valid points.

To get an idea of what differentiating features I should extract, I first had to graph a few of the time segments from each class, meal and no-meal, into glucose level vs. time line plots. Generally speaking, the lines plot generated from meal ranges look like a short, slight decrease followed by an upwards spike and then the beginning of a decrease. The no-meal data line plots typically just look like a slow decrease. One set of features I extracted was the parameters of fitting the points to a sine curve. Another was the coefficients of a third-degree polynomial curve I fit the points to. In order to capture the spike from meals I also included a curvature value and the largest average of 5 consecutive slopes. Then I added a few random features such as range, average slope, average acceleration, etc. In hindsight, the inclusion of a lot of these features was not very useful, a flaw I attempted to correct in project 3.

Because of the curse of dimensionality problem arising from learning with too many dimensions, the features cannot be used directly to train the model and then after predict using the model. Therefore, I used Principal Component Analysis (PCA) with the Sklearn library to reduce the number of features to the top 4 most differentiating features. PCA uses eigenvector analysis to reduce the dimensions of a dataset while maintaining the variability of the original dataset [2]. Standard scaling was also performed after PCA to standardize the data, making the distribution be the same across all mealtime/non-mealtime ranges for each feature.

I chose the Support Vector Machine (SVM) classification approach rather than decision trees, because on first pass, it performed better, and generally speaking, SVMs perform better in these types of problems whereas Decision trees' strong suit is more with categorical data. One study in the similar field of diabetic research, showed SVMs to work particularly well in comparison to other classification models for diabetes prediction [2]. However, in retrospect, I did not explore the option of using decision trees in the form of Random Forest, which is an ensemble of decision trees. There can be exceptions such as this one comparison between the SVM and Random Forest classifiers in the binary classification context found the SVM classifier slightly outperformed [3]. However, what was noteworthy in the study is that the accuracy of the top SVM model was still 90%, and the nature of the data set was more linear so a linear kernel was most successful for SVM, as opposed to the non-linear set for this project where a radial kernel was most successful.

To select the best hyperparameters for SVM, namely gamma, kernel, and C (the misclassification penalty parameter), I used 5-fold cross validation in the form of the RandomSearchCV from the Sklearn library. Rather than exhaustively search a predefined range of parameters like GridSearchCV, which I initially used, RandomSearchCV samples from the predefined range of parameters, which allowed for larger ranges for the parameters to be specified and the search to be done more quickly. Before training using the hyperparameters found from the cross validation, I separated 5% of the training samples so I could use them to test the model locally before testing on the submission site.

To evaluate the results of the test, I extracted the following metrics, which all hovered around 70%: F1 score, precision, and accuracy. In order to conclude the training phase of the classification and proceed to the prediction phase, I had to save the SVM, PCA, and scaling models in the form of Python objects to a file using the Pickle library. These files are then accessed by the testing Python file, which contains nearly parallel steps as the training file, minus the meal data/no-meal data extraction portion. The testing file I implemented first read from a csv file which would be provided to the program upon submission, and this file contains an unspecified number of rows with each row containing 2 hours (24 sensor readings) worth of glucose level readings. Then I invoked the same function for feature extraction for each of these 2-hour segments of data. Using the same Pickle library to load the model objects onto file, I extracted the same model objects and applied them to the extracted features of the test segments. The order of the application had to be the same as was applied in training: PCA transformation, the standard scaling transformation, and then finally SVM model prediction. The results (1 for meal and 0 for no meal) were then written to a csv file that was evaluated by Gradescope

2.3 Project 3: Clustering

The goal of the third project was to cluster segments of glucose level data, similar to the data from the last project, based off the amount of carb intake by the user in that time period. The requirement was to cluster using K-means and DBSCAN algorithms. The "actual", or one could say target, clusters are ranges of carb intake in increments of 20. For example, if the minimum carb intake is 20 and the maximum is 80, there should be 3 clusters: 20-40, 40-60, 60-80. I say "actual" in quotes because clustering is unsupervised learning meaning ground truth labels are not utilized in training; training only finds patterns using extracted features. The clusters are to be trained and predicted with the same data set. The final goal was to derive the sum of squared error (SSE), purity, and entropy metrics from the resulting clusters of each of the two approaches.

Since a significant portion of this project up until after feature extraction drew from project 2, I created a file with the functions I had written in project 2 to reuse them. Extracting the mealtime glucose level time segments were the same, but the no-meal time segments of the data were not needed for this project. As specified previously, the feature extraction was slightly modified to get better results. I removed many of the random features like average slope and range, as well as the coefficients of the third-degree polynomial curve fit. I also added time-to-peak and maximum CGM extraction excursion as features, which is the number of data points until the peak is reached and the maximum reading minus the sixth reading (the moment when the meal is eaten) divided by the sixth reading respectively. This new set of features provided better results according to the metrics that were calculated at the end.

Once again prior to passing the training or testing data through the algorithm, I first had the data go through PCA and standard scaling. Other than a random state which changes the initial location of the cluster centers, the only parameter needed for the K-means clustering algorithm is the number of clusters, which is just the number carb intake ranges explained earlier. Implementing the DBSCAN algorithm was more challenging. Unlike K-means, the parameters are the radius Esp and the minimum number of points within Esp distance from a point for it to be considered a core point. In order to make it more manageable to fine-tune the parameters, I used the k-nearest neighbor algorithm to consolidate them into one. This works by arbitrarily selecting a minimum points value and plotting KNN to find a suitable Esp value. The elbow on the KNN plot is the suitable Esp value, which I had the program calculate automatically so I could tune the minimum points value until the algorithm produced the correct number of clusters and decent performance metrics.

To calculate the purity and entropy metrics for each of the two clustering approaches, I used Sklearn to combine the cluster results, which was just a number for each data point specifying what cluster it belongs to, and the ground truth ranges into a contingency matrix. After that I could easily perform the purity calculations, which is just the summation of the most occurring cluster count of each range (the ground truth labels) divided by the total number of data points. I then used the Scipy library to perform the weighted entropy calculation for each cluster given the matrix. The SSE calculation differed for the two approaches since the Sklearn library contains a built-in calculation of the metric under its inertia attribute of the resulting object; the library does not contain the same for DBSCAN. Therefore, the calculation for that metric had to be done manually by finding the mean of each cluster and adding up the distance between all the points in the cluster from the mean for each cluster. Once all these metrics were collected, they were written to a csv file which would be evaluated by the submissions site,

3. Description of Results

The results of the projects came in two forms: the numerical (and visual, in the case of project 3) results of the project and the grade. Full credit was awarded for the first project, and 95% was awarded to the final two projects. Some of the numerical results outside of the grades had to be inferred by the grades. For example, the professor had indicated that for the first project full credit was received for results that were within the 10% margin of error, and for the second project the F1, precision, and recall scores had to be at least 60 to get full credit. The last project's requirements were less transparent, but the grade was based on the SSE, entropy and purity results.

3.1 Project 1 Results

The tables below show the metrics extracted from the data set in the project. The metrics represent the average percentage of readings in each category, with respect to 24 hours no matter the segment (night, daytime, all day). Night is defined by the project as midnight to 6AM, and daytime is defined as 6AM to midnight. As mentioned above, it can be inferred that these metrics are within the 10% margin of error, because the project received full points.

Table 1. CGM metrics in manual mode

Time of Day	Hyper glyce mia	Hyper glyce mia critica l	In range	In secondary range	Hypogl ycemia 1	Hypogl ycemia 2
Night	5.2%	1.1%	19.1%	14.1%	0.7%	0.0%
Day	29.2%	10.8%	41.2%	31.3%	4.6%	2.0%
All	30.7%	10.1%	64.1%	49.5%	5.2%	1.9%

Table 2. CGM metrics in auto mode

Time of Day	Hyper glyce mia	Hyper glyce mia critica l	In range	In secondary range	Hypogl ycemia 1	Hypogl ycemia 2
Night	2.6%	0.4%	21.8%	19.0%	0.6%	0.2%
Day	21.8%	5.1%	49.6%	36.9%	3.6%	1.1%
All	24.7%	5.4%	71.0%	55.5%	4.3%	1.3%

3.2 Project 2 Results

The scores calculated for the model on Gradescope was 60, 55, and 57 for recall, precision, and F1 respectively. The scores hovered near the 60 mark for full credit but did not quite make it. Calculating the scores locally on a different dataset than the one used on Gradescope for the auto-grading, the scores were closer to the 70 mark, which generally indicates a solid model. However, the fact that it performed better in that case probably means that I overfit the model to the training data.

Although I took steps to avoid it such as cross-validation, there were other steps I could have taken to improve the model. One could have been as simple as seeking out more data to fit the model, because the data that I trained the model with was the CGM data for just one person, and the model may have overfit to that person. The other which I learned after the course was ensembling. This involves creating multiple, different models to generate a prediction from each along with an algorithm like just picking the majority prediction as the final classification.

3.3 Project 3 Results

The results of the final project were somewhat hard to gauge due to clustering being unsupervised learning. However, there were still some ways of observing the results. The first is through the numbers calculated (that also determined the grade). Generally, the higher the purity and the lower the SSE and entropy are, the better the clustering is. There were also two clustering approaches performed on the data; thus, it was much easier to tell how good the clustering was relatively than objectively.

Prior to starting the project, my hunch was that K-means would perform better since the number of clusters was known, and K-means approach utilizes that as a parameter. On the other hand, DBSCAN does not utilize it. It did not appear DBSCAN was well suited for this case and the data proved so. The following table shows the purity and entropy calculations for each method. The SSE metric of either approach did not appear as a by-product of the auto-testing in Gradescope, and thus, was left out of the table.

Table 3. Statistics for each clustering approach

Approach	**Purity**	**Entropy**
K-means	0.54	1.38
DBSCAN	0.44	1.55

Not only statistically but visually one can see the clusters look more defined and separated with the K-means approach than with the DBSCAN approach. This, by no means is any indicator of how "correct" these clusters are, but rather, just an observation. For instance, DBSCAN is supposed to be more resistant to noise, which is why all the sparse points to the right have been identified as noise in this approach, as opposed to K-means which has identified these as two clusters. In one scenario, they very well may be noise points. However, on the other hand, they may be the clusters of the maximum and minimum ranges of carb intake, which would have fewer points. The following images plotted the points using the 3 PCA feature values and a color to indicate the cluster. This was a different data set than was used on the Gradescope grading from the metrics above.

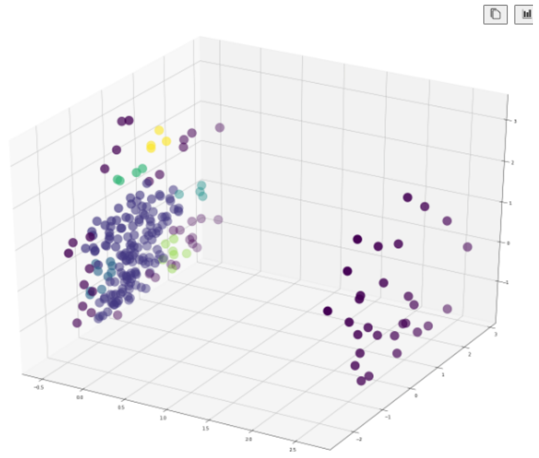

Figure 1. DBSCAN Clustering

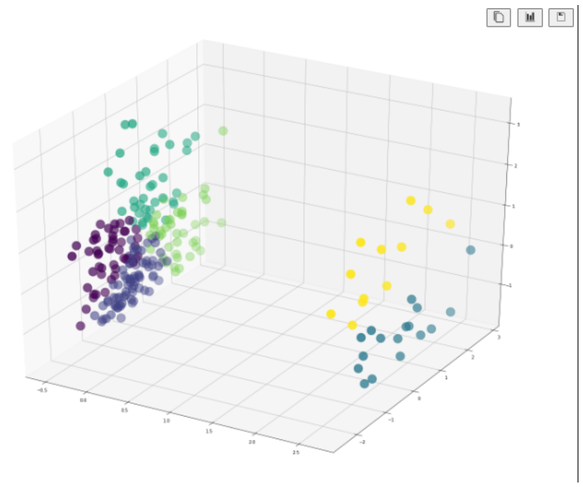

Figure 2. K-means Clustering

3.4 Conclusion

Throughout the course of these projects, I was able to learn a lot, particularly about the importance of the preparation of data. Real-life data will rarely be perfect, and I learned that first-hand in dealing with the data from the glucose level sensor and even manually entered data. Another key to preparing data was being able to visualize the data in a way that made sense to proceed and extract valid features. The data given for this project was thousands of entries long, making it hard to gather any insight just looking at it; therefore, it was important to graph the data in an appropriate manner to recognize useful patterns. I noticed the process takes a certain amount of persistence and creativity to improve results, especially as requirements become tougher to meet such as deadlines.

4. REFERENCES

[1] Wes McKinney. 2011. pandas: a Foundational Python Library for Data Analysis and Statistics.

[2] Ian T. Jolliffe and Jorge Cadima. 2016. Principal component analysis: a review and recent developments. Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences 374, 2065 (2016), 20150202. DOI=https//doi.org/10.1098/rsta.2015.0202

[3] Jobeda Jamal Khanam and Simon Y. Foo. 2021. A comparison of machine learning algorithms for diabetes prediction. ICT Express 7, 4 (2021), 432–439. DOI=https://doi.org/https://doi.org/10.1016/j.icte.2021.02.004

[4] Lukas Arnroth and Jonni Fiddler Dennis. 2016. Supervised Learning Techniques : A comparison of the Random Forest and the Support Vector Machine.