Milestone 5 Report

Engicoders
H. Bird, B. Karacelik, J. Peters, J. Ropotar, A. Rybka
Department of Computer Science, University of British Columbia
COSC310: Software Engineering
Dr. Gema Rodriguez-Perez
April 12th, 2024

(2%) Update the requirement document, including:

- **A brief description of the software you are building**
- **A list of user groups for your software, along with an example scenario for each user group of how they will use the software**
- **The final list of requirements of the software that was built. Incorporate feedback from M2.**

Overview of State

Our team has developed a Manufacturing Operations Webpage aimed at enhancing management's ability to forecast line scheduling by leveraging Machine Learning predictions based on existing data. As an Operations Manager, gauging the demand for specific product customizations can be challenging. Take Toyota, for instance, which offers its customers various trims, colours, and specification options. When optimizing a manufacturing assembly line, it's crucial to anticipate where demand will lie for particular features, enabling the creation of appropriate schedules and meeting deadlines. As a group of Manufacturing Engineering students, we recognized this challenge firsthand and sought to create a software application that empowers users, such as Operations Managers, to utilize existing datasets and machine learning algorithms for more accurate demand prediction.

The project gathers IoT data via an API from the "Thinkspeak" IoT cloud platform, covering various data types like pressure, temperature, and speed, as long as they're time-stamped. Targeting manufacturing engineers, the software includes data cleansing with filters and outlier removal. Sk-learn ML algorithms forecast future trends. Alarms alert users to high/low sensor values. Processed data is visualized for insights and exportability. An alert system identifies critical sensor values.

The software offers various functionalities tailored to regular users' needs. Users can test the software for free by selecting a preset database from a drop-down menu. They can filter and cleanse data, ensuring accurate visualizations. Users can choose from different visualization types and personalize them by manipulating the axes. Additionally, they can plot the direction of their data to predict future results and visualize multiple sensors simultaneously for comparison. Setting alerts for dangerous sensor values and managing alarms to prevent common issues like chattering are also available. To ensure confidence in data-driven decisions, users can view the range of accuracy for predicted data. Finally, premium users can download visualizations and alarm logs for future reference, facilitating easy access to saved data and analyses.

Status of Software Implementation

<u>Implemented Requirements</u>

W

- Sign up
- Sign in
- Upgrade
- Select data from API and use
- Filter data
- Select visualization type
- High low limits
- Alarm mgmt
- Prediction
- Display results
- Export results
- Alarm log
- Export alarm log
- Sign out
- Premium upload

- Seamless selection from API
- The webpage should run on any modern device, including mobile.
- The prediction algorithm should not take more than 1 minute to process future data
- User input data must be securely stored, as it could include confidential information
- The webpage must be able to handle multiple user interactions at the same time

The software implementation has successfully met the specified requirements, ensuring a seamless user experience. Users can easily sign up, sign in, and upgrade their accounts. The system allows for the selection of data from an API, enabling users to filter and visualize information according to their preferences. With high-low limits and alarm management functionalities in place, users can effectively monitor data fluctuations and set alerts accordingly. The prediction algorithm processes future data promptly, ensuring results are generated within a minute. Importantly, user input data is securely stored to protect confidential information. The webpage is designed to be responsive and compatible with all modern devices, including mobile, facilitating accessibility for users on the go. Additionally, the system can handle multiple user interactions simultaneously, providing a smooth and efficient experience. With features like exporting results and alarm logs, users have the flexibility to manage and analyze data effectively. Finally, users can securely sign out, and premium users can also upload data seamlessly.

- How many of your initial requirements that your team set out to deliver did you actually deliver (a checklist/table would help to summarize)? Were you able to deliver everything or are there things missing? Did your initial requirements sufficiently capture the details needed for the project?

Unimplemented Requirements

Initially, our project intended to allow users to input or select multiple sensors to visualize in the end. Unfortunately, we could not implement this into the program due to time constraints as well as it would cause difficulties with the way our project takes inputs from users. We also intended to have preset data from an API in a list for users to easily select, but now we make the users get the channel and route ID from a public channel in thinkspeak to access data to use in the program. Finally, the users are not able to manipulate the axis of the charts since that can not be implemented with the selected chart import. Chart.js automatically chooses the best axis to view the input data anyway.

Backlogged Tasks

When problems arose with user stories and tasks that were either changed or not possible, the tasks were altered and implemented or marked as not feasible and closed respectively. This left no tasks in the backlog by the end of the project, but the notable unfinished or changed tasks are task #34 multiple sensors which was marked as unfeasible, task #109 and #132 which changes the preset data list to inputs from the user which gets data from thinkspeak, and task #259 axis manipulation which was marked as unfeasible and unnecessary due to chart.js doing the work already.

System Architecture

The system is designed around 3 design patterns. Primarily, it focuses on an observer pattern between the data inputted and the alarms that observe it. Other classes operate around this central core. Alarm manager is a singleton that manages the alarms, keeping them present for modifications to prediction changes with the dataset. There can only be one alarm manager, as all alarms are treated as global. The reasoning for this was that the system can only process one data set at a time, so alarms, as created, should be maintained during changes in parameters on the set. There is another pattern implemented with the decorator pattern. This is used to dynamically add runtime functionality to the sensor class. It allows the sensor to extend functionality to do prediction and cleansing of the data.

There are two ways of adding data to the system. First data may be queried from the thinspeak API. This requires sending an HTTP GET request to the public thinkpeaks channel where the data exists. While one class handles these GET requests, a separate class handles the input of data from the user via an .xlsx file. Together these two classes represent the data acquisition functionality of the application.

Surrounding the data processing is the rest of the system. A login, registration, and user system was also created. Upon entering the website for the first time, or while not logged in, users are placed on a welcome splash that introduces the system. From there, they can log in or

register to access the main page. This page is where data can be processed and alarms set. This main page has different features depending on user type, where regular users can only use public thinkspeak channels, and premium users can upload their data. Upgrading is possible for a regular user, by entering a specific upgrades page. Furthermore, on the main page, users can set alarms in the sidebar.

An example flow for the system would be a user arriving on the welcome page, logging in, choosing a public sensor to use, or using their own data if they're a premium user, then setting alarms and plotting their data. If they were a premium user, they would then be able to download their data.

Overall, our project outcome is very similar to what we had originally intended. Our flow pattern is exactly what we had hoped to achieve. Some small features ended up different, however the core of the system is the same. One change was that initially it was planned that there would be pregenerated datasets that could be selected, but it was decided that it would be a better user experience to allow for public sensors to be used. This gave users the ability to see how features worked on any public sensor they could find. Alarms also changed, with them shifting from real time processing disrupts to post process notices. This was justified as the data was always historical or predicted data, meaning there was no need for a real time disrupt. The core of the system stayed consistent as the focus on taking a dataset, cleansing it, and predicting future trends in it.

- What is the architecture of the system? What are the key components? What design patterns did you use in the implementation? This needs to be sufficiently detailed so that the reader will have an understanding of what was built and what components were used/created. You will need to reflect on what you planned to build vs what you built.

Code Reuse

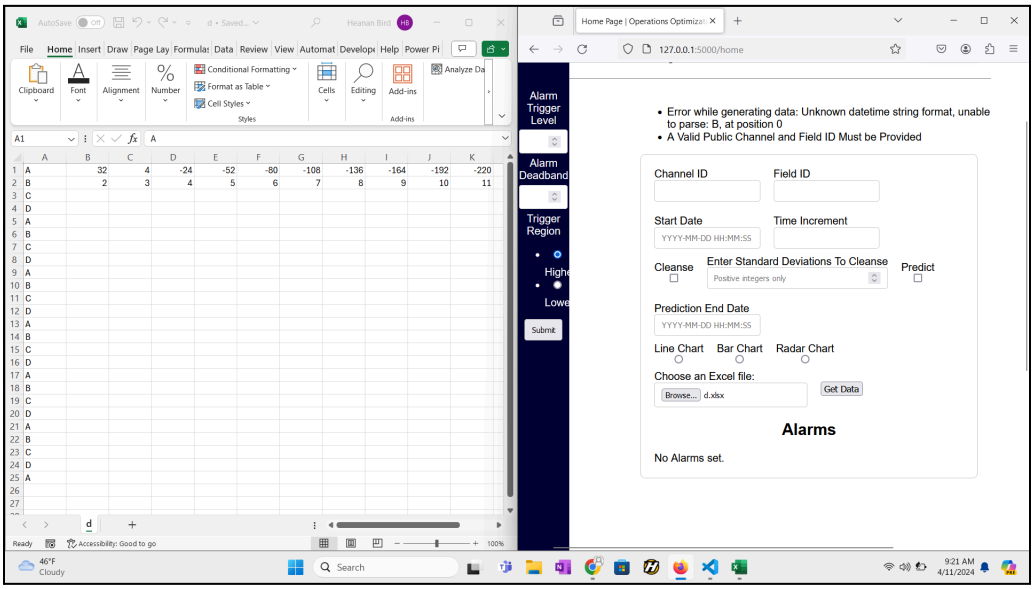
We worked to ensure as much code reuse as possible throughout this project. Anything that could be reused would be. This included things like the decorator classes for the sensor being reused to implement the observer pattern between the alarm and the sensor. Our user class used inheritance of the Flask user to avoid duplication of functionality and create an easier flow. All of our forms implemented Flask forms, which also aided in our code reuse. These features were used in these ways in order to make flow cleaner, and to save time. It prevented different sections of code doing essentially the same thing, and allowed us to make good use of inheritance within our program. Overall, this code reuse should make it so the program is easy to follow on the code side, and allowed for minimal duplication of features.

- What degree and level of re-use was the team able to achieve and why?

Known Bugs with Solution

Improper Formatting of Uploaded Data - Double Error Display:

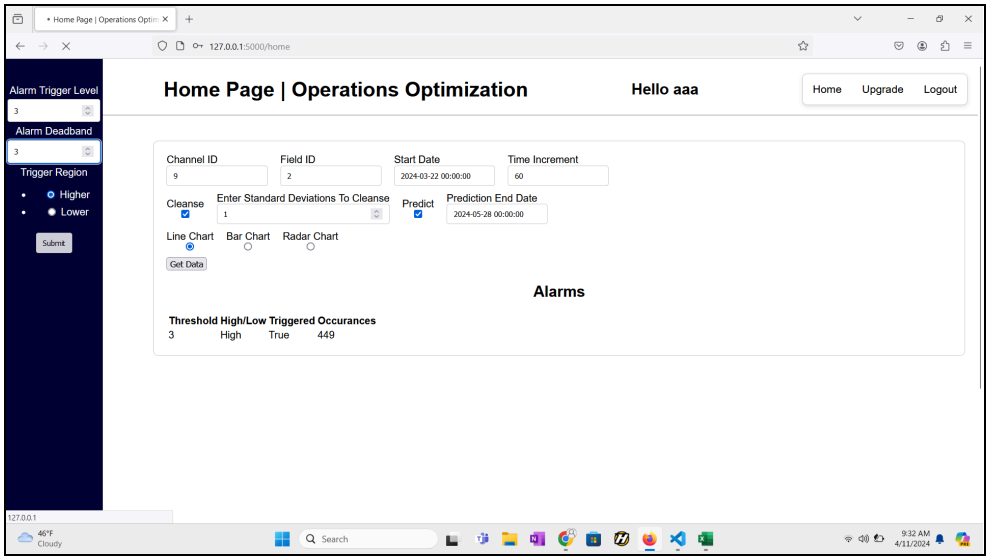
When data is uploaded that does not follow the proper format, one column of datetime parsable strings and one column of values, two errors are thrown, caught, and displayed. The first error "Error while generating data: Unknown datetime string format, unable to parse: B, at position 0" is correct and identifies the issue to the user. The second error "A Valid Public Channel and Field ID Must be Provided" does not relate to the uploaded data.



Fix: The "A Valid Public Channel and Field ID Must be Provided" error is thrown and displayed for unsuccessful plotting attempts regardless of the data source. This needs to be changed to specify if the error is from a data upload attempt vs a Thinspeak API retrieval attempt.

Reset of Prediction Model Fitting Via Alarm Manager Request:

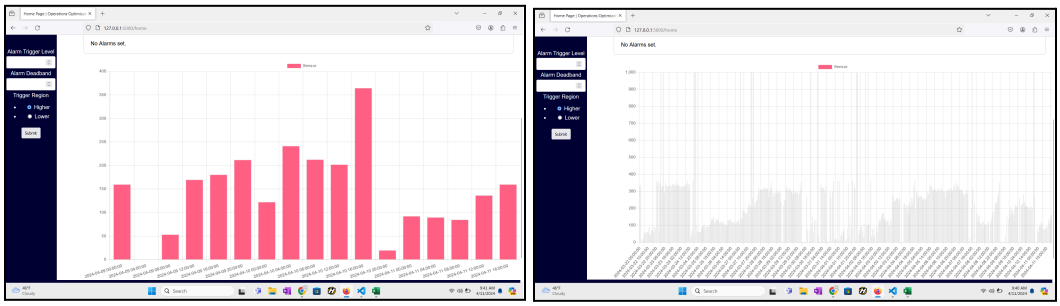
After data has been entered and a prediction is requested the web application will load as the model is fitted. If during this loading time, a value is entered to add an alarm to the alarm manager, the prediction will stop and no data will be displayed. Here the addition of the alarm is taking precedence over model fitting.



Fix: Both the Flask triggers for the Alarm and Data entry forms can be found in the same routing function. This allows for the interruption of one to occur via the submission of the other. To stop this each form submission and subsequent logic should be in a different routing form but routing to the same page. This would require refactoring and some minor logical changes of code.

Excessive Data Display Overlap On Bar Chart:

When an excessive quantity of data is provided to the chart and a bar chart is selected as the means of display, a display bug occurs. This bug is a result of the chart.js attempting to display a bar of data for each data point. As the number of datapoints grows chart.js automatically resizes the bars. But for excessive quantities of data, the bars are forced to overlap and an overlapped bar greys itself out. This means that the small quantities of data will display correctly but excessive data quantities will be difficult to see and read. The point at which this bug occurs is depend on the screen size being used.



Fix: A bar chart itself is useful for small and medium quantities of data. The best fix would be to limit the amount of data the bar chart selection can plot. Before the data

query is sent to thinskpeak the length of the query can be estimated. If too large an error can be thrown, caught, and displayed to the user restricting the use of the bar chart in the form input so as the stop the bug from occurring.

- Indicate any known bugs in the software.
- Explain how these bugs can be fixed.

Project Handover - Based on README.md

- Installation details needed to deploy the project
- Dependencies needed to deploy the project
- Any maintenance issues required to run the project (e.g., account information)

To run this project, the repository must be downloaded/cloned from GitHub onto a machine with Python 3 installed.

To run the Flask application on your local machine, follow these steps:

1. Open your terminal in the folder where the repository has been cloned
2. Install the required libraries & dependencies by using
   `pip install -r requirements.txt`
3. Initialize the database via the use of the following command in the terminal
   `pytest`
4. Navigate to the project directory using
   `cd src`
5. Finally, start the Flask application with the command
   `flask --app flaskr run`
6. In your terminal, the Flask application will start on localhost **127.0.0.1:5000**
7. Copy **127.0.0.1:5000** into a **Chrome** or **Firefox** browser and the web app will be accessible

Potential Maintenance Issues:

When starting the application two things may happen. First, you may already be logged in as a regular user. At this point, you may log out, upgrade, or use the application as is. Two, you may not be logged in at all. At this point, you may log in, and upgrade, but will not be able to use the application until an account has been made.

Reflections

W

1. How did your project management work for the team? What was the hardest thing, and what would you do the same/differently the next time you plan to complete a project like this?
2.

- One of the largest hurdles was the cross-functional skill set of our team. All members came from different programming backgrounds due to the unique curriculum of the engineering program. Conventainlayy all team members would be COSC students with a similar basis of knowledge. This difference was both a strength and a weakness. It brought unique perspectives but left room for assumed knowledge to create roadblocks. Resolve this required an “excessive” amount of communication to be clear about the needs of the project. I believe our approach was effective the one change would be to set up independent meetings earlier in the term so that roadblocks could be anticipated.

3. Do you feel that your initial requirements were sufficiently detailed for this project? Which requirements did you miss or overlook?

4. What did you miss in your initial planning for the project (beyond just the requirements)?

5. Would you (as a team) deal with testing differently in the future?

6. If you were to estimate the efforts required for this project again, what would you consider? (Really I am asking the team to reflect on the difference between what you thought it would take to complete the project vs what it took to deliver it).

- Understanding background programming frameworks
    - Learning about what flask is, how it works, understanding python, html, jinga, etc.
    - The time it takes to learn all these things before being able to start programming.

- Figuring out how to implement alarms and how time consuming that would be

7. What did your team do that you feel is unique or something that the team is especially proud of (was there a big learning moment that the team had in terms of gaining knowledge of a new concept/process that was implemented).

Our group was able to create a system that does something real with no prior experience working with programming. Some of us knew basic java or python, but nothing to the extent we knew by the end.