CS 4485.001

Computer Science Project

Mobile App for Visualization and Analysis of Data from Sensors

Table of Contents

Problem

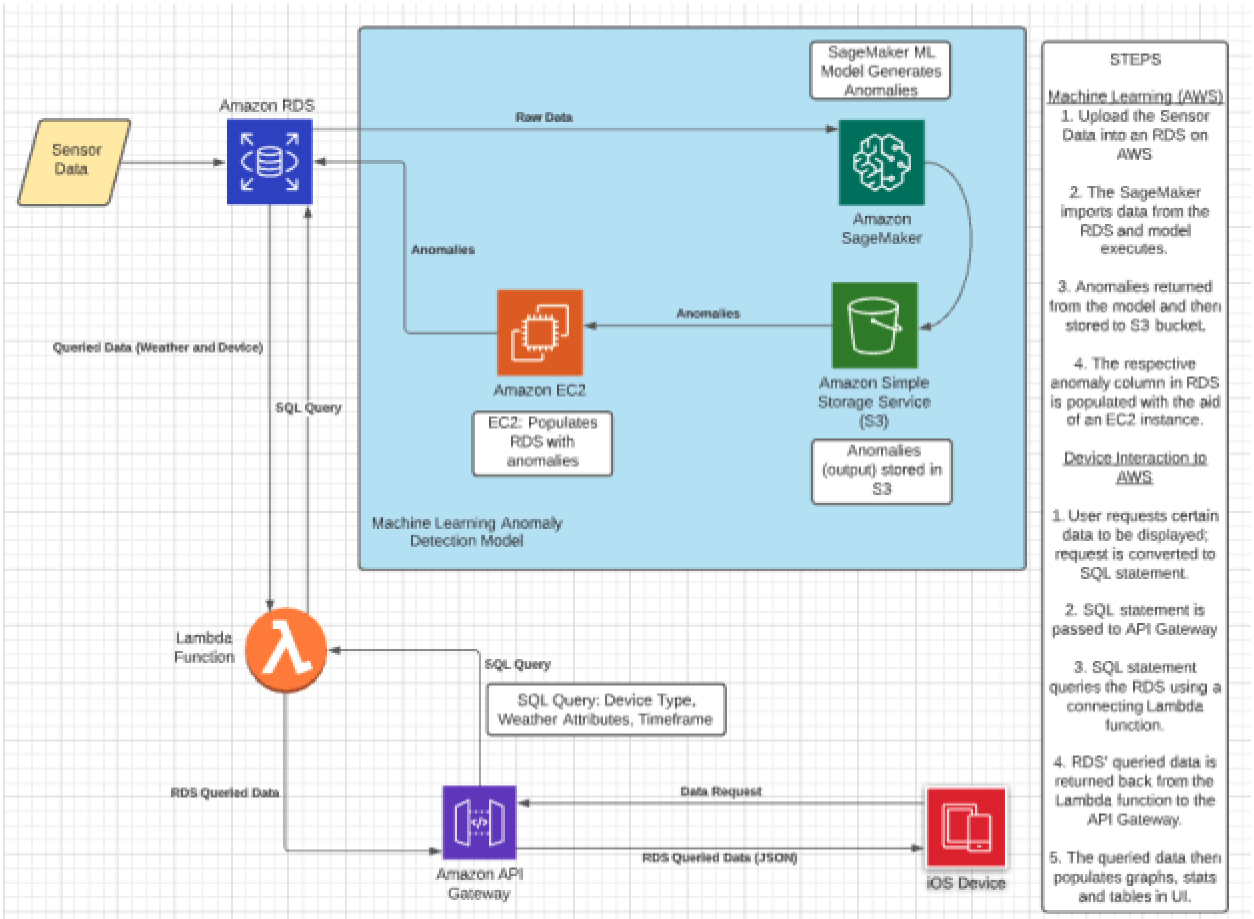
Electrical devices and appliances in our homes and various establishments that use too much energy contribute to both increased electricity costs and CO2 emissions. People do not have numerous options to visualize their energy consumption habits and visualizing how and when they are consuming energy will aid in reducing consumption wherever possible. Our goal is to inform consumers of energy use anomalies, like a left on stove or electrical short, so they can take steps to optimize their devices and reduce their overall energy costs and carbon footprint. Using our app the user could see what appliances are using the most electricity and begin to diagnose why by analyzing usage history and abnormalities. Once the problem has been identified steps can be taken to reduce energy consumption, such as repairing faulty equipment or buying more energy friendly appliances. The user can then ensure their implemented solution worked by comparing energy usage data from before and after their solution. Utilizing our ios phone application provides users the opportunity to check relevant data per appliance (like power usage, humidity, temperature, etc) in easy to digest graphs, which is not currently offered in the realm of energy consumption. In the status quo, energy providers offer mere data only about energy consumption, which doesn't offer descriptions for energy consumption per appliance. Understanding, analyzing, and visualizing each appliances' energy consumption gives valuable insights into energy saving techniques resulting in reduced energy bills. Users can get updates, based on their usage-vs-weather relationship, and these updates will result in better energy management to reduce consumption. The usage-vs-weather relationship will be generated by the ML models to detect and warn the user of any forthcoming anomaly.

Project Objective

Our objective is to develop an app to display data received from sensors (such as Temperature, Vibration, Humidity, Air Pressure, etc) on a mobile device (e.g., iPhone, or iPad). The sensors should be purchased and installed by the users to measure their energy consumption along with weather data, the weather data can be gathered by location (see "Resources Needed to Use the Application" section for examples). We will develop predictive models using machine learning methods. The purpose of the models is to establish thresholds for energy consumption, should the consumption exceed the threshold the models will generate alarms when unusual conditions/events are detected. An unusual condition/event could be caused by a faulty application, or user-driven faults, such as leaving the fridge door open, not shutting off the stovetop, etc. These alarms will be sent to the mobile device so that the user becomes aware and makes appropriate decisions/take suitable actions in order to minimize and prevent over usage. A cloud-based solution should be implemented with an intuitive UI. AWS integration will allow for more users to connect to this service without the need to expend their own devices' storage and/or computation power.

Overview

This project uses a SwiftUI front end with a AWS backend. Graph 1 depicted below shows a high level overview of the data flow for the entire project. First the AWS database is populated with sensor data. From here there are two distinct tracks that the data can flow through, machine learning anomaly detection or data display in the user interface. The specifics of what each of these services does and their contribution to the project is depicted in much more detail in the upcoming sections.


Sensor Data
Amazon RDS
Raw Data
SageMaker ML Model Generates Anomalies
Amazon SageMaker
Anomalies
Anomalies
Amazon EC2
EC2: Populates RDS with anomalies
Amazon Simple Storage Service (S3)
Anomalies (output) stored in S3
Machine Learning Anomaly Detection Model
Queried Data (Weather and Device)
SQL Query
Lambda Function
SQL Query
SQL Query: Device Type, Weather Attributes, Timeframe
RDS Queried Data
Data Request
RDS Queried Data (JSON)
Amazon API Gateway
iOS Device

STEPS

Machine Learning (AWS)

1. Upload the Sensor Data into an RDS on AWS
2. The SageMaker imports data from the RDS and model executes.
3. Anomalies returned from the model and then stored to S3 bucket.
4. The respective anomaly column in RDS is populated with the aid of an EC2 instance.

Device Interaction to AWS

1. User requests certain data to be displayed; request is converted to SQL statement.
2. SQL statement is passed to API Gateway
3. SQL statement queries the RDS using a connecting Lambda function.
4. RDS' queried data is returned back from the Lambda function to the API Gateway.
5. The queried data then populates graphs, stats and tables in UI.

Graph 1: Project Architecture Overview

Database Architecture

We started this project by designing a database that would contain the data given to us from the kaggle dataset (reference 2). First a ER diagram was created using MySQL Workbench and is depicted in figure 1, this diagram can be found on the github (reference 1) under the database folder labeled “Database Architecture ER Diagram.mwb”. Every table in this architecture has a primary key that is auto incrementing, not null, and unique. Each house (our kaggle dataset has only 1) has many devices and each device has device data, the time the data was collected, what energy use was recorded, and whether or not an anomaly was detected for this data row. Notice that the device type in devices table is an integer, we assigned each device in the kaggle dataset a number so we could easily parse them inside the backend and UI. The numbers corresponding to each device are shown in table 1.

Table 1: Device encoding scheme

Identifier	**Device**
1	time
2	Use [kW]
3	Gen [kW]
4	House overall
5	Dishwasher
6	Furnace
7	Home office
8	Fridge
9	Wine cellar
10	Garage door
11	Kitchen
12	Barn
13	Well
14	Microwave
15	Living room

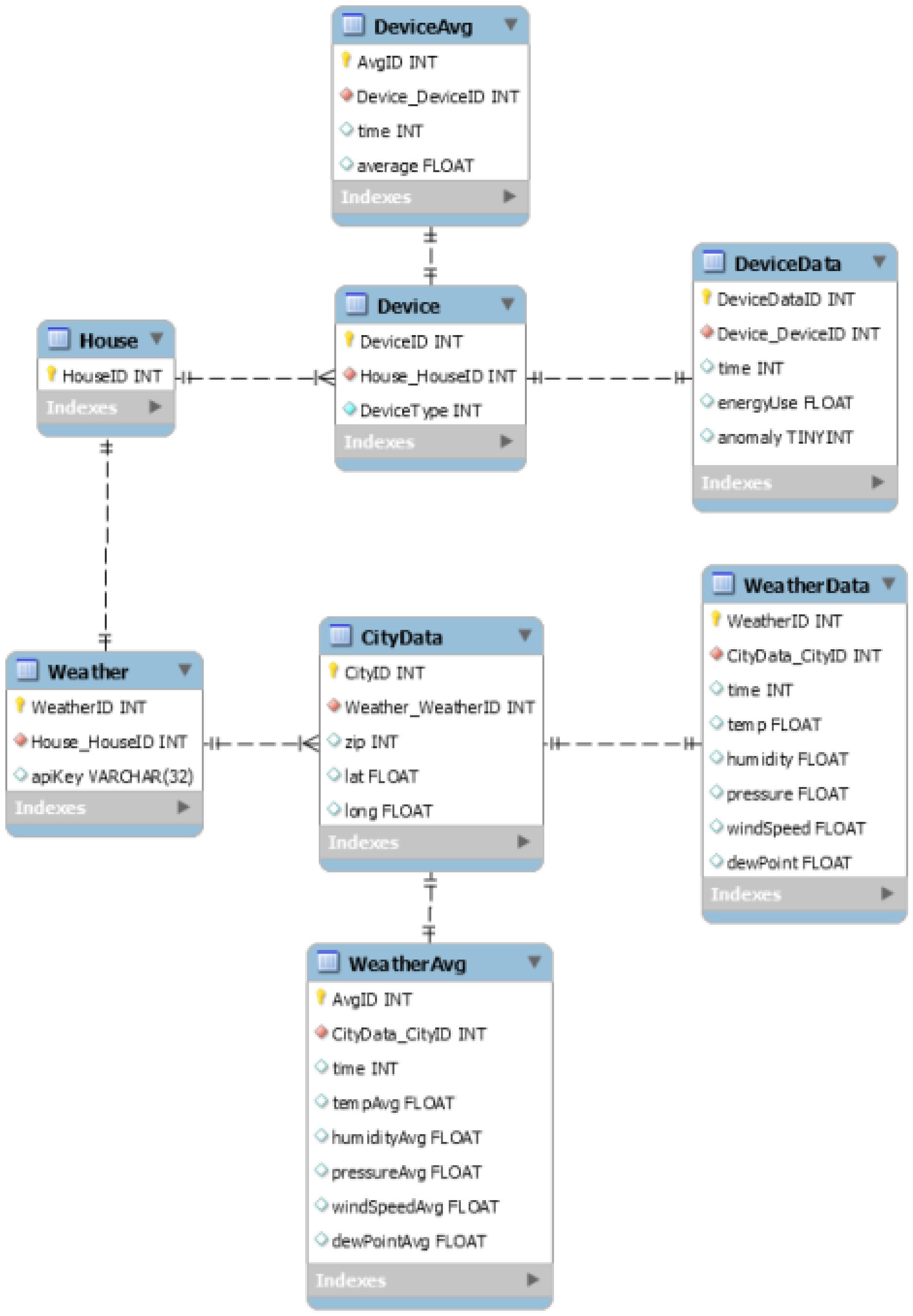


Figure 1: Database Architecture ER Diagram

Each house also has a weather table with an api key, this was included for incorporating live data from a weather api which we did not have time to implement in this project. For the same reason the weather table also can have multiple cities that all have different weather, again this was not implemented. Each city stores 6 measurements, time of recording the data, temperature, humidity, pressure, wind speed, and dewpoint. We did not include every measurement included in the kaggle dataset because some had no correlation to energy use, see machine learning section for more information.

The city and device tables also have their own average tables. The kaggle dataset collects its data every minute, and these tables hold the average across one hour of data. This was implemented to reduce the amount of data having to be queried in the user interface, and thus the wait times for data to appear was significantly reduced. These averages are calculated in the “seniordesign_average” lambda function and is discussed further in the AWS backend portion of this documentation.

Database Implementation

Once we had the architecture fleshed out we implemented it as a sql script using MySQL workbench’s forward engineer feature. Once the ER diagram is open go to File > Export > Forward Engineer SQL Create Script. This SQL file can also be found in the github (reference 1) in the database folder titled “database.sql”. Running this sql file will create the architecture necessary to hold the kaggle dataset.

Now that this architecture is implemented locally we imported the data from the kaggle dataset using a python script called “import.py” which can be found in the github (reference 1) in the database folder. This loads the data into a pandas dataframe, connects to the local database using pymysql, then creates the necessary tables and inserts all the data from the dataframe using sql inserts and a cursor. We attempted to implement the kaggle dataset into a AWS relational database (RDS) using this same method but the script would take much too long to insert all half a million rows (multiple days to execute). Instead we implemented the entire dataset to the local architecture (which is much faster, only took a few hours) and used the mysqldump utility to pack up the architecture and import it to the AWS RDS. The steps on how we did this can be found in the AWS documentation (reference 3). Now that the entire database (except for a few devices like the furnace that we did not use) is implemented in the AWS RDS we can begin implementing the lambda functions that will modify and query the data.

AWS Lambda

AWS Lambda was used for almost all (see AWS EC2) backend calculations required to make this project work. There were 7 lambda functions in total and I will describe the purpose and functionality for each below. We used two different run time environments to implement these functions, Node.js and python. I used Node.js over python for some of the functions because it found it easier to deal with the events passed from the api gateway (more on that later), however every function could have been implemented using python. Note that all the Node.js lambda functions define environmental variables at the beginning of the script, this is pulling the database endpoint, password, etc, defined in the configuration portion of the lambda function. This was done for the sake of readability, you could just as easily hard code this information straight into the Node.js file.

Weather

This function receives an event number from the api call that tells it which city we want to pull the average weather from in the database. With our implementation, as discussed in the Database Implementation portion of this document, we only have one city, this was done for the sake of expanding the project functionality in the future. The function queries the database, grabbing all the average weather data (one months worth), wraps it as a json object, and then returns it to the api call. See the source code for this function on the github (reference 1) under AWS Backend called “seniordesign_weather.zip”.

Device Average

This function is the same as the weather function above but for the device data. It gets the primary key of the device we want to average data for from the api calls event and uses it to query the database. Once it has this month of average data from the database it wraps it up in a json object then returns it to the api call. See the source code for this function on the github (reference 1) under AWS Backend called “seniordesign_deviceAVG.zip”.

Average

This function is what actually calculates and inserts the average data into the database that the above two functions use. All the calculation is done within the sql inserts using nested queries. Because it would take much too long to load the data in the user interface if this function was invoked every time we wanted average data, we instead scheduled it to run everyday at midnight using a AWS EventBridge rule. This is very simple to schedule, simply select what lambda function you want to schedule in EventBridge then use wildcards to schedule it at midnight every night like so, cron(0 0 * * ? *). Now this function will automatically calculate the averages for all 7 devices and the weather data every night at midnight. See the source code for this function on the github (reference 1) under AWS Backend called “seniordesign_average.zip”.

Anomaly

Every device data row in the database has an anomaly value that is null by default, if an anomaly is detected then this value is set to 1. This function selects all the anomalies that are not null from the past month by getting the most recent time value in the database and going back from that time 43,800 minutes (one month worth of minutes). It then returns the associated device data as a json object to the api caller. See the source code for this function on the github (reference 1) under AWS Backend called “seniordesign_anomaly.zip”.

24 Hour Weather

This function gets the most recent 24 hours of weather data from the database. It does this by getting the most recent time value the database has for weather and going backwards 24 hours from it. It then packages all the weather measurements for this time period into a json object and returns it to the api caller. See the source code for this function on the github (reference 1) under AWS Backend called “seniordesign_24hour_weather.zip”.

24 Hour Device

This function is the exact same as 24 hour weather except it gets device data. The function receives the primary key of the device it should access from the api calls event and uses it in the database query. It then gets the most recent time value from the database, goes back 24 hours, and returns the device data for this time chunk as a json object to the api caller. See the source code for this function on the github (reference 1) under AWS Backend called “seniordesign_24hour_device.zip”.

API to EC2

This function uses AWS systems manager (ssm) to invoke a python script held within a AWS EC2 instance for each of the 7 devices. The EC2 python scripts function is discussed in the AWS EC2 section below. The function evokes the EC2 python file using ssm.send_command and passes it the device primary key for what device it should work on. The function then enters a while loop that repeatedly checks the EC2 script's completion status and will only continue onto the next device once this device is finished. We had to implement this to solve concurrency issues. This function is also scheduled using AWS EventBridge to run every day at midnight exactly like the average function. No json object is returned with this function. See the source code for this function on the github (reference 1) under AWS Backend called “seniordesign_api_ec2.zip”.

AWS EC2

We needed to add to functionality within AWS to actually calculate anomalies. We first attempted to complete this by using AWS SageMaker to create the machine learning pickle file which is then inserted into a AWS S3 bucket. Then we made a lambda function that would load in this pickle file, it would be fed data queried from the database, calculate the anomalies, then insert back into the database which rows were anomalies by changing the device data anomaly value from null to 1. Here we ran into a major problem, the size of the libraries we had to use for the function to run were simply too large for an AWS Lambda function.

This function requires rather large python libraries like numpy, sklearn, pandas, etc. When we tried to import these libraries into the lambda function we hit a cap of AWS sets of 50 mb for a zipped file and 250 mb for an unzipped file. We tried many workarounds for this like importing the libraries through the lambda functions layers feature to no avail. After speaking with the technical mentor for this project, Mr. Sailendra Mishra, we decided to instead implement this function in a AWS EC2 instance, which has much larger file size limits, and invoke the EC2 script using a lambda function (AWS to EC2 under the AWS Lambda section above).

The functionality for this python script is a little different than what we first tried with the lambda function. We fully trained the machine learning model using AWS SageMaker and exported the pickle file to an AWS S3 bucket. The EC2 script then does the following in the order described:

1. Get the primary key of the device we want to access from the command line arguments
2. Connect to the AWS database
3. Get 24 hours worth of device data from the database
4. Load the machine learning pickle file from the S3 bucket
5. Predict anomalies using 24 hours of device data fetched in step 3
6. Get all average device data from the database for the selected device
7. Loop through every row of device data and set null anomaly value to 1

See the source code for this function on the github (reference 1) under AWS Backend called "ML_single_device.py".

AWS API Gateway

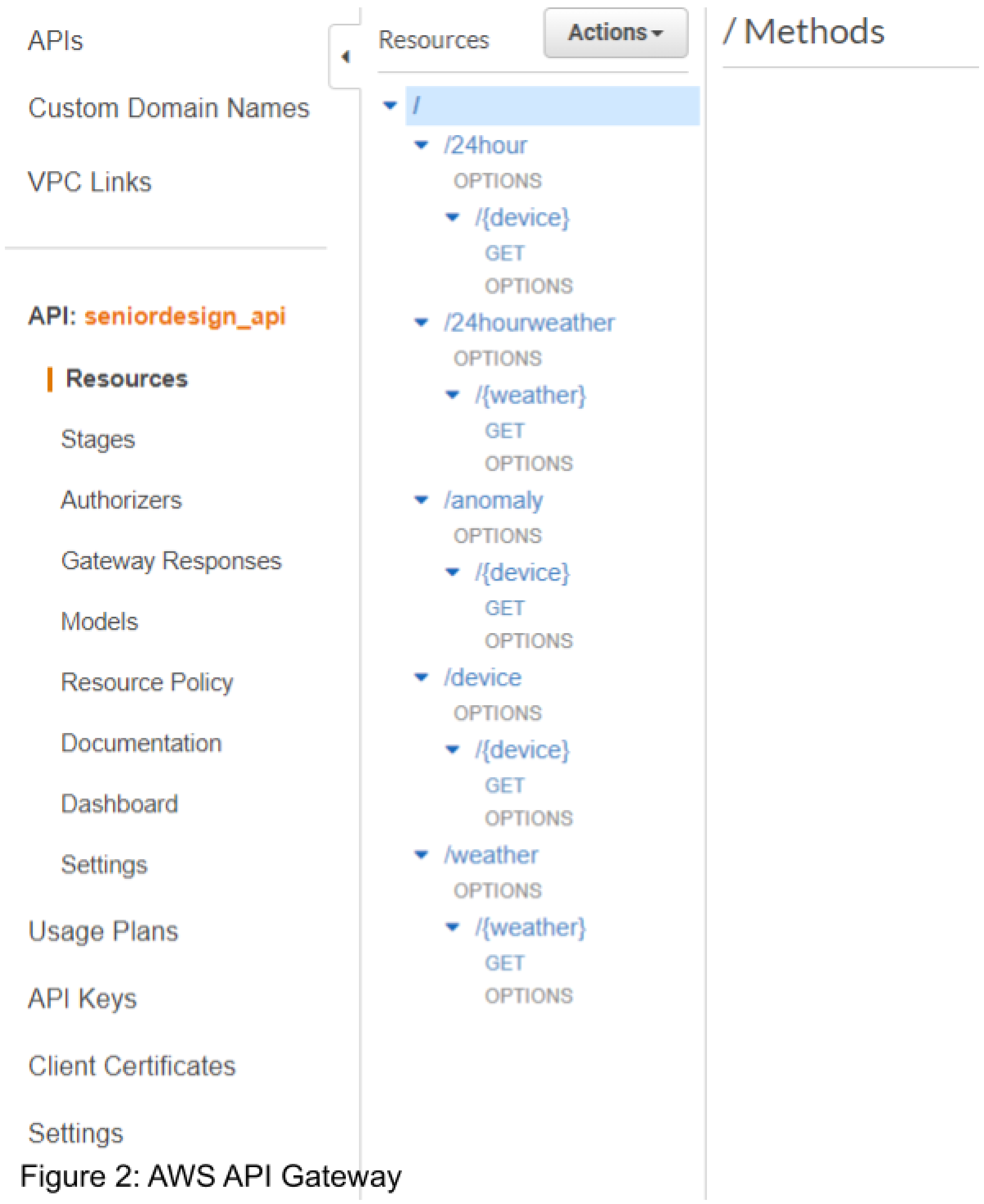
Mentioned multiple times in the AWS Lambda section of this documentation is api calls. We invoke the various AWS lambda functions using the AWS API Gateway. The API Gateway generate a unique endpoint link similar to a URL, this link has different resources added to the end of it that tells the API Gateway what lambda function we would like to to invoke, and then is also given the **primary key** (not the device number depicted in table 1) of the device we would like information on. For example, take the following API Gateway link:

- https://xkvpwwf8kb.execute-api.us-east-2.amazonaws.com/prd/24hour/1

Everything before "prd" is the endpoint automatically generated by the API Gateway. The "prd" portion of the link refers to what stage of production this gateway is currently in, it is kinda like a version of source control that AWS offers. This allows you to specify different versions of the gateway you would like to invoke, our project only had one stage, "prd". This endpoint has the resource "24hour" which tells the API Gateway we would like to invoke the "24 hour device" lambda function, the other available resources are listed in table 2 and an example of our gateway setup is shown in figure 2. It then passes this resource the primary key of "1" which is passed to the lambda function telling it we want the last 24 hours of device data for the device with the primary key of 1, the house overall. The lambda function then adds this primary key to the sql query, queries the database, wraps the data into a json object, and returns the data back through this call chain to the user interface where it is decoded. This medium article was helpful for us in the creation of this API Gateway functionality (reference 5).

Table 2: AWS API Gateway resources

Resource	Lambda Function	Primary Key
/24hour	24hour_device	Device
/24hourweather	24hour_weather	City (1)
/anomaly	anomaly	Device
/device	deviceAVG	Device
/weather	weather	City (1)



APIs

Custom Domain Names

VPC Links

API: seniordesign_api

Resources

Stages

Authorizers

Gateway Responses

Models

Resource Policy

Documentation

Dashboard

Settings

Usage Plans

API Keys

Client Certificates

Settings

Resources

Actions

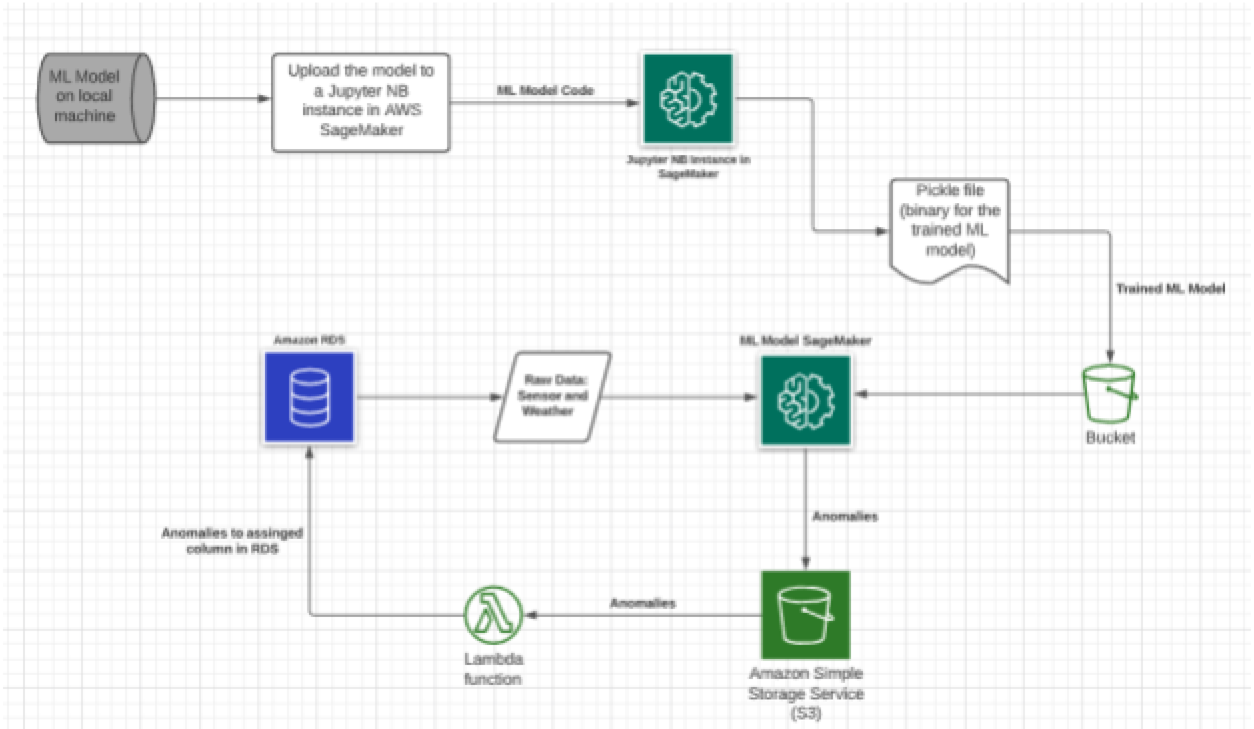
/ Methods

```
/
  /24hour
    OPTIONS
    /{device}
      GET
      OPTIONS
  /24hourweather
    OPTIONS
    /{weather}
      GET
      OPTIONS
  /anomaly
    OPTIONS
    /{device}
      GET
      OPTIONS
  /device
    OPTIONS
    /{device}
      GET
      OPTIONS
  /weather
    OPTIONS
    /{weather}
      GET
      OPTIONS
```

Figure 2: AWS API Gateway

Machine Learning

Goal of the ML Model is to predict patterns of anomalies in the data shown above. The feature considered for finding anomalies is time, as we found time to be a factor which had some correlation with the energy consumption per device (appliance). The language we chose for implementation of this model is Python. Please see (reference 4) for viewing the elementary data analysis we performed on the provided Kaggle dataset (reference 2). The dataset library is provided in the dataset; however, to emulate the application and functionality of the ML model, we decided to only preserve the relevant appliances and collectible, measurable weather attributes. The model which is currently in production is an implementation of linear regression model which breaks down the time object into weeks, days, hours, and minutes to find trends in the data. If there exists a data point outside of the tolerance levels, it is reported as an anomaly. Ultimately, each appliance will have varying tolerance levels but the model adapts to it based on the baselines it establishes for the energy for the respective appliance.


ML Model on local machine
Upload the model to a Jupyter NB instance in AWS SageMaker
ML Model Code
Jupyter NB instance in SageMaker
Pickle file (binary for the trained ML model)
Trained ML Model
Amazon RDS
Raw Data: Sensor and Weather
ML Model SageMaker
Bucket
Anomalies
Anomalies to assinged column in RDS
Anomalies
Lambda function
Amazon Simple Storage Service (S3)

Graph 2: Machine Learning Architecture

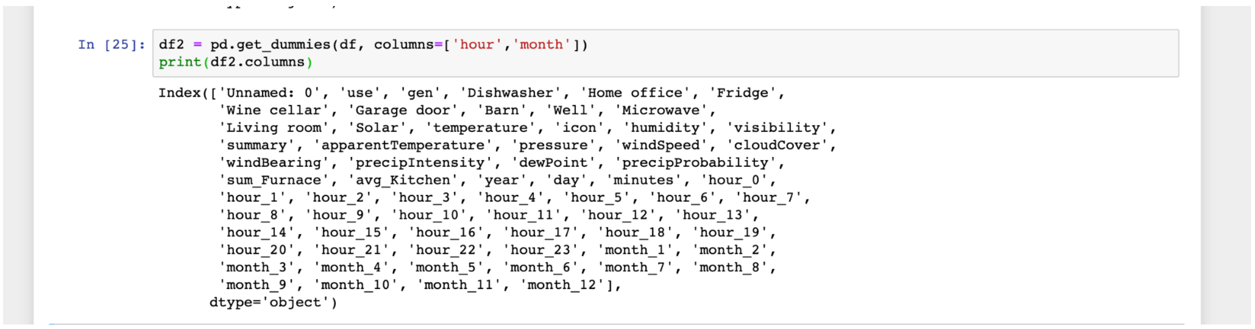
CS 4485.001

<u>How the model works</u>:

The model currently implemented is a linear regression model. The energy consumption values are the predicted (output values) whereas the time object is the independent feature. We break down time objects into hours and months using the dummies (Dummies basically means that if there is any value on that hour then it will be 1 other all will be 0. So for example **hour_1** will only be 1 when the hour is 1 else it will be 0 for all other values) and group data points. These data points are energy consumption at that specific hour, say 5 PM for every day of the month. Once the trend has been established, we try to predict the energy consumption at that specified time, by the hour. We split the model into train and test portions, the test portion being 10% of the data points in the data frame. Then, we fit the model into the Linear-Regression model in the scikit learn library. Once the predicted values are generated there compared with actual values and if the actual value exceeds the predicted value it is flagged as an anomaly in the resulting data frame.

The way we did it was that the Model is stored inside AWS Sagemaker. The file is named as **main_model.ipynb.** This file helps generate the model and then stores the model into a Pickle file and then that file is stored inside an S3 bucket name as **linear_reg.pk1.** Here I am including some screenshots of the code. If more help needed please feel free to contact me on yash291098@gmail.com

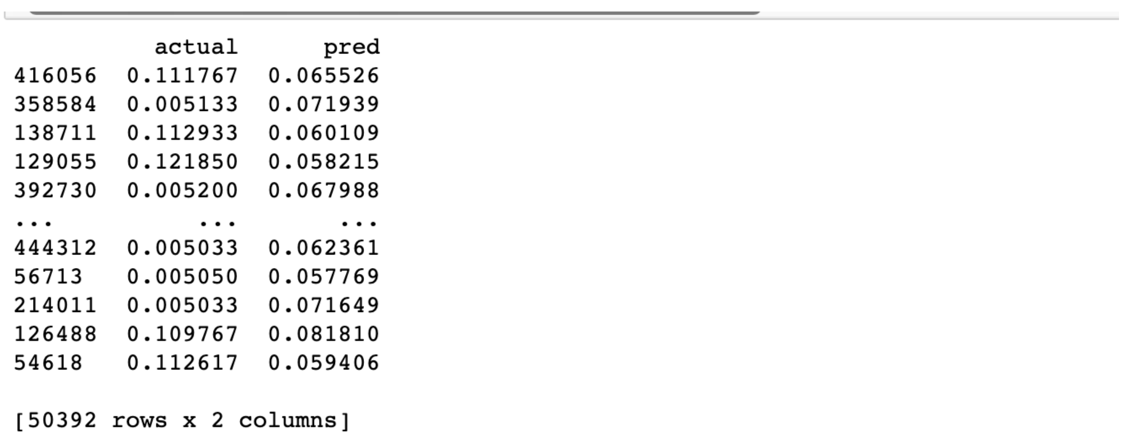
--------> The code file is present in Github as **main_model.py**



```
In [25]: df2 = pd.get_dummies(df, columns=['hour','month'])
         print(df2.columns)

         Index(['Unnamed: 0', 'use', 'gen', 'Dishwasher', 'Home office', 'Fridge',
                'Wine cellar', 'Garage door', 'Barn', 'Well', 'Microwave',
                'Living room', 'Solar', 'temperature', 'icon', 'humidity', 'visibility',
                'summary', 'apparentTemperature', 'pressure', 'windSpeed', 'cloudCover',
                'windBearing', 'precipIntensity', 'dewPoint', 'precipProbability',
                'sum_Furnace', 'avg_Kitchen', 'year', 'day', 'minutes', 'hour_0',
                'hour_1', 'hour_2', 'hour_3', 'hour_4', 'hour_5', 'hour_6', 'hour_7',
                'hour_8', 'hour_9', 'hour_10', 'hour_11', 'hour_12', 'hour_13',
                'hour_14', 'hour_15', 'hour_16', 'hour_17', 'hour_18', 'hour_19',
                'hour_20', 'hour_21', 'hour_22', 'hour_23', 'month_1', 'month_2',
                'month_3', 'month_4', 'month_5', 'month_6', 'month_7', 'month_8',
                'month_9', 'month_10', 'month_11', 'month_12'],
               dtype='object')
```

ML-1 : This image shows how the dummies are created for each hour



```
          actual      pred
416056  0.111767  0.065526
358584  0.005133  0.071939
138711  0.112933  0.060109
129055  0.121850  0.058215
392730  0.005200  0.067988
...          ...       ...
444312  0.005033  0.062361
56713   0.005050  0.057769
214011  0.005033  0.071649
126488  0.109767  0.081810
54618   0.112617  0.059406

[50392 rows x 2 columns]
```

ML-2 : shows example of actual and predicted values

<u>How the model was evaluated</u>:

Re-evaluate the model based on mean squared error and root mean squared error. The error is calculated with the predicted and actual values.
Calculated error for the deployed mode:

- RMSE (Root Mean Squared Error)
  - 0.0793
- MSE (Mean Squared Error)
- 0.00629
- MAE (Mean Absolute Error)
- 0.0440
- MAPE (Mean-absolute Percentage Error)
  - 131.033%
- Score
  - 2.106% the score is pretty low but we believe time-series models like ARIMA and LSTM should help increase the score.

What the errors represent: the low accuracy based on the error values shows that the trend in the energy consumption is not linear and needs a different model to capture the data points well considering the variability and the non-linearity of the data set.

User Interface

Our team did not have any experience in SwiftUI at the beginning of this project. Most of the UI was built from information we learned in the extensive series that can be found on O'Reilly called "SwiftUI - The Complete Developer Course and SwiftUI Bible" (reference 6). There are quite a few swift files in this project but I will discuss the main ones below. Note that the user interface code is held in a seterate GItHub here (reference 7).

AppData.swift

This file is where we define the data structures that hold the data from the api calls, and define the api calls themselves. There are 5 structures defined in total. DeviceData holds data from the deviceAVG lambda function (/device api resource). WeatherData holds data from the weather lambda function (/weather api resource). DayData holds data from the 24hour_device lambda function (/24hour api resource). WeatherDayData holds data from the 24hour_weather lambda function (/24hourweather api resource). Finally the AnomalyData structure holds data from the anomaly lambda function (/anomaly api resource).

Notice that the api calls are made asynchronously so that the application is not frozen when we are fetching data. Once the api call is made we decode the json object and then load it into the respective data structure defined above. Note that this file simply defines the functionality for fetching and storing data from the api calls, they are invoked in the DetailView.swift file discussed later.

ContentView.swift

This swift file declares the main app page that you see when you first enter the application. Notice the series of navigation links that each declare a new detail view or weather view (discussed later). They also pass the AppData.swift file a "selected" value which you notice is concatenated to the api call string as the device's primary key in the AppData file. This is how the api call knows what data we want depending on what device button we click. Each button also has a custom set of modifiers that define its look in the "ButtonModifiers" structure.

DetailView.swift

This detail view file and the weather view file discussed next are where most of the magic happens in this application. This file defines what the device pages look like when we click a button on the homepage to view a device. First this file displays the title of what device we are on by getting the selected value discussed in the ContentView.swift section. It then takes the arrays of float data obtained through the api calls and converts them to CGFloats using the SwiftUI map function so that we can use them in the custom defined graph function discussed more later. We then invoke the functionality defined in the AppData.swift using the SwiftUI onAppear function, so that the data only appears when it is all loaded.

We had some difficulty finding an up to date SwiftUI graph library that isn't outdated, so we created our own with the help of the (reference 8) youtube series. This helped us normalize the data set we receive from api calls and then graph it using the paths library.

WeatherView.swift

This view is very similar to DetailView.swift but operates on the weather data fetched from the database instead of device data. It starts by declaring a picker which allows the user to select which weather data they would like to view in the graphs (temperature, humidity, pressure, etc.). We then convert all the weather data measurements to CGFloats as we did in DetailView.swift so they can be used in our graphing function. We then load the month and day weather data in by invoking the functionality defined in AppData.swift and display the graphs.

References

1. Source Code
    - https://github.com/masonorsak/CS4485.001
2. Original Dataset
    - https://www.kaggle.com/taranvee/smart-home-dataset-with-weather-information
3. AWS Database Import
    - https://docs.aws.amazon.com/AmazonRDS/latest/UserGuide/MySQL.Procedural.Importing.NonRDSRepl.html
4. Data Analysis (Python)
    - https://colab.research.google.com/drive/1IZNrpIAGgReQQ4c7lW5kGAOhwKwfql2i?usp=sharing
5. API Gateway Creation
    - https://medium.com/@hk_it_er/create-lambda-and-api-gateway-nodejs-aws-serverless-to-rds-mysql-6a75243e61cc
6. SwiftUI - The Complete Developer Course and SwiftUI Bible
    - https://learning.oreilly.com/videos/swiftui-the/9781801070676/
7. User Interface GitHub
    - https://github.com/masonorsak/SeniorProject
8. Graphing in SwiftUI
    - https://www.youtube.com/watch?v=ZtOMxubWklw