IoT システムの設計 レポート

園田純太朗

2025 年 12 月 22 日

1,2 日目

基本的に 3 日目以降の内容と重複しており、コードについても教科書とほぼ同じであるため、省略する。ただし、照度の入力についてのみ解説する。照度の取得は以下のコードでできる。

```
int illuminance = analogRead(A0);
```

これは、照度 E と光ダイオードに流れる光電流 I の間には

$$\log_{10} I = \log_{10} E - 5.5 \Leftrightarrow I = 10^{-5.5} E \tag{1}$$

という関係があるが、下図 1 で光電流は

$$\frac{100\mathrm{k}\Omega}{320\mathrm{k}\Omega} \cdot 1024[V^{-1}] \cdot 1000[\Omega] \cdot E[V] \times \tag{2}$$

という形で出力される。 この式から明らかなように、各定数項が相互に打ち消し合う結果、最終的な出力値は物理的な照度の値とほぼ一致する。この結果は、回路設計段階で適切に定数が設定されていることを示している。したがって、実運用においてソフトウェア側で定数倍を加えるなどのデータ補正を行う必要はなく、センサーの出力値をそのまま照度として扱うことが可能である。

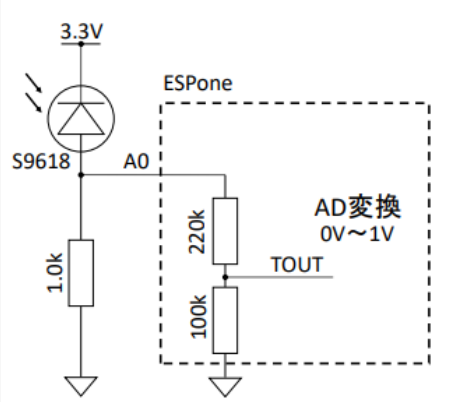

図 1: 照度センサの接続図

3 日目

TCP 接続についても基本的に教科書のとおりである。なお、SSH 接続は linux の ssh コマンドを用いて行った。

次に、これまでのコードを関数群として整理する課題について述べる。 個々の関数の実装は、既存のコードの引数等を調整するのみで可能である。しかし、すべての処理を単一のファイル（.ino)に記述すると、コードの肥大化を招き保守性が低下する。そこで、今後の拡張性を考慮し、機能ごとのファイル分割を行った。

Arduino 言語は C/C++をベースとしているため、分割にあたっては C++の標準的な手法であるヘッダーファイル（.h)とソースファイル（.cpp)による構成を採用した。以下に、スイッチ入力を検知する detectPushSWON 関数に関する実装の抜粋を示す。

ヘッダーファイルでは、#ifndef、#define、#endifを用いたインクルードガードを施し、多重定義を防止している。また、状態管理に用いる変数 int prev_stat については、extern 修飾子を用いて宣言することで、外部ファイルからの参照とグローバルな共有を可能とした。

```
#include "function_list.h"

int prev_stat = HIGH;
bool detectPushSWON() {
  int stat = digitalRead(2);
  if (stat == LOW && prev_stat == HIGH) {
    prev_stat = stat;
    return true;
  } else {
    prev_stat = stat;
    return false;
  }
}
```

リスト 1: function_list.cpp

```
#ifndef _FUNCTION_LIST_H
#define _FUNCTION_LIST_H

extern int prev_stat;
bool detectPushSWON();

#include <Arduino.h>
#endif
```

リスト 2: function_list.h

```
#include "function_list.h"

void setup() {
  Serial.begin(9600);
  pinMode(2, INPUT);
}

void loop() {
  // detectPushSWONのテストコード
  bool isChangedON = detectPushSWON();
  if(isChangedON){
    Serial.println("Pushed!");
  }else{
    Serial.println("not Pushed!");
  }
  delay(1000);
}
```

リスト 3: main.ino

テストケースについては getNPTTime 関数の引数に IP アドレスを入れたときのみ、正しく動作しなかったが、それ以外の単体テストはクリアした。IP アドレスを入力にいれた際に正しく動作しなかったのは入力の際に文字列型で IP アドレスを指定していたからだと思われる。実際、IP アドレスには IPAddress 型という専用の型が用意されているようである。

また、次のように正常に動くテストと、異常系のテストを同時に行うと正常系のテストの方も 0 を返し、正常に動作しなかった。これは正常系のテストが終了し、ポートが解放される前に、異常系のテストが行われてしまい、ポートの初期化に失敗してしまうためであると考えられる。

```
  //WiFiと接続されているときのgetNTPTimeのテストコード
  unsigned long unix_time = getNTPTime("ntp.nict.jp");
  Serial.print("url: ntp.nict.jp :: ");
  Serial.println(unix_time);
  unsigned long unix_time_2 = getNTPTime("www.gutp.jp");
  Serial.print("url: www.gutp.jp :: ");
  Serial.println(unix_time_2);
```

リスト 4: getNTPTime のテストコード

4,5 日目

こちらも基本的に教科書に記述されているコードのうち必要なものを組み合わせたものである。

実装にあたっては、第 3 日目の設計指針と同様、メインプログラムである setup() および loop() 関数内には詳細なロジックを記述せず、抽象度を高める構成とした。具体的には、第 3 日目に作成した関数群に加え、新たに 4 つの関数を定義して処理を委ねている。

ただし、時刻取得を行う getCurrentTime() 関数については、実装上の制約が生じた。関数内で定義した静的ローカル変数 static char str_time[30] を、呼び出し側の loop() 関数へ適切に参照渡し（またはポインタ返し）することが困難であった。このため、今回は設計の簡略化を優先し、時刻取得に関する処理は loop() 関数内に直接実装することとした。

```
//ディスプレイの文字を全消去し、色などの設定をし、カーソルを左上に合わせる。
void initDisplay();
//WiFiに接続する。接続できないときのエラー処理も記述する
bool setupWiFi(const char *ssid, const char *password);
//現在時刻をYYYY-MM-DD HH:MM:SSの形式で返す。
char *getCurrentTime();
//サーバーに接続する。接続できないときのエラー処理も記述する。
bool connectToServer(WiFiClient &client, const char *host, const int port);
//TCP接続し、接続状態を表示する。
void displayCurrentTCPStatus(WiFiClient &client);
```

以下にコードの全容を示す。ただし、エラー処理や print 関数、定数の定義などを省略した疑似コードである。

```
#include "funcList.h"
Adafruit_SSD1306 display(SCREEN_WIDTH, SCREEN_HEIGHT, &Wire, OLED_RESET);

void setup() {
  Serial.begin(115200);

  initDisplay();

  bool isSetupWiFi = setupWiFi(ssid, password);
  if (!isSetupWiFi) {while (1) delay(1000);}

  bool isSyncNTPTime = syncNTPTime(ntp_server);
  if (!isSyncNTPTime) {while (1) delay(1000);}
}

void loop() {
  if (now() / 30 != last_observed_time / 30) {
    int id = getDIPSWStatus();
    char *str_time = getCurrentTime();
    int illuminance = getIlluminance();
    bool isPeopleDetected = getMDStatus();
```

```
    WiFiClient client;
    bool isConnected =connectToServer(client, tcp_server_host, tcp_server_port);
    if (!isConnected) {return;}

    char send_buffer[100] = getCurrentTime();
    client.print(send_buffer);
    displayCurrentTCPStatus(client);

    client.stop();
    last_observed_time = now();
  }

  if (now() / 300 != last_sync_time / 300) {
    syncNTPTime(ntp_server);
    last_sync_time = now();
  }
}
```

また、サーバー側での処理についても簡単に解説する。以下サーバーのコードの抜粋である。

```
server = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server.bind((bind_ip, bind_port))
server.listen(5)

def handle_client(client_socket):
    while True:
        request = client_socket.recv(bufsize)
        try:
            data_str = request.decode("utf-8").strip()
            if data_str.startswith(("GET", "POST" )):#ほかにも列挙している
                client_socket.send(b"ERR\r\n")
                break

            data_fields = data_str.split(',')

            with open("arduino_data.csv", "a", newline='') as f:
                csv.writer(f).writerow(data_fields)

            client_socket.send(b"OK\r\n")

        except UnicodeDecodeError as e:
            client_socket.send(b"ERR\r\n")
        except BrokenPipeError:
            client_socket.send(b"ERR\r\n")
        except Exception as e:
            client_socket.send(b"ERR\r\n")
        finally:
            client_socket.close()

while True:
    client, addr = server.accept()
    client_handler = threading.Thread(target=handle_client, args=(client,))
    client_handler.start()
```

教科書に記述されている部分からの変更点について解説していく。まず、システムの安定性を高めるため、try-except 文による例外処理を導入した。具体的な改良点として、第 10 行目では HTTP リクエスト等の不正なアクセスを検知・拒否するロジックを加え、ブラウザからの不要な干渉を排除した。また、受信したカンマ区切りのデータ(CSV 形式)を一度リストに変換し、再度カンマ区切りに戻

す工程を挟むことで、データ形式の妥当性チェックを実現した。保存時には、with open 文のモードを a(追記モード)に指定し、既存データを損なうことなく CSV ファイルへ保存可能とした。

エラー処理に関しては、開発中に頻発した以下の 2 点について個別の対策を講じた。

- UnicodeDecodeError サーバーとクライアント間の文字コードの不一致に起因する。第 9 行目にて request.decode(“utf-8”) と明示的に指定することで、文字化けおよびエラーを解消した。
- BrokenPipeError サーバーのデータ送信完了前にクライアント側で接続終了(client.stop()）が実行されることで発生した。これに対し、適切な delay 関数を挿入し、通信の完遂を待機させることで接続の安定化を図った。

6 から 10 日目

6 日目から 10 日目においては、センサーデータの収集から機械学習による解析、そして動的な可視化までを一貫して行う基盤を構築した。全体像を図 2 に示す。

データ収集と中継（エッジ〜リレーサーバー） ESP One ボードより、TCP 通信を用いて工学部 2 号館のリレーサーバーへデータを送信した。

- プロトコル変換とセキュア通信（リレー〜バックエンド） リレーサーバーからローカルのバックエンドサーバーへデータを転送する際、セキュリティ制約により標準的な HTTP 通信が制限されていた。これに対応するため、ngrok を用いたトンネリングを導入し、HTTPS 規格による通信経路を確保した。本郷のサーバーから送信された HTTPS リクエストは、ngrok を経由してローカルのバックエンドサーバーへ安全にリダイレクトされる。
- データ解析と蓄積（バックエンド） バックエンドでは、受信データをデータベースに保存すると同時に、機械学習アルゴリズムである Isolation Forest を適用した。これにより、リアルタイムで外れ値の判定を行い、その結果を DB に記録する仕組みを構築した。
- 動的可視化（フロントエンド） フロントエンドでは、DB と常に同期し、新着データを動的に取得するグラフを実装した。10 秒間隔で更新されるデータをリアルタイムに描画し、更新時にはアニメーションを付加することで、視認性の高いユーザーインターフェースを実現した。

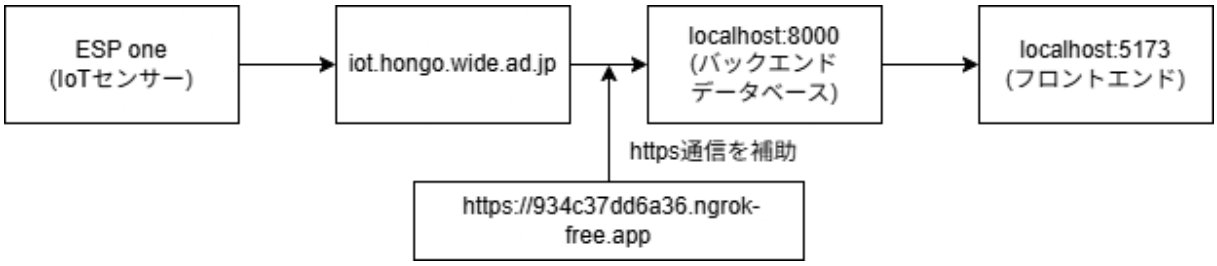

図 2: データの可視化と機械学習の全体像

コード量が多くなってしまったので、バックエンド、フロントエンドの解説については概略のみを示す。 コード全体を参照されたい場合は以下の github を参照していただきたい。また、可視化されたデータのアニメーションについての動画も github の README に記載されているため、そちらを参照していただきたい。注:README に記載された動画の読み込みが遅いので動画を見る際は public ディレクトリを参照していただけると幸いです。

url: https://github.com/junnamuzaurusu/IoTdevice

リレーサーバー

リレーサーバーではサーバー上に CSV ファイルとしてセンサーからのデータを記録することに加え、以下のような json 形式のデータをバックエンドサーバーに送信する役割を追加した。

```
data = {
    "mode": int(data_fields[0]),
```

```
    "date_time": data_fields[1],
    "illuminance": int(data_fields[2]),
    "is_people_detected": bool(int(data_fields[3]))
}
url = f"{"""ngrokによって生成されるURL"""}/raw-data"
response = requests.post(url, json=data)
```

バックエンドサーバー

バックエンドのサーバーは FastAPI と sqlalchemy を使用して構築した。

大まかなファイル構成は以下の通りである。

Backend/

├ algorithm/

├ migrations/

│ └ versions/

├ models/

├ schemas/

├ services/

├ app.py

└ api.py

algorithm ディレクトリでは Isolation Forest を用いてデータを学習し、異常値を検出する関数を作成した。Isolation Forest の実装は scikit-learn の IsolationForest 関数を用いて実装した。

migrations、models ディレクトリは取得したセンサーデータを保存するデータベースの作成のためのディレクトリである。models でデータベースの構造を定義し、alembic という規格を使うことで models から migration ファイルを自動的に生成する。この migration ファイルによってデータベースを作成する。なお、今回のデータベースの ER 図は以下の 図 3 の通りである。

RawData	
PK	id: str(26) NOT NULL
	mode: int (DIPSWの値)
	date_time: datatime(日時)
	illuminance: integer(照度センサーの値)
	is_people_detected: bool (人感センサーの値)
	is_outlier: bool(機械学習で外れ値判定されるか)

図 3: バックエンドの ER 図

schemas ディレクトリは POST や GET などのリクエストが来た時にどのようなデータを返すかの型を定義する役割を果たす。

services ディレクトリはリクエストを受け取った時にどのような処理を行うかを定義する役割を果たす。FastAPI の APIRouter 関数や SQLAlchemy の SessionLocal 関数を用いて、各 URL に来たリクエストに対する応答を記述している。

以上に述べたような関数群を api.py や app.py でまとめることでバックエンドを実装している。今回は他のサーバーも localhost で実装している。このときポートの異なるものは異なるオリジンとして

認識され、CORS によって通信が許可されない。そのため、CORS を許可する設定を app.py に記述する必要があった。

フロントエンドサーバー

フロントエンドは主に React を用いて実装した。 グラフの作成は recharts、タブによる画面の切り替えは headlessui の TabGroup を用いて実装した。

バックエンドからのデータの取得は useEffect 関数内で fetch 関数を非同期で実行することで行っている。

実験の様子

上記のコードをすべて実行し、ESP One ボードを 241 教室前に置くことで、フロントエンドのグラフが動的に更新されることが確認できた。ESP one ボードに表示されている内容を図 4 に、実験現場の様子を図 5 に、各サーバーのログを図 6,7,8,9 に示す。

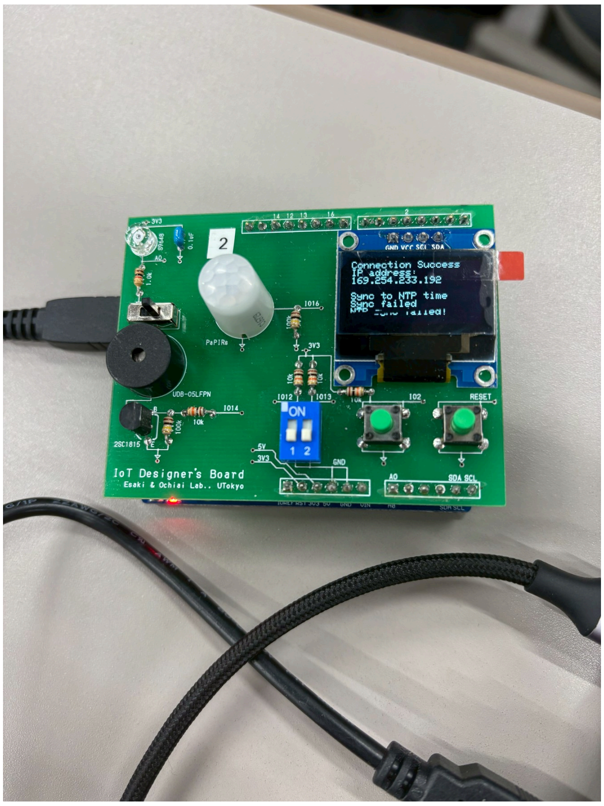

図 4: ESP one ボードのログ

図 5: 実験現場の様子

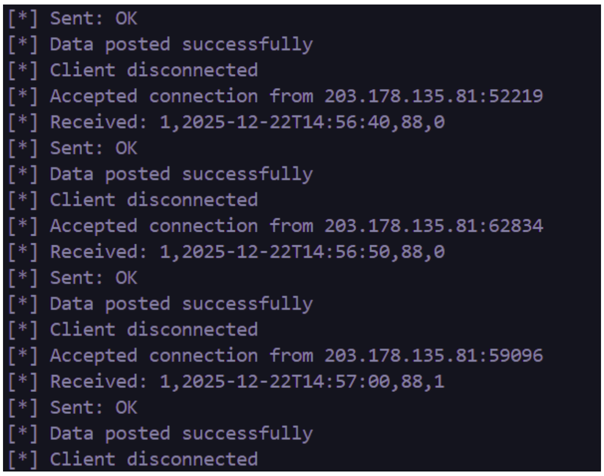

図 6: リレーサーバーのログ

図 7: バックエンドサーバーのログ

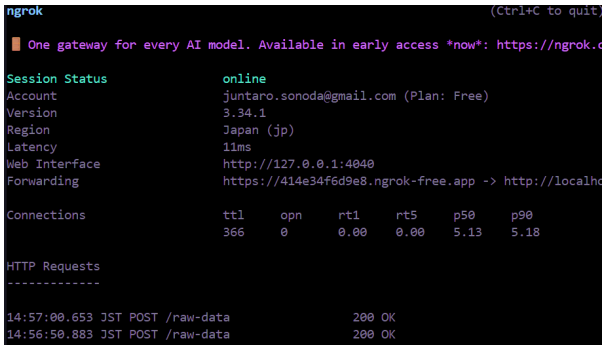

図 8: ngrok のログ

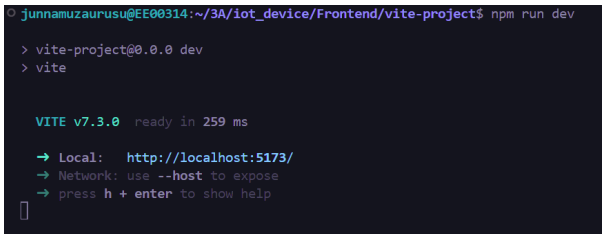

図 9: フロントエンドサーバーのログ

,

結果

上記のコードをすべて実行し、ESP One ボードを 241 教室前に置くことで、フロントエンドのグラフが動的に更新されることが確認できた。その結果を以下の図 10,11 に示す。

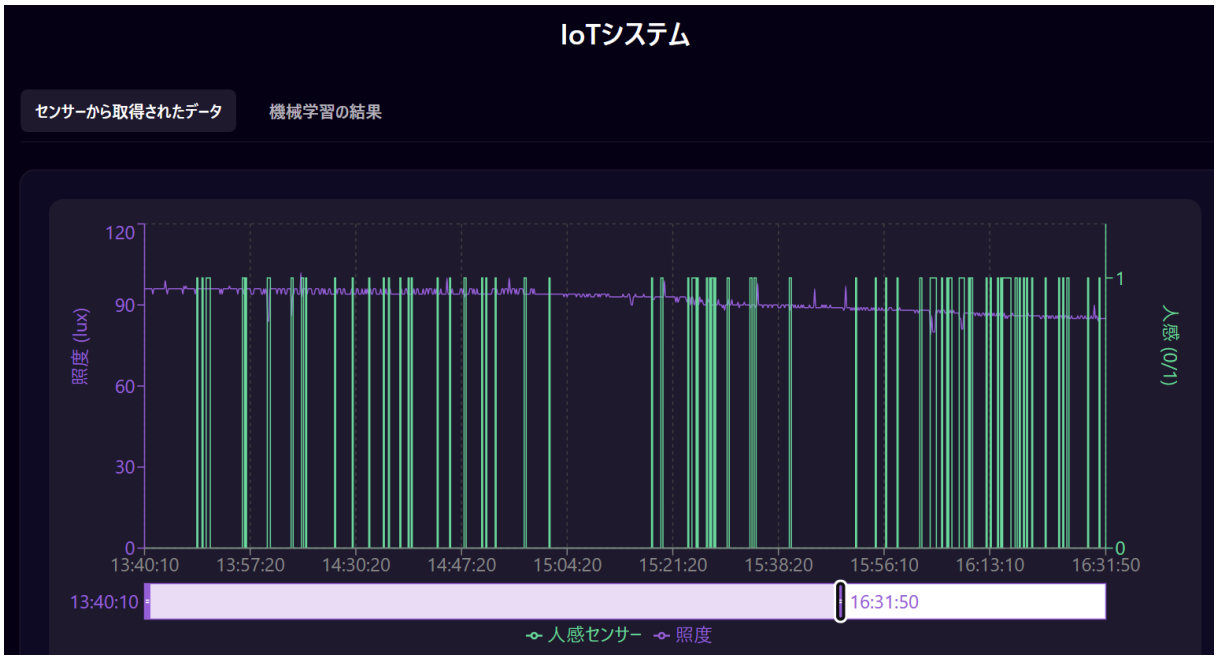

図 10: 観測結果（一日目）

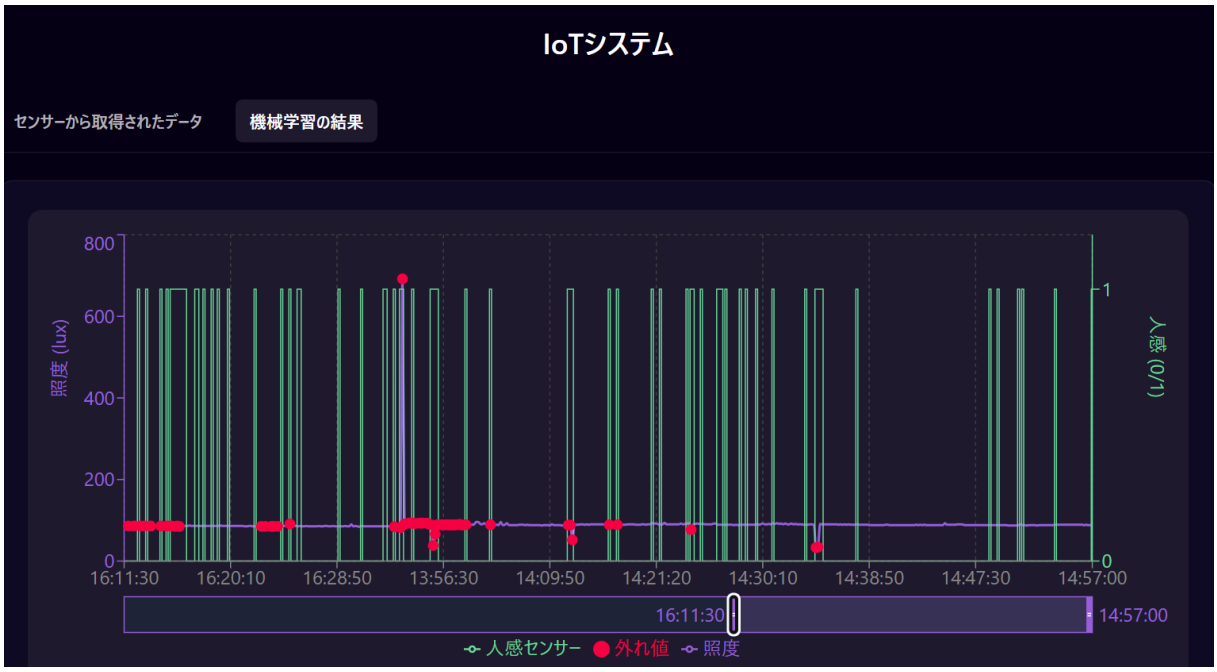

図 11: 観測結果（二日目）

図 10 より、時間の経過とともに照度が低下する傾向が確認された。これは日没に伴う自然な変化であると考えられる。人間は周囲の明るさに合わせて光への感度を動的に調整するため、室内の緩やかな明るさの変化には気づきにくい。一方、ESP one ボード(センサー）は、その微細な変化を正確に捉えている。

しかし、実運用においては、人間が感知できないレベルの細かな数値変化は重要ではない場合が多い。武内徹二(平成 9 年)によれば人間の明るさへの感度は照度の対数におおよそ依存する。そのため、人間の感覚に合わせ、照度データを対数グラフで表現する手法がより実用的であると考えられる。

その他に図 10 でみられる興味深い現象としては 16 時頃に人感センサーが反応したときに照度が急に減少する現象が 2 度確認できたことが挙げられる。これは人の影によって照明の光が遮られたことによると考えられる。

図 11 の機械学習の結果から、外れ値の検出精度について考察する。 まず、視覚的に明らかな外れ値(13:50 頃の約 600lux、14:35 頃の急激な照度低下など)については、モデルでも正しく外れ値として認識できている。

一方で、13:50 頃の照度 100lux 程度の通常範囲内のデータに対しても、誤って外れ値として検出されるケースが確認された。この要因は、外れ値判定の変数に「時刻」を含めてしまったことにあると考えられる。

今回のモデルは、データ取得の瞬間にその妥当性を判定する仕組みである。13:50 頃のデータが前日の同時刻のデータ傾向から乖離していたため、照度値自体は正常であっても「時間的な整合性がない」と判断され、誤検知に繋がったと推測される。

参考文献

[1] https://qiita.com/abek21/items/7739163085899b257cb8

[2] https://fastapi.tiangolo.com/ja/tutorial/cors/#_2

[3] https://headlessui.com/

[4] https://www.jstage.jst.go.jp/article/jieij1980/81/6/81_6_493/_pdf