Lab 6 Report: Automating Things (CSCE 438/838)

Group Members:

1. Khalil Ahmed
2. Apala Pramanik
3. Raoul Nya
4. Audrey Vazzana

Table of Contents

Introduction

In this lab, the goal was to integrate Machine Learning (ML) and Azure IoT services into an end-to-end system. This system involves:

- Building a predictive model using the **Rain in Australia** dataset.
- Creating IoT devices that interact with **Azure IoT Central**.
- Deploying an **Azure Function** to make decisions based on incoming data.

This report documents the steps taken, challenges encountered, and the final outcomes.

Requirements

1. **Data Splitting:** Randomly extract 5% of the dataset as a test set and train the model with the remaining 95%.
2. **IoT Central Application:** Create an IoT Central app with a device template and a **COMMAND** defined.
3. **Data-Sending Device:** Implement a Python-based IoT device that sends data to IoT Central using the 5% dataset.
4. **Command-Listening Device:** Develop another IoT Central device to listen for a **COMMAND** from the Azure function.
5. **Azure Function:** Create a function that predicts rain using the trained model and sends a command if rain is predicted.

Development Process and Results

1. Data Preparation and Model Training

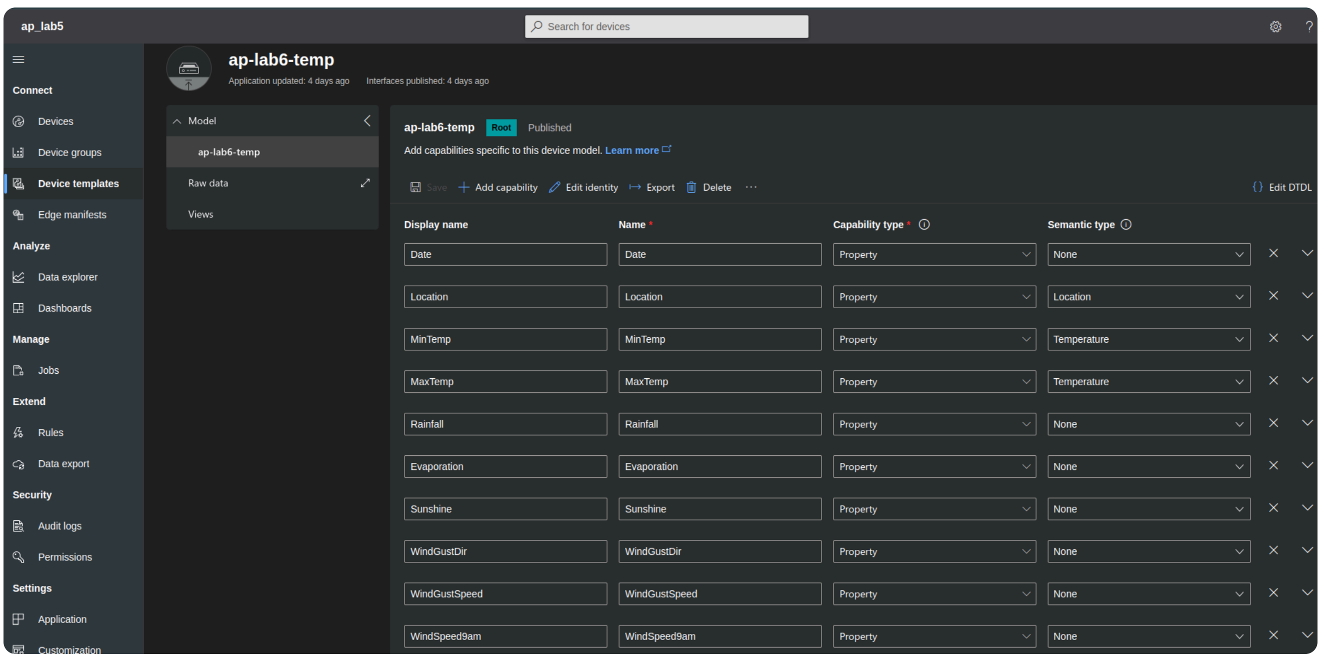
This code demonstrates the end-to-end process of preparing the **Rain in Australia** dataset, training a logistic regression model, and ensuring it can be saved and loaded correctly for deployment. The dataset is initially cleaned by transforming the date column into separate year, month, and day columns and by identifying categorical and numerical features. Missing categorical values are filled using the most frequent value, while numerical columns are handled by treating outliers with the IQR method and imputing missing values with the column mean. Encoding is applied to convert categorical data into numeric form for model compatibility.

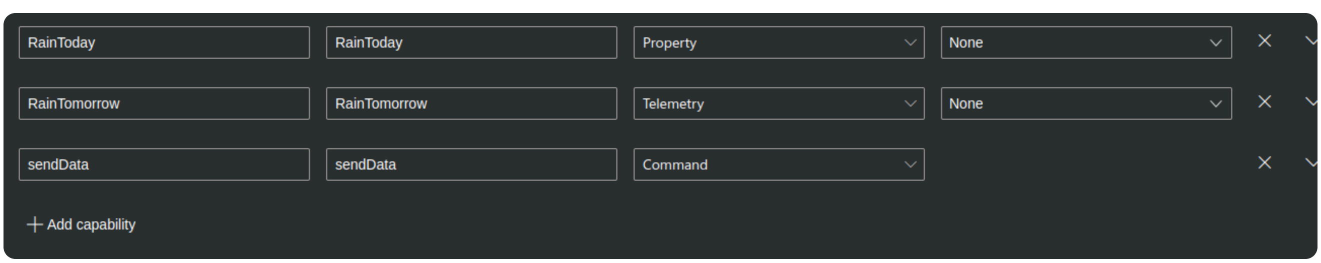
```
# Split off 5% of the data for validation before processing
rain_train, rain_validation =
train_test_split(rain, test_size=0.05, random_state=42)

# Save the 5% validation data into a CSV file
rain_validation.to_csv('rain_validation.csv', index=False)

# Split remaining data into features (X) and target (y)
X = rain_train.drop(['RainTomorrow'], axis=1)
y = rain_train['RainTomorrow']

# Split training and test sets (from the remaining 95%)
X_train, X_test, y_train, y_test =
train_test_split(X, y, test_size=0.2, random_state=0)
```

A key modification to this code, in accordance with the assignment requirements, is the extraction of **5% of the dataset for validation** before processing. This subset is saved separately in a CSV file for use later in IoT Central simulation. The remaining 95% of the data is split into training and test sets, with a standard scaler applied to prevent data leakage. A logistic regression model is trained on the scaled training data, and the model's accuracy is evaluated on the test set using metrics like accuracy score and classification report. The model is saved using `pickle` for future use in Azure IoT integration, ensuring consistency between the original and reloaded model predictions.

2. IoT Central Application Setup

A screenshot here showing the creation of the **device template** with a **COMMAND** defined.





3. Python-Based IoT Device to Send Data

Code Explanation: Sending Telemetry and Handling Commands with IoT Central

This code demonstrates how to connect a Python-based device to **Azure IoT Central**, send telemetry data from a pre-split dataset, and listen for commands from the IoT Central platform. Additionally, the code shows how a listening device can receive commands and respond.

Step-by-Step Explanation

1. Importing Required Libraries

The following libraries are used to manage time, load datasets, and interact with Azure IoT Central:

```
import time
import pandas as pd
from iotc.models import Command
from iotc import IoTCClient, IOTCConnectType, IOTCEvents
```

- **`time`** : Adds delays between telemetry transmissions.
- **`pandas`** : Used to load the pre-split 5% dataset for telemetry data.
- **`iotc`** : A library to handle device connections and events with IoT Central.

2. Setting IoT Central Device Credentials

The credentials for the device include **Scope ID**, **Device ID**, and **Device Key**, which are used to authenticate the device with IoT Central.

```
# IoT Central device credentials
scope_id = '0ne00D18E8C'
device_id = '1oqxy244di7'
device_key = 'm3k5AQsOKu9PRApxzXOFFEDaJSozi/CeG7dSEIQPnww='
```

3. Loading the 5% Validation Dataset

This dataset, saved as `rain_validation.csv` , contains weather observations that will be sent as telemetry data.

```
# Load the 5% split dataset (rain_validation.csv)
rain_validation = pd.read_csv("rain_validation.csv")
```

4. Defining a Function to Handle Incoming Commands

This function prints the name of any incoming command from IoT Central and sends a reply to confirm receipt.

```
# Function to handle incoming commands
def on_commands(command: Command):
    print(f"{command.name} command was sent")
    command.reply()
```

5. Establishing Connection to IoT Central

The `IoTCClient` object is initialized with the device's credentials, and a connection is established. We also register the **`on_commands`** handler to listen for commands.

```
# Connect to IoT Central
iotc = IoTCClient(
    device_id,
    scope_id,
    IOTCConnectType.IOTC_CONNECT_DEVICE_KEY,
    device_key
)

iotc.connect()
iotc.on(IOTCEvents.IOTC_COMMAND, on_commands)
```

6. Sending Telemetry Data to IoT Central

The code iterates over each row in the **5% split dataset**, formats the data as a dictionary, and sends it to IoT Central as telemetry. A short delay is added between transmissions.

```
# Sending telemetry data from the 5% split dataset
for index, row in rain_validation.iterrows():
    if iotc.is_connected():
        TEL = {
            'Location': row['Location'],
            'MinTemp': row['MinTemp'],
            'MaxTemp': row['MaxTemp'],
            'Rainfall': row['Rainfall'],
            'Evaporation': row['Evaporation'],
            'Sunshine': row['Sunshine'],
            'WindGustDir': row['WindGustDir'],
            'WindGustSpeed': row['WindGustSpeed'],
            'WindDir9am': row['WindDir9am'],
            'WindDir3pm': row['WindDir3pm'],
            'WindSpeed9am': row['WindSpeed9am'],
            'WindSpeed3pm': row['WindSpeed3pm'],
            'Humidity9am': row['Humidity9am'],
            'Humidity3pm': row['Humidity3pm'],
            'Pressure9am': row['Pressure9am'],
            'Pressure3pm': row['Pressure3pm'],
            'Cloud9am': row['Cloud9am'],
            'Cloud3pm': row['Cloud3pm'],
            'Temp9am': row['Temp9am'],
            'Temp3pm': row['Temp3pm'],
            'RainToday': row['RainToday']
        }

        # Send telemetry to IoT Central
        iotc.send_telemetry(TEL)
        print(f"Telemetry sent: {TEL}")

        # Sleep for a short duration before sending the next row of data
        time.sleep(10)  # Adjust as needed
```

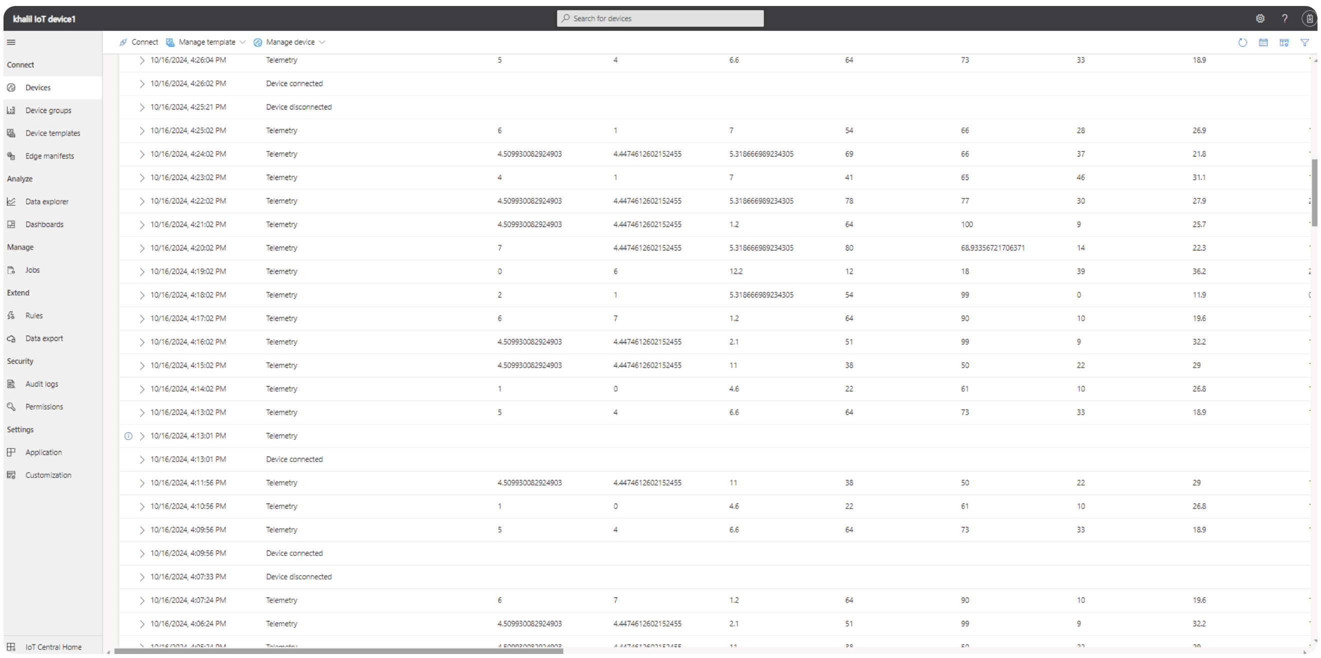
Explanation Summary

This code establishes a connection between a Python-based IoT device and Azure IoT Central. The device sends weather telemetry data from a 5% validation dataset and listens for incoming commands from IoT Central. If a command is received, it prints the command name and sends a confirmation reply.

TELEMETRY DATA SENT TO IOT CENTRAL

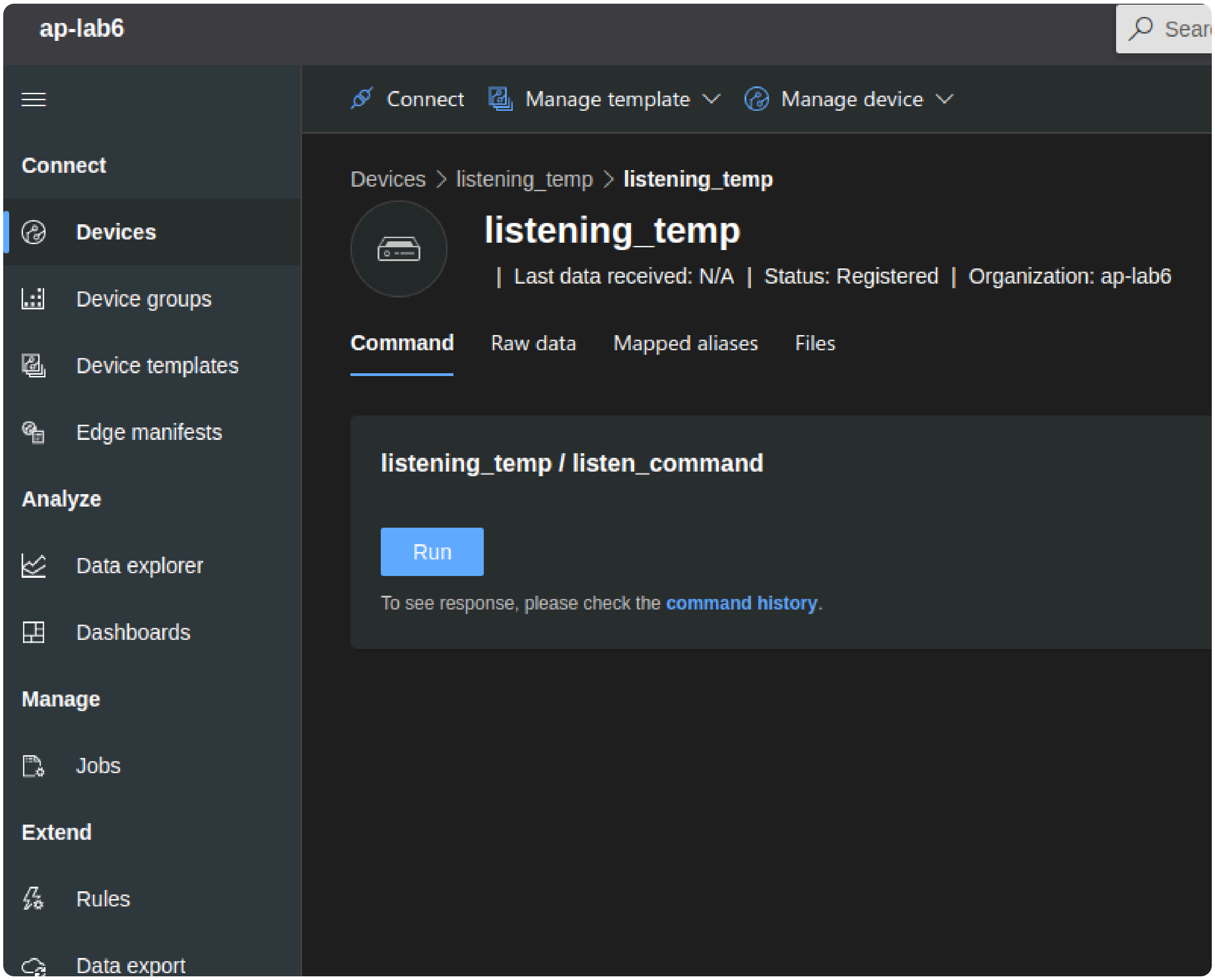
```
Telemetry sent: {'Location': np.float64(33.0), 'MinTemp': np.float64(15.0), 'MaxTemp': np.float64(18.9), 'Rainfall': np.float64(2.0), 'Evaporation': np.float64(6.6), 'Sunshine': np.float64(8.0), 'WindGustDir': np.float64(4.0), 'WindGustSpeed': np.float64(54.0), 'WindDir9am': np.float64(4.0), 'WindDir3pm': np.float64(11.0), 'WindSpeed9am': np.float64(24.0), 'WindSpeed3pm': np.float64(31.0), 'Humidity9am': np.float64(73.0), 'Humidity3pm': np.float64(64.0), 'Pressure9am': np.float64(1005.2), 'Pressure3pm': np.float64(1003.7), 'Cloud9am': np.float64(4.0), 'Cloud3pm': np.float64(5.0), 'Temp9am': np.float64(17.3), 'Temp3pm': np.float64(17.6), 'RainToday': np.float64(1.0)}
Sending telemetry message: {'Location': np.float64(10.0), 'MinTemp': np.float64(13.1), 'MaxTemp': np.float64(26.8), 'Rainfall': np.float64(0.0), 'Evaporation': np.float64(4.6), 'Sunshine': np.float64(10.9), 'WindGustDir': np.float64(0.0), 'WindGustSpeed': np.float64(39.83779225870396), 'WindDir9am': np.float64(0.0), 'WindDir3pm': np.float64(1.0), 'WindSpeed9am': np.float64(22.0), 'WindSpeed3pm': np.float64(15.0), 'Humidity9am': np.float64(61.0), 'Humidity3pm': np.float64(22.0), 'Pressure9am': np.float64(1013.0), 'Pressure3pm': np.float64(1009.0), 'Cloud9am': np.float64(0.0), 'Cloud3pm': np.float64(1.0), 'Temp9am': np.float64(16.9), 'Temp3pm': np.float64(25.9), 'RainToday': np.float64(0.0)}

Telemetry sent: {'Location': np.float64(10.0), 'MinTemp': np.float64(13.1), 'MaxTemp': np.float64(26.8), 'Rainfall': np.float64(0.0), 'Evaporation': np.float64(4.6), 'Sunshine': np.float64(10.9), 'WindGustDir': np.float64(0.0), 'WindGustSpeed': np.float64(39.83779225870396), 'WindDir9am': np.float64(0.0), 'WindDir3pm': np.float64(1.0), 'WindSpeed9am': np.float64(22.0), 'WindSpeed3pm': np.float64(15.0), 'Humidity9am': np.float64(61.0), 'Humidity3pm': np.float64(22.0), 'Pressure9am': np.float64(1013.0), 'Pressure3pm': np.float64(1009.0), 'Cloud9am': np.float64(0.0), 'Cloud3pm': np.float64(1.0), 'Temp9am': np.float64(16.9), 'Temp3pm': np.float64(25.9), 'RainToday': np.float64(0.0)}
```

TELEMETRY DATA RECEIVED AT IOT CENTRAL

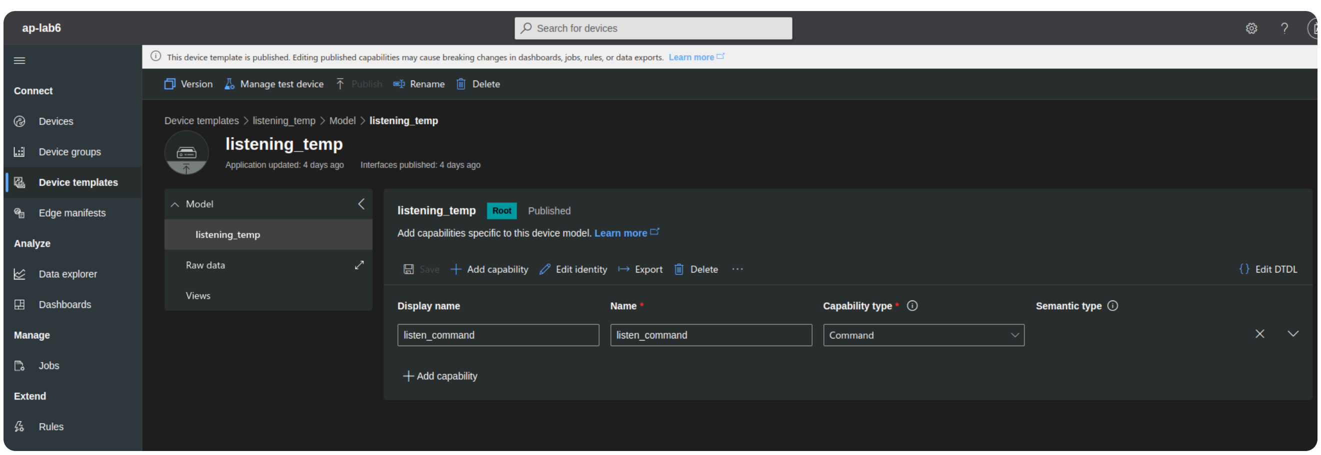


4. Command-Listening Device

LISTENING DEVICE CREATED



LISTENING DEVICE TEMPLATE



5. Azure Function to Process Data and Send Commands

Code Explanation: Azure Function for Weather Prediction and Device Command

This code defines an **Azure Function** that loads a pre-trained machine learning model, processes incoming HTTP requests, makes weather predictions, and sends commands to a listening device if rain is predicted. It showcases the integration of Azure Functions with machine learning and IoT Central through RESTful APIs.

Step-by-Step Explanation

1. Importing Required Libraries

The necessary libraries for Azure Functions, machine learning model loading, logging, and HTTP requests are imported.

```
import azure.functions as func
import joblib
import logging
import json
import requests
```

- **`azure.functions`** : Used to create Azure Functions and handle HTTP requests.
- **`joblib`** : Loads the pre-trained machine learning model from disk.
- **`logging`** : Logs important information, warnings, and errors for monitoring.
- **`json`** : Handles JSON data formats for API requests.
- **`requests`** : Sends HTTP POST requests to the listening IoT device.

2. Initializing the Azure Function App and Loading the Model

The function app is configured with **function-level authorization** for secured access. The machine learning model, previously trained and saved, is loaded using `joblib` .

```
app = func.FunctionApp(http_auth_level=func.AuthLevel.FUNCTION)

# Load the pre-trained model
model = joblib.load('iot_model_ap')
```

3. Creating the HTTP Trigger Function

The `http_trigger_lab6` function is triggered by an HTTP request. It logs the request and extracts relevant weather data features for prediction.

```
@app.route(route="http_trigger_lab6", auth_level=func.AuthLevel.FUNCTION)
def http_trigger_lab6(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    try:
        # Get request body data
        req_body = req.get_json()
        logging.info(f"Request body: {req_body}")
```

4. Extracting Features for Prediction

The function extracts weather-related features from the request body. These features are expected to match the input format required by the pre-trained model.

```
        # Extract the necessary features for prediction
        features = [
            'Location': req_body.get('Location'),
            'MinTemp': req_body.get('MinTemp'),
            'MaxTemp': req_body.get('MaxTemp'),
            'Rainfall': req_body.get('Rainfall'),
            'Evaporation': req_body.get('Evaporation'),
            'Sunshine': req_body.get('Sunshine'),
            'WindGustDir': req_body.get('WindGustDir'),
            'WindGustSpeed': req_body.get('WindGustSpeed'),
            'WindDir9am': req_body.get('WindDir9am'),
            'WindDir3pm': req_body.get('WindDir3pm'),
            'WindSpeed9am': req_body.get('WindSpeed9am'),
            'WindSpeed3pm': req_body.get('WindSpeed3pm'),
            'Humidity9am': req_body.get('Humidity9am'),
            'Humidity3pm': req_body.get('Humidity3pm'),
            'Pressure9am': req_body.get('Pressure9am'),
            'Pressure3pm': req_body.get('Pressure3pm'),
            'Cloud9am': req_body.get('Cloud9am'),
            'Cloud3pm': req_body.get('Cloud3pm'),
            'Temp9am': req_body.get('Temp9am'),
            'Temp3pm': req_body.get('Temp3pm'),
            'RainToday': req_body.get('RainToday')
        ]


        features = [features]  # Reshape into 2D array if necessary
```

5. Making a Prediction Using the Model

The features are passed to the loaded model to predict whether it will rain tomorrow. The model outputs `1` if rain is predicted and `0` otherwise.

```
        # Make a prediction using the loaded model
        prediction = model.predict(features)
        logging.info(f"Model prediction: {prediction}")
```

6. Sending a Command if Rain is Predicted

If rain is predicted, the function sends a **command** to the listening IoT device through an HTTP POST request. The command notifies the device to activate a rain alert.

```
        if prediction[0] == 1:  # If rain is predicted
            logging.info("Rain predicted for
                         tomorrow. Sending command to device...")

            # URL to send the command to the listening device
            device_command_url = 'https://command-url'

            # Command data to be sent
            command_data = {
                "command": "TurnOnRainAlert",
                "message": "Rain Predicted Tomorrow!"
            }
            headers = {'Content-Type': 'application/json'}

            # Send the command to the device
            response = requests.post(device_command_url,
                                     data=json.dumps(command_data),
                                     headers=headers)

            if response.status_code == 200:
                logging.info(f"Command sent to device successfully:
                             {response.text}")
                return func.HttpResponse(f"Rain predicted, command
                                         sent successfully: {response.text}'
                                         status_code=200)
            else:
                logging.error(f"Failed
                              to send command to device: {response.text}")

                return func.HttpResponse(f"Rain predicted, but command
                                         failed: {response.text}",
                                         status_code=500)
```

7. Handling the Case When No Rain is Predicted

If the model predicts no rain, a simple HTTP response is returned.

```
        else:
            logging.info("No rain predicted for tomorrow.")
            return func.HttpResponse("No rain predicted
                                     for tomorrow.", status_code=200)
```

8. Exception Handling

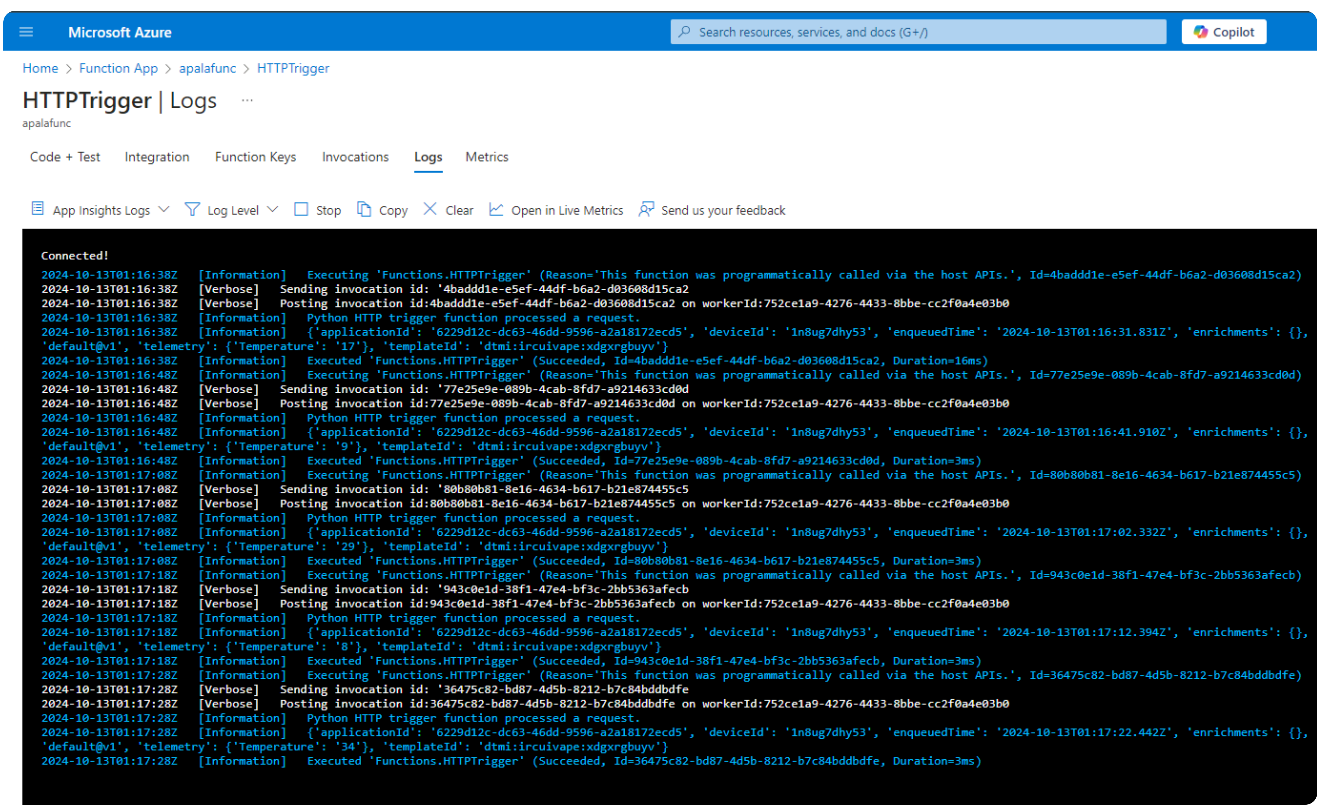
Any errors during the function's execution are caught, logged, and returned with a `500` status code.

```
    except Exception as e:
        logging.error(f"Error processing the request: {str(e)}")
        return func.HttpResponse(f"Error processing the
                                  request: {str(e)}", status_code=500)
```

The screenshot of logs from Azure Function indicating the successful run

```
PS C:\Users\kahmad2\desktop\lab6n> python listner.py
Syncing property '$version'
Listening for commands...
Received command: TurnOnRainAlert
Rain Predicted Tomorrow!
```

	PredictedRainTomorrow	RainTomorrow
0	Yes	1
1	Yes	0
2	Yes	0
3	Yes	0
4	Yes	0



Development Process Challenges and Troubleshooting

1. **Challenges:**
    - **API Authentication Errors:** During initial testing, the IoT Central device failed to connect due to an incorrect device key or expired API token. The device returned 401 Unauthorized errors, which indicated an issue with credentials.
    - **Data Format Mismatch:** The machine learning model expected input features in a specific 2D array format, but the incoming data from IoT Central requests was not properly reshaped, resulting in prediction errors.
    - **Connectivity Issues:** There were intermittent connection failures between the device and IoT Central, likely due to network instability.
    - **Azure Function Deployment Errors:** While deploying the Azure Function, issues with missing dependencies were encountered, causing the function to fail during execution.
    - **Command Delivery Failures:** The device occasionally failed to receive commands from the Azure function due to incorrect API URLs or device unavailability.

2. **Troubleshooting Steps:**
    - **API Authentication:** Double-checked the scope ID, device ID, and device key, ensuring that the correct credentials were being used. Generated a new API token to replace the expired one.
    - **Data Reshaping:** Added a step in the Azure Function to convert the features into a 2D array to match the model's expected input format, which resolved the prediction errors.
    - **Network Connectivity:** Monitored network conditions and added error handling with retries to ensure the device reconnected to IoT Central in case of disconnections.
    - **Azure Function Dependencies:** Updated the `requirements.txt` file in the Azure Function project to include all necessary libraries (e.g., `joblib`, `requests`) and redeployed the function.
    - **Command URL Corrections:** Verified the API URL and device ID used in the command requests. Added logging to track API responses, which helped identify and fix endpoint issues.

- **Device Availability:** Ensured that the listening IoT device was online and correctly registered in IoT Central before sending commands, preventing delivery failures.

References

- Azure IoT Central Documentation (https://learn.microsoft.com/en-us/rest/api/iotcentral/)
- Rain in Australia Dataset on Kaggle (https://www.kaggle.com/datasets/jsphyg/weather-dataset-rattle-package)
- Azure Functions Documentation (https://learn.microsoft.com/en-us/azure/azure-functions/)

Appendix

CODE FILE FOR ML PROCESSING

```
# Load all the libraries
import pandas as pd
import numpy as np
import pickle

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report
from sklearn.linear_model import LogisticRegression

# Load the data
dataset = 'weatherAUS.csv'
rain = pd.read_csv(dataset)

# Reduce the cardinality of date by splitting it into year month and
rain['Date'] = pd.to_datetime(rain['Date'])
rain['year'] = rain['Date'].dt.year
rain['month'] = rain['Date'].dt.month
rain['day'] = rain['Date'].dt.day
rain.drop('Date', axis = 1, inplace = True)

# Classify feature type
categorical_features = [
    column_name
    for column_name in rain.columns
    if rain[column_name].dtype == 'O'
]

numerical_features = [
    column_name
    for column_name in rain.columns
    if rain[column_name].dtype != 'O'
]

# Fill missing categorical values
# with the highest frequency value in the column
categorical_features_with_null = [
    feature
    for feature in categorical_features
    if rain[feature].isnull().sum()
]

for each_feature in categorical_features_with_null:
    mode_val = rain[each_feature].mode()[0]
    rain[each_feature].fillna(mode_val,inplace=True)

# Before treating the missing values
# in numerical values, treat the outliers
features_with_outliers = [
    'MinTemp',
```

```
    'MaxTemp',
    'Rainfall',
    'Evaporation',
    'WindGustSpeed',
    'WindSpeed9am',
    'WindSpeed3pm',
    'Humidity9am',
    'Pressure9am',
    'Pressure3pm',
    'Temp9am',
    'Temp3pm'
]

for feature in features_with_outliers:
    q1 = rain[feature].quantile(0.25)
    q3 = rain[feature].quantile(0.75)
    IQR = q3 - q1
    lower_limit = q1 - (IQR * 1.5)
    upper_limit = q3 + (IQR * 1.5)
    rain.loc[rain[feature]<lower_limit,feature] = lower_limit
    rain.loc[rain[feature]>upper_limit,feature] = upper_limit

# Treat missing values in numerical features
numerical_features_with_null = [
    feature
    for feature in numerical_features
    if rain[feature].isnull().sum()
]

for feature in numerical_features_with_null:
    mean_value = rain[feature].mean()
    rain[feature].fillna(mean_value,inplace=True)


# Encoding categorical values as integers
direction_encoding = {
    'W': 0, 'WNW': 1, 'WSW': 2, 'NE': 3, 'NNW': 4,
    'N': 5, 'NNE': 6, 'SW': 7, 'ENE': 8, 'SSE': 9,
    'S': 10, 'NW': 11, 'SE': 12, 'ESE': 13, 'E': 14, 'SSW': 15
}

location_encoding = {
    'Albury': 0,
    'BadgerysCreek': 1,
    'Cobar': 2,
    'CoffsHarbour': 3,
    'Moree': 4,
    'Newcastle': 5,
    'NorahHead': 6,
    'NorfolkIsland': 7
```

```
    'NorfolkIsland': 7,
    'Penrith': 8,
    'Richmond': 9,
    'Sydney': 10,
    'SydneyAirport': 11,
    'WaggaWagga': 12,
    'Williamtown': 13,
    'Wollongong': 14,
    'Canberra': 15,
    'Tuggeranong': 16,
    'MountGinini': 17,
    'Ballarat': 18,
    'Bendigo': 19,
    'Sale': 20,
    'MelbourneAirport': 21,
    'Melbourne': 22,
    'Mildura': 23,
    'Nhil': 24,
    'Portland': 25,
    'Watsonia': 26,
    'Dartmoor': 27,
    'Brisbane': 28,
    'Cairns': 29,
    'GoldCoast': 30,
    'Townsville': 31,
    'Adelaide': 32,
    'MountGambier': 33,
    'Nuriootpa': 34,
    'Woomera': 35,
    'Albany': 36,
    'Witchcliffe': 37,
    'PearceRAAF': 38,
    'PerthAirport': 39,
    'Perth': 40,
    'SalmonGums': 41,
    'Walpole': 42,
    'Hobart': 43,
    'Launceston': 44,
    'AliceSprings': 45,
    'Darwin': 46,
    'Katherine': 47,
    'Uluru': 48
}
boolean_encoding = {'No': 0, 'Yes': 1}


rain['RainToday'].replace(boolean_encoding, inplace = True)
rain['RainTomorrow'].replace(boolean_encoding, inplace = True)
rain['WindGustDir'].replace(direction_encoding,inplace = True)
rain['WindDir9am'].replace(direction_encoding,inplace = True)
rain['WindDir3pm'].replace(direction_encoding,inplace = True)
```

```
rain['WindDir3pm'].replace(direction_encoding,inplace = True)
rain['Location'].replace(location_encoding, inplace = True)

# See the distribution of the dataset
print(rain['RainTomorrow'].value_counts())

# Split off 5% of the data for validation before processing
rain_train, rain_validation = train_test_split(rain,
                              test_size=0.05, random_state=42)

# Save the 5% validation data into a CSV file
rain_validation.to_csv('rain_validation.csv', index=False)

# Split remaining data into features (X) and target (y)
X = rain_train.drop(['RainTomorrow'], axis=1)
y = rain_train['RainTomorrow']

# Split training and test sets (from the remaining 95%)
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                    test_size=0.2, random_state=0)

# Scale input using just the training set to prevent bias
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)

# Train the model
classifier_logreg = LogisticRegression(solver='liblinear', random_sta
classifier_logreg.fit(X_train, y_train)

# Test the model accuracy
y_pred = classifier_logreg.predict(X_test)

print(f"Accuracy Score: {accuracy_score(y_test,y_pred)}")
print("Classifcation report", classification_report(y_test,y_pred))

with open("iot_model_ap", "wb") as model_file:
 pickle.dump(classifier_logreg, model_file)

with open("iot_model_ap", "rb") as model_file:
 model = pickle.load(model_file)
y_pred_new = classifier_logreg.predict(X_test)

print("Output of loaded model is same
      as original model ?", all(y_pred == y_pred_new))
```

CODE FILE FOR SENDING DATA TO IOT CENTRAL

```
import time
import pandas as pd
from iotc.models import Command
from iotc import IoTCClient, IOTCConnectType, IOTCEvents

# IoT Central device credentials
scope_id = '0ne00D18E8C'
device_id = '1oqxy244di7'
device_key = 'm3k5AQsOKu9PRApxzXOFFEDaJSozi/CeG7dSEIQPnww='

# Load the 5% split dataset (rain_validation.csv)
rain_validation = pd.read_csv("rain_validation.csv")

# Function to handle incoming commands
def on_commands(command: Command):
    print(f"{command.name} command was sent")
    command.reply()

# Connect to IoT Central
iotc = IoTCClient(
    device_id,
    scope_id,
    IOTCConnectType.IOTC_CONNECT_DEVICE_KEY,
    device_key
)

iotc.connect()
iotc.on(IOTCEvents.IOTC_COMMAND, on_commands)

# Sending telemetry data from the 5% split dataset
for index, row in rain_validation.iterrows():
    if iotc.is_connected():
        TEL = {

            'Location': row['Location'],
            'MinTemp': row['MinTemp'],
            'MaxTemp': row['MaxTemp'],
            'Rainfall': row['Rainfall'],
            'Evaporation': row['Evaporation'],
            'Sunshine': row['Sunshine'],
            'WindGustDir': row['WindGustDir'],
            'WindGustSpeed': row['WindGustSpeed'],
            'WindDir9am': row['WindDir9am'],
            'WindDir3pm': row['WindDir3pm'],
            'WindSpeed9am': row['WindSpeed9am'],
            'WindSpeed3pm': row['WindSpeed3pm'],
            'Humidity9am': row['Humidity9am'],
            'Humidity3pm': row['Humidity3pm'],
            'Pressure9am': row['Pressure9am'],
            'Pressure3pm': row['Pressure3pm'],
```

```
                'Cloud9am': row['Cloud9am'],
                'Cloud3pm': row['Cloud3pm'],
                'Temp9am': row['Temp9am'],
                'Temp3pm': row['Temp3pm'],
                'RainToday': row['RainToday']
            }

            # Send telemetry to IoT Central
            iotc.send_telemetry(TEL)
            print(f"Telemetry sent: {TEL}")

            # Sleep for a short duration before sending the next row of da
            time.sleep(10)  # Adjust as needed (10 seconds per data entry)



"""
listening device

scope: 0ne00D18E8C
dev id: 13rio316qdu
primary : 6VdOgcQq5PagwDmn/MOXHtTiNhQzrKYZmGvrDfaZnaU=


"""
```

CODE FILE FOR PREDICTIONA AND SENDING COMMAND

```
import azure.functions as func
import joblib
import logging
import json
import requests

app = func.FunctionApp(http_auth_level=func.AuthLevel.FUNCTION)

model = joblib.load('iot_model_ap')



@app.route(route="http_trigger_lab6", auth_level=func.AuthLevel.FUNCTION)
def http_trigger_lab6(req: func.HttpRequest) -> func.HttpResponse:
    logging.info('Python HTTP trigger function processed a request.')

    try:
        # Get request body data
        req_body = req.get_json()
        logging.info(f"Request body: {req_body}")

        # Extract the necessary features for prediction
        features = [
             'Location': req_body.get('Location'),
            'MinTemp': req_body.get('MinTemp'),
            'MaxTemp': req_body.get('MaxTemp'),
            'Rainfall': req_body.get('Rainfall'),
            'Evaporation': req_body.get('Evaporation'),
            'Sunshine': req_body.get('Sunshine'),
            'WindGustDir': req_body.get('WindGustDir'),
            'WindGustSpeed': req_body.get('WindGustSpeed'),
            'WindDir9am': req_body.get('WindDir9am'),
            'WindDir3pm': req_body.get('WindDir3pm'),
            'WindSpeed9am': req_body.get('WindSpeed9am'),
            'WindSpeed3pm': req_body.get('WindSpeed3pm'),
            'Humidity9am': req_body.get('Humidity9am'),
            'Humidity3pm': req_body.get('Humidity3pm'),
            'Pressure9am': req_body.get('Pressure9am'),
            'Pressure3pm': req_body.get('Pressure3pm'),
            'Cloud9am': req_body.get('Cloud9am'),
            'Cloud3pm': req_body.get('Cloud3pm'),
            'Temp9am': req_body.get('Temp9am'),
            'Temp3pm': req_body.get('Temp3pm'),
            'RainToday': req_body.get('RainToday')
        ]

        # Convert features to the
```

```
        # correct format (e.g., NumPy array or DataFrame)
        # Assuming the model expects a 2D array input
        features = [features]  # Reshape into 2D array if necessary

        # Make a prediction using the loaded model
        prediction = model.predict(features)

        logging.info(f"Model prediction: {prediction}")

        # Check if the model predicts rain tomorrow
        # (assuming 'RainTomorrow' is 1 for rain, 0 for no rain)
        if prediction[0] == 1:  # Modify based on how your model encodes ra
            logging.info("Rain Predicted Tomorrow!"")

            # Send a command to the
            #  listening device
            #   (Assume the device is reachable via an API)
            device_command_url = 'https://command-url'
            command_data = {
                "command": "TurnOnRainAlert",
                "message": "Rain Predicted Tomorrow!"
            }
            headers = {'Content-Type': 'application/json'}

            response = requests.post(device_command_url,
                                     data=json.dumps(command_data),
                                     headers=headers)

            if response.status_code == 200:
                logging.info(f"Command sent
                             to device successfully: {response.text}")
                return func.HttpResponse(f"Rain predicted,
                                         command sent to device
                                         successfully: {response.text}""
                                         status_code=200)
            else:
                logging.error(f"Failed to send command
                              to device: {response.text}")
                return func.HttpResponse(f"Rain predicted,
                                         but failed to
                                         send command to device:
                                         {response.text}",
                                         status_code=500)
        else:
            logging.info("No rain predicted for tomorrow.")

            return func.HttpResponse("No rain predicted for tomorrow.",
                                     status_code=200)
```

```
    except Exception as e:
        logging.error(f"Error processing
                      the request: {str(e)}")
        return func.HttpResponse(f"Error
                                 processing the request: {str(e)}",
                                 status_code=500)
```