CACSC19: AI Hardware and Tools Workshop

Name:Chaitanya Yadav
Roll No: 2022UCA1849
Class: CSAI-1

Problem Statement

Traditional irrigation methods often waste water by following fixed schedules without accounting for real-time environmental conditions. This can lead to over-irrigation (wasting water) or under-irrigation (harming crops), especially in regions facing climate uncertainty or water scarcity.

Goal:
Develop a smart irrigation scheduling system that:

- Predicts soil moisture conditions using environmental sensor data.
- Recommends when and how much to irrigate.
- Supports deployment on low-power microcontrollers using TinyML.

Data Source

The dataset titled `Irrigation Scheduling.csv` contains various sensor readings including:

- Temperature
- Humidity
- Soil moisture
- Rainfall
- Altitude
- Other geographical features

Target Variable: `class`
Represents irrigation need with 4 labels:

- "Very Dry"
- "Dry"
- "Wet"
- "Very Wet"

Tech Stack & Tools Used

- Programming Languages: Python, R
- ML/DL Frameworks: PyTorch, scikit-learn
- Model Optimization: Knowledge Distillation (Teacher-Student architecture)
- Model Format: ONNX
- Visualization Tools: Power BI, Streamlit
- Data Streaming & Processing: PySpark
- DevOps: Docker
- Microcontroller Platform: ESP32 (simulated on Wokwi)

Project Workflow

Data Preprocessing & Analysis

- Initial exploration and cleaning are done using R.
- Scikit-learn is used for encoding and standardizing inputs.

Model Training

- A Teacher Model (deep neural network) is trained for high accuracy.
- A Student Model (lightweight DNN) is trained using knowledge distillation.

Model Export

- The student model is exported to ONNX format for hardware deployment.

Frontend Interface

- A Streamlit app visualizes predictions and model outputs interactively.
- Power BI is used to present insights, trends, and analytics from the data.

Real-time Streaming & Analytics

- PySpark is used for large-scale analysis and ingestion.

DevOps & Deployment

- Everything is containerized and versioned using Docker for reproducibility and deployment.

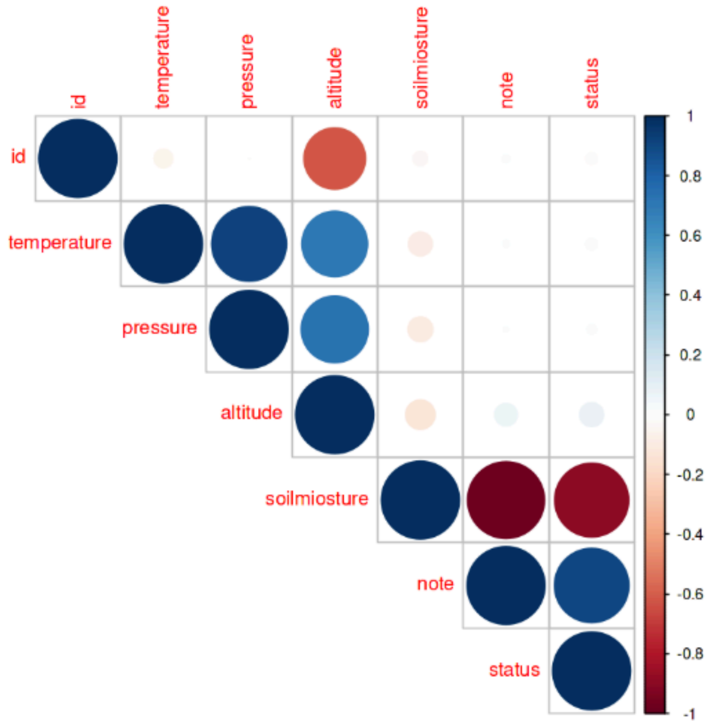
Hardware Interface

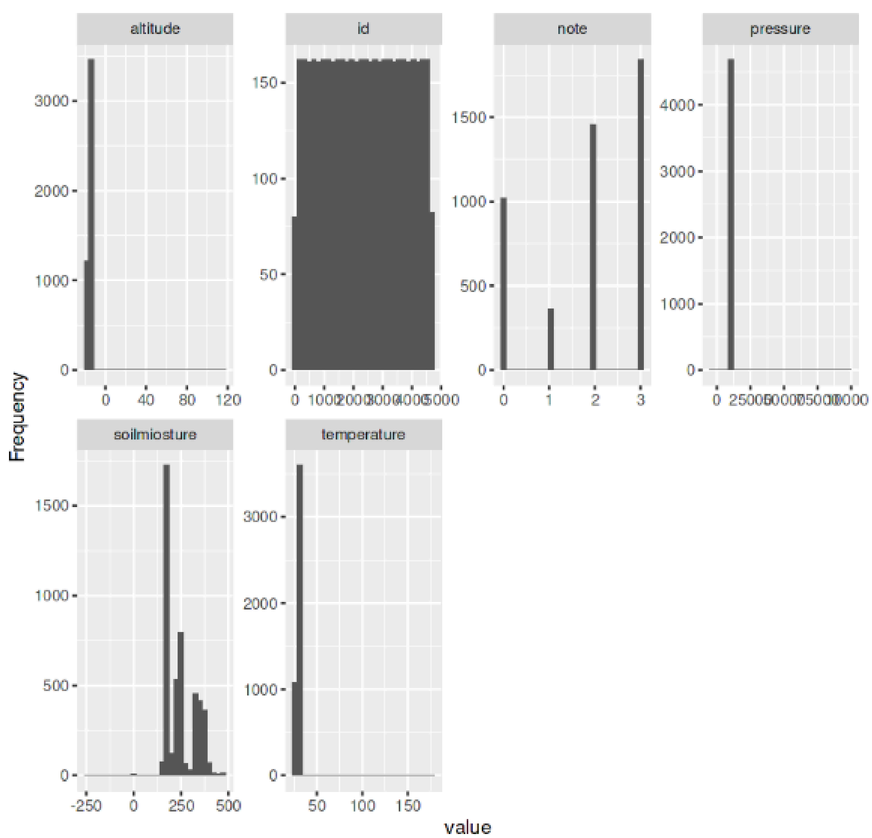
- The exported ONNX model runs on an ESP32 via Wokwi simulation.
- Real-time sensor inputs are used to make predictions and activate irrigation components.

1. Data Preprocessing & Analysis

```r
# Plot histograms for numeric variables
plot_histogram(data)

# Bar plots for categorical variables (if any)
plot_bar(data)
```

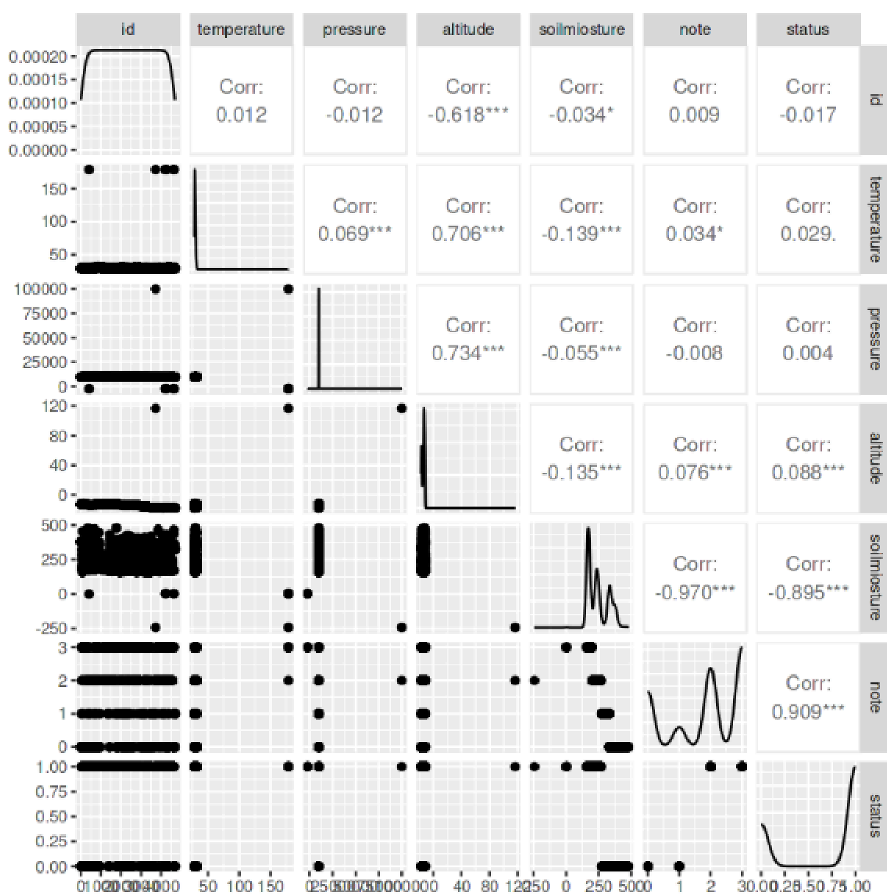
```r
# Correlation matrix for numeric variables
numeric_data <- select_if(data, is.numeric)
cor_matrix <- cor(numeric_data, use = "complete.obs")
corrplot(cor_matrix, method = "circle", type = "upper")

# Pair plots
ggpairs(numeric_data)
```

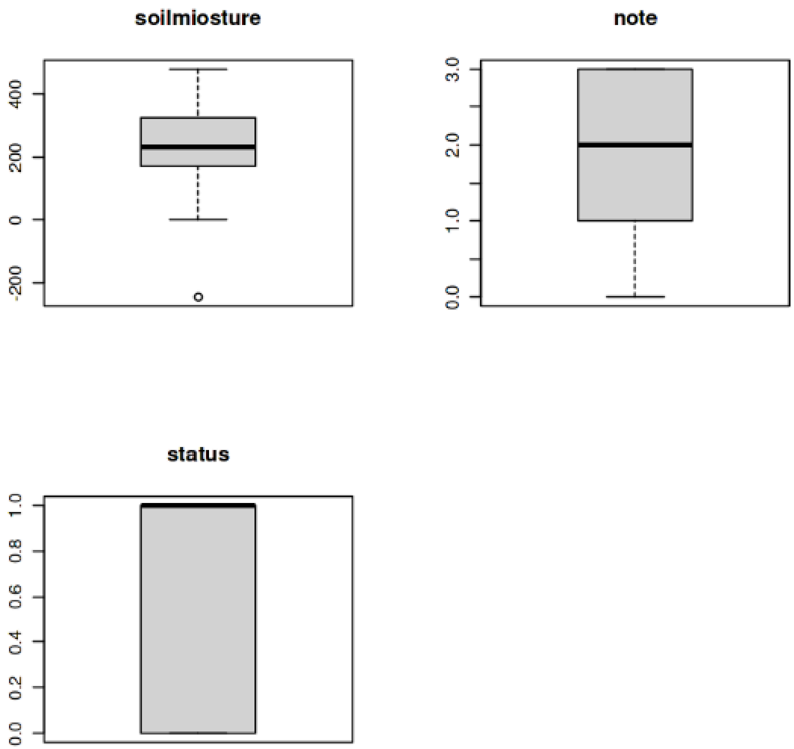

id
temperature
pressure
altitude
soilmiosture
note
status
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1


altitude
id
note
pressure
soilmiosture
temperature
Frequency
value

```
# Boxplots for numeric variables
numeric_cols <- colnames(numeric_data)
par(mfrow = c(2, 2))  # Adjust layout
for(col in numeric_cols){
  boxplot(numeric_data[[col]], main = col)
}
```

```
# Scatter plots for selected relationships
ggplot(data, aes(x = data[[1]], y = data[[2]])) +
  geom_point() +
  labs(title = paste(names(data)[1], "vs", names(data)[2]))
```





2. Model Training

```
Windsurf: Refactor | Explain
class TeacherNet(nn.Module):
    Windsurf: Refactor | Explain | Generate Docstring | X
    def __init__(self, input_size, num_classes):
        super(TeacherNet, self).__init__()
        self.fc1 = nn.Linear(input_size, 128)
        self.fc2 = nn.Linear(128, 64)
        self.fc3 = nn.Linear(64, 32)
        self.out = nn.Linear(32, num_classes)

    Windsurf: Refactor | Explain | Generate Docstring | X
    def forward(self, x):
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = F.relu(self.fc3(x))
        return self.out(x)
```

```python
# Define student model
Windsurf: Refactor | Explain
class StudentNet(nn.Module):
    Windsurf: Refactor | Explain | Generate Docstring | X
    def __init__(self, input_size, num_classes):
        super(StudentNet, self).__init__()
        self.fc1 = nn.Linear(input_size, 32)
        self.out = nn.Linear(32, num_classes)

    Windsurf: Refactor | Explain | Generate Docstring | X
    def forward(self, x):
        x = F.relu(self.fc1(x))
        return self.out(x)
```

```python
# Distillation training
student = StudentNet(X_train.shape[1], len(np.unique(y)))
# teacher = torch.load("teacher_model.pt")

import torch
from torch.serialization import add_safe_globals

# Register the class if it's a custom class
add_safe_globals([TeacherNet])

teacher = torch.load("teacher_model.pt", weights_only=False)
opt = torch.optim.Adam(student.parameters(), lr=0.001)

T, alpha = 3.0, 0.7

Windsurf: Refactor | Explain | Generate Docstring | X
def distill_loss(s_logit, t_logit, labels, T, alpha):
    h = F.cross_entropy(s_logit, labels)
    t_soft = F.log_softmax(t_logit / T, dim=1)
    s_soft = F.log_softmax(s_logit / T, dim=1)
    s_loss = F.kl_div(s_soft, t_soft, log_target=True, reduction='batchmean') * (T**2)
    return alpha * s_loss + (1 - alpha) * h

for epoch in range(20):
    student.train()
    teacher.eval()
    for xb, yb in train_loader:
        with torch.no_grad():
            t_logit = teacher(xb)
        s_logit = student(xb)
        loss = distill_loss(s_logit, t_logit, yb, T, alpha)
        opt.zero_grad(); loss.backward(); opt.step()

# Save final model
torch.save(student.state_dict(), "student_model.pt")
```

Do you
REditor

```
Classes are  ['Very Dry' 'Dry' 'Wet' 'Very Wet']
Epoch 1, Loss: 63.0976
Epoch 2, Loss: 3.1205
Epoch 3, Loss: 0.2682
Epoch 4, Loss: 0.1120
Epoch 5, Loss: 0.0602
Epoch 6, Loss: 0.0385
Epoch 7, Loss: 0.0267
Epoch 8, Loss: 0.0198
Epoch 9, Loss: 0.0154
Epoch 10, Loss: 0.0123
Epoch 11, Loss: 0.0099
Epoch 12, Loss: 0.0082
Epoch 13, Loss: 0.0069
Epoch 14, Loss: 0.0058
Epoch 15, Loss: 0.0050
Epoch 16, Loss: 0.0043
Epoch 17, Loss: 0.0038
Epoch 18, Loss: 0.0033
Epoch 19, Loss: 0.0029
Epoch 20, Loss: 0.0026
Epoch 21, Loss: 0.0023
Epoch 22, Loss: 0.0021
Epoch 23, Loss: 0.0018
Epoch 24, Loss: 0.0017
Epoch 25, Loss: 0.0015
✅ Teacher model saved as 'teacher_model.pt'
```

3. Model Export

```python
dummy_input = torch.randn(1, X_train.shape[1])
torch.onnx.export(student, dummy_input, "student_model.onnx", input_names=["input"], output_names=["output"],
                  dynamic_axes={"input": {0: "batch_size"}, "output": {0: "batch_size"}}, opset_version=11)
```

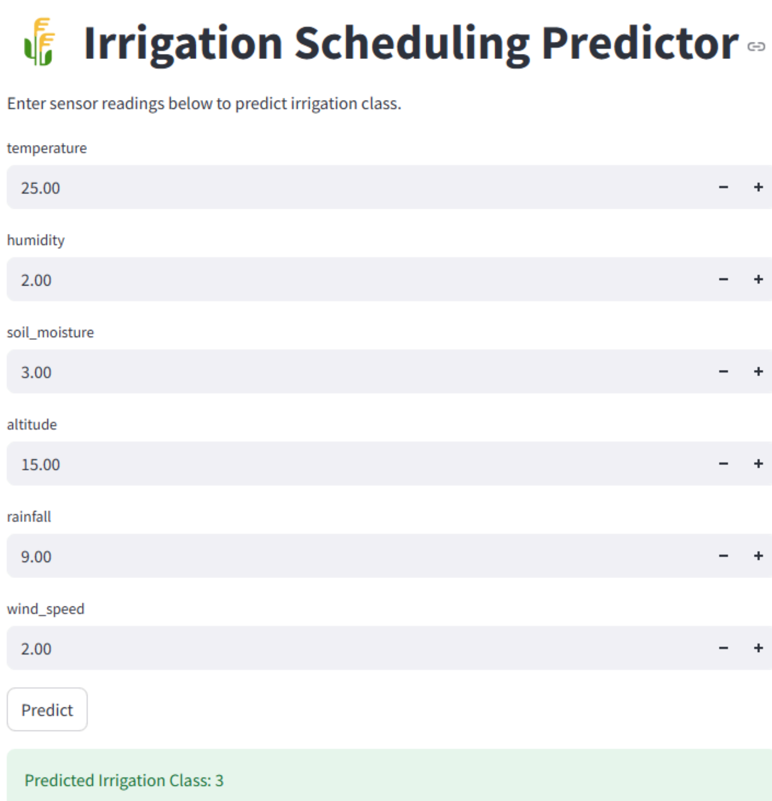
AIHT-FINALPROJECT

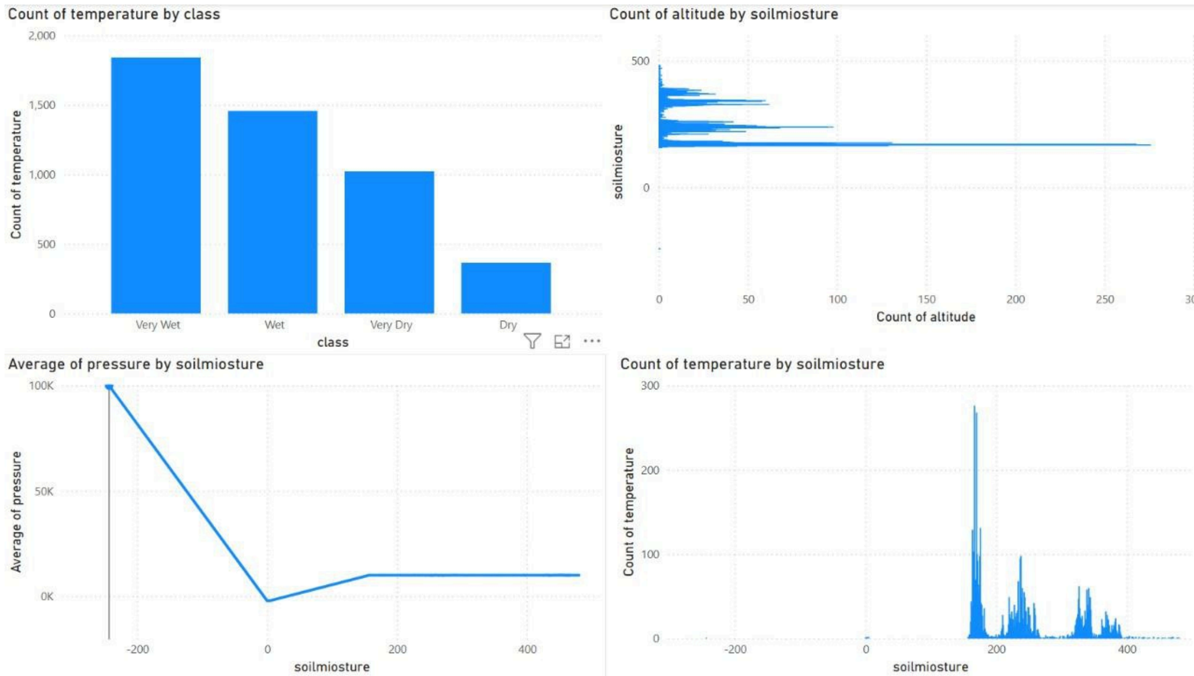
- __pycache__
- venv
- app.py
- converter.py
- Dockerfile
- eda-aihtproject.ipynb
- frontend.py
- Irrigation Scheduling.csv
- model_train_and_export.py
- README.md
- requirements.txt
- student_model.onnx
- student_model.pt
- teacher_model.pt
- teacher.py
- temp.py
- test_input.txt

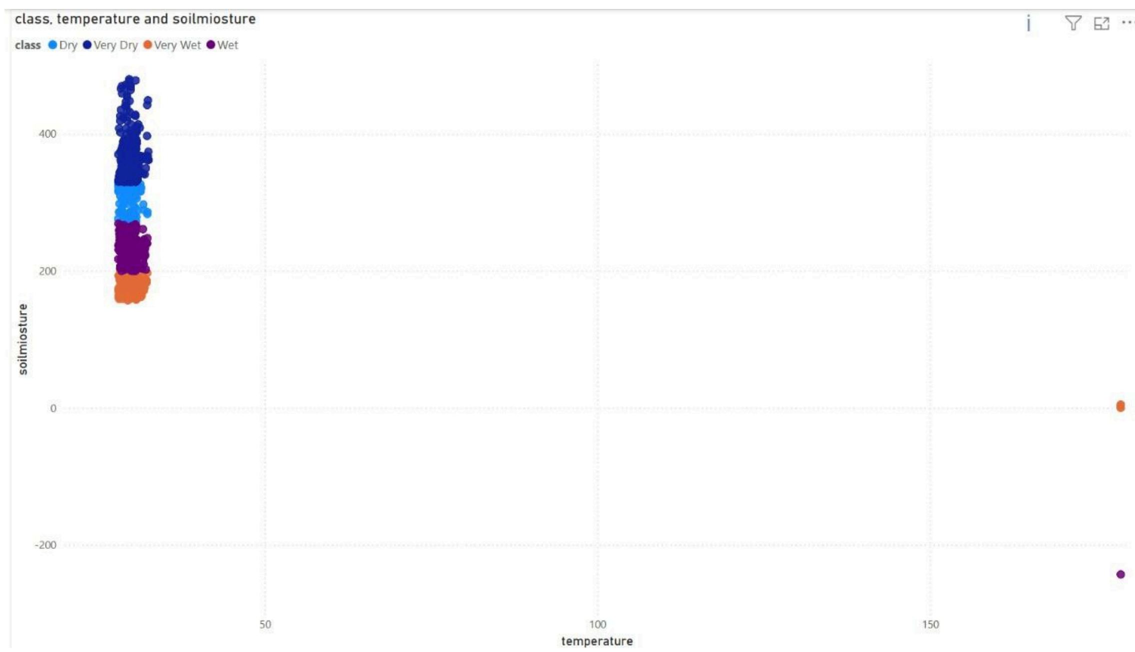
```python
from fastapi import FastAPI, File
import numpy as np
import onnxruntime as ort

app = FastAPI()
session = ort.InferenceSession("student_model.onnx")

Windsurf: Refactor | Explain | Generate Docstring | ×
@app.post("/predict")
def predict_from_file(file: bytes = File(...)):
    content = file.decode("utf-8").strip()
    features = np.array([list(map(float, content.split(',')))], dtype=np.float32)
    outputs = session.run(["output"], {"input": features})
    predicted_class = int(np.argmax(outputs[0]))
    return {"predicted_class": predicted_class}
```

4. Frontend

```python
import streamlit as st
import numpy as np
import onnxruntime as ort

# Load ONNX model
session = ort.InferenceSession("student_model.onnx")

st.title("🌾 Irrigation Scheduling Predictor")
st.write("Enter sensor readings below to predict irrigation class.")


feature_names = ["temperature", "humidity", "soil_moisture", "altitude", "rainfall", "wind_speed"]

inputs = []
for name in feature_names:
    value = st.number_input(f"{name}", value=0.0)
    inputs.append(value)

if st.button("Predict"):
    features = np.array([inputs], dtype=np.float32)
    output = session.run(["output"], {"input": features})
    predicted_class = int(np.argmax(output[0]))
    st.success(f"Predicted Irrigation Class: {predicted_class}")
```




Count of temperature by class
2,000
1,500
1,000
500
0
Count of temperature
Very Wet
Wet
Very Dry
Dry
class
Count of altitude by soilmiosture
500
0
soilmiosture
0
50
100
150
200
250
Count of altitude
Average of pressure by soilmiosture
100K
50K
0K
Average of pressure
-200
0
200
400
soilmiosture
Count of temperature by soilmiosture
300
200
100
0
Count of temperature
-200
0
200
400
soilmiosture


class, temperature and soilmiosture
class Dry Very Dry Very Wet Wet
400
200
0
-200
soilmiosture
50
100
150
temperature

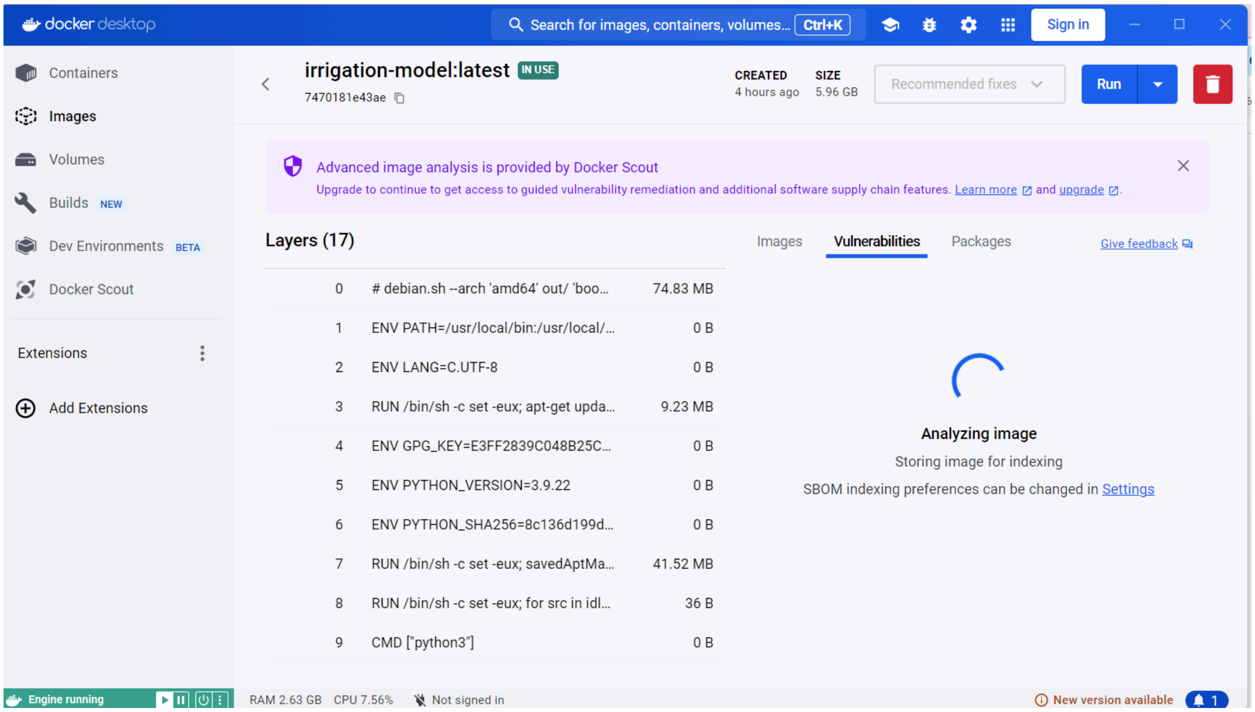
5. Real-time Streaming

spark_teacher.py > ...

```python
from pyspark.sql import SparkSession
from pyspark.sql.functions import col
from sklearn.preprocessing import LabelEncoder, StandardScaler
import pandas as pd
import numpy as np

# Initialize Spark
spark = SparkSession.builder.appName("IrrigationPipeline").getOrCreate()

# Load dataset
df_spark = spark.read.csv("Irrigation Scheduling.csv", header=True, inferSchema=True)

# Drop unnecessary columns and handle nulls
df_spark = df_spark.drop("id", "date", "time")
df_spark = df_spark.na.fill({"altitude": df_spark.select("altitude").agg({"altitude": "mean"}).first()[0]})

# Convert to Pandas for label encoding and scaling
df = df_spark.toPandas()

# Encode labels
le = LabelEncoder()
df["class_encoded"] = le.fit_transform(df["class"])

# Feature scaling
X = df.drop(columns=["class", "class_encoded"])
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
y = df["class_encoded"]

# Create final DataFrame
df_final = pd.DataFrame(X_scaled, columns=X.columns)
df_final["label"] = y

# Save scaled test set to simulate Kafka stream
train_df = df_final.sample(frac=0.8, random_state=42)
test_df = df_final.drop(train_df.index)
```

6. Deployment and DevOps

```dockerfile
FROM python:3.9-slim
WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY app.py .
COPY student_model.onnx .

EXPOSE 8000
CMD ["uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]
```



7. Hardware Interface

```python
!pip install Flask onnxruntime numpy

import flask
import onnxruntime as ort
import numpy as np
import os
import csv
from datetime import datetime
import threading

app = flask.Flask(__name__)

# --- Configuration ---
MODEL_PATH = 'student_model.onnx'  # Make sure this file is deployed with your app
INPUT_NAME = None
OUTPUT_NAME = None
EXPECTED_INPUT_SHAPE = (1, 6)
LOG_FILE = 'data_log.csv'  # <-- CSV log file
# --- End Configuration ---

session = None

def load_model():
    global session, INPUT_NAME, OUTPUT_NAME
    if not os.path.exists(MODEL_PATH):
        print(f"ERROR: Model file not found at {MODEL_PATH}")
        return False
    try:
        print(f"Loading ONNX model from {MODEL_PATH}...")
        session = ort.InferenceSession(MODEL_PATH, providers=['CPUExecutionProvider'])
        INPUT_NAME = session.get_inputs()[0].name
        OUTPUT_NAME = session.get_outputs()[0].name
        print(f"Model loaded successfully.")
        print(f"Input Name: {INPUT_NAME}, Expected Shape: {EXPECTED_INPUT_SHAPE}")
        print(f"Output Name: {OUTPUT_NAME}")
        return True
```

```python
@app.route('/infer', methods=['POST'])
def infer():
    if session is None:
        print("Model not loaded, attempting reload...")
        if not load_model():
            return flask.jsonify({"error": "Model not loaded on server"}), 500

    try:
        data = flask.request.get_json()
        if not data or 'inputs' not in data:
            return flask.jsonify({"error": "Missing 'inputs' key in JSON payload"}), 400

        input_values = data['inputs']

        if not isinstance(input_values, list) or len(input_values) != EXPECTED_INPUT_SHAPE[1]:
            return flask.jsonify({"error": f"Expected a list of {EXPECTED_INPUT_SHAPE[1]} float values in 'inputs'"}), 400

        input_array = np.array(input_values, dtype=np.float32).reshape(EXPECTED_INPUT_SHAPE)
        feeds = {INPUT_NAME: input_array}
        results = session.run([OUTPUT_NAME], feeds)
        output_data = results[0].flatten().tolist()

        log_to_csv(input_values, output_data)

        print(f"Received input: {input_values}, Produced output: {output_data}")
        return flask.jsonify({"outputs": output_data})

    except Exception as e:
        print(f"Error during inference: {e}")
        return flask.jsonify({"error": f"Inference error: {str(e)}"}), 500
```