MONITORING

CODE:-

Monitor.py

import pandas as pd

import time

class RealTimeMonitor:

    def \_\_init\_\_(self, data\_path):

        self.df = pd.read\_csv(data\_path, parse\_dates=['RealtimeClockDateandTime'])

        self.index = 0

    def get\_next\_reading(self):

        if self.index >= len(self.df):

            return None  # End of stream

        row = self.df.iloc[self.index]

        self.index += 1

        return row.to\_dict()

    def get\_history(self, limit=100):

        return self.df.iloc[max(0, self.index - limit):self.index]

    def reset(self):

        self.index = 0

STREAMLIT:-

UI/dashboard.py

import streamlit as st

import pandas as pd

import numpy as np

import time

import random

from datetime import datetime

import pydeck as pdk

from tensorflow.keras.models import load\_model

import joblib

import os

# Suppress oneDNN log clutter

os.environ["TF\_ENABLE\_ONEDNN\_OPTS"] = "0"

# App Config

st.set\_page\_config(page\_title="Smart Meter Monitoring", layout="wide")

st.title("🔌 Real-Time Smart Meter Monitoring Dashboard")

# Sidebar Controls

refresh\_rate = st.sidebar.slider("🔁 Refresh interval (sec)", 1, 10, 5)

data\_choice = st.sidebar.radio("🧪 Data Source", ["Real", "Simulated"])

# Load Monitoring Data

DATA\_PATH = "data/preprocessed\_data.csv" if data\_choice == "Real" else "data/simulated\_data.csv"

df = pd.read\_csv(

    DATA\_PATH,

    parse\_dates=["RealtimeClockDateandTime"],

    dayfirst=True  # Set this to True if your CSV uses DD-MM-YYYY format

)

# Simulated Real-Time Loop

placeholder = st.empty()

i = 0

while True:

    with placeholder.container():

        data\_batch = df.iloc[i:i+1]

        if data\_batch.empty:

            st.success("✔️ Stream complete.")

            break

        row = data\_batch.iloc[0]

        st.subheader(f"Meter ID: `{row['METERSNO']}` | Time: {row['RealtimeClockDateandTime']}")

        col1, col2, col3, col4 = st.columns(4)

        col1.metric("🔋 Voltage (V)", f"{row['Voltage']:.2f}")

        col2.metric("⚡ Phase Current (A)", f"{row['NormalPhaseCurrent']:.2f}")

        col3.metric("📊 Power Factor", f"{row['SystemPowerFactor']:.2f}")

        col4.metric("🌐 Frequency (Hz)", f"{row['Frequency']:.2f}")

        st.markdown("### 📈 Energy Usage")

        st.line\_chart(df.iloc[:i+1][['BlockEnergykWh', 'CumulativeEnergykWh']])

        st.markdown("### 🗺️ Location")

        st.pydeck\_chart(pdk.Deck(

            map\_style='mapbox://styles/mapbox/light-v9',

            initial\_view\_state=pdk.ViewState(

                latitude=row['Latitude'],

                longitude=row['Longitude'],

                zoom=12,

                pitch=50,

            ),

            layers=[

                pdk.Layer(

                    'ScatterplotLayer',

                    data=data\_batch,

                    get\_position='[Longitude, Latitude]',

                    get\_color='[200, 30, 0, 160]',

                    get\_radius=500,

                ),

            ],

        ))

    i += 1

    time.sleep(refresh\_rate)

# ------------------------

# 🔮 Energy Forecast Section

# ------------------------

st.markdown("---")

st.title("📊 7-Day Smart Meter Energy Forecast")

try:

    # Load Model and Scalers

    model = load\_model("models/final\_energy\_forecast\_model.h5")

    feature\_scaler = joblib.load("models/final\_feature\_scaler.pkl")

    target\_scaler = joblib.load("models/final\_target\_scaler.pkl")

    # File Uploader

    uploaded\_file = st.file\_uploader("14\_day\_inp", type="csv")

    if uploaded\_file is not None:

        new\_data = pd.read\_csv(uploaded\_file)

        if new\_data.shape[0] != 14:

            st.error("❌ Upload must contain exactly 14 rows (days) of input features.")

        else:

            # Predict

            input\_scaled = feature\_scaler.transform(new\_data)

            input\_seq = np.expand\_dims(input\_scaled, axis=0)

            pred = model.predict(input\_seq)

            pred = pred.reshape(7, 2)

            pred\_inv = target\_scaler.inverse\_transform(pred)

            # Display Prediction

            st.subheader("✅ Predicted Energy Usage (Next 7 Days)")

            forecast\_df = pd.DataFrame(pred\_inv, columns=["Predicted\_kWh", "Predicted\_kVAh"])

            forecast\_df.index = [f"Day {i+1}" for i in range(7)]

            st.dataframe(forecast\_df)

            st.line\_chart(forecast\_df)

except Exception as e:

    st.error(f"⚠️ Forecast model loading error: {e}")

Data/preprocessed\_data

SIMULATION

CODE

SIMULATE.PY

# simulation/simulate.py

import pandas as pd

import numpy as np

from datetime import timedelta

import random

def simulate\_variation(df):

    df\_sim = df.copy()

    # Add random fluctuations to key features

    df\_sim['NormalPhaseCurrent'] \*= np.random.normal(1.0, 0.05, size=len(df\_sim))

    df\_sim['Voltage'] += np.random.normal(0, 1.5, size=len(df\_sim))

    df\_sim['Frequency'] += np.random.normal(0, 0.01, size=len(df\_sim))

    df\_sim['CumulativeEnergykWh'] += np.random.normal(0.01, 0.05, size=len(df\_sim))

    return df\_sim

def simulate\_theft(df, theft\_meter\_ids):

    df\_theft = df.copy()

    for meter in theft\_meter\_ids:

        mask = df\_theft['METERSNO'] == meter

        df\_theft.loc[mask, 'CumulativeEnergykWh'] \*= 0.5  # Under-reporting

        df\_theft.loc[mask, 'SystemPowerFactor'] = df\_theft.loc[mask, 'SystemPowerFactor'] \* 0.6

    return df\_theft

def inject\_spikes(df, spike\_indices):

    df\_spike = df.copy()

    for idx in spike\_indices:

        df\_spike.at[idx, 'NormalPhaseCurrent'] \*= 2.5

        df\_spike.at[idx, 'Voltage'] += 20

    return df\_spike

# Usage Example

if \_\_name\_\_ == "\_\_main\_\_":

    df = pd.read\_csv("data/preprocessed\_data.csv")

    # Simulate

    df\_varied = simulate\_variation(df)

    df\_theft = simulate\_theft(df\_varied, theft\_meter\_ids=["A00002", "A00029"])

    df\_spiked = inject\_spikes(df\_theft, spike\_indices=[5, 15, 25])

    df\_spiked.to\_csv("data/simulated\_data.csv", index=False)

    print("✅ Simulation complete: saved as simulated\_data.csv")

DATASET:

SIMULATE\_DATA

PREDICTION ENERGY

MONTH

Train\_predict\_energy.py

##month prediction

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error

from keras.models import Sequential

from keras.layers import LSTM, Dense, Dropout, RepeatVector, TimeDistributed

from keras.callbacks import EarlyStopping

import joblib

# === Step 1: Load and preprocess data ===

df = pd.read\_csv('/mnt/data/cleaned.csv', parse\_dates=['DateTime'], index\_col='DateTime')

df = df[['KWHhh']]

df = df.resample('D').sum().dropna()  # Daily energy consumption

# === Step 2: Scale ===

scaler = MinMaxScaler()

scaled\_data = scaler.fit\_transform(df)

# === Step 3: Create sliding window sequences ===

def create\_multi\_step\_sequences(data, input\_steps, output\_steps):

    X, y = [], []

    for i in range(len(data) - input\_steps - output\_steps + 1):

        X.append(data[i:i + input\_steps])

        y.append(data[i + input\_steps:i + input\_steps + output\_steps])

    return np.array(X), np.array(y)

lookback = 30  # 30 days input

forecast\_horizon = 30  # predict next 30 days

X, y = create\_multi\_step\_sequences(scaled\_data, lookback, forecast\_horizon)

# === Step 4: Train-Test Split ===

split\_idx = int(len(X) \* 0.8)

X\_train, X\_test = X[:split\_idx], X[split\_idx:]

y\_train, y\_test = y[:split\_idx], y[split\_idx:]

# === Step 5: Reshape for LSTM ===

X\_train = X\_train.reshape((X\_train.shape[0], X\_train.shape[1], 1))

X\_test = X\_test.reshape((X\_test.shape[0], X\_test.shape[1], 1))

# === Step 6: Define model ===

model = Sequential()

model.add(LSTM(64, activation='relu', input\_shape=(lookback, 1)))

model.add(RepeatVector(forecast\_horizon))

model.add(LSTM(32, activation='relu', return\_sequences=True))

model.add(TimeDistributed(Dense(1)))

model.compile(optimizer='adam', loss='mse')

# === Step 7: Train model ===

early\_stop = EarlyStopping(patience=10, restore\_best\_weights=True)

model.fit(X\_train, y\_train, epochs=100, batch\_size=16, validation\_split=0.1, callbacks=[early\_stop])

# === Step 8: Predict and inverse transform ===

y\_pred = model.predict(X\_test)

y\_pred = y\_pred.reshape(y\_pred.shape[0], y\_pred.shape[1])

y\_test = y\_test.reshape(y\_test.shape[0], y\_test.shape[1])

y\_pred\_inv = scaler.inverse\_transform(y\_pred)

y\_test\_inv = scaler.inverse\_transform(y\_test)

# === Step 9: Evaluation ===

rmse = np.sqrt(mean\_squared\_error(y\_test\_inv.flatten(), y\_pred\_inv.flatten()))

mae = mean\_absolute\_error(y\_test\_inv.flatten(), y\_pred\_inv.flatten())

print("\n✅ Evaluation:")

print(f"RMSE: {rmse:.2f}")

print(f"MAE: {mae:.2f}")

# === Step 10: Visualization ===

plt.figure(figsize=(14, 6))

plt.plot(y\_test\_inv[0], label='Actual')

plt.plot(y\_pred\_inv[0], label='Predicted')

plt.title('Energy Usage Forecast (Next 30 Days)')

plt.xlabel('Day')

plt.ylabel('Energy (kWh)')

plt.legend()

plt.grid(True)

plt.tight\_layout()

plt.show()

# === Step 11: Save model and scaler ===

model.save('energy\_lstm\_month\_model.h5')

joblib.dump(scaler, 'energy\_scaler\_month.pkl')

print("✅ Model and scaler saved to disk.")

Week

Train\_predict\_energy.py

# train\_model\_fixed.py

####day prediction

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

from sklearn.preprocessing import MinMaxScaler

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error

from keras.models import Sequential

from keras.layers import LSTM, Dense, Dropout

from keras.callbacks import EarlyStopping

import joblib

# === Step 1: Load and Format Data ===

df = pd.read\_csv('cleaned.csv', parse\_dates=['DateTime'])

df.set\_index('DateTime', inplace=True)

# === Step 2: Select Energy Column and Resample ===

df = df[['KWHhh']].resample('D').sum().dropna()  # Daily energy usage

# === Step 3: Scale Data ===

scaler = MinMaxScaler()

scaled\_data = scaler.fit\_transform(df)

# === Step 4: Create Sequences ===

def create\_sequences(data, window\_size):

    X, y = [], []

    for i in range(len(data) - window\_size):

        X.append(data[i:i + window\_size])

        y.append(data[i + window\_size])

    return np.array(X), np.array(y)

lookback = 7

X, y = create\_sequences(scaled\_data, lookback)

# === Step 5: Train-Test Split ===

split\_idx = int(len(X) \* 0.8)

X\_train, X\_test = X[:split\_idx], X[split\_idx:]

y\_train, y\_test = y[:split\_idx], y[split\_idx:]

# === Step 6: Build LSTM Model ===

model = Sequential()

model.add(LSTM(64, input\_shape=(lookback, 1), return\_sequences=True))

model.add(Dropout(0.2))

model.add(LSTM(32))

model.add(Dropout(0.2))

model.add(Dense(1))

model.compile(loss='mse', optimizer='adam')

# === Step 7: Train Model ===

early\_stop = EarlyStopping(patience=10, restore\_best\_weights=True)

model.fit(X\_train, y\_train, epochs=100, batch\_size=16, validation\_split=0.1, callbacks=[early\_stop])

# === Step 8: Predict and Inverse Scale ===

y\_pred = model.predict(X\_test)

y\_pred\_inv = scaler.inverse\_transform(y\_pred)

y\_test\_inv = scaler.inverse\_transform(y\_test)

# === Step 9: Evaluate ===

rmse = np.sqrt(mean\_squared\_error(y\_test\_inv, y\_pred\_inv))

mae = mean\_absolute\_error(y\_test\_inv, y\_pred\_inv)

print(f"\n✅ Evaluation Results:")

print(f"RMSE: {rmse:.2f}")

print(f"MAE: {mae:.2f}")

# === Step 10: Plot Results ===

plt.figure(figsize=(12, 6))

plt.plot(y\_test\_inv, label='Actual')

plt.plot(y\_pred\_inv, label='Predicted')

plt.legend()

plt.title("Energy Usage Prediction (Next Day)")

plt.xlabel("Time Step")

plt.ylabel("Energy (kWh)")

plt.grid(True)

plt.tight\_layout()

plt.show()

# === Step 11: Save Model and Scaler ===

model.save("energy\_lstm\_model.h5")

joblib.dump(scaler, "energy\_scaler.pkl")

print("✅ Model and scaler saved to disk.")

Streamlit

App.py

import streamlit as st

st.set\_page\_config(page\_title="Energy Predictor", layout="centered")

import numpy as np

import pandas as pd

import joblib

from tensorflow.keras.models import load\_model

from io import BytesIO

# === Load Models and Scalers ===

@st.cache\_resource

def load\_all\_models():

    day\_model = load\_model("models/energy\_lstm\_model.h5", compile=False)

    month\_model = load\_model("models/energy\_lstm\_month\_model.h5", compile=False)

    day\_scaler = joblib.load("models/energy\_scaler.pkl")

    month\_scaler = joblib.load("models/energy\_scaler\_month.pkl")

    return day\_model, month\_model, day\_scaler, month\_scaler

day\_model, month\_model, day\_scaler, month\_scaler = load\_all\_models()

# === Styling ===

st.markdown("""

    <style>

    body {

        color: #ffffff;

        background-color: #0e1117;

    }

    .stButton>button {

        background-color: #3a3f52;

        color: white;

        border-radius: 8px;

        padding: 0.5em 1em;

    }

    </style>

""", unsafe\_allow\_html=True)

# === App Header ===

st.title("🔋 Energy Usage Predictor")

st.write("Upload past energy usage data to forecast the next day or next 30 days.")

# === Prediction Type ===

prediction\_type = st.radio("📊 Select Prediction Type:", ["Next Day", "Next Month"])

# === File Upload ===

expected\_days = 7 if prediction\_type == "Next Day" else 30

uploaded\_file = st.file\_uploader(f"📄 Upload CSV with last {expected\_days} daily energy values (column: `KWHhh`)", type=["csv"])

if uploaded\_file is not None:

    try:

        df = pd.read\_csv(uploaded\_file)

        if 'KWHhh' not in df.columns:

            st.error("❌ CSV must contain a `KWHhh` column.")

        elif len(df) < expected\_days:

            st.error(f"❌ Please provide at least {expected\_days} rows of data.")

        else:

            input\_values = df['KWHhh'].values[-expected\_days:]

            st.success(f"✅ Uploaded {len(input\_values)} records. Ready to predict.")

            if st.button("🔮 Predict"):

                try:

                    input\_array = np.array(input\_values).reshape(-1, 1)

                    if prediction\_type == "Next Day":

                        input\_scaled = day\_scaler.transform(input\_array).reshape(1, 7, 1)

                        prediction\_scaled = day\_model.predict(input\_scaled)

                        prediction = day\_scaler.inverse\_transform(prediction\_scaled)[0][0]

                        st.metric(label="📈 Predicted Energy (Tomorrow)", value=f"{prediction:.2f} kWh")

                        # Download

                        csv = pd.DataFrame({"Predicted kWh": [prediction]}).to\_csv(index=False)

                        st.download\_button("📥 Download Result CSV", data=csv, file\_name="next\_day\_prediction.csv", mime="text/csv")

                    else:  # Next Month

                        input\_scaled = month\_scaler.transform(input\_array).reshape(1, 30, 1)

                        prediction\_scaled = month\_model.predict(input\_scaled).reshape(30, 1)

                        prediction = month\_scaler.inverse\_transform(prediction\_scaled).flatten()

                        st.success("📆 Predicted Energy for Next 30 Days")

                        st.line\_chart(prediction)

                        # Download

                        result\_df = pd.DataFrame({

                            "Day": [f"Day {i+1}" for i in range(30)],

                            "Predicted kWh": prediction

                        })

                        csv = result\_df.to\_csv(index=False)

                        st.download\_button("📥 Download 30-Day Forecast", data=csv, file\_name="next\_month\_prediction.csv", mime="text/csv")

                except Exception as e:

                    st.error(f"❌ Prediction failed: {e}")

    except Exception as e:

        st.error(f"❌ Error reading file: {e}")

Dataset:-test22 for testing

Trained :-cleaned.csv

Energy billing

Bill.py

CODE

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.ensemble import RandomForestRegressor

from sklearn.metrics import mean\_squared\_error, r2\_score

import joblib

# Step 1: Load dataset

df = pd.read\_csv("Training\_Data.csv")

# Optional: Rename for clarity

df.rename(columns={"Amount\_Billed": "Previous\_Bill"}, inplace=True)

# Step 2: Select features and target

features = [

    'Energy\_Consumption\_KWh',

    'Units\_Consumed\_KWh',

    'Tariff\_Per\_KWh',

    'Average\_Daily\_Consumption\_KWh'

]

target = 'Projected\_Bill'

X = df[features]

y = df[target]

# Step 3: Split into training and testing sets (80/20)

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, y, test\_size=0.2, random\_state=42

)

# Step 4: Train the model

model = RandomForestRegressor(n\_estimators=100, random\_state=42)

model.fit(X\_train, y\_train)

# Step 5: Predict on test set

y\_pred = model.predict(X\_test)

# Step 6: Evaluate model

rmse = np.sqrt(mean\_squared\_error(y\_test, y\_pred))

r2 = r2\_score(y\_test, y\_pred)

print(f"✅ Model trained successfully!")

print(f"📉 RMSE: {rmse:.4f}")

print(f"📈 R² Score: {r2:.5f}")

# Step 7: Save the model for reuse

joblib.dump(model, "bill\_predictor\_model.pkl")

print("💾 Model saved as 'bill\_predictor\_model.pkl'")

import matplotlib.pyplot as plt

# Step 6.1: Plot Actual vs Predicted

plt.figure(figsize=(10, 6))

plt.scatter(y\_test, y\_pred, alpha=0.5, color='dodgerblue', label='Predicted vs Actual')

plt.plot([y\_test.min(), y\_test.max()], [y\_test.min(), y\_test.max()], 'r--', lw=2, label='Perfect Prediction')

plt.xlabel("Actual Projected Bill")

plt.ylabel("Predicted Projected Bill")

plt.title("Actual vs Predicted Electricity Bill")

plt.legend()

plt.grid(True)

plt.tight\_layout()

plt.show()

MODEL

BILL\_PREDICTION\_MODEL.PKL

STREAMLIT

Billingstream.py

import streamlit as st

import pandas as pd

import joblib

import smtplib

from email.mime.text import MIMEText

from email.mime.multipart import MIMEMultipart

# Load model

@st.cache\_resource

def load\_model():

    return joblib.load("models/bill\_predictor\_model.pkl")

model = load\_model()

# Send email function

def send\_email(to\_email, name, predicted, previous):

    sender\_email = "nikithnandi08@gmail.com"

    sender\_password = "sshz jpyi pibg jxev"  # Use App Password

    subject = "⚠️ High Electricity Usage Alert"

    body = f"""

    Dear {name},

    Our system predicts that your next electricity bill will be ₹{predicted:.2f},

    which is higher than your previous bill of ₹{previous:.2f}.

    Please consider reducing your usage to avoid higher charges.

    Regards,

    Energy Monitoring Team

    """

    msg = MIMEMultipart()

    msg['From'] = sender\_email

    msg['To'] = to\_email

    msg['Subject'] = subject

    msg.attach(MIMEText(body, 'plain'))

    try:

        server = smtplib.SMTP('smtp.gmail.com', 587)

        server.starttls()

        server.login(sender\_email, sender\_password)

        server.send\_message(msg)

        server.quit()

        st.success(f"✅ Email sent to {to\_email}")

    except Exception as e:

        st.error(f"❌ Failed to send email to {to\_email}: {e}")

# Streamlit UI

st.title("🔌 Electricity Bill Prediction & Alert System")

uploaded\_file = st.file\_uploader("📁 Upload your Training\_Data.csv", type=["csv"])

if uploaded\_file is not None:

    df = pd.read\_csv(uploaded\_file)

    df.rename(columns={"Amount\_Billed": "Previous\_Bill"}, inplace=True)

    features = [

        'Energy\_Consumption\_KWh',

        'Units\_Consumed\_KWh',

        'Tariff\_Per\_KWh',

        'Average\_Daily\_Consumption\_KWh'

    ]

    # Predict future bills

    df['Predicted\_Bill'] = model.predict(df[features])

    # Flag anomalies

    df['Anomaly\_Flag'] = df['Predicted\_Bill'] > df['Previous\_Bill'] \* 1.20

    st.subheader("📊 Predicted Results")

    st.dataframe(df[['Name', 'Email', 'Previous\_Bill', 'Predicted\_Bill', 'Anomaly\_Flag']])

    # Only show anomalous rows

    anomalies = df[df['Anomaly\_Flag'] == True]

    st.subheader("🚨 Anomalies Detected (High Usage)")

    st.dataframe(anomalies[['Name', 'Email', 'Previous\_Bill', 'Predicted\_Bill']])

    if st.button("📧 Send Alerts to All Anomalous Customers"):

        for \_, row in anomalies.iterrows():

            if pd.notna(row['Email']) and pd.notna(row['Name']):

                send\_email(

                    to\_email=row['Email'],

                    name=row['Name'],

                    predicted=row['Predicted\_Bill'],

                    previous=row['Previous\_Bill']

                )

DATA:-TRAINED\_DATA.CSV

FAULT DETECTION

FAULT.PY

import streamlit as st

import pandas as pd

import time

import os

import smtplib

from datetime import datetime

from email.message import EmailMessage

# === CONFIG ===

LOG\_DIR = "logs"

os.makedirs(LOG\_DIR, exist\_ok=True)

EMAIL\_ALERTS = True

ALERT\_EMAIL = "nikithnandi2004@gmail.com"   # ✅ Receiver

EMAIL\_SENDER = "nikithnandi08@gmail.com"    # ✅ Sender Gmail

EMAIL\_PASSWORD = "sshz jpyi pibg jxev"       # ✅ Gmail app password

# Streamlit Page Setup

st.set\_page\_config(page\_title="⚠️ Anomaly Monitoring", layout="wide")

st.title("📡 Smart Meter Monitoring & Anomaly Detection")

# Sidebar Alerts Box

st.sidebar.header("🚨 Live Alerts")

alert\_box = st.sidebar.empty()

# File Upload or default data

uploaded\_file = st.file\_uploader("📂 Upload Smart Meter CSV", type="csv")

if uploaded\_file:

    df = pd.read\_csv(uploaded\_file, parse\_dates=["RealtimeClockDateandTime"])

else:

    DATA\_PATH = "data/preprocessed\_data.csv"

    if os.path.exists(DATA\_PATH):

        df = pd.read\_csv(DATA\_PATH, parse\_dates=["RealtimeClockDateandTime"], dayfirst=True)

    else:

        st.warning("⚠️ Upload a file or add 'data/preprocessed\_data.csv'")

        st.stop()

# Anomaly Detection Function

def detect\_row\_anomalies(row):

    reasons = []

    if row['Voltage'] < 180 or row['Voltage'] > 250:

        reasons.append("Voltage anomaly")

    if row['SystemPowerFactor'] < 0.5:

        reasons.append("Low power factor")

    if row['ActivePower\_kW'] < 0:

        reasons.append("Negative active power")

    if row['Frequency'] < 48.5 or row['Frequency'] > 51.5:

        reasons.append("Frequency anomaly")

    if row['BlockEnergykWh'] == 0:

        reasons.append("Zero energy consumption")

    return reasons

# Email Alert Function

def send\_email\_alert(anomalies):

    if not anomalies or not EMAIL\_ALERTS:

        return

    msg = EmailMessage()

    msg["Subject"] = "⚠️ Smart Meter Anomaly Alert"

    msg["From"] = EMAIL\_SENDER

    msg["To"] = ALERT\_EMAIL

    body = "\n".join(anomalies)

    msg.set\_content(f"Anomalies Detected:\n\n{body}")

    try:

        with smtplib.SMTP("smtp.gmail.com", 587) as server:

            server.starttls()

            server.login(EMAIL\_SENDER, EMAIL\_PASSWORD)

            server.send\_message(msg)

        return True

    except Exception as e:

        st.error(f"❌ Email sending failed: {e}")

        return False

# Create daily log file

log\_file = os.path.join(LOG\_DIR, f"{datetime.today().strftime('%Y-%m-%d')}\_realtime\_log.txt")

anomaly\_log = []

# Real-Time Monitoring Loop

placeholder = st.empty()

for i in range(len(df)):

    with placeholder.container():

        row = df.iloc[i]

        st.subheader(f"⏱️ Time: {row['RealtimeClockDateandTime']}")

        st.metric("🔋 Voltage", f"{row['Voltage']:.2f} V")

        st.metric("⚡ Power (kW)", f"{row['ActivePower\_kW']:.2f}")

        st.metric("🔌 Power Factor", f"{row['SystemPowerFactor']:.2f}")

        st.metric("🌐 Frequency", f"{row['Frequency']:.2f} Hz")

        anomalies = detect\_row\_anomalies(row)

        timestamp = row['RealtimeClockDateandTime']

        alert\_lines = [f"[{timestamp}] - {reason}" for reason in anomalies]

        if anomalies:

            alert\_box.error("\n".join(alert\_lines))

            anomaly\_log.extend(alert\_lines)

            with open(log\_file, "a") as log:

                for line in alert\_lines:

                    log.write(line + "\n")

            send\_email\_alert(alert\_lines)  # ✅ Send email immediately

        else:

            alert\_box.success("✅ No anomalies detected")

        time.sleep(0.7)

# End of Monitoring

st.success("✅ Monitoring Completed")

if anomaly\_log:

    st.subheader("📋 Final Anomaly Summary")

    st.code("\n".join(anomaly\_log))

# Log Viewer Section

st.markdown("---")

st.subheader("📂 View Past Anomaly Logs")

if os.path.exists(LOG\_DIR):

    log\_files = sorted(os.listdir(LOG\_DIR), reverse=True)

    selected\_log = st.selectbox("📄 Select log file", log\_files)

    if selected\_log:

        with open(os.path.join(LOG\_DIR, selected\_log)) as f:

            st.code(f.read(), language="text")

else:

    st.info("No logs found.")

################################################code 1#############################################################

# import streamlit as st

# import pandas as pd

# import time

# import os

# import smtplib

# from datetime import datetime

# from email.message import EmailMessage

# # === CONFIG ===

# LOG\_DIR = "logs"

# os.makedirs(LOG\_DIR, exist\_ok=True)

# EMAIL\_ALERTS = True

# ALERT\_EMAIL = "nikithnandi2004@gmail.com"  # ✅ Replace with your recipient email

# EMAIL\_SENDER = "nikithnandi08@gmail.com"  # ✅ Your Gmail

# EMAIL\_PASSWORD = "sshz jpyi pibg jxev"     # ✅ App password only

# # Streamlit setup

# st.set\_page\_config(page\_title="🔍 Real-Time Anomaly Monitoring", layout="wide")

# st.title("📡 Smart Meter Monitoring & Anomaly Detection")

# # Sidebar for real-time alerts

# st.sidebar.header("🚨 Live Alerts")

# alert\_box = st.sidebar.empty()

# # File Upload or Default File

# uploaded\_file = st.file\_uploader("📂 Upload Smart Meter CSV", type="csv")

# if uploaded\_file:

#     df = pd.read\_csv(uploaded\_file, parse\_dates=["RealtimeClockDateandTime"])

# else:

#     DATA\_PATH = "data/preprocessed\_data.csv"

#     if os.path.exists(DATA\_PATH):

#         df = pd.read\_csv(DATA\_PATH, parse\_dates=["RealtimeClockDateandTime"], dayfirst=True)

#     else:

#         st.warning("⚠️ Upload a file or add 'data/preprocessed\_data.csv'")

#         st.stop()

# # --- Anomaly Detection Function ---

# def detect\_row\_anomalies(row):

#     reasons = []

#     if row['Voltage'] < 180 or row['Voltage'] > 250:

#         reasons.append("Voltage anomaly")

#     if row['SystemPowerFactor'] < 0.5:

#         reasons.append("Low power factor")

#     if row['ActivePower\_kW'] < 0:

#         reasons.append("Negative active power")

#     if row['Frequency'] < 48.5 or row['Frequency'] > 51.5:

#         reasons.append("Frequency anomaly")

#     if row['BlockEnergykWh'] == 0:

#         reasons.append("Zero energy consumption")

#     return reasons

# # --- Email Sender ---

# def send\_email\_alert(anomalies):

#     if not anomalies or not EMAIL\_ALERTS:

#         return

#     msg = EmailMessage()

#     msg["Subject"] = "⚠️ Smart Meter Anomaly Alert"

#     msg["From"] = EMAIL\_SENDER

#     msg["To"] = ALERT\_EMAIL

#     body = "\n".join(anomalies)

#     msg.set\_content(f"Anomalies Detected:\n\n{body}")

#     try:

#         with smtplib.SMTP("smtp.gmail.com", 587) as server:

#             server.starttls()

#             server.login(EMAIL\_SENDER, EMAIL\_PASSWORD)

#             server.send\_message(msg)

#         return True

#     except Exception as e:

#         st.error(f"❌ Email sending failed: {e}")

#         return False

# # Log file path

# log\_file = os.path.join(LOG\_DIR, f"{datetime.today().strftime('%Y-%m-%d')}\_realtime\_log.txt")

# # --- Real-Time Monitoring Loop ---

# placeholder = st.empty()

# anomaly\_log = []

# for i in range(len(df)):

#     with placeholder.container():

#         row = df.iloc[i]

#         st.subheader(f"⏱️ Time: {row['RealtimeClockDateandTime']}")

#         st.metric("🔋 Voltage", f"{row['Voltage']:.2f} V")

#         st.metric("⚡ Power (kW)", f"{row['ActivePower\_kW']:.2f}")

#         st.metric("🔌 Power Factor", f"{row['SystemPowerFactor']:.2f}")

#         st.metric("🌐 Frequency", f"{row['Frequency']:.2f} Hz")

#         # Detect anomalies

#         anomalies = detect\_row\_anomalies(row)

#         ts = row['RealtimeClockDateandTime']

#         alert\_lines = [f"[{ts}] - {reason}" for reason in anomalies]

#         if anomalies:

#             alert\_box.error("\n".join(alert\_lines))

#             anomaly\_log.extend(alert\_lines)

#             with open(log\_file, "a") as log:

#                 for line in alert\_lines:

#                     log.write(line + "\n")

#         else:

#             alert\_box.success("✅ No anomalies detected")

#         time.sleep(0.7)

# # Final Message

# st.success("✅ Monitoring Completed")

# # Show all collected anomalies

# if anomaly\_log:

#     st.subheader("📋 Final Anomaly Summary")

#     st.code("\n".join(anomaly\_log))

#     if st.button("📧 Send Email Alert"):

#         sent = send\_email\_alert(anomaly\_log)

#         if sent:

#             st.success("📨 Email sent successfully!")

# st.markdown("---")

# st.subheader("📂 View Past Anomaly Logs")

# if os.path.exists(LOG\_DIR):

#     log\_files = sorted(os.listdir(LOG\_DIR), reverse=True)

#     selected\_log = st.selectbox("Select log file", log\_files)

#     if selected\_log:

#         with open(os.path.join(LOG\_DIR, selected\_log)) as f:

#             st.code(f.read(), language="text")

# else:

#     st.info("No logs yet.")

DATA:-PREPROCESSED DATA.CSV

ANOMOLY PREDICTION

CODE

ANOMOLYDETECTION.PY

import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import MinMaxScaler

from xgboost import XGBClassifier

from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score, mean\_squared\_error, r2\_score

import matplotlib.pyplot as plt

import seaborn as sns

import joblib

# STEP 1: Load the Data

df = pd.read\_csv("generated\_anomaly\_training\_data.csv")

df['RealtimeClockDateandTime'] = pd.to\_datetime(df['RealtimeClockDateandTime'])

# STEP 2: Feature Selection

features = ['Voltage', 'SystemPowerFactor', 'ActivePower\_kW', 'Frequency', 'BlockEnergykWh']

X = df[features]

y = df['Anomaly']

# STEP 3: Normalize the Features

scaler = MinMaxScaler()

X\_scaled = scaler.fit\_transform(X)

# STEP 4: Train-Test Split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X\_scaled, y, test\_size=0.2, random\_state=42, stratify=y

)

# STEP 5: Train XGBoost Classifier

model = XGBClassifier(

    scale\_pos\_weight=len(y\_train[y\_train == 0]) / len(y\_train[y\_train == 1]),

    use\_label\_encoder=False,

    eval\_metric='logloss'

)

model.fit(X\_train, y\_train)

# STEP 6: Predictions & Evaluation

y\_pred = model.predict(X\_test)

y\_prob = model.predict\_proba(X\_test)[:, 1]

print("Confusion Matrix:\n", confusion\_matrix(y\_test, y\_pred))

print("\nClassification Report:\n", classification\_report(y\_test, y\_pred))

accuracy = accuracy\_score(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_prob)

rmse = np.sqrt(mean\_squared\_error(y\_test, y\_prob))

print(f"\nAccuracy: {accuracy:.4f}")

print(f"R² Score (based on probabilities): {r2:.4f}")

print(f"RMSE (based on probabilities): {rmse:.4f}")

# STEP 7: Predict All

df['predicted'] = model.predict(X\_scaled)

# STEP 8: Voltage Anomaly Plot

plt.figure(figsize=(14, 6))

plt.plot(df['RealtimeClockDateandTime'], df['Voltage'], label='Voltage')

plt.scatter(df[df['predicted'] == 1]['RealtimeClockDateandTime'],

            df[df['predicted'] == 1]['Voltage'],

            color='red', label='Predicted Anomaly', s=10)

plt.title("Voltage with Predicted Anomalies")

plt.xlabel("Timestamp")

plt.ylabel("Voltage")

plt.legend()

plt.tight\_layout()

plt.grid(True)

plt.show()

# STEP 9: Actual vs Predicted Anomalies

plt.figure(figsize=(14, 6))

plt.plot(df['RealtimeClockDateandTime'], df['Anomaly'], label='Actual Anomaly', linestyle='--')

plt.plot(df['RealtimeClockDateandTime'], df['predicted'], label='Predicted Anomaly')

plt.title("Actual vs Predicted Anomalies")

plt.xlabel("Timestamp")

plt.ylabel("Anomaly")

plt.legend()

plt.tight\_layout()

plt.grid(True)

plt.show()

# STEP 10: Confusion Matrix Heatmap

cm = confusion\_matrix(y\_test, y\_pred)

plt.figure(figsize=(6, 5))

sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',

            xticklabels=['Normal', 'Anomaly'],

            yticklabels=['Normal', 'Anomaly'])

plt.xlabel("Predicted")

plt.ylabel("Actual")

plt.title("Confusion Matrix")

plt.tight\_layout()

plt.show()

# STEP 11: Save the Model

joblib.dump(model, 'anomaly\_xgboost\_model.pkl')

joblib.dump(scaler, 'feature\_scaler.pkl')

MODELS

amomaly\_xgboost\_model.pkl

feature\_scaler.pkl

STREAMLIT

faultprediction.py

# anomaly\_streamlit\_app.py

import streamlit as st

import pandas as pd

import numpy as np

import joblib

import matplotlib.pyplot as plt

# Load trained model and scaler

model = joblib.load("models/anomaly\_xgboost\_model.pkl")

scaler = joblib.load("models/feature\_scaler.pkl")

st.set\_page\_config(page\_title="🔍 Anomaly Detector", layout="wide")

st.title("⚡ Smart Meter Anomaly Detection App")

st.sidebar.header("📂 Upload Data")

uploaded\_file = st.sidebar.file\_uploader("Upload CSV", type=["csv"])

if uploaded\_file is not None:

    df = pd.read\_csv(uploaded\_file)

    df['RealtimeClockDateandTime'] = pd.to\_datetime(df['RealtimeClockDateandTime'])

    features = ['Voltage', 'SystemPowerFactor', 'ActivePower\_kW', 'Frequency', 'BlockEnergykWh']

    if all(col in df.columns for col in features):

        # Preprocess input

        X = df[features]

        X\_scaled = scaler.transform(X)

        # Predict anomalies

        df['Predicted'] = model.predict(X\_scaled)

        df['Anomaly\_Label'] = df['Predicted'].apply(lambda x: "🔴 Anomaly" if x == 1 else "🟢 Normal")

        # Show summary

        st.success(f"✅ Data Processed: {len(df)} records")

        st.write("### 📊 Sample Output", df.head())

        # Plot results

        st.write("### 🔍 Anomaly Timeline (Voltage)")

        fig, ax = plt.subplots(figsize=(14, 5))

        ax.plot(df['RealtimeClockDateandTime'], df['Voltage'], label='Voltage')

        ax.scatter(df[df['Predicted'] == 1]['RealtimeClockDateandTime'],

                   df[df['Predicted'] == 1]['Voltage'],

                   color='red', label='Anomalies', s=10)

        ax.set\_title("Voltage with Predicted Anomalies")

        ax.set\_xlabel("Time")

        ax.set\_ylabel("Voltage")

        ax.legend()

        ax.grid(True)

        st.pyplot(fig)

        # Download results

        csv\_download = df.to\_csv(index=False).encode('utf-8')

        st.download\_button("📥 Download Results CSV", csv\_download, "anomaly\_predictions.csv", "text/csv")

    else:

        st.error("❌ Required columns missing: Ensure your CSV includes Voltage, SystemPowerFactor, ActivePower\_kW, Frequency, BlockEnergykWh")

else:

    st.info("📁 Upload a CSV file with smart meter readings to begin.")

DATA:-generated\_anomally\_training\_data.csv