College of computing

Department of Software Engineering

Machine Learning Project Documentation

Name : Eden Getachew

Id :1401182

ATM Maintenance Prediction System

This documentation outlines the steps, technologies, and methods involved in building and deploying the ATM Maintenance Prediction system using machine learning and API deployment techniques. The system aims to predict the maintenance status of an ATM based on sensor data.

Table of Contents

Contents Page No-

ATM Maintenance Prediction System Documentation

❖ Problem Definition

The ATM Maintenance Prediction System aims to predict the likelihood of an ATM machine requiring maintenance based on sensor data. By analyzing historical data from ATMs, we can identify patterns that may indicate when maintenance is needed, potentially preventing unexpected downtime and improving operational efficiency.

❖ Data Source and Description

The dataset used for this project is sourced from the AI4I 2020 dataset, which includes various sensor readings from ATM machines. The columns in the dataset are as follows:

- Air temperature [K]: Air temperature inside the ATM machine.
- Process temperature [K]: Temperature related to the ATM's internal processes.
- Rotational speed [rpm]: Speed of the ATM's internal mechanical parts.
- Torque [Nm]: The force being applied to the rotating parts.
- Tool wear [min]: The wear time of the tools inside the ATM.
- Machine Type: Two machine types, Type_M and Type_L (encoded as 0 or 1).

❖ Exploratory Data Analysis (EDA)

Findings

- Missing Data: The dataset contains a few missing values, which were handled during preprocessing.
- Feature Correlation: Initial analysis shows that Air temperature and Process temperature are highly correlated.
- Target Variable Distribution: The target variable is binary, representing whether an ATM requires maintenance (failure = 1, no failure = 0). The distribution of the target variable is imbalanced, with a higher number of instances of no failure.

❖ Visualizations

- Correlation Matrix: Visualizes correlations between numerical features.
- Feature Distribution: Plots of features like air temperature, rotational speed, and torque to check their distribution.
- Class Imbalance: A bar plot showing the distribution of the target variable.

❖ Data Preprocessing

Steps

1. Handling Missing Data: Missing values were either filled using mean imputation or removed.

2. Feature Scaling: Numerical features such as Air temperature, Process temperature, and Torque were normalized using MinMax scaling to ensure they are on the same scale.

3. One-Hot Encoding: The categorical variables Type_M and Type_L were one-hot encoded into binary columns.

```
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder
import pandas as pd

# Load the data
```

```
data = pd.read_csv("atm_data.csv")

# Handle missing values
data.fillna(data.mean(), inplace=True)

# Scale numerical features
scaler = MinMaxScaler()
data[['Air temperature [K]', 'Process temperature [K]', 'Rotational speed [rpm]', 'Torque [Nm]', 'Tool wear [min]']] = scaler.fit_transform(data[['Air temperature [K]', 'Process temperature [K]', 'Rotational speed [rpm]', 'Torque [Nm]', 'Tool wear [min]']])

# One-hot encode categorical variables
data = pd.get_dummies(data, columns=['Machine Type'])

# Split the data into training and testing sets
X = data.drop('Failure', axis=1)
y = data['Failure']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

❖ Model Selection and Training

A Random Forest Classifier was chosen due to its ability to handle large datasets with high feature dimensionality and its capacity to model complex relationships between the input features.

❖ Training

• Train-Test Split: 80% of the data was used for training, and 20% for testing.

• Model Training: The Random Forest model was trained on the prepared data.

```
from sklearn.ensemble import RandomForestClassifier
import joblib

# Train the Random Forest model
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)

# Save the model for later use
joblib.dump(model, 'atm_rf_model.pkl')
```

❖ Model Evaluation

• Accuracy: The model achieved an accuracy of 98% on the test set.

• Confusion Matrix: The confusion matrix showed a high true positive rate for predicting failures and a low false positive rate.

• Classification Report: The model performed well with an F1 score of 0.97.

```
from sklearn.metrics import classification_report, confusion_matrix

# Make predictions on the test set
y_pred = model.predict(X_test)

# Evaluate the model
print(classification_report(y_test, y_pred))
print(confusion_matrix(y_test, y_pred))
```

❖ Interpretation of Results

• The model has a high accuracy rate, indicating it is effective at distinguishing between failure and non-failure instances.

• The classification report shows that both precision and recall for predicting failures are high, which is critical in a maintenance prediction scenario.

❖ Deployment Details and Instructions

FastAPI Backend

The model is served using FastAPI for real-time predictions via an API endpoint. The backend accepts data in JSON format and returns predictions indicating whether maintenance is required.

Install dependencies:

```
pip install fastapi uvicorn pandas scikit-learn joblib
```

```
from fastapi import FastAPI, HTTPException
import joblib
import pandas as pd
```

```
from pydantic import BaseModel

# Load model
model = joblib.load('atm_rf_model.pkl')

# Define FastAPI app
app = FastAPI()

class InputData(BaseModel):
    Air_temperature_K: float
    Process_temperature_K: float
    Rotational_speed_rpm: float
    Torque_Nm: float
    Tool_wear_min: float
    Type_M: int
    Type_L: int

@app.post("/predict")
def predict(data: InputData):
    input_data = pd.DataFrame([data.dict()])
    prediction = model.predict(input_data)
    prediction_label = "Failure" if prediction[0] == 1 else "No Failure"
    return {"prediction": prediction_label}
```

Running the FastAPI Server:

```
uvicorn api:app --reload
```

❖ Streamlit Frontend

The Streamlit frontend provides an interactive user interface where users can input sensor data and receive predictions from the FastAPI backend.

Install dependencies:

```
pip install streamlit requests
```

```
import streamlit as st
import requests
import json

# Define the API URL
```

```
api_url = "https://your-fastapi-url/predict"

# Streamlit interface for inputs
st.title("ATM Maintenance Prediction")

# Input fields
air_temperature = st.number_input('Air temperature [K]', min_value=0.0, value=300.0)
# More input fields...

input_data = {
   "Air_temperature_K": air_temperature,
   # Add other inputs
}

# Convert input data to JSON
input_json = json.dumps(input_data)

# Send data to FastAPI
if st.button('Predict'):
   response = requests.post(api_url, data=input_json, headers={'Content-Type':
'application/json'})
   if response.status_code == 200:
      result = response.json()
      st.write(f"Prediction: {result['prediction']}")
```

❖ Potential Limitations and Future Improvements

1. Imbalanced Dataset: The model may struggle with predicting maintenance failures in cases where there are few failure instances. Future work can focus on addressing class imbalance using techniques like SMOTE or weighted loss functions.

2. Feature Engineering: Adding more sensor features and improving feature engineering could help enhance model accuracy.

3. Model Interpretability: Using tools like SHAP or LIME could provide better insights into how the model makes predictions.

4. Real-Time Updates: Integrating real-time data collection from ATMs and updating the model periodically would improve its prediction accuracy over time.