**Project Documentation**

**Project Name : Sentiment Analysis API with MLOps Integration**

**Date: 13/03/2025**

**Overview**

This project is a sentiment analysis application using machine learning and deep learning models to classify text reviews as positive or negative. It consists of a training module, a web interface, and containerization for deployment.

**Tech Stack**

Windows 11

Programming: Python3. 9

Libraries: Scikit-learn, TensorFlow/PyTorch, FastAPI, Pandas, NumPy

Model Deployment: FastAPI, Docker

Monitoring: MLflow, Prometheus, Grafana

Virtual Environment: ml

**Downloads :**

1. Download IMDB dataset for training models.

2. Prometheus and Grafana to set up monitoring dashboards.

**Project Structure**

ML\_Model-Sentiment\_Analysis/

│── aclImdb/ # Dataset (Train/Test folders)

│── Accuracy-Reports/ # Model accuracy reports & figures

│── Models/ # Trained models (Logistic Regression, Naïve Bayes, LSTM)

│── Templates/ # Frontend (login/index.html)

│── logs/ # API request/response logs

│── mlruns/ # MLflow data tracking

│── mlflow\_metrics.py # MLflow tracking setup

│── prometheus.yml # Prometheus configuration

│── Dockerfile # Docker container setup

│── docker-compose.yml # Docker Compose configuration

│── requirements.txt # Required dependencies

│── train.py # Model training & evaluation

│── main.py # FastAPI main application (Prediction API)

│── .env # Authentication credentials

**File Descriptions**

1. **train\_model.py**

**Downloaded IMDB dataset**

This script is responsible for training sentiment analysis models using traditional machine learning and deep learning techniques.

**Key Features:**

* Loads and cleans IMDB review dataset.
* Applies TF-IDF vectorization for traditional models.
* Handles class imbalance using SMOTE.
* Trains **Logistic Regression** and **Naive Bayes** classifiers with TF-IDF features.
* Trains an **LSTM neural network** for deep learning-based classification.
* Evaluates models using accuracy, confusion matrices, and classification reports.
* Saves trained models and vectorizers.
* Generates performance plots and stores results in a CSV file.

**Code-**

import os

import re

import string

import joblib

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

import seaborn as sns

import nltk

import tensorflow as tf

from tensorflow.keras.models import Sequential, load\_model

from tensorflow.keras.layers import Embedding, LSTM, Dense, Dropout

from tensorflow.keras.preprocessing.text import Tokenizer

from tensorflow.keras.preprocessing.sequence import pad\_sequences

from nltk.corpus import stopwords

from imblearn.over\_sampling import SMOTE

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

from sklearn.feature\_extraction.text import TfidfVectorizer

from sklearn.linear\_model import LogisticRegression

from sklearn.naive\_bayes import MultinomialNB

from sklearn.metrics import accuracy\_score, classification\_report, confusion\_matrix, ConfusionMatrixDisplay

# Download stopwords if not already downloaded

nltk.download("stopwords")

stop\_words = set(stopwords.words("english"))

# Function to clean text

def clean\_text(text):

    text = text.lower()

    text = re.sub(r'\d+', '', text)  # Remove numbers

    text = text.translate(str.maketrans('', '', string.punctuation))  # Remove punctuation

    text = " ".join([word for word in text.split() if word not in stop\_words])  # Remove stopwords

    return text.strip()

# Load IMDB dataset from given directory (expects 'pos' and 'neg' subfolders)

def load\_imdb\_data(data\_dir):

    data = {"review": [], "sentiment": []}

    for sentiment in ["pos", "neg"]:

        folder\_path = os.path.join(data\_dir, sentiment)

        for filename in os.listdir(folder\_path):

            with open(os.path.join(folder\_path, filename), "r", encoding="utf-8") as file:

                review = file.read()

                data["review"].append(clean\_text(review))

                data["sentiment"].append(1 if sentiment == "pos" else 0)

    return pd.DataFrame(data)

# Set paths (modify these paths to your local directories)

train\_path = r"aclImdb\train"

test\_path = r"aclImdb\test"

# Load datasets

df\_train = load\_imdb\_data(train\_path)

df\_test = load\_imdb\_data(test\_path)

# Remove duplicates & missing values

df\_train.drop\_duplicates(inplace=True)

df\_train.dropna(inplace=True)

df\_test.drop\_duplicates(inplace=True)

df\_test.dropna(inplace=True)

# Exploratory Data Analysis (EDA)

plt.figure(figsize=(6, 4))

sns.countplot(x=df\_train["sentiment"])

plt.title("Class Distribution (Train Data)")

plt.show()

print("\nDataset Info:")

print(df\_train.info())

print("\nSentiment Distribution:\n", df\_train["sentiment"].value\_counts())

# Split train data (80% train, 20% validation)

X\_train, X\_val, y\_train, y\_val = train\_test\_split(df\_train["review"], df\_train["sentiment"], test\_size=0.2, random\_state=42)

# TF-IDF Vectorization (for traditional models)

vectorizer = TfidfVectorizer(max\_features=5000, stop\_words="english")

X\_train\_tfidf = vectorizer.fit\_transform(X\_train)

X\_val\_tfidf = vectorizer.transform(X\_val)

X\_test\_tfidf = vectorizer.transform(df\_test["review"])

# Handle class imbalance using SMOTE (for TF-IDF based training)

# Note: We create new variables so that the original y\_train remains unchanged for LSTM

smote = SMOTE(random\_state=42)

X\_train\_tfidf\_res, y\_train\_tfidf\_res = smote.fit\_resample(X\_train\_tfidf, y\_train)

# Train Logistic Regression with GridSearchCV

param\_grid = {"C": [0.01, 0.1, 1, 10, 100]}

grid\_search = GridSearchCV(LogisticRegression(max\_iter=1000, class\_weight="balanced"), param\_grid, cv=3, scoring="accuracy")

grid\_search.fit(X\_train\_tfidf\_res, y\_train\_tfidf\_res)

best\_lr\_model = grid\_search.best\_estimator\_

# Train Naive Bayes Model

nb\_model = MultinomialNB()

nb\_model.fit(X\_train\_tfidf\_res, y\_train\_tfidf\_res)

# Tokenization & Padding for LSTM (using original X\_train and X\_val)

tokenizer = Tokenizer(num\_words=10000, oov\_token="<OOV>")

tokenizer.fit\_on\_texts(X\_train)

X\_train\_seq = pad\_sequences(tokenizer.texts\_to\_sequences(X\_train), maxlen=200, padding="post")

X\_val\_seq = pad\_sequences(tokenizer.texts\_to\_sequences(X\_val), maxlen=200, padding="post")

X\_test\_seq = pad\_sequences(tokenizer.texts\_to\_sequences(df\_test["review"]), maxlen=200, padding="post")

# Build and Train LSTM Model

lstm\_model = Sequential([

    Embedding(input\_dim=10000, output\_dim=128, input\_length=200),

    LSTM(128, return\_sequences=True),

    Dropout(0.3),

    LSTM(64),

    Dropout(0.3),

    Dense(1, activation="sigmoid")

])

lstm\_model.compile(loss="binary\_crossentropy", optimizer="adam", metrics=["accuracy"])

lstm\_history = lstm\_model.fit(X\_train\_seq, y\_train, epochs=5, batch\_size=64, validation\_data=(X\_val\_seq, y\_val))

# Function to plot confusion matrix

def plot\_confusion\_matrix(y\_true, y\_pred, model\_name):

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

    disp = ConfusionMatrixDisplay(confusion\_matrix=cm)

    disp.plot(cmap="Blues")

    plt.title(f"Confusion Matrix - {model\_name}")

    plt.show()

# Initialize list to store model evaluation results

comparison\_results = []

# Evaluate Logistic Regression & Naive Bayes

models = {"Logistic Regression": best\_lr\_model, "Naive Bayes": nb\_model}

for name, model in models.items():

    y\_val\_pred = model.predict(X\_val\_tfidf)

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

    val\_acc = accuracy\_score(y\_val, y\_val\_pred)

    test\_acc = accuracy\_score(df\_test["sentiment"], y\_test\_pred)

    comparison\_results.append({"Model": name, "Validation Accuracy": val\_acc, "Test Accuracy": test\_acc})

    # Save classification report

    with open(f"{name}\_classification\_report.txt", "w") as f:

        f.write(f"Validation Accuracy: {val\_acc}\n")

        f.write(f"Test Accuracy: {test\_acc}\n")

        f.write("Validation Report:\n")

        f.write(classification\_report(y\_val, y\_val\_pred))

        f.write("\nTest Report:\n")

        f.write(classification\_report(df\_test["sentiment"], y\_test\_pred))

    # Plot confusion matrix for this model

    plot\_confusion\_matrix(y\_val, y\_val\_pred, name)

# Evaluate LSTM Model

lstm\_val\_pred = (lstm\_model.predict(X\_val\_seq) > 0.5).astype("int32")

lstm\_test\_pred = (lstm\_model.predict(X\_test\_seq) > 0.5).astype("int32")

lstm\_val\_acc = accuracy\_score(y\_val, lstm\_val\_pred)

lstm\_test\_acc = accuracy\_score(df\_test["sentiment"], lstm\_test\_pred)

comparison\_results.append({"Model": "LSTM", "Validation Accuracy": lstm\_val\_acc, "Test Accuracy": lstm\_test\_acc})

# Save LSTM Classification Report

with open("LSTM\_classification\_report.txt", "w") as f:

    f.write(f"Validation Accuracy: {lstm\_val\_acc}\n")

    f.write(f"Test Accuracy: {lstm\_test\_acc}\n")

    f.write("Validation Report:\n")

    f.write(classification\_report(y\_val, lstm\_val\_pred))

    f.write("\nTest Report:\n")

    f.write(classification\_report(df\_test["sentiment"], lstm\_test\_pred))

# Plot confusion matrix for LSTM

plot\_confusion\_matrix(y\_val, lstm\_val\_pred, "LSTM")

# Plot LSTM training accuracy & loss curves

plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)

plt.plot(lstm\_history.history["accuracy"], label="Train Accuracy")

plt.plot(lstm\_history.history["val\_accuracy"], label="Validation Accuracy")

plt.xlabel("Epochs")

plt.ylabel("Accuracy")

plt.title("LSTM Model Accuracy")

plt.legend()

plt.subplot(1, 2, 2)

plt.plot(lstm\_history.history["loss"], label="Train Loss")

plt.plot(lstm\_history.history["val\_loss"], label="Validation Loss")

plt.xlabel("Epochs")

plt.ylabel("Loss")

plt.title("LSTM Model Loss")

plt.legend()

plt.show()

# Save Comparison Table as CSV

comparison\_df = pd.DataFrame(comparison\_results)

comparison\_df.to\_csv("Accuracy-Reprts/model\_comparison\_results.csv", index=False)

print("\nComparison Results:\n", comparison\_df)

# Save Models & Vectorizer

joblib.dump(best\_lr\_model, "Models/logistic\_regression\_model.pkl")   # Saved using Joblib

joblib.dump(nb\_model, "Models/naive\_bayes\_model.pkl")                  # Saved using Joblib

joblib.dump(vectorizer, "Models/tfidff\_vectorizer.pkl")                 # Saved using Joblib

# Save tokenizer as JSON

tokenizer\_json = tokenizer.to\_json()

with open("Models/tokenizer.json", "w", encoding="utf-8") as f:

    f.write(tokenizer\_json)

# Save LSTM model using Keras native save (HDF5 format)

lstm\_model.save("Models/lstm\_sentiment\_model.h5")

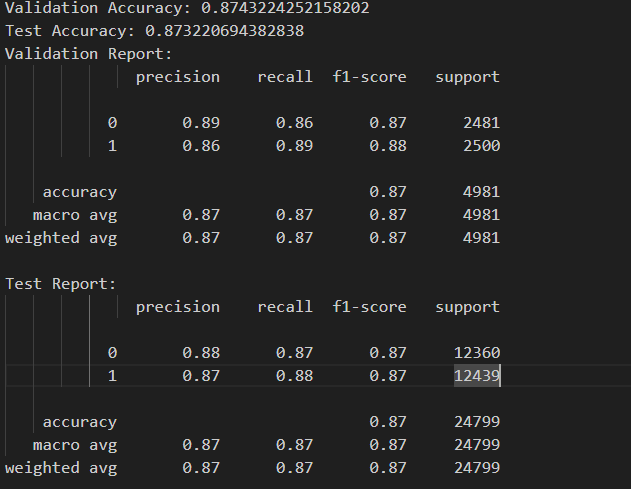
print("\n✅ All models, reports, plots, and comparison results saved successfully!")

**Outputs:**

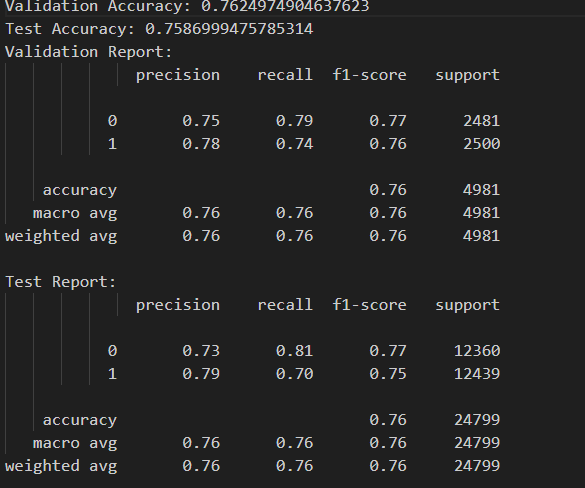
* Trained models: logistic\_regression\_model.pkl, naive\_bayes\_model.pkl, lstm\_sentiment\_model.h5
* Tokenizer: tokenizer.json
* Vectorizer: tfidff\_vectorizer.pkl
* Evaluation reports and plots
* Comparison results in model\_comparison\_results.csv

Reports:

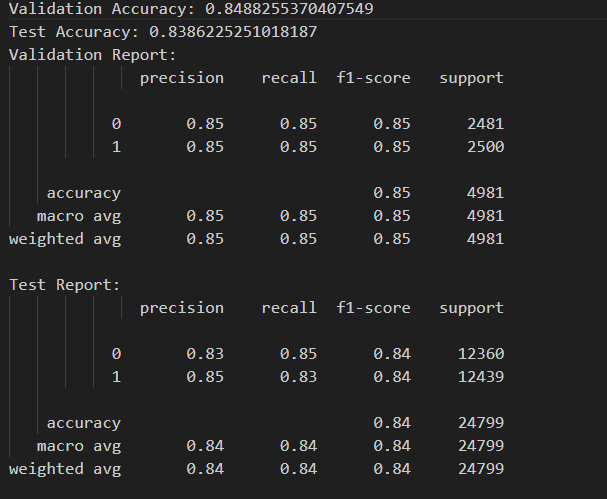
1. **Logistic Regression\_classification\_report.txt**

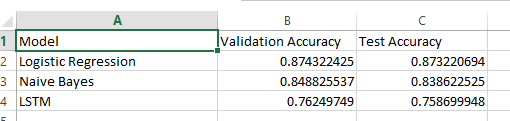
****

1. **LSTM\_classification\_report.txt**



1. **Naive Bayes\_classification\_report.txt**



1. **model\_comparison\_results.csv** 
2. **main.py**

Download grafana and Prometheus for monitoring dashboards

This script sets up a **FastAPI** web server to handle user input for sentiment analysis.

**Key Features:**

* Loads trained models (logistic\_regression\_model.pkl, naive\_bayes\_model.pkl, lstm\_sentiment\_model.h5).
* Uses **Jinja2** templates for rendering web pages.
* Accepts user input, preprocesses it, and predicts sentiment.
* Supports both traditional and deep learning models.
* Displays results on a web interface.

**Endpoints:**

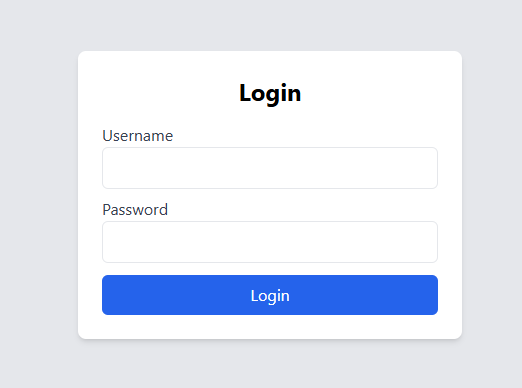
* / (Home page)
* /predict (Handles prediction requests)
* #main.py file                                                                                                                                                       from fastapi import FastAPI, Request, Form
* from fastapi.templating import Jinja2Templates
* from fastapi.responses import RedirectResponse
* import joblib
* import os
* import numpy as np
* import tensorflow as tf
* from tensorflow.keras.preprocessing.sequence import pad\_sequences
* from tensorflow.keras.preprocessing.text import tokenizer\_from\_json
* from dotenv import load\_dotenv
* from loguru import logger
* import mlflow
* from prometheus\_fastapi\_instrumentator import Instrumentator
* from starlette.middleware.sessions import SessionMiddleware
* # Load environment variables
* load\_dotenv()
* # Initialize FastAPI app
* app = FastAPI()
* # Add session middleware
* app.add\_middleware(SessionMiddleware, secret\_key="your\_secret\_key\_here")
* # Set up templates directory
* templates = Jinja2Templates(directory="templates")
* # Load credentials from .env
* VALID\_USERNAME = os.getenv("API\_USERNAME")
* VALID\_PASSWORD = os.getenv("API\_PASSWORD")
* # ✅ Initialize Loguru logging
* logger.add("logs/api.log", rotation="1 day", level="INFO", format="{time} {level} {message}")
* # ✅ Prometheus Monitoring
* Instrumentator().instrument(app).expose(app, endpoint="/metrics")
* # Model paths
* MODEL\_DIR = "Models"
* LR\_MODEL\_PATH = os.path.join(MODEL\_DIR, "logistic\_regression\_model.pkl")
* NB\_MODEL\_PATH = os.path.join(MODEL\_DIR, "naive\_bayes\_model.pkl")
* VECTORIZER\_PATH = os.path.join(MODEL\_DIR, "tfidff\_vectorizer.pkl")
* TOKENIZER\_PATH = os.path.join(MODEL\_DIR, "tokenizer.json")
* LSTM\_MODEL\_PATH = os.path.join(MODEL\_DIR, "lstm\_sentiment\_model.h5")
* # Load models safely
* try:
* lr\_model = joblib.load(LR\_MODEL\_PATH)
* nb\_model = joblib.load(NB\_MODEL\_PATH)
* vectorizer = joblib.load(VECTORIZER\_PATH)
* with open(TOKENIZER\_PATH, "r", encoding="utf-8") as f:
* tokenizer\_json = f.read()
* tokenizer = tokenizer\_from\_json(tokenizer\_json)
* lstm\_model = tf.keras.models.load\_model(LSTM\_MODEL\_PATH)
* logger.info("✅ All models loaded successfully!")
* except Exception as e:
* logger.error(f"❌ Error loading models: {e}")
* raise RuntimeError("Model loading failed. Check model files.")
* # Preprocess text
* def preprocess\_text(text):
* return text.lower().strip()
* # Prediction for Logistic Regression & Naive Bayes
* def predict\_tfidf(text, model):
* try:
* text\_tfidf = vectorizer.transform([text])
* prediction = model.predict(text\_tfidf)[0]
* return "Positive" if prediction == 1 else "Negative"
* except Exception as e:
* logger.error(f"❌ Error in predict\_tfidf: {e}")
* return "Prediction Failed"
* # Prediction for LSTM
* def predict\_lstm(text):
* try:
* text\_seq = pad\_sequences(tokenizer.texts\_to\_sequences([text]), maxlen=200, padding="post")
* prediction = lstm\_model.predict(text\_seq)
* return "Positive" if prediction[0][0] > 0.4 else "Negative"
* except Exception as e:
* logger.error(f"❌ Error in predict\_lstm: {e}")
* return "Prediction Failed"
* # ✅ MLflow Logging Function
* def log\_prediction(model\_name, text, prediction):
* mlflow.set\_tracking\_uri("http://mlflow:5000")  # Use MLflow container name
* mlflow.set\_experiment("Sentiment\_Analysis")
* with mlflow.start\_run():
* mlflow.log\_param("model", model\_name)
* mlflow.log\_param("input\_text", text)
* mlflow.log\_metric("prediction", 1 if prediction == "Positive" else 0)
* logger.info(f"📊 MLflow Logged: Model={model\_name}, Prediction={prediction}")
* # ✅ Middleware for Logging API Requests
* @app.middleware("http")
* async def log\_requests(request: Request, call\_next):
* logger.info(f"📩 Request: {request.method} {request.url}")
* response = await call\_next(request)
* logger.info(f"📤 Response: {response.status\_code}")
* return response
* # Authentication Function
* def is\_authenticated(request: Request):
* return request.session.get("authenticated", False)
* # Login Page
* @app.get("/login")
* def login\_page(request: Request):
* return templates.TemplateResponse("login.html", {"request": request})
* @app.post("/login")
* def login(request: Request, username: str = Form(...), password: str = Form(...)):
* if username == VALID\_USERNAME and password == VALID\_PASSWORD:
* request.session["authenticated"] = True
* return RedirectResponse(url="/", status\_code=303)
* return templates.TemplateResponse("login.html", {"request": request, "error": "Invalid credentials"})
* # Logout Route
* @app.get("/logout")
* def logout(request: Request):
* request.session.clear()
* return RedirectResponse(url="/login", status\_code=303)
* # Home Page - Render Input Form (Authenticated Users Only)
* @app.get("/")
* def home(request: Request):
* if not is\_authenticated(request):
* return RedirectResponse(url="/login")
* return templates.TemplateResponse("index.html", {"request": request, "prediction": None})
* # Handle Form Submission & Show Predictions
* @app.post("/")
* def predict(request: Request, text: str = Form(...)):
* if not is\_authenticated(request):
* return RedirectResponse(url="/login")
* processed\_text = preprocess\_text(text)
* predictions = {
* "Logistic Regression": predict\_tfidf(processed\_text, lr\_model),
* "Naive Bayes": predict\_tfidf(processed\_text, nb\_model),
* "LSTM": predict\_lstm(processed\_text)
* }
* for model, pred in predictions.items():
* log\_prediction(model, processed\_text, pred)
* return templates.TemplateResponse("index.html", {"request": request, "text": text, "prediction": predictions})

**3. login.html**

A **login page** designed with HTML for user authentication.

**Features:**

* Contains a form for user login.
* Sends authentication details to the backend.
* Uses Bootstrap for styling.

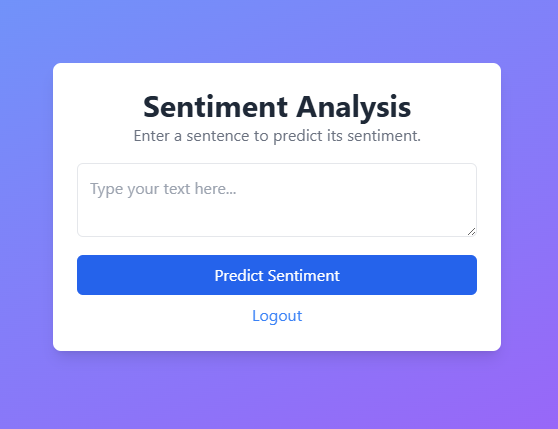


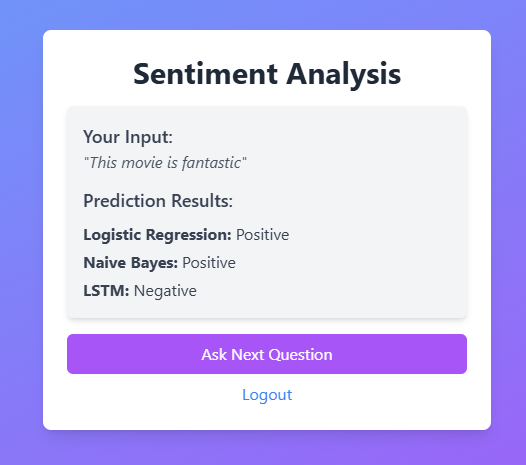
**4. index.html**

A **home page** that allows users to enter text and classify sentiment.

**Features:**

* Accepts user input (text review).
* Displays sentiment classification results.
* Provides an interface for switching between machine learning models.





### For Monitoring & Logging

To Track model performance I have used **MLflow**

For monitoring dashboards I have used **Prometheus and Grafana**.

* 1. Add mlflow\_metrics.py file

from flask import Flask

from prometheus\_flask\_exporter import PrometheusMetrics

app = Flask(\_\_name\_\_)

metrics = PrometheusMetrics(app)

# Custom metrics

metrics.info("mlflow\_server", "MLflow Tracking Server Metrics")

@app.route("/")

def index():

    return "MLflow Prometheus Exporter Running"

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

    app.run(host="0.0.0.0", port=5001)

* 1. Add Prometheus.yml
* scrape\_configs:
* - job\_name: "fastapi"
* static\_configs:
* - targets: ["fastapi-app:8080"]
* - job\_name: "mlflow"
* static\_configs:
* - targets: ["mlflow\_server:5000"]
* - job\_name: "mlflow\_metrics"
* static\_configs:
* - targets: ["mlflow\_server:5001"]  # Expose custom MLflow metrics
* - job\_name: "prometheus"
* static\_configs:
* - targets: ["prometheus:9090"]

**Test main.py**

* Main app - curl -X POST "http://127.0.0.1:8080/" -H "Content-Type: application/x-www-form-urlencoded" -u admin:password123 -d "text=I love this movie!"
* mlflow - curl -i http://localhost:8080
* grafana - curl -i http://localhost:3001
* prometheus - curl -i http://localhost:9090

**5. Dockerfile**

Defines the **Docker container** setup for deploying the application.

**Key Features:**

* Uses python:3.9 slim as the base image.
* Installs required dependencies from requirements.txt.
* Copies application files into the container.
* Exposes FastAPI server on port 8080.
* Runs the FastAPI application inside a container.

Codes

**Dockerfile**

# Use a lightweight Python image

FROM python:3.9-slim

# Set working directory inside the container

WORKDIR /app

# Copy the requirements file and install dependencies

COPY requirements.txt .

# Install dependencies without '--no-cache-dir'

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

# Copy the entire application

COPY . .

# Expose port 8080 for FastAPI

EXPOSE 8080

#without relosd for production

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8080"]

**Docker-compose.yml**

version: "3.8"

services:

  fastapi-app:

    build: .

    container\_name: fastapi\_app

    ports:

      - "8080:8080"

    depends\_on:

      - mlflow

      - prometheus

    environment:

      API\_USERNAME: admin

      API\_PASSWORD: password123

      MLFLOW\_TRACKING\_URI: http://mlflow:5000

    volumes:

      - ./logs:/app/logs

    healthcheck:

      test: ["CMD", "curl", "-f", "http://localhost:8080/"]

      interval: 30s

      retries: 3

  mlflow:

    image: ghcr.io/mlflow/mlflow:v2.5.0

    container\_name: mlflow\_server

    command: >

      sh -c "pip install prometheus-flask-exporter &&

             mlflow server --backend-store-uri sqlite:///mlflow.db --host 0.0.0.0 --port 5000"

    ports:

      - "5000:5000"

    environment:

      MLFLOW\_BACKEND\_STORE\_URI: sqlite:///mlflow.db

      MLFLOW\_ARTIFACT\_ROOT: /mlflow/artifacts

    volumes:

      - ./mlflow\_data:/mlflow

  prometheus:

    image: prom/prometheus

    container\_name: prometheus\_server

    ports:

      - "9090:9090"

    volumes:

      - ./prometheus.yml:/etc/prometheus/prometheus.yml

    command:

      - "--config.file=/etc/prometheus/prometheus.yml"

      - "--web.listen-address=:9090"

  grafana:

    image: grafana/grafana

    container\_name: grafana\_server

    ports:

      - "3001:3000"

    depends\_on:

      - prometheus

    environment:

      GF\_SECURITY\_ADMIN\_USER: admin

      GF\_SECURITY\_ADMIN\_PASSWORD: admin

    volumes:

      - grafana-data:/var/lib/grafana

volumes:

  grafana-data:

  mlflow\_data:

global:

  scrape\_interval: 15s  # Set the default scrape interval

scrape\_configs:

  - job\_name: "fastapi"

    static\_configs:

      - targets: ["fastapi-app:8080"]

  - job\_name: "mlflow"

    static\_configs:

      - targets: ["mlflow\_server:5000"]

  - job\_name: "mlflow\_metrics"

    static\_configs:

      - targets: ["mlflow\_server:5001"]  # Expose custom MLflow metrics

  - job\_name: "prometheus"

    static\_configs:

      - targets: ["prometheus:9090"]

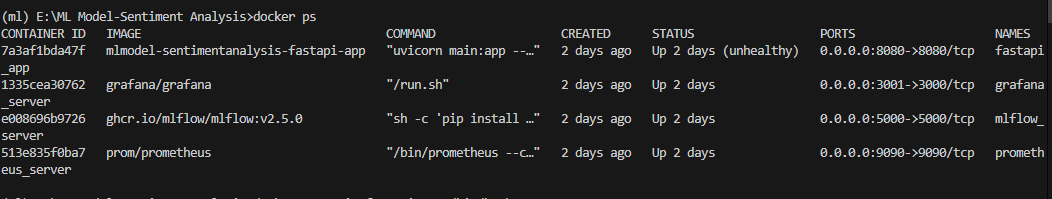
**Commands:**

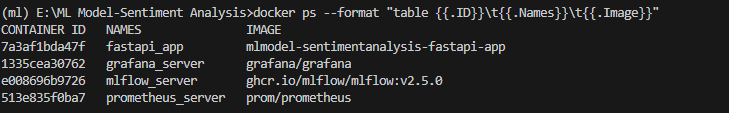
# Build the Docker image

* docker build -t mlmodel-sentimentanalysis-fastapi-app .

# Build and Run the container

* docker-compose up –build
* docker run -d -p 8080:8080 --name fastapi\_app mlmodel-sentimentanalysis-fastapi-app





**Docker Images**

All images are publicly available on Docker Hub. You can pull and run them using the following commands:

**1. Sentiment Analysis API**

docker pull pv271994/mlmodel-sentimentanalysis-fastapi-app

docker run -p 8080:8080 pv271994/mlmodel-sentimentanalysis-fastapi-app

**2. MLflow**

docker pull pv271994/mlflow

docker run -p 5000:5000 pv271994/mlflow

**3. Prometheus**

docker pull pv271994/prometheus

docker run -p 9090:9090 pv271994/prometheus

**4. Grafana**

docker pull pv271994/grafana

docker run -p 3001:3000 pv271994/grafana

Alternatively, use Docker Compose to run all services with one command:

docker-compose up --build -d

**6 Access UI Services**

Service URL

* FastAPI http://localhost:8080
* MLflow http://localhost:5000
* Grafana http://localhost:3001
* Prometheus <http://localhost:9090>

**7 Set up a GitHub repository with version control.**

* git init
* git remote add origin
* git remote add origin <https://github.com/pvlex27/MLOps-Sentiment-Analysis.git>
* git add .
* git commit -m "Initial commit with Docker and monitoring setup"
* git branch -M main
* git push -u origin main

**Deployment**

**Steps:**

1. **Train models** using train.py.
2. **Start FastAPI server** with main.py.
3. **Run web application** and classify user input.
4. **Deploy using Docker** for easy scalability.

**Conclusion**

This project provides a comprehensive sentiment analysis system using both traditional and deep learning models. It integrates a web interface and containerization for easy deployment and accessibility.

Future Enhancements

* 1. CI/CD Pipeline Integration
  2. Cloud Deployment (AWS/GCP/Azure)
  3. Auto-scaling with Kubernetes