App.py

# Import necessary libraries

import pandas as pd

import numpy as np

import tensorflow as tf

import re

from numpy import array

from keras.preprocessing.text import one\_hot, Tokenizer

from keras.models import Sequential, load\_model

from keras.layers import LSTM

from keras.layers.core import Activation, Dropout, Dense

from keras.layers import Flatten, GlobalMaxPooling1D, Embedding, Conv1D, LSTM

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

from fastapi import FastAPI, Request

from keras\_preprocessing.sequence import pad\_sequences

import nltk

from nltk.corpus import stopwords

from keras\_preprocessing.text import tokenizer\_from\_json

import io

import json

from pydantic import BaseModel

from typing import List

# Initialize FastAPI app

app = FastAPI()

# Stopwords list

stopwords\_list = set(stopwords.words('english'))

maxlen = 100

# Load model

model\_path = 'c1\_lstm\_model\_acc\_0.864.h5'

pretrained\_lstm\_model = load\_model(model\_path)

# Loading tokenizer

with open('b3\_tokenizer.json') as f:

data = json.load(f)

loaded\_tokenizer = tokenizer\_from\_json(data)

# Custom preprocessing

from b2\_preprocessing\_function import CustomPreprocess

custom = CustomPreprocess()

# Define the Pydantic model for input data

class ReviewRequest(BaseModel):

reviews: List[str]

# Define FastAPI routes

@app.get("/")

def read\_root():

return {"message": "Welcome to the LSTM Sentiment Analysis API!"}

@app.post("/predict/")

async def predict(request\_data: ReviewRequest):

query\_asis = request\_data.reviews

# Preprocess review text using the custom preprocessing function

query\_processed\_list = []

for query in query\_asis:

query\_processed = custom.preprocess\_text(query)

query\_processed\_list.append(query\_processed)

# Tokenize the instance using the loaded tokenizer

query\_tokenized = loaded\_tokenizer.texts\_to\_sequences(query\_processed\_list)

# Padding to have a max length of 100 tokens

query\_padded = pad\_sequences(query\_tokenized, padding='post', maxlen=maxlen)

# Passing tokenized instance to the LSTM model for predictions

query\_sentiments = pretrained\_lstm\_model.predict(query\_padded)

# Generate response for each review

results = []

for sentiment in query\_sentiments:

if sentiment[0] > 0.5:

results.append(f"Positive Review with probable IMDb rating as: {np.round(sentiment[0] \* 10, 1)}")

else:

results.append(f"Negative Review with probable IMDb rating as: {np.round(sentiment[0] \* 10, 1)}")

return {"predictions": results}

# To run the FastAPI app, use: `uvicorn script\_name:app --reload`

Create a file named Dockerfile in the same directory as your FastAPI app with the following content:

# Use an official Python runtime as a parent image

FROM python:3.9-slim

# Set the working directory in the container

WORKDIR /app

# Copy the current directory contents into the container at /app

COPY . /app

# Install any needed packages specified in requirements.txt

RUN pip install --no-cache-dir --upgrade pip \

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

# Expose the port FastAPI will run on

EXPOSE 8000

# Command to run the FastAPI server

CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000", "--reload"]

### Create the requirements.txt file

You need to list all the Python dependencies in this file. Run the following command to generate requirements.txt:

bash

Copy code

pip freeze > requirements.txt

Make sure requirements.txt includes packages like fastapi, uvicorn, tensorflow, nltk, etc.

If you want, you can manually create a requirements.txt like this:

Copy code

fastapi

uvicorn

pandas

numpy

tensorflow

nltk

keras

scikit-learn

### 3. Building and Running the Docker Image

#### Build the Docker image:

In the terminal, navigate to the directory with your Dockerfile and run the following command to build your Docker image:

bash

Copy code

docker build -t fastapi-lstm-app .

#### Run the Docker container:

Once the image is built, you can run it with:

bash

Copy code

docker run -p 8000:8000 fastapi-lstm-app

This will run the FastAPI app inside a Docker container and expose it at http://localhost:8000.

### 4. Access the FastAPI app:

Now, open your browser or use a tool like Postman to access your FastAPI app at:

arduino

Copy code

http://localhost:8000

You can interact with the FastAPI Swagger UI for testing the API at:

bash

Copy code

http://localhost:8000/docs

### Notes:

* Replace main:app in the CMD line of the Dockerfile with the correct file name and app instance (for example, if your FastAPI app is in a file named app.py, change it to app:app).
* Ensure you include nltk downloads in your app if needed (e.g., nltk.download('stopwords') during the app initialization phase).

To integrate Docker and FastAPI for Continuous Integration (CI) using GitHub Actions, you'll need to create a workflow configuration file in your GitHub repository. This configuration file will automate the process of building and testing the Docker image whenever a change is pushed to the repository.

### Steps to Integrate CI using GitHub Actions:

1. **Create the** .github/workflows/ci.yml **file**.
2. **Set up the GitHub Actions workflow to build and test the Docker image**.

### 1. Create the .github/workflows/ci.yml file

First, create a new directory in your repository called .github/workflows (if it doesn't already exist) and then create a file called ci.yml inside that directory. This YAML file will define the steps for GitHub Actions.

yaml

Copy code

name: CI for FastAPI Docker App

# This runs on every push or pull request to the main branchon:

push:

branches:

- main

pull\_request:

branches:

- main

jobs:

build:

runs-on: ubuntu-latest

steps:

# Checkout the repository containing the FastAPI code

- name: Checkout code

uses: actions/checkout@v2

# Set up Python

- name: Set up Python

uses: actions/setup-python@v3

with:

python-version: '3.9'

# Install Python dependencies (to test code before Docker build)

- name: Install dependencies

run: |

python -m pip install --upgrade pip

pip install -r requirements.txt

# Run tests (if you have any)

- name: Run tests

run: |

# Example command, replace with your actual test framework

pytest

# Set up Docker Buildx (this allows building multi-platform Docker images)

- name: Set up Docker Buildx

uses: docker/setup-buildx-action@v2

# Log in to DockerHub (optional, only if you want to push images to DockerHub)

- name: Log in to DockerHub

uses: docker/login-action@v2

with:

username: ${{ secrets.DOCKER\_USERNAME }}

password: ${{ secrets.DOCKER\_PASSWORD }}

# Build the Docker image

- name: Build Docker image

run: |

docker build -t fastapi-lstm-app .

# Push the Docker image to DockerHub (optional)

- name: Push Docker image

run: |

docker tag fastapi-lstm-app:latest ${{ secrets.DOCKER\_USERNAME }}/fastapi-lstm-app:latest

docker push ${{ secrets.DOCKER\_USERNAME }}/fastapi-lstm-app:latest if: github.event\_name == 'push'

# Run the Docker container to ensure everything works as expected

- name: Run Docker container

run: |

docker run -d -p 8000:8000 fastapi-lstm-app

### 2. Explanation of the Workflow:

* **Name**: This defines the name of the workflow.
* **Triggers**: The workflow will run on a push or pull request event targeting the main branch.
* **Jobs**:
  + **Build**: The job will run on an ubuntu-latest machine.
  + **Checkout Code**: This step checks out the repository code into the GitHub Actions runner.
  + **Set Up Python**: Installs Python and the necessary dependencies from requirements.txt.
  + **Run Tests**: This step runs tests using pytest (replace with your test command).
  + **Set Up Docker Buildx**: Docker Buildx is set up to build multi-platform images.
  + **Log in to DockerHub**: If you want to push the Docker image to DockerHub, log in using secrets stored in GitHub (DOCKER\_USERNAME and DOCKER\_PASSWORD).
  + **Build Docker Image**: This builds the Docker image defined in the Dockerfile.
  + **Push Docker Image (optional)**: Pushes the image to DockerHub if a push event occurs on the main branch.
  + **Run Docker Container**: This step verifies the container runs successfully by mapping port 8000.

### 3. Add Secrets in GitHub (if pushing to DockerHub)

If you want to push the Docker image to DockerHub, you’ll need to store your DockerHub credentials in GitHub Secrets.

1. Go to your repository on GitHub.
2. Click on Settings → Secrets and variables → Actions → New repository secret.
3. Add the following secrets:
   * DOCKER\_USERNAME: Your DockerHub username.
   * DOCKER\_PASSWORD: Your DockerHub password.

### 4. Push to GitHub

After setting up the GitHub Actions workflow and adding DockerHub credentials (if required), commit and push your code to GitHub. The workflow will automatically run on every push or pull request to the main branch.

### Summary:

* **Docker Image Build**: The workflow automates the Docker image build.
* **Testing**: Tests the Python dependencies and application.
* **CI Pipeline**: Automatically builds and optionally pushes Docker images using GitHub Actions, providing Continuous Integration for your FastAPI application.

4o

Yes, you can set up a **CI/CD pipeline using GitHub Actions** to deploy your **machine learning model** directly to AWS SageMaker without using a Docker image. This approach allows you to automate the process of training, updating, and deploying your model to SageMaker.

In this workflow, GitHub Actions will trigger the deployment whenever changes are pushed to the repository, automating the retraining of your machine learning model and its deployment to AWS SageMaker.

### Steps for CI/CD Deployment to AWS SageMaker:

1. **Prepare your Machine Learning Model**.
2. **Create a GitHub Actions Workflow for CI/CD**.
3. **Deploy Model to AWS SageMaker via GitHub Actions**.
4. **Configure AWS Credentials for GitHub Actions**.
5. **Monitor and Test the Deployment**.

### 1. Prepare Your Machine Learning Model

Your machine learning model script should be ready for training and deployment to AWS SageMaker. Typically, this includes:

* A script to **train the model** (e.g., train.py).
* A **model artifact** (e.g., saved as .pkl or .h5 file).
* The **inference code** (SageMaker handles how the model is used for inference).

You should ensure that your model can be trained using **SageMaker's built-in algorithms**, custom **training scripts** (e.g., in TensorFlow, PyTorch), or SageMaker's **SDK**.

### 2. Create a GitHub Actions Workflow for CI/CD

Here is an example of a CI/CD pipeline using GitHub Actions for deploying a machine learning model to AWS SageMaker:

#### .github/workflows/sagemaker-deploy.yml

yaml

Copy code

name: SageMaker Model Deployment

# Trigger this workflow on push to the main branch or on pull requestson:

push:

branches:

- main

pull\_request:

branches:

- main

jobs:

deploy:

runs-on: ubuntu-latest

steps:

# Step 1: Check out the code

- name: Checkout code

uses: actions/checkout@v2

# Step 2: Set up Python environment

- name: Set up Python

uses: actions/setup-python@v2

with:

python-version: '3.8'

# Step 3: Install dependencies (SageMaker SDK, boto3, etc.)

- name: Install dependencies

run: |

pip install --upgrade pip

pip install -r requirements.txt # Ensure you have SageMaker and AWS SDK in this file

# Step 4: Configure AWS credentials

- name: Configure AWS credentials

uses: aws-actions/configure-aws-credentials@v2

with:

aws-access-key-id: ${{ secrets.AWS\_ACCESS\_KEY\_ID }}

aws-secret-access-key: ${{ secrets.AWS\_SECRET\_ACCESS\_KEY }}

aws-region: us-east-1 # Update the region based on your setup

# Step 5: Upload training data to S3 (if needed)

- name: Upload data to S3

run: |

aws s3 cp ./data/ s3://your-s3-bucket/data/ --recursive

# Step 6: Train the model on SageMaker

- name: Train the model on SageMaker

run: |

python train.py # You can call a training script that triggers SageMaker training

# Step 7: Deploy the model to SageMaker

- name: Deploy model to SageMaker

run: |

python deploy\_model.py # Script to create a SageMaker endpoint

# Step 8: Invalidate SageMaker endpoint (optional)

- name: Invalidate endpoint (Optional)

run: |

# Optionally, stop the existing endpoint or create new versions

python invalidate\_endpoint.py # Script to manage model versions

### 3. Deploy Model to AWS SageMaker via GitHub Actions

You will need a script that handles training and deployment via AWS SageMaker. Here’s an overview of the process:

#### train.py (example for training)

This script uses the AWS SageMaker Python SDK to train your model and save the artifacts to S3:

python

Copy code

import sagemakerfrom sagemaker.tensorflow import TensorFlowimport boto3

# Define SageMaker session and role

sagemaker\_session = sagemaker.Session()

role = 'arn:aws:iam::<account-id>:role/<sagemaker-execution-role>' # Update with your role ARN

# Upload data to S3 (if needed)

data\_path = 's3://your-bucket-name/data/'

# Create TensorFlow estimator

estimator = TensorFlow(

entry\_point='train\_model.py', # Your training script

role=role,

framework\_version='2.3',

py\_version='py37',

instance\_count=1,

instance\_type='ml.m5.large',

hyperparameters={

'epochs': 10,

'batch\_size': 32,

}

)

# Train the model on SageMaker

estimator.fit(data\_path)

# Save the model to S3 after training

trained\_model\_artifact = estimator.model\_dataprint(f"Model trained and saved at: {trained\_model\_artifact}")

#### deploy\_model.py (example for deployment)

This script deploys the trained model to SageMaker:

python

Copy code

import sagemakerfrom sagemaker.model import Modelfrom sagemaker.tensorflow import TensorFlowModel

# Initialize the SageMaker session and model

sagemaker\_session = sagemaker.Session()

role = 'arn:aws:iam::<account-id>:role/<sagemaker-execution-role>' # Update with your role ARN

# Location of the trained model in S3

model\_artifact = 's3://your-bucket-name/model.tar.gz'

# Create a SageMaker TensorFlow Model for deployment

model = TensorFlowModel(

model\_data=model\_artifact,

role=role,

framework\_version='2.3',

py\_version='py37'

)

# Deploy the model as a real-time endpoint

predictor = model.deploy(

initial\_instance\_count=1,

instance\_type='ml.m5.large',

endpoint\_name='fastapi-lstm-endpoint'

)

print("Model deployed to SageMaker")

### 4. Configure AWS Credentials for GitHub Actions

To allow GitHub Actions to access your AWS account, you need to set up AWS credentials in the GitHub repository’s secrets:

1. Go to your GitHub repository.
2. Click on **Settings** → **Secrets and Variables** → **Actions** → **New repository secret**.
3. Add the following secrets:
   * AWS\_ACCESS\_KEY\_ID: Your AWS access key ID.
   * AWS\_SECRET\_ACCESS\_KEY: Your AWS secret access key.

These credentials are used in the GitHub Actions workflow to authenticate and authorize the operations in AWS.

### 5. Monitor and Test the Deployment

Once the model is deployed to SageMaker, you can monitor the SageMaker endpoint using the **AWS Console** or the **SageMaker SDK**.

You can test the endpoint using any HTTP client, such as curl or requests, by sending a POST request with the input data to the SageMaker endpoint URL.

Here’s how you can test the deployed model using Python:

python

Copy code

import boto3import json

# Initialize a SageMaker runtime client

client = boto3.client('sagemaker-runtime')

# Define the endpoint name and payload

endpoint\_name = 'fastapi-lstm-endpoint'

payload = json.dumps({"reviews": ["This movie was amazing!", "I did not like this movie."]})

# Invoke the SageMaker endpoint

response = client.invoke\_endpoint(

EndpointName=endpoint\_name,

ContentType="application/json",

Body=payload

)

# Print the predictionsprint(response['Body'].read().decode())

### Conclusion:

* **Continuous Integration**: GitHub Actions is used to automate testing, training, and deploying the machine learning model when changes are pushed.
* **Continuous Deployment**: The workflow automatically deploys the model to AWS SageMaker after successful training, ensuring that the most up-to-date model is always deployed in production.
* **AWS SageMaker**: Models are trained and deployed directly using SageMaker’s services without the need for Docker containerization, simplifying the deployment process.

This setup enables a smooth CI/CD pipeline for machine learning models using GitHub Actions and AWS SageMaker.