Rapport du Projet - Classificateur des Genres Musicaux

Rédigé par : Alaa DOUKH, Alaa HMEM et Sahar MILED

Introduction

Le projet vise à exploiter la technologie Docker pour créer un environnement dédié à la classification des genres musicaux, en utilisant deux services distincts : SVM_service et vgg19_service. Ces services, hébergés dans un même conteneur Docker, sont conçus pour classifier le genre musical d'un fichier audio au format WAV, transmis sous forme de données encodées en base64.

Objectifs

- Services de Classification :
    - svm_service : Utilise un modèle basé sur SVM pour classifier les genres musicaux.
    - vgg19_service : Utilise un modèle VGG19 pour effectuer la classification.
- Base de Données d'Entraînement : Le modèle de Machine Learning est entraîné sur le jeu de données GTZAN offrant une diversité de genres musicaux, disponible sur Kaggle : https://www.kaggle.com/andradaolteanu/gtzan-dataset-music-genre-classification
- Déploiement Simplifié : Utilisation de Docker et Docker-compose pour créer une image unique encapsulant les deux services, facilitant le déploiement.
- Intégration Continue avec Jenkins : Mise en place d'un environnement d'intégration continue pour automatiser le déploiement, les tests et garantir la qualité continue du projet.

Structure simplifiée du Projet

```bash
Music-Genre-Classifier/
|-- my_app/
|   |-- templates/
|   |   |-- upload.html
|   |   `-- result.html
|   |-- app.py
|   |-- Dockerfile
|   `-- requirements.txt
|-- svm_service/
|   |-- app.py
|   |-- svm-classification-model.h5
|   |-- tests
|   `-- requirements.txt
|-- vgg19_service/
|   |-- app.py
|   |-- vgg-classification-model.h5
|   |-- tests
```

```
|   `-- requirements.txt
|-- docker-compose.yml
|-- Jenkinsfile
```

Code

docker-compose

```bash
version: '3'
services:
my_app:
    build:
    context: ./my_app
    dockerfile: Dockerfile
    ports:
    - "3000:3000"
    volumes:
    - shared_volume:/Nouvarch/shared_volume
    environment:
    - FLASK_ENV=development
svm_service:
    build:
    context: ./svm_service
    dockerfile: Dockerfile
    ports:
    - "8000:8000"
    volumes:
    - shared_volume:/Nouvarch/shared_volume
    depends_on:
    - my_app
vgg19_service:
    build:
    context: ./vgg19_service
    dockerfile: Dockerfile
    ports:
    - "9000:9000"
    volumes:
    - shared_volume:/Nouvarch/shared_volume
    depends_on:
    - my_app
volumes:
shared_volume:
```

`my_app`: C'est le service principal, construit à partir du répertoire ./my_app en utilisant le fichier Dockerfile. Il expose le port 3000 et monte le volume partagé pour permettre la persistance des données.

`svm_service` et `vgg19_service`: Ce sont les services dépendants construits à partir des répertoires respectifs. Ils exposent les ports 8000 et 9000, respectivement. Ils dépendent du service my_app, ce qui signifie qu'ils ne démarreront que lorsque my_app sera prêt.

`volumes`: Il crée un volume nommé shared_volume qui est utilisé pour partager des données entre les différents services. Cela garantit la persistance des données même si les conteneurs sont détruits et recréés.

Ce fichier de composition est essentiel pour définir les relations entre les services et les configurations nécessaires pour les faire fonctionner ensemble dans un environnement Docker.

my_app (Front-End)

app.py

- Importation des modules :

```
import json
from flask import Flask, render_template, request
import requests
import base64
```

- json : Module pour travailler avec des données JSON.
- Flask : Framework web pour Python.
- render_template: Fonction pour rendre des modèles HTML avec Flask.
- request : Module Flask pour gérer les requêtes HTTP.
- requests : Bibliothèque HTTP pour envoyer des requêtes à d'autres services.
- base64 : Module pour encoder et décoder en base64.

```
svm_service_url = 'http://svm_service:8000'
vgg19_service_url = 'http://vgg19_service:9000'

@app.route('/')
def hello_world():
    return render_template('upload.html')
```

/ : La racine de l'application, renvoie le modèle HTML upload.html. /classify_audio : Traitement des fichiers audio, envoie les données encodées au service SVM. /classify_image : Traitement des fichiers image, envoie les données encodées au service VGG19.

```bash
@app.route('/classify_audio', methods=['POST'])
def classify():

    if 'fileInput' not in request.files:
        return "No file provided"

    music_file = request.files['fileInput']
    encoded_music_data = base64.b64encode(music_file.read()).decode('utf-8')
    response = requests.post(f'{svm_service_url}/classify', json={"music_data":
encoded_music_data})

    response_data = response.json()
    received_message = response_data.get("received_message", "No message
received")
    svm_response = response_data.get("response", "No response received")

    return render_template('result.html', received_message=received_message,
response=svm_response)
```

Cette fonction est appelée lorsque l'utilisateur soumet un fichier audio. Elle récupère le fichier, encode les données en base64, envoie une requête POST au service SVM, et affiche le résultat sur une page HTML.

```bash
@app.route('/classify_image', methods=['POST'])
def classify_image():

    if 'fileInput' not in request.files:
        return "No file provided"

    image_file = request.files['fileInput']

    encoded_image_data = base64.b64encode(image_file.read()).decode('utf-8')

    response = requests.post(f'{vgg19_service_url}/classify', json={"image_data":
encoded_image_data})

    try:
        response_data = response.json()
    except json.decoder.JSONDecodeError:
        print("Error: Invalid JSON response")
        response_data = {}

    received_message = response_data.get("received_message", "No message
received")
    vgg19_response = response_data.get("response", "No response received")
```

```
    return render_template('result.html', received_message=received_message,
response=vgg19_response)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=3000, debug=True, use_reloader=True)
```

Cette fonction est appelée lorsque l'utilisateur soumet un fichier image. Elle récupère le fichier, encode les données en base64, envoie une requête POST au service VGG19, et affiche le résultat sur une page HTML.

upload.html

```bash
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Classifier</title>
</head>
<body>
    <div class="container">
        <h1>Classifier</h1>

        <form method="post" enctype="multipart/form-data" id="uploadForm">
            <input type="radio" name="dataType" id="audioType" value="audio"
checked onchange="updateAttributes()">
            <label for="audioType">Audio</label>

            <input type="radio" name="dataType" id="imageType" value="image"
onchange="updateAttributes()">
            <label for="imageType">Image</label>

            <br><br>

            <label for="fileInput">Choose a file:</label>
            <input type="file" name="fileInput" id="fileInput" accept=".wav"
required>
            <br>

            <button type="submit" onclick="updateActionAttribute()">Upload and
Classify</button>
        </form>

        <script>
            function updateAttributes() {
                var audioType = document.getElementById("audioType");
```

```
                var fileInput = document.getElementById("fileInput");

                if (audioType.checked) {
                    fileInput.accept = ".wav";
                    document.getElementById("uploadForm").action =
"/classify_audio";
                } else {
                    fileInput.accept = ".png, .jpg, .jpeg";
                    document.getElementById("uploadForm").action =
"/classify_image";
                }
            }

            function updateActionAttribute(){
                var audioType = document.getElementById("audioType");
                var uploadForm = document.getElementById("uploadForm");

                if (audioType.checked) {
                    uploadForm.action = "/classify_audio";
                } else {
                    uploadForm.action = "/classify_image";
                }
            }
        </script>
    </div>
</body>
</html>
```

Ce fichier HTML crée une page web avec un formulaire permettant à l'utilisateur de télécharger un fichier pour la classification, avec la possibilité de choisir entre deux types de données : audio ou image. La page inclut des boutons radio pour la sélection du type de données, un champ de téléchargement de fichier, et un bouton pour l'envoi du formulaire. Des fonctions JavaScript sont utilisées pour ajuster dynamiquement les attributs du formulaire en fonction du type de données choisi.

result.html

```bash
<!DOCTYPE html>
<html lang="en">
<head>
    <link rel="stylesheet" href="{{ url_for('static', filename='style.css') }}">
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Music Genre Classifier</title>
</head>
<body>
    <div class="container">
```

```
        <h1>Music Genre Classifier</h1>

        <div class="result-container">
            <p><strong>  {{ response }} </strong></p>
        </div>
        <button type="submit" > <a style="color: #fff; text-decoration: none;"
href="/">Home</a></button>
    </div>
</body>
</html>
```

Ce fichier HTML définit une page de résultats pour le classificateur de genre musical. La page affiche la réponse du serveur, qui est dynamiquement insérée à l'aide de la variable `response` provenant du serveur. Il y a également un bouton "Home" qui permet de retourner à la page d'accueil du classificateur.

Dockerfile

```bash
FROM python:3-alpine3.15

WORKDIR ./

COPY . .

ENV STATIC_URL /static
ENV STATIC_PATH /app/static

COPY ./requirements.txt /requirements.txt
RUN pip install -r /requirements.txt

EXPOSE 3000

CMD ["python", "/app/app.py"]
```

Le fichier Dockerfile est utilisé pour construire une image Docker contenant une application Flask, définie dans le script Python app.py. Basée sur l'image officielle Python 3 avec Alpine Linux 3.15, la construction commence par la définition du répertoire de travail, suivi de la copie des fichiers locaux, y compris requirements.txt.

Des variables d'environnement sont définies pour les fichiers statiques, puis les dépendances Python sont installées. Le port 3000 est exposé pour indiquer où l'application écoute, et la commande par défaut lance l'application Flask.

Ce Dockerfile simplifie le déploiement de l'application, encapsulant toutes les dépendances et paramètres nécessaires dans un conteneur Docker autonome.

requirements.txt

```bash
Flask==2.0.1
Werkzeug==2.0.1
requests==2.26.0
```

Ces lignes définissent les versions spécifiques des bibliothèques Flask, Werkzeug et Requests que l'application utilise. Lorsque le Dockerfile est construit, ces dépendances seront installées dans l'environnement Python virtuel du conteneur, assurant une exécution cohérente de l'application.

svm_service

app.py

```bash
from flask import Flask, request, jsonify
import time
import base64
import os
import librosa
import numpy as np
from tensorflow.keras.models import load_model

app = Flask(__name__)

base_path = os.path.abspath(os.path.dirname(__file__))
model_path = os.path.join(base_path, "svm-classification-model.h5")
model = load_model(model_path)

genre_dict = {0: 'blues', 1: 'classical', 2: 'country', 3: 'disco',
            4: 'hiphop', 5: 'jazz', 6: 'metal', 7: 'pop', 8: 'reggae', 9: 'rock'}

def extract_features(file_path):
    audio, sample_rate = librosa.load(file_path, res_type='kaiser_fast')
    mfccs = librosa.feature.mfcc(y=audio, sr=sample_rate, n_mfcc=20)
    mfccs_processed = np.mean(mfccs.T, axis=0)
    return mfccs_processed

def predict_genre(file_path):
    features = extract_features(file_path)
    features = np.reshape(features, (1, -1))
    prediction = model.predict(features)
    print("Raw Prediction:", prediction)
    predicted_genre = np.argmax(prediction)
    predicted_genre_name = genre_dict.get(predicted_genre, "Unknown Genre")
    return predicted_genre_name
```

```
@app.route('/classify', methods=['POST'])
def classify():
    data = request.get_json()
    if data and "music_data" in data:
        encoded_music_data = data["music_data"]
        decoded_music_data = base64.b64decode(encoded_music_data)
        temp_wav_file = '/Nouvarch/shared_volume/temp_audio.wav'
        with open(temp_wav_file, 'wb') as temp_file:
            temp_file.write(decoded_music_data)

        genre = predict_genre(temp_wav_file)
        response_data = {"received_message": "Music file received and processed
successfully",
                         "response": f"Predicted Genre: {genre}"}
    else:
        response_data = {"received_message": "No music file received", "response":
"Error"}

    return jsonify(response_data)

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=8000)
```

Ce script Python représente une application Flask qui expose un endpoint POST `/classify` pour la classification du genre musical. Les bibliothèques importées incluent Flask pour la création de l'application web, des modules pour le traitement audio (librosa), la manipulation de données (NumPy), et la gestion de modèles de Machine Learning avec TensorFlow/Keras.

L'application est initialisée avec Flask, et un modèle SVM pré-entraîné est chargé pour la classification des genres musicaux. Un dictionnaire est utilisé pour mapper les indices de sortie du modèle aux genres correspondants.

Deux fonctions principales, extract_features et predict_genre, sont définies pour extraire des caractéristiques audio et effectuer des prédictions de genre musical.

L'endpoint `/classify` traite les requêtes POST, décode les données audio encodées en base64, utilise le modèle SVM pour prédire le genre musical, puis renvoie une réponse JSON avec le résultat de la prédiction.

Enfin, l'application Flask est exécutée sur l'adresse IP '0.0.0.0' et le port 8000. Ces éléments sont regroupés pour créer une application complète de classification de genre musical à l'aide de modèles SVM.

Dockerfile

```bash
FROM python:3.9-slim-buster

WORKDIR ./
```

```
ENV STATIC_URL /static
ENV STATIC_PATH /app/static

COPY ./requirements.txt /requirements.txt
RUN pip install -r /requirements.txt

COPY /app/svm-classification-model.h5 /app/svm-classification-model.h5

COPY . .

EXPOSE 8000

CMD ["python", "/app/app.py"]
```

Ce Dockerfile crée un environnement minimaliste propice à l'exécution d'une application Flask basée sur Python 3.9. Il installe les dépendances requises, copie le modèle SVM pré-entraîné, et configure le point d'entrée pour l'application Flask tout en exposant le port 8000 pour les connexions externes.

requirements.txt

```bash
Flask==2.0.1
Werkzeug==2.0.1
requests==2.26.0
librosa==0.10.1
numpy==1.26.2
tensorflow==2.15.0
```

Ces dépendances incluent le framework Flask pour le développement web, Werkzeug comme bibliothèque d'utilités pour Flask, requests pour effectuer des requêtes HTTP, librosa pour l'analyse de fichiers audio, numpy pour le traitement de tableaux, et TensorFlow pour le deep learning.

vgg19_service

app.py

```bash
from flask import Flask, request, jsonify
import time
import base64
import os
import librosa
```

```
import numpy as np
import base64
from tensorflow.keras.preprocessing import image
from tensorflow.keras.models import load_model

app = Flask(__name__)

base_path = os.path.abspath(os.path.dirname(__file__))
model_path = os.path.join(base_path, "vgg-classification-model.h5")
model = load_model(model_path)

def image_to_base64(image_path):
    with open(image_path, "rb") as image_file:
        encoded_image = base64.b64encode(image_file.read())
        return encoded_image.decode("utf-8")


@app.route('/classify', methods=['POST'])
def classify_image():
    data = request.get_json()

    if data and "image_data" in data:
        encoded_image_data = data["image_data"]
        decoded_image_data = base64.b64decode(encoded_image_data)

        temp_image_file = '/Nouvarch/shared_volume/temp_image.jpg'
        with open(temp_image_file, 'wb') as temp_file:
            temp_file.write(decoded_image_data)

        img = image.load_img(temp_image_file, target_size=(224, 224))
        img_array = image.img_to_array(img)
        img_array = np.expand_dims(img_array, axis=0)
        img_array /= 255.0

        predictions = model.predict(img_array)

        genres = ['blues', 'classical', 'country', 'disco', 'hiphop', 'jazz',
'metal', 'pop', 'reggae', 'rock']
        predicted_genre = genres[np.argmax(predictions)]

        response_data = {"received_message": "Image file received and processed
successfully",
                         "response": f"Predicted Genre: {predicted_genre}"}
    else:
        response_data = {"received_message": "No image file received", "response":
"Error"}

    return jsonify(response_data)

if __name__ == '__main__':
```

```
    app.run(host='0.0.0.0', port=9000)
```

Ce script Python représente une application Flask qui expose un endpoint POST `/classify` pour la classification d'images en utilisant un modèle VGG19 pré-entraîné. Il commence par importer les bibliothèques nécessaires, telles que Flask, base64, et TensorFlow/Keras pour la manipulation d'images et le chargement du modèle. L'application est initialisée, le chemin du modèle VGG19 est défini, et le modèle est chargé. Une fonction auxiliaire `image_to_base64` est définie pour convertir une image en base64.

L'endpoint `/classify` traite les requêtes POST, décode les données d'image encodées en base64, effectue la prédiction du genre musical à l'aide du modèle VGG19, puis renvoie une réponse JSON avec le résultat. Enfin, l'application Flask est exécutée sur l'adresse IP 0.0.0.0 et le port 9000. En résumé, ce script crée un service web Flask pour la classification d'images avec un modèle VGG19, offrant une interface simple via des requêtes POST et fournissant les résultats prédits en format JSON, le tout écoutant sur le port 9000.

```bash
FROM python:3.9-slim-buster

WORKDIR ./

ENV STATIC_URL /static
ENV STATIC_PATH /app/static

COPY ./requirements.txt /requirements.txt
RUN pip install -r /requirements.txt

COPY /app/vgg-classification-model.h5 /app/vgg-classification-model.h5

COPY . .

EXPOSE 9000

CMD ["python", "/app/app.py"]
```

Ce fichier Dockerfile définit la configuration pour construire une image Docker destinée à exécuter une application Flask de classification d'images à l'aide d'un modèle VGG19. Voici une explication concise de chaque instruction :

- FROM python:3.9-slim-buster : Utilise une image de base Python version 3.9 basée sur Debian Buster avec une empreinte réduite pour une meilleure efficacité.
- WORKDIR ./ : Définit le répertoire de travail à la racine du système de fichiers du conteneur.
- ENV STATIC_URL /static et ENV STATIC_PATH /app/static : Configure des variables d'environnement définissant l'URL statique et le chemin statique pour l'application Flask.

- COPY ./requirements.txt /requirements.txt : Copie le fichier requirements.txt local dans le répertoire /requirements.txt du conteneur.
- RUN pip install -r /requirements.txt : Installe les dépendances spécifiées dans requirements.txt en utilisant pip.
- COPY /app/vgg-classification-model.h5 /app/vgg-classification-model.h5 : Copie le modèle de classification VGG19 pré-entraîné dans le répertoire /app du conteneur.
- COPY . . : Copie le contenu local du projet dans le répertoire actuel du conteneur.
- EXPOSE 9000 : Indique que le conteneur écoutera sur le port 9000 lors de son exécution.
- CMD ["python", "/app/app.py"] : Définit la commande par défaut à exécuter lorsque le conteneur démarre, ici, l'exécution du script Python app.py.

jenkinsfile

```bash
pipeline {
    agent any

    environment {
        DOCKER_COMPOSE_VERSION = '1.29.2'
        REPORT_DIR = 'test-reports'
    }

    stages {
        stage('Build and Start Services') {
            steps {
                script {
                    sh "docker-compose up --build -d"
                }
            }
        }

        stage('Run Tests') {
            steps {
                script {
                    // Run SVM tests
                    sh "docker-compose exec my_app python svm_service/tests/test_app.py --junit-xml=${REPORT_DIR}/svm_test_results.xml"

                    // Run VGG19 tests
                    sh "docker-compose exec my_app python vgg19_service/tests/test_app.py --junit-xml=${REPORT_DIR}/vgg19_test_results.xml"
                }
            }
        }
```

```
        stage('Publish Test Results') {
            steps {
                script {
                    // Archive and publish JUnit test results
                    junit "${REPORT_DIR}/svm_test_results.xml"
                    junit "${REPORT_DIR}/vgg19_test_results.xml"
                }
            }
        }
    }

    post {
        always {
            // Stop Services
            script {
                sh "docker-compose down"
            }
        }
    }
}
```

Ce Jenkinsfile définit un pipeline Jenkins pour un environnement Docker avec deux services, SVM et VGG19. Le pipeline construit, déploie, exécute des tests, et publie les résultats. La première étape construit et démarre les services Docker. Ensuite, les tests SVM et VGG19 sont exécutés, les résultats sont sauvegardés en format JUnit. Enfin, les résultats des tests sont publiés. Après chaque build, les services Docker sont arrêtés. Ce pipeline assure un déploiement et une validation continus des services.