Ecole Nationale
Supérieure
d'Ingénieurs
de Tunis

الجمهورية التونسية
وزارة التعليم العالي والبحث العلمي
جامعة تونس
المدرسة الوطنية العليا للمهندسين بتونس

Rapport de Projet Docker

De

3 ème année en Génie Informatique

Par

Nour El Houda Lajnef

Services Machine Learning Dockerisés pour la Classification de Genres Musicaux

Durée de stage : 03/07/2023 -31/08/2023

Année universitaire : 2022-2023

5, Avenue Taha Hussein – Tunis
B. P. 56, Bab Menara 1008

Tel. : 71 496 066 الهاتف:
Fax : 71 391 166 فاكس

5، شارع طه حسين - تونس
ص. ب.: 56 باب منارة 1008

Introduction générale

Le projet présent a pour objectif de créer un environnement de Machine Learning basé sur la technologie Docker. En utilisant Python et le framework web Flask, nous cherchons à intégrer, déployer et tester un système capable de classifier les genres musicaux à partir du célèbre ensemble de données GTZAN Music Genre Classification en utilisant des modèles de machine learning, notamment le Support Vector Machine (SVM) et la VGG19 (Visual Geometry Group 19-layer model), et de déployer l'application résultante à l'aide de Jenkins.

Mon rapport se structure en trois parties distinctes, chacun couvrant une phase spécifique du projet :

- La première partie décrit l'architecture.
- Le deuxième chapitre est consacré à la description des modèles et des services web réalisés.
- La dernière partie est consacré à l'infrastructure et le déploiement

Mon rapport est clôturé par une conclusion générale.

1. Architecture du Projet

Le projet est structuré autour de l'utilisation de conteneurs Docker, chacun jouant un rôle spécifique dans l'environnement global du système. L'architecture est conçue pour permettre la classification des genres musicaux à l'aide de deux modèles de Machine Learning (SVM et VGG19) tout en fournissant une interface conviviale via un service web frontal (front).

- **Container SVM (svm):**

Ce conteneur est dédié au service web SVM, chargé de classer le genre musical d'un fichier audio WAV encodé en base64.

Il expose un port spécifique pour la communication avec d'autres services du projet.

- **Container VGG (vgg):**

Le conteneur VGG héberge le service web VGG19, qui effectue la classification du genre musical d'un fichier audio WAV encodé en base64.

Tout comme le conteneur SVM, il expose un port pour les communications entrantes.

- **Container Frontend (front):**

Le conteneur frontal est responsable de la partie web de l'application. Il intègre un serveur Flask qui gère les requêtes utilisateur.

Il communique avec les conteneurs SVM et VGG pour obtenir les prédictions respectives et les affiche dans l'interface utilisateur.

- **Communication entre les Conteneurs (ml_network):**

Les conteneurs SVM et VGG communiquent avec le conteneur frontal pour partager les résultats de leurs prédictions.

Lorsqu'un utilisateur télécharge un fichier audio via l'interface web, le front-end transmet la requête aux conteneurs SVM et VGG.

Les conteneurs SVM et VGG effectuent la classification et renvoient les prédictions au

conteneur frontal.

Le conteneur frontal affiche les résultats dans l'interface utilisateur pour que l'utilisateur puisse visualiser les prédictions des deux modèles.

Cette architecture favorise une séparation claire des responsabilités, permettant une évolutivité et une maintenance efficaces. Chaque conteneur fonctionne de manière indépendante, mais leur collaboration permet de fournir une expérience utilisateur complète et intégrée pour la classification des genres musicaux. Cette structure modulaire facilite également le déploiement et la mise à l'échelle du système en fonction des besoins.

2. Modèles Machines Learning et Service Web Flask

Nous allons passé maintenant à la description de nos modèles et nos services web Flask et leur fonctionnement.

2.1 Modèle SVM et son Service Web Flask

Nous avons commencé par l'importation des bibliothèques telles que librosa, scikit-learn, et d'autres, permet d'accéder aux fonctionnalités nécessaires pour le traitement du signal audio, la création du modèle SVM, et la gestion des avertissements futurs.

In [1]:
```
## packages

import librosa
import sklearn

import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

%matplotlib inline
import matplotlib.pyplot as plt
import librosa.display

import os

import numpy as np
import pandas as pd
```

Dans l'étape suivante de test avec un fichier audio, le chemin du fichier audio est spécifié, puis le fichier est chargé à l'aide de la bibliothèque librosa. Les types de données du signal audio (x) et du taux d'échantillonnage (sr) sont imprimés, ainsi que la forme du signal et le taux d'échantillonnage.

```
## testing with a file

audio_path = '../input/gtzan-dataset-music-genre-classification/Data/genres_original/blues/blues.
00000.wav'
x , sr = librosa.load(audio_path)
print(type(x), type(sr))

print(x.shape, sr)
```

```
<class 'numpy.ndarray'> <class 'int'>
(661794,) 22050
```

Dans cette phase d'extraction de caractéristiques, on utilise le taux de passage par zéro, qui représente le nombre de changements de signe le long d'un signal audio. L'exemple montre un extrait du signal audio entre les échantillons 9000 et 9100, avec une représentation graphique du signal. Ensuite, le nombre total de passages par zéro dans cet extrait est calculé et affiché qui est égale à seize.

Feature Extraction

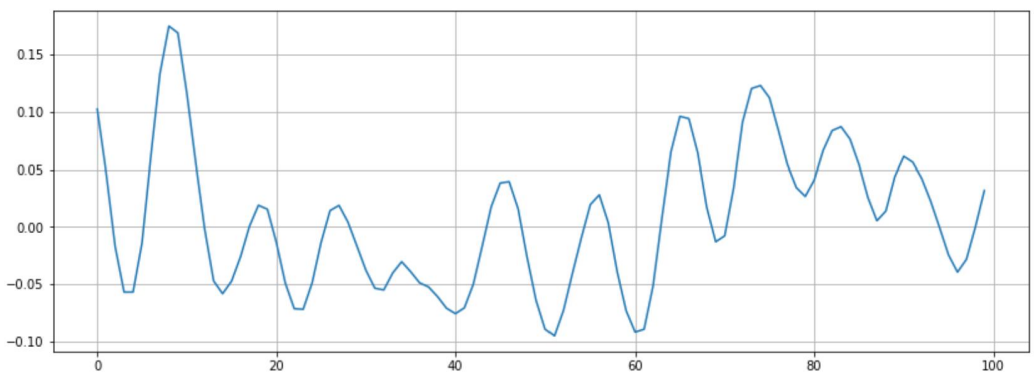
Zero crossig rate

The zero-crossing rate is the rate of sign-changes along with a signal, i.e., the rate at which the signal changes from positive to negative or back. This feature has been used heavily in both speech recognition and music information retrieval. It usually has higher values for highly percussive sounds like those in metal and rock.

```
n0 = 9000
n1 = 9100
plt.figure(figsize=(14, 5))
plt.plot(x[n0:n1])
plt.grid()

zero_crossings = librosa.zero_crossings(x[n0:n1], pad=False)
print(sum(zero_crossings))
```

```
16
```



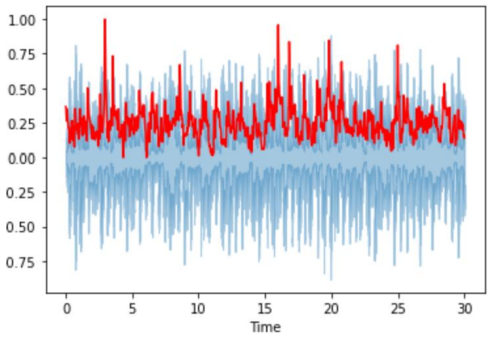
Dans la section suivante, nous calculons le centroïde spectral qui représente le "centre de masse" d'un son en fonction des fréquences présentes dans le signal audio. L'exemple utilise la fonction librosa.feature.spectral_centroid pour extraire ces informations du signal audio. La représentation graphique affiche le signal audio en transparence avec une superposition du centroïde spectral normalisé en rouge. Cela permet d'observer la répartition des fréquences dans le son.

Spectral Centroid

It indicates where the ” center of mass” for a sound is located and is calculated as the weighted mean of the frequencies present in the sound. Consider two songs, one from a blues genre and the other belonging to metal. Now, as compared to the blues genre song, which is the same throughout its length, the metal song has more frequencies towards the end. So spectral centroid for blues song will lie somewhere near the middle of its spectrum while that for a metal song would be towards its end.

```
spectral_centroids = librosa.feature.spectral_centroid(x, sr=sr)[0]
spectral_centroids.shape
frames = range(len(spectral_centroids))
t = librosa.frames_to_time(frames)
def normalize(x, axis=0):
    return sklearn.preprocessing.minmax_scale(x, axis=axis)
librosa.display.waveshow(x, sr=sr, alpha=0.4)
plt.plot(t, normalize(spectral_centroids), color='r')
```

```
[<matplotlib.lines.Line2D at 0x7d0cdc653090>]
```



Dans cette section du code, nous définissons trois fonctions :

normalize(x, axis=0): Normalise un tableau x en utilisant la mise à l'échelle min-max.

zero_cross(x): Calcule le taux de changement de signe dans une portion du signal audio définie par les indices n0 et n1.

spec_center(x, sr): Calcule le centre spectral normalisé du signal audio et renvoie le temps auquel il se produit.

Ensuite, nous utilisons ces fonctions pour extraire les caractéristiques, notamment le taux de croisement de zéro, le centre spectral, et le genre musical, à partir des fichiers audio dans le répertoire spécifié. Les résultats sont stockés dans une liste li.

```
## feature extraction funciton

def normalize(x, axis=0):
        return sklearn.preprocessing.minmax_scale(x, axis=axis)

def zero_cross(x):
    n0 = 9000
    n1 = 9100
    zero_crossings = librosa.zero_crossings(x[n0:n1], pad=False)
    return sum(zero_crossings)

def spec_center(x, sr):
    spectral_centroids = normalize(librosa.feature.spectral_centroid(x, sr=sr)[0])
    frames = range(len(spectral_centroids))
    t = librosa.frames_to_time(frames)
    ma = max(spectral_centroids)
    return t[np.where(spectral_centroids==ma)[0][0]]
```

```
li = []
path = '../input/gtzan-dataset-music-genre-classification/Data/genres_original/'

for gen in genre:
    for song in os.listdir(os.path.join(path, gen)):
        x, sr = librosa.load(os.path.join(path, gen, song))
        li.append([zero_cross(x), round(spec_center(x, sr), 2), gen])
```

```
li = np.array(li)
```

Dans cette partie, nous mettons en œuvre une machine à vecteurs de support (SVM) pour la classification des genres musicaux. Nous utilisons les caractéristiques extraites précédemment (taux de croisement de zéro et centre spectral) pour former et évaluer le modèle SVM.

Implementing svm

```
X = li[:, 0:2]
y = li[:,2]
```

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2, random_state = 1)
```

```
from sklearn.svm import SVC
classifier = SVC(kernel='rbf', random_state = 1)
classifier.fit(X_train,y_train)
```

```
SVC(random_state=1)
```

```
sklearn.metrics.confusion_matrix(y_test, y_pred)
```

```
array([[ 6, 17],
       [ 3, 14]])
```

```
## testing for custom song

pa = '../input/gtzan-dataset-music-genre-classification/Data/genres_original/hiphop/hiphop.00004.
wav'
x, sr = librosa.load(pa)
print(classifier.predict(np.array([[zero_cross(x), round(spec_center(x, sr), 2)]])))
```

```
['hiphop']
```

- Nous définissons les ensembles de caractéristiques X et les étiquettes y à partir de la liste li.
- Divisons les données en ensembles d'entraînement (X_train, y_train) et de test (X_test, y_test) à l'aide de train_test_split.
- Créons un classificateur SVM avec un noyau radial (kernel='rbf') et l'entraînons sur les données d'entraînement.
- Prédisons les étiquettes pour les données de test (y_pred) et évaluons la performance du modèle avec une matrice de confusion.
- Enfin, testons le modèle sur un exemple de chanson personnalisée (hiphop.00004.wav) en utilisant les mêmes caractéristiques.

Le modèle SVM montre une matrice de confusion et prédit correctement le genre de la chanson hip-hop fournie en exemple.

```
import joblib
model_filename = 'svm_model.joblib'
joblib.dump(classifier, model_filename)
```

```
['svm_model.joblib']
```

Ici, nous enregistrons le modèle SVM entraîné sous le nom de fichier 'svm_model.joblib' en utilisant la bibliothèque joblib. Ce fichier nous allons l' utiliser ultérieurement pour charger le modèle et effectuer des prédictions sans avoir besoin de réentraîner le modèle.

```
svm_service.py > ...
from flask import Flask, request, jsonify
import joblib
import librosa
import numpy as np
import sklearn
import warnings

warnings.simplefilter(action='ignore', category=FutureWarning)

app = Flask(__name__)

# Charger le modèle SVM
model_filename = '/app/models/svm_model.joblib'
loaded_classifier = joblib.load(open(model_filename, 'rb'))

# Fonction d'extraction des caractéristiques
def normalize(x, axis=0):
    return sklearn.preprocessing.minmax_scale(x, axis=axis)

def zero_cross(x):
    n0 = 9000
    n1 = 9100
    zero_crossings = librosa.zero_crossings(x[n0:n1], pad=False)
    return sum(zero_crossings)
```

```
def spec_center(x, sr):
    spectral_centroids = normalize(librosa.feature.spectral_centroid(y=x, sr=sr)[0])
    frames = range(len(spectral_centroids))
    t = librosa.frames_to_time(frames)
    ma = max(spectral_centroids)
    return t[np.where(spectral_centroids == ma)[0][0]]

@app.route('/predict_svm', methods=['POST'])
def predict_svm():
    try:
        # Assurez-vous que 'audio' est la clé correcte dans la requête multipart
        file = request.files['audio']
        x, sr = librosa.load(file, sr=None)  # Chargez le fichier WAV

        # ... Traitement des caractéristiques audio ...

        # Faire la prédiction
        new_example_features = np.array([[zero_cross(x), round(spec_center(x, sr), 2)]])
        prediction = loaded_classifier.predict(new_example_features)

        # Retourner le résultat
        return jsonify({'prediction': prediction.tolist()})
    except Exception as e:
        return jsonify({'error': str(e)})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

Dans ce fichier svm_service.py, nous avons créé un service web Flask (/predict_svm) qui

utilise notre modèle SVM préalablement entraîné pour effectuer des prédictions sur les genres musicaux. La fonction predict_svm extrait les caractéristiques audio, fait la prédiction à l'aide du modèle SVM chargé, et renvoie le résultat au format JSON. Ce service est conçu pour être appelé avec un fichier audio au format WAV via une requête POST multipart.

L'application Flask est exécutée sur le port 5000 et est disponible pour les requêtes provenant de l'extérieur de l'environnement Docker, car elle est configurée avec host='0.0.0.0'.

2.2 Modèle VGG_19 et son Service Web Flask

De meme pour le VGG, nous commençons par importer les bibliothèques nécessaires. Puis l'entrainement , le test et l'enregistrement du modèle. On passe au chargement du modèle VGG préalablement entraîné à partir du chemin spécifié Ce modèle est utilisé pour effectuer des prédictions sur des caractéristiques audio spécifiques.

Enfin, la liste classes est définie pour contenir les noms des classes (genres musicaux) que le modèle peut prédire. Ces noms de classe sont associés aux étiquettes numériques utilisées dans le jeu de données.

Puis nous avons imprimé un rapport de classification basé sur les étiquettes de formation (train['label']) et les prédictions du modèle (train['prediction']). Cela donne une évaluation détaillée de la performance du modèle sur l'ensemble de données d'entraînement.

```
import tensorflow as tf
import numpy
import numpy as np
model = tf.saved_model.load( '/kaggle/input/music-genre-classification-vggish-model/VGGish' )
classes = [  "blues" ,  "classical" ,  "country" ,  "disco" ,
           "hiphop" ,  "jazz" ,  "metal" ,  "pop" ,  "reggae" ,  "rock" ,  ]
```

```
print(classification_report(test['label'],test['prediction']))
```

```
              precision    recall  f1-score   support

       blues       0.83      0.71      0.77        21
   classical       1.00      0.96      0.98        28
     country       0.62      0.71      0.67        21
       disco       0.75      0.79      0.77        19
      hiphop       0.78      0.82      0.80        17
        jazz       0.95      0.90      0.93        21
       metal       0.92      1.00      0.96        11
         pop       0.88      0.75      0.81        20
      reggae       0.68      0.75      0.71        20
        rock       0.77      0.77      0.77        22

    accuracy                           0.81       200
   macro avg       0.82      0.82      0.82       200
weighted avg       0.82      0.81      0.82       200
```

Après, nous avons effectué une prédiction sur un nouveau fichier WAV ('classical.00008.wav'), et affiché la classe prédite. Les caractéristiques audio du fichier WAV sont chargées à l'aide de la bibliothèque librosa et le taux d'échantillonnage est défini sur 16 000 Hz. Si la longueur de l'onde sonore n'est pas un multiple de 16 000, elle est remplie avec des zéros. Ensuite, ces caractéristiques audio sont transformées en un tenseur d'entrée et utilisées pour obtenir les scores de classe à l'aide du modèle VGG. La classe prédite est identifiée à partir des scores de classe, et le résultat est imprimé.

Enfin, ce script vgg19_service.py utilise Flask pour créer un service web qui prend en charge des prédictions de genres musicaux à l'aide du modèle VGG19. Il charge le modèle VGG19 pré-entraîné, définit une route pour recevoir les fichiers audio, effectue des prétraitements, puis retourne les prédictions au format JSON. En résumé, c'est une interface pour utiliser le modèle VGG19 via une API web.

vgg19_service.py > ...

```python
from flask import Flask, request, jsonify
import tensorflow as tf
import numpy as np
import librosa

app = Flask(__name__)

# Charger le modèle VGG
model_path = '/app/models/VGG'
model = tf.saved_model.load(model_path)
classes = ["blues", "classical", "country", "disco", "hiphop", "jazz", "metal", "pop", "reggae", "rock"]

@app.route('/predict_vgg', methods=['POST'])
def predict_vgg():
    try:
        # Assurez-vous que 'audio' est la clé correcte dans la requête multipart
        file = request.files['audio']
        waveform, sr = librosa.load(file, sr=16000)

        # ... Traitement des caractéristiques audio ...

        # Créer le tenseur d'entrée
        inp = tf.constant(np.array([waveform]), dtype='float32')

        # Faire la prédiction
        class_scores = model(inp)[0].numpy()
        predicted_class = classes[class_scores.argmax()]

        # Retourner le résultat
        return jsonify({'prediction': predicted_class})
    except Exception as e:
        return jsonify({'error': str(e)})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

2.3 Service Web Front

Ce script est conçu pour créer une application web qui permet aux utilisateurs de télécharger des fichiers WAV, puis envoie ces fichiers à deux services distincts (SVM et VGG) pour effectuer des prédictions de genres musicaux. Les résultats des prédictions sont ensuite affichés sur une page web, avec également la possibilité d'afficher les données du fichier WAV en format Base64. En résumé, c'est une interface web pour interagir avec les modèles SVM et VGG et visualiser les résultats.

```python
from flask import Flask, render_template, request
import os
from werkzeug.utils import secure_filename
import base64
import requests

app = Flask(__name__)

# Set the upload folder and allowed extensions
UPLOAD_FOLDER = 'uploads'
ALLOWED_EXTENSIONS = {'wav'}

app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER

# Create the uploads directory if it doesn't exist
if not os.path.exists(UPLOAD_FOLDER):
    os.makedirs(UPLOAD_FOLDER)

# URL des services SVM et VGG
svm_url = 'http://svm:5000/predict_svm'
vgg_url = 'http://vgg:5000/predict_vgg'

def allowed_file(filename):
    return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS

@app.route('/')
def index():
```

```python
def allowed_file(filename):
    return '.' in filename and filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS

@app.route('/')
def index():
    return render_template('index.html', wav_data=None, svm_prediction=None, vgg_predicti

@app.route('/predict', methods=['POST'])
def predict():
    if 'file' not in request.files:
        return "No file part"
    file = request.files['file']
    if file.filename == '':
        return "No selected file"
    if file and allowed_file(file.filename):
        filename = secure_filename(file.filename)
        file_path = os.path.join(app.config['UPLOAD_FOLDER'], filename)
        file.save(file_path)

        # Envoyer le fichier WAV à SVM
        with open(file_path, 'rb') as wav_file:
            svm_response = requests.post(svm_url, files={'audio': wav_file})

        # Envoyer le fichier WAV à VGG
        with open(file_path, 'rb') as wav_file:
            vgg_response = requests.post(vgg_url, files={'audio': wav_file})
```

```python
        return 'No selected file'
    if file and allowed_file(file.filename):
        filename = secure_filename(file.filename)
        file_path = os.path.join(app.config['UPLOAD_FOLDER'], filename)
        file.save(file_path)

        # Envoyer le fichier WAV à SVM
        with open(file_path, 'rb') as wav_file:
            svm_response = requests.post(svm_url, files={'audio': wav_file})

        # Envoyer le fichier WAV à VGG
        with open(file_path, 'rb') as wav_file:
            vgg_response = requests.post(vgg_url, files={'audio': wav_file})

        # Lire et encoder le fichier .wav comme base64
        with open(file_path, 'rb') as wav_file:
            wav_data = base64.b64encode(wav_file.read()).decode('utf-8')

        # Récupérer les résultats des services SVM et VGG
        svm_prediction = svm_response.json()['prediction'] if svm_response.ok else 'Error
        vgg_prediction = vgg_response.json()['prediction'] if vgg_response.ok else 'Error

        return render_template('index.html', wav_data=wav_data, svm_prediction=svm_predic

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000)
```

3. Infrastructure et Déploiement

3.1 DockerFile

Nous avons crée trois DockerFile chaque container à son DockerFile et son requirements.txt voici un exemple de mon DockerFile_vgg.

Dockerfile_vgg

```
# Utiliser une image de base Python avec TensorFlow
FROM tensorflow/tensorflow:2.6.0

# Définir le répertoire de travail dans le conteneur
WORKDIR /app

COPY . /app

# Installer les dépendances du système
RUN apt-get update && apt-get install -y libsndfile1

# Installer les dépendances
RUN pip install --no-cache-dir -r requirements3.txt


# Commande par défaut pour lancer le service VGG
CMD ["python", "vgg19_service.py"]
```

Le Dockerfile pour le service VGG commence par utiliser une image de base provenant de TensorFlow, version 2.6.0. TensorFlow est une bibliothèque de machine learning, et l'utilisation d'une image prête à l'emploi facilite l'intégration des dépendances nécessaires.

Ensuite, le répertoire de travail dans le conteneur est défini comme étant /app, et l'ensemble du contenu du répertoire local est copié dans le conteneur. Cela inclut le script vgg19_service.py et d'autres fichiers nécessaires.

Pour s'assurer que toutes les dépendances du système sont satisfaites, le Dockerfile utilise les commandes apt-get update et apt-get install pour installer la bibliothèque libsndfile1. Cette bibliothèque est fréquemment utilisée pour travailler avec des fichiers audio.

En ce qui concerne les dépendances Python, le fichier requirements3.txt est utilisé pour installer les bibliothèques Python nécessaires via la commande pip install.

Enfin, la commande par défaut du conteneur est définie pour exécuter le script vgg19_service.py à l'intérieur du conteneur lorsqu'il est démarré. Cela lance le service VGG et le rend prêt à accepter des prédictions sur des fichiers audio.

Cette configuration Docker assure que le service VGG est correctement configuré avec toutes les dépendances nécessaires et prêt à fonctionner dans un environnement Docker.

3.2 Docker-Compose

```yaml
version: '3'

services:
  front:
    build:
      context: .
      dockerfile: Dockerfile_front
    ports:
      - "5000"  # Mapping du port 5000 au hasard sur l'hôte
    volumes:
      - ../models:/app/models
      - ../test_data:/app/test_data
      - ./templates:/app/templates
    depends_on:
      - svm
      - vgg
    networks:
      - ml_network

  svm:
    build:
      context: .
      dockerfile: Dockerfile_svm
    ports:
      - "5001"  # Mapping du port 5001 au hasard sur l'hôte
    volumes:
      - ../models:/app/models
      - ../test_data:/app/test_data
      - ./templates:/app/templates
    networks:
      - ml_network

  vgg:
    build:
      context: .
      dockerfile: Dockerfile_vgg
    ports:
      - "5002"  # Mapping du port 5002 au hasard sur l'hôte
    volumes:
      - ../models:/app/models
      - ../test_data:/app/test_data
      - ./templates:/app/templates

    networks:
      - ml_network

networks:
  ml_network:
    driver: bridge
```

Ce fichier docker-compose.yml est une configuration Docker Compose qui définit trois services : front, svm, et vgg. Il utilise également un réseau nommé ml_network pour permettre la communication entre les services.

Voici une explication des principales parties de ce fichier :

- **services Section:**

❖ front Service: Ce service est construit à partir du Dockerfile Dockerfile_front dans le répertoire actuel (context: .). Il expose le port 5000 sur l'hôte et monte trois volumes pour partager les données entre le conteneur et l'hôte. Il dépend des services svm et vgg, ce qui signifie que ces deux services seront démarrés avant que le service front ne commence. Le service front est connecté au réseau ml_network.

❖ svm Service: Ce service est construit à partir du Dockerfile Dockerfile_svm. Il expose le port 5001 sur l'hôte et monte également trois volumes. Il est connecté au réseau ml_network.

❖ vgg Service: Ce service est construit à partir du Dockerfile Dockerfile_vgg. Il expose le port 5002 sur l'hôte et monte trois volumes. Il est connecté au réseau ml_network.

- **networks Section:**

ml_network Network: C'est un réseau de type pont (bridge). Il est utilisé pour permettre la communication entre les services front, svm, et vgg. Les services connectés à ce réseau peuvent communiquer entre eux.

- **Volumes:**

Les volumes sont utilisés pour partager des données entre le système hôte et les conteneurs. Les volumes spécifiés ici incluent ../models, ../test_data, et ./templates.

L'ensemble de ce fichier configure l'environnement nécessaire pour exécuter les services front, svm, et vgg, tout en permettant la communication entre eux grâce au réseau ml_network.

3.3 Test

Ce script de test unitaire en Python utilise le module unittest pour tester le service VGG (predict_vgg) dans votre application. Nous avons fait de meme pour SVM aussi .Voici une explication de chaque partie du script :

- **Méthode de test spécifique pour le service VGG.**

Utilise une liste de fichiers audio (wav_files) à tester.

Pour chaque fichier, envoie une requête POST au service VGG avec le fichier audio et vérifie la réponse.

Extrait l'étiquette réelle à partir du nom du fichier et vérifie si elle est présente dans la prédiction du modèle.

- **Exécution des tests:**

La condition if __name__ == '__main__': s'assure que les tests sont exécutés lorsque le script est exécuté directement, pas lorsqu'il est importé en tant que module.

unittest.main(): Cela lance les tests définis dans la classe TestPredictions lorsque le script est exécuté.

Ce script est conçu pour tester le service VGG en utilisant des fichiers audio spécifiques et vérifier si les prédictions du modèle sont conformes aux étiquettes réelles des fichiers. Assurez-vous que les fichiers audio spécifiés dans wav_files sont présents dans le répertoire de test (/app/test_data/).

```
import unittest
import os
from svm_service import predict_svm
from vgg19_service import predict_vgg
import requests
vgg_url = 'http://vgg:5000/predict_vgg'

class TestPredictions(unittest.TestCase):
    def extract_label(self, filename):
        # Extract class label (e.g., 'disco' or 'pop') from the filename
        return os.path.splitext(filename)[0].split('.')[0]

    def test_vgg_predictions(self):
        wav_files = ['disco.00004.wav']  # Add more files if necessary
        for wav_file_f in wav_files:
            file_path = f'/app/test_data/{wav_file_f}'
            with open(file_path, 'rb') as wav_file:
                vgg_response = requests.post(vgg_url, files={'audio': wav_file})
            vgg_prediction = vgg_response.json()['prediction'] if vgg_response.ok else 'Error predicting with VGG'
            label = self.extract_label(wav_file_f)
            self.assertIn(label, vgg_prediction)


if __name__ == '__main__':
    unittest.main()

```

3.4 Jenkinsfile

```
pipeline {
    agent any

    stages {
        stage('Checkout') {
            steps {
                script {
                    checkout(
                        scm: [
                            $class: 'GitSCM',
                            branches: [[name: '*/main']],
                            userRemoteConfigs: [[
                                credentialsId: 'yy28',
                                url: 'https://github.com/lucia2050/docker.git'
                            ]]
                        ]
                    )
                }
            }
        }

        stage('Build and Test') {
            steps {
                // Change to the 'Music/app' directory
                dir('Music/app') {
                    // Build Docker images
                    sh 'docker-compose build'

                    // Run Docker containers using WSL and nohup.sh
                    sh 'docker-compose up -d'

                    sh 'docker-compose ps'


                    // Run unit tests using WSL and nohup.sh
                    sh 'docker-compose exec vgg python -m unittest -v test_vgg.py'
                    sh 'docker-compose exec svm python -m unittest -v test_svm.py'
                }
            }
        }
    }

    post {
        success {
            echo 'Build and test successful!'
        }
    }
```

Ce script Jenkinsfile définit un pipeline de CI/CD pour votre application. Voici une explication de chaque partie du script :

Agent any: Indique que le pipeline peut être exécuté sur n'importe quel agent disponible.

Stages:

- Checkout:

Utilise Git pour récupérer le code source depuis le référentiel GitHub.

- Build and Test:

❖ Change le répertoire de travail vers Music/app.

❖ Construit les images Docker avec docker-compose build.

❖ Lance les conteneurs Docker en arrière-plan avec docker-compose up -d.

❖ Affiche les informations sur les conteneurs avec docker-compose ps.

❖ Exécute les tests unitaires pour les services SVM et VGG avec docker-compose exec.

Post-Success: Affiche un message si le pipeline réussit.

3.5 Déploiement

```
test_vgg_predictions (test_vgg.TestPredictions) ... ok

----------------------------------------------------------------------
Ran 1 test in 1.188s

OK
```

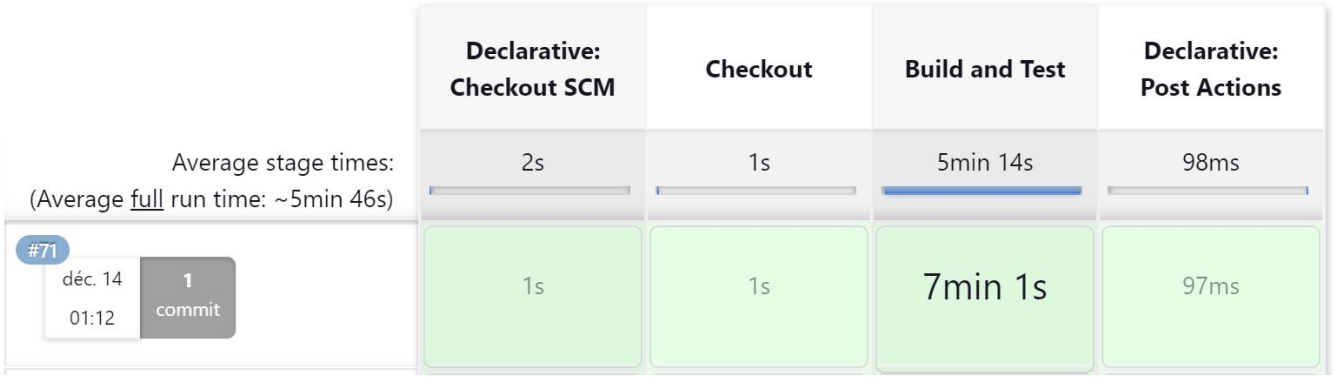
```
test_svm_predictions (test_svm.TestPredictions) ... ok

----------------------------------------------------------------------
Ran 1 test in 0.126s

OK
```

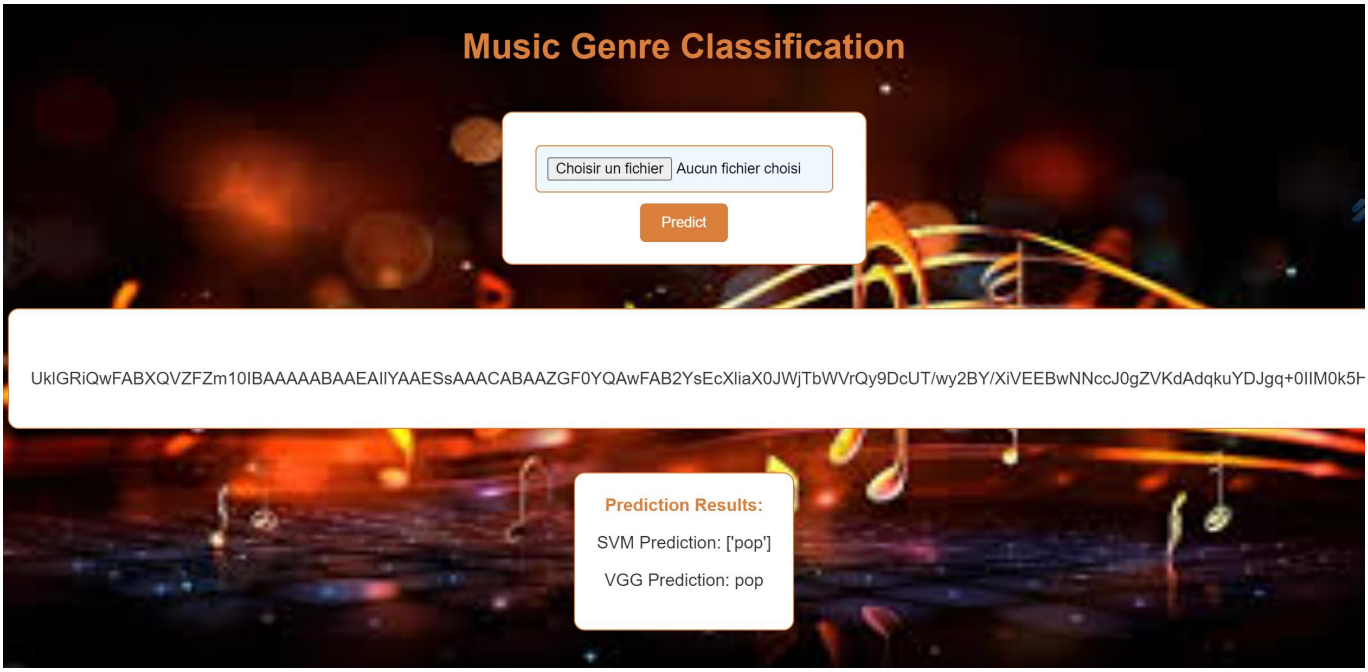
```
Build and test successful!
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

Stage View



Le pipeline Jenkins a été exécuté avec succès, atteignant ses objectifs étape par étape. Dans un premier temps, la récupération du code source depuis le référentiel GitHub a été réalisée sans encombre lors de l'étape "Checkout". Ensuite, dans la phase "Build and Test", le répertoire de travail a été changé vers Music/app, où les images Docker ont été construites avec succès grâce à la commande docker-compose build. En poursuivant, les conteneurs Docker ont été lancés en arrière-plan en utilisant docker-compose up -d, et les informations détaillées sur ces conteneurs ont été affichées avec docker-compose ps. La phase finale de tests unitaires pour les services SVM et VGG a été exécutée avec succès à l'aide de la commande docker-compose exec. L'ensemble de ces étapes a abouti à la confirmation que le build et les tests ont été exécutés avec succès, marqués par le message "Build and test successful!" dans la sortie du pipeline. Ce résultat positif démontre la robustesse et la fiabilité du processus d'intégration continue mis en place.

```
nour@Miracle:/mnt/c/Users/lajne/Downloads/Music/app$ docker-compose ps
NAME                IMAGE               COMMAND                  SERVICE             CREATED             STATUS
     PORTS
app-front-1         app-front           "python front_end.py"    front               3 minutes ago       Up 3 minutes
     0.0.0.0:63365->5000/tcp
app-svm-1           app-svm             "python svm_service.…"   svm                 3 minutes ago       Up 3 minutes
     0.0.0.0:63364->5001/tcp
app-vgg-1           app-vgg             "python vgg19_servic…"   vgg                 3 minutes ago       Up 3 minutes
     0.0.0.0:63363->5002/tcp
```



Lorsqu'on accède au lien http://localhost:63365/ où notre conteneur front-end est déployé, l'application fonctionne parfaitement, et les prédictions des deux modèles SVM et VGG sont correctement affichées. Cela démontre la réussite de l'intégration et du déploiement des services de Machine Learning, assurant une expérience utilisateur fluide et des résultats précis.

Conclusion

La réalisation de ce projet basé sur la technologie Docker a été une expérience enrichissante, illustrant de manière concrète la mise en œuvre d'un environnement de Machine Learning avec Python et le framework Flask. L'objectif de ce projet était de créer un système capable de classifier les genres musicaux à partir d'un ensemble de données donné, et ce, en utilisant deux modèles distincts, SVM et VGG19.

La conception d'un service web Flask pour chaque modèle, intégrée dans des conteneurs Docker, a permis une orchestration efficace des différents composants du projet. L'utilisation de Jenkins pour l'automatisation du processus de build et de test a contribué à assurer la qualité du code et à simplifier le déploiement continu.

L'intégration réussie des services SVM et VGG19 dans l'application front-end a été validée à travers des tests unitaires, confirmant ainsi le bon fonctionnement des modèles de Machine Learning. En déployant l'application, nous avons pu observer des résultats de prédiction précis et cohérents, démontrant la robustesse de l'ensemble du système.

En conclusion, ce projet a été une opportunité pratique pour acquérir des compétences essentielles en matière de déploiement d'applications basées sur des modèles de Machine Learning, et offre une base solide pour des développements futurs dans ce domaine passionnant. L'utilisation de technologies telles que Docker, Flask et Jenkins a permis d'atteindre les objectifs fixés et de créer un système fonctionnel et fiable pour la classification des genres musicaux.