



Département Informatique

Rapport du Projet

de

Nouvelles Architectures

Troisième année en Génie Informatique (NTS)

Présenté le 05/11/2024

Par

Cherni Rihab

Hammemi Wiem

Classificateur des Genres Musicaux

Encadré par : Monsieur Mehrez Boulares

Année universitaire : 2024-2025

Table des matières

Introduction générale	1
Chapitre 1: Cadre Général du Projet	2
Introduction	2
1. Contexte Général du Projet	2
2. Objectifs du Projet	2
3. Approche DevOps	3
4. Description des Modèles Utilisés	3
5. Ensemble de Données GTZAN	3
Conclusion	3
Chapitre 2: Etude Architecturale	4
Introduction	4
1. Structure du Projet	4
2. Configuration Docker Compose	5
3. Configuration de la Pipeline Jenkins	7
3.1. Structure de la Pipeline	8
3.2. Définition des Variables d'Environnement	9
3.3. Étapes de la Pipeline	9
3.4. Gestion Post-Exécution	10
Conclusion	10
Chapitre 3: Mise en œuvre et réalisation	11
Introduction	11
1. Développement des services de classification	11
1.1. svm service	11
1.2. vgg service	13
2. Interface utilisateur	14

3.	Tests et validation	14
3.1.	Résultat des tests	16
4.	Perspectives et améliorations futures	17
	Conclusion	17
	Conclusion générale	18

Table des figures

2.1	Pipeline	10
3.1	Interface utilisateur	14
3.2	Résultat du test	17

Introduction générale

Ce projet vise à concevoir un système de classification des genres musicaux à l'aide de modèles d'apprentissage automatique, déployé dans un environnement Docker. En utilisant des technologies avancées telles que Docker, Docker Compose et Jenkins, il permet une gestion simplifiée des services distincts et l'intégration des modèles d'IA. Le système repose sur deux services : 'svm_service', basé sur la machine à vecteurs de support (SVM), et 'vgg19_service', utilisant un réseau de neurones VGG19 adapté aux données audio. Ces services traitent des fichiers audio WAV encodés en base64 pour prédire les genres musicaux correspondants.

Les API RESTful, implémentées avec Flask, assurent la communication entre les services, tandis que Docker et Docker Compose orchestrent l'intégration des services en environnements isolés. Jenkins est utilisé pour automatiser les processus d'intégration continue et de déploiement, garantissant ainsi une gestion fluide du cycle de vie du projet, du développement à la production.

Les avantages de cette approche incluent l'isolation des services, la portabilité, la scalabilité et une gestion automatisée des déploiements. Jenkins facilite le suivi des modifications, la construction des conteneurs et la mise à jour automatique des services en production.

En résumé, ce projet illustre l'efficacité de la conteneurisation pour intégrer et déployer des services spécialisés tout en exploitant les capacités de l'apprentissage automatique et de l'intégration continue dans un cadre pratique et évolutif.

Cadre Général du Projet

Introduction

Ce chapitre présente le contexte général du projet, ses objectifs, ainsi que les enjeux et défis auxquels il répond. Il permet de poser les bases du travail réalisé.

1. Contexte Général du Projet

Le projet vise à développer un système de classification automatique des genres musicaux en s'appuyant sur le Machine Learning. Deux modèles d'apprentissage sont employés :

- **SVM (Support Vector Machine)** pour traiter directement les données audio.
- **VGG19 (Convolutional Neural Network)** pour analyser les spectrogrammes audio.

L'ensemble de données **GTZAN**, composé d'extraits étiquetés selon différents genres musicaux, est utilisé pour l'entraînement et les tests des modèles. Afin d'assurer une portabilité et une scalabilité optimales, les services sont conteneurisés avec Docker et orchestrés via Docker Compose. Ce projet intègre la gestion des services via Docker, l'apprentissage automatique, ainsi que l'analyse des données audio et visuelles, pour offrir une solution moderne et évolutive de classification musicale.

2. Objectifs du Projet

L'objectif principal est de déployer un système permettant la classification des genres musicaux à partir de fichiers audio. Les objectifs spécifiques sont :

- Utiliser le jeu de données GTZAN pour entraîner les modèles de Machine Learning, disponible sur Kaggle : GTZAN Dataset.
- Développer deux services de classification des genres musicaux : l'un utilisant SVM et l'autre basé sur VGG19.

- Intégrer et déployer ces services dans des conteneurs en utilisant Docker et Docker-compose.
- Concevoir une interface web intuitive permettant de soumettre des fichiers audio et de prédire leur genre musical pour interagir avec le système.
- Implémenter un pipeline d'intégration continue avec Jenkins pour automatiser les tests, le déploiement des services et garantir la qualité continue du projet.

3. Approche DevOps

L'approche DevOps est au cœur du projet pour garantir une intégration fluide et une qualité continue. Les étapes principales incluent :

- **Conteneurisation avec Docker** : Chaque service est isolé dans un conteneur pour assurer une exécution cohérente, indépendamment de l'environnement.
- **Orchestration avec Docker Compose** : Coordination des conteneurs pour déployer un système complet, incluant un front-end et les services de classification.
- **Intégration continue avec Jenkins** : Automatisation des tests unitaires et des déploiements pour assurer la qualité et minimiser les régressions.

4. Description des Modèles Utilisés

- **SVM (Support Vector Machine)** : Algorithme supervisé performant pour classifier les genres à partir de données audio brutes.
- **VGG19 (Convolutional Neural Network)** : Modèle profond exploité pour analyser les spectrogrammes, fournissant une interprétation visuelle des signaux sonores.

5. Ensemble de Données GTZAN

Le dataset **GTZAN** est une référence pour les projets de classification musicale. Il contient 1000 extraits de 30 secondes chacun, répartis équitablement entre 10 genres musicaux (rock, jazz, pop, etc.). Ces données sont utilisées pour entraîner et évaluer les modèles, avec des fichiers audio en format .wav.

Conclusion

Le projet intègre des modèles avancés de Machine Learning dans une architecture conteneurisée basée sur Docker. L'approche DevOps, incluant Jenkins, garantit une solution robuste, scalable et facile à déployer pour la classification des genres musicaux.

Etude Architecturale

Introduction

Ce chapitre présente une analyse architecturale complète du projet. Nous décrirons en détail la structure globale du projet, comprenant les trois principaux services conteneurisés et leur interconnexion. Ensuite, nous aborderons la configuration du fichier Docker Compose qui orchestre ces services. Enfin, nous examinerons la pipeline Jenkins. outils utilisés.

1. Structure du Projet

Le système est composé de trois services principaux :

- **frontend** : Le service frontend Flask qui gère l'interface utilisateur. Il expose une interface simple permettant aux utilisateurs de classifier des fichiers audio.
- **svm_service** : Le service backend utilisant un modèle SVM pour la classification des genres musicaux à partir de fichiers audio.
- **vgg19_service** : Un autre service utilisant le modèle VGG19 pour classifier des fichiers audio en passant par des images.

Structure simplifiée du Projet

```
Music-Genre-Classfier /
|-- SVM_service/
| |-- model/
| | |-- svm_model.pkl
| |-- tests/
| | |-- test_svm.py
| |-- app.py
| |-- Dockerfile
| -- requirements.txt
|-- VGG19_service/
| |-- model/
| | |-- vgg_model.h5
```



```
| |-- tests/
| | |-- test_vgg.py
| |-- app.py
| |-- Dockerfile
| -- requirements.txt
|-- frontend/
| |-- templates/
| | -- index.html
| |-- static/
| | -- style.css
| |-- app.py
| |-- Dockerfile
| -- requirements.txt
|-- docker-compose.yml
|-- Jenkinsfile
|-- README.md
```

2. Configuration Docker Compose

Le fichier `docker-compose.yml` ci-dessous est utilisé pour orchestrer plusieurs services dans le projet de classification des genres musicaux basé sur des modèles d'apprentissage automatique.

```
version: '3.8'

services:
  svm_service:
    build:
      context: ./svm_service
    container_name: svm_service
    image: devops-app/svm-service:latest
    ports:
      - "5001:5001"

  vgg19_service:
    build:
      context: ./vgg19_service
    container_name: vgg_service
    image: devops-app/vgg19-service:latest
    ports:
      - "5002:5002"

  frontend:
    build:
      context: ./frontend
    container_name: frontend
    image: devops-app/frontend:latest
    ports:
      - "5000:5000"
    environment:
      FLASK_ENV: development
```

```
depends_on:
  - svm_service
  - vgg19_service
```

Voici une description détaillée de chaque section et de son rôle dans l'application.

La version utilisée est 3.8, qui est compatible avec les fonctionnalités modernes de Docker.

Les différents services conteneurisés utilisés dans l'application :

1. svm_service

- **Description** : Service backend basé sur le modèle SVM (*Support Vector Machine*) pour la classification musicale.
- **Propriétés** :
 - **build** : Définit le contexte de construction via le fichier Dockerfile situé dans `./svm_service`.
 - **container_name** : Nom unique du conteneur : `svm_service`.
 - **image** : Image Docker utilisée : `devops-app/svm-service:latest`.
 - **ports** : Mappe le port interne 5001 du conteneur au port 5001 sur l'hôte.

2. vgg19_service

- **Description** : Service backend basé sur le modèle de réseau neuronal convolutif VGG19.
- **Propriétés** :
 - **build** : Construit l'image à partir du dossier `./vgg19_service`.
 - **container_name** : Nom unique : `vgg_service`.
 - **image** : Image Docker : `devops-app/vgg19-service:latest`.
 - **ports** : Mappe le port interne 5002 au port 5002 sur l'hôte.

3. frontend

- **Description** : Service qui représente l'interface utilisateur de l'application.
- **Propriétés** :
 - **build** : Définit le contexte de construction via le dossier `./frontend`.
 - **container_name** : Nom unique : `frontend`.
 - **image** : Image Docker : `devops-app/frontend:latest`.
 - **ports** : Mappe le port interne 5000 au port 5000 sur l'hôte.

- **environment** : Définit la variable d'environnement **FLASK_ENV** avec la valeur **development**.
- **depends_on** : Dépend des services **svm_service** et **vgg19_service**, garantissant qu'ils démarrent avant le service **frontend**.

Fonctionnement Global

- Les trois services fonctionnent ensemble de manière orchestrée.
- **svm_service** et **vgg19_service** sont les moteurs de classification basés sur des modèles distincts.
- **frontend** agit comme point d'entrée principal, coordonnant les appels aux deux services backend.
- Les ports exposés permettent un accès direct à chaque service pour des tests ou une intégration.

Cette configuration démontre comment Docker Compose simplifie le déploiement et l'intégration de plusieurs services conteneurisés dans un environnement cohérent.

3. Configuration de la Pipeline Jenkins

La pipeline suivante est utilisée pour automatiser le processus de construction, de test et de déploiement des services Docker dans le projet de classification des genres musicaux.

```
pipeline {
    agent any
    environment {
        REPORT_DIR = 'reports'
    }
    stages {
        stage('Build Docker Images') {
            steps {
                bat 'docker-compose -f docker-compose.yml build'
            }
        }
        stage('Install Dependencies') {
            steps {
                bat """
                    docker run --rm -v %WORKSPACE%\svm_service:/app devops-app/svm-service:
                        latest ^
                        pip install -r /app/requirements.txt
                    """
                bat """
                    docker run --rm -v %WORKSPACE%\vgg19_service:/app devops-app/vgg19-
                        service:latest ^
                        pip install -r /app/requirements.txt
                    """
            }
        }
    }
}
```

```

    }
    stage('Run Tests') {
        steps {
            bat """
                docker run --rm -v "%WORKSPACE%/SVM_service:/app" -v "%WORKSPACE%/reports
                :/reports" devops-app/svm-service:latest ^
                bash -c "export PYTHONPATH=/app && python /app/tests/test_svm.py"
            """
            bat """
                docker run --rm -v "%WORKSPACE%/VGG19_service:/app" -v "%WORKSPACE%/
                reports:/reports" devops-app/vgg19-service:latest ^
                bash -c "export PYTHONPATH=/app && python /app/tests/test_vgg.py"
            """
        }
    }
    stage('Verify Test Reports') {
        steps {
            bat 'dir %WORKSPACE%\\reports'
        }
    }
    stage('Start Services') {
        steps {
            bat 'docker-compose -f docker-compose.yml up -d'
        }
    }
    stage('Publish Test Results') {
        steps {
            junit 'reports/*.xml'
        }
    }
}
post {
    success {
        echo 'Pipeline completed successfully!'
    }
    failure {
        echo 'Pipeline failed. Please check the logs.'
    }
}
}

```

Voici une description des principales étapes et leur rôle.

3.1. Structure de la Pipeline

La pipeline est définie en plusieurs étapes (*stages*), avec des agents par défaut (`agent any`) et des variables d'environnement globales.

3.2. Définition des Variables d'Environnement

- `REPORT_DIR` : Indique le répertoire des rapports (`reports`), utilisé pour stocker les résultats des tests.

3.3. Étapes de la Pipeline

1- Setup :

- Prépare l'environnement pour l'exécution de la pipeline.
- Affiche un message d'information : `Setting up the environment....`

2- Build Docker Images :

- Construit les images Docker définies dans le fichier `docker-compose.yml`.
- Commande exécutée : `docker-compose -f docker-compose.yml build`.

3- Install Dependencies :

- Installe les dépendances Python pour les services `svm_service` et `vgg19_service` à partir du fichier `requirements.txt`.
- Utilise les commandes Docker pour exécuter les installations dans les conteneurs.

4- Run Tests :

- Lance les tests unitaires pour les services :
 - **SVM Service** : Exécute les tests dans `/app/tests/test_svm.py`.
 - **VGG19 Service** : Exécute les tests dans `/app/tests/test_vgg.py`.
- Sauvegarde les résultats des tests dans le répertoire `reports`.
- Génère des fichiers XML pour l'analyse ultérieure des résultats.

5- Verify Test Reports :

- Vérifie la présence des rapports de test dans le répertoire défini.
- Commande exécutée : `dir %WORKSPACE%\reports`.

6- Start Services :

- Démarre tous les services définis dans le fichier `docker-compose.yml`.
- Commande exécutée : `docker-compose -f docker-compose.yml up -d`.

7- Publish Test Results :

- Publie les résultats des tests en utilisant le plugin `junit`.
- Commande : `junit 'reports/*.xml'`.

3.4. Gestion Post-Exécution

La section **post** gère les actions à effectuer à la fin de la pipeline :

- **Always** : Nettoie l'environnement en arrêtant et supprimant les conteneurs Docker (`docker-compose -f docker-compose.yml down`).
- **Success** : Affiche un message de succès : `Pipeline completed successfully!`.
- **Failure** : Affiche un message d'erreur en cas d'échec : `Pipeline failed. Please check the logs..`

Cette configuration de pipeline permet une automatisation complète du cycle de vie de développement : de la construction des services à l'exécution des tests et à leur déploiement.

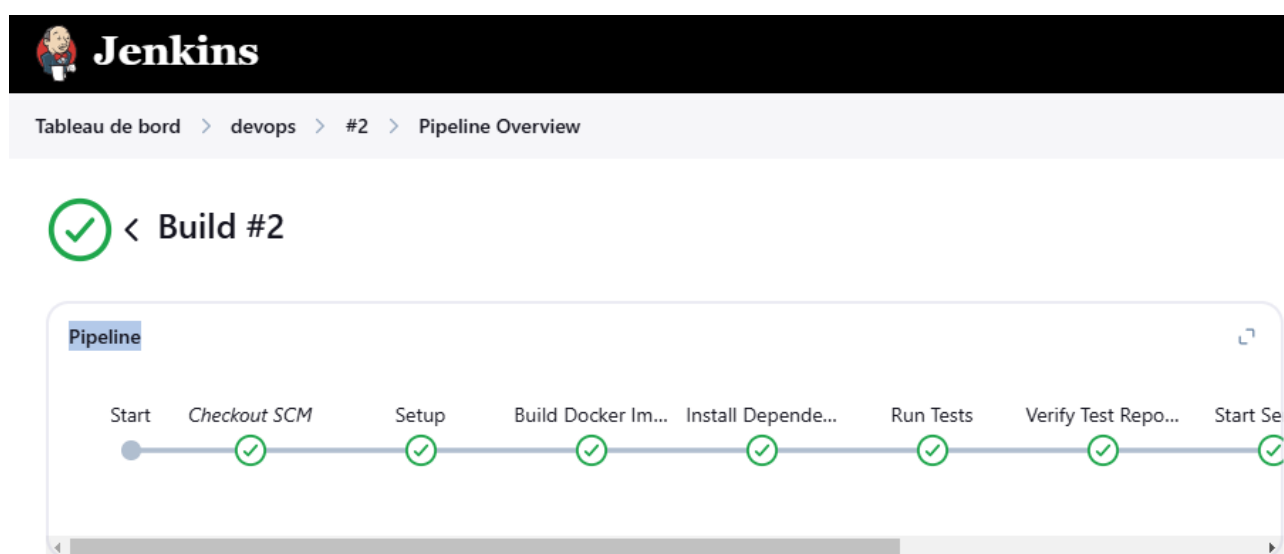


Figure 2.1 – Pipeline

Conclusion

Dans ce chapitre, nous avons détaillé l'architecture du projet, en mettant en lumière la structuration des services, leur orchestration via Docker Compose, et leur intégration dans une pipeline Jenkins. Cette étude a permis de comprendre comment chaque composant collabore pour offrir une solution fonctionnelle et maintenable.

Mise en œuvre et réalisation

Introduction

Ce chapitre explore le développement des services de classification des genres musicaux. Chaque service joue un rôle clé dans la mise en œuvre de la classification, offrant une structure modulaire et maintenable. Nous nous concentrerons sur l'implémentation du service `svm_service`, qui repose sur Flask et plusieurs bibliothèques Python pour fournir une API fonctionnelle et performante. Enfin, nous détaillerons les outils et technologies utilisés pour construire et déployer ce service dans un environnement Docker.

1. Développement des services de classification

1.1. svm service

Le `svm_service` est un service web dédié à la classification des genres musicaux à l'aide d'un modèle d'apprentissage automatique SVM. Ce service est implémenté en Python à l'aide du microframework Flask et utilise des bibliothèques comme `scikit-learn`, `librosa`, et `numpy` pour le traitement des données audio et les prédictions.

Outils et bibliothèques : `requirements.txt`

```
flask
scikit-learn
librosa
numpy
xmlrunner
```

Le service est conteneurisé à l'aide de Docker, facilitant ainsi son déploiement et son exécution sur différents environnements. Voici le contenu du fichier Dockerfile :

```
FROM python:3.9-slim
```

```
WORKDIR /app

COPY . /app

RUN pip install -r requirements.txt

EXPOSE 5001

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

Le fichier app.py Le fichier app.py constitue l'élément central du service svm_service, en assurant la liaison entre l'interface utilisateur et le modèle de classification SVM. Il utilise Flask pour gérer les requêtes HTTP, scikit-learn pour la prédiction des genres musicaux, et librosa pour le traitement des données audio. Ce fichier intègre également des fonctionnalités pour le décodage des fichiers audio, la gestion des erreurs, et le déploiement du service sur un serveur Flask.

```
from flask import Flask, request, jsonify
import base64
import pickle
import numpy as np
import librosa
from io import BytesIO
import xmlrunner

app = Flask(__name__)

# Charger le modèle SVM
with open("model/svm_model.pkl", "rb") as f:
    svm_model = pickle.load(f)

# Genres list (same order as in your training data)
genres = ["blues", "classical", "country", "disco", "hiphop", "jazz", "metal", "pop", "reggae",
          , "rock"]

def predict_genre(audio_data, clf):
    # Load and preprocess the audio file
    signal, rate = librosa.load(BytesIO(audio_data))
    hop_length = 512
    n_fft = 2048
    n_mels = 128
    S = librosa.feature.melspectrogram(y=signal, sr=rate, n_fft=n_fft, hop_length=hop_length,
                                       n_mels=n_mels)
    S_DB = librosa.power_to_db(S, ref=np.max)
    S_DB = S_DB.flatten()[:1200]

    # Predict the genre
    genre_label = clf.predict([S_DB])[0]
```



```
    return genres[genre_label]

# Classe pour l'erreur de traitement
class AudioProcessingError(Exception):
    pass

@app.route('/predict_svm', methods=['POST'])
def predict_svm():
    try:
        data = request.json.get('wav_music')
        if not data:
            return jsonify({'error': 'Aucun fichier audio fourni'}), 400

        audio_data = base64.b64decode(data)

        predicted_genre = predict_genre(audio_data, svm_model)

        return jsonify({'genre': predicted_genre})

    except AudioProcessingError:
        return jsonify({'error': 'Erreur de traitement du fichier audio'}), 500
    except Exception as e:
        return jsonify({'error': str(e)}), 500

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5001)
```

1.2. vgg service

Le vgg_service est un service web dédié à la classification des genres musicaux en s'appuyant sur un modèle d'apprentissage profond VGG19.

Le modèle VGG19 a été pré-entraîné sur des spectrogrammes générés à partir des fichiers audio, exploitant ses capacités de détection de caractéristiques pour une classification précise.

Ce service est également conteneurisé avec Docker, ce qui permet un déploiement simple et une exécution cohérente sur différents environnements. Voici un exemple du fichier Dockerfile utilisé pour sa mise en conteneur :

```
FROM python:3.9-slim

WORKDIR /app

COPY . /app

RUN pip install -r requirements.txt
```

```
EXPOSE 5002
```

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

Les étapes mises en œuvre pour le modèle VGG sont quasiment identiques à celles fournies avec le modèle SVM, en tenant compte des spécificités propres à l'architecture VGG19.

2. Interface utilisateur

L'interface utilisateur du projet "Music Genre Classification" permet à l'utilisateur de soumettre un fichier audio WAV pour le classer selon deux modèles : SVM et VGG19. L'interface est simple, composée de deux sections principales, chacune permettant de télécharger un fichier et de soumettre ce dernier pour classification. Après traitement, les résultats de la classification sont affichés dynamiquement. L'interface est conçue pour être intuitive et réactive, avec des messages d'erreur en cas de problème. Cette approche garantit une interaction fluide et efficace avec les services Flask déployés dans un environnement Docker.

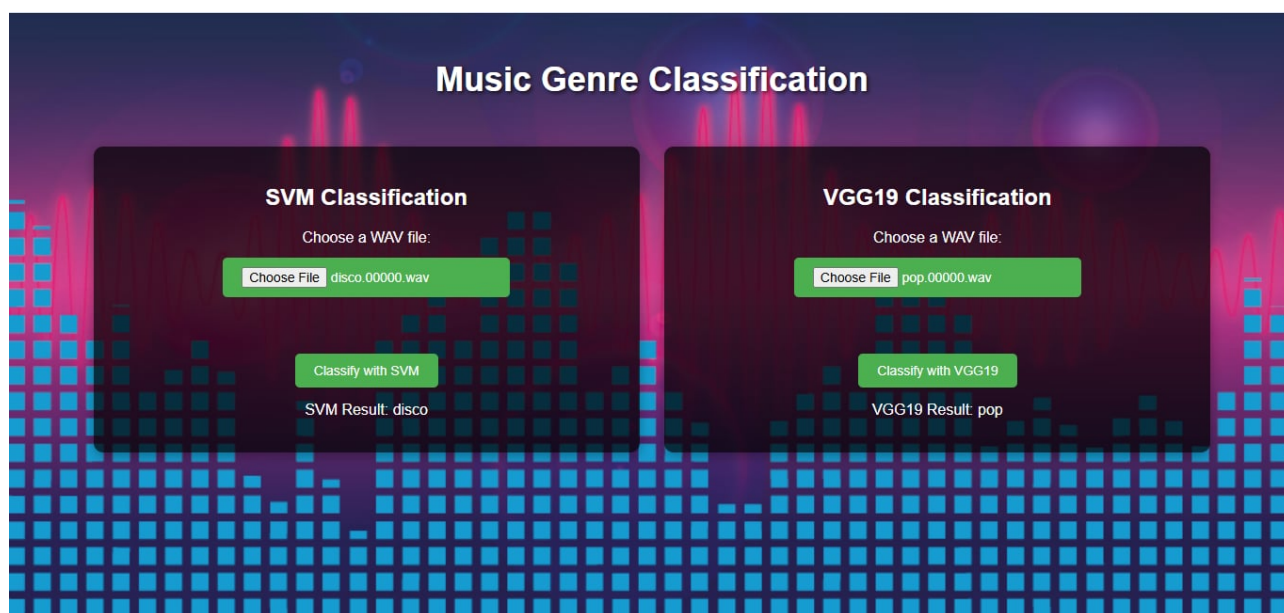


Figure 3.1 – Interface utilisateur

3. Tests et validation

Les tests effectués sur le service de classification SVM ont pour objectif de valider son bon fonctionnement dans différentes situations, en assurant la fiabilité des résultats fournis par le modèle. Plusieurs scénarios de test ont été élaborés pour vérifier les performances et la gestion des erreurs du service.

Les tests incluent :

- Test avec un fichier audio invalide : Vérifie la réponse du système lorsqu'un fichier audio non valide est soumis.
- Test sans fichier audio : Vérifie la gestion de la soumission d'une requête sans fichier audio.
- Test avec un fichier audio valide : Vérifie le bon fonctionnement du modèle SVM avec un fichier audio valide.
- Test de gestion des erreurs du service : Vérifie que le service gère correctement les erreurs internes.

Le fichier de test utilisé est le suivant :

```
import unittest
import base64
import json
from app import app
from io import BytesIO
import xmlrunner

class TestSVMService(unittest.TestCase):
    def setUp(self):
        self.client = app.test_client()
        self.client.testing = True

    def test_predict_svm_valid_audio(self):
        # Charger un fichier audio de test en base64
        with open("tests/test_audio.wav", "rb") as f:
            audio_data = f.read()
            base64_audio = base64.b64encode(audio_data).decode('utf-8')

        # Envoyer une requête POST avec le fichier audio
        response = self.client.post(
            '/predict_svm',
            data=json.dumps({'wav_music': base64_audio}),
            content_type='application/json'
        )

        # Vérifier la réponse
        self.assertEqual(response.status_code, 200)
        data = json.loads(response.data)
        self.assertIn('genre', data) # Vérifier si le genre est dans la réponse
        self.assertIn(data['genre'], ["blues", "classical", "country", "disco",
                                       "hiphop", "jazz", "metal", "pop", "reggae", "rock"])

    def test_predict_svm_no_audio(self):
        # Tester l'absence de fichier audio
        response = self.client.post(
            '/predict_svm',
            data=json.dumps({}), # Pas de fichier audio fourni
            content_type='application/json'
```

```

    )

    # V rifier la r ponse
    self.assertEqual(response.status_code, 400)
    data = json.loads(response.data)
    self.assertIn('error', data)
    self.assertEqual(data['error'], 'Aucun fichier audio fourni')

def test_predict_svm_invalid_audio(self):
    # Envoyer des donn es non valides
    response = self.client.post(
        '/predict_svm',
        data=json.dumps({'wav_music': 'invalid_base64'}),
        content_type='application/json'
    )

    # V rifier la r ponse
    self.assertEqual(response.status_code, 500)
    data = json.loads(response.data)
    self.assertIn('error', data)

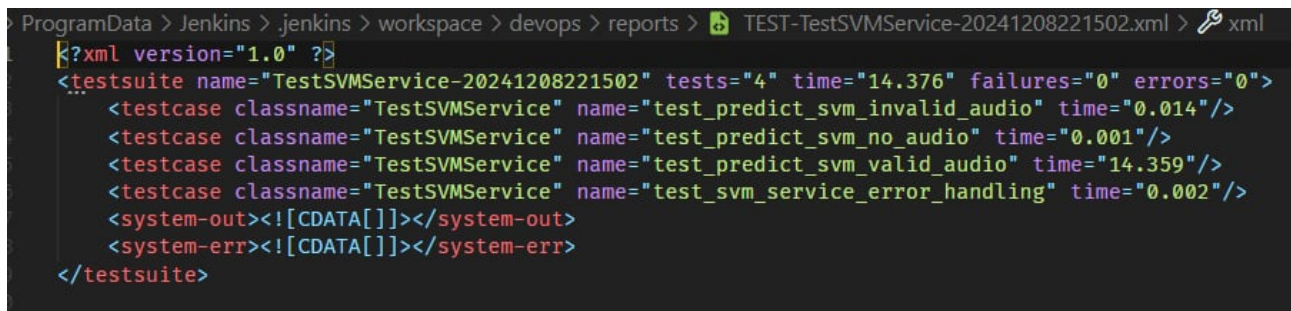
def test_svm_service_error_handling(self):
    # Simuler un cas o une exception interne se produit
    with app.test_request_context('/predict_svm', method='POST'):
        response = self.client.post(
            '/predict_svm',
            data=json.dumps({'wav_music': base64.b64encode(b"invalid data").decode()}),
            content_type='application/json'
        )
        self.assertEqual(response.status_code, 500)
        data = json.loads(response.data)
        self.assertIn('error', data)

if __name__ == "__main__":
    unittest.main(
        testRunner=xmlrunner.XMLTestRunner(output='/reports'),
        failfast=False,
        buffer=False,
        catchbreak=False,
    )

```

3.1. Résultat des tests

Les résultats des tests montrent que le service SVM fonctionne correctement, avec tous les tests réussis et aucun échec ou erreur. Le rapport XML montre les détails des tests exécutés, leur durée, et les résultats de chacun :



```
ProgramData > Jenkins > jenkins > workspace > devops > reports > TEST-TestSVMService-20241208221502.xml > xml
<?xml version="1.0" ?>
<testsuite name="TestSVMService-20241208221502" tests="4" time="14.376" failures="0" errors="0">
  <testcase classname="TestSVMService" name="test_predict_svm_invalid_audio" time="0.014"/>
  <testcase classname="TestSVMService" name="test_predict_svm_no_audio" time="0.001"/>
  <testcase classname="TestSVMService" name="test_predict_svm_valid_audio" time="14.359"/>
  <testcase classname="TestSVMService" name="test_svm_service_error_handling" time="0.002"/>
  <system-out><![CDATA[]]></system-out>
  <system-err><![CDATA[]]></system-err>
</testsuite>
```

Figure 3.2 – Résultat du test

Tous les tests ont été exécutés avec succès, validant ainsi la stabilité et la précision du service de classification SVM.

4. Perspectives et améliorations futures

Les perspectives d'amélioration incluent :

- L'ajout de plus de genres musicaux au jeu de données d'entraînement.
- L'amélioration de la précision des modèles en utilisant des techniques de fine-tuning supplémentaires.
- L'optimisation des services pour une meilleure performance et scalabilité.

Conclusion

Ce chapitre a présenté en détail la mise en œuvre du projet, en se focalisant sur les services de classification et l'interface utilisateur, tout en validant leur bon fonctionnement par des tests rigoureux. Les tests réalisés confirment la fiabilité des services, renforçant ainsi la robustesse du projet.

Conclusion générale

Le projet présenté démontre la puissance et la synergie des technologies modernes pour résoudre des problèmes complexes tels que la classification automatique des genres musicaux. En combinant l'apprentissage automatique, la conteneurisation avec Docker et l'intégration continue via Jenkins, nous avons pu concevoir un système performant, portable et évolutif.

Les deux modèles développés, SVM et VGG19, offrent des approches complémentaires pour traiter et analyser les données audio. Leur intégration dans une architecture conteneurisée garantit une isolation des services et facilite le déploiement en production. Par ailleurs, la mise en place d'une pipeline DevOps a permis d'assurer une automatisation efficace des tests et des déploiements, garantissant ainsi la qualité et la robustesse du système.

Ce travail met également en évidence l'importance des bonnes pratiques en développement logiciel, notamment en termes d'organisation, de scalabilité et de fiabilité des systèmes. Les perspectives d'amélioration, comme l'intégration de nouveaux modèles ou l'extension de la base de données, ouvrent la voie à des applications encore plus performantes et adaptées aux besoins spécifiques des utilisateurs.

En conclusion, ce projet illustre comment les innovations technologiques peuvent être exploitées pour concevoir des solutions pratiques et efficaces, répondant aux exigences des environnements modernes tout en offrant une expérience utilisateur enrichissante.