PyTune

Rapport MLOps

DataScientest promo MLOps septembre 2022

Yann HUET

Table des matières

Index des figures

I. INTRODUCTION

Dans le cadre de la formation de « Machine Learning Engineer » j'ai réalisé un projet de mis en production d'un modèle de recommandation musicale. Ce projet est la suite logique du projet fil rouge PyTune réalisé lors de la formation « Data Scientist » en 2021 (cf *Rapport fil rouge - Datascientest – PyTune.pdf).*

Comme base de départ j'ai choisi de partir du modèle le plus performant entraîné lors de la partie data science à savoir le modèle als_implicit de la librairie Implicit (cf 9.b.iii du rapport cité ci-dessus). Le but de ce projet sera de mettre en production ce modèle à l'aide d'une API, d'une base de données ainsi que d'un système de monitoring du modèle.

Ce projet est disponible sur le dépôt GIT suivant : https://github.com/Alumet/pytune.git

II. Choix des métriques

Le sujet du choix de la métrique à été traité dans le rapport « *Rapport fil rouge - Datascientest – PyTune.pdf » cf chapitre 7. Comme décrit, les métriques les plus pertinentes dans le cas d'un système de recommandation sont le P@K, NDGC et L'AUC.*

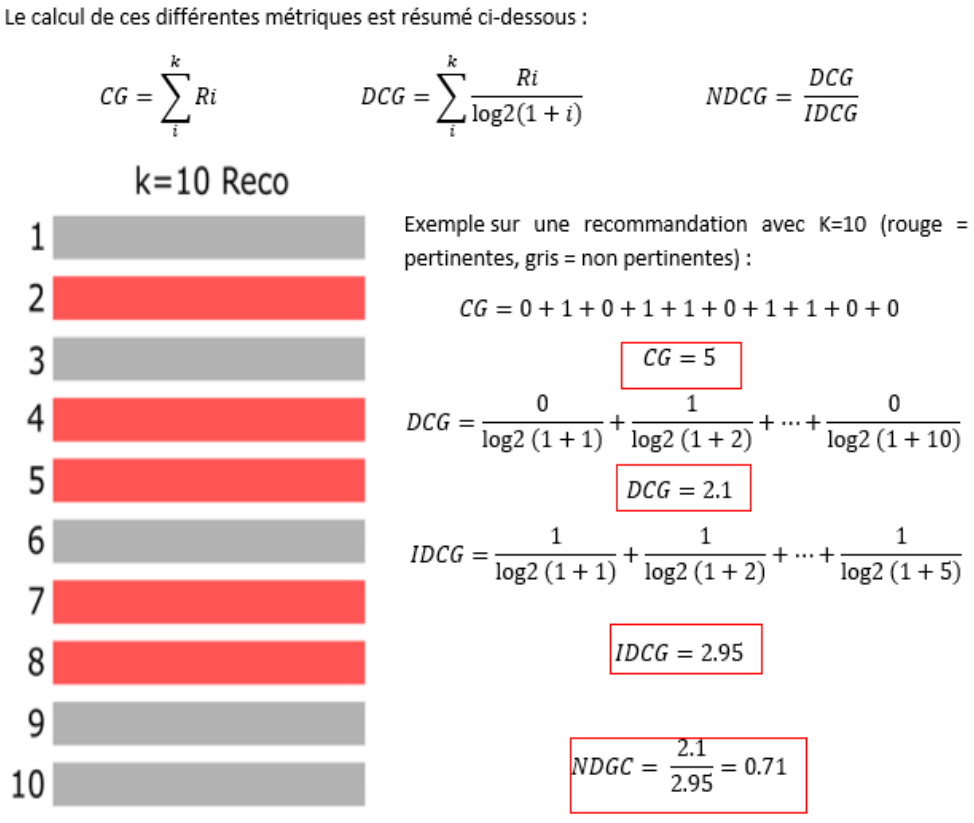

Figure 1: Métrique NDGC

Dans notre cas, le NDGC sera la métrique principalement utilisée pour évaluer et monitorer le modèle. Elle nous permet de mesurer la capacité du modèle à placer en tête dans les N première recommandations le plus d'éléments pertinents. Au final ce n'est pas intéressent de pouvoir recommander un bon élément à la 200^{ém} place d'une liste alors que l'utilisateur final n'aura d'interaction qu'avec les 20 premiers. Cependant, l'AUC donnera une vision plus globale sur bonne performance du modèle sur l'ensemble des données (cf *Rapport fil rouge - Datascientest – PyTune.pdf » cf chapitre 7)*

III. API

L'API sera sécurisée par un système d'authentification et devra fournir à l'utilisateur une liste de recommandation personnalisée de N titres. Elle fera aussi l'interface avec la base de données pour réaliser une recherche basique de titre ainsi que de loger les évènements d'écoutes (interactions user/item).

Les routes seront divisées en deux groupes, celles réservées aux utilisateurs et celles réservées aux administrateurs. Seule une route de test sera disponible librement pour tester le bon fonctionnement de l'API.

L'API sera développée autour de la libraire FastAPI disponible sous Python et isolée dans un conteneur docker et mis à disposition sur DockerHub.

III.I. End-points et fonctionnalités

- **« / »** Retourne « running » si en vie
- **« /login »** Permet de tester si un utilisateur existe et si son login est valide
- **« /search »** Réaliser une recherche simple d'un titre dans la base de donnée
- **« /recommendation »** Avoir une recommandation personnalisée de N titres
- **« /similar »** Avoir N titre similaire un autre
- **« /event »** Ajouter un évènement d'écoute dans la base de donnée
- **« *admin*/model/reaload »** Charger le modèle de production
- **« admin/model/train »** Lancer un entraînement du modèle sur l'ensemble des données
- **« admin/user »** Ajouter un nouvel utilisateur (username + pw)
- **« admin/recommendation »** Avoir une recommandation de N titres pour un utilisateur spécifique

IV. Base de données

La base de données devra stocker les informations relatives aux titres, artistes et utilisateurs ainsi que l'ensemble des interactions user/items. Considérant que les champs n'ayant pas vocation à évoluer et la bonne structuration des données j'ai fait le choix d'utiliser MySQL.

Elle sera composée des tables suivantes :

- User : Information sur les utilisateurs
- Tracks : information sur les titres de musiques
- Artists : information sur les artistes
- User_item : interaction en utilisateurs et titres de musique
- Prediction : permet de garder une trace des recommandations à un moment donné
- Training : garde une trace des métriques pour chaque entraînement du modèle

IV.I. Structure de la base de donnée.

clés primaires clés secondaires

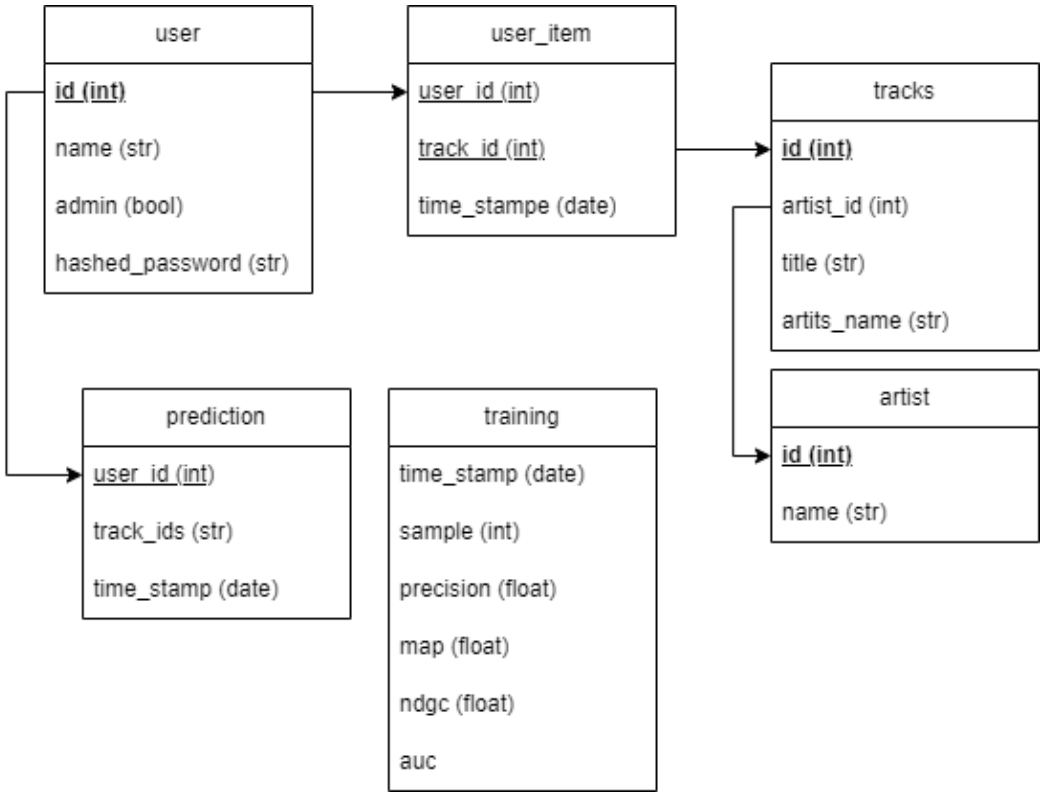

Figure 2: Diagrame base de données

V. AirFlow

AirFlow couvrira la partie automatisation et monitoring. Il nous permettra de :

- mettre à jour les données (à partir du fichier CSV)
- entraîner un nouveau modèle
- comparer les performances du nouveau modèle avec le dernier en production
- alerter dans le cas d'une dérive du modèle
- mettre en production le nouveau modèle sur l'API

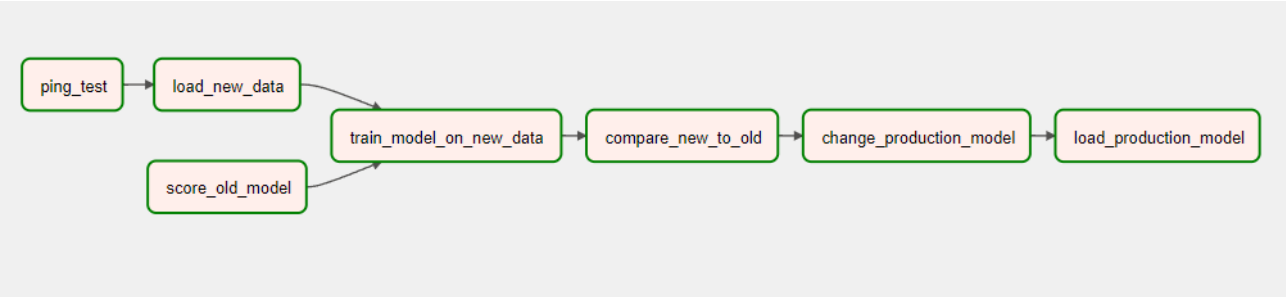

Figure 3: Workflow pytune_dag.py

VI. WebApp

Pour compléter la solution sur le plan « front », une interface web simple sera développée à l'aide de la librairie Streamlit. Elle permettra à un utilisateur de s'identifier, rechercher un titre, avoir une recommandation personnalisée et d'écouter les titres à partir de 'YouTube'.

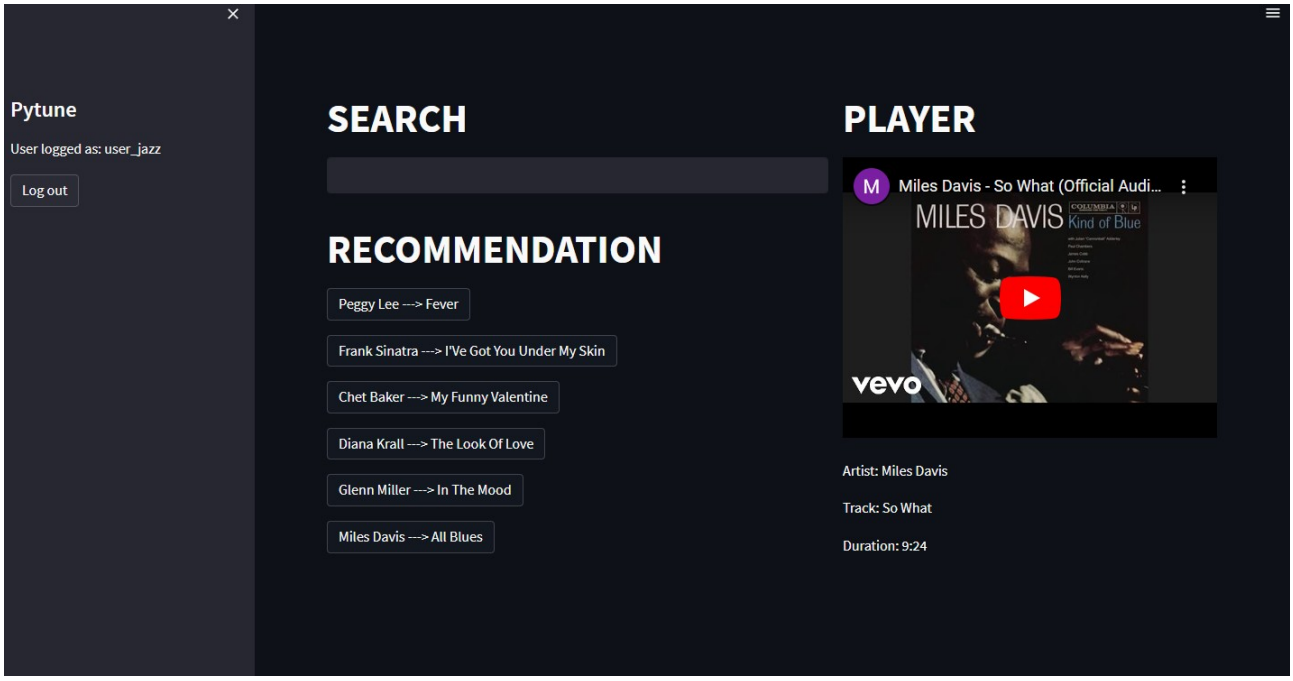

Figure 4: Pytune webb app interface

VII. Docker

Pour un déploiement plus aisée, les différentes parties de cette solution seront isolées à l'aide de conteneur docker. Ces docker seront les suivants :

- **pytune_api :** API python sous FastAPI
- **pytune_bdd :** Base de donnée MySQL
- **pytune_webapp :** Webapp python sous Streamlit
- **Airflow :** Airflow (surchagé avec custom build)

Ces conteneurs seront organisés en trois docker compose qui seront utilisables de manière modulaire autour de l'application principale (API) et seront donc tous regroupé sur un unique network prédéfini nommé **pytune**.

- **Core App**
  - pytune_api
  - pytune_bdd
- **Airflow**
  - airflow_flower
  - airflow_airflow-scheduler
  - airflow_airflow-webserver
  - airflow_airflow-worker
  - postgres:13

- redis:latest

- **Front**
  - pytune_webapp

VIII. CI/CD

La partie CI/CD sera gérée à l'aide des github-actions. Le workflow sera séparé en deux :

- La partie test unitaires qui seront déclenchés sur les branches « preproduciton » et « main »
- la partie docker qui fera le build et le push des images :
  - alumet/pytune_api:latest
  - alumet/pytune_webapp:latest