

Sommaire

I.	Introduction.....	3
1)	Problématique.....	3
2)	Présentation du jeu de données	3
3)	Environnement de développement	3
II.	Modélisation/Notebook.....	4
1)	Notebook fourni	4
2)	Adaptation du notebook à la problématique	4
III.	API.....	5
1)	Choix de FastAPI	Erreur ! Signet non défini.
2)	Description de l'API	5
3)	Description de la classe ModelFromFiles.....	5
4)	Utilisation d'un type Enum.....	6
IV.	Test de l'API.....	6
1)	Description	6
2)	Tests effectués	7
3)	Résultats du test.....	10
V.	Docker	8
1)	Docker de l'API	8
a.	Source	8
b.	Commandes	9
2)	Docker du script de test	9
a.	Source	9
b.	Commandes	10
3)	Docker Compose	10
VI.	K8S.....	11
1)	Deployment.....	11
c.	Présentation	11
d.	Source	11
e.	Commandes	12

2) Service	12
a. Présentation	12
b. Source	12
c. Commandes	13
3) Ingress	13
a. Présentation	13
b. Source	13
c. Commandes	13
4) Accès	13
VII. Conclusion	14

I. Introduction

1) Problématique

Le but de ce projet est de **mettre en production un modèle d'analyse de sentiment** construit sur le jeu de données de commentaires sur Disneyland.

<https://www.kaggle.com/arushchillar/disneyland-reviews>

L'objectif ici est de déployer des modèles déjà fournis. Attention, en production, les modèles ne devront pas être relancés.

Une API devra permettre **d'interroger les différents modèles**. Les utilisateurs pourront aussi interroger l'API pour **accéder aux performances de l'algorithme** sur les jeux de tests. Enfin, il faut permettre aux utilisateurs **d'utiliser une identification basique**. On pourra utiliser la liste d'utilisateurs/mots de passe suivante :

alice: wonderland

bob: builder

clementine: mandarine

2) Présentation du jeu de données

Le jeu de données inclut 42,000 avis sur 3 parcs Disneyland - Paris, Californie et Hong Kong, postés par des visiteurs sur Trip Advisor.

Colonnes:

Review_ID: unique id given to each review

Rating: ranging from 1 (unsatisfied) to 5 (satisfied) => target

Year_Month: when the reviewer visited the theme park

Reviewer_Location: country of origin of visitor

Review_Text: comments made by visitor => feature

Disneyland_Branch: location of Disneyland Park

3) Environnement de développement

La formation nous a permis d'utiliser différents outils de MLE sous Linux.

Ayant utilisé cet environnement lors des projets de validation des différentes parties de la formation, j'ai décidé de transposer ces principes sous un **environnement Windows**.

Pour cela, j'ai utilisé Visual Studio Code.

Cet IDE offre des fonctionnalités intéressantes pour un tel projet

- IntelliSense
- débogage (pas à pas, points d'arrêt ...)
- intégration de Docker (avec Docker Desktop)
- déploiement facilité dans le Cloud Azure
- nombreuses extensions selon les besoins

II. Modélisation/Notebook

1) Notebook fourni

Le jeu de données est partitionné en jeu d'entraînement et jeu de test.

On applique un **CountVectorizer** sur les features (i.e. la colonne "Review_Text").

Le notebook fourni contient **4 modèles** distincts :

- **Modèle 1**: LogisticRegression() appliquée sur l'ensemble des données d'entraînement.
- **Modèles 2/3/4**: RandomForestClassifier(n_estimators=20, max_depth=5) appliqué uniquement sur les parcs HK/Californie/Paris

2) Adaptation du notebook à la problématique

2 objets sont nécessaires pour appliquer les modèles au sein de l'API sans devoir les entraîner à chaque fois :

- le **CountVectorizer**

et

- le **Modèle**

qui auront été tous les deux générés à partir des données d'entraînement.

On sauvegarde donc ces 2 objets en format **pickles** pour chacun des modèles.

On obtient alors les fichiers suivants qui seront stockés dans le sous-répertoire /model_pickles/ :

- **count_vectorizer{i}.pkl**
- **model{i}.pkl**

Ces fichiers permettront de faire des prédictions sur les données saisies par l'utilisateur de l'API.

De plus, afin d'optimiser l'API, 2 autres objets nécessaires au préprocessing des données sont sauvegardés dans ce même sous-répertoire :

- NLTKWordTokenizer.pkl
- stopwords.pkl







III. API

[damienld/ML-Project: MLE Training Project \(Datascientest\) \(github.com\)](https://github.com/damienld/ML-Project)

FastAPI a été utilisé pour développer l'API.

Authorize 

default

GET	/get_username	Get Username	 
GET	/	Index	
GET	/get_performance	Get Performance	
POST	/text_to_sentiment/	Text To Sentiment	 

1) Description de l'API

L'API comporte 4 endpoint :

- "/" : retourne {score : 1} si l'API fonctionne
- "/get_username" : retourne le **username** actuellement utilisé *[nécessite une authentification préalable]*
- "/get_performance" : retourne le **score** du modèle
- "/text_to_sentiment/" : retourne la **prédiction** (score de 1 à 5) associé au texte saisi *[nécessite une authentification préalable]*

2) Description de la classe ModelFromFiles

Les 2 derniers endpoint s'appuient sur la classe « **ModelFromFiles** » définie dans le fichier **model_to_load.py**

Cette classe permet de :

- **Charger un modèle** (parmi les 4 existants) à partir des fichiers pré-enregistrés
« count_vectorizer{i}.pkl » et « model{i}.pkl »
⇒ `def _load_from_pickles_files(self)`
- **Pré-traiter un texte** afin de le rendre exploitable par le modèle
⇒ `def preprocess(self, text, pkl_stopwords, pkl_tokenizer)`
- **Effectuer une prédiction** à partir d'un texte
⇒ `def predict(self, text, pkl_stopwords, pkl_tokenizer)`

3) Utilisation d'un type Enum

Afin d'empêcher la saisie d'un numéro de modèle inexistant, il a été choisi d'utiliser un type Enum :

```
class EnumModel(IntEnum):
    AllBranch = 1
    HK = 2
    California = 3
    Paris = 4
```

Ainsi, FastAPI renvoie automatiquement un code HTTP 422 si le numéro de modèle ne correspond pas lors de l'appel d'une des fonctions. Ceci évite de devoir contrôler cette valeur manuellement ou de définir des contraintes au niveau des paramètres de fonctions.

IV. Test de l'API

[damienld/MLE-Project](#) test: [TEST Project for MLE-Project \(github.com\)](#)

1) Description

Le script `api_test.py` doit être appelé avec les paramètres suivants :

- 1: "127.0.0.1" (ip address)
- 2: "8000" (port)
- 3: "permissions"(endpoint)
- 4: (sentence)
- 5: (model_index)
- 6: (username)
- 7: (password)

Ce script se base sur un **dictionnaire** :

- **Clé** : concaténation des **paramètres** d'appel possibles
- **Valeur** : un tuple (**code_http_attendu**, **score_attendu**)

Lorsqu'un test est lancé on cherche dans ce dictionnaire quels valeurs sont attendus (code http, score). Si ces valeurs sont correctes, le test est réussi.

2) Tests effectués

On teste les paramètres suivants (chaque paramètre est séparé par un « # »).

`lst_sentences` est une liste de phrase à tester.

```
#test du endpoint « / » => [code HTTP attendu = 200, score attendu = 1]
    '####': (200, '1')

#test => bad username, 401 attendu
    'text_to_sentiment#{sentence}#1#alice1#wonderland'.format(sentence=lst_sentences[0]): (401, '')

#test => bad password, 401 attendu
    'text_to_sentiment#{sentence}#1#alice#wonderland1'.format(sentence=lst_sentences[0]): (401, '')

#test bad model_index=6 => status 422 attendu
    'text_to_sentiment#{sentence}#6#alice#wonderland'.format(sentence=lst_sentences[0]): (422, '')

#test modèle 1=> score = 1
    'text_to_sentiment#{sentence}#1#alice#wonderland'.format(sentence=lst_sentences[0]): (200, '[1]')

#test modèle 1=> score = 3
    'text_to_sentiment#{sentence}#1#alice#wonderland'.format(sentence=lst_sentences[1]): (200, '[3]')

#test modèle 2=> score = 4
    'text_to_sentiment#{sentence}#2#alice#wonderland'.format(sentence=lst_sentences[2]): (200, '[4]')

#test modèle 3=> score = 3
    'text_to_sentiment#{sentence}#3#alice#wonderland'.format(sentence=lst_sentences[3]): (200, '[3]')

#test modèle 4=> score = 2
    'text_to_sentiment#{sentence}#4#alice#wonderland'.format(sentence=lst_sentences[4]): (200, '[2]') #test => score = 2
```

V. Docker

Les images Docker sont disponibles sous Docker Hub.

[Docker MLE Project](#)

[Docker MLE Project Test](#)

1) Docker de l'API

a. Source

For more information, please refer to <https://aka.ms/vscode-docker-python>

FROM python:3.7-slim-buster

EXPOSE 8000

Keeps Python from generating .pyc files in the container

ENV PYTHONDONTWRITEBYTECODE=1

Turns off buffering for easier container logging

ENV PYTHONUNBUFFERED=1

Install pip requirements

COPY requirements.txt .

RUN python -m pip install -r requirements.txt

#copy /model_pickles and check

COPY model_pickles/ /model_pickles/

RUN ls -la /model_pickles/*

WORKDIR /app

COPY . /app

Creates a non-

root user with an explicit UID and adds permission to access the /app folder

For more info, please refer to <https://aka.ms/vscode-docker-python-configure-containers>

RUN adduser -u 5678 --disabled-password --gecos "" appuser && chown -

R appuser /app

USER appuser

During debugging, this entry point will be overridden. For more information, please refer to <https://aka.ms/vscode-docker-python-debug>

CMD ["gunicorn", "--bind", "0.0.0.0:8000", "-k", "uvicorn.workers.UvicornWorker", "api:app"]

b. Commandes

`docker image build . -t dami1ld/mleproject:latest`

`docker image push dami1ld/mleproject:latest`

`docker image pull dami1ld/mleproject:latest`

2) Docker du script de test

a. Source

`# For more information, please refer to https://aka.ms/vscode-docker-python`

`FROM python:3.7-slim-buster`

`# Keeps Python from generating .pyc files in the container`

`ENV PYTHONDONTWRITEBYTECODE=1`

`# Turns off buffering for easier container logging`

`ENV PYTHONUNBUFFERED=1`

`# Install pip requirements`

`COPY requirements.txt .`

`RUN python -m pip install -r requirements.txt`

`WORKDIR /app`

`COPY . /app`

`# Creates a non-`

`root user with an explicit UID and adds permission to access the /app folder`

`# For more info, please refer to https://aka.ms/vscode-docker-python-configure-containers`

`RUN adduser -u 5678 --disabled-password --gecos "" appuser && chown -`

`R appuser /app`

`USER appuser`

`# During debugging, this entry point will be overridden. For more information, please refer to https://aka.ms/vscode-docker-python-debug`

`# !!! Replace $ by \$ as it is a special character`

`CMD python3 api_test.py "127.0.0.1" "8000" "" "" "" "" "" \`

`&& python3 api_test.py "127.0.0.1" "8000" "text_to_sentiment" "Visited 21 5 2014.`

`This park is a joke, three main rides were closed in one park..." \`

`&& python3 api_test.py "127.0.0.1" "8000" "text_to_sentiment" "Visited 21 5 2014.`

`This park is a joke, three main rides were closed in one park..." 1 "alice" "wonderland1" \`

```
&& python3 api_test.py "127.0.0.1" "8000" "text_to_sentiment" "Visited 21 5 2014.  
This park is a joke, three main rides were closed in one park..." 6 "alice" "wonderland" \
```

```
&& python3 api_test.py "127.0.0.1" "8000" "text_to_sentiment" "Visited 21 5 2014.  
This park is a joke, three main rides were closed in one park..." 1 "alice" "wonderland" \
```

b. Commandes

```
docker image build . -t dami1ld/mleprojecttest:latest
```

```
docker image push dami1ld/mleprojecttest:latest
```

```
docker image pull dami1ld/mleprojecttest:latest
```

3) Docker Compose

Le Docker Compose permet de lancer les 2 conteneurs. D'abord le Docker de l'API puis ensuite le Docker de test de l'API qui affiche dans la console le résultat de chacun des tests.

Les 2 communiquent au travers du réseau « host »

```
version: '3.4'
```

```
services:
```

```
  mleproject:
```

```
    image: dami1ld/mleproject:latest
```

```
    network_mode: host
```

```
  mleproject_test:
```

```
    depends_on:
```

```
      - mleproject
```

```
    image: dami1ld/mleprojecttest:latest
```

```
    network_mode: host
```

4) Résultats du test

Sous Docker Desktop, on obtient par exemple (2 tests):

```

=====
API test
=====
Request done at "/text_to_sentiment"
| sentence=Visited 21 5 2014. This park is a joke, three main rides were closed in one park...
| model_index=6
| username="alice"
| password="wonderland"
=> Test(expected vs actual) / HTTP Status: 422 vs 422 / Score: vs ==> success

8 argument(s) are sent

=====
API test
=====
Request done at "/text_to_sentiment"
| sentence=Visited 21 5 2014. This park is a joke, three main rides were closed in one park...
| model_index=1
| username="alice"
| password="wonderland"
=> Test(expected vs actual) / HTTP Status: 200 vs 200 / Score: [1] vs [1] ==> success

```

VI. K8S

Les fichiers Kubernetes sont stockés dans le sous-répertoire /k8s/

1) Deployment

a. Présentation

Un fichier deployment.yml est créé. Il permet le lancement du conteneur de l'API qui répondra aux requêtes sur le **port 8000** du cluster. Ce conteneur sera répliqué **3 fois**.

Le conteneur est chargé depuis **l'image stockée en ligne dans Docker Hub**.

b. Source

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: mle-deployment
  labels:
    app: mle-api
spec:
  replicas: 3
  selector:
    matchLabels:
      app: mle-api
  template:
    metadata:

```

```

labels:
  app: mle-api
spec:
  containers:
  - name: mle-api
    image: dami1ld/mleproject:latest
    ports:
    - containerPort: 8000

```

c. Commandes

kubectl create -f deployment.yml

kubectl get deployment

(if needed:) kubectl delete deployments mle-deployment

```

ubuntu@ip-172-31-47-245:~/projetMLE$ kubectl create -f deployment.yml
deployment.apps/mle-deployment created
ubuntu@ip-172-31-47-245:~/projetMLE$ kubectl get deployment
NAME                READY    UP-TO-DATE    AVAILABLE    AGE
mle-deployment      0/3      3             0            5s

```

(Après quelques secondes les 3 répliques sont bien lancées.)

2) Service

a. Présentation

Un fichier service.yml est créé afin de rendre disponible les 3 répliques sur le port 8000 qui sera mappé au port 8000 défini pour le conteneur dans le Deployment.

.

b. Source

```

apiVersion: v1
kind: Service
metadata:
  name: mle-service
spec:
  type: ClusterIP
  ports:
  - port: 8000
    protocol: TCP
    targetPort: 8000
  selector:
    app: mle-api

```

c. Commandes

kubectl create -f service.yml

kubectl get deployment

(if needed:) kubectl delete service mle-service

```
ubuntu@ip-172-31-47-245:~/projetMLE$ kubectl create -f service.yml
service/mle-service created
ubuntu@ip-172-31-47-245:~/projetMLE$ kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	24s
mle-service	ClusterIP	10.106.254.117	<none>	8001/TCP	5s

3) Ingress

a. Présentation

Un fichier ingress.yml est créé afin de rendre le service disponible depuis l'extérieur.

b. Source

apiVersion: networking.k8s.io/v1

kind: Ingress

metadata:

name: mle-ingress

spec:

defaultBackend:

service:

name: mle-service

port:

number: 8000

c. Commandes

kubectl create -f ingress.yml

kubectl get ingress

(if needed:) kubectl delete ingress mle-ingress

```
ubuntu@ip-172-31-47-245:~/projetMLE$ kubectl get ingress
```

NAME	CLASS	HOSTS	ADDRESS	PORTS	AGE
mle-ingress	<none>	*	192.168.49.2	80	39s

4) Accès

54.194.241.188:8001/api/v1/namespaces/default/services/mle-service/proxy/get_performance?modelindex=1

[Kubernetes Dashboard](#)

VII. Conclusion

L'API a été déployée dans le Cloud Azure <https://disneyreviews.azurewebsites.net/docs#/> (attention le 1^{er} appel peut prendre quelques minutes car la config serveur minimale est utilisée).

J'aurais souhaité exploiter le workflow Github de publication automatique mais j'ai manqué de temps pour creuser cette piste.

J'aurais aussi souhaité stocker les username/password dans une BDD MySQL intégrée dans un autre container.

Et enfin j'aurais aimé déployer la partie kubernetes dans Azure.

Ce fût une bonne expérience de tester les outils de développement Windows (VS Code, Docker Desktop, Azure) qui ont amélioré ma productivité lors du développement de ce projet.