DATASCIENTEST – PROJETS 2

Stéphane REYNAL

(Flask / Docker / Kubernetes)

Objectifs :

1- Développement de modèles de scoring de sentiment de commentaires.

2- Construction d'une API Flask permettant d'interroger les différents modèles.

3- Containeurisation de l'API via Docker et tests CI/CD.

4- Déploiement via Kubernetes.

Livrables :

a. Le code source de l'API ainsi que les dépendances au format standard requirements.txt

b. Un Dockerfile pour l'API containeurisée,

c. Les Dockerfiles pour chaque containeur de test,

d. Dans un dossier, l'ensemble des fichiers utilisés pour créer les tests,

e. Un docker-compose.yml pour lancer l'ensemble des tests.

f. Les fichiers de déploiement Kubernetes.

Requirements

⇨ Packages: pandas, Flask, flask_httpauth, NLTK, Sklearn, Requests, `vaderSentiment`

⇨ Docker, minikube,

⇨ Fichiers: 'reviews.csv', 'credentials.csv'

1- Développement de modèles de scoring de sentiment

Model 1 : classification via régression logistique.

⇨ Fichier produit : model.py

- Objectif : attribuer un score positif/négatif aux commentaires des visiteurs d'un parc de loisirs (Disneyland) (score de -1 à +1).
- Entrainement du modèle : on scinde le jeu de données en "avis positif" (rating 4 et 5 = +1) et "avis négatifs" (rating 1 et 2 = -1). On supprime les avis neutre (rating 3). Chaque review est normalisée, préprocessée, stemmée, et vectorisée (tfidf).
- Prédict: le modèle permet d'attribuer un score de sentiment pour chaque commentaire laissé par un visiteur d'un des parcs Disneyland à travers le monde.
    - -1<score<0 => sentiment négatif
    - 0<score<1 => sentiment positif

On enregistre le model de score de sentiment et le transformeur TD-IDF, tous deux entrainés sur le dataset d'entrainement, ainsi que les résultats du modèle sur le jeu de test. Ces enregistrements se font au format .pkl.

⇨ fichiers produits: 'model1.pkl', 'vectorizer1.pkl', 'model1_test_score.pkl'

Model 2 : package vaderSentiment

2- Construction d'une API Flask permettant d'interroger les différents modèles

Cette API permet d'interroger les 2 modèles. Les utilisateurs peuvent aussi interroger l'API pour accéder aux performances de l'algorithme sur les jeux de tests.

⇨ Fichier produit : api.py

Endpoints :

- /status : retourne 1 si l'API fonctionne.
- /permissions : retourne les droits de l'utilisateur.
- /v1/test : retourne la performance de l'algorithme sur le jeu de test.
- /v1/sentiment?sentence=<sentence> : retourne le score de sentiment attribué à la sentence par le model 1.
- /v2/sentiment?sentence=<sentence> : retourne le score de sentiment attribué à la sentence par le model 2.

Enfin les utilisateurs peuvent utiliser une identification basique via le header Authentication et qui encode username:password en base 64.

L'API est accessible via :

```
curl -i -X GET http://user:password@0.0.0.0:5002/
```

Les utilisateurs qui veulent utiliser une identification basique utilisent la commande de la forme :

```
curl -H "Authorization: Basic Base64user:password" -X GET http://0.0.0.0:5002/
```

L'encodage user:password se fait comme suit :

```
echo -n user:password | base64
```

3- Containeurisation de l'API via Docker

On construit ici un container Docker pour déployer facilement l'API. Une série de tests est menée pour tester l'API contenairisée. Pour cela on crée trois fichier de test python et un fichier docker-compose.yml.

Authentication (fichier authentication_test_senti.py)

Dans ce premier test, nous vérifions que la logique d'identification fonctionne bien. Pour cela, on effectue des requêtes de type GET sur le point d'entrée /permissions. Nous savons que deux utilisateurs existent alice et bob et leurs mots de passes sont wonderland et builder. Nous allons essayer un 3e test avec un mot de passe qui ne fonctionne pas : clementine et mandarine.
Les deux premières requêtes renvoient un code d'erreur 200 alors que la troisième renvoit un code d'erreur 401 (on effectue les test avec un user:pwd encodé en base 64)

Authorization (fichier authorization_test_senti.py)

Dans ce deuxième test, nous vérifions que la logique de gestion des droits de nos utilisateurs fonctionne correctement. Nous savons que bob a accès uniquement à la v1 alors que alice a accès aux deux versions. Pour chacun des utilisateurs, nous faisons une requête sur les points d'entrée /v1/sentiment et /v2/sentiment: on fournit les authentifications username :password encodées en base 64 et l'argument « sentence » qui contient la phrase à analyser.

Content (fichier content_test_senti.py)

Dans ce dernier test, nous vérifions que l'API fonctionne comme elle doit fonctionner. Nous testons les phrases suivantes avec le compte d'alice :

- That sucks and this is expensive. It is the worst experience in my life, could not be worse
- Life is beautiful, amazing and magic

Pour chacune des versions du modèle, on récupère un score positif pour la première phrase et un score négatif pour la deuxième phrase. Le test consiste à vérifier la positivité ou négativité effective du score.

Construction des tests

Pour chacun des tests, nous créons un container séparé qui effectue ces tests. L'idée est d'avoir un container par test pour permettre de ne pas changer tout le pipeline de test dans le cas où un des composantes seulement est modifié.

Lorsqu'un test est effectué, si une variable d'environnement `LOG` vaut 1, alors une trace est imprimée dans un fichier api_test.log

4- Déploiement via Kubernetes

Enfin, on crée un fichier de déploiement ainsi qu'un Service et un Ingress avec Kubernetes pour permettre le déploiement de l'API sur au moins 3 Pods.

Installation de minikube

```
$ curl -LO https://storage.googleapis.com/minikube/releases/latest/minikube-linux-amd64
$ sudo install minikube-linux-amd64 /usr/local/bin/minikube
```

Ensuite on exécute `$ minikube start` pour lancer le cluster minikube
Pour lancer le dashboard de minikube :

```
$ minikube dashboard --url=true
```

Puis, dans un autre terminal :

```
$ Curl -LO https://storage.googleapis.com/kubernetes-release/release/v1.21.0/bin/linux/amd64/kubectl
$ chmod +x ./kubectl
$ sudo mv ./kubectl /usr/local/bin/kubectl
$ kubectl version --client
```

Pour lancer le dashboard depuis la machine locale :

```
$ kubectl proxy --address='0.0.0.0' --disable-filter=true
```

La première commande devrait retourner un lien avec l'adresse du dashboard sur la machine virtuelle. Par exemple :

```
$ http://127.0.0.1:36351/api/v1/namespaces/kubernetes-dashboard/services/http:kubernetes-dashboard:/proxy/
```

Déploiement des Pods

Nous créons un fichier yml pour déployer les pods (my-flaskapi-Deployment.yml).
Créons le service en utilisant l'interface en ligne de commande kubectl :

```
$ kubectl create -f my-flaskapi-Deployment.yml
```

Exposition de l'API

Nos trois répliques sont bien lancées mais on doit à présent la rendre disponible. Pour cela, nous pouvons créer un Service de type ClusterIP (fichier my-flaskapi-Service.yml)

```
$ kubectl apply -f my-flaskapi-Service.yml
```

L'API est disponible à l'intérieur du cluster mais on doit à présent créer un Ingress pour exposer le Service à l'extérieur du cluster.
On crée le fichier my-flaskapi-Ingress.yml puis on exécute la commande suivante :

```
$ kubectl create -f my-flaskapi -Ingress.yml
```

Nb : Par défaut, minikube ne permet pas d'utiliser les ingress. Il faut exécuter la commande `$ minikube addons enable ingress` pour permettre de les utiliser.