Summary

I. Introduction

1) Problematic

The goal of this project is to put into production a sentiment analysis model built on the Disneyland commentary dataset. https://www.kaggle.com/arushchillar/disneyland-reviews

The objective here is to deploy models already provided. Be careful, in production, the models should not be relaunched.

An API should make it possible to query the different models. Users will also be able to query the API to access the performance of the algorithm on the test sets. Finally, users must be allowed to use basic identification. We can use the following list of users / passwords:

alice: wonderland

bob: builder

clementine: mandarin

2) Presentation of the dataset

The dataset includes 42,000 reviews of 3 Disneyland parks - Paris, California and Hong Kong, posted by visitors on Trip Advisor.

Columns:

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) Development environment

The training allowed us to use different tools of MLE under Linux.

Having used this environment during the validation projects of the different parts of the training, I decided to transpose these principles into a Windows environment.

For this I used Visual Studio Code.

This IDE offers some cool features for such a project

- intelliSense

- debugging (step by step, breakpoints ...)

- Docker integration (with Docker Desktop)

- easy deployment in the Azure Cloud

- various extensions as needed

II. Modeling / Notebook

1) Notebook provided

The data set is partitioned into a training set and a test set.

We apply a CountVectorizer to the features (ie the "Review_Text" column).

The supplied notebook contains 4 distinct models:

- Model 1: LogisticRegression () applied to all training data.

- Models 2/3/4: RandomForestClassifier (n_estimators = 20, max_depth = 5) only applied on HK / California / Paris parks

2) Adaptation of the notebook to the problem

2 objects are necessary to apply the models within the API without having to train them each time:

- the CountVectorizer

and

- the model

which will both have been generated from the training data.

We therefore save these 2 objects in pickle format for each of the models.

We then obtain the following files which will be stored in the / model_pickles / subdirectory:

- count_vectorizer {i} .pkl

- model {i} .pkl

These files will make it possible to make predictions on the data entered by the API user.

In addition, in order to optimize the API, 2 other objects necessary for data preprocessing are saved in this same sub-directory:

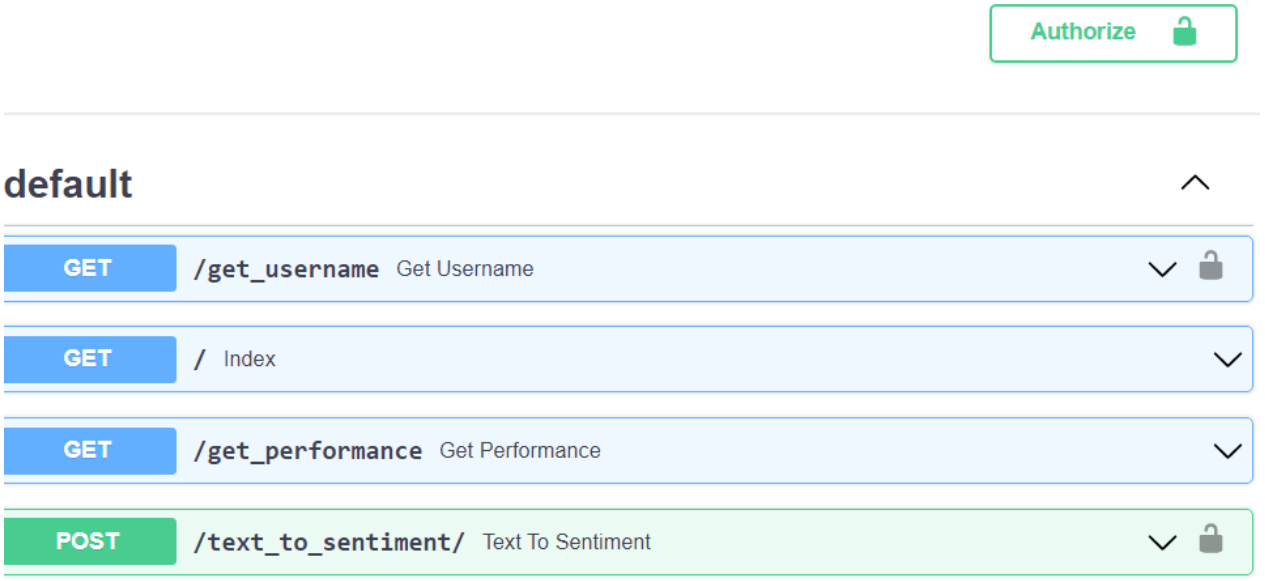
- NLTKWordTokenizer.pkl

- stopwords.pkl

III. API

damienld / MLE-Project: MLE Training Project (Datascientest) (github.com)

FastAPI was used to develop the API.



1) API Description

The API has 4 endpoints:

- `"/"`: returns {score: 1} if the API is working
- `"/ get_username"`: returns the username currently used [requires prior authentication]
- `"/ get_performance"`: returns the model score
- `"/ text_to_sentiment /"`: returns the prediction (score from 1 to 5) associated with the text entered [requires prior authentication]

2) Description of the ModelFromFiles class

The last 2 endpoints are based on the “ModelFromFiles” class defined in the model_to_load.py file

This class allows you to:

- Load a model (among the 4 existing ones) from the pre-saved files "count_vectorizer {i} .pkl" and "model {i} .pkl"

⇨ `def_load_from_pickles_files(self)`

- **Pre-process a text** in order to make it exploitable by the model
  ⇨ `defpreprocess(self,text,pkl_stopwords,pkl_tokenizer)`

- **Make a prediction** from text
  ⇨ `defpredict(self,text,pkl_stopwords,pkl_tokenizer)`

3) Using an Enum type

In order to prevent the entry of a non-existent model number, it was chosen to use an Enum type:

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

Thus, FastAPI automatically returns an HTTP 422 code if the model number does not match when one of the functions is called. This avoids having to control this value manually or to define constraints on the function parameters.

IV. API test

damienld / MLE-Project_test: TEST Project for MLE-Project (github.com)

1) Description

The api_test.py script must be called with the following parameters:

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

This script is based on a dictionary:

- **Key** : concatenation of possible call parameters
- **Value**: a tuple (expected_http_code, expected_score)

When a test is launched, we look in this dictionary for what values are expected (http code, score). If those values are correct, the test is successful.

2) Tests carried out

We test the following parameters (each parameter is separated by a "#").

`lst_sentences` is a list of sentences to test.

```
#test of the endpoint "/" => [expected HTTP code = 200, expected score = 1]
'####': (200,'1')

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

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

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

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

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

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

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

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

V. Docker

Docker images are available under Docker Hub.

Docker MLE Project

Docker MLE Project Test

1) API Docker

a. Source

```
# For more information, please refer to https://aka.ms/vscode-docker-python
FROMpython:3.7-slim-buster

EXPOSE8000

# Keeps Python from generating .pyc files in the container
NSPYTHONDONTWRITEBYTECODE=1

# Turns off buffering for easier container logging
NSPYTHONUNBUFFERED=1

# Install pip requirements
COPY requirements.txt.
RUN python-mpipinstall-rrequirements.txt

#copy / model_pickles and check
COPY model_pickles // model_pickles /
RUN ls-the/ 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-u5678--disabled-password--gecos""tap&&chown-Rtap/ app
USER tap

# 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. Orders

docker image build. -t dami1ld / mleproject: latest

docker image push dami1ld / mleproject: latest

docker image pull dami1ld / mleproject: latest

2) Test script docker

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
NS PYTHONDONTWRITEBYTECODE = 1

# Turns off buffering for easier container logging
NS 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 tap

# 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. Orders

docker image build. -t dami1ld / mleprojecttest: latest

docker image push dami1ld / mleprojecttest: latest

docker image pull dami1ld / mleprojecttest: latest

3) Docker Compose

The Docker Compose allows you to launch the 2 containers. First the API Docker then the API test Docker which displays in the console the result of each of the tests.

The 2 communicate through the "host" network

```
version:'3.4'
services:
mleproject:
picture:dami1ld / mleproject: latest
network_mode:host
mleproject_test:
depends_on:
-mleproject
picture:dami1ld / mleprojecttest: latest
network_mode:host
```

4) Test results

Under Docker Desktop, we obtain for example (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

Kubernetes files are stored in the / k8s / subdirectory

1) Deployment

a. Presentation

A deployment.yml file is created. It allows the launch of the API container which will respond to requests on theport 8000of the cluster. This container will be replicated3 times.

The container is loaded from the image stored online in 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
picture:dami1ld / mleproject: latest
ports:
-containerPort:8000
```

c. Orders

kubectl create -f deployment.yml

kubectl get deployment

(if needed :) kubect 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
```

(After a few seconds the 3 aftershocks are well launched.)

2) Service

a. Presentation

A service.yml file is created in order to make the 3 replicas available on the port 8000 which will be mapped to port 8000 defined for the container in the Deployment.

.

b. Source

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

c. Orders

kubectl create -f service.yml

kubectl get deployment

(if needed:) kubect 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. Presentation

An ingress.yml file is created in order to make the service available from outside.

b. Source

```
apiVersion:networking.k8s.io/v1
kind:Ingress
```

```
metadata:
name:mle-ingress
spec:
defaultBackend:
service:
name:mle-service
Harbor:
number:8000
```

c. Orders

kubectl create -f ingress.yml

kubectl get ingress

(if needed:) kubect 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) Access

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

Kubernetes Dashboard

VII. Conclusion

The API has been deployed in the Azure Cloudhttps://disneyreviews.azurewebsites.net/docs#/ (beware the 1st call may take a few minutes because the minimum server config is used).

I would have liked to use the Github auto-publish workflow, but I ran out of time to dig into this lead.

I would also have liked to store the username / password in a MySQL database integrated in another container.

And eventually I would have liked to deploy the kubernetes part in Azure.

It was a good experience to test the Windows development tools (VS Code, Docker Desktop, Azure) which improved my productivity during the development of this project.