Rapport Technique

Brief ChatBot Simplon

Amaury - Baptiste LB - Christelle - Paul

SOMMAIRE

1. Etat de l'art sur les chatbots

Il existe deux grandes catégories de chatbots. Les chatbots simples et les chatbots dits "intelligents" basés sur de l'IA.

Dans les chatbots simples sans IA :

- Les *rule-based chatbots.* Ces chatbots fonctionnent avec un flux en arborescence afin d'aider le visiteur avec ses demandes. Le chatbot va guider l'utilisateur avec des questions prévues d'avance pour arriver à la résolution de son problème. Tout est prédéfini de la structure aux réponses. Les développeurs contrôlent donc totalement la conversation. En général, les questions sont présentées au visiteur sous forme de boutons cliquables.

  Avantages : Ces chatbots sont faciles d'emploi pour les visiteurs. Ils sont sans IA donc ne nécessitent pas d'apprentissage. Leur implémentation est simple.
  Inconvénients : En revanche, l'interaction est peu naturelle et le chatbot ne peut pas apprendre de lui-même. Il faut l'améliorer à la main.

Dans les chatbots complexes avec IA, il y a deux sous-catégories majeures :

- Les *driven-based chatbots* sont des modèles entraînés sur base de données et lorsqu'une question est posée par utilisateur, le modèle assigne des scores à la question. Il renvoie ensuite la réponse associée à la question en base au score le plus similaire via un mécanisme de prédiction.

  Ce type de chatbot avec IA est plus adapté à des sujets fermés. Les réponses sont pré-construites donc on contrôle plutôt bien la conversation.

- Le deuxième type de chatbot avec IA est le *generative chatbot.* Ce type de chatbot étudie les mots de la question et génère une réponse avec de la *Natural Language Generation* (NLG).

  Il s'agit du type de chatbot le plus complexe à réaliser. Il donne potentiellement les rendus conversationnels les plus "naturels" mais il y a un risque d'erreurs de grammaire. De plus, la conversation peut beaucoup plus facilement dévier qu'avec le chatbot *driven-based* donc il est plus adapté à des sujets ouverts

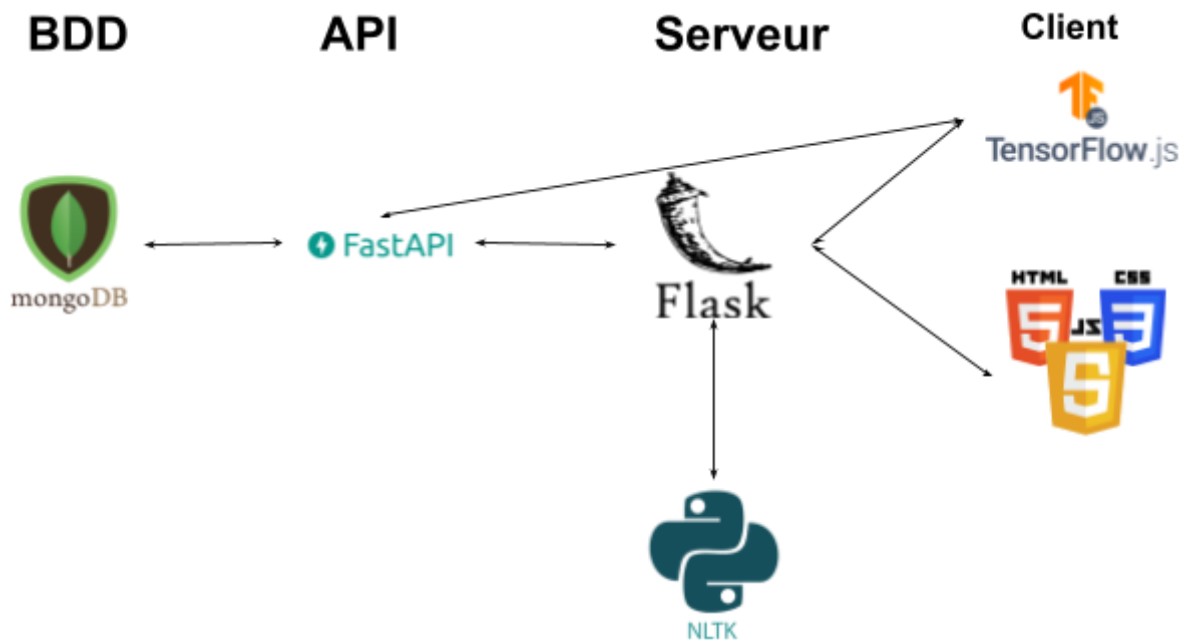
2. Choix pour la solution

Modèle retenu :

Le choix de modèle s'est porté sur Tensorflow (et son intégration de fonctionnalités de Keras), en particulier la fonction LSTM (Long Short-Term Memory) qui permet d'entraîner le modèle suivant des tentatives de prédire les prochains symboles d'une séquence. Un formatage important des données est effectué pour les simplifier, en particulier le stemming. Le but est une reconnaissance acceptable de la structure de base des lignes données en exemple, afin de permettre une estimation de nouvelles lignes envoyées par l'utilisateur.

```
model = Sequential()
model.add(Input(shape=(input_shape,)))
model.add(Embedding(vocabulary+1, 10))
model.add(Bidirectional(LSTM(40, return_sequences=True)))
model.add(Flatten())
model.add(Dense(output_length, activation="softmax"))
```

Infrastructure retenu :



3. Modèle

La base de données :
On récupère la base sur son serveur MongoDB, en un ensemble de dictionnaires transformé en DataFrame. Elle est double, constituée de parties française et anglaise. Chaque partie possède plus de 1100 strings de questions/inputs réparties en 46 catégories/tags, chaque catégorie ayant généralement une ou plusieurs réponses en output. Une catégorie donnée possède des variations de phrases autour d'un même thème, susceptibles de recevoir le même type de réponse. Les phrases sont variées suivant leur structure, l'usage de synonymes, etc...

Traitement de la question de l'utilisateur :

La ligne d'input, que ce soit lors de l'entraînement ou lors de l'usage pratique, arrive sous forme de string et reçoit un traitement similaire, en isolant ses caractéristiques :

- Une première caractéristique est la langue utilisée, anglais ou français. La database étant organisée par tags, on peut simplement les relever (toutes les tags anglaises commencent par "en_"). Par contre, une phrase soumise par l'utilisateur sera analysée pour déterminer sa langue. Après plusieurs essais, nous avons opté pour l'application de la fonction detect_language() du module TextBlob sur chaque mot de la phrase, et de retenir le résultat qui revient le plus souvent entre français et anglais.
- Une deuxième caractéristique est simplement de relever si la phrase se termine ou non par un point d'interrogation, ce qui a une grande importance sur la structure et le sens de la phrase, structure qui ne serait autrement pas conservée par l'étape suivante.
- La phase la plus importante est finalement le stemming, appliqué en anglais ou français (suivant le langage retenu) grâce à NLTK. Les mots sont ramenés à leurs seules racines, et les stopwords peu utiles à l'identification des phrases sont éliminés.

	inputs	tags	texts
85	Élève en Intelligence Artificielle	apprenant	elev intelligent artificiel 0 francais
1714	Peut-on rejoindre la formation si on n'est pas...	pas_demandeur	peut rejoindr format si demandeur emploi ? fra...
1390	Comment se passe le procédé de sélection ?	selection	comment pass proced select ? francais
1526	Qui decide la liste de compétences à avoir pou...	compétences	decid list competent avoir rentr format 0 fran...
1504	Est ce que je peux demander d'être recruteé d...	nombre_recruter	peux demand etre recrute entrepris prefer 0 fr...
...	...	...	...
1358	Choix d'un étudiant ?	choix_alternant	choix etudi ? francais
387	Profs	enseignants	prof 0 francais
1397	Comment contacter mon futur tuteur ?	tuteur	comment contact futur tuteur 0 francais
1571	Est-il possible de rejoindre la formation si o...	salarie	possibl rejoindr format si travaill ? francais
1625	Il y t-il beaucoup d'oofres de location de log...	logement	beaucoup oofr locat log pre ecol ? francais

Une fois les données originelles transformées, elles sont encodées, toujours séparément suivant la langue. Plus tard, une requête d'utilisateur renverra à l'un ou l'autre encodage.

- Le Tokenizer de Tensorflow est appliqué avec un nombre de mots de 2000, pour renvoyer la fréquence des mots présents. Un padding est appliqué pour égaliser la longueur des séquences. On se retrouve avec un vocabulaire anglais de plus de 300 mots, et un équivalent français d'environ 450.
- LabelEncoder est appliqué sur la colonne des tags pour les transformer en valeurs numériques. Elle est choisie comme variable cible.

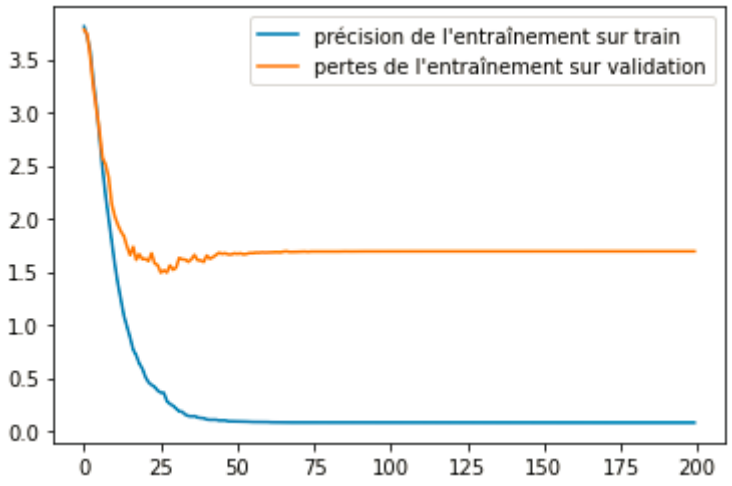
Entraînement des modèles :
Les modèles pour les lignes anglaises et françaises sont entraînés séparément mais suivant la même méthode. Les lignes sont séparées entre données d'entraînement et de validation (20% des données), même si les modèles finals les utilisent toutes pour profiter de l'ensemble de la base de données. Les séquences sont les suivantes :

- Une forme d'input standardisé est relevée (de longueur 9 en anglais et 12 en français).
- L'Embedding de Tensorflow est appliqué suivant la taille du vocabulaire, pour obtenir un format en vecteurs.
- LSTM (Long Short-Term Memory) de Tensorflow est appliqué de manière bidirectionnelle, pour renforcer le processus de feedback du modèle en testant les prédictions LSTM dans les deux sens des séquences..
- Flatten et Dense de Tensorflow sont également appliqués pour obtenir un modèle aux dimensions simplifiées mais qui conserve les connexions entre les neurones.

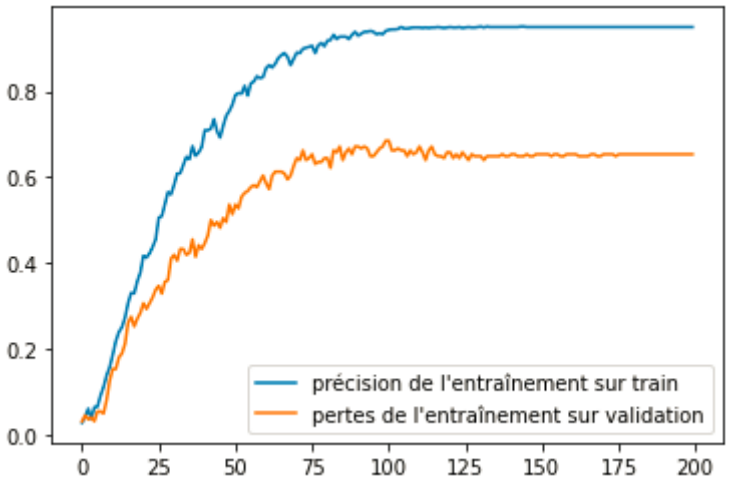
Evaluation du modèle :

Les modèles sont finalement compilés sur 200 epochs, avec comme métrique principale l'accuracy (qui arrive à environ 98% pour le français et 94% pour l'anglais), mais avec un relevé des pertes de validations, qui après entraînement tombent à moins de 1.5%.

Loss:



Accuracy :


0.8
0.6
0.4
0.2
0.0
0
25
50
75
100
125
150
175
200
précision de l'entraînement sur train
pertes de l'entraînement sur validation

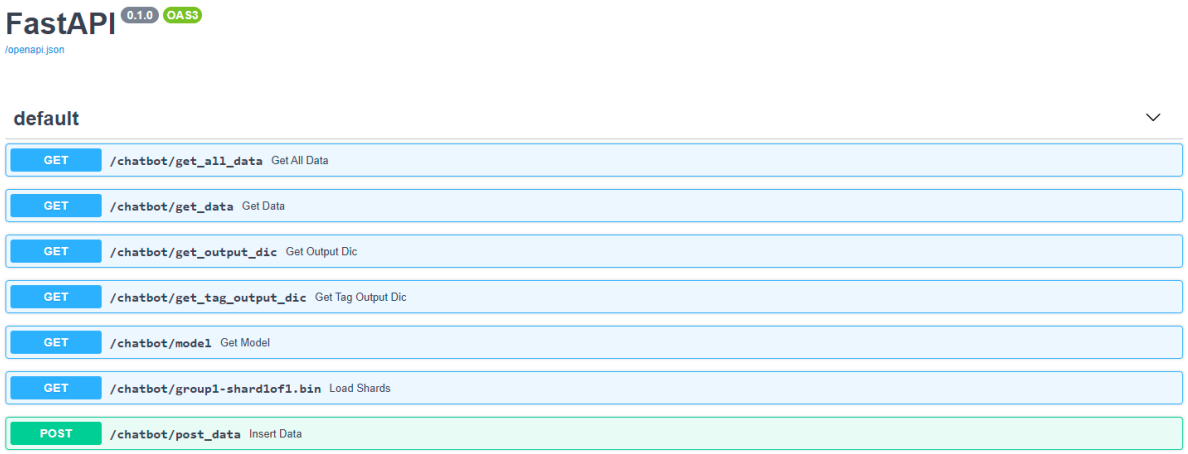
4. Back-end

BDD

Pour la base de données, nous nous sommes tournés vers du NoSQL afin d'avoir une base modulable. Nous avons donc choisi **MongoDB**. Le service Atlas nous permettant d'avoir une base en commun sur laquelle travailler. La base est composée d'un document par tag, dans lequel nous remplissons nos listes d'input et d'output. Afin de pouvoir manipuler celle-ci, le fichier `connecteur.py` nous permet un accès à la base et offre diverses méthodes (extraction, insertion, ajout, modifications, …)

API

Afin de pouvoir communiquer entre notre BDD et notre serveur Flask, nous avons mis en place une API. Pour produire celle-ci, nous nous tournons vers la librairie **FastAPI** qui présente de bonnes performances et une certaine facilité de mise en œuvre. L'API nous permettra également d'envoyer les fichiers relatifs au modèle au client côté front.



Serveur Flask

Côté serveur, afin de gérer l'affichage des diverses pages et gérer les différents flux d'informations, notre choix s'est porté sur le framework open-source **Flask**.

Notre serveur est composé de 3 routes :

- **/index :** Gère l'affichage de notre page HTML ainsi que l'initialisation de l'historique de conversations.
- **/pretreatment :** Traite la requête AJAX en provenance du client. Cette route permet donc de récupérer le message de l'utilisateur afin de pré-traiter le message avant de le renvoyer au modèle sous forme d'array. Le message est également enregistré dans l'historique de conversation.
- **/get_tag :** Traite la deuxième requête AJAX afin de récupérer la réponse du bot depuis la base de donnée correspondant au tag prédit par le modèle. Enregistre également la réponse dans l'historique.

Afin de permettre le prétraitement du message entré et l'historisation de la conversation, notre serveur Flask est lié à deux scripts.

Le premier, *`clean_message.py`* est composé de notre classe **Pretreatment**. Celle-ci regroupe les diverses méthodes afin de transformer le message entré par l'utilisateur *(cf la partie "Traitement de la question de l'utilisateur" pour plus de détails)*.

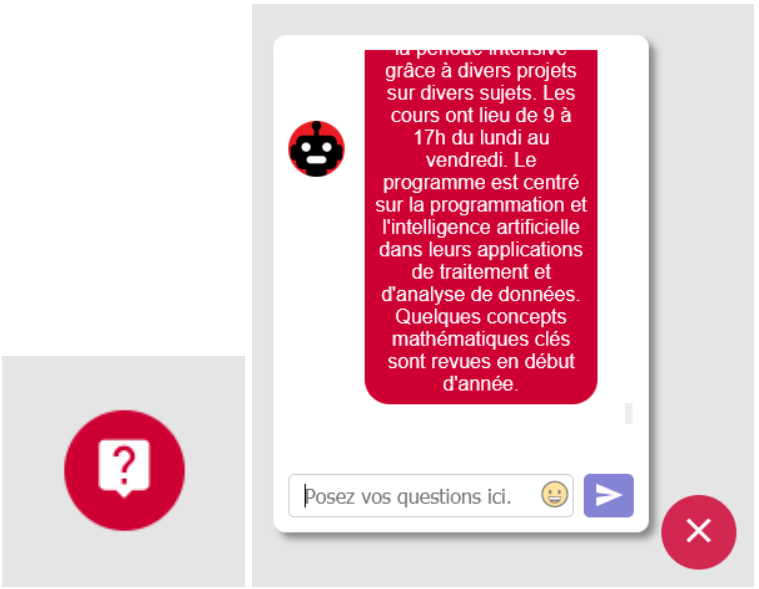
Le deuxième script *`historique.py`* nous permet de mettre en place l'historisation des messages. Notre classe est composée de 3 méthodes :

- **create_historique()** : si premier échange de l'utilisateur avec le bot créer un fichier historique sous format csv.
- **load_historique()** : charge le fichier correspondant à l'id de l'utilisateur de ses conversations précédentes.
- **save_message()** : sauvegarde un message dans le fichier historique de l'utilisateur.

5. Front-end

Pour la partie front-end de notre projet, nous avons opté pour une solution assez classique avec une page web en HTML servie par une application Flask en Python. Le HTML est mis en page par un fichier de style en CSS et dynamisé par un script en JavaScript.

Le cœur du projet est une fenêtre de chat ouvrable/refermable de type pop-up et adaptable à n'importe quelle page HTML. Visuellement, nous nous sommes inspiré du style graphique simplon caractérisé par beaucoup de rouges tendres, du blanc et des notes de bleu.



Pour des détails graphiques, quelques imports de scripts ont été réalisés via des cdn. Même s'il s'agit d'une pratique courante en développement web, il est à noter qu'en production, il serait peut-être plus pertinent d'installer directement les librairies de ces scripts sur le serveur via des technologies comme npm et ce afin d'éviter de se reposer sur des sources distantes qui peuvent potentiellement être hors d'accès ou compromises.

Nous avons mis l'accent sur l'agréabilité d'usage du chat à l'aide de feedbacks visuels plaisants et de petites features qui simplifient l'utilisation. Pour illustrer cela, on peut par exemple citer le bouton qui déploie la fenêtre *pop-up* de chat qui tressaille joliment ou bien les fonctions JS de défilement automatique du chat ou bien d'auto-focus sur la zone de saisie afin d'éviter à l'utilisateur de se fatiguer inutilement.

Explication de l'insertion de messages :

Les messages de l'utilisateur et du chatbot sont insérés visuellement dans la page à l'aide du javascript. L'idée est simple, on dispose d'une zone de chat destinée à contenir et afficher les messages. Lorsque l'utilisateur envoie un message via le formulaire de saisie ou bien que le chatbot répond, on va dynamiquement générer grâce au JS une division HTML complète qui contient le message correctement formaté avec ses balises et l'avatar sous forme de variable. Cette variable va ensuite être insérée à la fin de la zone de chat comme un nouveau message.

```
let temp = `<div class="income-msg">
            <img class="avatar animate__animated animate__fadeInLeft"
            src="${avatarBot}" alt="avatar du chatbot">
            <span class="msg animate__animated animate__zoomIn">
            ${reponse}
            </span>
            </div>`

chatArea.insertAdjacentHTML("beforeend", temp)
scrollToBottom();
```

Ce mode de fonctionnement est simple à gérer et permet de récupérer et d'insérer à la volée un message obtenu via la prédiction avec notre modèle IA et envoyé par l'application Flask via requête Ajax.

Intégration du modèle :

_Pour ce projet, le modèle sera intégré côté client grâce à la librairie **Tensorflow js**. Cette méthode nous permet d'obtenir de meilleures performances dans les temps de réponses de notre chatbot puisque le traitement s'effectuera côté client (sur la machine de l'utilisateur) et non sur le serveur.

Cette intégration s'effectue en plusieurs parties. Pour commencer, nous installons la librairie tensorflow js :

pip install tensorflowjs

Notre modèle étant au format de la librairie python Keras (***.h5***) nous devons le convertir dans un format reconnu par Tensorflow js. Pour ceci, la librairie nous met à disposition l'outil ***tensorflowjs_converter***, que nous pouvons utiliser dans la console :

tensorflowjs_converter --input_format=keras --output_format=tfjs_layers_model ./chemin_model_h5/model.h5 ./nom_dossier_enregistrement_modeljs

Après conversion nous obtenons un fichier ***model.json*** ainsi qu'un fichier de poids binaires.. Nous pouvons ensuite le charger côté front :

```
const modelURL = 'http://localhost:5000/chatbot/model';
const model = await tf.loadLayersModel(modelURL);
console.log('Modèle Chargé')
```

Nous chargeons le modèle grâce à la fonction ***loadLayersModel().*** Celle-ci prend une url vers le model comme argument. Nous lui donnons l'url de l'api qui pointe vers notre ***modèle*** et le fichier de poids.
loadLayersModel() renvoie un objet Promise. Cela signifie que cette fonction promet de renvoyer le modèle à un moment donné dans le futur. Le mot clé **await** met en pause l'exécution de cette fonction d'encapsulation jusqu'à ce que la promesse soit résolue et que le modèle soit chargé. C'est pourquoi nous utilisons une fonction asynchrone.

Le modèle chargé, nous pouvons finalement effectuer nos prédictions sur le message prétraité :

```
let prediction = model.predict(reponse);
let label = prediction.argMax(axis = 1).dataSync()[0];
console.log('Prédiction :', label);

let probabilities = tf.softmax(prediction).dataSync()[label];
console.log('Proba Label :', probabilities);
```

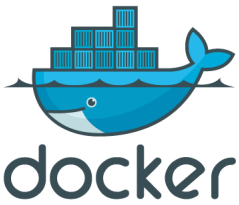
Ingestion du message prétraité par le modèle :

Notre message prétraité est de type array. Cependant, notre modèle tensorflow js attend un tenseur. Nous transformons alors notre message avant d'être ingéré dans le modèle :

```
reponse = tf.tensor(reponse);
```

6. Déploiement

Afin que notre chatbot puisse être déployé sur un serveur de production (ex: EC2 sur AWS), nous devons conteneurisé notre application. Nous réaliserons cette tâche grâce à **Docker**.



Docker est un outil qui peut empaqueter une application et ses dépendances dans un conteneur isolé, qui pourra être exécuté sur n'importe quel serveur.

Nous commençons par créer un **Dockerfile** pour l'API ainsi que pour le serveur Flask. Le Dockerfile nous permet de créer une image propre à notre application permettant sa construction dans un container au moment voulu.

```
FROM tiangolo/uvicorn-gunicorn-fastapi:python3.7

WORKDIR ./
COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY . .
```

Dockerfile FastAPI

```
FROM python:3.8-slim-buster

WORKDIR .

COPY requirements.txt requirements.txt
RUN pip3 install -r requirements.txt

COPY . .

RUN python -m nltk.downloader stopwords
```

Dockerfile Flask serveur

Afin d'associer ces deux images dans un seul container nous utiliserons un fichier ***Docker-compose.yml.***
Compose est un outil permettant de définir et d'exécuter des **applications** Docker **multi-conteneurs**. Avec Compose, vous utilisez un fichier YAML pour configurer les services de votre application. Ensuite, avec une seule commande, vous créez et démarrez tous les services à partir de votre configuration.

```yaml
api:
    build:
        context: ./API
    container_name: api
    ports:
        - 5000:5000
    volumes:
        - .:/API
    command: uvicorn api:api --reload --workers 1 --host 0.0.0.0 --port 5000
    # depends_on:
    #     - mongodb
    environment:
        - ENVIRONMENT=dev
        - TESTING=0

flask_app:
    build:
        context: ./app
    container_name: flask_app
    ports:
        - 5001:5001
    volumes:
        - .:/app
    environment:
        FLASK_ENV: development
    command: flask run --host 0.0.0.0 --port 5001
```

Pour que l'ensemble fonctionne correctement, nous définissons quelques paramètres :

- *build* et *context* qui nous permettent de cibler les Dockerfile
- nous définissons les *ports* interne/externe au conteneur sur lesquels nous pouvons interagir avec les conteneurs
- définir des *volumes* permet la persistance de nos fichiers locaux. Nos répertoires sont ainsi copié dans le conteneur lors de sa création
- *command* permet d'exécuter des commandes au lancement du container. Ici, nous lançons donc l'API et le serveur Flask.

7. Problèmes non résolus et axes d'améliorations

- La base de données peut toujours demander plus de données et de formulations différentes, les résultats restent encore limités ou présentent des réponses hors contexte lors d'une demande non connue du modèle.

- Rendre possible un ré-entraînement du modèle après labellisation de l'historique de conversation pour nourrir le modèle avec de nouvelles demandes.

- La vitesse de réponse du bot semble être difficile à prévoir suivant les machines.

- Transférer la base mongoDB Atlas une fois relativement complète vers une base conteneurisée sur Docker afin d'améliorer le temps de réponse.

- Déployer l'application sur un serveur de production type AWS EC2 afin de rendre l'application accessible au grand public.