Domestic Intelligent Vocal Assistant

Projet chef-d'oeuvre de Nicolas CHAPON

DiVA

"Science sans conscience n'est que ruine de l'âme"

Rabelais - Pantagruel

Microsoft SIMPLON.CO

SOMMAIRE

Introduction

Le marché des assistants domestiques vocaux est en plein essor. Ainsi, les mastodontes du web proposent tous leur enceinte connectée au web qui permet à l'utilisateur d'agir sur de la domotique, de manipuler un agenda en ligne ou encore d'accéder à ses comptes de streaming.

Si on ne peut nier le confort qu'apportent ces technologies, on peut légitimement se poser la question de l'utilisation des données personnelles de leurs utilisateurs par ces grands groupes. Sont-elles juste stockées ? Ou bien sont-elles vendues à des fins commerciales, ou encore utilisées par ces mêmes groupes pour leur propre compte - création de nouveaux modèles d'intelligence artificielle, amélioration de modèles existants - afin d'enrichir leurs offres ?

Pourrait-on alors envisager le développement d'un assistant domestique performant qui fonctionnerait de façon autonome sur un réseau privé ?

Problématique

Il existe un phénomène de défiance à l'encontre des géants du numérique dans une part de plus en plus importante de la population. Pour autant, ces personnes ne souhaitent pas se couper d'avancées technologiques qui leur apportent plus de confort ou plus d'amusement.

Je propose donc de m'employer à développer un assistant personnel domestique autonome et indépendant. Je souhaite que son code soit open source afin que chacun puisse savoir comment il fonctionne. Pour un déploiement aisé sur tout type de plateforme, je produirai une application encapsulée dans un Docker.

Dans un premier temps, on peut imaginer un outil très simple, pour gérer des listes de courses puis enrichir cet outil au fur et à mesure.

Analyse des besoins

En tant qu'utilisateur d'assistant vocal, je n'ai pas envie que les géants du net puissent jouer avec mes données personnelles.

En tant qu'utilisateur, je ne veux pas qu'un appareil, dont je n'ai pas le contrôle complet, écoute en permanence ce que je dis.

En tant qu'utilisateur, je veux garder mes données personnelles au sein de mon réseau privé.

En tant *qu'utilisateur,* je veux pouvoir accéder à une liste pour gérer mes courses.

En tant *qu'utilisateur*, je veux pouvoir ajouter un item à ma liste.

En tant *qu'utilisateur*, je veux pouvoir supprimer un item dont je n'ai plus besoin de ma liste.

En tant *qu'utilisateur*, je veux pouvoir lister les items présents dans ma liste pour être sûr de ne rien oublier.

En tant *qu'utilisateur*, je veux pouvoir recevoir ma liste sur un support mobile afin de l'avoir avec moi pour aller faire les courses.

Gestion de projet

Méthodologie de projet

J'ai choisi de m'appuyer en partie sur les principes *Agile* du développement logiciel pour mener à bien ce projet.

J'ai donc découpé mon projet en grandes *Feature*, elles-mêmes subdivisées en *User Stories* puis en *Tasks*. Ce découpage est basé sur l'utilisation de l'outil *Microsoft Azure DevOps* dont je détaillerai les principes de fonctionnement un peu plus loin dans ce rapport. Pour réaliser ce découpage, j'ai tenté d'avoir une vision de la plus grande échelle vers la plus petite, du plus global au plus particulier en détaillant chaque pièce principale en petits items.

Ceci m'a permis d'avoir une vision plus claire des développements à réaliser, tout en essayant d'estimer le niveau de difficulté de chaque tâche. Il est en effet plus aisé d'évaluer le temps que prendra une tâche à réaliser quand elle est moins importante et recoupe moins de fonctionnalités. De plus cela m'a permis d'établir un planning qui me permettait de mesurer l'avancement du projet au jour le jour et de pouvoir rectifier rapidement les éventuelles dérives tout en laissant de la place à l'imprévu.

J'ai créé un projet *Azure DevOps* dans lequel j'ai démarré une *backlog*. Je me suis organisé en sprints d'une semaine.

Outils

Azure DevOps

J'ai choisi d'utiliser *Microsoft Azure DevOps* en premier lieu parce que je l'utilise en entreprise. Je n'ai donc pas eu à apprendre à me servir de cet outil pour ce projet. J'ai ainsi eu l'occasion d'explorer des fonctionnalités que je ne manipule pas habituellement.

Ensuite ce qui me séduit dans cet outil, c'est la centralisation de différents aspects de la gestion de projet au même endroit.
Lors de la réalisation de ce projet, j'ai utilisé les fonctionnalités de gestion de projet - *Boards* - et de versionnage du code - *Repos*.

Boards

J'ai utilisé principalement les partie *Backlogs* et *Sprints* de cette partie d'*Azure DevOps*. Les items qualifiés dans la *backlog* se découpent facilement en tâches rattachées les unes aux autres par des parents communs, ce qui permet facilement d'aller du global au particulier et d'avoir une vision claire des développements à réaliser.

L'onglet *Sprints* permet d'organiser le travail dans le temps. On peut donc assigner les cartes décrites dans la *Backlog* à des sprints. De nombreux graphiques sont accessibles et mesurent - au choix - les tendances de *burn* ou de *up* des items.

Pour que cet outil soit optimisé, il demande un gros travail de préparation en amont des développements, notamment dans l'évaluation de la durée des tâches ou de leur difficulté (il y a évidemment un lien de cause à effet !). Étant seul pour ce projet, dans sa conception et sa réalisation, j'ai fait le choix de me cantonner à une utilisation assez basique de l'outil.

Enfin, le fait que la gestion de projet et les repos soient centralisés au même endroit permet de créer des liens entre les tâches de la *backlog* et les différentes versions du code.

Repos

J'ai choisi d'utiliser la partie *Repos* d'*Azure DevOps* pour bénéficier des avantages de la centralisation des outils.

En effet, on peut facilement relier une tâche de développement dans la *backlog* à une branche de développement. Je détaillerai plus loin mes choix en matière de versionnage du code, mais on peut noter ici l'intérêt de cette fonctionnalité : ceci m'a permis de détecter plus facilement d'où peut provenir un bug - quand est-il apparu, pendant le développement de quelle fonctionnalité - et ainsi de pouvoir le corriger plus aisément.

Enfin on peut relier au repo des *Pipelines* automatisées qui peuvent s'exécuter suite à différentes actions sur le repo, comme par exemple, un build automatique de l'application, des tests et enfin une release en production suite à un merge d'une fonctionnalité dans la branche principale.

Git

Pour versionner mon code, j'ai utilisé *Git*, un outil open-source massivement utilisé par les entreprises et les développeurs. Il est aujourd'hui essentiel d'utiliser de tels outils pour pouvoir manipuler le code source en ayant la possibilité de revenir en arrière. De plus, l'utilisation des branches de *git* permet de travailler parallèlement sur différentes parties du code sans que les autres n'en soient affectées.

J'ai choisi d'appliquer les principes du workflow *git flow* qui préconise l'utilisation de branches éphémères pour chaque feature à développer, d'une branche de développement et d'une branche de production.
La mise en place de ce workflow est aisée au sein d'*Azure Devops*, comme je l'ai dit précédemment, puisqu'on peut créer les branches de features du repos directement depuis les cartes de la *backlog*.

Conda

L'application Diva est développée en *Python*, j'ai donc utilisé l'outil *Conda* qui permet de gérer différentes versions de *Python* et des librairies utilisées sans conflits d'un projet à

l’autre. Les environnements sont séparés et chaque projet peut s’appuyer sur des dépendances différentes.

Étapes principales

La première grande étape de ce projet en a été la définition : que doit faire l’application Diva au minimum ? De ce questionnement est venu le découpage de mes histoires utilisateurs.

Ensuite, j’ai choisi quatre grands axes pour développer l’application :

- Les tâches liées à la partie chatbot / interface
- Les tâches liées à la base de données
- Les tâches liées à l’intelligence artificielle
- Les tâches transverses

Pour la partie intelligence artificielle de l’application, j’ai d’abord eu une phase de recherche et de documentation avant de démarrer les développements.

Intelligence artificielle

L'application DiVA est un assistant virtuel. Elle s'appuie sur les techniques de traitement du langage naturel - *Natural Language Processing* dans la langue de Shakespear - pour fonctionner.
Dans ce type d'applications, on distingue deux phases.

La classification d'intentions

Il s'agit dans un premier temps de savoir quelle est l'intention de l'utilisateur. Le programme doit donc réaliser une classification entre plusieurs intentions. Dans le cas d'une liste, on peut vouloir ajouter ou supprimer quelque chose par exemple. Il s'agit donc d'un problème de classification.

L'extraction d'entités

Dans un deuxième temps, le programme doit comprendre quel est l'objet à utiliser. Il s'agit là encore d'une classification. Cela permet de reconnaître des entités et d'en extraire les données.

"Ajoute du beurre dans la liste de courses"

Ici l'intention est *AddItem* et on peut extraire deux entités, par exemple, *ListItem* avec la valeur *beurre* et *ListName* avec la valeur *courses*. On pourrait représenter cette phrase - utterance - en base de données sous la forme suivante :

```
{
  "utterance": "ajoute du beurre à la liste de courses",
  "intent": "AddItem",
  "entities": [
    {
      "entity": "ListItem",
      "value": "beurre"
    },
    {
      "entity": "ListName",
      "value": "courses"
    }
  ]
}
```

Dans le cadre de ce projet, j'ai choisi de me concentrer en premier lieu sur la classification des intentions. Il existe aujourd'hui sur le marché une quantité d'outils permettant de réaliser ces tâches.

Modélisation

J'ai choisi de me concentrer pour ce projet sur la classification d'intentions. J'ai, pour réaliser cette classification, choisi d'utiliser un modèle d'apprentissage profond. En effet, ce type de modèle est privilégié dans le domaine du traitement du langage naturel.

J'ai utilisé la librairie *Keras* et un modèle *LSTM* - Long Short-Term Memory - pour ce projet.

L'architecture du modèle est assez simple.

```
model = Sequential()
model.add(Embedding(
   input_dim=MAX_NB_WORDS,
   output_dim=EMBEDDING_DIM,
   input_length=MAX_SEQUENCE_LENGTH
))
model.add(LSTM(units=100))
model.add(Dense(4, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer=optimizer,
              metrics=['accuracy'])
```

Le modèle est de type séquentiel.
Il est composé d'une couche d'embedding pour réduire la dimension des vecteurs représentants les mots. Un avantage de l'utilisation de l'embedding est qu'il va créer des vecteurs proches pour les mots dont le sens est proche et permettre ainsi au modèle de mieux généraliser à des mots qu'il ne connaît pas mais dont la sémantique est proche des données d'entraînement.
Il y a ensuite la couche LSTM composée de cent cellules.
La dernière couche est la couche de sortie qui permet d'obtenir des probabilités sur chacune des quatre classes à prédire.

Pour une classification multi-classes, j'ai choisi comme fonction de loss categorical cross entropy.

Dataset

J'ai dans un premier temps écrit un premier jeu de données à la main. Je l'ai écrit sous forme d'un csv qui comporte des phrases d'exemple et leur intention associée.
Pour enrichir ce dataset, j'ai ensuite proposé à mon entourage un formulaire pour avoir des exemples supplémentaires avec *Google Forms*. Les réponses de ce formulaire peuvent être extraites sous un format *Google Sheet*. J'ai reporté à la main ces données dans mon fichier csv initial.
Pour améliorer le processus, je pourrais écrire un script python qui permettrait d'automatiser cette tâche.

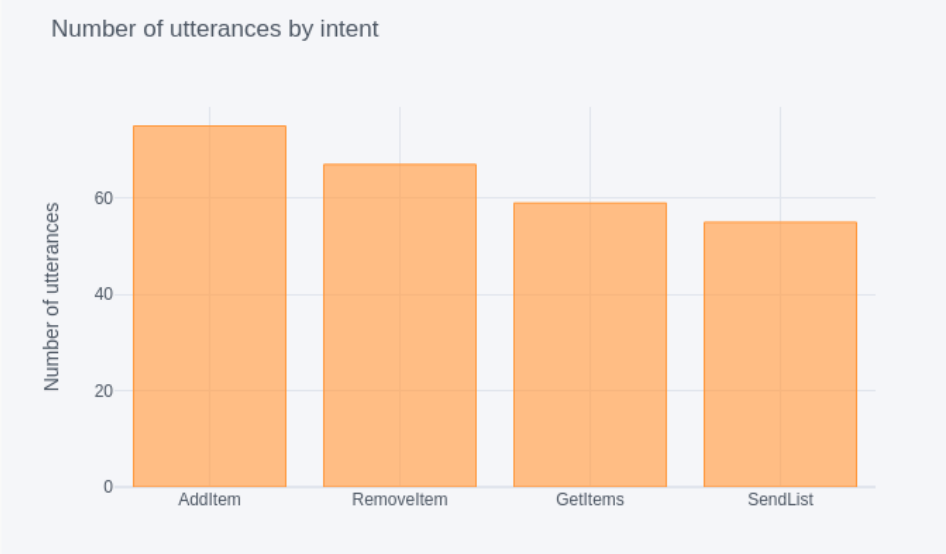
Une problématique importante quand on traite le langage naturel est que chaque utilisateur possède sa propre façon d'exprimer une intention. Aussi, il est essentiel de mettre en place un mécanisme d'amélioration continue du modèle prédictif au fur et à mesure de son utilisation pour enrichir le dataset et ainsi affiner les prédictions du modèle .

Description du jeu de données

Le jeu de données initial est composé de phrases d'exemple et de leur intention associée. Il y a quatre intentions différentes :

- **AddItem** -> Ajouter un élément à une liste
- **RemoveItem** -> Retirer un élément d'une liste
- **GetItems** -> Lire tous les éléments présents dans une liste
- **SendList** -> Envoyer la liste par email

Les quatre classes sont plutôt assez équilibrées.



Préparation des données

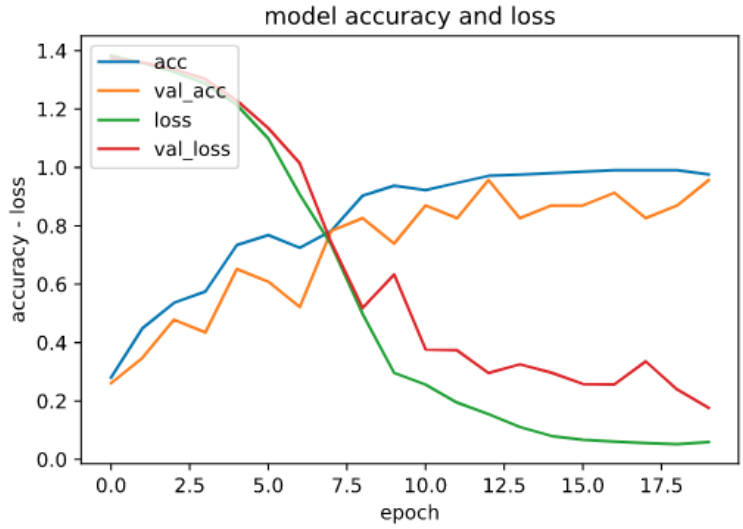
Méthodes

Métriques

Les métriques retenues pour l'évaluation du modèle sont l'accuracy et la loss. L'accuracy est représentative de l'importance d'une erreur alors que la loss, elle représente le nombre d'erreurs. On va donc chercher à maximiser l'accuracy, essayer d'être le plus proche possible de la vérité et à minimiser la loss, à se tromper le moins de fois possible.

Pour évaluer ces métriques, j'ai dans un premier temps séparé le dataset en deux. Une partie d'entraînement qui comporte quatre-vingt-dix pourcents du dataset initial appelée *trainset* et une partie qui contient les dix pourcents restants appelée *testset*. De plus, lors de l'entraînement du modèle, j'ai choisi d'utiliser dix pourcents du *trainset* pour créer un set de validation *validationset*.

Ceci permet à chaque époque d'obtenir des mesures de l'accuracy et de la loss sur le *trainset* et le *validationset*.



Ensuite, j'ai utilisé la méthode *test_on_batch* du modèle sur le *testset* pour pouvoir évaluer les performances du modèle sur des données qu'il n'a pas vu pendant la phase d'entraînement.

```
print(model.metrics_names)
print(model.test_on_batch(X_test, y_test))
['loss', 'accuracy']
[0.35126569867134094, 0.8775510191917419]
```

Recherche

Durant la phase de recherche, j'ai choisi d'utiliser la librairie open source *MlFlow* qui m'a permis de rapidement effectuer plusieurs itérations en variant les paramètres du modèle. En effet, cette librairie permet d'organiser l'entraînement du modèle sous forme de *Runs* avec la possibilité de consigner des valeurs (variables, résultats) ainsi que différents graphiques. La librairie met également à disposition une interface web qui permet de visualiser les résultats des runs et de les comparer entre eux.

J'ai principalement orienté les recherches autour de la taille du batch et du nombre d'époques.

Je joins en annexes 3 de ce rapport quelques captures d'écran de l'interface *MlFlow*.

Production

Pour la mise en production du modèle, j'ai choisi d'utiliser les pipelines *ScikitLearn* afin de pouvoir facilement passer le modèle de la phase d'entraînement à la phase de prédiction.

Cette facilité à pouvoir entraîner un modèle et à le remettre rapidement en production est particulièrement essentielle dans une optique d'amélioration continue du modèle prédictif.

J'ai écrit une API pour le modèle dans le but de l'exposer pour l'inférence d'une part, mais également dans l'optique de pouvoir également revoir les données prédites durant l'exploitation du modèle en production. Ceci permettra de pouvoir corriger les prédictions erronées du modèle et de rajouter ces nouvelles entrées labellisées aux données d'entraînement. Petit à petit, le jeu de données va donc s'enrichir directement avec les phrases des utilisateurs et cela rendra le modèle de plus en plus robuste.

Je détaillerai dans la partie Backend de ce rapport la mise en API du modèle.

Résultat

Compte tenu de la taille du dataset initial, les résultats sont plutôt satisfaisants. On peut de plus imaginer qu'avec l'utilisation du chatbot par différents utilisateurs et la revue des conversations passées, les données d'entraînement grossiront au fur et à mesure pour apporter plus de robustesse au modèle.

La faible taille du dataset initial a probablement été compensée par le fait que les exemples pour une intention donnée ne varient que très peu. Il n'y a pas tellement de façons d'exprimer la volonté d'ajouter un item à une liste.

Application

Base de données

J'ai choisi d'utiliser *MongoDb* qui est un moteur de base de données NoSql. J'ai fait ce choix parce que dans le contexte d'un chatbot, la problématique va être d'extraire des informations d'un texte. Ainsi, des phrases d'entrée émises par un utilisateur, on pourra classifier des intentions, mais aussi parfois des entités.

Les bases NoSql apportent plus de flexibilité que les bases SQL. La contrepartie étant qu'il faut être d'autant plus vigilant sur la validation des données qui entrent et sortent de la base. J'ai mis en place des schémas pour chaque collection afin d'avoir une validation des données que je détaillerai dans la partie Backend de ce rapport.

Collections

Il y a quatre collections dans la base de données. Chaque document dans une collection possède à minima un champ identifiant unique *_id*.

Conversation

Cette collection regroupe les conversations passées entre un utilisateur et le bot. Chaque document contient deux champs : *timestamp* qui permet de savoir à quelle date a eu lieu la conversation et *prompts* qui est une liste d'objets qui *prompt* dont chacun contient les données échangées. On peut ainsi savoir quelle phrase a été envoyée, si elle a été envoyée par un utilisateur ou le bot et, dans le cas d'une entrée utilisateur, l'intention reconnue par le modèle prédictif.

Cette collection permet de garder une trace des échanges et de pouvoir revoir ensuite les conversations passées et éventuellement de pouvoir corriger les erreurs du modèle pour enrichir les données d'apprentissage. Cela permet également de voir sur quel type de données le modèle fait des erreurs.

List

La collection *list* permet de stocker les listes. Le schéma d'un document *list* est composé d'un champ *name*, d'un champ liste d'*items* et de champs *created_at* et *updated_at*. Les *items* sont des objets qui contiennent un nom, une quantité optionnelle et une unité de mesure optionnelle.

User

La collection *user* sert à enregistrer un utilisateur. Les documents contiennent des champs *firstname*, *lastname*, *email* (unique dans une collection), *password* (hashé pour la sécurité) et deux champs *created_at* et *updated_at*.

Training Data

La collection *training_data* contient l'ensemble des données d'entraînement pour le modèle prédictif. Un document est constitué d'une phrase *utterance*, de l'intention associée *intent* et de deux champs *created_at* et *updated_at*.

Backend

Api base de données

J'ai développé une API - Application Programming Interface - pour accéder à la base de données afin de séparer la manipulation des données du chatbot en lui-même. J'ai utilisé *FastAPI* en python pour la réaliser. J'ai également utilisé un *ODM* - Object Document Mapper - *ODMantic* pour modéliser les données de la base dans l'application. *FastAPI* vient nativement avec un *ORM* - Object Relation Mapper - *Pydantic*, mais il y avait des soucis pour manipuler les ObjectId qui sont utilisés avec MongoDB.

De plus *FastAPI* incite fortement à l'utilisation des types python qui sont même indispensables pour la modélisation des données et en définir les schémas. Enfin, la documentation est agréable et ergonomique ! Elle est accompagnée de tutoriels qui permettent une prise en main rapide de l'outil.

Modèles

J'ai créé une classe par modèle de données dont j'avais besoin. Ces classes permettent de manipuler les données sous forme d'objets python. Ce sont les propriétés de ces classes qui définissent le schéma des documents. L'utilisation des types pythons permet au sein des classes de valider le format des données qui sont manipulées. Le nom que l'on donne à la classe est le nom de la collection qu'elle représente.

```python
# Python import
from typing import List, Optional
from datetime import datetime

# Dependencies import
from odmantic import Model

# Local Import
from .item import Item


class TodoList(Model):
   name: str
   items: List[Item] = []
   created_at: Optional[datetime]
   updated_at: Optional[datetime] = datetime.now()

   class Config:
       collection = 'list'
```

Dans cet exemple qui représente une liste on repère les champs obligatoires, ils sont seulement typés. On peut assigner des valeurs par défaut comme pour le champ *items* qui est une liste d'*Item* (un autre modèle de données) et qui est une liste vide à la création d'un document si aucun item n'y est associé. J'ai choisi de mettre les champs *created_at* et *updated_at* optionnels pour qu'ils ne soient gérés que depuis l'API elle-même. On assigne une valeur au champ *created_at* à l'insertion d'un document via la méthode de route qui lui est associée et le champ *updated_at* est mis à jour à chaque modification de manière automatique grâce à la valeur par défaut dans sa définition.

On note dans cet exemple que je ne pouvais pas appeler ma classe *List* car c'est un mot clé du langage python. La classe imbriquée *Config* permet de donner le nom que l'on souhaite à la collection.

Router

J'ai utilisé un router pour séparer les manipulations afférentes à chaque modèle de données. Ainsi, le fichier principal reste concis et chaque modèle possède sa propre route.

```python
# Dependencies imports
import uvicorn
from fastapi import FastAPI

# Local modules imports
from router import users, lists, items, conversations, data


app = FastAPI()

app.include_router(lists.router)
app.include_router(items.router)
app.include_router(users.router)
app.include_router(conversations.router)
app.include_router(data.router)


if __name__ == "__main__":
   uvicorn.run(app, host="0.0.0.0", port=8000)
```

CRUD

Sur chacune des routes j'ai appliqué les principes de *CRUD* - Create Read Update Delete - qui permettent la création, la lecture, la modification et la suppression d'une ressource. Ces actions s'appuient sur les verbes HTTP POST, GET, PUT/PATCH et DELETE. J'ai choisi pour des raisons de simplification d'associer PUT et PATCH à la même méthode.

Encore une fois, l'utilisation des types pythons permettent une validation des données à l'entrée et à la sortie de chacune des méthodes qui implémentent ces verbes.

Je joins en annexe 1 de ce document un exemple de router.

Api de prédiction

l'API du modèle, elle aussi développée avec *FastAPI*, comporte deux routes :

Une route *train* en GET et qui permet de lancer l'entraînement d'un nouveau modèle prédictif.
Je joins en annexe 2.1 de ce document le code d'entraînement du modèle.

Une route *predict* en POST qui permet d'obtenir une classification de l'intention de l'utilisateur suite à une phrase entrée par lui.
Je joins en annexe 2.2 de ce document le code qui permet d'obtenir une prédiction.

Chatbot

Pour la partie chatbot, j'ai développé une interface en console. Le principe est simple : l'utilisateur entre une phrase, le chatbot appelle l'API de prédiction sur la route */predict* en envoyant dans le corps de la requête la phrase entrée par l'utilisateur. Le modèle envoie sa prédiction, et à partir de cette prédiction, le chatbot pourra effectuer différentes tâches. À ce stade du projet, l'extraction d'intentions n'étant pas encore en place, il paraît peu utile d'ajouter des éléments dans une liste. Le chatbot se contente donc de dire quelle intention a été détectée par le modèle.

Une application asynchrone

Le chatbot effectuant des appels à des APIs externes, il est essentiel qu'il soit développé avec des méthodes asynchrones.

J'ai utilisé la librairie *requests* pour effectuer les appels aux APIs externes et j'ai utilisé la méthode *.run()* de la librairie *asyncio* pour pouvoir rendre l'application asynchrone.

```python
# Python's imports
import asyncio

# Local Python imports
from diva import Diva


async def main():
    diva = Diva()
    conversing = True

    print("Bonjour, je suis DiVa votre assistant de gestion de listes :)")
    message = input("\nQue puis-je faire pour vous ? \n- ")

    while conversing is True:
        conversing = await diva.on_incomming_message(message)

        if conversing:
            message = input("\nQue puis-je faire d'autre ? \n- ")
```

```
asyncio.run(main())
```

main.py de l'application chatbot

La méthode *on_incomming_message* du code ci-dessus fait partie de la classe *Diva* et encapsule la logique du chatbot. C'est la seule méthode "publique" de la classe. Si je mets des guillemets autours du mot publique, c'est qu'en python, la notion de publique ou privée n'existe pas, mais il est convenu que les méthodes de classe débutant par un underscore doivent être considérées comme étant privées.

C'est la méthode *_process_user_input* qui fait l'appel à l'API du modèle.

```
async def _process_user_input(self, utterance):
        if utterance == "merci":
            return None

        data = json.dumps({"utterance": utterance})
        r = await requests.post(self.PREDICTION_URL, data)
        return r.json()
```

Qualité

Qualité du code

Pour tenter d'écrire un code de qualité, j'essaye de m'appuyer sur les bonnes pratiques établies par la communauté python à travers les PEP (Python Enhancement Proposal). Pour ce faire, j'utilise le linter *Flake8* qui est exigeant et se base sur les prescriptions PEP8.

J'ai également essayé d'écrire du code *DRY* - Don't repeat yourself - en utilisant des fonctions et la factorisation quand cela était nécessaire.

J'ai enfin commencé à utiliser les types python. Ceux-ci sont utiles pour plusieurs raisons. Ils permettent rapidement de savoir ce que représentent les variables ou ce que doit être le retour d'une fonction, mais ils aident aussi durant la phase de développement puisqu'ils permettent à l'éditeur de code d'avoir plus de suggestion. C'est un double bénéfice qui aide à la clarté du code et à la productivité. Cela permet aussi une première forme de validation des données au sein de l'application.

Tests

J'ai utilisé l'outil *Postman* pour développer les différentes APIs. J'ai créé des collections pour pouvoir essayer chaque route. L'avantage de cet outil est qu'il permet de pouvoir réaliser des appels sans avoir encore écrit les méthodes qui vont consommer les APIs. Ainsi, on peut sécuriser les APIs avant de débuter les développements de leurs consommateurs et ne pas avoir à gérer des erreurs à trop d'endroits différents.

Améliorations et futur de l'application

J'ai trouvé ce projet très riche sur beaucoup d'aspects et je pense que je continuerai à travailler sur mon application DiVA pour pouvoir apporter des améliorations sur plusieurs points.

Comme je l'ai énoncé dans ce rapport, le code est hébergé sur un repo privé au sein d'*Azure DevOps*. Je mettrai le code en open source sur un repo Github.

Générales

Je voudrais utiliser *Docker* pour pouvoir être capable de déployer facilement l'application sur différentes plateformes. Je pense notamment au *Raspberry Pi* qui serait un très bon support pour ce type d'usage.

Machine learning

Le modèle a manqué de données d'entraînement, et les résultats sont perfectibles. J'ai exploré les pipelines Scikit Learn pour Keras. Je pense que pouvoir utiliser des méthodes d'optimisation des hyperparamètres telles que *GridSearch* seraient bénéfiques.

Sur le principe des *DevOps* on pourrait mettre en place un entraînement automatisé du modèle prédictif avec les nouvelles données récoltées lors de l'utilisation de l'application. Si les résultats sont meilleurs, le nouveau modèle est mis en production.

Il faudrait, pour que l'application remplisse pleinement son rôle, mettre en place l'extraction d'intentions.

Base de données

Il faudrait mettre en place un backup automatisé de la base de données à l'aide, par exemple, des tâches CRON. Chaque semaine, la base de données est sauvegardée sur un support externe à l'application.

Chatbot

Le chatbot est pourvu d'une interface en console. Je souhaite pouvoir interagir avec le bot avec la voix.

Avec l'appui d'un *NLP* plus robuste, on pourra aussi gérer plusieurs listes différentes.

Front End

Je n'ai pas mis en place d'interface graphique, mais je souhaite le faire pour pouvoir avoir un œil sur les conversations passées du chatbot. Ceci permettra de voir sur quelles phrases le modèle a fait des erreurs et de les corriger pour fournir de nouvelles données d'entraînement labellisées.

Tests

Je pourrais mettre en place des tests unitaires pour les développements futurs. Les librairies comme *Pytest* permettent de garantir le bon fonctionnement du code et de se prémunir des effets de bord indésirables. Ils assurent une qualité et une maintenabilité au code.

Annexes

Annexe 1 - List Router

```python
# pythons import
from typing import List, Union
from datetime import datetime
import re

# dependencies import
from fastapi import APIRouter, HTTPException
from odmantic import AIOEngine, ObjectId

# local import
from models.todolist import TodoList

engine = AIOEngine(database="diva")
router = APIRouter(
   tags=["lists"],
   responses={404: {"description": "not found"}}
)

# CREATE LIST
@router.post("/lists", response_model=TodoList, status_code=201)
async def add_list(list: TodoList):
   check_list = engine.find_one(TodoList, TodoList.name == list.name)
   if check_list is not None:
       raise HTTPException(
           status_code=405,
           detail="Une liste avec ce nom existe déjà."
       )
   list.created_at = datetime.now()
   return await engine.save(list)

# READ ALL COLLECTION, OR COLLECTION BY NAME
@router.get("/lists", response_model=Union[List[TodoList], TodoList])
async def get_lists(q: str = None):
   if q is None:
       return await engine.find(TodoList)
   return await engine.find(TodoList, {"name": re.compile(q)})

# UPDATE COLLECTION
```

```python
@router.patch("/lists/{list_id}", response_model=TodoList)
@router.put("/lists/{list_id}", response_model=TodoList)
async def update_list_by_id(list_id: ObjectId, input: TodoList):
    list = await engine.find_one(TodoList, TodoList.id == list_id)
    if list is None:
        raise HTTPException(
            status_code=404,
            detail="Cette liste n'existe pas."
        )
    list.name = input.name or list.name
    list.items = input.items or list.items
    return engine.save(list)

# DELETE COLLECTION
@router.delete("/lists/{list_id}")
async def delete_list_by_id(list_id: ObjectId):
    list = await engine.find_one(TodoList, TodoList.id == list_id)
    if list is None:
        raise HTTPException(
            status_code=404,
            detail="Cette liste n'existe pas."
        )
    await engine.delete(list)
```

Annexe 2.1 - Entraînement du modèle

```python
@router.get("/train")
def train():
    # load data
    print('Loading Data ...')
    db_url = "http://localhost:8000/data"

    try:
        df = pd.read_json(db_url)
    except Exception as e:
        raise RuntimeError('Failed to connect to database') from e

    df = df[['utterance', 'intent']].drop_duplicates()
    df = df.sample(frac=1)

    # Taking care of X
    print('Preprocessing')
```

```
text_cleaner = FunctionTransformer(clean_utterance)

token_transformer = FunctionTransformer(tokenizer)

sequencer = FunctionTransformer(sequence)

# Taking care of y
y = np.array(pd.get_dummies(df.intent))

# Modeling
def create_model(activation='softmax', optimizer='adam'):
    model = Sequential()
    model.add(Embedding(
        input_dim=1000,
        output_dim=64,
        input_length=10
    ))
    model.add(LSTM(units=100))
    model.add(Dense(4, activation=activation))

    model.compile(loss='categorical_crossentropy',
                  optimizer=optimizer,
                  metrics=['accuracy'])

    return model

clf = KerasClassifier(
    build_fn=create_model,
    validation_split=0.3,
    batch_size=16,
    epochs=40
)

pipe = pipeline.Pipeline([
    ('cleaning', text_cleaner),
    ('tokenize', token_transformer),
    ('sequencer', sequencer),
    ('lstm', clf)
])

pipe.fit(df['utterance'], y)

directory = os.path.dirname(os.path.realpath(__file__))
model_step = pipe.steps.pop(-1)[1]
```

```
    with open(os.path.join(directory, '../services/pipeline.pkl'), 'wb') as f:
        f.write(pickle.dumps(pipe))
    models.save_model(
        model_step.model,
        os.path.join(directory, '../services/model.h5')
    )
    return {"text": "new model trained"}
```

Annexe 2.2 - Prédiction d'intention

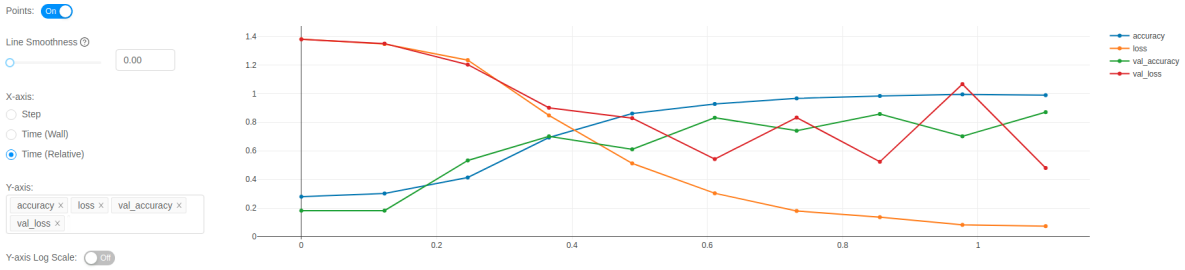
```
@router.post("/predict")
def predict(utt: UtterancePredictionPayload):
    directory = os.path.dirname(os.path.realpath(__file__))
    print(directory)
    pipe = pickle.load(open(f'{directory}/../services/pipeline.pkl', 'rb'))
    model = models.load_model(os.path.join(directory, '../services/model.h5'))
    pipe.steps.append(('lstm', model))

    pred = pipe.predict([utt.utterance])
    result = np.asarray(pred[0])

    if max(result) == result[0]:
        return {
            "intent": "AddItem",
            "score": result[0].item()
        }
    elif max(result) == result[1]:
        return {
            "intent": "GetItem",
            "score": result[1].item()
        }
    elif max(result) == result[2]:
        return {
            "intent": "RemoveItem",
            "score": result[2].item()
        }
    else:
        return {
            "intent": "SendList",
            "score": result[3].item()
        }
```

Annexe 3 - MlFlow

▼ Metrics

Name	Value
accuracy	0.994
loss	0.032
val_accuracy	0.896
val_loss	0.429

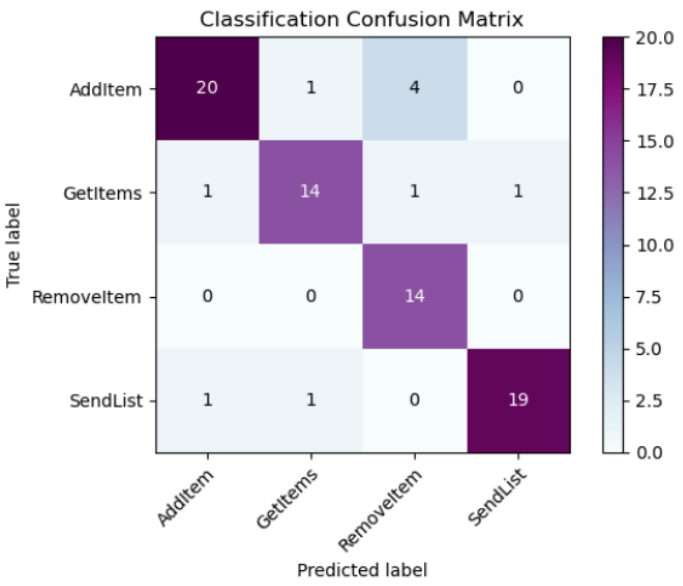
Log des metrics au sein d'MlFlow



Affichage des courbes en fonction des epoques

	precision	recall	f1-score	support
AddItem	0.91	0.80	0.85	25
GetItems	0.88	0.82	0.85	17
RemoveItem	0.74	1.00	0.85	14
SendList	0.95	0.90	0.93	21
accuracy			0.87	77
macro avg	0.87	0.88	0.87	77
weighted avg	0.88	0.87	0.87	77

Rapport de classification


Classification Confusion Matrix
AddItem
GetItems
RemoveItem
SendList
True label
20
1
4
0
1
14
1
1
0
0
14
0
1
1
0
19
AddItem
GetItems
RemoveItem
SendList
Predicted label
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0

Annexe 3 - The zen of python

```
>>> import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
>>>
```