Chatbot Industriel

Morel Louis, Monet Anaïs SDIA

Septembre 2020

1 Introduction

L'objectif de ce projet est de développer un code de prise en compte automatique de demande d'impression par Chatbot et de prévoir l'ordonnancement des tâches d'impression. Pour se faire, ce projet s'articule en deux étapes. La première consiste à élaborer un modèle de classification des demandes. La seconde étape réalise l'analyse d'une requête utilisateur, la classifie et en extrait les informations pertinentes.

2 Première étape

2.1 Création de la base de données

Pour démarrer ce projet, nous avons d'abord procéder à la création d'une base de données constituée d'exemples de message associé à son attente. Pour accéler la création de cette base de données, nous avons chacun créer 200 lignes d'exemples dans deux fichiers excel distincts comportant le même header. Notre objectif lors de cette création était d'éviter les doublons afin que notre base de données soit la plus diversifiée possible pour un meilleur apprentissage de notre modèle. Concernant les messages associés à l'attente "autre", nous avons inclus des messages qui n'avaient aucun rapport avec une demande d'impression mais aussi des demandes non conformes ou même des messages ambigus (voir Table 1, ligne 3 et ligne 4).

Table 1: Extrait de la base de données

Attente	Message
impression	Je souhaiterais imprimer le document doc8 de 9 pages
impression	impression doc6 54 pages
autre	impression doc8 67
autre	Où en est l'impression du document doc8 et des ses 56 pages ?

2.2 Création du modèle

2.2.1 Nettoyage des données

Pour créer notre modèle, nous avons choisi d'utiliser Python avec la librairie Spacy. Spacy est une librairie Open-source supportant plus de 49 langues et est très utilisée en traitement automatique des langues.
Le processus de création du modèle est le suivant. Tout d'abord, nous récupérons les fichiers .xlsx dans notre dossier data que nous stockons dans un DataFrame df. La première étape dans ce processus est le nettoyage des données. Par conséquent avant de créer notre modèle, nous avons pré-traité notre DataFrame en supprimant les doublons. Nous découpons les données en jeu d'apprentissage et de test. Nous avons décidé de répartir 80% des données pour l'apprentisage et le reste pour la partie test.

2.2.2 Traitement du texte

Pour traiter le contenu des messages, nous avons utilisé Spacy. La fonction slip_into_lemmas_spacy() va découper les mots. Ici, la lemmatisation nous semble être un choix légitime. En effet cette méthode permet de dériver les mots et d'en extraire un sens. Conscient qu'elle a une complexité en temps plus élevée que la méthode de stemming, qui se ramène à la racine du mot, nous avons jugé plus adaptée la lemmatisation puisque le sens du dialogue est ici très important.
Pour la construction de notre modèle, nous avons choisi d'utiliser des pipelines. Ces pipelines nous permettent de décrire des chaînes de traitement spécifiques pour la colonne message. Ces pipelines vont inclure le pré-traitement du texte ainsi que les étapes de classificiation (Figure 1).

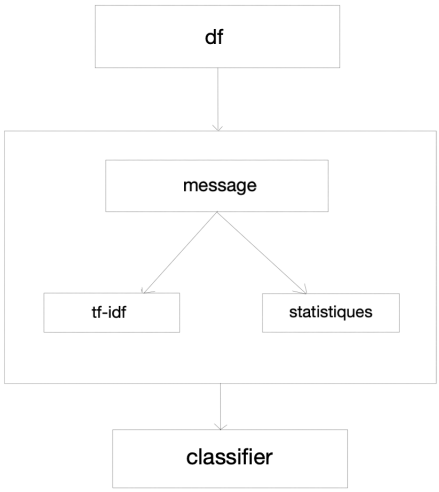

Figure 1: Pipeline

Tout d'abord on définit la classe SingleColumnSelector qui permet de sélectionner une colonne de notre table de données à partir de son nom.
Puis à l'aide de TfidVectorizer nous allons transformer dans la première pipeline le texte en sac de mots. Les mots seront écrit en minuscules et les mots vides seront supprimés. TfidVectorizer appartient à la librairie Scikit-Learn et convertit une collection de documents en matrice de caractéristiques. Cela va d'abord compter le nombre de mots dans l'ensemble de nos documents (ici message) puis va calculer les valeurs IDF (Inverse Document Frequency) et TF-IDF. L'IDF est un indicateur nous permettant de déterminer la fréquence de l'usage d'un mot. C'est-à-dire plus un mot est fréquemment utilisé à travers notre collection plus son score est faible ce qui signifie qu'il est moins important que ceux ayant un score élevé. La valeur TF-IDF d'un mot w est calculée à l'aide de la valeur IDF qui lui est associée. Cette valeur associe à w le poids qu'il a dans un document. Entre d'autres termes plus w est unique plus son score est élevé. Plus il est commun plus son score est bas.
Lors du processus de création du modèle, nous nous étions arrêter ici. Nous ne pensions pas avoir besoin de la seconde pipeline expliquée ci dessous. Nous avons alors entraîner notre modèle et nous nous sommes rendu compte que cela ne fonctionnait absolument pas. Même si nous avions une accuracy d'en moyenne 0.88, il y avait overfitting. Pour remédier à ce problème nous avons ajouté une deuxième pipeline.
La deuxième pipeline consiste à déduire une information supplémentaire concernant la colonne message. On va étudier sa longueur en nombre de caractères à l'aide de la classe TextStats(). Cette information sera transformée par sklearn.feature_extraction.DictVectorizer(). Cette fonction va transformer l'information précédente en un trait utilisable pour l'apprentissage sous forme de vecteur.
Ensuite on combine les deux pipelines pour obtenir une chaîne globale de traitement qui produira l'union des traits générés indépendamment par chacune des pipelines. Cette chaîne globale est preprocess_pipeline (cf model_train.py).
Pour passer à l'apprentissage, on effectue une chaîne complète en créant une pipeline nommée classifier_pipeline à partir de preprocess_pipeline et d'un algorithme de notre choix.

Table 2: Accuracy obtenue avec le jeu de données y_test

Algorithme	Accuracy
RandomForestClassifier	0.9
MultinomialNB	0.88
LogisticRegression	0.88
DecisionTreeClassifier	0.83

3 Seconde étape

3.1 Choix techniques pour la réalisation du Chatbot

Notre choix s'est tourné vers le langage Python. Nous avons décidé de réaliser ce Chatbot à l'aide d'un serveur Flask. La page d'ouverture est une page qui permet à l'utilisateur d'entrer son login avec son mot de passe au préalable créer avec la page signup. Ces deux pages sont des templates open-source trouvés sur internet que nous avons adapté aux besoins de notre projet. En ce qui concerne l'apparence du Chatbot, nous n'avons pas codé une interface graphique "from scratch". Nous avons également téléchargé un template open-source trouvé sur internet pour récupérer les fichiers HTML et CSS. Ce modèle convenait à nos attentes. C'est-à-dire une page où l'utilisateur peut intéragir avec le chatbot avec une entrée de texte et un bouton "Send". Nous avons ajouté un bouton qui permet de visualiser le planning des impressions de la semaine que nous avons créé et un autre qui permet à l'utilisateur de se déconnecter.

3.2 Partie analyse

3.2.1 Classification

Le modèle ayant été sauvegardé dans le fichier model.pkl à l'aide de la librairie pickle. Nous le chargeons juste dans notre main.py pour nous permettre de classifier les messages entrés par les utilisateurs. On récupère ensuite ces messages grâce à une balise form et la méthode POST de Flask. Lorsque le modèle prédit "autre", le chatbot répond qu'il n'est pas capable de répondre à la demande de l'utilisateur. En revanche si il prédit "impression" alors le chatbot répond qu'il lance l'impression du fichier confirmé avec son nombre de pages.

3.2.2 Récupération des données pertinentes

Le nom du document à imprimer et son nombre de page sont récupérés grâce au parsing de ce dernier. Lorsque le modèle prédit "impression" on parse le message plusieurs fois afin de récupérer les informations pertinentes à l'aide de la fonction split(). Pour récupérer le nombre de pages, nous procédons au parsing du message selon 'pages' qui nous permet d'obtenir deux éléments (ce qui précède 'pages' et ce qui suit 'pages'). Puis on parse le premier élément selon les espaces afin de récupérer seulement le nombre de pages qui est le dernier élément de ce dernier parsing. Pour le nom du document, on parse le message suivant 'doc' puis suivant les espaces. Ensuite il y a deux cas, si le premier élément est 'ument' alors le nom du document est l'élément suivant. Sinon, le nom du document est le premier élément (numéro) précédé de 'doc'. Le pseudo de l'utilisateur est lui récupéré grâce à la création d'une base de données database_user.sqlite au préalable créer avec une table User qui contient un id, un email, un pseudo et le mot de passe hashés de l'utilisateur.

3.3 Ordonnancement des tâches

Pour la création du planning, nous avons opté pour une méthode différente que celle initialement choisie. La méthode initiale consitait à créer une matrice où les lignes représentaient les jours et les colonnes les heures. Cette matrice contenait des 1 ou des 0. 0 si l'heure était libre et 1 sinon. Grâce à cette matrice nous écrivions le planning dans un fichier excel. Cependant, le fait d'utiliser des matrices n'était pas très optimal et notre excel n'avait qu'une précision à l'heure prêt. Pour remédier à ce problème nous avons pensé à avoir un fichier json stockant toutes les impressions planifiées qui permettrait d'avoir une précision à la seconde. Ce json a la forme suivante : Le titre, l'utilisateur, le nombre

```
{
    "titre": "doc8"
    "utilisateur" : "Louis",
    "start" : "MM/DD/AAAA HH:MM:SS" ,
    "end" : "MM/DD/AAAA HH:MM:SS ",
    "jour" : 1,
    "pages" : 25
}
```

Listing 1: Exemple d'un objet de calendrier.json

de pages sont récupérés par le procédé expliqué à la partie 3.2.2. Tout ce qui concerne date et heure, nous avons utilisé time et datetime de Python. "jour" est un chiffre compris entre 1 et 7. "start" est déterminé par des comparaisons réalisées entre l'impression précédente (si existante) et l'heure limite que nous avons fixé à 19:00:00.

4 Conclusion

Il aurait été intéressant de constituer une base de données plus élevée pour que notre modèle soit capable de détecter les cas ambigus et tout les cas d'impression, ce qui n'est pas le cas ici. Notre accuracy est relativement bonne mais des imperfections demeurent lors de la classification de certaines demandes. De plus notre code n'est pas assez optimisé surtout concernant la partie qui s'occupe d'écrire de le fichier calendrier.json. Notre approche est naïve, par manque de temps nous n'avons pas pu l'optimiser. Dans un objectif d'industrialisation de notre chatbot, il serait intéressant de pouvoir choisir les horaires d'ouvertures de l'imprimante. On trouve que cette amélioration est plus pertinente si nous avions mis en place des droits utilisateurs. Où l'on pourrait imaginer que seul l'administrateur aurait les droits de choisir les heures d'ouverture pour les impressions. Pour aller plus loin dans notre projet, il aurait été intéressant de demander confirmation à l'utilisateur sur le nom du document à imprimer et

sur son nombre de pages. De plus, nous avons toujours un problème selon les impressions qui doivent s'arrêter avant 19h. Notre programme autorise les impressions lancées avant 19h peu importe le nombre de pages ce qui n'est pas l'effet escompté. Concernant le planning des impressions, il y a une amélioration graphique à apporter. En effet les impressions prévues pendant des semaines différentes s'empilent sur la colonne qui correspond à leur jour d'impression. Il aurait fallu créer une nouvelle page pour chaque semaine.

References

• https://machinelearningmastery.com/save-load-machine-learning-models-python-scikit-learn/

• https://towardsdatascience.com/lemmatization-in-natural-language-processing-nlp-and-machine-learning-a4416f69a7b6

• https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html

• https://kavita-ganesan.com/what-is-inverse-document-frequency/#.X3WLFi8is1J