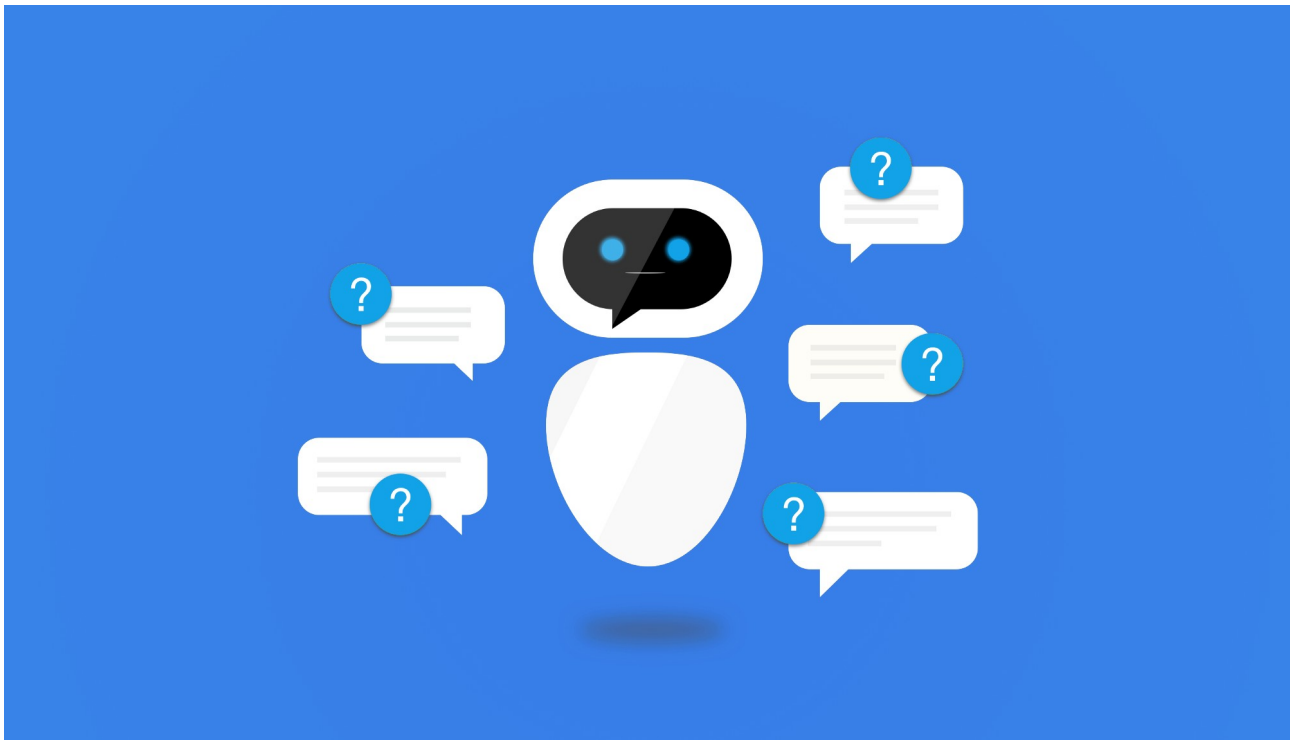
Rapport Technique Chatbot



Erwan Cloatre – Eva Moulard – Julien Furiga

Table des matières

I. Introduction

I.1. Objectifs du projet

L’objectif de se projet est de réaliser un chatbot pour un site web fictif pour l’école Microsoft. Plusieurs contraintes nous sont imposées ici :

- Les prédictions doivent se faire coté client (utilisation de Tensorflow JS)
- Le chatbot doit savoir si il s’adresse a un partenaire de l’école ou a un futur étudiant.
- Le champs des questions possible doit être large
- Une base de donnée doit être mise en place, ainsi qu’une api
- Une potentelle traduction en anglais

I.2. Mise au point sur les chatbot

Un chatbot est un robot conversationnel qui permet de répondre automatiquement, via une messagerie instantanée, aux requêtes/demandes d'un client (un être humain). Pour mettre cela en place sur un site web, plusieurs solution technique existent, on peut les consigner en 4grandes familles :

- Menu/button-based chatbots :

L’utilisateur est simplement guidé sur une interaface graphique composée de bouton et de choix prédéfinis.

- Rules-based chatbot :

Ici, on essaye de penser a toute les questions qu’un utilisateur va poser, et simplement comparer ce que l’utilisateur va taper aux questions déjà écrite pour pouvoir y répondre. Au niveau du code, celui ci sera uniquement composé de conditions.

- Retrieval chatbots :

Dans cette catégorie, on va analyser le texte de l’utilisateur et essayer de le classifier dans une des catégorie définie lors de la creation de notre modèle de classification. L’avantage est qu’il permet de potentiellement mieux comprendre l’entrée utilisateur.

- Contextual chatbots :

Ce dernier est le plus avancé, mais également le plus complexe a mettre en place. Dans celui-ci, on va également utiliser l'IA et le machine learning mais notre modèle va évoluer au cours du temps, en étant capable de s'améliorer et de s'adapter aux différentes demandes de l'utilisateur.

I.3. Choix pour le projet

Ici, nous avons fait le choix de nous orienter vers un modèle combiné entre un rule-based chatbot (que nous utiliserons en 1ere lieu pour connaître le type d'utilisateur) et un retrieval chatbot. En effet celui-ci est le meilleur compromis pour notre projet étant donné le temps imparti. Egalement, l'interet d'apprentissage sur un contextual chatbot serait limité étant donné que celui-ci fait partie d'un POC et ne sera pas utilisé en condition réelle.

Nous allons maintenant pouvoir voir toutes les étapes de création de notre projet.

II. Dataset

II.1. Forme des données

Pour ce projet de chabot, nous partons de zéro, et comme nous avons décidé d'utiliser de l'intelligence artificielle pour ce chatbot il nous fallait donc des données d'entraînement. Pour ces données d'entraînement, nous avons créé un dataset contenant des intentions, des questions et des réponses sous la forme d'une liste de dictionnaire composé de la classe, des questions et des réponses.

Exemple d'un dictionnaire de la liste :


```
"tag":"greetings",
"input":["hey salut", "bonjour", "yo", "salut", "salutation", "bien le bonjour"],
"responses_learner":["Bonjour !, Comment puis-je vous aider ?", "Bonjour !, Avez-vous besion d'aider ?", "Bonjour !, Que voulez vous ?"],
"responses_business":["Bonjour !, Comment puis-je vous aider ?", "Bonjour !, Avez-vous besion d'aider ?", "Bonjour !, Que voulez vous ?"]
```


Figure 1: Capture d'écran d'une classe

Avec tag le nom de la classe, *input* est une liste de questions et *responses_learner / responses_business* sont des listes de réponse soit pour un apprenant soit pour une entreprise.

II.2. Contenu

Dans ce dataset il y a plusieurs classes portant sur un sujet susceptible d'être évoqué pendants une conversation entre un apprenants ou un entreprise et le chatbot.

La répartition est la suivante :

Nom de la classe	nombre de question
greetings	6
goodbye	12
whatismicrosoftia	7
whatissimplon	6
whatisdevia	4
prerequisite	8
price	8
pedagogy	4
inscription	4

nextsession	3
pcspecs	7
timetable	7
whatisia	6
whyia	4
potentialia	6
educatioprogram	5
software	3
poleemploi	3
financilahelp	5
nature	4
validation	4

Ce qui nous donne un total de 116 questions réparties entre les 21 classes.

II.3. Enregistrement

Pour l'enregistrement des données, nous avons décidé de lui donner une forme qui facilite l'importation dans la base. Dans le fichier les classes sont a la suite des une des autre dans sans être dans une liste.

III. Fonctionnement technique

III.1. Prétraitement du texte

III.1.A. Tensorflow Tonkenizer

Documentation officiel : [tf.keras.preprocessing.text.Tokenizer]

Pour « tokenizer » les phrases nous avons choisi de prendre un module proposé par Tensorflow pour que ce soit plus simple et plus rapide.

```
tf.keras.preprocessing.text.Tokenizer(
    num_words=None,
    filters='!"#$%&()*+,-./:;<=>?@[\\]^_`{|}~\t\n',
    lower=True, split=' ', char_level=False, oov_token=None,
    document_count=0, **kwargs
)
```

Figure 2: Classe Tokenizer de Tensorflow

Il permet de vectoriser un corpus de texte, en transformant chaque texte soit en une séquence d'entiers (chaque entier étant l'index d'un « token » dans un dictionnaire) soit en un vecteur où le coefficient pour chaque « token » pourrait être binaire, basé sur le nombre de mots , basé sur tf-idf. Par défaut, toute ponctuation est supprimée, transformant les textes en séquences de mots séparés par des espaces (les mots peuvent inclure le caractère '). Ces séquences sont ensuite divisées en listes de « token ». Ils seront ensuite indexés ou vectorisés.

Nous devons ensuite utiliser la classe *fit_on_textes(),*
pour mettre à jour le vocabulaire interne, basé sur la liste de nos phrases.
Dans le cas où les textes contiennent des listes, il est supposé que
chaque entrée des listes est un « token ».

```
fit_on_texts(
    texts
)
```

Figure 3: Métode Tensorflow

Résultat sur notre dataset après avoir *fit_on_textes()* :

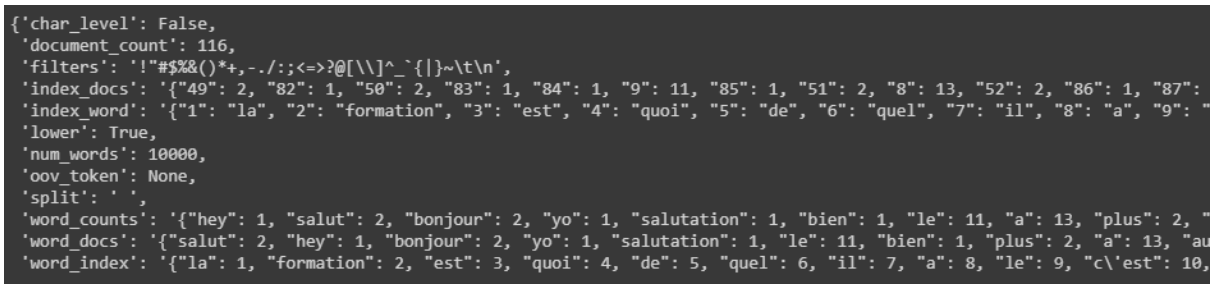


Figure 4: Capture d'écran de la configuration du Tokenizer

Dans notre cas, nous transformons chaque phrase en une séquence d'entiers avec *texts_to_sequences().*

Cela permet de transformer chaque texte des phrases en une séquence d'entiers.

Seuls les mots les plus fréquents de *num_words-1* seront pris en compte. Et seuls les mots connus du tokenizer seront pris en compte.

```
texts_to_sequences(
    texts
)
```

Figure 5: Métode Tensorflow

III.1.B. Sklearn LabelEncoder

Documentation officiel : [sklearn.preprocessing.LabelEncoder — scikit-learn 0.24.2 documentation]

Parce que le modèle après ces prédictions retourne un entier, qui doit correspondre à une classe de notre dataset, nous avons donc encodé les classes pour qu'elle corresponde un entier.

Pour ce faire et comme pour les raisons du Tokenizer (plus simple et facile) nous avons utilisé le label encoder de sklearn.

```
sklearn.preprocessing.LabelEncoder
```

Qui encode les étiquettes cibles(ici nos classes) avec une valeur comprise entre 0 et n_classes-1.

Nous avons passé la liste des classe de notre dataset avec *fit_transform(),*

qui met à jour le *LabelEncoder()* et retourne les étiquettes encodées.

Résultat du *LabelEncoder()* sur nos classes:

Figure 6: Capture d'écran du début du résultat du LabelEncoder()

III.2. Le Modèle

Documentation officiel : [Keras API reference]

Pour ce projet de chabot nous somme arrivé a un ANN simple.

Voici le modèle utilisé dans ce projet de chatbot :

```
model_test = tf.keras.Sequential()
model_test.add(Embedding(vocabulary+1, 137, input_length=input_shape))
model_test.add(GlobalMaxPooling1D())
model_test.add(Dense(470, activation='relu'))
model_test.add(Dropout(0.3))
model_test.add(Dense(62, activation='relu'))
model_test.add(Dropout(0.5))
model_test.add(Dense(output_lenght, activation='softmax'))
```

Figure 7: Capture d'écran du modèle

III.2.A. Composition du modèle

Voici la composition détaillée du modèle.

Embedding

Le modèle est composé d'un premier « layer » *Embedding() :* Qui transforme les entiers positifs (index) en vecteurs denses de taille fixe.

par exemple. [[4], [20]] -> [[0,25, 0,1], [0,6, -0,2]]

```
tf.keras.layers.Embedding(
    input_dim,
    output_dim,
    embeddings_initializer="uniform",
    embeddings_regularizer=None,
    activity_regularizer=None,
    embeddings_constraint=None,
    mask_zero=False,
    input_length=None,
    **kwargs
)
```

Figure 8: Classe Embedding de Keras

GlobalMaxPooling1D

Suivie du deuxième « layer » *GlabalMaxPooling1d() :*

```
tf.keras.layers.GlobalMaxPooling1D(data_format="channels_last", **kwargs)
```

Figure 9: Classe GlobalMaxPooling1D de keras

qui permet de sous-échantillonner la représentation d'entrée en prenant la valeur maximale sur la dimension temporelle.

Dense Relu

Suivie ensuite de deux « layer » *Dense()* avec une activation *relu :*

```
tf.keras.layers.Dense(
    units,
    activation=None,
    use_bias=True,
    kernel_initializer="glorot_uniform",
    bias_initializer="zeros",
    kernel_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    bias_constraint=None,
    **kwargs
)
```

Figure 10: Classe Dense de keras

Ils implémentent l'opération:
output = activation(dot(input, kernel) + bias)

où l'activation est la fonction d'activation élément par élément passée comme argument d'activation(*relu* dans notre cas), le noyau est une matrice de poids créée par la couche et le biais est un vecteur de biais créé par la couche (applicable uniquement si *use_bias* vaut *True*).
Les deux « layer » de *Dense()* sont séparé par un *Dropout()* de 0.3

Dense Softmax

Et pour le dernier « layer » qui sera donc la sortie nous avons aussi mit un Dense() mais avec un activation *Softmax* et une taille égale au nombre de classe de notre dataset.
Ce qui va nous servir à classer les phrases en entrée du modèle dans une des classe.
Mais juste avant ce « layer » ce trouve un *Dropout()* de 0.5

III.2.B. Compilation

Une fois le modèle construit il faut le compiler avec un optimizer, une « loss » et des métrique.

```
opt = tf.keras.optimizers.Adam(learning_rate=0.001)
model_test.compile(loss='sparse_categorical_crossentropy', optimizer=opt, metrics=['accuracy'])
```

Figure 11: Capture d'écran des paramètres de compilation

Loss

Pour la « loss » nous avons choisi *sparse_categorical_crossentropy() :*

```
tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=False, reduction="auto", name="sparse_categorical_crossentropy"
)
```

Figure 12: Classe de la « loss » utilisé

Elle permet de calculer la perte d'entropie croisée entre les étiquettes et les prédictions.

Cette fonction s'utilise lorsqu'il existe au moins deux classes d'étiquettes. Et attend à ce que les étiquettes soient fournies sous forme d'entiers.

Ce qui correspond exactement à nos classe une fois passer a travers le *LabelEncoder()*

Optimizer

Pour l'optimiser nous avons choisi *adam() :*

Il s'agit d'un optimiseur qui implémente l'algorithme Adam.

L'optimisation Adam est une méthode de descente de gradient stochastique basée sur l'estimation adaptative des moments du premier et du second ordre.

```
tf.keras.optimizers.Adam(
    learning_rate=0.001,
    beta_1=0.9,
    beta_2=0.999,
    epsilon=1e-07,
    amsgrad=False,
    name="Adam",
    **kwargs
)
```

Figure 13: Classe de l'optimizer Adam

Metrics

Et pour la métrique nous avons pris *accuracy* pour avoir la précision du modèle

```
tf.keras.metrics.Accuracy(name="accuracy", dtype=None)
```

Figure 14: Classe de la métrique accuracy

Cette métrique calcule la fréquence à laquelle les prédictions égalent les étiquettes.

Et elle crée deux variables locales, *total* et *count*, qui sont utilisées pour calculer la fréquence à laquelle *y_pred* correspond à *y_true*.

Cette fréquence est finalement renvoyée sous forme de précision binaire: une opération idempotente qui divise simplement le *total* par le *count*.

III.2.C. Entraînement

Pour l'entraînement de ce modèle nous avons un *batch_size=5*, un nombre *d'epochs=30*

```
model_test.fit(X_train, y_train, batch_size=5,epochs=30, validation_data=(x_test,y_test), shuffle=True)
```

Figure 15: Capture d'écran des paramètre d'entraînement du modèle

III.2.D. Évaluation

Pour évaluer le modèle nous avons utilisé plusieurs méthodes : graphiques et la méthode *evaluate()* de tensorflow.

Graphique

Pour voir si le modèle ne se surentraîne pas nous avons fait des graphiques pour la *loss* et l'*accuracy.*

Loss/ Validaion loss

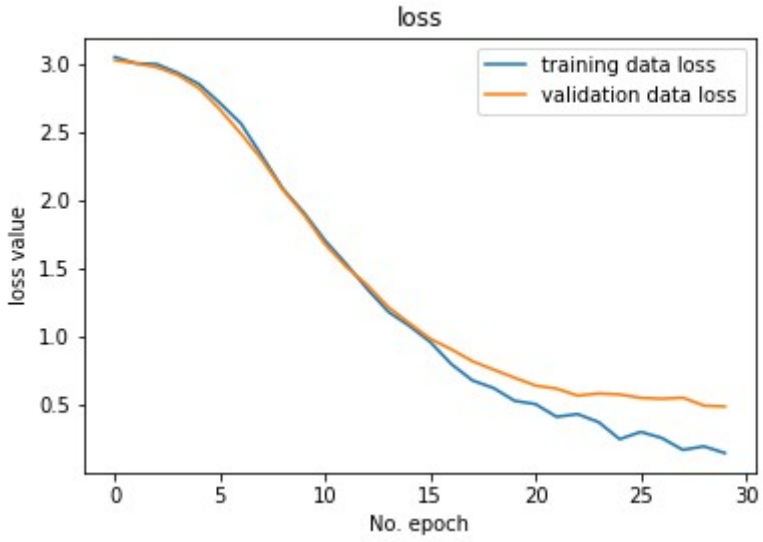


Figure 16: Graph de la loss et la val_loss

Accuracy / Validation accuracy

Figure 17: Graph de l'accuracy et la val_accuracy

Matrice de confusion

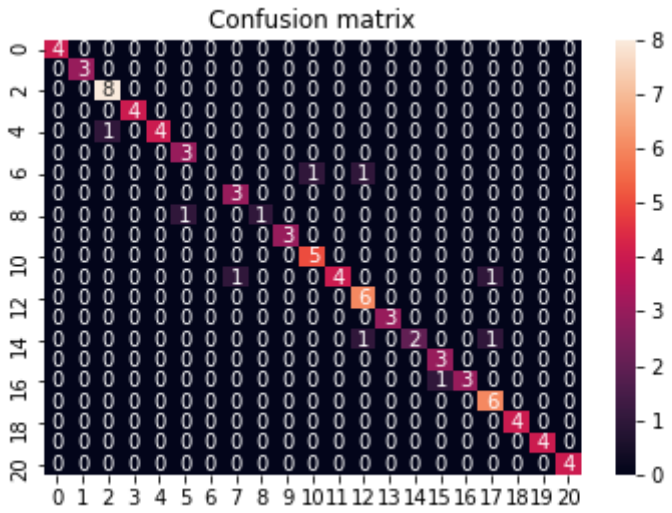


Figure 18: Heatmap de la matrice de confusion du modèle

Méthode evaluate() de Tensorflow

Nous avons aussi utilisé la méthod de tensorflow *evaluate() :*

```
model.evaluate(x_test, y_test)
```

Figure 19: Capture d'écran de la méthode evaluate()

Ce qui nous retourne la *loss* et l'*accuracy* sur des données de test.

```
{"loss": 0.49244773387908936, "accuracy": 0.895348846912384}
```

Figure 20: Capture d'écran du resulata

Ici, nos metrics sont plutots bonnes. En effet, on constate que l'on a une accuracy sur le jeux de test de 89 %. Les courbes différentes courbes nous montrent clairement que nous n'avons pas de problème d'over/under fitting. En effet la loss_validation est toujours plus haute que la loss_train.

Les chiffres sur notre matrice de confusion nous montre bien que nous avons une répartition homogène de nos erreurs, aucune classe n'est vraiment mal classifiée.

IV. Chatbot Front

IV.1. Choix technique

Le site web tourne sur un serveur Flask avec un backend en python. Pour la gestion du front et des pages web, l'application est gérée en Javascript. L'UI du chatbot a été trouvée sur codepen puis réadaptée a nos besoins.

La première étape de fonctionnement est la récupération du statut de l'utilisateur. Cette partie est codée directement en javascript avec une analyse du l'entrée utilisateurs. Suite a cela, une variable est fixée pour la durée du chargement de la page. Celle-ci nous sera utile ultérieurement afin de définir notre appel a l'api. En effet, ce dernier différera en fonction du type d'utilisateurs.

Une fois le type d'utilisateur récupéré, le JS envoie un message de salutation et le partie retrivial chatbot commence. Les données saisies par l'utilisateur sont récupérées dans un variable, puis envoyées par une requete Ajax a notre backend. Dans ce dernier, nous avons préalablement récupérés le Tokenizer utilisé lors de l'entrainement. Une fois les données traités, nous récupérons le vecteur créé, le formatons en tenseur puis le passons a notre modèle préalablement chargé afin d'effectuer notre prédiction. Celle-ci est récupérée, puis renvoyé en ajax a notre backend afin d'effectuer une requete sur notre base pour répondre a la question dans le thème prédit par notre modèle.

Une fois la réponse récupérée, elle est envoyée au JS afin d'etre affichée au a l'utilisateur.

V. API & Base de donnée

V.1. Base de donnée

La base de donnée selectionnée est Mongo DB, car les fichiers attendus en entrée par notre modèle sont en JSON , et que le format document est le plus pratique pour notre utilisation. En effet, chaque document peut être identifié par son tag qui est unique. De plus, la mise en place de cette dernière en container docker est facile et rapide

V.2. API

L'api a été réalisé avec FastAPI et containeurisé dans le même compose que la base de donnée. Elle possède un seul endpoint, qui nous retourne uniquement une phrase prise au hasard dans le pool de réponse du tag demandé pour le type d'utilisateur demandé. Elle est de la forme :

<addresse-ip-VM>/responses/<tag>/<type-de-personne>

VI. Recommandation

VI.1. Recommandation pour les données

VI.1.A. Données d'entraînement

Le dataset actuel est composé de 116 questions réparties dans 21 classes. Ce qui est peu pour des données d'entraînement.
Nous recommandons de faire beaucoup plus de questions et de classes, vers les 1000 questions et 60 classes, et qu'elle soit très variée.

VI.1.B. Données de test

Les données de test sont composées de quelques questions pour chaque classe.
La recommandation est qu'il faudrait un plus grand jeu de test créer qui n'a rien à voir avec le jeu d'entraînement.

VI.2. Recommandation pour le modèle

Les recommandation pour le modèle, en plus des recommandation pour les données, serais de faire plus de recherches et de tests sur sa composition et ces hyperparamètres.

VII. Conclusion :

En conclusion, nous pouvons voir que les objectifs du projet ont été atteints. Le site web est en place, ainsi que l'api et la base de donnée. Ce que l'on peut voir également, c'est qu'au niveau du chatbot, on constate qu'en condition réelle, avec des utilisateurs différent, sa puissance reste limitée. Cette dernière pourrait, comme précisé plus haut, être améliorée par un agrandissement du dataset et des catégories classées. On pourrait imaginer une récupération des données saisie par l'utilisateur et un traitement manuel des réponses afin d'agrandir le dataset. On pourrait également prévoir par la partie sur la traduction en anglais qui n'a pas été réalisée par manque de temps.