

UMMTO
TIZI-AUZOU



RAPPORT

PAR :

Belkessa Amine
Zeraoui Yani
Mabed Salim

Sommaire

Introduction

État de l'art

Approches adoptées

Mise en œuvre

Nos Données

BERT et Transformers

Réseaux de neurones

TFIDF

Comparaison des méthodes

Conclusion

Introduction

Dans le paysage numérique en constante évolution d'aujourd'hui, les entreprises et les organisations cherchent sans cesse des moyens innovants pour améliorer leur interaction avec les utilisateurs. Parmi les technologies émergentes qui transforment cette interaction, les chatbots se démarquent comme des outils puissants permettant d'automatiser et de personnaliser les interactions avec les utilisateurs à travers diverses plateformes en ligne.

Un chatbot, également connu sous le nom d'agent conversationnel, est un programme informatique conçu pour simuler une conversation humaine, ceci à l'aide de l'intelligence artificielle (IA) et du traitement du langage naturel (NLP). Ces agents conversationnels sont devenus omniprésents dans nos vies quotidiennes, que ce soit pour fournir un support client, automatiser des tâches simples, ou même offrir des conseils personnalisés.

Ayant déjà les bases de l'intelligence artificielle et du traitement automatique du langage naturelle on s'est donner le défi de créer deux chatbots et de les faire parler entre eux , dans ce rapport nous expliquons les différentes méthodes utiliser pour atteindre cet objectif.

État de l'art

Dans le domaine de la conception de chatbots, plusieurs techniques et outils sont utilisés pour créer des systèmes conversationnels intelligents. Parmi les notions théoriques essentielles à comprendre pour ce projet, nous pouvons citer :

NLTK (Natural Language Toolkit) :

NLTK est une bibliothèque Python largement utilisée pour le traitement du langage naturel. Elle offre une gamme de fonctionnalités telles que la tokenisation, la lemmatisation, le marquage morphologique, l'analyse syntaxique, etc. Ces fonctionnalités sont cruciales pour le prétraitement des données textuelles avant de les utiliser dans des modèles de chatbot.

Keras :

Keras est une bibliothèque open-source haut niveau pour le développement rapide de réseaux de neurones. Intégré à TensorFlow, il permet de créer, d'entraîner et de déployer des modèles de deep learning avec une syntaxe simple et intuitive.

Tokenisation :

La tokenisation est le processus de division d'une phrase en unités plus petites appelées "tokens". Ces tokens peuvent être des mots, des phrases ou même des caractères individuels, selon les besoins. La tokenisation est une étape fondamentale dans le traitement du langage naturel, car elle permet de représenter le texte sous une forme adaptée à l'analyse informatique.

PyTorch et Transformers :

PyTorch est une bibliothèque de machine learning développée par Facebook, offrant une approche dynamique et flexible pour la construction de modèles. Les Transformers sont des modèles d'apprentissage profond largement utilisés pour les tâches de traitement du langage naturel, introduits dans le cadre du modèle BERT.

BERT (Bidirectional Encoder Representations from Transformers) :

BERT est un modèle de langue pré-entraîné développé par Google, basé sur l'architecture des Transformers. Il a révolutionné de nombreux domaines du traitement du langage naturel en raison de sa capacité à capturer les contextes bidirectionnels des mots dans une phrase.

TensorFlow :

TensorFlow est une bibliothèque open-source développée par Google, largement utilisée pour la création de modèles de machine learning et de deep learning. Il offre une grande flexibilité pour la conception et l'entraînement de modèles de chatbot, notamment avec l'utilisation de bibliothèques supplémentaires comme Keras.

GPT-2 (Generative Pre-trained Transformer 2) :

GPT-2 est un modèle de langage développé par OpenAI, basé sur l'architecture des Transformers. Il est connu pour sa capacité à générer du texte cohérent et humain-comme, ce qui en fait un choix populaire pour les tâches de génération de texte dans les chatbots.

Lemmatisation :

La lemmatisation est le processus de réduction des mots fléchis (ou des variantes) à leur forme de base, appelée lemmes. Par exemple, les mots "marche", "marchait" et "marchant" seraient tous réduits au lemme "marcher". Cela permet de normaliser le texte et de réduire la dimensionnalité de l'espace des mots, ce qui peut améliorer les performances des modèles de chatbot.

TF-IDF (Term Frequency-Inverse Document Frequency) :

TF-IDF est une technique de pondération largement utilisée en recherche d'information et en extraction de caractéristiques textuelles. Elle évalue l'importance relative d'un terme dans un document par rapport à une collection de documents. La pondération est basée sur deux composantes principales : la fréquence du terme dans le document (TF) et l'inverse de la fréquence du terme dans l'ensemble des documents (IDF). Cette méthode est souvent utilisée pour la représentation vectorielle de documents dans des modèles de classification de texte et peut être utile pour l'extraction de caractéristiques dans le contexte de la conception de chatbots.

Quelques travaux clés réalisés dans le domaine :

1. Building a Simple Chatbot from Scratch in Python (TFIDF) :

Lien : <https://medium.com/analytics-vidhya/building-a-simple-chatbot-in-python-using-nltk-7c8c8215ac6e>

- Un chatbot basé sur la récupération qui utilise tf-idf pour faire correspondre la requête de l'utilisateur avec la phrase pertinente dans les corpus.

2. Chatbot using BERT :

Lien : <https://medium.com/geekculture/simple-chatbot-using-bert-and-pytorch-part-1-2735643e0baa>

3. Build a chat bot from scratch using Python and TensorFlow

Lien : <https://handsonai.medium.com/build-a-chat-bot-from-scratch-using-python-and-tensorflow-fd189bcfae45>

Approches adoptées

Approche Adoptée avec BERT

L'architecture globale du système utilise le modèle de langage BERT (Bidirectional Encoder Representations from Transformers) pour traiter les questions des utilisateurs . Voici comment l'approche adoptée avec BERT est intégrée dans le système :

1. Prétraitement des Données :

- Les données d'entraînement sont **prétraitées** pour créer un **vocabulaire de mots uniques**, des classes d'intentions et des documents contenant des paires de mots et de tags (intentions). Les mots sont lemmatisés et les caractères non pertinents sont supprimés, tout comme dans l'approche avec le modèle séquentiel.

2. Utilisation de BERT pour la Classification des Intentions :

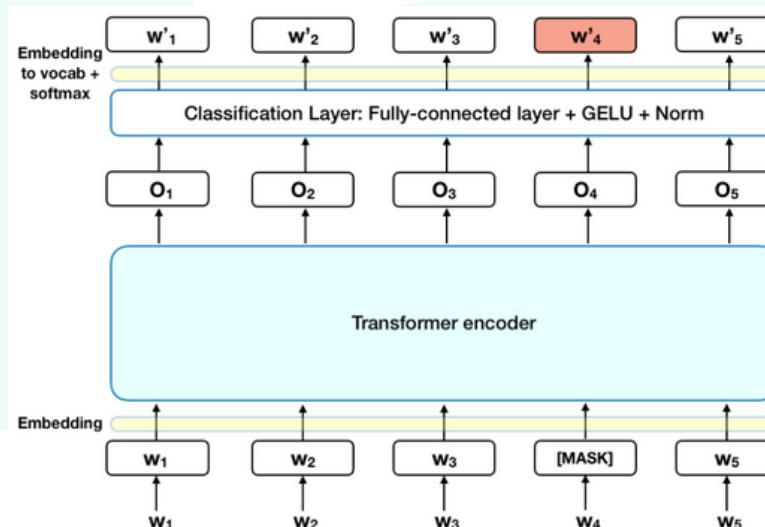
- Contrairement à l'approche séquentielle, où un réseau de neurones est entraîné pour classer les questions dans des intentions prédefinies, avec **BERT**, nous utilisons le modèle pré-entraîné pour la classification des séquences. **BERT** est capable de capturer les relations contextuelles entre les mots dans une phrase grâce à son architecture transformer bidirectionnelle.

3. Fine-Tuning de BERT pour la Classification d'Intentions :

- Le modèle **BERT** est **fine-tuné** sur les données **prétraitées** spécifiques à notre tâche de classification d'intentions. Cela implique d'adapter les poids du modèle pré-entraîné pour qu'il puisse mieux comprendre les nuances des questions liées à la santé mentale. L'optimisation est effectuée en minimisant la perte de catégorisation croisée, tout comme dans l'approche séquentielle.

4. Prédiction des Intentions et Réponses :

- Une fois que **BERT** est **fine-tuné**, il est capable de prendre une question en entrée et de prédire l'intention associée. Cette intention est ensuite utilisée pour sélectionner une réponse appropriée à partir d'un ensemble de réponses prédefinies, tout comme dans l'approche avec le modèle séquentiel.



Approche Adoptée avec le Modèle Séquentiel de TensorFlow (Réseaux de neurones)

L'architecture globale du système utilise un modèle séquentiel de TensorFlow pour traiter les questions des utilisateurs. Voici comment l'approche adoptée avec ce modèle séquentiel est intégrée dans le système :

1. Prétraitement des Données :

- Les données d'entraînement sont **prétraitées** pour créer un **vocabulaire de mots uniques**, des classes d'intentions et des documents contenant des paires de mots et de tags (intentions). Les mots sont lemmatisés et les caractères non pertinents sont supprimés pour simplifier le traitement.

2. Construction du Sac de Mots (Bag of Words) :

- Chaque question est représentée sous forme de vecteur numérique à l'aide d'une technique de sac de mots (**Bag of Words**). Dans cette approche, chaque mot est représenté par un indice dans un vecteur de taille fixe, et le vecteur contient des valeurs binaires indiquant si chaque mot est présent ou non dans la question.
- Cette représentation est utilisée comme entrée pour le modèle séquentiel.

3. Entraînement du Modèle Séquentiel :

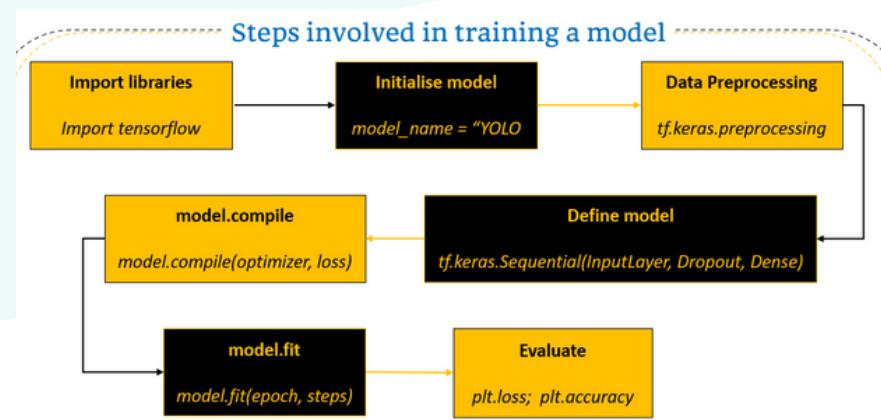
- Un modèle séquentiel est construit à l'aide de **TensorFlow** et **Keras**. Ce modèle comporte plusieurs couches de neurones, y compris des couches denses entièrement connectées. Les données d'entrée (les sacs de mots) sont alimentées dans le modèle, qui est ensuite entraîné sur plusieurs Epochs à l'aide d'un algorithme d'optimisation comme la descente de gradient stochastique (SGD).

4. Classification des Intentions :

- Pendant l'entraînement, le modèle séquentiel apprend à associer chaque sac de mots à une intention spécifique. Une fois que le modèle est entraîné, il peut prendre une question en entrée, la représenter comme un sac de mots, et prédire l'intention associée à cette question.

5. Sélection et Génération de Réponses :

- Après avoir prédit l'intention d'une question, le modèle sélectionne une réponse appropriée à partir d'un ensemble de réponses pré-définies associées à chaque intention. La réponse sélectionnée est renvoyée à l'utilisateur comme réponse au sein de la conversation.



Approche Adoptée avec TFIDF

L'architecture globale du système utilise un modèle séquentiel de TensorFlow pour traiter les questions des utilisateurs. Voici comment l'approche adoptée avec ce modèle séquentiel est intégrée dans le système :

1. **Traitement des données**
2. **Vectorisation des modèles de phrases** : Les modèles de phrases sont transformés en vecteurs numériques à l'aide de la technique TF-IDF (Term Frequency-Inverse Document Frequency). Cela permet de représenter chaque modèle de phrase sous forme de vecteur numérique, où chaque composante du vecteur correspond à l'importance relative d'un mot dans le modèle de phrase par rapport à l'ensemble des modèles de phrases.
3. **Encodage des étiquettes** : Les étiquettes sont encodées en utilisant la méthode LabelEncoder de la bibliothèque scikit-learn. Cela permet de représenter les étiquettes sous forme de valeurs numériques.
4. **Calcul de la similarité** : Lorsqu'une requête est soumise au chatbot, elle est également transformée en un vecteur numérique à l'aide du même processus de vectorisation TF-IDF que celui utilisé pour les modèles de phrases. Ensuite, la similarité cosinus est calculée entre le vecteur de la requête et tous les vecteurs de modèles de phrases. La similarité cosinus mesure la similarité directionnelle entre deux vecteurs dans un espace vectoriel. Dans ce contexte, elle est utilisée pour mesurer la similarité entre la requête de l'utilisateur et les modèles de phrases dans l'ensemble d'entraînement.
5. **Identification de la meilleure correspondance** : L'index du modèle de phrase ayant la plus grande similarité cosinus avec la requête de l'utilisateur est identifié. En utilisant cet index, le tag correspondant au modèle de phrase est récupéré.
6. **Sélection d'une réponse** : Enfin, une réponse appropriée est sélectionnée au hasard parmi les réponses associées au tag trouvé dans les données d'entraînement.

Nos Données

Le pétrole de l'intelligence artificielle n'est rien d'autre que les données, sans données l'IA ne serait pas possible. Ainsi en cherchons attentivement sur plusieurs sites on a pu choisir les données d'entraînement qui nous conviennent.

Description du dataset:

Le jeu de données choisi pour l'entraînement est un fichier JSON (java script Object notation) contenant des exemples de discussion question réponse, salutation réponse ... L'élément racine est intents il contient des sous-éléments avec les propriétés :

Tag : Est le titre de l'élément (le sujet que l'élément traite)

Patterns: Représente plusieurs cas d'approche pour aborder le sujet 'Tag' dans une discussion

Responses : Représente les différentes réponses possibles aux Patterns.

```
{"tag": "night",
  "patterns": ["Good night"],
  "responses": ["Good night. Get some proper sleep",
                "Good night. Sweet dreams."]
},
```

Exemple d'un élément du Dataset

Le dataset est présent sur Kaggle sous le nom "Mental Health Conversational Data".

Amélioration du Dataset

Le dataset original récupéré n'était pas assez complet pour créer un chatbot qui soit assez générale, alors on a ajouté quelques lignes pour qu'il soit plus générale.

Le dataset d'origine contenait 300 lignes sur la psychologie et l'assistance, ce qui est parfait car un bon chatbot doit se préoccuper de la personne qui l'utilise, en plus de ça on a voulu que notre chatbot ait des connaissances générales sur plusieurs sujets.

Ainsi on a ajouté des tags à ce dataset en l'a aidant de ChatGPT en lui demandant d'ajouter des tags sur différents sujets. Le dataset final contient 1023 lignes.

BERT et Transformers

Importation des bibliothèques :

```
💡 Click here to ask Blackbox to help you code faster
# Ignorer les avertissements pour améliorer la lisibilité du code
import warnings
warnings.filterwarnings('ignore')

Python
```

Importation des bibliothèques

```
💡 Click here to ask Blackbox to help you code faster
import json
import numpy as np
import pandas as pd
import random
from matplotlib import pyplot as plt
import seaborn as sns
from wordcloud import WordCloud,STOPWORDS
import missingno as msno

from sklearn.feature_extraction.text import CountVectorizer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, precision_recall_fscore_support

from keras.preprocessing import text
import keras
from keras.models import Sequential
from keras.layers import Dense, Embedding, LSTM, Dropout
from keras.callbacks import ReduceLROnPlateau
```

- **warnings.filterwarnings('ignore')** : Cette ligne permet de masquer les avertissements pour rendre le code plus lisible.
- **Importations de bibliothèques** : Les différentes bibliothèques nécessaires pour le projet, notamment pour la visualisation des données, le prétraitement, l'entraînement de modèles, etc.

Chargement du Fichier JSON :

Fonction pour Charger un Fichier JSON

```
💡 Click here to ask Blackbox to help you code faster
def load_json_file(filename):
    with open(filename) as f:
        file = json.load(f)
    return file

filename = 'C:/Users/AKAM/Downloads/intents.json'

intents = load_json_file(filename)

Python
```

- **def load_json_file(filename)** : Cette fonction est définie pour charger un fichier JSON à partir d'un chemin de fichier donné.
- **with open(filename) as f** : Utilisation de la déclaration **with** pour ouvrir le fichier spécifié en mode lecture.
- **file = json.load(f)** : Chargement du contenu du fichier JSON dans la variable **file**.
- **return file** : Retourne le contenu du fichier JSON.
- **filename = 'C:/Users/AKAM/Downloads/intents.json'** : Chemin du fichier JSON à charger.
- **intents = load_json_file(filename)** : charger le contenu du fichier JSON dans **intents**.

Création d'un DataFrame :

Création d'un DataFrame Pandas Vide

```
💡 Click here to ask Blackbox to help you code faster
def create_df():
    df = pd.DataFrame({
        'Pattern' : [],
        'Tag' : []
    })

    return df

df = create_df()
df
```

Python

Pattern Tag

- `def create_df()` : Cette fonction est définie pour créer un DataFrame vide avec deux colonnes, 'Pattern' et 'Tag'.
- `df = pd.DataFrame({'Pattern' : [], 'Tag' : []})` : Utilisation de la bibliothèque pandas pour créer un DataFrame avec les colonnes spécifiées.
- `return df` : Retourne le DataFrame nouvellement créé.
- `df = create_df()` : Appel de la fonction pour créer un DataFrame vide.
- `df` : Affiche le DataFrame créé, qui est vide pour le moment.

Extraction d'Informations depuis un Fichier JSON :

Extraction d'Informations depuis un Fichier JSON vers un DataFrame

```
💡 Click here to ask Blackbox to help you code faster
def extract_json_info(json_file, df):
    for intent in json_file['intents']:
        for pattern in intent['patterns']:
            sentence_tag = [pattern, intent['tag']]
            df.loc[len(df.index)] = sentence_tag

    return df

df = extract_json_info(intents, df)
df.head()
```

Python

- `def extract_json_info(json_file, df)` : Cette fonction prend en entrée un fichier JSON et un DataFrame vide, puis extrait les informations pertinentes pour les ajouter au DataFrame.
- `for intent in json_file['intents']` : Boucle à travers chaque intention dans le fichier JSON.
- `for pattern in intent['patterns']` : Boucle à travers chaque pattern dans les intentions.
- `sentence_tag = [pattern, intent['tag']]` : Crée une liste avec le pattern et le tag associé.
- `df.loc[len(df.index)] = sentence_tag` : Ajoute la liste comme une nouvelle ligne dans le DataFrame.
- `df = extract_json_info(intents, df)` : Appel de la fonction pour extraire les informations du fichier JSON et les ajouter au DataFrame.
- `df.head()` : Affiche les premières lignes du DataFrame pour vérification.

	Pattern	Tag
0	Hi	greeting
1	Hey	greeting
2	Is anyone there?	greeting
3	Hi there	greeting
4	Hello	greeting

Affichage de la Forme du DataFrame :

Fonction pour Afficher les Dimensions d'un DataFrame

```
💡 Click here to ask Blackbox to help you code faster
def print_shape_df(df, ds_name="df"):
    print(f"{ds_name} dataset has {df.shape[0]} rows and {df.shape[1]} columns")

print_shape_df(df, "Chatbot")
```

Chatbot dataset has 232 rows and 2 columns

- **def print_shape_df(df, ds_name="df")** : Cette fonction prend en entrée un **DataFrame** et un **nom de dataset** (par défaut "df") et imprime la forme du DataFrame.
- **print(f"{ds_name} dataset has {df.shape[0]} rows and {df.shape[1]} columns")** : Utilisation de la fonction **print** pour afficher le nombre de lignes et de colonnes du DataFrame.
- **print_shape_df(df, "Chatbot")** : Appel de la fonction pour imprimer la forme du DataFrame "Chatbot"

Impression des Informations du DataFrame :

Fonction pour Afficher les Informations d'un DataFrame

```
💡 Click here to ask Blackbox to help you code faster
def print_dfInfo(df, ds_name="df"):
    print(f"The info of {ds_name} dataset\n")
    print(df.info())

print_dfInfo(df, "Chatbot")
```

- **def print_dfInfo(df, ds_name="df")** : Cette fonction prend en entrée un **DataFrame** et un **nom de dataset** (par défaut "df") et imprime les informations détaillées du DataFrame.
- **print(f"The info of {ds_name} dataset\n")** : Affiche une ligne de titre pour indiquer les informations du dataset.
- **print(df.info())** : Utilisation de la fonction **info()** de pandas pour afficher des informations détaillées sur le **DataFrame**, y compris le type de données, le nombre de valeurs non nulles, etc.
- **print_dfInfo(df, "Chatbot")** : Appel de la fonction pour imprimer les informations détaillées du DataFrame "Chatbot".

The info of Chatbot dataset

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 232 entries, 0 to 231
Data columns (total 2 columns):
 #   Column   Non-Null Count  Dtype  
--- 
 0   Pattern   232 non-null   object  
 1   Tag       232 non-null   object  
dtypes: object(2)
memory usage: 5.4+ KB
None
```

- Il y a un total de 232 entrées (lignes) dans le **DataFrame**, avec un index allant de 0 à 231.
- Il y a un total de 2 colonnes dans le **DataFrame**.
 - 0 Pattern 232 **non-null object** : La première colonne nommée "Pattern" contient 232 valeurs non nulles de type objet.
 - 1 Tag 232 **non-null object** : La deuxième colonne nommée "Tag" contient également 232 valeurs non nulles de type objet.
- dtypes: object(2)** : Indique que les deux colonnes du **DataFrame** sont de type objet.
- memory usage: 5.4+ KB** : Le **DataFrame** utilise environ 5,4 kilo-octets (**KB**) de mémoire.
- None** : C'est le résultat renvoyé par la fonction **info()**, et il est affiché car la fonction imprime directement les informations plutôt que de les renvoyer.

Nombre de Classes dans le Dataset :

Fonction pour Compter le Nombre de Classes dans une Colonne

```
💡 Click here to ask Blackbox to help you code faster
def num_classes(df, target_col, ds_name="df"):
    print(f"The {ds_name} dataset has {len(df[target_col].unique())} classes")
num_classes(df, 'Tag', "Chatbot")
```

Python

The Chatbot dataset has 80 classes

- def num_classes(df, target_col, ds_name="df")** : Cette fonction prend en entrée un **DataFrame**, le nom de la colonne cible (ici, 'Tag'), et un nom de dataset (par défaut "df"). Elle imprime ensuite le nombre de classes uniques dans la colonne cible.
- print(f"The {ds_name} dataset has {len(df[target_col].unique())} classes")** : Affiche le nombre de classes uniques dans la colonne spécifiée.
- num_classes(df, 'Tag', "Chatbot")** : Appel de la fonction pour imprimer le nombre de classes uniques dans la colonne 'Tag' du DataFrame "Chatbot".

Vérification des Valeurs Nulles dans le Dataset :

Fonction pour Vérifier les Valeurs Nulles dans un DataFrame

```
💡 Click here to ask Blackbox to help you code faster
def check_null(df, ds_name='df'):
    print(f"Null Values in each col in the {ds_name} dataset:\n")
    print(df.isnull().sum())
check_null(df, "Chatbot")
```

Python

Null Values in each col in the Chatbot dataset:

```
Pattern    0
Tag       0
dtype: int64
```

Fonction de Tracé d'un Graphique à Barres :

```
def count_plot(x, df, title, xlabel, ylabel, width, height, order=None, rotation=False, palette='winter', hue=None):
    # Calcul du nombre total de données dans le DataFrame
    ncount = len(df)

    # Création de la figure avec la taille spécifiée
    plt.figure(figsize=(width, height))

    # Utilisation de Seaborn pour créer le graphique à barres
    ax = sns.countplot(x=x, data=df, palette=palette, order=order, hue=hue)

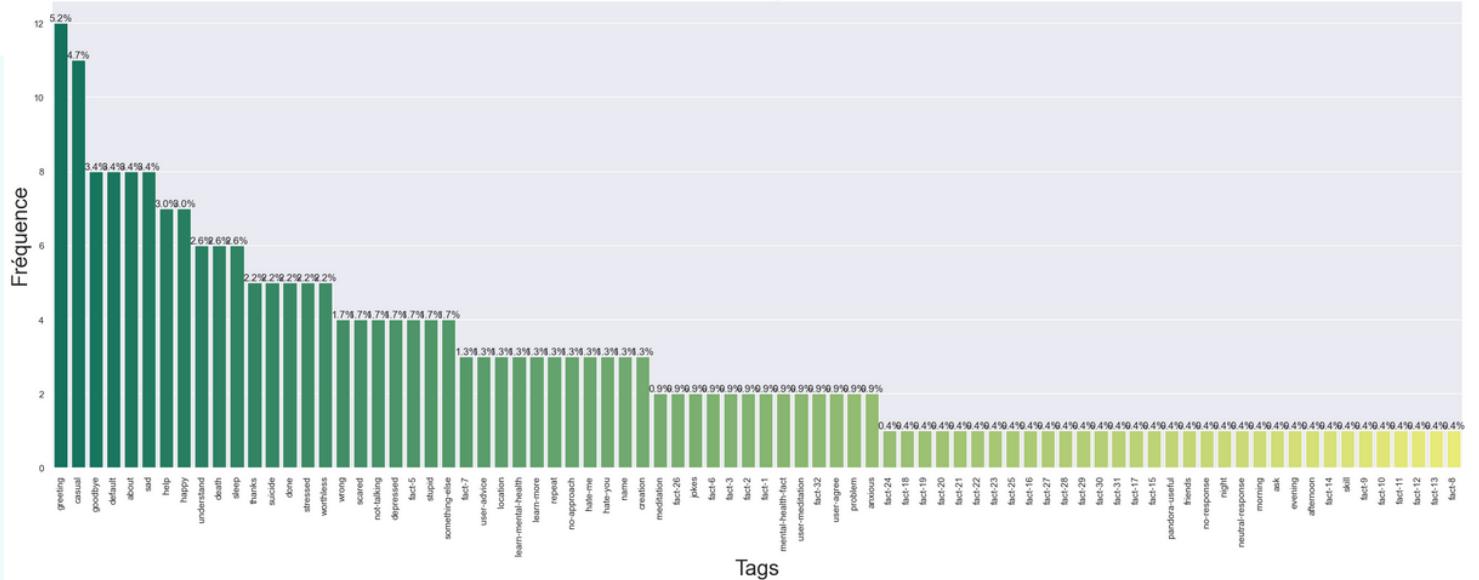
    # Ajout du titre et ajustement de la taille des étiquettes
    plt.title(title, fontsize=20)
    if rotation:
        plt.xticks(rotation='vertical')
    plt.xlabel(xlabel, fontsize=25)
    plt.ylabel(ylabel, fontsize=25)

    # Positionnement de l'étiquette de l'axe des y à gauche
    ax.yaxis.set_label_position('left')

    # Ajout d'annotations pour indiquer les pourcentages au-dessus des barres
    for p in ax.patches:
        x = p.get_bbox().get_points()[0, 0]
        y = p.get_bbox().get_points()[1, 1]
        ax.annotate(f'{:.1f}%'.format(100. * y / ncount), (x.mean(), y),
                    ha='center', va='bottom') # alignement du texte

    # Affichage du graphique
    plt.show()
```

- Cette fonction prend en entrée différentes informations pour créer un graphique à barres avec **Seaborn**.
 - **x** : Colonne à utiliser pour les valeurs sur l'axe des x.
 - **df** : DataFrame contenant les données.
 - **title** : Titre du graphique.
 - **xlabel** : Libellé de l'axe des x.
 - **ylabel** : Libellé de l'axe des y.
 - **width** et **height** : Dimensions de la figure.
 - **order** : Ordre spécifié des catégories sur l'axe des x.
 - **rotation** : Rotation des étiquettes sur l'axe des x.
 - **palette** : Palette de couleurs à utiliser.
 - **hue** : Colonne pour la séparation des données (optionnelle).
- **ncount = len(df)** : Calcul du nombre total de données dans le DataFrame.
- **plt.figure(figsize=(width, height))** : Création de la figure avec la taille spécifiée.
- **ax** : Utilisation de **Seaborn** pour créer un graphique à barres.
- **plt.title(title, fontsize=20)** : Ajout du titre au graphique.
- **plt.xlabel** : Ajout des étiquettes aux axes **x** et **y**.
- **ax.yaxis.set_label_position('left')** : Positionnement de l'étiquette de l'axe des y à gauche.
- **plt.show()** : Affichage du graphique à barres.
- **sns.set** : Cette ligne ajuste la taille de la police pour une meilleure lisibilité.
- **order** : Définition de l'ordre des catégories en fonction de leur fréquence.
- **count_plot(...)** : Appel de la fonction pour tracer le graphique à barres de la distribution des tags dans le DataFrame "Chatbot".



Prétraitement des Motifs :

Prétraitement du Texte avec NLTK Stemming

```
💡 Click here to ask Blackbox to help you code faster
# Initialisation du stemmer
stemmer = PorterStemmer()

# Liste des mots à ignorer lors du prétraitement
ignore_words = ['?', '!', ',', '.']

# Fonction pour prétraiter un motif en le convertissant en minuscules, en le tokenisant et en appliquant le stemming
def preprocess_pattern(pattern):
    words = word_tokenize(pattern.lower())
    stemmed_words = [stemmer.stem(word) for word in words if word not in ignore_words]
    return " ".join(stemmed_words)

# Application du prétraitement à la colonne 'Pattern' du DataFrame
df['Pattern'] = df['Pattern'].apply(preprocess_pattern)
```

Python

- **Initialisation du Stemmer :** Le code utilise le **stemmer de Porter (PorterStemmer)** pour réduire les mots à leur forme de base.
- **Liste des Mots à Ignorer :** Certains mots de ponctuation sont ignorés lors du prétraitement.
- **Fonction `preprocess_pattern(pattern)` :**
 - `words = word_tokenize(pattern.lower())` : Le motif est converti en minuscules, puis tokenisé en mots individuels.
 - `stemmed_words` : Chaque mot est soumis au **stemming** (réduction à sa forme de base) en utilisant le **stemmer** de Porter, à condition qu'il ne figure pas dans la liste des mots à ignorer.
 - `return " ".join(stemmed_words)` : Les mots après **stemming** sont rassemblés en une seule chaîne de caractères.
- **Application du Prétraitement à la Colonne 'Pattern' :**
 - `df['Pattern']` : La fonction **preprocess_pattern** est appliquée à chaque élément de la colonne 'Pattern' du **DataFrame**.

	Pattern	Tag
0	hi	greeting
1	hey	greeting
2	is anyone there	greeting
3	hi there	greeting
4	hello	greeting

Création d'un WordCloud :

Visualisation du Nuage de Mots avec WordCloud

```
💡 Click here to ask Blackbox to help you code faster
# Création d'une figure avec une taille spécifiée
plt.figure(figsize=(20, 20))

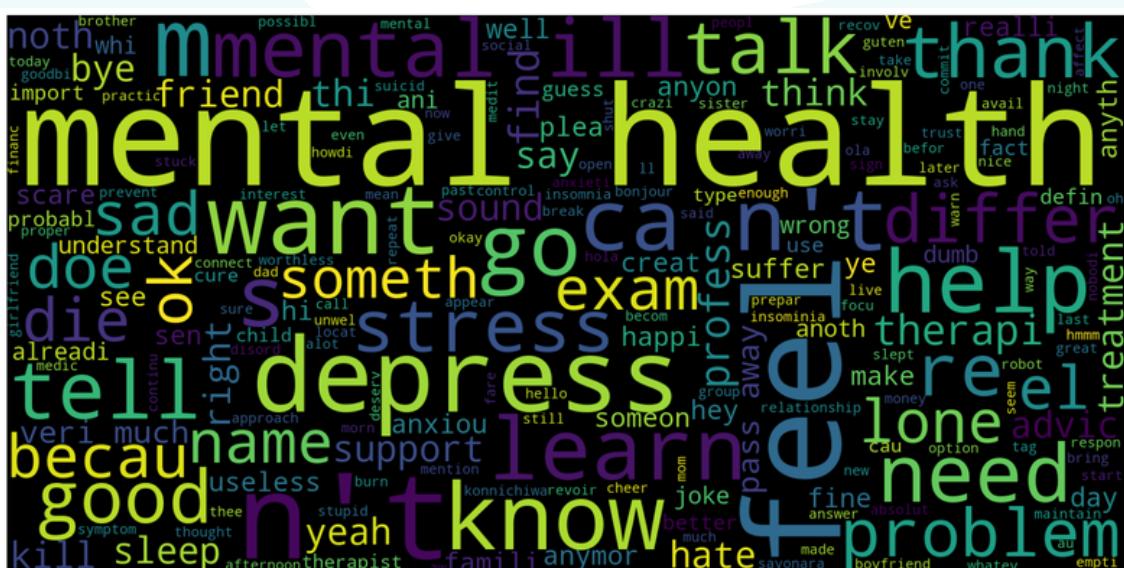
# Génération du WordCloud à partir des motifs dans la colonne 'Pattern'
wc = WordCloud(max_words=2000, width=1600, height=800, stopwords=STOPWORDS).generate(' '.join(df['Pattern']))

# Affichage du WordCloud avec interpolation bilinéaire
plt.imshow(wc, interpolation='bilinear')

# Désactivation des étiquettes sur les axes
plt.xticks([])
plt.yticks([])

# Affichage du WordCloud généré
plt.show()
```

- `plt.figure(figsize=(20, 20))` : Création d'une figure avec une taille spécifiée de 20x20 pouces.
 - `wc` : Génération du `WordCloud` à partir des motifs dans la colonne 'Pattern' du `DataFrame`. Le `WordCloud` peut afficher jusqu'à 2000 mots et a une taille de 1600x800 pixels.
 - `plt.imshow(wc, interpolation='bilinear')` : Affichage du `WordCloud` avec une interpolation bilinéaire pour une meilleure qualité visuelle.
 - Désactivation des étiquettes sur les axes :
 - `plt.xticks([])` et `plt.yticks([])` : Désactivation des étiquettes sur les axes `x` et `y` pour une présentation plus propre.
 - `plt.show()` : Affichage du `WordCloud` généré.



Extraction du Corpus :

Extraction du Corpus à partir de la Colonne 'Pattern'

```
💡 Click here to ask Blackbox to help you code faster
def get_corpus(series):
    words = []
    for text in series:
        for word in text.split():
            words.append(word.strip())
    return words

# Appel de la fonction pour extraire le corpus de la colonne 'Pattern'
corpus = get_corpus(df.Pattern)

# Affichage des cinq premiers éléments du corpus
corpus[:5]
```

Python

```
['hi', 'hey', 'is', 'anyon', 'there']
```



```
💡 Click here to ask Blackbox to help you code faster
# Affichage du nombre total de mots dans le corpus
print(f"Le dataset contient {len(corpus)} mots")
```

Python

```
Le dataset contient 1075 mots
```

- **def get_corpus(series)** : Cette fonction prend une série (colonne) en entrée et extrait un corpus en séparant chaque mot.
- **for text in series** : La fonction itère à travers chaque élément de la série.
- **for word in text.split()** : Chaque texte est divisé en mots, et la fonction itère à travers chaque mot.
- **words.append(word.strip())** : Chaque mot est ajouté à la liste **words** après avoir été débarrassé des espaces indésirables.
- **return words** : La fonction renvoie la liste de mots du corpus.
- **corpus = get_corpus(df.Pattern)** : Appel de la fonction pour extraire le corpus de la colonne 'Pattern' du DataFrame.
- **corpus[:5]** : Affichage des cinq premiers éléments du corpus.
- **print(f"Le dataset contient {len(corpus)} mots")** : Affichage du nombre total de mots dans le corpus.

Comptage des Mots les Plus Fréquents :

Identification des Mots les Plus Fréquents dans le Corpus

```
💡 Click here to ask Blackbox to help you code faster
from collections import Counter

# Utilisation de Counter pour compter les occurrences de chaque mot dans le corpus
counter = Counter(corpus)

# Extraction des dix mots les plus fréquents sous forme de dictionnaire
most_common = dict(counter.most_common(10))

# Affichage des mots les plus fréquents
most_common
```

Python

```
{'i': 107,
 'you': 43,
 'what': 32,
 'mental': 26,
 'do': 25,
 'to': 24,
 "n't": 21,
 'about': 20,
 'me': 19,
 'health': 19}
```

- Importation de la classe **Counter** de la bibliothèque `collections`.
- `counter = Counter(corpus)` : Utilisation de **Counter** pour compter les occurrences de chaque mot dans le corpus.
- `most_common = dict(counter.most_common(10))` : Extraction des dix mots les plus fréquents sous forme de dictionnaire.
- `most_common` : Affichage des mots les plus fréquents.

Extraction des Libellés Uniques :

Extraction des Libellés Uniques de la Colonne 'Tag'

```
💡 Click here to ask Blackbox to help you code faster
# Extraction des libellés uniques de la colonne 'Tag' du DataFrame
labels = df2['Tag'].unique().tolist()

# Nettoyage des espaces autour des libellés
labels = [s.strip() for s in labels]

# Affichage des libellés après nettoyage
labels
```

Python

```
['greeting',
'morning',
'afternoon',
'evening',
'night',
'goodbye',
'thanks',
'no-response',
'neutral-response',
'about',
'skill',
'creation',
'name',
'help',
'sad',
```

- `labels = df2['Tag'].unique().tolist()` : Extraction des libellés uniques de la colonne 'Tag' du DataFrame et conversion en liste.
- `labels = [s.strip() for s in labels]` : Nettoyage des espaces autour des libellés en utilisant une compréhension de liste pour appliquer la méthode `strip()` à chaque libellé.
- `labels` : Affichage des libellés après nettoyage.

Calcul du Nombre Total de Libellés et Création de Mappings:

Création de Mappings entre Identifiants et Libellés

```
💡 Click here to ask Blackbox to help you code faster
# Calcul du nombre total de libellés
num_labels = len(labels)

# Création d'un mapping d'identifiants vers libellés
id2label = {id: label for id, label in enumerate(labels)}

# Création d'un mapping de libellés vers identifiants
label2id = {label: id for id, label in enumerate(labels)}
```

✓ 0.0s Python

- `num_labels` : Calcul du nombre total de libellés en utilisant la fonction `len()`.
- `id2label = {id: label for id, label in enumerate(labels)}` : Création d'un mapping d'identifiants vers libellés en utilisant la fonction `enumerate()` pour obtenir les paires (`identifiant, libellé`).
- `label2id = {label: id for id, label in enumerate(labels)}` : Création d'un mapping de libellés vers identifiants, similaire à `id2label`.

id2label :

```
💡 Click here to ask Blackbox to help you code faster
id2label
✓ 0.0s
Python
{0: 'greeting',
 1: 'morning',
 2: 'afternoon',
 3: 'evening',
 4: 'night',
 5: 'goodbye',
 6: 'thanks',
 7: 'no-response',
 8: 'neutral-response',
 9: 'about',
 10: 'skill',
 11: 'creation',
 12: 'name',
 13: 'help',
 14: 'sad',
 15: 'stressed',
 16: 'worthless',
 17: 'depressed',
 18: 'happy',
 19: 'casual',
 20: 'anxious',
 21: 'not-talking',
 22: 'sleep',
 23: 'scared',
 24: 'death',
 ...
}
```

label2id :

```
💡 Click here to ask Blackbox to help you code faster
label2id
✓ 0.0s
Python
{'greeting': 0,
 'morning': 1,
 'afternoon': 2,
 'evening': 3,
 'night': 4,
 'goodbye': 5,
 'thanks': 6,
 'no-response': 7,
 'neutral-response': 8,
 'about': 9,
 'skill': 10,
 'creation': 11,
 'name': 12,
 'help': 13,
 'sad': 14,
 'stressed': 15,
 'worthless': 16,
 'depressed': 17,
 'happy': 18,
 'casual': 19,
 'anxious': 20,
 'not-talking': 21,
 'sleep': 22,
 'scared': 23,
 'death': 24,}
```

Ajout de la Colonne 'labels' en Utilisant le Mapping :

Ajout de la Colonne 'labels' basée sur le Mapping Label2Id

```
💡 Click here to ask Blackbox to help you code faster
# Utilisation de la fonction map pour ajouter la colonne 'labels' en utilisant le mapping label2id
df2['labels'] = df2['Tag'].map(lambda x: label2id[x.strip()])

# Affichage des cinq premières lignes du DataFrame après l'ajout de la colonne 'labels'
df2.head()
```

Python

- **df2['labels']** : Utilisation de la fonction **map** pour ajouter la colonne **'labels'** au **DataFrame** en utilisant le **mapping label2id**. La fonction **lambda** est utilisée pour appliquer la transformation à chaque élément de la colonne **'Tag'** (nettoyage des espaces avec **strip()**).
- **df2.head()** : Affichage des cinq premières lignes du **DataFrame** après l'ajout de la colonne **'labels'**.

	Pattern	Tag	labels
0	Hi	greeting	0
1	Hey	greeting	0
2	Is anyone there?	greeting	0
3	Hi there	greeting	0
4	Hello	greeting	0

Extraction des Motifs et des Étiquettes :

Extraction de la Liste des Motifs (X)

```
💡 Click here to ask Blackbox to help you code faster
# Extraction de la liste des motifs à partir de la colonne 'Pattern'
X = list(df2['Pattern'])

# Affichage des cinq premiers motifs dans la liste
X[:5]
```

Python

['Hi', 'Hey', 'Is anyone there?', 'Hi there', 'Hello']

Extraction de la Liste des Étiquettes (y)

```
💡 Click here to ask Blackbox to help you code faster
# Extraction de la liste des étiquettes à partir de la colonne 'labels'
y = list(df2['labels'])

# Affichage des cinq premières étiquettes dans la liste
y[:5]
```

Python

[0, 0, 0, 0, 0]

- **X = list(df2['Pattern'])** : Extraction de la liste des motifs à partir de la colonne **'Pattern'** du **DataFrame**.
- **X[:5]** : Affichage des cinq premiers motifs dans la liste.
- **y = list(df2['labels'])** : Extraction de la liste des étiquettes à partir de la colonne **'labels'** du **DataFrame**.
- **y[:5]** : Affichage des cinq premières étiquettes dans la liste.

Division des Données en Ensembles d'Entraînement et de Test :

Division des Données en Ensembles d'Entraînement et de Test

```
💡 Click here to ask Blackbox to help you code faster
# Division des données en ensembles d'entraînement et de test
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=123)
```

Python

- **train_test_split()** : Cette fonction de **scikit-learn** est utilisée pour diviser les données en ensembles **d'entraînement** et de **test**. Les paramètres **X** et **y** sont les motifs et les étiquettes respectivement. Le paramètre **random_state** est utilisé pour garantir la reproductibilité des résultats.
- **X_train, X_test, y_train, y_test** : Les quatre variables contiennent les données d'entraînement (**X_train, y_train**) et de test (**X_test, y_test**).

Utilisation de BERT pour la Classification de Séquences :

Configuration du Modèle BERT pour la Classification de Séquences

```
💡 Click here to ask Blackbox to help you code faster
from transformers import BertTokenizer, BertForSequenceClassification

# Spécification du modèle BERT pré-entraîné à utiliser
model_name = "bert-base-uncased"

# Définition de la longueur maximale des séquences d'entrée
max_len = 256

# Initialisation du tokenizer BERT
tokenizer = BertTokenizer.from_pretrained(model_name, max_length=max_len)

# Chargement du modèle BERT pour la classification de séquences
model = BertForSequenceClassification.from_pretrained(model_name,
                                                       num_labels=num_labels,
                                                       id2label=id2label,
                                                       label2id=label2id)
```

Python

- Importation du **tokenizer** et du modèle **BERT** spécifiquement conçu pour la classification de séquences.
- **model_name = "bert-base-uncased"** : Spécification du modèle **BERT** pré-entraîné à utiliser. Ici, "**bert-base-uncased**" indique le modèle **BERT** de base sans casse.
- **max_len = 256** : Définition de la longueur maximale des séquences d'entrée. Cela sera important lors du prétraitement des données.
- **tokenizer** : Initialisation du **tokenizer BERT** à partir du modèle pré-entraîné spécifié et avec la longueur maximale définie.
- **model** : Chargement du modèle BERT pré-entraîné pour la classification de séquences. Les paramètres **num_labels**, **id2label**, et **label2id** sont utilisés pour adapter le modèle à notre tâche spécifique avec les libellés définis précédemment.

Encodage des Données avec le Tokenizer BERT :

Encodage des Données d'Entraînement et de Test avec le Tokenizer BERT

```
💡 Click here to ask Blackbox to help you code faster
# Encodage des données d'entraînement avec le tokenizer BERT
train_encoding = tokenizer(X_train, truncation=True, padding=True)

# Encodage des données de test avec le tokenizer BERT
test_encoding = tokenizer(X_test, truncation=True, padding=True)
```

Python

Encodage des Données Complètes avec le Tokenizer BERT

```
💡 Click here to ask Blackbox to help you code faster
# Encodage des données complètes avec le tokenizer BERT
full_data = tokenizer(X, truncation=True, padding=True)
```

Python

- **tokenizer(X_train, truncation=True, padding=True)** : Utilisation du tokenizer BERT pour encoder les données d'entraînement (X_train). Les paramètres **truncation=True** et **padding=True** sont utilisés pour tronquer et rembourrer les séquences selon la longueur maximale définie.
- **tokenizer(X_test, truncation=True, padding=True)** : Encodage des données de test (X_test) de la même manière que les données d'entraînement.
- **tokenizer(X, truncation=True, padding=True)** : Encodage des données complètes (X) avec le tokenizer BERT. Cela peut être utile lors de la visualisation des **embeddings** ou d'autres analyses.

Classe DataLoader pour PyTorch :

```
💡 Click here to ask Blackbox to help you code faster
class DataLoader(Dataset):

    def __init__(self, encodings, labels):
        #Initialisation de la classe DataLoader.
        self.encodings = encodings
        self.labels = labels

    def __getitem__(self, idx):
        #Obtention d'un élément du jeu de données.
        item = {key: torch.tensor(val[idx]) for key, val in self.encodings.items()}
        item['labels'] = torch.tensor(self.labels[idx])
        return item

    def __len__(self):
        #Obtention de la longueur du jeu de données.
        return len(self.labels)
```

Python

- **__init__(self, encodings, labels)** : Le constructeur de la classe prend les encodages et les étiquettes en tant que paramètres et initialise les attributs de la classe.
- **__getitem__(self, idx)** : Méthode spéciale pour obtenir un élément du jeu de données à l'index spécifié (idx). Elle renvoie un dictionnaire contenant les encodages et les étiquettes pour l'élément correspondant.
- **__len__(self)** : Méthode spéciale pour obtenir la longueur du jeu de données, c'est-à-dire le nombre total d'éléments dans le jeu de données.

Création des DataLoader pour l'Entraînement et les Tests :

Création des DataLoader pour l'Entraînement et les Tests

```
💡 Click here to ask Blackbox to help you code faster
# Création des DataLoader pour l'entraînement et les tests
train_dataloader = DataLoader(train_encoding, y_train)
test_dataloader = DataLoader(test_encoding, y_test)
```

Python

Création du DataLoader pour l'Ensemble de Données Complet

```
💡 Click here to ask Blackbox to help you code faster
# Création du DataLoader pour l'ensemble de données complet
fullDataLoader = DataLoader(full_data, y_test)
```

Python

- **train_dataloader** : Création d'un DataLoader pour l'ensemble d'entraînement avec les encodages **train_encoding** et les étiquettes **y_train**.
- **test_dataloader** : Création d'un DataLoader pour l'ensemble de test avec les encodages **test_encoding** et les étiquettes **y_test**.
- **fullDataLoader** : Création d'un DataLoader pour l'ensemble de données complet avec les encodages **full_data** et les étiquettes **y_test**.

Fonction pour Calculer les Métriques :

Fonction pour Calculer les Métriques

```
💡 Click here to ask Blackbox to help you code faster
def compute_metrics(pred):
    # Calcul des métriques d'évaluation (précision, rappel, F1, exactitude) à partir des prédictions.
    labels = pred.label_ids
    preds = pred.predictions.argmax(-1)
    precision, recall, f1, _ = precision_recall_fscore_support(labels, preds, average='macro')
    acc = accuracy_score(labels, preds)

    return {
        'Accuracy': acc,
        'F1': f1,
        'Precision': precision,
        'Recall': recall
    }
```

Python

- **pred.label_ids** : Les vraies étiquettes dans les prédictions.
- **pred.predictions.argmax(-1)** : Les prédictions du modèle, obtenues en prenant l'indice de la classe avec la probabilité maximale.
- **precision_recall_fscore_support(labels, preds, average='macro')** : Calcul des précisions, rappels, et F1-scores pour chaque classe, puis calcul de la moyenne pondérée (moyenne macro).
- **accuracy_score(labels, preds)** : Calcul de l'exactitude en comparant les vraies étiquettes aux prédictions.
- **return {...}** : Retourne un dictionnaire contenant les métriques calculées (exactitude, F1, précision, rappel).

Configuration des Paramètres d'Entraînement :

Configuration des Arguments d'Entraînement

```
💡 Click here to ask Blackbox to help you code faster
training_args = TrainingArguments(
    output_dir='./output',
    do_train=True,
    do_eval=True,
    num_train_epochs=100,
    per_device_train_batch_size=32,
    per_device_eval_batch_size=16,
    warmup_steps=100,
    weight_decay=0.05,
    logging_strategy='steps',
    logging_dir='./multi-class-logs',
    logging_steps=50,
    evaluation_strategy="steps",
    eval_steps=50,
    save_strategy="steps",
    load_best_model_at_end=True
)
```

Python

- **output_dir='./output'** : Répertoire où les modèles entraînés seront sauvegardés.
- **do_train=True** : Indique si l'entraînement doit être effectué.
- **do_eval=True** : Indique si l'évaluation doit être effectuée.
- **num_train_epochs=100** : Nombre total d'époques d'entraînement.
- **per_device_train_batch_size=32** : Taille du lot par périphérique pour l'entraînement.
- **per_device_eval_batch_size=16** : Taille du lot par périphérique pour l'évaluation.
- **warmup_steps=100** : Nombre d'étapes pour le réchauffement de l'optimiseur.
- **weight_decay=0.05** : Terme de décroissance du poids pour éviter le surajustement.
- **logging_strategy='steps'** : Stratégie de journalisation basée sur le nombre d'étapes.
- **logging_dir='./multi-class-logs'** : Répertoire où les journaux seront sauvegardés.
- **logging_steps=50** : Fréquence de journalisation des étapes.
- **evaluation_strategy="steps"** : Stratégie d'évaluation basée sur le nombre d'étapes.
- **eval_steps=50** : Fréquence d'évaluation des étapes.
- **save_strategy="steps"** : Stratégie de sauvegarde basée sur le nombre d'étapes.
- **load_best_model_at_end=True** : Charge le meilleur modèle à la fin de l'entraînement.

Configuration du Trainer pour l'Entraînement :

Configuration du Trainer pour l'Entraînement

```
💡 Click here to ask Blackbox to help you code faster
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataloader,
    eval_dataset=test_dataloader,
    compute_metrics=compute_metrics
)
```

Python

- **model=model** : Le modèle que le Trainer utilisera pour l'entraînement.
- **args=training_args** : Les arguments d'entraînement configurés précédemment.
- **train_dataset=train_dataloader** : Le DataLoader contenant les données d'entraînement.
- **eval_dataset=test_dataloader** : Le DataLoader contenant les données de test.
- **compute_metrics=compute_metrics** : La fonction pour calculer les métriques d'évaluation.

Entraînement du Modèle avec le Trainer :

```
💡 Click here to ask Blackbox to help you code faster
# Entraînement du modèle en utilisant le Trainer
trainer.train()

8% | 50/600 [05:35<55:46, 6.09s/it]
{'loss': 4.2193, 'learning_rate': 2.5e-05, 'epoch': 8.33}

8% | 50/600 [05:37<55:46, 6.09s/it]
{'eval_loss': 4.061669826507568, 'eval_Accuracy': 0.15517241379310345, 'eval_F1': 0.10022983552395318, 'eval_Precision': 0.09523809523809523
17% | 100/600 [11:20<52:18, 6.28s/it]
{'loss': 3.3349, 'learning_rate': 5e-05, 'epoch': 16.67}

17% | 100/600 [11:23<52:18, 6.28s/it]
{'eval_loss': 3.4604804515838623, 'eval_Accuracy': 0.27586206896551724, 'eval_F1': 0.1521856766421984, 'eval_Precision': 0.15006503158677073
25% | 150/600 [16:24<41:43, 5.56s/it]
{'loss': 1.969, 'learning_rate': 4.5e-05, 'epoch': 25.0}

25% | 150/600 [16:27<41:43, 5.56s/it]
{'eval_loss': 2.8732590675354004, 'eval_Accuracy': 0.3620689655172414, 'eval_F1': 0.21142266824085007, 'eval_Precision': 0.21306818181818182
33% | 200/600 [21:39<40:25, 6.06s/it]
{'loss': 1.0454, 'learning_rate': 4e-05, 'epoch': 33.33}

33% | 200/600 [21:41<40:25, 6.06s/it]
{'eval_loss': 2.6497275829315186, 'eval_Accuracy': 0.4482758620689655, 'eval_F1': 0.2828960155490768, 'eval_Precision': 0.27575315840621967,
42% | 250/600 [26:47<36:30, 6.26s/it]
{'loss': 0.5757, 'learning_rate': 3.5e-05, 'epoch': 41.67}
```

Évaluation du Modèle sur les Ensembles d'Entraînement et de Test :

```
💡 Click here to ask Blackbox to help you code faster
# Évaluation du modèle sur les ensembles d'entraînement et de test
evaluation_results = [trainer.evaluate(eval_dataset=dataset) for dataset in [train_dataloader, test_dataloader]]

# Création d'un DataFrame avec les résultats d'évaluation
evaluation_df = pd.DataFrame(evaluation_results, index=["train", "test"]).iloc[:, :5]
```

- `trainer.evaluate(eval_dataset=dataset)` : Utilisation du Trainer pour évaluer le modèle sur un ensemble de données spécifique (entraînement ou test).
- `[trainer.evaluate(eval_dataset=dataset) for dataset in [train_dataloader, test_dataloader]]` : Création d'une liste contenant les résultats d'évaluation pour les ensembles d'entraînement et de test.
- `pd.DataFrame(evaluation_results, index=["train", "test"]).iloc[:, :5]` : Création d'un DataFrame à partir des résultats d'évaluation avec les métriques telles que l'exactitude, la F1, la précision, le rappel, et la perte.

```
100% | 11/11 [00:12<00:00, 1.15s/it]
100% | 4/4 [00:01<00:00, 2.19it/s]

  eval_loss  eval_Accuracy  eval_F1  eval_Precision  eval_Recall
train  0.059699        1.000000  1.000000        1.000000  1.000000
test   2.664119        0.482759  0.335218        0.328125  0.376736
```

Les résultats d'évaluation du modèle semblent indiquer une très bonne performance sur l'ensemble d'entraînement, avec une **exactitude**, une **précision**, un **rappel** et un **score F1** tous égaux à 1. Cela suggère que le modèle a appris parfaitement les exemples d'entraînement et peut les reproduire avec précision.

Cependant, sur l'ensemble de **test**, les performances sont moins optimales. L'**exactitude** est d'environ 48%, ce qui indique que le modèle ne généralise pas aussi bien sur des données qu'il n'a pas vu pendant l'entraînement. Les **scores F1**, **précision** et **rappel** sont également inférieurs, montrant que le modèle a du mal à généraliser correctement.

ESauvegarde du Modèle BERT et du Tokenizer, Chargement du Modèle Re-entraîné

sauvegarder le modèle BERT

```
💡 Click here to ask Blackbox to help you code faster
# Chemin où sauvegarder le modèle BERT
model_path_BERT = "chatbot"

# Sauvegarde du modèle et du tokenizer
trainer.save_model(model_path_BERT)
tokenizer.save_pretrained(model_path_BERT)

# Chargement du modèle re-entraîné
retrained_model = BertForSequenceClassification.from_pretrained(model_path_BERT, num_labels=num_labels, id2label=id2label, label2id=label2id)
```

Python

- **model_path_BERT = "chatbot"** : Spécification du chemin où sauvegarder le modèle BERT.
- **trainer.save_model(model_path_BERT)** : Sauvegarde du modèle entraîné à l'emplacement spécifié.
- **tokenizer.save_pretrained(model_path_BERT)** : Sauvegarde du tokenizer associé au modèle.
- **retrained_model** : Chargement du modèle re-entraîné à partir du chemin spécifié, avec les mêmes configurations que l'entraînement initial.

Chargement du Modèle BERT pour le Chatbot:

chargement du modèle

```
💡 Click here to ask Blackbox to help you code faster
# Chemin où le modèle BERT a été sauvegardé
model_path = "chatbot"

# Chargement du modèle et du tokenizer pour l'utilisation dans le chatbot
model = BertForSequenceClassification.from_pretrained(model_path)
tokenizer = BertTokenizerFast.from_pretrained(model_path)

# Configuration du pipeline pour l'analyse de sentiment avec le modèle BERT
chatbot = pipeline("sentiment-analysis", model=model, tokenizer=tokenizer)
```

Python

- **model_path = "chatbot"** : Spécification du chemin où le modèle BERT a été sauvegardé.
- **BertForSequenceClassification.from_pretrained(model_path)** : Chargement du modèle BERT entraîné à partir du chemin spécifié.
- **BertTokenizerFast.from_pretrained(model_path)** : Chargement du tokenizer associé au modèle BERT.
- **pipeline("sentiment-analysis", model=model, tokenizer=tokenizer)** : Configuration d'un pipeline pour l'analyse de sentiment utilisant le modèle BERT chargé. Cela te permettra d'utiliser le modèle directement pour effectuer des prédictions d'analyse de sentiment.

```
[{'label': 'greeting', 'score': 0.9764207005500793}]
```

Il semble que la phrase "Hello" ait été classée comme une salutation (**greeting**) avec une forte confiance, comme indiqué par le score élevé de **0.9764**. Cela suggère que le modèle **BERT** a correctement identifié la nature de la phrase comme étant une salutation.

Fonction de Chat Interactive avec le Chatbot BERT :

```
def chat(chatbot):  
  
    print("Chatbot: Hi! I am your virtual assistance, Feel free to ask, and I'll do my best to provide you with answers and assistance..")  
    print("Type 'quit' to exit the chat\\n\\n")  
  
    text = input("User: ").strip().lower()  
  
    while(text != 'quit'):  
        # Obtention du score de confiance pour l'analyse de sentiment  
        score = chatbot(text)[0]['score']  
  
        # Vérification du score de confiance  
        if score < 0.8:  
            print("Chatbot: Sorry, I can't answer that confidently.\\n")  
            text = input("User: ").strip().lower()  
            continue  
  
        # Obtention de l'étiquette prédite  
        label = label2id[chatbot(text)[0]['label']]  
        # Sélection d'une réponse aléatoire parmi les réponses associées à l'étiquette  
        response = random.choice(intents['intents'][label]['responses'])  
  
        # Affichage de la réponse du chatbot  
        print(f"Chatbot: {response}\\n")  
  
        # Attente de la prochaine entrée utilisateur  
        text = input("User: ").strip().lower()
```

• **Message de Bienvenue :**

- La fonction commence par afficher un message de bienvenue, invitant l'utilisateur à poser des questions ou demander de l'aide.

• **Entrée Utilisateur :**

- L'utilisateur est invité à entrer un message via la fonction **input**.

• **Boucle de Chat :**

- La boucle **while** est utilisée pour maintenir la conversation tant que l'utilisateur ne saisit pas "**quit**".

• **Analyse de Sentiment :**

- Le modèle BERT analyse le sentiment du message utilisateur et retourne un score de confiance.

• **Réponse du Chatbot :**

- Si le score de confiance est inférieur à **0.8**, le **chatbot** indique qu'il ne peut pas répondre avec confiance.
- Si le score est suffisamment élevé, le **chatbot** choisit une réponse aléatoire associée à l'étiquette prédite.

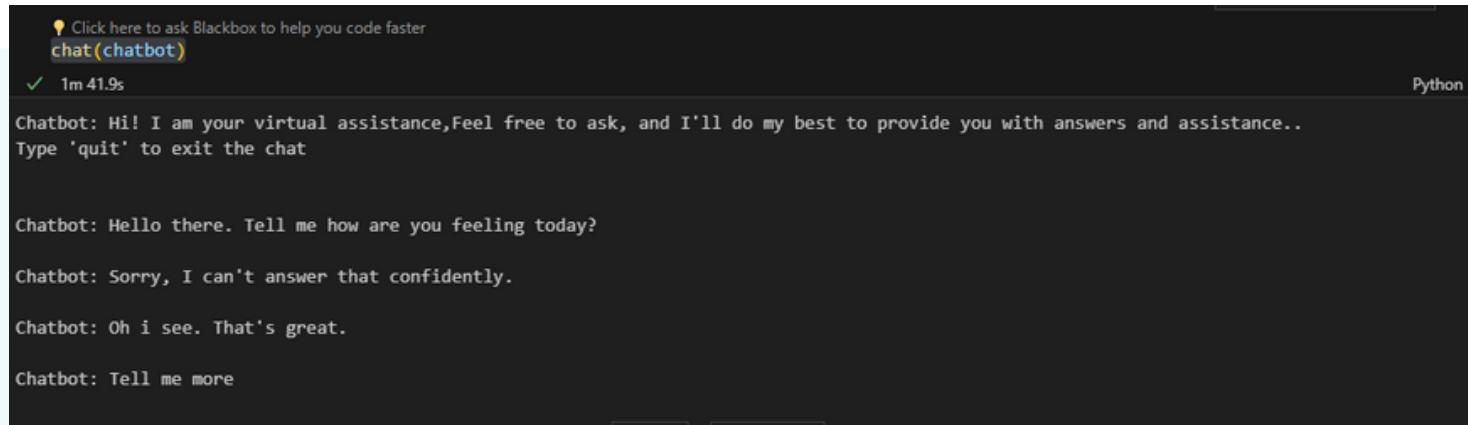
• **Nouvelle Entrée Utilisateur :**

- L'utilisateur est invité à saisir un nouveau message à la fin de chaque échange.

• **Fin de la Conversation :**

- La conversation se termine lorsque l'utilisateur entre "**quit**".

Test du Chatbot avec l'Interaction Utilisateur :



```
💡 Click here to ask Blackbox to help you code faster
chat(chatbot)
✓ 1m 41.9s
Python

Chatbot: Hi! I am your virtual assistance,Feel free to ask, and I'll do my best to provide you with answers and assistance..
Type 'quit' to exit the chat

Chatbot: Hello there. Tell me how are you feeling today?

Chatbot: Sorry, I can't answer that confidently.

Chatbot: Oh i see. That's great.

Chatbot: Tell me more
```

Il semble que le **chatbot** a répondu de manière appropriée à la première entrée, en demandant à l'utilisateur comment il se sent aujourd'hui. Cependant, il a ensuite indiqué qu'il ne pouvait pas répondre avec confiance à la deuxième entrée. Cela peut indiquer un manque de confiance dans la réponse du modèle pour certains types d'entrées. Enfin, le **chatbot** a répondu de manière générale à la troisième entrée, En demandant de continuer à partager davantage d'informations.

Fonction de Génération de Question Simple avec GPT-2 :

```
# Extraire les motifs (patterns) de toutes les intentions
all_patterns = [pattern for intent in intents_data['intents'] for pattern in intent['patterns']]

# Charger le modèle GPT-2 pré-entraîné
gpt2_model = GPT2LMHeadModel.from_pretrained("gpt2")
gpt2_tokenizer = GPT2Tokenizer.from_pretrained("gpt2")

def generate_simple_gpt2_question():
    random_pattern = random.choice(all_patterns)

    # Ajouter "Generate a question:" avant le motif
    input_text = f"Generate a question: {random_pattern}"

    # Encoder le texte
    input_ids = gpt2_tokenizer.encode(input_text, return_tensors="pt", max_length=512)

    # Générer la séquence
    output = gpt2_model.generate(input_ids, max_length=150, num_beams=5, no_repeat_ngram_size=2, top_k=50, top_p=0.95, temperature=0.7)

    # Décoder la séquence
    generated_sequence = gpt2_tokenizer.decode(output[0], skip_special_tokens=True)

    # Extraire la question terminée après un point d'interrogation, un point, ou entre guillemets
    match = re.search(r"([.!?]|\"[^\"]*\")", generated_sequence)
    if match:
        generated_question = match.group()
    else:
        generated_question = generated_sequence

    return generated_question
```

Chargement du fichier intents.json :

Le code commence par charger le fichier `intents.json`, qui contient les intentions et les motifs associés utilisés pour la génération de question.

Extraction des motifs :

Tous les motifs (patterns) des intentions sont extraits du fichier `intents.json`.

Chargement du modèle GPT-2 :

Le modèle GPT-2 pré-entraîné est chargé à l'aide de la classe `GPT2LMHeadModel`.

Génération de la Question :

Une fonction `generate_simple_gpt2_question` est définie pour générer une question simple en utilisant le modèle GPT-2.

Un motif aléatoire est sélectionné parmi tous les motifs disponibles dans le fichier `intents.json`.

Ce motif est utilisé comme entrée pour générer une séquence de texte avec GPT-2.

La séquence générée est découpée pour extraire la question terminée après un point d'interrogation, un point, ou entre guillemets.

La question générée est renvoyée.

Utilisation de la Fonction :

La fonction `generate_simple_gpt2_question` est utilisée pour générer une question simple.

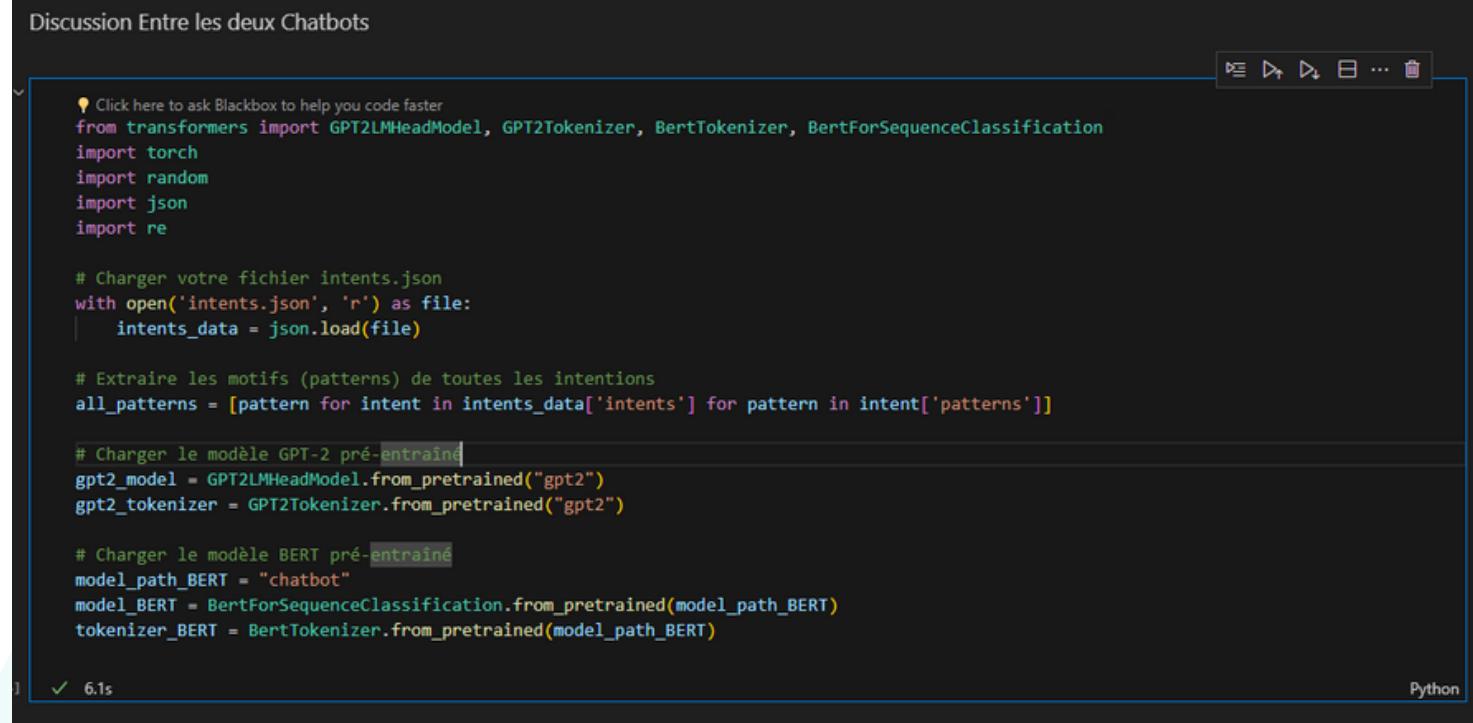
Affichage de la Question Générée :

La question générée est affichée à la fin.

```
Truncation was not explicitly activated but `max_length` is provided a specific value, please use `truncation=True` to explicitly truncate e
The attention mask and the pad token id were not set. As a consequence, you may observe unexpected behavior. Please pass your input's `atten
Setting `pad_token_id` to `eos_token_id` :50256 for open-end generation.
Question générée: Generate a question: I can't do anything.
```

Chargement des Modèles GPT-2 et BERT pour le Chatbot :

Discussion Entre les deux Chatbots



```
Click here to ask Blackbox to help you code faster
from transformers import GPT2LMHeadModel, GPT2Tokenizer, BertTokenizer, BertForSequenceClassification
import torch
import random
import json
import re

# Charger votre fichier intents.json
with open('intents.json', 'r') as file:
    intents_data = json.load(file)

# Extraire les motifs (patterns) de toutes les intentions
all_patterns = [pattern for intent in intents_data['intents'] for pattern in intent['patterns']]

# Charger le modèle GPT-2 pré-entraîné
gpt2_model = GPT2LMHeadModel.from_pretrained("gpt2")
gpt2_tokenizer = GPT2Tokenizer.from_pretrained("gpt2")

# Charger le modèle BERT pré-entraîné
model_path_BERT = "chatbot"
model_BERT = BertForSequenceClassification.from_pretrained(model_path_BERT)
tokenizer_BERT = BertTokenizer.from_pretrained(model_path_BERT)
```

- **Chargement du Fichier `intents.json` :**
 - Le code commence par charger le fichier `intents.json`, qui contient les intentions et les motifs associés utilisés pour la génération de question.
- **Extraction des Motifs :**
 - Tous les motifs (patterns) des intentions sont extraits du fichier `intents.json`.
- **Chargement du Modèle GPT-2 :**
 - Le modèle GPT-2 pré-entraîné est chargé à l'aide de la classe `GPT2LMHeadModel`.
 - Le tokenizer associé au modèle GPT-2 est également chargé à l'aide de la classe `GPT2Tokenizer`.

- **Chargement du Modèle BERT :**
- Le modèle BERT pré-entraîné pour la classification de séquences est chargé à partir du chemin spécifié.
- Le tokenizer associé au modèle BERT est également chargé à partir du même chemin.

Génération d'une Question Simple avec GPT-2 :

```
def generate_simple_gpt2_question():
    random_pattern = random.choice(all_patterns)

    # Ajouter "Generate a question:" avant le motif
    input_text = f" {random_pattern}"

    # Encoder le texte
    input_ids = gpt2_tokenizer.encode(input_text, return_tensors="pt", max_length=512)

    # Générer la séquence
    output = gpt2_model.generate(input_ids, max_length=150, num_beams=5, no_repeat_ngram_size=2, top_k=50, top_p=0.95, temperature=0.7)

    # Décoder la séquence
    generated_sequence = gpt2_tokenizer.decode(output[0], skip_special_tokens=True)

    # Extraire la question terminée après un point d'interrogation, un point, ou entre guillemets
    match = re.search(r"(.*[.!?])|(\\".*\\")", generated_sequence)
    if match:
        generated_question = match.group()
    else:
        generated_question = generated_sequence

    return generated_question
```

- **Fonction de Génération de Question Simple avec GPT-2 :**
 - La fonction `generate_simple_gpt2_question()` est définie pour générer une question simple en utilisant le modèle GPT-2.
- **Sélection Aléatoire d'un Motif :**
 - Un motif aléatoire est sélectionné parmi tous les motifs disponibles dans le fichier `intents.json`.
- **Préparation de l'Entrée Textuelle :**
 - Le motif sélectionné est utilisé pour former une entrée textuelle à laquelle est ajoutée la phrase "Generate a question:".
- **Encodage et Génération :**
 - Le texte encodé est passé au modèle GPT-2 pour générer une séquence de mots.
 - Plusieurs paramètres de génération sont spécifiés pour contrôler le processus de génération.
- **Décodage de la Séquence :**
 - La séquence générée est décodée à l'aide du tokenizer pour obtenir la question complète.
- **Extraction de la Question :**
 - La question est extraite de la séquence générée en recherchant un point d'interrogation, un point ou une phrase entre guillemets.

Génération d'une Réponse avec le Modèle BERT :

```
def generate_bert_response(question):
    encoding = tokenizer_BERT(question, return_tensors='pt', truncation=True, padding=True)

    # Passer le tensor d'entrée à travers le modèle BERT
    outputs = model_BERT(**encoding)

    # Récupérer la prédiction de classification
    logits = outputs.logits
    predicted_class = torch.argmax(logits, dim=1).item()

    # Utiliser la classe prédite pour obtenir une réponse aléatoire dans les intentions
    responses = intents_data['intents'][predicted_class]['responses']

    return random.choice(responses)
```

- **Fonction de Génération de Réponse avec BERT :**

- La fonction `generate_bert_response(question)` est définie pour générer une réponse en utilisant le modèle BERT.

- **Encodage de la Question :**

- La question donnée en entrée est encodée à l'aide du `tokenizer` associé au modèle BERT.

- **Passage à travers le Modèle BERT :**

- Le tensor encodé est passé à travers le modèle BERT pour obtenir les `logits` de sortie.

- **Prédiction de Classification :**

- Les `logits` sont utilisés pour prédire la classe à laquelle la question appartient.

- **Sélection Aléatoire d'une Réponse :**

- Les réponses associées à la classe prédite sont extraites à partir du fichier `intents.json`.
- Une réponse est choisie aléatoirement parmi les réponses disponibles pour cette classe.

Conversation entre deux Chatbots :

```
def chat_between_bots():
    print("MAHMOUD Bot: Hi! I'm the MAHMOUD the chatbot.")
    print("BMZ Bot: hello I am BMZ a chatbot model")
    print("Let's start our conversation.")

    for _ in range(3): # Nombre de questions dans la discussion (pour la démo, vous pouvez ajuster cela)
        # Générer une question avec GPT-2
        gpt2_question = generate_simple_gpt2_question()
        print(f"MAHMOUD: {gpt2_question}\n")

        # Obtenir la réponse de BERT
        bert_response = generate_bert_response(gpt2_question)
        print(f"BMZ: {bert_response}\n")

        # Obtenir une nouvelle question pour GPT-2
        gpt2_question = generate_simple_gpt2_question()
        print(f"MAHMOUD: {gpt2_question}\n")

        # Obtenir la réponse de BERT
        bert_response = generate_bert_response(gpt2_question)
        print(f"BMZ: {bert_response}\n")

    # Lancer la discussion entre les deux chatbots
    chat_between_bots()
```

- Fonction de Conversation entre les Bots :
 - La fonction `chat_between_bots()` est définie pour simuler une conversation entre deux chatbots.
- Message d'Introduction :
 - Les deux chatbots, MAHMOUD et BMZ, se présentent et commencent la conversation.
- Échange de Questions et Réponses :
 - Pour chaque tour de conversation, MAHMOUD génère une question aléatoire à l'aide de GPT-2.
 - BMZ, le deuxième chatbot, répond à la question de MAHMOUD en utilisant un modèle BERT entraîné.
 - Ensuite, MAHMOUD génère une autre question, et le processus se répète.
 - Le nombre de tours de conversation peut être ajusté en modifiant la valeur de la boucle `for`.

```

MAHMOUD: I need help.

The attention mask and the pad token id were not set. As a consequence, you may observe unexpected behavior. Please pass your input's `atten
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
BMZ: Sure. Tell me how can i assist you

The attention mask and the pad token id were not set. As a consequence, you may observe unexpected behavior. Please pass your input's `atten
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
MAHMOUD: What is Depression?

BMZ: A mental health disorder characterised by persistently depressed mood or loss of interest in activities, causing significant impairment

The attention mask and the pad token id were not set. As a consequence, you may observe unexpected behavior. Please pass your input's `atten
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
MAHMOUD: I can't take it anymore.

BMZ: I first want to let you know that you are not alone in your feelings and there is always someone there to help . you can always change

The attention mask and the pad token id were not set. As a consequence, you may observe unexpected behavior. Please pass your input's `atten
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
MAHMOUD: I feel ok with that.

BMZ: Oh i see. That's great.

The attention mask and the pad token id were not set. As a consequence, you may observe unexpected behavior. Please pass your input's `atten
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
MAHMOUD: I need some advice.

BMZ: Okay what do you need advice on?

```

Les résultats de la conversation entre les deux chatbots, MAHMOUD et BMZ, semblent être dans une plage acceptable. Voici quelques observations :

1. **Interaction fluide:** La conversation entre les deux chatbots semble relativement fluide, avec des réponses cohérentes et pertinentes à chaque étape de la discussion.
2. **Compréhension des questions:** Les chatbots semblent bien comprendre les questions posées par l'autre et y répondent de manière appropriée, ce qui suggère une bonne capacité de traitement du langage naturel.
3. **Variété dans les réponses:** Les réponses des chatbots semblent variées et adaptées au contexte, ce qui évite une impression de répétition ou de prévisibilité dans la conversation.

En résumé, les résultats de la conversation entre les deux chatbots semblent être satisfaisants en termes d'interaction naturelle, de compréhension du langage

Réseaux de neurones

Prétraitement des données d'intention pour le chatbot :

```
💡 Click here to ask Blackbox to help you code faster
import random
import json
import pickle
import numpy as np
import tensorflow as tf

✓ 21.9s
Python

WARNING:tensorflow:From c:\Users\AKAM\AppData\Local\Programs\Python\Python310\lib\site-packages\keras\src\losses.py:2976: The name tf.losses
```

Configuration de l'Analyseur Linguistique avec NLTK

```
💡 Click here to ask Blackbox to help you code faster
import nltk
from nltk.stem import WordNetLemmatizer
lemmatizer = WordNetLemmatizer()

Python
```

Chargement des Intentions à partir d'un Fichier JSON

```
💡 Click here to ask Blackbox to help you code faster
intents = json.loads(open('C:/Users/AKAM/Downloads/intents(1).json').read())

Python
```

1. **Importations et initialisations :** Il importe les bibliothèques nécessaires et initialise le **lemmatizer de NLTK** pour la lemmatisation des mots.

2. **Chargement des données d'intention :** Le code charge les données d'intention à partir d'un fichier JSON spécifié. Ces données contiennent des intentions prédéfinies avec des modèles de phrases associés et des étiquettes.

Prétraitement des mots et des étiquettes d'intention:

Initialisation des Listes pour le Traitement des Intentions

```
💡 Click here to ask Blackbox to help you code faster
words = []
classes = []
documents = []
ignoreLetters = ['?', '!', '.', ',']

Python
```

Traitement des Intentions pour la Construction des Données

```
💡 Click here to ask Blackbox to help you code faster
for intent in intents['intents']:
    for pattern in intent['patterns']:
        wordList = nltk.word_tokenize(pattern)
        words.extend(wordList)
        documents.append((wordList, intent['tag']))
        if intent['tag'] not in classes:
            classes.append(intent['tag'])

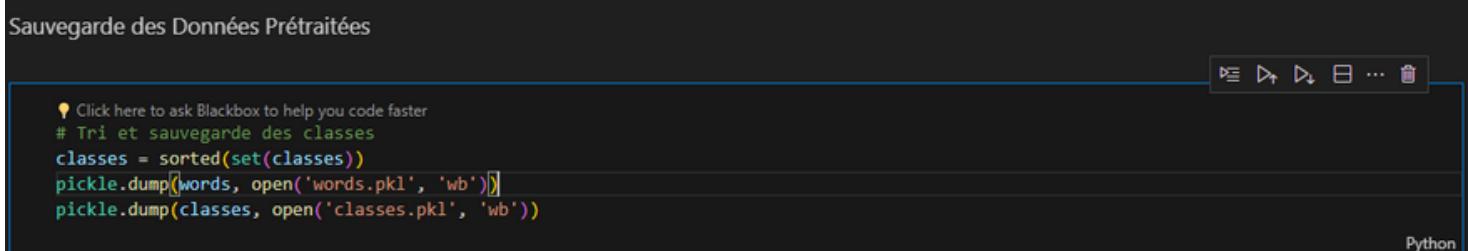
# Lemmatisation des mots et suppression des caractères ignorés
words = [lemmatizer.lemmatize(word) for word in words if word not in ignoreLetters]

# Suppression des duplicates et tri des mots
words = sorted(set(words))

Python
```

1. **Initialisation des listes** : Elle initialise des listes vides pour stocker les mots, les étiquettes et les documents (couples de mots et d'étiquettes).
2. **Extraction des mots et des étiquettes** : Elle parcourt chaque intention dans les données d'intention et extrait les modèles de phrases (**patterns**) associés. Pour chaque modèle de phrase, elle tokenise les mots à l'aide de **NLTK** et les ajoute à la liste des mots. Elle crée également des couples de mots et d'étiquettes, puis les ajoute à la liste des documents. Si une étiquette n'est pas déjà présente dans la liste des classes, elle l'ajoute également.
3. **Lemmatisation et nettoyage** : Elle lemmatise ensuite les mots à l'aide du **lemmatizer** de **NLTK** et supprime les caractères ignorés comme les points d'interrogation, les points d'exclamation, les points et les virgules.
4. **Suppression des doublons et tri** : Enfin, elle supprime les doublons des mots et les trie par ordre alphabétique. Cela garantit que chaque mot unique est représenté une seule fois dans la liste des mots, ce qui facilite la création de vecteurs de mots pour l'entraînement du modèle.

Tri et Sauvegarde des Classes et des Mots :



```
# Click here to ask Blackbox to help you code faster
# Tri et sauvegarde des classes
classes = sorted(set(classes))
pickle.dump(words, open('words.pkl', 'wb'))
pickle.dump(classes, open('classes.pkl', 'wb'))
```

1. **classes** : Cette ligne trie les classes extraites des données d'intention. Les classes sont triées dans un ordre spécifique, généralement par ordre alphabétique ou numérique, en utilisant la fonction **sorted**. La fonction **set** est utilisée pour éliminer les doublons et garantir qu'aucune classe n'est répétée.
2. **pickle.dump()** : Cette instruction sauvegarde la liste de mots dans un fichier binaire à l'aide du module **pickle**. Les mots sont stockés dans un fichier nommé '**words.pkl**', qui est ouvert en mode écriture binaire ('**wb**'). Cette étape est cruciale car elle permet de sauvegarder les mots pour une utilisation ultérieure, comme la **tokenization** des entrées utilisateur lors de l'utilisation du modèle.
3. **pickle.dump(classes, open('classes.pkl', 'wb'))** : De manière similaire à l'étape précédente, cette instruction sauvegarde la liste de classes dans un fichier binaire nommé '**classes.pkl**'. Les classes triées sont stockées dans ce fichier pour être utilisées ultérieurement, comme pour l'encodage des classes lors de l'entraînement du modèle. L'ouverture du fichier se fait en mode écriture binaire ('**wb**') pour assurer une sauvegarde correcte des données.

Préparation des Données d'Entraînement :

Initialisation des Données d'Entraînement

```
💡 Click here to ask Blackbox to help you code faster
training = []
outputEmpty = [0] * len(classes)
```

Python

Préparation des Données d'Entraînement

```
💡 Click here to ask Blackbox to help you code faster
for document in documents:
    bag = []
    wordPatterns = document[0]
    wordPatterns = [lemmatizer.lemmatize(word.lower()) for word in wordPatterns]

    # Construction du sac de mots
    for word in words:
        bag.append(1) if word in wordPatterns else bag.append(0)

    # Construction de la sortie attendue
    outputRow = list(outputEmpty)
    outputRow[classes.index(document[1])] = 1

    # Ajout des données d'entraînement
    training.append(bag + outputRow)
```

Python

1. **training** : Initialise une liste vide **training** qui sera utilisée pour stocker les données d'entraînement.
2. **outputEmpty** : Crée une liste **outputEmpty** contenant des zéros avec une longueur égale au nombre de classes. Cette liste servira de modèle pour les étiquettes de sortie attendues.
3. Parcourt chaque document dans la liste **documents**, où chaque document contient une liste de mots (**wordList**) et une étiquette (**tag**).
4. Construction du Sac de Mots :
 - o Les mots de chaque document sont lemmatisés et convertis en minuscules.
 - o Pour chaque mot dans la liste de mots (**words**), la présence ou l'absence du mot dans la liste de mots du document est représentée par **1** ou **0** respectivement. Ces valeurs sont ajoutées au sac de mots (**bag**).
5. Construction de la Sortie Attendue :
 - o Une liste de zéros, **outputRow**, est créée avec une longueur égale au nombre de classes.
 - o L'indice correspondant à la classe du document est défini sur **1** pour indiquer la classe attendue.
6. Ajout des Données d'Entraînement : Chaque sac de mots et sa sortie attendue sont ajoutés à la liste **training**.

Construction et Entraînement du Modèle de Chatbot :

Mé lange des Données d'Entraînement

```
💡 Click here to ask Blackbox to help you code faster
random.shuffle(training)
training = np.array(training)
```

Python

Définition du Modèle de Chatbot avec TensorFlow

```
💡 Click here to ask Blackbox to help you code faster
trainX = training[:, :len(words)]
trainY = training[:, len(words):]

model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(128, input_shape=(len(trainX[0]),), activation = 'relu'))
model.add(tf.keras.layers.Dropout(0.5))
model.add(tf.keras.layers.Dense(64, activation = 'relu'))
model.add(tf.keras.layers.Dropout(0.5))
model.add(tf.keras.layers.Dense(len(trainY[0]), activation='softmax'))
```

Python

1. `random.shuffle()` : Mé lange aléatoirement les données d'entraînement pour éviter tout biais dans l'ordre des échantillons.

2. Transformation en **Numpy Array** :

- `training` est converti en un tableau **Numpy** pour une manipulation plus efficace.

3. Division des Données :

- `trainX` contient les sacs de mots, tandis que `trainY` contient les sorties attendues.
- `trainX` contient toutes les colonnes jusqu'à la longueur des mots (`len(words)`), représentant les sacs de mots.
- `trainY` contient toutes les colonnes après la longueur des mots, représentant les sorties attendues.

4. Construction du Modèle :

- `tf.keras.Sequential()` : Crée un modèle séquentiel.
- `tf.keras.layers.Dense()` : Couche dense avec **128** neurones et fonction d'activation **ReLU**. La première couche nécessite l'entrée spécifiée avec la forme `(len(trainX[0]))`.
- `tf.keras.layers.Dropout()` : Couche de **dropout** pour éviter le **surapprentissage**.
- `tf.keras.layers.Dense()` : Couche dense avec **64** neurones et activation **ReLU**.
- `tf.keras.layers.Dropout()` : Couche de **dropout** supplémentaire.
- `tf.keras.layers.Dense()` : Couche dense de sortie avec un nombre de neurones égal au nombre de classes et activation **softmax** pour obtenir des probabilités de chaque classe.

Construction et Entraînement du Modèle de Chatbot :

Mé lange des Données d'Entraînement

```
💡 Click here to ask Blackbox to help you code faster
random.shuffle(training)
training = np.array(training)
```

Python

Définition du Modèle de Chatbot avec TensorFlow

```
💡 Click here to ask Blackbox to help you code faster
trainX = training[:, :len(words)]
trainY = training[:, len(words):]

model = tf.keras.Sequential()
model.add(tf.keras.layers.Dense(128, input_shape=(len(trainX[0]),), activation = 'relu'))
model.add(tf.keras.layers.Dropout(0.5))
model.add(tf.keras.layers.Dense(64, activation = 'relu'))
model.add(tf.keras.layers.Dropout(0.5))
model.add(tf.keras.layers.Dense(len(trainY[0]), activation='softmax'))
```

Python

1. `random.shuffle()` : Mé lange aléatoirement les données d'entraînement pour éviter tout biais dans l'ordre des échantillons.

2. Transformation en **Numpy Array** :

- `training` est converti en un tableau **Numpy** pour une manipulation plus efficace.

3. Division des Données :

- `trainX` contient les sacs de mots, tandis que `trainY` contient les sorties attendues.
- `trainX` contient toutes les colonnes jusqu'à la longueur des mots (`len(words)`), représentant les sacs de mots.
- `trainY` contient toutes les colonnes après la longueur des mots, représentant les sorties attendues.

4. Construction du Modèle :

- `tf.keras.Sequential()` : Crée un modèle séquentiel.
- `tf.keras.layers.Dense()` : Couche dense avec **128** neurones et fonction d'activation **ReLU**. La première couche nécessite l'entrée spécifiée avec la forme `(len(trainX[0]))`.
- `tf.keras.layers.Dropout()` : Couche de **dropout** pour éviter le **surapprentissage**.
- `tf.keras.layers.Dense()` : Couche dense avec **64** neurones et activation **ReLU**.
- `tf.keras.layers.Dropout()` : Couche de **dropout** supplémentaire.
- `tf.keras.layers.Dense()` : Couche dense de sortie avec un nombre de neurones égal au nombre de classes et activation **softmax** pour obtenir des probabilités de chaque classe.

Compilation Entraînement et Sauvegarde du Modèle de Chatbot :

Compilation du Modèle avec l'Optimiseur SGD

```
💡 Click here to ask Blackbox to help you code faster
sgd = tf.keras.optimizers.SGD(learning_rate=0.01, momentum=0.9, nesterov=True)
model.compile(loss='categorical_crossentropy', optimizer=sgd, metrics=['accuracy'])
```

Python

Entraînement du Modèle et Sauvegarde

```
💡 Click here to ask Blackbox to help you code faster
model.fit(trainX, trainY, epochs=200, batch_size=5, verbose=1)
model.save('chatbot_model.h5')
print('Done')
```

Python

```
Epoch 1/200
WARNING:tensorflow:From c:\Users\AKAM\AppData\Local\Programs\Python\Python310\lib\site-packages\keras\src\utils\tf_utils.py:492: The name tf
WARNING:tensorflow:From c:\Users\AKAM\AppData\Local\Programs\Python\Python310\lib\site-packages\keras\src\engine\base_layer_utils.py:384: Th
130/130 [=====] - 4s 8ms/step - loss: 5.3209 - accuracy: 0.0123
Epoch 2/200
130/130 [=====] - 1s 8ms/step - loss: 5.2930 - accuracy: 0.0154
Epoch 3/200
130/130 [=====] - 1s 6ms/step - loss: 5.2504 - accuracy: 0.0247
Epoch 4/200
```

1. Compilation du Modèle :

- **sgd** : Définit l'optimiseur **Stochastic Gradient Descent (SGD)** avec un taux d'apprentissage de **0.01**, un **momentum** de **0.9** et la méthode **Nesterov** activée.
- **model.compile()** : Compile le modèle en utilisant la perte de **catégorical crossentropy** comme fonction de perte, l'optimiseur **SGD** défini précédemment et la métrique d'exactitude pour évaluer les performances.

2. Entraînement du Modèle :

- **model.fit()** : Entraîne le modèle sur les données d'entraînement **trainX** et **trainY** pour **200 Epochs** avec une taille de lot de **5**. L'argument **verbose=1** permet d'afficher les détails de l'entraînement pendant le processus.

3. Sauvegarde du Modèle :

- **model.save()** : Sauvegarde le modèle entraîné au format **H5** dans un fichier nommé **chatbot_model.h5**.

4. Message de Confirmation :

- **print('Done')** : Affiche un message de confirmation une fois l'entraînement terminé et le modèle sauvegardé.

Résultats d'entraînement : Les résultats d'entraînement du modèle de chatbot montrent la perte et la précision pour chaque **epoch** sur un total de **200 epochs**. La perte diminue progressivement tandis que la précision augmente, ce qui indique que le modèle s'améliore avec l'entraînement.

La dernière précision (**accuracy**) du modèle de chatbot, mesurée après la dernière **epoch** d'entraînement, est de **0.6256**, ce qui équivaut à environ **62,56%**. Cela signifie que, sur l'ensemble des données d'entraînement, le modèle a correctement classé environ **62,56%** des exemples.

Une précision de **62,56%** peut être considérée comme raisonnable.

Chargement des Données et du Modèle du Chatbot :

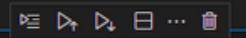
Importation des Modules et Chargement du Modèle

```
💡 Click here to ask Blackbox to help you code faster
import random
import pickle
import numpy as np
import nltk
from nltk.stem import WordNetLemmatizer
from tensorflow.keras.models import load_model
```

Python

Chargement des Données et du Modèle

```
💡 Click here to ask Blackbox to help you code faster
words = pickle.load(open('words.pkl', 'rb'))
classes = pickle.load(open('classes.pkl', 'rb'))
model = load_model('chatbot_model.h5')
```



1. Chargement des Données :

- **words** : Charge les mots **prétraités** à partir du fichier **words.pkl**.
- **classes** : Charge les classes **prétraitées** à partir du fichier **classes.pkl**.

2. Chargement du Modèle :

- **model** : Charge le modèle du **chatbot** à partir du fichier **chatbot_model.h5** enregistré précédemment à l'aide de **Keras**.

Prétraitement du Texte pour l'Entrée du Modèle :

Fonctions de Prétraitement

```
💡 Click here to ask Blackbox to help you code faster
lemmatizer = WordNetLemmatizer()

def clean_up_sentence(sentence):
    sentence_words = nltk.word_tokenize(sentence)
    sentence_words = [lemmatizer.lemmatize(word.lower()) for word in sentence_words]
    return sentence_words
```

Python

Fonction de Création du Sac de Mots (BoW)

```
💡 Click here to ask Blackbox to help you code faster
def bow(sentence, words, show_details=True):
    sentence_words = clean_up_sentence(sentence)
    bag = [0]*len(words)
    for s in sentence_words:
        for i, w in enumerate(words):
            if w == s:
                bag[i] = 1
                if show_details:
                    print(f"found in bag: {w}")
    return np.array(bag)
```

• Initialisation du Lemmatizer :

- **lemmatizer**: Initialisation de l'objet **WordNetLemmatizer** pour la lemmatisation des mots.

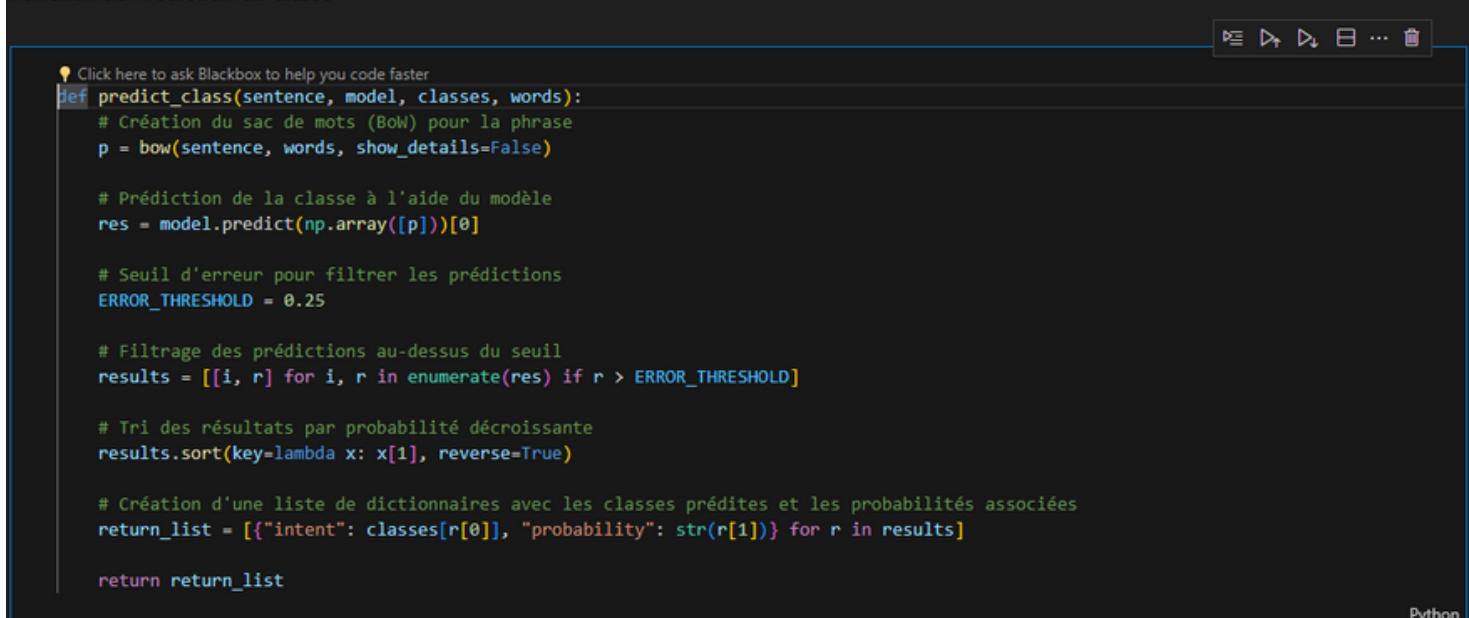
• Nettoyage de la Phrase :

- **def clean_up_sentence()** : Définition de la fonction **clean_up_sentence** qui prend une phrase en entrée et effectue le prétraitement nécessaire.
- **sentence_words**: Tokenisation de la phrase en mots individuels à l'aide de **nltk**.

- **sentence_words** : Lemmatisation de chaque mot de la phrase pour normaliser les formes de mots.
- La fonction renvoie une liste des mots de la phrase après prétraitement.
- Création du Sac de Mots (**Bag of Words**) :
 - **def bow()** : Définition de la fonction **bow** qui crée un sac de mots pour une phrase donnée.
 - **bag** : Initialise un vecteur de zéros de longueur égale au nombre de mots dans le corpus.
 - Parcours de chaque mot de la phrase et vérification s'il est présent dans le corpus de mots :
 - Si le mot est présent, l'indice correspondant dans le vecteur **bag** est mis à 1.
 - Si **show_details** est vrai, affiche le mot trouvé dans le sac de mots.
 - La fonction renvoie un tableau **numpy** représentant le sac de mots pour la phrase donnée.

Prédiction de la Classe à partir de la Phrase :

Fonction de Prédiction de Classe



```

Click here to ask Blackbox to help you code faster
def predict_class(sentence, model, classes, words):
    # Création du sac de mots (BoW) pour la phrase
    p = bow(sentence, words, show_details=False)

    # Prédiction de la classe à l'aide du modèle
    res = model.predict(np.array([p]))[0]

    # Seuil d'erreur pour filtrer les prédictions
    ERROR_THRESHOLD = 0.25

    # Filtrage des prédictions au-dessus du seuil
    results = [[i, r] for i, r in enumerate(res) if r > ERROR_THRESHOLD]

    # Tri des résultats par probabilité décroissante
    results.sort(key=lambda x: x[1], reverse=True)

    # Création d'une liste de dictionnaires avec les classes prédites et les probabilités associées
    return_list = [{"intent": classes[r[0]], "probability": str(r[1])} for r in results]

    return return_list

```

Python

1. Prédiction de la Classe :

- **def predict_class()** : Définition de la fonction **predict_class** qui prend en entrée une phrase, le modèle du **chatbot**, les classes et les mots du corpus.
- Crée un sac de mots (**BoW**) pour la phrase à l'aide de la fonction **bow** définie précédemment.
- Utilise le modèle pour prédire la classe correspondante à partir du sac de mots.
- Applique un seuil d'erreur pour filtrer les prédictions ayant une probabilité inférieure au seuil spécifié (**ERROR_THRESHOLD**).
- Trie les résultats par probabilité décroissante.
- Crée une liste de dictionnaires contenant les classes prédites et les probabilités associées.
- Renvoie cette liste de résultats.

Réponse du Chatbot :

Fonction de Réponse du Chatbot

```
💡 Click here to ask Blackbox to help you code faster
def chatbot_response(text):
    # Chargement du modèle, des mots, et des classes depuis les fichiers
    model = load_model('chatbot_model.h5')
    words = pickle.load(open('words.pkl', 'rb'))
    classes = pickle.load(open('classes.pkl', 'rb'))

    # Prédiction de la classe du texte
    ints = predict_class(text, model, classes, words)

    # Récupération de la classe prédictive
    tag = ints[0]['intent']

    # Recherche de la classe dans les intentions
    for intent in intents['intents']:
        if intent['tag'] == tag:
            # Sélection d'une réponse aléatoire associée à la classe
            response = random.choice(intent['responses'])
            return response

Python
```

- **def chatbot_response(text)** : Définition de la fonction **chatbot_response** prenant en entrée le texte saisi par l'utilisateur.
- Chargement du modèle de **chatbot** ainsi que des mots et des classes depuis les fichiers sauvegardés précédemment.
- Prédiction de la classe du texte en utilisant la fonction **predict_class**, qui retourne une liste de dictionnaires contenant les classes prédictives et les probabilités associées.
- Récupération de la classe prédictive à partir des résultats de prédiction.
- Recherche de la classe dans les intentions pour trouver les réponses associées à cette classe.
- Sélection aléatoire d'une réponse parmi les réponses associées à la classe prédictive.
- Renvoi de la réponse sélectionnée.

Interaction Utilisateur-Chatbot :

```
WARNING:tensorflow:5 out of the last 5 calls to <function Model.make_predict_function.<locals>.predict_function at 0x0000017DD9B1A7A0> triggered by op 'Placeholder' (op_type: 0).
1/1 [=====] - 8s 8s/step
ChatBot: Hello there. Glad to see you're back. What's going on in your world right now?
WARNING:tensorflow:6 out of the last 6 calls to <function Model.make_predict_function.<locals>.predict_function at 0x0000017DD9B1B9A0> triggered by op 'Placeholder' (op_type: 0).
1/1 [=====] - 6s 6s/step
ChatBot: That's great to hear. I'm glad you're feeling this way.
1/1 [=====] - 7s 7s/step
ChatBot: I want to help you. I really do. But in order for me to help you, you're gonna have to talk to me.
1/1 [=====] - 7s 7s/step
ChatBot: Any time!
```

Interaction entre le modél BMZ et gpt_2 :

```
Commençons notre conversation.
MAHMOUD: How can I find a mental health professional for myself or my child?

1/1 [=====] - 8s 8s/step
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
BMZ: Feeling comfortable with the professional you or your child is working with is critical to the success of the treatment. Finding the right professional can be challenging, but it's important to take the time to find someone who you feel comfortable with.

MAHMOUD: I'm scared. I don't know what to do," he said.

1/1 [=====] - 8s 8s/step
Setting `pad_token_id` to `eos_token_id`:50256 for open-end generation.
BMZ: I understand how you feel. Don't put yourself down because of it.

MAHMOUD: What treatment options are available?

1/1 [=====] - 8s 8s/step
BMZ: Just as there are different types of medications for physical illness, different treatment options are available for individuals with mental health conditions. These may include therapy, medication, or a combination of both.
```

TFIDF

- **Chargement des données et prétraitement :**

Les données sont chargées à partir d'un fichier JSON nommé "intents.json", qui contient des intentions (intents) de conversation avec des modèles de phrases associés à des tags et des réponses.

Les phrases de chaque intention sont extraites ainsi que leurs tags correspondants.

```
import json

# Charger les données depuis le fichier intent Chargement en cours...
with open('intents.json', 'r', encoding='utf-8') as file:
    data = json.load(file)

# Afficher les données pour vérification
print(json.dumps(data, indent=2))
```

- **Préparation des données et Vectorisation des patterns avec TF-IDF :**

Les patterns (modèles de phrases) sont convertis en vecteurs TF-IDF à l'aide de la classe TfidfVectorizer de scikit-learn. Cela permet de représenter chaque phrase sous forme de vecteur numérique basé sur la fréquence des mots dans le corpus.

Les tags sont encodés en utilisant LabelEncoder de scikit-learn.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.preprocessing import LabelEncoder
import numpy as np

# Extraire les patterns et les tags
patterns = []
tags = []
for intent in data['intents']:
    for pattern in intent['patterns']:
        patterns.append(pattern)
        tags.append(intent['tag'])

# Effectuer une tokenisation sur les phrases
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(patterns)

# Encoder les tags
encoder = LabelEncoder()
y = encoder.fit_transform(tags)

print("Patterns:", patterns)
print("Tags:", tags)
print("Tokenized patterns:", X.toarray())
print("Encoded tags:", y)
```

- **Fonction de réponse avec TF-IDF :**

La fonction `respond(query)` prend une requête de l'utilisateur en entrée.

Elle vectorise la requête en utilisant le même `TfidfVectorizer`.

Elle calcule la similarité cosinus entre la requête et tous les patterns d'entraînement.

Elle sélectionne le pattern d'entraînement avec la plus grande similarité et trouve le tag correspondant.

Enfin, elle sélectionne une réponse aléatoire associée à ce tag dans les données.

```
from sklearn.metrics.pairwise import cosine_similarity

def respond(query):
    # Transformer la requête en un vecteur de caractéristiques
    query_vector = vectorizer.transform([query])

    # Calculer la similarité entre le vecteur de requête et tous les vecteurs de phrases d'entraînement
    similarities = cosine_similarity(query_vector, X)

    # Trouver l'index de la phrase d'entraînement avec la plus grande similarité
    best_match_index = np.argmax(similarities)

    # Trouver le tag correspondant à cette phrase
    best_match_tag = tags[best_match_index]

    # Trouver une réponse correspondant à ce tag
    for intent in data['intents']:
        if intent['tag'] == best_match_tag:
            response = np.random.choice(intent['responses'])
            break

    return response

# Test de la fonction de réponse
query = "Good morning"
print("Query:", query)
print("Response:", respond(query))
```

- **Boucle de conversation**

Une boucle `while` permet à l'utilisateur de taper des requêtes et de recevoir des réponses.

La fonction `respond` est utilisée pour générer une réponse à chaque requête.

L'utilisateur peut taper "bye" pour quitter la boucle.

```
while True:
    query = input("Vous: ")
    if query.lower() == "bye":
        print("Chatbot: Au revoir!")
        break
    response = respond(query)
    print("Chatbot:", response)
```

- **Génération de questions avec GPT-2 :**

Un modèle GPT-2 pré-entraîné est chargé à l'aide de la bibliothèque transformers. Une fonction `generate_gpt2_question()` est définie pour générer une question en utilisant GPT-2. Elle prend un pattern aléatoire parmi ceux disponibles et génère une question complète en utilisant GPT-2.

```

gpt2_model = GPT2LMHeadModel.from_pretrained("gpt2")
gpt2_tokenizer = GPT2Tokenizer.from_pretrained("gpt2")

# Définir explicitement pad_token_id sur eos_token_id
gpt2_model.config.pad_token_id = gpt2_tokenizer.eos_token_id

# Fonction pour générer une question avec GPT-2
def generate_gpt2_question():
    random_pattern = random.choice(patterns)
    input_text = f"{random_pattern}" # Ajout de "Generate a question:" avant le pattern

    # Encoder le texte
    input_ids = gpt2_tokenizer.encode(input_text, return_tensors="pt", max_length=512)

    # Définir le masque d'attention
    attention_mask = torch.ones(input_ids.shape, dtype=torch.long)

    # Générer la séquence
    output = gpt2_model.generate(input_ids, attention_mask=attention_mask, max_length=150, num_beams=5,
                                no_repeat_ngram_size=2, top_k=50, top_p=0.95, temperature=0.7)

    # Décoder la séquence
    generated_sequence = gpt2_tokenizer.decode(output[0], skip_special_tokens=True)

    # Extraire la question complétée après un point d'interrogation, un point ou entre guillemets
    match = re.search(r"(.*[.!?])|(.*\")", generated_sequence)
    if match:
        generated_question = match.group()
    else:
        generated_question = generated_sequence

    return generated_question

```

- **Simulation de conversation entre les chatbots :**

Une fonction `chat_between_bots()` est définie pour simuler une conversation entre les deux chatbots.

Dans cette fonction, chaque bot échange tour à tour des questions et des réponses générées avec les deux méthodes différentes (TF-IDF et GPT-2).

```

def chat_between_bots():
    print("MAHMOUD: Hi! ")
    print("BMZ: Hello, I am BMZ, a chatbot model.")
    print("Let's start our conversation.")

    for _ in range(3): # Nombre de questions dans la discussion (vous pouvez ajuster cela à des fins de démonstration)
        # Générer une question avec GPT-2
        gpt2_question = generate_gpt2_question()
        print(f"MAHMOUD: {gpt2_question}\n")

        # Obtenir la réponse de TF-IDF
        tfidf_response_text = respond(gpt2_question)
        print(f"BMZ: {tfidf_response_text}\n")

    # Lancer la conversation entre les deux chatbots
    chat_between_bots()

```

Test du Chatbot avec l'Interaction Utilisateur :

```
Vous: i hate you
Chatbot: Forgive me if i did anything to offend you. I only want to help
Vous: I am so sad
Chatbot: I'm sorry to hear that. I'm here for you. Talking about it might help. So, tell me why do you think you're feeling this way?
Vous: no one likes me
Chatbot: It's only natural to feel this way. Tell me more. What else is on your mind?
Vous: I want to die
Chatbot: I'm very sorry to hear that but you have so much to look forward to. Please seek help by contacting: 9152987821.
```

Le chatbot adapte ses réponses en fonction du contexte de la conversation. Lorsque l'utilisateur exprime des émotions négatives, le chatbot répond avec empathie et offre des ressources d'aide. Lorsque l'utilisateur change de sujet vers des sujets plus légers ou informatifs, le chatbot répond en engageant une conversation sur ces sujets. Cela montre une capacité du chatbot à comprendre et à répondre à différents types de requêtes de l'utilisateur.

Résultats et Comparaison

Pour ce qui est des métriques d'accuracy, precision, recall et F-score , on a eu des résultats assez moyens, avec pas beaucoup de différences entre les méthodes, ceci est dûs à la quantité limités des données qu'on a utilisé.

Pour comparer entre les méthodes , on utilisera d'autres mesures tel que le temps d'exécution, interprétabilité, Utilisation de ressources, Capacité à Généraliser, précision ...

MESURE / MODEL	BERT	RÉSEAUX DE NEURONES	TFIDF
TEMPS D'EXÉCUTION	✓	✓ ✓	✓ ✓ ✓
INTERPRÉTABILITÉ	✓ ✓	✓ ✓	✓ ✓ ✓
UTILISATION DE RESSOURCES	✓	✓ ✓	✓ ✓ ✓
CAPACITÉ À GÉNÉRALISER	✓ ✓ ✓	✓ ✓	✓
COMPRÉHENSION DES SUBTILITÉS DU LANGAGE	✓ ✓ ✓	✓ ✓	✓
PRÉCISION DANS LES TÂCHES NLP COMPLEXES	✓ ✓ ✓	✓ ✓	✓

BERT offre une grande précision et une capacité de généralisation élevée, mais au prix de temps d'exécution plus longs et de ressources plus importantes. Les réseaux de neurones offrent une certaine flexibilité et peuvent être adaptés à des tâches spécifiques, tandis que TF-IDF est rapide et simple, mais peut manquer de capacité à généraliser dans des cas plus complexes.

BERT se distingue par sa capacité à capturer des informations contextuelles riches et complexes, ce qui lui permet d'obtenir des performances supérieures dans un large éventail de tâches de NLP par rapport aux méthodes traditionnelles telles que TF-IDF et même parfois par rapport aux réseaux de neurones non-transformers.



Conclusion

L'étude de la conception et de l'implémentation de deux chatbots capables de communiquer entre eux nous a permis de plonger au cœur d'un domaine passionnant de l'intelligence artificielle et du traitement du langage naturel. À travers cette expérience, nous avons pu constater de première main la complexité et les défis inhérents à la création d'agents conversationnels capables de dialoguer de manière autonome.

La réalisation de ces chatbots interconnectés a mis en lumière l'importance cruciale de la planification minutieuse, de la conception réfléchie et du développement itératif. Du choix des technologies sous-jacentes à la modélisation des dialogues, chaque étape du processus de création a nécessité une attention particulière pour assurer la cohérence, la pertinence et la fluidité des échanges entre les deux entités conversationnelles.

En observant le dialogue entre ces deux chatbots, nous avons également pu apprécier la façon dont les algorithmes d'apprentissage automatique et les techniques de traitement du langage naturel évoluent pour permettre des interactions plus sophistiquées et plus naturelles. L'adaptabilité et l'amélioration continue sont essentielles pour garantir que ces agents conversationnels restent pertinents et utiles dans un environnement en constante évolution.