
Desarrollo de Aplicaciones y Servicios Inteligentes

ChatCare

Integrantes

Jesús Espadas

Gabriele Petroni

Ricardo Palomares

Tabla de contenidos

Introducción y objetivos	3
Solución planteada	4
Datos de entrada	5
Dataset Kaggle	5
Datasets HuggingFace	6
Fine-tuning	6
Datos de salida	8
Conversaciones ChatBots intermedios	8
Respuestas conversacionales	8
Descripción de scripts desarrollados	9
ChatBot inicial	9
ChatBot intermedio para estrategia sin etiquetado	10
Agregación de datasets	10
Resultados	11
ChatBots intermedios para estrategia con etiquetado	11
Agregación manual de datasets	11
Agregación automática de datasets	13
Generación de conversaciones	14
Instrucciones para instalación y ejecución	15
ChatBot intermedio sin tag	15
Carga de los json	15
Carga de librerías	16
Carga y guardado de modelos	16
Entrenamiento	16
Bloques del chat	17
ChatBots intermedios con tag	18
01-chatbot-for-mental-health-aggregate.ipynb	18
02-automatic_tagging.py	20
03-chatbot-for-mental-health-generate-jsonl.ipynb	21
ChatBot Fine-Tuning	23
Entrenamiento	23
Resultados	25
Despliegue y prueba	25
Conclusiones	26

Introducción y objetivos

La salud mental es un tema candente actualmente. Proporcionar un ChatBot que pueda ayudar a detectar emociones y reconfortar al usuario cuando no tiene acceso inmediato a profesionales puede resultar muy útil a la vez que proporciona un desafío de programación mediante IA.

La intención es preparar un ChatBot construido con Python que partirá de un notebook disponible públicamente. Con él se realizará un entrenamiento y se mantendrán charlas con un usuario humano, guardando el resultado de las conversaciones en un formato que, posteriormente, se introducirá como datos de entrenamiento para ChatGPT, con el objetivo de hacer un fine-tuning y construir el ChatBot definitivo, el cual será el producto final y que tratará con los usuarios para intentar ayudarle con sus problemas emocionales.

Solución planteada

Para alcanzar el objetivo, primero hemos decidido crear varios ChatBots por nuestra cuenta, utilizando un notebook de ejemplo hecho en Python que encontramos en Kaggle. A este notebook le fuimos haciendo modificaciones y aplicando diferentes datasets de conversaciones de ChatBots (centrados en temas de salud mental) para crear distintos modelos.

En medio de la creación de modelos para generar conversaciones para alimentar al fine-tuning, decidimos usar dos vías. Una consistió en trabajar con ChatBots utilizando un sistema de clasificación o tags de sentimientos o emociones detectados, para dar una respuesta en relación al tag detectado; en este caso se iría expandiendo un primer dataset agregando tags y datos de entrenamiento.

El otro grupo de ChatBots se ha desarrollado sin utilizar el método de tags: directamente se entrena al modelo utilizando un sistema de pregunta y respuesta usando diferentes datasets de gran tamaño. Aquí el objetivo es generar varios modelos cada uno usando un dataset diferente y un último modelo que contenga una mezcla de varios datasets.

Una vez desarrollados los modelos planteados, hemos generado con ellos las conversaciones antes descritas, eligiendo cuáles consideramos correctas para un ChatBot de ayuda para la salud mental, para hacer un buen entrenamiento del ChatBot final (en nuestro caso tenemos varios centenares de conversaciones entre usuario y asistente). Aquí queremos explotar las habilidades que tiene ChatGPT e intentar especializarlo como ayudante para gente que sufra de problemas de salud mental.

Todo el proyecto se ha incorporado en el repositorio en GitHub accesible desde la siguiente dirección web:

<https://github.com/gabrielepetroni/chatcare>

Datos de entrada

Los datos de entrada se explican para cada script, para mayor detalle.

Dataset Kaggle

Se comenzó el desarrollo a partir de un notebook encontrado en Kaggle (<https://www.kaggle.com/code/jocelyndumlao/ChatBot-for-mental-health-conversations>) que incluye un dataset en formato JSON compuesto por 80 entradas con tres atributos:

- **Tag:** una etiqueta que representa el sentimiento u orientación de la entrada. Ejemplos de etiquetas son: hello, help, mental-illness-definition...
- **Patterns:** un array que, a su vez, contiene una o más frases que representan posibles entradas en el chat del usuario. Todas ellas representan el mismo sentimiento u orientación delimitado por la etiqueta.
- **Responses:** un array que, a su vez, contiene una o más frases que representan posibles respuestas del ChatBot apropiadas para cualquiera de las entradas en el array Patterns.

Durante el desarrollo, se usó este mismo formato con otros ficheros que aumentaban el contenido del dataset. En total, se dispone de los siguientes ficheros en este formato:

- **intents-original.json:** el fichero original disponible junto con el notebook original en Kaggle.
- **intents-last-manual-addition.json:** versión que acumula al fichero anterior etiquetas y contenido agregados manualmente mediante uno de los notebooks desarrollado para esta tarea.
- **intents-auto.json:** versión que acumula al fichero anterior contenido agregado y etiquetado automáticamente mediante el script de transformers.
- **intents.json:** versión final que acumula al fichero anterior el resultado de una pequeña revisión manual y más etiquetas y contenido agregados manualmente mediante el notebook correspondiente.

- **intents-old.json**: versión de respaldo del fichero intents.json generada por el notebook de etiquetado y agregado manual.

Para usar cualquiera de estos ficheros con los notebooks que cargan el dataset en formato original es preciso renombrar el fichero deseado al nombre original, **intents.json**.

Datasets HuggingFace

El dataset original del notebook de Kaggle se consideró demasiado pequeño para poder realizar un entrenamiento de fine-tuning apropiado, por lo que se buscaron más. Se encontraron varios en HuggingFace, entre los cuales se usaron:

https://huggingface.co/datasets/PrinceAyush/Mental_Health_conv

<https://huggingface.co/datasets/marmikpandya/mental-health>

Estos datasets tienen en común dos cosas:

- No tienen una columna de etiqueta.
- Los pares de patrón y respuesta son univaluados; es decir, no son arrays con varias entradas asociadas a un mismo sentimiento o intención, bien como entrada del usuario o como posible salida. Son, por el contrario, entradas 1:1 de patrón de pregunta y respuesta.

Fine-tuning

El formato de ficheros de fine-tuning viene determinado por la infraestructura de OpenAI en la plataforma de computación en la nube Azure. Los detalles se explican en la página [Customize a model with fine-tuning](#) y se resumen en que se pide un fichero JSONL (cada línea del fichero es una estructura JSON válida) con el formato siguiente:

```
{ "messages": [  
  
  { "role": "system", "content": "Descripción de la personalidad o papel del asistente" },  
  
  { "role": "user", "content": "Entrada del usuario" },  
  
  { "role": "assistant", "content": "Respuesta del asistente GPT" }  
  
]}
```

Aunque en el ejemplo anterior solo hay una participación de cada interlocutor en cada línea JSON, el formato permite incluir diálogos más largos, con sucesivos intercambios entre usuario y asistente.

Datos de salida

Conversaciones ChatBots intermedios

Como se ha explicado en puntos anteriores, hemos creado varios ChatBots de formas y con datos diferentes que nos permiten generar varias conversaciones para entrenar al ChatBot final usando el Fine-Tuning.

Estas conversaciones se han ido generando a medida que conversábamos con los diferentes ChatBots. De esta forma pensábamos generar mayor variedad de conversaciones y hacer mejor el entrenamiento. Se han generado 100 conversaciones utilizando el método de ChatBot intermedio sin tag guardados en el fichero conv-ft1.txt (este fichero se procesa en fase del Fine-Tuning para pasar los datos a JSONL). Por otro lado, se han generado diálogos etiquetados, directamente en formato JSONL de las 5.106 entradas del dataset PrinceAyush utilizando el método de ChatBot intermedio con tag guardados en el fichero conv-ft2.jsonl.

Como se ha explicado anteriormente estas conversaciones serán usadas como entrada para el ChatBot final.

Respuestas conversacionales

Una vez entrenado el ChatBot final las salidas deberán ser respuestas coherentes a las peticiones del usuario, que le ayuden con sus problemas o le den consejos que permitan tratar su situación.

Descripción de scripts desarrollados

ChatBot inicial

El ChatBot inicial no se desarrolló íntegramente. En su lugar, se partió de uno encontrado en el repositorio de modelos y datasets de Kaggle (<https://www.kaggle.com/code/jocelyndumlao/ChatBot-for-mental-health-conversations>). En ningún momento el objetivo era obtener como producto final una evolución de este ChatBot, sino usarlo para generar conversaciones que se volcarían en el formato JSONL que necesita GPT a través de los servicios de Azure para hacer el fine-tuning del modelo GPT.

Pronto se observó que el ChatBot no era enteramente funcional. Aunque el formato del dataset asociado (ya descrito en los apartados anteriores) y el entrenamiento del modelo parecían óptimos como punto de partida, una vez hecho este, el ChatBot no utilizaba las respuestas incluidas en el dataset. En su lugar, para cada entrada interactiva del usuario se predecía la etiqueta según el modelo entrenado y, a continuación utilizaba un simple if-else que solo procesaba dos o tres etiquetas de las contenidas en el dataset, ofreciendo respuestas fijas sin acudir a las disponibles en dataset, y una respuesta genérica en todos los demás casos.

Se procedió a corregir la función **generate_response** para tomar, a partir de la etiqueta predicha por el modelo, una de las respuestas disponibles en el dataset (elegida al azar) para esa etiqueta. El modelo con el dataset original presenta este rendimiento:



Como se ve, la distribución es muy irregular, con unas pocas etiquetas con resultados altos y el resto sin datos, dando lugar a una media ponderada francamente baja.

Tras ello, se procedió a usar el ChatBot un par de sesiones y pronto quedó patente, como era de esperar, que su efectividad era muy baja. Por ello, se optaron por varias vías alternativas:

- Incrementar el dataset usando otros disponibles. Dado que los encontrados con una estructura similar en cuanto a preguntas y respuestas no estaban etiquetados, había que etiquetar sus entradas de forma asistida.
- Utilizar otros datasets disponibles, prescindiendo del etiquetado. En este caso se suprimió el etiquetado como objetivo de la predicción y se usó en su lugar la respuesta asociada a cada pregunta.

ChatBot intermedio para estrategia sin etiquetado

Agregación de datasets

Se han usado 3 datasets diferentes obtenidos de Hugging Face:

<https://huggingface.co/datasets?sort=trending&search=mental+health>

Esta web nos ha permitido buscar varios datasets centrados en el tema de la salud mental. Con estos datasets se han creado 4 diferentes ChatBots: 3 de ellos con cada uno de los datasets utilizados, y el cuarto conteniendo los 3 datasets cargados. Con esto buscamos tener conversaciones diferentes para la parte del Fine-Tuning y también observar si podemos obtener algún ChatBot de buen nivel entre medias.

Los datasets seleccionados se han llamado **data_NoEtiqueta.json** (con 7576 filas pregunta-respuesta), **data_NoEtiqueta_2.json** (con 1268 filas pregunta-respuesta) y **data_NoEtiqueta_3.json** (con 995 filas pregunta-respuesta). Cada dataset se procesaba individualmente con su correspondiente ChatBot y el último ChatBot contaba con un dataset combinado de 8539 filas de pregunta-respuesta. En la cantidad de filas no se han contado filas repetidas puesto que han sido eliminadas, para evitar problemas a la hora de chatear con el ChatBot. También se han eliminado columnas extra que contenían datos extra que para el ChatBot no eran necesarios.

Resultados

Se han conseguido elaborar 4 ChatBots diferentes y se han guardado sus modelos para que sea más fácil cargarlos. Los modelos se llaman **model_Jesus_1_v1.joblib**, **model_Jesus_2_v1.joblib**, **model_Jesus_3_v1.joblib** y **model_Jesus_4_v1.joblib**, siendo **model_Jesus_4_v1.joblib** el modelo que contiene los 3 datasets combinados. El resto tiene cada uno de los datasets. En el apartado [ChatBot intermedio sin tag](#) del capítulo Instrucciones para instalación y ejecución se explica cómo cargar estos modelos.

De entre todos los modelos, el que mejores resultados ha arrojado es el modelo **model_Jesus_1_v1** pues es el que mejor contesta. El principal problema encontrado es que tiende a usar 3 respuestas de forma muy genérica, pero en la mayoría de ocasiones tiene sentido el contexto.

En cuanto a los otros modelos, han dado más problemas, puesto que por algún motivo que no hemos conseguido averiguar, muchas de las respuestas de los ChatBots son nada (espacios en blanco). Hemos intentado limpiar los datasets para evitar estos casos pero seguía apareciendo el problema.

A pesar de los problemas se han conseguido sacar varias conversaciones (100 en total) a base de probar diferentes preguntas y contexto, consiguiendo buenas respuestas por parte de los ChatBots.

ChatBots intermedios para estrategia con etiquetado

Agregación manual de datasets

Para usar el ChatBot corregido, manteniendo el etiquetado, con una mayor efectividad era necesario incrementar el dataset. También se renombraron varias etiquetas con contenido genérico (“fact-15”, “fact-16”, etc.) a nombres más aproximados a la colección de patrones y respuestas asociados a las mismas.

Para incrementar el dataset se desarrolló una variación del notebook, que se ha llamado de agregación, en la que se pide un fichero correspondiente al dataset extra, los nombres de las columnas que representan la entrada (patterns) y la salida (responses), así como el número de fila desde el que comenzar y el de iteraciones en la ejecución del script.

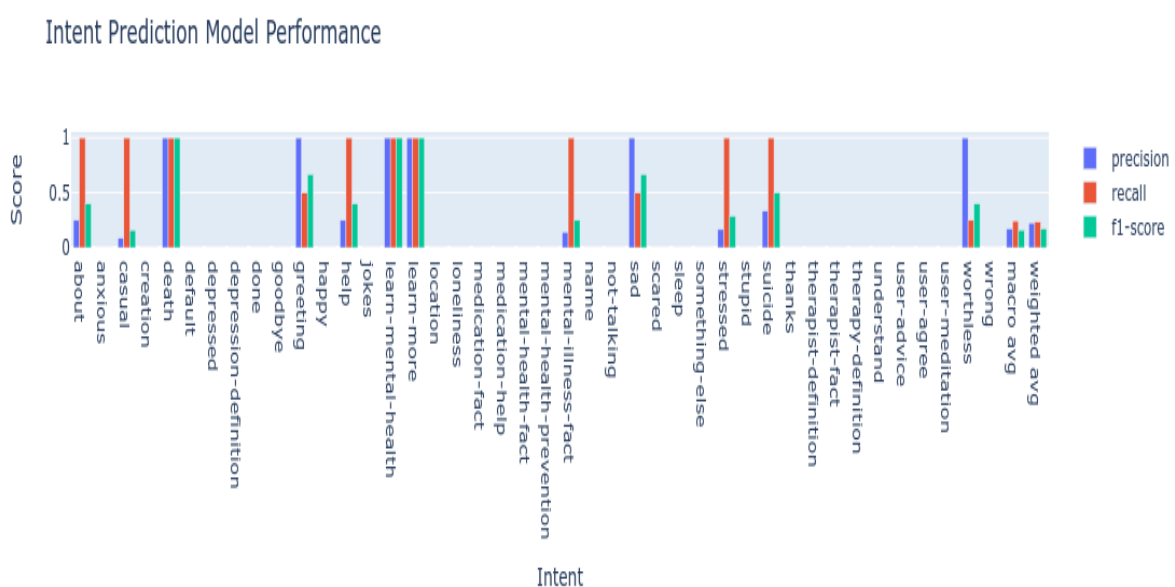
Con esos valores, el script entrena el modelo con el dataset *normal*. A continuación, comienza a leer entradas del dataset extra y las pasa al modelo entrenado

como si fuera la entrada del usuario. El modelo predice la etiqueta y ofrece una respuesta, pero además pregunta al usuario si la etiqueta es correcta y ofrece las siguientes opciones:

- Si es correcta, permite añadir al dataset original la respuesta del dataset extra.
- Si no es correcta, presenta la lista completa numerada de etiquetas generadas a partir del dataset con el que se ha entrenado el modelo, y pide al usuario que indique el número de etiqueta correcta, o -1 si no hay ninguna aplicable.
 - Si hay una etiqueta correcta, se añade al dataset normal el par de pregunta y respuesta del dataset extra.
 - Si no hay ninguna etiqueta apropiada, se permite añadir una nueva etiqueta y se incorpora esta, junto con el par de pregunta y respuesta del dataset extra.

Este proceso se repite para el número de iteraciones indicado, tras lo cual se renombra a **intents-old.json** el fichero JSON del dataset normal con el que se ha entrenado el modelo y se vuelca como nuevo fichero **intents.json** el que hay en memoria con las entradas añadidas, presentando un pequeño resumen y recordando cuál sería la siguiente línea inicial a partir de la cual procesar el dataset extra.

Se procesaron de esta forma 100 entradas del dataset extra PrinceAyush. Tras ello, el modelo presenta estos datos de rendimiento:



Puede observarse que, si bien sigue existiendo una irregularidad alta, al menos hay un mayor número de etiquetas con resultados y la media ponderada ha subido ligeramente.

Agregación automática de datasets

Como el tiempo para hacer el etiquetado manual resulta demasiado largo, se han explorado otras vías. Se ha preparado un script (esta vez no como notebook) que usa PyTorch y Transformers. Mediante el uso de inferencia por el método **pipelines()**, con la tarea “**zero-shot-classification**”, el modelo por defecto Distilbert base en inglés (<https://huggingface.co/distilbert/distilbert-base-uncased-finetuned-sst-2-english>) y pasándole la colección de etiquetas del modelo original más el resultado de alimentarlo manualmente con las primeras 100 entradas del dataset extra de PrinceAyush, se ha procedido a etiquetar automáticamente las siguientes 550 entradas. No se ha completado el tratamiento de las 5000 entradas del fichero porque el tiempo de proceso en los equipos disponibles ha resultado ser muy alto (más de un minuto, de media, por entrada):

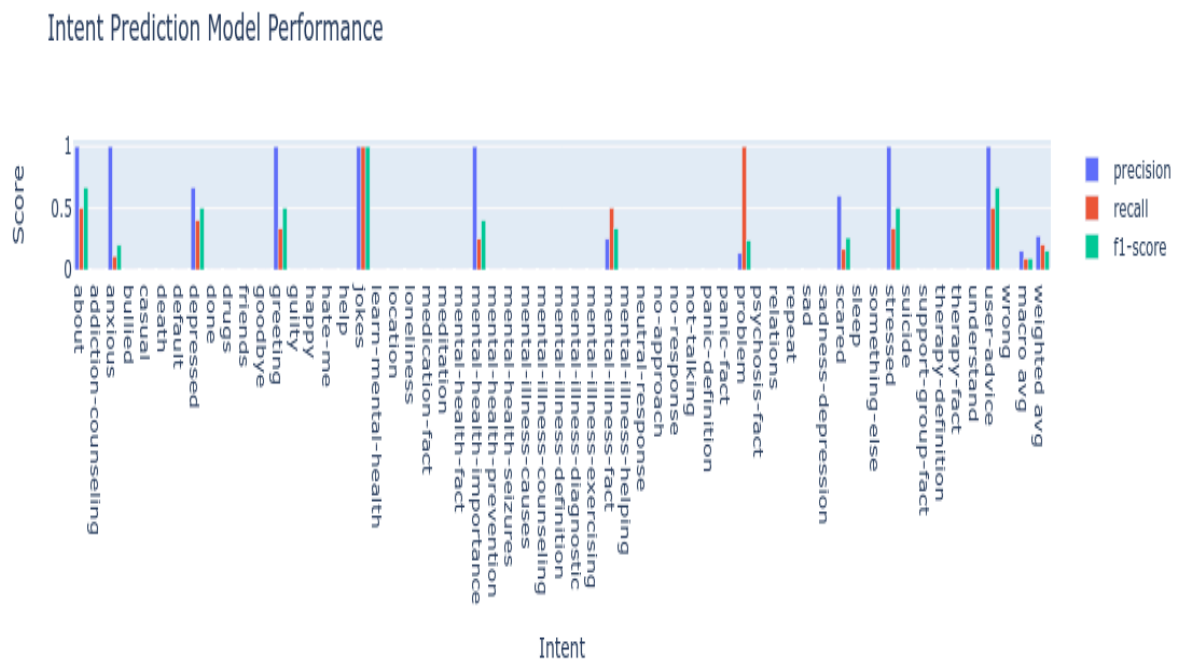
```
Number of inputs processed: 550
Elapsed time: 12:06:46.368051 seconds
Average process time for each input: 0:01:19.284306
```

El resultado no ha sido perfecto, principalmente debido a tres factores:

- El dataset original incluye etiquetas un tanto genéricas, tales como “ask”, “problem” y similares. De hecho, hubo que eliminar la entrada con etiqueta “ask” del dataset original porque todas las entradas del otro dataset que consistían en una pregunta se etiquetaban como “ask”.
- El dataset extra incluye contenido heterogéneo: hay bloques de entradas en las que la pregunta no es tal, sino una frase motivacional, con una respuesta que es continuación de la anterior. Estas entradas han debido ser suprimidas.
- El dataset extra incluye contenido preciso que no está contemplado, ni de forma cercana, por el dataset original, tal como cuestiones relacionadas con la intimidad sexual de adolescentes en sus primeras relaciones. Al no haber entradas similares en el dataset original, el etiquetado de este tipo de entradas es, necesariamente, incorrecto (se recuerda que el script no trata de generar nuevas etiquetas).

Generación de conversaciones

Con el dataset en formato original incrementado en su tamaño casi por 20, se ha desarrollado otro notebook, a partir del de agregación manual, que inyecta las entradas del dataset extra elegido, predice la intención, genera una respuesta y va generando entradas en el formato JSONL requerido para fine-tuning. Con el dataset de entrenamiento generado por el script de etiquetado automático, el rendimiento ha resultado ser este:



Aunque el número de etiquetas diferentes dificulta un tanto su lectura, y aunque se mantienen muchas etiquetas sin valores en la gráfica, se aprecia una disminución en los valores de recuerdo (recall).

Instrucciones para instalación y ejecución

ChatBot intermedio sin tag

En este apartado se explica cómo utilizar cualquiera de los modelos explicados anteriormente.

Carga de los json

Este apartado no es necesario porque los modelos ya están creados, es principalmente por si se quiere probar la carga de los ficheros. Los ficheros deberían estar en el mismo directorio donde está el notebook y son `data_NoEtiqueta_1.json`, `data_NoEtiqueta_2.json` y `data_NoEtiqueta_3.json`.

En la configuración entregada, como está actualmente el notebook, este está pensado para cargar todos los datasets mezclados. Si se quiere cargar solo uno de los ficheros se recomienda comentar todos los `with open(...)` menos el del fichero que se quiera probar, cambiar `init_df_x` a `init_df` y comentar desde `frames = [...]` hasta el ultimo `init_df.dropna(...)`.

```
#ESTE BLOQUE ES PARA CARGAR LOS DATASETS.
import json
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)
import os

#X = init_df['input'] #data_Jesus
#y = init_df['output'] #data_Jesus
#X = init_df['questionTitle'] #data_Jesus_2
#y = init_df['answerText'] #data_Jesus_2
#X = init_df['instruction'] #data_Jesus_3
#y = init_df['output'] #data_Jesus_3

with open('data_Jesus.json', 'r') as f:
    data = json.load(f)
    init_df_1 = pd.DataFrame(data)
    init_df_1 = init_df_1.drop_duplicates(subset = ['input']) #data_Jesus
    init_df_1 = init_df_1.rename(columns={'input': 'X', 'output': 'Y'})

with open('data_Jesus_2.json', 'r') as f:
    data = json.load(f)
    init_df_2 = pd.DataFrame(data)
    init_df_2 = init_df_2.drop_duplicates(subset = ['questionTitle']) #data_Jesus_2
    init_df_2 = init_df_2.rename(columns={'questionTitle': 'X', 'answerText': 'Y'})

with open('data_Jesus_3.json', encoding="utf8") as f:
    data = json.load(f)
    #init_df = pd.DataFrame(data['intents'])
    init_df_3 = pd.DataFrame(data)
    init_df_3 = init_df_3.drop_duplicates(subset = ['instruction']) #data_Jesus_3
    init_df_3 = init_df_3.rename(columns={'instruction': 'X', 'output': 'Y'})

frames = [init_df_1, init_df_2, init_df_3]

init_df = pd.concat(frames)
init_df = init_df.drop_duplicates(subset = ['X'])
init_df = init_df.drop_duplicates(subset = ['Y'])
init_df.dropna(subset=['X'], inplace=True)
init_df.dropna(subset=['Y'], inplace=True)
init_df
```

Carga de librerías

El siguiente bloque sí es necesario ejecutarlo siempre porque importa todas las librerías necesarias para que todo funcione.

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.svm import SVC
from sklearn.metrics import classification_report
import plotly.graph_objects as go
from joblib import dump, load
from sklearn import svm
from sklearn import datasets

#X = init_df['input'] #data_Jesus
#y = init_df['output'] #data_Jesus
#X = init_df['questionTitle'] #data_Jesus_2
#y = init_df['answerText'] #data_Jesus_2
#X = init_df['instruction'] #data_Jesus_3
#y = init_df['output'] #data_Jesus_3
X = init_df['X'] #data_Jesus_3
y = init_df['Y'] #data_Jesus_3
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, random_state=42)

# Vectorize the text data using TF-IDF
vectorizer = TfidfVectorizer()
X_train_vec = vectorizer.fit_transform(X_train)
X_test_vec = vectorizer.transform(X_test)
```

Carga y guardado de modelos

Los siguientes dos bloques sirven para guardar y cargar respectivamente los modelos. Recomendamos usar el de cargar para los modelos `model_Jesus_1_v1.joblib`, `model_Jesus_2_v1.joblib`, `model_Jesus_3_v1.joblib` y `model_Jesus_4_v1.joblib` porque ahorra mucho tiempo y se trabaja directamente con los modelos que nosotros hemos usado (los modelos se incluyen en el repositorio comprimidos en un archivo ZIP, debido a su tamaño).

```
#GUARDAR MODELO con el nombre que desees
dump(model, 'model_Jesus_4_v1.joblib')
```

```
['model_Jesus_4_v1.joblib']
```

```
#CARGAR MODELO usando el nombre del fichero y el path si no esta en el directorio actual.
model = load('model_Jesus_4_v1.joblib')
```

Entrenamiento

El siguiente bloque se encarga de hacer el entrenamiento. No se recomienda ejecutar este bloque porque puede tardar bastante tiempo y no hace falta si se usa el bloque de carga del modelo, que se ha explicado en el bloque anterior.


```
# ENTRENAMIENTO Y PREPARACION DEL MODELO
model = SVC()
model.fit(X_train_vec, y_train)

# Predict intents for the testing set
y_pred = model.predict(X_test_vec)

# Evaluate the model's performance
report = classification_report(y_test, y_pred, output_dict=True, zero_division=0)
report
```

Bloques del chat

Estos dos últimos bloques se utilizan para arrancar el ChatBot si los pasos anteriores se han seguido correctamente y es necesario ejecutar ambos si se quieren probar los modelos.

```
import random
# Prediction Model Deployment

# Function to predict intents based on user input
def predict_intent(user_input):
    # Vectorize the user input
    user_input_vec = vectorizer.transform([user_input])

    # Predict the intent
    intent = model.predict(user_input_vec)[0]

    return intent
```

```
# Example usage
while True:
    # Get user input
    user_input = input("User: ")

    # genera la respuesta
    respuesta = predict_intent(user_input)

    print("Chatbot:", respuesta)
```

ChatBots intermedios con tag

01-chatbot-for-mental-health-aggregate.ipynb

Este notebook Python, variación del notebook original, permite agregar entradas al dataset original desde otro dataset distinto.

Para su ejecución en un entorno Jupyter requiere:

- Un fichero intents.json en la misma carpeta que el notebook con el dataset en formato original usado como partida.
- Uno o varios ficheros con datasets adicionales en formato JSON que tengan una colección de entradas de diccionario con al menos un elemento que contenga la entrada de usuario y otro con la respuesta del asistente correspondiente. El fichero elegido y los nombres de las claves de diccionario se preguntan al operador durante la ejecución del notebook.

El funcionamiento es similar a este:



```
import fnmatch
for dirname, _, filenames in os.walk('.'):
    for filename in filenames:
        if fnmatch.fnmatch(filename, '*.json'):
            print(filename)

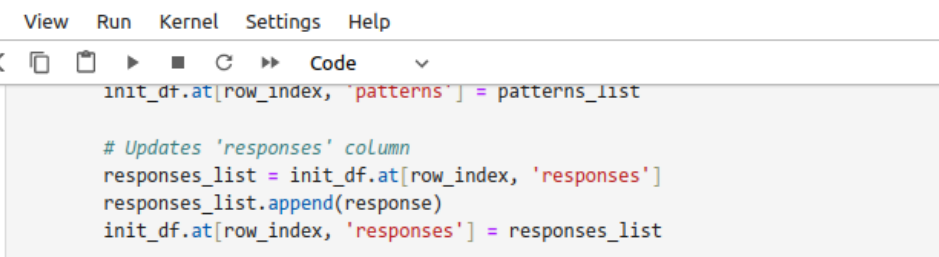
cl_output_file.json
cl_output_file_test.json
intents-auto.json
intents-old.json
cl_output_file_formatted.json
intents-initial.json
amod_mental_health_counseling_conversation.json
cl_output_file_formatted_short.json
intents-last-manual-addition.json
intents.json
original_intents.json

[*]: # We're adding knowledge to intents.json by injecting prompt-response pairs from an extra dataset
extra_dataset = input("Enter another dataset file to be added: ")
Enter another dataset file to be added: cl_output_file_formatted.json
```

Debe conocerse el formato del dataset adicional para identificar las claves de diccionario de las entradas que se procesarán. Por ejemplo, la siguiente imagen muestra la estructura del archivo `cl_output_file_formatted.json`. Debe tenerse en cuenta que en algunas entradas todas las claves tienen valor y en otras es `questionTitle` la que contiene la pregunta:

JSON	Datos sin procesar	Cabeceras
<div> <div> Guardar Copiar Contraer todo Expandir todo (lento) Filtrar JSON </div> </div>		
▶ 1274:	{...}	
▶ 1275:	{...}	
▶ 1276:	{...}	
▶ 1277:	{...}	
▶ 1278:	{...}	
▶ 1279:	{...}	
▼ 1280:	<div> <div>questionTitle:</div> <div>"</div> </div> <div> <div>▼ questionText:</div> <div>"He's gone for 11 weeks for a band camp. It's week two right now. We used to be Intimate phone. Is it normal for me to more sensitive in general?"</div> </div> <div> <div>▼ answerText:</div> <div>"Well yes, physical intimacy does give all sorts of positive feelings. The sudden withdr</div> </div>	
▼ 1281:	<div> <div>questionTitle:</div> <div>"</div> </div> <div> <div>▼ questionText:</div> <div>"My husband had an emotional affair with his ex-wife in November. She invited him to dir was. He had been drinking, and I told him not to come home that night. The next morning, wasn't going. I found out in April that she did go. I gave him the chance to get everyth slept in the same bed, and he said yes, both nights. His daughter wasn't there the secon</div> </div> <div> <div>▼ answerText:</div> <div>"I agree with you that professional counseling is a wise choice for your relationship.Yc toward you and the ex.The topic is very heated, which is why discussing these matters al and get so upset the conversation goes off track and unintended hurts happen.In a therap be much more productive than trying to do so only with the two of you."</div> </div>	

El notebook solicita las claves de diccionario, la entrada que se quiere comenzar a procesar y el número de entradas del stint:



The screenshot shows a Jupyter Notebook window titled "jupyter chatbot-for-mental-health-aggregate" with a "Last Checkpoint: 6 days ago" status. The interface includes a top menu bar with "File", "Edit", "View", "Run", "Kernel", "Settings", and "Help". Below the menu is a toolbar with icons for saving, adding, deleting, copying, pasting, running, and other actions. The main area displays a code cell with the following Python code:

```
init_df.at[row_index, 'patterns'] = patterns_list

# Updates 'responses' column
responses_list = init_df.at[row_index, 'responses']
responses_list.append(response)
init_df.at[row_index, 'responses'] = responses_list

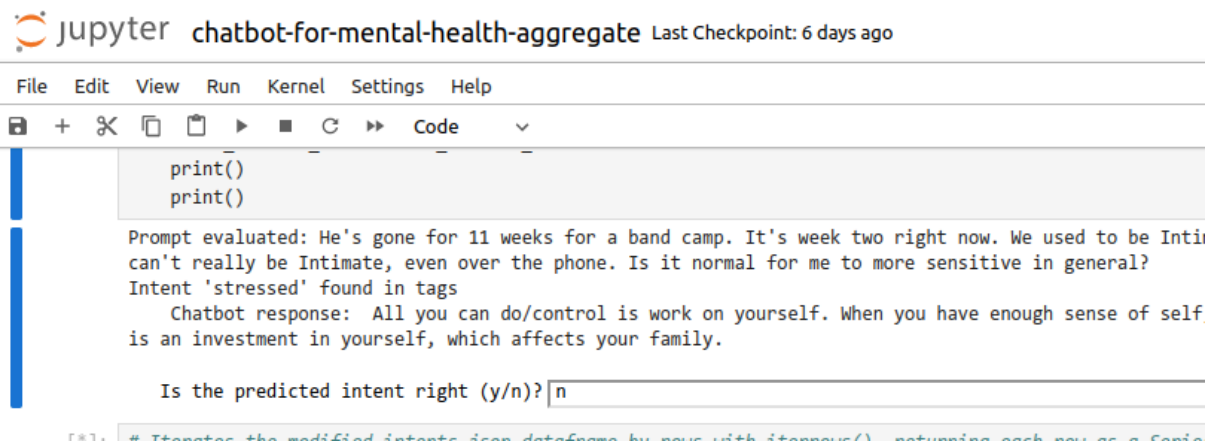
return init_df
```

Below the code cell, the execution output is shown, starting with a prompt character "[*]:". The output displays the input prompts and the user's responses:

```
[*]: extra_pattern_column_name = input("Enter the name of column acting as pattern / user prompt: ")
      extra_response_column_name = input("Enter the name of column acting as response: ")
      extra_init_row = eval(input("Enter initial row to evaluate: "))
      extra_stint_length = eval(input("Enter length of row stint (how many rows to evaluate now): "))

      Enter the name of column acting as pattern / user prompt: questionText
      Enter the name of column acting as response: answerText
      Enter initial row to evaluate: 1280
      Enter length of row stint (how many rows to evaluate now): 10
```

A partir de ese momento se inyectan en el modelo de clasificación del chatbot original las entradas del dataset adicional, se predice la etiqueta y se calcula una respuesta según esta etiqueta. Se presentan ambos datos al operador y se pregunta si la predicción es correcta:



```

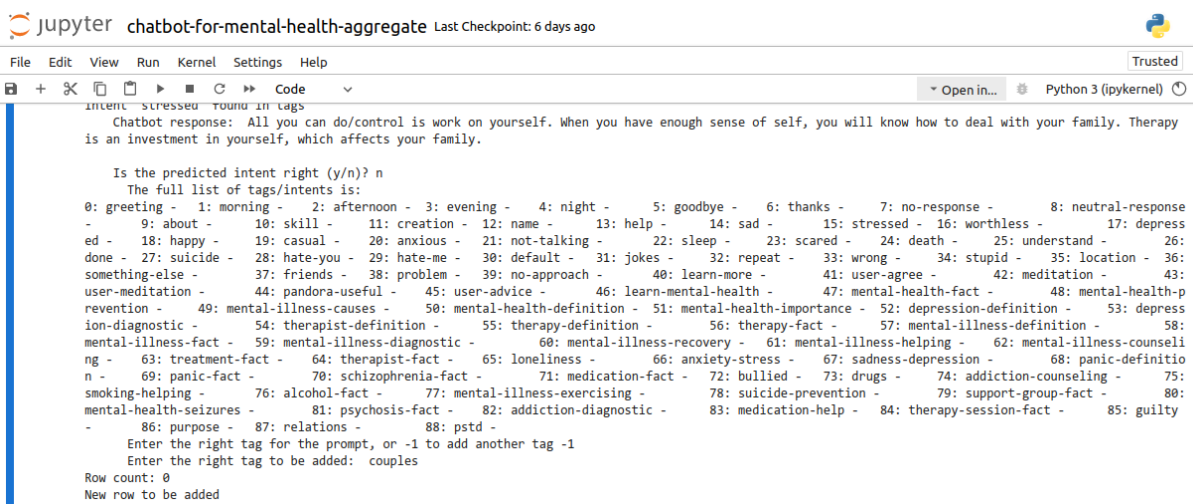
print()
print()

Prompt evaluated: He's gone for 11 weeks for a band camp. It's week two right now. We used to be Inti
can't really be Intimate, even over the phone. Is it normal for me to more sensitive in general?
Intent 'stressed' found in tags
Chatbot response: All you can do/control is work on yourself. When you have enough sense of self
is an investment in yourself, which affects your family.

Is the predicted intent right (y/n)? n

```

Si se indica que no, se presenta la lista de etiquetas y se pide el número de la correcta. Puede indicarse -1 si ninguna se ajusta a la entrada de usuario y es necesario añadir otra etiqueta al dataset original:



```

print()
print()

Chatbot response: All you can do/control is work on yourself. When you have enough sense of self, you will know how to deal with your family. Therapy
is an investment in yourself, which affects your family.

Is the predicted intent right (y/n)? n
The full list of tags/intents is:
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: depress
18: happy - 19: casual - 20: anxious - 21: not-talking - 22: sleep - 23: scared - 24: death - 25: understand - 26:
done - 27: suicide - 28: hate-you - 29: hate-me - 30: default - 31: jokes - 32: repeat - 33: wrong - 34: stupid - 35: location - 36:
something-else - 37: friends - 38: problem - 39: no-approach - 40: learn-more - 41: user-agree - 42: meditation - 43:
user-meditation - 44: pandora-useful - 45: user-advice - 46: learn-mental-health - 47: mental-health-fact - 48: mental-health-p
49: mental-illness-causes - 50: mental-health-definition - 51: mental-health-importance - 52: depression-definition - 53: depress
54: therapist-definition - 55: therapy-definition - 56: therapy-fact - 57: mental-illness-definition - 58:
mental-illness-fact - 59: mental-illness-diagnostic - 60: mental-illness-recovery - 61: mental-illness-helping - 62: mental-illness-counseli
ng - 63: treatment-fact - 64: therapist-fact - 65: loneliness - 66: anxiety-stress - 67: sadness-depression - 68: panic-definitio
n - 69: panic-fact - 70: schizophrenia-fact - 71: medication-fact - 72: bullied - 73: drugs - 74: addiction-counseling - 75:
smoking-helping - 76: alcohol-fact - 77: mental-illness-exercising - 78: suicide-prevention - 79: support-group-fact - 80:
mental-health-seizures - 81: psychosis-fact - 82: addiction-diagnostic - 83: medication-help - 84: therapy-session-fact - 85: guilty
86: purpose - 87: relations - 88: pstd
Enter the right tag for the prompt, or -1 to add another tag -1
Enter the right tag to be added: couples
Row count: 0
New row to be added

```

Al terminar la ronda, se renombra intents.json a intents-old.json y se guarda una nueva versión de intents.json con el contenido añadido.

02-automatic_tagging.py

Script Python que inyecta prompts de usuario del dataset `cl_output_file_formatted_short.json` en un modelo Distilbert con la tarea de clasificación de texto **zero-shot-classification** y la colección de etiquetas del dataset en el formato original (idealmente, tras haber mejorado su colección de etiquetas). Al terminar, guarda un fichero `intents-auto.json` que puede ser renombrado a `intents.json` para su uso con los notebooks de conversación, agregación o generación del JSONL.

Para su ejecución en consola o en un entorno de desarrollo como Spyder requiere:

- Un fichero intents.json en la misma carpeta que el notebook con el dataset en formato original usado como partida.
- El fichero cl_output_file_formatted_short.json con una versión parcial del dataset adicional en formato JSON. Podría usarse la versión completa modificando el nombre del fichero en la línea 69 del script.

```
62
63
64     with open('intents.json', 'r') as f:
65         data = json.load(f)
66     init_df = pd.DataFrame(data['intents'])
67     tags = init_df['tag']
68
69     with open('cl_output_file_formatted_short.json', 'r') as f:
70         dataset_extra = json.load(f)
71     extra_df = pd.DataFrame(dataset_extra)
72     extra_df = extra_df.dropna(subset = ['questionTitle'])
73     extra_df = extra_df.drop_duplicates(subset = ['questionTitle'])
74     patterns = extra_df['questionTitle']
75
```

Una vez iniciada la ejecución el funcionamiento es desatendido. Se van mostrando las entradas procesadas con el etiquetado generado por el pipeline. Al terminar, se guarda el fichero intents-auto.json con el dataset en formato original resultante y se presenta un resumen de las entradas procesadas y el tiempo empleado.

03-chatbot-for-mental-health-generate-jsonl.ipynb

Notebook Python que genera pares de pregunta-respuesta mediante el chatbot original y los guarda en el formato de fine-tuning necesario para GPT3.5turbo.


Para ello, el notebook carga el dataset en formato original intents.json, tras lo cual solicita un dataset adicional al del notebook original, el nombre de las columnas que representan la pregunta del usuario y la respuesta del asistente, un número de elemento en el que comenzar y un número de iteraciones (opcionalmente todas las que contenga el dataset adicional). Entonces va inyectando como prompt de usuario líneas del dataset adicional, prediciendo la etiqueta con el clasificador del notebook original y guardando en el archivo conv-ft2.jsonl el par prompt - respuesta en el formato JSONL que requiere GPT-3.5turbo para fine-tuning.

Para su ejecución en un entorno Jupyter requiere:

- Un fichero intents.json en la misma carpeta que el notebook con el dataset en formato original usado como partida.

- Un fichero con el dataset adicional en formato JSON que tenga una colección de entradas de diccionario con al menos un elemento que contenga la entrada de usuario y otro con la respuesta del asistente correspondiente. Los nombres de las claves de diccionario se preguntan al operador durante la ejecución del notebook.

Tras solicitar los datos de forma análoga al notebook de agregación manual, se lleva a cabo la generación del fichero JSONL de fine-tuning. El proceso es rápido, como puede verse en el resumen que aparece al finalizar aquella.



```
[24]: et = datetime.datetime.now()
print("conv-ft2.jsonl saved")
print("Round summary:")
print(f"    Extra dataset filename: {extra_dataset}")
print(f"    Initial row in extra dataset: {extra_init_row}")
print(f"    Stint length: {extra_stint_length}")
print(f"    Process time: {(et - st)}")
print()

mental-health-fine-tuning.jsonl saved
Round summary:
    Extra dataset filename: cl_output_file.json
    Initial row in extra dataset: 0
    Stint length: 5016
    Process time: 0:00:18.734245
```

ChatBot Fine-Tuning

El objetivo final de este proyecto, una vez obtenidos los resultados de los modelos Python, es la creación de diferentes datasets en formato .jsonl (JSON Line) para entrenar modelos GPT en la plataforma OpenAI gestionada por Microsoft Azure.

A través del proceso de afinamiento (en inglés fine-tuning) vamos a modificar las capas internas de la red neuronal de nuestra *instancia* de modelo, en este caso un modelo GPT 3.5 Turbo versión 0124. ¿Por qué utilizar el fine tuning? Gracias a este proceso, podemos evitar entrenar desde cero un modelo nuevo, ahorrando en tiempo, recursos y datos necesarios para construir una base de conocimiento lo suficientemente robusta. El proceso de fine-tuning, de hecho, sólo modifica parte del modelo preentrenado: la base de conocimiento (si bien sufre algunos cambios) se queda intacta. Nuestro objetivo es que el modelo sufra un reajuste de pesos internos, favoreciendo sólo aquellas capas que nos permitan alcanzar el objetivo deseado.

Entrenamiento

Primero, seleccionamos un modelo como base para nuestro reentrenamiento: en nuestro caso, hemos elegido el modelo GPT (Generative Pre-trained Transformer) en su versión 3.5 Turbo 0124 por su extensa base de conocimiento. Ya que nuestro objetivo es una IA capaz de comunicarse y de tener conversaciones, parte fundamental del resultado es que nuestra IA sepa siempre qué contestar y cómo: la fase de afinamiento nos permite “otorgar” a la IA un enfoque, un carácter (que es algo importante si queremos simular una interacción entre dos personas).

Para entrenar el modelo, hemos construido un dataset en formato .jsonl (que es el formato aceptado por OpenAI) a partir de las conversaciones y de los patrones de respuestas generados por los modelos anteriores en Python.

Hay dos tipos distintos de entrenamiento posibles para un modelo conversacional como es el GPT: fine tuning con patrones de respuesta estándar o pesado. En ambos casos, es necesario especificar un “contexto” general para los dos. Sirve de guía para nuestra IA, de manera que se fije en un enfoque específico. Ejemplo:

“You’re an AI assistant that likes to make a lot of jokes”

En este caso estamos describiendo una IA que suele hacer chistes al hablar, algo que influencia la manera de comunicarse al recibir una petición del usuario. Es probable por lo tanto, que las respuestas contengan chistes de diferentes tipos, según la definición de “chiste” aprendida por la IA.

- Estándar
 - Sigue la estructura: Contexto + Pregunta + Respuesta
 - Ejemplo:
 - USER: *“I’m suffering from panic attacks”*
 - IA: *“I’m so sorry to hear that. Do you want to tell me more about this?”*

- Pesada
 - Sigue la estructura: Contexto + Pregunta + Respuestas Pesadas
 - Ejemplo:
 - USER: *“I’m suffering from panic attacks”*
 - IA: *“I’m so sorry to hear that. Do you want to tell me more about this?”*
+ 0.8
 - IA: *“That’s a very hard thing to say. Are you sure you are experiencing panic attacks?”* + 0.2
 - En este caso, la respuesta con el peso más alto es la respuesta preferida; sin embargo es posible que debido a factores causales, la respuesta se parezca a la segunda con peso 0.2

Cargado el fichero .json en la plataforma de OpenAI y escogido el modelo, podemos empezar el entrenamiento que tardará una cantidad de tiempo en función del tamaño del dataset utilizado. Finalizado el entrenamiento, obtendremos un fichero Excel con el resultado de la sesión de fine tuning. Sucesivamente, el modelo recién entrenado aparecerá en la lista de modelos desplegados para que pueda ser utilizado según lo planteado.

Resultados

Podemos ver en general, que la pérdida (loss) va descendiendo en cada interacción hacía el 0, es decir que estamos cada vez afinando el modelo con más precisión.

Además de las métricas, para comprobar que nuestro fine tuning ha producido los resultados esperados, hemos comparado las respuestas recibidas entre un modelo normal sin fine tuning (el mismo chatgpt) y nuestro modelo. Hemos notado un cambio de enfoque importante: nuestro modelo, al recibir una pregunta como “He sufrido un ataque de pánico” pide al usuario más detalles, haciendo preguntas e incentivando una conversación mientras que el modelo sin fine tuning, corta la conversación, recomendando al usuario acudir a un profesional de la salud mental.

Despliegue y prueba

Los modelos, una vez creados, hay que desplegarlos para que se puedan utilizar. El proceso de despliegue es parecido al proceso de afinamiento: seleccionamos un modelo, le asignamos un nombre y fijamos algunos parámetros. Una vez desplegado, podemos interactuar con el modelo a través de llamadas APIs o utilizando el Playground de OpenAI.

El Playground consta con una interfaz que simula un chat y permite cambiar algunos parámetros, como por ejemplo el contexto del chat (para modificar su comportamiento, como se ha explicado anteriormente).

Algunos parámetros adicionales a tener en cuenta son la **Temperatura** y el **Punto K**. La **Temperatura** está relacionada con la *certidumbre* de las respuestas: es decir, cuanto más alta sea la temperatura, más probable será que la respuesta obtenida sea menos afín al modelo entrenado (incertidumbre). Mientras más baja sea la temperatura, más *segura* será la respuesta. El Punto K, análogamente, cambia la manera en la que interaccionan las capas.

Utilizando el Playground, hemos probado el modelo y comprobado la eficacia del fine tuning realizado.

Conclusiones

En cuanto al fine-tuning, es evidente que es un proceso que supone un gasto que no se puede ignorar, al ser elevado especialmente cuando se trata de entrenar más modelos. Además, en general cualquier interacción con el modelo supone gastos que hay que calcular para medir la rentabilidad del proyecto. Seguramente hacen falta mucho más datos de los generados actualmente, sin embargo estamos satisfechos con el haber cambiado el enfoque original del modelo de Chat GPT.