PsychoGPT - Manual Técnico

Guía técnica del proyecto

02/06/2023
CES JUAN PABLO II
Álvaro Sánchez Ruiz, Iván Lumbreras Martín

Índice

Contenido

INTRODUCCIÓN

OBJETIVO DEL MANUAL.

El objetivo de este manual técnico es proporcionar una guía completa y detallada sobre el desarrollo e implementación de una aplicación basada en inteligencia artificial (IA) utilizando el modelo de Huggingface y su integración con una aplicación móvil en Flutter. El manual está diseñado para ayudar a los desarrolladores a comprender el proceso desde la preparación de los datos y el entrenamiento del modelo mediante fine-tuning, hasta la generación de una API y la creación de la interfaz de usuario en Flutter.

BREVE DESCRIPCIÓN DE IA.

Mediante el proceso de fine-tuning usando el modelo pre-entrenado con más de 1.4millones de parámetros. Utilizando varias bases de datos de psicología con conversaciones paciente-doctor para entrenarlo.

BREVE DESCRIPCIÓN DE LA APLICACIÓN EN FLUTTER.

Con Flutter hemos desarrollado una aplicación multiplataforma que cuenta con un inicio de sesión y un registro conectados a Firebase, además de la propia funcionalidad de la aplicación para hacer consultas a la Api, acceder al historial de consultas y ver información del perfil.

PREPARACIÓN DE DATOS

RECOPILACIÓN DE LOS DATOS.

Para la recopilación de los datos, investigamos en distintos sitios web sobre bases de datos con conversaciones públicas psicólogo-paciente, desgraciadamente no hay demasiadas por Google, ya que compartir este tipo de conversaciones doctor-paciente es ilegal. Entonces descubrimos una web llamada HuggingFace en la cual se encuentran muchísimas bases de datos públicas, y daba la casualidad de que había una específicamente para psicología. La cual disponía de la estructura perfecta para ser usada input-output. Descargamos el JSON y por el momento lo almacenamos.

PROCESAMIENTO DE DATOS PREVIO AL ENTRENAMIENTO.

Para el procesamiento de datos usamos este código.

```python
class ChatData(Dataset):
    def __init__(self,path:str,tokenizer):
      # Carga los datos desde el archivo JSON
        self.data = json.load(open(path,"r"))

        self.X = []
        for i in self.data:
          # Obtiene los textos de entrada y salida del chat
            self.X.append(i['input'])
            self.X.append(i['output'])

          # Concatena los textos de entrada y salida en un solo texto,
          # agrega marcas especiales para indicar el inicio y el final del texto,
          # y lo almacena en la lista self.X
        for idx,i in enumerate(self.X):
            try:
                self.X[idx] = "<startofstring> "+i+" <bot>: "+self.X[idx+1]+" <endofstring>"
            except:
                break

        self.X = self.X[:-1]
        # Codifica los textos utilizando el tokenizer proporcionado
        self.X_encoded = tokenizer(self.X,max_length=40, truncation=True, padding="max_length", return_tensors="pt")
        self.input_ids = self.X_encoded['input_ids']
        self.attention_mask = self.X_encoded['attention_mask']

    def __len__(self):
      # Devuelve la longitud de los datos
        return len(self.X)

    def __getitem__(self,idx):
       # Devuelve un par de tensores que representan el input_ids y la atención para un ejemplo en particular
        return (self.input_ids[idx],self.attention_mask[idx])
```

1.Para empezar tenemos que cargar el JSON previamente descargado

2.Una vez hecho tenemos que extraer del JSON los input-output.

3. Concatena los textos y agrega las marcas especiales para indicar el inicio y fin del texto, para que el modelo los entienda.

4. Después una vez tenemos la lista bien ordenada y limpia, tenemos que usar un tokenizador para que el modelo sea capaz de entender el json y sea capaz de aprender de esas frases.

5. Almacenar los tensores de input_aids y attention_mask para usarlos posteriormente.

ENTRENAMIENTO DEL MODELO (FINE-TUNING)

ELECCIÓN DEL MODELO BASE DE HUGGINGFACE.

Para la elección del modelo igual usamos la misma web de HuggingFace ya que dispone de modelos base completamente gratuitos, ya depende tus recursos computacionales, es decir, cuantos mas recursos tengas podrás usar un modelo con más parámetros y por ende más precisa será la pregunta y mas tipos de preguntas admitirá. Como nuestros recursos eran pequeños y limitados ya que usamos Google Colab probamos varios modelos hasta encontrar el que mejor se adaptada a nuestra situación como dialoGPT-medium de Microsoft o blenderbot-400mb-small de Facebook. Al final decidimos pagar por Google colab pro y usar gpt2 con unos 1.5 millones de parámetros ya que consideramos que los otros no nos proporcionaban respuestas con demasiada coherencia al ser modelos tan pequeños.

CONFIGURACIÓN DEL ENTORNO DE ENTRENAMIENTO.

¡Para su configuración simplemente tendremos que poner la línea! pip install y el módulo que queramos añadir. De esta manera:

```
#en google colab se usa esta estructura para instalarlos
!pip install transformers
#imports para las librerias que vamos a utilizar
```

FINE-TUNING DEL MODELO CON EL ARCHIVO JSON.

Nosotros para este proceso usaremos esta función

```
def train(chatData, model, optim):
    # Función de entrenamiento del modelo
    epochs = 12
    # Número de épocas de entrenamiento
    #Cuantas mas epocas o epochs mas preciso será pero más recursos computacionales consume, puede ocurrir el sobreajuste (0 precisión en la respuesta)

    for i in tqdm.tqdm(range(epochs)):
        # For para a recorrer las épocas(epochs) utilizando la libreria tqdm para mostrar una barra de progreso
        for X, a in chatData:
            X = X.to(device)
            a = a.to(device)
            # Mover los datos a la GPU si está disponible ya que por GPU es mas eficiente
            optim.zero_grad()
            # Reiniciar los gradientes acumulados
            loss = model(X, attention_mask=a, labels=X).loss
            # Calcular la pérdida del modelo, la perdida sirve para monitorizar y ver la precisión de las respuestas
            loss.backward()
            # Retropropagación de la pérdida para calcular los gradientes. Este paso es esencial para poder ajustar los pesos del modelo y mejorar su rendimiento.
            optim.step()
            # Actualización de los pesos del modelo utilizando el optimizador

        torch.save(model.state_dict(), "/content/drive/MyDrive/IA/model_state/model_state2.pt")
        model.config.save_pretrained("/content/drive/MyDrive/IA/model_config/")
        tokenizer.save_pretrained("/content/drive/MyDrive/IA/tokenizer_info/")
        # Guardar el estado del modelo y los archivos relacionados como la configuracion del modelo y su tokenizador.

        print(infer("Me he estado sintiendo muy ansioso últimamente y no sé por qué."))
        print(infer("Da tres consejos para mantenerse saludable."))
        print(infer("Escribe una historia corta en tercera persona sobre un protagonista que tiene que tomar una decisión de carrera importante."))
        print(infer("Me siento tan solo en mi vida, no tengo a nadie con quien hablar."))
        # Ejecutar una funcion que va generando respuestas a estas preguntas para ver como va mejorando la calidad de las respuestas según se entrena
```

El proceso de fine-tuning implica ajustar un modelo de lenguaje preentrenado para que se adapte a un dominio o tarea específica. En este caso, se realiza el fine-tuning utilizando un archivo JSON que contiene los datos del chat. El código carga los datos del archivo y los prepara para su procesamiento previo al entrenamiento. Los textos de entrada y salida se concatenan, se agregan

marcas especiales y se codifican utilizando el tokenizer. Luego, se realiza un bucle de entrenamiento donde se mueven los datos a la GPU (si está disponible), se calcula la pérdida del modelo, se retropropaga la pérdida y se actualizan los pesos del modelo utilizando un optimizador. Finalmente, se guardan el estado del modelo, la configuración y el tokenizador.

OPTIMIZACIÓN Y AJUSTE DE HIPERPARÁMETROS.

```
# Función de entrenamiento del modelo
epochs = 12
# Número de épocas de entrenamiento
#Cuantas mas epocas o epochs mas preciso será pero más recursos computacionales consume, puede ocurrir el sobreajuste (0 precisión en la respuesta)
```

```
optim = Adam(model.parameters(), lr=1e-4)
# Crear el optimizador Adam para ajustar los pesos del modelo con una tasa de aprendizaje de 1e-5 , cuanto mas pequeña es la medida mas preciso es pero mas recursos consume para entrenarlo
```

```
chatData = DataLoader(chatData, batch_size=256)
# Crear un DataLoader para cargar los datos de chat en lotes de tamaño 256
```

El proceso de optimización y ajuste de hiperparámetros es crucial para obtener un modelo de IA con un rendimiento óptimo. En este código, se utiliza el optimizador Adam con una tasa de aprendizaje de 1e-4 para ajustar los pesos del modelo durante el entrenamiento. La elección de la tasa de aprendizaje es importante, ya que determina la rapidez con la que el modelo aprende de los datos. Es posible ajustar este hiperparámetro para obtener mejores resultados. Además, se especifica un total de 12 épocas de entrenamiento, que determina cuántas veces se recorrerán todos los datos de entrenamiento. A medida que se realiza el entrenamiento, se pueden observar mejoras en la calidad de las respuestas generadas por el modelo. Por último, usamos un número de lotes de tamaño que se procesan en paralelo de 256, esto simplemente sirve para obtener una mayor precisión en la respuesta, pero cuanto mayor es mas tiempo y recursos consume.

EVALUACIÓN DEL MODELO ENTRENADO.

Después de completar el entrenamiento del modelo, es importante evaluar su desempeño. En este código, se utiliza una función "infer" para generar respuestas del modelo a diferentes preguntas. Esto permite evaluar la calidad de las respuestas y ver cómo mejora a medida que se entrena el modelo. Se muestran algunos ejemplos de preguntas y las respuestas generadas por el modelo entrenado.

```
# Función de inferencia utilizando el modelo entrenado
def infer(inp):
    inp = "<startofstring> " + inp + " <bot>: "
    inp = tokenizer.encode(inp, return_tensors="pt")
    inp = inp.to(device)
    output = model.generate(inp, max_length=100, num_return_sequences=1)
    output = tokenizer.decode(output[0])
    return output
```

GENERACIÓN DE LA API Y RESPUESTA DEL MODELO.

INSTALACIONES NECESARIAS PARA GOOGLE COLAB.

Para preparar el entorno simplemente usamos las siguientes líneas:

```
#Instalaciones necesarias para el google colab
"""
!pip install transformers
!pip install mtranslate
!pip install torch
!pip install langdetect
!pip install flask
!pip install flask_cors

!pip install pyngrok
"""

!ngrok authtoken 2QG1taYso6NHZZDuwVIFBMLk8kP_7rwSdYfNS3HZJ7XEeH1oN
```

IMPORTACIÓN DE BIBLIOTECAS NECESARIAS.

Una vez descargadas es necesario implementarlas para ello ejecutamos todos los imports que serán estos

```
# Importar las bibliotecas necesarias
from pyngrok import ngrok
from transformers import GPT2Tokenizer, GPT2LMHeadModel
from mtranslate import translate
import torch
from langdetect import detect
from flask import Flask, request, jsonify
from flask_cors import CORS
import os
```

CREACIÓN DE UNA INSTANCIA DE LA APLICACIÓN FLASK.

Para la creación de la instancia en flask simplemente debemos escribir esta línea para iniciar el servidor.

```
app = Flask(__name__)
```

HABILITACIÓN DE CORS.

Importante realizar este paso ya que sin esto nos dará error ya que debemos habilitar todas las peticiones, por defecto restringe las que no son seguras.

```
CORS(app, resources={r"/*": {"origins": "*"}})
```

DEFINICIÓN DE LA RUTA DEL ENDPOINT PARA GENERAR RESPUESTAS.

Importante establecer una ruta a la que recibir los datos, la cual establecemos con app.route

```
@app.route('/generate-response/<question>', methods=['POST'])
```

IMPLEMENTACIÓN DE LA GENERACIÓN DE RESPUESTAS.

```python
def generate_response(question):
  # Obtener la pregunta del cuerpo de la solicitud en formato JSON
    question = request.json['question']

    # Verificar si hay una GPU disponible para utilizarla, de lo contrario utilizar la CPU
    device = "cuda" if torch.cuda.is_available() else "cpu"

    # Detectar el idioma de la pregunta
    source_lang = detect(question)

    # Traducir la pregunta al inglés si no está en inglés
    if source_lang != "en":
        question = translate(question, "en")

    # Carga el tokenizador preentrenado GPT2 (Para que el modelo pueda entender la pregunta)
    tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
    tokenizer.add_special_tokens({"pad_token": "<pad>",
                                  "bos_token": "<startofstring>",
                                  "eos_token": "<endofstring>"})
    tokenizer.add_tokens(["<bot>:"])

    # Carga el modelo de GPT2
    model = GPT2LMHeadModel.from_pretrained("gpt2")
    model.resize_token_embeddings(len(tokenizer))
    #Carga el estado del modelo preentrenado con las bases de datos de piscologia al modelo de gpt2
    model.load_state_dict(torch.load("/content/drive/MyDrive/psycho.pt", map_location=device))
    model.to(device)
    model.eval()
```

```python
# Preprocesar la pregunta agregando los tokens especiales para que el modelo pueda entender la pregunta
input_text = "<startofstring> " + question + " <bot>: "
input_ids = tokenizer.encode(input_text, return_tensors="pt").to(device)

# Generar la respuesta utilizando el modelo
output = model.generate(input_ids, max_length=100, num_return_sequences=1)
output_text = tokenizer.decode(output[0], skip_special_tokens=True)

# Traducir la respuesta al idioma original de la pregunta si no está en inglés
if source_lang != "en":
    output_text = translate(output_text, source_lang, "en")

# Obtener el texto generado por el modelo eliminando el texto anterior a "<bot>:" y posterior a "<bot>:"
start_token = "<bot>:"
end_token = "<bot>:"
start_index = output_text.find(start_token)
end_index = output_text.find(end_token, start_index + len(start_token))
if start_index != -1 and end_index != -1:
    output_text = output_text[start_index + len(start_token):end_index].strip()

# Devolver la respuesta generada como JSON
return jsonify({'response': output_text})
```

1. Obtener la pregunta del cuerpo de la solicitud en formato JSON.
2. Verificar si hay una GPU disponible para utilizarla, de lo contrario, utilizar la CPU.
3. Detectar el idioma de la pregunta utilizando la función "detect".
4. Traducir la pregunta al inglés si no está en inglés utilizando la función "translate".
5. Cargar el tokenizador preentrenado GPT2 y agregar los tokens especiales necesarios para que el modelo pueda entender la pregunta.
6. Cargar el modelo de GPT2 preentrenado.
7. Cargar el estado del modelo preentrenado con las bases de datos de psicología al modelo de GPT2.
8. Mover el modelo a la GPU si está disponible.
9. Establecer el modelo en modo de evaluación.
10. Preprocesar la pregunta agregando los tokens especiales utilizando el tokenizador.
11. Generar la respuesta utilizando el modelo.
12. Decodificar la salida generada por el modelo y obtener el texto de la respuesta.
13. Traducir la respuesta al idioma original de la pregunta si no está en inglés.
14. Eliminar el texto anterior y posterior a "<bot>:" para obtener solo la respuesta generada.
15. Devolver la respuesta generada como JSON.

EJECUCIÓN DE LA APLICACIÓN FLASK CON NGROK.

como JSON. Una vez acabada esa función lo siguiente seria abrir el servidor para que reciba respuestas, pero en este caso hay que iniciarlo con un intermediario ya que por defecto Google Colab no tiene localhost.

```
if __name__ == '__main__':
    # Obtener el puerto aleatorio disponible en el entorno o utilizar el puerto 5000 por defecto
    port = int(os.environ.get('PORT', 5000))

    # Iniciar el túnel de Ngrok para crear una URL pública y accesible desde Internet
    # Este paso es necesario ya que desde google colab no se puede hacer el server flask ya que lo bloquea entonces usamos un intermediario.
    ngrok_tunnel = ngrok.connect(port)
    print('Public URL:', ngrok_tunnel.public_url)

    # Ejecutar la aplicación Flask utilizando la URL proporcionada por Ngrok
    app.run(host='0.0.0.0', port=port)
```

DESARROLLO DE LA APLICACIÓN EN FLUTTER.

Para la aplicación utilizable, se crea un proyecto en Flutter con varias pantallas, primeramente las de login y registro que se conectan a Firebase mediante las librerías pertinentes para Flutter (firebase_core y firebase_auth). Por otra parte, se crea una página home que deriva al usuario a la pantalla que elija al pulsar el menú inferior. Esto se logra creando 3 clases, una para cada pantalla con sus respectivos contenidos y se redirigen mediante el menú a la pantalla Home. Para el uso de la Api se hace una petición por enlace a la Api y se almacena la respuesta en la BBDD y se muestra en la pantalla de consultas.

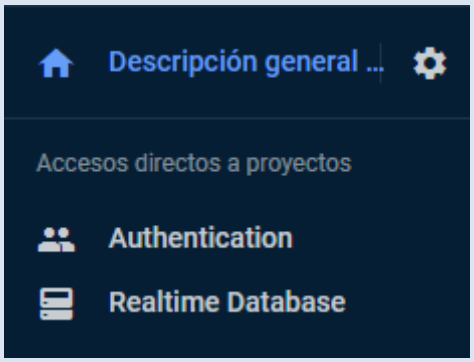
```yaml
dependencies:
  flutter:
    sdk: flutter
  # Dependencias para conectar con Firebase
  firebase_auth: ^4.6.2
  firebase_core: ^2.13.0
  firebase_database: ^10.2.2
  # Dependencia para conectar con la API
  http: ^1.0.0

  # The following adds the Cupertino Icons font to your application.
  # Use with the CupertinoIcons class for iOS style icons.
  cupertino_icons: ^1.0.2
```

```dart
class _HomeState extends State<Home> { // Clase que contiene la parte lógica y gráfica
  int indexActual = 0;
  final List<Widget> pantallas = [ // Instancia de las pantallas accesibles desde el menú inferior de navegación
    Api(),
    Historial(),
    Cuenta(),
  ];

  @override
  Widget build(BuildContext context) { // Parte gráfica que contiene la barra superior, el contenido principal que se elige según la opción
    return Scaffold(
      appBar: AppBar( // Barra superior
        title: const Text('PsychoGPT'),
        backgroundColor: Color(0xff3990F4),
        automaticallyImplyLeading: false,
        centerTitle: true,
      ),  // AppBar
      body: pantallas[indexActual], // Contenido de la pantalla
      bottomNavigationBar: BottomNavigationBar( // Barra de navegación inferior
          currentIndex: indexActual,
          type: BottomNavigationBarType.shifting,
          onTap: (index) => setState(() => indexActual = index),
          items: const <BottomNavigationBarItem>[
            BottomNavigationBarItem( // Botón para la pantalla Consulta
              icon: Icon(Icons.chat_bubble), label: 'Consulta', backgroundColor: Color(0xff3990F4)),  // BottomNavigationBarItem
            BottomNavigationBarItem( // Botón para la pantalla Historial
              icon: Icon(Icons.history), label: 'Historial', backgroundColor: Color(0xff00E5A4)),  // BottomNavigationBarItem
            BottomNavigationBarItem( // Botón para la pantalla Usuario
              icon: Icon(Icons.person), label: 'Usuario', backgroundColor: Color(0xffF83F4C)),  // BottomNavigationBarItem
          ]),  // <BottomNavigationBarItem>[], BottomNavigationBar
    );  // Scaffold
  }
}
```

```dart
final response = await http.post(
  // Variable para hacer la petición a la Api
  Uri.parse(
      'https://ce28-35-227-68-196.ngrok-free.app/generate-response/$query'),
  headers: headers,
  body: requestBody,
);

if (response.statusCode == 200) {
  // Si se obtiene respuesta de la Api se almacena en la variable 'respuesta' y se almacena en la base de datos
  Map<String, dynamic> responseData = json.decode(response.body);
  String outputText = responseData['response'];
  setState(() {
    respuesta = outputText;
  });
  await rutaDB
      .child('usuarios/$usuarioId/consultas/$query')
      .set(// Almacenamiento de la pregunta y la respuesta en la base de datos en formato Json
          {
    'pregunta': query.toString(),
    'respuesta': respuesta,
  });
} else {
  // Si no se obtiene respuesta o da error, se guarda un mensaje de error en lugar de la respuesta
  setState(() {
    respuesta = 'Error en la solicitud';
  });
}
```

FIREBASE.

Para contar con una base de datos optamos por Firebase de Google, en este caso utilizamos los módulos de Firebase Auth para el inicio de sesión y registro, y para los datos almacenados utilizamos Realtime Database




```dart
await rutaDB
    .child('usuarios/$usuarioId/consultas/$query')
    .set(// Almacenamiento de la pregunta y la respuesta en la base de datos en formato Json
        {
  'pregunta': query.toString(),
  'respuesta': respuesta,
});
```

```
void iniciarSesion(String email, String password) async {
  if (email == "" || password == "") {
    const snackBar = SnackBar(
    ├─ content: Text("Faltan datos."),
    );  // SnackBar
    ScaffoldMessenger.of(context).showSnackBar(snackBar);
  } else {
    try {
      await FirebaseAuth.instance.signInWithEmailAndPassword(
        email: email,
        password: password,
      );
    } on FirebaseAuthException catch (e) {
      print(e.message);
      print(e.code);
      String mensajeError;
      if (e.code == 'user-not-found') {
        mensajeError = 'Usuario no encontrado';
      } else if (e.code == 'wrong-password') {
        mensajeError = 'Contraseña incorrecta';
      } else if (e.code == 'invalid-email') {
        mensajeError = 'Email no válido';
      } else {
        mensajeError = 'Esta cuenta no existe o algún dato es incorrecto';
      }
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text(mensajeError)),
      );
    }
  }
```