



# Sistemas conversacionales híbridos aplicados al sector turístico

## **Autor:**

Alberto Pérez Blasco

## **Tutor:**

José Francisco Quesada Moreno

Trabajo de Fin de Máster, del Máster Universitario en Lógica,  
Computación e Inteligencia Artificial.

Universidad de Sevilla

28 de agosto de 2023



# Tabla de Contenidos

<b>1. Introducción</b>	<b>5</b>
1.1. Problema a resolver . . . . .	5
1.1.1. Selección de las API's . . . . .	5
1.1.2. Interpretación del lenguaje natural . . . . .	6
1.1.3. Respuesta en lenguaje natural . . . . .	6
<b>2. Estado del arte</b>	<b>7</b>
2.1. Token . . . . .	7
2.2. Embedding . . . . .	8
2.3. Transformer . . . . .	9
2.3.1. Encoder . . . . .	9
2.3.2. Decoder . . . . .	11
2.4. Modelo de lenguaje . . . . .	14
2.5. Prompt . . . . .	16
<b>3. Propuesta</b>	<b>17</b>
3.1. Alcance . . . . .	17
3.2. Diseño . . . . .	17
3.2.1. Arquitectura . . . . .	21
3.3. Modelado de datos . . . . .	24
3.4. Implementación . . . . .	30
3.4.1. Implementación conexión API Tiempo . . . . .	33
3.4.2. Implementación conexión API's Ayuntamiento de Málaga . .	33
3.4.3. Implementación de la integración de voz . . . . .	35
3.5. Evaluación . . . . .	35
3.6. Casos de Uso . . . . .	35
3.6.1. Casos finales . . . . .	36
3.6.2. Trazas de los casos anteriores . . . . .	37
<b>4. Resultados y discusión</b>	<b>41</b>
<b>5. Conclusiones y Trabajo Futuro</b>	<b>42</b>
<b>A. Anexo I: Repositorio y organización</b>	<b>45</b>
<b>B. Anexo II: Conversaciones con ChatGPT para generación ejemplos y extracción de entidades</b>	<b>46</b>
B.1. Tiempo . . . . .	46
B.2. Cargadores . . . . .	46
B.3. Aparcamientos . . . . .	46
B.4. Monumentos . . . . .	46
B.5. Primer generador (pruebas con ChatGPT) . . . . .	46

## Resumen y Palabras Clave

La interacción entre los usuarios y los sistemas conversacionales ha adquirido una importancia creciente en el día a día, por lo que con este Trabajo de Fin de Máster, surge la idea de aplicarla al ámbito turístico, donde la calidad de la experiencia del usuario es fundamental. Se decidieron escoger los modelos de lenguaje, ya que en este contexto han demostrado ser herramientas poderosas para mejorar la interacción y ofrecer respuestas más precisas y relevantes al usuario.

Este documento, presenta un estudio que se centra en la aplicación de modelos de lenguaje en sistemas conversacionales híbridos en el sector turístico. El objetivo principal es aprovechar las capacidades de los modelos grandes de lenguaje (*LLM's*) para enriquecer la interacción con los usuarios, proporcionando respuestas más naturales.

A lo largo de este documento, se analiza el estado del arte en sistemas conversacionales explorando las limitaciones en este caso de uso, examinando los avances en modelos de lenguaje y se identifica la necesidad de adaptar estos modelos a las necesidades específicas del sector turístico. Por otro lado, la implementación se basa en el diseño de un sistema de este estilo que combina modelos de lenguaje con la integración de *API's* (interfaz que permite la conexión entre dos sistemas) de información turística, con una Interfaz de Usuario sencilla en consola.

Este documento, ofrece una visión detallada sobre la aplicación de modelos de lenguaje en sistemas conversacionales en el ámbito turístico, destacando la importancia de mejorar la interacción con los usuarios. Finalmente, las propuestas de mejora ofrecen una base sólida para futuros desarrollos, con el objetivo de conseguir una aplicación que ofrezca un mayor nivel de capacidades.

### Palabras clave:

- **Modelos de lenguaje.** Algoritmos de inteligencia artificial que permiten a las máquinas entender y generar lenguaje natural.
- **Sistemas conversacionales.** Tecnología que permite la interacción entre humanos y máquinas a través del lenguaje natural
- **API.** Interfaz de programación de aplicaciones que permiten a un sistema acceder a información y servicios externos ofrecidos por otro sistema.
- ***Fine-tuning*.** Proceso de ajustar los modelos de lenguaje preentrenados a tareas y dominios específicos.
- **Arquitectura híbrida.** Combinación de diferentes tecnologías y enfoques para lograr un mejor rendimiento y funcionalidad en los sistemas conversacionales.

# Capítulo 1

## Introducción

La idea de este problema, surge al explorar las funcionalidades que ofrecen los modelos grandes de lenguaje (*LLM's*) orientados a la conversación con el usuario, que han surgido en la actualidad como ChatGPT, Google Bard, Microsoft Bing Chat... Al ser la mayoría modelos de lenguaje y no tener acceso a más recursos (únicamente con los datos que han sido entrenados), la idea principal es desarrollar un sistema que se aproveche de estas capacidades, pudiendo dar respuestas certeras en base a la conexión a datos en tiempo real (*API*), ofreciendo así un servicio especialista sobre un campo concreto.

En el caso de este TFM, se escogió realizar esta personalización en torno al campo del turismo, más concretamente sobre el turismo de la ciudad de Málaga, debido a que su ayuntamiento ofrece datos en tiempo real actualizados y fáciles de obtener. Los ámbitos escogidos dentro de este campo fueron:

- **Ocupación de aparcamientos municipales.**
- **Cargadores de vehículos eléctricos.**
- **Descripciones de monumentos.**
- **Tiempo.** (En este caso, se podrá realizar para cualquier lugar al que se tenga acceso desde la *API*, es decir, no tiene que ser únicamente de Málaga).

### 1.1. Problema a resolver

El problema a tratar, se dividió en varias fases debido a que se tuvo que realizar un proceso de investigación de todos los apartados, para así poder determinar el alcance de la aplicación.

#### 1.1.1. Selección de las *API's*

En primer lugar, se tuvieron que buscar los datos en tiempo real y a su vez los servicios que se querían ofrecer, ya que era necesario que los datos fuesen de calidad y estuviesen actualizados, al encontrarse muchas bases de datos en estado obsoleto. Una vez realizada la investigación, se decidió acotar el problema a un lugar concreto, debido a que encontrar una base de datos en tiempo real tan amplia es prácticamente imposible, por lo que se decidió buscar los datos en tiempo real que ofrecían los distintos ayuntamientos de la región de Andalucía, dándose el caso de que el de Málaga era el que mejores características aportaba (actualización en tiempo real, fácil acceso y abundancia de datos distintos).

En el caso de otras provincias importantes como Sevilla o Granada, se buscó información pero en ambas se encontraban obsoletos ya que llevaban años sin actualizarse.

Para que el problema no se restringiese únicamente a Málaga, se decidió buscar otra *API* que tuviese unos datos más amplios, decidiendo que estos fuesen el clima en el tiempo actual.

### 1.1.2. Interpretación del lenguaje natural

Por otro lado, se tuvo que realizar un proceso de investigación para seleccionar el modelo de lenguaje adecuado a utilizar durante el desarrollo, probando en todos ellos la tarea a realizar, extraer de una cadena de texto en lenguaje natural la acción que se quería realizar con los datos asociados a la misma. Un ejemplo ideal, sería el siguiente:

¿Qué tiempo hace en Sevilla?

```
{"intent": "get_weather", "entities": {"city": "Sevilla"}}
```

Figura 1.1: Caso ideal de conversión del lenguaje natural a *JSON*

En este caso, se puede apreciar como se quiere la salida en base a la entrada y la correlación del atributo a obtener. El formato de la salida no es arbitrario, debido a que este formato (*JSON*), es un equivalente a los diccionarios de *Python* y son fáciles de manejar a la hora de programar, por lo que se podrían ser controlados por el ordenador para pasar los datos directamente y de forma correcta a las *API*'s.

Para conseguir este resultado, era necesario personalizar un modelo de lenguaje, ya que la tarea a realizar era muy específica por lo que primero se tuvo que realizar una búsqueda de varios modelos de lenguaje y posteriormente, comprobar si se podía realizar esta personalización y a su vez si el modelo base era capaz de comprender la tarea, ya que si el modelo base la entiende, hay más posibilidades de que el modelo personalizado sea capaz de lograrla.

Para estas pruebas, se introducían como entrada varios casos en los que se realizaba la conversión comentada en la imagen 1.1 de forma satisfactoria (ejemplos de entrada) para comprobar si el modelo de lenguaje, era capaz de interpretar la salida que se quería obtener basándose en la entrada

### 1.1.3. Respuesta en lenguaje natural

Otro apartado importante para el que había que escoger un modelo de lenguaje, era el de la formulación de una respuesta a modo chat para el usuario, basada en los datos proporcionados por la base de datos en tiempo real, por lo que para ello, se tuvo que buscar en los principales modelos de lenguaje orientados a los sistemas conversacionales y comprobar cuáles ofrecían una mayor capacidad de respuesta y a su vez, una *API* con la que se consiguiese una buena conectividad.

Una vez explicada por encima la problemática de la elección del alcance y los modelos de lenguaje, en el siguiente capítulo 2 se van a explicar los conceptos básicos que los engloban.

## Capítulo 2

# Estado del arte

### 2.1. Token

Un token, es la unidad mínima en la que se divide un texto para poder ser interpretado por el *transformer* (arquitectura de red neuronal utilizada). Los tokens, son secuencias de caracteres encontradas en el texto y para obtenerlos, se utilizan herramientas de *tokenización*, que separan el texto en estas unidades. Un ejemplo sería la herramienta de OpenAI para GPT-3 (cada modelo de lenguaje suele tener su propia herramienta):

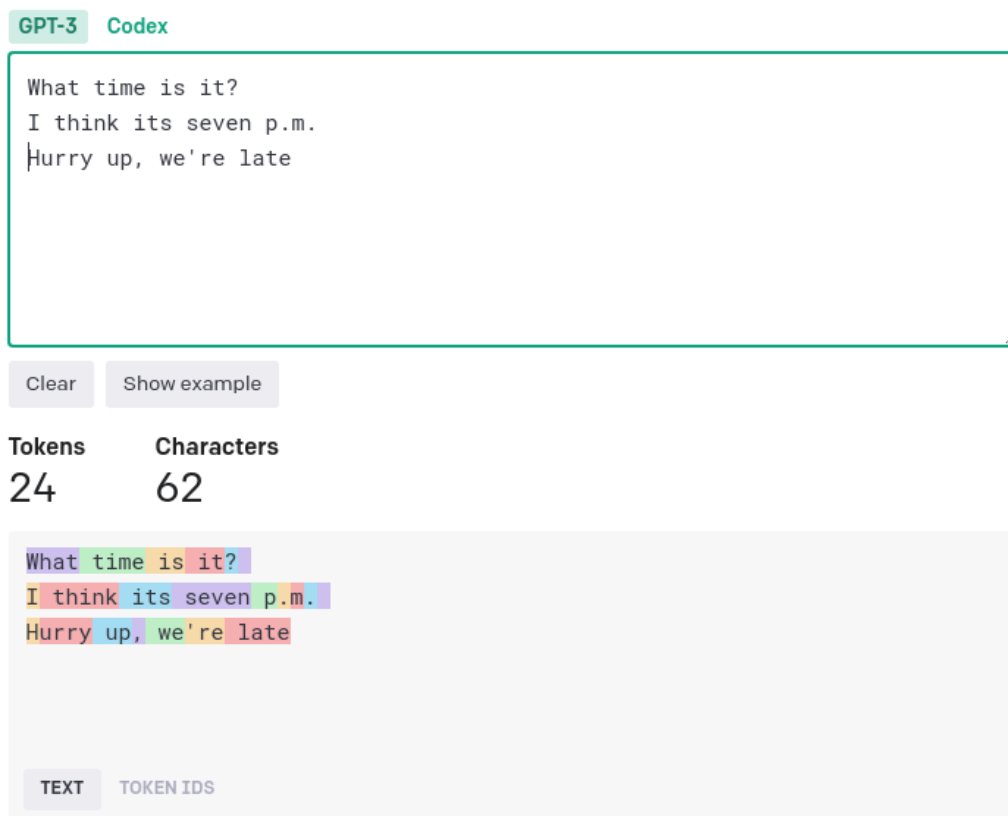


Figura 2.1: Ejemplo de *tokenización* para GPT-3

Como se puede ver en la figura 2.1, se obtienen 24 tokens (cada uno siendo un color del texto) y se puede apreciar que la mayoría de palabras las entiende enteras y las convierte un token (al ser inglés, ya que todos estos modelos están preparados principalmente para el inglés), pero algunas palabras como “Hurry” o “p.m.” las interpreta como varios tokens.

## 2.2. Embedding

Un *embedding*, es la representación vectorial de cada token. Para obtener los *embeddings*, se utilizan algoritmos propios en cada modelo, debido a que están entrenados para ver las relaciones entre las palabras y que así de esta forma, cada grupo de palabras relacionadas entre sí, salgan en una zona del espacio vectorial cercanas. Un ejemplo de espacio vectorial para distintas palabras sería:



Figura 2.2: Ejemplo de un espacio vectorial

En la figura 2.2, se pueden ver distintas palabras representadas en el espacio vectorial, apreciándose como las palabras que tienen una relación se encuentran en una zona bastante cercana, por ejemplo los animales en el centro o las frutas en las zonas exteriores.



## 2.3. Transformer

Los *transformers*[4][5], son un tipo de arquitectura de red neuronal que trata de gestionar la dependencia entre la entrada y salida de un texto basándose en la atención y en la recurrencia. Estas arquitecturas, son la base de los *LLM*'s, por lo que se va a explicar en profundidad el funcionamiento de las mismas, para así poder entender los modelos de lenguaje:

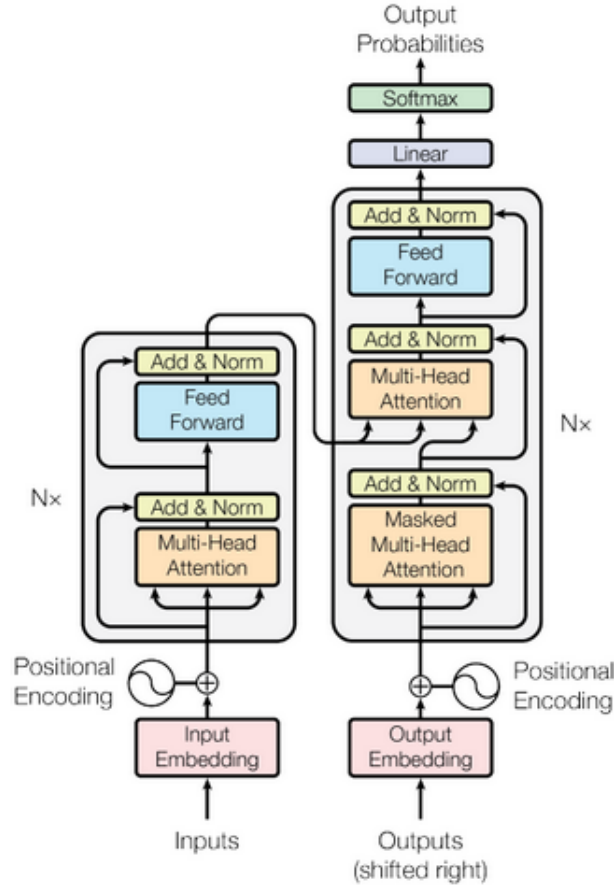


Figura 2.3: Arquitectura de un *transformer*

Como se puede apreciar en la imagen, un *transformer* está formado dos bloques *encoder* (izda) y *decoder* (dcha), constando cada uno de:

### 2.3.1. Encoder

Está formado por un conjunto de 6 capas idénticas ( $N = 6$ ), en la que cada una está formada por:

- **Multi-Head Attention Mechanism.** Este mecanismo, permite que el modelo atienda conjuntamente a información de distintos subespacios de representación, ubicados en distintas posiciones. Para ello, en lugar de tener una única atención, utilizan múltiples cabezas (*Multi-Head*) para capturar todos los detalles y relaciones en los datos de entrada. En cada cabeza, se proyectarán las consultas  $Q$ , las claves  $K$  y los valores  $V$ , siendo:
  - $Q$ . Las consultas, son las representaciones de los tokens de entrada a los que se les desea asignar ponderaciones de atención.

- $K$ . Las claves, son las representaciones de los tokens de entrada que se utilizan para calcular las similitudes o relevancias entre las consultas y las claves. Cada consulta, se compara con el resto de claves determinando cómo de relacionadas están.
- $V$ . Los valores, son las representaciones de los tokens de entrada que se utilizan para construir la salida ponderada de la atención. Para ello, se multiplican por las ponderaciones de atención para obtener una representación ponderada de los tokens de entrada que se utilizarán posteriormente.

Estas tres entradas, se utilizan para calcular las ponderaciones de atención en cada cabeza de atención, utilizando la fórmula:

$$head_i = Attention(QW_i^Q, KW_i^K, VW_i^V)$$

Siendo las proyecciones matrices de parámetros tal que  $W_i^{q|k|v} \in \mathbb{R}^{d_{model} \times d_{q|k|v}}$  ( $d$  hace referencia a las dimensiones del modelo y de las consultas, claves o valores respectivamente) Calculándose la atención con la siguiente fórmula:

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V$$

En esta función, se realiza el producto escalar de las consultas con las claves traspuestas para estimar la alineación de los vectores (*embeddings*), devolviendo un peso para cada palabra del texto. Posteriormente, se normaliza al dividir por  $\sqrt{d_k}$  y se aplica *softmax* para conseguir el peso de la palabra entre un rango 0-1. Finalmente, se multiplicarán estos pesos por el valor para obtener las palabras importantes. Un ejemplo sería:

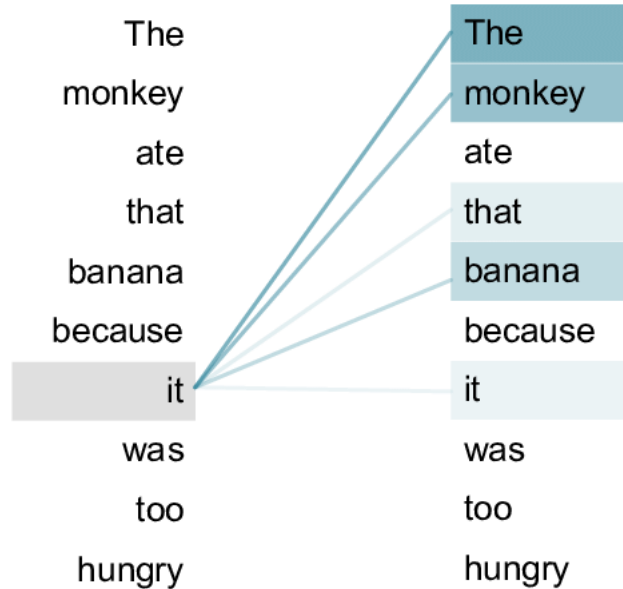


Figura 2.4: Ejemplo de obtención de la atención

Como se puede ver en la figura 2.4, se relaciona 'it' con el 'mono' y con la 'banana', debido a que puede hacer referencia a ambas palabras, resaltando con un color fuerte el caso que es más probable.

Una vez explicado el cálculo para una única cabeza, el cálculo para múltiples cabezas sería el siguiente:

$$MultiHead(Q, K, V) = Concat(head_1, \dots, head_h)W^O$$

Siendo  $h$  el número paralelo de cabezas (capas de atención) que se van a utilizar y  $W^O \in \mathbb{R}^{h d_v \times d_{model}}$ . Por otro lado, para reducir la dimensión de cada

cabeza, se divide la dimensión total entre  $h$ , quedándonos para el caso de los *transformers*:

$d_k = d_v = d_{model}/h = 512/8 = 64$  (todas salidas de 512 dimensiones para facilitar la conexión entre las arquitecturas).

Con esta reducción, el coste computacional sería similar a una única cabeza de atención con total dimensionalidad, pero con la ventaja de tener múltiples cabezas independientes (paralelismo).

- **Fully connected feed-forward network.** Aparte de la arquitectura anterior, también cuentan con una red neuronal totalmente conectada hacia delante, es decir, sin recurrencia (2.5). Esta, consiste de dos transformaciones lineales con una capa de activación *ReLU*:

$FFN(x) = \max(0, xW_1 + b_1)W_2 + b_2$  Donde  $x$ , es la entrada a la red totalmente conectada,  $W_1, b_1$  son los pesos y sesgos de la primera capa lineal y  $W_2, b_2$  los sesgos y pesos de la segunda capa lineal. La función de activación *ReLU* ( $\max(0, x)$ ), se aplica elemento por elemento a la salida de la primera capa lineal, convirtiendo los valores negativos a cero y manteniendo sin cambios los positivos.

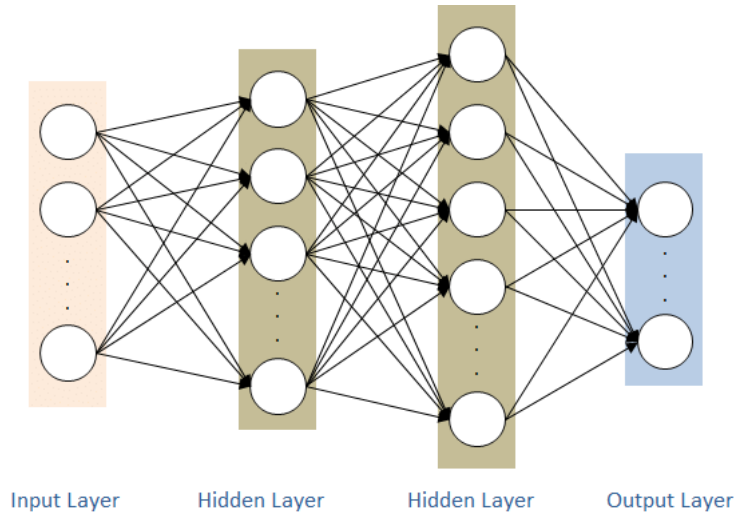


Figura 2.5: Feed Forward FCN

### 2.3.2. Decoder

También está compuesto por 6 capas ( $N = 6$ ) y el *decoder*, también incluye otra arquitectura que es la *Masked Multi-Head Attention*. Esta sería muy similar a la *Multi Head Attention*, debido a que se enmascaran las posiciones futuras para evitar que el modelo tenga acceso a la información del futuro durante la generación. Esta máscara, se aplica estableciendo los valores de atención correspondientes a las posiciones futuras en la matriz de pesos a un valor muy negativo (o cero), de modo que en la etapa de *softmax* se anulen y no contribuyan a las ponderaciones de atención, garantizando que durante la generación, cada posición solo pueda atender a las posiciones anteriores, permitiendo una generación autorregresiva paso a paso que no permita el acceso a información futura.

En la imagen 2.3, se puede apreciar como en el *encoder* y *decoder* aparece un salto antes de la entrada de cada bloque, esto se debe a que utilizan también conexiones residuales para abordar el problema del descenso del gradiente, preservar la información original y facilitar el aprendizaje de representaciones más detalladas, ya que cada capa puede especializarse en aprender características adicionales en lugar de tener que aprender la representación completa desde cero.

[1] La principal diferencia de las conexiones residuales con las CNN, es que se propagan las entradas de cada bloque residual hacia su salida, por lo que se reduce el desvanecimiento del gradiente, es decir, un bloque residual es un módulo que utiliza conexiones residuales para saltar por encima de algunas capas en lugar de atravesarlas, permitiendo a la red ser mucho más profunda sin tener que sufrir el problema del desvanecimiento del gradiente producido en las redes de este estilo. Un ejemplo sería el de la imagen siguiente:

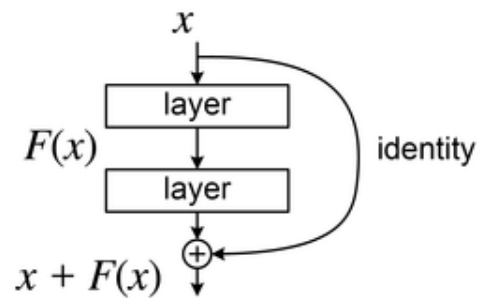
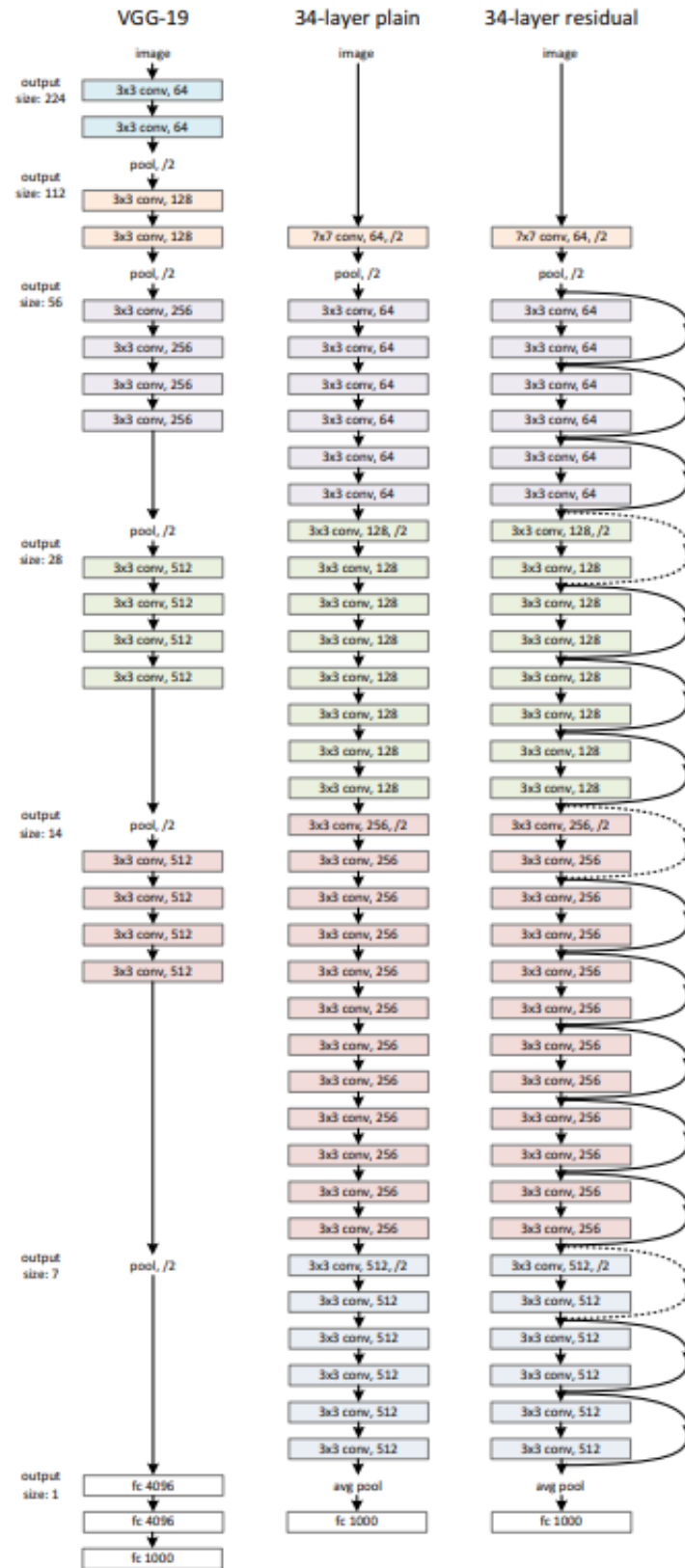


Figura 2.6: Bloque residual

La fórmula para un bloque residual, sería la siguiente:  $y = F(x) + x$ , siendo:

- $x$  es la entrada al bloque residual
  
  
  
  
  
  
  
  
  
  
- $F(x)$  es una función no lineal que transforma la entrada
  
  
  
  
  
  
  
  
  
  
- $y$  es la salida del bloque residual

La suma, se aplica a una función de activación no lineal para producir la salida final del bloque residual.

Figura 2.7: Arquitectura *ResNet* vs otros modelos

Finalmente, otro aspecto a destacar en la imagen 2.3, es el *Positional encoding*, que es una arquitectura para agregar información de posición a los datos de entrada, ya que de esta forma, el modelo puede distinguir entre diferentes posiciones en la secuencia de entrada, al no tener en cuenta la atención por sí sola la posición. En la figura 2.4 se puede apreciar que depende de la posición, debido a que el *positional*

*encoding*, se agrega a los *embeddings* de entrada en el codificador y el decodificador. El *positional encoding*, utiliza dos fórmulas, una para las dimensiones pares y otras para las impares, generando patrones sinusoidales para la representación de los tokens en cada secuencia. Las fórmulas utilizadas son:

- **Dimensiones pares:**  $PE_{(pos,2i)} = \sin(pos/10000^{2i/d_{model}})$
- **Dimensiones impares:**  $PE_{(pos,2i+1)} = \cos(pos/10000^{2i/d_{model}})$

Por último, el entrenamiento para los modelos de los *transformers*, dependerá de la tarea para las que se quieran realizar debido a que se podrán entrenar con un conjunto de datos u otro para adaptarlos a la misma. En el caso de [5], utilizan pares de Inglés-Alemán e Inglés-Francés, constando de 4.5 millones de pares de oraciones y 36 millones respectivamente, por lo que el modelo en este caso estaba enfocado a realizar la traducción.

## 2.4. Modelo de lenguaje

Un modelo de lenguaje [3], es un modelo de aprendizaje automático, que funciona prediciendo cuál será la siguiente palabra dentro de una oración, teniendo en cuenta el contexto de las palabras anteriores. Las probabilidades, ayudan a elegir la palabra entre varias alternativas. Los modelos de lenguaje, se pueden entrenar con grandes cantidades de texto en un idioma específico, para que aprendan las probabilidades estadísticas y lingüísticas del mismo, pudiendo utilizar distintos enfoques como:

- N-gramas
- Modelos ocultos de Markov
- Redes neuronales Recurrentes (RNN)
- Transformers

El objetivo principal de un modelo de lenguaje, es capturar la estructura y las regularidades del lenguaje, permitiendo que genere texto coherente y realista. Estos modelos se podrán utilizar para distintos tipos de aplicaciones como generación de texto automática, traducción, análisis de sentimiento...

La funcionalidad para la que están diseñados, dependerá de la forma en la que sean entrenados y de la arquitectura del modelo, debido a que dependiendo de la selección o modificación de los mismos, estará enfocado a una función u otra.

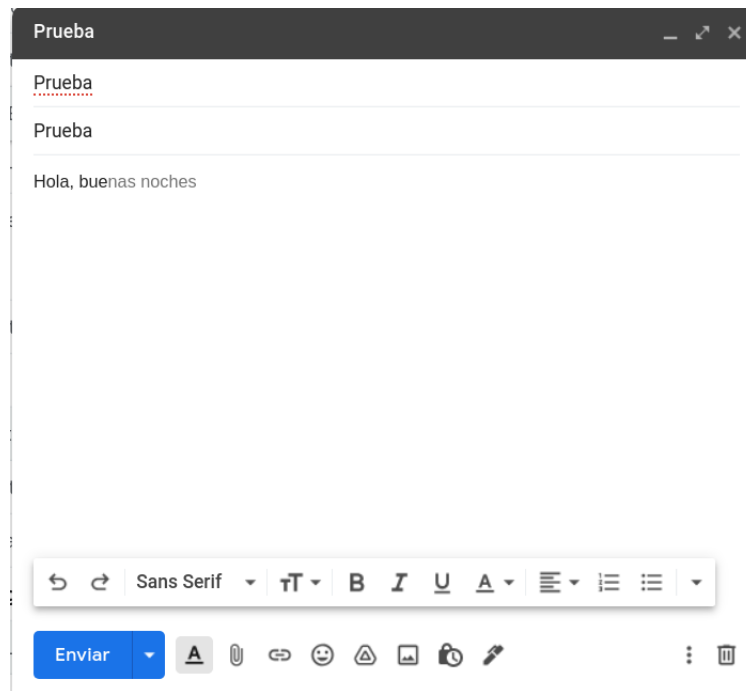


Figura 2.8: Caso de modelo de lenguaje para recomendar continuación de la frase

Para el caso con el que se va a trabajar, será necesario un *Large Language Model* (*LLM*), debido a la capacidad que tienen estos modelos para interpretar patrones y comprender el lenguaje. Esto se debe a que están específicamente diseñados y entrenado en una escala muy grande, constando de billones de parámetros y requiriendo un gran poder de cálculo y recursos computacionales para entrenarlos.

La principal diferencia entre un *LLM* y un modelo de lenguaje convencional se puede apreciar en la escala y el rendimiento, ya que los *LLMs* tienden a superar en rendimiento a los modelos de lenguaje más pequeños en varias tareas de procesamiento de lenguaje natural, como traducción automática, generación de texto, respuesta a preguntas, completado de oraciones...

Finalmente, dentro de los modelos de lenguaje se puede realizar una técnica denominada *fine-tuning*, la cual sirve para adaptar un modelo preentrenado (que ya cumplía una tarea específica) a otra tarea o dominio específico. En los modelos de lenguaje, tratará de utilizar un modelo de lenguaje para ajustarlo con un conjunto de datos específico, por lo que puede utilizar todo su conocimiento aprendido en la tarea anterior para la nueva tarea concreta ajustando los parámetros/pesos del modelo.

## 2.5. Prompt

Una *prompt*, es el fragmento de texto utilizado como entrada para solicitar una respuesta específica de un modelo de lenguaje, proporcionando contexto y guiando al modelo de lenguaje para generar una respuesta asociada a la tarea o información solicitada. La generación de la misma, será importante debido a que la calidad de la respuesta puede variar en función de su redacción. Un ejemplo sería:

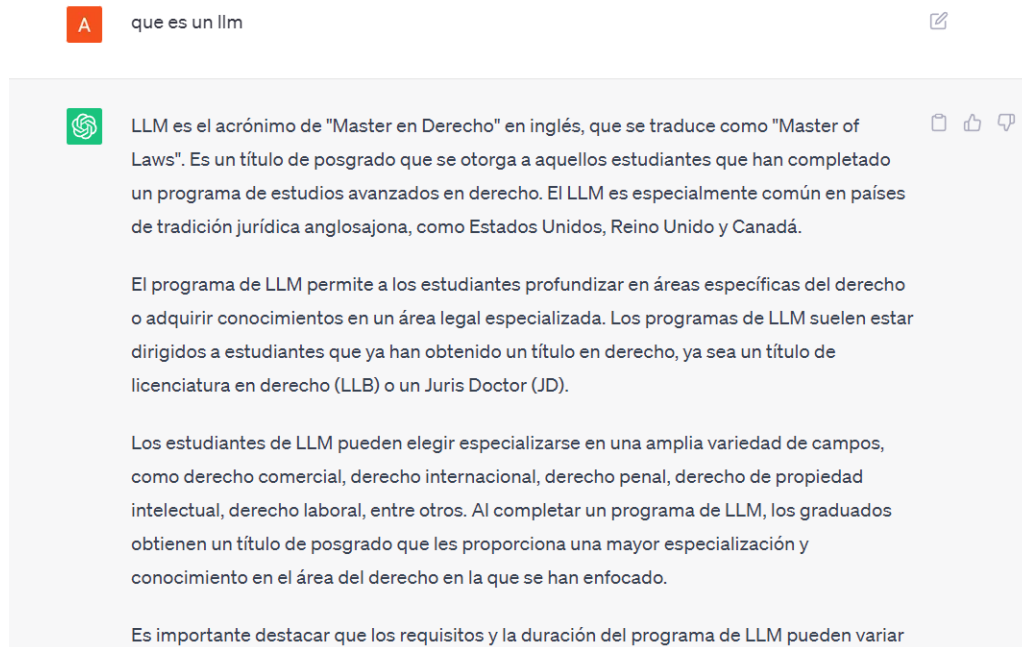


Figura 2.9: Primera *Prompt*

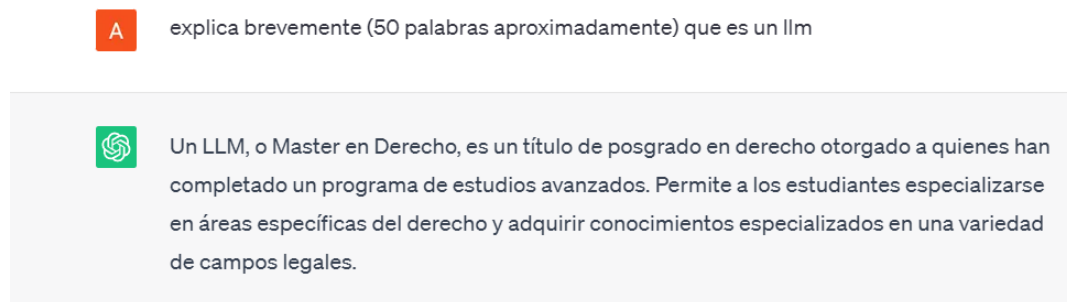


Figura 2.10: Segunda *Prompt*

Como se puede apreciar en las imágenes 2.9 2.10, dependiendo de la *prompt* el modelo de lenguaje devolverá una respuesta en función de la entrada y el contexto proporcionado. Por otro lado, se puede apreciar como sufre un pequeño sesgo, ya que está explicando un tema relacionado con la abogacía (máster en derecho), pero en este caso, se quería una explicación sobre LLM (Large Language Model), que habría que poner con el nombre completo (no con el acrónimo) para que el modelo de lenguaje entienda que no se está solicitando información del máster, sino del modelo de lenguaje.



## Capítulo 3

# Propuesta

### 3.1. Alcance

Una vez explicados todos los conceptos previos, en la aplicación se tratará de implementar un *ChatBot* similar a *ChatGPT* (Bot conversacional) pero enfocado al turismo. Para este caso, se va a reducir el rango de la aplicación (alcance), ya que realizarlo para el turismo en general de todos los lugares, requeriría la interconexión de varias fuentes de datos distintas, con un largo proceso de personalización para las mismas y por otro lado, no todos los lugares cuentan con bases de datos en tiempo real (*API's*) actualizadas, por lo que se va a implementar un bot conversacional, que admite las siguientes tareas:

- Información sobre un monumento que se encuentre en Málaga [6]
- Ocupación sobre el estado de un parking concreto de Málaga [7]
- Información sobre un cargador eléctrico concreto de Málaga [8]
- El clima de cualquier ciudad del mundo (no se sabe el alcance concreto, debido a que no se menciona en la *API*). [12]

Como se ha comentado anteriormente, se han seleccionado *API's* del ayuntamiento de Málaga, debido a que las mantiene correctamente actualizadas en tiempo real y es de los pocos lugares que se proporcionan (muchos otros ayuntamientos no las ofrecen).

Para el caso del tiempo, es necesario registrarse en *RapidAPI* (página web orientada a ofrecer *API's* de forma gratuita o de pago) y buscar por alguna *API* del tiempo que fuese gratuita, ya que únicamente con eso, se obtienen las claves de acceso a la misma. La escogida fue la siguiente: <https://rapidapi.com/apininjas/api/weather-by-api-ninjas/>

### 3.2. Diseño

Una vez seleccionadas las *API's*, había que decidir la selección de modelo de lenguaje que realice la traducción de lenguaje natural, a las tareas a realizar con el formato especificado en 1.1, por lo que se barajaron estas propuestas:

- **API de OpenAI.** [3] [11] Al ser gratuito, se realizaron las pruebas con ChatGPT (modelo de lenguaje GPT-3.5), introduciendo en el propio mensaje de entrada (*prompt*) varios ejemplos de conversiones ideales y posteriormente la conversión que se quería realizar. En el caso de este modelo de lenguaje, identificaba de manera satisfactoria el *intent* o acción y las entidades (parámetros). El único problema sería, que las capacidades de *fine-tuning* y de conexión con

la *API*, son de pago en este caso. Dentro de OpenAI, existen varios modelos de lenguaje que son capaces de detectar estas tareas y de realizar un *fine-tuning* sobre ellos. Para la selección de uno de ellos, será necesario saber el precio y las características de cada uno:

Nombre Modelo	\$ por 1k tokens (train)	\$ por 1k tokens (usage)	Max tokens	Embeddings output (dimensiones)
Ada	0.0004	0.0016	2048	2048
Babbage	0.0006	0.0024	2048	2048
Curie	0.0030	0.01200	2048	2048
Davinci	0.0300	0.1200	4000	2048

Cuadro 3.1: Comparación de los modelos de GPT-3 [13]

Estos modelos, únicamente sirven para *fine-tuning* debido a que fueron superados por los modelos de GPT-3.5 Y GPT-4, los cuales tienen mayores capacidades y rendimiento. Por otro lado, el cálculo de tokens, se realiza sumando los tokens de entrada (*prompt*) y la salida que se produce por el modelo de lenguaje, para el caso de 1800 tokens de entrada y 200 tokens de salida, la consulta tendría un coste de 2000 tokens. Finalmente, al crear una nueva cuenta de OpenAI, ofrecen un crédito de 5\$, por lo que para realizar las pruebas con estos modelos, se podría realizar gratuitamente.

- **GPT4All.** [15] Este modelo de lenguaje, se puede encontrar en <https://github.com/nomic-ai/gpt4all> y está en constante desarrollo, por lo que los errores encontrados, puede ser que en la versión actual se hayan resuelto. A la hora de realizar las pruebas, en este modelo se descargaba una interfaz y el modelo en el propio ordenador con unos pesos, obtenidos a raíz de personalizar el modelo de lenguaje *LLaMa* con 13 billones de parámetros y GPT-J para la conversación. Los problemas encontrados, fueron los siguientes:
  - No se aceptan más de 255 caracteres [16], debido a que *stdin* (entrada estándar, en este caso teclado) con *scanf* (función para leer de teclado en C) tienen restringido a ese límite. Este problema, se daba con la interfaz gráfica del usuario propuesta por los autores. Por otro lado, el modelo de lenguaje, debería aceptar 2048 tokens, así que en futuras versiones, no debería darse este error.
  - A la hora de realizar el *fine-tuning*, los autores todavía no tienen desarrollada esta funcionalidad, por lo que no se podría adaptar el modelo de lenguaje a la funcionalidad requerida (conversión de lenguaje natural a JSON).

### Short Term

- (IN PROGRESS) Train a GPT4All model based on GPTJ to alleviate llama distribution issues.
- (IN PROGRESS) Create improved CPU and GPU interfaces for this model.
- (Done) [Integrate llama.cpp bindings](#)
- (Done) [Create a good conversational chat interface for the model.](#)
- (Done) [Allow users to opt in and submit their chats for subsequent training runs](#)

### Medium Term

- (NOT STARTED) Integrate GPT4All with [Atlas](#) to allow for document retrieval.
  - BLOCKED by GPT4All based on GPTJ
- (NOT STARTED) Integrate GPT4All with Langchain.
- (IN PROGRESS) [Build easy custom training scripts to allow users to fine tune models.](#)

Figura 3.1: Lista de mejoras de *GPT4ALL*, en la que el *fine-tuning* todavía estaba en desarrollo (13/04/2023)

- **LLaMA & Alpaca.** [19] Este modelo de lenguaje (<https://medium.com/@martin-thissen/llama-alpaca-chatgpt-on-your-local-computer-tutorial-17adda704c23> ~:text=Alpaca%20Model%20The%20Alpaca%20model%20is%20a%20fine-tuned, which%20can%20be%20thought%20of%20as%20%E2%80%9CChatGPT%20behaviour%E2%80%9D.), combina el modelo de lenguaje *LLaMA* (utilizado en el caso anterior) y un modelo diseñado para seguir instrucciones denominado Alpaca. En este modelo, no se detectaba bien el formato JSON para las *prompts* (mensajes de entrada), por lo que se decidió por descartar este modelo.

El mensaje de entrada utilizado (*prompt*) para las pruebas fue el siguiente:

```
Infer the [JSON] for "" given the examples:
{
  1: {
    "text": "que tiempo hizo en sevilla el lunes?",
    "intent": "get_weather",
    "entities": {
      "city": "Sevilla",
      "date": "27/03/2023"
    }
  },
  2: {
    "text": "como está el día hoy en Valladolid",
    "intent": "get_weather",
    "entities": {
      "city": "Valladolid",
      "date": "28/03/2023"
    }
  },
  3: {
    "text": "En barcelona, cuantos grados hace mañana",
    "intent": "get_weather",
    "entities": {
      "city": "Barcelona",
      "date": "29/03/2023"
    }
  },
  4: {
    "text": "En Valencia",
    "entities": {
      "city": "Valencia"
    }
  },
  5: {
    "text": "Para el 25 de abril",
    "entities": {
      "date": "25/04/2023"
    }
  }
}
Assume that today is "28/03/2023".
```

Figura 3.2: *Prompt* para pruebas de selección del modelo de lenguaje

En estas pruebas, se tenía que extraer la acción de obtener el tiempo. En ellas, se introduce dentro de las dos comillas que se encuentran después del *for*, la frase de la que el modelo de lenguaje tendrá que obtener las del *JSON*. Como se ha comentado, estas pruebas eran únicamente para saber si el modelo de lenguaje entendía esta estructura. Por ello, aparecen dos entidades y en el caso de que se extraigan las dos es correcto.

Una vez vistas las distintas posibilidades, se decidió utilizar la *API* de OpenAI junto al modelo de *ada* para realizar el *fine-tuning*, debido a que es el más económico y la tarea a realizar no es muy compleja.

Para la comunicación con las *API's*, se utilizará la librería de *requests*, que sirve

para realizar este tipo de tareas (comunicarse a través de *Python* con páginas externas). Para la *API* del tiempo, se realizará una petición *GET* y para las *API's* del ayuntamiento de Málaga, se va a descargar el *csv* que proporcionado en página web de cada *API*, debido a que es más sencillo de manejar que realizar una consulta, ya que con la consulta no se puede obtener la información de los datos a consultar al tener comas para delimitar las columnas y aparecer comas en los valores de las filas, por lo que no se puede diferenciar correctamente a que se refiere cada información, y con el *csv* ya están delimitadas.

Para obtener una respuesta en lenguaje natural basada de los *csv*, se va a utilizar otro modelo de la *API* de OpenAI (debido a que es fácil de comunicar y con el crédito de 5\$ se pueden realizar ambas tareas), denominado *GPT-3.5 Completion Turbo*, el cual está optimizado para chatear con el usuario y tiene un coste de 0.0015\$/1000 tokens de entrada y 0.002\$/1000 tokens de salida y número máximo de tokens 4096, por lo que para la tarea de formatear la salida en lenguaje natural para el usuario es suficiente.

Finalmente, para la comunicación del usuario se utilizará una terminal al ejecutar el fichero de *Python* con la que se podrá conversar con el *ChatBot*.

### 3.2.1. Arquitectura

Para entender mejor las conexiones que se realizan, en la figura 3.3 se aprecian los distintos componentes que integran la aplicación:

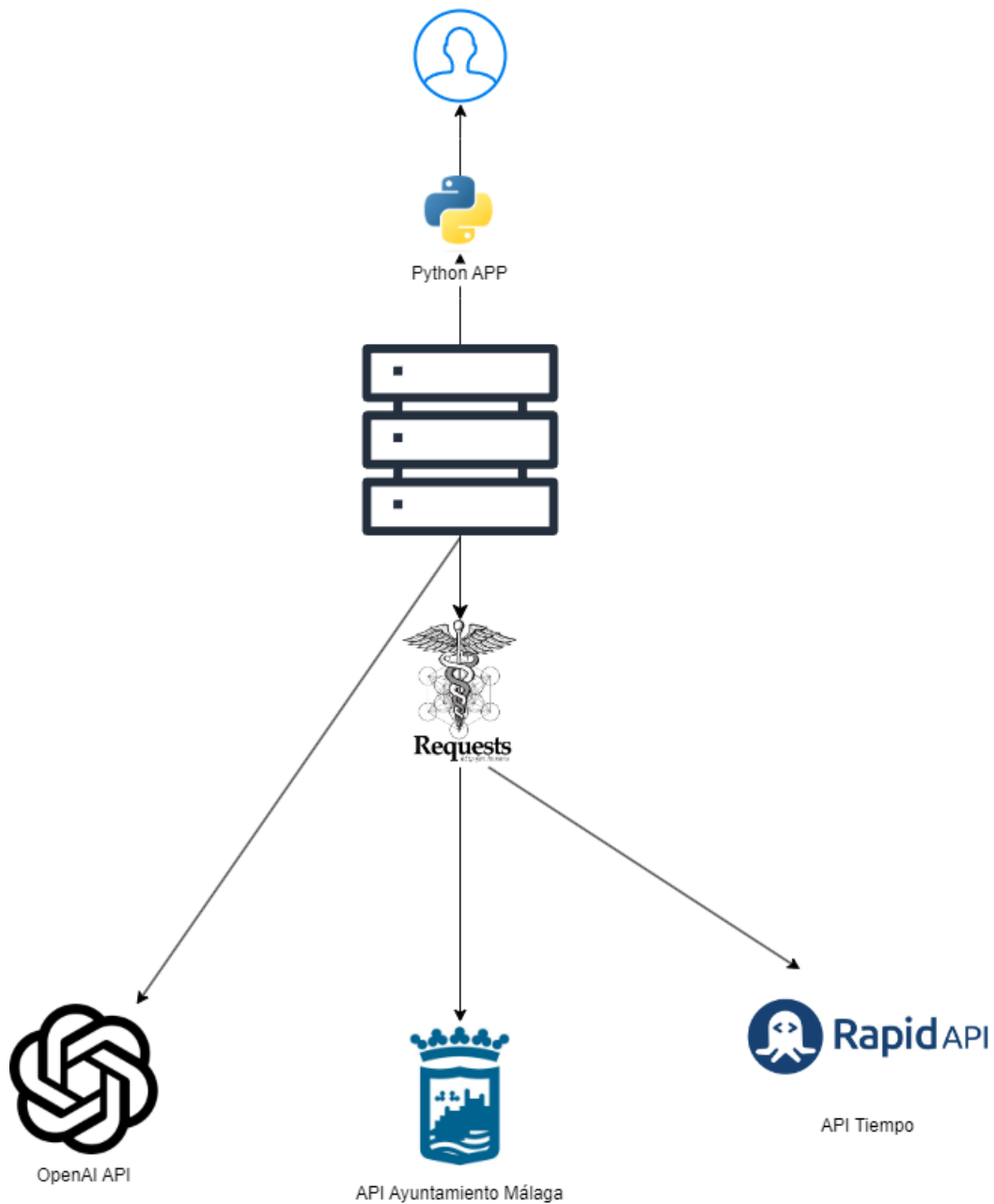


Figura 3.3: Diseño arquitectónico de la aplicación

Por otro lado, en las figuras, se puede apreciar como sería la comunicación entre los distintos apartados de la aplicación, para así poder entender mejor el funcionamiento de la misma y las distintas conexiones que se realizan y los motivos para la realización de las mismas

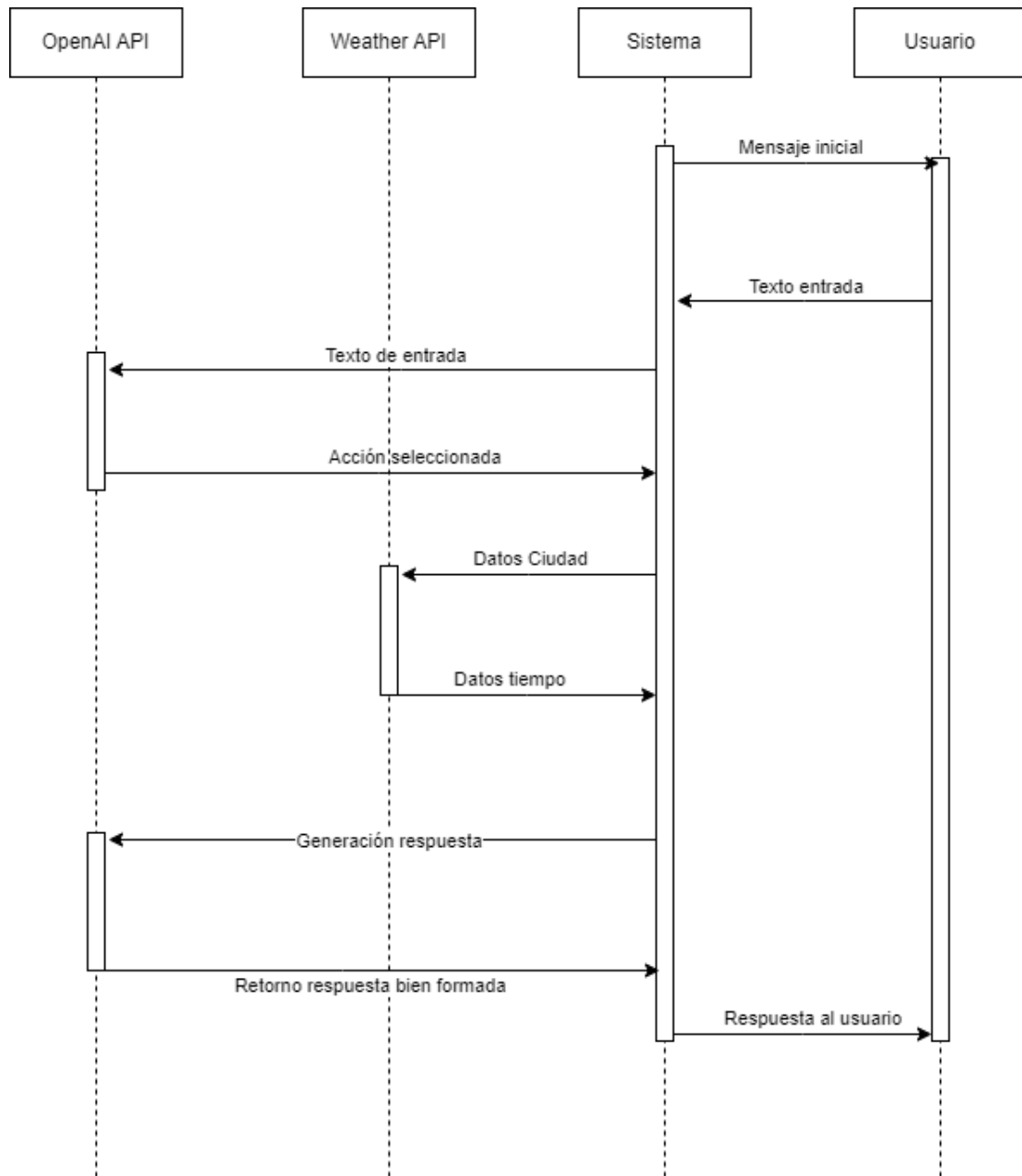


Figura 3.4: Diagrama de comunicación cuando el usuario solicita saber el tiempo en un lugar

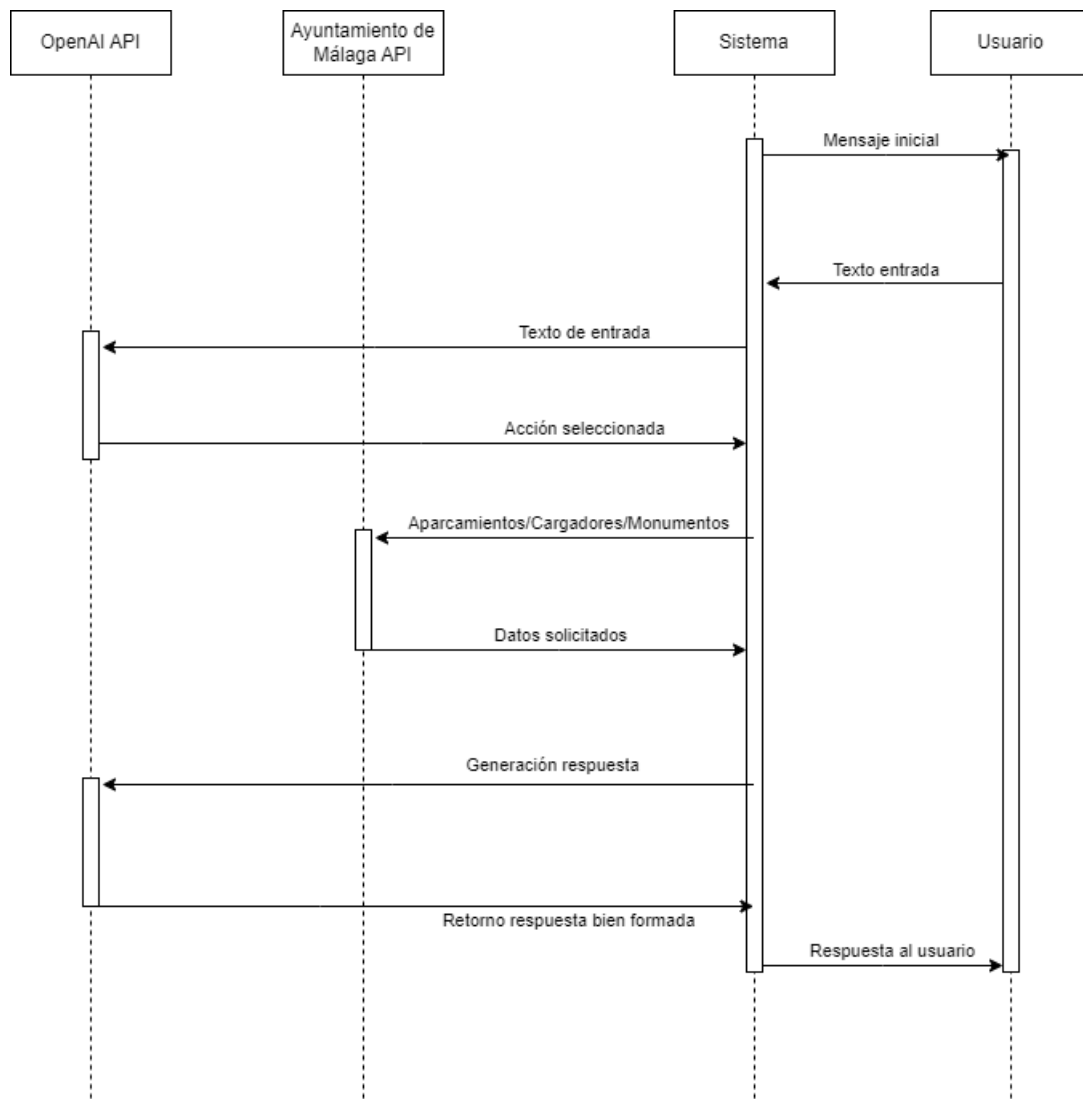


Figura 3.5: Diagrama de comunicación cuando el usuario solicita cualquier tarea de la *API* de Málaga

### 3.3. Modelado de datos

En este apartado, se va a comentar el proceso de generación de los datos para la realización del *fine-tuning*. En primer lugar, se redactaron de forma manual un lote ejemplos en unos *.txt* para cada acción, para comprobar el funcionamiento de la *prompt* [21] y si con los ejemplos, se cubrían la mayoría de casos. Un ejemplo de *prompt*, sería el comentado en 3.2 para el caso del reconocimiento de dos entidades (ya que al ser más complejo se comprobaría si el modelo de lenguaje se adaptaba a los datos) y otro tipo de *prompt*, podría ser el de los monumentos.

```
Infer the [JSON] for "donde esta el castillo de gibralfaro?" given the examples:
{
  1: {
    "text": "en que lugar puedo ver la giralda?",
    "intent": "get_monument_location",
    "entities": {
      "monument": "Giralda"
    }
  },
  2: {
    "text": "donde puedo ver la plaza de españa",
    "intent": "get_monument_location",
    "entities": {
      "city": "Plaza de España"
    }
  },
  3: {
    "text": "En que sitio está la sagrada familia",
    "intent": "get_monument_location",
    "entities": {
      "city": "Sagrada Familia"
    }
  },
  4: {
    "text": "Dónde puedo encontrar el Coliseo Romano?",
    "intent": "get_monument_location",
    "entities": {
      "monument": "Coliseo Romano"
    }
  },
  5: {
    "text": "En qué lugar se encuentra la Torre Eiffel?",
    "intent": "get_monument_location",
    "entities": {
      "monument": "Torre Eiffel"
    }
  },
  6: {
    "text": "Donde está el Taj Mahal?",
    "intent": "get_monument_location",
    "entities": {
      "monument": "Taj Mahal"
    }
  },
  7: {
    "text": "En qué sitio se encuentra el Cristo Redentor?",
    "intent": "get_monument_location",
    "entities": {
      "monument": "Cristo Redentor"
    }
  },
  8: {
    "text": "Dónde puedo ver la Gran Muralla China?",
    "intent": "get_monument_location",
    "entities": {
      "monument": "Gran Muralla China"
    }
  }
}
Assume that today is [today-date].
```

Figura 3.6: *Prompt* utilizada para monumentos



Una vez creados los 8 ejemplos para cada tipo de acción que se iba a realizar, se comprobó que funcionaba bien para todas acciones y que al mezclar dos ejemplos distintos, detectaba si una frase pertenecía a uno u otro, es decir, si se ponen 4 ejemplos de monumentos y 4 ejemplos de parkings, se detectaba que acción quería hacer la frase de entrada.

Una vez finalizadas las pruebas con todos casos, se procedió a realizar la investigación de como debía ser el formato de los datos para realizar el *fine-tuning* del modelo. Según [10], tiene que estar en formato *JSONL*, es decir, un fichero *JSON* en el que cada línea es un par de prompt-respuesta con el siguiente formato:

```
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}
{"prompt": "<prompt text>", "completion": "<ideal generated text>"}
```

Figura 3.7: Formato de los datos para el *fine-tuning*

Por lo que se realizó la modificación manual de todos los ejemplos anteriores para que cumpliera este formato. Un ejemplo de estos casos, sería el siguiente:

```
{"prompt": "que aparcamientos hay disponibles en la avenida de la victoria?", "completion": "{\"intent\": \"get_free_parkings\", \"entities\": {\"avenue\": \"Avenida de la Victoria\"}}"}
{"prompt": "donde puedo ver la plaza de espa\u00f1a", "completion": "{\"intent\": \"get_monument_location\", \"entities\": {\"monument\": \"Plaza de Espa\u00f1a\"}}"}
{"prompt": "en la calle Mara Curie puedo cargar mi coche?", "completion": "{\"intent\": \"get_electric_charger\", \"entities\": {\"avenue\": \"Calle Mara Curie\"}}"}
{"prompt": "En barcelona, cuantos grados hace ma\u00f1ana.", "completion": "{\"intent\": \"get_weather\", \"entities\": {\"city\": \"Barcelona\"}}"}

```

Figura 3.8: Ejemplos formateados uno de cada acción

Cuando estaban listos todos los ejemplos, se utilizó la herramienta que ofrece OpenAI para comprobar la calidad de los datos de entrenamiento para el *fine-tuning*. La instalación se realiza con *pip install -upgrade openai*, que incluye varias herramientas que ayudan a realizar esta personalización. Para añadir la *API Key* (clave única asociada a la cuenta para tener los datos de la misma) no se pudo realizar por el procedimiento normal, por lo que se incluía en el propio comando con el parámetro *-k* y posteriormente la clave entre comillas. El procedimiento para realizar el *fine-tuning*, fue el siguiente:

- ***openai tools fine-tunes.prepare-data -f LOCAL\_FILE***. Este comando, es una herramienta que valida los datos dando sugerencias y formateando los datos, al preguntar si se quiere reescribir el fichero *JSONL* nuevo. La primera ejecución se realizó con los 8 pares de cada ejemplo (30 en total aproximadamente). En esta, se obtuvo esta salida:

Analyzing...

- Your file contains 31 prompt-completion pairs. In general, we
  - recommend having at least a few hundred examples. We've
  - found that performance tends to linearly increase for every
  - doubling of the number of examples
- Your data does not contain a common separator at the end of
  - your prompts. Having a separator string appended to the end
  - of the prompt makes it clearer to the fine-tuned model where
  - the completion should begin. See
  - <https://platform.openai.com/docs/guides/fine-tuning/preparing-your-dataset>
  - for more detail and examples. If you intend to do open-ended
  - generation, then you should leave the prompts empty
- All completions start with prefix ``{"intent": "get``. Most of
  - the time you should only add the output data into the
  - completion, without any prefix

- All completions end with suffix ``"}}``
- The completion should start with a whitespace character (`` ``).
- This tends to produce better results due to the tokenization
- we use. See
- <https://platform.openai.com/docs/guides/fine-tuning/preparing-your-dataset>
- for more details

Based on the analysis we will perform the following actions:

- [Recommended] Add a suffix separator ``->`` to all prompts
- [Y/n]: Y
- [Recommended] Remove prefix ``{"intent": "get"}`` from all
- completions [Y/n]: Y
- [Recommended] Add a whitespace character to the beginning of
- the completion [Y/n]: Y

Your data will be written to a new JSONL file. Proceed [Y/n]: Y

Wrote modified file to ``.\\fine-tuning_prepared (1).jsonl``  
 Feel free to take a look!

Now use that file when fine-tuning:

```
> openai api fine_tunes.create -t ".\\fine-tuning_prepared
→ (1).jsonl"
```

After you've fine-tuned a model, remember that your prompt has

- to end with the indicator string ``->`` for the model to
- start generating completions, rather than continuing with
- the prompt. Make sure to include ``stop=[""]`` so that the
- generated texts ends at the expected place.

Once your model starts training, it'll approximately take 2.87

- minutes to train a ``curie`` model, and less for ``ada`` and
- ``babbage``. Queue will approximately take half an hour per
- job ahead of you.

Como se puede apreciar, se recomienda un mínimo de 100 ejemplos para realizar el *fine-tuning*, por lo que se decidió utilizar ChatGPT para generar más ejemplos de cada acción, en total 120 (30 por acción). Durante la generación de estos ejemplos, se siguió este procedimiento:

- Para cada acción, se crea un nuevo chat con los ejemplos, debido a que si se realizaban todos en el mismo chat, existía bias con las anteriores ya que eran ejemplos y redacciones muy similares.
- A la hora de delimitar el inicio y fin de cada ejemplo, se introducen números delante (aunque luego se eliminasen) para que se logre una mejor comprensión.
- Después de estas recomendaciones que se vieron durante las pruebas, un ejemplo de *prompt* es el siguiente

Basándote en estos ejemplos:

```
1.{\"prompt\":\"en que lugar puedo ver la giralda?\",\"completion\":{\"intent\":
\\\"get_monument_location\\\",\\\"entities\\\":{\\\"monument\\\":\\\"Giralda\\\"}}}}
2.{\"prompt\":\"donde puedo ver la plaza de españa\",\"completion\":{\"intent\":
\\\"get_monument_location\\\",\\\"entities\\\":{\\\"city\\\":\\\"Plaza de España\\\"}}}}
3.{\"prompt\":\"En que sitio está la sagrada familia\",\"completion\":{\"intent\":
\\\"get_monument_location\\\",\\\"entities\\\":{\\\"city\\\":\\\"Sagrada Familia\\\"}}}}
4.{\"prompt\":\"Dónde puedo encontrar el Coliseo Romano?\",\"completion\":{\"intent\":
\\\"get_monument_location\\\",\\\"entities\\\":{\\\"monument\\\":\\\"Coliseo Romano\\\"}}}}
5.{\"prompt\":\"En qué lugar se encuentra la Torre Eiffel?\",\"completion\":{\"intent\":
\\\"get_monument_location\\\",\\\"entities\\\":{\\\"monument\\\":\\\"Torre Eiffel\\\"}}}}
6.{\"prompt\":\"Donde está el Taj Mahal?\",\"completion\":{\"intent\":
\\\"get_monument_location\\\",\\\"entities\\\":{\\\"monument\\\":\\\"Taj Mahal\\\"}}}}
7.{\"prompt\":\"En qué sitio se encuentra el Cristo Redentor?\",\"completion\":{\"intent\":
\\\"get_monument_location\\\",\\\"entities\\\":{\\\"monument\\\":\\\"Cristo Redentor\\\"}}}}
8.{\"prompt\":\"Dónde puedo ver la Gran Muralla China?\",\"completion\":{\"intent\":
\\\"get_monument_location\\\",\\\"entities\\\":{\\\"monument\\\":\\\"Gran Muralla China\\\"}}}}
Genera variaciones de los mismos que no hayan aparecido
```

Figura 3.9: *Prompt* para generar más ejemplos

- En algunos casos, había que pedir que reformulase o que cambiase los ejemplos ya que no funcionaba correctamente al utilizar los mismos lugares o no variar la formulación, es decir, la redacción de la pregunta (*prompt*).
- Para comparar si los ejemplos eran muy similares, se generó un fichero en el que aparecían la *prompt* y los *intent*. Este fichero, se generó pasando a ChatGPT todos los ejemplos e indicando que se extrajesen la *prompt* e *intents* bien formateadas:

```
Prompt: "que aparcamientos hay disponibles en malaga?"
Ciudad: "Malaga"

Prompt: "que aparcamientos tengo por cáceres?"
Ciudad: "Cáceres"

Prompt: "podre estacionar el coche en algún lado de cerdeña"
Ciudad: "Cerdeña"
```

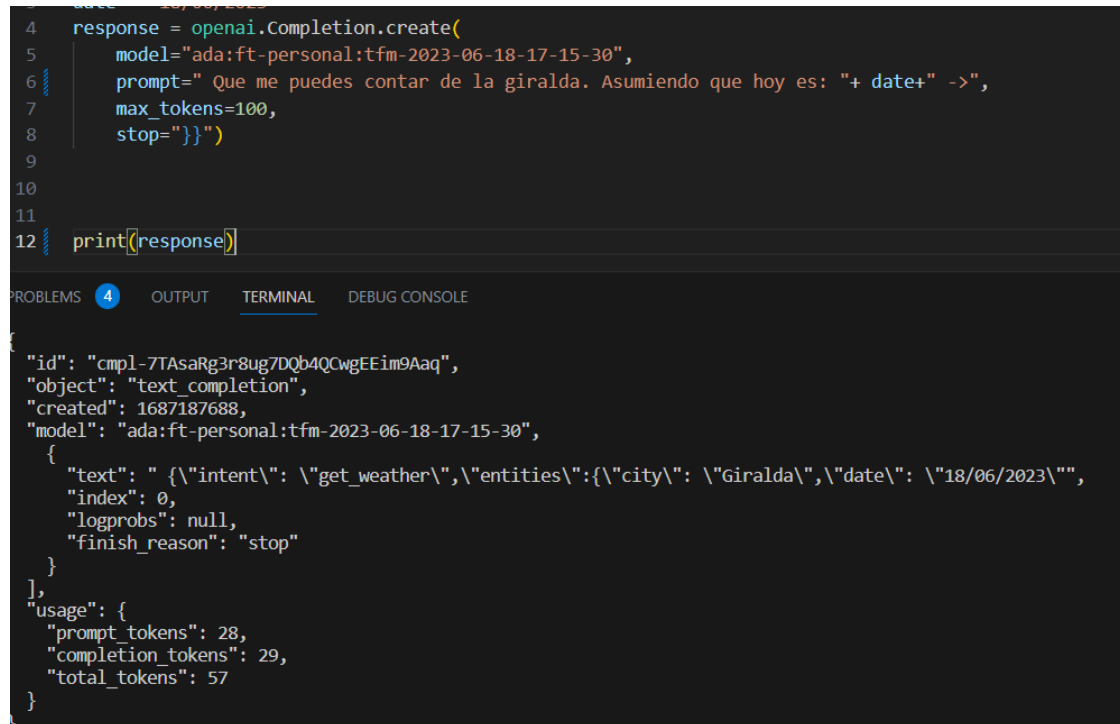
Figura 3.10: Texto con *prompts* e *intents*

- Finalmente, es necesaria una revisión manual de los ejemplos (por ello generación del fichero anterior), ya que en algunos casos se repetían los lugares o incluso se llegaban a realizar modificaciones del formato.

Los principales problemas que se encontraron a la hora de generar estos ejemplos con ChatGPT fueron:

- Las comillas a partir de *completion* van escapadas para que el formato del *JSONL* fuese correcto, pero a la hora de generar los ejemplos ChatGPT no las detectaba y no las ponía, por lo que se tenían que escribir manualmente para el resto de ejemplos.
- En el caso de la *prompt* del tiempo, había una mayor dificultad para variar debido a que al haber dos variables distintas que identificar, no llegaba a realizar las modificaciones de forma completa. En este caso, había que modificar lugares, posteriormente las fechas y finalmente la redacción, ya que más de dos cosas no es capaz de modificar en la misma ejecución. A

la hora de calcular las fechas, también se realizaba mal, por lo que será mejor hacerlo de forma manual. Finalmente, se decidió eliminar la fecha (al ser un foco de errores por la interpretación) ya que se comprobó que en la *API* únicamente se ofrecía la predicción del tiempo actual. Esta decisión se tomó en versiones posteriores (en las primeras se modificaron los ejemplos de forma manual) al ver que realizando el *fine-tuning*, se daba un error en el que se pedía información sobre un monumento y al tener incluida la fecha (para saber el día que se pedía el tiempo), el modelo pensaba que la acción era dar el tiempo y se equivocaba (3.11)



```

4 response = openai.Completion.create(
5     model="ada:ft-personal:tfm-2023-06-18-17-15-30",
6     prompt="Que me puedes contar de la giralda. Asumiendo que hoy es: "+ date+ " ->",
7     max_tokens=100,
8     stop="}}")
9
10
11
12 print(response)

```

```

{"id": "cmpl-7TAsaRg3r8ug7DQb4QCwgEEim9Aaq",
 "object": "text_completion",
 "created": 1687187688,
 "model": "ada:ft-personal:tfm-2023-06-18-17-15-30",
  {
    "text": " {\"intent\": \"get_weather\", \"entities\": {\"city\": \"Giralda\", \"date\": \"18/06/2023\"",
    "index": 0,
    "logprobs": null,
    "finish_reason": "stop"
  }
},
 "usage": {
  "prompt_tokens": 28,
  "completion_tokens": 29,
  "total_tokens": 57
}

```

Figura 3.11: Error de acción debido a que el tiempo tiene dos entidades (puede ser que en cuanto hay dos entidades se asocia a tiempo)

- En los lugares en los que aparecían calles, se encontró que se tenían plazas o frases para indicar la ubicación de una calle que estaba en una avenida, por lo que se añadió un pequeño lote de 5 ejemplos sustituyendo los anteriores arbitrariamente.

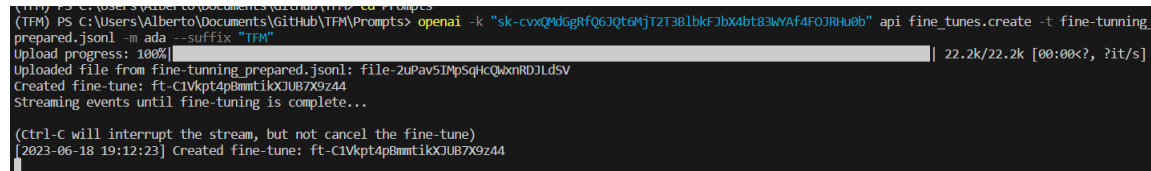
Una vez formateado correctamente, se volvió a utilizar la herramienta para ver la calidad del ejemplo, teniendo las mismas recomendaciones en el análisis. A continuación se indica la sugerencia y la decisión que se tomó con cada una:

- Add a suffix separator ' ' to all prompts [Y/n]: **Y**, debido a que así el modelo de lenguaje, sabrá donde tiene que empezar a completar
- Remove prefix "{ \"intent\" : \"get\" }" from all completions [Y/n]. **N**, debido a que como se comprobó en la versión anterior, es necesario para el intent, ya que si se elimina, se pierde una parte importante del JSON de salida
- Add a whitespace character to the beginning of the completion [Y/n]: **Y**, ya que es una medida que no afecta a los pasos posteriores y si se recomienda, será porque el fine-tuning tendrá un mejor rendimiento.

Una vez creado el conjunto de datos de una forma aceptada por la herramienta, se procedió a ejecutar el siguiente comando para crear el modelo:

```
openai api fine\_tunes.create -t \<TRAIN\_FILE\_ID\_OR\_PATH\> -m <BASE\_MODEL>
```

Donde *TRAIN\\_FILE\\_ID\\_OR\\_PATH* es la ruta al fichero de salida de la herramienta anterior y *BASE\\_MODEL*, el modelo base que se quiere utilizar como pre-entrenamiento para el *fine-tuning* (en este caso *ada*). La salida de este comando fue:



```
(TFM) PS C:\Users\Alberto\Documents\Github\TFM\Prompts> openai -k "sk-cvxQndGgRfQ6JQt6MjT2T3B1bkF3bX4bt83wAf4F0JRhu0b" api fine\_tunes.create -t fine-tuning-prepared.jsonl -m ada --suffix "TFM"
Upload progress: 100% | 22.2k/22.2k [00:00<?, ?it/s]
Uploaded file from fine-tuning_prepared.jsonl: file-2uPav5IMpSqHcQbnRDJLdSV
Created fine-tune: ft-C1Vkt4p8mtikXU87X9z44
Streaming events until fine-tuning is complete...

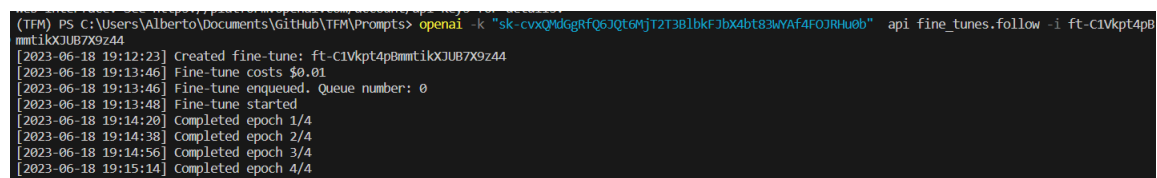
(Ctrl-C will interrupt the stream, but not cancel the fine-tune)
[2023-06-18 19:12:23] Created fine-tune: ft-C1Vkt4p8mtikXU87X9z44
```

Figura 3.12: Creación de la tarea del primer entrenamiento y carga del fichero de datos

Una vez creada la tarea del entrenamiento, se indica que para visualizar el estado del mismo, habrá que utilizar otro comando:

```
openai api fine\_tunes.follow -i <YOUR\_FINE\_TUNE\_JOB\_ID>
```

En el que se ve todo el proceso:



```
(TFM) PS C:\Users\Alberto\Documents\Github\TFM\Prompts> openai -k "sk-cvxQndGgRfQ6JQt6MjT2T3B1bkF3bX4bt83wAf4F0JRhu0b" api fine\_tunes.follow -i ft-C1Vkt4p8mtikXU87X9z44
[2023-06-18 19:12:23] Created fine-tune: ft-C1Vkt4p8mtikXU87X9z44
[2023-06-18 19:13:46] Fine-tune costs $0.01
[2023-06-18 19:13:46] Fine-tune enqueued. Queue number: 0
[2023-06-18 19:13:48] Fine-tune started
[2023-06-18 19:14:20] Completed epoch 1/4
[2023-06-18 19:14:38] Completed epoch 2/4
[2023-06-18 19:14:56] Completed epoch 3/4
[2023-06-18 19:15:14] Completed epoch 4/4
```

Figura 3.13: *Fine-tuning* finalizado

En la imagen, se puede apreciar el coste de realizar el *fine-tuning* del mismo y el número de la cola en el que se encuentra (en este caso no había otros trabajos pendientes). Finalmente, se aprecia las distintas *epochs* que se realizan sobre el conjunto de datos para completar el *fine-tuning*. Estas capturas se realizaron sobre la primera versión del *fine-tuning*, que como se ha comentado tuvo errores (3.11) ya que las dos entidades en *get\\_weather* daban a confusión, por lo que se generó otro modelo *fine-tuned* (v2) con estos casos eliminados (únicamente la entidad de ciudad en la acción de obtener el tiempo). Después de trabajar con esta versión, había un error en los datos, y es que los casos de parking estaban preparados con nombres de ciudades y en la *API* se encontraban con nombres de calles, por lo que se tuvo que realizar una tercera modificación (v3) contemplando lo anterior. De paso, se modificaron las acciones de obtener parkings y cargadores eléctricos, ya que estaban escritas con guion en lugar de barra baja, es decir:

- *get-electric-charger* → *get\\_electric\\_charger*
- *get-free-parkings* → *get\\_free\\_parkings*

Finalmente, este modelo *fine-tuned*, fue el definitivo para seguir trabajando en el proyecto.

### 3.4. Implementación

Para implementar todos los casos, se han ido probando por separado las conexiones con cada *API* creando pequeños módulos para finalmente, unirlos todos en un único fichero conjunto, debido a que de esta forma, era más fácil comprobar los casos de error y de obtener los datos correctamente, es decir, se ha ido comprobando el funcionamiento por separado de cada función implementada en el código. Para las pruebas finales, se ha enfrentado el chat a distintos casos para comprobar si realizaba todas las funciones correctamente con distintas formas de escribir.

La implementación final, se llevó a cabo en un fichero *Python* como se ha comentado utilizando la terminal para la salida y entrada del programa (mensajes del usuario de entrada y de salida por parte del bot). Para realizar este apartado, se utilizó el siguiente bucle:

```
bot_name = "Personalized-ChatGPT"
print("\033[37mHola! Soy un modelo de lenguaje basado en GPT-3-Ada, capaz de conectarse\n"+
      "con algunas aplicaciones en tiempo real, pudiendo realizar estas funciones: \n"+
      "- Información sobre monumentos de Málaga\n"+
      "- Información sobre ocupación de parkings en Málaga\n"+
      "- Información sobre cargadores eléctricos en Málaga\n"+
      "- Información sobre el tiempo actual en un cualquier lugar\n"+
      "¿Qué desea? (Escriba exit si quiere salir)")
while True:
    sentence = input("\033[33;4mUsuario:\033[0m ")
    if sentence == "exit":
        break
    elif sentence == "voz":
        respuesta = listen()
        speak(respuesta)
        print(f"\n\033[4;36m{bot_name}:\033[0m" + " \033[37m"+ respuesta + "\n" )
    else:
        print(f"\n\033[4;36m{bot_name}:\033[0m" + " \033[37m"+ getAction(sentence) + "\n" )
```

En el código, se puede apreciar que se utilizan códigos de color [17] para diferenciar el mensaje que escribe el bot del que escribe el usuario, subrayando y con color amarillo para el usuario y subrayando y con color cyan para el bot. El programa finalizará cuando el usuario escriba *exit* y la respuesta la escribirá llamando a *getAction*, que devuelve la respuesta correctamente formateada y escrita en lenguaje natural.

```
def getAction(sentence):
    openai.api_key = "sk-cvxQMdGgRfQ6JQt6MjT2T3B1bkFJbX4bt83WYAf4FOJRHu0b"

    response = openai.Completion.create(
        model="ada:ft-personal:tfm-v3-2023-06-20-16-52-50",
        prompt=sentence+" ->",
        max_tokens=100,
        stop="}}")
    try:
        diccionario_respuesta = json.loads(response.choices[0].text+"}}")
    except ValueError:
        diccionario_respuesta = json.loads("{\"intent\":\"\", \"entities\":\"\"}")

    switch = {
        "get_free_parkings": getParkingsFunction,
        "get_monument_location": getMonumentsFunction,
        "get_weather": getTemperatureFunction,
        "get_electric_charger": getElectricChargersFunction
    }

    accionEscogida = switch.get(diccionario_respuesta["intent"], funcionNoEncontrada)
    return accionEscogida(diccionario_respuesta["entities"])
```

En este método, en primer lugar se llama al modelo de lenguaje *fine-tuned*, pasando por parámetros:

- El nombre con el que se crea a modelo, la *prompt* (frase pasada por el usuario, añadiendo '' como recomiendan en la documentación de la *API*)
- Los máximos tokens con los que puede completar (no hay casos con calles o lugares muy largos)
- Condición de parada, debido a que si no, el modelo de lenguaje suele alargar hasta que se queda sin tokens.

Por otro lado, se creará un diccionario utilizando *json.loads()* (convierte una cadena de texto en formato *JSON* a un diccionario de *Python*), pero añadiendo '}}' ya que al ser la condición de parada, no se devuelve en la respuesta. Finalmente, se captura la excepción para el caso de que el formato del *JSON* sea inválido (no se ha detectado la acción a realizar) y se implementa un *switch* para *Python* utilizando un diccionario con una función asociada a cada clave (las 4 acciones indicadas). Para el caso de que no encuentre la clave en el diccionario, se utilizará un método auxiliar, el cual retorna que la acción que se quiere realizar no está disponible.

A la hora de implementar las acciones, se van a dividir en dos posibilidades, ya que la implementación de las *API's* del Ayuntamiento de Málaga, es distinta a la del tiempo. Primero, hay que comentar como se va a realizar la conexión con ChatGPT y la *prompt* que se le va a pasar para obtener la respuesta final:

```
def getResponse(data):
    openai.api_key = "sk-cvxQMdGgRfQ6JQt6MjT2T3B1bkFJbX4bt83WYAf4FOJRHu0b"
    response = openai.ChatCompletion.create(
        model="gpt-3.5-turbo",
        max_tokens=1500,
        messages=[
            {"role": "user", "content": data},
        ]
    )
    return(response.choices[0]["message"]["content"])
```

En este caso la conexión es bastante similar al caso del modelo *fine-tuned*, pero llamando a *ChatCompletion* en lugar de a *Completion*. Por otro lado, el máximo de tokens será de 1500 ya que es un tamaño considerable para entrada y respuesta y en el mensaje, habrá que indicar que el autor es el usuario (también se puede hacer como asistente, sistema o función) y la *prompt* variará dependiendo de la acción de la siguiente forma:

- **Tiempo.** Elabora un pronóstico del tiempo basado en estos datos:

*Ciudad:* {ciudad}

*Temperatura:* {temperatura}

*Sensación Térmica:* {sensación térmica}

*Humedad:* {humedad}

*Temperatura mínima:* {temperatura mínima}

*Temperatura máxima:* {temperatura máxima}

*Velocidad del viento:* {velocidad del viento}

- **Parkings.** Explica brevemente los datos del parking basado en:

*Nombre:* {nombre}

*Dirección:* {dirección}

*Capacidad:* {capacidad}

*Plazas libres actuales:* {plazas libres actuales}

- **Cargadores eléctricos.** Explica los detalles del punto de carga eléctrica basado en los siguientes datos:

*Dirección:* {dirección}

*Descripción:* {descripción}

*Tipo de carga:* {tipo de carga}

*Plazas disponibles:* {plazas disponibles}

*Número de puntos:* {número de puntos}

- **Monumentos.** Explica los detalles del monumento desde los siguientes datos:

*Nombre:* {nombre}

*Descripción:* {descripción}

*Dirección:* {dirección}

*Horarios:* {horarios}

*Precios:* {precios}

*Tarjeta joven:* {tarjeta joven}

Las dos llaves después del texto, hacen referencia a que en ese lugar irán los datos específicos para cada caso.



### 3.4.1. Implementación conexión API Tiempo

```
def getTemperatureFunction(entities):
    url = "https://weather-by-api-ninjas.p.rapidapi.com/v1/weather"

    querystring = {"city":entities["city"]}

    headers = {
        "X-RapidAPI-Key": "d08972acafmsh47671a8215e3be1p1fa5afjsncfa144723584",
        "X-RapidAPI-Host": "weather-by-api-ninjas.p.rapidapi.com"
    }

    response = json.loads(str(requests.request("GET", url, headers=headers, params=querystring).text))
    if "error" in response.keys():
        return "Localidad no encontrada en nuestros datos"
    mensajeCompletion = ("Elabora un pronóstico del tiempo basado en estos datos:\n"+
        "Ciudad: " + str(entities["city"]) + "\n"+
        "Temperatura: " + str(response["temp"]) + "\n"+
        "Sensación Térmica: " + str(response["feels_like"]) + "\n"+
        "Humedad: " + str(response["humidity"]) + "\n" +
        "Temperatura Mínima: " + str(response["min_temp"]) + "\n"+
        "Temperatura Máxima: " + str(response["max_temp"]) + "\n"+
        "Velocidad del viento: " + str(response["wind_speed"]) + "\n")
    return getResponse(mensajeCompletion)
```

Para obtener el tiempo, se utiliza la librería *requests* como se ha comentado anteriormente y se utiliza el ejemplo por defecto de la página de la *API*, pero cambiando el nombre del ejemplo por la ciudad de entrada. Finalmente, se trata la respuesta para saber si está la ciudad escogida y en caso de acierto, se devuelve la respuesta de ChatGPT, enviando el mensaje a la función anterior con el caso para el tiempo.

### 3.4.2. Implementación conexión API's Ayuntamiento de Málaga

```
def getElectricChargersFunction(entities):
    url = "https://datosabiertos.malaga.eu/recursos/urbanismoEInfraestructura/equipamientos/da_cve-25830.csv"
    parkings = getDataAPIMalaga(url)
    for indice, fila in parkings.iterrows():

        similarity = checkSimilarity(entities["avenue"].lower(), ' '.join(fila["NOMBRE"].split()[1:]).lower())
        if similarity == True:
            aux = dict(eval(fila["INFOESP"])[0])
            mensajeCompletion = ("Explica los detalles del punto de carga eléctrica basado en los siguientes datos:\n" +
                "Dirección: " + str(fila["DIRECCION"]).replace("\n", " ") + "\n" +
                "Descripción: " + str(fila["DESCRIPCION"]).replace("\n", " ") + "\n" +
                "Tipo de carga: " + str(aux["Tipologia_de_carga"]).replace("\n", " ") + "\n" +
                "Plazas disponibles: " + str(aux["Plazas_de_vehiculo_disponibles_para_recarga"]).replace("\n", " ") + "\n" +
                "Número de puntos: " + str(aux["Numero_de_puntos_de_recarga"]).replace("\n", " ") + "\n")
            return getResponse(mensajeCompletion)
    return "Cargador eléctrico no encontrado"
```

El método es bastante similar al anterior, pero en este caso, se utilizan dos funciones auxiliares, la primera es *getDataAPIMalaga* y la segunda *checkSimilarity*

```
def checkSimilarity(texto1, texto2):
    model = SentenceTransformer('sentence-transformers/all-mpnet-base-v2')
    similarity_score = util.cos_sim(model.encode([texto1]), model.encode([texto2]))
    #print(similarity_score.item())
    return True if similarity_score.item() >= 0.8 else False
```

Este método, se ha implementado debido a que en muchos casos se producen errores de escritura por parte del usuario o de comprensión por parte del modelo de lenguaje a la hora de detectar las *entities*, por lo que aunque cueste un poco más de tiempo la ejecución, se utiliza un modelo de similaridad semántica para calcular la similaridad entre la entrada y las entidades que se encuentran en los *csv*. Tras realizar varias pruebas, se concluyó que con más de un 0.8 de similaridad se da por válido. Este modelo pertenece a *sentence-transformers*, que es una librería de *Python* con la que se puede implementar esta tarea de una forma muy sencilla. Esta librería, obtiene los *embeddings* en 768 dimensiones de frases y párrafos para que se puedan utilizar en tareas como *clustering* o búsqueda semántica. Dentro de esta librería, se utiliza el modelo *sentence-transformers/all-mpnet-base-v2* debido a que es uno de los modelos con los que se ha obtenido un mejor rendimiento en este tipo de tareas [24] ya que se consiguen los mejores números en todos los ámbitos.

Finalmente, este modelo es un *fine-tuned* realizado con 1 billón de pares de frases del modelo pre-entrenado *microsoft/mpnet-base* (<https://huggingface.co/sentence-transformers/all-mpnet-base-v2>)

```
def getDataAPIMalaga(url):
    apis = {
        ("https://datosabiertos.malaga.eu/recursos/urbanismoEInfraestructura/"+
         "equipamientos/da_cultura_ocio_monumentos-25830.csv"): "monumentos.csv",
        ("https://datosabiertos.malaga.eu/recursos/urbanismoEInfraestructura/"+
         "equipamientos/da_cve-25830.csv"): "cargadores.csv",
        ("https://datosabiertos.malaga.eu/recursos/aparcamientos/"+
         "ocupappublicosmun/ocupappublicosmun.csv"): "aparcamientos.csv"
    }
    if url in apis.keys():
        archivo = requests.get(url, stream=True)

        with open(apis[url], 'wb') as f:
            for ch in archivo:
                f.write(ch)
        df = pd.read_csv(apis[url])
        return df.set_index("poiID") if apis[url] == "aparcamientos.csv" else df.set_index("ID")
    else:
        return None
```

Este último método, sirve para descargar el fichero *csv* asociado a cada caso debido a todo lo comentado anteriormente. Para ello, se descarga byte a byte y se escribe en el *csv* asociado, modificando el índice en función del fichero ya que la columna de identificación varía en cada uno.

### 3.4.3. Implementación de la integración de voz

Para la implementación de una integración de voz completa, se utilizan distintas librerías de conversión de audio a texto y de texto a audio, para realizar una conversación completa mediante la voz.

```
def listen():
    listener = sr.Recognizer()
    try:
        with sr.Microphone() as source:
            print(f"\n\033[4;36m{bot_name}:\033[0m" + " \033[37m" + "Escuchando..." + "\n" )

            pc = listener.listen(source)
            rec = listener.recognize_google(pc, language='es-ES')
            rec = rec.lower()
            print(f"\n\033[4;36m{bot_name}:\033[0m" + " \033[37m" + "Consultando: " + rec + "?\n" )
            return getAction(rec + "?")
    except Exception as e:
        print(e)
```

En el apartado de reconocimiento de la voz del usuario, se utiliza la librería de *speech-recognition* que trata de capturar el sonido con un *Recognizer* mientras se escuche sonido por el micrófono y posteriormente, se utiliza el sistema de reconocimiento de voz deseado, siendo este caso *recognize\_google* [22] [23].

```
def speak(text):
    tts = gTTS(text, lang='es-es')
    tts.save('pruebas.mp3')
    playsound("pruebas.mp3")
    remove("pruebas.mp3")
```

A la hora de reproducir los sonidos, se utiliza la librería *gtts* [14] que sirve para pasar de texto a audio. Para la reproducción del sonido desde *Python*, no se encontró ninguna librería que aceptase la entrada en lugar de utilizar un fichero, por lo que finalmente se optó por utilizar *playsound* [16] que reproduce el audio de un fichero y una vez reproducido, eliminarlo para no tener ficheros innecesarios.

## 3.5. Evaluación

El apartado de evaluación, no se ha llevado a cabo en el TFM, debido a que se centró más en el proceso de investigación de todas las capacidades que ofrece el *fine-tuning* y los modelos de lenguaje, creando una pequeña aplicación para poder comprobar su aplicabilidad.

## 3.6. Casos de Uso

En este apartado, se va a poner un ejemplo para cada función que puede realizar la aplicación, mostrando las distintas entradas y salidas que se obtienen desde cada apartado para ver todas las fases e interconexiones de las distintas *API's*

### 3.6.1. Casos finales

#### Aparcamientos

**Usuario:** Como esta el parking de la plaza cruz de humilladero

**Personalized-ChatGPT:** El parking "Cruz De Humilladero" se encuentra ubicado en la Plaza Cruz De Humilladero. Tiene una capacidad total de 100 plazas. En el momento actual, cuenta con 95 plazas libres, lo que significa que hay actualmente cinco plazas ocupadas en el estacionamiento.

Figura 3.14: Ejemplo de búsqueda del estado de un parking

#### Cargadores Eléctricos

**Usuario:** Puedo cargar el coche eléctrico en la calle lope de rueda?

**Personalized-ChatGPT:** El punto de carga eléctrica se encuentra en la dirección CALLE LOPE DE RUEDA, 250. Se trata de una infraestructura de estaciones de recarga para vehículos eléctricos bajo concesión demanial, lo que significa que está gestionada por una entidad pública.

El tipo de carga que ofrece este punto de carga es semirápida, lo que implica que el tiempo de carga será más rápido que en una carga lenta, pero aún más lento que en una carga rápida.

En este punto de carga hay disponibles 2 plazas, lo que significa que dos vehículos eléctricos pueden realizar la carga de forma simultánea.

El número de puntos que ofrece este punto de carga es 1, por lo que solo un vehículo puede ser conectado a la vez. Si hubiera más vehículos que deseen cargar, tendrían que hacer cola hasta que un punto de carga esté disponible.

Figura 3.15: Ejemplo de búsqueda del estado de un cargador eléctrico

#### Monumentos

**Usuario:** Que me puedes contar de la Abadía de Santa Ana?

**Personalized-ChatGPT:**

- El monumento se llama "Abadía de Santa Ana".
- Fue construido en el siglo XVI, aunque el edificio actual es del siglo XIX.
- La última intervención en el edificio se realizó en los años 80 y 90 del siglo XX para ampliar la comunidad y habilitar espacios para el Museo de Arte Sacro.
- La estructura del templo consta de una nave única dividida en dos tramos cubiertos con bóveda de medio cañón y arcos fajones. La capilla mayor tiene planta semicircular y una cubierta de cuarto de esfera.
- El coro se encuentra en la parte delantera del edificio, con una tribuna que se abre hacia la zona del presbiterio.
- En el exterior del monumento destaca una portada de piedra blanca con líneas clásicas. Esta portada tiene un arco de medio punto y está flanqueada por pilastras toscanas cajeadas. En el centro de la fachada se encuentra una hornacina con la imagen de Santa Ana, que data del siglo XVIII.
- La dirección del monumento es CALLE CÍSTER, 13.
- No se proporcionan horarios ni precios.
- No se acepta la Tarjeta Joven.

Figura 3.16: Ejemplo de búsqueda del estado de un monumento

## Tiempo

```
Usuario: Que tiempo hace en Ateca??  
Personalized-ChatGPT: Basado en los datos proporcionados, podemos prever que el clima en Ateca será bastante estable y sin cambios significativos en las próximas horas. La temperatura actual de 22 grados parece ser la temperatura máxima y mínima para hoy. La sensación térmica coincide con la temperatura real, lo que significa que no habrá sensación de frío o calor extremo. La humedad del 68% sugiere una cantidad moderada de humedad en el aire, lo que podría hacer que el clima se sienta un poco bochornoso.  
  
La velocidad del viento es baja, con una velocidad de 1.46, lo que indica que no habrá una presencia notable de viento. En resumen, se espera que el clima en Ateca sea estable y sin cambios bruscos en las próximas horas, con temperatura y sensación térmica de 22 grados, humedad moderada y baja velocidad del viento.
```

Figura 3.17: Ejemplo de búsqueda del estado del tiempo en un lugar

### 3.6.2. Trazas de los casos anteriores

A continuación, se han desglosado los ejemplos anteriores mostrando en cada uno de ellos:

- *JSON* generado por el modelo *fine-tuned*
- Similitud semántica entre el texto detectado por el modelo anterior y el proporcionado por la *API* del Ayuntamiento de Málaga (caso de tiempo no es necesaria)
- Mensaje enviado a ChatGPT para que genere la respuesta

En todos los casos, se podrá apreciar como la respuesta de ChatGPT varía debido a que este modelo no ofrece siempre la misma respuesta, a no ser que se introduzcan unos ejemplos muy concretos en la *prompt* y se quiera una salida definida (en este caso, únicamente se introducen los casos para que redacte).

## Aparcamientos

```
Usuario: Como esta el parking de la plaza cruz del humilladero_  
Reconocimiento entidades OpenAI (fine-tuned): {'intent': 'get_free_parkings', 'entities': {'avenue': 'Plaza Cruz del Humilladero'}}  
Los textos: plaza cruz del humilladero y plaza cruz de humilladero tienen una similitud de 0.9925348162651062  
Mensaje enviado a ChatGPT (prompt): Explica brevemente los datos del parking basado en:  
Nombre: Cruz De Humilladero  
Dirección: Plaza Cruz De Humilladero  
Capacidad: 100.0  
Plazas Libres Actuales: 95  
  
Personalized-ChatGPT: El parking Cruz De Humilladero se encuentra ubicado en la Plaza Cruz De Humilladero. Tiene una capacidad de 100 plazas de aparcamiento. En este momento, hay 95 plazas libres disponibles.
```

Figura 3.18: Traza del ejemplo de búsqueda del estado de un parking

En la figura anterior, se puede apreciar el funcionamiento en detalle del modelo de similitud semántica, debido a que se introduce una diferencia con respecto al caso anterior (original), ya que el lugar correcto es la 'plaza cruz de humilladero' y el usuario introduce 'plaza cruz del humilladero', caso que si se utilizase la comparación de los caracteres de forma literal, no daría un buen resultado.

## Cargadores Eléctricos

**Usuario:** Puedo cargar el coche eléctrico en la calle lope de rueda?  
Reconocimiento entidades OpenAI (fine-tuned): {'intent': 'get\_electric\_charger', 'entities': {'avenue': 'Calle Lope de Rueda'}}  
Los textos: calle lope de rueda y calle lope de rueda tienen una similaridad de 1.0  
Mensaje enviado a ChatGPT (prompt): Explica los detalles del punto de carga eléctrica basado en los siguientes datos:  
Dirección: CALLE LOPE DE RUEDA, 250  
Descripción: Infraestructura de estaciones de recarga para vehículos eléctricos bajo concesión demanial  
Tipo de carga: Semirápida  
Plazas disponibles: 2  
Número de puntos: 1

**Personalized-ChatGPT:** El punto de carga eléctrica se encuentra en la CALLE LOPE DE RUEDA, número 250. Es una infraestructura que ha sido concesionada y destinada a estaciones de recarga para vehículos eléctricos.

El tipo de carga que ofrece este punto de carga es semirápida, lo que significa que tiene una velocidad de carga más rápida que la carga normal, pero más lenta que la carga rápida. Esto permite cargar el vehículo de manera más eficiente y rápida que en cargadores convencionales.

En cuanto a las plazas disponibles, hay dos espacios destinados para vehículos eléctricos en este punto de carga. Esto significa que pueden cargar dos vehículos eléctricos al mismo tiempo, lo que ofrece una mayor disponibilidad y capacidades de carga.

Finalmente, se menciona que hay un único punto de carga disponible en este lugar. Esto indica que hay una única conexión de carga a la que se pueden conectar los vehículos eléctricos para iniciar la recarga.

Figura 3.19: Traza del ejemplo de búsqueda del estado de un cargador eléctrico



## Monumentos

En este caso, destaca lo comentado anteriormente de la variación de la salida de ChatGPT, ya que en el caso en el que no se mostraban las trazas, ChatGPT ofrecía una salida separada por varios ítems, pero en este caso, ha dado como resultado una salida con una forma más redactada que la anterior (aunque no aparezca completa en la imagen).

```
Usuario: Que me puedes contar de la Abadía de Santa Ana?
Reconocimiento entidades OpenAI (fine-tuned): {'intent': 'get_monument_location', 'entities': {'monument': 'Abadía de Santa Ana'}}
Los textos: abadía de santa ana y abadía de santa ana tienen una similaridad de 1.000000
1192092896
Mensaje enviado a ChatGPT (prompt): Explica los detalles del monumento desde los siguientes datos:
Nombre: Abadía de Santa Ana
Descripción: PERIODO: Siglo XVI. El edificio actual, según proyecto de Jerónimo Cuervo, en 1878, es el que podemos visitar. Las últimas intervenciones corresponden a los años 80 y 90 del siglo XX, donde se busca satisfacer las necesidades de ampliación de la comunidad y habilitar espacios amplios para la ubicación definitiva del Museo de Arte Sacro. La estructura del templo se compone de una nave única dividida en dos tramos cubiertos con bóveda de medio cañón con arcos fajones. La capilla mayor, de planta semicircular, se remata en una cubierta de cuarto de esfera. El coro se eleva a los pies del edificio, abriéndose la tribuna a la zona del presbiterio. En el exterior destaca la portada de piedra blanca, de líneas clásicas, con arco de medio punto flanqueado por pilastras toscanas cajeadas. Centra la fachada una hornacina con la imagen de Santa Ana, original del S.XVIII.
Dirección: CALLE CÍSTER, 13
Horarios: nan
Precios: nan
Tarjeta Joven: No

Personalized-ChatGPT: El monumento en cuestión se llama Abadía de Santa Ana y fue construido en el siglo XVI. Sin embargo, el edificio actual que podemos visitar fue diseñado por Jerónimo Cuervo en 1878. Durante los años 80 y 90 del siglo XX, se llevaron a cabo intervenciones para satisfacer las necesidades de ampliación de la comunidad y habilitar espacios para el Museo de Arte Sacro.

La estructura del templo está compuesta por una nave única dividida en dos tramos, los cuales están cubiertos con bóvedas de medio cañón con arcos fajones. La capilla mayor tiene una planta semicircular y cuenta con una cubierta en forma de cuarto de esfera. El coro se encuentra en la parte frontal del edificio y tiene una tribuna que da hacia la zona del presbiterio.

En cuanto al exterior del monumento, destaca una portada de piedra blanca con líneas clásicas. Esta portada cuenta con un arco de medio punto y está flanqueada por pilastras toscanas cajeadas. En el centro de la fachada hay una hornacina que contiene una imagen de Santa Ana que data del siglo XVIII.

En cuanto a la dirección, el monumento se encuentra en la calle Císter número 13. Respecto a los horarios de visita y los precios de entrada, no se proporciona información en los datos proporcionados. Además, no se menciona si se acepta la Tarjeta Joven para obtener
```

Figura 3.20: Traza del ejemplo de búsqueda del estado de un monumento

## Tiempo

```
Usuario: Que tiempo hace en Ateca?
Reconocimiento entidades OpenAI (fine-tuned): {'intent': 'get_weather', 'entities': {'city': 'Ateca'}}
Mensaje enviado a ChatGPT (prompt): Elabora un pronóstico del tiempo basado en estos datos:
Ciudad: Ateca
Temperatura: 22
Sensación Térmica: 22
Humedad: 68
Temperatura Mínima: 22
Temperatura Máxima: 22
Velocidad del viento: 1.46

Personalized-ChatGPT: Según los datos proporcionados, se puede pronosticar que en Ateca la temperatura se mantendrá constante en 22 grados Celsius durante todo el día. La sensación térmica también se mantendrá en 22 grados, lo que indica que la percepción de temperatura será similar a la temperatura real.

La humedad es del 68%, lo que indica que el ambiente puede sentirse algo húmedo. No se esperan cambios significativos en la temperatura mínima y máxima, ambas se mantendrán en 22 grados. Esto sugiere que la temperatura durante la noche será similar a la del día.

La velocidad del viento es de 1.46, lo que indica que habrá una brisa suave. En general, se espera un clima estable y tranquilo en Ateca, con temperatura constante y una ligera brisa.
```

Figura 3.21: Traza del ejemplo de búsqueda del estado del tiempo en un lugar



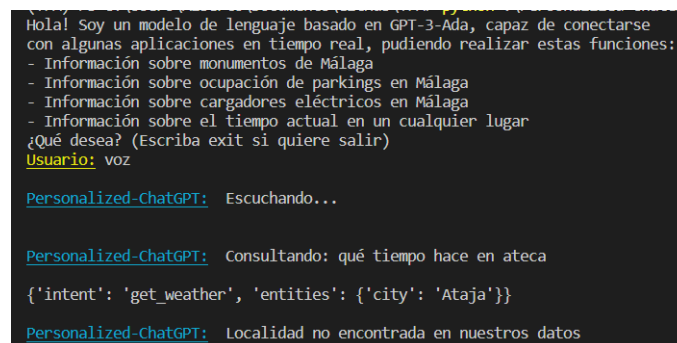
## Capítulo 4

# Resultados y discusión

Los resultados obtenidos en este trabajo, se han podido observar en el apartado anterior debido a que en este, se comenta el funcionamiento de las distintas funcionalidades, al haber desarrollado la aplicación con el funcionamiento esperado. Por otro lado, se podría haber conseguido un mejor rendimiento en cuanto al tema de tiempos, ya que la conexión con las *API's* de OpenAI, tiene un gran tiempo de procesamiento (al utilizar los dos modelos de lenguaje) y el modelo de similaridad semántica, también es bastante lento al tener que comparar con todos los casos del fichero de *csv* de la *API* de Málaga (dependiendo de la tarea). Finalmente, a la hora de detectar las tareas y las entidades muchas veces se cometen fallos dependiendo del tipo de formulación.

Lo anterior, se puede deber al utilizar el modelo base *ada* para el *fine-tuning* (el peor de todos), ya que cuenta con un menor número de dimensiones y de parámetros, pero al ser una aplicación tan cerrada y solo tratarse de identificar cuatro tareas, no merecía la pena realizar las pruebas con un modelo de lenguaje de mayores prestaciones.

Un caso concreto en el que se pudo apreciar este fallo, fue a la hora de implementar el reconocimiento de voz, ya que al no detectar el signo de interrogación el modelo de lenguaje *fine-tuned* no conseguía identificar las entidades correctamente cambiándolos por otros distintos como se puede apreciar en la imagen inferior:



```
Hola! Soy un modelo de lenguaje basado en GPT-3-Ada, capaz de conectarse
con algunas aplicaciones en tiempo real, pudiendo realizar estas funciones:
- Información sobre monumentos de Málaga
- Información sobre ocupación de parkings en Málaga
- Información sobre cargadores eléctricos en Málaga
- Información sobre el tiempo actual en un cualquier lugar
¿Qué desea? (Escriba exit si quiere salir)
Usuario: voz
Personalized-ChatGPT: Escuchando...
Personalized-ChatGPT: Consultando: qué tiempo hace en ateca
{'intent': 'get_weather', 'entities': {'city': 'Ataja'}}
Personalized-ChatGPT: Localidad no encontrada en nuestros datos
```

Figura 4.1: Error de detección errónea cuando la entrada es muy distinta a los ejemplos con los que se hace el *fine-tuning*

Con este caso comentado, se podría concluir que la aplicación está diseñada para casos que sean bastante similares a los incluidos en el ejemplo, así que para una mejora de la aplicación sería necesaria una mejora o bien del modelo de lenguaje (cambiar a *Babbage*, *Curie* o *Davinci*) o aumentar el lote de ejemplos con casos más variados creándolos de forma manual, ya que ChatGPT no llega a tener el suficiente nivel de 'emergencia' para poder crear frases completamente nuevas que no aparezcan en los ejemplos de la *prompt*.

## Capítulo 5

# Conclusiones y Trabajo Futuro

Como conclusiones del trabajo, en este proyecto se ha aprendido de las posibilidades que ofrece el *fine-tuning* ya que con un simple lote de 120 ejemplos, se ha conseguido obtener un modelo de lenguaje que realice todas las tareas de forma satisfactoria, por lo que con un mayor lote de ejemplos, se podrían realizar aplicaciones más complejas con modelos de lenguaje completos al existir otras variaciones del empleado pero con una complejidad mayor. Como se ha comentado en los resultados, se podrían haber obtenido unos tiempos de procesamiento menores, pero al implementar el modelo de similaridad semántica para que la aplicación tenga una mayor utilidad, se decidió dejarlo de esta forma. En cuanto a las versiones futuras de la aplicación, se pueden destacar las siguientes posibilidades:

- Usar unas *API*'s más complejas en las que se utilicen varias entidades, pero en este caso, modificando el modelo base empleado (con *ada* se producían errores a la hora de detectar la fecha del tiempo como se ha comentado), para ofrecer una aplicación más completa.
- Reducir los tiempos de espera, principalmente del modelo de similaridad semántica, pudiendo implementar otro tipo de técnicas
- En base a estas *API*'s introducir otro tipo de acciones que se puedan desarrollar como la de listar todos parkings con una menor ocupación, por ejemplo, implementando un módulo de geolocalización para lograr indicar los parkings/monumentos/cargadores más cercanos en base a la ubicación del usuario
- Añadir más *API*'s del Ayuntamiento de Málaga (al contar con cientos) para así poder crear un *ChatBot* especializado en el municipio de Málaga, ofreciendo posibilidades reales a los usuarios para saber como visitar Málaga o que ofrece la ciudad de una forma sencilla, pero para ello, sería necesario escalar la generación de ejemplos para tener unos rendimientos ajustados (en lugar de 30 ejemplos por *API*, unos 100 más o menos)

# Bibliografía

- [1] He, K. et al. (2015) *Deep residual learning for image recognition*, *arXiv.org*. Retrieved 28 June 2023, from <https://arxiv.org/abs/1512.03385>
- [2] *La guía máxima para el modelo de lenguaje GPT-3 de OpenAI*. (2022). Retrieved 28 June 2023, from <https://www.twilio.com/blog/la-guia-maxima-para-el-modelo-de-lenguaje-gpt-3-de-openai>
- [3] *Modelos de lenguaje en el NLP*. (2022). Retrieved 28 June 2023, from <https://monica-echeverrt.medium.com/modelos-de-lenguaje-en-el-nlp-8922dc34753b>
- [4] *Transformer: domina el mundo (NLP): explicación SENCILLA*. (2022). Retrieved 28 June 2023, from <https://www.themachinelearners.com/transformer/#:~:text=que%20nos%20importan.-,Auto-atenci%C3%B3n%20por%20multi-cabeza,para%20despu%C3%A9s%20concatenar%20los%20resultados>
- [5] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., & Gomez, A. et al. (2022). *Attention Is All You Need*. Retrieved 28 June 2023, from <https://arxiv.org/abs/1706.03762>
- [6] *Monumentos - conjuntos de Datos - Datos Abiertos Ayto. Málaga*. Retrieved 28 June 2023, from <https://datosabiertos.malaga.eu/dataset/monumentos>
- [7] *Cargadores de Vehículos Eléctricos - Datos Abiertos Ayto. Málaga*. Retrieved 28 June 2023, from <https://datosabiertos.malaga.eu/dataset/cargadores-de-vehiculos-electricos>
- [8] *Ocupación Aparcamientos Públicos Municipales - CSV - Datos Abiertos Ayto. Málaga*. Retrieved 28 June 2023, from <https://datosabiertos.malaga.eu/dataset/ocupacion-aparcamientos-publicos-municipales/resource/0dcf7abd-26b4-42c8-af19-4992f1ee60c6>
- [9] *Openai (2022) Openai-cookbook/examples/fine-tuned\_classification.ipynb at main · openai/openai-cookbook, GitHub*. Retrieved 28 June 2023, from [https://github.com/openai/openai-cookbook/blob/main/examples/Fine-tuned\\_classification.ipynb](https://github.com/openai/openai-cookbook/blob/main/examples/Fine-tuned_classification.ipynb)
- [10] *OpenAI platform Fine Tuning*. Retrieved 28 June 2023, from <https://platform.openai.com/docs/guides/fine-tuning>
- [11] *OpenAI platform Introduction*. Retrieved 28 June 2023, from <https://platform.openai.com/docs/guides/chat/introduction>
- [12] *Weather by API-ninjas API documentation (apininjas): Rapidapi*. Retrieved 28 June 2023, from <https://rapidapi.com/apininjas/api/weather-by-api-ninjas/>
- [13] *OpenAI Pricing*. Retrieved 28 June 2023, from : <https://openai.com/pricing>
- [14] *gTTS Documentation*. Retrieved 28 June 2023, from: <https://gtts.readthedocs.io/en/latest/>

- [15] *Nomic-ai/gpt4all: Gpt4all: An ecosystem of open-source chatbots trained on a massive collections of clean assistant data including code, stories and dialogue, GitHub*. Retrieved 28 June 2023, from : <https://github.com/nomic-ai/gpt4all#readme>
- [16] *Nomic-Ai Prompt limit? · issue 74 · nomic-ai/gpt4all, GitHub*. Retrieved 28 June 2023, from: <https://github.com/nomic-ai/gpt4all/issues/74>
- [17] *Pherkad (1970) Dar color a las salidas en la consola*. Retrieved 28 June 2023, from: <https://python-para-impacientes.blogspot.com/2016/09/dar-color-las-salidas-en-la-consola.html>
- [18] *TaylorSMarks/playsound: Pure Python, Cross Platform, single function module with no dependencies for playing sounds., GitHub*. Retrieved 28 June 2023, from: <https://github.com/TaylorSMarks/playsound>
- [19] *Thissen, M. (2023) Llama amp; Alpaca: ‘CHATGPT’ on your local computer: Tutorial, Medium*. Retrieved 28 June 2023, from: <https://medium.com/@martin-thissen/llama-alpaca-chatgpt-on-your-local-computer-tutorial-17adda704c23#:~:text=Alpaca%20Model%20The%20Alpaca%20model%20is%20a%20fine-tuned,which%20can%20be%20thought%20of%20as%20%E2%80%9CChatGPT%20behaviour%E2%80%9D>.
- [20] *Caelen, O. (2023) Unleashing the power of GPT: How to fine-tune your model, Medium*. Retrieved 28 June 2023, from: <https://towardsdatascience.com/unleashing-the-power-of-gpt-how-to-fine-tune-your-model-da35c90766c4>
- [21] *White, J. et al. (2023) A prompt pattern catalog to enhance prompt engineering with chatgpt, arXiv.org*. Retrieved 28 June 2023, from: <https://arxiv.org/abs/2302.11382>
- [22] *Web speech API*. Retrieved 28 June 2023, from: <https://wicg.github.io/speech-api/>
- [23] *Reconocimiento de Voz — Jose Ricardo Zapata*. Retrieved 28 June 2023, from: [https://joserzapata.github.io/courses/mineria-audio/reconocimiento\\_voz/](https://joserzapata.github.io/courses/mineria-audio/reconocimiento_voz/)
- [24] *Pretrained Models - Sentence-Transformers documentation*. Retrieved 28 June 2023, from: [https://www.sbert.net/docs/pretrained\\_models.html](https://www.sbert.net/docs/pretrained_models.html)

## Apéndice A

# Anexo I: Repositorio y organización

Para trabajar con el proyecto, se utilizó *Github* como herramienta de control de versiones. El repositorio no está organizado ya que al no ser un proyecto real y tratarse de realizar pruebas, se han separado en varios ficheros *.py*. La organización sería la siguiente:

- **main:** Se encuentran todos los documentos con los que se han realizado pruebas y a su vez, se almacenan todos los ficheros *.csv* que se descargan de las *API's* del Ayuntamiento Málaga, siendo el fichero final sería *Personalized-ChatGPT.py*. Por otro lado, en *requirements.txt*, se encuentran todas las librerías que se han instalado en el repositorio *conda* (aunque algunas no son necesarias al haberse utilizado para probar algunas herramientas que al final no se han llegado a utilizar en la versión final)
- **media:** Se almacenan imágenes y ficheros relevantes que se han generado durante el *fine-tuning* del modelo, también se encuentran los diseños de la arquitectura de la aplicación mostrados en el capítulo 3
- **Prompts:** Se encuentran los ficheros con las primeras pruebas, el fichero de extracción de entidades, los distintos ficheros de *fine-tuning* finales de las distintas versiones generadas por la herramienta y el fichero sobre el que se han puesto todos los ejemplos del *fine-tuning*.

El repositorio, se puede encontrar en el siguiente enlace: <https://github.com/alberto-atk/TFM>

## Apéndice B

# Anexo II: Conversaciones con ChatGPT para generación ejemplos y extracción de entidades

A continuación, se adjuntan las conversaciones que se mantuvieron con ChatGPT para la mayor parte de la generación del lote de ejemplos para el *fine-tuning*. En algunos casos, el lote de ejemplos final (localizado en el repositorio), puede variar con esta versión debido a que se introdujeron casos distintos o se tuvieron que realizar modificaciones ya comentadas por cambios en el código o requisitos del *fine-tuning*. A la hora de mostrarlos, se va a adjuntar un enlace con el que se puede ver el chat.

### B.1. Tiempo

`https://chat.openai.com/share/9bb1f250-f8f0-4e79-9b04-49a5c726b402`

### B.2. Cargadores

`https://chat.openai.com/share/0983d764-6002-4ae1-b78b-b6e4063e791d`

### B.3. Aparcamientos

`https://chat.openai.com/share/45bbbb48-379c-4997-a89b-17e79421b8bb`

### B.4. Monumentos

`https://chat.openai.com/share/44a78c64-827b-44b5-aa09-2412e93e6dca`

### B.5. Primer generador (pruebas con ChatGPT)

`https://chat.openai.com/share/719a9006-5d38-490d-a3b5-f0672b5150b0`