

---

Experimentos con la API de ChatGPT  
Experiments with the ChatGPT API

---



Trabajo de Fin de Grado  
Curso 2023–2024

**Autores**

Mario Alcolea Alcolea  
Jose Javier Carrasco Ferri

**Director**

Miguel Palomino Tarjuelo

Grado en Ingeniería Informática  
Facultad de Informática  
Universidad Complutense de Madrid



# Resumen

## Experimentos con la API de ChatGPT

Desde hace un par de años, el uso de aplicaciones basadas en inteligencia artificial ha experimentado un notable aumento en la esfera de los usuarios no expertos. La introducción de la tecnología Transformer ha sido un factor determinante en este crecimiento ya que en poco tiempo de desarrollo los modelos de lenguaje grandes han demostrado ser capaces de realizar tareas de procesamiento de lenguaje natural de forma eficiente y precisa.

Este nuevo panorama tecnológico abre oportunidades para el desarrollo de aplicaciones innovadoras que aprovechen estas nuevas tecnologías. Por ende, el propósito de este estudio es realizar un análisis del uso de la API de uno de estos modelos de lenguaje de gran tamaño en el desarrollo de aplicaciones.

La utilización de las APIs de estos modelos de lenguaje marca un nuevo enfoque en el desarrollo de aplicaciones de software. En este proyecto, hemos creado una aplicación web que emplea las API de diferentes modelos para gestionar interacciones con el usuario. Esta aplicación actúa como tutor, corrector y juez de problemas de programación relacionados con la algoritmia y estructuras de datos. Gracias al uso de las API de ChatGPT y Claude, la aplicación puede proporcionar asistencia, correcciones y evaluaciones relacionadas con estos problemas.

Además, se llevó a cabo una evaluación para determinar la fiabilidad y las ventajas de utilizar esta tecnología en comparación con los métodos de desarrollo de software más convencionales.

## Palabras clave

Inteligencia Artificial, Modelos de Lenguaje, ChatGPT, API, Aplicaciones Web, Desarrollo de Software, Tutor, Juez.



# Abstract

## Experiments with the ChatGPT API

In recent years, the use of applications based on artificial intelligence has experienced a notable increase in the sphere of common technology users. The introduction of Transformer technology has been a driving factor in this growth, as in a short development time, large-scale language models have proven to be extremely useful for a wide range of users.

This new technological landscape opens opportunities for the development of innovative applications that take advantage of these new technologies. Therefore, the purpose of this study is to perform an analysis of the use of the API of one of these large language models in application development.

The use of the APIs of these language models marks a new approach in the development of software applications. In this project, we have created a web application that uses the API from different models to manage user interactions. This application acts as a tutor, corrector and judge of programming problems related to algorithms and data structures. Using the API from ChatGPT and Claude, the app can provide support, fixes, and reviews related to these issues.

Additionally, a comprehensive evaluation was conducted to determine the reliability and benefits of using this technology compared to more conventional software development methods.

## Keywords

Artificial Intelligence, Language Models, ChatGPT, API, Web Applications, Software Development, Tutor, Judge.



# Índice

<b>1. Introducción</b>	<b>1</b>
1.1. Motivación y objetivos . . . . .	1
1.2. Contexto y antecedentes . . . . .	1
1.3. Inteligencia Artificial . . . . .	2
1.3.1. Procesamiento de Lenguaje Natural . . . . .	2
1.3.2. ¿Qué es ChatGPT? . . . . .	2
1.3.3. Diferentes tipos de modelos inteligencia artificial . . . . .	2
1.4. Contribuciones . . . . .	3
<b>2. Modelos de Lenguaje Grandes (LLM)</b>	<b>5</b>
2.1. Funcionamiento . . . . .	5
2.1.1. Transformadores . . . . .	6
2.1.2. Tokenización . . . . .	7
2.1.3. <i>Prompting</i> . . . . .	7
2.2. Utilidad de LLM en desarrollo de aplicaciones . . . . .	8
2.3. ChatGPT . . . . .	8
2.4. Otros Modelos de Lenguaje Grandes . . . . .	9
2.5. Benchmarks . . . . .	10
<b>3. Desarrollo de la aplicación</b>	<b>11</b>
3.1. Introducción . . . . .	11
3.2. Herramientas . . . . .	12
3.2.1. API de OpenAI . . . . .	12
3.2.2. API de Anthropic . . . . .	14
3.2.3. Django . . . . .	14
3.3. Arquitectura de la aplicación . . . . .	17
3.3.1. Controlador de las API . . . . .	17
3.3.2. Implementación de la API de OpenAI . . . . .	18
3.3.3. Aplicación . . . . .	25
3.3.4. Seguridad . . . . .	32
<b>4. Evaluación y Trabajo Futuro</b>	<b>35</b>

4.1. Fiabilidad de la herramienta . . . . .	35
4.1.1. Rendimiento . . . . .	37
4.1.2. Comparación con otras herramientas de tipo juez . . . . .	38
4.2. Desarrollo con modelos de lenguaje . . . . .	38
4.2.1. Otros ejemplos de uso aplicaciones . . . . .	39
4.3. Trabajo futuro . . . . .	39
4.3.1. Entrenamiento específico de un modelo la aplicación . . . . .	39
4.3.2. Función de chat . . . . .	40
4.3.3. Funcionalidad para profesores . . . . .	40
<b>Conclusiones</b>	<b>41</b>
<b>Conclusions</b>	<b>43</b>
<b>Aportaciones Personales</b>	<b>45</b>

# Introducción

## 1.1. Motivación y objetivos

El crecimiento exponencial en la adopción de la tecnología de la inteligencia artificial ha generado un interés en explorar sus posibles aplicaciones y en cómo puede contribuir al desarrollo de soluciones innovadoras para desafíos en el proceso de desarrollo software.

Este impulso nos ha llevado a profundizar en la investigación para comprender mejor cómo podemos aprovechar esta tecnología de manera efectiva en la creación de aplicaciones y en la resolución de una amplia gama de problemas. Desde abordar obstáculos técnicos hasta encontrar soluciones para desafíos complejos, queremos explorar el potencial que esta tecnología ofrece para impulsar avances significativos en el desarrollo de aplicaciones y en la resolución de problemas en múltiples sectores y disciplinas.

En nuestro caso, nos centramos en aplicaciones útiles en el mundo académico. Como estudiantes de informática, vemos mucho potencial en el uso efectivo de la inteligencia artificial aplicada a la docencia.

## 1.2. Contexto y antecedentes

En la actualidad, estamos inmersos en un panorama de intensa competencia en el ámbito del desarrollo de la inteligencia artificial (IA). Las principales empresas tecnológicas a nivel mundial están canalizando enormes inversiones financieras con el objetivo de concebir y perfeccionar los productos más innovadores y avanzados en este campo. Este fervor competitivo refleja la creciente importancia y el potencial transformador de la IA en diversos sectores y disciplinas.

Uno de estos sectores, es el de la educación y la docencia. Nuestro caso más cercano es el del grado de Ingeniería Informática, donde vemos que los modelos de lenguaje grandes pueden ser de gran ayuda para mejorar las herramientas que ya

existen. Aplicaciones web como DomJudge evalúan código según ciertos parámetros. Sin embargo, solo hace de juez y no ofrece retroalimentación sobre los fallos en el código. Esto nos llevó a pensar que es necesario para los alumnos obtener una respuesta más informativa sobre lo que hay que corregir y ahí es donde una aplicación web más completa puede solucionar este problema.

El objetivo de la aplicación es ofrecer una respuesta didáctica para el alumno sobre qué debe cambiar en su código para que pueda pasar el juez y dar la solución como correcta. Para ello vamos a hacer uso de modelos de lenguaje grandes para dar respuestas educativas a distintos tipos de problemas relacionados con la algoritmia y las estructuras de datos.

## 1.3. Inteligencia Artificial

### 1.3.1. Procesamiento de Lenguaje Natural

El procesamiento de lenguaje natural es el campo de la inteligencia artificial que trata de comunicarse con los usuarios a través de lenguas naturales, como son los idiomas humanos. Para esto se utilizan distintos tipos de técnicas como puede ser el *word embedding*, que designa un conjunto de métodos de aprendizaje para representar palabras de un texto mediante vectores numéricos reales. A esas palabras se les denomina tokens. Con esto podemos originar diferentes aplicaciones para comunicar la máquina con el usuario, como por ejemplo ChatGPT.

### 1.3.2. ¿Qué es ChatGPT?

ChatGPT es un chatbot de inteligencia artificial generativa desarrollado por OpenAI especializado en el modelado de lenguaje. Ha sido creado para mantener conversaciones y responder preguntas a través de entradas de texto, más conocidas como *prompts*, lo más precisa y claramente posible. ChatGPT ha sido entrenado con técnicas de aprendizaje automático y procesamiento de lenguaje natural, sin acceso a Internet pero programado para responder preguntas sobre una amplia variedad de temas.

### 1.3.3. Diferentes tipos de modelos inteligencia artificial

Podemos clasificar algunos de los modelos de inteligencia artificial más populares según su función:

- **Modelos para generación de texto:** Los modelos de lenguaje grandes (LLM) son capaces de generar texto de manera coherente y con sentido. Pueden responder preguntas y mantener conversaciones.
- **Modelos para generación de imágenes:** Este tipo de modelos generan imágenes a partir de una descripción textual o de un conjunto de imágenes de entrada.

- **Modelos para generación de música:** Estos modelos son capaces de componer música a través de la descripción de un estilo musical, instrumentos y letra. Además, pueden generar música a partir de un fichero de audio dado como entrada.
- **Modelos para generación de vídeo:** Estos modelos son capaces de generar vídeo de forma similar a la generación de imágenes. Generan vídeo a partir de una descripción textual usando las mismas técnicas que los modelos de generación de imágenes, pero escalado y optimizado para vídeo.

## 1.4. Contribuciones

Las contribuciones de este trabajo son una estructuración sobre significados y usos de varios modelos de lenguaje y su utilidad en el desarrollo software, además de una aplicación sobre corrección didáctica de ejercicios que implican código, que pueda servir para asignaturas como Fundamentos de la Algoritmia, Estructura de Datos, Métodos Algorítmicos en Resolución de Problemas, etc.



# Modelos de Lenguaje Grandes (LLM)

## 2.1. Funcionamiento

Los Modelos de Lenguaje Grandes, o LLM por sus siglas en inglés, *Large Language Models*, son algoritmos de aprendizaje automático diseñados para entender y generar texto en lenguaje humano de manera similar a como lo hacen las personas. Estos modelos son capaces de comprender y producir texto coherente y relevante en respuesta a una variedad de entradas de texto, como preguntas, solicitudes o instrucciones.

El funcionamiento básico de un LLM implica una arquitectura de red neuronal profunda, como las redes neuronales recurrentes (RNN), las redes neuronales convolucionales (CNN) o los transformadores. Estas arquitecturas permiten al modelo aprender patrones complejos en el lenguaje humano a través del entrenamiento con grandes cantidades de datos textuales.

Durante el entrenamiento, un LLM es expuesto a enormes conjuntos de datos que contienen texto en varios idiomas y de diversas fuentes, como libros, artículos de noticias, sitios web y redes sociales. El modelo aprende a predecir la siguiente palabra o token en una secuencia de texto dada la historia previa de tokens. Este proceso se conoce como “aprendizaje autoregresivo” y es fundamental para la capacidad del modelo para generar texto coherente y fluido.

El desarrollo de un LLM implica múltiples etapas. En primer lugar, se preprocesan los datos para limpiarlos y estructurarlos adecuadamente. Luego, se entrena el modelo utilizando técnicas de aprendizaje supervisado, donde el objetivo es minimizar la diferencia entre las predicciones del modelo y los datos reales. Durante el entrenamiento, se ajustan los pesos y parámetros del modelo para mejorar su capacidad para capturar patrones en el lenguaje humano.

Es importante destacar que el entrenamiento de un LLM requiere una gran cantidad de recursos computacionales, especialmente cuando se utilizan modelos muy grandes y conjuntos de datos masivos. Además, el proceso de entrenamiento puede

llevar mucho tiempo, desde días hasta semanas o incluso meses, dependiendo del tamaño del modelo y la cantidad de datos utilizados.

Una vez que el modelo ha sido entrenado con éxito, se puede utilizar para una variedad de tareas relacionadas con el procesamiento del lenguaje natural, como la generación de texto, la traducción automática, el resumen de texto, el análisis de sentimientos y mucho más.

### 2.1.1. Transformadores

Los transformadores son una arquitectura de red neuronal innovadora que ha impulsado muchos de los avances recientes en el procesamiento del lenguaje natural (NLP). Los transformadores se utilizan en una variedad de modelos de lenguaje grandes y han demostrado ser altamente efectivos en tareas como la traducción automática, la generación de texto y el análisis de sentimientos.

La arquitectura de los transformadores se basa en la atención, un mecanismo que permite al modelo enfocarse en partes específicas del texto de entrada durante el procesamiento. La atención permite que el modelo considere la relevancia de cada palabra o token en relación con todas las demás palabras o tokens en el texto de entrada, lo que le permite capturar relaciones de largo alcance y contextos complejos de manera más efectiva que las arquitecturas de redes neuronales recurrentes o convolucionales tradicionales.

Los transformadores consisten en un codificador y un decodificador, cada uno compuesto por múltiples capas de atención y capas de redes neuronales totalmente conectadas. Durante la fase de codificación, el modelo procesa el texto de entrada y lo convierte en una representación vectorial de alta dimensionalidad que captura su significado y contexto. Durante la fase de decodificación, el modelo utiliza esta representación para generar una secuencia de salida.

Los transformadores se entrenan utilizando técnicas de aprendizaje supervisado con grandes conjuntos de datos textuales. Durante el entrenamiento, se ajustan los pesos y parámetros del modelo utilizando algoritmos de optimización para minimizar la diferencia entre las predicciones del modelo y los datos reales.

### 2.1.2. Tokenización

La tokenización es un paso interno que ocurre dentro del modelo mismo durante el procesamiento de las solicitudes de los usuarios. Cuando envías una solicitud, el modelo primero tokeniza el texto de entrada. Este proceso implica dividir el texto en unidades más pequeñas llamadas tokens, que generalmente corresponden a palabras individuales, partes de palabras o caracteres. Esta tokenización se realiza para que el modelo pueda procesar el texto de manera más eficiente y efectiva.

Una vez que el texto se ha tokenizado, el modelo puede comprenderlo mejor y generar una respuesta adecuada. Después de procesar la entrada tokenizada, el modelo produce una secuencia de tokens que representa la respuesta generada. La tokenización se realiza automáticamente como parte del proceso de inferencia del modelo y no es visible ni controlable directamente por el usuario.

### 2.1.3. *Prompting*

El *prompting* es una técnica de guía sobre un modelo entrenado que consiste en proporcionar unas instrucciones específicas, conocidas como *prompts*, para guiar su generación de texto. El *prompt* es una oración o frase que describe la tarea o el contexto en el que se espera que el modelo genere una respuesta.

Para entender la relevancia del *prompting* en el entrenamiento de modelos de lenguaje es crucial considerar sus ventajas respecto a otros enfoques. Con el aprendizaje supervisado convencional la obtención de datos para entrenar un LLM puede ser un proceso muy costoso, llevando aproximadamente un mes. El propio proceso de entrenamiento del modelo con estos datos puede extenderse hasta tres meses adicionales. Luego, se requiere tiempo adicional, otros tres meses, para obtener un modelo completamente instalable y listo para ser utilizado por los usuarios.

En contraste, el uso de un enfoque basado en *prompts* puede reducir drásticamente el tiempo de trabajo necesario. En el caso de ChatGPT, generar un *prompt* específico puede llevar tan solo minutos o, en el peor de los casos, horas. Asimismo, el tiempo necesario para obtener un sistema completamente utilizable también se ve significativamente reducido. Esta disminución en el tiempo de trabajo es especialmente valiosa en entornos donde la velocidad y la eficiencia son críticas, como en el desarrollo de aplicaciones de inteligencia artificial para el procesamiento del lenguaje natural. Además, el enfoque basado en *prompts* puede ofrecer una mayor flexibilidad y adaptabilidad, permitiendo a los desarrolladores ajustar y refinar rápidamente los modelos en función de las necesidades cambiantes de los usuarios y del contexto.

Existen varias estrategias para obtener *prompts* efectivos [8]. Una de las más comunes es proporcionar ejemplos de entrada y salida esperados al modelo, lo que le permite aprender a generar respuestas coherentes y precisas. Por ejemplo, si queremos resaltar cierta información de la entrada podemos añadir información sobre delimitadores o etiquetas que se deben tener en cuenta. Otra estrategia es utilizar

*prompts* que describan claramente la tarea o el contexto en el que se espera que el modelo genere una respuesta. Por ejemplo, si se desea que el modelo responda a preguntas sobre un tema específico, el *prompt* podría incluir una breve descripción del tema y algunas preguntas de ejemplo. Otra buena idea es dividir tareas complejas en subtareas más simples. Al igual que en ingeniería de software es una buena práctica descomponer un problema complejo en sus componentes simples, lo mismo sirve para un modelo de lenguaje grande. Es más probable que existan errores en tareas complejas que en tareas simples. Una forma de conseguirlo es especificar en el *prompt* que trabaje en la solución paso a paso. Por ejemplo, para aplicaciones que requieren conversaciones muy extensas se pueden resumir o filtrar los mensajes anteriores.

## 2.2. Utilidad de LLM en desarrollo de aplicaciones

Los LLM ofrecen a los usuarios una experiencia adaptada y contextualizada. Esta capacidad adquiere particular relevancia en entornos profesionales, donde un LLM puede funcionar como un asistente para usuarios ofreciendo información especializada. Es aquí donde encontramos un uso de los LLM en el desarrollo de aplicaciones software. Con esta tecnología, es posible delegar algunas funcionalidades en peticiones a estos modelos y tratar la respuesta proporcionada. Mediante instrucciones detalladas de la forma de la respuesta en la petición podemos conseguir un resultado interpretable por el resto de métodos de desarrollo software.

Por ejemplo, consideremos una aplicación de servicio al cliente en línea para una empresa de telecomunicaciones. Los clientes pueden ingresar consultas sobre sus servicios, tales como “¿Cuál es mi saldo actual?”, “¿Cuándo vence mi factura?” o “¿Cómo puedo cambiar mi plan de datos?”. Al integrar un modelo de lenguaje grande en la aplicación, podemos delegar estas consultas de usuario en una petición al modelo, con unas instrucciones concretas proporcionadas por el programador. El modelo sería capaz de consultar la base de datos y devolver como respuesta una estructura de datos para ser procesada y mostrada al usuario. Esta interacción con el modelo puede ser escalada a muchas funcionalidades más complejas, dependiendo del tipo de aplicación y del modelo que se use.

## 2.3. ChatGPT

ChatGPT es una aplicación que usa modelos de lenguaje grandes implementados con la arquitectura basada en transformadores, desarrollada por OpenAI. Su diseño se fundamenta en una red neuronal de gran escala, pre-entrenada mediante el aprendizaje supervisado en grandes volúmenes de datos textuales provenientes de diversas fuentes. Su modelo nos proporciona unas funciones muy variadas, ya sea aportar una solución a problemas prácticos, creación de textos concretos, consultar información y asistir en la toma de decisiones.

ChatGPT cuenta con diferentes modelos de lenguaje grandes llamados *GPT*. El modelo *GPT-3.5 Turbo* es el más extendido, estando disponible su uso para todos los usuarios. Sin embargo, existe también los modelos *GPT-4* y *GPT-4 Turbo*, que son más potentes y capaces de realizar tareas más complejas, ofreciendo al usuario más posibilidades.

Además, ChatGPT nos ofrece una serie de herramientas para desarrolladores que nos permite integrar sus funcionalidades en nuestras aplicaciones. Estas herramientas nos permiten realizar peticiones al modelo, que podemos procesar y mostrar al usuario.

## 2.4. Otros Modelos de Lenguaje Grandes

Además de ChatGPT, existen otros modelos de lenguaje grandes que han sido desarrollados por diferentes organizaciones y empresas tecnológicas. Estos otros modelos también ofrecen funcionalidades similares a las de ChatGPT, como la generación de texto, la traducción automática, el análisis de sentimientos y la respuesta a preguntas. Algunos de los modelos de lenguaje grandes más populares son:

- **Gemini:** Desarrollado por Google, Gemini es un conjunto de modelos de lenguaje grande que tuvo su lanzamiento en 2023. Cuenta con varios modelos diferentes, variando en la capacidad de procesamiento. Su modelo más potente, *Gemini Ultra* está optimizado para tareas de alta complejidad y gran volumen de datos. Otro modelo de la familia Gemini es *Gemini Nano*, que es más ligero y está diseñado para poder ser ejecutado en dispositivos con menos recursos como móviles o tabletas.
- **Claude:** Desarrollado por Anthropic, Claude es otro conjunto de modelos de lenguaje grande lanzados en marzo de 2023. Su versión más reciente es la 3.0, formada por 3 modelos diferentes. Al igual que Gemini, estos modelos varían en capacidad de procesamiento y están diseñados para adaptarse a diferentes necesidades de los usuarios. Claude destaca por el tamaño permitido para la entrada de texto, 200.000 tokens, que aproximadamente equivale a 150.000 palabras.
- **Meta Llama 3:** Desarrollado por Meta, este conjunto de modelos de lenguaje grande es uno de los pocos que son de código abierto. Sus diferentes versiones se diferencian en el tamaño de los parámetros del modelo, uno contando 8 billones de parámetros y el otro contando con 70 billones.

Estos modelos de lenguaje grandes ofrecen a los usuarios una amplia gama de funcionalidades y capacidades, pero es importante saber cómo elegir el modelo adecuado para cada caso de uso. Para poder comparar entre los diferentes modelos, existen una serie de *benchmarks* que nos permiten comparar el rendimiento de los modelos en diferentes tareas.

## 2.5. Benchmarks

Un *benchmark* relevante es el GPQA (*Graduate-Level Google-Proof Q&A Benchmark*) [11]. Este *benchmark* está diseñado para evaluar la capacidad de los modelos de lenguaje grandes para responder correctamente a 448 preguntas de respuesta múltiple acerca de temas como biología, física y química. Existen tres variaciones del GPQA con diferentes longitudes de texto de entrada en la pregunta: *extended* (546), *main* (448) y *diamond* (198). Actualmente el modelo que lidera el GPQA es el *Claude 3 Opus*, el modelo más potente de Anthropic.

Otro *benchmark* relevante para este proyecto es el *HumanEval* [4]. Este *benchmark* se usa para medir la correcta funcionalidad para sintetizar código. Consiste en 164 problemas de programación, donde se evalúa la comprensión del problema, algoritmos y matemática. De los modelos de lenguaje grandes mencionados anteriormente, el que lidera este *benchmark* actualmente es de nuevo *Claude 3 Opus*. Para este proyecto es especialmente relevante, ya que se espera que el modelo de lenguaje grande que se use sea capaz de responder a preguntas sobre código y programación.

Existen *benchmarks* que se pueden usar para comparar el rendimiento de modelos de lenguaje grandes acerca de otras funcionalidades. El *benchmark HellaSwag* evalúa la capacidad de los modelos sobre el sentido común. Esta compuesto por preguntas de respuesta múltiple diseñadas para que un humano pueda responderlas con facilidad, pero que un modelo de lenguaje grande pueda encontrar dificultades.

## Desarrollo de la aplicación

### 3.1. Introducción

En este capítulo se describe el desarrollo de la aplicación **AlgoritmIA**, que se basa en las API de varios modelos de lenguaje grande y en el framework Django para aplicaciones web. Se detallan las herramientas utilizadas, la arquitectura de la aplicación y el modelo de datos empleado. El código de la aplicación se encuentra en el siguiente repositorio [12].

La aplicación web AlgoritmIA es un tutor para estudiantes de programación que ofrece diferentes ayudas al aprendizaje mediante la resolución de problemas. En otras herramientas basadas en inteligencia artificial, como en la aplicación web de chat de ChatGPT o Github Copilot, el usuario recibe la solución directamente. Sin embargo, en AlgoritmIA, la aplicación guía al usuario hacia la solución asegurando un proceso de aprendizaje. Para ello, se proporcionan al usuario diferentes opciones sobre el enunciado del problema:

- **Casos de prueba:** Se proporcionan casos de prueba para que el usuario pueda probar si su código tiene la salida esperada.
- **Esqueleto del código:** Se genera un esqueleto del código con la estructura básica del problema que incluye la lógica de entrada por teclado y salida por pantalla.
- **Eficiencia:** Se indica la complejidad en espacio y tiempo en el caso mejor que se puede resolver el problema. Si el usuario ha proporcionado código solución, se hace un análisis de la complejidad en espacio y tiempo del código.
- **Pistas:** Se aportan varias indicaciones útiles para guiar al usuario en la resolución del problema, como el tipo de problema o método de resolución.
- **Errores:** Se corrigen los errores del código proporcionado por el usuario en el caso de haberlos. En caso contrario se felicita al usuario por su solución.

## 3.2. Herramientas

### 3.2.1. API de OpenAI

La aplicación AlgoritmIA se ha desarrollado para ser capaz de utilizar diferentes modelos de lenguaje. Dependiendo del modelo de lenguaje que se utilice, se tiene que utilizar una API u otra. Muchas de estas API de estos modelos tienen una estructura similar y se utilizan de la misma forma. Para ejemplificar el uso de estas API vamos a usar la API de OpenAI. Una API (*Application Programming Interface*) es un conjunto de funciones y procedimientos que nos permiten la comunicación entre diferentes aplicaciones. En este caso, la API de OpenAI nos permite realizar llamadas al modelo de lenguaje grande de *GPT-3.5 Turbo*. Oficialmente la API está disponible vía una librería de Python o de `Node.js`, aunque también soporta librerías para otros lenguajes creados por una comunidad de desarrolladores.

Como primer paso para utilizar la API es necesario registrarse en la página web de OpenAI y obtener una clave de desarrollador, ya sea para un proyecto o una organización. La clave de desarrollador es un token de acceso privado y obligatorio para autenticar las solicitudes realizadas. La clave se puede guardar en una variable de entorno o en un fichero de configuración. Hay que cuidar la privacidad de la clave y no compartirla en repositorios o con los usuarios de la aplicación, ya que los recursos ofrecidos por OpenAI no son gratuitos. En segundo lugar es necesario instalar el paquete de Python de openAI usando:

```
pip install openai
```

Para esta aplicación solo necesitamos usar los modelos de generación de texto de OpenAI. Para enviar una solicitud se envía una petición HTTP que contiene las entradas y nuestra clave de la API y se recibe una respuesta con la salida generada por el modelo. La parte principal de la entrada es una lista de mensajes representados por diccionarios en los que a cada mensaje se le asigna un rol y un contenido. El rol puede ser *user* o *assistant* o *system* y el contenido es el texto del mensaje (*prompt*) como se explica en la documentación [6]. Un ejemplo de petición al modelo sería:

```
from openai import OpenAI
client = OpenAI()

response = client.chat.completions.create(
    model="gpt-3.5-turbo",
    messages=[
        {"role": "system", "content": "Eres un asistente de\
propósito general."},
        {"role": "user", "content": "Cuál es el sentido de la vida?"},
        {"role": "assistant", "content": "42"},
        {"role": "user", "content": "Por qué 42?"}
    ]
)
```

- **Rol system:** Establece el comportamiento del asistente. Por ejemplo, proporcionar instrucciones específicas sobre cómo debe comportarse durante la conversación. Sin embargo, se debe tener en cuenta que el mensaje del sistema es opcional y es probable que el comportamiento del modelo sin un mensaje del sistema sea similar al uso de un mensaje genérico como “Eres un asistente servicial”.
- **Rol user:** Los mensajes del usuario. Son las solicitudes a las que el asistente responde.
- **Rol assistant:** Las respuestas previas almacenadas del modelo. También pueden ser escritos por el programador para dar ejemplos del comportamiento deseado.

Todos los mensajes se leen y escriben mediante tokens. El número de tokens en una llamada a la API impacta en el coste de la misma, ya que se factura por cada token, y en el tiempo de respuesta de la llamada, ya que más tokens implican más tiempo de procesamiento. El límite máximo de tokens permitidos en una llamada es de 4097 para el modelo *GPT-3.5 Turbo*. Tanto los tokens que forman la entrada y la salida cuentan para este total, por lo tanto es importante vigilar el tamaño de los textos de instrucciones y el tamaño del texto de la respuesta. Se puede comprobar cuántos tokens se han usado en una solicitud en la estructura de datos en la que el modelo nos devuelve su respuesta [7].

Tras enviar una solicitud al modelo (como la del ejemplo anterior), se recibe una respuesta con el siguiente formato:

```
{
  "choices": [
    {
      "finish_reason": "stop",
      "index": 0,
      "message": {
        "content": "42",
        "role": "assistant"
      },
      "logprobs": null
    }
  ],
  "created": 1677664795,
  "id": "chatcmpl-7QyqpwhqwjicIEznoc6Q47XAYW",
  "model": "gpt-3.5-turbo-0613",
  "object": "chat.completion",
  "usage": {
    "completion_tokens": 1,
    "prompt_tokens": 13,
    "total_tokens": 14
  }
}
```

La respuesta tiene un formato JSON y contiene diferentes campos. Los más relevantes son:

- **Campo choices:** Lista de opciones generadas por el modelo en respuesta a la entrada. Sus contenidos más relevantes son la respuesta con el rol de asistente y el motivo de finalización.
- **Campo usage:** Información sobre el uso de tokens en la solicitud. Se detalla el número de tokens usados en la entrada, en la salida y el total.

### 3.2.2. API de Anthropic

Anthropic nos proporciona una API muy similar a la de OpenAI. De nuevo, para poder utilizarla es necesario registrarse en la página web de Anthropic y obtener una clave de desarrollador. De forma similar a OpenAI, para poder utilizar la API de Anthropic es necesario instalar el paquete de Python de Anthropic usando:

```
pip install anthropic
```

Una vez instalado, las llamadas al modelo de lenguaje de Anthropic se realizan de forma similar a las de OpenAI. Contiene parámetros similares como el modelo a utilizar, el mensaje de sistema y el mensaje del usuario.

```
import anthropic

client = anthropic.Anthropic(
    api_key="my_api_key",
)

message = client.messages.create(
    model="claude-3-opus-20240229",
    system="Respond only in Yoda-speak.",
    messages=[
        {"role": "user", "content": "How are you today?"}
    ]
)
```

Estas similitudes entre las API nos permite cambiar de una a otra de forma sencilla. La aplicación se ha desarrollado pensando en la posibilidad de cambiar entre diferentes API sin tener que realizar cambios sustanciales en el resto del código.

### 3.2.3. Django

Django es un *framework* de desarrollo web de código abierto para la creación de aplicaciones web en Python [5]. Django incluye muchas funcionalidades que facilitan el desarrollo como la autenticación de usuarios, la gestión de bases de datos, la generación de formularios, etc. Además estas funcionalidades están implementadas de forma segura ayudando a prevenir ataques comunes en aplicaciones web. Se puede instalar fácilmente Django con el siguiente comando:

```
pip install Django
```

Para comenzar un proyecto en Django es necesario crear un nuevo proyecto. Este proyecto es un conjunto de ficheros de configuración y aplicaciones para un sitio web. Para crear un nuevo proyecto se utiliza el comando:

```
django-admin startproject nombre_proyecto
```

El fichero más relevante tras la creación es el fichero de configuración `settings.py`. En este fichero se configuran las aplicaciones que forman parte del proyecto, la base de datos, la configuración de la aplicación, la configuración de la autenticación, etc. Con el proyecto creado, se puede crear una nueva aplicación utilizando el comando:

```
python manage.py startapp nombre_aplicacion
```

Una aplicación en Django es un conjunto de ficheros que implementan una funcionalidad específica. Es necesario añadir la aplicación al proyecto en el fichero `settings.py` para que Django la reconozca. De nuevo se generan ficheros de configuración y plantillas que se pueden modificar para adaptar la aplicación a las necesidades del proyecto. Los ficheros más importantes de una aplicación son:

- `models.py`: Define los modelos de datos que se utilizan en la aplicación. Luego se pueden gestionar con el ORM (*Object Relational Mapping*) que se haya configurado.
- `views.py`: Define las funciones que se ejecutan cuando se recibe una petición HTTP.
- `urls.py`: Define las rutas de la aplicación y las funciones que se ejecutan en cada ruta.

El ORM de Django es una capa de abstracción que permite interactuar con la base de datos sin la obligación de usar SQL. Django soporta diferentes bases de datos como SQLite, MySQL, PostgreSQL, etc. Por defecto se utiliza SQLite, pero se puede cambiar la configuración en el fichero `settings.py`. En `models.py` se definen las clases que representan las tablas de la base de datos. Una definición de un modelo en Django sigue la misma estructura que una definición de una clase:

```
from django.db import models

class Fruta(models.Model):
    nombre = models.CharField(max_length=100)
    precio = models.FloatField()
```

Esta definición crea una tabla en la base de datos igual que si se hubiera creado con SQL. Como no se ha especificado una clave primaria, Django añade automáticamente un campo `id` que actúa como clave. El ORM de Django permite realizar peticiones a la base de datos de forma sencilla. Los objetos creados a partir de esta definición se pueden guardar en la base de datos con el método `save()` y se pueden consultar con el método `objects.all()` o con filtros como `objects.filter()`.

Para gestionar las vistas, Django nos proporciona un lenguaje de plantillas HTML. Este lenguaje de plantillas permite la inclusión de variables, la creación

de bucles y condicionales, la herencia de plantillas, etc. Los datos y la lógica de la plantilla de Django se especifican entre llaves y porcentajes. Un ejemplo de una plantilla HTML en Django sería:

```
<!DOCTYPE html>
<html>
<head>
  <title>{{ title }}</title>
</head>
<body>
  <h1>{{ title }}</h1>
  {% if user.is_authenticated %}
    <p>Bienvenido {{ user.username }}</p>
  {% else %}
    <p>Por favor, inicia sesión</p>
  {% endif %}
</body>
</html>
```

En este ejemplo, la plantilla recibe una variable `title` y una variable `user`. Si el usuario está autenticado, se muestra un mensaje de bienvenida con el nombre de usuario. En caso contrario, se muestra un mensaje pidiendo que se inicie sesión.

Django también tiene implementado un sistema de autenticación de usuarios. Este sistema permite gestionar el registro de usuarios, el inicio de sesión, el cierre de sesión, etc. Existen diferentes formas de gestionar la autenticación. Por defecto, Django utiliza la aplicación `django.contrib.auth` que incluye métodos seguros para registrar y autenticar usuarios. También podemos delegar la autenticación a un servicio externo, como una red social, a través de la aplicación `social_django`. En concreto, la aplicación `social_django` permite la autenticación de usuarios a través de Google con el emparejamiento del sistema OAuth2.0 [2], un protocolo de autorización que permite a una aplicación de terceros obtener acceso limitado a los recursos protegidos de un usuario sin la necesidad de que el usuario divida sus credenciales de acceso. En nuestro caso, solo la información necesaria para autenticarse. Es un protocolo seguro que se puede simplificar con el siguiente flujo:

1. El usuario hace clic en el botón de inicio de sesión con Google.
2. El usuario es redirigido a la página de inicio de sesión de Google.
3. El usuario inicia sesión en Google.
4. Google le pide al usuario que autorice la aplicación a acceder a su información.
5. Google redirige al usuario de vuelta a la aplicación con un código de autorización.
6. La aplicación intercambia el código de autorización por un token de acceso.
7. La aplicación utiliza el token de acceso para acceder a la información del usuario.

Para poder configurar la autenticación con Google, es necesario crear un proyecto en la consola de desarrolladores de Google y obtener las credenciales de `OAuth2.0`. Estas credenciales incluyen un ID de cliente y un secreto de cliente que se utilizan para autenticar la aplicación. Además, es necesario configurar las rutas de redirección una vez que el usuario se ha autenticado.

### 3.3. Arquitectura de la aplicación

La aplicación **AlgoritmIA** se ha desarrollado con el *framework* Django y se puede dividir en dos módulos diferentes. Por un lado, el módulo controlador de las API que se encarga de la comunicación con el modelo de lenguaje elegido y, por otro lado, el módulo de la aplicación web que se encarga de la interacción con el usuario. El objetivo de esta separación es seguir un patrón de diseño MVC (Modelo-Vista-Controlador) para separar la lógica de las solicitudes al modelo del código que gestiona las vistas.

#### 3.3.1. Controlador de las API

En este módulo se encuentran las clases y funciones que se encargan de enviar las solicitudes al modelo y recibir las respuestas. El controlador de las API se encarga de gestionar las solicitudes al modelo de lenguaje escogido. La estructura del módulo comienza con una clase controladora, que delega las solicitudes a diferentes clases que implementan los métodos dependiendo del modelo que se esté utilizando. El primer método de esta clase controladora es el constructor que obtiene el modelo e inicializa el cliente de la API correspondiente.

```
from .openai_manager import OpenAIManager
from .claude_manager import ClaudeManager

OPENAI_MODELS = ["gpt-3.5-turbo", "gpt-4-turbo", ...]
CLAUDE_MODELS = ["claude-1.0", "claude-2.0", ...]

MODEL_API MANAGERS = {
    'OPENAI': OpenAIManager,
    'CLAUDE': ClaudeManager,
}

class MultiModelManager:

    def __init__(self, model):
        if model in OPENAI_MODELS:
            self.manager = MODEL_API MANAGERS.get('OPENAI')(model)
        elif model in CLAUDE_MODELS:
            self.manager = MODEL_API MANAGERS.get('CLAUDE')(model)

    ...
```

Esta definición de la clase permite la adición de nuevos manejadores de API simplemente añadiendo una nueva clase y añadiendo el modelo a la lista de modelos soportados. La clase `MultiModelManager` delega las solicitudes a la clase correspondiente en función del modelo que se esté utilizando usando su instancia almacenada en la variable `manager`. Las clases deben implementar los métodos necesarios definidos en la clase controladora. En nuestra aplicación, se ha definido un método para cada tipo de solicitud que se quiera consultar acerca del problema. Estos métodos son:

```
def get_test_cases(self, prompt):
    return self.manager.get_test_cases(prompt)

def get_skeleton(self, prompt):
    return self.manager.get_skeleton(prompt)

def get_efficiency(self, prompt):
    return self.manager.get_efficiency(prompt)

def get_hints(self, prompt):
    return self.manager.get_hints(prompt)

def get_errors(self, prompt):
    return self.manager.get_errors(prompt)

def moderation(self, prompt):
    return self.manager.moderation(prompt)
```

Estos métodos delegan las solicitudes y las implementaciones se realizan de acuerdo al uso de la API correspondiente al modelo. La clase controladora se instancia en el fichero `settings.py` del proyecto junto con el modelo que se va a utilizar. Esto permite cambiar el modelo de lenguaje simplemente cambiando una sola sentencia en el fichero de configuración.

### 3.3.2. Implementación de la API de OpenAI

La implementación que gestiona las solicitudes y respuestas a un modelo de OpenAI se ha encapsulado en la clase `OpenAIManager`. El primer método de esta clase es el constructor que inicializa el cliente de OpenAI. Para ello es necesario importar la librería de OpenAI y las librerías de Python `os` y `dotenv` para leer la clave de la API. Para inicializar el cliente es necesario pasar la clave de la API como parámetro. La clave de la API se guarda en una variable de entorno o en un fichero `secrets.env` para proteger la privacidad de la clave y la leemos con la función `os.environ.get()` si es una variable de entorno y con la función `load_dotenv(dotenv_path)` si está en el fichero `secrets.env`. Guardamos el objeto del cliente en la variable `client`. Este objeto se utiliza en los métodos de la clase para enviar solicitudes al modelo de lenguaje. También guardamos el modelo de lenguaje que se va a utilizar en la variable `model`, necesario también para enviar la solicitud al modelo.

```

class OpenAIManager:
    def __init__(self, model):
        dotenv_path = os.path.join\
            (os.path.dirname(__file__), '..', '..', 'secrets.env')
        load_dotenv(dotenv_path)
        self.client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))
        self.model = model

```

A continuación, se implementan los métodos definidos en la clase controladora `MultiModelManager` que se encargan de enviar solicitudes al modelo, dependiendo de la petición que se quiera realizar. Para cada uno de ellos usamos el método `chat.completions.create()` de la API de OpenAI para obtener una respuesta del modelo. Acompañamos la solicitud con un mensaje de sistema que guía al modelo en la respuesta. Todos estos mensajes comparten una introducción y están seguidos de instrucciones específicas para cada método. Todos los mensajes están definidos en el fichero `system_messages.py`. El mensaje de introducción es:

```

intro = """You are a tutor who needs to provide a help in solving |
coding problems.|
Answer always in the users chosen language.|
For example, if the user provides the problem in spanish, |
answer in spanish.|
For example, if the user provides the problem in english, |
answer in english.|
The user will input a coding problem via text.|
The user must provide a problem statement. |
The user can also provide a code solution, |
but its optional.|
Make sure you are answering in the users language.|
The user problem statement will be provided|
between this delimiter: '---'.|
The user problem code will be provided|
between this delimiter: '```'.|
"""

```

En este mensaje se establece el contexto de la conversación y se dan las instrucciones para el modelo. Es importante destacar que vamos a proporcionar al modelo un problema con enunciado y opcionalmente un código. Para que el modelo los diferencie usamos los delimitadores indicados. En el fichero `system_messages.py` se concatenan el mensaje de introducción con los mensajes específicos de cada método. Después, se agrupan en un diccionario para poder acceder a ellos de forma sencilla.

### 3.3.2.1. Métodos para cada consulta

La implementación de los métodos es similar para cada uno de ellos. Se envía una solicitud al modelo con un mensaje de sistema y un mensaje específico para la consulta. Como ejemplo se muestra el método `get_test_cases()`. Esta función se encarga de pedir al modelo casos de prueba para el problema dado. Los casos de prueba son una serie de entradas para el código del problema y las salidas esperadas. El mensaje de sistema para esta solicitud es:

```

test_cases = intro + """\
    Provide at least 5 test cases for the code solution.\
    Don't provide the code solution. Only cases.\
    Make sure yo are only providing cases.\
    The test cases must include the input and the expected output.\
    The test cases must be provided between this delimiter: '---'.\
    The test cases must be provided in the following this guide \
    in the language that you are answering:\
    An expected output in spanish would be:\
        Entrada: ...
        Salida esperada: ...

        Entrada: ...
        Salida esperada: ... \
    Another example:\
        Input: ...
        Expected_output: ...

        Input: ...
        Expected_output:... \
    """

```

En este mensaje se le indica al modelo que el usuario debe proporcionar al menos 5 casos de prueba para la solución del problema. También indicamos al modelo que los casos de prueba se deben proporcionar siguiendo un formato específico para que podamos procesarlos en el código de la aplicación. En la aplicación implementamos esta función con la llamada al modelo de la siguiente forma:

```

def get_test_cases(self, prompt):
    response = self.client.chat.completions.create(
        model=self.model,
        messages=[
            {'role':'system', 'content':system_messages['test_cases']},
            {"role": "user", "content": prompt}],
        temperature=0,
        max_tokens=1000,
    )

    return \
        response.choices[0].message.content.split('---')[1].strip()

```

En este método se pasan por parámetro el enunciado del problema en la variable `prompt` y el modelo de lenguaje utilizado. Asignamos a la variable `response` la respuesta del modelo a la solicitud. En la llamada al modelo, además del enunciado y las instrucciones, se especifica la temperatura y el número máximo de tokens que se pueden usar en la respuesta. La temperatura es un parámetro que controla la creatividad del modelo. Cuanto mayor sea la temperatura, más creativa será la respuesta. En nuestro caso, la temperatura es 0 para que la respuesta sea más precisa. Finalmente, se devuelve el contenido de la respuesta que contiene el texto de los casos de prueba. Como hemos especificado en el mensaje de sistema, los casos de prueba están delimitados por los caracteres “---”, por lo que separamos el contenido de la respuesta por este carácter y devolvemos la parte que contiene los casos de prueba. El resto de métodos se implementa de forma análoga. Lo que cam-

bia en cada método es el mensaje de sistema especificando las instrucciones para el modelo. A continuación, se describen las instrucciones de sistema para cada consulta:

**Esqueleto:** Esta función se encarga de pedir al modelo el esqueleto del código para el problema dado. El esqueleto es la definición de la función que pide el problema a resolver y las funciones donde se da el código que maneja la entrada y salida según los casos esperados. El mensaje de sistema para esta solicitud es:

```
skeleton = intro + """\n    Provide a skeleton of the code solution.|\n    The skeleton must include three methods:|\n    - An implemented and complete method that reads test cases from|\n    the user via keyboard input.|\n    This method must not be empty.|\n    - An incomplete method that only contains the function signature|\n    and parameters of the function that solves the problem.|\n    This is the only method that must be empty.|\n    Create any data structure or variable that you consider|\n    necessary.|\n    - An implemented and complete method that prints the output of|\n    the code solution.|\n    This method must not be empty.|\n    Do not include any type of text in your answer,|\n    only a code block.|\n    Do not provide any hints to the user.|\n    The code must be provided between this delimiter: ''''|\n    followed by the coding language you are using.|\n    """
```

En este mensaje se le indica al modelo que debe proporcionar al usuario el esqueleto del código solución del problema. También indicamos al modelo que debe aportar funciones para realizar las entradas y salidas del problema. Para poder obtener el esqueleto es necesario cerciorarse de que si está aportando una función con la solución se la quite, porque hay veces en las que el sistema ignora la instrucción asignada para que solo aporte el esqueleto y da la solución completa de todas formas, por tanto hay que indicarle lo que queremos de manera muy específica.

**Eficiencia:** Esta función pide al modelo un análisis de la eficiencia del problema dado, es decir la complejidad en tiempo y en espacio del mismo. La eficiencia es la explicación de la complejidad del problema y el valor de esta en tiempo y espacio. Para ello el mensaje de sistema se divide en tres, uno general en el que se van a basar los dos restantes, y los otros dos que son mensajes de sistema especiales y son los casos en los que el usuario da código solución o no. El mensaje de sistema general para esta solicitud es:

```

efficiency_central = intro + """\
    Provide an analysis of the best efficiency of the code solution.\
    Think about the most eficiente way to solve the problem\
    in time and space.\
    Don't provide the code solution. Only the analysis.\
    The analysis must be the time and space complexity.\
    The analysis must be provided in Big O notation.\
    """

```

En este mensaje se le indica al modelo que debe mostrar un análisis sobre la eficiencia del problema. Para conseguir esto se le indica que solo queremos tener el análisis, nada de código, y en dicho análisis incluir la complejidad en tiempo y espacio con un formato concreto. El formato especificado para el sistema es:

```

format = """Only provide two statement of analysis following\
    this guide in the language you are answering:\
    An expected output in spanish would be this:\
    Complejidad en tiempo: O( )\
    El algoritmo tiene complejidad en tiempo...\
    Complejidad en espacio: O( )\
    El algoritmo tiene complejidad en espacio...\
    Another example:\
    Time complexity: O( )\
    The algorithm has a time complexity...\
    Space complexity: O( )\
    The algorithm has a space complexity...\
    These two statements must be provided\
    between this delimiter: '---'.\
    """

```

Los mensajes de sistema específicos para esta solicitud son:

```

efficiency1 = efficiency_central + """\
    Provide the efficiency analysis of the most efficient\
    solution for the problem.\
    """ + format

```

En este mensaje se especifica el análisis para el propio enunciado, que es el caso en el que el usuario no aporta un código solución.

```

efficiency2 = efficiency_central + """\
    Provide the analysis of the efficiency from the code solution\
    given by the user.\
    The code is delimited by '''.\
    """ + format

```

En este mensaje se especifica el análisis de la complejidad sobre el código aportado por el usuario. En ambos mensajes se hace uso del mensaje de sistema general para seguir el mismo formato a la hora de escribir el análisis.

Esta función tiene una llamada al sistema más específica que el resto ya que debe diferenciar si el usuario aporta código o no, y en función de esto usar un mensaje de sistema u otro.

```
def get_efficiency(self, prompt):
    if ''' in prompt:
        response = self.client.chat.completions.create(
            model=self.model,
            messages=[
                {'role': 'system',
                 'content': system_messages['efficiency_code']},
                {"role": "user",
                 "content": prompt}
            ],
            temperature=0,
            max_tokens=1000,
        )
        return response.choices[0].message.content
    else:
        response = self.client.chat.completions.create(
            model=self.model,
            messages=[
                {'role': 'system',
                 'content': system_messages['efficiency_problem']},
                {"role": "user",
                 "content": prompt}
            ],
            temperature=0,
            max_tokens=1000,
        )
        return response.choices[0].message.content
```

Para diferenciar si el usuario ha enviado código o no buscamos dentro del prompt el delimitador asociado al código que es (``) para así elegir el mensaje de sistema correspondiente.

**Pistas:** Pistas se encarga de pedir al modelo distintas formas de resolver el problema sin llegar a dar el código ni la solución. Las pistas son texto en los que se indica qué tipo de problema es (recursivo, iterativo, etc.) y ciertas formas de afrontar el problema dado. El mensaje de sistema para esta solicitud es:

```
hints = intro + """\
    Provide at least couple of hints about how to solve the problem.\
    The hints must not include code.\
    One of the hints must be the type of problem it is.\
    Make sure the type of problem you provide is the proper\
    way to solve it.\
    If the problem can be resolved in different ways,\
    provide all the ways.\
    Output only a numbered list of hints.\
    The hints must be provided between this delimiter: '---'.\
    """
```

En este mensaje se indica al modelo que debe proporcionar una serie de pistas sobre cómo resolver el problema. Para ello se exige que las pistas no incluyan código, ya que si da código puede sacar la solución directamente y lo que buscamos es que en base a las pistas el usuario pueda sacar la solución. Si el problema tiene diferentes maneras de resolverse debe proporcionarlas.

**Errores:** Esta función se encarga de pedir al modelo los diferentes errores que tiene el código aportado por el usuario para resolver el problema dado. Los errores son instrucciones del código que pueden hacer que la solución falle y no aporte los valores esperados. El mensaje de sistema para esta solicitud es:

```
errors = intro + """\n    Only if there is code present in the prompt,\n    provide some errors in the code and the efficiency\n    that provokes the wrong functioning of the problem.\n    If the code provided by the user is correct \n    write that it is correct.\n    The errors must not provide a solution.\n    Explain the errors to the user.\n    Output between this delimiter: '---'.\n    """
```

En este mensaje se indica que proporcione los diferentes errores en el código que puedan provocar el mal funcionamiento del problema. También, en el caso de que así sea, si la eficiencia no es la mejor posible debe indicarlo. Si la solución proporcionada por el usuario es correcta y no tiene errores se indica. Para que a la hora de proporcionar los errores no de la solución directa, se pide que no aporten código y que sean explicados.

### 3.3.3. Aplicación

El módulo de la aplicación es el conjunto de ficheros que se encarga de la interacción con el usuario. En este módulo se gestionan las solicitudes HTTP, se gestiona la base de datos, se renderizan las plantillas HTML y se comunican con el controlador de la API para obtener las respuestas del modelo. Dentro del *framework* Django esto se puede implementar en una sola aplicación. Para realizar la configuración inicial de la aplicación en el proyecto Django se tienen que definir las rutas de la aplicación en el fichero `urls.py` del proyecto. Además es necesario añadir el nombre de la aplicación a la lista `INSTALLED_APPS` en el fichero `settings.py`. A continuación, se describen los ficheros más importantes de la aplicación y su funcionalidad siguiendo el flujo de uso de la aplicación.

#### 3.3.3.1. Login y registro

El primer paso para utilizar la aplicación es autenticarse. Para ello, se ha implementado un sistema de autenticación de usuarios con la aplicación por defecto `django.contrib.auth`, la aplicación `social_django` y la consola de desarrolladores de Google. La implementación comienza definiendo las rutas de autenticación en el fichero `urls.py` de la aplicación.

```
path('login/', views.login_view, name='login'),  
path('logout/', login_required(views.logout_view), name='logout'),
```

En estas rutas se especifica la vista que se ejecuta cuando se accede a la ruta y el nombre de la ruta. Estas rutas se almacenan en una lista `urlpatterns`. La vista de inicio de sesión se ha implementado en la función `login_view` y la vista de cierre de sesión en la función `logout_view`, ambas implementadas en el fichero `views.py`.

- `login_view`: Esta función simplemente renderiza la plantilla `login.html` que contiene un mensaje de bienvenida y el botón de inicio de sesión con Google.

```
<a href="{% url 'social:begin' 'google-oauth2' %}">  
  <button class="ucm-button">  
    Inicia sesión con tu cuenta de UCM  
  </button>  
</a>
```

El botón contiene un enlace a la ruta de inicio de sesión de Google. Esta ruta se especifica en el fichero `urls.py` del proyecto porque utiliza la aplicación `social_django`.

- `logout_view`: Esta función cierra la sesión del usuario y redirige al usuario a la página de login.

Una vez el usuario se ha autenticado, se le añade a dos tablas en la base de datos de usuarios de Django; al modelo de usuario estándar de Django y al modelo de usuario de `social_django`. En el modelo de usuario estándar se almacenan datos como el nombre de usuario, email, permisos, etc. En el modelo de usuario de `social_django` se almacenan datos específicos de la autenticación delegada, como el ID de usuario de Google y el token de acceso. La vista final de autenticación se muestra en la figura 3.1.

### 3.3.3.2. Vista principal

Una vez autenticado, el usuario es redirigido a la vista principal de la aplicación. Esta vista se define en las rutas de la aplicación como:

```
path('', login_required(views.main), name='index'),
```

En esta vista se muestra al usuario un formulario para introducir el enunciado del problema y opcionalmente un código. Con este formulario, se muestran unos botones interruptor para seleccionar el tipo de consulta que se quiere realizar. Se pueden seleccionar uno o varios para obtener diferentes tipos de ayuda. Una vez seleccionados los botones, se envía el formulario al servidor con una petición POST. La vista final se puede ver en la figura 3.2. La implementación de la vista principal se encuentra en el método `main` del fichero `views.py`. El flujo del método es el siguiente:

```
@login_required(login_url='tutor:login')
@require_http_methods(["GET", "POST"])
def main(request):

    if request.method == "GET": # Si es una petición GET
        form = SolucionForm()
        return render(request, 'index.html', {'form': form})

    # Si es una petición POST
    form = SolucionForm(request.POST)
    ...
```

El método contiene dos decoradores. El primero, `@login_required`, redirige al usuario a la página de inicio de sesión si no está autenticado. El segundo, llamado `@require_http_methods`, solo permite peticiones GET y POST. Si la petición es GET, se renderiza la plantilla `index.html` con el formulario vacío. Si la petición es POST, se asigna el formulario con los datos de la petición para poder ser procesado. El formulario se ha definido en el fichero `forms.py` y contiene los siguientes campos:

```
class SolucionForm(forms.Form):
    enunciado = forms.CharField(widget=forms.Textarea(
        attrs={'rows': 6, 'cols': 80,
              'class': 'text-area',
              'placeholder': PLACEHOLDER_ENUNCIADO}))

    codigo = forms.CharField(widget=forms.Textarea
        (attrs={'rows': 6, 'cols': 80,
              'class': 'code-area',
              'placeholder': PLACEHOLDER_CODIGO,
              'spellcheck': 'false'}), required=False)
```

El formulario contiene dos campos, uno para el enunciado del problema y otro para el código. Usamos los *widgets* `Textarea` para que los campos sean de varias líneas y se puedan introducir textos largos. Además, se han añadido atributos para personalizar el tamaño y el estilo de los campos. El campo de código es opcional, por lo que se ha añadido el atributo `required=False`. Esta forma de definir los formularios en Django permite la validación de los datos en la función `main` de la siguiente forma:

```

...
if not form.is_valid(): # Si el formulario no es valido
    return HttpResponseRedirect("Formulario invalido")

enunciado = "---" + form.cleaned_data['enunciado'] + "---"
codigo = "'''" + form.cleaned_data['codigo'] + "'''" \
    if form.cleaned_data['codigo'] else ""
prompt = enunciado + codigo
...

```

Si el formulario no es válido, se devuelve un error 400. Si el formulario es válido se obtienen los datos del formulario con el método `cleaned_data` de forma segura y se preparan para ser enviados al modelo. Se añaden los delimitadores para diferenciar entre el enunciado y el código. Se realizan las peticiones al modelo con los métodos de la clase `MultiModelManager` cuya instancia `model_manager` se ha definido en el fichero `settings.py` y se obtienen las respuestas.

```

# Los campos se asignan a las funciones
fields = {
    'casos_prueba': model_manager.get_test_cases,
    'esqueleto': model_manager.get_skeleton,
    'eficiencia': model_manager.get_efficiency,
    'pistas': model_manager.get_hints,
    'errores': model_manager.get_errors,
}

# Obtiene la respuesta del modelo de los campos solicitados,
# llamando a la función correspondiente
for field, function in fields.items():
    if request.POST.get(field) == '1':
        locals()[field] = function(prompt)

# Obtiene las variables locales
# que almacenan las respuestas del modelo
local_vars = vars()

# Crea una nueva consulta en la base de datos
# o actualiza una existente
consulta, created = Consulta.objects.get_or_create(
    usuario=request.user,
    enunciado=form.cleaned_data['enunciado'],
    defaults={
        'codigo_fuente': form.cleaned_data['codigo'],
    }
)

# Si la consulta ya existía, actualiza los campos solicitados
if not created:
    for field in fields.keys():
        if request.POST.get(field) == '1':
            setattr(consulta, field, local_vars.get(field))
    consulta.codigo_fuente = form.cleaned_data['codigo']
    consulta.save()

```

En esta sección del código se comprueba qué botones interruptor se han seleccionado, se valida y se realizan las peticiones al modelo. Se guardan las respuestas en variables locales para ser mostradas en la plantilla. Además, se guarda la consulta en la base de datos con el modelo `Consulta`. Este modelo se ha definido en el fichero `models.py` de la siguiente forma:

```
class Consulta(models.Model):
    id = models.AutoField(primary_key=True)
    usuario =
        models.ForeignKey('auth.User', on_delete=models.CASCADE)
    enunciado =
        models.CharField(max_length=1000)
    codigo_fuente =
        models.CharField(max_length=1000, null=True, blank=True)
    fecha = models.DateTimeField(auto_now_add=True)
    casos_prueba =
        models.CharField(max_length=1000, null=True, blank=True)
    esqueleto =
        models.CharField(max_length=1000, null=True, blank=True)
    eficiencia =
        models.CharField(max_length=1000, null=True, blank=True)
    pistas =
        models.CharField(max_length=1000, null=True, blank=True)
    errores =
        models.CharField(max_length=1000, null=True, blank=True)

    def __str__(self):
        return
            f"Consulta ({self.id} - {self.fecha} - {self.enunciado})"
```

En este modelo se guardan los datos de la consulta, como el enunciado, el código, la fecha, las respuestas del modelo a las consultas y el usuario que la ha realizado. El modelo se ha definido con un campo `id` que actúa como clave primaria y se ha definido un método `__str__` para mostrar una representación del modelo en la consola de administración de Django.

Una vez procesadas las consultas, se renderiza la plantilla `index.html` con las respuestas del modelo, además del formulario para realizar nuevas consultas. Las plantillas de la vista principal se han construido heredando de una plantilla base que contiene la estructura común de todas las páginas de la aplicación. En la plantilla `index.html` se definen los elementos específicos de la vista además de los ficheros JavaScript necesarios para el funcionamiento de la página. En la plantilla `base.html` se define la cabecera de la página la importación de los estilos CSS. Es el fichero del que extiende `index.html`. En la cabecera que se define en este fichero se incluye un botón de cierre de sesión y un botón que te redirige a la vista de consultas realizadas. Se puede ver la barra de navegación en la figura 3.3. Las vistas de consultas realizadas se han definido en las siguientes rutas:

```
path('consultas/', login_required(views.consultas),
     name='consultas'),
path('consultas/<int:id>/', login_required(views.consulta_id),
     name='consulta_id'),
```

En la vista de consultas se muestra un historial de las consultas realizadas por el usuario. Cada consulta se muestra en una lista de los enunciados de las consultas. Si se pulsa en un enunciado, se redirige a la vista de la consulta con el ID de la consulta en la URL. En la vista de la consulta se muestra el enunciado, el código y las respuestas del modelo a las consultas. Estas vistas se han implementado en las funciones `consultas` y `consulta_id` del fichero `views.py`.

```
@login_required(login_url='tutor:login')
def consultas(request):
    """Muestra las consultas realizadas |
    por el usuario autenticado."""
    # Obtiene las consultas realizadas por el usuario autenticado
    query = Consulta.objects.filter(usuario=request.user)
        .values_list('enunciado', 'id').order_by('-id')
    # Limita la longitud de los enunciados a 50 caracteres
    query = [(enunciado[:50] + '...' if len(enunciado) > 50
        else enunciado, id) for enunciado, id in query]
    return render(request, 'consultas.html', {'consultas': query})

@login_required(login_url='tutor:login')
def consulta_id(request, id):
    """Muestra los detalles de una consulta."""
    consulta = Consulta.objects.get(id=id)
    esqueleto = None
    if consulta.esqueleto:
        esqueleto = ast.literal_eval(consulta.esqueleto)
    return render(request, 'consulta_id.html',
        {'consulta': consulta,
        'esqueleto': esqueleto})
```

En la función `consultas` se obtienen las consultas realizadas por el usuario autenticado usando el método `filter` del modelo `Consulta`. Esta es una de las formas de realizar consultas a la base de datos en Django usando el ORM. Se obtienen los enunciados y los IDs de las consultas y se limita la longitud de los enunciados a 50 caracteres. La vista en la que se muestran se puede ver en la figura 3.5. En la función `consulta_id` se obtiene la consulta con el ID proporcionado en la URL y se renderiza la plantilla `consulta_id.html` con los detalles de la consulta.

### 3.3.3.3. Capturas de pantalla

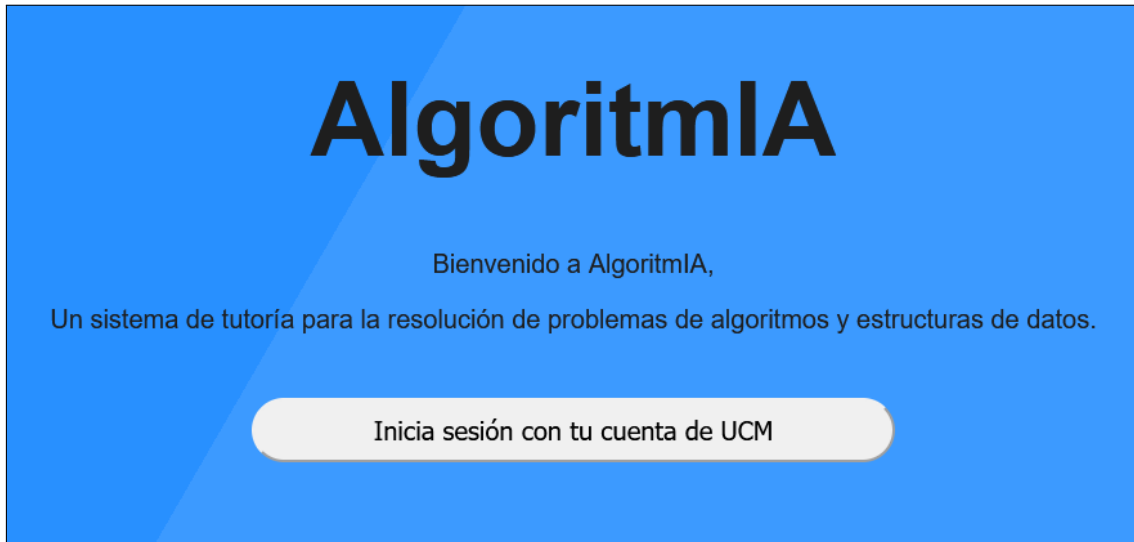


Figura 3.1: Al abrir la aplicación se nos pide iniciar sesión con nuestra cuenta de UCM.

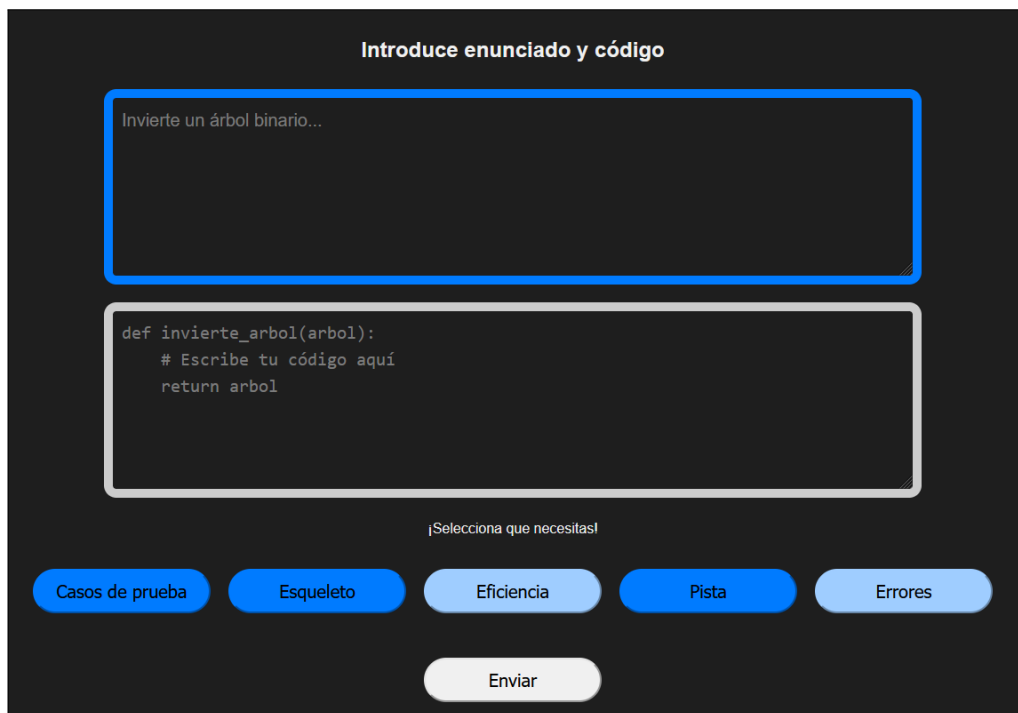


Figura 3.2: Formulario de consulta donde se escriben el enunciado y el código, además de los botones de acción



Figura 3.3: La barra de navegación incluye un botón de cierre de sesión y un botón de consultas realizadas. Además si hacemos click en el título de la aplicación volvemos a la vista principal.

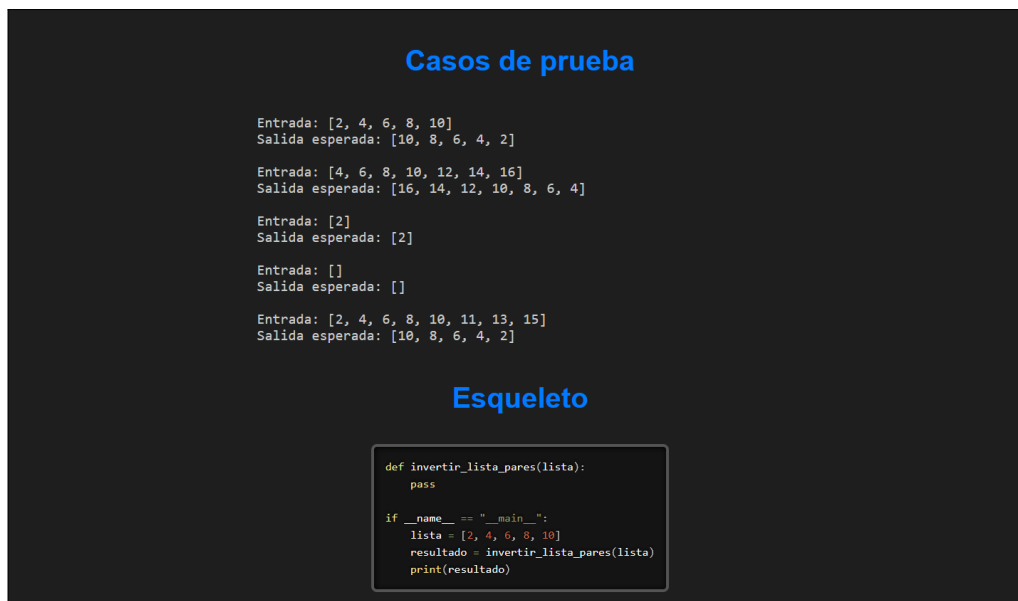


Figura 3.4: La salida de las consultas se muestran en forma de lista.

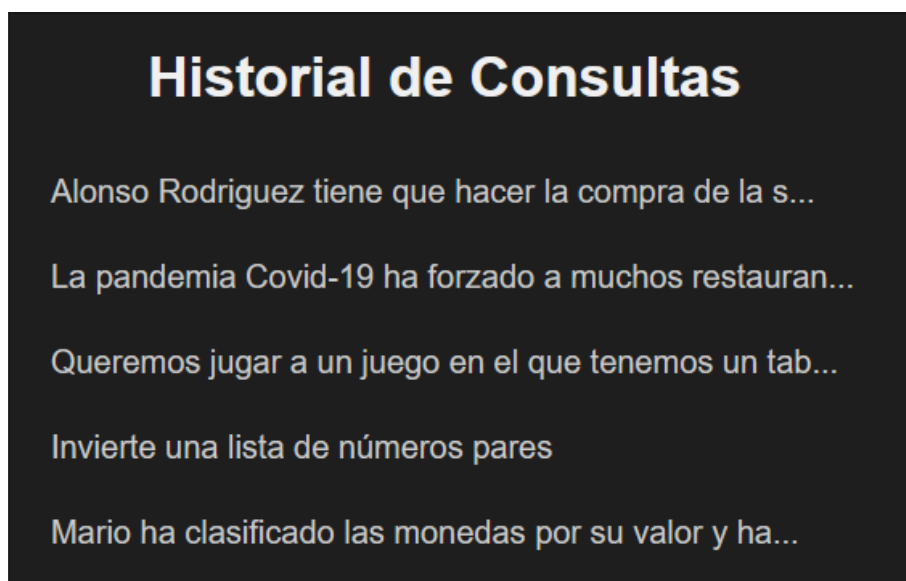


Figura 3.5: El historial de consultas se ordena de más reciente a más antigua y cada consulta se muestra con su enunciado.

### 3.3.4. Seguridad

Cuando se desarrolla una aplicación web, es importante tener en cuenta la seguridad contra ataques. Django proporciona una serie de medidas de seguridad para proteger la aplicación contra los ataques más comunes, pero también hemos tomado decisiones de diseño para mejorar la seguridad de la aplicación. Otro aspecto importante de la seguridad de la aplicación ha sido la seguridad del uso del modelo de lenguaje ya que es un recurso no gratuito de un tercero y no queremos que se use de forma maliciosa o alternativa a la que se ha diseñado. A continuación, se describen las medidas de seguridad implementadas en la aplicación.

#### 3.3.4.1. Autenticación

La seguridad de la aplicación comienza con la autenticación de los usuarios. Django proporciona por defecto un sistema de autenticación seguro que cifra las contraseñas con todas las medidas de seguridad necesarias. Cuando un usuario se añade a la base de datos, se cifra la contraseña con un algoritmo de hash seguro, usando sal y ralentizado. Aunque esta es una medida de seguridad robusta, nosotros hemos decidido delegar la autenticación en un servicio externo, en este caso Google, para no tener que gestionar las contraseñas de los usuarios. La autenticación delegada es un método seguro que permite a los usuarios autenticarse con sus credenciales de Google sin necesidad de compartir sus contraseñas. En la base de datos de Django se almacenan dos tablas, una con los datos de usuario estándar y otra con los datos de autenticación de Google.

Esta decisión de diseño mejora la seguridad de la aplicación porque no almacenamos las contraseñas de los usuarios en la base de datos. Aunque se filtre la base de datos, los atacantes solo podrían obtener los tokens de acceso de Google, que son temporales y se pueden revocar en cualquier momento. La obtención de estos tokens también se realiza de forma segura con el protocolo OAuth2.0 [3].

#### 3.3.4.2. Formularios

Los formularios son elementos críticos en la seguridad de una aplicación web ya que permiten al usuario enviar datos al servidor. Django proporciona un sistema de formularios con varias medidas de seguridad para proteger la aplicación contra ataques CSRF (*Cross-Site Request Forgery*) e inyección de código. A continuación se describen estas medidas de seguridad:

- **CSRF:** Django protege la aplicación contra ataques CSRF con un token de seguridad que se añade a los formularios. Este token se genera de forma aleatoria y se almacena en la sesión del usuario. Cuando se envía un formulario, se comprueba que el token enviado coincide con el token almacenado en la sesión. Si no coinciden, la petición se rechaza. Esta medida de seguridad evita que un atacante envíe peticiones maliciosas desde otro sitio web. Hemos incluido este token en el formulario principal al definir la plantilla:

```
{% if form %}
<form id="consulta_form" class="form" method="post">
  {% csrf_token %}
  {{ form.enunciado }}
  {{ form.codigo }}
```

- **Inyección de código:** Django protege la aplicación contra la inyección de código con un sistema de validación de formularios. Cuando se envía un formulario, Django valida los datos del formulario para asegurarse de que no contienen código malicioso. Esto se hace con el método `cleaned_data` de los formularios. Este método limpia los datos del formulario y los convierte en un formato seguro para ser procesados por la aplicación.

### 3.3.4.3. Moderación de consultas

Los modelos de lenguaje grandes son capaces de generar texto de forma autónoma y creativa. Aunque las instrucciones de sistema proporcionadas al modelo guían la respuesta, no se puede predecir el contenido del mensaje del usuario. Por tanto, es posible que un atacante envíe consultas no relacionadas con el uso de la aplicación. Esto supone un riesgo de seguridad ya que el modelo de lenguaje es un recurso no gratuito y no queremos que se use de forma abusiva o maliciosa. Para mitigar este riesgo, hemos implementado una función de moderación previo a la realización de las consultas. Esta función hace uso del modelo de lenguaje con una petición cuya instrucción es la evaluación del contenido de los datos que ha enviado el usuario. Si el contenido no es adecuado, se rechaza la consulta y se notifica al usuario. Hemos implementado esta función en la clase manejadora de la API de forma análoga a las demás consultas con las siguientes instrucciones:

```
moderation = """\
  Your task is to determine if the coding problem provided \
  by the user is in fact a coding problem. \
  When given a user message as input (delimited by '---'), \
  respond with Y or N: \
  Y - if the user is providing the problem statement, \
  and it is a coding problem \
  N - otherwise \
  Output a single character. \
  """
```

En este mensaje se le indica al modelo que debe determinar si el problema proporcionado por el usuario es un problema de programación. Si el modelo responde con 'Y', se acepta la consulta y se procesa. Si el modelo responde con 'N', se rechaza la consulta y se notifica al usuario. Esta función de moderación se usa en la función `main` de la vista principal:

```
moderation = model_manager.moderation(prompt)
if moderation == "N":
    moderation = "Por favor ponga un problema y un código,\
    en caso de tenerlo, de programación. Mis funcionalidades\
    solo sirven para ese tipo de problemas."
    return render(request, 'index.html',
                  {'form': form,
                   'moderation': moderation
                  })
```

De esta forma evitamos gastar los recursos del modelo de lenguaje en consultas no relacionadas con el uso de la aplicación. Además, protegemos la aplicación contra posibles generaciones de texto malicioso o inapropiado aunque el modelo ya está entrenado para no generar contenido inapropiado [9].

# Capítulo 4

## Evaluación y Trabajo Futuro

### 4.1. Fiabilidad de la herramienta

Una de las características más importantes y diferentes de esta aplicación es que el resultado de las consultas está delegado en un modelo de lenguaje. Esto implica que la fiabilidad de la herramienta depende en gran medida de la calidad de los modelos y de las instrucciones que se le proporcionen. Un modelo de lenguaje más avanzado es capaz de interpretar mejor las instrucciones y realizar consultas más complejas.

Nosotros hemos utilizado modelos de lenguaje como *GPT-3.5 Turbo* o *Claude-3-Haiku*, debido a su bajo coste de entrada. Sin embargo, este modelo no ha sido entrenado específicamente para la aplicación y sus tareas, por lo que el refinamiento de las instrucciones de sistema ha sido crucial para garantizar la mejor fiabilidad posible. Nos hemos encontrado con que, en general, el modelo de lenguaje es capaz de interpretar correctamente las instrucciones y realizar consultas de forma satisfactoria. Sin embargo, durante el proceso de desarrollo hemos ido encontrando casos en los que el modelo no interpretaba correctamente las instrucciones.

Las primeras instrucciones proporcionadas al modelo fueron simples y directas, pero no conseguían buenos resultados. Por ejemplo las instrucciones para obtener casos de prueba al comienzo del desarrollo eran:

```
test_cases = intro + """\n    Provide at least 5 test cases for the code solution.\n    The test cases must include the input and the expected output.\n    The test cases must be provided between this delimiter: '---'.\n    The test cases must be provided in the following format:\n    'input: expected_output'.\n    """
```

Estas instrucciones producían resultados insatisfactorios ya que el modelo devolvía información no pedida o no relevante. Por ejemplo, en el caso de las pruebas, una de las salidas que se obtenía era la siguiente:

To solve this problem, we can use dynamic programming. We will create a function that takes the target value and the array of scores as input. The function will return the minimum number of darts needed to achieve the target value.

Here is the Python implementation:

```
def min_darts_to_reach_target(target, scores):
    dp = [float('inf')] * (target + 1)
    dp[0] = 0

    for i in range(1, target + 1):
        for score in scores:
            if i - score >= 0:
                dp[i] = min(dp[i], dp[i - score] + 1)

    return dp[target]

# Test cases
print(min_darts_to_reach_target(20, [1, 5, 10]))
# input: 20, expected_output: 2
print(min_darts_to_reach_target(25, [3, 6, 9]))
# input: 25, expected_output: 3
print(min_darts_to_reach_target(30, [2, 4, 6]))
# input: 30, expected_output: 5
print(min_darts_to_reach_target(15, [1, 2, 3, 4]))
# input: 15, expected_output: 4
print(min_darts_to_reach_target(18, [2, 5, 7]))
# input: 18, expected_output: 3
```

In this code, we initialize a list `dp` with infinity values and set `dp[0]` to 0. We then iterate through the target values and scores to calculate the minimum number of darts needed to reach each target value. Finally, we return the minimum number of darts needed to reach the given target value.

Esta respuesta, aunque completa y correcta, no es lo que se espera de la consulta. Devuelve secciones de texto y de código que no son solicitadas. Para mejorar las instrucciones, se ha optado por ser más específicos y directos. Las instrucciones para solo obtener casos de prueba han tenido que ser modificadas hasta llegar a la siguiente salida:

```
Entrada:
Puntuaciones de la diana: [10, 20, 30, 40, 50]
Valor a conseguir: 100
Salida esperada: 3

Entrada:
Puntuaciones de la diana: [5, 10, 15, 20, 25]
Valor a conseguir: 50
Salida esperada: 2

Entrada:
Puntuaciones de la diana: [3, 5, 7, 11, 13]
Valor a conseguir: 25
Salida esperada: 3

Entrada:
Puntuaciones de la diana: [1, 3, 5, 7, 9]
Valor a conseguir: 15
Salida esperada: 3

Entrada:
Puntuaciones de la diana: [2, 4, 6, 8, 10]
Valor a conseguir: 20
Salida esperada: 2
```

Con la última versión (descrita en el capítulo 3) de las instrucciones el modelo ya devuelve solo los casos de prueba sin incluir ni texto ni código listo para ser procesado por la aplicación.

El método que hemos seguido para aumentar la fiabilidad de las respuestas ha sido iterativo. Hemos ido probando diferentes instrucciones y analizando las respuestas para ver si se ajustaban a lo esperado. En caso de no ser así, se modificaban las instrucciones y se volvía a probar. Este proceso ha sido necesario para obtener resultados interpretables por la aplicación.

### 4.1.1. Rendimiento

Vamos a analizar el rendimiento de la aplicación en función de la cantidad de tiempo que tarda en responder a las consultas. Hemos notado que el tiempo de respuesta a una sola consulta es aceptablemente rápido, en torno a 1 segundo. Sin embargo, el tiempo de respuesta aumenta considerablemente al realizar 3 o más consultas en la misma petición, llegando alrededor de los 70 segundos si se piden las 5 consultas sobre un mismo problema. Con estos resultados, podemos ver que el tiempo de respuesta es aceptable para una consulta, pero es demasiado largo para varias consultas en la misma petición del problema. Esto es debido a las limitaciones de la API de OpenAI en su modo gratuito [10], que limita a 3 el número de peticiones por minuto. La API de Anthropic en su modo gratuito lo limita a 5 peticiones por minuto [1]. Para mejorar el rendimiento de la aplicación, habría que mejorar el plan de suscripción a un modelo superior al gratuito.

### 4.1.2. Comparación con otras herramientas de tipo juez

Esta herramienta se diferencia de otras herramientas de tipo juez en que proporciona ayuda a la hora de resolver problemas de programación. En lugar de limitarse a evaluar la solución propuesta por el usuario y dar un veredicto de si es correcta o no, esta herramienta proporciona información adicional que puede ayudar al usuario a resolver el problema y a aprender de él. Otras herramientas tipo juez están implementadas sin modelos de lenguaje, por lo que no pueden proporcionar información ni analizar en profundidad el código proporcionado. El desarrollo con un modelo de lenguaje grande facilita mucho la implementación de este tipo de funcionalidades. La aplicación está pensada para ser utilizada en conjunto con estas herramientas de tipo juez, para proporcionar la experiencia de aprendizaje más completa.

## 4.2. Desarrollo con modelos de lenguaje

Los modelos de lenguaje grandes son una herramienta muy potente para el desarrollo de aplicaciones. Permiten a los desarrolladores añadir funcionalidades avanzadas a sus aplicaciones sin tener que desarrollarlas desde cero. En el caso de esta aplicación, el uso de un modelo de lenguaje ha permitido añadir funcionalidades de análisis y generación de código que hubieran sido muy difíciles de implementar de otra forma. Además, el uso de un modelo de lenguaje permite a los desarrolladores centrarse en la lógica de la aplicación y delegar en el modelo tareas complejas.

El uso de modelos de lenguaje es especialmente útil en aplicaciones de tipo asistente. Estas aplicaciones son aquellas donde el usuario preferiría interactuar con la aplicación mediante lenguaje natural. En este tipo de aplicaciones los modelos de lenguaje permiten a los usuarios usar la aplicación de forma intuitiva y natural, sin tener que aprender funcionalidades específicas y complejas. Además, los modelos de lenguaje permiten a los desarrolladores designar instrucciones al modelo para guiarlo en la realización de las tareas del usuario. Por ejemplo, en esta aplicación las instrucciones proporcionadas al modelo le indican qué tipo de información debe devolver y cómo debe devolverla. En la aplicación nativa de chat de ChatGPT es probable que el modelo no devuelva la información pedida por el usuario sin que este tenga que especificar correctamente las instrucciones para su consulta, ya que en la funcionalidad de chat el modelo tiene otras instrucciones por defecto. El uso de modelos de lenguaje no solo sirve para generar un texto final para el usuario, también es posible utilizarlos para generar una salida que puede ser procesada por código. Por ejemplo, podemos dar instrucciones al modelo para que genere un objeto JSON o genere más código que pueda ser ejecutado por la aplicación.

Sin embargo, el uso de modelos de lenguaje no está exento de problemas. Uno de los problemas más importantes es la fiabilidad de las respuestas ya que no las controla el desarrollador. En el caso de esta aplicación, hemos tenido que iterar sobre las instrucciones al modelo para obtener respuestas satisfactorias. Aún así hemos encontrado casos en los que el modelo no interpretaba correctamente los problemas y devolvía respuestas no acordes a las instrucciones.

Otro problema que surge del uso de las API es el coste. Los modelos de lenguaje llevan asociados un coste computacional alto del que se encargan las empresas que los desarrollan. Es por eso que el uso intensivo de las API de los modelos de lenguaje más avanzados tienen un coste elevado, lo que puede limitar su uso en aplicaciones con un presupuesto ajustado. Además, el cargo es por uso, que se mide en tokens tanto de entrada como de salida. Esto puede limitar de alguna forma el uso masivo de las aplicaciones que utilicen estos modelos si no se puede asumir el coste.

### 4.2.1. Otros ejemplos de uso aplicaciones

Existen otros casos de uso en los que se puede utilizar modelos de lenguaje para proporcionar a los usuarios funcionalidades avanzadas. Algunos ejemplos son:

- **Análisis de sentimientos:** Un modelo de lenguaje puede ser utilizado para analizar el sentimiento de un texto y devolver una puntuación de positividad o negatividad.
- **Sintetización de contenidos:** Los modelos de lenguaje pueden hacer uso de un texto largo para resumirlo y devolver un texto más corto y conciso. También podrían ser capaces de entender código y generar documentación y comentarios.
- **Traducción de idiomas:** Las capacidades de un modelo de lenguaje pueden ser utilizadas para traducir un texto de un idioma a otro delegando en el modelo la tarea de traducción.
- **Asistentes empresariales:** Un modelo de lenguaje puede ser utilizado para aprender sobre los procesos de una empresa y proporcionar información y ayuda a los empleados.
- **Asistentes creativos:** Es posible utilizar modelos de lenguaje para generar ideas creativas y ayudar a creadores en su proceso creativo, sugiriendo ideas y completando textos.

## 4.3. Trabajo futuro

### 4.3.1. Entrenamiento específico de un modelo la aplicación

Como se ha mencionado anteriormente, el modelo de lenguaje utilizado no ha sido entrenado específicamente para la aplicación. Esto implica que el modelo no está optimizado para las consultas que se le realizan. Un modelo de lenguaje entrenado específicamente para la aplicación con los enunciados de los problemas y formatos de código sería capaz de devolver resultados más precisos y relevantes.

Para entrenar un modelo de lenguaje específico para la aplicación se necesitaría una gran cantidad de datos. Estos datos podrían ser extraídos de plataformas de problemas de programación, como LeetCode o HackerRank. Se podrían extraer los enunciados de los problemas y los casos de prueba, y utilizarlos para entrenar el modelo. También sería posible entrenar al modelo con enunciados de problemas de las asignaturas de algoritmia del grado en Informática.

### **4.3.2. Función de chat**

Una funcionalidad interesante que se podría añadir a la aplicación es una función de chat. Esta función permitiría a los usuarios interactuar con el modelo de lenguaje haciendo preguntas sobre las consultas previas que han realizado. Por ejemplo, un usuario podría preguntar al modelo por la explicación de un caso de prueba específico, o por una aclaración sobre una pista. La implementación de esta funcionalidad tendría que tener en cuenta las respuestas ya dadas e implicaría un uso más intensivo del modelo de lenguaje.

### **4.3.3. Funcionalidad para profesores**

Se podría extender el uso de la aplicación de forma que sirva como una plataforma para profesores y alumnos. Los profesores podrían subir problemas de programación y los alumnos podrían resolverlos utilizando la aplicación. Los profesores podrían delegar las dudas habituales a la aplicación, y los alumnos podrían recibir ayuda inmediata. Esta aplicación podría ser utilizada en conjunto con las aplicaciones ya usadas tipo juez, para proporcionar una experiencia de aprendizaje más completa.

# Conclusiones

En este trabajo hemos explorado el potencial de los modelos de lenguaje grandes para la creación de aplicaciones web. Hemos desarrollado una aplicación web que utiliza un modelo de lenguaje para responder a consultas de los usuarios. La aplicación permite a los usuarios realizar consultas sobre distintos aspectos de la programación y obtener respuestas claras y concisas.

El proceso de desarrollo de la aplicación ha sido un reto interesante. Hemos tenido que enfrentarnos a problemas técnicos y de diseño, y hemos tenido que tomar decisiones importantes en cuanto a la arquitectura y la implementación de la aplicación. Hemos tenido que aprender a utilizar las API de diferentes modelos y trabajar con modelos de lenguaje grandes.

El uso de modelos de lenguaje grandes en el desarrollo de aplicaciones es una tendencia en alza. Estos modelos ofrecen una gran capacidad para entender y generar texto, y pueden ser utilizados en una amplia variedad de aplicaciones, brillando especialmente en aquellas que proporcionan asistencia a los usuarios. Es una tecnología en constante evolución y mejora, y es probable que veamos nuevas formas de utilizar estos modelos para el desarrollo de aplicaciones en el futuro.



# Conclusions

In this paper we have explored the potential of large language models for creating web applications. We have developed a web application that uses a language model to respond to user queries. The application allows users to ask queries about different aspects of programming problems and get clear and concise answers.

The process of developing the application has been an interesting challenge. We have had to deal with technical and design issues, and we have had to make important important decisions regarding the architecture and implementation of the application. We had to learn how to use the API from different models and how to work with large language models.

The use of large language models in application development is a growing trend. These models offer a great ability to understand and generate text, and can be used in a wide variety of applications, especially shining in those that provide assistance to users. It is a technology that is constantly evolving and improving, and it is likely that we will see new ways of using these models for software development in the future.



# Aportaciones Personales

Al comienzo de este proyecto, ambos nos hemos centrado en estudiar y comprender el funcionamiento de los modelos de lenguajes grandes y sus aplicaciones en el ámbito del desarrollo de software. Los dos miembros del equipo realizamos cursos relacionados con la API de ChatGPT y la creación de aplicaciones. También hemos investigado sobre los distintos modelos de lenguaje disponibles y sus características como alternativas a la API de ChatGPT.

En la fase de diseño, también hemos investigado sobre diferentes marcos de trabajo sobre los que desarrollar la aplicación. Jose Javier ha investigado sobre el uso de Flask y Django, proponiendo finalmente el desarrollo de una aplicación web usando Django ya que Jose Javier tenía una breve experiencia previa. Una vez acordada esta decisión, Mario ha investigado sobre su uso y funcionamiento. Ambos decidimos qué funcionalidades implementar y cómo hacerlo. Finalmente, ambos nos decidimos por usar primero la API de ChatGPT para la implementación de la aplicación.

En la fase de desarrollo, hemos trabajado conjuntamente en la implementación de la aplicación. Como la aplicación está dividida en dos partes, Jose Javier primero implementó el primer módulo de las API. Mario, por su parte, refinó y completó el resto del módulo. Una vez que este estuvo completo, Jose Javier trabajó en los estilos, la interfaz de usuario y las funcionalidades generales de la aplicación, como el historial y la autenticación, mientras que Mario se encargó del proceso iterativo de pruebas y correcciones del modelo. Durante el proceso de desarrollo, decidimos añadir soporte para que la aplicación funcionase con varios modelos de lenguaje, por lo que ambos investigamos sobre las APIs de otros modelos y cómo integrarlos en la aplicación. Jose Javier se encargó de la integración de la API de Anthropic, mientras que Mario se encargó de seguir el proceso iterativo de pruebas y correcciones.

Pese a estas divisiones del trabajo en paralelo, la mayoría del proyecto se ha realizado en conjunto, como la toma de decisiones del diseño de la interfaz o de las funcionalidades a implementar.



# Bibliografía

- [1] Anthropic. Rate limits. <https://docs.anthropic.com/en/api/rate-limits>.
- [2] Google Cloud. OAUTH 2.0 Introduction. <https://cloud.google.com/apigee/docs/api-platform/security/oauth/oauth-introduction?hl=es-419>.
- [3] Ed. D. Hardt. The OAuth 2.0 Authorization Framework, 2012. <https://datatracker.ietf.org/doc/html/rfc6749>.
- [4] Mark Chen et al. Evaluating Large Language Models Trained on Code. 2021.
- [5] Django Software Foundation. Django documentation. <https://docs.djangoproject.com/en/5.0/>.
- [6] OpenAI. Chat completions API. <https://platform.openai.com/docs/guides/text-generation/chat-completions-api>.
- [7] OpenAI. Chat Completions response format. <https://platform.openai.com/docs/guides/text-generation/chat-completions-response-format>.
- [8] OpenAI. Prompt engineering. <https://platform.openai.com/docs/guides/prompt-engineering>.
- [9] OpenAI. Usage Policies. <https://openai.com/policies/usage-policies/>.
- [10] OpenAI. Usage tiers. <https://platform.openai.com/docs/guides/rate-limits/usage-tiers>.
- [11] David Rein, Betty Li Hou, Asa Cooper Stickland, Jackson Petty, Richard Yuanzhe Pang, Julien Dirani, Julian Michael, and Samuel R. Bowman. GPQA: A Graduate-Level Google-Proof Q&A Benchmark, 2023.
- [12] Jose Javier Carrasco y Mario Alcolea Alcolea. Repositorio de GitHub de Algoritmia. <https://github.com/WhiteMoon-Knight/TFGPT/tree/main/AppWeb/TFG>.

