Instituto Tecnológico de Costa Rica

Campus Local San Carlos

Escuela de Ingeniería en Computación

Segundo Proyecto

Inteligencia Artificial

Grupo 50

Estudiantes:

Sebastian Matey

Liz Salazar

Roosevelt Pérez

Profesor:

Efrén Jimenez Delgado

II Semestre 2025

Resumen

El presente documento detalla el desarrollo del proyecto Mariantonieta-IA, elaborado como parte del curso de Inteligencia Artificial de la carrera de Ingeniería en Computación del Instituto Tecnológico de Costa Rica. Este proyecto tiene como objetivo diseñar e implementar un asistente virtual inteligente que combine diferentes ramas de la inteligencia artificial, como el aprendizaje automático, el reconocimiento de voz, la visión por computadora y el procesamiento del lenguaje natural.

A lo largo del documento se presenta la solución planteada, basada en una arquitectura modular de microservicios que permite integrar diversos modelos de machine learning especializados, cada uno orientado a un dominio distinto. También se describe la arquitectura general de la plataforma, el funcionamiento del coordinador LLM (Llama 3 con Ollama) y los procesos de comunicación entre los servicios.

El informe aborda además las arquitecturas de los modelos de aprendizaje automático utilizados, explicando su diseño, justificación y entrenamiento. Posteriormente, se exponen los resultados obtenidos, junto con un análisis de su desempeño y las posibles mejoras futuras.

Finalmente, este proyecto propone una forma innovadora de aplicar la inteligencia artificial como coordinadora de un backend inteligente, capaz de orquestar múltiples modelos y servicios autónomos bajo una misma lógica conversacional. Este enfoque puede ser aprovechado en diversos entornos, como sistemas de atención al cliente, plataformas educativas, asistentes empresariales o herramientas de apoyo médico, demostrando el potencial de los LLM y la arquitectura de microservicios para construir soluciones escalables y adaptables a distintos contextos.

1. Solución Planteada

El proyecto **Mariantonieta-IA** propone el desarrollo de un asistente virtual inteligente y multimodal capaz de procesar información proveniente del texto, la voz y la imagen, utilizando únicamente componentes desarrollados dentro del propio sistema. Su objetivo es ofrecer una experiencia conversacional natural, en la que el usuario pueda realizar preguntas o solicitudes y recibir respuestas generadas mediante modelos de Machine Learning distribuidos.

La solución se construyó bajo una **arquitectura de microservicios**, donde cada módulo representa un dominio de conocimiento específico, como predicción de precios de Bitcoin, retrasos de vuelos, riesgo de ACV, análisis agrícolas o recomendaciones de películas. Cada microservicio funciona como una API independiente que ejecuta su propio modelo especializado y devuelve resultados precisos y estructurados.

En el centro de la plataforma se encuentra un **backend principal desarrollado con FastAPI**, que actúa como concentrador de comunicaciones. Este backend recibe las consultas del usuario y coordina el flujo interno con los microservicios. La toma de decisiones está a cargo de un **modelo de lenguaje de gran tamaño (LLM)**, específicamente **Llama 3 ejecutado localmente mediante Ollama**, que interpreta las preguntas en lenguaje natural, determina qué microservicio debe atenderlas y sintetiza una respuesta conversacional basada en los resultados obtenidos.

El sistema también incorpora capacidades multimodales sin depender de servicios externos. El módulo de **reconocimiento de voz (Speech-to-Text)** y el módulo de **análisis de imágenes** se ejecutan de forma local utilizando modelos optimizados y contenedorizados dentro de la arquitectura de microservicios. Esto permite al asistente procesar audio e imágenes de manera privada, eficiente y completamente autónoma.

Finalmente, el proyecto incluye mecanismos de registro y monitoreo de eventos (logging rotativo), scripts para ejecutar y desplegar los servicios, y una estructura organizada que facilita la extensión futura del sistema. Gracias a su modularidad, Mariantonieta-IA puede integrar nuevos modelos o dominios sin afectar la arquitectura existente, consolidándose como una plataforma flexible y escalable para experimentación con inteligencia artificial aplicada.

1.1 Arquitectura general de la plataforma

La arquitectura de Mariantonieta-IA está basada en el paradigma de microservicios, lo que permite distribuir las distintas funcionalidades del sistema en módulos independientes, fácilmente escalables y mantenibles. Cada microservicio encapsula un dominio de conocimiento específico, su modelo de machine learning y la lógica necesaria para recibir peticiones, procesar datos y devolver resultados a través de una API REST.

El núcleo del sistema es un servidor central desarrollado en FastAPI (Python), el cual actúa como punto de entrada para todas las solicitudes del usuario. Este servidor coordina los diferentes microservicios y expone los endpoints principales, entre ellos /ask, encargado de la comunicación con el modelo de lenguaje de gran tamaño (LLM) que dirige el flujo de interacción.

El LLM, (implementado con Llama 3, como se mencionó con anterioridad) y ejecutado localmente mediante Ollama, cumple el rol de coordinador semántico: interpreta el mensaje del usuario, extrae parámetros relevantes mediante prompt engineering y decide qué microservicio debe atender la consulta. Una vez obtiene la respuesta estructurada, la transforma en una salida conversacional coherente en lenguaje natural.

Cada microservicio se encuentra implementado en su propio módulo dentro del directorio api/routes/ y cuenta con un modelo entrenado y serializado en la carpeta ml_models/. Estos servicios cubren áreas como: Predicción de precios de Bitcoin (modelo

Prophet). Estimación de retrasos de vuelos (Random Forest). Evaluación de riesgo de accidente cerebrovascular (Árbol de Decisión). Valoración inmobiliaria (Random Forest Regressor). Predicción de precios de aguacate (CatBoost). Recomendación de películas (modelo híbrido KNN + similitud de géneros).

Además, la plataforma integra módulos complementarios de **reconocimiento facial y voz**, ejecutados completamente de forma local dentro de la arquitectura del sistema.

El módulo de **análisis facial** utiliza un modelo propio de visión por computadora para detectar rostros, estimar emociones y procesar características visuales sin depender de servicios externos. Este componente implementa sus propios cálculos de coincidencia y métricas como IoU para evaluar la calidad de las detecciones.

Por su parte, el componente de **Speech-to-Text (STT)** también se ejecuta localmente mediante un modelo optimizado para transcripción de audio, permitiendo que el asistente procese comandos y preguntas por voz manteniendo la privacidad y reduciendo la latencia.

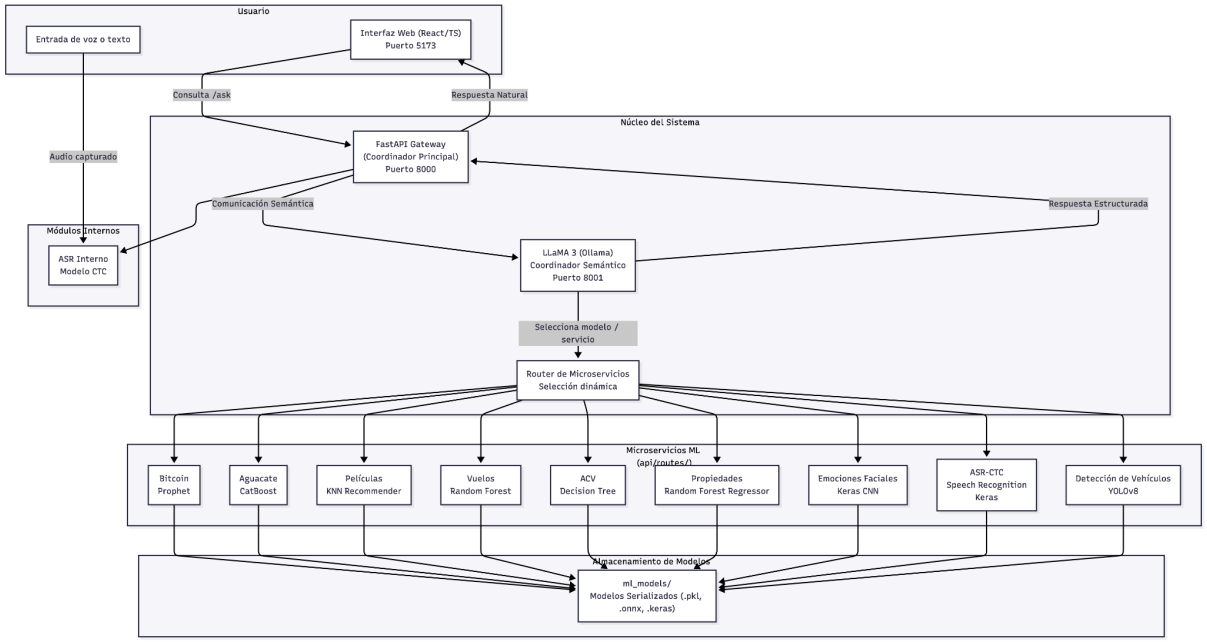

Fig 1: Arquitectura de la aplicación (elaboración propia)

1.1.1 Coordinador LLM (Ollama + Llama 3)

El coordinador LLM es el componente central encargado de interpretar las solicitudes del usuario y determinar qué microservicio del sistema debe responder. Este módulo utiliza un modelo de lenguaje de gran tamaño (LLM), específicamente Llama 3, ejecutado localmente mediante la plataforma Ollama, lo que permite operar sin depender de servicios externos en la nube y mantener un mayor control sobre el procesamiento de datos. El coordinador cumple el rol de núcleo cognitivo del sistema, actuando como intermediario entre el usuario y los distintos servicios de aprendizaje automático. Cuando se recibe una consulta en lenguaje natural, el LLM analiza el mensaje, identifica el dominio del problema (por ejemplo, predicciones de Bitcoin, riesgo de ACV o recomendación de películas) y extrae los parámetros necesarios para la llamada al microservicio correspondiente. El flujo de trabajo típico del coordinador se puede resumir en los siguientes pasos:

1. Recepción de la solicitud: El usuario envía una pregunta o comando a través del endpoint /ask del backend central.
2. Interpretación del contexto: El LLM analiza la intención del usuario mediante técnicas de prompt engineering y reconocimiento semántico.
3. Extracción de parámetros: Utilizando plantillas de extracción definidas en el módulo llm/extract_params.py, el sistema determina qué datos o argumentos requiere el modelo especializado (por ejemplo, fechas, variables de entrada o nombres de entidad).
4. Selección del microservicio: Con base en el dominio identificado, el LLM consulta la lista de modelos disponibles (llm/available_models.py) y selecciona el endpoint correspondiente.
5. Invocación del servicio: Se realiza una llamada REST al microservicio elegido, el cual procesa la información y devuelve un resultado estructurado en formato JSON.
6. Generación de la respuesta: Finalmente, el LLM interpreta la salida del modelo y genera una respuesta en lenguaje natural que es enviada al usuario.

Este enfoque permite una coordinación dinámica y flexible entre los diferentes modelos de la plataforma, logrando que el asistente actúe como un sistema unificado y coherente, a pesar de estar compuesto por múltiples servicios independientes. El uso de Ollama ofrece además ventajas en rendimiento y privacidad, ya que el modelo Llama 3 puede ejecutarse localmente sin necesidad de conexión externa, garantizando tiempos de respuesta estables y evitando la exposición de información sensible. Gracias a esta capa de coordinación, Mariantonieta-IA logra integrar diversos tipos de inteligencia artificial bajo un solo flujo conversacional, optimizando la interacción con el usuario de forma natural y contextual.

1.2 Frontend

El frontend del proyecto fue implementado utilizando React con TypeScript, aprovechando las ventajas de un entorno de desarrollo rápido y eficiente proporcionado por Vite. La aplicación utiliza TailwindCSS para el diseño de la interfaz, facilitando la creación de una interfaz responsiva y de alto rendimiento con clases de utilidad. Se integra Axios para realizar solicitudes HTTP a las APIs del backend.

Para la estructura del proyecto, se emplean herramientas de calidad de código como ESLint, lo que garantiza la coherencia del código y minimiza errores. Además, se utilizaron bibliotecas como Lucide-React y React-Icons para la inclusión de iconos, mientras que Class-Variance-Authority y clsx permiten gestionar clases dinámicas de manera eficiente en componentes de React.

Los scripts de desarrollo están configurados para compilar el código mediante TypeScript y Vite, proporcionando una experiencia de recarga rápida durante el desarrollo. Esto permite a los desarrolladores hacer cambios inmediatos y probarlos sin necesidad de reiniciar el servidor, optimizando el tiempo de desarrollo.

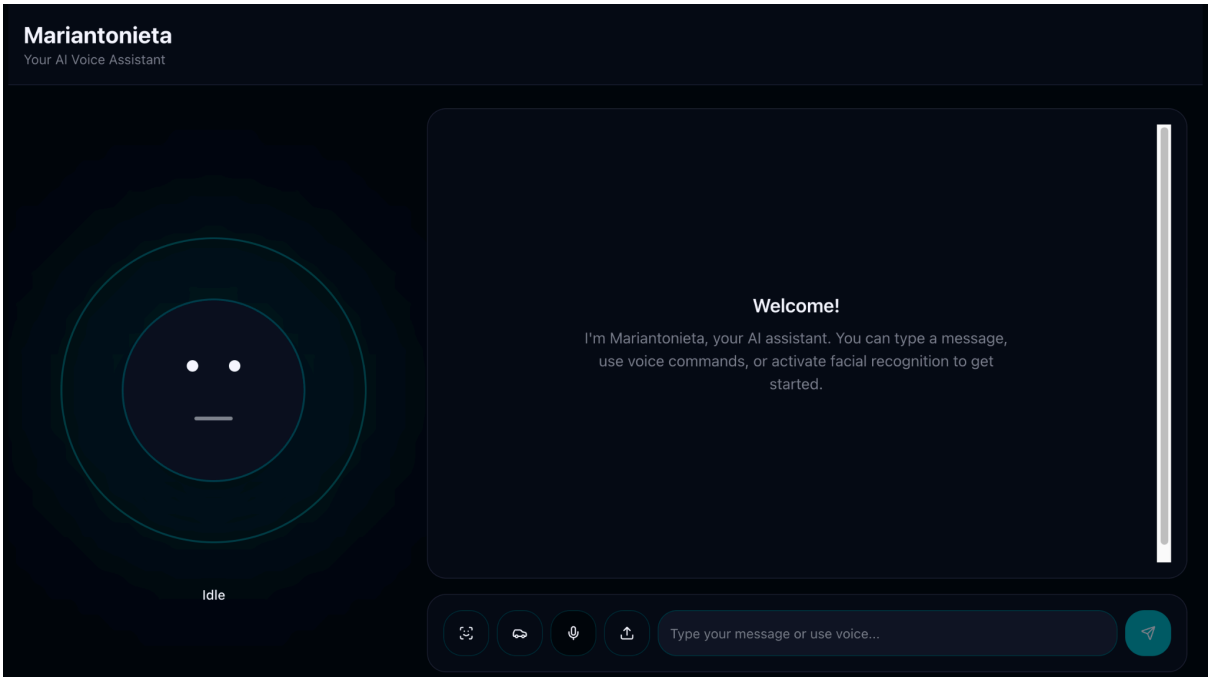

Fig 2: Interfaz de usuario al iniciar

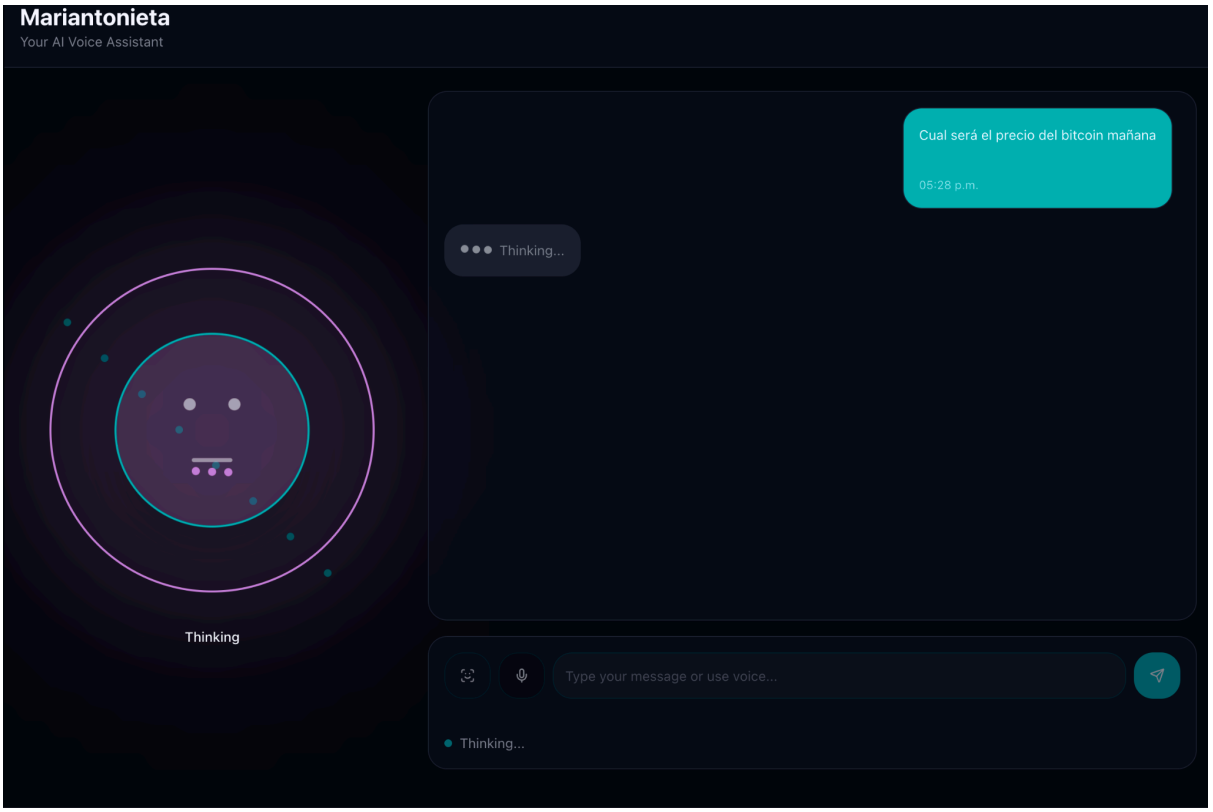

Fig 3: Interfaz de usuario con el modelo procesando una consulta

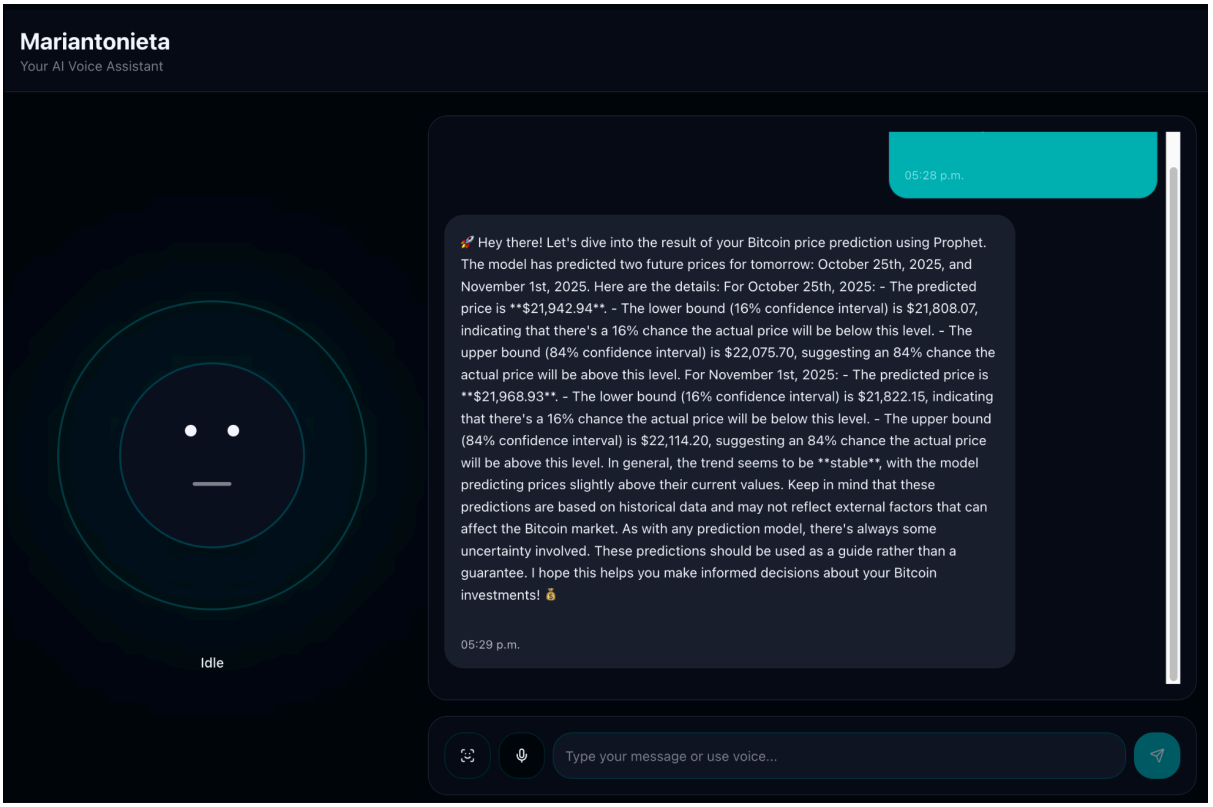

Fig 4: Interfaz de usuario con una respuesta generada

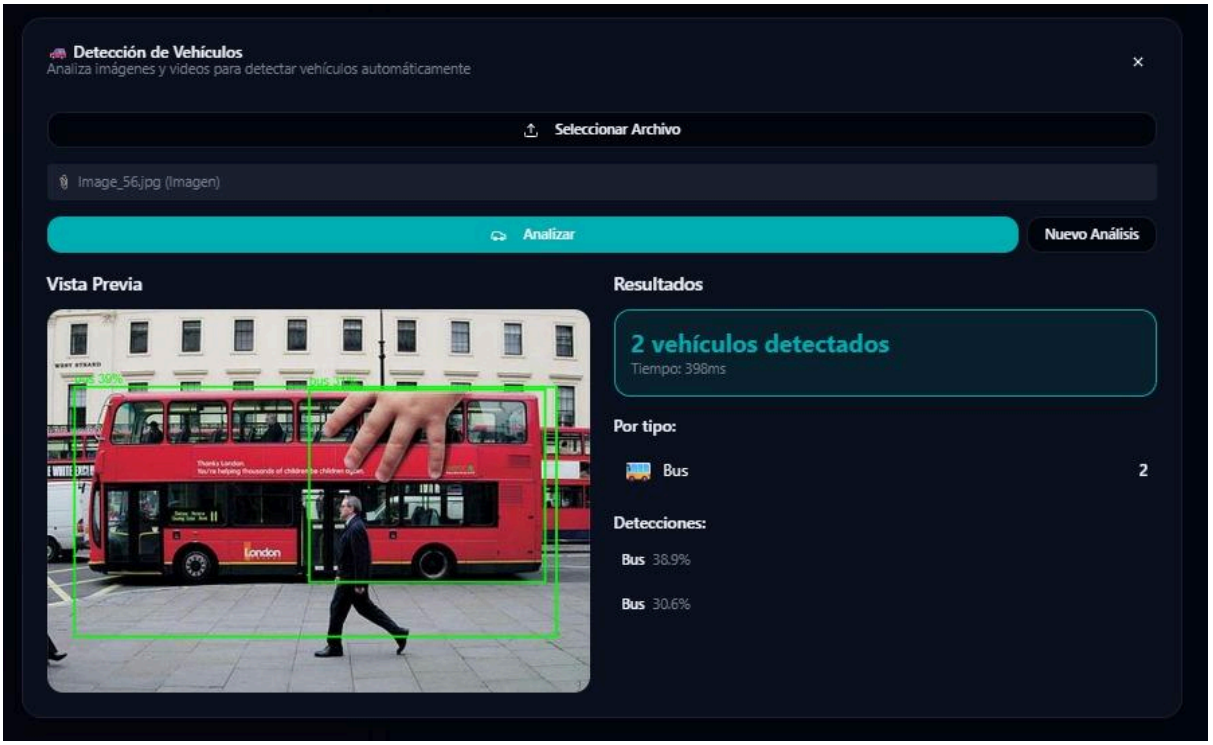

Fig 5: Interfaz para la detección de vehículos

1.3 Integración reconocimiento facial

La integración del reconocimiento facial se implementó como un módulo completamente interno dentro de la aplicación principal construida con FastAPI. A diferencia de la versión inicial, esta nueva arquitectura **elimina la dependencia de servicios externos** (como Google Cloud Vision) y utiliza exclusivamente un **modelo Keras local** de reconocimiento de emociones junto con un **detector de rostros basado en ONNX YuNet u Haar Cascade**, lo que permite un procesamiento más rápido, seguro y sin conexión a Internet.

El archivo **main.py** continúa actuando como el orquestador central del sistema. Durante el arranque, este módulo verifica la disponibilidad del modelo local almacenado en **dl_models/emotion_model.keras** y los archivos del detector de rostros. Si ambos están disponibles, activa una bandera interna (**FACE_AVAILABLE**) que habilita dinámicamente la inclusión del router dedicado a la funcionalidad facial. En caso de que el modelo o el detector no estén disponibles, el sistema omite esta funcionalidad sin afectar el resto de la API, manteniendo el diseño modular y tolerante a fallos.

El archivo **face_routes.py** define ahora los endpoints de análisis facial basados en este nuevo pipeline. El endpoint principal **/face/analyze** acepta archivos de imagen en formato **.jpg, .jpeg** o **.png,** valida el tipo y luego convierte el archivo recibido en un flujo de bytes (**BytesIO**). Este flujo se envía al servicio interno de emociones mediante la función **detect_faces_with_keras**, que encapsula todo el proceso de detección, recorte y clasificación de emociones mediante el modelo local. La ruta está diseñada para manejar errores comunes como imágenes inválidas, problemas de lectura o fallos internos del modelo, devolviendo códigos HTTP adecuados altamente descriptivos.

El servicio devuelve una lista de diccionarios con la posición del rostro y su emoción correspondiente, lo que permite enriquecer el frontend o generar análisis adicionales. Además, el módulo incorpora optimizaciones como **carga diferida**, **caché de predicciones**

y **procesamiento batch**, mejorando el rendimiento en tiempo real sin recurrir a recursos externos. Finalmente, la API guarda automáticamente cada imagen procesada junto con la emoción detectada, facilitando auditorías, datasets internos y trazabilidad del sistema.

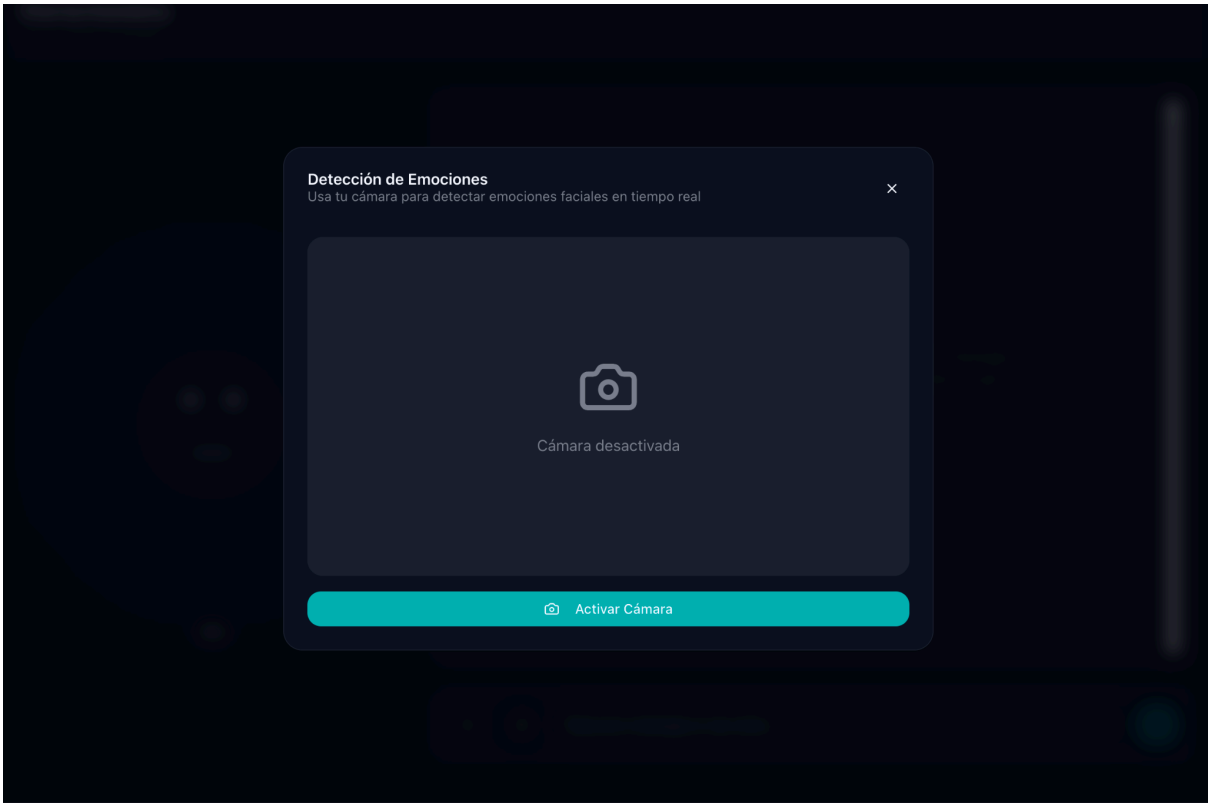

Fig 6: Inicialización del módulo de reconocimiento de emociones

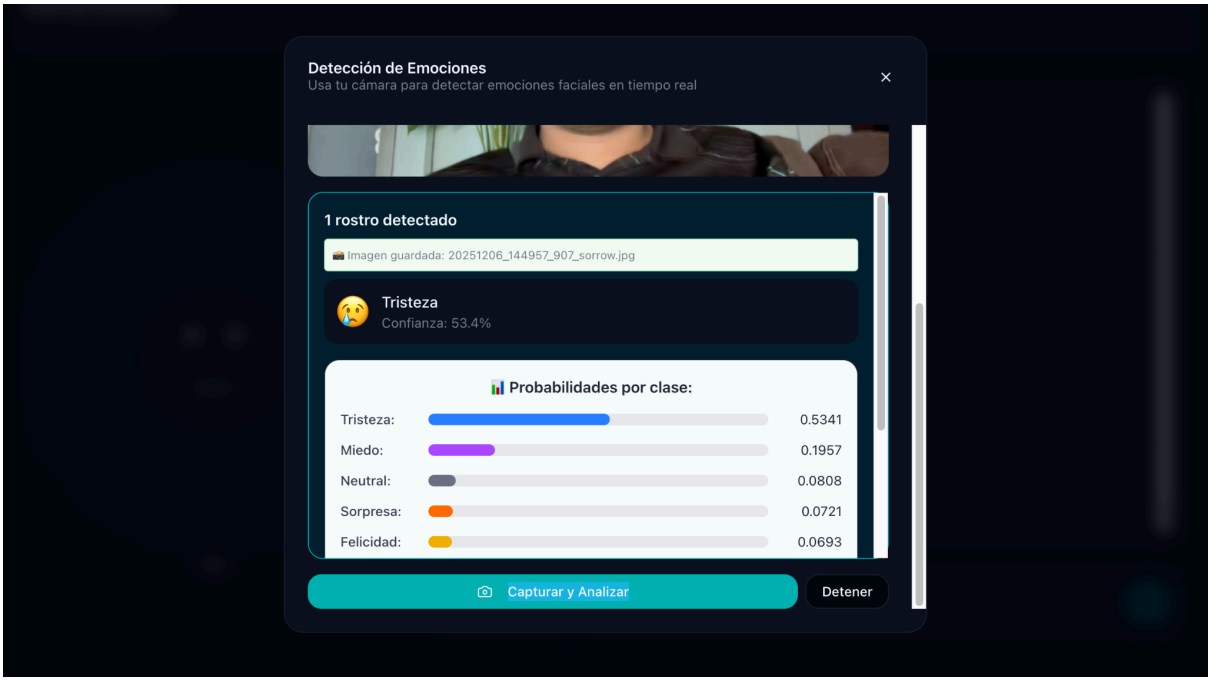

Fig 7: Módulo de reconocimiento de emociones en ejecución

1.4 Integración del reconocimiento de voz

La integración del reconocimiento de voz a texto (STT) en el proyecto se implementó como un módulo dentro de la aplicación principal FastAPI, utilizando un modelo de deep learning tipo DeepSpeech2 entrenado específicamente para este propósito y almacenado como mariantonieta_asr_ctc.h5. El archivo principal main.py actúa como orquestador, intentando cargar el router de Speech-to-Text al iniciar la aplicación. Si el módulo dl_models.asr está disponible y correctamente configurado, main.py incluye el router de STT en la API mediante app.include_router(stt_router), permitiendo gestionar las peticiones de transcripción de audio. De esta forma, la funcionalidad de reconocimiento de voz se activa solo cuando las dependencias del modelo están presentes, garantizando que la aplicación siga funcionando incluso si el módulo de STT no está disponible en el entorno.

El archivo stt.py contiene la lógica específica para manejar el proceso de transcripción de voz a texto. Define un endpoint POST en /stt/ que recibe archivos de audio en formato WAV, valida el tipo de contenido para asegurar que sea un archivo de audio

válido, y lo guarda temporalmente en el directorio uploads. Una vez almacenado, el archivo se procesa mediante el módulo asr_model.py, que carga el modelo pre-entrenado mariantonieta_asr_ctc.h5 y se encarga de preprocesar el audio (extracción de espectrograma mel, normalización), alimentarlo a través de una red neuronal tipo DeepSpeech2 (compuesta por capas convolucionales, recurrentes bidireccionales y entrenada con CTC loss) y decodificar la salida del modelo en texto legible utilizando el vocabulario de caracteres definido en vocab_characters.json. En caso de que el archivo no sea válido o se presenten problemas durante el procesamiento, el sistema maneja adecuadamente las excepciones mediante HTTPException, devolviendo mensajes de error claros al usuario con códigos de estado HTTP apropiados.

El flujo completo de la integración comienza cuando el usuario graba audio usando el micrófono o selecciona un archivo WAV propio desde el frontend. El archivo es enviado al endpoint /stt/ de la API, donde es validado y procesado por el modelo mariantonieta_asr_ctc.h5. El modelo genera una transcripción que es devuelta en formato JSON, incluyendo el nombre del archivo original y el texto transcrito. La interfaz del frontend incluye dos opciones para capturar audio: un botón de micrófono que graba hasta 5 segundos de audio y lo convierte automáticamente a formato WAV mono PCM16 a 16kHz antes de enviarlo, y un botón de carga que permite seleccionar archivos de audio locales. Esta integración permite que el proyecto convierta audio en texto de manera eficiente utilizando un modelo entrenado internamente. El modelo trabaja bien en base a audios dentro de su mismo dataset sin embargo al utilizar audios grabados por uno propio el modelo decae gravemente, esto se debe a varias limitaciones claras: el modelo fue entrenado solo con una locutora, en condiciones de estudio y con un tipo de inglés muy específico, mientras que mi voz tiene otro acento, otro micrófono y más ruido de fondo. Es decir, el modelo funciona bien dentro del dominio de LJSpeech, pero generaliza mal a condiciones reales distintas.

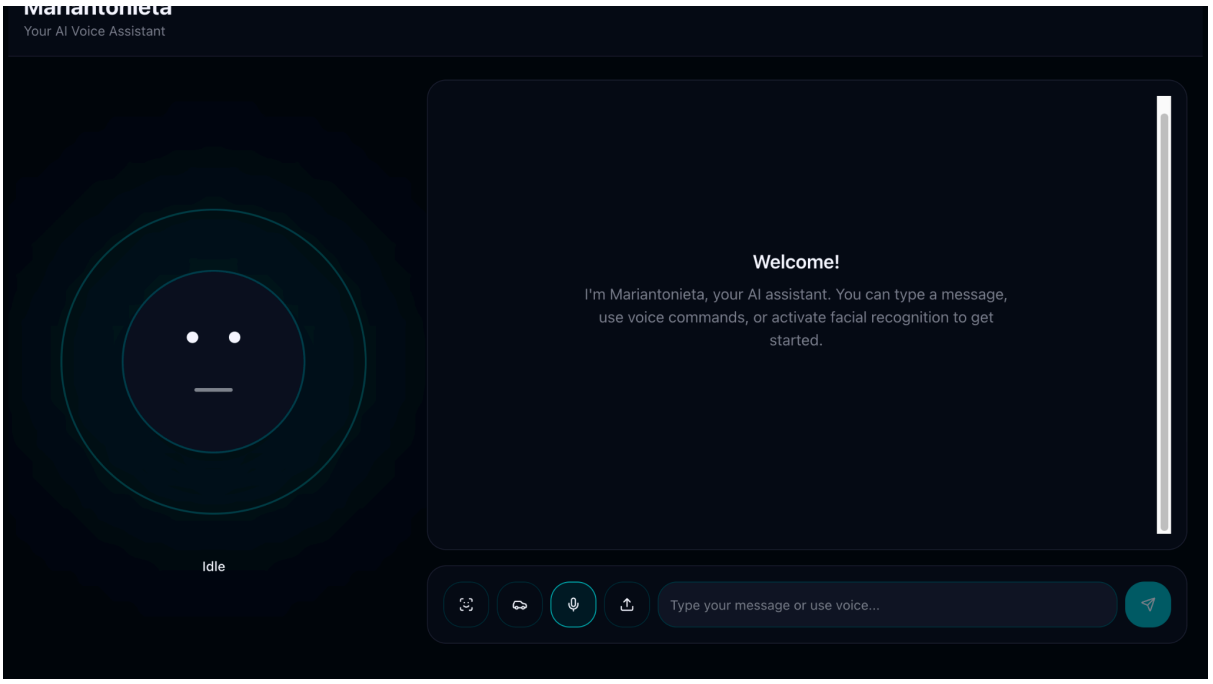

Fig 8: Botón para dictar al sistema

1.5 Integración de la detección de vehículos

La integración del sistema de detección de vehículos en Mariantonieta representa una expansión natural de las capacidades de visión artificial de la aplicación. El núcleo del sistema utiliza un modelo YOLOv8 previamente entrenado para reconocer cuatro tipos de vehículos, que son automóviles, camiones, autobuses y motocicletas. Este modelo, se guardó en un archivo especializado que ahora reside en la carpeta de modelos de la aplicación, desde donde se carga una única vez al iniciar el servicio para optimizar el rendimiento.

La arquitectura del backend sigue los mismos patrones modulares establecidos en otras funcionalidades. Se creó una carpeta dedicada para el servicio de detección de vehículos, que actúa como intermediario entre el modelo de IA y las solicitudes del frontend. La API REST ofrece dos capacidades principales, que son el análisis de imágenes individuales, que retorna la ubicación exacta de cada vehículo con cajas delimitadoras y

niveles de confianza, y procesamiento de videos mediante un muestreo inteligente de fotogramas que permite al usuario ajustar el balance entre precisión y velocidad de análisis.

El frontend fue diseñado para mantener la coherencia visual con el resto de la aplicación. Se añadió un botón con icono de automóvil junto al de reconocimiento facial, que abre una ventana modal dividida en dos secciones, vista previa del archivo y resultados del análisis. La interfaz maneja tanto imágenes como videos, mostrando para imágenes un lienzo interactivo con rectángulos de detección dibujados sobre los vehículos identificados, y para videos, estadísticas completas como el promedio de vehículos por fotograma y conteos agregados por tipo. Los resultados se presentan con emojis representativos y desgloses detallados que facilitan la comprensión inmediata.

La integración se completó actualizando el sistema de monitoreo de salud de la aplicación para incluir verificaciones del servicio de vehículos, y enriqueciendo la documentación automática con información sobre los nuevos endpoints. Todo el flujo de comunicación utiliza los mismos patrones HTTP establecidos, con manejo consistente de errores y respuestas en formato JSON, demostrando la extensibilidad de la arquitectura y facilitando futuras incorporaciones de nuevos módulos de inteligencia artificial.

2. Arquitecturas

La plataforma Mariantonieta-IA integra distintos modelos de machine learning y deep learning que fueron desarrollados y entrenados de manera independiente, cada uno orientado a resolver un tipo de problema particular dentro de su propio dominio. Estos modelos representan el núcleo predictivo del sistema y permiten que el asistente ofrezca respuestas inteligentes basadas en datos reales.

Cada modelo fue diseñado siguiendo un flujo de trabajo común que incluye la preparación de los datos, el análisis exploratorio (EDA), la selección de características relevantes, el entrenamiento y validación, y finalmente la serialización del modelo en

formato pkl para su uso en producción. Este proceso garantiza que los modelos puedan integrarse fácilmente dentro del backend, manteniendo consistencia y portabilidad entre los distintos microservicios.

Las arquitecturas implementadas abarcan tanto algoritmos de regresión como de clasificación y recomendación, utilizando técnicas tradicionales de aprendizaje supervisado como árboles de decisión, bosques aleatorios (Random Forest), modelos de boosting (CatBoost) y modelos de series de tiempo (Prophet). Cada uno fue seleccionado considerando la naturaleza de los datos, la complejidad del problema y la necesidad de interpretabilidad en el contexto de uso.

2.1 Modelos desarrollados por dominio

2.1.1 Predicción de precios de Bitcoin (Prophet)

La arquitectura implementa un sistema de pronóstico del precio de cierre de Bitcoin, diseñado como un problema de predicción de series temporales utilizando el modelo Prophet, desarrollado por Meta. Este enfoque permite capturar tendencias no lineales, estacionalidades y variaciones abruptas, características propias del comportamiento de las criptomonedas.

El modelo se entrena a partir del conjunto de datos histórico bitcoin_price_Training.csv, que contiene información financiera diaria, incluyendo la fecha y valores de apertura, cierre y volumen. En la etapa de preprocesamiento, los registros se ordenan cronológicamente, se normalizan las variables numéricas y se verifica la ausencia de valores faltantes, confirmando la completitud de la serie temporal. El análisis exploratorio inicial evidencia la alta volatilidad del precio de Bitcoin, lo cual justifica el uso de un modelo flexible y capaz de adaptarse a fluctuaciones dinámicas.

Posteriormente, se indexa la serie por fecha y se establece una división 80/20 entre los subconjuntos de entrenamiento y prueba como referencia. Para cumplir con la estructura

requerida por Prophet, las columnas se renombran a ds (fecha) y y (valor objetivo). Durante la fase de entrenamiento, se activa el parámetro daily_seasonality=True con el fin de capturar patrones diarios de comportamiento del mercado.

Una vez entrenado, el modelo genera un horizonte de predicción de 180 días, del cual se obtienen las estimaciones (yhat) correspondientes a los precios futuros. Estas predicciones se evalúan utilizando métricas de error estándar.

Además, se generan visualizaciones que muestran la trayectoria estimada del precio, junto con los componentes de tendencia y estacionalidad, lo que permite interpretar mejor la dinámica del modelo.

Finalmente, el modelo entrenado se serializa mediante joblib y se guarda en un archivo .pkl. Este artefacto incluye tanto el modelo ajustado como la información de configuración y las métricas de referencia, quedando listo para su integración en el microservicio correspondiente del backend o para futuras iteraciones de mejora y comparación de desempeño.

2.1.2 Predicción de retrasos de vuelos (Random Forest)

La arquitectura implementa un sistema de predicción de retrasos en vuelos, planteado como un problema de regresión supervisada y desarrollado utilizando el algoritmo Random Forest. El modelo se entrena a partir del conjunto de datos histórico *DelayedFlights.csv*, el cual contiene información operativa y temporal de vuelos realizados en los Estados Unidos.

El proceso inicia con la preparación y limpieza del dataset. Se eliminan los registros correspondientes a vuelos cancelados o desviados y se depuran los valores nulos presentes en la variable objetivo *ArrDelay* (retraso en la llegada). Los valores faltantes en otras columnas se reemplazan por cero, obteniendo una base de datos completa y consistente para el modelado.

Posteriormente, se selecciona un conjunto de atributos relevantes que representan distintos aspectos del comportamiento de los vuelos:

- Variables de calendario: *Month*, *DayofMonth* y *DayOfWeek*.
- Horarios planificados y reales: *DepTime*, *CRSDepTime* y *CRSArrTime*.
- Identificadores categóricos: *UniqueCarrier*, *Origin* y *Dest*.
- Factores operativos: *Distance* y *DepDelay*.

Las variables categóricas se transforman mediante Label Encoding para convertirlas en índices numéricos compatibles con modelos basados en árboles. Con la matriz de características procesada, se realiza una división 80/20 entre los conjuntos de entrenamiento y prueba, utilizando una semilla aleatoria fija (42) para garantizar reproducibilidad.

El modelo final se implementa con un RandomForestRegressor configurado con 200 árboles, una profundidad máxima de 20 niveles y ejecución paralela (n_jobs=-1) para optimizar el rendimiento. Tras el entrenamiento, se generan predicciones sobre el conjunto de prueba y se calculan métricas de evaluación que permiten medir la precisión del sistema:

- MAE (Mean Absolute Error) para cuantificar el error promedio en minutos.
- RMSE (Root Mean Squared Error) para penalizar desviaciones grandes.
- R^2 (Coeficiente de Determinación) para estimar la proporción de varianza explicada por el modelo.

Además del rendimiento, se analiza la importancia de las variables dentro del bosque aleatorio, observándose que los factores más influyentes son el retraso en la salida y los horarios programados de despegue y llegada, lo que coincide con el comportamiento real de los sistemas de transporte aéreo.

Finalmente, el modelo entrenado se serializa mediante la librería joblib, almacenando no solo el artefacto del modelo, sino también las columnas empleadas, los codificadores utilizados y las métricas de referencia.

2.1.3 Predicción de ACV (Árbol de Decisión)

La arquitectura implementada en el notebook acv_notebook.ipynb aborda la predicción de accidentes cerebrovasculares (ACV) como un problema de clasificación binaria. El proceso inicia con la carga y limpieza de los datos, seguida de un análisis exploratorio de datos que revela características clave, como la distribución de las variables de edad, presión arterial, y el índice de masa corporal (IMC), entre otras. En este análisis, se observa que ciertas variables como la hipertensión y la diabetes presentan una fuerte correlación con el riesgo de sufrir un ACV, lo que se confirma mediante la matriz de correlación. Además, se identifican patrones de desbalanceo en la clase objetivo, donde la cantidad de pacientes que han tenido un ACV (clase 1) es menor en comparación con aquellos que no lo han tenido (clase 0).

Para preparar los datos, se realizan varias transformaciones como la imputación de valores faltantes, especialmente en la columna bmi. Además, se divide el conjunto de datos en tres partes: un 60% para entrenamiento, un 20% para validación y un 20% para prueba, utilizando split estratificado para asegurar que la distribución de las clases en cada conjunto sea representativa de la población total.

El núcleo de la arquitectura es un modelo de Árbol de Decisión, que se utiliza debido a su capacidad para manejar relaciones no lineales entre las variables y proporcionar interpretabilidad en las predicciones. Este modelo clasifica a los pacientes en dos clases: aquellos que sufrirán un ACV y aquellos que no lo harán. Se optimiza utilizando métricas de rendimiento como precisión, recall, y F1-Score, que permiten medir de manera efectiva la capacidad del modelo para predecir correctamente los casos de ACV. Además, se evalúa la AUC-ROC para determinar qué tan bien el modelo discrimina entre las dos clases. El

modelo se entrena sobre el conjunto de entrenamiento, y se evalúa utilizando el conjunto de prueba para asegurar su generalización.

2.1.4 Predicción de precios de aguacate (CatBoost)

La arquitectura implementada en el notebook avocado_model.ipynb aborda la predicción del precio del aguacate como un problema de regresión de series temporales. El proceso inicia con la carga y limpieza de los datos, seguido de un análisis exploratorio (EDA) que identifica características clave, una fuerte estacionalidad que fue confirmada en el gráfico de promedios mensuales, alta multicolinealidad entre las variables de volumen detectada en la matriz de correlación y heterogeneidad por región vista en los diagramas de caja. Para preparar los datos, se aplica una robusta ingeniería de características donde se extraen señales de calendario (mes, semana del año), se transforman las variables de volumen sesgadas usando log1p, y se crean ratios de tamaño de bolsas para mitigar la multicolinealidad. De manera crucial, para capturar la dependencia temporal, se generan características de rezagos (lags) y medias móviles (rolling means) del precio y del volumen, utilizando shift(1) para evitar la fuga de datos.

El núcleo de la arquitectura es un modelo de Gradient Boosting, específicamente el CatBoostRegressor. Este modelo fue elegido por su capacidad superior para manejar relaciones no lineales e interacciones, y su ventaja distintiva de procesar variables categóricas (region, type) de forma nativa, evitando el one-hot encoding. Para la validación, los datos se dividen siguiendo un estricto orden cronológico en conjuntos de entrenamiento, lo cual es esencial para problemas de series temporales. El modelo se entrena utilizando el conjunto de entrenamiento, optimizando la métrica de Error Absoluto Medio (MAE), y emplea una estrategia de parada temprana basada en el rendimiento del conjunto de validación para seleccionar la mejor iteración, la 630 en este caso, y prevenir el sobreajuste.

2.1.5 Recomendación de películas (KNN híbrido)

La arquitectura construye un sistema de recomendación de películas utilizando un enfoque de Filtrado Colaborativo. Inicialmente, se adquieren y comprenden los datos de dos fuentes, "movies.csv" (información de películas) y "ratings.csv" (calificaciones de usuarios). Estos datos se cargan y se someten a un preprocesamiento que incluye análisis descriptivo, verificación de limpieza y exploración para entender las distribuciones de calificaciones y géneros. Un paso clave en la preparación es la creación de una matriz usuario-ítem, donde las filas son usuarios, las columnas son películas y los valores son las calificaciones. Dado que esta matriz es inherentemente dispersa (muchos valores faltantes), se aplica una imputación, rellenando los vacíos con la media global de las calificaciones para facilitar el modelado.

El núcleo del sistema es el algoritmo K-Nearest Neighbors (KNN), configurado para encontrar los 10 usuarios más similares (vecinos) a un usuario dado, utilizando la similitud del coseno como métrica para comparar sus patrones de calificación. Una vez entrenado el modelo KNN con la matriz usuario-ítem imputada, se utiliza para generar recomendaciones, identificando películas que les gustaron a usuarios similares. Adicionalmente, se implementa una función para predecir la calificación específica que un usuario daría a una película, calculándola como un promedio ponderado de las calificaciones de sus vecinos más cercanos.

2.1.6 Predicción de objetos

El modelo desarrollado se basa en la arquitectura YOLOv8 (You Only Look Once), una Red Neuronal Convolucional (CNN) de última generación diseñada específicamente para la detección de objetos en tiempo real. Su funcionamiento se distingue por ser un detector de una sola etapa, lo que significa que procesa la imagen completa en una única pasada a través de la red, en lugar de escanearla por secciones.

Para asegurar que el sistema reconozca vehículos a diferentes distancias, la arquitectura incluye un componente intermedio que fusiona la información visual a distintas escalas. Esto permite que el modelo detecte con la misma precisión un autobús grande en primer plano o una motocicleta pequeña al fondo de la calle. Finalmente, la predicción se realiza en la cabeza del modelo, el cual utiliza un enfoque moderno “anchor-free” y desacoplado. Esto significa que la red separa internamente la tarea de decidir qué es el objeto (clasificación) de la tarea de calcular dónde está exactamente (regresión de la caja), logrando así resultados más precisos y flexibles que las versiones anteriores.

2.1.7 Predicción de emociones (MobileNetV3 + Keras + Detector local)

La arquitectura implementada para el reconocimiento de emociones faciales aborda este problema como una tarea de **clasificación multiclase**, ejecutada completamente de manera local y sin depender de servicios externos. El sistema combina un **modelo Keras basado en MobileNetV3Large** con un **detector de rostros local** (YuNet en formato ONNX u Haar Cascade), lo que permite detectar y analizar rostros en imágenes con bajo costo computacional y alta velocidad.

El proceso inicia con la carga y preparación del dataset, compuesto por imágenes etiquetadas en distintas categorías emocionales. Se realiza una limpieza inicial de archivos corruptos y un análisis exploratorio para evaluar el balance entre clases, la distribución de tamaños y la calidad de las muestras. Debido a la variabilidad en iluminación, poses y resolución, se aplican técnicas de **data augmentation** como rotaciones leves, volteos horizontales y variaciones de brillo, con el fin de aumentar la robustez del modelo ante escenarios reales.

Para la fase de detección, cada imagen pasa primero por un módulo de localización facial. Este componente utiliza un modelo ONNX optimizado (YuNet) o, como alternativa, un clasificador Haar Cascade para identificar las regiones donde se encuentra el rostro. Una

vez detectadas las coordenadas, cada rostro se recorta y se normaliza a un tamaño estándar de **224 × 224 píxeles**, convirtiéndolo a formato RGB. Luego se aplica la función **preprocess_input** de MobileNetV3 para escalar los valores de los píxeles según el formato requerido por la arquitectura base.

El núcleo del sistema es un **modelo de aprendizaje profundo basado en MobileNetV3Large**, seleccionado por su alta eficiencia y capacidad para extraer características relevantes en imágenes con poca capacidad de cómputo. Este modelo actúa como extractor de características, sobre el cual se construye una capa densa final encargada de clasificar el rostro en una de las emociones disponibles (por ejemplo: feliz, triste, neutral, sorprendido, etc.). El entrenamiento se realiza utilizando el optimizador Adam, con una función de pérdida de entropía cruzada categórica, y un esquema de validación 80/20 para evaluar el rendimiento durante el entrenamiento.

El modelo converge tras varias épocas, mostrando mejoras progresivas en accuracy y estabilidad de la pérdida. Una evaluación posterior sobre el conjunto de prueba permite medir el rendimiento mediante métricas como **accuracy**, **F1-Score macro** y **matriz de confusión**, verificando la capacidad del sistema para clasificar emociones incluso en condiciones de iluminación difíciles o rostros parcialmente ocluidos. Además, se observa que el modelo generaliza adecuadamente gracias a la combinación de MobileNetV3 y el uso intensivo de data augmentation.

Finalmente, el modelo se serializa empleando el formato **.keras**, lo que permite conservar tanto la topología como los pesos optimizados. Este archivo se integra directamente en el backend del sistema mediante un microservicio dedicado, donde es cargado de forma diferida (lazy loading) y se procesa cada imagen recibida a través del flujo de detección, recorte, preprocesamiento y predicción. La arquitectura modular facilita futuras mejoras, permitiendo sustituir el detector o ajustar las clases emocionales sin afectar el resto del sistema.

2.2 Integración de los modelos con la API

La arquitectura integra múltiples micro-APIs de machine learning bajo un servidor central de FastAPI.

El archivo **api/main.py** agrupa cada servicio especializado (por ejemplo, **/bitcoin, /movies**) y expone endpoints globales como **/ask, /health** y **/models**, asegurando registro unificado y control de disponibilidad.

El flujo de conversación es gestionado por **llm/coordinator.py**, donde se interpreta la consulta del usuario, extrae parámetros mediante **llm/extract_params.py** y selecciona el microservicio adecuado según el mapeo definido en **llm/available_models.py.**

Cada micro-API en **api/routes**/ carga de forma perezosa su modelo serializado desde **ml_models**/, utiliza Pydantic para validar las solicitudes y respuestas, mantiene un esquema común de logging y ofrece su propio endpoint /**health**.

Tras procesar la predicción, el coordinador usa nuevamente el LLM para generar una respuesta contextual y natural, según el **response_type** de cada modelo.

El resultado final se devuelve al cliente a través de /**ask**, aunque los usuarios avanzados pueden consumir directamente los endpoints específicos de cada servicio.

3. Justificación del Modelo Desarrollado

El desarrollo del proyecto Mariantonieta-IA se fundamenta en la necesidad de construir un sistema inteligente capaz de integrar múltiples dominios de aprendizaje automático dentro de una misma plataforma, manteniendo al mismo tiempo independencia, escalabilidad y seguridad en el manejo de datos.

Una de las principales decisiones de diseño fue implementar una arquitectura basada en microservicios, lo que permitió que cada integrante del equipo trabajara de manera autónoma en su propio modelo de machine learning y su respectiva API. Este enfoque favoreció el desarrollo paralelo, redujo los conflictos de integración y facilitó la incorporación de nuevos modelos sin afectar la estabilidad del sistema. Cada microservicio encapsula su lógica, dependencias y modelo entrenado, exponiendo únicamente un endpoint REST estandarizado que puede ser consumido por el coordinador general.

Además, la adopción de un modelo de lenguaje de gran tamaño (LLM) local, en este caso Llama 3 ejecutado mediante Ollama, responde tanto a criterios técnicos como económicos. A nivel de seguridad, esta elección garantiza que toda la información procesada por el asistente —incluyendo consultas, parámetros y posibles datos sensibles— se mantenga de forma local, sin requerir el envío de información a servidores externos. Esto permite cumplir con buenas prácticas de privacidad y confidencialidad de datos, especialmente en escenarios donde se manejan dominios médicos o financieros.

Desde el punto de vista económico y operativo, el uso de un LLM local resulta más rentable que depender de servicios de terceros como la API de OpenAI, cuyo costo aumenta con el volumen de consultas. En cambio, Ollama permite ejecutar modelos de lenguaje de manera gratuita y eficiente, aprovechando los recursos de hardware disponibles sin incurrir en costos adicionales por uso o tráfico.

3.1 Comparación con alternativas posibles

Durante el diseño de Mariantonieta-IA se evaluaron distintas alternativas arquitectónicas y tecnológicas antes de definir el enfoque final. Cada una ofrecía ventajas en ciertos aspectos, pero también limitaciones en términos de flexibilidad, costo y control sobre los datos.

Una de las principales alternativas consideradas fue la implementación de una arquitectura monolítica, donde todos los modelos y servicios se integrarían dentro de una única aplicación. Si bien este enfoque simplifica la comunicación interna y el despliegue inicial, presenta serios inconvenientes para un proyecto colaborativo. En un entorno con múltiples desarrolladores y modelos de machine learning heterogéneos, el mantenimiento y la integración continua se vuelven complejos. Un error o cambio en un modelo podría afectar a toda la aplicación. Por ello, se optó por una arquitectura de microservicios, que brinda independencia total entre módulos, favorece la escalabilidad y permite el desarrollo simultáneo de cada API de manera autónoma.

En cuanto al modelo de coordinación, se analizó la posibilidad de usar servicios de inteligencia artificial en la nube, como OpenAI GPT-5, Google Vertex AI o Azure OpenAI Service. Aunque estas plataformas ofrecen capacidades avanzadas y fácil integración, su uso implica costos recurrentes por consulta, además de la exposición de datos a servidores externos. Dado que uno de los objetivos del proyecto era garantizar la privacidad y la soberanía de la información, se descartó este enfoque en favor de una solución local basada en Ollama, utilizando el modelo Llama 3, que permite ejecutar un LLM directamente en el entorno local del servidor sin costos adicionales por uso.

3.3 Aportes del sistema a la aplicación de IA en la vida real

El presente proyecto constituye una demostración práctica del potencial de la inteligencia artificial aplicada en entornos reales. Su diseño modular, basado en microservicios y coordinado por un modelo de lenguaje local, sienta las bases para el desarrollo de plataformas inteligentes adaptables a diferentes contextos de uso.

Uno de los principales aportes del sistema es evidenciar cómo la IA puede funcionar no solo como un componente aislado de predicción o análisis, sino como un coordinador central capaz de orquestar múltiples servicios. Este enfoque permite integrar distintos

modelos especializados —por ejemplo, de predicción, clasificación, recomendación o análisis de sentimientos— bajo una misma interfaz conversacional, ofreciendo una experiencia unificada al usuario.

Gracias a esta arquitectura, el proyecto puede ser fácilmente extendido o adaptado a otras áreas fuera del ámbito académico. Entre sus posibles aplicaciones destacan:

1. **Sistemas de atención al cliente**, donde un LLM coordina módulos de respuesta automática, análisis de emociones del usuario mediante reconocimiento facial y comprensión de voz en tiempo real.
2. **Plataformas educativas inteligentes**, en las que el asistente puede responder dudas, guiar procesos de aprendizaje o evaluar el progreso del estudiante.
3. **Asistentes empresariales o administrativos**, capaces de interactuar con bases de datos, generar reportes y ofrecer soporte interno mediante lenguaje natural.
4. **Aplicaciones médicas o de bienestar**, donde los microservicios de análisis de voz o rostro podrían integrarse con modelos de diagnóstico o detección temprana de anomalías.

4. Análisis de Resultados Obtenidos

La etapa de análisis de resultados tiene como objetivo evaluar el rendimiento de los distintos modelos de *machine learning* desarrollados para la plataforma **Mariantonieta-IA**, verificando su capacidad predictiva, su comportamiento frente a los datos de prueba y su adecuación a los objetivos del proyecto.

Dado que cada microservicio aborda un dominio distinto —como finanzas, transporte, medicina, agricultura o entretenimiento—, los modelos fueron evaluados de manera independiente, utilizando métricas específicas para cada tipo de problema (clasificación o regresión). Este enfoque permite obtener una visión más precisa de la efectividad individual de cada modelo, así como de su contribución al desempeño global del asistente.

4.1 Resultados por dominio

4.1.1 Bitcoin

La evaluación del modelo de predicción del precio de Bitcoin se encuentra registrada en el archivo **notebooks/bitcoin/bitcoin_notebook_v2.ipynb**, específicamente en la celda 29 del cuaderno de trabajo. En esta fase, se combinaron las predicciones generadas por el modelo **Prophet** (*yhat*) con los valores reales de cierre para calcular las principales métricas de desempeño.

Los resultados obtenidos fueron los siguientes:

- **MSE (Mean Squared Error):** 11 422.81
- **MAE (Mean Absolute Error):** 72.84 USD
- **RMSE (Root Mean Squared Error):** 106.88 USD

El valor de **MAE** cercano a 73 USD indica que, en promedio, las predicciones diarias del modelo difieren unos 73 dólares respecto al precio real de cierre. Por su parte, el **RMSE** de aproximadamente 107 USD penaliza más los desvíos grandes, reflejando la magnitud típica del error esperable en un activo tan volátil como Bitcoin.

Es importante señalar que la evaluación se realizó sobre el mismo periodo de entrenamiento, sin una separación explícita entre conjuntos de entrenamiento y prueba. Por tanto, las métricas reflejan principalmente la capacidad de ajuste histórico del modelo (*in-sample performance*) y no necesariamente su capacidad de generalización a datos futuros.

Como pasos futuros, se recomienda:

1. **Recalcular las métricas sobre un conjunto de prueba reciente**, con el fin de evaluar el comportamiento del modelo en escenarios fuera de muestra.

2. **Incorporar métricas relativas**, como el **MAPE (Mean Absolute Percentage Error)**, que permiten interpretar el error en términos porcentuales respecto al precio de cierre.

4.1.2 Vuelos

Los resultados del modelo de predicción de retrasos en vuelos muestran un **desempeño altamente satisfactorio**, evidenciando la efectividad del algoritmo **Random Forest** para este tipo de problema.

Según el análisis documentado en el cuaderno **notebooks/flights/flights_notebook.ipynb**, el modelo fue entrenado con una partición 80/20 del conjunto de datos (*random_state = 42*) y evaluado sobre el bloque de prueba. Los resultados obtenidos fueron:

- **MAE (Mean Absolute Error):** 10.89 minutos
- **RMSE (Root Mean Squared Error):** 16.11 minutos
- **R^2 (Coeficiente de Determinación):** 0.914

Estas métricas reflejan que el modelo logra una **precisión promedio de aproximadamente 11 minutos** de error respecto al retraso real, mientras que las desviaciones más grandes se mantienen dentro de un rango de **16 minutos**. El coeficiente de determinación del **91 %** indica que el modelo explica la gran mayoría de la varianza presente en los retrasos, lo que confirma su solidez predictiva.

El rendimiento alcanzado se sustenta en un proceso de preparación de datos adecuado, que incluyó la **codificación de etiquetas** para aerolíneas y aeropuertos, y una configuración de **200 árboles** con **profundidad máxima de 20 niveles**, parámetros que ofrecen un equilibrio entre capacidad de generalización e interpretabilidad del modelo. Además, el análisis de importancia de variables sugiere que factores como el retraso en la salida y los horarios programados son los más influyentes en la estimación final.

En términos de mejora, se recomienda **analizar los residuos por rango horario o aerolínea** para detectar posibles sesgos, así como **validar el modelo en un bloque temporal más reciente**, con el fin de comprobar su capacidad de generalización más allá del periodo de entrenamiento (2008).

4.1.3 ACV

La evaluación del modelo Árbol de Decisión muestra un rendimiento aceptable, con una precisión de 71.33%, un recall de 68.00%, y un F1-Score de 31.78%. La AUC-ROC de 0.8201 indica una buena capacidad de discriminación entre las clases, aunque el modelo presenta un desbalance, favoreciendo más la clase negativa (sin ACV), con una precisión de 98% y un recall de 71% para esa clase.

El análisis de la matriz de confusión muestra 692 casos correctamente clasificados como negativos (sin ACV) y 37 casos positivos (con ACV), pero también 280 falsos positivos y 13 falsos negativos. El reporte de clasificación revela que, aunque la clase negativa es bien identificada, la clase positiva tiene un recall alto (74%) pero una precisión muy baja (12%).

La importancia de las características destaca variables como la presión arterial, el IMC, y la hipertensión como las más influyentes en la predicción del ACV. Sin embargo, el modelo aún tiende a predecir mayormente la clase negativa, debido al desbalance en los datos.

4.1.4 Aguacate

La evaluación del modelo CatBoostRegressor demuestra un rendimiento robusto y una fuerte capacidad de generalización para predecir el precio promedio del aguacate. Las métricas clave en el conjunto de prueba fueron un Error Absoluto Medio (MAE) de 0.1181, un Error Cuadrático Medio Raíz (RMSE) de 0.1660 y un Error Porcentual Absoluto Medio (MAPE) de 8.21%. Estos valores son muy consistentes con los obtenidos en el conjunto de

validación (VAL MAE 0.1380), lo que indica que el modelo no está sobreajustado y es estable. El análisis gráfico de los valores reales frente a los predichos muestra una fuerte correlación, con los puntos agrupados estrechamente alrededor de la línea diagonal. El histograma de residuos confirma la falta de sesgo significativo, ya que está centrado correctamente en cero.

El análisis de la importancia de las características revela que el modelo captura eficazmente la naturaleza de la serie temporal del precio. La variable más influyente es el precio de la semana anterior (AveragePrice_lag_1), seguida por las medias móviles y la proporción de bolsas grandes (large_ratio). Esto subraya que la memoria a corto plazo, la estacionalidad anual y la composición del producto son determinantes clave del precio. Al visualizar la predicción en el tiempo, el modelo sigue con precisión la tendencia y la estacionalidad del precio real, aunque tiende a suavizar los picos y valles extremos, un comportamiento común en modelos de regresión.

4.1.5 Películas

La evaluación del modelo de recomendación KNN arrojó un Error Cuadrático Medio Raíz (RMSE) de 1.11, calculado sobre una muestra de datos. Este valor indica una precisión razonable en la estimación de las calificaciones que los usuarios darían a las películas, sugiriendo que el error promedio entre la predicción y el valor real es moderado dentro de la escala de calificación utilizada. Adicionalmente, el análisis de la distribución de las calificaciones predichas mostró una clara tendencia del modelo a generar valores centrados alrededor de 3.2, lo que implica una propensión a predecir calificaciones moderadas más frecuentemente que calificaciones extremas (muy altas o muy bajas).

4.1.6 Detección de vehículos

El modelo entrenado demostró un rendimiento sobresaliente para la tarea de detección de tráfico urbano, alcanzando una Precisión Promedio Media global del 94.3%.

Este resultado indica que el sistema es capaz de localizar y clasificar correctamente la gran mayoría de los vehículos presentes en las imágenes de prueba. Al analizar el desempeño individual por clases, se observó que categorías con características visuales muy distintivas, como los Automóviles y Camiones, obtuvieron las puntuaciones más altas (superiores al 95%), validando que la arquitectura YOLOv8n logró generalizar eficazmente las características morfológicas de estos vehículos a pesar de la variabilidad en ángulos e iluminación del dataset.

Por otro lado, el análisis de la matriz de confusión revela que el modelo mantiene una robustez notable, con una tasa de falsos positivos muy baja. Aunque la clase Autobús presentó una precisión ligeramente inferior (aprox. 89%) en comparación con las demás, aún así el sistema sigue siendo altamente confiable para aplicaciones de monitoreo en tiempo real. En conjunto, estas métricas confirman que la estrategia de Transfer Learning fue exitosa, permitiendo adaptar un modelo pre-entrenado a un entorno específico con un conjunto de datos limitado pero bien balanceado.

4.1.7 Detección de emociones

El entrenamiento del modelo de reconocimiento de emociones se desarrolló en dos fases con el objetivo de evaluar su capacidad de aprendizaje y su comportamiento en un dominio tan complejo como el del dataset FER2013. En la primera etapa, donde la base convolucional de MobileNetV3 permaneció congelada, el modelo alcanzó un *training accuracy* de 41.47% y un *validation accuracy* de 47.15% tras diez épocas de entrenamiento, con pérdidas finales de 1.5450 y 1.4031 respectivamente. El tiempo por época osciló entre 300 y 530 segundos, evidenciando un proceso estable en el que la pérdida disminuyó progresivamente y la precisión de validación se mantuvo superior a la de entrenamiento, indicando una adecuada capacidad inicial de generalización. En la segunda fase se aplicó *fine-tuning* descongelando parcialmente las capas del modelo base, lo que permitió un ajuste más fino a las características específicas del conjunto de datos. Después de diez

épocas adicionales, el modelo registró un *training accuracy* de 43.42% y un *validation accuracy* de 48.08%, acompañado de pérdidas de 1.4850 y 1.3784. Esta etapa representó una mejora de 0.93 puntos porcentuales en la precisión de validación, confirmando la efectividad del *transfer learning* sin provocar sobreajuste.

Una vez completado el entrenamiento, el modelo fue evaluado en el conjunto de prueba, alcanzando un *test accuracy* de 47.42% y una pérdida de 1.3814 en un tiempo total de 46 segundos para procesar 449 lotes. Estos valores se encuentran dentro del rango esperado para arquitecturas basadas en MobileNet aplicadas al FER2013, un dataset que es reconocido por su dificultad debido a la baja resolución, variabilidad en condiciones de iluminación y expresiones ambiguas. Como ejemplo ilustrativo, se analizó una imagen de prueba empleando el flujo completo del sistema —detección del rostro, recorte, preprocesamiento y predicción— obteniendo como resultado la emoción *happy* con una confianza del 64.40%. La distribución de probabilidades mostró un comportamiento coherente, asignando valores elevados a categorías relacionadas y manteniendo proporciones realistas entre emociones cercanas.

5. Conclusiones

El desarrollo del proyecto Mariantonieta-IA permitió demostrar de manera práctica cómo diferentes áreas de la inteligencia artificial pueden integrarse dentro de una arquitectura modular, flexible y escalable. La combinación de modelos de machine learning, visión por computadora, reconocimiento de voz y procesamiento del lenguaje natural dio lugar a un asistente conversacional multimodal capaz de interpretar información proveniente del texto, la voz y las imágenes, y de generar respuestas coherentes y contextualizadas para el usuario.

La adopción de una arquitectura basada en microservicios fue fundamental para el éxito del sistema, ya que permitió distribuir el trabajo entre los diferentes integrantes del

equipo sin generar dependencia directa entre los componentes. Cada modelo se desarrolló, entrenó y desplegó de forma independiente, integrándose posteriormente mediante APIs REST. Este enfoque facilitó la mantenibilidad, promovió la reutilización de componentes y garantizó que nuevos servicios pudieran incorporarse en el futuro sin afectar la estabilidad del sistema.

Otro aspecto clave fue la incorporación de un modelo de lenguaje local —Llama 3 ejecutado mediante Ollama— como núcleo de coordinación semántica. Su ejecución local permitió mantener la privacidad de los datos y evitar costos asociados a servicios externos, aumentando la autonomía tecnológica del proyecto y reduciendo la latencia en la comunicación interna. Esta decisión reforzó la visión de un asistente completamente autocontenido, capaz de operar sin depender de proveedores externos.

Asimismo, los modelos de procesamiento visual y de voz fueron ejecutados de forma local, reemplazando las dependencias anteriores con soluciones propias optimizadas para el sistema. El módulo de reconocimiento facial emplea un modelo interno de visión por computadora capaz de detectar rostros y estimar emociones sin necesidad de servicios en la nube, mientras que el componente de Speech-to-Text utiliza un modelo local de transcripción que permite habilitar interacciones por voz manteniendo la privacidad y el control total sobre los datos procesados.

En términos de desempeño, los distintos modelos predictivos y de análisis mostraron resultados adecuados para la naturaleza de cada dominio. Tanto los modelos tradicionales (como Prophet, Random Forest, Árboles de Decisión y CatBoost) como los modelos de deep learning presentaron métricas consistentes y útiles dentro del contexto del asistente. Estos resultados evidencian que la combinación de modelos especializados con un coordinador basado en lenguaje natural es efectiva para construir sistemas conversacionales más inteligentes y versátiles.

En conjunto, Mariantonieta-IA no solo cumple con los objetivos académicos del curso, sino que también demuestra ser una propuesta viable para escenarios reales. Su arquitectura modular, sumada al uso de modelos locales, lo convierte en una base sólida para aplicaciones como asistentes empresariales, sistemas educativos inteligentes, soluciones de soporte técnico, herramientas de análisis interactivo o incluso plataformas médicas y de bienestar digital. En conclusión, este proyecto confirma que el uso combinado de microservicios y modelos de lenguaje locales representa una ruta prometedora para el diseño de asistentes inteligentes seguros, eficientes y completamente autosuficientes.

6. Bibliografía

Facebook Research. (2017). *Prophet: Forecasting at scale* [Software]. GitHub. https://github.com/facebook/prophet

Google Cloud. (2024). *Cloud Vision API documentation*. *Google Cloud Platform*. https://cloud.google.com/vision

Google Cloud. (2024). *Speech-to-Text API documentation*. *Google Cloud Platform*. https://cloud.google.com/speech-to-text

Joblib Developers. (2024). *Joblib: Efficient serialization of Python objects* [Software]. https://joblib.readthedocs.io

LangChain AI. (2024). *LangChain: Framework for developing applications powered by language models* [Software]. https://github.com/langchain-ai/langchain

Ollama. (2024). *Ollama: Run Llama and other large language models locally* [Software]. https://ollama.ai

Tiangolo, S. (2018). *FastAPI: Modern, fast (high-performance) web framework for building APIs with Python* [Software]. https://fastapi.tiangolo.com

7. Anexos

A. Enlace al repositorio del proyecto

https://github.com/SMatey/Mariantonieta-IA

B. Estructura de carpetas del proyecto

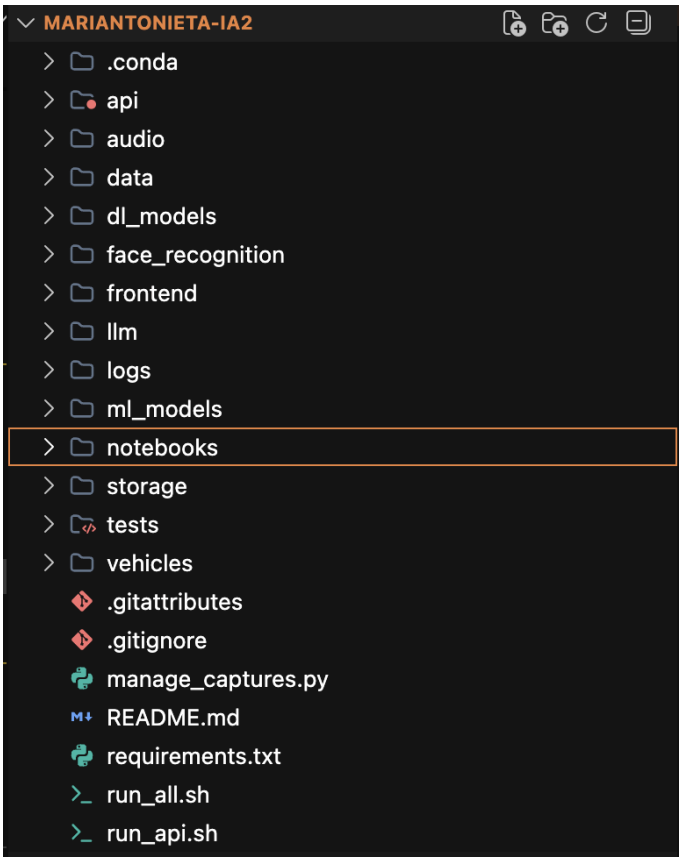

Fig 9: Estructura del Proyecto

C. Ejemplo de logs y llamadas API

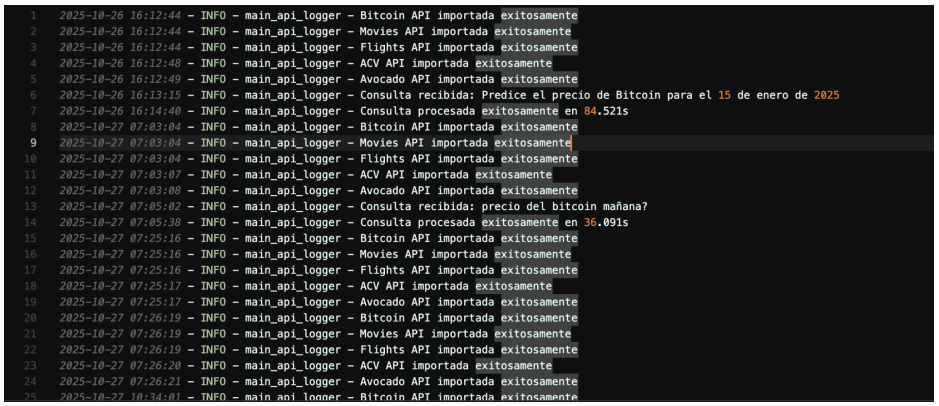

Fig 10: Logs de la API principal