**Documentación Detallada de la API para Predicción y Evaluación de Precios de Autos**

**Introducción**

Esta API, desarrollada con FastAPI, proporciona tres servicios principales relacionados con la predicción y evaluación de precios de automóviles:

1. Predicción del precio óptimo basado en las especificaciones de un automóvil.

2. Evaluación de un precio propuesto por el vendedor en comparación con el precio óptimo.

3. Recomendación de un automóvil basado en el presupuesto y preferencias del usuario.

El modelo de predicción se ha entrenado previamente y se carga desde un archivo 'modelo.pkl'. La API también utiliza un conjunto de datos de automóviles almacenado en 'car\_price\_prediction.csv' para proporcionar recomendaciones.

**Descripción de los servicios**

**Servicio 1: Predicción del precio óptimo ('/predict\_price/')**

Ruta: '/predict\_price/'

Método HTTP: 'POST'

Descripción: Dado un conjunto de especificaciones de un automóvil, el servicio predice el precio óptimo utilizando un modelo de aprendizaje automático previamente entrenado.

Entrada:

Un objeto JSON que representa las especificaciones del automóvil:

json

{

"levy": float,

"manufacturer": "string",

"model": "string",

"prod\_year": int,

"category": "string",

"leather\_interior": bool,

"fuel\_type": "string",

"engine\_volume": float,

"mileage": float,

"cylinders": int,

"gear\_box\_type": "string",

"drive\_wheels": "string",

"doors": int,

"wheel": "string",

"color": "string",

"airbags": int

}

Salida:

Un objeto JSON con el precio óptimo predicho:

json

{

"optimal\_price": float

}

Proceso:

1. Se recibe el objeto 'CarSpecifications'.

2. Se convierte en un DataFrame de Pandas.

3. Se aplican transformaciones a las columnas categóricas utilizando 'LabelEncoder'.

4. Se realiza la predicción utilizando el modelo cargado.

5. Se devuelve el precio óptimo.

**Servicio 2: Evaluación del precio propuesto ('/evaluate\_price/')**

Ruta: '/evaluate\_price/'

Método HTTP: 'POST'

Descripción: Evalúa si el precio propuesto por un vendedor es razonable en comparación con el precio óptimo predicho por el modelo.

Entrada:

Un objeto JSON que incluye las especificaciones del automóvil y el precio propuesto:

json

{

"specifications": { ... }, // Mismo formato que en /predict\_price/

"proposed\_price": float

}

Salida:

Un objeto JSON con la evaluación:

json

{

"evaluation": "string"

}

Proceso:

1. Se reciben las especificaciones y el precio propuesto.

2. Se predice el precio óptimo utilizando el mismo proceso que en el Servicio 1.

3. Se compara el precio propuesto con el predicho.

4. Si la diferencia es menor al 10%, se considera razonable.

5. Se devuelve la evaluación correspondiente.

**Servicio 3: Recomendación de automóvil ('/recommend\_car/')**

Ruta: '/recommend\_car/'

Método HTTP: 'POST'

Descripción: Sugiere el mejor automóvil disponible basado en el presupuesto y las preferencias del usuario.

Entrada:

Un objeto JSON con el presupuesto y preferencias opcionales:

json

{

"budget": float,

"manufacturer": "string", // Opcional

"category": "string", // Opcional

"fuel\_type": "string", // Opcional

"min\_prod\_year": int // Opcional

}

Salida:

Un objeto JSON con el automóvil recomendado o un mensaje si no se encontraron coincidencias:

json

{

"recommended\_car": { ... } // Detalles del automóvil recomendado

}

// O

{

"recommendation": "No cars found matching the criteria."

}

Proceso:

1. Se carga el conjunto de datos desde 'car\_price\_prediction.csv'.

2. Se filtra el conjunto de datos según el presupuesto y las preferencias.

3. Si no hay autos que coincidan, se devuelve un mensaje indicándolo.

4. Si hay coincidencias, se devuelve el primer automóvil filtrado como recomendación.

**Revisión de seguridad**

La seguridad es un aspecto crítico en cualquier API. A continuación, se detallan los puntos de seguridad considerados y las posibles mejoras:

Validación de entrada

- Uso de pydantic: Se utilizan modelos de datos ('BaseModel') para validar y tipar las entradas de los usuarios, lo que ayuda a prevenir inyecciones de código y entradas maliciosas.

- Limitaciones:

- Validación insuficiente de datos categóricos: No se validan los valores permitidos para campos categóricos, lo que podría llevar a errores si se proporcionan valores inesperados.

- Validación de rangos numéricos: No se especifican validaciones para asegurar que los valores numéricos estén dentro de rangos razonables (por ejemplo, años de producción futuros, valores negativos, etc.).

**Manejo de excepciones**

- Captura de excepciones: Se capturan excepciones durante la predicción y se devuelven como errores HTTP 500 con detalles.

- Limitaciones:

- Exposición de detalles internos: Al devolver el mensaje de excepción directamente, se podrían filtrar detalles internos del sistema que un atacante podría aprovechar.

- No manejo de excepciones en todas las operaciones: Algunas operaciones, como la carga del modelo y del conjunto de datos, podrían fallar y no están envueltas en bloques 'try-except'.

**Carga de modelos y datos**

- Rutas absolutas: Se utiliza una ruta absoluta para cargar el conjunto de datos, lo que puede generar problemas de seguridad y portabilidad.

- Limitaciones:

- Control de acceso: No hay control sobre quién puede acceder a qué servicios; todos están abiertos públicamente.

**Configuración CORS**

- Permitir orígenes: Se configura 'allow\_origins=["\*"]', permitiendo solicitudes desde cualquier origen.

- Limitaciones:

- Exposición a solicitudes no deseadas: Esto es útil para desarrollo, pero en producción debería restringirse a orígenes confiables.

Posibles mejoras en seguridad

- Validación estricta de datos:

- Definir enumeraciones ('Enum') para campos categóricos.

- Implementar validaciones de rangos para valores numéricos.

- Manejo seguro de excepciones:

- Devolver mensajes genéricos de error en lugar de detalles específicos.

- Control de acceso y autenticación:

- Implementar autenticación para proteger ciertos endpoints.

- Restricción de orígenes CORS:

- Configurar 'allow\_origins' con una lista específica de orígenes confiables en producción.

- Uso de rutas relativas:

- Evitar rutas absolutas al cargar archivos, para mejorar la seguridad y portabilidad.

**Patrones utilizados**

Modelo-Vista-Controlador (MVC)

- Modelo: Representado por los datos y el modelo de predicción ('modelo.pkl').

- Vista: No aplica directamente, ya que es una API, pero las respuestas JSON actúan como la vista.

- Controlador: Las funciones de los endpoints manejan la lógica de negocio y coordinan entre el modelo y la vista.

Inyección de dependencias

- Carga del modelo y dataset: El modelo y el conjunto de datos se cargan globalmente y se utilizan dentro de las funciones, lo que simplifica el acceso pero puede mejorarse utilizando la inyección de dependencias que ofrece FastAPI.

Uso de pydantic para validación

- Modelos de datos: Se utilizan clases que heredan de 'BaseModel' para definir y validar los datos de entrada.

Separación de responsabilidades

- Funciones claras: Cada endpoint tiene una responsabilidad específica y su lógica está contenida dentro de su función.

**Uso de middlewares**

Middleware de CORS

- Funcionalidad: Permite o restringe solicitudes de orígenes distintos al servidor.

- Configuración sctual: Está configurado para permitir solicitudes desde cualquier origen ('allow\_origins=["\*"]').

Middleware de logging

- Funcionalidad: Registra cada solicitud entrante y el tiempo que toma procesarla.

- Detalles registrados:

- ID de la solicitud (si se proporciona).

- Método HTTP y URL.

- Tiempo de procesamiento de la solicitud.

- Uso de decoradores: Se utiliza el decorador '@app.middleware("http")' para aplicar el middleware a todas las solicitudes HTTP.

**Modularización del código**

Actualmente, todo el código reside en un solo archivo. Aunque es funcional para aplicaciones pequeñas, es recomendable modularizar el código para mejorar su mantenibilidad y escalabilidad.

Posibles mejoras en modularización

- Separación por módulos:

- Modelos de datos ('models.py'): Definir los 'BaseModel' de Pydantic en un archivo separado.

- Servicios o rutas ('routes.py'): Mover los endpoints a un archivo dedicado.

- Utilidades ('utils.py'): Funciones auxiliares, como la carga y transformación de datos.

- Configuración ('config.py'): Variables de configuración y constantes.

- Carga del modelo:

- Utilizar una función de inicialización o inyección de dependencias para cargar el modelo de manera más segura y eficiente.

- Middleware en archivos separados:

- Definir middlewares en archivos separados para mantener el código limpio.

Beneficios de la modularización

- Mantenibilidad: Facilita la actualización y el mantenimiento del código.

- Escalabilidad: Permite agregar nuevas funcionalidades sin que el archivo principal se vuelva inmanejable.

- Colaboración: Mejora la organización del proyecto, facilitando el trabajo en equipo.