Documento Técnico: API de Predicción de Categorías

Versión: 1.0

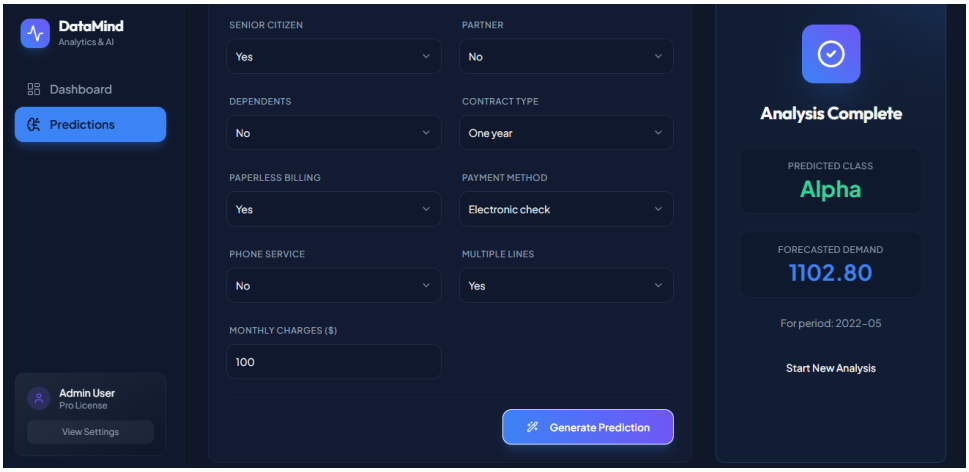
Fecha: 18 de enero de 2026

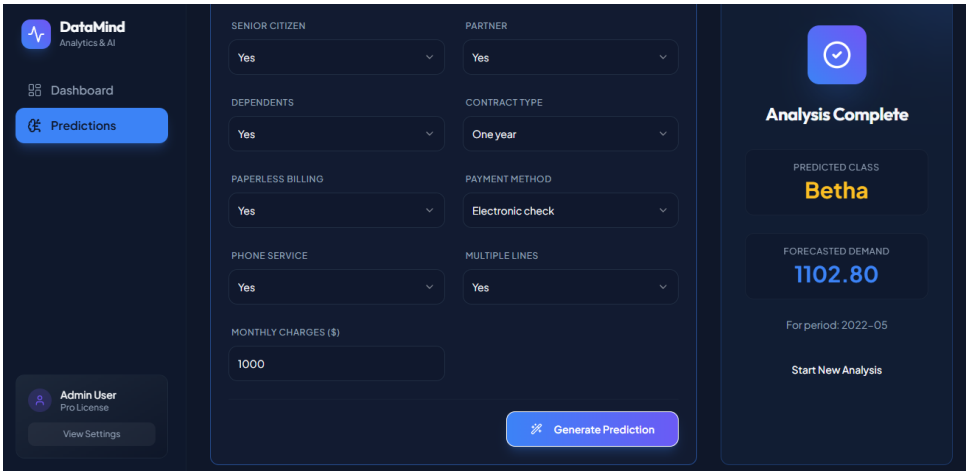
Autor: Robinson A. Moncada

1. Resumen Ejecutivo

Este documento describe la arquitectura y el funcionamiento de una API de predicción desarrollada con **FastAPI**. La API expone un único endpoint `POST /predict/` que recibe datos de entrada en formato JSON, los procesa y utiliza un modelo de machine learning pre-entrenado **(`trained_model.pkl`)** para predecir una de dos categorías: 'Alpha' o 'Betha'.

El objetivo de este servicio es proporcionar una interfaz ligera, rápida y escalable para consumir el modelo de predicción en aplicaciones de producción.





2. Arquitectura General

El sistema sigue una arquitectura simple y directa, típica de un servicio de inferencia de machine learning:

```
Cliente (Aplicación Web/Móvil/Servicio)
    ↓ (Petición HTTP POST con JSON)
  [API FastAPI - main.py]
    ↓ (1. Validación de datos con Pydantic)
  [Preprocesamiento de Datos]
    ↓ (2. Codificación y Escalado)
  [Modelo de ML - trained_model.pkl]
    ↓ (3. Predicción)
  [Postprocesamiento]
    ↓ (4. Mapeo de resultado a texto)
    ↑ (Respuesta HTTP con JSON)
Cliente (Recibe {"prediction": "Alpha/Betha"})
```

Componentes clave:

1. **Servidor API (FastAPI):** Orquesta el flujo de la solicitud, maneja la validación de entrada y devuelve la respuesta.

2. **Preprocesador:** Transforma los datos crudos del cliente al formato numérico que el modelo espera.

3. **Modelo de Machine Learning**: Un artefacto serializado (`.pkl`) que realiza la predicción.

4. Artefactos de Soporte: El `scaler.pkl` para el escalado de datos numéricos y los mapeos de codificación definidos en el código.

3. Descripción de Componentes y Flujo de Trabajo

A continuación se detalla el flujo interno de la API, basado en el archivo `main.py`.

3.1. Dependencias y Framework

* **FastAPI:** Framework web de alto rendimiento para construir APIs. Proporciona validación de datos automática a través de Pydantic y genera documentación interactiva (Swagger UI).

* **Pydantic (`BaseModel`):** Se utiliza para definir el esquema de los datos de entrada (`PredictRequest`), garantizando la validación de tipos y estructura.

* **Pandas:** Librería fundamental para la manipulación de datos. Se usa para estructurar la entrada en un `DataFrame`, que es el formato que el modelo y el `scaler` esperan.

* **Joblib:**Se utiliza para cargar los artefactos serializados en disco, como el modelo de machine learning (`trained_model.pkl`) y el escalador de características (`scaler.pkl`).

3.2. Esquema de Datos de Entrada (`PredictRequest`)

El endpoint espera un cuerpo de solicitud en formato `application/json` con la siguiente estructura:

```json
{
  "Partner": "string",
  "Dependents": "string",
  "Service1": "string",
  "Service2": "string",
  "Security": "string",
  "OnlineBackup": "string",
  "DeviceProtection": "string",
  "TechSupport": "string",
  "Contract": "string",
```

"PaperlessBilling": "string",

"PaymentMethod": "string",

"Charges": "float",

"Demand": "int"

}

```

Como se puede observar en la documentación generada por FastAPI (imagen adjunta), todos los campos son obligatorios y tienen tipos específicos. La variable `Demand` es un entero, pero **no se utiliza en el proceso de predicción**, lo que sugiere que podría ser una característica obsoleta o reservada para futuras versiones del modelo.

### 3.3. Preprocesamiento de Datos

Este es el paso más crítico para asegurar que los datos enviados a la API sean compatibles con los datos con los que se entrenó el modelo.

#### 1. Codificación de Variables Categóricas:

-Se utiliza un diccionario de mapeo fijo (`label_encoders`) para transformar las características categóricas (ej. 'Yes'/'No', 'Month-to-month') a valores numéricos (ej. 1/0, 0/1/2).

**-Importante:** Este mapeo debe ser idéntico al utilizado durante el entrenamiento del modelo. Cualquier cambio en los valores de entrada (ej. un nuevo método de pago) que no esté en el diccionario causará un error (`NaN`).

#### 2.Escalado de Variables Numéricas:

* La característica `Charges` es un valor numérico que debe ser escalada.

* Se carga un `scaler` pre-entrenado (`scaler.pkl`) y se aplica a la columna `Charges` usando `scaler.transform()`.

* Crítico: Se debe usar el *mismo* scaler que se usó para los datos de entrenamiento para mantener la consistencia y evitar predicciones erróneas.
```

3.4. Inferencia y Postprocesamiento

1. **Predicción:**Una vez que los datos están preprocesados, se pasan al modelo cargado (`model.predict(input_data)`). El modelo devuelve una predicción numérica (0 o 1).

2. **Mapeo de Clases:** El resultado numérico se mapea a su etiqueta correspondiente usando el diccionario `class_mapping`:

   * **`0` -> `'Alpha'`**
   * **`1` -> `'Betha'`**

3. **Respuesta:** La API devuelve un objeto JSON simple con la predicción:

   ```json
   { "prediction": "Alpha" }
   ```

4. Guía de Despliegue y Puesta en Marcha

Para ejecutar este servicio en un entorno local o de desarrollo, siga estos pasos:

1. **Prerrequisitos:**
   * Python 3.7+
   * pip (gestor de paquetes de Python)

2. **Archivos Necesarios:**
   * `main.py` (el código de la API)
   * `trained_model.pkl` (el modelo de machine learning)
   * `scaler.pkl` (el escalador de características)

3. **Instalación de Dependencias:**

   ```bash
   ```

pip install "fastapi[all]" pandas scikit-learn joblib

```

*(Nota: `scikit-learn` es una dependencia implícita necesaria para `joblib` y el scaler).*

4. Ejecución del Servidor:

```bash

uvicorn main:app --reload

```

* Esto iniciará el servidor en `http://127.0.0.1:8000`.

* La documentación interactiva (Swagger UI) estará disponible en `http://127.0.0.1:8000/docs`.

5. Consideraciones y Mejoras Futuras

Gestión de Errores: La validación de Pydantic manejará errores de tipo y campos faltantes. Sin embargo, se podría mejorar el manejo de valores categóricos no vistos (que resultarían en `NaN` después del `map`) para devolver un error HTTP 400 más descriptivo.

Logging: Implementar un sistema de logging para registrar las solicitudes, las predicciones y cualquier error que ocurra. Esto es crucial para el monitoreo en producción.

Versionado del Modelo: La implementación actual carga un archivo de modelo fijo. Para producción, se debería considerar un sistema de versionado de modelos (ej. `model_v1.pkl`, `model_v2.pkl`) y un mecanismo para cambiar el modelo activo sin necesidad de reiniciar el servicio.

Pruebas Automatizadas: Desarrollar un conjunto de pruebas unitarias (para el preprocesamiento) y de integración (para el endpoint `/predict/`) para asegurar la calidad y robustez del código ante futuros cambios.

Optimización: Para un alto volumen de solicitudes, se podría explorar el uso de un servidor ASGI más robusto como `Gunicorn` con workers de `Uvicorn`.
```

6. Referencias

Documentación de FastAPI: https://fastapi.tiangolo.com/

Documentación de Pydantic: https://pydantic-docs.helpmanual.io/

Documentación de Pandas: https://pandas.pydata.org/docs/

Documentación de Scikit-learn: https://scikit-learn.org/stable/