

Pokémon TCG Price Predictor



Victor Quero Valencia

Resumen

Este proyecto tiene como objetivo desarrollar una aplicación interactiva que permita a coleccionistas, entusiastas e inversores explorar una amplia base de datos de cartas del juego Pokémon TCG y predecir sus precios futuros mediante técnicas de Machine Learning. Para ello, se diseñó un flujo de adquisición de datos que combina metadatos oficiales (nombre, set, rareza, artista, tipos) con precios históricos obtenidos inicialmente mediante bots de web-scraping (Walle, Firulai y Mapache), y finalmente consumiendo una API comunitaria que ofrece snapshots mensuales de precios. Todos los datos se almacenan y gestionan en Google BigQuery, facilitando consultas dinámicas desde la aplicación.

En el núcleo predictivo se implementó un modelo MLP (Multi-Layer Perceptron) cross-sectional, entrenado sobre dos snapshots de precios separados por 29 días y un extenso encoding de características categóricas y numéricas. Adicionalmente, se exploró un enfoque de doble pipeline con LightGBM para segmentos de precios bajos y altos, y se planteó un futuro modelo LSTM para aprovechar series temporales más densas. La aplicación se despliega en Streamlit, ofreciendo filtros, visualizaciones y predicciones bajo demanda, complementada con un dashboard en Power BI para análisis estratégico.

Keywords: Pokémon TCG, Predicción de precios, Machine Learning, Perceptrón Multicapa (MLP), LightGBM, LSTM, Web-scraping, API comunitaria, BigQuery, Streamlit, Power BI, Metadatos de cartas, Snapshots mensuales, Preprocesamiento de datos, Segmentación de precios.

Abstract

This project aims to build an interactive application for collectors, enthusiasts, and investors to explore a comprehensive Pokémon TCG card database and forecast future market prices using Machine Learning. The data pipeline merges official card metadata (name, set, rarity, artist, types) with historical price snapshots—initially scraped via custom bots (Walle, Firulai, Mapache) and later retrieved from a community API providing monthly price snapshots. All information is housed in Google BigQuery, enabling dynamic queries within the application.

At its core lies an MLP (Multi-Layer Perceptron) model trained on two price snapshots separated by 29 days, leveraging extensive one-hot encoding of categorical features and scaling of numerical inputs. A dual-pipeline LightGBM approach was also implemented for low- and high-value cards, while a future LSTM time-series model is proposed to harness richer historical data. Deployed via Streamlit, the app offers dynamic filters, visualizations, and on-demand price forecasts, complemented by a Power BI dashboard for strategic analysis.

Keywords: Pokémon TCG, Price prediction, Machine Learning, Multi-Layer Perceptron (MLP), LightGBM, LSTM, Web scraping, Community API, BigQuery, Streamlit, Power BI, Card metadata, Monthly snapshots, Data preprocessing, Price segmentation.

Índice

2. Adquisición y Preprocesamiento de Datos

- Metadatos de las Cartas
- Datos de Precios

2.1. Bot Walle: Autocompletado y Gestión de Modificadores de Nombre

- Propósito Principal
- Propósito Secundario
- Funcionamiento y Características Clave del Código
- Logro Reportado
- Limitación Reportada
- En Resumen

2.2. Bot Firulai: Búsqueda por URL Directa en Cardmarket.com

- Propósito
- Funcionamiento y Contexto del Código
- Ineficiencia Reportada (“Ineficiente debido a parámetros numéricos aleatorios en las URLs”)
- En Esencia

2.3. Bot Mapache: Búsqueda con Filtros y Desafíos del Historial de Precios

- Propósito Principal
- Propósito Secundario (Desafío)
- Funcionamiento y Contexto del Código
- Logro Reportado
- Limitación Reportada (Fallo en Acceso al Historial de Precios)
- En Síntesis

2.4. Solución Final y Estrategia de Adquisición de Datos

2.5. Estructura de la Base de Datos (Google BigQuery)

- Tablas de Precios Mensuales (`monthly_YYYY_MM_DD`)
- Tabla de Metadatos (`card_metadata`)
- Proceso de Creación

2.6. Adquisición de Datos para la Aplicación en Tiempo Real

2.7. Preprocesamiento de Datos para ML (Offline en Google Colab)

- Feature Engineering
- Transformaciones

3. Desarrollo del Modelo de Machine Learning (MLP)

3.1. Adaptación de la Estrategia: Del Doble Pipeline al MLP

3.2. Arquitectura y Configuración del Modelo MLP (`RedNeuronal.ipynb`)

- Carga de Datos
- Preprocesamiento
- Características de Entrada
- Arquitectura de la Red Neuronal
- Parámetros de Entrenamiento
- Ponderación de Muestras
- Proceso de Entrenamiento y Evaluación
- Guardado de Artefactos

3.3. Despliegue y Pruebas de Inferencia del Modelo MLP ([PruebasModelos.ipynb](#))

3.4. Visión a Futuro: Modelo LSTM y Evolución de *days_diff*

4. Doble Modelo de Pipeline: Segmentación y Predicción

4.1. Carga y Preparación de Datos

4.2. Segmentación de Datos por Umbral

4.3. Pipeline para Precios Bajos con LightGBM

4.4. Pipeline para Precios Altos con LightGBM y Afinamiento de Hiperparámetros

4.5. Función de Predicción Mixta

– Evaluación Global del Modelo Mixto

4.6. Visualización de Errores del Modelo Mixto

5. Implementación y Despliegue de la Aplicación

– Arquitectura de datos y flujo de snapshots mensuales en BigQuery

– Aplicación Streamlit

- Carga de Datos y conexión a BigQuery

- Carga del Modelo Local y preprocesadores

- Lógica de Visualización y filtros

- Sección de Detalle de Carta

- Predicción en Inferencia con el MLP

– **Integración con Power BI** para análisis estratégico

6. Trabajo Futuro

– Continuación y Expansión de la Adquisición Histórica

– Automatización Completa del Flujo de Datos

– Integración y Uso del Modelo LSTM

– Selección Dinámica del Horizonte de Predicción

– Visualizaciones Gráficas Avanzadas y de Predicción

– Implementación de un Chatbot Asistente

– Mejora Continua del Pipeline de Preprocesamiento

– Escalabilidad y Optimización del Despliegue

– Exploración de Factores de Precio Adicionales

– Segmentación Avanzada y Análisis de Cartas de Alto Valor

2. Adquisición y Preprocesamiento de Datos

El proyecto necesitaba dos tipos principales de información:

- **Metadatos de las Cartas:**
 - Nombre
 - Set (Expansión)
 - Número de carta
 - Rareza
 - Artista
 - Tipo de carta
 - Tipos elementales
 - Supertipos
 - Subtipos
 - *Fuente:* Dataset oficial de Pokémon TCG.
- **Datos de Precios:**
 - Historial de precios de mercado.

Obtener los datos históricos de precios presentó un desafío. Cardmarket, una fuente principal, limita el acceso masivo a esta información. Por ello, se intentó el *web scraping* mediante bots: "Walle", "Firulai" y "Mapache".

2.1 Bot Walle: Autocompletado y Gestión de Modificadores de Nombre

- **Propósito Principal:**
 - Mejorar la búsqueda de cartas en Cardmarket.com usando la función de autocompletado.
 - Manejar complejidades en nombres de cartas e IDs de sets para evitar filtros manuales inestables.
- **Propósito Secundario:**
 - Evitar sets de cartas exclusivamente digitales (sin mercado físico en Cardmarket).
- **Funcionamiento y Características Clave del Código:**
 - **Enfoque:** Se centra en la barra de búsqueda principal y la lista de autocompletado, no en formularios de filtro.
 - **Carga y Preprocesamiento de Datos:**
 - Carga metadatos desde el mismo CSV que otros bots.

- **Mejora Importante:** Normalización de IDs de set (ej., "sv8.5" se convierte en "sv8") para buscar con la abreviatura correcta.

```
Python

# Eliminar decimales tipo "sv8.5" → "sv8"
if "." in set_id:
    set_id = set_id.split(".")[0]
```

- Incluye una lista `invalid_sets` para excluir sets digitales (ej., "shining-revelry"), reduciendo búsquedas fallidas.
- **Detección de Modificadores en el Nombre:**
 - Enriquece el nombre de la carta para la búsqueda reconociendo prefijos/sufijos comunes (ej., "Lv.", "Mega", "Shiny").
 - Busca si el nombre original ya los contiene o si el nombre base + modificador existe en el dataset.

```
Python

modificadores = ['Lv.', 'nvl', 'Mega', 'Dark', 'Shiny']
full_name = name
for mod in modificadores:
    if mod.lower() in name.lower():
        full_name = name
        break
elif f"{name} {mod}" in df['name'].values:
    full_name = f"{name} {mod}"
```

- **Construcción de la Consulta de Búsqueda:**
 - Crea un término de búsqueda (`search_term`) muy específico: nombre completo (con modificadores) + abreviatura del set de Cardmarket + `localid` de la carta.
 - Formato (`abreviatura ID`): Crucial para que Cardmarket identifique cartas únicas en el autocompletado.
- **Interacción con Barra de Búsqueda y Autocompletado:**
 - Introduce el `search_term` en el campo de búsqueda principal.
 - Usa `WebDriverWait` para esperar resultados de autocompletado que contengan el identificador de la carta.
 - Si lo encuentra, hace clic directo en el resultado. Usa `execute_script` y `ActionChains` como respaldo.

```
Python

try:
    # ... espera y búsqueda de resultados de autocompletado ...
    for r in resultados:
        if objetivo in r.text:
            # ... clic en el resultado correcto ...
            break
except TimeoutException:
    print(f"⌚ No se encontró en autocompletado. Intentando 'Mostrar to
```

- **Fallback "Mostrar todos":**
 - Si la carta no aparece en autocompleteado, hace clic en "Mostrar todos".
 - En la página de resultados, itera por las filas de la tabla para encontrar y hacer clic en la carta.
- **Manejo de Pop-ups y Consola:**
 - Cierra banner de cookies y pop-overs informativos.
 - Incluye `print statements` descriptivos ("TRUMPBOT ha cerrado...", "TRUMPBOT va a buscar:").
- **Logro Reportado:**
 - **Mejora en Precisión de Búsqueda:** Aprovechó el autocompleteado de Cardmarket, aumentando fiabilidad y eficiencia para encontrar páginas de cartas correctas y obtener información básica (como precio actual).
- **Limitación Reportada:**
 - **Dificultad Persistente en Extracción de Datos de Gráficos:** Al igual que Mapache, no pudo extraer el historial de precios de los gráficos debido a:
 - ★ Medidas anti-bot robustas de Cardmarket.
 - ★ Datos de gráficos cargados asíncronamente o vía APIs protegidas.
 - ★ Posible ofuscación, encriptación o necesidad de autenticación de sesión avanzada.

En Resumen: Walle perfeccionó la localización de cartas imitando la interacción del usuario con el autocompleteado, pero el acceso a datos protegidos como el historial de precios siguió siendo un desafío.

2.2 Bot Firulai: Búsqueda por URL Directa en Cardmarket.com

- **Propósito:**
 - Obtener información de cartas construyendo URLs directas, sin usar formularios de búsqueda, para mayor rapidez y eficiencia.
- **Funcionamiento y Contexto del Código:**
 - **Carga y Filtrado de Datos:**
 - Lee CSV con metadatos.
 - Filtra filas sin nombre, set o `localid`.
 - Excluye sets "inválidos".
 - Para pruebas, usa una muestra de 100 cartas.
 - **Normalización de Nombres (`normalize_name`):**
 - Transforma nombres de cartas (ej., "Pikachu & Zekrom-GX") a un formato compatible con URLs (ej., "Pikachu-Zekrom-GX").

- Reemplaza acentos, puntuación y ajusta prefijos/sufijos.

```
Python

def normalize_name(name):
    # ... (reemplazos de caracteres) ...
    name = name.replace("Lv.", "Lv").replace("GX", "-GX").replace("EX",
    name = name.replace(" ", "-").replace("--", "-").strip("-")
    return name
```

- **Construcción de URLs (`urls_a_probar`):**
 - Genera múltiples URLs posibles para cada carta basadas en: nombre de set normalizado, nombre de carta normalizado y `localid`.
 - Incluye variantes con sufijos como -V1, -V2, -V3, suponiendo que Cardmarket podría usarlos para diferentes versiones.

```
Python

urls_a_probar = [
    f"https://www.cardmarket.com/es/Pokemon/Products/Singles/{set_name}",
    f"https://www.cardmarket.com/es/Pokemon/Products/Singles/{set_name}",
    f"https://www.cardmarket.com/es/Pokemon/Products/Singles/{set_name}",
    # ... y otras variantes ...
]
```

- **Acceso y Verificación:**
 - Navega a cada URL generada con Selenium.
 - Intenta cerrar pop-ups.
 - Verifica el éxito esperando un elemento `<h1>` y comprobando si su texto contiene el `local_id` y la abreviatura del set.

```
Python

abrir_link_como_humano(driver, url) # Simula una navegación humana
# ... intento de cerrar cookies y popover ...
WebDriverWait(driver, 8).until(EC.presence_of_element_located((By.XPATH
titulo = driver.find_element(By.XPATH, '//h1').text.strip()
if str(local_id_int) in titulo and abbreviation in titulo:
    print(f"✅ ENCONTRADO: {titulo}")
    aciertos += 1
    encontrado = True
    break # Si encuentra la carta, pasa a la siguiente
```

- **Simulación Humana y Métricas:**
 - Incluye pausas aleatorias para imitar comportamiento humano.
 - Registra aciertos y fallos en `firulai_log.txt`.
 - **Ineficiencia Reportada: "Ineficiente debido a parámetros numéricos aleatorios en las URLs"**
- Se refiere a la impredecibilidad de la estructura exacta de las URLs de Cardmarket.

- **Problemas:**
 - **Variantes inesperadas:** Sufijos no contemplados, orden de componentes diferente, códigos alfanuméricos únicos para promocionales o artes alternativos.
 - **Falta de `localid` en la URL:** No todas las cartas lo incluyen de forma predecible.
 - **Excesivas Peticiones:** Al no poder adivinar la URL consistentemente, realiza múltiples peticiones por carta, muchas resultando en errores 404 o páginas incorrectas, lo que ralentiza el proceso.

En Esencia: El enfoque de Firulai de construir URLs directamente fue limitado por la variabilidad en cómo Cardmarket las estructura, llevando a baja eficiencia.

2.3 Bot Mapache: Búsqueda con Filtros y Desafíos del Historial de Precios

- **Propósito Principal:**
 - Simular la interacción de un usuario humano para encontrar y obtener precios actuales de cartas usando los filtros de búsqueda del sitio.
- **Propósito Secundario (Desafío):**
 - Intentar acceder al historial de precios de cada carta.
- **Funcionamiento y Contexto del Código:**
 - **Carga y Preparación de Datos:**
 - Carga metadatos del CSV.
 - **Procesamiento de `localid` (`limpiar_numero`):** Genera diversas variantes del número de carta (ej., "1", "01", "001", "SWH001") que podrían usarse en filtros.



```
Python

def limpiar_numero(localid):
    # ... (lógica para extraer y generar variantes numéricas/alfanuméricas)
    return result, stripped
```

- Asigna `set_id` y ordena el DataFrame por un orden predefinido de sets y luego por número de carta.
- **Configuración e Interacción del Driver:**
 - Inicializa un driver de Chrome.
 - Navega a la página de búsqueda de Cardmarket ([Products/Search](#)) para usar filtros.
- **Manejo de Pop-ups y Captchas:**

- Funciones para cerrar banner de cookies, pop-overs e intentar resolver/cerrar captchas simples.
 - **Uso de Filtros de Búsqueda (Diferencia Clave con Firulai):**
 - **Selecciona la Expansión:** Encuentra el menú desplegable de expansiones y selecciona la correcta.

```
Python

select = wait.until(EC.presence_of_element_located((By.NAME, 'idExpansion'))
opciones = select.find_elements(By.TAG_NAME, 'option')
for option in opciones:
    if option.text.strip() == expansion:
        option.click()
        break
```

- **Introduce el Nombre de la Carta:** Escribe el nombre en el campo de texto correspondiente.
 - **Realiza la Búsqueda:** Hace clic en el botón "Search".
 - **Manejo de Resultados y Navegación:**
 - **Resultado Directo:** Si hay una sola coincidencia, Cardmarket redirige a la página de la carta.
 - **Lista de Resultados:** Itera sobre las filas de la tabla de resultados. Compara el número de carta extraído con las variantes generadas. Si hay coincidencia, hace clic en el enlace. Si no, intenta ir a la "Next page".
 - **Extracción de Datos de Gráficos (`extraer_datos_grafica`):**
 - Intenta obtener el historial de precios buscando un elemento `<script>` específico que contiene JSON con datos del gráfico.
 - Si tiene éxito, guarda la información en `charts_data.csv`.

```
Python   
  
def extraer_datos_grafica(driver):  
    # ... (lógica para encontrar, extraer y parsear JSON de la gráfica)  
    script_element = driver.find_element(By.XPATH, '/html/body/main/div')  
    script_text = script_element.get_attribute('innerHTML')  
    # ... (parseo de JSON y extracción de labels y data) ...  
    return [labels, data]
```

- **Simulación Humana y Logging:**
 - Mantiene pausas aleatorias.
 - Guarda resumen de aciertos/fallos y un log en `mapache_log.txt`.
 - **Logro Reportado:**
 - **Obtención de Precios Actuales:** Exitoso gracias a la interacción con la interfaz de usuario (filtros), navegando fiablemente a las páginas correctas donde los precios actuales son visibles.
 - **Limitación Reportada (Fallo en Acceso al Historial de Precios)**
 - Impedido por medidas anti-bot sofisticadas de Cardmarket.

- **Possibles Razones:**
 - **Carga Asíncrona y Ofuscación:** Datos del gráfico cargados dinámicamente o JSON ofuscado.
 - **Detección de Acceso a API Interna:** Cardmarket usa APIs internas para datos de gráficos; accesos automáticos pueden ser bloqueados o requerir autenticación avanzada.
 - **Limitación de Tasa (Rate Limiting):** Acceso frecuente a datos históricos puede activar límites.
 - **CAPTCHAs Invasivos/Detectores de Bots Avanzados:** Sistemas que detectan automatización (ej., análisis de comportamiento del ratón/teclado).

En Síntesis: Mapache mejoró la búsqueda de cartas y obtención de precios actuales al imitar la navegación humana. Sin embargo, las medidas anti-bot frustraron el acceso al historial de precios.

(Nota: El código de los 3 bots se puede consultar en sus respectivos archivos .py. También se incluye el archivo .py donde están definidas todas las equivalencias de los sets utilizadas por los bots.)

2.4 Solución Final y Estrategia de Adquisición de Datos

Finalmente, se encontró una **API comunitaria** que proporciona precios de los últimos 30 días. Esto condujo a una nueva estrategia de adquisición a largo plazo:

- Un **script manual** se ejecutará a principios de cada mes.
 - Este script almacenará un **snapshot mensual** de precios.
-

2.5 Estructura de la Base de Datos (Google BigQuery)

La base de datos principal es Google BigQuery, con las siguientes tablas:

1. **Tablas de Precios Mensuales (monthly_YYYY_MM_DD):**
 - Snapshots periódicos del precio de mercado (`cm_averageSellPrice`).
 - Para cada carta (`id`).
 - Con una fecha específica.
2. **Tabla de Metadatos (card_metadata):**
 - Información descriptiva de cada carta:
 - `id, name, artist, rarity, set_name, types, supertype, subtypes`, etc.

<!-- end list -->

- **Proceso de Creación:**
 - Detallado en el notebook: [API Crear base de datos.ipynb](#) (extracción de datos de la API).
 - Subida de datos locales a Google BigQuery gestionada y consultable en: [API con bigquery.ipynb](#).
-

2.6 Adquisición de Datos para la Aplicación en Tiempo Real

Implica los siguientes pasos:

1. **Conexión segura a BigQuery:** Usando el cliente Python oficial y autenticación vía Streamlit Secrets.
 2. **Consulta dinámica:** A la tabla de precios más reciente y a la tabla `card_metadata`.
 3. **Unión (merge):** De los datos de precios con los metadatos en memoria.
-

2.7 Preprocesamiento de Datos para Machine Learning (Offline en Google Colab)

Incluyó:

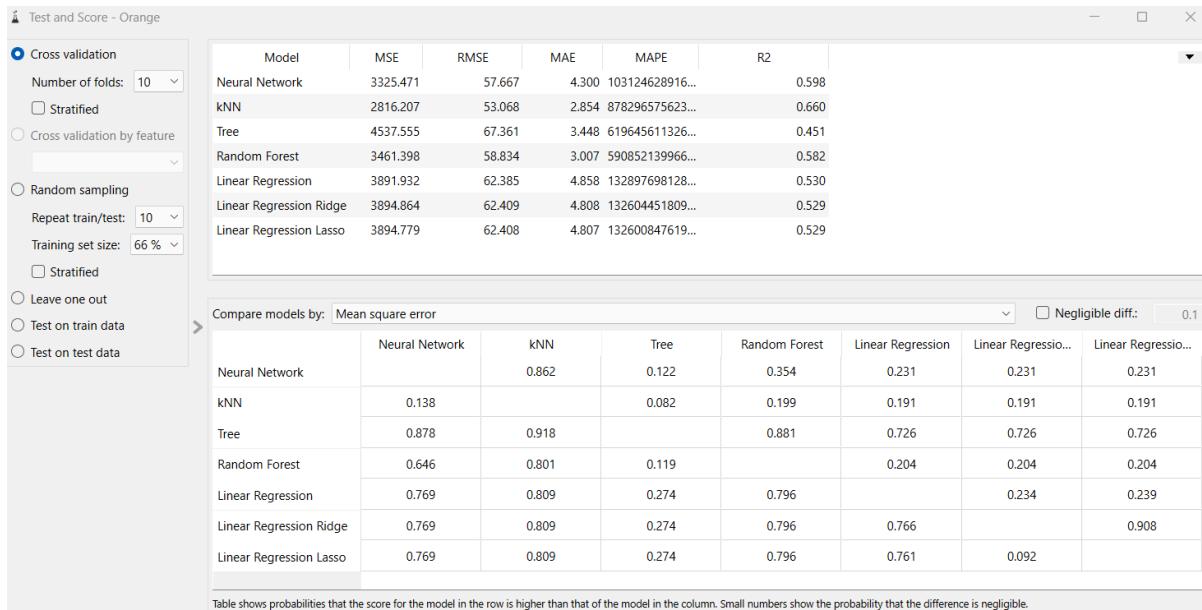
- **Feature Engineering:**
 - Logaritmo del precio actual: `price_t0_log = log(1 + price_t0)`.
 - Diferencia en días entre snapshots: `days_diff` (constante de 29.0 para el MLP).
- **Transformaciones:**
 - `StandardScaler` para características numéricas.
 - `OneHotEncoder` para características categóricas (`artist_name`, `pokemon_name`, `rarity`, `set_name`, `types`, `supertype`, `subtypes`).
 - El `OneHotEncoder` se entrenó con los valores exactos de la base de datos.

3. Desarrollo del Modelo de Machine Learning (MLP)

El corazón de la funcionalidad predictiva del proyecto reside en el modelo de Machine Learning, cuya evolución y diseño están detallados en el notebook [RedNeuronal.ipynb](#). Aunque inicialmente se exploró un enfoque de "doble pipeline" (como el discutido en [DobleModelo \(2\).ipynb](#)), las limitaciones de datos dirigieron el proyecto hacia un modelo **MLP (Multi-Layer Perceptron) Cross-Sectional**.

3.1. Adaptación de la Estrategia del Modelo: Del Doble Pipeline al MLP

La estrategia inicial de modelado, influenciada por herramientas como Orange Data Mining,



sugería una segmentación del mercado de cartas Pokémon TCG. Se observó que las cartas con precios significativamente diferentes (especialmente por encima y por debajo de 50€) parecían seguir patrones de comportamiento distintos. Esta observación motivó la idea de entrenar dos modelos separados, uno para cartas de bajo valor y otro para las de alto valor. Las pruebas iniciales con este enfoque dual, documentadas en [DobleModelo \(2\).ipynb](#), mostraron un potencial prometedor para lograr una mayor precisión en rangos de precios específicos.

Sin embargo, este camino se encontró con una limitación crítica: la **escasez de datos históricos extensos**. En el momento del desarrollo, el proyecto solo disponía de información de precios de los últimos 30 días, obtenida de una API comunitaria. Esto se traducía en un historial de precios muy limitado, esencialmente dos *snapshots*: el precio actual y el de aproximadamente 29-30 días antes. Esta restricción temporal impidió que el enfoque de dos modelos separados fuera lo suficientemente robusto para predecir un horizonte futuro variable o para capturar tendencias a largo plazo. Un modelo que intentara segmentar el mercado con tan pocos puntos de datos temporales podría fácilmente sobreajustarse o carecer de la generalización necesaria.

Dada esta limitación, la estrategia se reorientó. Se decidió implementar un modelo que pudiera aprovechar al máximo la información de esos dos *snapshots* para una predicción a un **horizonte fijo y cercano**. Esto llevó a la implementación del modelo **MLP (Multi-Layer Perceptron) Cross-Sectional**, cuyo objetivo es predecir el precio futuro basándose en un único *snapshot* de precio actual (`price_t0`) y los metadatos de la carta. La característica clave para la predicción, `days_diff` (la diferencia de días entre el precio actual y el futuro a predecir), fue fijada en **29.0 días** en el entrenamiento del MLP en [RedNeuronal.ipynb](#).

Esta decisión no es arbitraria; se deriva directamente de la forma en que se adquieren los datos de la API (precios de los últimos 30 días), permitiendo al modelo aprender la relación entre el precio actual y el precio esperado aproximadamente un mes después, dada la periodicidad de los *snapshots* disponibles. Es, por tanto, una predicción del "siguiente punto" disponible en el tiempo, un enfoque pragmático ante la falta de una serie temporal más densa.

3.2. Arquitectura y Configuración del Modelo MLP ([RedNeuronal.ipynb](#))

El modelo MLP implementado en [RedNeuronal.ipynb](#) presenta una arquitectura y configuración específicas, diseñadas para extraer el máximo valor de los datos disponibles.

Carga de Datos: Los datos se cargan desde BigQuery utilizando las siguientes configuraciones:

- PROJECT_ID: "pokemon-cards-project"
- BQ_DATASET: "pokemon_dataset"
- BQ_TABLE_META: "card_metadata"

Las consultas SQL utilizadas para obtener los precios y metadatos son las siguientes:

- **Precios:**

```
SQL
SELECT
    PARSE_DATE('%Y_%m_%d', _TABLE_SUFFIX) AS fecha,
    id AS card_id,
    cm_averageSellPrice AS precio
FROM `pokemon-cards-project.pokemon_dataset.monthly_*`
WHERE _TABLE_SUFFIX BETWEEN '2020_01_01' AND FORMAT_DATE('%Y_%m_%d', CURRENT_DATE)
ORDER BY card_id, fecha
```

- **Metadatos:**

```
SQL
SELECT
    id AS card_id,
    artist AS artist_name,
    name AS pokemon_name,
    rarity,
    set_name,
    types,
    supertype,
    subtypes
FROM `pokemon-cards-project.pokemon_dataset.card_metadata`
```

Se utiliza un diccionario `release_dates` para añadir la fecha de lanzamiento de cada set a los metadatos.

Preprocesamiento: Una vez cargados y fusionados los datos de precios y metadatos, se realiza una imputación contextual de los precios faltantes en tres niveles (por `rarity` y `set_name`, por `pokemon_name`, y por `artist_name`), y finalmente una imputación global con la mediana.

El código detecta que hay 2 fechas únicas (`[Timestamp('2025-04-01 00:00:00'), Timestamp('2025-04-30 00:00:00')]`), lo que activa el Pipeline A: MLP.

- **Características de Entrada:** El modelo recibe un vector concatenado de 4865 características. Esto incluye:
 - 2 características numéricas escaladas: `price_t0_log` (el logaritmo natural del precio actual más uno, para manejar la asimetría de precios y distribuciones sesgadas) y `days_diff` (fijado en `29.0`, calculado como la diferencia de días entre las dos fechas disponibles). La transformación numérica se realiza con `StandardScaler`.
 - 4863 características categóricas codificadas con One-Hot Encoding: Proviene de variables como `artist_name`, `pokemon_name`, `rarity`, `set_name`, `types`, `supertype`, y `subtypes`. La lista de columnas categóricas utilizadas es `cat_cols_meta = ["artist_name", "pokemon_name", "rarity", "set_name", "types", "supertype", "subtypes"]`. El `OneHotEncoder` fue entrenado con el conjunto completo de la base de datos para asegurar el manejo de todas las categorías posibles.
- **Arquitectura de la Red Neuronal:** El modelo es una red secuencial construida con `tf.keras.Sequential`. La configuración de las capas es la siguiente:
 - Primera Capa Densa: `layers.Dense` con 16 neuronas y activación ReLU. Inmediatamente después, se aplica una capa `Dropout` con una tasa de 0.3.
 - Segunda Capa Densa: Otra capa `layers.Dense` con 16 neuronas y activación ReLU, seguida nuevamente por una capa `Dropout` con tasa de 0.3.
 - Capa de Salida: Una única neurona con activación lineal.

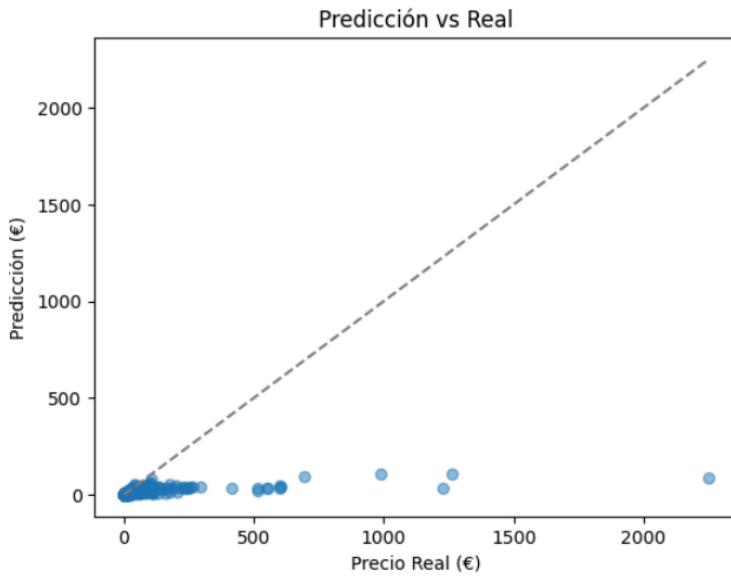
- El `model.summary()` muestra un total de 78,145 parámetros entrenables.

Model: "functional"		
Layer (type)	Output Shape	Param #
features (InputLayer)	(None, 4865)	0
dense (Dense)	(None, 16)	77,856
dropout (Dropout)	(None, 16)	0
dense_1 (Dense)	(None, 16)	272
dropout_1 (Dropout)	(None, 16)	0
dense_2 (Dense)	(None, 1)	17

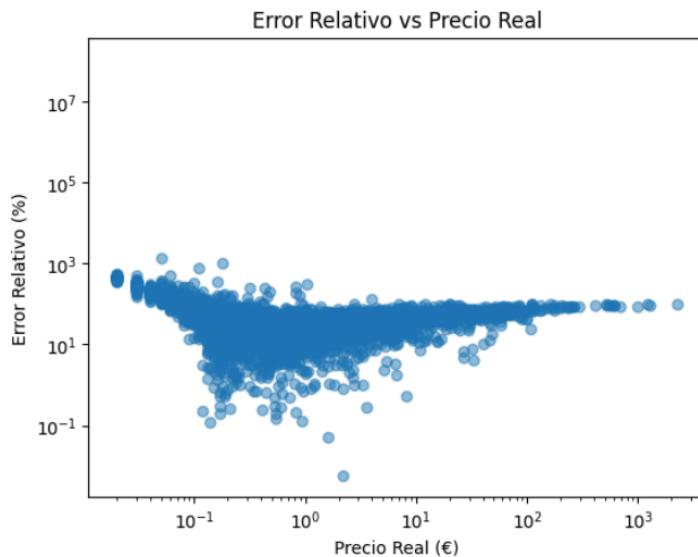
Total params: 78,145 (305.25 KB)
Trainable params: 78,145 (305.25 KB)
Non-trainable params: 0 (0.00 B)

- **Parámetros de Entrenamiento:**

- Variable Objetivo (y): Se entrenó el modelo para predecir el logaritmo transformado del precio futuro (`price_t1_log = np.log1p(df_wide["price_t1"])`).
- Función de Pérdida: Se empleó la Huber loss (`tf.keras.losses.Huber`) con `delta=1.0`.
- Optimizador: Se utilizó el optimizador Adam (`tf.keras.optimizers.Adam`).
- Métricas de Evaluación: Durante el proceso de entrenamiento, se monitorearon el MAE (Mean Absolute Error) y la `Huber Loss`. La compilación del modelo se realiza con `model.compile(optimizer="adam", loss=Huber(delta=1.0), metrics=["mae"])`.
- **Ponderación de Muestras (Sample Weighting):** Se aplicó un `sample_weight` de 2.0 a las cartas con precios superiores a 50€. El cálculo de los pesos se hizo como `weights = 1.0 / (1.0 + df_wide["price_t1"].values)`, lo que significa que a menor `price_t1`, mayor peso. Esto se hace para dar mayor importancia a la predicción de cartas de alto valor.
- **Proceso de Entrenamiento y Evaluación:** El modelo fue entrenado en Google Colab con 50 épocas y un `batch_size` de 32. La evolución de las métricas durante el entrenamiento es fundamental para evaluar el rendimiento. Se pueden observar las curvas de pérdida y MAE tanto para el conjunto de entrenamiento como para el de validación.



La evaluación final en el conjunto de validación mostró un Val MAE (€): 5.1825. Las métricas de validación en la escala logarítmica fueron: Val MAE: 0.1969 y Val RMSE: 0.3341.



- **Guardado de Artefactos:** Una vez entrenado, el modelo y los preprocesadores se guardaron:
 - El modelo TensorFlow se exportó en formato `.keras` como `"mlp_price_forecast.keras"`.
 - El `StandardScaler` numérico se serializó usando `joblib` como `"scaler_mlp_num.pkl"`.
 - El `OneHotEncoder` categórico se serializó usando `joblib` como `"ohe_mlp_cat.pkl"`.

3.3. Despliegue y Pruebas de Inferencia del Modelo MLP (`PruebasModelos.ipynb`)

Una vez que el modelo MLP fue entrenado y sus artefactos (el modelo guardado y los preprocesadores) fueron exportados desde `RedNeuronal.ipynb`, el siguiente paso crítico fue validar su operacionalización y capacidad para realizar inferencias en un entorno similar al de producción. Este proceso se llevó a cabo en la notebook `PruebasModelos.ipynb`, la cual se diseñó específicamente para cargar el modelo y sus componentes auxiliares, y luego simular predicciones sobre nuevos datos de entrada.

Los objetivos principales de esta notebook fueron:

- **Verificación de la Carga de Artefactos:** Se comprobó que el modelo MLP, guardado en formato TensorFlow `SavedModel`, pudiera ser cargado exitosamente usando `tf.keras.layers.TFSMLayer`. Este es un formato robusto que facilita el despliegue. Adicionalmente, los preprocesadores clave (`OneHotEncoder` para características categóricas y `StandardScaler` para numéricas) se cargaron mediante `joblib`. La correcta carga de estos elementos es fundamental, ya que aseguran que los datos de entrada para la inferencia se transformen exactamente de la misma manera que se hizo durante el entrenamiento, manteniendo la consistencia del *pipeline*.
- **Implementación del Pipeline de Inferencia:** La notebook encapsula la lógica de predicción en una función dedicada (`predict_price_with_local_tf_layer`). Esta función es crucial porque replica todo el proceso de transformación de datos de entrada:
 - Toma los metadatos de una carta y su precio actual.
 - Aplica la transformación logarítmica (`np.log1p`) al precio actual, creando `price_t0_log`.
 - Confirma que la característica `days_diff` se fija en 29.0 días, el mismo valor utilizado en el entrenamiento del MLP.
 - Utiliza los `StandardScaler` y `OneHotEncoder` cargados para escalar las características numéricas y codificar las categóricas, respectivamente.
 - Concatena todas las características preprocesadas en un único vector de 4865 entradas, listo para ser consumido por el modelo.
 - Finalmente, la predicción cruda del modelo (un valor logarítmico) se post-procesa aplicando la inversa de la transformación logarítmica (`np.expm1`) para obtener el precio predicho en la escala de euros original y comprensible.
- **Realización de Pruebas Aleatorias:** Para validar la función de predicción, se seleccionaron y procesaron 10 cartas aleatorias. A estas cartas se les asignó un precio actual simulado (dentro de un rango realista), lo que permitió probar el *pipeline* completo sin depender de datos de mercado en vivo durante la fase de desarrollo. Los resultados de estas pruebas se imprimieron en la consola, mostrando

el nombre de cada carta, su precio actual simulado, el precio predicho por el modelo y la diferencia (delta) entre ambos. Esto confirmó que el modelo cargado, junto con sus preprocesadores, operaba según lo esperado y podría generar predicciones

En esencia, `PruebasModelos.ipynb` sirvió como un banco de pruebas vital para asegurar que el modelo entrenado y sus componentes auxiliares estuvieran listos para ser integrados en la aplicación final, validando la integridad del *pipeline* de inferencia.

3.4. Visión a Futuro: El Modelo LSTM y la Evolución de `days_diff`

Aunque el MLP Cross-Sectional es la solución actual y efectiva dadas las limitaciones de datos, la visión a futuro del proyecto incluye la implementación de un **Pipeline B: LSTM (Long Short-Term Memory) Time-Series**.

La transición a un modelo LSTM se fundamenta en la expectativa de acumular un historial de precios más profundo y denso en Google BigQuery, gracias a la ejecución mensual continua del *script* de adquisición de datos. Una vez que se disponga de una serie temporal más extensa para cada carta, el LSTM será ideal para:

- **Capturar Dependencias Temporales:** Los LSTMs son arquitecturas de redes neuronales recurrentes (RNN) diseñadas específicamente para modelar secuencias de datos. Pueden aprender y recordar patrones de larga duración, lo que es esencial para predecir la evolución de precios en el tiempo.
- **Identificar Tendencias y Estacionalidades:** Con más datos históricos, el LSTM podrá detectar tendencias alcistas o bajistas, así como posibles estacionalidades o patrones cíclicos en los precios de las cartas.
- **Predicción de Horizontes Variables:** A diferencia del `days_diff` fijo del MLP, un modelo LSTM permitiría que la característica `days_diff` tenga una fórmula dinámica o se convierta en una variable de entrada que represente el horizonte de predicción deseado por el usuario. Por ejemplo, el usuario podría querer saber el precio de una carta en 7, 30, 90 o incluso 180 días. Un LSTM, al operar sobre secuencias, puede adaptarse a la predicción de un valor futuro en un punto determinado de una secuencia temporal, permitiendo una mayor flexibilidad y utilidad para el usuario final.

La implementación del LSTM representa el siguiente paso lógico y la evolución natural del proyecto, aprovechando el crecimiento de la base de datos histórica para ofrecer predicciones más sofisticadas y versátiles.

4. Doble Modelo de Pipeline: Segmentación y Predicción

Para optimizar la precisión de las predicciones, se implementó una estrategia de doble modelo, segmentando el conjunto de datos de cartas en dos grupos: "precios bajos" y

"precios altos" basados en un umbral de 30€. Esta aproximación permite entrenar modelos especializados que capturan mejor las características intrínsecas de cada segmento.

El proceso general se divide en las siguientes etapas, desde la carga de datos hasta la evaluación de los modelos segmentados:

4.1 Carga y Preparación de Datos

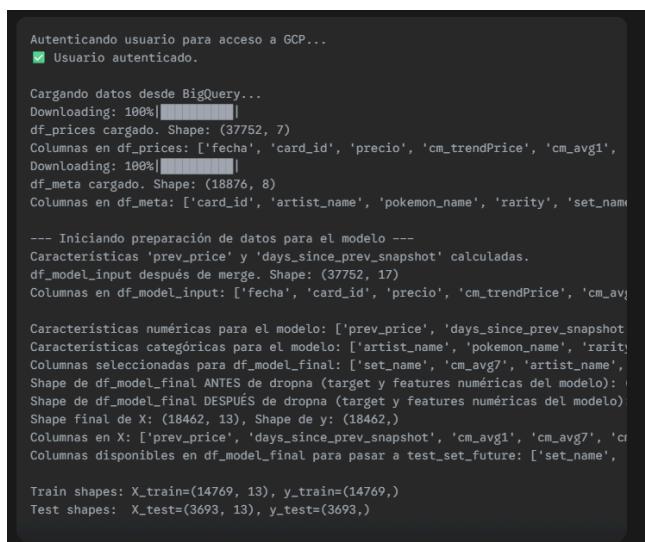
Inicialmente, se montó Google Drive para acceder a los archivos y se autenticó el usuario para conectar con Google Cloud Platform. Los datos de precios (últimos 30 días) y metadatos de las cartas se cargaron desde BigQuery. Se verificó la existencia de las columnas necesarias para el modelo, como `cm_trendPrice`, `cm_avg1`, `cm_avg7`, `cm_avg30`.

Se realizó un preprocesamiento de los datos, incluyendo:

- Conversión de la columna `fecha` a formato `datetime`.
- Cálculo de `prev_price` (precio del snapshot anterior) y `days_since_prev_snapshot` (días desde el snapshot anterior).
- Fusión de los datos de precios con los metadatos de las cartas (`artist_name`, `pokemon_name`, `rarity`, `set_name`, `types`, `supertype`, `subtypes`).

Se definieron las características numéricas y categóricas que el modelo usaría. Las columnas categóricas se procesaron para asegurar que fueran de tipo `category` y se manejaron valores nulos o formatos inconsistentes (especialmente en `types` y `subtypes`). Finalmente, se eliminaron las filas con valores `NaN` en el `target` o en las características numéricas esenciales, y el conjunto de datos se dividió en entrenamiento y prueba (`X_train`, `X_test`, `y_train`, `y_test`).

Código de la Celda 1 (Carga y Preparación de Datos):



```
Autenticando usuario para acceso a GCP...
✓ Usuario autenticado.

Cargando datos desde BigQuery...
Downloading: 100%|██████████|
df_prices cargado. Shape: (37752, 7)
Columnas en df_prices: ['fecha', 'card_id', 'precio', 'cm_trendPrice', 'cm_avg1',
Downloaded: 100%|██████████|
df_meta cargado. Shape: (18876, 8)
Columnas en df_meta: ['card_id', 'artist_name', 'pokemon_name', 'rarity', 'set_name', 'types', 'supertype', 'subtypes']

--- Iniciando preparación de datos para el modelo ---
Características 'prev_price' y 'days_since_prev_snapshot' calculadas.
df_model_input después de merge. Shape: (37752, 17)
Columnas en df_model_input: ['fecha', 'card_id', 'precio', 'cm_trendPrice', 'cm_avg1', 'cm_avg7', 'cm_avg30', 'rarity', 'set_name', 'supertype', 'subtypes', 'prev_price', 'days_since_prev_snapshot']

Características numéricas para el modelo: ['prev_price', 'days_since_prev_snapshot']
Características categóricas para el modelo: ['artist_name', 'pokemon_name', 'rarity', 'set_name', 'supertype', 'subtypes']
Columnas seleccionadas para df_model_final: ['set_name', 'cm_avg7', 'artist_name', 'rarity', 'supertype', 'subtypes']
Shape de df_model_final ANTES de dropna (target y features numéricas del modelo): (37752, 13)
Shape de df_model_final DESPUÉS de dropna (target y features numéricas del modelo): (18462, 13)
Shape final de X: (18462, 13), Shape de y: (18462,)
Columnas en X: ['prev_price', 'days_since_prev_snapshot', 'cm_avg1', 'cm_avg7', 'cm_avg30', 'rarity', 'set_name', 'supertype', 'subtypes']
Columnas disponibles en df_model_final para pasar a test_set_future: ['set_name', 'supertype', 'subtypes']

Train shapes: X_train=(14769, 13), y_train=(14769,)
Test shapes: X_test=(3693, 13), y_test=(3693,)
```

4.2 Segmentación de Datos por Umbral

Se fijó un umbral de 30.0 € para la columna `precio` (`y`). Los datos se dividieron en dos conjuntos: `mask_low` para precios menores a 30 € y `mask_high` para precios iguales o mayores a 30 €. Cada conjunto se dividió a su vez en entrenamiento y prueba. Se definió un `ColumnTransformer` (`preprocessor`) que aplicaría `OneHotEncoder` a las características categóricas, ignorando categorías desconocidas, y pasaría las numéricas sin transformar (`remainder='passthrough'`).

Código de la Celda 2 (Segmentación de Datos):

```
Python

threshold = 30.0
mask_low  = y < threshold
mask_high = y >= threshold
# ... (resto del código de la Celda 2) ...
preprocessor = ColumnTransformer([
    ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_features_for_model),
], remainder='passthrough')
```

4.3 Pipeline para Precios Bajos con LightGBM

Para el segmento de precios bajos, se construyó un `Pipeline` que primero aplica el `preprocessor` definido (para las características categóricas) y luego un `LGBMRegressor`. Se configuró `LightGBM` con `objective='mae'` (Error Absoluto Medio) para optimizar esta métrica, `n_estimators=100`, y se silenciaron los mensajes de advertencia y los logs de entrenamiento. El modelo se entrenó con los precios transformados a escala logarítmica (`np.log1p(y_tr_low)`), y las predicciones se volvieron a la escala original (`np.expm1`).

Código de la Celda 3 (Pipeline Precios Bajos):

```
import warnings
warnings.filterwarnings(
    "ignore",
    message=".*force_all_finite.*",
    category=FutureWarning
)
# ... (resto del código de la Celda 3) ...
pipe_low = Pipeline([
    ('preprocessor', preprocessor),
    ('regressor', lgb.LGBMRegressor(
        objective='mae',
        random_state=42,
        n_estimators=100,
        force_row_wise=True,
        verbosity=-1
    ))
])
y_tr_low_log = np.log1p(y_tr_low)
pipe_low.fit(X_tr_low, y_tr_low_log)
y_log_pred_low = pipe_low.predict(X_te_low)
y_pred_low      = np.expm1(y_log_pred_low)
rmse_low = np.sqrt(mean_squared_error(y_te_low, y_pred_low))
mae_low  = mean_absolute_error(y_te_low, y_pred_low)
print(f"Low-price → RMSE: {rmse_low:.3f} €, MAE: {mae_low:.3f} €")
```

Salida de la Celda 3:

```
Low-price → RMSE: 1.261 €, MAE: 0.379 €
```

4.4 Pipeline para Precios Altos con LightGBM y Afinamiento (Hyperparameter Tuning)

Para el segmento de precios altos, se definió un `preprocessor_high` más complejo que incluye `StandardScaler` para las características numéricas y `OneHotEncoder` para las categóricas. Se creó un `Pipeline` similar (`pipe_high`) con un `LGBMRegressor` (`n_estimators=200`). Este modelo también se entrenó en escala logarítmica.

Además, se realizó un afinamiento de hiperparámetros (`RandomizedSearchCV`) para `pipe_high` para encontrar la mejor combinación de `n_estimators`, `max_depth`, `num_leaves` y `min_child_samples` que minimizara el Error Absoluto Medio (MAE).

Código de la Celda 4 (Pipeline Precios Altos y Afinamiento):

```
warnings.filterwarnings("ignore", category=ConvergenceWarning)
warnings.filterwarnings("ignore", message=".*force_all_finite.*")

preprocessor_high = ColumnTransformer([
    ('num', StandardScaler(), numeric_features_for_model),
    ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_features_for_model)
])
# ... (resto del código de la Celda 4) ...
pipe_high.fit(X_tr_high, y_tr_high_log)
y_log_pred_high = pipe_high.predict(X_te_high)
y_pred_high      = np.expm1(y_log_pred_high)
rmse_high        = np.sqrt(mean_squared_error(y_te_high, y_pred_high))
mae_high         = mean_absolute_error(y_te_high, y_pred_high)
print(f"High-price → RMSE: {rmse_high:.3f} €, MAE: {mae_high:.3f} €")

# Afinamiento de hiperparámetros
param_dist_high = {
    'regressor__n_estimators': randint(50, 500),
    'regressor__max_depth': [None, 10, 20, 30],
    'regressor__num_leaves': randint(20, 200),
    'regressor__min_child_samples': randint(5, 50)
}
search_high = RandomizedSearchCV(
    pipe_high,
    param_distributions=param_dist_high,
    n_iter=10,
    cv=3,
    scoring='neg_mean_absolute_error',
    random_state=42,
    n_jobs=-1,
    verbose=2
)
search_high.fit(X_tr_high, np.log1p(y_tr_high))
best_high = search_high.best_estimator_
print("High-price best params:", search_high.best_params_)
```

Salida de la Celda 4 (Pipeline Precios Altos y Afinamiento):

```
High-price → RMSE: 223.286 €, MAE: 51.449 €  
Fitting 3 folds for each of 10 candidates, totalling 30 fits  
High-price best params: {'regressor__max_depth': 20, 'regressor__min_child_samples':
```

La evaluación del modelo de precios altos afinado mostró una ligera mejora en el MAE.

4.5 Función de Predicción Mixta

Se definió una función `predict_mixed` que toma un DataFrame de entrada, las columnas de características esperadas por los *pipelines*, y el nombre y valor de la columna de umbral (por defecto `cm_avg7` y 30.0 €). Esta función segmenta el DataFrame de entrada según el umbral y aplica el `pipe_low` o `pipe_high` (o `best_high` si se usó el afinado) apropiado para cada segmento. Las predicciones logarítmicas se transforman de nuevo a euros antes de ser devueltas.

Código de la Celda 6 (Función `predict_mixed`):

```
Python  
  
def predict_mixed(  
    X_data_with_threshold_col: pd.DataFrame,  
    feature_cols_for_pipeline: list,  
    threshold_col_name: str = 'cm_avg7',  
    threshold_value: float = 30.0  
) -> pd.Series:  
    # ... (resto del código de la función predict_mixed) ...
```

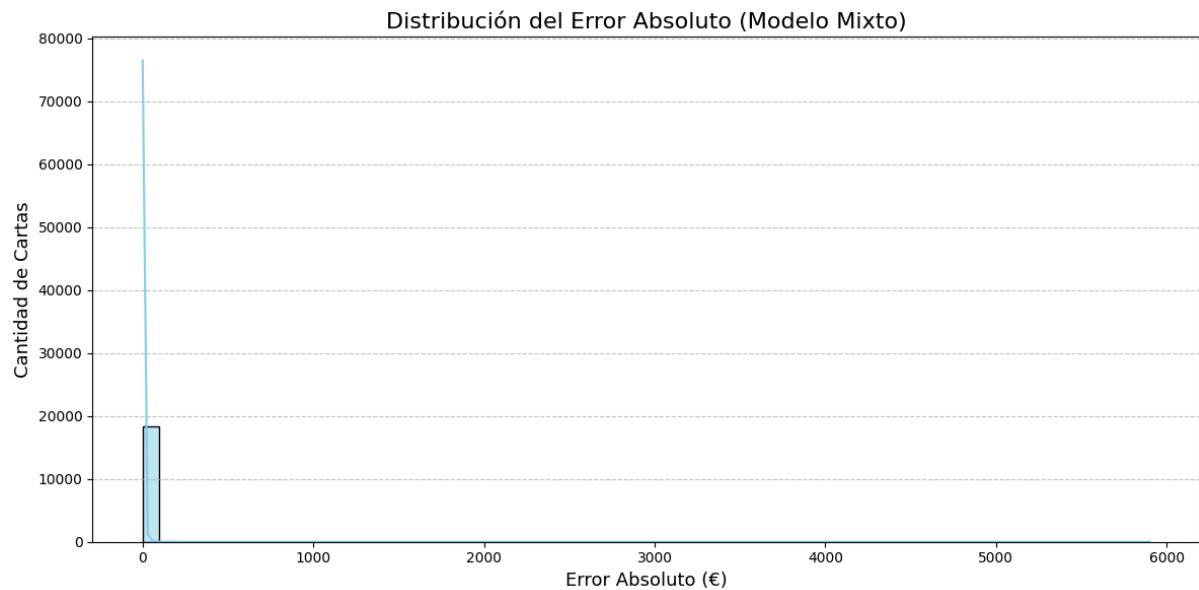
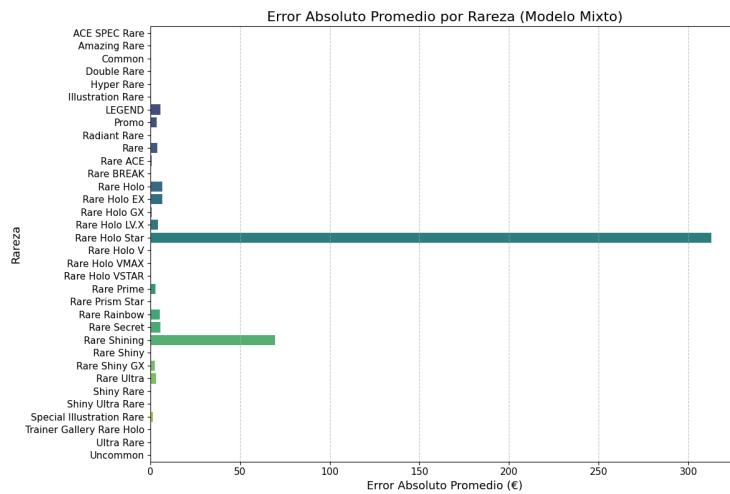
4.5 Evaluación Global del Modelo Mixto

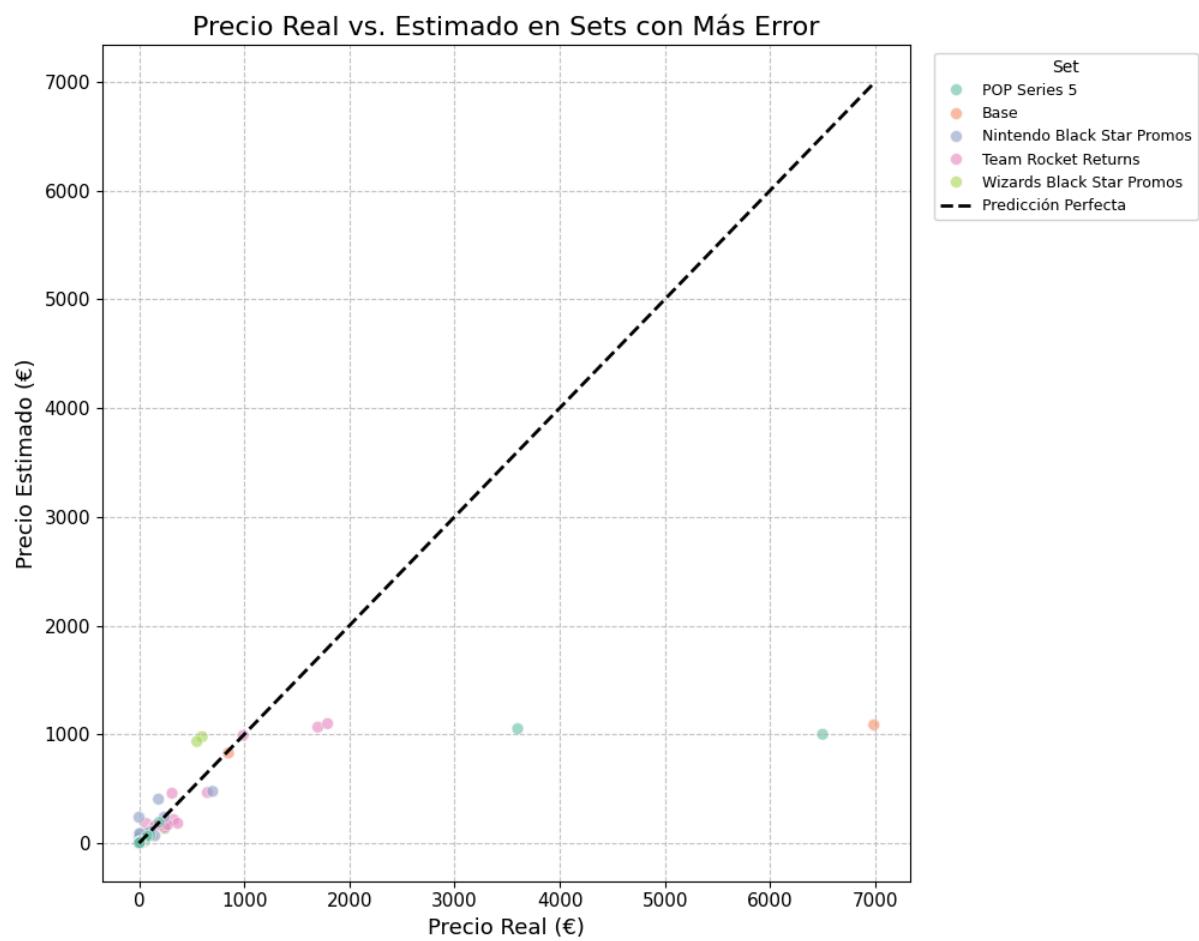
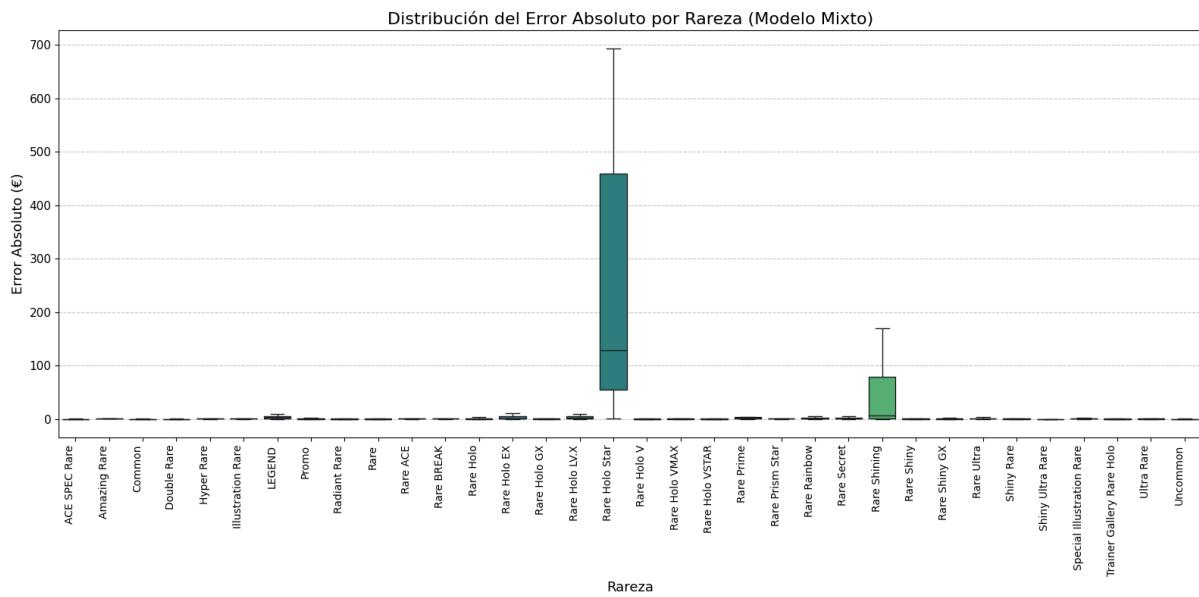
Finalmente, se llamó a la función `predict_mixed` con el conjunto de datos completo (X) y se evaluó el rendimiento global del modelo mixto. Esto permite obtener una métrica combinada de cómo los dos modelos segmentados funcionan en conjunto.

Los resultados muestran que el modelo mixto (global) alcanza un RMSE de **65.88 €** y un MAE de **2.39 €**.

4.6 Visualización de Errores del Modelo Mixto

Para comprender mejor el rendimiento del modelo, se generaron gráficos que muestran el error absoluto promedio por rareza y los 15 sets con mayor error absoluto promedio. Se creó un DataFrame `df_resultado` que combina los precios reales y las predicciones, y calcula las diferencias, errores absolutos y errores porcentuales.

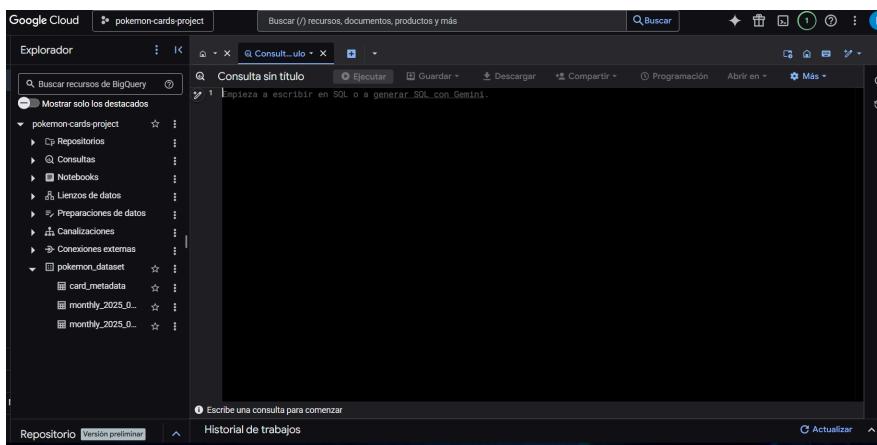




5. Implementación y Despliegue de la Aplicación

La **arquitectura de datos** facilita la ingestión periódica y el acceso eficiente. Los *snapshots* mensuales de precios obtenidos de la API siguen un flujo automatizado:

- Un *script* o bot consume la API mensualmente y descarga los datos de precios (últimos 30 días).
- Estos archivos se almacenan temporalmente en Google Drive y localmente.
- Un segundo proceso automatizado monitorea Google Drive para ingerir los nuevos archivos en **Google BigQuery**. Cada *snapshot* mensual se carga en una tabla separada (`monthly_YYYY_MM_DD`), manteniendo un registro histórico.



Google BigQuery se eligió por su escalabilidad, integración con Google Cloud Platform y su interfaz de consulta SQL, permitiendo la extracción eficiente de datos para:

- Servir datos a la aplicación interactiva Streamlit.
- Facilitar el análisis exploratorio y la preparación de datos para el entrenamiento de modelos (offline).
- Permitir la conexión con herramientas de Business Intelligence como Power BI.

La aplicación interactiva, desarrollada con **Streamlit** y desplegada en **Streamlit Cloud** desde **GitHub**, interactúa con esta infraestructura así:

```

Pokemon_TCG_Price_Predictor/
├── app.py                      # Aplicación principal (Streamlit)
├── requirements.txt              # Dependencias
└── model_files/                 # Modelo ML + Preprocesadores
    ├── saved_model.pb
    ├── variables/
    ├── ohe_mlp_cat.pkl
    └── scaler_mlp_num.pkl

```

Carga de Datos para la App: Al inicio y al cambiar filtros, la aplicación consulta Google BigQuery (vía Streamlit Secrets) para obtener metadatos y los precios más recientes. Los datos de precios se unen (*merge*) con los metadatos en memoria.

Carga del Modelo Local: El modelo MLP ([SavedModel](#)) y sus preprocesadores ([.pkl](#)) se incluyen en el repositorio de GitHub ([model_files/](#)). Se cargan en memoria al inicio de la aplicación usando `@st.cache_resource` y `tf.keras.layers.TFSMLayer` para Keras 3.

Lógica de Visualización:

link app: <https://apppokemon.streamlit.app/>

- **Carga Inicial (sin filtros):** Muestra una selección aleatoria de cartas destacadas (por rareza 'Special Illustration Rare') con imágenes grandes y el set en el pie de foto. La tabla principal de resultados se oculta. Una carta aleatoria con precio se selecciona automáticamente para mostrar sus detalles.



- **Al Aplicar Filtros:** La sección de cartas destacadas se oculta. La tabla principal de resultados ("Resultados de Cartas") se muestra usando `st-aggrid`, permitiendo interacción, paginación, búsqueda y ordenación.

The screenshot shows the 'Explorador de Cartas Pokémon TCG' interface. On the left, there's a sidebar titled 'Filtros y Opciones' with dropdown menus for 'Categoría' (set to 'Pokémon'), 'Set(s)' (set to 'Choose an option'), 'Pokéname (Nombre Base)' (set to 'Choose an option'), 'Rareza(s)' (set to 'Choose an option'), and 'Ordenar por Precio (Trend)': 'Ascendente'. Below these are sections for 'Pokéname' and 'Set'. At the bottom of the sidebar, it says 'Pokéname TCG Explorer v1.22 | MLP & LGBM'. The main area is titled 'Resultados de Cartas' and contains a table with 200 results. The columns are: ID, Nombre Carta, Categoría, Set, Rareza, Artista, and Precio (€). The first few rows include: 'base1-4 Charizard', 'pp05-16 Espeon', 'pp05-17 Umbreon', 'ex11-111 Groudon', 'es8-107 Rayquaza', 'ex7-108 Torchic', 'ex7-107 Mudkip', 'sm9-170 Latios & Latias-GX', 'ex13-103 Meowth', 'sm97-215 Unbroken VMAX', 'ex11-113 Metagross', 'ex11-112 Kyurem', 'ex7-109 Troxico', 'ex15-100 Charizard + 5', and 'base1-15 Venusaur'. The table has a scroll bar on the right. At the bottom right of the table area, it says 'Page Size: 1 to 25 of 200' and 'Page 1 of 8'.

- Sección de Detalle de Carta:** Un panel fijo visible cuando una carta está seleccionada. Muestra la imagen ampliada, metadatos clave, precio actual y enlaces externos a plataformas de venta como Cardmarket.
- Predicción en Inferencia:** Al seleccionar una carta con precio registrado y con el modelo MLP cargado:
 - Se extraen datos relevantes (`price`, metadatos) de la carta seleccionada.
 - Se prepara la entrada para el modelo replicando el preprocessamiento *offline*: `price_t0_log = log(1 + price_actual)`, `days_diff` constante (29.0), y mapeo de categóricas a `OneHotEncoder` (manejando nulos y formatos).
 - Se aplican `scaler_local_preprocessor` y `ohe_local_preprocessor`.
 - Se concatenan las características preprocesadas en el orden correcto (forma `(1, 4865)`).
 - Se realiza la predicción con `model_layer(**input_dict)`.
 - La predicción cruda (escala logarítmica) se post-procesa con `np.expm1()` para obtener el precio estimado en euros.
 - La predicción se muestra en la interfaz de usuario, usualmente con `st.metric`.

This screenshot shows the 'Detalle de Carta Seleccionada' (Selected Card Detail) page for the card 'Brock's Primeape'. The card itself is shown on the left with its artwork, stats (70 HP), and abilities. To the right, detailed information is provided:

- Brock's Primeape**
- ID:** gym2-35
- Categoría:** Pokéname
- Set:** Gym Challenge
- Rareza:** Uncommon
- Artista:** Ken Sugimori
- Precio Actual (€):** €3.28

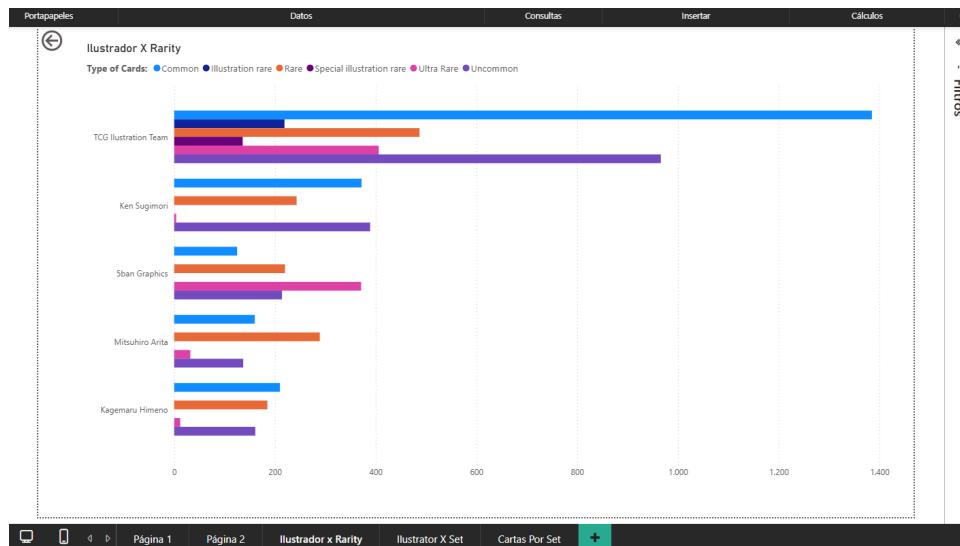
 Below this section, there's a 'Estimaciones de Precio' (Price Estimates) section with three buttons:

- Ver en Cardmarket**
- Estimar Precio Futuro (MLP)**
- Calcular Precio Justo (LGBM)**

Además, los datos en **Google BigQuery** son accesibles mediante **SQL** para integración con herramientas de Business Intelligence (BI) como **Power BI**. Se ha creado un **dashboard de análisis** en Power BI para una visión agregada y profunda, facilitando:

- **Análisis de Tendencias Históricas:** Evolución de precios para sets, rarezas o grupos de cartas.
- **Comparativas de Rendimiento:** Analizar qué sets o rarezas han tenido mejor desempeño.
- **Distribución de Precios:** Entender la dispersión de precios dentro de un set o por rareza.
- **Identificación de Oportunidades:** Identificar cartas o sets infravalorados o con tendencias de crecimiento para decisiones de inversión o colecciónismo.

La aplicación Streamlit es una herramienta de consulta y predicción bajo demanda, mientras que el dashboard de Power BI es una herramienta de análisis estratégico que explora patrones y tendencias a un nivel más granular y agregado, complementando el proyecto.



6. Trabajo Futuro

Este proyecto establece las bases para futuras mejoras:

- **Continuación y Expansión de la Adquisición Histórica:** Mantener y optimizar la ejecución mensual del *script* de adquisición de precios para construir un historial de datos más profundo y granular en BigQuery, crucial para modelos de series temporales.
- **Automatización Completa del Flujo de Datos:** Automatizar el *pipeline* de datos, desde la obtención de *snapshots* mensuales de la API hasta la ingestión automática en Google BigQuery y la actualización periódica de la aplicación Streamlit.
- **Integración y Uso del Modelo LSTM:** Adaptar la aplicación Streamlit para cargar y usar el modelo LSTM a medida que se acumule suficiente historial de precios, modificando la lógica de predicción y considerando la selección dinámica del modelo más apropiado.
- **Selección Dinámica del Horizonte de Predicción:** Permitir al usuario seleccionar interactivamente el número de días para la predicción si se implementa el LSTM o se reentrena un MLP con `days_diff` variable.

- **Visualizaciones Gráficas Avanzadas y de Predicción:** Integrar gráficos interactivos en la sección de detalle de la carta en Streamlit que muestren la evolución histórica del precio y superpongan la predicción del modelo.
- **Implementación de un Chatbot Asistente:** Integrar un *chatbot* para responder preguntas comunes sobre cartas, sets, rarezas, tendencias de mercado y ofrecer consejos basados en datos y predicciones.
- **Mejora Continua del Pipeline de Preprocesamiento:** Refinar el tratamiento de datos faltantes e inconsistentes, y explorar técnicas de ingeniería de características más avanzadas.
- **Escalabilidad y Optimización del Despliegue:** Evaluar la migración de componentes del sistema a soluciones más robustas y escalables (ej. TensorFlow Serving, Vertex AI Endpoints) si la carga de usuarios o el tamaño de los modelos aumentan.
- **Exploración de Factores de Precio Adicionales:** Investigar la incorporación de otras fuentes de datos que influyan en el precio, como la popularidad del Pokémon, su relevancia en el juego competitivo o eventos comunitarios.
- **Segmentación Avanzada y Análisis de Cartas de Alto Valor:** Crear un segmento específico para cartas de muy alto valor económico (Ultra-High Tier, >500€) para desarrollar modelos o enfoques analíticos más adecuados a sus características.