El Proceso de Entrenamiento: Una Conversación con el Código train_model.py

Imagina que eres el director de orquesta y el script train_model.py es tu partitura.

Acto 1: Recopilar a los Músicos (Carga y Preparación de Datos)

Líneas 11-28: Carga de Datos desde BigQuery

codePython

print("Paso 1: Cargando datos desde la tabla Gold 'modelo_final'...")

... (código de conexión) ...

df = pd.read_gbq(query, project_id=PROJECT_ID, credentials=credentials)

- **Qué hacemos:** Nos conectamos a BigQuery y descargamos nuestra tabla gold_data.modelo_final completa.
- **Cómo lo hacemos:** Usamos la librería pandas-gbq, que actúa como un traductor. Le damos nuestras credenciales (credentials) y una pregunta en SQL (query), y se encarga de hablar con Google Cloud para traer los resultados y ponerlos en una tabla de Pandas (df).
- **Por qué:** Este es el punto de partida. Traemos todos nuestros datos, ya limpios y unidos, desde nuestro Data Warehouse a nuestro entorno de trabajo en Python.

Líneas 30-45: Preparación de Datos y Feature Engineering

codePython

df['fecha'] = pd.to_datetime(df['fecha'])

df.set_index('fecha', inplace=True)

... (bucle para convertir a 'category') ...

- **Qué hacemos:** Realizamos los últimos ajustes a nuestros datos para que el modelo los entienda mejor.
- **Cómo lo hacemos:**
    - pd.to_datetime: Nos aseguramos de que la columna fecha sea tratada como una fecha real, no como un texto.
    - df.set_index('fecha', ...): Establecemos la fecha como el índice de la tabla. Esto es crucial para tratar los datos como una **serie temporal**.
    - astype('category'): Le decimos a Pandas que columnas como id_geografia no son números con los que se pueda calcular, sino "etiquetas" o "categorías". Esto ayuda a XGBoost a interpretarlas correctamente y ahorra mucha memoria.

Acto 2: Separar las Partituras (Selección y División de Datos)

Líneas 47-59: Selección de Características y Objetivo

codePython

target = 'consumo_kwh'

features = [col for col in df.columns if col not in [...]]

X = df[features]

y = df[target]

- **Qué hacemos:** Separamos nuestra tabla en dos partes: las "preguntas" y las "respuestas".
- **Cómo lo hacemos:**
    - y = df[target]: y es nuestra variable **objetivo (target)**. Es la columna consumo_kwh, la "respuesta" que queremos que el modelo aprenda a predecir.
    - X = df[features]: X es nuestro conjunto de **características (features)**. Son todas las demás columnas, las "pistas" o "preguntas" que le daremos al modelo (temperatura, día de la semana, población, etc.) para que pueda adivinar la respuesta y.

Líneas 61-71: División de Datos (Train/Test Split)

codePython

test_size = int(len(df) * 0.2)

X_train, X_test = X[:-test_size], X[-test_size:]

y_train, y_test = y[:-test_size], y[-test_size:]

- **Qué hacemos:** Dividimos nuestros datos históricos en dos: un conjunto grande para "estudiar" y uno pequeño para el "examen final".
- **Cómo lo hacemos:** A diferencia de una división aleatoria, aquí lo hacemos de forma **cronológica**. X[:-test_size] coge el primer 80% de los datos (el pasado más lejano) para entrenar. X[-test_size:] coge el último 20% (el pasado más reciente) para probar.
- **Por qué:** Esto simula una situación real. Entrenamos el modelo con lo que sabíamos hasta, por ejemplo, finales de 2024, y luego probamos si es capaz de predecir correctamente los datos de 2025.

Acto 3: El Ensayo (Entrenamiento y Evaluación)

Líneas 73-84: Entrenamiento del Modelo XGBoost

codePython

model = xgb.XGBRegressor(...)

model.fit(X_train, y_train, eval_set=[(X_test, y_test)], verbose=False)

- **Qué hacemos:** ¡Este es el momento mágico! Le decimos al algoritmo que aprenda.
- **Cómo lo hacemos:**

    - xgb.XGBRegressor(...): Creamos una instancia del "aprendiz" de XGBoost, configurando algunos de sus parámetros (como la max_depth o profundidad máxima de sus "árboles de decisión").
    - model.fit(...): Este es el comando clave. Le pasamos los datos de entrenamiento (X_train, y_train) y le decimos "aprende la relación entre estas preguntas y estas respuestas". También le pasamos los datos de prueba (eval_set) para que, mientras aprende, vaya comprobando si está mejorando o no (early_stopping_rounds).
- **Qué obtenemos:** Al final de este paso, la variable model ya no está vacía. Ahora es un **objeto entrenado** que contiene la "fórmula" matemática que ha descubierto para predecir el consumo.

Líneas 86-93: Evaluación del Modelo

codePython

```
y_pred = model.predict(X_test)

mape = mean_absolute_percentage_error(y_test, y_pred)

print(f"RESULTADO FINAL -> MAPE: {mape:.4f}")
```

- **Qué hacemos:** Calificamos el "examen" del modelo.
- **Cómo lo hacemos:**
    - model.predict(X_test): Le damos al modelo las "preguntas del examen" (X_test) y nos devuelve sus "respuestas" (y_pred).
    - mean_absolute_percentage_error(y_test, y_pred): Comparamos las respuestas del modelo (y_pred) con las respuestas correctas que teníamos guardadas (y_test) y calculamos el error porcentual medio (MAPE).
- **Qué obtenemos:** Un único número (el MAPE) que nos dice qué tan bueno es nuestro modelo.

Acto 4: Entender la Mente del Músico (Interpretación)

Líneas 99-122: Interpretación del Modelo (SHAP)

codePython

```
explainer = shap.Explainer(model)

shap_values = explainer(X_test)

shap.summary_plot(shap_values, X_test, ...)

plt.savefig('shap_summary_final.png')
```

- **Qué hacemos:** Le preguntamos al modelo: "¿En qué te has fijado más para hacer tus predicciones?".

- **Cómo lo hacemos:** Usamos la librería shap. shap.Explainer "envuelve" a nuestro modelo para poder interrogarlo. explainer(X_test) calcula la contribución de cada feature para cada una de las predicciones en el conjunto de prueba. summary_plot resume toda esa información en un gráfico de barras.
- **Qué obtenemos:** Una imagen (shap_summary_final.png) que nos muestra visualmente qué características (temperatura, población, día de la semana, etc.) fueron las más influyentes para el modelo.