1. **¿Qué Estamos Haciendo?**

Estamos construyendo un modelo basado en **redes neuronales LSTM** para predecir la cantidad de ventas semanales en distintas ciudades de Ecuador. Para mejorar la precisión, hemos combinado **datos históricos de ventas** con información externa obtenida de **API's** (clima, salario, empleo, etc.).

El modelo recibe una **ventana de 16 semanas** de datos y predice la cantidad de ventas para la siguiente semana.

📌 **Estrategias utilizadas:**  
✅ Eliminación de **outliers** y transformación logarítmica.  
✅ **Normalización** de variables para que el modelo aprenda mejor.  
✅ **Reducción de sobreajuste** con **Dropout**, **Batch Normalization**, y **Early Stopping**.  
✅ Combinación de **Conv1D + LSTM** para capturar patrones temporales.  
✅ **Pérdida personalizada** para enfatizar en predicciones más precisas en valores altos.

## ****Carga y Preprocesamiento de Datos****

El primer paso es leer los datos, limpiar inconsistencias y transformar las variables en un formato útil para el modelo.

df = pd.read\_csv('./data/datos\_combinados.csv')

📌 **Pasos clave:**  
✅ Convertimos fechas en un formato estructurado.  
✅ Creamos variables adicionales como **season** (estación del año) y **is\_big\_city** (ciudades grandes).  
✅ Agrupamos los datos **semanalmente**.  
✅ Aplicamos **winsorización** y **logaritmo** para reducir la influencia de valores extremos.  
✅ **Normalizamos** los datos con StandardScaler.

df\_weekly['qty'] = winsorize(df\_weekly['qty'], limits=[**0**, **0.05**])

df\_weekly['qty\_log'] = np.log(df\_weekly['qty'] + **1**) # Transformación logarítmica

scaler = StandardScaler()

df\_weekly[features] = scaler.fit\_transform(df\_weekly[features])

📌 **¿Por qué esto es importante?**

* **Los datos originales pueden tener valores extremos (outliers)** que afectan el entrenamiento.
* **La normalización ayuda a que todas las variables tengan una escala similar** y mejora la estabilidad del modelo.

## ****Creación de Secuencias para LSTM****

LSTM trabaja con **secuencias de datos en el tiempo**. Necesitamos transformar nuestros datos en pequeños fragmentos de 16 semanas para que la red aprenda los patrones históricos.

ventana = **16**

X\_feats, y\_all = [], []

**for** \_, subset **in** df\_weekly.groupby('areacity'):

target\_vals = subset['qty\_log'].values

feats = subset[features].values

**for** i **in** range(ventana, len(subset)):

X\_feats.append(feats[i-ventana:i])

y\_all.append(target\_vals[i])

📌 **¿Qué hace esto?**

* Para cada ciudad, tomamos **16 semanas de datos como entrada** y usamos la **semana siguiente como salida**.
* Convertimos la información en **matrices NumPy**, que son el formato que TensorFlow usa.
* Agregamos **ruido gaussiano** para mejorar la generalización.

X\_feats\_noisy = X\_feats + np.random.normal(loc=**0.0**, scale=**0.005**, size=X\_feats.shape)

📌 **¿Por qué agregamos ruido?**

* Para evitar que el modelo aprenda **patrones demasiado específicos** y sobreajuste.
* Es una técnica conocida como **data augmentation** en series temporales.

## ****Construcción del Modelo LSTM Mejorado****

Nuestro modelo tiene la siguiente arquitectura:

1️. **Capa Conv1D:** Extrae patrones espaciales de las series temporales.  
2️. **Capa MaxPooling1D:** Reduce dimensionalidad y mejora eficiencia.  
3️. **Dos capas LSTM:** Capturan dependencias a largo plazo en los datos.  
4️. **Capa densa + Batch Normalization + Dropout:** Evitan sobreajuste y mejoran la estabilidad.

feats\_input = Input(shape=(ventana, len(features)), name='feats\_input')

x = Conv1D(filters=**128**, kernel\_size=**3**, activation='relu', padding='same')(feats\_input)

x = MaxPooling1D(pool\_size=**2**)(x)

x = LSTM(**128**, return\_sequences=True, dropout=**0.15**)(x)

x = LSTM(**32**, return\_sequences=False, dropout=**0.15**)(x)

fusion = Dense(**64**, activation='relu')(x)

fusion = BatchNormalization()(fusion)

fusion = Dropout(**0.25**)(fusion)

output = Dense(**1**, activation='linear')(fusion)

📌 **¿Por qué combinamos Conv1D + LSTM?**  
✅ Conv1D ayuda a capturar **patrones cortos** en los datos.  
✅ LSTM es excelente para aprender **patrones largos** en series temporales.

## ****Entrenamiento del Modelo****

model.compile(optimizer=Adam(learning\_rate=**0.0002**), loss=custom\_loss)

📌 **¿Qué técnicas usamos para evitar sobreajuste?**  
✅ **Dropout:** Apaga neuronas aleatorias para evitar que el modelo memorice los datos.  
✅ **Batch Normalization:** Estabiliza la activación de las neuronas.  
✅ **Early Stopping:** Detiene el entrenamiento si el modelo deja de mejorar.  
✅ **ReduceLROnPlateau:** Reduce la tasa de aprendizaje si no hay mejoras.

callbacks=[

EarlyStopping(monitor='val\_loss', patience=**15**, restore\_best\_weights=True),

ReduceLROnPlateau(monitor='val\_loss', factor=**0.5**, patience=**5**, min\_lr=**1e-5**)

]

📌 **¿Por qué usamos una función de pérdida personalizada?**

* Queremos penalizar más los errores en **valores de ventas altos**.

**def** **custom\_loss**(y\_true, y\_pred):

error = K.abs(y\_true - y\_pred)

penalty = K.exp(y\_true / K.max(y\_true))

**return** K.mean(error \* penalty)

## ****Evaluación del Modelo****

📌 **Métricas utilizadas:**  
✅ **RMSE (Root Mean Square Error):** Error absoluto en las predicciones.  
✅ **MAE (Mean Absolute Error):** Diferencia promedio entre la predicción y el valor real.  
✅ **wMAPE (Weighted Mean Absolute Percentage Error):** Métrica clave en negocios.

**def** **weighted\_mape**(y\_true, y\_pred):

y\_true, y\_pred = np.array(y\_true), np.array(y\_pred)

w = y\_true / (y\_true.sum() + **1e-10**)

rel\_error = np.abs(y\_true - y\_pred) / (y\_true + **1e-10**)

**return** np.sum(w \* rel\_error) \* **100**

📌 **¿Cómo interpretamos los resultados?**

* Un **wMAPE bajo** significa que el modelo es **preciso**.
* Un **RMSE alto** sugiere que el modelo tiene problemas con valores extremos.

**print**(f"📌 RMSE : {rmse:.2f}")

**print**(f"📌 MAE : {mae:.2f}")

**print**(f"📌 wMAPE : {wmape:.2f}%")

**print**(f"✅ PRECISIÓN : {precision:.2f}%")

## ****Preguntas****

🔹 **¿Por qué usaste una transformación logarítmica en la variable objetivo?**  
📌 Para reducir el efecto de valores extremos y mejorar la estabilidad del modelo.

🔹 **¿Por qué combinaste Conv1D con LSTM?**  
📌 Conv1D capta patrones locales, mientras que LSTM aprende dependencias temporales largas.

🔹 **¿Cómo manejaste el sobreajuste?**  
📌 Con técnicas como Dropout, Batch Normalization, Early Stopping y Regularización.

🔹 **¿Cómo interpretas el wMAPE?**  
📌 Es el porcentaje de error ponderado por la magnitud de las ventas reales.

**🔹 ¿Qué es Conv1D y por qué lo usamos en el modelo?**

Conv1D es una capa de **convolución unidimensional** usada para procesar datos secuenciales.

📌 **¿Cómo funciona?**

* La convolución actúa como un **filtro deslizante** sobre los datos de entrada.
* Detecta **patrones locales en series temporales** (por ejemplo, picos en ventas o estacionalidad).
* Permite que el modelo **aprenda estructuras más abstractas** antes de pasar los datos a la capa LSTM.

📌 **Ejemplo con imágenes (análogamente a los datos de ventas):**

* En visión computacional, Conv2D detecta bordes o formas en imágenes.
* En series temporales, Conv1D detecta cambios de tendencia o repeticiones en datos de ventas.

### **¿Por qué combinamos Conv1D con LSTM?**

✅ Conv1D detecta **patrones cortos** y los reduce.  
✅ LSTM es excelente para **recordar relaciones largas en el tiempo**.  
✅ Juntos permiten que el modelo **vea la serie desde diferentes perspectivas**.

**🔹 ¿Qué hace cada técnica utilizada para mejorar el modelo?**

### **🔵 Dropout**

❌ Problema: Un modelo con demasiadas conexiones puede memorizar los datos (sobreajuste).  
✅ Solución: Dropout **apaga algunas neuronas aleatoriamente** en cada paso del entrenamiento.  
📌 **Esto fuerza al modelo a aprender patrones más generales en los datos.**

### **🔵 Batch Normalization**

❌ Problema: Cuando los valores de activación cambian mucho, el entrenamiento es inestable.  
✅ Solución: Batch Normalization **normaliza** los valores de cada capa, estabilizando el modelo.  
📌 **Esto permite entrenar más rápido y mejorar precisión.**

### **🔵 Early Stopping**

❌ Problema: Si entrenamos demasiado, el modelo aprende demasiado bien los datos y pierde capacidad de generalizar.  
✅ Solución: EarlyStopping detiene el entrenamiento si la pérdida en validación no mejora después de varias épocas.  
📌 **Esto previene el sobreajuste.**

### **🔵 ReduceLROnPlateau**

❌ Problema: A veces, el modelo deja de mejorar porque la tasa de aprendizaje es alta.  
✅ Solución: Reduce automáticamente el **learning rate** cuando el modelo no mejora.  
📌 **Esto permite entrenar con precisión fina sin reentrenar desde cero.**

**🔹 ¿Los resultados obtenidos son buenos?**

### **📌 Interpretación de las métricas del modelo mejorado**

📊 **Entrenamiento:**

* **RMSE:** 14,603
* **MAE:** 9,793
* **wMAPE:** 18.90%
* **Precisión:** **81.10%**

📊 **Test (evaluación con datos reales no vistos en el entrenamiento):**

* **RMSE:** 19,957
* **MAE:** 12,407
* **wMAPE:** 21.90%
* **Precisión:** **78.10%**

📌 **Conclusión:**  
✔️ La precisión del **81.1% en entrenamiento** es **buena** y **78.1% en test** sigue siendo aceptable.  
✔️ **wMAPE bajo (18.9%-21.9%)** significa que el error relativo en las predicciones es bajo.  
✔️ **El aumento de RMSE en test (mayor error en valores altos) indica que el modelo podría mejorar su generalización.**  
✔️ **Dado que no hay una gran brecha entre entrenamiento y test, no hay sobreajuste severo.**

**¿Cómo podríamos mejorar?**  
✅ Usar **más datos de entrenamiento**.  
✅ Probar una **mayor regularización** o **más dropout**.  
✅ **Optimizar la estructura de la red** (por ejemplo, más filtros en Conv1D).

**EXPLICACIÓN DE GRÁFICAS RESULTANTES**

## ****📊 1. Comparación de Predicciones vs. Valores Reales (Entrenamiento)****

📌 **¿Qué representa?**

* Cada punto azul representa una predicción del modelo frente a su valor real.
* La línea roja es la referencia y = x (predicción perfecta).

📌 **¿Cómo interpretarla?**  
✔️ La mayoría de los puntos están **cerca de la línea roja** → El modelo predice bien.  
✔️ Algunos puntos se **alejan de la línea** → Errores en algunos valores extremos.  
✔️ Dispersión moderada pero aceptable.

## ****📊 2. Comparación de Predicciones vs. Valores Reales (Test)****

📌 **¿Qué representa?**

* Igual que la anterior, pero con **datos que el modelo no vio durante el entrenamiento**.

📌 **¿Cómo interpretarla?**  
✔️ Sigue un **buen alineamiento con la línea roja**, lo que indica **buena generalización**.  
✔️ Se observa **más dispersión que en entrenamiento**, lo que es normal.  
✔️ Algunos valores altos están **subestimados**, lo que indica que el modelo tiene dificultades con valores extremos.

## ****📊 3. Evolución de Ventas Reales vs. Predicciones en Test****

📌 **¿Qué representa?**

* La **línea azul** muestra las ventas reales a lo largo del tiempo.
* La **línea naranja punteada** son las predicciones del modelo.

📌 **¿Cómo interpretarla?**  
✔️ **Las tendencias generales se capturan bien**.  
✔️ Hay **pequeños desfases en algunos picos**, pero sigue la forma general.  
✔️ **El modelo sufre con cambios bruscos**, lo cual es esperable en problemas de series temporales.

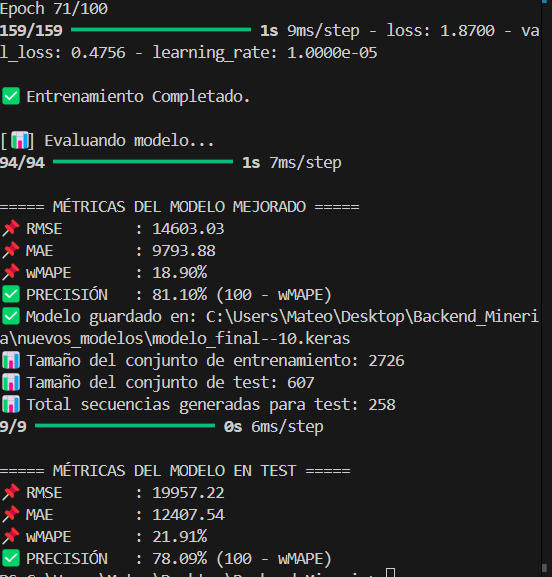
## ****📊 4. Curva de Pérdida del Modelo (Loss Function)****

📌 **¿Qué representa?**

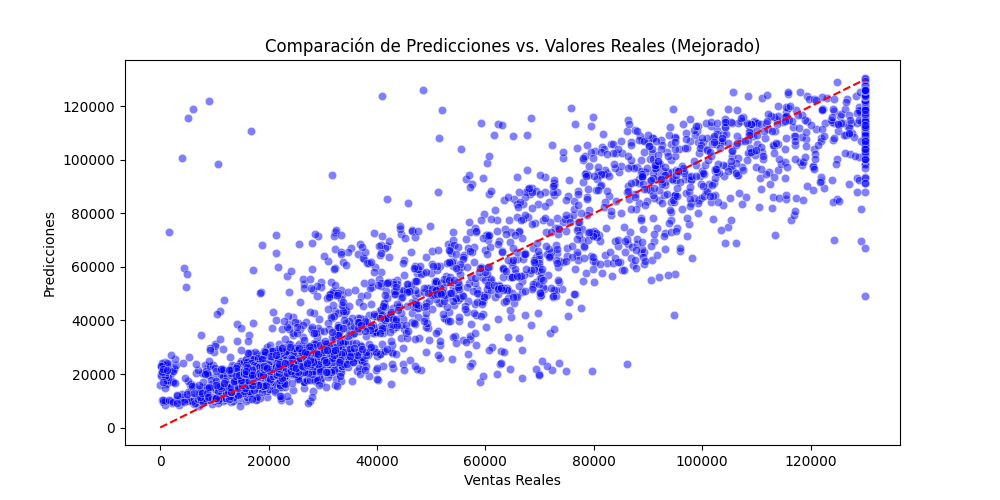
* **Línea azul:** Pérdida en entrenamiento.
* **Línea naranja:** Pérdida en validación (datos no vistos).

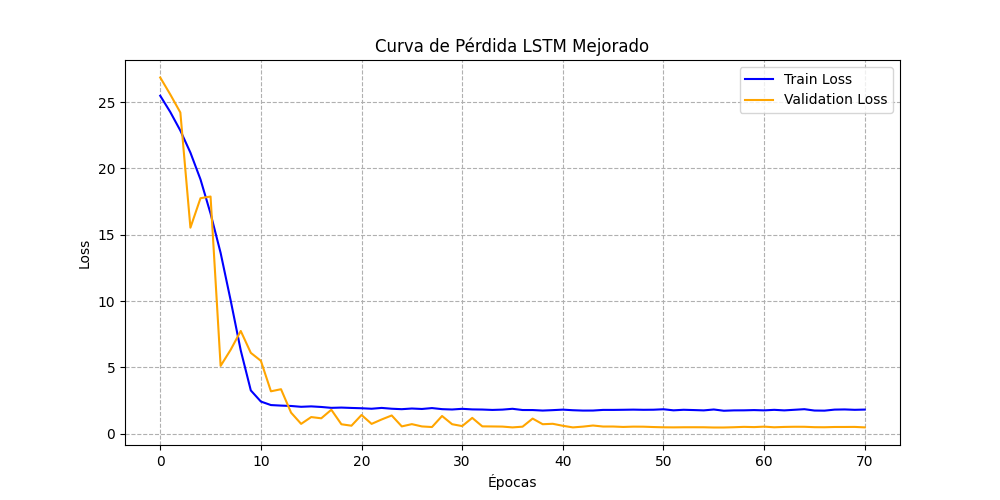
📌 **¿Cómo interpretarla?**  
✔️ **Ambas pérdidas disminuyen rápido en las primeras épocas** → Buen aprendizaje inicial.  
✔️ **Las curvas se estabilizan y son similares** → No hay sobreajuste grave.  
✔️ **La validación no aumenta al final** → Se entrenó bien con EarlyStopping.

----------GRÁFICAS-----------

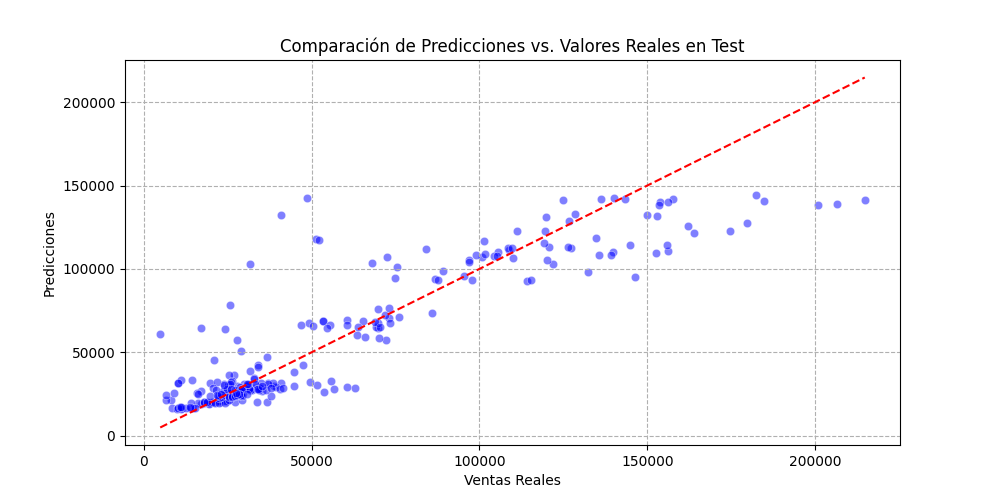


RESULTADOS DEL ENTRENAMIENTO

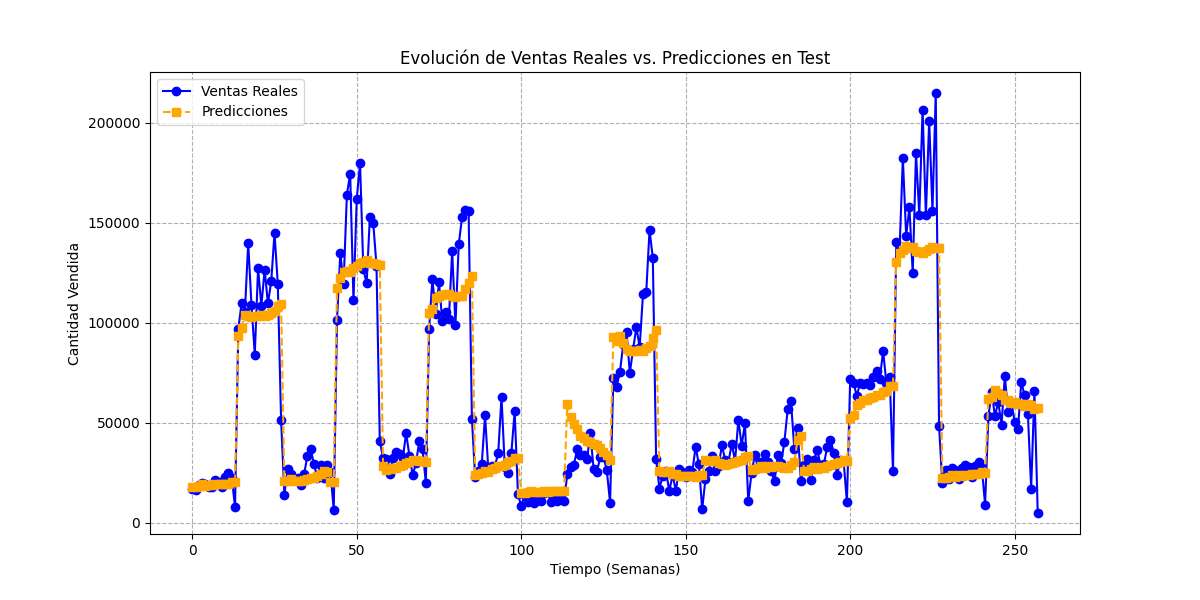


COMPARACIÓN DE PREDICCIONES CON VALORES REALES (ENTRENAMIENTO)

CURVA DE PERDIDA



COMPARACIÓN DE PREDICCIONES CON VALORES REALES (DE TEST)



EVOLUCIÓN DE VENTAS REALES VS PREDICCIONES (EN TEST)

**--------------API--------------**

# **1️. ¿Qué estamos haciendo aquí?**

Este código es una **API REST** creada con **FastAPI** que permite **realizar predicciones de ventas** a partir de un modelo LSTM previamente entrenado.

📌 **¿Cómo funciona?**

1. Cargamos el modelo LSTM guardado (modelo\_final--10.keras).
2. Preprocesamos los datos históricos de ventas.
3. Normalizamos las variables para que el modelo pueda usarlas.
4. Implementamos **dos endpoints** (/predict\_range y /predict\_custom):
   * **/predict\_range**: Predice las ventas para un rango de semanas.
   * **/predict\_custom**: Permite enviar datos personalizados y obtener una predicción.

**📌 Importación de Librerías**

**import** **os**

**import** **numpy** **as** **np**

**import** **pandas** **as** **pd**

**import** **tensorflow** **as** **tf**

**from** **tensorflow.keras.models** **import** load\_model

**from** **fastapi** **import** FastAPI, HTTPException

**from** **pydantic** **import** BaseModel

**from** **datetime** **import** datetime

**from** **sklearn.preprocessing** **import** StandardScaler

**import** **tensorflow.keras.backend** **as** **K**

✅ tensorflow.keras.models.load\_model → Carga el modelo guardado.  
✅ FastAPI → Crea una API REST para interactuar con el modelo.  
✅ Pydantic → Define la estructura de los datos que recibirá la API.  
✅ StandardScaler → Normaliza los datos de entrada para que tengan una escala adecuada.

**📌 Función de Pérdida Personalizada**

**def** **custom\_loss**(y\_true, y\_pred):

error = K.abs(y\_true - y\_pred)

penalty = K.exp(y\_true / K.max(y\_true))

**return** K.mean(error \* penalty)

📌 **¿Qué hace esta función?**

* Calcula la diferencia **absoluta** entre la predicción y el valor real.
* **Penaliza más los errores en valores altos** de ventas, usando una función exponencial.

📌 **¿Por qué usamos una pérdida personalizada?**  
✅ En negocios, **predecir mal valores altos de ventas es más costoso** que fallar en valores bajos.  
✅ Evita que el modelo solo se enfoque en valores pequeños y mejore su precisión en grandes ventas.

**📌 Carga del Modelo**

MODEL\_PATH = r"C:\Users\Mateo\Desktop\Backend\_Mineria\nuevos\_modelos\modelo\_final--10.keras"

model = load\_model(MODEL\_PATH, custom\_objects={"custom\_loss": custom\_loss})

📌 **¿Qué estamos haciendo aquí?**

* Cargamos el modelo entrenado (modelo\_final--10.keras).
* Incluimos la función de pérdida personalizada (custom\_loss).

**📌 Carga y Preprocesamiento de Datos**

FEATURES = ['ipc', 'tasa\_empleo', 'salario\_nominal', 'salario\_real', 'qty\_trend', 'season', 'is\_big\_city']

scaler = StandardScaler()

DATA\_PATH = "./data/datos\_combinados.csv"

df = pd.read\_csv(DATA\_PATH)

📌 **¿Qué estamos haciendo aquí?**

* Definimos **las columnas** que usará el modelo (FEATURES).
* Cargamos los **datos históricos de ventas**.
* Creamos un **escalador (StandardScaler)** para normalizar las variables.

df['date'] = pd.to\_datetime(df[['year', 'month', 'day']])

df['year\_week'] = df['date'].dt.strftime('%Y-%U')

big\_cities = ['QUITO', 'GUAYAQUIL', 'CUENCA']

df['is\_big\_city'] = df['areacity'].apply(**lambda** x: **1** **if** x **in** big\_cities **else** **0**)

df['season'] = df['date'].dt.month.apply(**lambda** x: **1** **if** x **in** [**1**, **2**, **3**, **4**, **5**, **10**, **11**, **12**] **else** **0**)

📌 **¿Qué estamos haciendo aquí?**  
✅ Convertimos la fecha (year, month, day) en un formato estructurado.  
✅ Creamos nuevas variables:

* **is\_big\_city** → Marca si la ciudad es grande (1) o pequeña (0).
* **season** → Indica si es invierno (1) o verano (0).

df\_weekly = df.groupby(['year\_week', 'areacity'], as\_index=False).agg({

'qty': 'sum',

'ipc': 'mean',

'tasa\_empleo': 'mean',

'salario\_nominal': 'mean',

'salario\_real': 'mean',

'is\_big\_city': 'max'

})

df\_weekly['qty\_trend'] = df\_weekly.groupby('areacity')['qty'].transform(**lambda** x: x.rolling(window=**4**, min\_periods=**1**).mean())

scaler.fit(df\_weekly[FEATURES])

📌 **¿Qué estamos haciendo aquí?**  
✅ **Agrupamos las ventas semanalmente por ciudad**.  
✅ Calculamos **qty\_trend** (promedio móvil de 4 semanas).  
✅ **Normalizamos** los datos para que el modelo los use correctamente.

**📌 Creación de la API con FastAPI**

app = FastAPI()

📌 **FastAPI nos permite:**  
✅ Crear una API para hacer predicciones en tiempo real.  
✅ Recibir datos y devolver resultados en formato JSON.

**📌 Definición de los Endpoints**

**🔵 /predict\_range - Predicción para un rango de fechas**

**class** **PredictionRangeRequest**(BaseModel):

ciudad: str

fecha\_inicio: str

fecha\_fin: str

📌 **Define el formato de la solicitud:**

* ciudad: Ciudad para la que queremos predecir.
* fecha\_inicio: Fecha inicial del rango.
* fecha\_fin: Fecha final del rango.

**@app.post**("/predict\_range")

**def** **predict\_range**(request: PredictionRangeRequest):

📌 **Este endpoint:**  
✅ Recibe una ciudad y un rango de fechas.  
✅ Obtiene los datos previos necesarios (últimas 16 semanas).  
✅ **Realiza predicciones semana por semana.**  
✅ Devuelve los resultados en un JSON.

**🔵 /predict\_custom - Predicción con datos personalizados**

**class** **PredictionCustomRequest**(BaseModel):

ciudad: str

ultimas\_semanas: list # Lista de diccionarios con datos de las últimas 16 semanas

📌 **Este endpoint:**  
✅ Recibe **16 semanas de datos personalizados**.  
✅ Normaliza los datos y hace una **predicción única**.  
✅ Útil para probar predicciones con datos externos o simulados.

**@app.post**("/predict\_custom")

**def** **predict\_custom**(request: PredictionCustomRequest):

📌 **¿Cómo funciona?**

1. **Valida que se envíen exactamente 16 semanas de datos.**
2. **Convierte los datos en un DataFrame y los normaliza.**
3. **Hace la predicción y devuelve la cantidad estimada de ventas.**