

Paquetes

```
pip install -q skforecast
```

Esto instala la librería **skforecast**, que permite usar modelos de regresión (scikit-learn, XGBoost, LightGBM, etc.) para pronósticos de series temporales. El argumento `-q` significa *quiet*, para que no muestre tanto texto en consola.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
```

- **numpy (np)**: librería base para cálculos numéricos, vectores y matrices.
- **pandas (pd)**: usada para manejar datos en forma de DataFrame y series temporales.
- **matplotlib.pyplot (plt)**: para graficar datos y resultados.
- **seaborn (sns)**: agrega estilos y funciones gráficas más atractivas que matplotlib.

Estas son las librerías de **análisis exploratorio y visualización**.

```
from xgboost import XGBRegressor
from lightgbm import LGBMRegressor
from sklearn.ensemble import RandomForestRegressor
```

Aquí se importan **modelos base** de aprendizaje automático:

- **XGBRegressor**: implementa *gradient boosting* con árboles (XGBoost).
- **LGBMRegressor**: versión optimizada de boosting (LightGBM).
- **RandomForestRegressor**: bosque de árboles de decisión (de scikit-learn).

Estos modelos sirven como regresores dentro del forecaster de `skforecast`.

```
from skforecast.recursive import ForecasterRecursive
```

Clase principal de `skforecast` para construir un modelo de pronóstico. Funciona entrenando un regresor para un **paso adelante** y luego usando sus predicciones como insumos en los siguientes pasos (**forecasting recursivo**).

```
from skforecast.model_selection import TimeSeriesFold,
grid_search_forecaster, backtesting_forecaster
```

- **TimeSeriesFold**: división de datos en ventanas temporales, respeta el orden de la serie (a diferencia de KFold clásico).
- **grid_search_forecaster**: permite probar distintas combinaciones de hiperparámetros y lags para elegir la mejor.
- **backtesting_forecaster**: evalúa el modelo simulando pronósticos repetidos en distintos puntos históricos (como un “testeo hacia atrás”).

Estas funciones permiten **validación y selección de modelos** en series temporales.

```
from skforecast.preprocessing import TimeSeriesDifferentiator
```

Transformador para **diferenciar** una serie temporal. La diferenciación es útil para quitar tendencia o hacer que una serie no estacionaria se convierta en estacionaria antes de entrenar el modelo.

```
from sklearn.metrics import mean_squared_error, mean_absolute_error
```

Funciones de **evaluación de error**:

- **MSE (Mean Squared Error)**: penaliza errores grandes, útil para medir precisión en predicción.
 - **MAE (Mean Absolute Error)**: da el error promedio en las mismas unidades de la demanda (más interpretable).
-

Datos

```
data = pd.read_excel("DemandaCOES_.xlsx", skiprows=3)
```

Se carga el archivo Excel que contiene la demanda eléctrica. El argumento `skiprows=3` indica que se deben saltar las tres primeras filas del archivo, normalmente son títulos o notas que no forman parte de la tabla de datos.

```
data['FECHA'] = pd.to_datetime(data['FECHA'], format='%d/%m/%Y %H:%M')
```

La columna `FECHA` se convierte en un objeto de tipo fecha-hora (`datetime`). Se especifica el formato exacto en que vienen los datos: día/mes/año seguido de horas y minutos.

```
data.set_index('FECHA', inplace=True)
```

La columna `FECHA` se convierte en el índice del DataFrame. Esto es importante porque a partir de aquí los registros estarán organizados como una serie temporal.

```
data.rename(columns={'EJECUTADO': 'Demand'}, inplace=True)
```

Se cambia el nombre de la columna `EJECUTADO` a `Demand`, que será la variable objetivo (la serie de consumo eléctrico que se quiere analizar y pronosticar).

```
data = data.asfreq("30min")
```

Se fija la frecuencia del índice temporal a intervalos de 30 minutos. Esto asegura que toda la serie tenga un espaciado uniforme; si faltan valores en el archivo original, se insertan filas con índices correspondientes y valores vacíos (`NaN`).

```
data.head()
```

Se muestran las primeras filas del DataFrame para verificar que la estructura y los cambios realizados estén correctos.

```
data.Demand.plot(figsize=(16,6), label="Consumo Electrico")
```

Se grafica la serie `Demand` completa, en un lienzo de 16x6 pulgadas. El argumento `label` define el texto que aparecerá en la leyenda para identificar la curva como "Consumo Eléctrico".

```
data.Demand.rolling(48).mean().plot(label='Media móvil (24 horas)')
```

Se calcula una **media móvil** de la demanda con una ventana de 48 observaciones. Como los datos están en intervalos de 30 minutos, 48 puntos equivalen a un día completo (24 horas). El resultado suaviza la curva original y se grafica en la misma figura, con etiqueta "Media móvil (24 horas)".

```
plt.legend()
```

Activa la leyenda en la esquina del gráfico, mostrando las etiquetas de cada serie para diferenciarlas.

```
plt.show()
```

Muestra el gráfico en pantalla.

```
data.Demand.plot(figsize=(16,6), label="Consumo Electrico")
```

Se dibuja la serie completa de demanda eléctrica en una figura de 16x6 pulgadas. La curva representa los valores observados tal cual están en el DataFrame.

```
data.Demand.rolling(48*7).mean().plot(label='Media móvil (7 días)')
```

Aquí se calcula una **media móvil** con una ventana de 48*7 observaciones. Como cada día tiene 48 intervalos de 30 minutos, al multiplicar por 7 se obtiene el equivalente a una semana. El promedio móvil suaviza la serie resaltando la tendencia semanal y reduce el ruido de corto plazo.

```
plt.legend()
```

Se añade la leyenda que mostrará las etiquetas de ambas líneas: la demanda original y la media móvil de 7 días.

```
plt.show()
```

Se renderiza el gráfico final en pantalla.

Efecto cíclico

```
data['ciclo'] = data.index.map(lambda t: (t.hour * 60 + t.minute) / (24 * 60))
```

Se crea una nueva columna llamada `ciclo`. Para cada registro del índice (que es de tipo fecha-hora), se calcula la cantidad de minutos transcurridos desde la medianoche (`t.hour * 60 + t.minute`). Ese valor se divide entre el total de minutos que tiene un día ($24 \times 60 = 1440$). El resultado es un número entre 0 y 1 que indica la **posición relativa dentro del día**:

- 0 corresponde a las 00:00,
- 0.5 al mediodía,
- valores cercanos a 1 son la noche.

```
data[['Demand', 'ciclo']][-48*7:].plot(figsize=(16,8), subplots=True)
```

Se seleccionan las columnas `Demand` y `ciclo` pero solo en la última semana (-48*7 equivale a 7 días con 48 intervalos de 30 minutos cada uno). Se grafican como subgráficas independientes dentro de una misma figura de tamaño 16x8. Esto permite visualizar cómo evoluciona la demanda real frente al ciclo intradía generado.

```
plt.show()
```

Se muestra el gráfico final.

Efecto diario

```
days_translation = {  
    'Monday': '1Lunes',  
    'Tuesday': '2Martes',  
    'Wednesday': '3Miércoles',  
    'Thursday': '4Jueves',  
    'Friday': '5Viernes',  
    'Saturday': '6Sábado',  
    'Sunday': '0Domingo'  
}
```

Se define un diccionario que traduce el nombre de los días de la semana (que `pandas` devuelve en inglés) a nombres en español. Se antepone un número para que alfabéticamente los días queden en orden lógico de domingo a sábado.

```
data['dia'] = data.index.day_name().map(days_translation)
```

Se crea una nueva columna `dia` con el nombre traducido y numerado de cada día según el índice de fecha-hora. Así cada registro tiene asignado si fue domingo, lunes, etc.

```
data2 = pd.get_dummies(data, columns=['dia'], dtype=int)
```

Se generan variables dummy (columnas binarias 0/1) para cada día de la semana. Por ejemplo, `dia_0Domingo` vale 1 si la observación ocurrió en domingo y 0 en otro caso. Esto convierte la información categórica en formato numérico para que pueda usarse en modelos de ML.

```
data2.columns
```

Se listan todas las columnas de `data2` para verificar que las dummies se hayan creado correctamente.

```
data2[['Demand', 'dia_0Domingo', 'dia_1Lunes', 'dia_2Martes', 'dia_3Miércoles',  
       [-48*7:]].plot(figsize=(16, 32), subplots=True)
```

Se selecciona la columna `Demand` junto con todas las dummies de día de semana. Se grafican únicamente los últimos 7 días de datos (48 intervalos por día \times 7). El parámetro `subplots=True` coloca cada variable en su propio gráfico vertical dentro de la figura de tamaño 16x32.

```
plt.show()
```

Se muestra el conjunto de gráficos donde se puede ver la evolución de la demanda junto a la activación de cada día de la semana.

Efecto de los feriados

```
import holidays
```

Se importa la librería `holidays`, que contiene calendarios oficiales de distintos países y permite generar listas de feriados automáticamente.

```
pe = holidays.Peru(years=[2023, 2024], observed=True)
```

Se crea un objeto con los feriados de Perú para los años 2023 y 2024. El argumento `observed=True` hace que se incluyan también los días "observados" o trasladados cuando un feriado cae en fin de semana.

```
data2['feriado'] = data2.index.normalize().isin(pe).astype(int)
```

- `data2.index.normalize()` convierte el índice de fecha-hora en solo fecha (elimina la hora).
- `.isin(pe)` comprueba si esa fecha está dentro del calendario de feriados de Perú.
- `.astype(int)` transforma el resultado booleano (True/False) en 1 y 0. El resultado es una nueva columna `feriado` que vale 1 en días festivos y 0 en días normales.

```
data2[['Demand', 'feriado']][-48*7: ].plot(figsize=(16,6), subplots=True)
```

Se grafican, para la última semana de datos, la demanda eléctrica y la variable binaria `feriado`. Al usar `subplots=True`, cada variable aparece en su propio gráfico, apilados verticalmente, dentro de una figura de 16x6.

```
plt.show()
```

Se muestra la figura final, donde se puede observar la relación entre la demanda y si hubo o no feriado en esos días.

Partición de la serie

```
steps = 48*7 # separar la última semana
```

Aquí se define el horizonte de evaluación. Como cada día tiene 48 registros (30 minutos cada uno), multiplicar por 7 da una semana completa. Ese tramo final será el conjunto de prueba.

```
x_train = data2.drop(columns=['Demand'])[:-steps]
x_test = data2.drop(columns=['Demand'])[-steps:]
```

Se dividen las variables explicativas (exógenas):

- `x_train` contiene todos los predictores menos la semana final.
- `x_test` contiene solo la última semana.

```
y_train = data2['Demand'][:-steps]
y_test = data2['Demand'][-steps:]
```

Lo mismo pero para la variable objetivo (`Demand`). `y_train` es lo que el modelo usará para aprender, mientras que `y_test` se guarda para medir el rendimiento fuera de muestra.

Por qué es importante particionar: En series temporales no se puede evaluar un modelo con los mismos datos con los que se entrena, porque eso daría una falsa sensación de exactitud. La idea de separar la última semana (`steps`) es simular el escenario real: entrenas el modelo con el pasado y luego lo enfrentas a datos que nunca

ha visto (el futuro). Así puedes comprobar si el modelo realmente generaliza y predice bien lo que viene después, y no solo memoriza el historial.

Además, en forecasting el tiempo siempre avanza en una dirección: entrenas con lo antiguo y predices lo más nuevo. Por eso la partición siempre se hace “dejando al final” el conjunto de prueba, respetando la cronología.

BackTesting

```
forecaster = ForecasterRecursive(  
    regressor = XGBRegressor(random_state=...),  
    lags      = ...  
)
```

El **forecaster** es el objeto principal que hace forecasting recursivo.

- `regressor` recibe el modelo base (por ejemplo, `XGBRegressor`). Aquí se define qué algoritmo se usará para aprender.
- `lags` indica cuántos pasos hacia atrás de la serie se van a usar como predictores. Un valor mayor implica más memoria del sistema.

```
cv = TimeSeriesFold(  
    steps          = ...,  
    initial_train_size = ...,  
    refit          = ...,  
    fixed_train_size = ...  
)
```

Este objeto define la **validación cruzada para series temporales**.

- `steps` es el tamaño de cada bloque de validación (por ejemplo, un día).
- `initial_train_size` es la cantidad de observaciones con que se inicia el primer entrenamiento.
- `refit` indica si el modelo debe reentrenarse en cada fold o no.
- `fixed_train_size` controla si el tamaño de la ventana de entrenamiento se mantiene fijo o se expande al avanzar en el tiempo.

Esto asegura que la validación respete la secuencia temporal: se entrena en el pasado y se valida en el futuro.

```
lags_grid = [...]
```

Aquí se define la lista de **valores candidatos para los lags**. Sirve para explorar si el modelo funciona mejor usando, por ejemplo, un día completo de memoria, dos días, o una semana.

```
param_grid = {  
    'n_estimators': [...],  
    'max_depth':   [...]  
}
```

Es el **espacio de búsqueda de hiperparámetros** del modelo base.

- `n_estimators` controla el número de árboles que entrena XGBoost.
- `max_depth` controla la profundidad de cada árbol, es decir, su complejidad.

Con este grid, el proceso probará diferentes combinaciones para encontrar la más efectiva.

```
results_grid = grid_search_forecaster(  
    forecaster = forecaster,  
    y = y_train,  
    cv = cv,  
    param_grid = param_grid,  
    lags_grid = lags_grid,  
    metric = 'mean_squared_error',  
    return_best = True,  
    n_jobs = 'auto',  
    verbose = False  
)
```

La función que hace el **grid search**.

- `forecaster` es el objeto que queremos optimizar.
- `y` es la serie objetivo sobre la que se entrena y valida.
- `cv` define cómo se dividen los datos en folds temporales.
- `param_grid` es el conjunto de hiperparámetros del modelo que se van a probar.
- `lags_grid` son los distintos números de lags que se testean.
- `metric` define la métrica de error que se optimiza (ej. MSE).
- `return_best=True` asegura que el forecaster quede configurado con la mejor combinación encontrada.
- `n_jobs` indica cuántos núcleos usar para paralelizar la búsqueda.
- `verbose` controla el nivel de detalle en los mensajes que se muestran.

👉 La utilidad conjunta es:

1. **Validación cruzada temporal (TimeSeriesFold):** simula varios escenarios históricos de "entrenar en el pasado, validar en el futuro".
2. **Backtesting:** es la aplicación práctica de esa validación en múltiples puntos, para tener una estimación más robusta del error.
3. **Grid search de hiperparámetros y lags:** prueba distintas memorias de la serie y configuraciones del modelo base, eligiendo la que minimiza el error definido en `metric`.

Ajuste del modelo

```
regressor = XGBRegressor(n_estimators=..., max_depth=...,  
random_state=...)
```

Se instancia el **modelo base** de XGBoost pero ahora con los **mejores hiperparámetros** que se encontraron en el grid search.

- `n_estimators` es la cantidad de árboles en el ensamble.
 - `max_depth` es la profundidad máxima de cada árbol, que regula la complejidad.
 - `random_state` garantiza reproducibilidad.
-

```
forecaster = ForecasterRecursive(  
    regressor = regressor,  
    lags      = ...  
)
```

Se crea un nuevo **forecaster** con ese regressor optimizado. El parámetro `lags` se fija al valor que resultó más conveniente en la búsqueda, en este caso 96, lo que significa que el modelo usará las últimas 96 observaciones (dos días) como predictores.

```
forecaster.fit(y = y_train)
```

Aquí se entrena el forecaster sobre toda la parte de entrenamiento (`y_train`). Es decir, ya no en los folds de validación, sino usando todo el historial disponible previo al periodo de prueba, para que aprenda con la mayor cantidad de datos posible.

```
predictions = forecaster.predict(steps = steps)
```

Se generan las predicciones recursivas para el horizonte definido en `steps` . Cada predicción se alimenta como lag del siguiente paso, hasta completar la última semana (48*7 registros).

```
fig, ax = plt.subplots(figsize=(12, 4))  
y_train[-steps: ].plot(ax=ax, label='train')  
y_test.plot(ax=ax, label='test')  
predictions.plot(ax=ax, label='predictions')  
ax.legend()  
plt.show()
```

Se grafican tres curvas en la misma figura:

- El final del conjunto de entrenamiento (`y_train[-steps:]`) para ver la transición.
- El conjunto de prueba real (`y_test`), que representa lo que realmente ocurrió en la última semana.
- Las predicciones del modelo (`predictions`), para comparar visualmente la cercanía a los valores observados.

Esto permite verificar de manera gráfica si el modelo logra seguir la forma de la serie, anticipar picos y no perderse en la tendencia.

Metrica de error

```
error_mse1 = mean_squared_error(  
    y_true = y_test,  
    y_pred = predictions  
)
```

Aquí se calcula el **Mean Squared Error (MSE)** comparando los valores reales (`y_test`) contra las predicciones (`predictions`).

- `y_true` es la demanda real en el periodo de prueba.
- `y_pred` son los valores que generó el modelo para ese mismo periodo. El MSE mide la diferencia promedio al cuadrado, penalizando más los errores grandes.

```
print(f"Test error 1 (MSE): {error_mse1}")
```

Se imprime el valor obtenido para el MSE, que representa cuánto se equivocó el modelo en la semana de prueba.

Este es un paso clave porque, más allá de la gráfica, necesitas cuantificar el rendimiento con métricas objetivas. En forecasting eléctrico es común usar varias métricas (MSE, MAE, MAPE) para tener una visión más completa del error y no depender de una sola medida.

¿Cómo agregamos variables exógenas?

Para extender el proceso univariado y trabajar con exógenas (ciclo horario, día de la semana, feriados), se deben añadir tres pasos clave:

```
# 1. Durante el grid search
results_grid = grid_search_forecaster(
    forecaster = forecaster,
    y           = y_train,
    exog        = x_train,    # incluir exógenas
    cv          = cv,
    param_grid  = {...},
    lags_grid   = [...]
)

# 2. En el entrenamiento final
forecaster.fit(y = y_train, exog = x_train)

# 3. En la predicción
predictions = forecaster.predict(steps = steps, exog = x_test)
```

De esta manera, el forecaster combina los **lags de la serie** con las **variables externas conocidas en el futuro**, mejorando la capacidad de anticipar patrones regulares (horarios, días y feriados).