Actualizacion de statsmodels ya que originalmente me daba conflictos y no me dejaba ejecutar muchas funcionalidades

```
!pip install statsmodels --upgrade
```

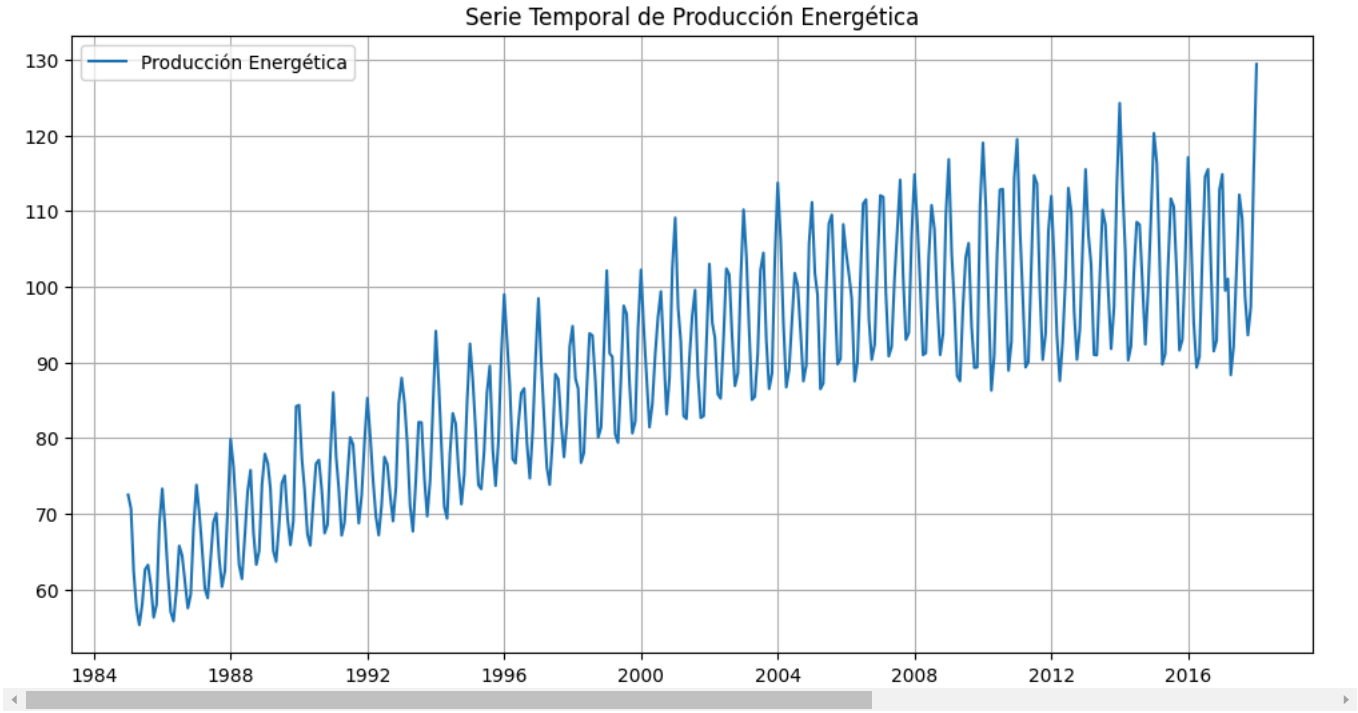
```
Requirement already satisfied: statsmodels in /usr/local/lib/python3.10/dist-packages (0.14.4)
Requirement already satisfied: numpy<3,>=1.22.3 in /usr/local/lib/python3.10/dist-packages (from statsmodels) (1.26.4)
Requirement already satisfied: scipy!=1.9.2,>=1.8 in /usr/local/lib/python3.10/dist-packages (from statsmodels) (1.13.1)
Requirement already satisfied: pandas!=2.1.0,>=1.4 in /usr/local/lib/python3.10/dist-packages (from statsmodels) (2.2.2)
Requirement already satisfied: patsy>=0.5.6 in /usr/local/lib/python3.10/dist-packages (from statsmodels) (1.0.1)
Requirement already satisfied: packaging>=21.3 in /usr/local/lib/python3.10/dist-packages (from statsmodels) (24.2)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.10/dist-packages (from pandas!=2.1.0,>=1.4->sta
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.10/dist-packages (from pandas!=2.1.0,>=1.4->statsmodels)
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.10/dist-packages (from pandas!=2.1.0,>=1.4->statsmodels
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-packages (from python-dateutil>=2.8.2->pandas!=2.1.0
```

Importamos librerias y establecemos configuracion inicial para mas adelante

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.stats.diagnostic import acorr_ljungbox
from sklearn.metrics import mean_squared_error
import itertools
import random
import warnings

# Configuración inicial
warnings.filterwarnings('ignore')

# Configuración de parámetros del algoritmo
initial_temperature = 10
final_temperature = 0.001
cooling_rate = 0.9
iterations = 30
```

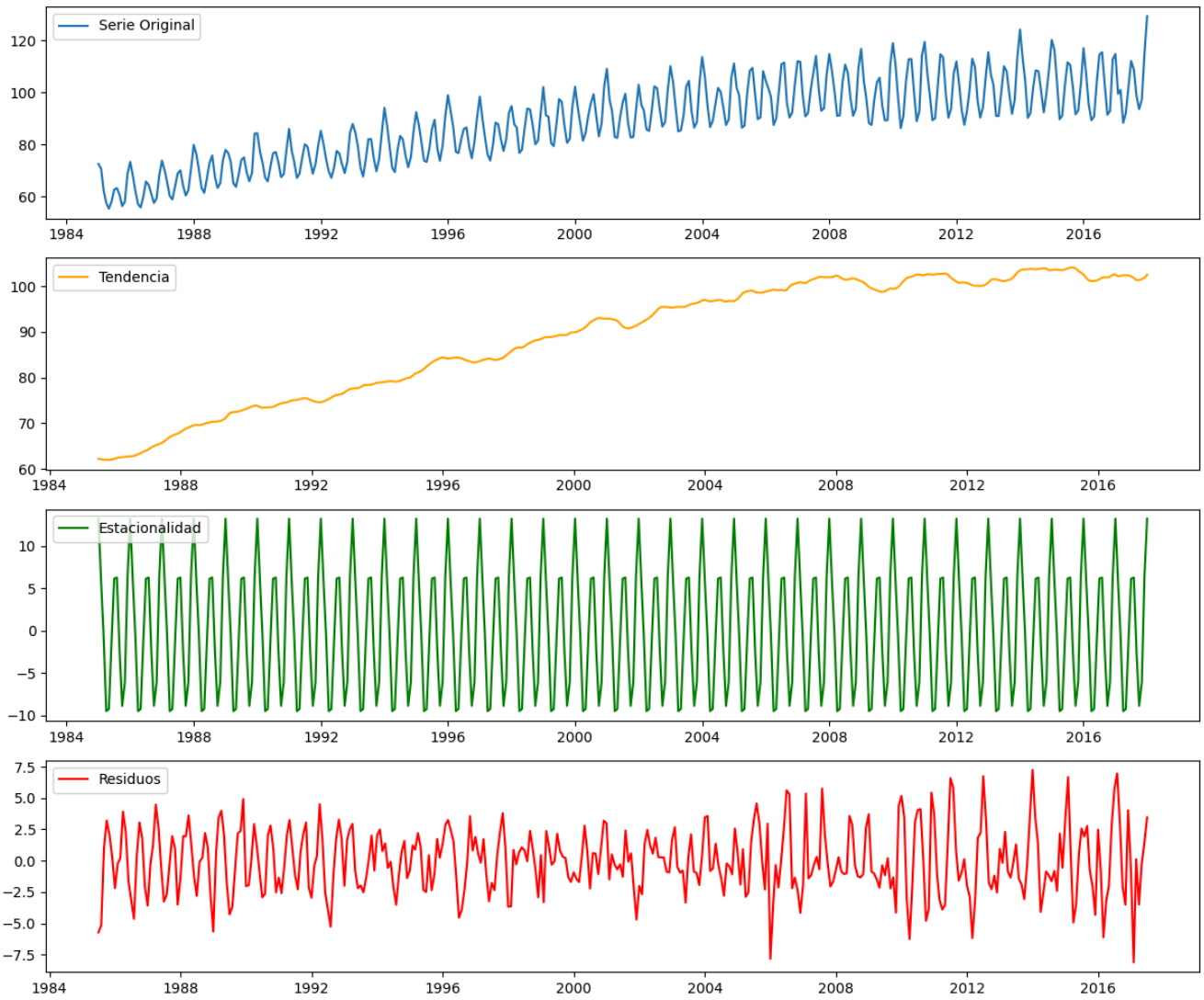
Cargamos y visualizamos datos

```
# Cargar datos
file_path = '/content/drive/My Drive/PracticasMaster/P8/Electric_Production.csv'
data = pd.read_csv(file_path)
data['DATE'] = pd.to_datetime(data['DATE'])
data.set_index('DATE', inplace=True)
data = data.asfreq('MS')

# Visualizar datos
plt.figure(figsize=(12, 6))
plt.plot(data['IPG2211A2N'], label='Producción Energética')
plt.title('Serie Temporal de Producción Energética')
plt.grid()
plt.legend()
plt.show()
```



Descomponemos la serie temporal

```
from statsmodels.tsa.seasonal import seasonal_decompose

# Descomponer la serie
decomposition = seasonal_decompose(data['IPG2211A2N'], model='additive', period=12)

# Visualizar componentes
plt.figure(figsize=(12, 10))

plt.subplot(4, 1, 1)
plt.plot(data['IPG2211A2N'], label='Serie Original')
plt.legend(loc='upper left')

plt.subplot(4, 1, 2)
plt.plot(decomposition.trend, label='Tendencia', color='orange')
plt.legend(loc='upper left')

plt.subplot(4, 1, 3)
plt.plot(decomposition.seasonal, label='Estacionalidad', color='green')
plt.legend(loc='upper left')

plt.subplot(4, 1, 4)
plt.plot(decomposition.resid, label='Residuos', color='red')
plt.legend(loc='upper left')

plt.tight_layout()
plt.show()
```



Construimos el grafo de soluciones

- Creamos el grafo que representará todas las posibles configuraciones de parámetros p, d, q, P, D, Q que cumplan las restricciones.
- Cada nodo sera una configuración única de parámetros y cada arista conectará configuraciones vecinas que difieran en uno de sus valores.

```
def generar_nodos():
    nodos = []
    for p in range(8):
        for d in range(2):
            for q in range(8):
                for P in range(8):
                    for D in range(2):
                        for Q in range(8):
                            nodos.append((p, d, q, P, D, Q))
    return nodos

# Crear grafo
import networkx as nx
grafo = nx.Graph()
nodos = generar_nodos()
grafo.add_nodes_from(nodos)

# Agregar conexiones válidas
for nodo in nodos:
```

```
        for i in range(len(nodo)):
            vecino = list(nodo)
            for cambio in [-1, 1]:
                vecino[i] += cambio
                if vecino[i] >= 0 and vecino[i] < (8 if i in [0, 2, 3, 5] else 2):
                    grafo.add_edge(nodo, tuple(vecino))

    print(f"Total de nodos: {len(grafo.nodes)}")
    print(f"Total de aristas: {len(grafo.edges)}")
```

```
Total de nodos: 16384
Total de aristas: 90112
```

Generación de vecinos

```
def generate_neighbor(solution):
    p, d, q, P, D, Q = solution
    neighbor = list(solution)
    index_to_modify = random.randint(0, len(solution) - 1)
    change = random.choice([-1, 1])
    neighbor[index_to_modify] += change

    # Aplicar restricciones
    neighbor[0] = max(0, min(neighbor[0], 7))  # p
    neighbor[1] = max(0, min(neighbor[1], 1))  # d
    neighbor[2] = max(0, min(neighbor[2], 7))  # q
    neighbor[3] = max(0, min(neighbor[3], 7))  # P
    neighbor[4] = max(0, min(neighbor[4], 1))  # D
    neighbor[5] = max(0, min(neighbor[5], 7))  # Q

    return tuple(neighbor)
```

Definicion funciones de evaluacion

```
def evaluate_solution(solution, train, validation):
    try:
        p, d, q, P, D, Q = solution
        model = SARIMAX(train, order=(p, d, q), seasonal_order=(P, D, Q, 12))
        results = model.fit(disp=False)
        forecast = results.forecast(steps=len(validation))
        rmse = np.sqrt(mean_squared_error(validation, forecast))
        lb_test = acorr_ljungbox(results.resid, lags=[6], return_df=True)
        if (lb_test['lb_pvalue'] < 0.05).any():
            return float('inf')
        return rmse
    except Exception as e:
        return float('inf')
```

Implementación del Algoritmo Simulated Annealing

```
# Algoritmo de Recocido Simulado Mejorado
def simulated_annealing(train, validation, initial_solution, max_iter=30, initial_temp=10, cooling_rate=0.9):
    current_solution = initial_solution
    best_solution = initial_solution
    best_rmse = float('inf')
    temperature = initial_temp

    for iteration in range(max_iter):
        neighbor = generate_neighbor(current_solution)
        current_rmse = evaluate_solution(current_solution, train, validation)
        neighbor_rmse = evaluate_solution(neighbor, train, validation)

        if neighbor_rmse < current_rmse:
            current_solution = neighbor
        else:
            delta = neighbor_rmse - current_rmse
            acceptance_probability = np.exp(-delta / temperature)
            if np.random.rand() < acceptance_probability:
                current_solution = neighbor

        if current_rmse < best_rmse:
            best_solution = current_solution
            best_rmse = current_rmse
```

```
        temperature *= cooling_rate
        print(f"Iteración {iteration + 1}: Mejor Solución: {best_solution}, RMSE: {best_rmse:.4f}")

    return best_solution, best_rmse
```

Ejecución del Algoritmo

```
train_data = data[:-50]
validation_data = data[-50:]

initial_solution = (1, 1, 1, 1, 1, 1)
best_params, best_rmse = simulated_annealing(train_data['IPG2211A2N'], validation_data['IPG2211A2N'], initial_solution)

print(f"Evaluación final del modelo:")
print(f"Mejores parámetros: {best_params}")
print(f"RMSE en conjunto de validación: {best_rmse}")
```

```
Iteración 1: Mejor Solución: (1, 1, 1, 1, 1, 2), RMSE: 3.6462
Iteración 2: Mejor Solución: (1, 1, 1, 1, 1, 2), RMSE: 3.6379
Iteración 3: Mejor Solución: (1, 1, 1, 1, 1, 2), RMSE: 3.6379
Iteración 4: Mejor Solución: (1, 1, 1, 1, 1, 2), RMSE: 3.6379
Iteración 5: Mejor Solución: (1, 1, 1, 1, 1, 2), RMSE: 3.6379
Iteración 6: Mejor Solución: (1, 1, 1, 1, 1, 2), RMSE: 3.6379
Iteración 7: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 8: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 9: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 10: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 11: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 12: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 13: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 14: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 15: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 16: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 17: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 18: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 19: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 20: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 21: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 22: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 23: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 24: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 25: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 26: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 27: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 28: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 29: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Iteración 30: Mejor Solución: (1, 1, 1, 0, 1, 1), RMSE: 3.6346
Mejores parámetros encontrados: (1, 1, 1, 0, 1, 1)
Mejor RMSE obtenido: 3.634611386778892
```

Aplicamos test de Ljung-Box para comprobar residuos finales

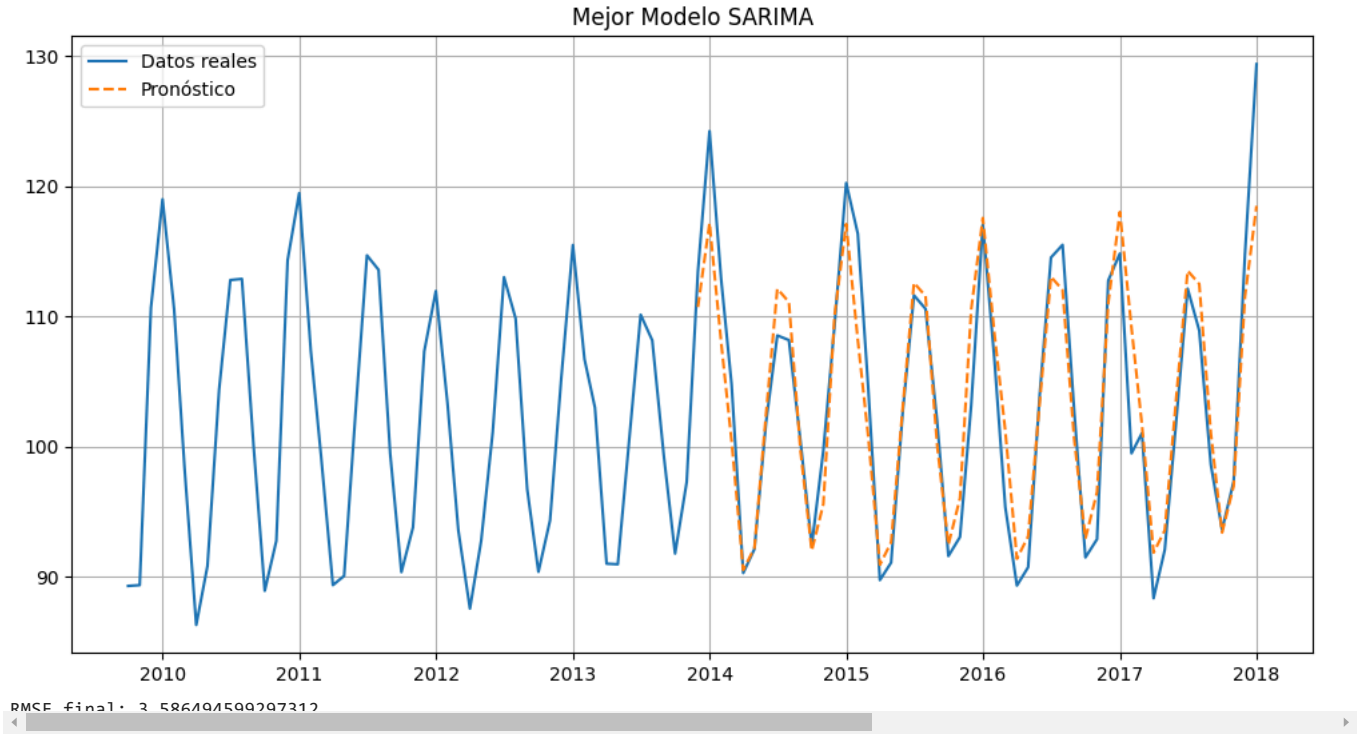
```
from statsmodels.stats.diagnostic import acorr_ljungbox

# Validación de residuos finales
print("\nValidación de residuos con el test Ljung-Box:")
residuos_finales = best_model.resid

# Test Ljung-Box
ljung_box_results = acorr_ljungbox(residuos_finales, lags=[6], return_df=True)

# Mostrar resultados
print("Resultados del test Ljung-Box para residuos finales:")
print(ljung_box_results)

# Comprobación de autocorrelación
if any(ljung_box_results['lb_pvalue'] < 0.05):
    print("Advertencia: Los residuos presentan autocorrelación significativa en algunos retardos.")
else:
    print("Los residuos no presentan autocorrelación significativa.")
```

```

Validación de residuos con el test Ljung-Box:
Resultados del test Ljung-Box para residuos finales:
     lb_stat  lb_pvalue
6  6.182309   0.403081
Los residuos no presentan autocorrelación significativa.
```

Validación final de la solución

```
# Validar residuos finales
best_model = SARIMAX(train_data['IPG2211A2N'], order=(best_params[0], best_params[1], best_params[2]),
                     seasonal_order=(best_params[3], best_params[4], best_params[5], 12)).fit(disp=False)
residuos_finales = best_model.resid
ljung_box_results = acorr_ljungbox(residuos_finales, lags=[6], return_df=True)
print(ljung_box_results)

# Visualizar ajuste
plt.figure(figsize=(12, 6))
plt.plot(data.index[-100:], data['IPG2211A2N'][-100:], label='Datos reales')
plt.plot(validation_data.index, best_model.forecast(steps=50), label='Pronóstico', linestyle='--')
plt.legend()
plt.title('Mejor Modelo SARIMA')
plt.grid()
plt.show()

# RMSE final
final_rmse = np.sqrt(mean_squared_error(validation_data, forecast))
print(f"RMSE final: {final_rmse}")
```

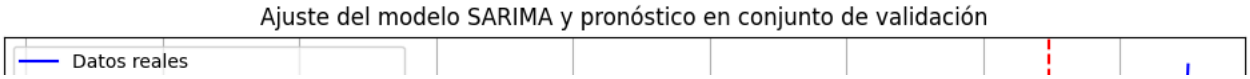
```
     lb_stat  lb_pvalue
6  6.182309   0.403081
```




```
RMSE final: 3.586494599297312
```

Visualizacion final resultados

```
plt.figure(figsize=(12, 6))

# Datos reales
plt.plot(data.index, data['IPG2211A2N'], label='Datos reales', color='blue')

# Predicciones del modelo
train_pred = best_model.fittedvalues
validation_pred = best_model.get_forecast(steps=len(validation_data)).predicted_mean

plt.plot(train_data.index, train_pred, label='Ajuste del modelo (entrenamiento)', color='orange')
plt.plot(validation_data.index, validation_pred, label='Pronóstico (validación)', color='green')

# Líneas y etiquetas
plt.axvline(validation_data.index[0], color='red', linestyle='--', label='Inicio validación')
plt.title('Ajuste del modelo SARIMA y pronóstico en conjunto de validación')
plt.xlabel('Fecha')
plt.ylabel('Consumo eléctrico')
plt.legend()
plt.grid()
plt.show()
```


Ajuste del modelo SARIMA y pronóstico en conjunto de validación
Datos reales