TAREA SERIES TEMPORALES

Adriana Acero Fernández-Villacañas

Máster en Big Data, Data Science e Inteligencia Artificial
Universidad Complutense de Madrid

1 Introducción: presentación de la serie a analizar. (0.5)

Los datos elegidos corresponden a la evolución de la demanda eléctrica en España (megavatios, MW). Se pueden obtener de la página oficial de la Red Eléctrica Española: https://www.ree.es. Gracias a la documentación disponible de la API (https://www.ree.es/es/apidatos) se han sacado los datos diarios por año, desde 2011 a 2023, con el siguiente script:

```
import requests
import csv

url = 'https://apidatos.ree.es/es/datos/demanda/evolucion?start_date=2023-01-01T00:00&end_date=
2023-12-31T00:00&time_trunc=day'

headers = {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
}

response = requests.get(url, headers=headers)

if response.status_code == 200:
    data = response.json()

    with open('Tarea/demanda_electrica2023.csv', 'w', newline='') as file:
        writer = csv.writer(file)

        writer.writerow(['Fecha', 'Demanda'])

        for item in data['included'][0]['attributes']['values']:
            fecha = item['datetime']
            demanda = item['value']

            writer.writerow([fecha, demanda])

    print("Datos guardados en demanda_electrica2023.csv")
else:
    print("Error en la solicitud:", response.status_code)
```

De esta manera, modificando el año de la URL (y el nombre del fichero que se guarda) se ha obtenido un csv por año con los datos diarios de la demanda eléctrica.

Una vez obtenidos los datos diarios, se han llevado a cabo un proceso de limpieza del formato de las fechas y concatenación de los datos en un solo archivo:

```
import pandas as pd
import glob
import os

os.chdir(r'.\Tarea') #Verificar directorio actual de trabajo

pattern = 'demanda_electrica*.csv'

csv_files = glob.glob(pattern)

for file in csv_files:
    df = pd.read_csv(file)

    df['Fecha'] = df['Fecha'].str.replace('T00:00:00.000+01:00', '', regex=False)
    df['Fecha'] = df['Fecha'].str.replace('T00:00:00.000+02:00', '', regex=False)
```

```
    df.to_csv(file, index=False)


    #### fusionamos los csvs: ######


dfs = []
for file in csv_files:
    df = pd.read_csv(file)
    dfs.append(df)

df_concatenado = pd.concat(dfs, ignore_index=True)
df_concatenado['Fecha'] = df_concatenado['Fecha'].str.replace('T00:00:00.000+01:00', '', regex=False)
df_concatenado['Fecha'] = df_concatenado['Fecha'].str.replace('T00:00:00.000+02:00', '', regex=False)

df_concatenado.to_csv('demanda_electrica_2011_2023.csv', index=False)
```

Por último, se han seleccionado los valores máximos de cada mes para adecuarlo a la tarea, ya que se requería un conjunto de datos de unos 150 datos observados:

```
import pandas as pd
df = pd.read_csv('demanda_electrica_2011_2023.csv', index_col='Fecha', parse_dates=True)

df['Año'] = df.index.year
df['Mes'] = df.index.month

df_maximo_por_mes = df.groupby(['Año', 'Mes'])['Demanda'].max().reset_index()

df_maximo_por_mes['Año'] = df_maximo_por_mes['Año'].astype(int)
df_maximo_por_mes['Mes'] = df_maximo_por_mes['Mes'].astype(int)

df_maximo_por_mes['Fecha'] = pd.to_datetime(df_maximo_por_mes['Año'].astype(str) + '-' +
                                            df_maximo_por_mes['Mes'].astype(str) + '-01')
df_maximo_por_mes.drop(['Año', 'Mes'], axis=1, inplace=True)
df_maximo_por_mes.to_csv('demanda_electrica_maxima_mes_2011_2023.csv', index=False)
```

El archivo de datos final contiene 156 registros, el índice Fecha y la columna Demanda con los valores máximos cada mes de la demanda eléctrica, importándolo al jupyter notebook de trabajo principal:

```
df = pd.read_csv('demanda_electrica_maxima_mes_2011_2023.csv', index_col='Fecha', parse_dates=True)
```

```
df.head()
```

Fecha	Demanda
2011-01-01	925838.099
2011-02-01	896370.497
2011-03-01	872249.280
2011-04-01	739283.981
2011-05-01	768988.409

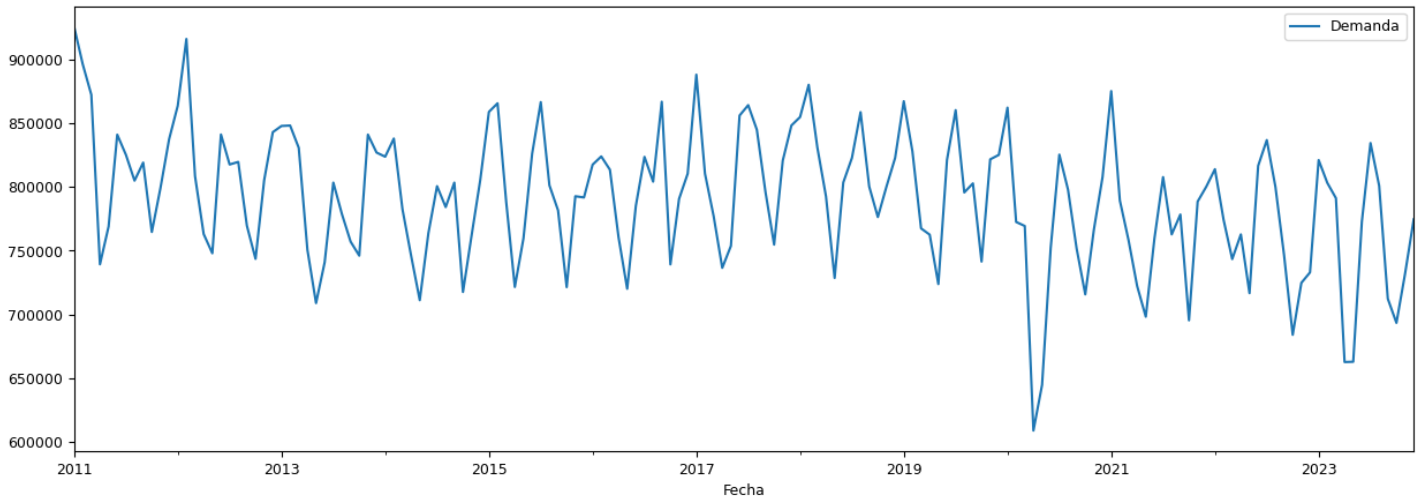
```
df.dtypes
```

```
Demanda    float64
dtype: object
```

Este conjunto de datos es intersante ya que al anticipar la demanda futura, los operadores podrían ajustar la producción energética, minimizando el desperdicio y mejorando la sostenibilidad.

2 Representación gráfica y descomposición estacional (si tuviera comportamiento estacional). (1.5)

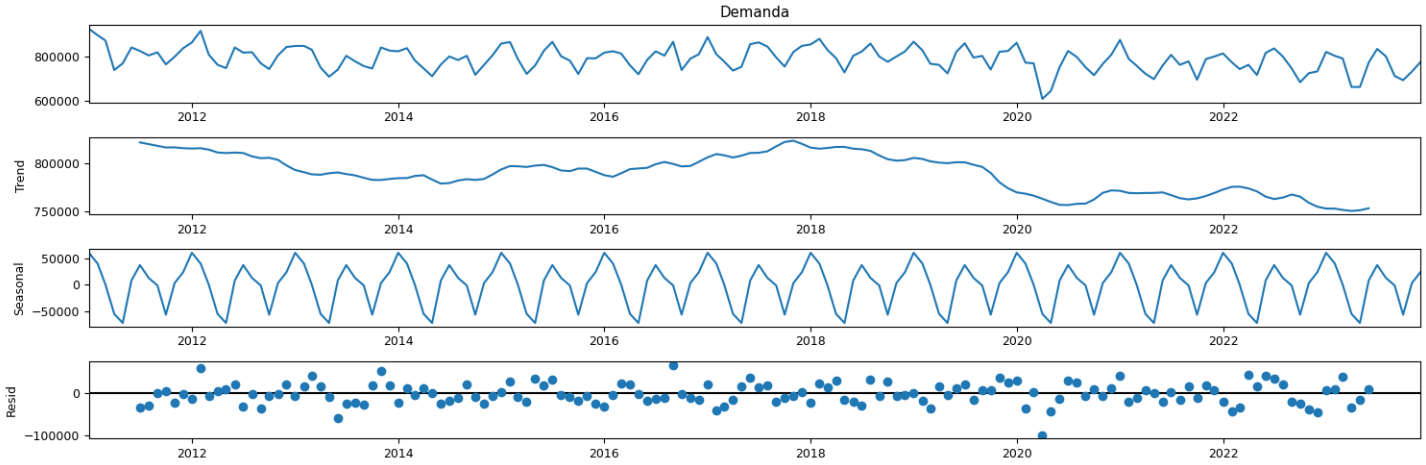
```
df.plot()
plt.show()
```



Como se puede observar, la serie tiene un marcado carácter estacional: la demanda repunta en los meses de invierno y verano, mientras que los mínimos corresponden a los meses de primavera y otoño. Además, existe un mínimo global en la gráfica en el año 2020. Esto puede ser debido a la pandemia de COVID-19, puesto que se redujo la actividad en las industrias, en las oficinas, servicios, etc.

Por otra parte, se ha considerado una descomposición aditiva, ya que parece que la tendencia y estacionalidad son constantes a lo largo del tiempo:

```
from statsmodels.tsa.seasonal import seasonal_decompose
add_decomposition = seasonal_decompose(df['Demanda'], model='additive', period=12)
descomposicion_grafica = add_decomposition.plot()
```



3 Para comprobar la eficacia de los métodos de predicción que vamos a hacer en los siguientes apartados reservamos los últimos datos observados (TEST, un periodo en las series estacionales o aproximadamente 10 observaciones), para comparar con las predicciones realizadas por cada uno de los métodos. Luego ajustamos los modelos sobre la serie sin esos último datos (TRAIN) en los siguientes apartados.

Se comprueba si se ha detectado automáticamente la frecuenca de los índices:

```
df.index
```

```
DatetimeIndex(['2011-01-01', '2011-02-01', '2011-03-01', '2011-04-01',
               '2011-05-01', '2011-06-01', '2011-07-01', '2011-08-01',
               '2011-09-01', '2011-10-01',
               ...
               '2023-03-01', '2023-04-01', '2023-05-01', '2023-06-01',
               '2023-07-01', '2023-08-01', '2023-09-01', '2023-10-01',
               '2023-11-01', '2023-12-01'],
              dtype='datetime64[ns]', name='Fecha', length=156, freq=None)
```

Como freq=None, hay que especificar la frecuencia manualmente:

```
df.index.freq = 'MS' # MS es Month Start
```

A continuación, ya se pueden construir los conjuntos de datos train y test:

```
N_test = 12

train = df.iloc[:-N_test]
test = df.iloc[-N_test:]

train_idx = df.index <= train.index[-1]
test_idx = df.index > train.index[-1]
```

4 Encontrar el modelo de suavizado exponencial más adecuado, mostrando una tabla con los estimadores de los parámetros del modelo elegido. Para dicho modelo, representar gráficamente la serie observada y la suavizada con las predicciones para el periodo TEST. Mostrar una tabla con las predicciones. (2)

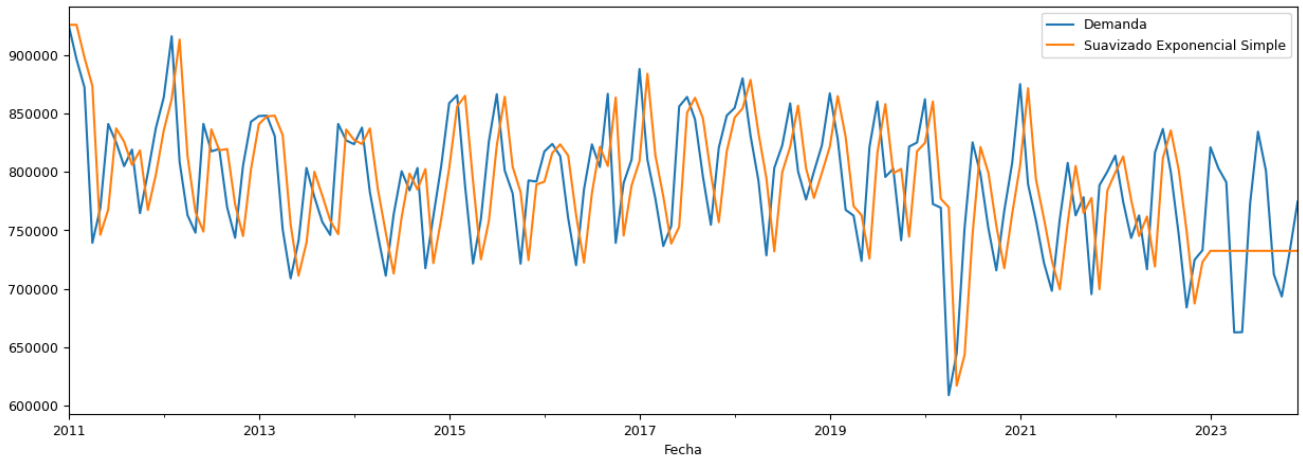
En este apartado se van a evaluar los modelos de suavizado exponencial simple, el de Holt y el de Holt-Winters. Dada la presencia de estacionalidad en los datos, se anticipa que el modelo de Holt-Winters será el más adecuado para capturarla y proporcionar el mejor ajuste. Además, se va a utilizar la métrica del error cuadrático medio raíz (RMSE) para comparar el desempeño de los modelos.

4.1. Modelo de suavizado exponencial simple (SES)

```
from statsmodels.tsa.holtwinters import SimpleExpSmoothing, Holt, ExponentialSmoothing
import itertools
from sklearn.metrics import mean_squared_error
```

```
ses = SimpleExpSmoothing(train['Demanda'], initialization_method='legacy-heuristic')
res_ses = ses.fit()

df.loc[train_idx,'Suavizado Exponencial Simple'] = res_ses.fittedvalues
df.loc[test_idx, 'Suavizado Exponencial Simple'] = res_ses.forecast(N_test)
```

```
df[['Demanda', 'Suavizado Exponencial Simple']].plot()
```

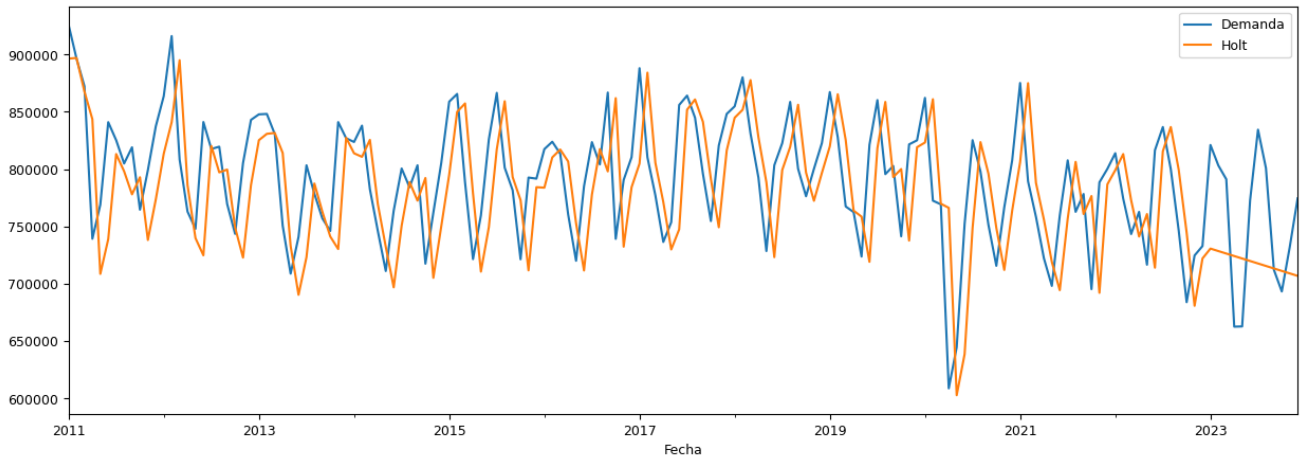



```
print("Root Mean Square Error (TRAIN): ", mean_squared_error(train['Demanda'],res_ses.fittedvalues,
                                                             squared=False))
print("Root Mean Square Error (TEST): ", mean_squared_error(test['Demanda'],res_ses.forecast(N_test),
                                                             squared=False))
```

```
Root Mean Square Error (TRAIN):  52478.84050379847
Root Mean Square Error (TEST):  62211.7164045061
```

4.2. Modelo lineal de Holt

```
holt = Holt(train['Demanda'], initialization_method='legacy-heuristic')
res_h = holt.fit()

df.loc[train_idx, 'Holt'] = res_h.fittedvalues
df.loc[test_idx, 'Holt'] = res_h.forecast(N_test)

df[['Demanda', 'Holt']].plot()
```



```
print("Root Mean Square Error (TRAIN): ", mean_squared_error(train['Demanda'],res_h.fittedvalues,
                                                squared=False))
print("Root Mean Square Error (TEST): ", mean_squared_error(test['Demanda'],res_h.forecast(N_test),
                                                squared=False))
```

```
Root Mean Square Error (TRAIN):  54085.51425732554
Root Mean Square Error (TEST):  67282.5463381124
```

4.3. Modelo Holt-Winters (y con selección de hiperparámetros)

Para seleccionar los hiperparámetros óptimos del modelo Holt-Winters se ha empleado un enfoque de validación progresiva (walkforward), complementado con una búsqueda exhaustiva (grid search).

Primero se establecen las variables de horizonte de pronóstico y el número de iteraciones en la validación:

```
h= 12 #horizonte de pronóstico
steps = 13
NTest = len(df)-h-steps +1
```

Listas con todas las opciones de hiperparámetros:

```
lista_tendencias = ['add', 'mul']
lista_estacionalidades = ['add','mul']
lista_tendencia_amortiguada = [True,False]
lista_init_methods = ['estimated','heuristic','legacy-heuristic']
lista_boxcox = [True, False,0]
```

La función walkforward implementa la validación progresiva iterando sobre el conjunto de datos, utilizando segmentos consecutivos como entrenamiento y prueba para cada iteración. La configuración específica de hiperparámetros para el modelo de Holt-Winters se evalúa en cada paso, calculando el RMSE entre las predicciones y los valores reales del segmento de prueba. Este proceso se repetirá para cada combinación de hiperparámetros generada por la búsqueda exhaustiva.

```
def walkforward(tipo_tendencia, tipo_estacionalidad, tendencia_amortiguada, init_method, use_boxcox, debug=False):
    errores = []
    ultimo = False # marcador que nos dirá si hemos procesado la última fila del dataframe
    pasos_completados = 0 #cuando se complete el for, será el mismo número que le hemos puesto en
    #la variable steps

    for end_of_train in range(NTest, len(df)-h+1):
        train = df.iloc[:end_of_train]
        test = df.iloc[end_of_train:end_of_train + h]

        if test.index[-1] == df.index[-1]:
            ultimo = True

        pasos_completados +=1

        hw = ExponentialSmoothing(train['Demanda'], initialization_method= init_method, trend=tipo_tendencia,
                          damped_trend=tendencia_amortiguada, seasonal=tipo_estacionalidad,
                          seasonal_periods=12, use_boxcox=use_boxcox)
        res_hw = hw.fit()


        fcast = res_hw.forecast(h)
        error = mean_squared_error(test['Demanda'], fcast, squared=False)
```

```
        errores.append(error)

    if debug:
        print("último registro: ", ultimo)
        print("Pasos completados: ", pasos_completados)

    return np.mean(errores)
```

Se itera sobre todos los posibles conjuntos de hiperparámetros y se registra la métrica RMSE (del conjunto de datos test) de cada configuración:

```
tupla_de_listas_de_opciones = (lista_tendencias, lista_estacionalidades, lista_tendencia_amortiguada,
                               lista_init_methods, lista_boxcox, )

mejor_score = float('inf')
mejor_opcion = None

for x in itertools.product(*tupla_de_listas_de_opciones):
    score = walkforward(*x)

    if score < mejor_score:
        print("La mejor score hasta el momento: ", score)
        mejor_score = score
        mejor_opcion = x

print("Mejor puntuación (Root Mean Squared Error): " ,mejor_score)
print("Mejores opciones de hiperparámetros: ",mejor_opcion)
```

```
Mejor puntuación (Root Mean Squared Error):  34874.541943558346
Mejores opciones de hiperparámetros:  ('mul', 'add', False, 'estimated', True)
```

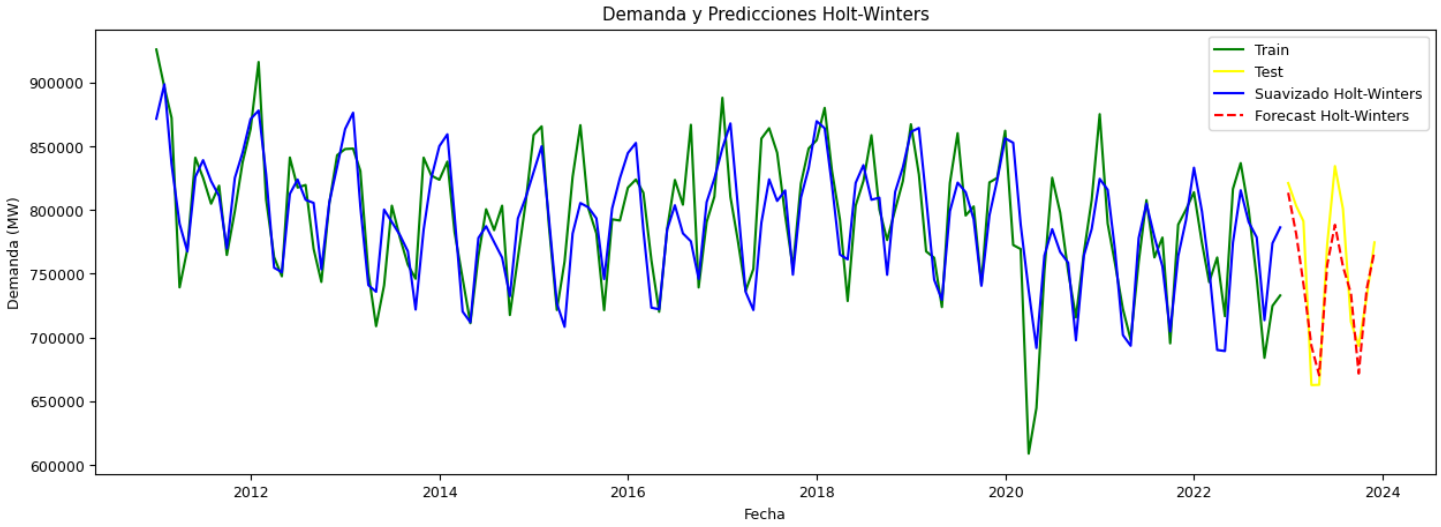
Se observa que, de los modelos suavizados, éste presenta el menor RMSE. Adicionalmente, se muestra en la salida anterior cuáles son los hiperparámetros que hay que pasarle al método ExponentialSmoothing para minimizar esta métrica. A continuación, se genera el modelo con esos hiperparámetros:

```
holtwinters = ExponentialSmoothing(train['Demanda'],trend='mul', seasonal='add', damped_trend=False,
                                                    initialization_method='estimated', use_boxcox=True)
res_hw = holtwinters.fit()

df.loc[train_idx, 'Holt-Winters'] = res_hw.fittedvalues
df.loc[test_idx, 'Holt-Winters'] = res_hw.forecast(N_test)
```

Y la representación gráfica:

```
plt.plot(df.index[train_idx], df.loc[train_idx, 'Demanda'], label='Train', color='green')

plt.plot(df.index[test_idx], df.loc[test_idx, 'Demanda'], label='Test', color='yellow')

plt.plot(df.index[train_idx], df.loc[train_idx, 'Holt-Winters'], label='Suavizado Holt-Winters', color='blue')

plt.plot(df.index[test_idx], df.loc[test_idx, 'Holt-Winters'], label='Forecast Holt-Winters', color='red'
                                                ,linestyle='--')

plt.legend()
```

```
plt.title('Demanda y Predicciones Holt-Winters')
plt.xlabel('Fecha')
plt.ylabel('Demanda (MW)')

plt.show()
```



Por último, se muestran la tabla con los estimadores de los parámetros del modelo de Holt-Winters y otra tabla con las predicciones:

```
from tabulate import tabulate
headers = ['Nombre', 'Param', 'Valor', 'Optimizado']
table_str=tabulate(res_hw.params_formatted, headers, tablefmt='fancy_grid')
print(table_str)
```

Nombre	Param	Valor	Optimizado
smoothing_level	alpha	0.181786	True
smoothing_trend	beta	0.0001	True
smoothing_seasonal	gamma	0.157349	True
initial_level	l.0	2.40888e+13	True
initial_trend	b.0	0.997414	True
initial_seasons.0	s.0	3.45853e+12	True
initial_seasons.1	s.1	4.78985e+12	True
initial_seasons.2	s.2	3.24705e+11	True
initial_seasons.3	s.3	-3.20843e+12	True
initial_seasons.4	s.4	-3.95219e+12	True
initial_seasons.5	s.5	-1.0194e+11	True
initial_seasons.6	s.6	6.76264e+11	True
initial_seasons.7	s.7	-2.18325e+11	True
initial_seasons.8	s.8	-7.32408e+11	True
initial_seasons.9	s.9	-3.41257e+12	True
initial_seasons.10	s.10	3.13271e+11	True
initial_seasons.11	s.11	2.06323e+12	True

El parámetro alpha indica que el modelo da un peso moderado a las observaciones más recientes, aunque conserva

cierta memoria en las observaciones pasadas. También se observa un valor beta de la tendencia muy bajo.

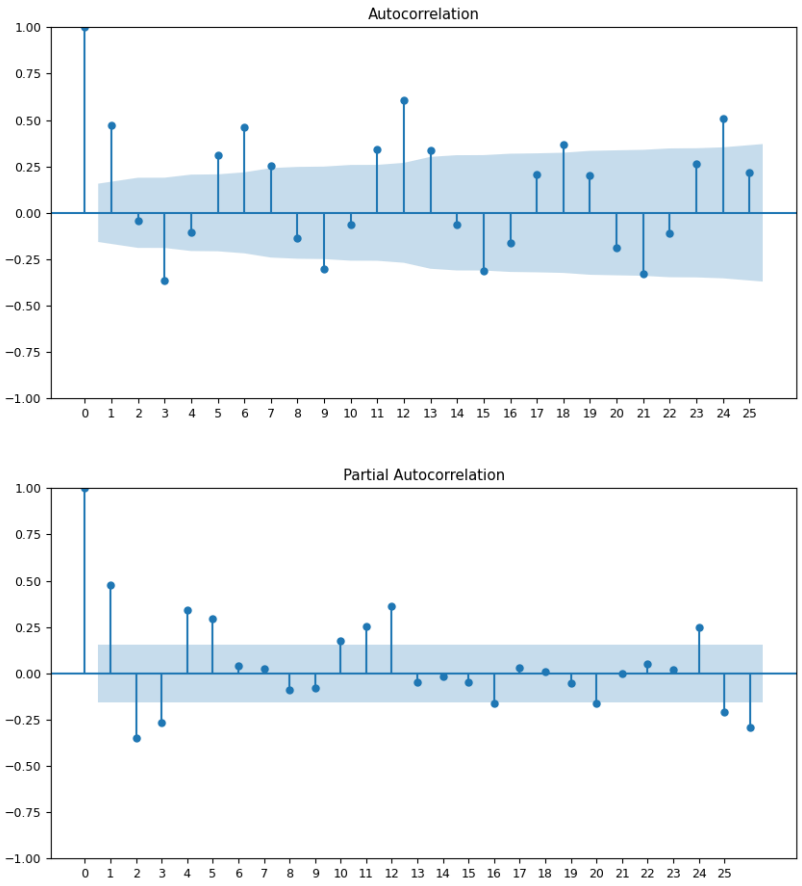
```
predicciones_df = df.loc[test_idx, ['Demanda', 'Holt-Winters']]
predicciones_df['Fecha'] = predicciones_df.index
predicciones_df = predicciones_df[['Fecha', 'Demanda', 'Holt-Winters']]
table_str2=tabulate(predicciones_df, 'keys', tablefmt='fancy_grid', showindex=False)
print(table_str2)
```

Fecha	Demanda	Holt-Winters
2023-01-01 00:00:00	821055	813473
2023-02-01 00:00:00	803175	781108
2023-03-01 00:00:00	791156	742060
2023-04-01 00:00:00	662633	693285
2023-05-01 00:00:00	662854	670198
2023-06-01 00:00:00	772763	756836
2023-07-01 00:00:00	834383	788335
2023-08-01 00:00:00	800909	755769
2023-09-01 00:00:00	712298	734338
2023-10-01 00:00:00	693332	671679
2023-11-01 00:00:00	731976	738000
2023-12-01 00:00:00	774527	767669

5 Representar la serie y los correlogramas. Según el resultado de los correlogramas, decidir qué modelo puede ser ajustado. Ajustar el modelo adecuado comprobando que sus residuales están incorrelados.

5.1. Correlogramas de la serie:

```
from statsmodels.tsa.stattools import adfuller #test AUGMENTED DICKEY-FULLER
from statsmodels.graphics.tsaplots import plot_pacf, plot_acf

fig, ax = plt.subplots(figsize=(10,5))
plot_acf(df['Demanda'], ax = ax, lags=25)
ax.set_xticks(range(26))
ax.set_xticklabels(range(26))
plt.show()

fig, ax = plt.subplots(figsize=(10,5))
plot_pacf(df['Demanda'], ax = ax, lags=25)
ax.set_xticks(range(26))
ax.set_xticklabels(range(26))
plt.show()
```



Se puede observar, especialmente en el correlograma de la autocorrelación, un marcado carácter estacional, un patrón que se repite cada 6 meses. Se tienen valores significativamente positivos en los retardos 6, 12, 18 y 24, mientras que en los retardos 3, 9, 15 y 21 son negativos. Además, en el correlograma de la autocorrelación parcial destaca el retardo 1 siendo significativamente positivo. Por tanto, se probará más adelante un modelo con parámetros $p = 1$, $m = 6$, $D = 1$.

También se comprueba la estacionariedad de los datos con el test Dickey-Fuller Aumentado:

```
def adf(datos):
    res = adfuller(datos)
    print('Estadístico T: ', res[0])
    print('p-valor: ', res[1])

    if res[1] < 0.05:
        print('Estacionario')
    else:
        print('No estacionario')
```

```
adf(df['Demanda'])
```

```
Estadístico T:  -1.2038513389392718
p-valor:  0.6719367726771039
No estacionario
```

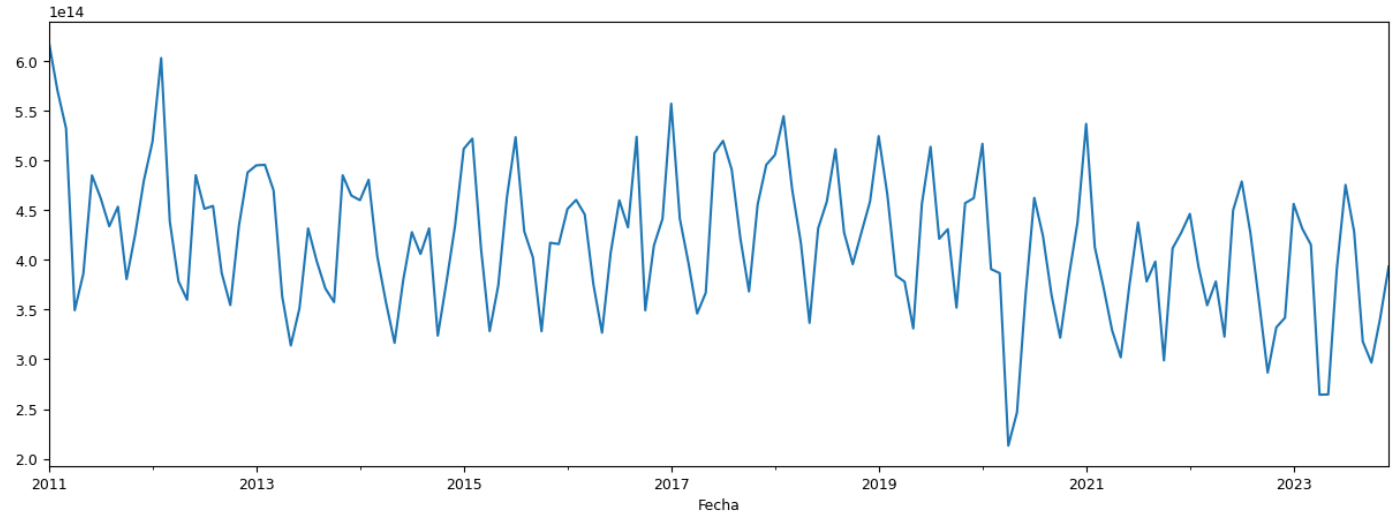
Siguiendo la metodología Box-Jenkins estudiada, a continuación se ha probado con una transformación Box-Cox ante la incertidumbre de la homogeneidad de la varianza. Posteriormente se reevalúan la estacionariedad de los datos transformados y los correlogramas:

```
data, lamda =  boxcox(df['Demanda'])
lamda #mostramos el valor del parámetro de lambda de Box-Cox
```

```
2.5472717710607555
```

```
df['boxcoxDemanda'] = data

adf(df['boxcoxDemanda'])

df['boxcoxDemanda'].plot()
```

```
Estadístico T:  -1.3231391391051854
p-valor:  0.6185207734257168
No estacionario
```



Adicionalmente, se añaden los datos transformados como una nueva variable en los conjuntos de entrenamiento y prueba:

```
train = train.join(df['boxcoxDemanda'])
test = test.join(df['boxcoxDemanda'])
```

Dado que la transformación por sí sola no garantiza la estacionariedad, se implementó un paso adicional de diferenciación sobre los datos transformados:

```
df['diffboxcoxDemanda'] = df['boxcoxDemanda'].diff()
adf(df['diffboxcoxDemanda'].dropna())
```

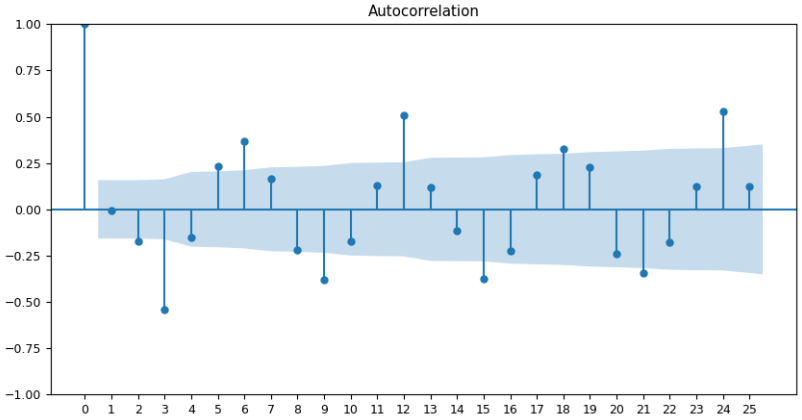
```
Estadístico T:  -10.00479027124606
p-valor:  1.8436718592975295e-17
Estacionario
```

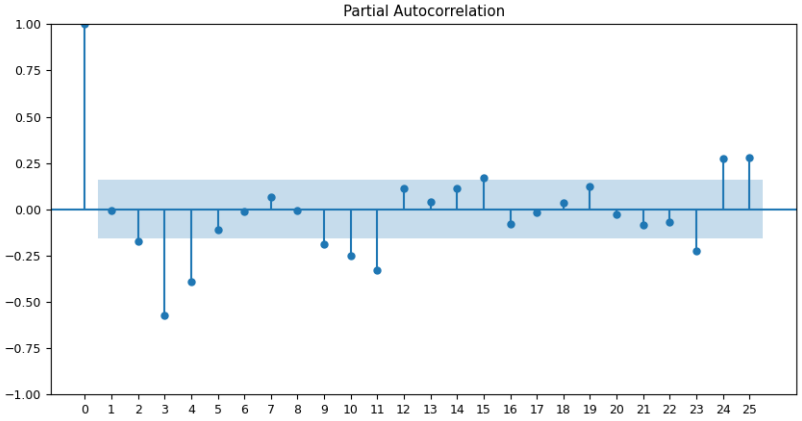
Los datos diferenciados sí son significativamente estacionarios.

Basado en el siguiente paso del método de Box-Jenkins, se representaron los correlogramas de los datos transformados y diferenciados:

```
fig, ax = plt.subplots(figsize=(10,5))
plot_acf(df['diffboxcoxDemanda'].dropna(), ax = ax, lags=25)
ax.set_xticks(range(26))
ax.set_xticklabels(range(26))
plt.show()
```

```
fig, ax = plt.subplots(figsize=(10,5))
plot_pacf(df['diffboxcoxDemanda'].dropna(), ax = ax, lags=25)
ax.set_xticks(range(26))
ax.set_xticklabels(range(26))
plt.show()
```





En vista de los resultados, parece que existe cierta estacionalidad también en la función de autocorrelación parcial, por tanto, se probará también con el parámetro $Q = 1$. En resumen, se ha generado el siguiente modelo ARIMA manual:

```
manual = ARIMA(train['boxcoxDemanda'], order=(1,1,0), seasonal_order=(0,1,1,6))
res_manual = manual.fit()
res_manual.summary()
```

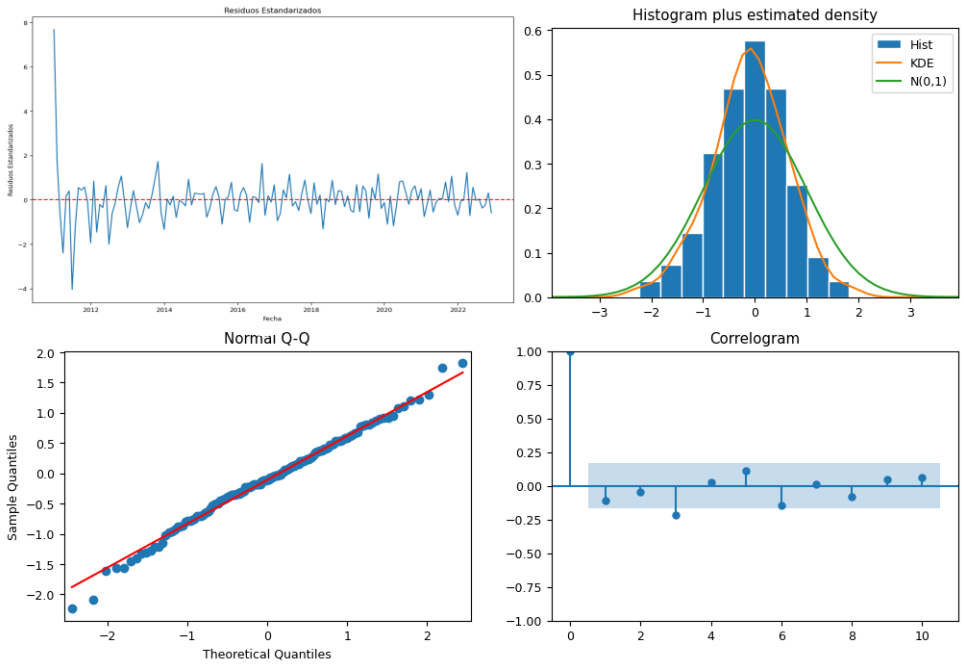
SARIMAX Results

Dep. Variable:	boxcoxDemanda	No. Observations:	144
Model:	ARIMA(1, 1, 0)x(0, 1, [1], 6)	Log Likelihood	-4536.970
Date:	Sun, 10 Mar 2024	AIC	9079.941
Time:	15:23:11	BIC	9088.701
Sample:	01-01-2011	HQIC	9083.501
	- 12-01-2022		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	-0.3008	0.120	-2.507	0.012	-0.536	-0.066
ma.S.L6	-0.8300	0.146	-5.703	0.000	-1.115	-0.545
sigma2	5.271e+27	3.17e-30	1.66e+57	0.000	5.27e+27	5.27e+27

Ljung-Box (L1) (Q):	1.70	Jarque-Bera (JB):	1.14
Prob(Q):	0.19	Prob(JB):	0.56
Heteroskedasticity (H):	0.59	Skew:	-0.18
Prob(H) (two-sided):	0.08	Kurtosis:	3.27

- Los coeficientes de este modelo son todos significativos.
- Test Ljung-Box: sugiere que no hay autocorrelaciones significativas en los residuos del modelo a nivel 1, indicando que el modelo ha capturado adecuadamente la autocorrelación en los datos. En otras palabras, los residuales son independientes y están incorrelados, tal y como se puede comprobar en el correlograma de la siguiente imagen del diagnóstico (ya que quedan todos los retardos más o menos dentro de las bandas de confianza).
- Test Jarque-Bera: indican que no hay suficiente evidencia como para rechazar la hipótesis nula de normalidad en los residuos, sugiriendo que los residuos del modelo se distribuyen de manera normal.
- Heterocedasticidad: podría sugerir cierta heterocedasticidad en los residuos, aunque no es estadísticamente significativa al nivel convencional del 5 %.



Por último, se calcula la métrica RMSE. Para ello, se va a invertir la transformación Box-Cox y así poder comparar con el resto de modelos:

```
from scipy.special import inv_boxcox

train_pred_manual_inv = inv_boxcox(train_pred_manual, lamda)
real_train_original = train['Demanda'][1:]
rmse_train_original = mean_squared_error(real_train_original, train_pred_manual_inv, squared=False)
print(f"RMSE para los datos TRAIN en la escala original: {rmse_train_original}")

forecast_pred = res_manual.get_forecast(N_test)
forecast = forecast_pred.predicted_mean
forecast_inv = inv_boxcox(forecast, lamda)
real_test_original = test['Demanda']
rmse_test_original = mean_squared_error(real_test_original, forecast_inv, squared=False)
print(f"RMSE para los datos TEST en la escala original: {rmse_test_original}")
```

```
RMSE para los datos TRAIN en la escala original: 44828.73415255811
RMSE para los datos TEST en la escala original: 50358.95985374023
```

6 Ajustar un modelo ARIMA con ajuste automático. Comparar los resultados con el manual y elegir el mejor. (1)

Se evaluaron varios modelos ARIMA con ajuste automático con diferentes periodos de estacionalidad m, para determinar el ajuste más adecuado a los datos. La Tabla 1 resume los modelos considerados:

Cuadro 1: Resumen de modelos ARIMA con ajuste automático.

m	Modelo	Parámetros no significativos	Prob(Q)	AIC	BIC
3	ARIMA(1,1,1)(3,0,1)[3]	ar.S.L9 (0.058)	0.24	9428.554	9449.294
6	ARIMA(1,1,1)(2,0,2)[6]	ar.S.L6 (0.678), ma.S.L6 (0.764)	0.34	9438.686	9459.426
12	ARIMA(1,1,1)(1,0,1)[12]	Ninguno	0.71	9438.453	9453.267

El modelo ARIMA(1,1,1)(1,0,1)[12], correspondiente a $m = 12$, fue seleccionado por presentar una combinación óptima de menor número de parámetros no significativos, junto con valores aceptables en las pruebas de heterocedasticidad y autocorrelación (Ljung-Box).

```
import pmdarima as pm
modelo_auto1 = pm.auto_arima(train['boxcoxDemanda'],max_D=12, max_d=2, max_P=3,
max_p=3,max_Q=3, max_q=3, max_order=14,m=12,trace=True, suppress_warnings=True, seasonal=True)
modelo_auto1.summary()
```

SARIMAX Results

Dep. Variable:	y	No. Observations:	144
Model:	SARIMAX(1, 1, 1)x(1, 0, 1, 12)	Log Likelihood	-4714.227
Date:	Mon, 11 Mar 2024	AIC	9438.453
Time:	17:23:27	BIC	9453.267
Sample:	01-01-2011	HQIC	9444.473
	- 12-01-2022		
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	0.4946	0.087	5.692	0.000	0.324	0.665
ma.L1	-0.9699	0.051	-19.200	0.000	-1.069	-0.871
ar.S.L12	0.9506	0.061	15.484	0.000	0.830	1.071
ma.S.L12	-0.8261	0.125	-6.589	0.000	-1.072	-0.580
sigma2	3.205e+27	1.59e-29	2.02e+56	0.000	3.2e+27	3.2e+27

Ljung-Box (L1) (Q):	0.14	Jarque-Bera (JB):	0.97
Prob(Q):	0.71	Prob(JB):	0.62
Heteroskedasticity (H):	0.63	Skew:	0.14
Prob(H) (two-sided):	0.11	Kurtosis:	3.29

Para comparar este modelo con ajuste automático con el manual de la sección anterior, vamos a calcular el RMSE como se ha hecho anteriormente invirtiendo la transformación de Box-Cox:

```
test_pred , confint = modelo_auto1.predict(n_periods=N_test, return_conf_int=True)
train_pred = modelo_auto1.predict_in_sample(start=1, end=-1)

train_pred_auto1_inv = inv_boxcox(train_pred ,lamda)
forecast_auto1_inv = inv_boxcox(test_pred, lamda)

real_train_original_auto1 = train['Demanda'][1:]
real_test_original_auto1 = test['Demanda']

rmse_train_original_auto1 = mean_squared_error(real_train_original_auto1, train_pred_auto1_inv,squared=False)
print(f"RMSE para los datos TRAIN en la escala original: {rmse_train_original_auto1}")

rmse_test_original_auto1 = mean_squared_error(real_test_original_auto1, forecast_auto1_inv, squared=False)
print(f"RMSE para los datos TEST en la escala original: {rmse_test_original_auto1}")
```

```
RMSE para los datos TRAIN en la escala original: 37367.4422870198
RMSE para los datos TEST en la escala original: 37745.779442195984
```

Por último, se compara el modelo manual ARIMA(1, 1, 0)(0, 1, 1, 6) y el modelo de ajuste automático elegido ARIMA(1, 1, 1)(1, 0, 1, 12). La elección del mejor modelo se basa en una evaluación de su rendimiento predictivo (RMSE), la significancia de sus parámetros, y el cumplimiento de las suposiciones estadísticas:

Cuadro 2: Comparación de Modelos ARIMA Manual y Ajuste Automático

Criterio	**Modelo Manual**	**Modelo Automático**
RMSE Train	44828.73	37367.44
RMSE Test	50358.96	37745.78
AIC	9079.941	9438.453
BIC	9088.701	9453.267
Prob(Q) Ljung-Box	0.19	0.71
Prob(JB) Jarque-Bera	0.56	0.62
Heteroskedasticity (H)	0.59	0.63

El modelo de ajuste automático muestra un mejor rendimiento predictivo (no solamente son más bajos los valores de RMSE, sino que también son muy similares tanto para los datos de train como para los de test). El modelo manual presenta valores más altos de esta métrica, además la diferencia entre dicho valor con respecto a los conjuntos de train (44828,73) y el de test (50358,73) indica sobreajuste. Por ello, a pesar de tener un AIC y BIC ligeramente más alto (tiene un parámetro más) se considera mejor el de ajuste automático.

Desde el punto de vista de las pruebas estadísticas de los residuos, ambos modelos pasan la prueba de Ljung-Box, indicando que no hay autocorrelaciones significativas en los residuos, siendo mejor para el modelo automático.

Las pruebas de Jarque-Bera y de heteroscedasticidad también muestran buenos resultados para ambos modelos, sugiriendo que los residuos son aproximadamente normales y que la varianza de los residuos es constante a lo largo del tiempo.

En conclusión, a pesar de que ambos modelos satisfacen las condiciones estadísticas básicas, el modelo ajustado automáticamente destaca por su mejor precisión predictiva (RMSE) y su consistencia entre los conjuntos de entrenamiento y prueba, lo que demuestra su capacidad de generalización y evita el sobreajuste. Por esto se convierte en la elección preferente.

7 Escribir la expresión algebraica del modelo ajustado con los parámetros estimados. (1)

El modelo elegido en la sección anterior ARIMA(1, 1, 1)(1, 0, 1, 12) tiene la siguiente notación algebraica usando el operador retardo B:

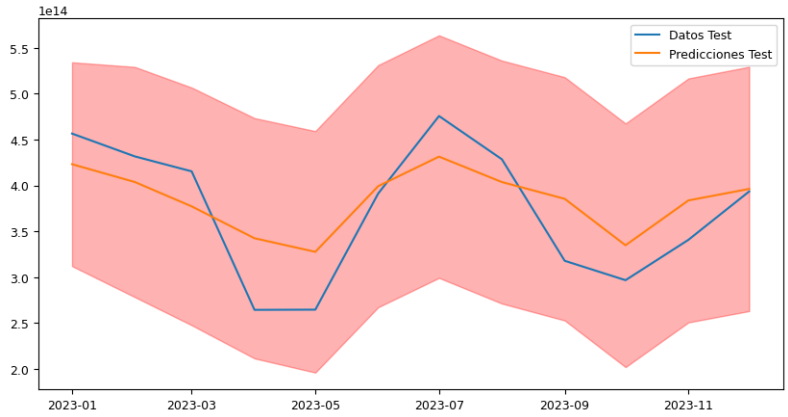
$$(1 - \Phi_1 B^{12})(1 - \phi_1 B)(1 - B)Y_t = (1 + \Theta_1 B^{12})(1 + \theta_1 B)Z_t$$

donde Z_t es el término de error en el tiempo t, $\Phi_1 = 0.9506$, $\phi_1 = 0.4946$, $\Theta_1 = -0.8261$ y $\theta_1 = -0.9699$.

8 Calcular las predicciones y los intervalos de confianza para las unidades de tiempo que se considere oportuno, dependiendo de la serie, siguientes al último valor observado. Representarlas gráficamente. (1)

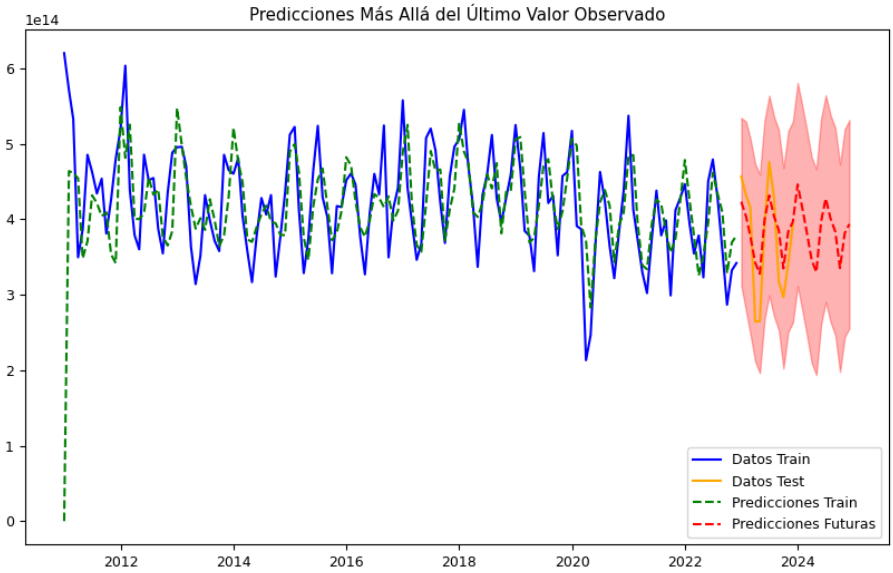
Primero, se van a representar las predicciones de los datos test, los intervalos de confianza y los datos test reales que reservamos de los datos observados:

```
test_pred , confint = modelo_auto1.predict(n_periods=N_test, return_conf_int=True)
fig, ax = plt.subplots(figsize=(10,5))
ax.plot(test.index, test['boxcoxDemanda'], label='test data')
ax.plot(test.index, test_pred, label='forecast')
ax.fill_between(test.index, confint[:,0], confint[:,1], color='red', alpha=0.3)
ax.legend()
```



Como se pide calcular más allá de los datos observados (el último dato test observado es en diciembre de 2023), se va a realizar un pronóstico de 2023 y 2024:

```
in_sample_forecast = modelo_auto1.predict_in_sample()
in_sample_index = train.index

n_extras = 24
forecast, confint = modelo_auto1.predict(n_periods=n_extras, return_conf_int=True)
ultima_fecha = train.index[-1]
pred_index = pd.date_range(start=ultima_fecha, periods=n_extras + 1, freq='MS')[1:]

fig, ax = plt.subplots(figsize=(10, 6))
ax.plot(train.index, train['boxcoxDemanda'], label='Datos Train', color='blue')
ax.plot(test.index, test['boxcoxDemanda'], label='Datos Test', color='orange')
ax.plot(in_sample_index, in_sample_forecast, label='Predicciones Train', color='green', linestyle='--')
ax.plot(pred_index, forecast, label='Predicciones Futuras', color='red', linestyle='--')
ax.fill_between(pred_index, confint[:, 0], confint[:, 1], color='red', alpha=0.3)
ax.set_title('Predicciones Más Allá del último Valor Observado')
ax.legend(loc='lower right')

plt.show()
```



9 Comparar las predicciones obtenidas con cada uno de los métodos (suavizado y ARIMA) con los valores observados que habíamos reservados antes. Conclusiones. (1)

En el apartado 4.3. se calculó el RMSE del método de suavizado Holt-Winters exponencial: 34874,54. Este valor, si es comparado con el del método ARIMA de ajuste automático (37745,78), sugiere que el método de suavizado destaca por tener un menor error en sus predicciones.

Además, el método de Holt-Winters está específicamente diseñado para capturar tanto la tendencia como la estacionalidad, siendo adecuado para el tipo de serie temporal analizado ya que muestra patrones estacionales claros (influenciados por factores como las estaciones del año y los hábitos de consumo).

Por otra parte, Holt-Winters destaca por su simplicidad computacional y eficacia en capturar la tendencia y estacionalidad de forma intuitiva. Considerando la complejidad computacional, la facilidad de interpretación y la especificidad de la aplicación a series con marcado carácter estacional, el suavizado de Holt-Winters lo considero como la opción más pragmática para nuestro análisis.