

# sarima-forecast

June 5, 2025

```
[1]: import pandas as pd
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_absolute_error
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
import time
import os

warnings.filterwarnings("ignore")

[2]: # 1. Cargar datos
df_2022 = pd.read_csv('2022_limpio.csv', sep=';',
    ↳ parse_dates=['fecha_de_factura'], dayfirst=True)
df_2023 = pd.read_csv('2023_limpio.csv', sep=';',
    ↳ parse_dates=['fecha_de_factura'], dayfirst=True)
df = pd.concat([df_2022, df_2023])

[3]: # 2. Limpieza
df['cantidad_neta'] = pd.to_numeric(df['cantidad_neta'], errors='coerce')
df = df.dropna(subset=['cantidad_neta'])

[4]: # 3. Agrupar ventas diarias por cliente y ciudad
ventas_diarias = df.groupby(['fecha_de_factura', 'solicitante',
    ↳ 'CIUDAD'])['cantidad_neta'].sum().reset_index()
ventas_diarias.columns = ['fecha', 'solicitante', 'CIUDAD', 'cantidad_neta']

[5]: # 4. Generar lista de combinaciones cliente + ciudad con al menos 180 días
clientes_unicos = ventas_diarias.groupby(['solicitante', 'CIUDAD']).size().
    ↳ reset_index(name='n_dias')
clientes_filtrables = clientes_unicos[clientes_unicos['n_dias'] >= 180]

[6]: # 5. Fecha de corte
fecha_corte = pd.to_datetime("2023-07-01")

[7]: # 6. Evaluar cada combinación con SARIMA
resultados = []
```

```

[8]: # 7. Bucle por cada cliente + ciudad
predictions_agg = []

if not os.path.exists('plots'):
    os.makedirs('plots')

for _, row in clientes_filtrables.iterrows():
    cliente = row['solicitante']
    ciudad = row['CIUDAD']

    subset = ventas_diarias[
        (ventas_diarias['solicitante'] == cliente) &
        (ventas_diarias['CIUDAD'] == ciudad)
    ].sort_values('fecha')

    subset = subset.set_index('fecha').asfreq('D')
    subset['cantidad_neta'] = subset['cantidad_neta'].fillna(0)

    train = subset[subset.index < fecha_corte]
    test = subset[subset.index >= fecha_corte]

    if len(train) < 180 or len(test) < 30:
        continue

    try:
        # Entrenar SARIMA
        start_train_time = time.time()
        model = SARIMAX(train['cantidad_neta'],
                        order=(1, 1, 1),
                        seasonal_order=(1, 1, 1, 7), # semanal
                        enforce_stationarity=False,
                        enforce_invertibility=False)
        results = model.fit(dispatch=False)
        end_train_time = time.time()
        train_time_seconds = end_train_time - start_train_time

        # Pronóstico
        start_inference_time = time.time()
        pred = results.predict(start=test.index[0], end=test.index[-1])
        end_inference_time = time.time()
        inference_time_seconds = end_inference_time - start_inference_time

        # Evaluación
        mae = mean_absolute_error(test['cantidad_neta'], pred)
        promedio_real = test['cantidad_neta'].mean()
        mae_relativo = mae / promedio_real if promedio_real != 0 else None

```

```

    accuracy = 1 - mae / promedio_real if promedio_real != 0 else None
    precision = 1 - mae / test['cantidad_neta'].mean() if
↪test['cantidad_neta'].mean() != 0 else None
    recall = 1 - mae / test['cantidad_neta'].mean() if
↪test['cantidad_neta'].mean() != 0 else None
    f1_score = 2 * (precision * recall) / (precision + recall) if
↪(precision is not None and recall is not None) else None

    comparacion_filtrada = test[test['cantidad_neta'] != 0]
    pred_filtrada = pred[test['cantidad_neta'] != 0]
    if len(comparacion_filtrada) == 0:
        mape = None
    else:
        mape = (abs((comparacion_filtrada['cantidad_neta'] - pred_filtrada)
↪/ comparacion_filtrada['cantidad_neta'])).mean() * 100

    resultados.append({
        'cliente': cliente,
        'ciudad': ciudad,
        'MAE': mae,
        'Promedio_ventas_test': promedio_real,
        'MAE_relativo': mae_relativo,
        'MAPE (%)': mape,
        'accuracy': accuracy,
        'precision': precision,
        'recall': recall,
        'f1_score': f1_score,
        'observaciones': len(subset),
        'tiempo_entrenamiento_segundos': train_time_seconds,
        'tiempo_inferencia_segundos': inference_time_seconds
    })

    data_plot = pd.DataFrame({
        'fecha': test.index,
        'real': test['cantidad_neta'],
        'pred': pred
    })
    data_plot['ciudad'] = ciudad
    # data_plot['cliente'] = cliente # se omite cliente para no generar
↪tantas gráficas
    predictions_agg.append(data_plot)

    # Graficar Serie temporal: Ventas reales vs. Predicción (se guardan en
↪carpeta plots)
    plt.figure(figsize=(12, 6))

```

```

plt.plot(test['cantidad_neta'].index, test['cantidad_neta'],
↳label='Ventas Reales')
plt.plot(test['cantidad_neta'].index, pred, label='Predicción',
↳linestyle='--')
plt.title(f'Ventas Reales vs. Predicción - Cliente: {cliente} | Ciudad:
↳{ciudad}')
plt.xlabel('Fecha')
plt.ylabel('Cantidad Neta')
plt.legend()
plt.savefig(f'plots/{ciudad}_cliente_{cliente}_real_vs_pred.png',
↳bbox_inches='tight')
plt.close()

# Graficar Scatter plot: Predicción vs. Ventas reales (se guardan en
↳carpeta plots)
plt.figure(figsize=(8, 6))
plt.scatter(test['cantidad_neta'], pred, alpha=0.6)
plt.plot([test['cantidad_neta'].min(), test['cantidad_neta'].max()],
↳[test['cantidad_neta'].min(), test['cantidad_neta'].max()], 'r--')
plt.title(f'Scatter: Predicción vs. Ventas reales - Cliente: {cliente}
↳| Ciudad: {ciudad}')
plt.xlabel('Ventas Reales')
plt.ylabel('Predicción')
plt.savefig(f'plots/{ciudad}_cliente_{cliente}_scatter.png',
↳bbox_inches='tight')
plt.close()

except Exception as e:
    print(f"Error en {cliente} - {ciudad}: {e}")
    continue

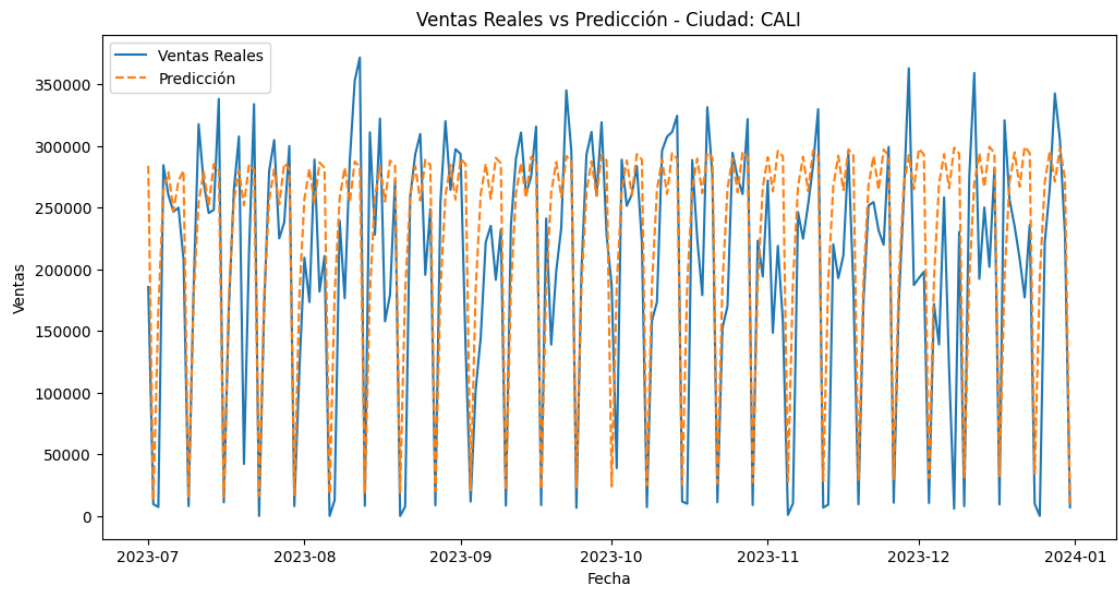
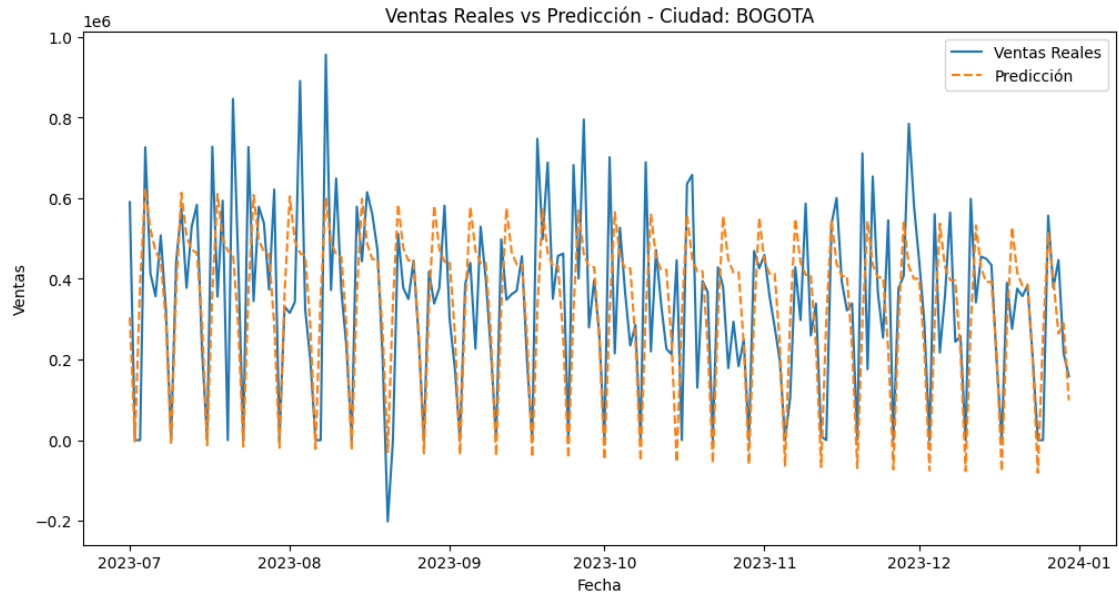
```

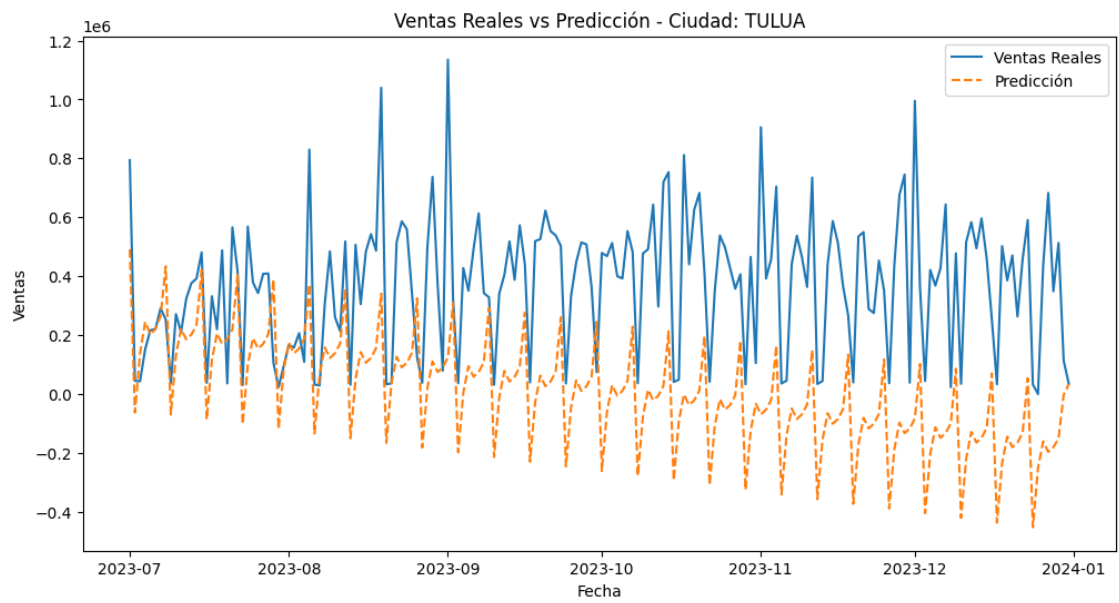
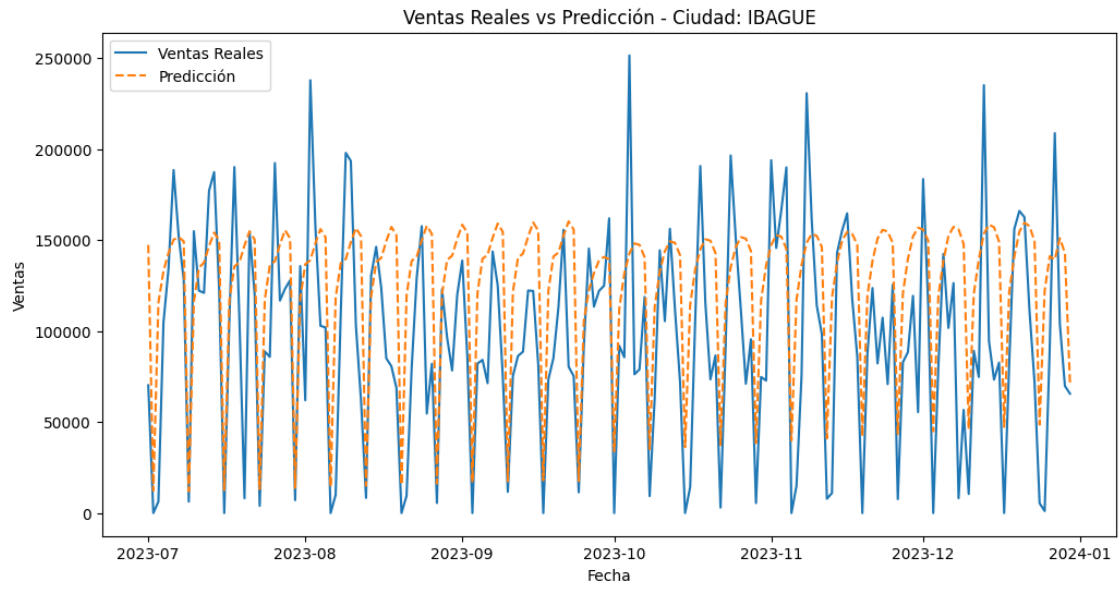
[9]: `df_predictions = pd.concat(predictions_agg, ignore_index=True)`

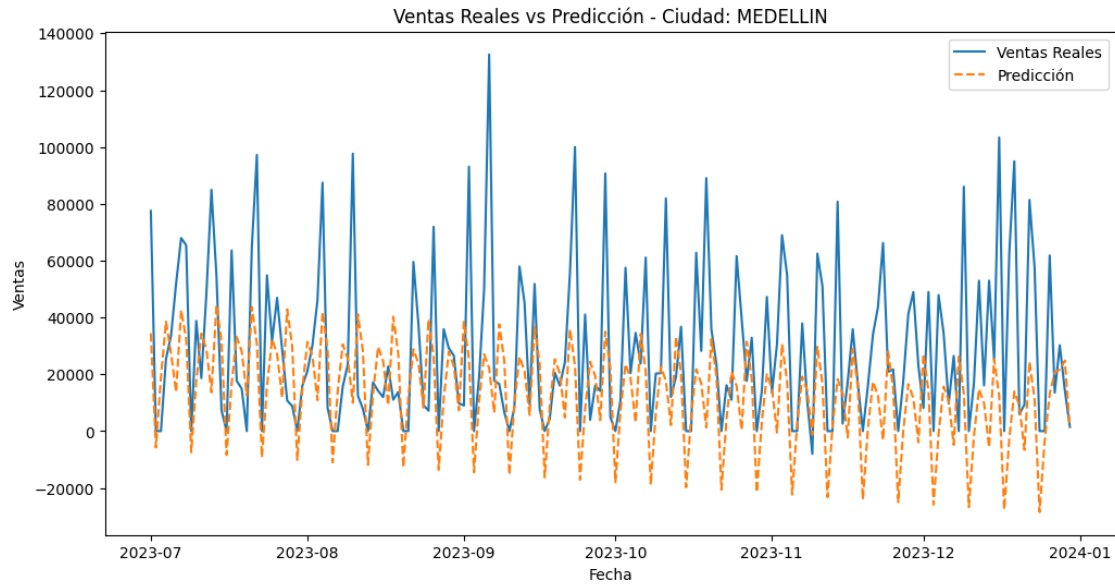
```

# 8. Gráfica por cada ciudad
ciudades = df_predictions['ciudad'].unique()
for city in ciudades:
    group = df_predictions[df_predictions['ciudad'] == city].groupby('fecha').
↳agg({'real': 'sum', 'pred': 'sum'}).reset_index()
    plt.figure(figsize=(12,6))
    plt.plot(group['fecha'], group['real'], label='Ventas Reales')
    plt.plot(group['fecha'], group['pred'], label='Predicción', linestyle='--')
    plt.title(f'Ventas Reales vs Predicción - Ciudad: {city}')
    plt.xlabel('Fecha')
    plt.ylabel('Ventas')
    plt.legend()
    plt.show()

```

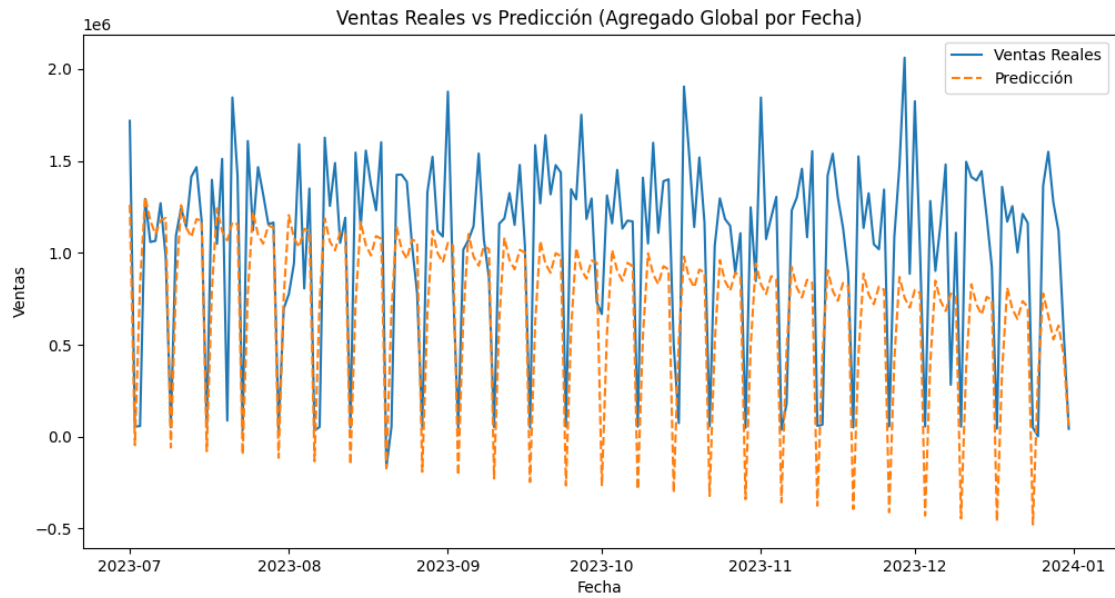






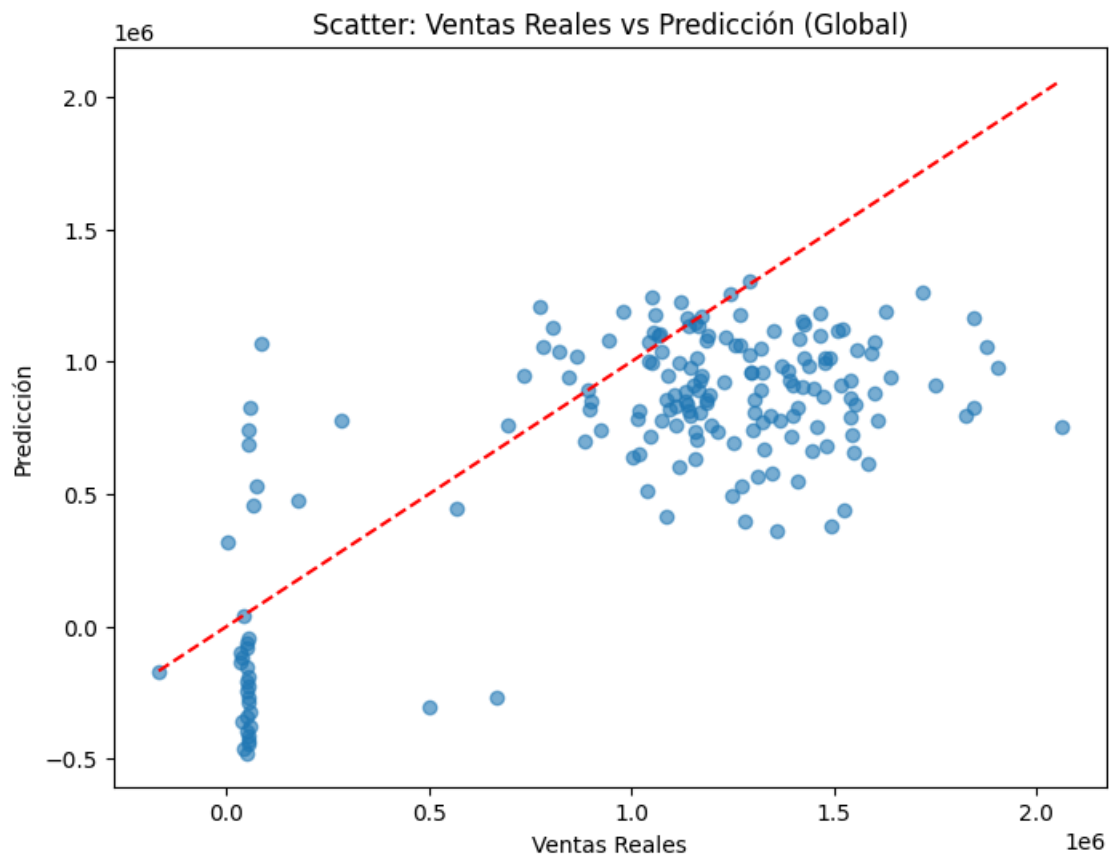
```
[10]: # 9. Grafica Ventas Reales vs Predicción (Agregado Global por Fecha)
df_total = df_predictions.groupby('fecha').agg({'real': 'sum', 'pred': 'sum'}).
    ↪reset_index()

plt.figure(figsize=(12,6))
plt.plot(df_total['fecha'], df_total['real'], label='Ventas Reales')
plt.plot(df_total['fecha'], df_total['pred'], label='Predicción',
    ↪linestyle='--')
plt.title('Ventas Reales vs Predicción (Agregado Global por Fecha)')
plt.xlabel('Fecha')
plt.ylabel('Ventas')
plt.legend()
plt.show()
```



```
[11]: # 11. Grafica Scatter: Ventas Reales vs Predicción (Global)
plt.figure(figsize=(8,6))
plt.scatter(df_total['real'], df_total['pred'], alpha=0.6)
plt.plot([df_total['real'].min(), df_total['real'].max()],
         [df_total['real'].min(), df_total['real'].max()],
         'r--')
plt.title('Scatter: Ventas Reales vs Predicción (Global)')
plt.xlabel('Ventas Reales')
plt.ylabel('Predicción')
plt.show()
```





[12]: # 12. Mostrar resultados

```
df_resultados = pd.DataFrame(resultados).sort_values('MAE')
df_resultados
```

```
[12]:
```

	cliente	ciudad	MAE	Promedio_ventas_test	MAE_relativo \
660	1032353.0	BOGOTA	8.017327	1.099291	7.293181
174	1008423.0	IBAGUE	12.578714	38.360656	0.327907
43	1001221.0	TULUA	13.016912	42.388889	0.307083
456	1027079.0	BOGOTA	13.623633	47.083333	0.289351
656	1032204.0	IBAGUE	13.925435	21.546961	0.646283
..	...	...	...	...	...
122	1004155.0	CALI	44906.685363	44697.125683	1.004688
638	1031842.0	BOGOTA	47894.036745	37203.666667	1.287347
35	1000936.0	BOGOTA	59155.057352	71675.781421	0.825314
256	1018222.0	BOGOTA	98723.972513	86064.327869	1.147095
257	1018222.0	TULUA	379342.509776	227240.918033	1.669341

	MAPE (%)	accuracy	precision	recall	f1_score	observaciones \
660	114.139268	-6.293181	-6.293181	-6.293181	-6.293181	580
174	19.436135	0.672093	0.672093	0.672093	0.672093	725

43	22.332359	0.692917	0.692917	0.692917	0.692917	722
456	20.187598	0.710649	0.710649	0.710649	0.710649	724
656	40.911528	0.353717	0.353717	0.353717	0.353717	668
..	...	...	...	...	...	
122	612.759593	-0.004688	-0.004688	-0.004688	-0.004688	728
638	1317.832300	-0.287347	-0.287347	-0.287347	-0.287347	724
35	427.735717	0.174686	0.174686	0.174686	0.174686	727
256	454.288381	-0.147095	-0.147095	-0.147095	-0.147095	727
257	980.923251	-0.669341	-0.669341	-0.669341	-0.669341	729

	tiempo_entrenamiento_segundos	tiempo_inferencia_segundos
660	0.447444	0.000000
174	0.785737	0.006985
43	0.762930	0.003717
456	0.653875	0.006026
656	0.724003	0.000000
..	...	...
122	0.418740	0.003676
638	0.300768	0.000000
35	0.474579	0.006765
256	0.389498	0.005022
257	0.271512	0.004955

[688 rows x 13 columns]

```
[13]: # 13. Clasificación por MAPE
def clasificar_mape(mape):
    if pd.isna(mape):
        return 'Sin datos'
    elif mape < 10:
        return 'Excelente'
    elif mape < 20:
        return 'Buena'
    elif mape < 50:
        return 'Aceptable'
    else:
        return 'Mala'

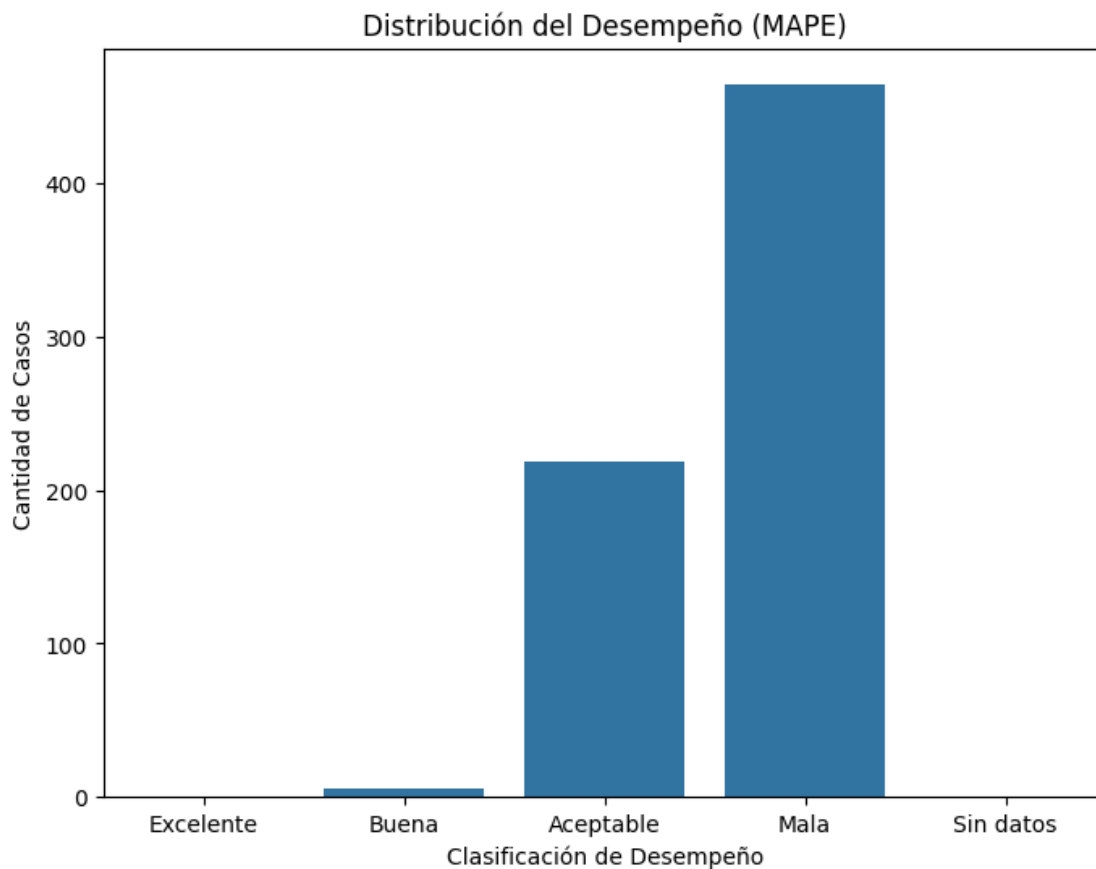
df_resultados['desempeño'] = df_resultados['MAPE (%)'].apply(clasificar_mape)
```

```
[14]: print("\nDistribución del desempeño (porcentaje):")
print(df_resultados['desempeño'].value_counts(normalize=True) * 100)
print(df_resultados['MAPE (%)'].mean())
```

Distribución del desempeño (porcentaje):  
desempeño

```
Mala          67.441860
Aceptable     31.686047
Buena          0.872093
Name: proportion, dtype: float64
145.94025954029976
```

```
[15]: # 14. Distribución del desempeño (por MAPE)
plt.figure(figsize=(8, 6))
sns.countplot(x='desempeño', data=df_resultados, order=['Excelente', 'Buena', 'Aceptable', 'Mala', 'Sin datos'])
plt.title('Distribución del Desempeño (MAPE)')
plt.xlabel('Clasificación de Desempeño')
plt.ylabel('Cantidad de Casos')
plt.show()
```



```
[16]: # 15. Clasificación por accuracy
def clasificar_accuracy(accuracy):
    if pd.isna(accuracy):
        return 'Sin datos'
    elif accuracy > 0.9:
```

```

        return 'Excelente'
    elif accuracy > 0.75:
        return 'Buena'
    elif accuracy > 0.6:
        return 'Aceptable'
    else:
        return 'Mala'

```

```

df_resultados['desempeño_accuracy'] = df_resultados['accuracy'].
    ↪ apply(clasificar_accuracy)

```

```

[17]: print("\nDistribución del desempeño de accuracy (porcentaje):")
      print(df_resultados['desempeño_accuracy'].value_counts(normalize=True) * 100)
      print(df_resultados['accuracy'].mean())

```

```

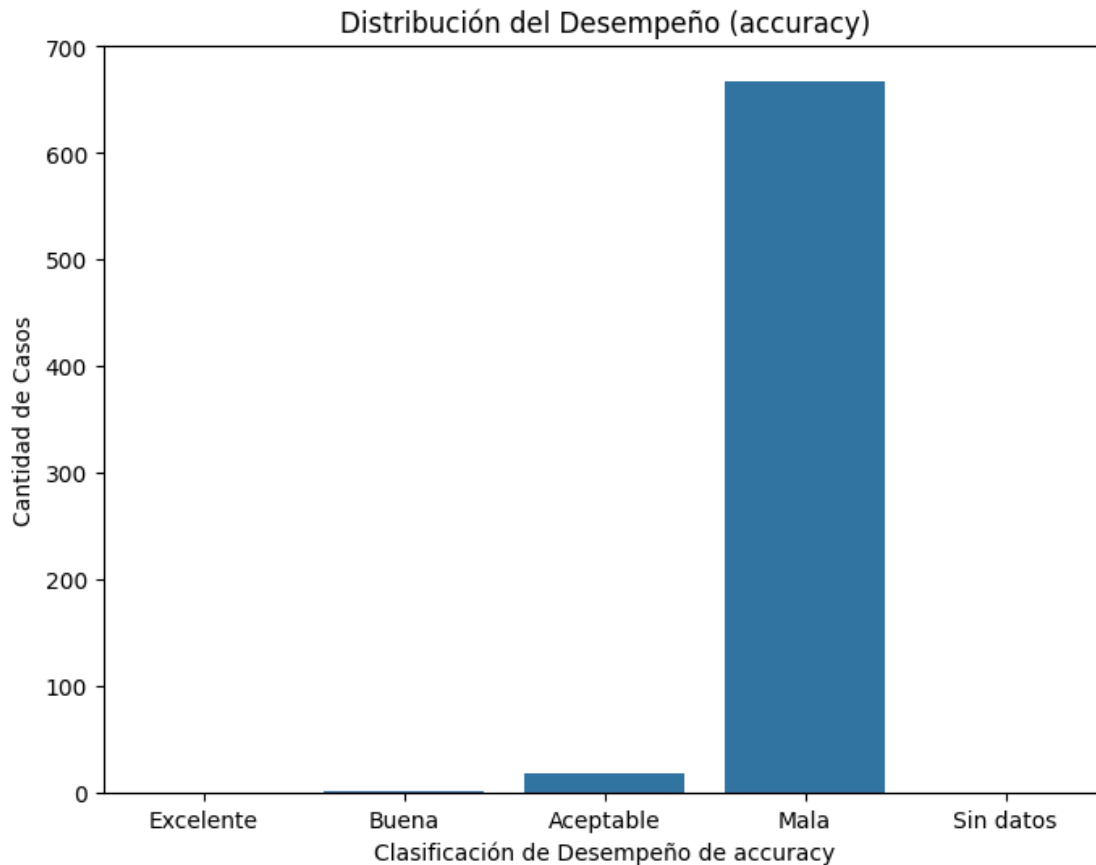
Distribución del desempeño de accuracy (porcentaje):
desempeño_accuracy
Mala          96.947674
Aceptable     2.761628
Buena         0.290698
Name: proportion, dtype: float64
-0.3037574782030685

```

```

[18]: # 16. Distribución del desempeño (por accuracy)
      plt.figure(figsize=(8, 6))
      sns.countplot(x='desempeño_accuracy', data=df_resultados, order=['Excelente',
      ↪ 'Buena', 'Aceptable', 'Mala', 'Sin datos'])
      plt.title('Distribución del Desempeño (accuracy)')
      plt.xlabel('Clasificación de Desempeño de accuracy')
      plt.ylabel('Cantidad de Casos')
      plt.show()

```



```
[19]: # Análisis de Complejidad (Tiempos de Entrenamiento e Inferencia)

print("\n--- Resumen de Tiempos de Complejidad ---")

# Filtrar solo casos donde se pudo calcular el tiempo (evitar errores None o
↳NaN)
df_tiempos = df_resultados.dropna(subset=['tiempo_entrenamiento_segundos',
↳'tiempo_inferencia_segundos'])

if not df_tiempos.empty:
    # Estadísticas para el tiempo de entrenamiento
    total_train_time = df_tiempos['tiempo_entrenamiento_segundos'].sum()
    avg_train_time = df_tiempos['tiempo_entrenamiento_segundos'].mean()
    median_train_time = df_tiempos['tiempo_entrenamiento_segundos'].median()
    max_train_time = df_tiempos['tiempo_entrenamiento_segundos'].max()
    min_train_time = df_tiempos['tiempo_entrenamiento_segundos'].min()

    print(f"\n**Tiempos de Entrenamiento (en segundos) para {len(df_tiempos)}
↳modelos:**")
```

```

    print(f" Tiempo Total: {total_train_time:.2f} segundos ({total_train_time /
↪ 60:.2f} minutos)")
    print(f" Promedio por modelo: {avg_train_time:.4f} segundos")
    print(f" Mediana por modelo: {median_train_time:.4f} segundos")
    print(f" Máximo por modelo: {max_train_time:.4f} segundos")
    print(f" Mínimo por modelo: {min_train_time:.4f} segundos")

    # Estadísticas para el tiempo de inferencia
    total_inference_time = df_tiempos['tiempo_inferencia_segundos'].sum()
    avg_inference_time = df_tiempos['tiempo_inferencia_segundos'].mean()
    median_inference_time = df_tiempos['tiempo_inferencia_segundos'].median()
    max_inference_time = df_tiempos['tiempo_inferencia_segundos'].max()
    min_inference_time = df_tiempos['tiempo_inferencia_segundos'].min()

    print(f"\n**Tiempos de Inferencia (en segundos) para {len(df_tiempos)}_
↪modelos:**")
    print(f" Tiempo Total: {total_inference_time:.2f} segundos_
↪({total_inference_time / 60:.2f} minutos)")
    print(f" Promedio por modelo: {avg_inference_time:.4f} segundos")
    print(f" Mediana por modelo: {median_inference_time:.4f} segundos")
    print(f" Máximo por modelo: {max_inference_time:.4f} segundos")
    print(f" Mínimo por modelo: {min_inference_time:.4f} segundos")

    # Visualización de la distribución de tiempos
    plt.figure(figsize=(14, 6))
    plt.subplot(1, 2, 1)
    sns.histplot(df_tiempos['tiempo_entrenamiento_segundos'], bins=30, kde=True)
    plt.title('Distribución del Tiempo de Entrenamiento por Modelo')
    plt.xlabel('Tiempo (segundos)')
    plt.ylabel('Frecuencia')

    plt.subplot(1, 2, 2)
    sns.histplot(df_tiempos['tiempo_inferencia_segundos'], bins=30, kde=True)
    plt.title('Distribución del Tiempo de Inferencia por Modelo')
    plt.xlabel('Tiempo (segundos)')
    plt.ylabel('Frecuencia')
    plt.tight_layout()
    plt.show()

else:
    print("No hay datos de tiempo disponibles para analizar.")

```

--- Resumen de Tiempos de Complejidad ---

```

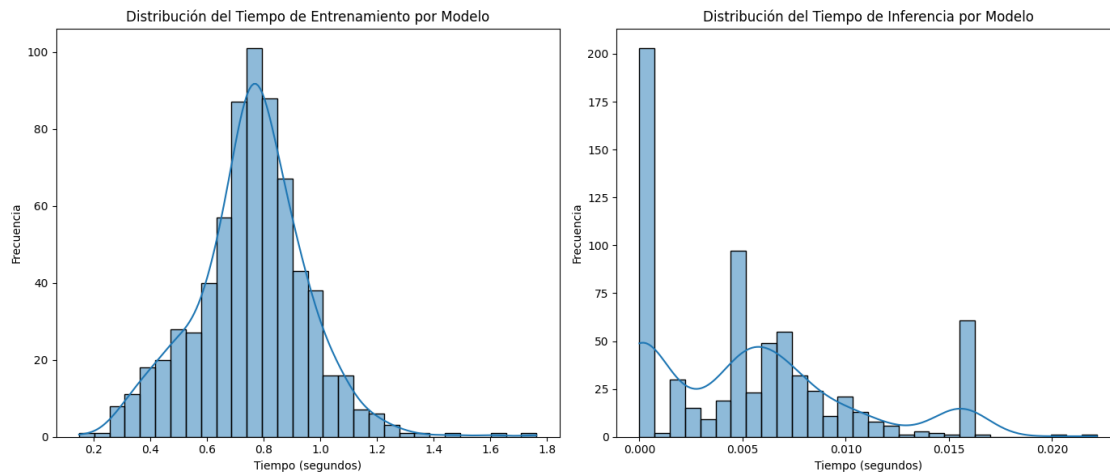
**Tiempos de Entrenamiento (en segundos) para 688 modelos:**
Tiempo Total: 521.95 segundos (8.70 minutos)

```

Promedio por modelo: 0.7586 segundos  
Mediana por modelo: 0.7643 segundos  
Máximo por modelo: 1.7629 segundos  
Mínimo por modelo: 0.1491 segundos

**\*\*Tiempos de Inferencia (en segundos) para 688 modelos:\*\***

Tiempo Total: 3.68 segundos (0.06 minutos)  
Promedio por modelo: 0.0053 segundos  
Mediana por modelo: 0.0050 segundos  
Máximo por modelo: 0.0222 segundos  
Mínimo por modelo: 0.0000 segundos



```
[20]: # Guardar resultados en Excel
df_resultados.to_excel('resultados_modelo_sarima.xlsx', index=False)
```