2) Time series

August 23, 2023

1 Introducción

En esta notebook se analizará la serie de tiempo de BTC. El desarrollo consta de tres partes: la primera prepara el dataset, la segunda aplica modelos de predicción y la tercera analiza estacionariedad tanto para la serie de tiempo como para los residuos de los modelos

2 1) Preparación previa

2.1 Carga de librerías

[1]:
```
import pandas as pd
import numpy as np

import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import seaborn as sns
%matplotlib inline

import statsmodels.api as sm
import statsmodels.formula.api as smf
import statsmodels.tsa.api as smt
from statsmodels.tsa.stattools import adfuller, acf, pacf
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima.model import ARIMA
# from statsmodels.graphics.tsaplots import plot_predict
from statsmodels.tsa.holtwinters import SimpleExpSmoothing

from scipy import stats
from statistics import mode

from sklearn.model_selection import train_test_split, GridSearchCV,␣
↪TimeSeriesSplit
from sklearn.metrics import mean_squared_error
from sklearn.preprocessing import StandardScaler

# Se debe instalar pmdarima
from pmdarima import auto_arima #!pip install pmdarima
```

```
# Se debe instalar prophet
from prophet import Prophet #!pip install prophet
from prophet.diagnostics import cross_validation
import itertools
from prophet.diagnostics import performance_metrics

import warnings
warnings.filterwarnings('ignore')
import logging
```

2.2 Lectura y armado del dataset

[2]:
```
df = pd.read_csv('https://raw.githubusercontent.com/Agustin-Bulzomi/Projects/
↪main/Programming/Digital%20House%20(Python)/Support%20Files/Final%20Project/
↪coin_Bitcoin.csv', delimiter = ',')
```

Se realizan las modificaciones del dataset pertinentes para el análisis de series de tiempo

[3]:
```
df['Date'] = pd.to_datetime(df['Date'])
df.index = pd.PeriodIndex(df.Date, freq = 'D')
df.head()
```

[3]:
```
            SNo     Name Symbol       Date        High         Low  \
Date
2013-04-29    1  Bitcoin    BTC 2013-04-29  147.488007  134.000000
2013-04-30    2  Bitcoin    BTC 2013-04-30  146.929993  134.050003
2013-05-01    3  Bitcoin    BTC 2013-05-01  139.889999  107.720001
2013-05-02    4  Bitcoin    BTC 2013-05-02  125.599998   92.281898
2013-05-03    5  Bitcoin    BTC 2013-05-03  108.127998   79.099998

                  Open       Close  Volume     Marketcap
Date
2013-04-29  134.444000  144.539993     0.0  1.603769e+09
2013-04-30  144.000000  139.000000     0.0  1.542813e+09
2013-05-01  139.000000  116.989998     0.0  1.298955e+09
2013-05-02  116.379997  105.209999     0.0  1.168517e+09
2013-05-03  106.250000   97.750000     0.0  1.085995e+09
```

Se agrega la columna Time index, necesaria para algunos modelos futuros

[4]:
```
df['timeIndex'] = pd.Series(np.arange(len(df['Close'])), index = df.index)

df.head()
```

[4]:
```
            SNo     Name Symbol       Date        High         Low  \
Date
2013-04-29    1  Bitcoin    BTC 2013-04-29  147.488007  134.000000
```

```
2013-04-30    2  Bitcoin    BTC 2013-04-30  146.929993  134.050003
2013-05-01    3  Bitcoin    BTC 2013-05-01  139.889999  107.720001
2013-05-02    4  Bitcoin    BTC 2013-05-02  125.599998   92.281898
2013-05-03    5  Bitcoin    BTC 2013-05-03  108.127998   79.099998

                  Open       Close  Volume     Marketcap  timeIndex
Date
2013-04-29  134.444000  144.539993     0.0  1.603769e+09          0
2013-04-30  144.000000  139.000000     0.0  1.542813e+09          1
2013-05-01  139.000000  116.989998     0.0  1.298955e+09          2
2013-05-02  116.379997  105.209999     0.0  1.168517e+09          3
2013-05-03  106.250000   97.750000     0.0  1.085995e+09          4
```

[5]:
```
df.tail()
```

[5]:
```
             SNo     Name Symbol       Date         High          Low  \
Date
2021-02-23  2858  Bitcoin    BTC 2021-02-23  54204.92976  45290.59027
2021-02-24  2859  Bitcoin    BTC 2021-02-24  51290.13669  47213.49816
2021-02-25  2860  Bitcoin    BTC 2021-02-25  51948.96698  47093.85302
2021-02-26  2861  Bitcoin    BTC 2021-02-26  48370.78526  44454.84211
2021-02-27  2862  Bitcoin    BTC 2021-02-27  48253.27010  45269.02577

                   Open        Close        Volume     Marketcap  timeIndex
Date
2021-02-23  54204.92976  48824.42687  1.061020e+11  9.099260e+11       2857
2021-02-24  48835.08766  49705.33332  6.369552e+10  9.263930e+11       2858
2021-02-25  49709.08242  47093.85302  5.450657e+10  8.777660e+11       2859
2021-02-26  47180.46405  46339.76008  3.509680e+11  8.637520e+11       2860
2021-02-27  46344.77224  46188.45128  4.591095e+10  8.609780e+11       2861
```

Se crean dummies de los meses, que serán utilizadas luego en un modelo que analiza estacionalidad

[6]:
```
df['Month'] = df['Date'].dt.month
df['Year'] = df['Date'].dt.year
dummies_mes = pd.get_dummies(df['Month'], drop_first = True, prefix = 'Month')
df = df.join(dummies_mes)
df.sample(10)
```

[6]:
```
             SNo     Name Symbol       Date         High          Low  \
Date
2020-05-30  2589  Bitcoin    BTC 2020-05-30   9704.030309   9366.729418
2020-04-24  2553  Bitcoin    BTC 2020-04-24   7574.196026   7434.181556
2018-07-11  1900  Bitcoin    BTC 2018-07-11   6444.959961   6330.470215
2018-08-04  1924  Bitcoin    BTC 2018-08-04   7497.490000   6984.070000
2016-10-23  1274  Bitcoin    BTC 2016-10-23    661.129028    653.885986
2013-12-28   244  Bitcoin    BTC 2013-12-28    747.059998    705.349976
```

```
2013-06-17    50  Bitcoin    BTC 2013-06-17    102.209999     99.000000
2020-10-26  2738  Bitcoin    BTC 2020-10-26  13225.297760  12822.382330
2015-09-08   863  Bitcoin    BTC 2015-09-08    245.781006    239.677994
2013-07-05    68  Bitcoin    BTC 2013-07-05     80.000000     65.526001

                    Open         Close        Volume     Marketcap  …  \
Date                                                                …
2020-05-30   9438.914009   9700.414072  3.272298e+10  1.783900e+11  …
2020-04-24   7434.181556   7550.901027  3.463653e+10  1.385120e+11  …
2018-07-11   6330.770020   6394.709961  3.644860e+09  1.096320e+11  …
2018-08-04   7438.670000   7032.850000  4.268390e+09  1.209000e+11  …
2016-10-23    657.620972    657.070984  5.447460e+07  1.047210e+10  …
2013-12-28    737.979981    727.830017  3.250580e+07  8.869919e+09  …
2013-06-17     99.900002    101.699997  0.000000e+00  1.149418e+09  …
2020-10-26  13031.201250  13075.247700  2.946146e+10  2.422510e+11  …
2015-09-08    239.845993    243.606995  2.687920e+07  3.554439e+09  …
2013-07-05     79.989998     68.431000  0.000000e+00  7.784112e+08  …

            Month_3  Month_4  Month_5  Month_6  Month_7  Month_8  Month_9  \
Date
2020-05-30        0        0        1        0        0        0        0
2020-04-24        0        1        0        0        0        0        0
2018-07-11        0        0        0        0        1        0        0
2018-08-04        0        0        0        0        0        1        0
2016-10-23        0        0        0        0        0        0        0
2013-12-28        0        0        0        0        0        0        0
2013-06-17        0        0        0        1        0        0        0
2020-10-26        0        0        0        0        0        0        0
2015-09-08        0        0        0        0        0        0        1
2013-07-05        0        0        0        0        1        0        0

            Month_10  Month_11  Month_12
Date
2020-05-30         0         0         0
2020-04-24         0         0         0
2018-07-11         0         0         0
2018-08-04         0         0         0
2016-10-23         1         0         0
2013-12-28         0         0         1
2013-06-17         0         0         0
2020-10-26         1         0         0
2015-09-08         0         0         0
2013-07-05         0         0         0

[10 rows x 24 columns]
```

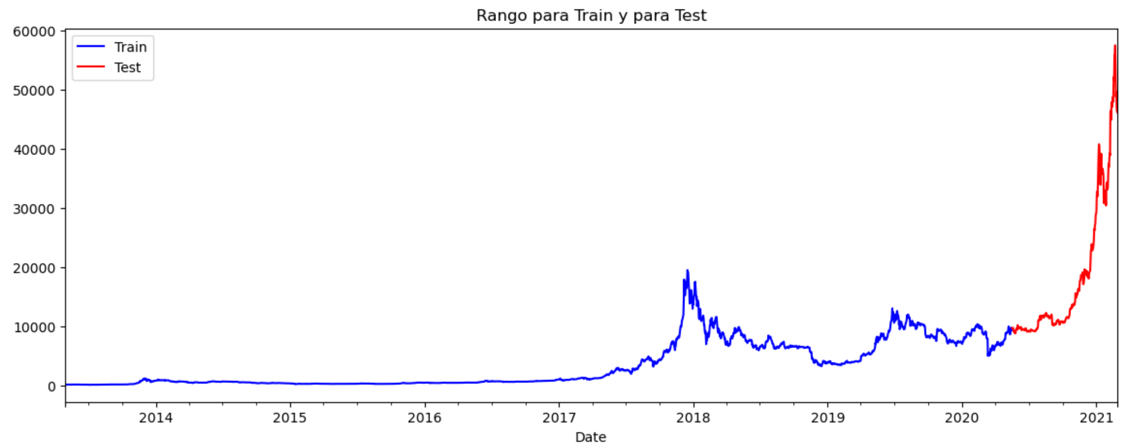
2.3 División de Train y Test

[7]:
```
# Se procede con el tradicional 90-10, recomendado para time series y/o␣
 ↪datasets muy grandes
df_train, df_test = train_test_split(df, test_size=0.1, shuffle=False)

print("train shape", df_train.shape)
print("test shape", df_test.shape)
```

```
train shape (2575, 24)
test shape (287, 24)
```

Ploteo de los dos datasets obtenidos:

[8]:
```
pd.plotting.register_matplotlib_converters()
f, ax = plt.subplots(figsize = (14,5))
df_train.plot(kind = 'line', x = 'Date', y = 'Close', color = 'blue', label =␣
 ↪"Train", ax = ax)
df_test.plot(kind = 'line', x = 'Date', y = 'Close', color = 'red', label =␣
 ↪"Test", ax = ax)
ax.legend(loc = 'upper left')
plt.title("Rango para Train y para Test")
plt.show()
```



Visualmente ya se puede ver que es prácticamente imposible que un modelo de predicción estime el ascenso que tuvo el BTC en el último medio año, por lo que el split 90-10 tradicional no va a permitir evaluar modelos acertadamente. Se procede a hacer un split manual para analizar solo el último mes disponible de 2021 (febrero) como test.

[9]:
```
train_size_date = df[df['Date'] <= (df['Date'].max() - pd.DateOffset(days=31))].
 ↪shape[0]
df_train, df_test = train_test_split(df, train_size=train_size_date,␣
 ↪shuffle=False)
```

```python
print("train shape", df_train.shape)
print("test shape", df_test.shape)
```

```
train shape (2831, 24)
test shape (31, 24)
```

[10]:
```python
df_test.head()
```

[10]:
```
             SNo     Name Symbol        Date         High          Low  \
Date
2021-01-28  2832  Bitcoin    BTC  2021-01-28  33858.31099  30023.20683
2021-01-29  2833  Bitcoin    BTC  2021-01-29  38406.26096  32064.81419
2021-01-30  2834  Bitcoin    BTC  2021-01-30  34834.70830  32940.18691
2021-01-31  2835  Bitcoin    BTC  2021-01-31  34288.33148  32270.17602
2021-02-01  2836  Bitcoin    BTC  2021-02-01  34638.21349  32384.22811

                   Open        Close        Volume     Marketcap  …  \
Date                                                              …
2021-01-28  30441.04182  33466.09636  7.651716e+10  6.229100e+11  …
2021-01-29  34318.67169  34316.38765  1.178950e+11  6.387690e+11  …
2021-01-30  34295.93504  34269.52154  6.514183e+10  6.379250e+11  …
2021-01-31  34270.87759  33114.35775  5.275454e+10  6.164530e+11  …
2021-02-01  33114.57724  33537.17682  6.140040e+10  6.243490e+11  …

            Month_3  Month_4  Month_5  Month_6  Month_7  Month_8  Month_9  \
Date
2021-01-28        0        0        0        0        0        0        0
2021-01-29        0        0        0        0        0        0        0
2021-01-30        0        0        0        0        0        0        0
2021-01-31        0        0        0        0        0        0        0
2021-02-01        0        0        0        0        0        0        0

            Month_10  Month_11  Month_12
Date
2021-01-28         0         0         0
2021-01-29         0         0         0
2021-01-30         0         0         0
2021-01-31         0         0         0
2021-02-01         0         0         0

[5 rows x 24 columns]
```

Ploteo de los dos datasets obtenidos:

[11]:
```python
pd.plotting.register_matplotlib_converters()
f, ax = plt.subplots(figsize = (14,5))
df_train.plot(kind = 'line', x = 'Date', y = 'Close', color = 'blue', label =␣
↪"Train", ax = ax)
```

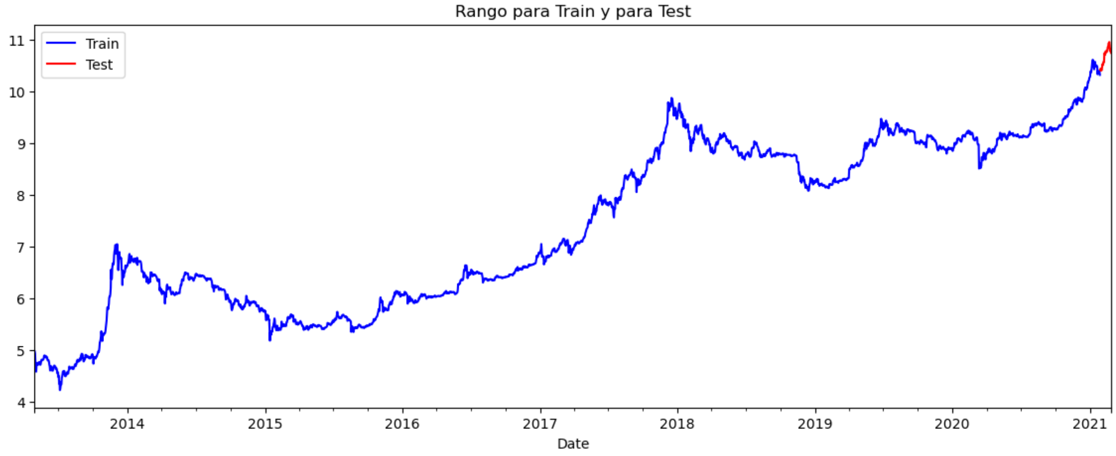
```
df_test.plot(kind = 'line', x = 'Date', y = 'Close', color = 'red', label =␣
 ↪"Test", ax = ax)
ax.legend(loc = 'upper left')
plt.title("Rango para Train y para Test")
plt.show()
```

2.4 Generación de la serie en escala logarítmica

[12]:
```
df_train['log_value'] = np.log(df_train['Close'])
df_test['log_value'] = np.log(df_test['Close'])
```

Ploteo del Target y Test:

[13]:
```
pd.plotting.register_matplotlib_converters()
f, ax = plt.subplots(figsize = (14,5))
df_train.plot(kind = 'line', x = 'Date', y = 'log_value', color = 'blue', label␣
 ↪= "Train", ax = ax)
df_test.plot(kind = 'line', x = 'Date', y = 'log_value', color = 'red', label =␣
 ↪"Test", ax = ax)
ax.legend(loc = 'upper left')
plt.title("Rango para Train y para Test")
plt.show()
```



3 2) Modelos

Se utilizará una plétora de herramientas y recursos para analizar las series de tiempo y sus implicancias. En cada paso se irá visualizando los resultados y almacenando su información para, al final de la notebook, compararlos

Se define una función para calcular el RMSE:

[14]:
```
def RMSE(actual, predicted):
  mse = (predicted - actual) ** 2
  rmse = np.sqrt(mse.sum() / mse.count())
  return rmse
```

Se define una función para calcular el MAPE:

[15]:
```
def MAPE(actual, predicted):
  actual, predicted = np.array(actual), np.array(predicted)
  return np.mean(np.abs((actual - predicted) / actual)) * 100
```

Se define una función para crear los gráficos de cada modelo:

[16]:
```
def plot_time_series(df_train, df_test, model_name, series = 'Close'):
  fig, axes = plt.subplots(1,2, figsize = (16, 6))

  df_train.plot(kind = "line", y = [series, model_name], ax = axes[0])
  axes[0].set_title("Train Data", size = 16)
  axes[0].set_xlabel("Year", size = 14)

  df_test.plot(kind = "line", y = [series, model_name], ax = axes[1])
  axes[1].set_title("Test Data", size = 16)
  axes[1].set_xlabel("Date", size = 14)
  # Por el salto de mes que se da de enero a febrero, el código plotea los x␣
↪ticks con variedad de formatos. Se lo modifica:
```

```
date_format = mdates.DateFormatter('%b-%d')
axes[1].xaxis.set_major_formatter(date_format)
weekday_locator = mdates.WeekdayLocator(byweekday = mdates.MO)
axes[1].xaxis.set_major_locator(weekday_locator)

# Se agregan el RMSE y el MAPE debajo de los plots
rmse = RMSE(df_test[series], df_test[model_name])
mape = MAPE(df_test[series], df_test[model_name])
rmse_mape_text = f"RMSE = {rmse:.2f} | MAPE = {mape:.2f}%"
plt.text(0.5, 0.0, rmse_mape_text, ha = 'center', transform = fig.
↪transFigure, fontsize = 14)

#Se agrega el título del plot
formatted_model_name = model_name.replace("_", " ").title()
plt.suptitle(f"Predicción del precio de BTC con {formatted_model_name}", size␣
↪= 18, y = 1.02)

plt.show()
```

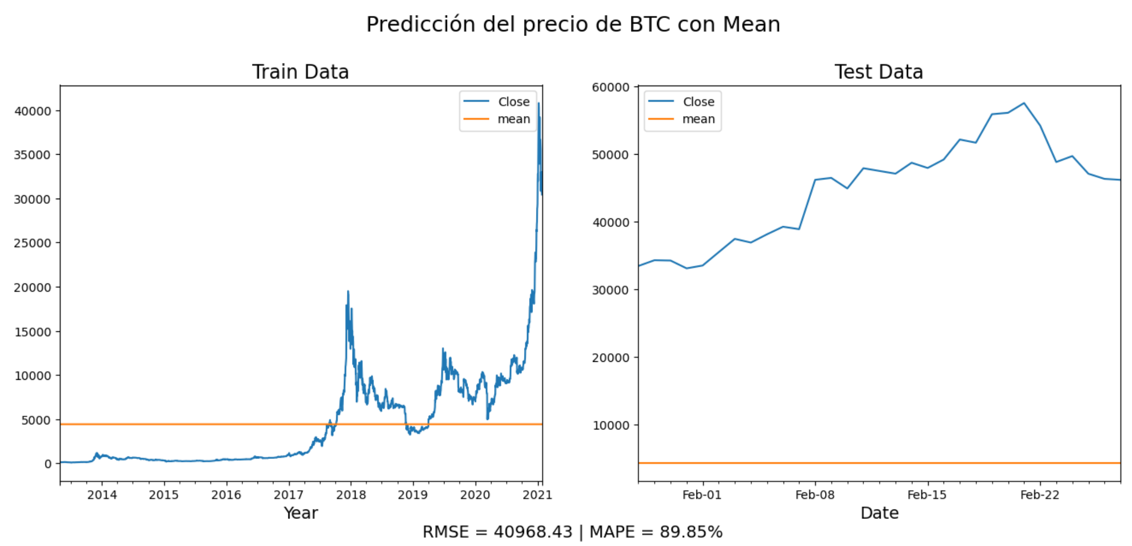
3.1 a) Mean

Se aplica el modelo de media constante a train y test:

[17]:
```
# Se calcula el promedio:
model_mean_pred = df_train['Close'].mean()

# La predicción es fija y es la misma para el set de testeo y de entrenamiento:
df_train['mean'] = model_mean_pred
df_test['mean'] = model_mean_pred
```

Ploteo de las predicciones vs la serie real y cálculo de RMSE y MAPE:

[18]:
```
plot_time_series(df_train, df_test, 'mean')
```



Se guardan los resultados en un DataFrame: El mismo será reutilizado para almacenar los resultados de los distintos modelos a utilizar

[19]:
```
df_results = pd.DataFrame(columns = ["Model", "RMSE","MAPE"])
df_results.loc[0, "Model"] = "Mean"
df_results.loc[0, "RMSE"] = round(RMSE(df_test['Close'], df_test['mean']),1)
df_results.loc[0, "MAPE"] = round(MAPE(df_test['Close'], df_test['mean']),1)
df_results
```

[19]:

	Model	RMSE	MAPE
0	Mean	40968.4	89.8

3.2 b) Random Walk

Se crea el shift de target en train:

[20]:
```
df_train['close_shift'] = df_train['Close'].shift()
# La primera observación va a quedar en nan, por lo que se reemplaza por el␣
↪valor siguente:
df_train['close_shift'].fillna(method = 'bfill', inplace = True)
df_train.head()
```

[20]:

Date	SNo	Name	Symbol	Date	High	Low
2013-04-29	1	Bitcoin	BTC	2013-04-29	147.488007	134.000000
2013-04-30	2	Bitcoin	BTC	2013-04-30	146.929993	134.050003
2013-05-01	3	Bitcoin	BTC	2013-05-01	139.889999	107.720001
2013-05-02	4	Bitcoin	BTC	2013-05-02	125.599998	92.281898
2013-05-03	5	Bitcoin	BTC	2013-05-03	108.127998	79.099998

Date	Open	Close	Volume	Marketcap	…	Month_6
2013-04-29	134.444000	144.539993	0.0	1.603769e+09	…	0
2013-04-30	144.000000	139.000000	0.0	1.542813e+09	…	0
2013-05-01	139.000000	116.989998	0.0	1.298955e+09	…	0
2013-05-02	116.379997	105.209999	0.0	1.168517e+09	…	0
2013-05-03	106.250000	97.750000	0.0	1.085995e+09	…	0

Date	Month_7	Month_8	Month_9	Month_10	Month_11	Month_12
2013-04-29	0	0	0	0	0	0
2013-04-30	0	0	0	0	0	0
2013-05-01	0	0	0	0	0	0
2013-05-02	0	0	0	0	0	0
2013-05-03	0	0	0	0	0	0

```
             log_value         mean  close_shift
Date
2013-04-29    4.973556  4415.425613   144.539993
2013-04-30    4.934474  4415.425613   144.539993
2013-05-01    4.762088  4415.425613   139.000000
2013-05-02    4.655958  4415.425613   116.989998
2013-05-03    4.582413  4415.425613   105.209999

[5 rows x 27 columns]
```

Se crea el shift de target en test:

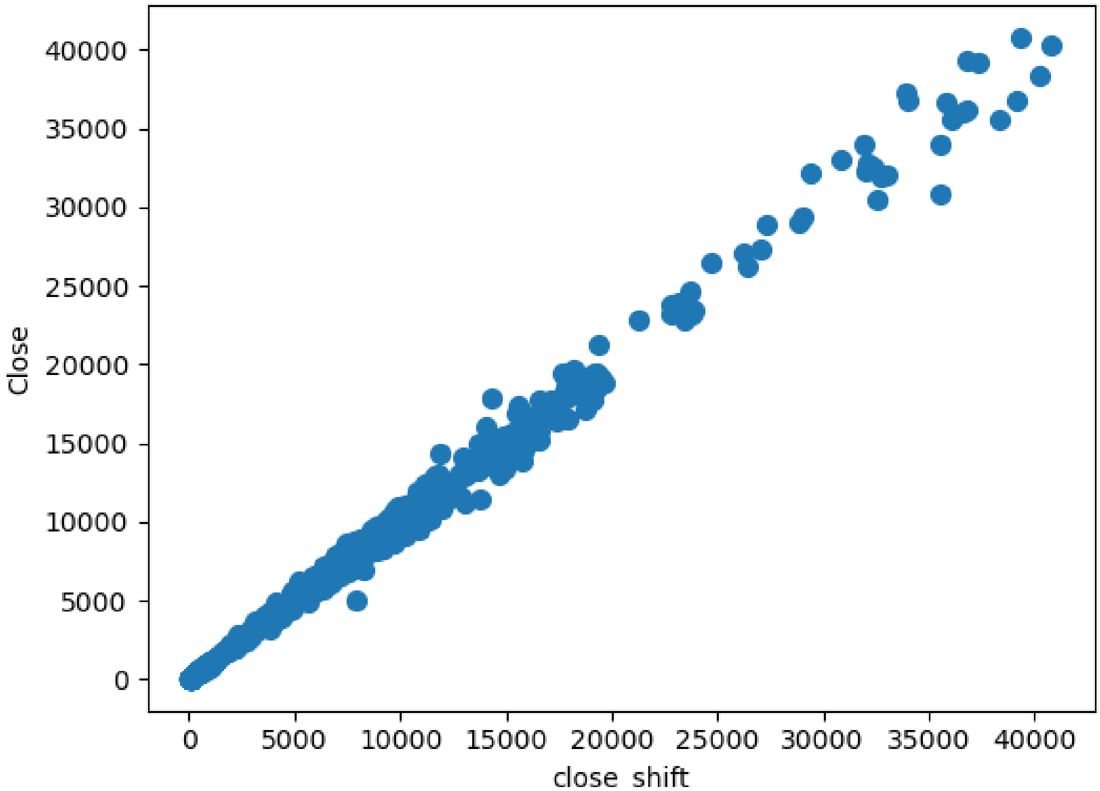
[21]:
```python
df_test['close_shift'] = df_test['Close'].shift()
# Se puede reemplazar el primer nan con el último valor del set de
 ↪entrenamiento:
df_test.iloc[0,26] = df_train.iloc[-1,0]
df_test.head()
```

[21]:
```
             SNo     Name Symbol        Date          High           Low  \
Date
2021-01-28  2832  Bitcoin    BTC  2021-01-28   33858.31099   30023.20683
2021-01-29  2833  Bitcoin    BTC  2021-01-29   38406.26096   32064.81419
2021-01-30  2834  Bitcoin    BTC  2021-01-30   34834.70830   32940.18691
2021-01-31  2835  Bitcoin    BTC  2021-01-31   34288.33148   32270.17602
2021-02-01  2836  Bitcoin    BTC  2021-02-01   34638.21349   32384.22811

                   Open         Close        Volume      Marketcap  …  \
Date                                                                …
2021-01-28  30441.04182   33466.09636  7.651716e+10   6.229100e+11  …
2021-01-29  34318.67169   34316.38765  1.178950e+11   6.387690e+11  …
2021-01-30  34295.93504   34269.52154  6.514183e+10   6.379250e+11  …
2021-01-31  34270.87759   33114.35775  5.275454e+10   6.164530e+11  …
2021-02-01  33114.57724   33537.17682  6.140040e+10   6.243490e+11  …

             Month_6  Month_7  Month_8  Month_9  Month_10  Month_11  Month_12  \
Date
2021-01-28         0        0        0        0         0         0         0
2021-01-29         0        0        0        0         0         0         0
2021-01-30         0        0        0        0         0         0         0
2021-01-31         0        0        0        0         0         0         0
2021-02-01         0        0        0        0         0         0         0

             log_value         mean  close_shift
Date
2021-01-28   10.418288  4415.425613   2831.00000
2021-01-29   10.443378  4415.425613  33466.09636
2021-01-30   10.442012  4415.425613  34316.38765
```

```
2021-01-31  10.407722  4415.425613  34269.52154
2021-02-01  10.420410  4415.425613  33114.35775

[5 rows x 27 columns]
```

Lag de un período:

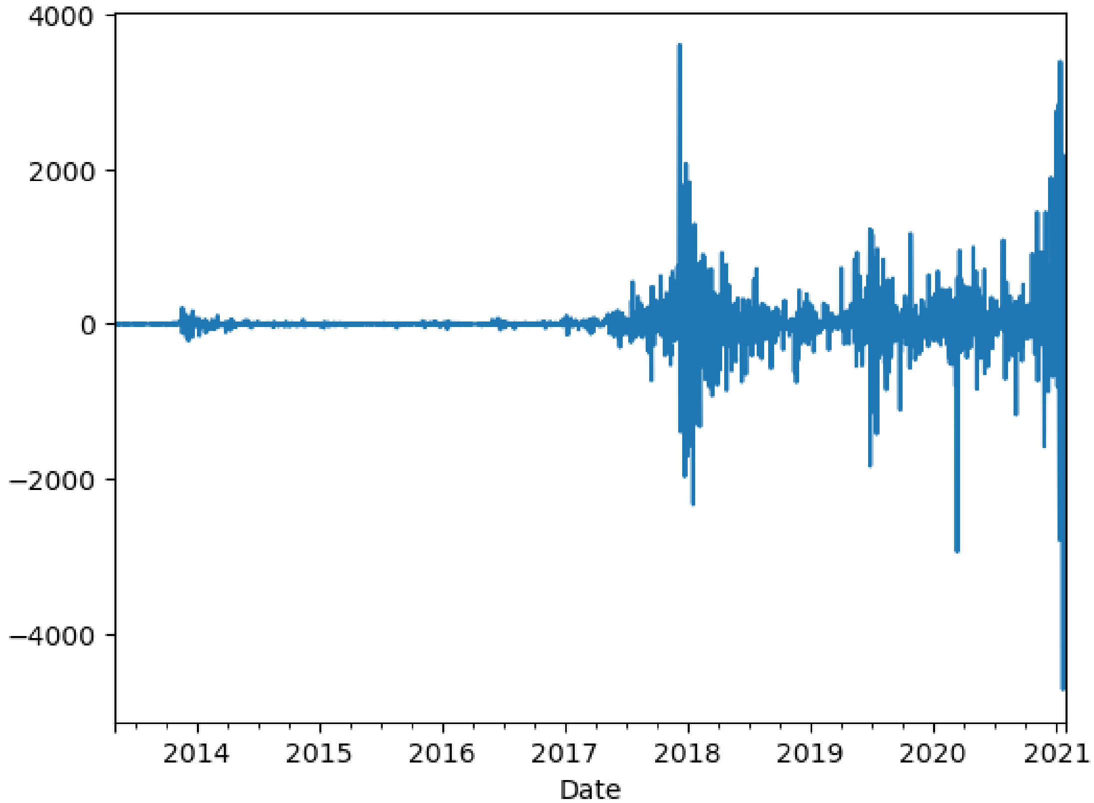
[22]:
```
df_train.plot(kind = 'scatter', y = 'Close', x = 'close_shift', s = 50);
```



Diferencias entre Target y el lag:

[23]:
```
df_train['close_diff'] = df_train['Close'] - df_train['close_shift']
df_train['close_diff'].plot()
```

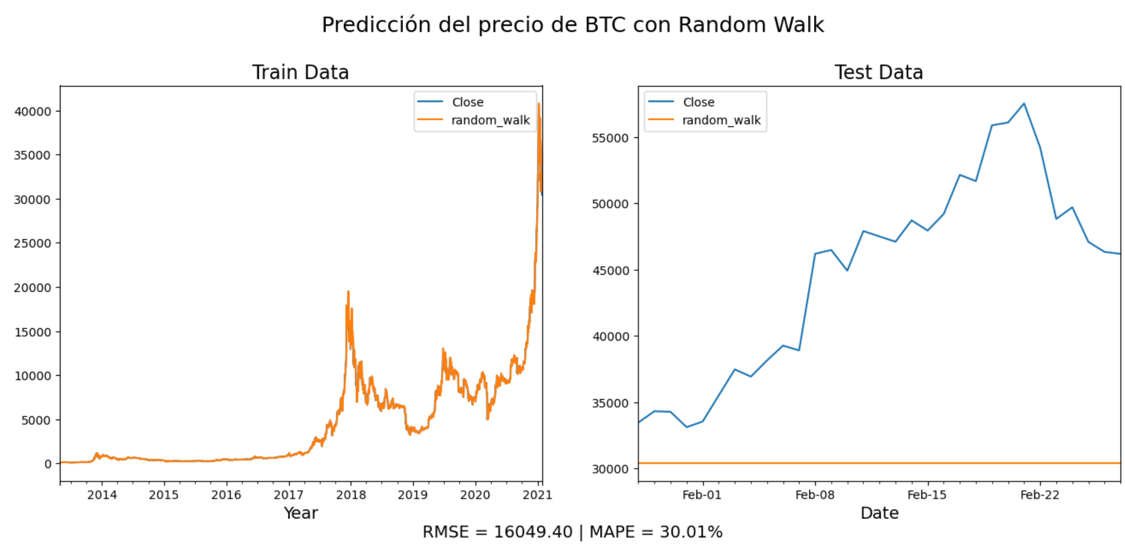
[23]: <Axes: xlabel='Date'>



Ploteo de las predicciones vs la serie real y cálculo de RMSE y MAPE:

[24]:
```
df_train['random_walk'] = df_train['close_shift']
df_test['random_walk'] = pd.Series(df_train['Close'][-1], index = df_test.index)
```

[25]:
```
plot_time_series(df_train, df_test, 'random_walk')
```



Se almacenan los valores de RMSE y MAPE

[26]:
```
df_results.loc[1, "Model"] = "Random Walk"
df_results.loc[1, "RMSE"] = round(RMSE(df_test['Close'],␣
↪df_test['random_walk']),1)
df_results.loc[1, "MAPE"] = round(MAPE(df_test['Close'],␣
↪df_test['random_walk']),1)
df_results
```

[26]:

	Model	RMSE	MAPE
0	Mean	40968.4	89.8
1	Random Walk	16049.4	30.0

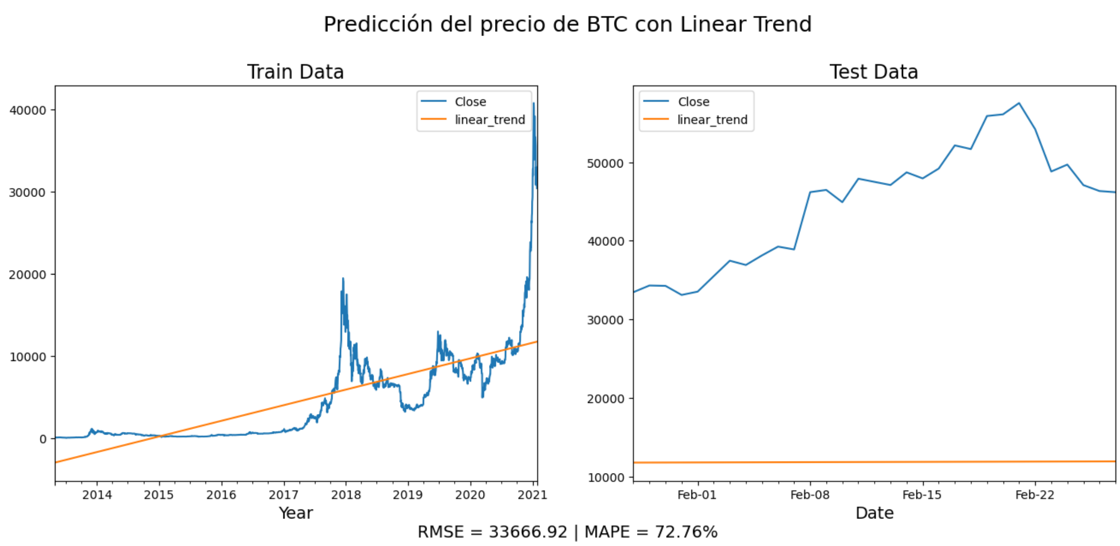
3.3 c) Linear Trend

Se crea una columna en train con el predict:

[27]:
```
model_linear = smf.ols('Close ~        timeIndex', data = df_train).fit()

df_train['linear_trend'] = model_linear.predict(df_train['timeIndex'])
df_test['linear_trend'] = model_linear.predict(df_test['timeIndex'])
```

Ploteo de las predicciones vs las series reales, en train y test:

[28]:
```
plot_time_series(df_train, df_test, 'linear_trend')
```



Se almacenan los valores de RMSE y MAPE

[29]:
```
df_results.loc[2, "Model"] = "Linear Trend"
df_results.loc[2, "RMSE"] = round(RMSE(df_test['Close'],␣
 ↪df_test['linear_trend']),1)
df_results.loc[2, "MAPE"] = round(MAPE(df_test['Close'],␣
 ↪df_test['linear_trend']),1)
df_results
```

[29]:

	Model	RMSE	MAPE
0	Mean	40968.4	89.8
1	Random Walk	16049.4	30.0
2	Linear Trend	33666.9	72.8

3.4 d) Back Log Transformation + Linear Trend

Se fitea el modelo Linear Trend con escala logarítmica

[30]:
```
model_log = smf.ols('log_value ~ timeIndex', data = df_train).fit()
```

[31]:
```
model_log.summary()
```

[31]:
```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:              log_value   R-squared:                       0.835
Model:                            OLS   Adj. R-squared:                  0.835
Method:                 Least Squares   F-statistic:                 1.434e+04
Date:                Wed, 23 Aug 2023   Prob (F-statistic):               0.00
Time:                        16:27:56   Log-Likelihood:                -2770.3
No. Observations:                2831   AIC:                             5545.
Df Residuals:                    2829   BIC:                             5556.
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
Intercept      4.8828      0.024    201.754      0.000       4.835       4.930
timeIndex      0.0018   1.48e-05    119.736      0.000       0.002       0.002
==============================================================================
Omnibus:                      222.150   Durbin-Watson:                   0.004
Prob(Omnibus):                  0.000   Jarque-Bera (JB):              270.407
Skew:                           0.746   Prob(JB):                     1.91e-59
Kurtosis:                       2.747   Cond. No.                     3.27e+03
==============================================================================

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly
specified.
```

```
[2] The condition number is large, 3.27e+03. This might indicate that there are
strong multicollinearity or other numerical problems.
"""
```

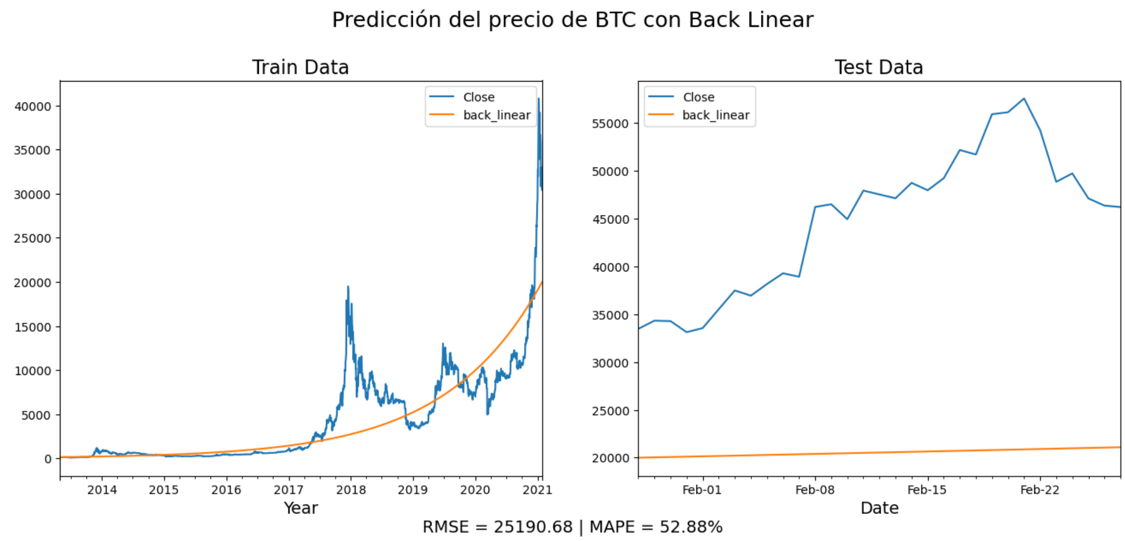
[32]:
```python
df_train['log_linear'] = model_log.predict(df_train[['timeIndex']])
df_test['log_linear'] = model_log.predict(df_test[['timeIndex']])
```

Se invierte la escala logarítmica del modelo anterior

[33]:
```python
df_train['back_linear'] = np.exp(df_train['log_linear'])
df_test['back_linear'] = np.exp(df_test['log_linear'])
```

Ploteo de las predicciones vs las series reales, en train y test:

[34]:
```python
plot_time_series(df_train, df_test, 'back_linear')
```



Se almacenan los valores de RMSE y MAPE

[35]:
```python
df_results.loc[3, "Model"] = "Back Log Linear"
df_results.loc[3, "RMSE"] = round(RMSE(df_test['Close'],␣
↪df_test['back_linear']),1)
df_results.loc[3, "MAPE"] = round(MAPE(df_test['Close'],␣
↪df_test['back_linear']),1)
df_results
```

[35]:

	Model	RMSE	MAPE
0	Mean	40968.4	89.8
1	Random Walk	16049.4	30.0
2	Linear Trend	33666.9	72.8
3	Back Log Linear	25190.7	52.9

3.5 e) Back Log Transformation + Linear Trend + Estacionalidad

En la tercera sección de esta notebook se utilizarán algunas herramientas para analizar la estacionalidad de la serie. Sin embargo, igualmente se analiza un caso de modelo con agregado de estacionalidad para ver si aporta al resultado.

Creación del modelo con dummies

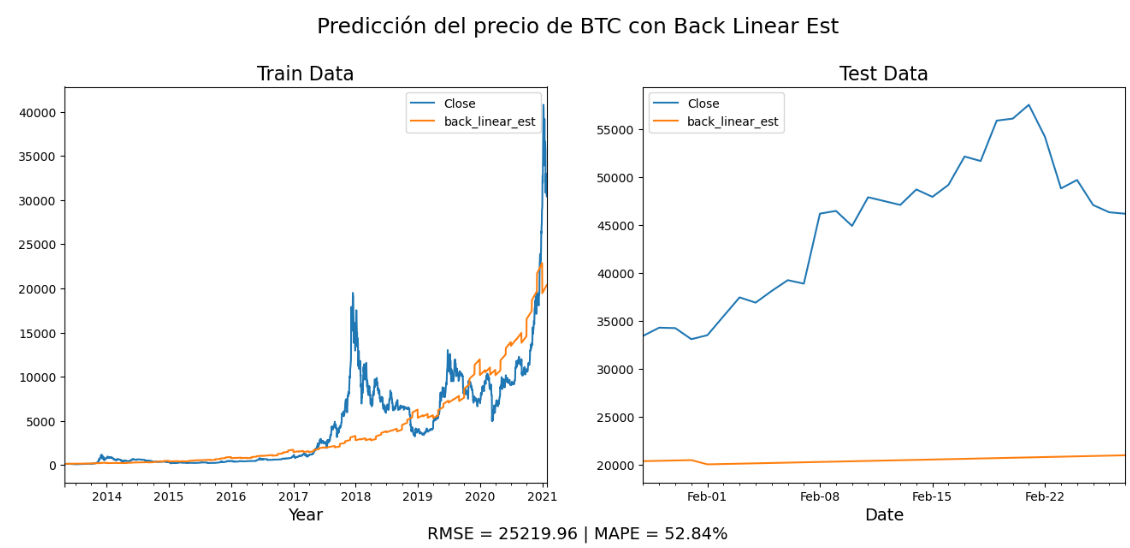
[36]:
```
log_linear_est = smf.ols('log_value ~ timeIndex + Month_2 + Month_3 + Month_4 + Month_5 + Month_6 + Month_7 + Month_8 + Month_9 + Month_11 + Month_12', data= df_train).fit()

df_train['log_linear_est'] = log_linear_est.predict(df_train[['timeIndex', 'Month_2', 'Month_3', 'Month_4', 'Month_5', 'Month_6','Month_7', 'Month_8', 'Month_9','Month_10','Month_11','Month_12']])

df_test['log_linear_est'] = log_linear_est.predict(df_test[['timeIndex', 'Month_2', 'Month_3', 'Month_4', 'Month_5', 'Month_6','Month_7', 'Month_8', 'Month_9','Month_10','Month_11','Month_12']])
```

Se invierte la escala logarítmica del modelo anterior

[37]:
```
df_train['back_linear_est'] = np.exp(df_train['log_linear_est'])
df_test['back_linear_est'] = np.exp(df_test['log_linear_est'])
```

Ploteo de las predicciones vs las series reales, en train y test:

[38]:
```
plot_time_series(df_train, df_test, 'back_linear_est')
```



Se comparan los valores de RMSE y MAPE del Back Transformation sin y con el agregado de los meses A pesar de que Back Transformation con y sin Estimate parecen idénticos, hay unas mínimas diferencias en los decimales.

```
[39]: print("El RMSE de Back Transformation es ", RMSE(df_test['Close'],␣
  ↪df_test['back_linear']), ", mientras que el de Back Transformation +␣
  ↪Estacionalidad es ", RMSE(df_test['Close'], df_test['back_linear_est']), ".
  ↪\n La diferencia es de", (RMSE(df_test['Close'], df_test['back_linear']) -␣
  ↪RMSE(df_test['Close'], df_test['back_linear_est'])))
print("\nEl MAPE de Back Transformation es ", MAPE(df_test['Close'],␣
  ↪df_test['back_linear']), ", mientras que el de Back Transformation +␣
  ↪Estacionalidad es ", MAPE(df_test['Close'], df_test['back_linear_est']), ".
  ↪\n La diferencia es de", (MAPE(df_test['Close'], df_test['back_linear']) -␣
  ↪MAPE(df_test['Close'], df_test['back_linear_est'])))
```

```
El RMSE de Back Transformation es  25190.68314519895 , mientras que el de Back
Transformation + Estacionalidad es  25219.96294723701 .
 La diferencia es de -29.27980203806146

El MAPE de Back Transformation es  52.87858925532839 , mientras que el de Back
Transformation + Estacionalidad es  52.84438410973474 .
 La diferencia es de 0.03420514559365273
```

Por ser ínfima la diferencia, no se almacena el valor del modelo con Estimate incluido.

3.6 f) Simple Smoothing

Se aplica Cross Validation para averiguar el nivel óptimo de Simple Smoothing del train data.

```
[40]: # Se estandarizan los datos
scaler = StandardScaler()
values_standardized = scaler.fit_transform(df_train['Close'].values.reshape(-1,␣
  ↪1)).flatten()

# Se define el rango de hiperparametros a teastear
hyperparam_range = np.linspace(0.001, 1, num=100)

# Se calcula el error de cada hiperparámetro utilizando CV
tscv = TimeSeriesSplit(n_splits=5)
mse_errors = []
for alpha in hyperparam_range:
    errors = []
    for train, test in tscv.split(values_standardized):
        model = SimpleExpSmoothing(values_standardized[train]).
  ↪fit(smoothing_level=alpha, optimized=False)
        predictions_standardized = model.forecast(len(test))
        actual_standardized = values_standardized[test]
        predictions = scaler.inverse_transform(predictions_standardized.
  ↪reshape(-1, 1)).flatten()
        actual = scaler.inverse_transform(actual_standardized.reshape(-1, 1)).
  ↪flatten()
        error = mean_squared_error(predictions, actual)
```

```
        errors.append(error)
    mse_errors.append(np.mean(np.array(errors)))

# Se encuentra el hiperparámetro óptimo
optimal_alpha = hyperparam_range[np.argmin(mse_errors)]
print('Optimal alpha:', optimal_alpha)
```

```
Optimal alpha: 0.03127272727272727
```

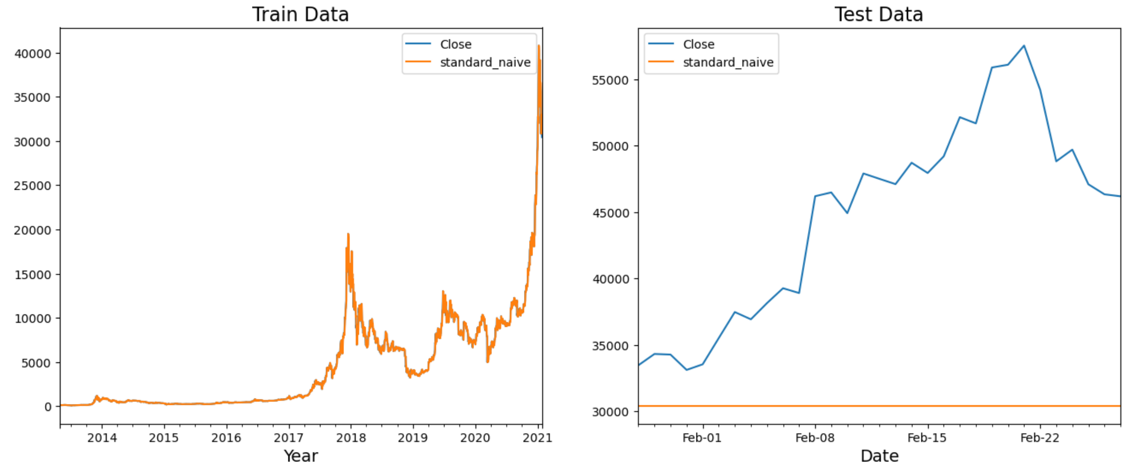
Se fitean varios modelos Se realizará el proceso 3 veces para comparar los resultados en test. El primer caso será uno sin suavizado: en ese caso, Simple Smoothing equivale a un modelo Standard Naive (por lo que se espera que el resultado sea el mismo que el que se obtuvo aplicando Random Walk). Los otros dos serán con distintos grados de suavizado, siendo uno el obtenido mediante Cross Validation.

[41]:
```
model_no_smoothing = SimpleExpSmoothing(df_train['Close']).fit(smoothing_level␣
 ↪= 1, optimized = False)
df_train['standard_naive'] = model_no_smoothing.fittedvalues
df_test['standard_naive'] = model_no_smoothing.forecast(len(df_test))

model_simple_smoothing = SimpleExpSmoothing(df_train['Close']).
 ↪fit(smoothing_level = 0.1, optimized = False)
df_train['simple_smoothing'] = model_simple_smoothing.fittedvalues
df_test['simple_smoothing'] = model_simple_smoothing.forecast(len(df_test))

model_strong_simple_smoothing = SimpleExpSmoothing(df_train['Close']).
 ↪fit(smoothing_level = optimal_alpha, optimized = False)
df_train['strong_simple_smoothing'] = model_strong_simple_smoothing.fittedvalues
df_test['strong_simple_smoothing'] = model_strong_simple_smoothing.
 ↪forecast(len(df_test))
```

Ploteo de las predicciones vs las series reales, en train y test:

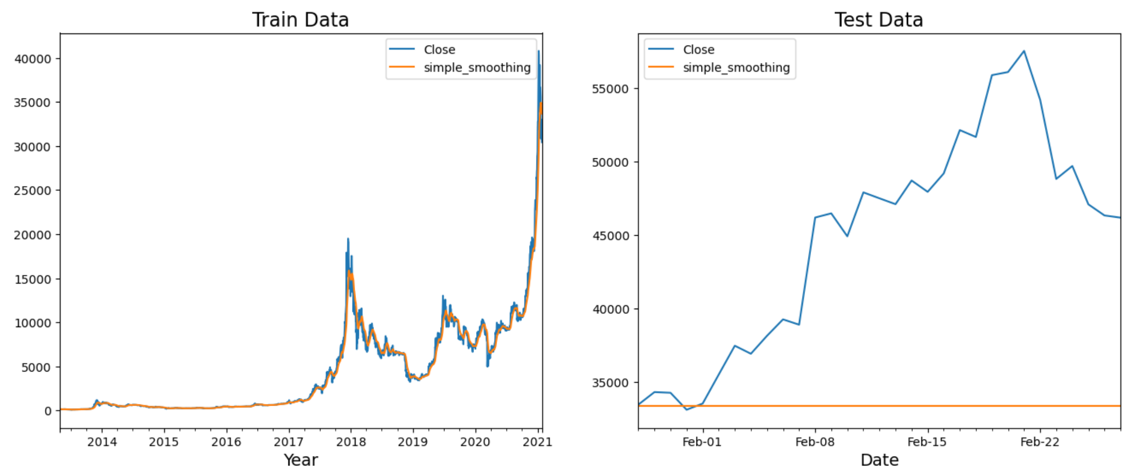
[42]:
```
plot_time_series(df_train, df_test, 'standard_naive')
plot_time_series(df_train, df_test, 'simple_smoothing')
plot_time_series(df_train, df_test, 'strong_simple_smoothing')
```

Predicción del precio de BTC con Standard Naive

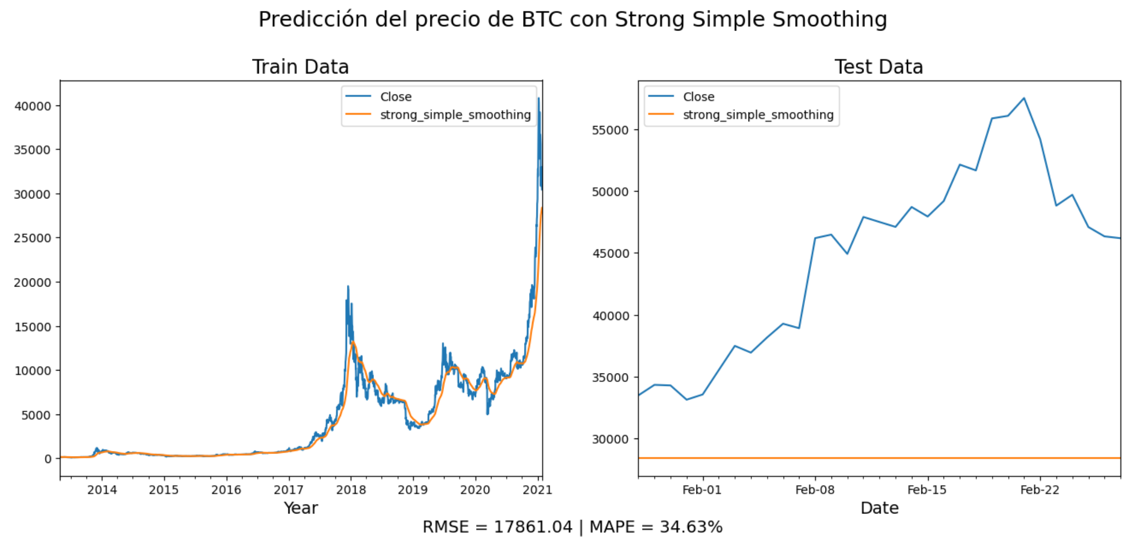


RMSE = 16049.40 | MAPE = 30.01%

Predicción del precio de BTC con Simple Smoothing



RMSE = 13517.46 | MAPE = 23.35%



Como se puede observar, el resultado obtenido utilizando el alpha que arroja el cross validation sobre train presenta underfitting, ya que da un rendimiento inferior en comparación con un alpha superior. En otras palabras, el mayor suavizado es menos eficiente que no suavizar (Standard Naive), y mucho menos que un suavizado leve. Esto es razonable en casos como el del presente dataset: los patrones históricos de 2014 a 2017 tienen poca relevancia para predecir movimientos actuales en el precio, en comparación a tendencias recientes como lo son las del 2020 con el boom de las criptomonedas. Un alpha más alto, con menor suavizado y mayor overfitting, podría capturar mejor las fluctuaciones a corto plazo que son más significativas en mercados criptográficos en constante evolución.

Se almacenan los valores de RMSE y MAPE

[43]:
```
df_results.loc[4, "Model"] = "Simple Smoothing"
df_results.loc[4, "RMSE"] = round(RMSE(df_test['Close'],␣
↪df_test['simple_smoothing']),1)
df_results.loc[4, "MAPE"] = round(MAPE(df_test['Close'],␣
↪df_test['simple_smoothing']),1)
df_results
```

[43]:

	Model	RMSE	MAPE
0	Mean	40968.4	89.8
1	Random Walk	16049.4	30.0
2	Linear Trend	33666.9	72.8
3	Back Log Linear	25190.7	52.9
4	Simple Smoothing	13517.5	23.4

3.7 g) ARIMA

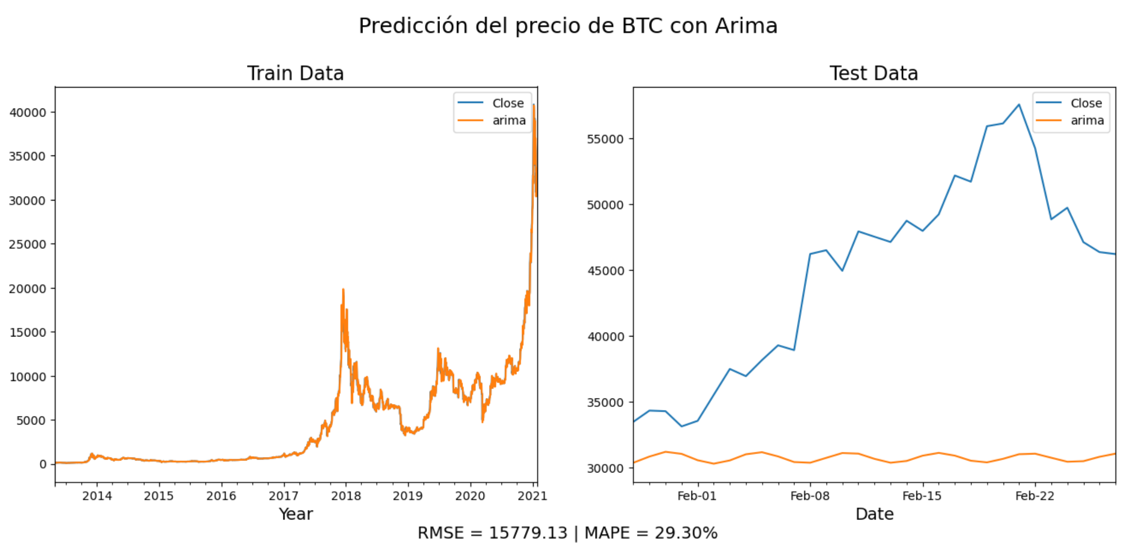
[44]:
```
stepwise_fit = auto_arima(df_train['Close'], trace = True, suppress_warnings =
    True)
model_ARIMA = ARIMA(df_train['Close'], order = stepwise_fit.order)
```

```
Performing stepwise search to minimize aic
 ARIMA(2,1,2)(0,0,0)[0] intercept   : AIC=40848.812, Time=1.91 sec
 ARIMA(0,1,0)(0,0,0)[0] intercept   : AIC=40883.765, Time=0.11 sec
 ARIMA(1,1,0)(0,0,0)[0] intercept   : AIC=40871.221, Time=0.24 sec
 ARIMA(0,1,1)(0,0,0)[0] intercept   : AIC=40870.558, Time=0.27 sec
 ARIMA(0,1,0)(0,0,0)[0]             : AIC=40884.713, Time=0.06 sec
 ARIMA(1,1,2)(0,0,0)[0] intercept   : AIC=40873.105, Time=0.85 sec
 ARIMA(2,1,1)(0,0,0)[0] intercept   : AIC=40872.143, Time=1.01 sec
 ARIMA(3,1,2)(0,0,0)[0] intercept   : AIC=40842.726, Time=2.64 sec
 ARIMA(3,1,1)(0,0,0)[0] intercept   : AIC=40870.089, Time=1.76 sec
 ARIMA(4,1,2)(0,0,0)[0] intercept   : AIC=40843.842, Time=3.23 sec
 ARIMA(3,1,3)(0,0,0)[0] intercept   : AIC=40844.026, Time=4.69 sec
 ARIMA(2,1,3)(0,0,0)[0] intercept   : AIC=40843.222, Time=3.43 sec
 ARIMA(4,1,1)(0,0,0)[0] intercept   : AIC=40847.809, Time=3.20 sec
 ARIMA(4,1,3)(0,0,0)[0] intercept   : AIC=40846.343, Time=3.44 sec
 ARIMA(3,1,2)(0,0,0)[0]             : AIC=40843.528, Time=1.53 sec

Best model:  ARIMA(3,1,2)(0,0,0)[0] intercept
Total fit time: 28.406 seconds
```

[45]:
```
df_train['arima'] = model_ARIMA.fit().fittedvalues
```

[46]:
```
forecast_ARIMA = model_ARIMA.fit().get_forecast(steps=len(df_test))
df_test['arima'] = forecast_ARIMA.predicted_mean.values
```

[47]:
```
plot_time_series(df_train, df_test, 'arima')
```



Se observa el summary:

[48]:
```
print(model_ARIMA.fit().summary())
```

```
                               SARIMAX Results
==============================================================================
Dep. Variable:                  Close   No. Observations:                 2831
Model:                 ARIMA(3, 1, 2)   Log Likelihood              -20415.764
Date:                Wed, 23 Aug 2023   AIC                          40843.528
Time:                        16:28:35   BIC                          40879.216
Sample:                    04-29-2013   HQIC                         40856.403
                         - 01-27-2021
Covariance Type:                  opg
==============================================================================
                 coef    std err          z      P>|z|      [0.025      0.975]
------------------------------------------------------------------------------
ar.L1          0.9247      0.015     62.199      0.000       0.896       0.954
ar.L2         -1.0336      0.012    -86.818      0.000      -1.057      -1.010
ar.L3          0.0685      0.006     10.901      0.000       0.056       0.081
ma.L1         -0.8548      0.015    -57.243      0.000      -0.884      -0.826
ma.L2          0.9395      0.013     72.490      0.000       0.914       0.965
sigma2      1.092e+05    764.096    142.970      0.000    1.08e+05    1.11e+05
================================================================================
===
Ljung-Box (L1) (Q):                   0.01   Jarque-Bera (JB):
148229.19
Prob(Q):                              0.91   Prob(JB):
0.00
Heteroskedasticity (H):             338.19   Skew:
0.13
Prob(H) (two-sided):                  0.00   Kurtosis:
38.45
================================================================================
===

Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-
step).
```

Se almacenan los valores de RMSE y MAPE

[49]:
```
df_results.loc[5, "Model"] = "ARIMA"
df_results.loc[5, "RMSE"] = round(RMSE(df_test['Close'], df_test['arima']),1)
df_results.loc[5, "MAPE"] = round(MAPE(df_test['Close'], df_test['arima']),1)
df_results
```

[49]:

	Model	RMSE	MAPE
0	Mean	40968.4	89.8
1	Random Walk	16049.4	30.0
2	Linear Trend	33666.9	72.8
3	Back Log Linear	25190.7	52.9
4	Simple Smoothing	13517.5	23.4
5	ARIMA	15779.1	29.3

3.8 h) Prophet

Buscamos la mejor combinación de hiperparámetros: Prophet requiere que la columna Date se llame “ds” y la columna de los precios “y”.

[50]:
```
df_train['ds'] = df_train['Date']
df_test['ds'] = df_test['Date']
df_train['y'] = df_train['Close']
df_test['y'] = df_test['Close']
```

[51]:
```
# Este código toma mucho tiempo. Dependiendo del poder de procesamiento de la␣
↪computador, puede demorar alrededor de una hora.

# Como el proceso produce más de 100 líneas de output por cada iteración de␣
↪prueba de hiperparámetros, se lo reduce via logging:
logger = logging.getLogger('cmdstanpy')
logger.addHandler(logging.NullHandler())
logger.propagate = False
logger.setLevel(logging.CRITICAL)
# De esta manera, solo se produce la barra de progreso de cada iteración.

# Se crea una grilla con distintos valores de parámetros posibles a testear
param_grid = {
    'changepoint_prior_scale': [0.001, 0.005, 0.01, 0.1, 0.5, 1],
    'seasonality_prior_scale': [1, 10, 20, 30, 40],
    'seasonality_mode' : ('additive', 'multiplicative'),
    'daily_seasonality' : [False, True]}

# Genera todas las combinaciones de parámetros
all_params = [dict(zip(param_grid.keys(), v)) for v in itertools.
↪product(*param_grid.values())]
rmses = []

# Usa cross validation para evaluar los parámetros
for params in all_params:
    m = Prophet(**params).fit(df_train)  # Fitea el modelo con los parámetros␣
↪obtenidos
    df_cv = cross_validation(m, initial='2500 days', period= '15 days',␣
↪horizon='31 days')
```

```python
    df_p = performance_metrics(df_cv, rolling_window=1)
    rmses.append(df_p['rmse'].values[0])
```

```
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
```

```
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
```

```
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
0%|          | 0/20 [00:00<?, ?it/s]
```

```
0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]

0%|          | 0/20 [00:00<?, ?it/s]
```

Análisis de cuál tuvo mejor rendimiento:

[52]:
```python
# Se convierten los resultados en un df
tuning_results = pd.DataFrame(all_params)
tuning_results['rmse'] = rmses

# Se encuentran los mejores hiperparámetros
best_params = tuning_results.loc[tuning_results['rmse'].idxmin()]
print("Best parameters:")
```

```
print(best_params)
```

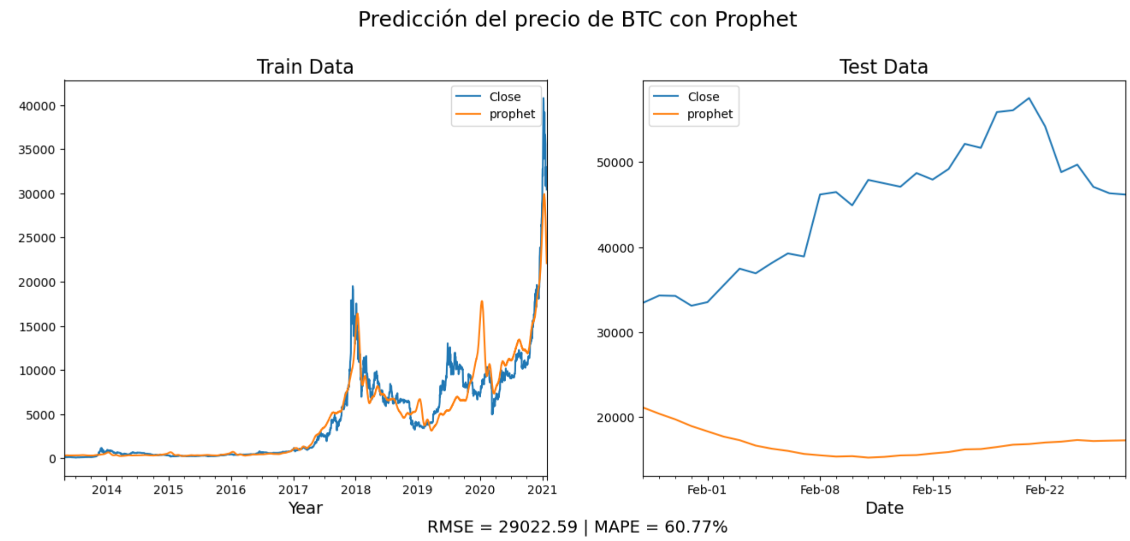
```
Best parameters:
changepoint_prior_scale            0.005
seasonality_prior_scale               10
seasonality_mode          multiplicative
daily_seasonality                   True
rmse                         6768.708651
Name: 27, dtype: object
```

Aplicación de los hiperparámetros sobre el modelo y fiteo

[53]:
```
# Se crea el modelo con los mejores parámetros obtenidos
prophet_model = Prophet(changepoint_prior_scale =␣
 ↪best_params['changepoint_prior_scale'],
                        seasonality_prior_scale =␣
 ↪best_params['seasonality_prior_scale'],
                        seasonality_mode = best_params['seasonality_mode'],
                        daily_seasonality = best_params['daily_seasonality'])

# Se fitea el modelo
prophet_model.fit(df_train)

# Se crea un Dataframe para realizar las predicciones
future = prophet_model.make_future_dataframe(periods=len(df_test), freq='D')

# Se realizan las predicciones
forecast = prophet_model.predict(future)
```

[54]:
```
# Se insertan los resultados en el dataset utilizando el mismo esquema que el␣
 ↪de los anteriores modelos
forecast_values = forecast['yhat'].values
df_train['prophet'] = forecast_values[:len(df_train)]
df_test['prophet'] = forecast_values[-len(df_test):]
```

[55]:
```
plot_time_series(df_train, df_test, "prophet")
```



Se almacenan los valores de RMSE y MAPE

[56]:
```
df_results.loc[6, "Model"] = "Prophet"
df_results.loc[6, "RMSE"] = round(RMSE(df_test['Close'], df_test['prophet']),1)
df_results.loc[6, "MAPE"] = round(MAPE(df_test['Close'], df_test['prophet']),1)
df_results
```

[56]:

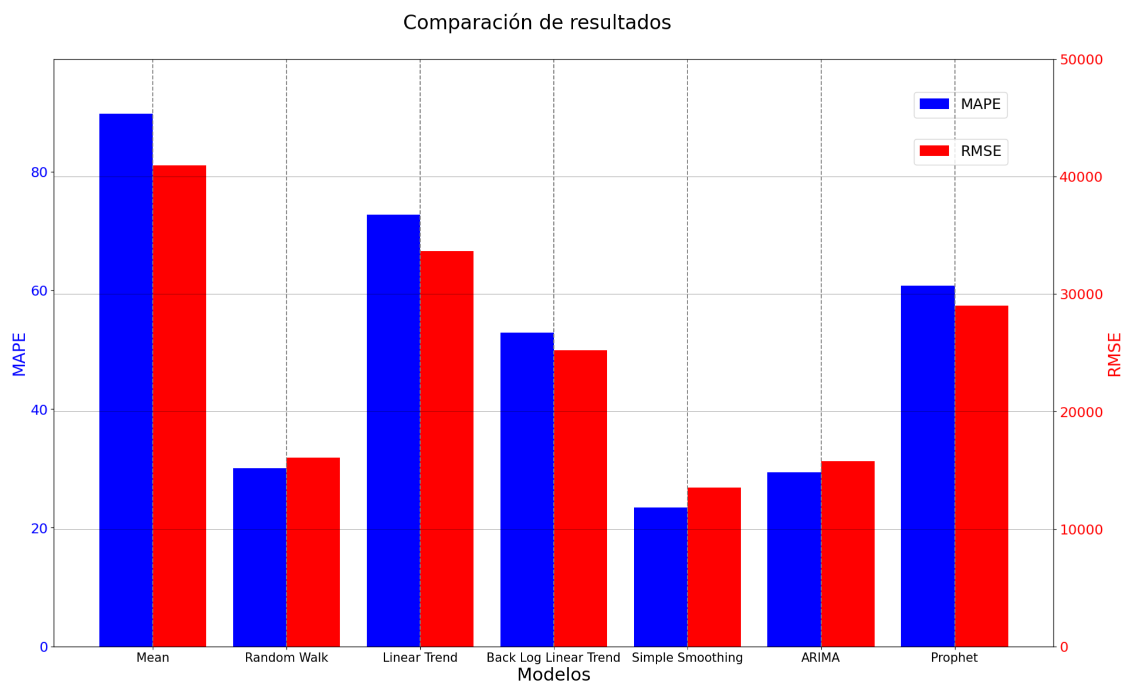
	Model	RMSE	MAPE
0	Mean	40968.4	89.8
1	Random Walk	16049.4	30.0
2	Linear Trend	33666.9	72.8
3	Back Log Linear	25190.7	52.9
4	Simple Smoothing	13517.5	23.4
5	ARIMA	15779.1	29.3
6	Prophet	29022.6	60.8

4 3) Comparación de resultados

Análisis de RMSE y MAPE visualizado

[57]:
```
fig, ax1 = plt.subplots(figsize = (22, 13))
ax1.set_xlabel('Modelos', fontsize = 22)
ax1.set_ylabel('MAPE', fontsize = 20, color = 'b')
ax1.bar(df_results.index - 0.2, df_results.MAPE, width = 0.4, color = 'b',␣
↪linewidth = 2, label = "MAPE")
ax1.tick_params(axis = 'y', labelcolor = 'b', labelsize = 17)
ax1.tick_params(axis = 'x', labelsize = 15)
ax1.set_ylim([0, 99])

ax2 = ax1.twinx()
```

```
ax2.set_ylabel('RMSE', fontsize = 20, color = 'r')
ax2.bar(df_results.index + 0.2, df_results.RMSE, width = 0.4, color = 'r',␣
 ↪linewidth = 2, label = "RMSE")
ax2.tick_params(axis = 'y', labelcolor = 'r', labelsize = 17)
ax2.set_ylim([0, 50000])

plt.axvline(x = 'Mean', color = 'grey', linestyle = '--', lw = 1.3)
plt.axvline(x = 'Random Walk',color = 'grey', linestyle = '--', lw = 1.3)
plt.axvline(x = 'Linear Trend', color = 'grey', linestyle = '--', lw = 1.3)
plt.axvline(x = 'Back Log Linear Trend' , color = 'grey', linestyle = '--', lw␣
 ↪= 1.3)
plt.axvline(x = 'Simple Smoothing', color = 'grey', linestyle = '--', lw = 1.3)
plt.axvline(x = 'ARIMA', color = 'grey', linestyle = '--', lw = 1.3)
plt.axvline(x = 'Prophet', color = 'grey', linestyle = '--', lw = 1.3)

plt.grid(which = 'major', axis = 'y', color = 'black', lw = 0.4, alpha = 0.6)
plt.suptitle("Comparación de resultados", fontsize = 24, y = 0.94)
legend1 = ax1.legend(loc = (0.86, 0.9), fontsize = 18)
legend2 = ax2.legend(loc = (0.86, 0.82), fontsize = 18)

plt.show()
```



5 3) Análisis de estacionalidad y autocorrelación

A continuación, se analizarán ACF, PACF y Dickey Fuller sobre la serie de tiempo de BTC y sobre los residuos de algunos modelos

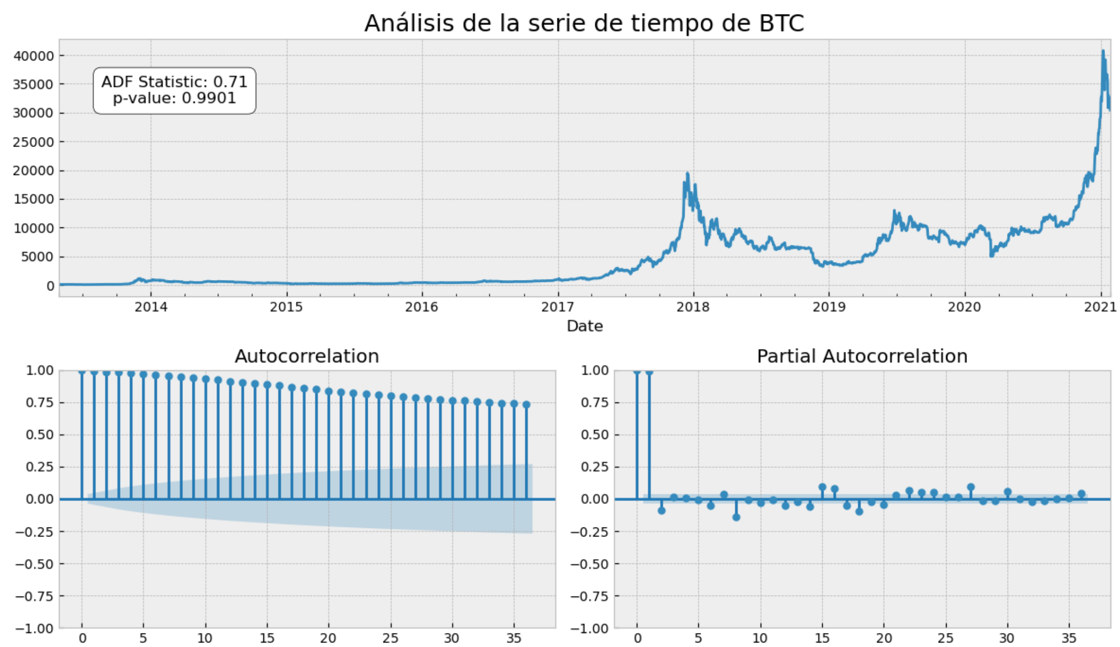
Se crea una función para plotear una serie con información sobre ACF, PACF y su estacionalidad:

[77]:
```python
def tsplot(y, model_name = None, lags = None, figsize = (12, 7), style = 'bmh'):
    """
    Plotea la serie de tiempo, el ACF y PACF y el test de Dickey-Fuller

    y - serie de tiempo
    model_name - nombre del modelo con default None para cuando se desee␣
↪plotear la serie de tiempo de BTC, en vez de los residuos del modelo
    lags - cuántos lags incluir para el cálculo de la ACF y PACF
    """
    if not isinstance(y, pd.Series):
        y = pd.Series(y)

    with plt.style.context(style):
        fig = plt.figure(figsize=figsize)
        layout = (2, 2)

        # Se definen ejes
        ts_ax = plt.subplot2grid(layout, (0, 0), colspan=2)
        acf_ax = plt.subplot2grid(layout, (1, 0))
        pacf_ax = plt.subplot2grid(layout, (1, 1))

        y.plot(ax=ts_ax)

        # Se obtiene el p-value con H0: raiz unitaria presente
        result = sm.tsa.stattools.adfuller(y)
        p_value = result[1]

        if model_name is not None:
            ts_ax.set_title(f"Análisis de los residuos del modelo␣
↪{model_name}", fontsize=18)
        else:
            ts_ax.set_title("Análisis de la serie de tiempo de BTC",␣
↪fontsize=18)

        # Se agrega el texto del Dickey Fuller
        adf_text = f"ADF Statistic: {round(result[0],2)}\n"
        adf_text += f"p-value: {round(result[1],4)}"

        # Se añade el texto del Dickey Fuller como anotación
```

```
        annotation_box = dict(boxstyle='square,pad=0.5', facecolor='white',␣
↪edgecolor='black', alpha=1)
        annotation = ts_ax.annotate(adf_text, xy=(0.11, 0.75), xycoords='axes␣
↪fraction', ha='center', fontsize=12, bbox=annotation_box)

        # Se añade un cuadro para el texto de Dickey Fuller
        annotation_bbox = annotation.get_bbox_patch()
        annotation_bbox.set_boxstyle("round,pad=0.3", pad=0.5)

        # Plot de autocorrelacion
        smt.graphics.plot_acf(y, lags=lags, ax=acf_ax)
        # Plot de autocorrelacion parcial
        smt.graphics.plot_pacf(y, lags=lags, ax=pacf_ax)
        plt.tight_layout()
```

5.1 Serie de BTC

[78]:
```
tsplot(df_train['Close'], lags = 36)
```



5.2 Residuos

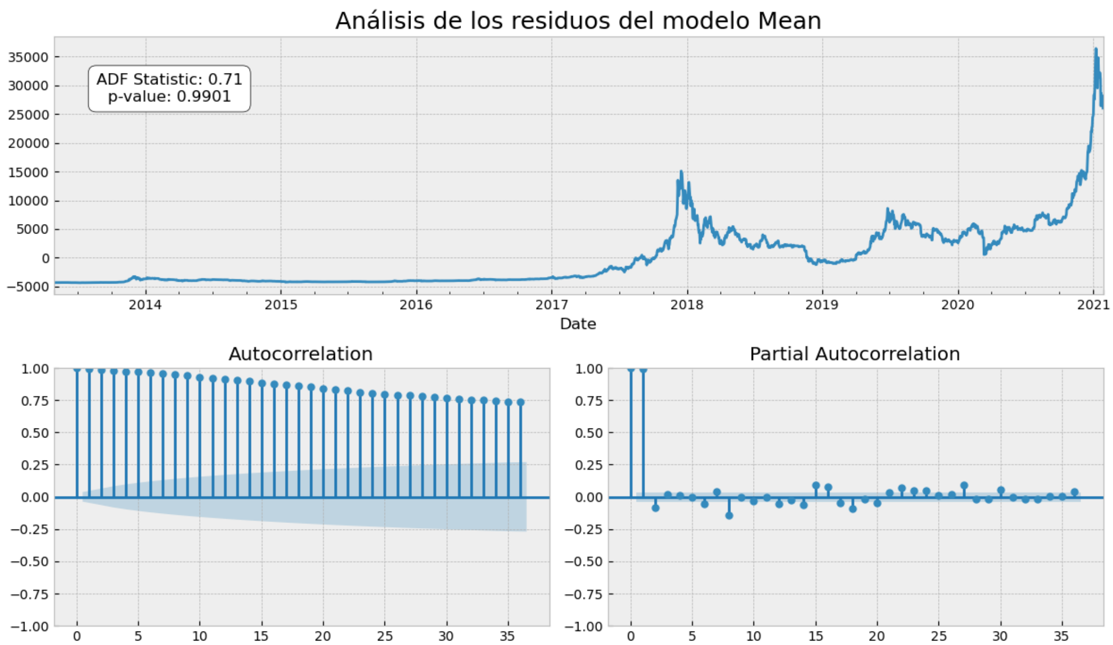
Se crea la variable de cada residuo

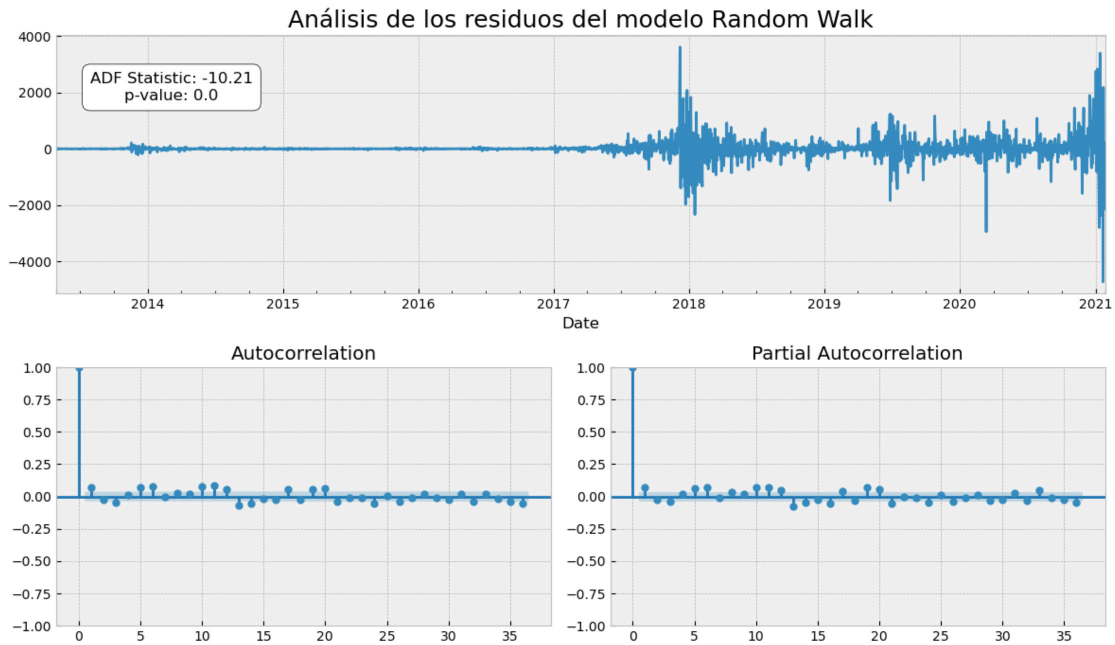
[79]:
```
residue_mean = df_train['Close'] - df_train['mean']
residue_random_walk = df_train['Close'] - df_train['random_walk']
residue_linear_trend = df_train['Close'] - df_train['linear_trend']
residue_back_linear = df_train['Close'] - df_train['back_linear']
```

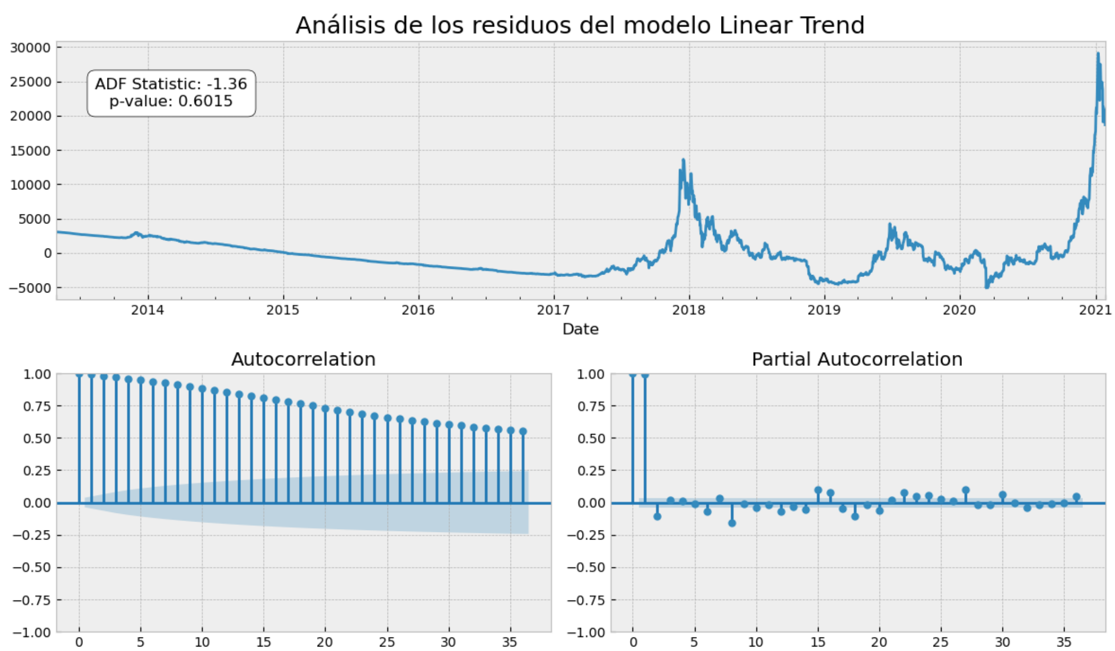
```
residue_simple_smoothing = df_train['Close'] - df_train['simple_smoothing']
residue_arima = df_train['Close'] - df_train['arima']
residue_prophet = df_train['Close'] - df_train['prophet']
```

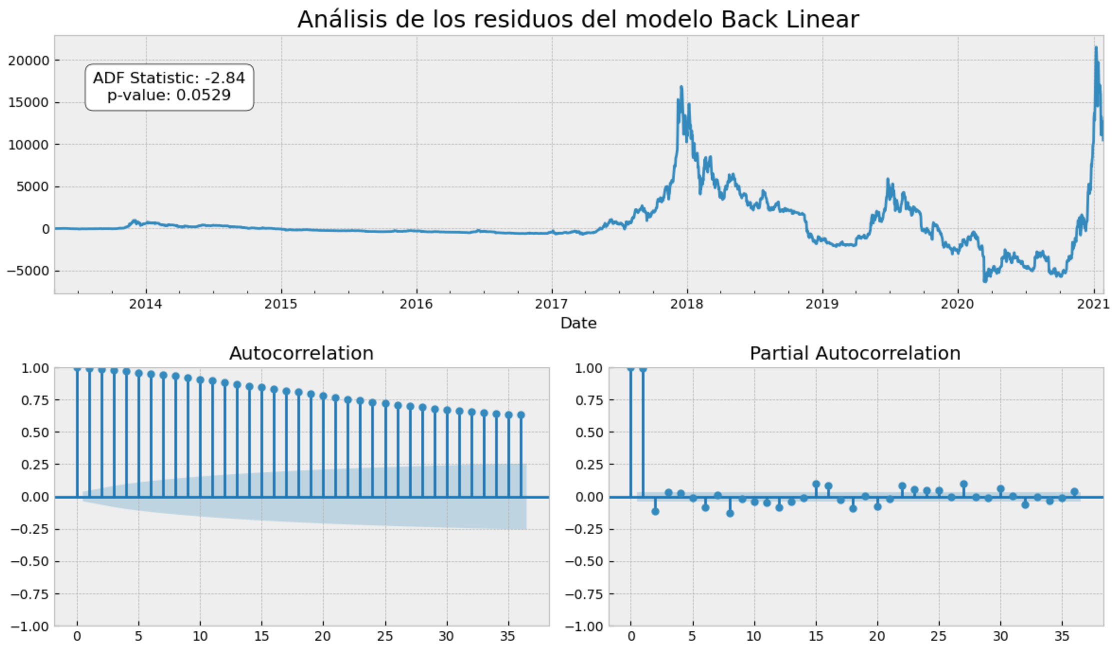
Se plotean todos los residuos

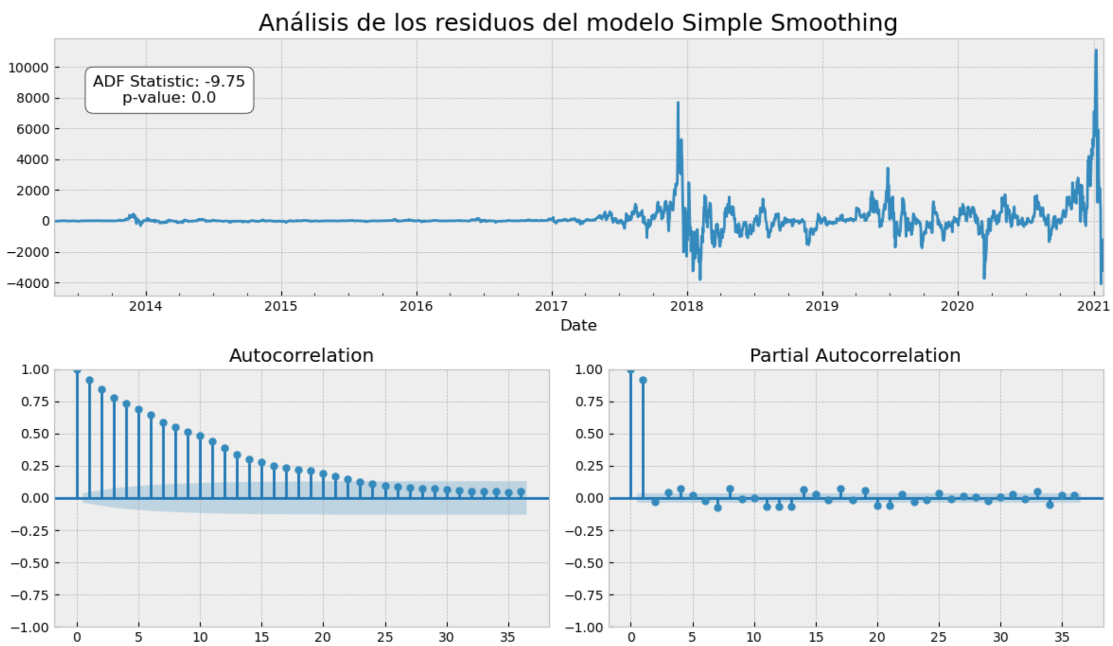
[80]:
```
residues = [residue_mean, residue_random_walk, residue_linear_trend,␣
 ↪residue_back_linear, residue_simple_smoothing, residue_arima,␣
 ↪residue_prophet]

for residue, name in zip(residues, ["Mean", "Random Walk", "Linear Trend",␣
 ↪"Back Linear", "Simple Smoothing", "ARIMA", "Prophet"]):
    tsplot(residue, model_name=name, lags=36)
```

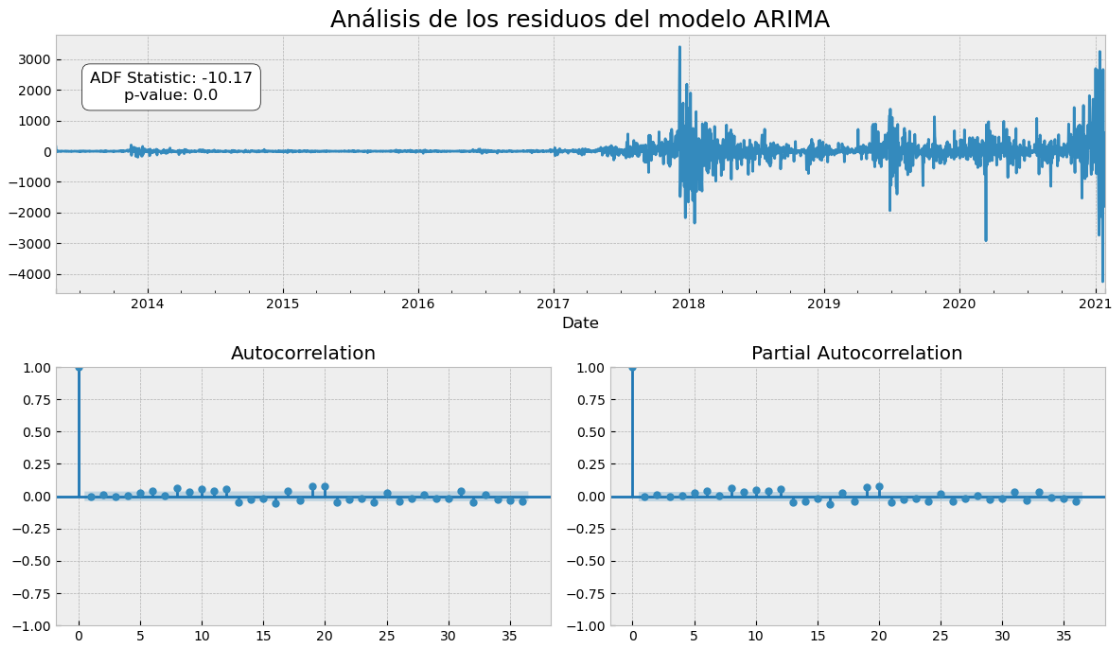


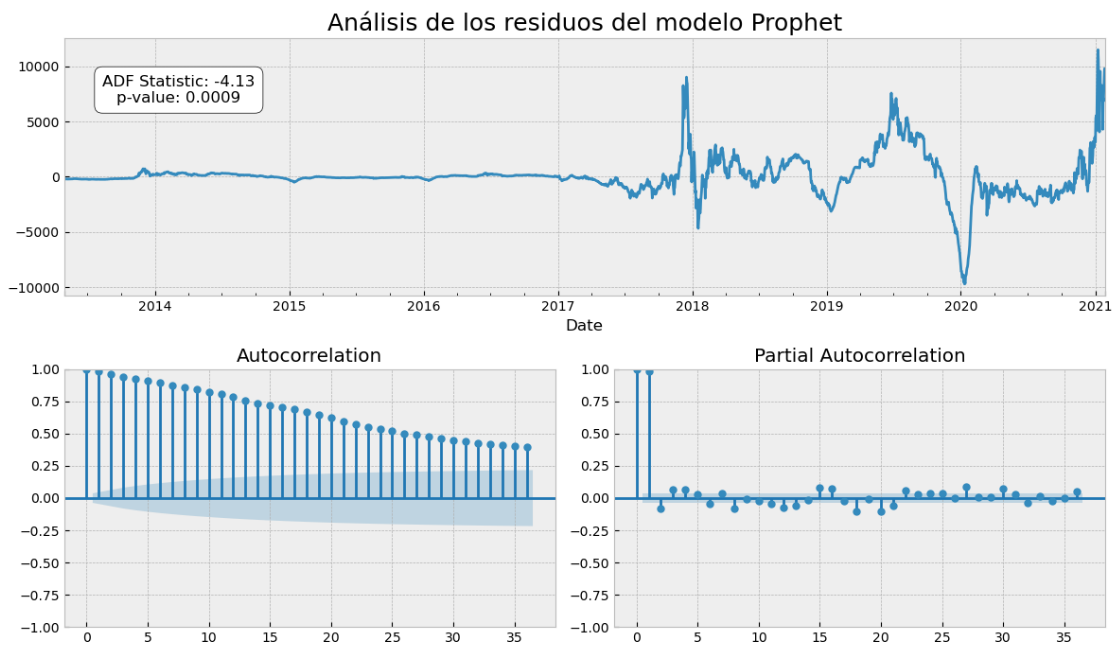

Análisis de los residuos del modelo Random Walk
ADF Statistic: -10.21
p-value: 0.0
Date
Autocorrelation
Partial Autocorrelation


Análisis de los residuos del modelo Linear Trend
ADF Statistic: -1.36
p-value: 0.6015
Date
Autocorrelation
Partial Autocorrelation


Análisis de los residuos del modelo Back Linear
ADF Statistic: -2.84
p-value: 0.0529
Date
Autocorrelation
Partial Autocorrelation


Análisis de los residuos del modelo Simple Smoothing
ADF Statistic: -9.75
p-value: 0.0
Date
Autocorrelation
Partial Autocorrelation





5.3 Análisis

Serie de tiempo de BTC La estadística ADF de 0.7, junto con un p value de 0.99, indican en conjunto que la prueba de Dickey-Fuller Aumentada no logra proporcionar una evidencia sólida en contra de la presencia de una raíz unitaria en la serie de tiempo del BTC. Esto sugiere que los datos siguen siendo no estacionarios, potencialmente mostrando tendencias y patrones que pueden afectar el análisis y la predicción.

Residuos de modelos Algunos valores de p value y de estadística de ADF son resultados esperados:

1) Los valores para los residuos de Mean son idénticos a los de la serie de BTC.
2) Los valores para los residuos de Random Walk, Simple Smoothing con poco suavizado y ARIMA en la configuración elegida dan estacionariedad perfecta. Esto se debe a que los valores que los modelos proponen para train son sumamente similares a los actuales de train.

Algunas conclusiones que se pueden realizar sobre otros modelos son:

1) Linear Trend
    - Estadística ADF: -1.36
    - Valor p: 0.6
    - Análisis: La estadística ADF de -1.36 sugiere no estacionariedad, ya que no es fuertemente negativa. El valor p alto de 0.6 indica que no hay suficiente evidencia para rechazar la hipótesis nula de una raíz unitaria, lo que respalda la idea de no estacionariedad.
2) Back Log Linear Trend
    - Estadística ADF: -2.84
    - Valor p: 0.05
    - Análisis: La estadística ADF de -2.84 es más negativa, lo que sugiere una evidencia más fuerte en contra de la presencia de una raíz unitaria e indica una mayor probabilidad de estacionariedad. El valor p de 0.05 es relativamente bajo, lo que indica que existe alguna evidencia para rechazar la hipótesis nula de una raíz unitaria, lo cual concuerda con la estadística ADF que sugiere potencial estacionariedad.
3) Prophet
    - Estadística ADF: -4.13
    - Valor p: 0.0009
    - Análisis: La estadística ADF muy baja de -4.13 sugiere una evidencia sólida en contra de la presencia de una raíz unitaria e indica una alta probabilidad de estacionariedad. El valor p muy bajo de 0.0009 confirma esta evidencia, rechazando fuertemente la hipótesis nula de una raíz unitaria y respaldando la conclusión de estacionariedad.