time_series_real_data_sunspots

January 6, 2026

1 Time Series Analysis Notebook (Real Dataset)

This notebook demonstrates an end-to-end time series workflow on a **real dataset**: **Monthly Sunspot Numbers** (commonly used in time series research and included with `statsmodels`).

It covers: - Data loading and exploration - Visualization and seasonality/trend inspection - Stationarity checks (ADF/KPSS) - ACF/PACF diagnostics - Baseline forecasting - ARIMA model selection and fitting - Forecasting and accuracy evaluation

[1]:
```
# If you're running this on a fresh environment, install dependencies:
# !pip -q install statsmodels pandas numpy matplotlib

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt

from statsmodels.datasets import sunspots
from statsmodels.tsa.seasonal import STL
from statsmodels.tsa.stattools import adfuller, kpss
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima.model import ARIMA

from sklearn.metrics import mean_absolute_error, mean_squared_error

plt.rcParams['figure.figsize'] = (12, 4)
```

1.1 1) Load the data

The `sunspots` dataset contains historical monthly sunspot activity. We'll convert it into a proper `pandas.Series` with a datetime index.

[2]:
```
data = sunspots.load_pandas().data
data.head(), data.tail(), data.shape
```

[2]:
```
(      YEAR  SUNACTIVITY
 0  1700.0          5.0
 1  1701.0         11.0
 2  1702.0         16.0
 3  1703.0         23.0
```

```
 4  1704.0         36.0,
         YEAR  SUNACTIVITY
 304  2004.0         40.4
 305  2005.0         29.8
 306  2006.0         15.2
 307  2007.0          7.5
 308  2008.0          2.9,
 (309, 2))
```

<img src="https://raw.githubusercontent.com/msfasha/307304-Data-Mining/main/images/sunspots.jpg

[3]:
```python
# The dataset includes columns: 'YEAR' and 'SUNACTIVITY'
# Convert YEAR (float) into monthly dates.
# Many versions store fractional year (e.g., 1700.0833...), which encodes month.
# We'll reconstruct a monthly DateTimeIndex robustly.

year_float = data['YEAR'].to_numpy()
sun = data['SUNACTIVITY'].to_numpy()

# Convert fractional years to month index (1..12)
years = np.floor(year_float).astype(int)
frac = year_float - years
months = np.clip(np.round(frac * 12 + 1).astype(int), 1, 12)

dates = pd.to_datetime(
    {'year': years, 'month': months, 'day': 1}
)

ts = pd.Series(sun, index=dates).sort_index()
ts.name = "Monthly Sunspot Activity"

ts.head(), ts.index.min(), ts.index.max(), ts.isna().sum()
```

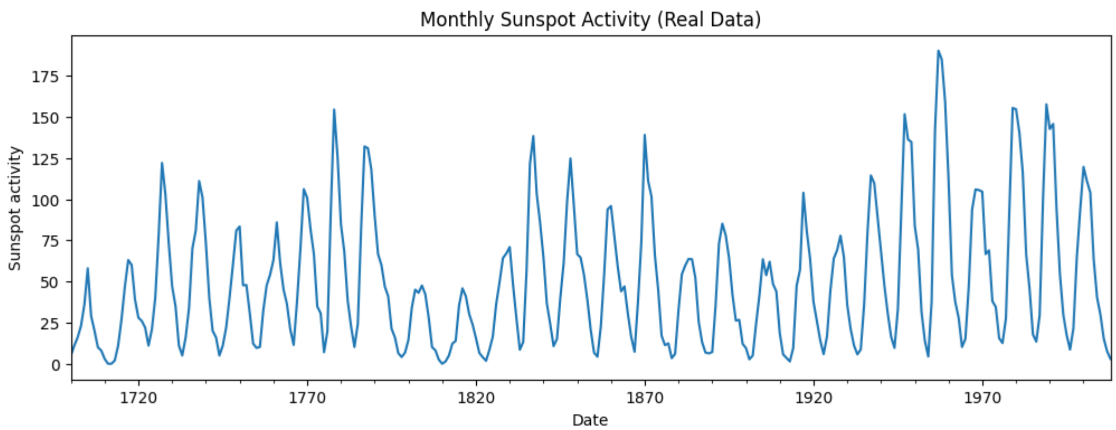
[3]:
```
(1700-01-01     5.0
 1701-01-01    11.0
 1702-01-01    16.0
 1703-01-01    23.0
 1704-01-01    36.0
 Name: Monthly Sunspot Activity, dtype: float64,
 Timestamp('1700-01-01 00:00:00'),
 Timestamp('2008-01-01 00:00:00'),
 0)
```

1.2 2) Quick exploration

[4]:
```python
ts.describe()
```

```
[4]: count     309.000000
     mean       49.752104
     std        40.452595
     min         0.000000
     25%        16.000000
     50%        40.000000
     75%        69.800000
     max       190.200000
     Name: Monthly Sunspot Activity, dtype: float64
```

```
[5]: ts.plot()
     plt.title("Monthly Sunspot Activity (Real Data)")
     plt.xlabel("Date")
     plt.ylabel("Sunspot activity")
     plt.show()
```



1.3 3) Train/Test split

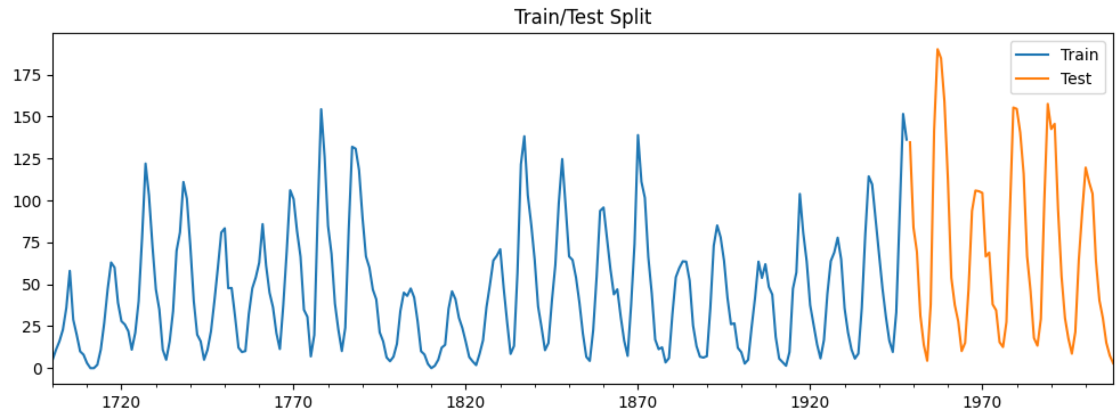
We'll hold out the last few years as a test set.

```
[6]: # Hold out the last 5 years (60 months)
     test_horizon = 60

     train = ts.iloc[:-test_horizon]
     test = ts.iloc[-test_horizon:]

     train.index.min(), train.index.max(), test.index.min(), test.index.max(),␣
      ↪len(train), len(test)
```

```
[6]: (Timestamp('1700-01-01 00:00:00'),
      Timestamp('1948-01-01 00:00:00'),
      Timestamp('1949-01-01 00:00:00'),
```

```
 Timestamp('2008-01-01 00:00:00'),
 249,
 60)
```

[7]:
```python
train.plot(label="Train")
test.plot(label="Test")
plt.title("Train/Test Split")
plt.legend()
plt.show()
```

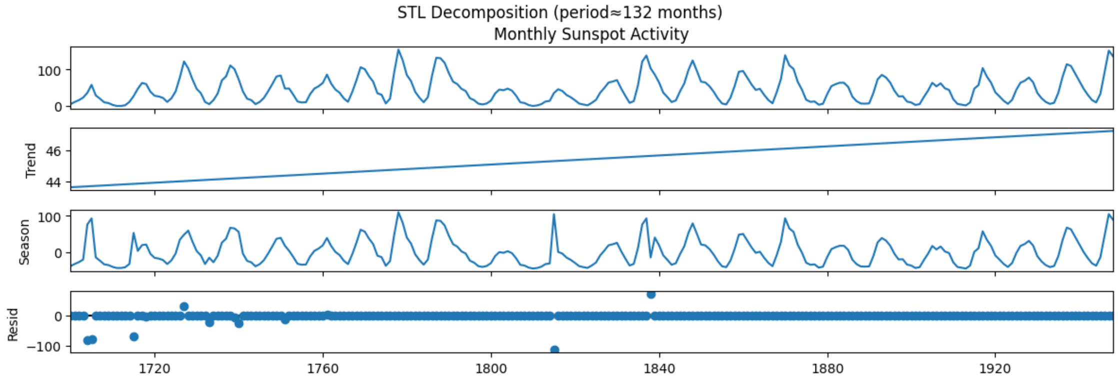


1.4 4) Decomposition with STL

STL (Seasonal-Trend decomposition using Loess) is useful for exploring trend and repeating structure.

Sunspots exhibit **cyclical behavior** (roughly ~11 years), which is not strictly a fixed seasonal pattern like monthly retail seasonality, but STL can still provide insight.

[8]:
```python
# Choose a period. For sunspots, an ~11-year cycle   132 months.
stl = STL(train, period=132, robust=True)
res = stl.fit()

res.plot()
plt.suptitle("STL Decomposition (period 132 months)", y=1.02)
plt.show()
```



1.5 5) Stationarity checks (ADF and KPSS)

- **ADF** tests the null hypothesis that a unit root exists (non-stationary). Low p-value → reject null → more stationary (is stationary).
- **KPSS** tests the null hypothesis that the series is stationary. Low p-value → reject null → non-stationary.

[9]:
```
def adf_test(series):
    out = adfuller(series.dropna(), autolag="AIC")
    return {
        "ADF statistic": out[0],
        "p-value": out[1],
        "n_lags": out[2],
        "n_obs": out[3]
    }

def kpss_test(series, regression="c"):
    stat, pval, lags, crit = kpss(series.dropna(), regression=regression,␣
 ↪nlags="auto")
    return {
        "KPSS statistic": stat,
        "p-value": pval,
        "n_lags": lags
    }

adf_test(train), kpss_test(train)
```

```
C:\Users\me\AppData\Local\Temp\ipykernel_16104\4149292106.py:11:
InterpolationWarning: The test statistic is outside of the range of p-values
available in the
look-up table. The actual p-value is greater than the p-value returned.

  stat, pval, lags, crit = kpss(series.dropna(), regression=regression,
nlags="auto")
```

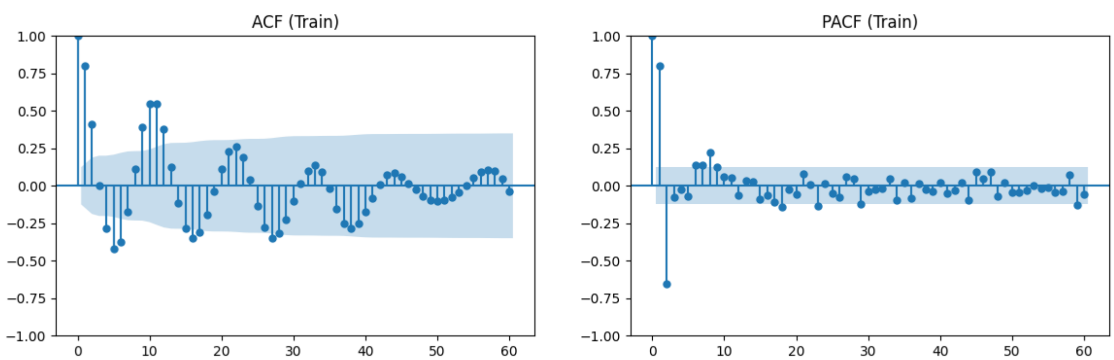
[9]:
```
({'ADF statistic': -2.9748603692623097,
  'p-value': 0.037309759442603924,
  'n_lags': 8,
  'n_obs': 240},
 {'KPSS statistic': 0.10762408792487697, 'p-value': 0.1, 'n_lags': 6})
```

ADF vs KPSS (train window) - ADF statistic -2.97, p 0.037 → reject the unit-root null at 5% → evidence for stationarity. - KPSS statistic 0.108, p 0.10 → fail to reject the stationarity null → also supports stationarity.
- Both tests agree the level series looks stationary, so differencing is likely unnecessary before fitting AR/MA terms on this window.

1.6 6) ACF / PACF diagnostics

These plots help inform AR/MA order choices: - ACF: autocorrelation at different lags - PACF: partial autocorrelation after removing intermediate effects

[10]:
```
fig, ax = plt.subplots(1, 2, figsize=(14, 4))
plot_acf(train, lags=60, ax=ax[0])
plot_pacf(train, lags=60, ax=ax[1], method="ywm")
ax[0].set_title("ACF (Train)")
ax[1].set_title("PACF (Train)")
plt.show()
```



- **ACF (Train):** Shows a slowly decaying, oscillatory pattern (alternating positive/negative correlations), indicating a persistent **cyclical** time series rather than white noise. No sharp cutoff → not a pure MA process.
- **PACF (Train):** Has **significant spikes at lags 1 and 2**, with most later lags within the confidence bounds → classic signature of an **AR(2)** structure.
- **Implication:** The series appears **stationary (no differencing needed, d = 0)** but with a pronounced cycle (consistent with the ~11-year sunspot cycle).
- **Recommended starting model: ARIMA(2,0,0)** (i.e., AR(2)); optionally compare against nearby alternatives like ARIMA(3,0,0) or ARIMA(2,0,1) using AIC/BIC and residual diagnostics.

1.7 7) Baseline forecast

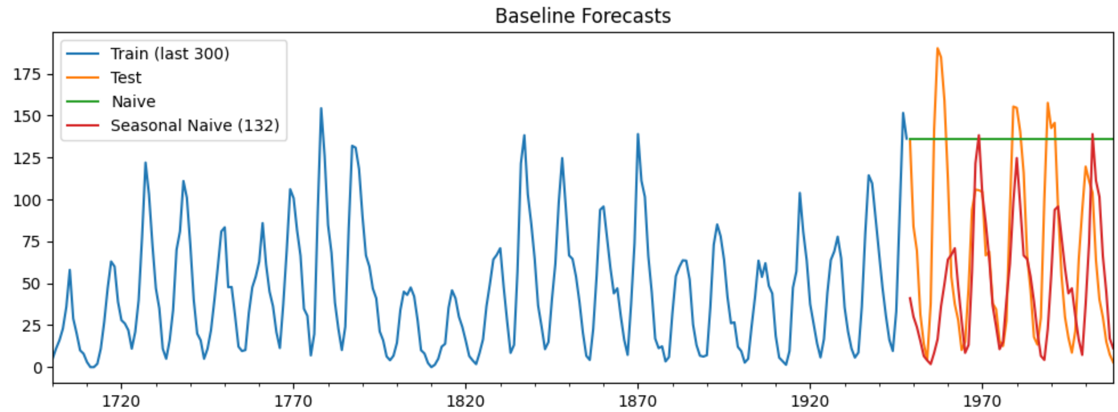
A simple baseline helps contextualize more complex models. We'll use: - **Naive** forecast: last observed value - **Seasonal naive (cycle)**: last value from 132 months ago (approx sunspot cycle)

[11]:
```python
def naive_forecast(train, steps):
    return pd.Series([train.iloc[-1]] * steps, index=test.index)

def seasonal_naive_forecast(train, steps, season_len):
    # For each step, use the value from season_len months earlier
    idx = np.arange(len(train), len(train) + steps)
    vals = [train.iloc[i - season_len] if (i - season_len) >= 0 else train.
 ↪iloc[-1] for i in idx]
    return pd.Series(vals, index=test.index)

naive_pred = naive_forecast(train, len(test))
snaive_pred = seasonal_naive_forecast(train, len(test), season_len=132)

def eval_forecast(y_true, y_pred, label="model"):
    mae = mean_absolute_error(y_true, y_pred)
    mse = mean_squared_error(y_true, y_pred)
    rmse = np.sqrt(mse)
    return pd.Series({"MAE": mae, "RMSE": rmse}, name=label)

pd.concat([
    eval_forecast(test, naive_pred, "Naive"),
    eval_forecast(test, snaive_pred, "Seasonal Naive (132)")
], axis=1)
```

[11]:

	Naive	Seasonal Naive (132)
MAE	71.850000	39.693333
RMSE	83.172025	55.591825

[12]:
```python
plt.figure(figsize=(12,4))
train.iloc[-300:].plot(label="Train (last 300)")
test.plot(label="Test")
naive_pred.plot(label="Naive")
snaive_pred.plot(label="Seasonal Naive (132)")
plt.title("Baseline Forecasts")
plt.legend()
plt.show()
```



1.8 8) ARIMA model selection (lightweight grid search)

We'll do a small ARIMA(p,d,q) search using AIC on the training data.
For many real projects you'd expand this, add seasonal terms (SARIMA), and/or use time series cross-validation.

Note: Sunspots have strong cyclicality; a pure ARIMA may not perfectly capture multi-year cycles, but it's useful for demonstration.

[13]:
```
import warnings
warnings.filterwarnings("ignore")

def arima_grid_search(series, p_range=range(0, 6), d_range=range(0, 3),␣
↪q_range=range(0, 6)):
    best = {"aic": np.inf, "order": None, "model": None}
    for p in p_range:
        for d in d_range:
            for q in q_range:
                if p == 0 and d == 0 and q == 0:
                    continue
                try:
                    model = ARIMA(series, order=(p,d,q)).fit()
                    if model.aic < best["aic"]:
                        best = {"aic": model.aic, "order": (p,d,q), "model":␣
↪model}
                except Exception:
                    continue
    return best

best = arima_grid_search(train, p_range=range(0,6), d_range=range(0,3),␣
↪q_range=range(0,6))
best["order"], best["aic"]
```

[13]: ((4, 1, 4), 2032.3223382311585)

1.9 9) Fit best ARIMA and forecast

[14]:
```python
best_order = best["order"]
arima_fit = ARIMA(train, order=best_order).fit()
arima_fit.summary()
```

[14]:

Dep. Variable:	Monthly Sunspot Activity	**No. Observations:**	249
Model:	ARIMA(4, 1, 4)	**Log Likelihood**	-1007.161
Date:	Tue, 06 Jan 2026	**AIC**	2032.322
Time:	06:05:24	**BIC**	2063.943
Sample:	01-01-1700	**HQIC**	2045.052
	- 01-01-1948		
Covariance Type:	opg		

	coef	**std err**	**z**	**P> |z|**	**[0.025**	**0.975]**
ar.L1	0.3083	0.076	4.054	0.000	0.159	0.457
ar.L2	0.4090	0.056	7.309	0.000	0.299	0.519
ar.L3	0.0234	0.073	0.321	0.748	-0.119	0.166
ar.L4	-0.7126	0.061	-11.666	0.000	-0.832	-0.593
ma.L1	-0.0789	0.083	-0.950	0.342	-0.242	0.084
ma.L2	-0.6739	0.046	-14.544	0.000	-0.765	-0.583
ma.L3	-0.4903	0.067	-7.359	0.000	-0.621	-0.360
ma.L4	0.6604	0.075	8.863	0.000	0.514	0.806
sigma2	196.0160	15.522	12.628	0.000	165.593	226.439

Ljung-Box (L1) (Q):	0.54	**Jarque-Bera (JB):**	26.87
Prob(Q):	0.46	**Prob(JB):**	0.00
Heteroskedasticity (H):	1.09	**Skew:**	0.43
Prob(H) (two-sided):	0.71	**Kurtosis:**	4.36

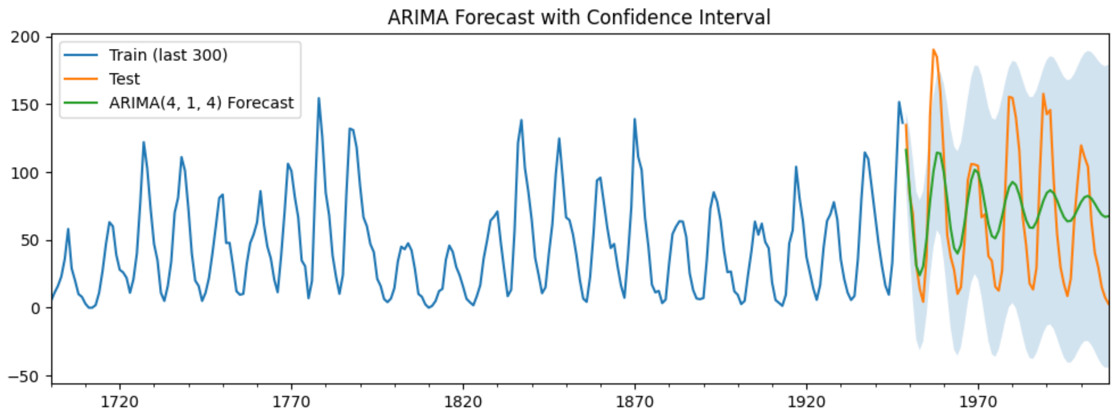
Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).

[15]:
```python
forecast_res = arima_fit.get_forecast(steps=len(test))
pred = forecast_res.predicted_mean
ci = forecast_res.conf_int()

metrics = eval_forecast(test, pred, f"ARIMA{best_order}")
metrics
```

[15]:
```
MAE     32.326951
RMSE    38.581238
Name: ARIMA(4, 1, 4), dtype: float64
```

[16]:
```python
plt.figure(figsize=(12,4))
train.iloc[-300:].plot(label="Train (last 300)")
test.plot(label="Test")
```

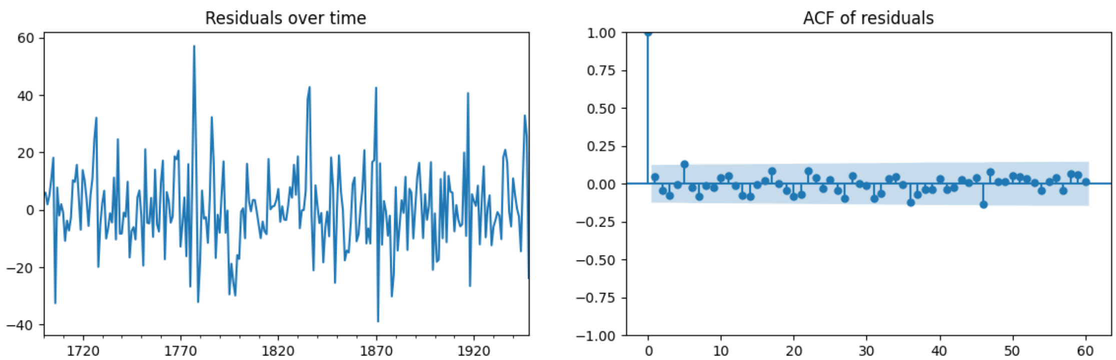
```
pred.plot(label=f"ARIMA{best_order} Forecast")

plt.fill_between(ci.index, ci.iloc[:,0], ci.iloc[:,1], alpha=0.2)
plt.title("ARIMA Forecast with Confidence Interval")
plt.legend()
plt.show()
```

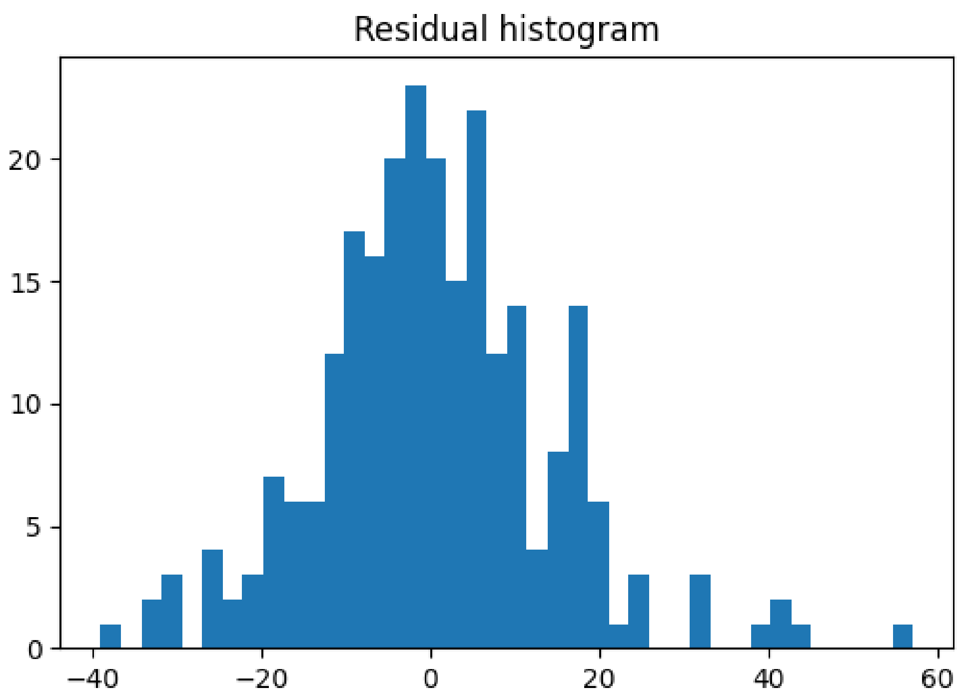


1.10 10) Residual diagnostics

A good model's residuals should look roughly like white noise: - No obvious autocorrelation left - Mean near zero - Roughly stable variance

[17]:
```
resid = arima_fit.resid

fig, ax = plt.subplots(1, 2, figsize=(14,4))
resid.plot(ax=ax[0])
ax[0].set_title("Residuals over time")
plot_acf(resid.dropna(), lags=60, ax=ax[1])
ax[1].set_title("ACF of residuals")
plt.show()

plt.figure(figsize=(6,4))
plt.hist(resid.dropna(), bins=40)
plt.title("Residual histogram")
plt.show()
```





1.11 11) Comparison summary

[18]:
```
summary = pd.concat([
    eval_forecast(test, naive_pred, "Naive"),
    eval_forecast(test, snaive_pred, "Seasonal Naive (132)"),
    eval_forecast(test, pred, f"ARIMA{best_order}")
], axis=1).T.sort_values("RMSE")

summary
```

[18]:

	MAE	RMSE
ARIMA(4, 1, 4)	32.326951	38.581238
Seasonal Naive (132)	39.693333	55.591825
Naive	71.850000	83.172025