Time series analysis: ARMA, ARIMA, SARIMA

ΕΠΛ 428: IOT PROGRAMMING

Dr. Panayiotis Kolios
Assistant Professor, Dept. Computer Science,
KIOS CoE for Intelligent Systems and Networks
Office: FST 01, 116
Telephone: +357 22893450 / 22892695
Web: https://www.kios.ucy.ac.cy/pkolios/

Πανεπιστήμιο Κύπρου

- **MA models**: The current value of the series depends linearly on the series' mean and a set of prior (observed) white noise error terms
- **AR models**: The current value of the series depends linearly on its own previous values and on a stochastic term (an imperfectly predictable term)

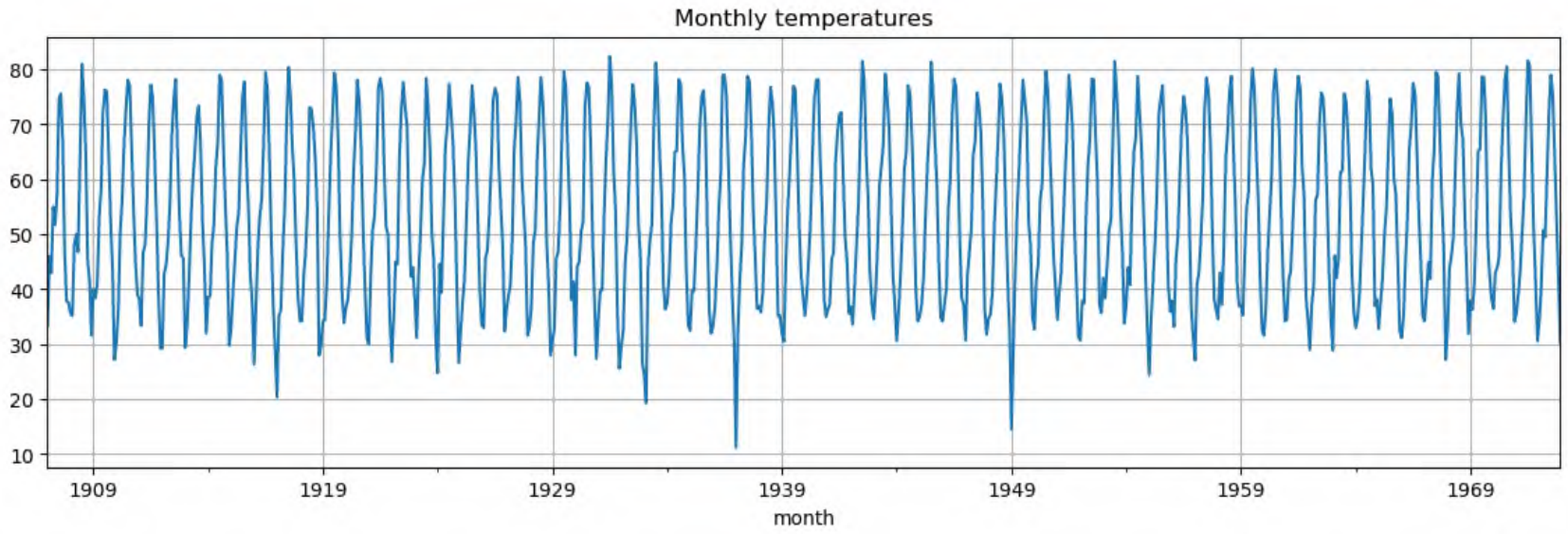
- What about combining AR and MA models, resulting to:
  - ARMA (Autoregressive Moving Average)
  - ARIMA (Autoregressive Integrated Moving Average)
  - SARIMA (ARIMA model for data with seasonality)

- ARMA model combines:
  - Autoregressive (AR) model of order p
  - Moving average (MA) model of order q

- Used when we have autocorrelation between outcomes and their past values, there will be a pattern in the time series
- Predict future values with a confidence level proportional to
  - strength of the relationship
  - proximity to known values (prediction weakens the further out we go)
- ARMA also assumes stationarity
- Fitting requires some observations, >100

- Monthly average temperatures example

```
# load data and convert to datetime
monthly_temp =
pd.read_csv('https://zenodo.org/records/10951538/files/arima_temp.csv?download=1',
                            skipfooter=2,
                            header=0,
                            index_col=0,
                            names=['month', 'temp'],
                            engine='python')
monthly_temp.index = pd.to_datetime(monthly_temp.index)
monthly_temp['temp'].plot(grid=True, figsize=(14, 4), title="Monthly
temperatures");
```



Πανεπιστήμιο Κύπρου

- Monthly average temperatures example

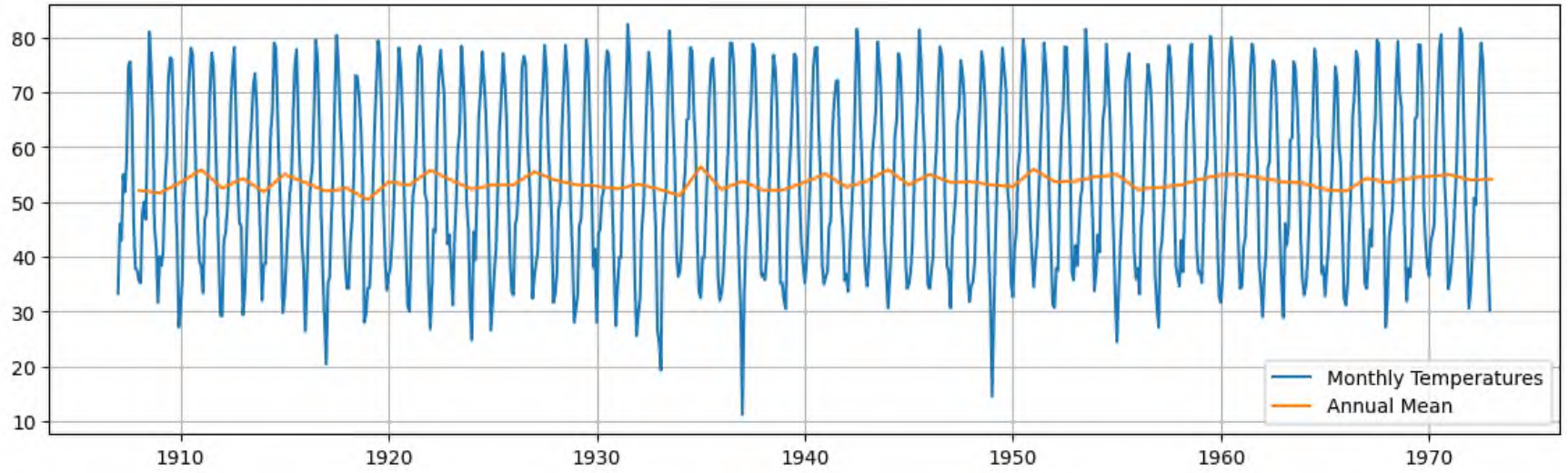
```
# data and stats
monthly_temp.head()
monthly_temp.describe()
```

month	temp
1907-01-01	33.3
1907-02-01	46.0
1907-03-01	43.0
1907-04-01	55.0
1907-05-01	51.8

	temp
count	792.000000
mean	53.553662
std	15.815452
min	11.200000
25%	39.675000
50%	52.150000
75%	67.200000
max	82.400000

- Monthly average temperatures example

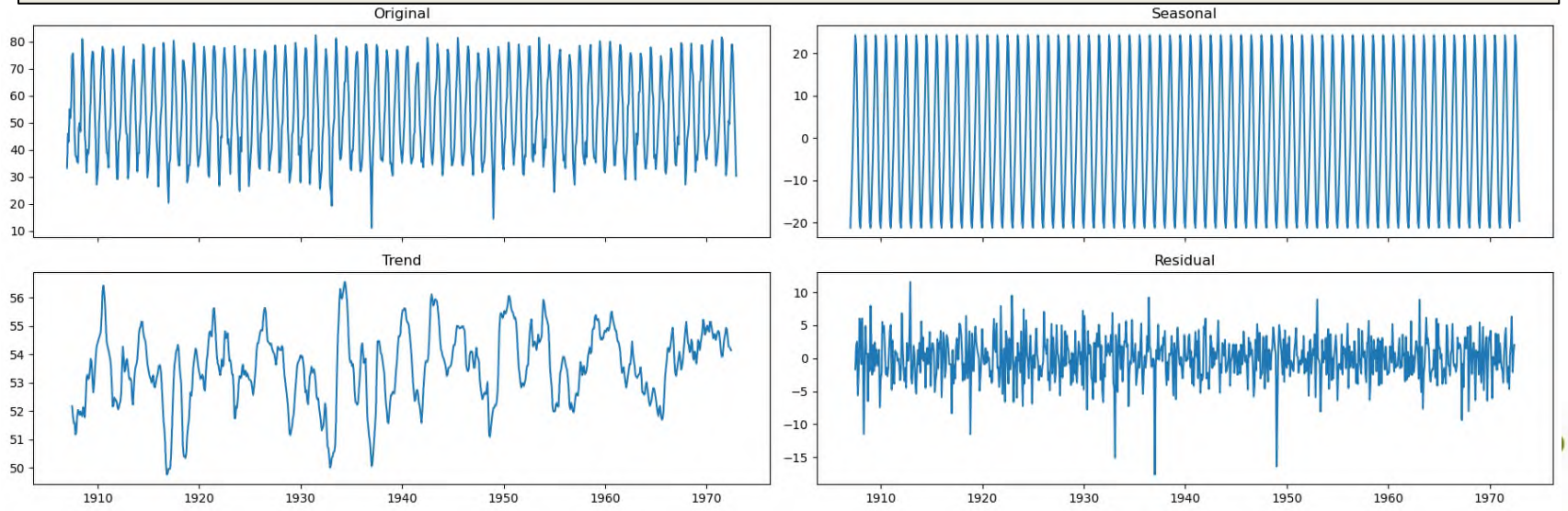
```
# Compute annual mean
annual_temp = monthly_temp.resample('A').mean()
annual_temp.index.name = 'year'

plt.figure(figsize=(14, 4))
plt.plot(monthly_temp, label="Monthly Temperatures")
plt.plot(annual_temp, label="Annual Mean")
plt.grid(); plt.legend();
```



- Indication that mean used to be constant over the years

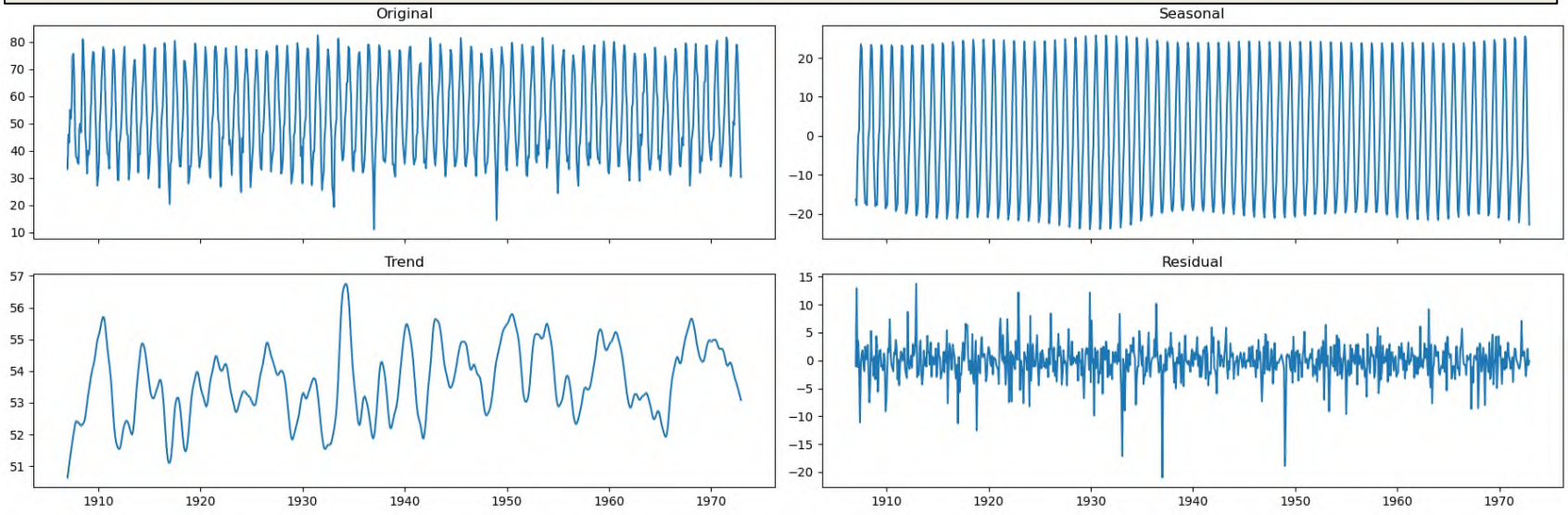
- Trend and seasonal components extraction

```
# seasonal_decompose method
decomposition = seasonal_decompose(x=monthly_temp['temp'], model='additive',
period=12)
seasonal, trend, resid = decomposition.seasonal, decomposition.trend,
decomposition.resid

fig, axs = plt.subplots(2,2, sharex=True, figsize=(18,6))
axs[0,0].plot(monthly_temp['temp'])
axs[0,0].set_title('Original')
axs[0,1].plot(seasonal)
axs[0,1].set_title('Seasonal')
axs[1,0].plot(trend)
axs[1,0].set_title('Trend')
axs[1,1].plot(resid)
axs[1,1].set_title('Residual')
plt.tight_layout()
```



- Trend and seasonal components extraction

```
# STL method
decomposition = STL(endog=monthly_temp['temp'], period=12, seasonal=13,
robust=True).fit()
seasonal, trend, resid = decomposition.seasonal, decomposition.trend,
decomposition.resid

fig, axs = plt.subplots(2,2, sharex=True, figsize=(18,6))
axs[0,0].plot(monthly_temp['temp'])
axs[0,0].set_title('Original')
axs[0,1].plot(seasonal)
axs[0,1].set_title('Seasonal')
axs[1,0].plot(trend)
axs[1,0].set_title('Trend')
axs[1,1].plot(resid)
axs[1,1].set_title('Residual')
plt.tight_layout()
```

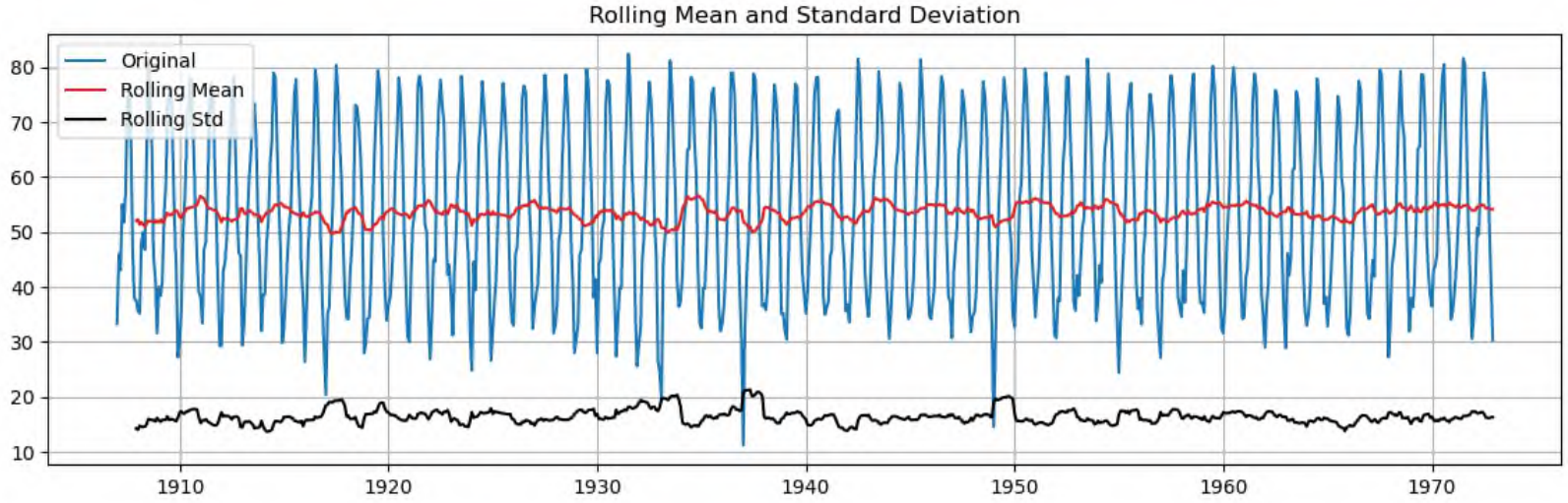


- ARMA modeling stages
  - Model Identification
  - Model Estimation
  - Model Evaluation

- Before performing model identification we need to:
  - Determine if the time series is stationary.
    - Use ADF test
    - Look at the rolling mean and std
  - Determine if the time series has seasonal component.
- ARMA Model identification
  - Model identification consists in finding the orders p and q of the AR and MA components

```
# STL method
adftest(monthly_temp.temp)
```

- ADF Statistic: -6.48
- p-value: 0.000
- Critical Values: ['1%: -3.44', '5%: -2.87', '10%: -2.57']



- Cleary however there is a periodic component and mean and std change locally within the period

Πανεπιστήμιο Κύπρου

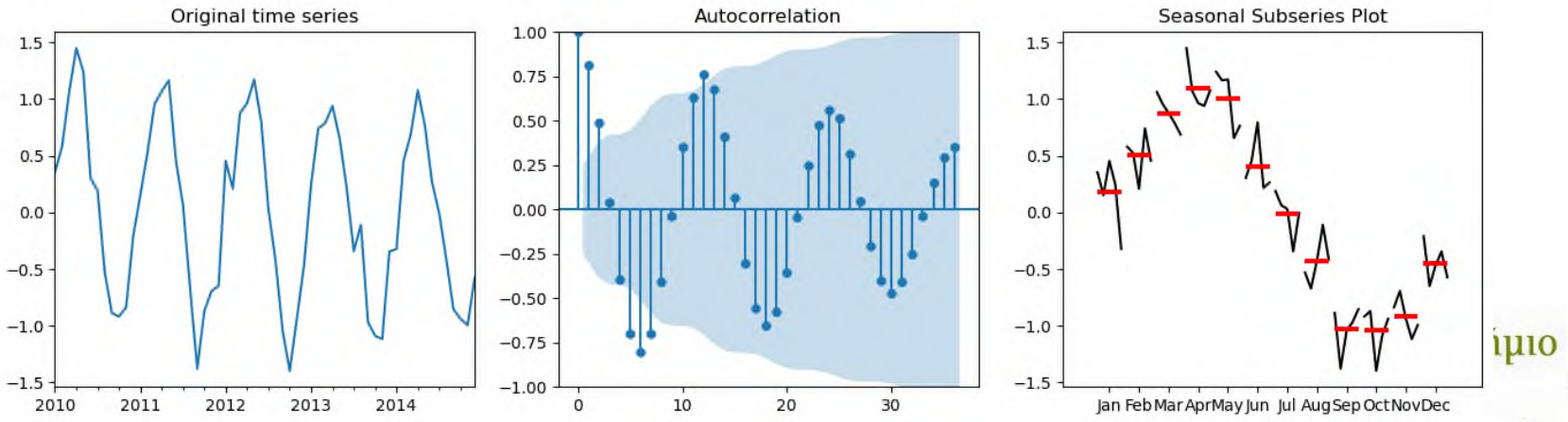
- Determine seasonality
  - Autocorrelation plot
  - Seasonal subseries plot (month plot)
  - Fourier Transform

```
# run ADF on annual means
adftest(annual_temp.temp, plots=False) # no point in plotting the rolling mean/std
here
```

  - ADF Statistic: -7.88
  - p-value: 0.000
  - Critical Values: ['1%: -3.54', '5%: -2.91', '10%: -2.59']

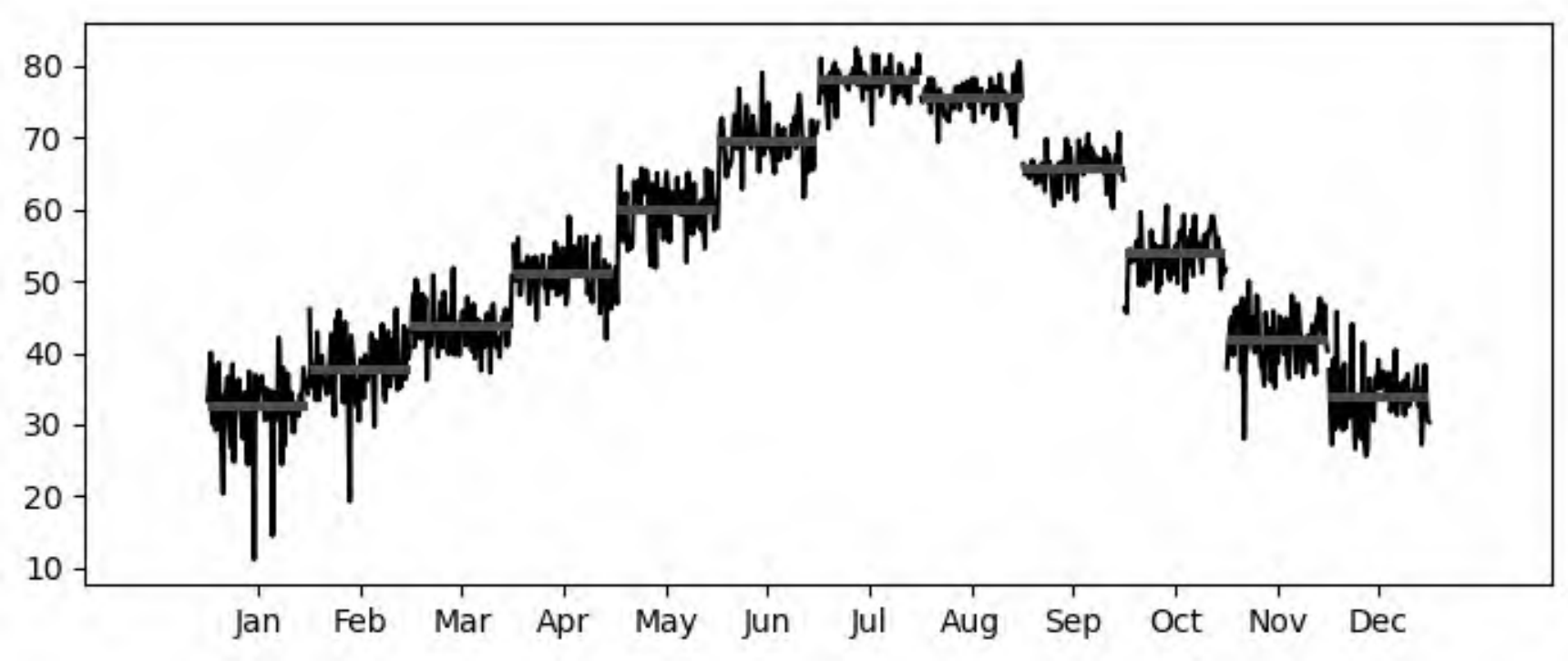
- Determine seasonality

```
# Generate synthetic time series data
dates = pd.date_range(start='2010-01-01', periods=60, freq='M')  # Monthly data
for 5 years
seas = 12 # change this and see how the plots change
data = np.sin(np.arange(60)*2*np.pi/seas) + np.random.normal(loc=0, scale=0.2,
size=60)  # Seasonal data with noise
series = pd.Series(data, index=dates)
fig, axes = plt.subplots(1,3,figsize=(16,4))
series.plot(ax=axes[0], title="Original time series")

# ACF Plot
plot_acf(series, lags=36, ax=axes[1]);

# Convert series to a DataFrame and add a column for the month
df = series.to_frame(name='Value')
df['Month'] = df.index.month

# Seasonal Subseries Plot
month_plot(df['Value'], ax=axes[2]); axes[2].set_title("Seasonal Subseries Plot");
```



- Determine seasonality

```
# and with real data
_, ax = plt.subplots(1,1, figsize=(7,3))
month_plot(monthly_temp, ax=ax)
plt.tight_layout();
```



- Numerical value of the main periodicity use Fourier Transform

```
# and with real data
dominant_period, _, _ = fft_analysis(monthly_temp['temp'].values)
print(f"Dominant period: {np.round(dominant_period)}")
```

  - Dominant period: 12.0

- Removing main seasonality $L = 12$ using differencing

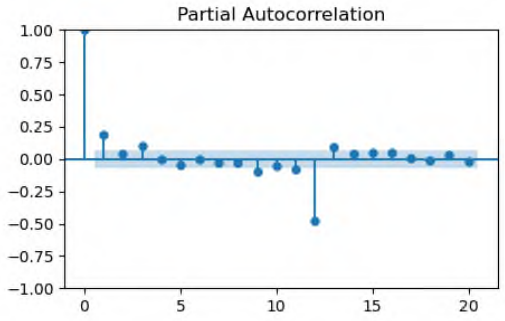
```
# removing seasonality
monthly_temp['Seasonally_Differenced'] = monthly_temp['temp'].diff(12)
# Drop nan
monthly_temp_clean = monthly_temp.dropna()
monthly_temp_clean
```

month	temp	Seasonally_Differenced
1908-01-01	35.6	2.3
1908-02-01	35.2	-10.8
1908-03-01	48.1	5.1
…	…	…
1972-12-01	30.3	-0.3

- Compute AR and MA order p and q with
  - Autocorrelation function (ACF)
  - Partial autocorrelation function (PACF)
- AR(p)
  - Plot 95% confidence intervals on PACF
  - Choose lag p such that PACF becomes insignificant for p+1
  - If process depends on previous values of itself then it is AR
  - If process depends on previous errors then it is an MA
  - AR propagates shocks infinitely
  - AR exhibits exponential decay in ACF and a cut-off in PACF

```
# plot PACF
_, ax = plt.subplots(1, 1, figsize=(5, 3))
plot_pacf(monthly_temp_clean['Seasonally_Differenced'], lags=20, ax=ax);
```

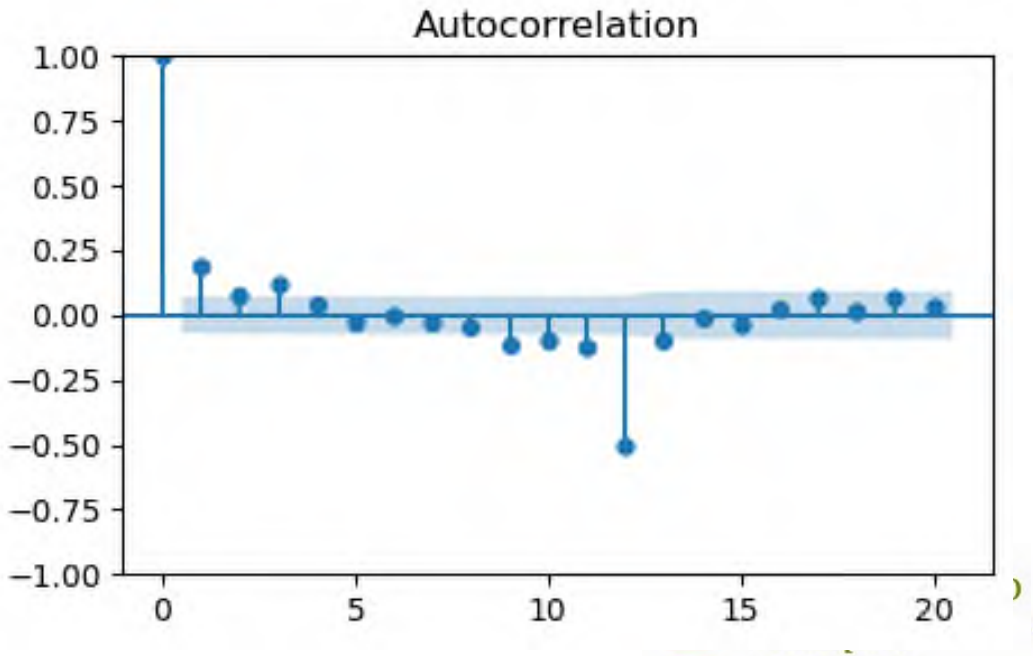
- $p = 1$ or 2 or 3
- High PAC at higher lags
  due to seasonal components



- MA(q)
  - Plot 95% confidence intervals on ACF
  - Choose lag q such that ACF becomes statistically 0 for q+1
  - MA propagates shock up to q lags only
  - MA exhibits exponential decay in PACF and a cut-off in ACF

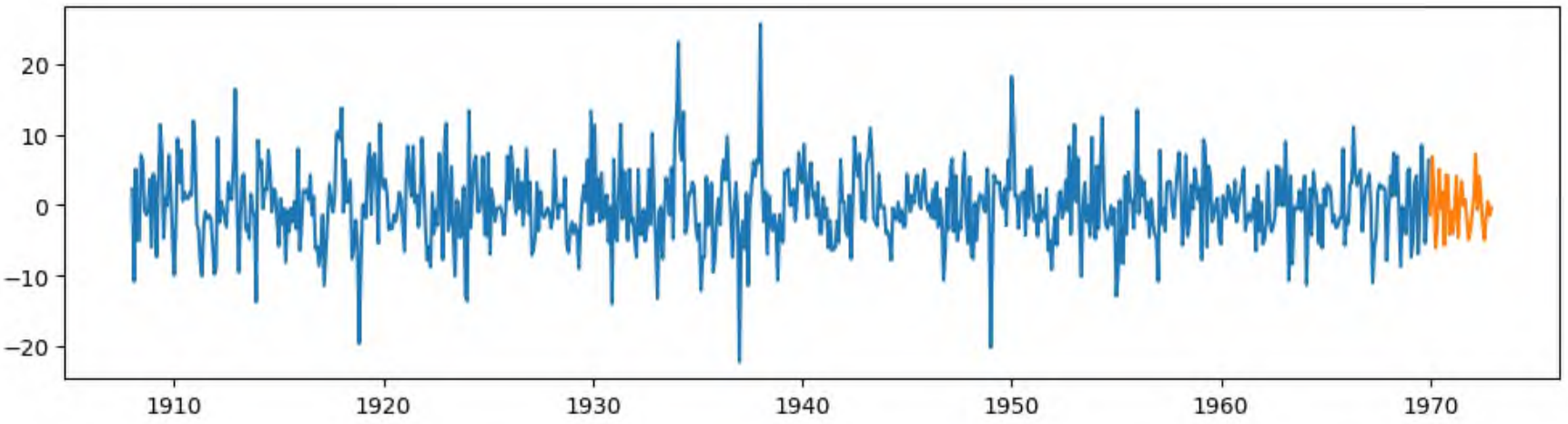
```
# plot ACF
_, ax = plt.subplots(1, 1, figsize=(5, 3))
plot_acf(monthly_temp_clean['Seasonally_Differenced'], lags=20, ax=ax);
```

  - Non-zero AC at lag 1 and 3
  - $q = 1$ or 2 or 3



- Model estimation
  - Having obtained p and q
  - Estimate $\theta_1, \ldots, \theta_p$ for AR and $\theta_1, \ldots, \theta_q$ for MA
    - Complex and non-linear problem
    - Use nonlinear least squares or maximum likelihood estimation

```
# Split data into training and test set
train = monthly_temp_clean['Seasonally_Differenced'][:-36]
test = monthly_temp_clean['Seasonally_Differenced'][-36:]

plt.figure(figsize=(12,3))
plt.plot(train)
plt.plot(test);
```



```
# Fit model
model = ARIMA(train, order=(3, 0, 3))  # ARIMA with d=0 is equivalent to ARMA
fit_model = model.fit()

print(fit_model.summary())
```

SARIMAX Results

Dep. Variable:	Seasonally_Differenced	No. Observations:	744
Model:	ARIMA(3, 0, 3)	Log Likelihood	-2248.119
Date:	Tue, 07 May 2024	AIC	4512.238
Time:	16:24:06	BIC	4549.134
Sample:	01-01-1908 - 12-01-1969	HQIC	4526.460
Covariance Type:	opg		

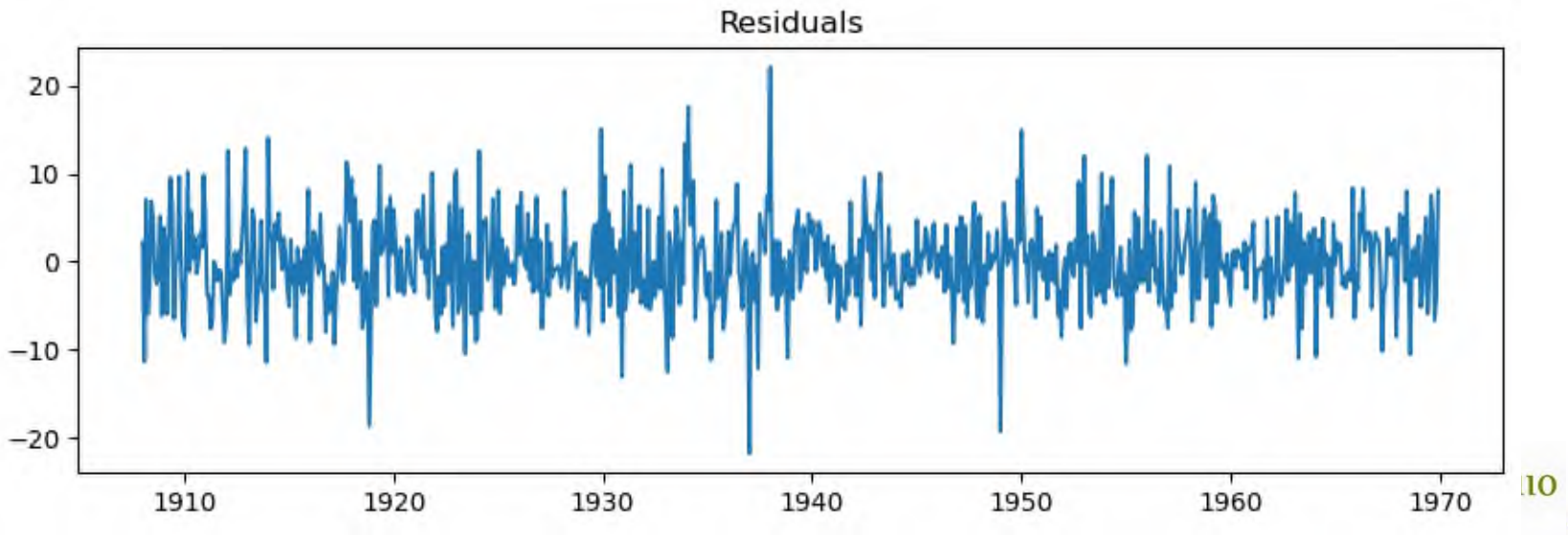
	coef	std err	z	P>|z|	[0.025	0.975]
const	0.0644	0.245	0.263	0.793	-0.416	0.544
ar.L1	-0.0891	0.137	-0.648	0.517	-0.358	0.180
ar.L2	-0.8305	0.023	-35.754	0.000	-0.876	-0.785
ar.L3	0.0521	0.120	0.435	0.664	-0.183	0.287
ma.L1	0.2842	0.136	2.092	0.036	0.018	0.551
ma.L2	0.9966	0.012	84.858	0.000	0.974	1.020
ma.L3	0.2394	0.136	1.759	0.079	-0.027	0.506
sigma2	24.3189	1.008	24.134	0.000	22.344	26.294

Ljung-Box (L1) (Q):	0.03	Jarque-Bera (JB):	56.84
Prob(Q):	0.86	Prob(JB):	0.00
Heteroskedasticity (H):	0.77	Skew:	0.12
Prob(H) (two-sided):	0.04	Kurtosis:	4.33

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

- ARMA Model Validation
  - Check the residuals, i.e., what the model was not able to fit
  - Residuals should approximate Gaussian distribution (aka white noise)
  - Otherwise, need to select a better model

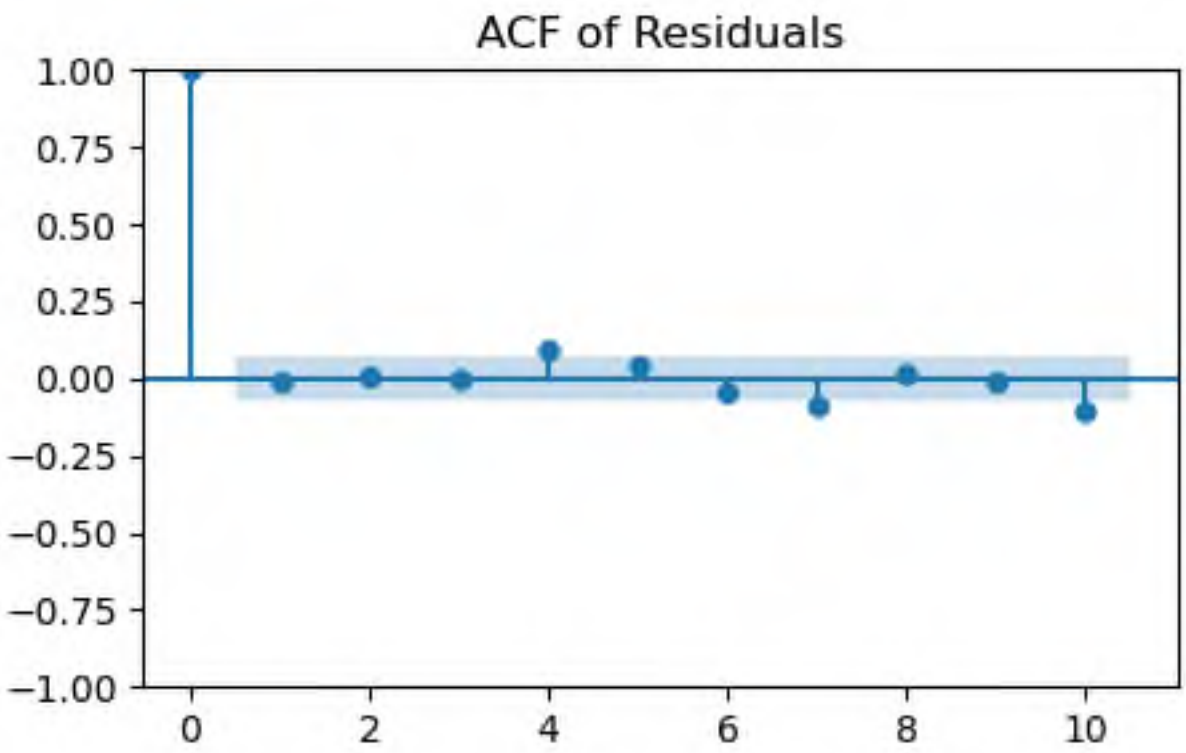
```
# compute residuals ACF
residuals = fit_model.resid

plt.figure(figsize=(10,3))
plt.plot(residuals)
plt.title("Residuals");
```



- ARMA Model Validation
  - Residuals approximate noise?
  - Visual inspection
    - ACF plot
    - Histogram
    - QQ plot
  - Statistical tests
    - Normality
    - Autocorrelation
    - Heteroskedasticity

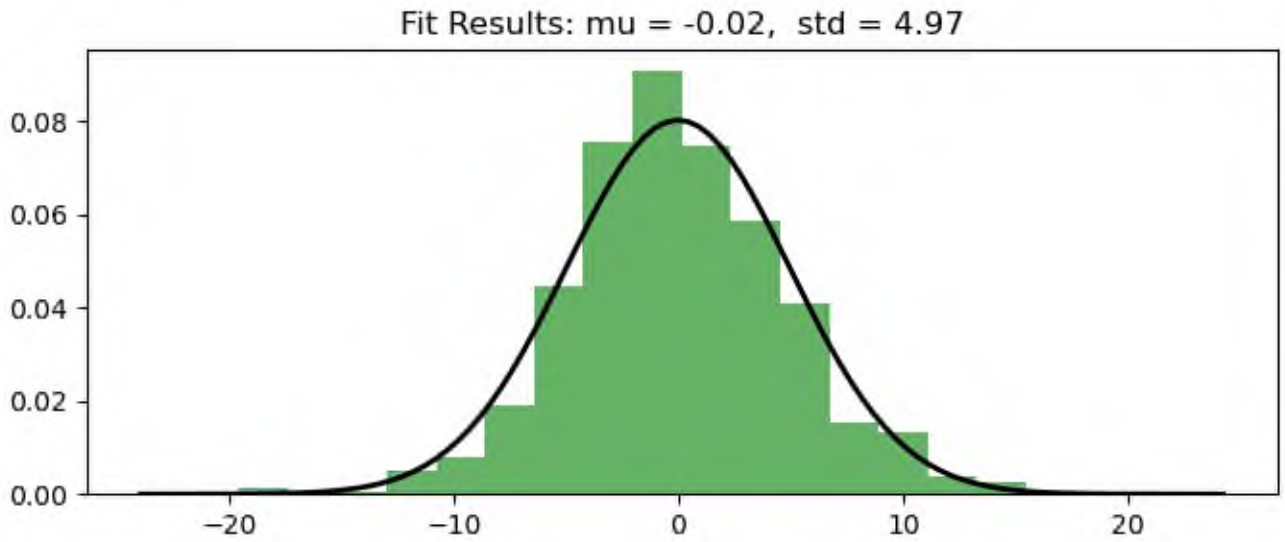
- ARMA Model Validation
  - ACF plot to check for autocorrelation in the residuals

```
# ACF of residuals
_, ax = plt.subplots(1, 1, figsize=(5, 3))
plot_acf(residuals, lags=10, ax=ax)
plt.title('ACF of Residuals')
plt.show()
```



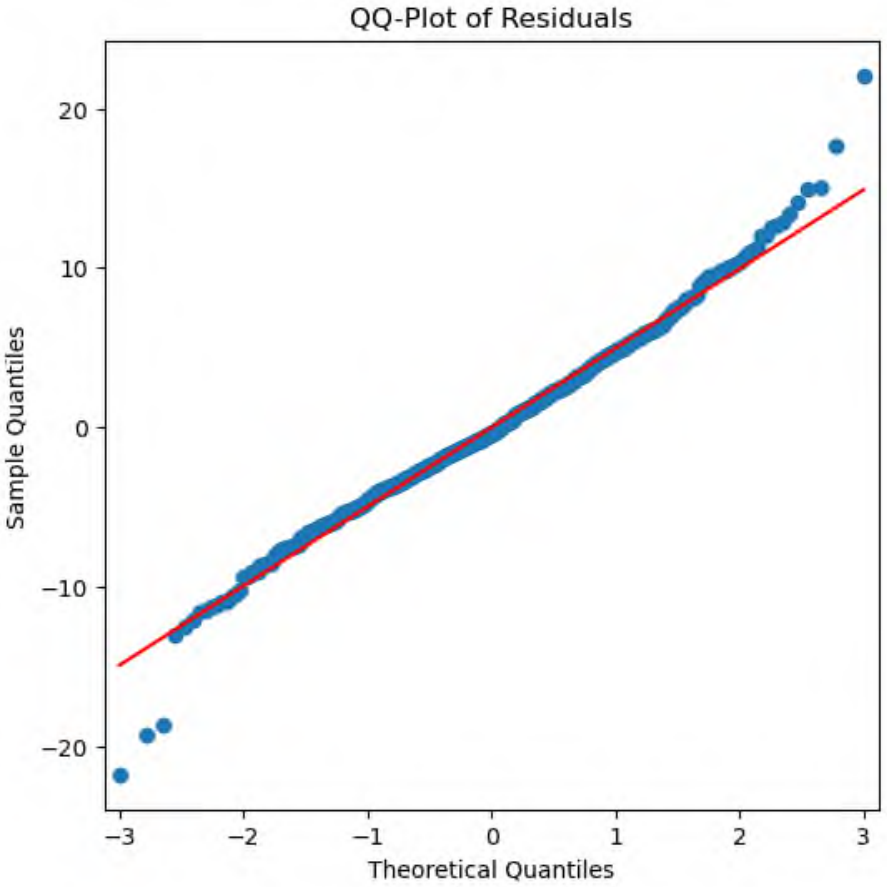
- ARMA Model Validation
  - Histogram of residuals

```
# Histogram
plt.figure(figsize=(8,3))
plt.hist(residuals, bins=20, density=True, alpha=0.6, color='g')
# Add the normal distribution curve
xmin, xmax = plt.xlim()
x = np.linspace(xmin, xmax, 100)
p = stats.norm.pdf(x, np.mean(residuals), np.std(residuals))
plt.plot(x, p, 'k', linewidth=2)
title = "Fit Results: mu = %.2f,  std = %.2f" % (np.mean(residuals),
np.std(residuals))
plt.title(title)
plt.show()
```



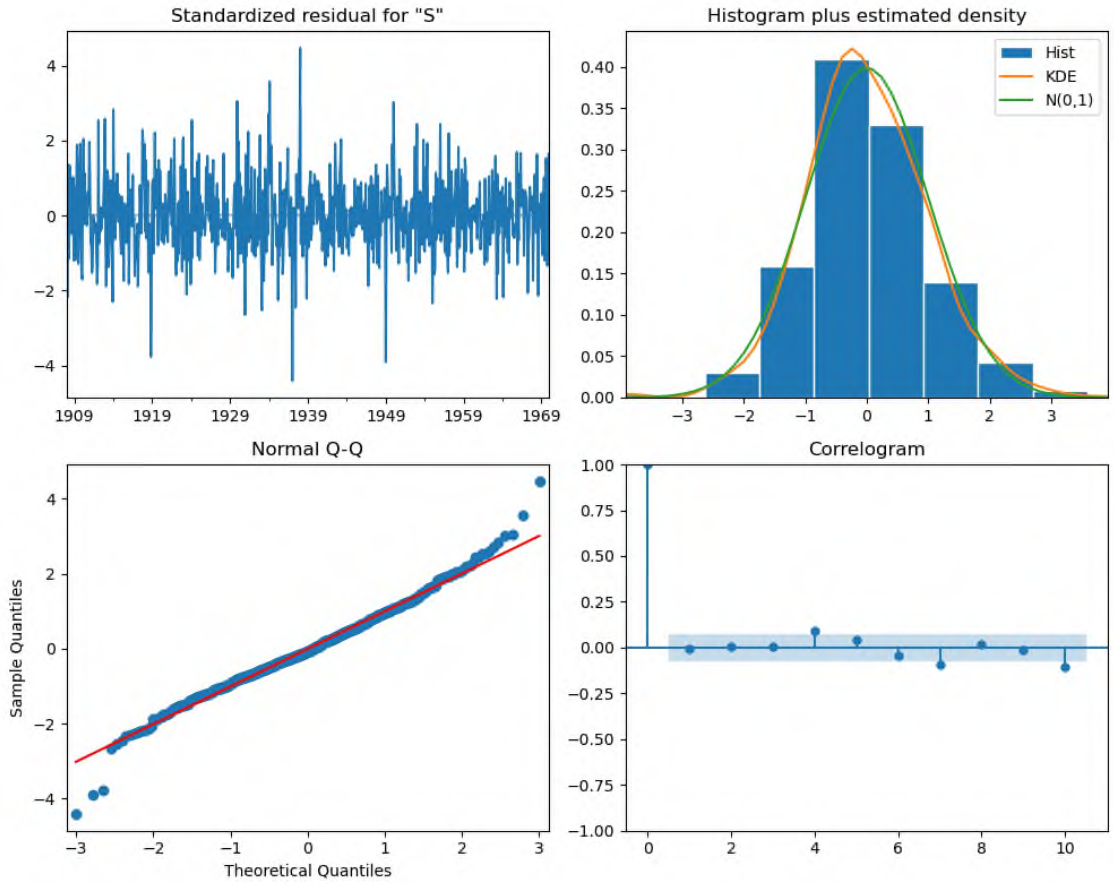
- ARMA Model Validation
  - QQ-plot of residuals

```
# QQ-Plot
_, ax = plt.subplots(1, 1, figsize=(6, 6))
qqplot(residuals, line='s', ax=ax)
plt.title('QQ-Plot of Residuals')
plt.show()
```



- ARMA Model Validation
  - Summarized using plot_diagnostics()

```
# plot diagnostics
fit_model.plot_diagnostics(figsize=(10, 8))
plt.tight_layout();
```



- ARMA Model Validation
  - Beyond plots, statistical test for Normality
  - Jarque-Bera and Shapiro-Wilk tests
  - H_0: the residuals are normally distributed

```
# Statistical Jarque-Bera test
norm_val, norm_p, skew, kurtosis = fit_model.test_normality('jarquebera')[0]
print('Normality (Jarque-Bera) p-value:{:.3f}'.format(norm_p))
```

  - Normality (Jarque-Bera) p-value:0.000

```
# Statistical Shapiro-Wilk test
shapiro_test = stats.shapiro(residuals)
print(f'Normality (Shapiro-Wilk) p-value: {shapiro_test.pvalue:.3f}')
```

  - Normality (Shapiro-Wilk) p-value: 0.000

  - Hence reject H_0 since p-values are small
  - Residuals are not normally distributed (this is what these tests say)

- ARMA Model Validation
- Autocorrelation: Ljung-Box test
  - H_0: the residuals are independently distributed (no autocorrelation)

```
# Statistical Autocorrelation test
statistic, pval = fit_model.test_serial_correlation(method='ljungbox', lags=10)[0]
print(f'Ljung-Box p-value: {pval.mean():.3f}')
```

  - Not always obvious how many lags to use
  - Ljung-Box p-value: 0.343

- Autocorrelation: Durbin Watson test

```
# Statistical Autocorrelation test in the residuals
durbin_watson =
ss.stats.stattools.durbin_watson(fit_model.filter_results.standardized_forecasts_e
rror[0, fit_model.loglikelihood_burn:])
print('Durbin-Watson: d={:.2f}'.format(durbin_watson))
```

  - Durbin-Watson: d=2.01 (values between 1-3, 2 ideal no serial correlation)

- ARMA Model Validation
- Heteroskedasticity test
  - Tests for change in variance between residuals
  - H_0: no heteroskedasticity, alternative H_A
  - H_A: increasing, H_0: variance not decreasing throughout the series
  - H_A: decreasing, H_0: variance not increasing throughout the series
  - H_A: two-sided, H_0: variance not changing throughout the series

```
_, pval = fit_model.test_heteroskedasticity('breakvar',
alternative='increasing')[0]
print(f'H_a: Increasing - pvalue:{pval:.3f}')
```

  - H_a: Increasing - pvalue:0.980

```
_, pval = fit_model.test_heteroskedasticity('breakvar',
alternative='decreasing')[0]
print(f'H_a: Decreasing - pvalue:{pval:.3f}')
```

  - H_a: Decreasing - pvalue:0.020

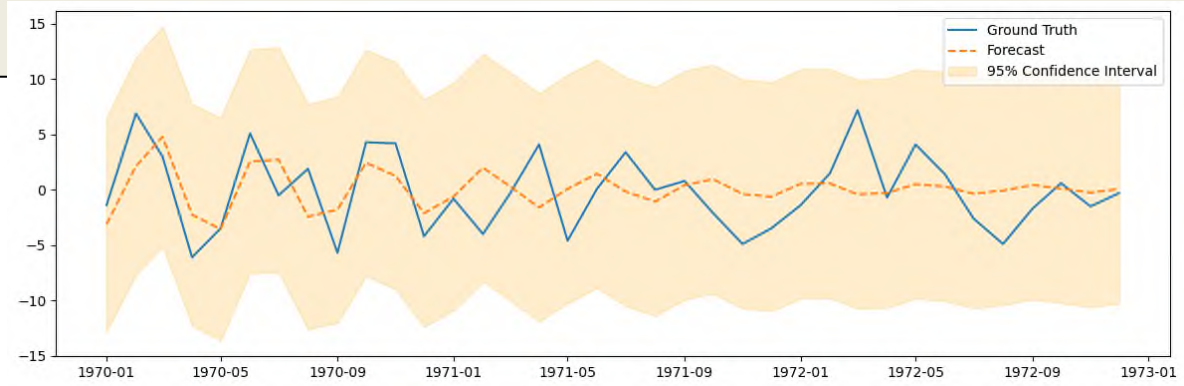
```
_, pval = fit_model.test_heteroskedasticity('breakvar', alternative='two-
sided')[0]
print(f'H_a: Two-sided - pvalue:{pval:.3f}')
```

  - H_a: Two-sided - pvalue:0.040

- ARMA Model Validation
  - Independence:
    - ✅ ACF plot
    - ✅ Ljung-Box test
    - ✅ Durbin Watson test
  - Normality:
    - ✅ Histogram/Density plot
    - 🤔 QQ-plot
    - ❌ Jarque-Bera (reliable for large sample size).
    - ❌ Shapiro-Wilk (reliable for large sample size).
  - Heteroskedasticity
    - ❌ Heteroskedasticity test
- Tests are a bit inconclusive
- No strong evidence that the model is either very good or very bad
- Probably worth investigate other models, e.g., ARMA(2,0,2) and repeat the tests

- Once the model is fit, we can use it to predict the test data
- Predictions come in a form of a distribution
    - ARMA performs a probabilistic forecasting
- The mean (mode) of this distribution correspond to the most likely value and correspond to our forecast
- The rest of the distribution can be used to compute confidence intervals

```
# ARMA forecasting
pred_summary = fit_model.get_prediction(test.index[0], test.index[-1]).summary_frame()
plt.figure(figsize=(12, 4))
plt.plot(test.index, test, label='Ground Truth')
plt.plot(test.index, pred_summary['mean'], label='Forecast', linestyle='--')
plt.fill_between(test.index, pred_summary['mean_ci_lower'], pred_summary['mean_ci_upper'],
                 color='orange', alpha=0.2, label='95% Confidence Interval')
plt.legend()
plt.tight_layout();
```



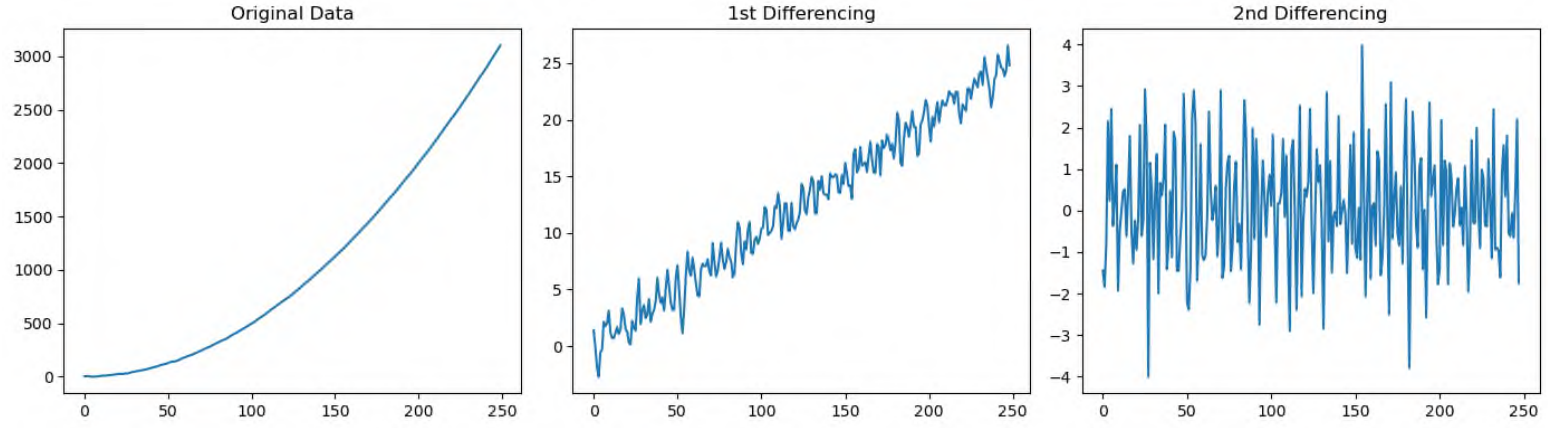
AUTO REGRESSIVE INTEGRATED MOVING AVERAGE

- ARIMA stands for Auto Regressive Integrated Moving Average
- ARIMA models have three components:
  - AR model
  - Integrated component (more on this shortly)
  - MA model
- ARIMA model is denoted ARIMA(p, d, q)
  - p is the order of the AR model
  - d is the number of times to difference the data
  - q is the order of the MA model
  - p, d, q are all nonegative integers
- Differencing nonstationary time series data one or more times can make it stationary but needs caution

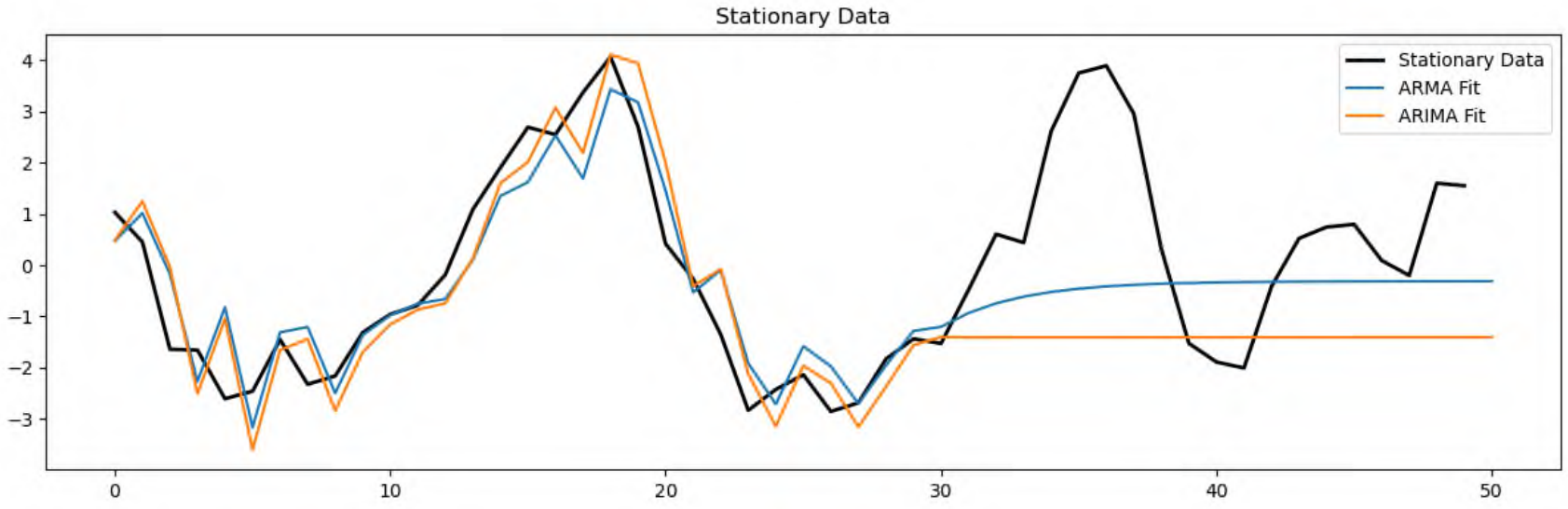
- Differencing through the Integrated (I) component of ARIMA
  - d number of times to perform lag 1 difference on data
  - $d = 0$: no differencing
  - $d = 1$: difference once
  - $d = 2$: difference twice
- While ARMA model is suitable for stationary time series where the mean and variance do not change over time
- The ARIMA model effectively models non-stationary time series by differencing the data
- In practice, ARIMA makes the time series stationary before applying the ARMA model

```
# Generate synthetic stationary data with an ARMA(1,1) process
n = 250
ar_coeff = np.array([1, -0.7]) # The first value refers to lag 0 and is always 1.
In addition, AR coeff are negated.
ma_coeff = np.array([1, 0.7])  # The first value refers to lag 0 and is always 1
arma_data = ss.tsa.arima_process.ArmaProcess(ar_coeff,
ma_coeff).generate_sample(nsample=n, burnin=1000)


# Generate a synthetic non-stationary data (needs to be differenced twice to be
stationary)
t = np.arange(n)
non_stationary_data = 0.05 * t**2 + arma_data  # Quadratic trend

fig, axes = plt.subplots(1, 3, figsize=(14, 4))
axes[0].plot(non_stationary_data)
axes[0].set_title('Original Data')
axes[1].plot(diff(non_stationary_data, k_diff=1))
axes[1].set_title('1st Differencing')
axes[2].plot(diff(non_stationary_data, k_diff=2))
axes[2].set_title('2nd Differencing')
plt.tight_layout();
```

```
# Generate synthetic stationary data with an ARMA(1,1) process
n = 250
ar_coeff = np.array([1, -0.7]) # The first value refers to lag 0 and is always 1.
In addition, AR coeff are negated.
ma_coeff = np.array([1, 0.7])  # The first value refers to lag 0 and is always 1
arma_data = ss.tsa.arima_process.ArmaProcess(ar_coeff,
ma_coeff).generate_sample(nsample=n, burnin=1000)


# Generate a synthetic non-stationary data (needs to be differenced twice to be
stationary)
t = np.arange(n)
non_stationary_data = 0.05 * t**2 + arma_data  # Quadratic trend

fig, axes = plt.subplots(1, 3, figsize=(14, 4))
axes[0].plot(non_stationary_data)
axes[0].set_title('Original Data')
axes[1].plot(diff(non_stationary_data, k_diff=1))
axes[1].set_title('1st Differencing')
axes[2].plot(diff(non_stationary_data, k_diff=2))
axes[2].set_title('2nd Differencing')
plt.tight_layout();
```

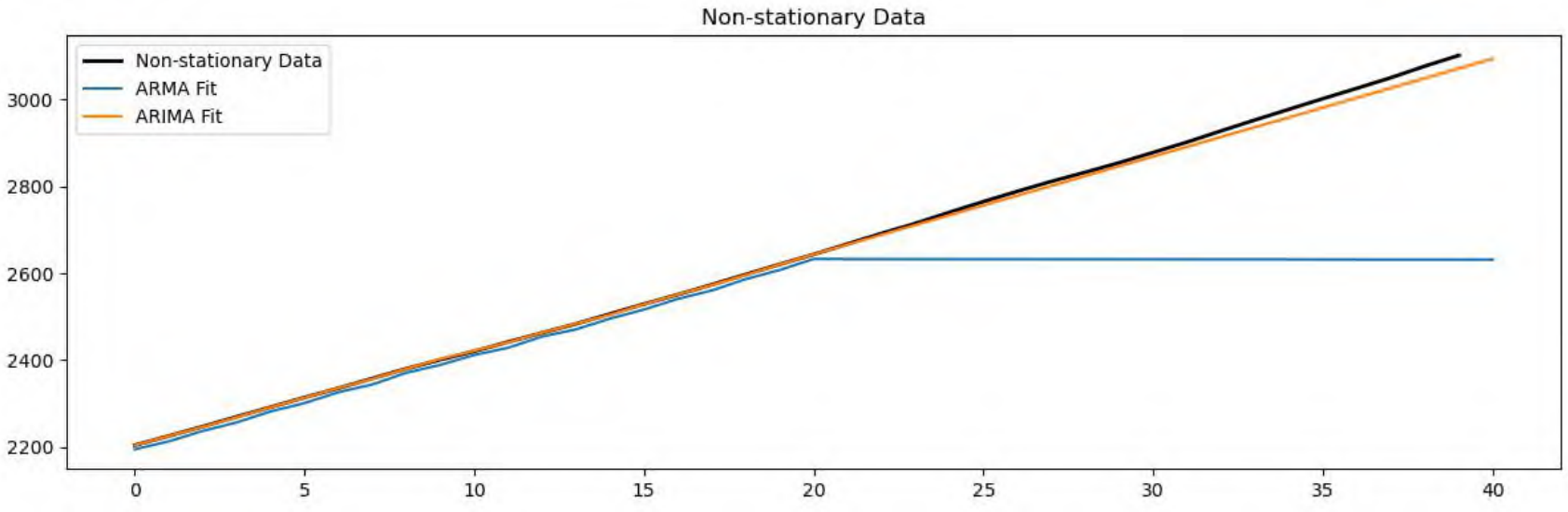


```
# Fit models to stationary data
arma_model = ARIMA(arma_data[:-20], order=(1, 0, 1)).fit()
arima_model = ARIMA(arma_data[:-20], order=(1, 1, 1)).fit()

plt.figure(figsize=(12, 4))
plt.plot(arma_data[-50:], 'k', label='Stationary Data', linewidth=2)
plt.plot(arma_model.predict(200,250), label='ARMA Fit')
plt.plot(arima_model.predict(200, 250), label='ARIMA Fit')
plt.legend()
plt.title('Stationary Data')
plt.tight_layout();

print(len(arma_model.predict(10)))
```



Πανεπιστήμιο Κύπρου

```
# Fit models to non-stationary data
arma_model = ARIMA(non_stationary_data[:-20], order=(1, 0, 1)).fit()
arima_model = ARIMA(non_stationary_data[:-20], order=(1, 2, 1)).fit()

plt.figure(figsize=(12, 4))
plt.plot(non_stationary_data[-40:], 'k', label='Non-stationary Data', linewidth=2)
plt.plot(arma_model.predict(210,250), label='ARMA Fit')
plt.plot(arima_model.predict(210,250), label='ARIMA Fit')
plt.legend()
plt.title('Non-stationary Data')
plt.tight_layout();
```

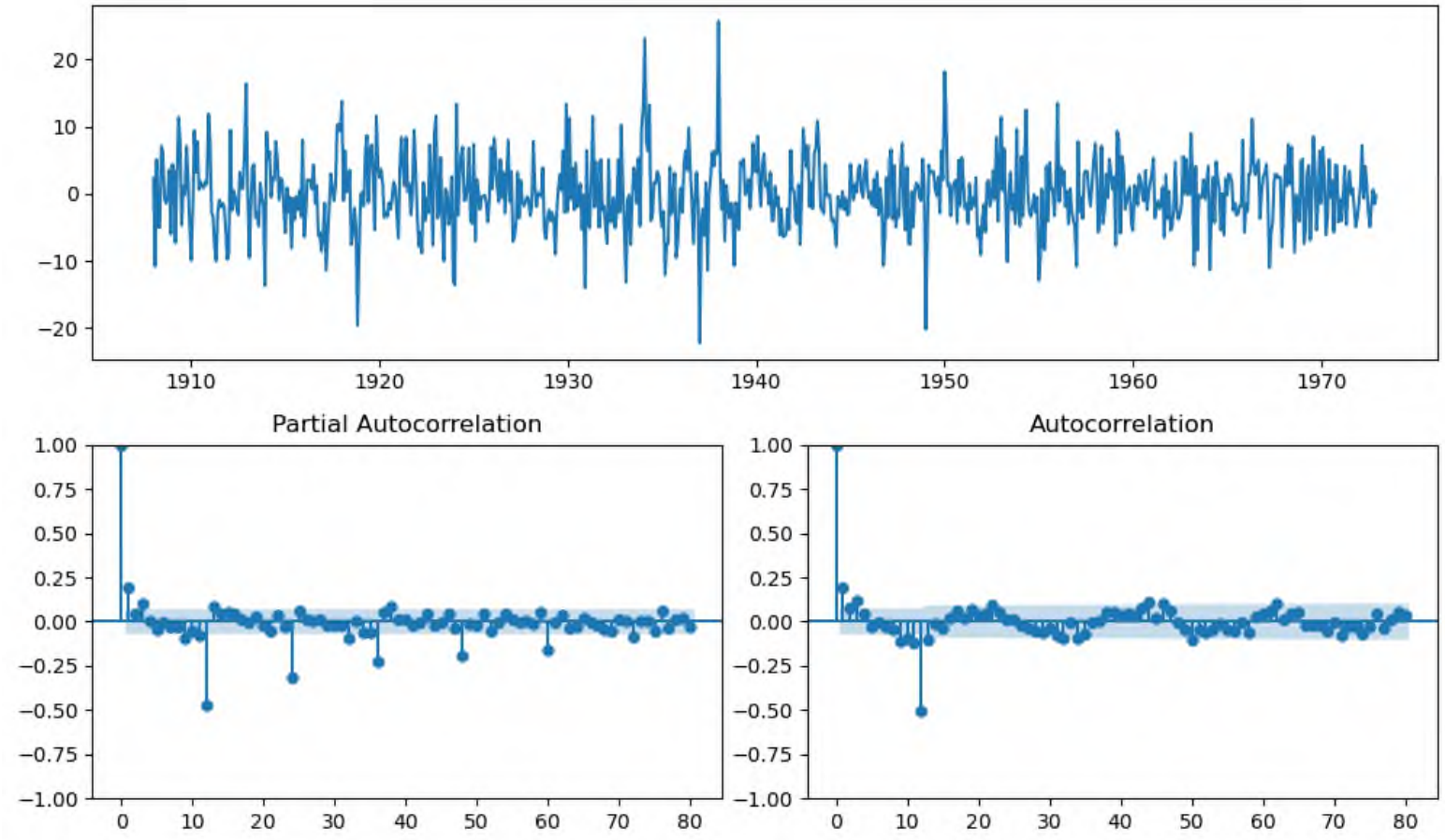


SEASONAL ARIMA (SARIMA)

- To apply ARMA and ARIMA, we must remove the seasonal component
  - After computing the predictions we had to put the seasonal component back
- It would be convenient to directly work on data with seasonality
- SARIMA is an extension of ARIMA that includes seasonal terms
- SARIMA model specified as SARIMA(p, d, q) x$(\mathrm{P}, \mathrm{D}, \mathrm{Q}, s)$:
  - Regular ARIMA components (p, d, q)
  - Seasonal components $(\mathrm{P}, \mathrm{D}, \mathrm{Q}, s)$:
    - P: Seasonal autoregressive order
    - D: Seasonal differencing order
    - Q: Seasonal moving average order
    - s: number of timesteps for a single seasonal period

- How to select $(\mathrm{P}, \mathrm{D}, \mathrm{Q}, s)$:
  - s:
    - Seasonality is easy to compute
  - P and Q:
    - Spikes at s^{th} lags (and multiples $(s \times n)^{th}$ of ACF/PACF plots
    - Select lags with largest spikes as candidates
  - D:
    - Number of seasonal differencing for stationary to be achieved
    - Determined by trial and error or examining the seasonal difference data
- Rule of thumb
  - Before selecting P and Q ensure that the series is seasonally stationary by applying seasonal differencing if needed (D)
  - Look the ACF plot to identify the seasonal moving average order Q
    - Look for significant autocorrelations at seasonal lags (multiples of s)
    - If the ACF plot shows a slow decay in the seasonal lags, it suggests a need for a seasonal MA component (Q)
  - Look at the PACF plot to identify the seasonal autoregressive order P
    - Look for significant spikes at multiples of the seasonality s
    - A sharp cut-off in the PACF at a seasonal lag suggests the number of AR terms (P) needed

```
diff_ts = monthly_temp['temp'].diff(periods=12).dropna()
fig = plt.figure(figsize=(10, 6))
ax1 = plt.subplot2grid((2, 2), (0, 0), colspan=2)
ax1.plot(diff_ts)
ax2 = plt.subplot2grid((2, 2), (1, 0))
plot_pacf(diff_ts, lags=80, ax=ax2)
ax3 = plt.subplot2grid((2, 2), (1, 1))
plot_acf(diff_ts, lags=80, ax=ax3)
plt.tight_layout();
```



```
# fit SARIMA monthly based on helper plots
sar = ss.tsa.statespace.sarimax.SARIMAX(monthly_temp[:750].temp,
                                        order=(2,1,2),
                                        seasonal_order=(0,1,1,12),
                                        trend='c').fit(disp=False)
sar.summary()
```

SARIMAX Results

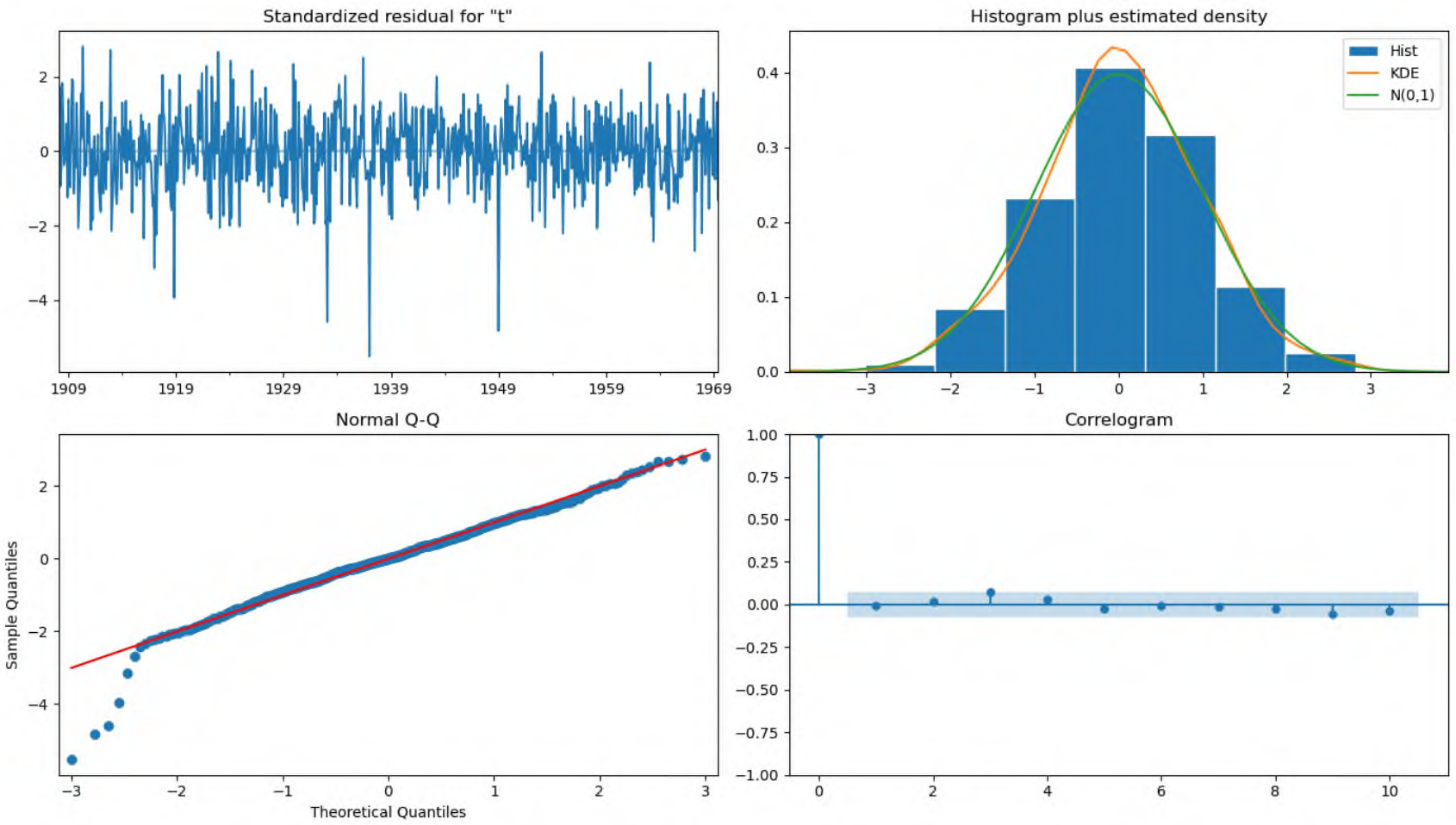
Dep. Variable:	temp	No. Observations:	750
Model:	SARIMAX(2, 1, 2)x(0, 1, [1], 12)	Log Likelihood	-2026.063
Date:	Tue, 07 May 2024	AIC	4066.127
Time:	16:24:10	BIC	4098.345
Sample:	01-01-1907	HQIC	4078.551
	- 06-01-1969		
Covariance Type: opg			

	coef	std err	z	P>|z|	[0.025	0.975]
intercept	-6.67e-05	0.000	-0.374	0.709	-0.000	0.000
ar.L1	-0.7658	0.170	-4.496	0.000	-1.100	-0.432
ar.L2	0.1790	0.042	4.262	0.000	0.097	0.261
ma.L1	-0.0543	0.182	-0.299	0.765	-0.411	0.302
ma.L2	-0.9433	0.173	-5.447	0.000	-1.283	-0.604
ma.S.L12	-0.9816	0.039	-24.909	0.000	-1.059	-0.904
sigma2	13.3628	0.955	13.993	0.000	11.491	15.234

Ljung-Box (L1) (Q):	0.03	Jarque-Bera (JB):	202.05
Prob(Q):	0.86	Prob(JB):	0.00
Heteroskedasticity (H):	0.77	Skew:	-0.55
Prob(H) (two-sided):	0.04	Kurtosis:	5.32

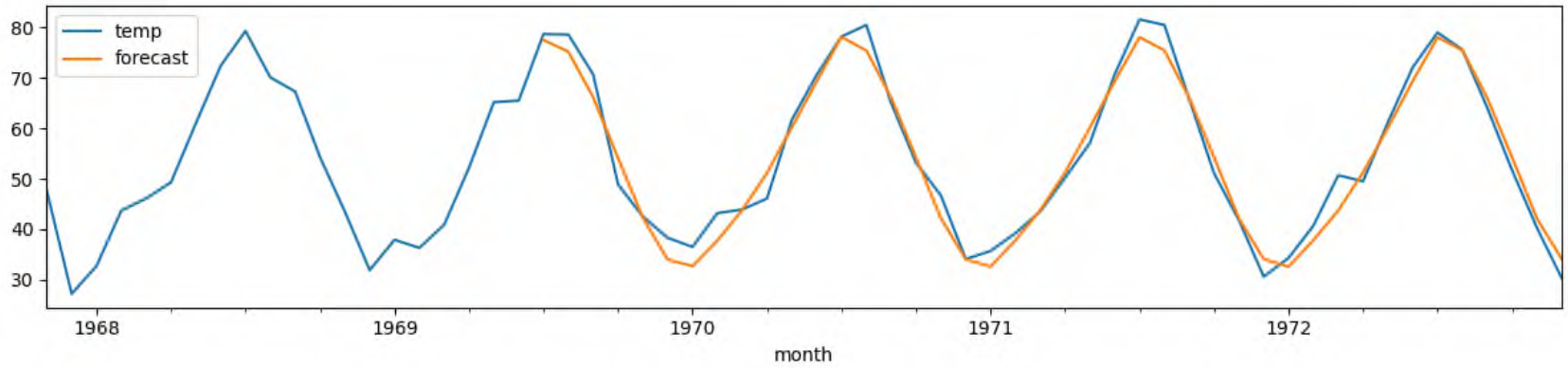
Warnings:
[1] Covariance matrix calculated using the outer product of gradients (complex-step).

```
sar.plot_diagnostics(figsize=(14, 8))
plt.tight_layout();
```



```
monthly_temp['forecast'] = sar.predict(start = 750, end= 792, dynamic=False)
monthly_temp[730:][['temp', 'forecast']].plot(figsize=(12, 3))
plt.tight_layout();
print(f"MSE: {mse(monthly_temp['temp'][-42:],monthly_temp['forecast'][-
42:]):.2f}")
```

- MSE: 9.21



AUTOARIMA

- Clearly identifying the optimal SARIMA model parameters is hard
- Requires careful analysis, trial and errors, and some experience
- Cheatsheet to help summarizes some rules of thumb

ACF Shape	Indicated Model
Exponential, decaying to zero	AR model. Use the PACF to identify the order of the AR model.
Alternating positive and negative, decaying to zero	AR model. Use the PACF to identify the order.
One or more spikes, rest are essentially zero	MA model, order identified by where plot becomes zero.
Decay, starting after a few lags	Mixed AR and MA (ARMA) model.
All zero or close to zero	Data are essentially random.
High values at fixed intervals	Include seasonal AR term.
No decay to zero	Series is not stationary.

- Alternatively use automated procedures
- AutoARIMA
  - Specify the maximum range of values to try
  - Goes on to find the best configuration among the alternatives

```
# Split the data into train and test sets
train, test = monthly_temp[:750].temp, monthly_temp[750:].temp

# Use auto_arima to find the best ARIMA model
model = pm.auto_arima(train,
                      start_p=0, start_q=0,
                      test='adf',       # Use adftest to find optimal 'd'
                      max_p=2, max_q=2, # Maximum p and q
                      m=12,             # Seasonality
                      start_P=0, start_Q=0,
                      max_P=2, max_Q=2, # Maximum P and Q
                      seasonal=True,    # Seasonal ARIMA
                      d=None,           # Let model determine 'd'
                      D=1,              # Seasonal difference D
                      trace=True,       # Print status on the fits
                      error_action='ignore',
                      suppress_warnings=True,
                      stepwise=True)    # Stepwise search to find the best model
```

```
# Summarize the model
print(model.summary())

# Frecast future values
monthly_temp['forecast'] = model.predict(n_periods=len(test))
monthly_temp[730:][['temp', 'forecast']].plot(figsize=(12, 3))
plt.tight_layout();
print(f"MSE: {mse(monthly_temp['temp'][-42:],monthly_temp['forecast'][-
42:]):.2f}")
```

SARIMAX Results

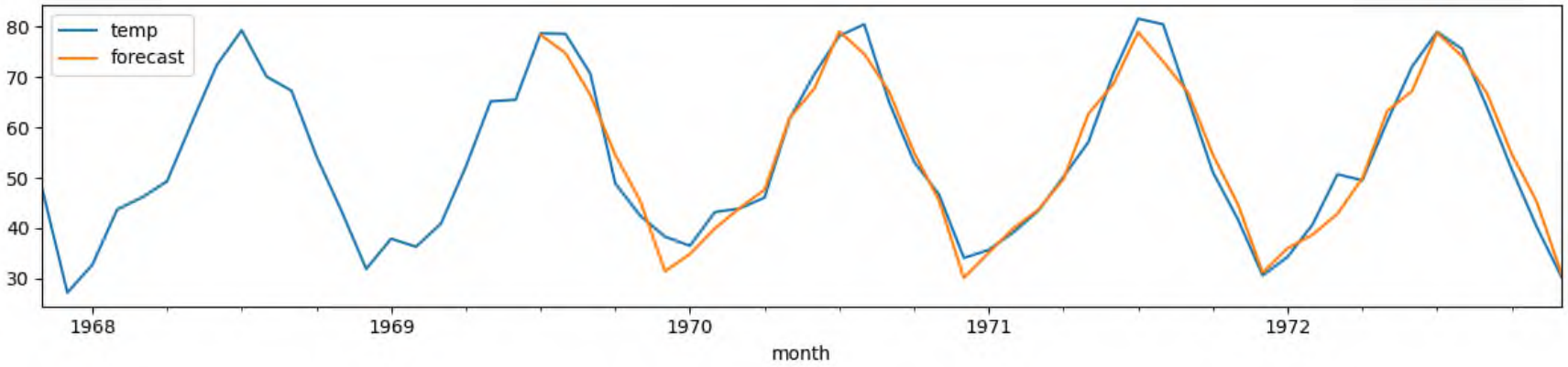
Dep. Variable:	y	No. Observations:	750
Model:	SARIMAX(1, 0, 1)x(2, 1, [], 12)	Log Likelihood	-2108.234
Date:	Tue, 07 May 2024	AIC	4226.467
Time:	16:24:39	BIC	4249.487
Sample:	01-01-1907 - 06-01-1969	HQIC	4235.344
Covariance Type:	opg		

	coef	std err	z	P>|z|	[0.025	0.975]
ar.L1	0.5814	0.144	4.027	0.000	0.298	0.864
ma.L1	-0.4101	0.162	-2.528	0.011	-0.728	-0.092
ar.S.L12	-0.6818	0.029	-23.405	0.000	-0.739	-0.625
ar.S.L24	-0.3479	0.030	-11.737	0.000	-0.406	-0.290
sigma2	17.5740	0.668	26.315	0.000	16.265	18.883

Ljung-Box (L1) (Q):	0.03	Jarque-Bera (JB):	145.56
Prob(Q):	0.87	Prob(JB):	0.00
Heteroskedasticity (H):	0.76	Skew:	-0.39
Prob(H) (two-sided):	0.04	Kurtosis:	5.03

Warnings: [1] Covariance matrix calculated using the outer product of gradients (complex-step). MSE: 11.12

```
# Summarize the model
print(model.summary())

# Frecast future values
monthly_temp['forecast'] = model.predict(n_periods=len(test))
monthly_temp[730:][['temp', 'forecast']].plot(figsize=(12, 3))
plt.tight_layout();
print(f"MSE: {mse(monthly_temp['temp'][-42:],monthly_temp['forecast'][-
42:]):.2f}")
```



- Can be computationally expensive, especially for large datasets and when exploring a wide range of models
- Lacks the qualitative insights a human might bring to the modeling process
    - Include understanding business cycles, external factors, or anomalies in the data
- Defaults in *auto_arima* may not be optimal for all time series data
    - range of values to explore should be adjusted properly each time
    - almost as tricky as doing manual model selection
- Requires a sufficiently long time series to accurately identify patterns and seasonality
- Selection of the best model is typically based on statistical criteria such as AIC or BIC
    - Might not always align with practical performance metrics such as MSE

- Selecting the best ARIMA model involves iteration and refinement

- A common approach is to select a set of candidates for (p, d, q) x$(\mathrm{P}, \mathrm{D}, \mathrm{Q}, s)$ and fit a model for each combination

- For each resulting model:
  - Analyze the residuals using the visual techniques and the statistical tests discussed before
  - Evaluate the prediction performance
  - Evaluate its complexity

- Use MSE and/or MAPE metrics to evaluate prediction performance
- Mean Squared Error (MSE)
  - average of the squared differences between the observed values and the predictions

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \widehat{Y}_i)^2$$

  - where Y_i is the observed data and $\widehat{Y}_i$ is the predicted value
  - lower MSE is preferred, indicating better fit to the data

- Mean Absolute Percentage Error (MAPE)
  - average of the absolute percentage errors of predictions

$$MAPE = \frac{100\%}{n} \sum_{i=1}^{n} \frac{Y_i - \widehat{Y}_i}{Y_i}$$

  - where Y_i is the observed data and $\widehat{Y}_i$ is the predicted value
- MAPE expresses errors as a percentage, making it straightforward to understand the magnitude of forecasting errors
- Also applicable when inspecting the relative size of the errors rather than absolute size
- Also MAPE is useful when the magnitude of the data varies significantly

Use AIC or BIC to estimate the model's complexity

- Akaike Information Criterion (AIC)
  - Measures the relative quality of statistical models for a given set of data
  - trade-off between the goodness of fit of the model and the complexity of the model.
  - $AIC = 2k - 2\ln(\hat{L})$, where k is the number of parameters in the model, and $\hat{L}$ is the maximized value of the likelihood function for the model
  - The model with the lowest AIC value is preferred, as it fits the data well but is not overly complex

Use AIC or BIC to estimate the model's complexity

- Bayesian Information Criterion (BIC)
  - Another criterion for model selection, introducing a stronger penalty for models with more parameters
  - $\mathrm{B}IC = \ln(n)\, k - 2\ln(\hat{L})$, where n is the number of observations, k is the number of parameters, and $\hat{L}$ is the maximized likelihood

- A lower BIC value indicates a better model, preferring simpler models to complex ones, especially as the sample size increases.

- Restricting the search with Exploratory Data Analysis (EDA)
- Grid search can be very expensive if done exaustively, especially on limited hardware
- An Exploratory Data Analysis can help to significantly reduce the number of candidates to try out
- Selecting the candidates for differentiation
- Let's start by identifying all the candidates for seasonal and general differencing
- Already know that the main seasionality is for example $s = 12$.
- Should we apply first the general or the seasonal differencing?
    - If seasonal patterns are dominant and the goal is to remove seasonality before addressing any trend, start with seasonal differencing.
    - If the trend is the predominant feature, you might start with standard differencing

```
# create all combinations of differencing orders, applying seasonal differencing
first and then general differencing
def differencing(timeseries, s, D_max=2, d_max=2):

    # Seasonal differencing from 0 to D_max
    seas_differenced = []
    for i in range(D_max+1):
        timeseries.name = f"d0_D{i}_s{s}"
        seas_differenced.append(timeseries)
        timeseries = timeseries.diff(periods=s)
    seas_df = pd.DataFrame(seas_differenced).T

    # General differencing from 0 to d_max
    general_differenced = []
    for j, ts in enumerate(seas_differenced):
        for i in range(1,d_max+1):
            ts = ts.diff()
            ts.name = f"d{i}_D{j}_s{s}"
            general_differenced.append(ts)
    gen_df = pd.DataFrame(general_differenced).T

    # concatenate seasonal and general differencing dataframes
    return pd.concat([seas_df, gen_df], axis=1)

# create the differenced series
diff_series = differencing(monthly_temp['temp'], s=12, D_max=2, d_max=2)
diff_series
```

month	d0_D0_s12	d0_D1_s12	d0_D2_s12	d1_D0_s12	d2_D0_s12	d1_D1_s12	d2_D1_s12	d1_D2_s12	d2_D2_s12
1907-01-01	33.3	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
1907-02-01	46.0	NaN	NaN	12.7	NaN	NaN	NaN	NaN	NaN
1907-03-01	43.0	NaN	NaN	-3.0	-15.7	NaN	NaN	NaN	NaN
1907-04-01	55.0	NaN	NaN	12.0	15.0	NaN	NaN	NaN	NaN
1907-05-01	51.8	NaN	NaN	-3.2	-15.2	NaN	NaN	NaN	NaN
...	...	...	...	...	...	...	...	...	...
1972-08-01	75.6	-4.9	-4.9	-3.4	-10.3	-2.3	1.7	1.1	8.4
1972-09-01	64.1	-1.7	-2.5	-11.5	-8.1	3.2	5.5	2.4	1.3
1972-10-01	51.7	0.6	2.7	-12.4	-0.9	2.3	-0.9	5.2	2.8
1972-11-01	40.3	-1.5	3.4	-11.4	1.0	-2.1	-4.4	0.7	-4.5
1972-12-01	30.3	-0.3	3.2	-10.0	1.4	1.2	3.3	-0.2	-0.9

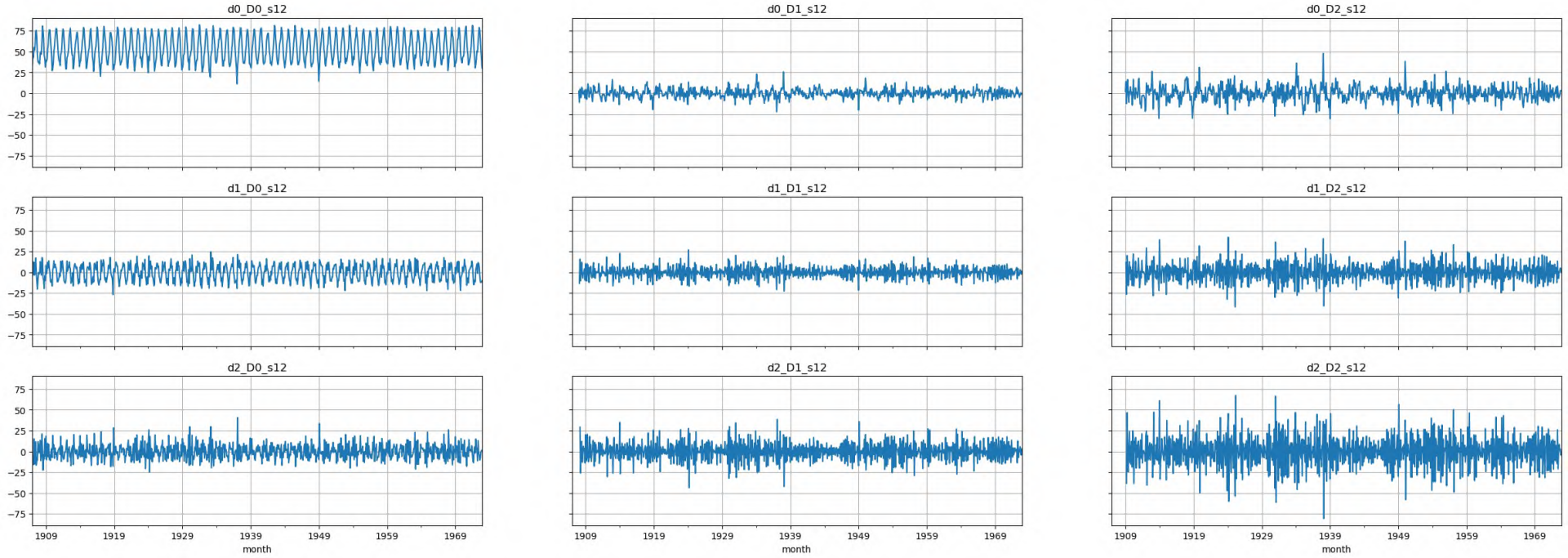
- Filter-out non-stationary candidates
  - Among all the differenced time series, keep only those that are stationary (according to ADF)

```
# create a summary of test results of all the series
def adf_summary(diff_series):
    summary = []

    for i in diff_series:
        # unpack the results
        a, b, c, d, e, f = adfuller(diff_series[i].dropna())
        g, h, i = e.values()
        results = [a, b, c, d, g, h, i]
        summary.append(results)

    columns = ["Test Statistic", "p-value", "#Lags Used", "No. of Obs. Used",
               "Critical Value (1%)", "Critical Value (5%)", "Critical Value (10%)"]
    index = diff_series.columns
    summary = pd.DataFrame(summary, index=index, columns=columns)

    return summary
```

```
# create the summary
summary = adf_summary(diff_series)

# filter away results that are not stationary
summary_passed = summary[summary["p-value"] < 0.05]
summary_passed
```

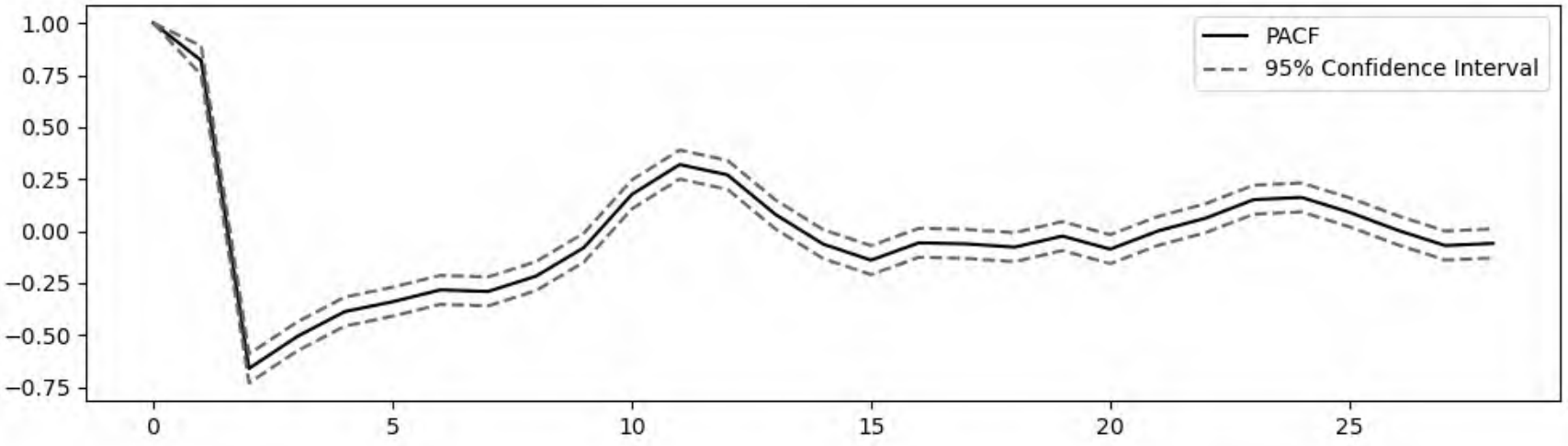
	Test Statistic	p-value	#Lags Used	No. of Obs. Used	Critical Value (1%)	Critical Value (5%)	Critical Value (10%)
d0_D0_s12	-6.481466	1.291867e-08	21	770	-3.438871	-2.865301	-2.568773
d0_D1_s12	-12.658082	1.323220e-23	12	767	-3.438905	-2.865316	-2.568781
d0_D2_s12	-10.416254	1.751310e-18	14	753	-3.439064	-2.865386	-2.568818
d1_D0_s12	-12.302613	7.391771e-23	21	769	-3.438882	-2.865306	-2.568775
d2_D0_s12	-15.935084	7.651998e-29	17	772	-3.438849	-2.865291	-2.568767
d1_D1_s12	-11.846173	7.390517e-22	20	758	-3.439006	-2.865361	-2.568804
d2_D1_s12	-18.352698	2.234953e-30	21	756	-3.439029	-2.865371	-2.568810
d1_D2_s12	-12.221559	1.104309e-22	20	746	-3.439146	-2.865422	-2.568837
d2_D2_s12	-15.080476	8.468854e-28	20	745	-3.439158	-2.865427	-2.568840

```
# output indices as a list
index_list = pd.Index.tolist(summary_passed.index)

# use the list as a condition to keep stationary time-series
passed_series = diff_series[index_list].sort_index(axis=1)

# Plot the final set of time series
fig, axes = plt.subplots(3, 3, figsize=(30, 10), sharex=True, sharey=True)
for i, ax in enumerate(axes.flatten()):
    passed_series.iloc[:,i].plot(ax=ax)
    ax.set_title(passed_series.columns[i])
    ax.grid()
```



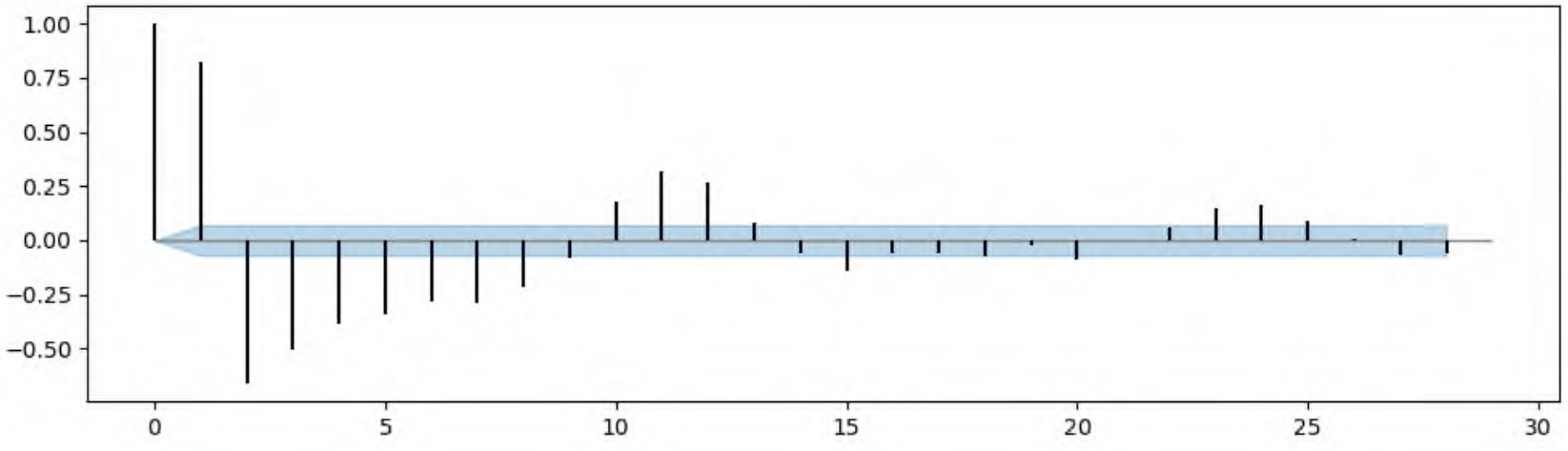
- Leverage the ACF and PACF functions for the orders of the AR and MA component, both in the general and the seasonal part, $p, q, \mathrm{P}, \mathrm{Q}$

```
PACF, PACF_ci = pacf(passed_series.iloc[:,0].dropna(), alpha=0.05)

# Plot PACF
plt.figure(figsize=(10,3))
plt.plot(PACF, color='k', label='PACF')
plt.plot(PACF_ci, color='tab:blue', linestyle='--', label=['95% Confidence
Interval', ''])
plt.legend()
plt.tight_layout();
```



- Leverage the ACF and PACF functions for the orders of the AR and MA component, both in the general and the seasonal part, $p, q, \mathrm{P}, \mathrm{Q}$

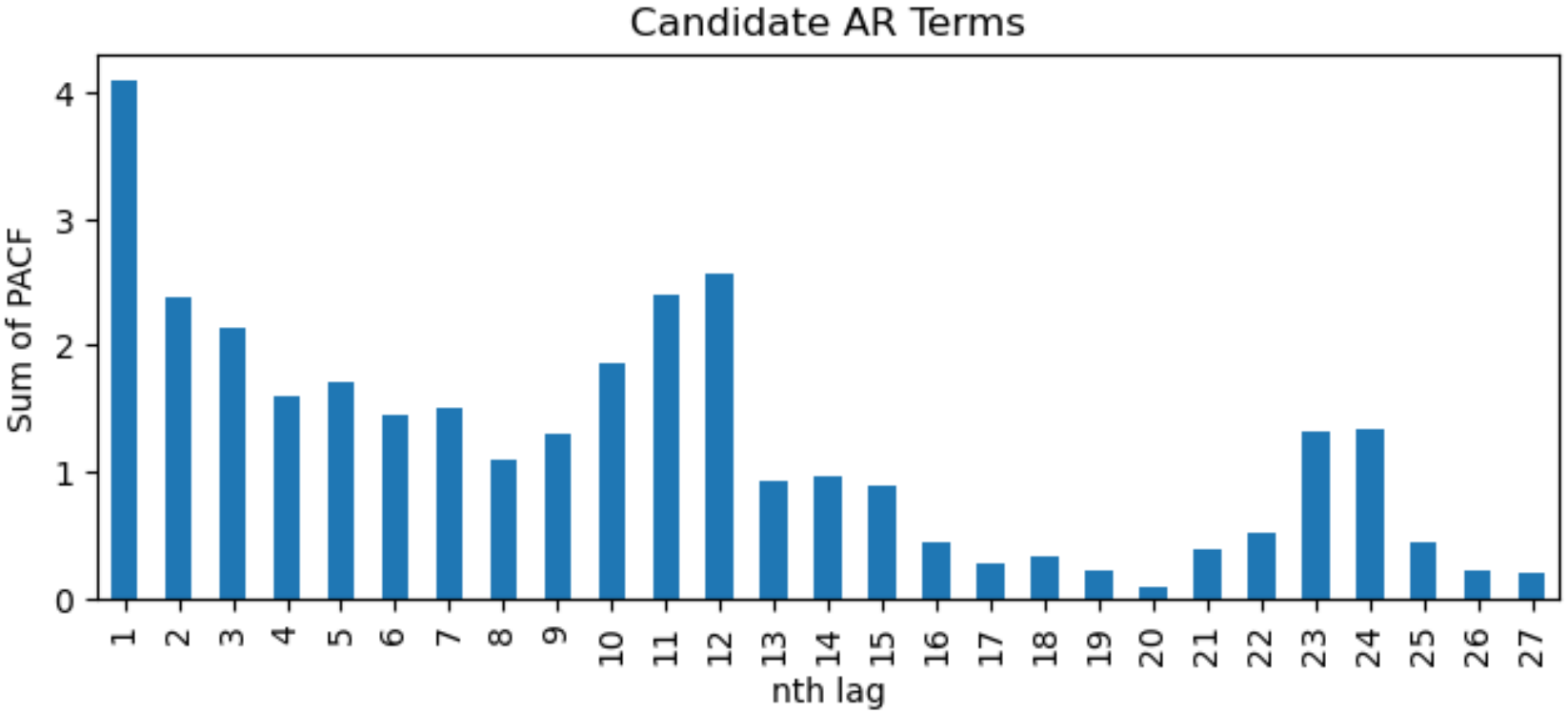
```
# subtract the confidence interval from the PACF to center the CI in zero
plt.figure(figsize=(10,3))
plt.fill_between(range(29), PACF_ci[:,0] - PACF, PACF_ci[:,1] - PACF,
color='tab:blue', alpha=0.3)
plt.hlines(y=0.0, xmin=0, xmax=29, linewidth=1, color='gray')

# Display the PACF as bars
plt.vlines(range(29), [0], PACF[:29], color='black')
plt.tight_layout();
```



Πανεπιστήμιο Κύπρου

```
df_sp_p = pd.DataFrame() # create an empty dataframe to store values of
significant spikes in PACF plots
for i in passed_series:
    # unpack the results into PACF and their CI
    PACF, PACF_ci = pacf(passed_series[i].dropna(), alpha=0.05, method='ywm')

    # subtract the upper and lower limits of CI by PACF to centre CI at zero
    PACF_ci_ll = PACF_ci[:,0] - PACF
    PACF_ci_ul = PACF_ci[:,1] - PACF

    # find positions of significant spikes representing possible value of p & P
    sp1 = np.where(PACF < PACF_ci_ll)[0]
    sp2 = np.where(PACF > PACF_ci_ul)[0]

    # PACF values of the significant spikes
    sp1_value = abs(PACF[PACF < PACF_ci_ll])
    sp2_value = PACF[PACF > PACF_ci_ul]

    # store values to dataframe
    sp1_series = pd.Series(sp1_value, index=sp1)
    sp2_series = pd.Series(sp2_value, index=sp2)
    df_sp_p = pd.concat((df_sp_p, sp1_series, sp2_series), axis=1)

# Sort the dataframe by index
df_sp_p = df_sp_p.sort_index()

# visualize sums of values of significant spikes in PACF plots ordered by lag
df_sp_p.iloc[1:].T.sum().plot(kind='bar', title='Candidate AR Terms',
xlabel='nth lag', ylabel='Sum of PACF', figsize=(8,3));
```

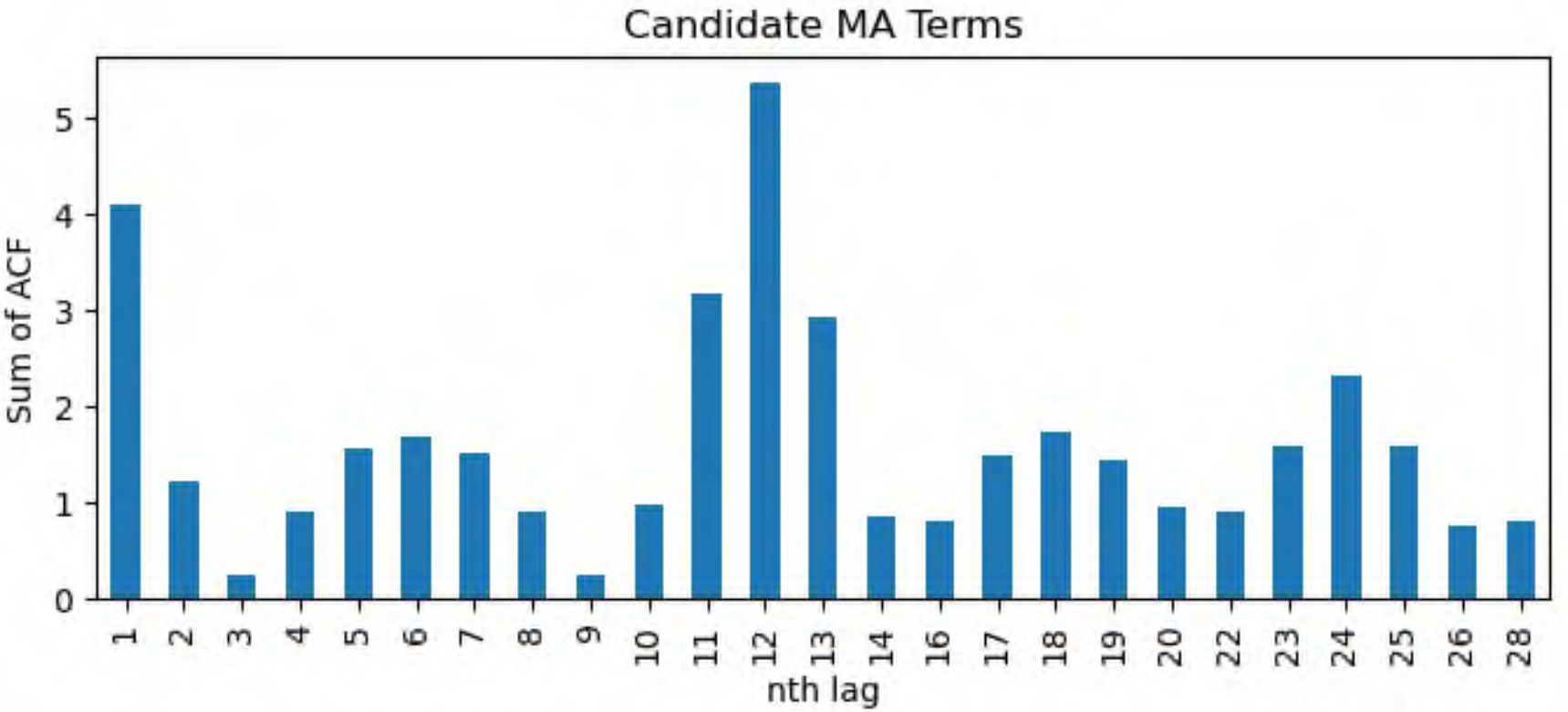
SELECT CANDIDATES FOR SARIMA COEFFICIENTS



Πανεπιστήμιο Κύπρου

```
df_sp_q = pd.DataFrame()
for i in passed_series:
    # unpack the results into ACF and their CI
    ACF, ACF_ci = acf(passed_series[i].dropna(), alpha=0.05)

    # subtract the upper and lower limits of CI by ACF to centre CI at zero
    ACF_ci_ll = ACF_ci[:,0] - ACF
    ACF_ci_ul = ACF_ci[:,1] - ACF

    # find positions of significant spikes representing possible value of q & Q
    sp1 = np.where(ACF < ACF_ci_ll)[0]
    sp2 = np.where(ACF > ACF_ci_ul)[0]

    # ACF values of the significant spikes
    sp1_value = abs(ACF[ACF < ACF_ci_ll])
    sp2_value = ACF[ACF > ACF_ci_ul]

    # store values to dataframe
    sp1_series = pd.Series(sp1_value, index=sp1)
    sp2_series = pd.Series(sp2_value, index=sp2)
    df_sp_q = pd.concat((df_sp_q, sp1_series, sp2_series), axis=1)

# Sort the dataframe by index
df_sp_q = df_sp_q.sort_index()

# visualize sums of values of significant spikes in ACF plots ordered by lags
df_sp_q.iloc[1:].T.sum().plot(kind='bar', title='Candidate MA Terms',
xlabel='nth lag', ylabel='Sum of ACF', figsize=(8,3));
```

Select candidates for SARIMA coefficients



Πανεπιστήμιο Κύπρου

```
# possible values
p = [1, 2, 3]
d = [0, 1]
q = [1, 2]
P = [0, 1]
D = [0, 1, 2]
Q = [0, 1]
s = [12]

# create all combinations of possible values
pdq = list(product(p, d, q))
PDQm = list(product(P, D, Q, s))

print(f"Number of total combinations: {len(pdq)*len(PDQm)}")
```

- Train the models
  - Defined a function that takes every model configuration and trains a model
  - For each model, we save the MSE, MAPE, AIC and BIC

```
warnings.simplefilter("ignore")
def SARIMA_grid(endog, order, seasonal_order):
    # create an empty list to store values
    model_info = []
    #fit the model
    for i in tqdm(order):
        for j in seasonal_order:
            try:
                model_fit = SARIMAX(endog=endog, order=i,
seasonal_order=j).fit(disp=False)
                predict = model_fit.predict()
                # calculate evaluation metrics: MAPE, RMSE, AIC & BIC
                MAPE = (abs((endog-predict)[1:])/(endog[1:])).mean()
                MSE = mse(endog[1:], predict[1:])
                AIC = model_fit.aic
                BIC = model_fit.bic
                # save order, seasonal order & evaluation metrics
                model_info.append([i, j, MAPE, MSE, AIC, BIC])
            except:
                continue
    # create a dataframe to store info of all models
    columns = ["order", "seasonal_order", "MAPE", "MSE", "AIC", "BIC"]
    model_info = pd.DataFrame(data=model_info, columns=columns)
    return model_info
```

```
# create train-test-split
train = monthly_temp['temp'].iloc[:int(len(monthly_temp)*0.9)]
test = monthly_temp['temp'].iloc[int(len(monthly_temp)*0.9):]
start = time.time()
# fit all combinations into the model
model_info = SARIMA_grid(endog=train, order=pdq, seasonal_order=PDQm)
end = time.time()
print(f'time required: {end - start :.2f}')
```

```
# 10 least MAPE models (best models according to performance)
least_MAPE = model_info.nsmallest(10, "MAPE")
least_MAPE
```

	order	seasonal_order	MAPE	MSE	AIC	BIC
140	(3, 1, 2)	(1, 1, 1, 12)	0.069031	16.775574	3860.325753	3896.722959
45	(1, 1, 2)	(1, 1, 1, 12)	0.069051	16.774401	3855.635083	3882.932987
127	(3, 1, 1)	(1, 0, 1, 12)	0.069052	17.111851	3960.626656	3992.593363
129	(3, 1, 1)	(1, 1, 1, 12)	0.069079	16.741013	3856.094709	3887.942264
81	(2, 1, 1)	(1, 1, 1, 12)	0.069124	16.802831	3857.202572	3884.500476
135	(3, 1, 2)	(0, 1, 1, 12)	0.069126	16.792647	3859.571180	3891.418735
39	(1, 1, 2)	(0, 1, 1, 12)	0.069130	16.781150	3854.502839	3877.251093
123	(3, 1, 1)	(0, 1, 1, 12)	0.069166	16.746288	3854.844955	3882.142859
93	(2, 1, 2)	(1, 1, 1, 12)	0.069172	16.805178	3860.077580	3891.925136
75	(2, 1, 1)	(0, 1, 1, 12)	0.069203	16.811213	3856.176615	3878.924868

```
# 10 least MSE models (best models according to performance)
least_MSE = model_info.nsmallest(10, "MSE")
least_MSE
```

	order	seasonal_order	MAPE	MSE	AIC	BIC
129	(3, 1, 1)	(1, 1, 1, 12)	0.069079	16.741013	3856.094709	3887.942264
123	(3, 1, 1)	(0, 1, 1, 12)	0.069166	16.746288	3854.844955	3882.142859
45	(1, 1, 2)	(1, 1, 1, 12)	0.069051	16.774401	3855.635083	3882.932987
140	(3, 1, 2)	(1, 1, 1, 12)	0.069031	16.775574	3860.325753	3896.722959
39	(1, 1, 2)	(0, 1, 1, 12)	0.069130	16.781150	3854.502839	3877.251093
135	(3, 1, 2)	(0, 1, 1, 12)	0.069126	16.792647	3859.571180	3891.418735
81	(2, 1, 1)	(1, 1, 1, 12)	0.069124	16.802831	3857.202572	3884.500476
93	(2, 1, 2)	(1, 1, 1, 12)	0.069172	16.805178	3860.077580	3891.925136
75	(2, 1, 1)	(0, 1, 1, 12)	0.069203	16.811213	3856.176615	3878.924868
87	(2, 1, 2)	(0, 1, 1, 12)	0.069289	16.825125	3859.565036	3886.862941

```
# 10 least AIC models (best models according to performance)
least_AIC = model_info.nsmallest(10, "AIC")
least_AIC
```

	order	seasonal_order	MAPE	MSE	AIC	BIC
43	(1, 1, 2)	(1, 0, 1, 12)	132.159673	5.501298e+07	132.106052	159.506087
3	(1, 0, 1)	(0, 1, 1, 12)	0.080283	6.149156e+01	3846.181890	3864.386212
9	(1, 0, 1)	(1, 1, 1, 12)	0.080153	6.147133e+01	3846.879649	3869.635051
15	(1, 0, 2)	(0, 1, 1, 12)	0.080286	6.148300e+01	3847.914835	3870.670236
51	(2, 0, 1)	(0, 1, 1, 12)	0.080283	6.148531e+01	3847.985625	3870.741026
99	(3, 0, 1)	(0, 1, 1, 12)	0.080295	6.144977e+01	3848.292676	3875.599158
21	(1, 0, 2)	(1, 1, 1, 12)	0.080146	6.146176e+01	3848.568559	3875.875041
57	(2, 0, 1)	(1, 1, 1, 12)	0.080145	6.146432e+01	3848.650344	3875.956826
105	(3, 0, 1)	(1, 1, 1, 12)	0.080096	6.143063e+01	3849.069448	3880.927010
117	(3, 0, 2)	(1, 1, 1, 12)	0.080081	6.140204e+01	3849.084348	3885.492991

```
# 10 least BIC models (best models according to performance)
least_BIC = model_info.nsmallest(10, "BIC")
least_BIC
```

	order	seasonal_order	MAPE	MSE	AIC	BIC
43	(1, 1, 2)	(1, 0, 1, 12)	132.159673	5.501298e+07	132.106052	159.506087
3	(1, 0, 1)	(0, 1, 1, 12)	0.080283	6.149156e+01	3846.181890	3864.386212
9	(1, 0, 1)	(1, 1, 1, 12)	0.080153	6.147133e+01	3846.879649	3869.635051
15	(1, 0, 2)	(0, 1, 1, 12)	0.080286	6.148300e+01	3847.914835	3870.670236
51	(2, 0, 1)	(0, 1, 1, 12)	0.080283	6.148531e+01	3847.985625	3870.741026
27	(1, 1, 1)	(0, 1, 1, 12)	0.069318	1.683683e+01	3855.820879	3874.019482
99	(3, 0, 1)	(0, 1, 1, 12)	0.080295	6.144977e+01	3848.292676	3875.599158
21	(1, 0, 2)	(1, 1, 1, 12)	0.080146	6.146176e+01	3848.568559	3875.875041
57	(2, 0, 1)	(1, 1, 1, 12)	0.080145	6.146432e+01	3848.650344	3875.956826
63	(2, 0, 2)	(0, 1, 1, 12)	0.080282	6.146983e+01	3849.361800	3876.668282

```
# check if there are overlaps between MAPE and MSE
set(least_MAPE.index) & set(least_MSE.index)
```

{39, 45, 75, 81, 93, 123, 129, 135, 140}

```
# the best model by each metric
L1 = model_info[model_info.MAPE == model_info.MAPE.min()]
L2 = model_info[model_info.MSE == model_info.MSE.min()]
L3 = model_info[model_info.AIC == model_info.AIC.min()]
L4 = model_info[model_info.BIC == model_info.BIC.min()]

best_models = pd.concat((L1, L2, L3, L4))
best_models
```

	order	seasonal_order	MAPE	MSE	AIC	BIC
140	(3, 1, 2)	(1, 1, 1, 12)	0.069031	1.677557e+01	3860.325753	3896.722959
129	(3, 1, 1)	(1, 1, 1, 12)	0.069079	1.674101e+01	3856.094709	3887.942264
43	(1, 1, 2)	(1, 0, 1, 12)	132.159673	5.501298e+07	132.106052	159.506087
43	(1, 1, 2)	(1, 0, 1, 12)	132.159673	5.501298e+07	132.106052	159.506087

- Take the best models, compute the predictions and evaluate their performance in terms of MAPE on the w.r.t. test data.

```
# Take the configurations of the best models
ord_list = [tuple(best_models.iloc[i,0]) for i in range(best_models.shape[0])]
s_ord_list = [tuple(best_models.iloc[i,1]) for i in
range(best_models.shape[0])]
preds, ci_low, ci_up, MAPE_test = [], [], [], []

# Fit the models and compute the forecasts
for i in range(4):
    model_fit = SARIMAX(endog=train, order=ord_list[i],
                        seasonal_order=s_ord_list[i]).fit(disp=False) # Fit the
model
    pred_summary = model_fit.get_prediction(test.index[0],
                                            test.index[-1]).summary_frame() #
Compute preds
    # Store results
    preds.append(pred_summary['mean'])
    ci_low.append(pred_summary['mean_ci_lower'][test.index])
    ci_up.append(pred_summary['mean_ci_upper'][test.index])
    MAPE_test.append((abs((test-pred_summary['mean'])/(test)).mean()))
```

```
# visualize the results of the fitted models
fig, axs = plt.subplots(nrows=2, ncols=2, figsize=(24,6),
                        sharex=True, sharey=True)
titles = [f'Least MAPE Model {ord_list[0]} x {s_ord_list[0]}',
          f'Least MSE Model {ord_list[1]} x {s_ord_list[1]}',
          f'Least AIC Model {ord_list[2]} x {s_ord_list[2]}',
          f'Least BIC Model {ord_list[3]} x {s_ord_list[3]}']
k = 0
for i in range(2):
    for j in range(2):
        axs[i,j].plot(monthly_temp['temp'], label='Ground Truth')
        axs[i,j].plot(preds[k], label='Prediction')
        axs[i,j].set_title(titles[k] + f' -- MAPE test: {MAPE_test[k]:.2%}')
        axs[i,j].legend()
        axs[i,j].axvline(test.index[0], color='black', alpha=0.5, linestyle='--')
        axs[i,j].fill_between(x=test.index, y1=ci_low[k], y2=ci_up[k],
color='orange', alpha=0.2)
        axs[i,j].set_ylim(bottom=20, top=90)
        axs[i,j].set_xlim(left=monthly_temp.index[-120],
right=monthly_temp.index[-1])
        k += 1
plt.tight_layout()
plt.show()
```