UNIVERSITY OF SCIENCE AND TECHNOLOGY OF HANOI

DEPARTMENT OF INFORMATION AND COMMUNICATION TECHNOLOGY

Research and Development

GROUP PROJECT REPORT

Time Series Analysis of FoodStuffCPI Data and Bangladesh_C Data

By

Student group number 02

Student 1 name - Student ID
Student 2 name - Student ID
Student 3 name - Student ID
Student 4 name - Student ID
Student 5 name - Student ID
Student 6 name - Student ID

Hanoi, November 2025

Introduction

This report presents a comprehensive time series analysis of two distinct datasets. Part 1 focuses on forecasting foodstuff-related Consumer Price Index (CPI) data using Autoregressive Integrated Moving Average (ARIMA) and Seasonal ARIMA (SARIMA) models. This section provides a detailed walkthrough of the model building process, from data exploration and stationarity testing to parameter selection and forecast evaluation.

Part 2 shifts focus to historical economic data for Bangladesh, employing statistical methods to detect anomalies and structural change points. Using Z-Score, Isolation Forest, and change-point detection algorithms, this section aims to identify significant events and shifts in key economic indicators, linking them to potential real-world economic phenomena. Together, these analyses demonstrate the application of a range of time series techniques for both forecasting and historical event detection.

Part 1. Time Series Analysis of Foodstuff CPI Data

Summary:

This part details the process of developing ARIMA and SARIMA models to forecast Foodstuff CPI. It covers initial data visualization, stationarity assessment, parameter identification using ACF/PACF plots, and model selection via AIC and RMSE criteria. The analysis concludes with a forecast and evaluation of model performance.

Introduction

The Consumer Price Index (CPI) is a critical economic indicator that measures the average change over time in the prices paid by urban consumers for a market basket of consumer goods and services. Analyzing and forecasting CPI, particularly for essential components like foodstuffs, is vital for policymakers, economists, and businesses for strategic planning and inflation management [1].

This analysis employs time series methodology to model and forecast two CPI-related series. The core of the analysis rests on the Box-Jenkins ARIMA methodology, a powerful framework for modeling non-stationary time series data. We will systematically explore the data's properties, ensure stationarity, identify appropriate model orders (p, d, q), estimate model parameters, and evaluate their forecasting accuracy before extending the analysis to include seasonal components with SARIMA models. The dataset under review is "Data01_02_FoodstuffCPI.csv", which includes the following key series:

- Column 1: Time
- Column 2: Year
- Column 3: Month
- Column 4: Series 1 - Foodstuff_toDecCP
- Column 5: Series 2 - CPI_toPM
- Column 6: Series 3 - CPI rate

Import necessary library

Code

```
import warnings
warnings.filterwarnings("ignore")

import os
import random
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from statsmodels.tsa.stattools import adfuller, kpss
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.statespace.sarimax import SARIMAX
from statsmodels.stats.diagnostic import acorr_ljungbox
from sklearn.metrics import mean_squared_error, mean_absolute_error
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import TimeSeriesSplit

random.seed(42)
np.random.seed(42)
```

Time series description

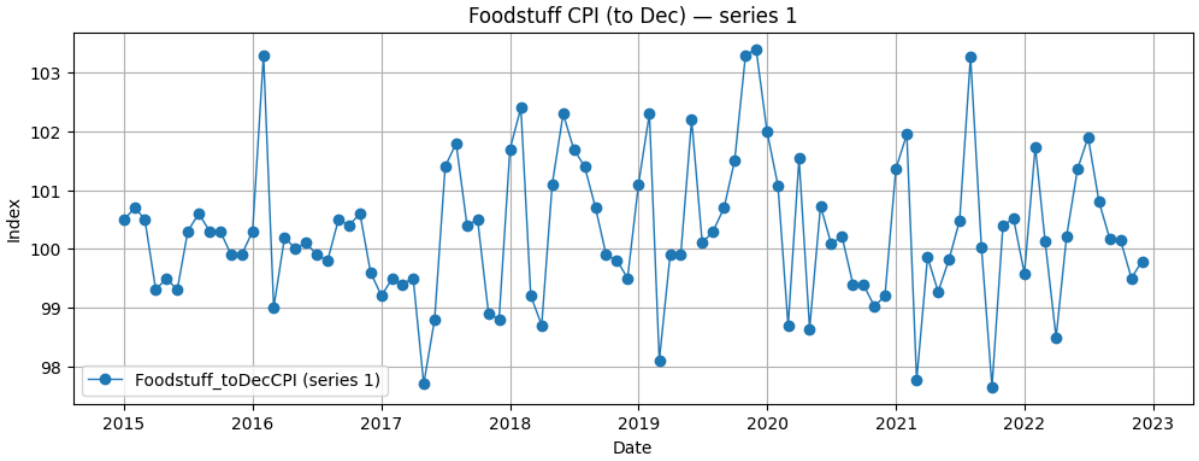
A. Time series plot

A1. Foodstuff_toDecCPI (series 1)

Code

```
ts1 = df['Foodstuff_toDecCPI'].astype(float)


plt.figure(figsize=(12,4))
plt.plot(ts1, marker='o', linewidth=1, label='Foodstuff_toDecCPI (series 1)')
plt.title('Foodstuff CPI (to Dec)  series 1')
plt.ylabel('Index')
plt.xlabel('Date')
plt.grid(True)
plt.legend()
plt.show()
```

Result



Comment

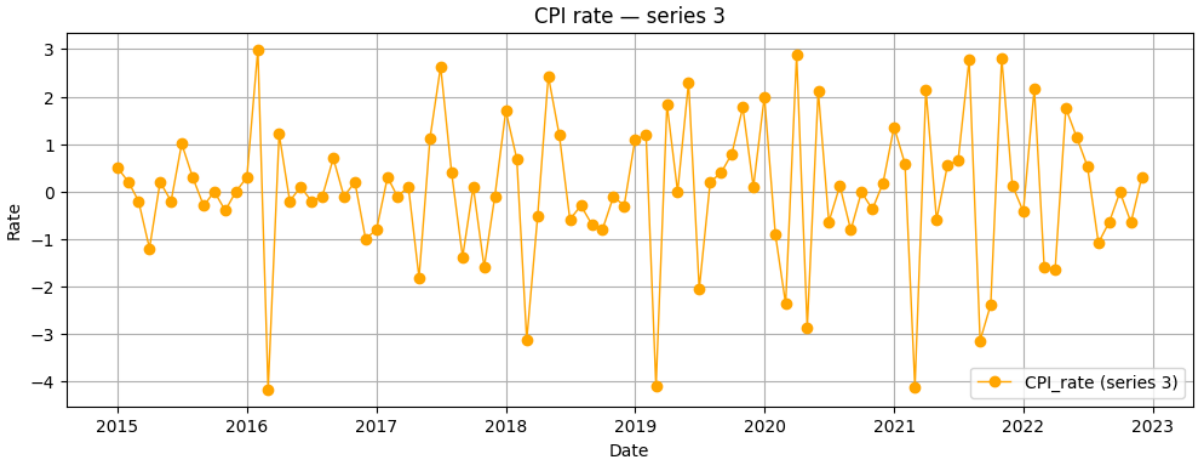
The plot for Series 1 (Foodstuff_toDecCPI) shows the index fluctuating around a mean value of approximately 100. There is no obvious long-term trend (upward or downward slope), suggesting that the series may be trend-stationary. The variance appears relatively constant over time. There are noticeable seasonal-like patterns, with peaks and troughs occurring at regular intervals, likely corresponding to annual cycles in food prices.

A2. CPI rate (series 3)

Code

```
ts3 = df['CPI_rate'].astype(float)

plt.figure(figsize=(12,4))
plt.plot(ts3, marker='o', linewidth=1, label='CPI_rate (series 3)',
       color='orange')
plt.title('CPI rate  series 3')
plt.ylabel('Rate')
plt.xlabel('Date')
plt.grid(True)
plt.legend()
plt.show()
```

Result



Comment

The plot for Series 3 (CPI rate) exhibits behavior characteristic of a stationary process. The series oscillates around a mean of zero, without any discernible trend. It displays significant volatility, with sharp, short-lived spikes in both positive and negative directions. This type of pattern, known as mean reversion, is common in financial and economic rate series [2]. The absence of a trend and constant variance strongly suggests stationarity, which will be formally tested later.

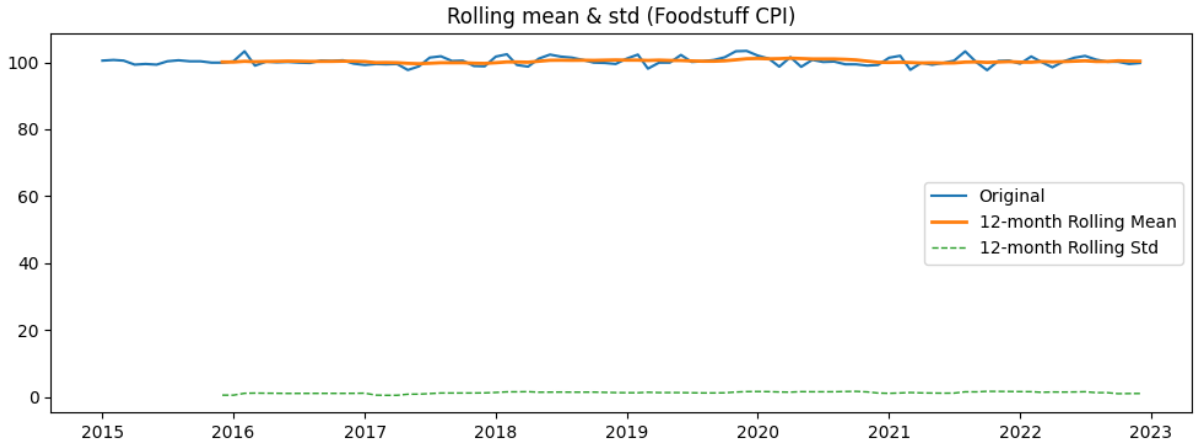
B. Time Series Trend Checking

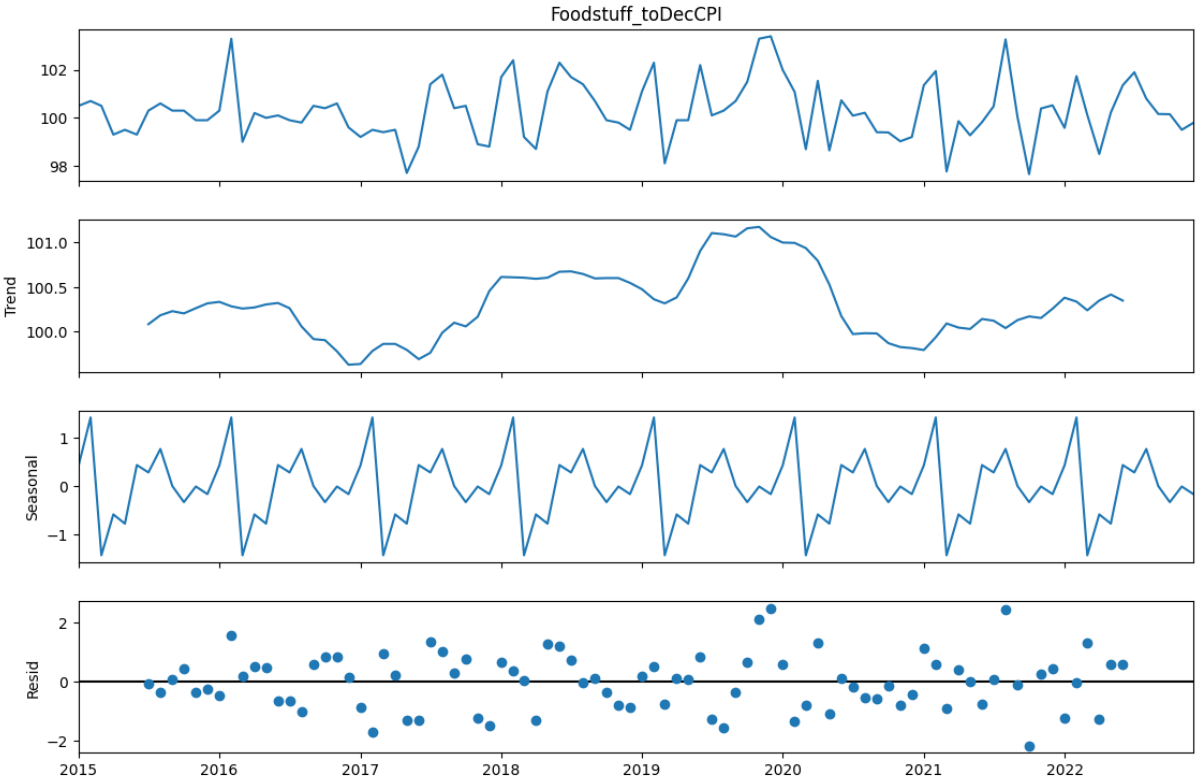
Code

```
# rolling mean (12-month) for monthly data
plt.figure(figsize=(12,4))
plt.plot(ts1, label='Original')
plt.plot(ts1.rolling(window=12).mean(), label='12-month Rolling Mean',
linewidth=2)
plt.plot(ts1.rolling(window=12).std(), label='12-month Rolling Std',
linewidth=1, linestyle='--')
plt.title('Rolling mean & std (Foodstuff CPI)')
plt.legend(); plt.show()

# seasonal decomposition
from statsmodels.tsa.seasonal import seasonal_decompose
try:
    decomposition = seasonal_decompose(ts1.dropna(), model='additive',
    period=12)
    fig = decomposition.plot()
    fig.set_size_inches(12,8)
    plt.show()
except Exception as e:
    print("Decompose error (maybe short series):", e)
```

B1. Foodstuff_toDecCPI (series 1)

Result





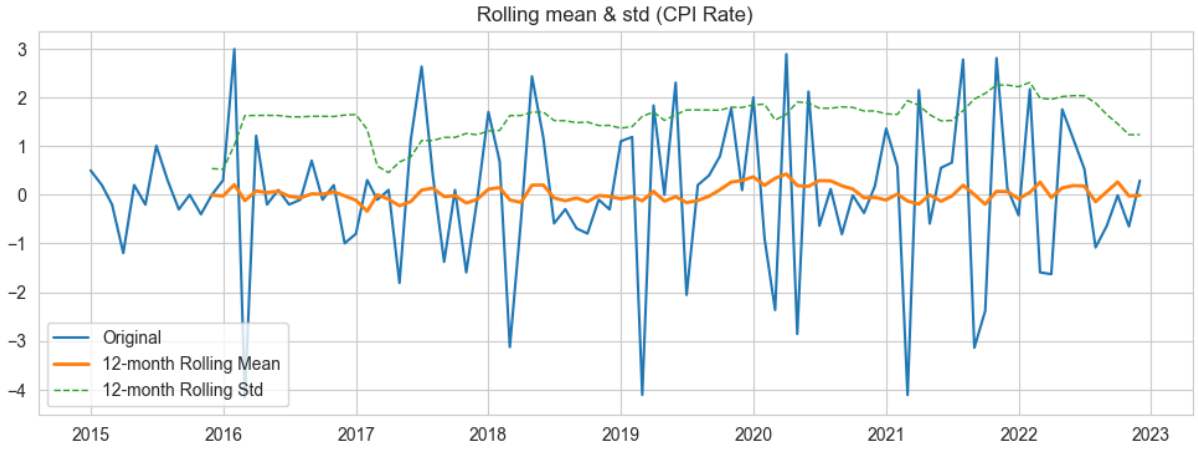
Comment

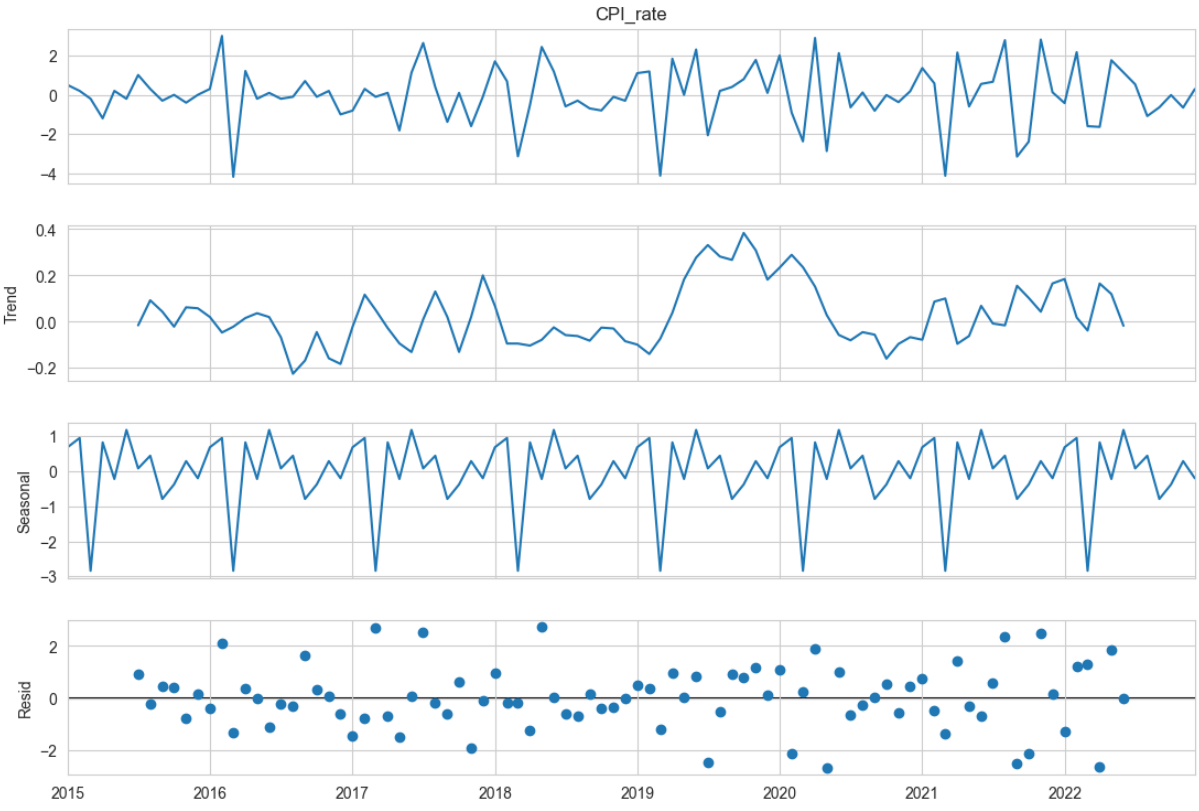
The rolling statistics for Series 1 confirm the initial visual assessment. The 12-month rolling mean is very stable and hovers just above 100, indicating a lack of a significant trend. The rolling standard deviation is also remarkably flat and close to zero, suggesting constant variance (homoscedasticity).

The seasonal decomposition plot effectively isolates the underlying components. The ‘Trend‘ component is almost flat, confirming trend stationarity. The ‘Seasonal‘ component shows a clear, repeating annual pattern, which is the dominant source of variation in the series. The ‘Resid‘ (residuals) appear to be random noise, centered around zero, which is ideal.

B2. CPI rate (series 3)

Result





Comment

For the CPI rate (Series 3), the rolling mean fluctuates around zero, consistent with a stationary series. The rolling standard deviation shows some variability but does not exhibit a persistent trend.

The decomposition plot for Series 3 reveals a weak, oscillating trend component that largely stays close to zero. A seasonal pattern is also evident, though it is less pronounced relative to the random noise in the residual component. The 'Resid' component contains a significant portion of the series' total variance, indicating that much of the month-to-month change in the CPI rate is random and not easily predictable by deterministic components alone.

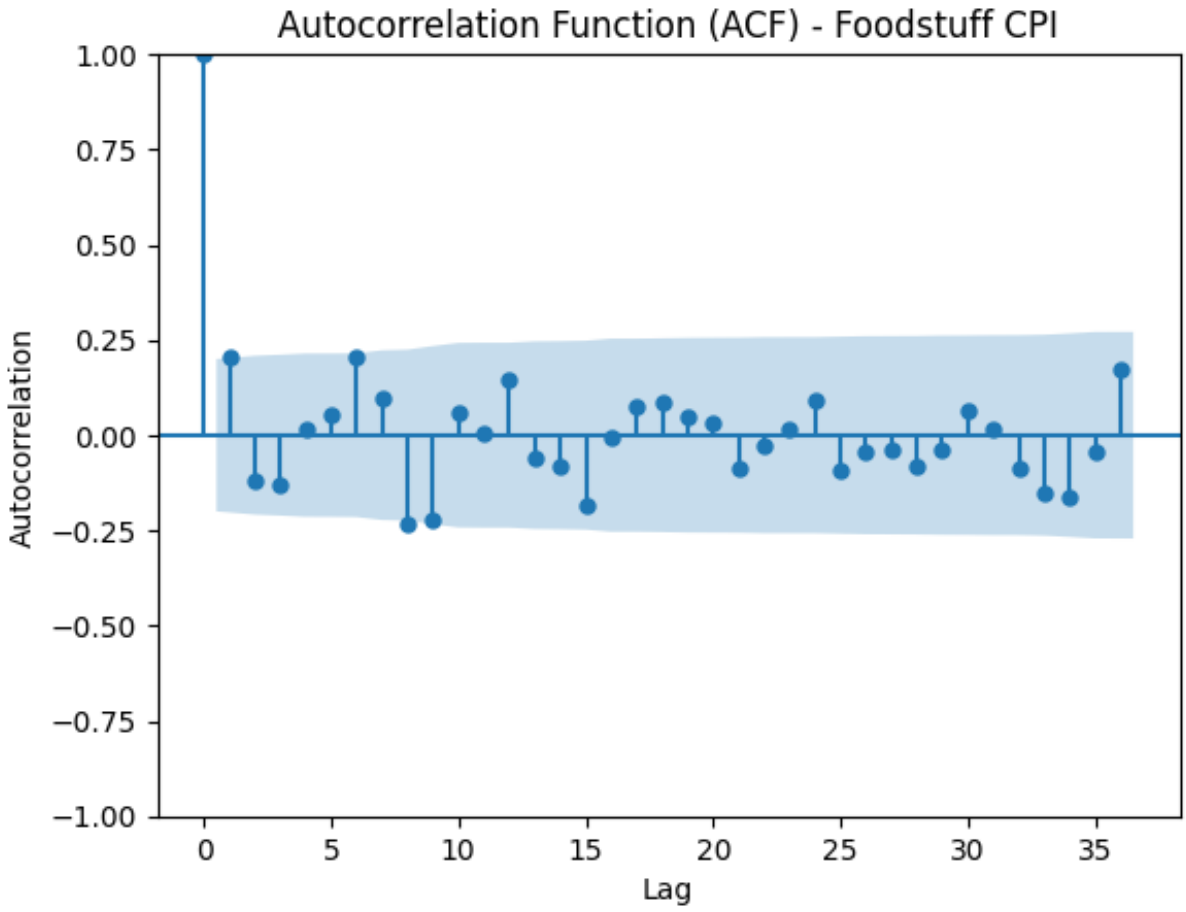
C. Autocorrelation Function

C1. Foodstuff_toDecCPI (series 1)

Code

```
plt.figure(figsize=(10,4))
plot_acf(ts1.dropna(), lags=36)
plt.title('Autocorrelation Function (ACF) - Foodstuff CPI')
plt.xlabel('Lag')
plt.ylabel('Autocorrelation')
plt.show()
```

Result



Comment

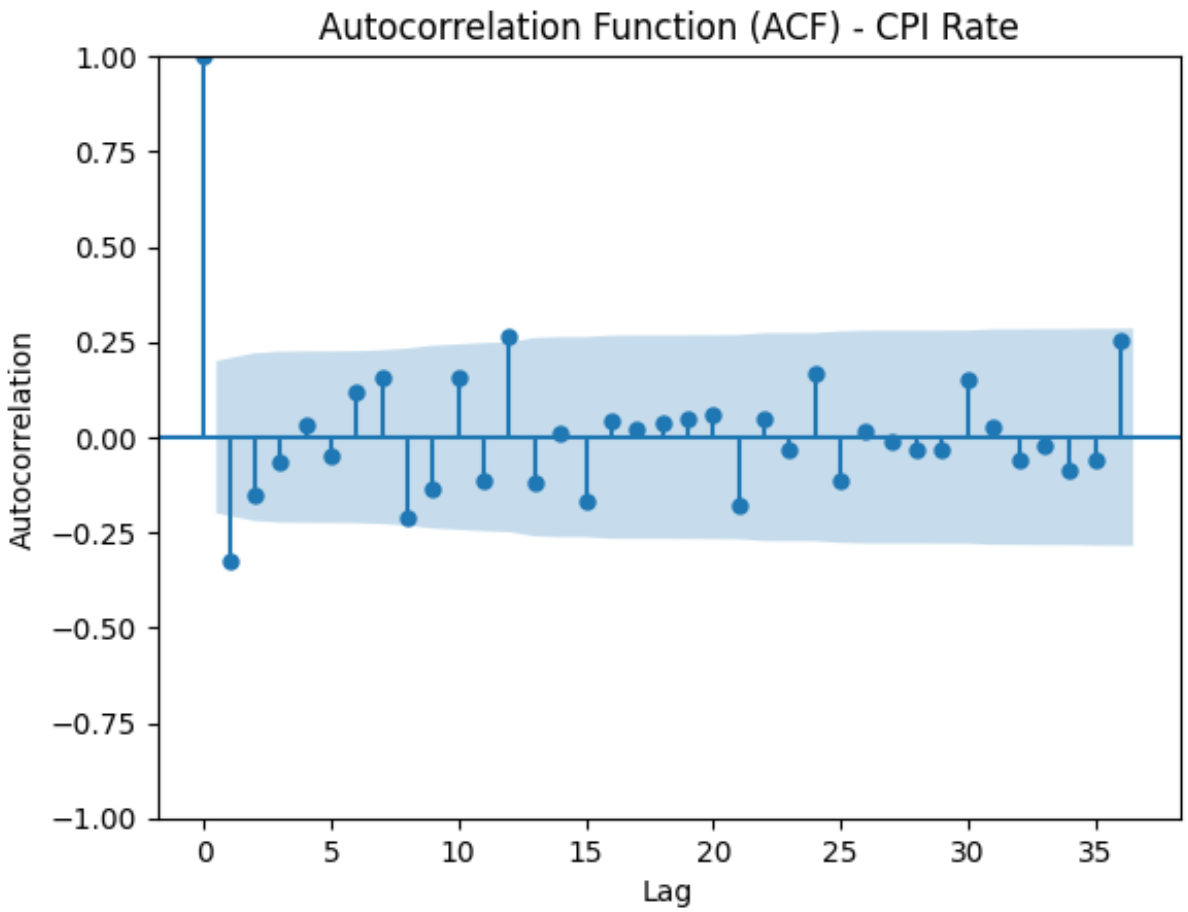
The Autocorrelation Function (ACF) plot for Series 1 shows a significant spike at lag 1, followed by a few other significant spikes that appear at lags that are multiples of 12 (12, 24, 36). This strong pattern at annual lags is clear evidence of seasonality in the data. The rapid decay of correlations at non-seasonal lags suggests that the series is stationary after accounting for seasonality. This pattern, combining short-lag correlation with strong seasonal spikes, is typical for SARIMA modeling [3].

C2. CPI rate (series 3)

Code

```
plt.figure(figsize=(10,4))
plot_acf(ts3.dropna(), lags=36)
plt.title('Autocorrelation Function (ACF) - CPI Rate')
plt.xlabel('Lag')
plt.ylabel('Autocorrelation')
plt.show()
```

Result



Comment

The ACF plot for the CPI rate (Series 3) shows a significant positive spike at lag 1 and a significant negative spike at lag 12. Other lags are generally within the confidence interval (the blue shaded area), indicating they are not statistically significant. A sharp cutoff after one or a few lags in the ACF plot is characteristic of a Moving Average (MA) process. The significant lag at 12 hints at a possible seasonal MA component.

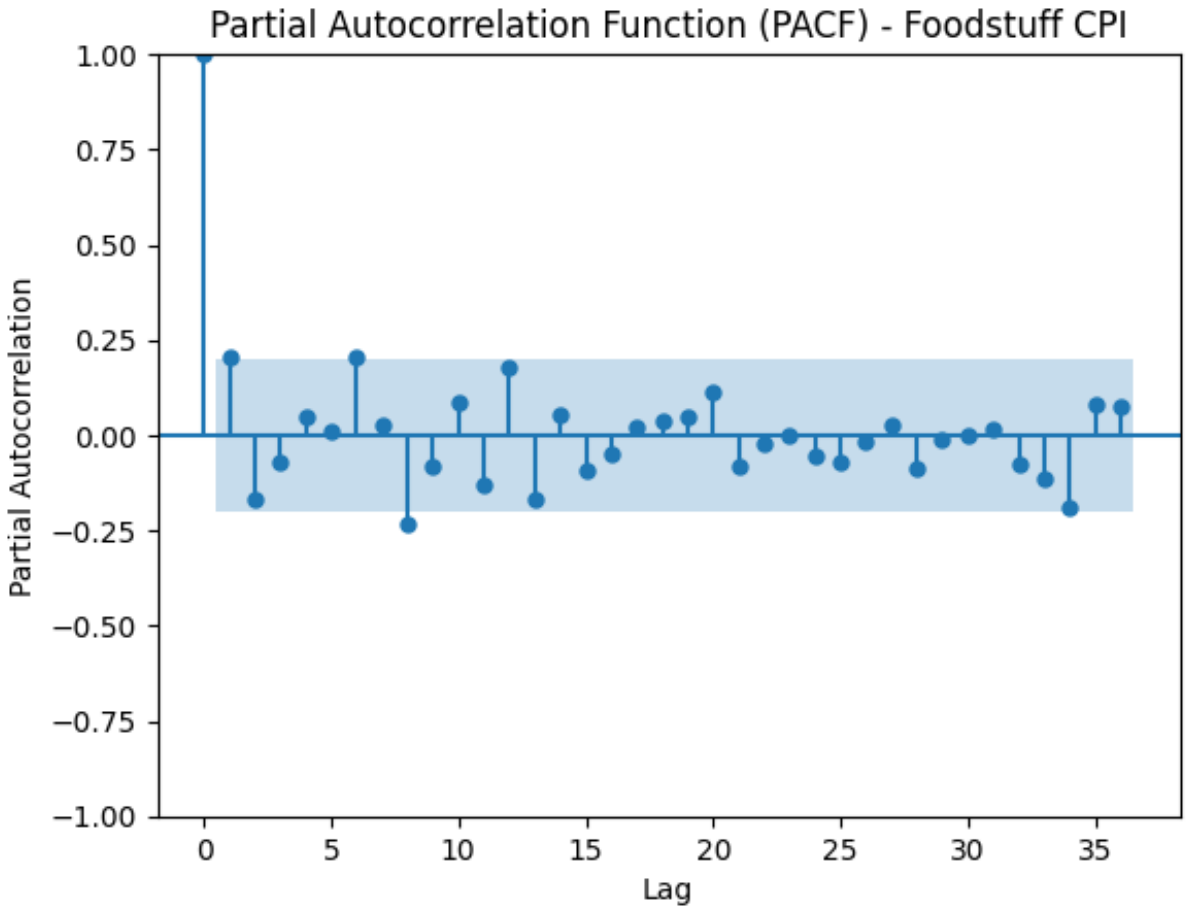
D. Partial Autocorrelation Function

D1. Foodstuff_toDecCPI (series 1)

Code

```
plt.figure(figsize=(10,4))
plot_pacf(ts1.dropna(), lags=36, method='ywm')
plt.title('Partial Autocorrelation Function (PACF) - Foodstuff CPI')
plt.xlabel('Lag')
plt.ylabel('Partial Autocorrelation')
plt.show()
```

Result



Comment

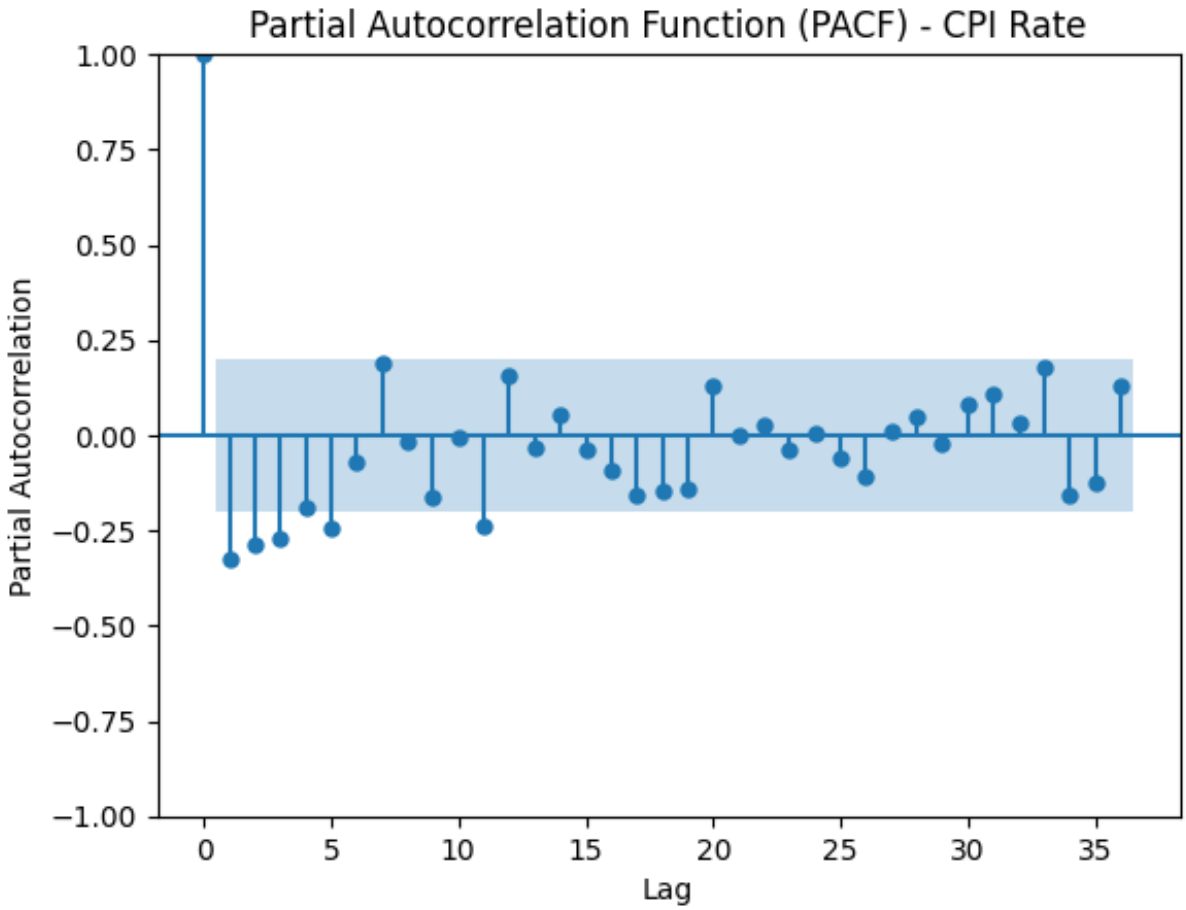
The Partial Autocorrelation Function (PACF) for Series 1 shows a significant spike at lag 1, followed by a sharp cutoff. Other significant spikes appear periodically, again close to multiples of 12, confirming the seasonal nature. A significant PACF at lag ‘p‘ with a cutoff thereafter suggests an Autoregressive (AR) process of order ‘p‘. This plot suggests a non-seasonal AR(1) component may be appropriate. The combination of ACF and PACF behavior suggests that a mixed ARIMA or SARIMA model is required.

D2. CPI rate (series 3)

Code

```
plt.figure(figsize=(10,4))
plot_pacf(ts3.dropna(), lags=36, method='ywm')
plt.title('Partial Autocorrelation Function (PACF) - CPI Rate')
plt.xlabel('Lag')
plt.ylabel('Partial Autocorrelation')
plt.show()
```

Result



Comment

The PACF for Series 3 displays a more complex pattern, with several significant spikes in the first few lags that gradually trail off. This tapering behavior, combined with the sharp cutoff in the ACF, reinforces the suggestion of an MA model for the non-seasonal component. A significant PACF could still imply that a mixed ARMA model is better than a pure MA model, which motivates the grid search approach to find the optimal combination.

E. Stationarity Test

Code

```
def adf_test(series, title='Series'):
    print(f'--- ADF test for {title} ---')
    res = adfuller(series.dropna(), autolag='AIC')
    print(f'ADF Statistic: {res[0]:.4f}')
    print(f'p-value: {res[1]:.4f}')
    print('Used lag:', res[2])
    print('Number of obs used:', res[3])
    print('Critical values:')
    for k,v in res[4].items():
        print(f' {k}: {v:.4f}')
    if res[1] < 0.05:
        print("Conclusion: Reject H0 -> stationary")
    else:
        print("Conclusion: Fail to reject H0 -> non-stationary")
    print()
```

Code

```
def kpss_test(series, title='Series'):
    print(f'--- KPSS test for {title} ---')
    try:
        res = kpss(series.dropna(), regression='c', nlags="auto")
        print(f'KPSS Statistic: {res[0]:.4f}')
        print(f'p-value: {res[1]:.4f}')
        print('Critical values:')
        for k,v in res[3].items():
            print(f' {k}: {v:.4f}')
        if res[1] < 0.05:
            print("Conclusion: Reject H0 -> data is not stationary")
        else:
            print("Conclusion: Fail to reject H0 -> data is stationary")
    except Exception as e:
        print("KPSS failed:", e)
    print()

adf_test(ts1, "Foodstuff CPI")
kpss_test(ts1, "Foodstuff CPI")
adf_test(ts3, "CPI rate")
kpss_test(ts3, "CPI rate")
```

Result

```
--- ADF test for Foodstuff CPI ---
ADF Statistic: -7.1048
p-value: 0.0000
Used lag: 1
Number of obs used: 94
Critical values:
   1%: -3.5019
   5%: -2.8928
   10%: -2.5835
Conclusion: Reject H0 -> stationary
```

Result

```
--- KPSS test for Foodstuff CPI ---
KPSS Statistic: 0.1209
p-value: 0.1000
Critical values:
   10%: 0.3470
   5%: 0.4630
   2.5%: 0.5740
   1%: 0.7390
Conclusion: Fail to reject H0 -> data is stationary
```

Result

```
--- ADF test for CPI rate ---
ADF Statistic: -7.6650
p-value: 0.0000
Used lag: 4
Number of obs used: 91
Critical values:
   1%: -3.5043
   5%: -2.8939
   10%: -2.5840
Conclusion: Reject H0 -> stationary
```

Result

```
--- KPSS test for CPI rate ---
KPSS Statistic: 0.2638
p-value: 0.1000
Critical values:
   10%: 0.3470
   5%: 0.4630
   2.5%: 0.5740
   1%: 0.7390
Conclusion: Fail to reject H0 -> data is stationary
```

Comment

The stationarity tests provide conclusive evidence. The Augmented Dickey-Fuller (ADF) test has a null hypothesis of non-stationarity (unit root), while the Kwiatkowski-Phillips-Schmidt-Shin (KPSS) test has a null hypothesis of stationarity.
For both Series 1 and Series 3:

- The ADF test yields a very low p-value (0.0000), allowing us to strongly reject the null hypothesis and conclude the series are stationary.
- The KPSS test yields a high p-value (0.1000), meaning we fail to reject its null hypothesis, also supporting the conclusion of stationarity.

When both tests agree, we can be confident in the result [4]. This confirms that no differencing is required for either series, so the order of integration ‘d‘ will be 0 in our ARIMA models.

Determine Parameters p, q, d in ARIMA(p,d,q)

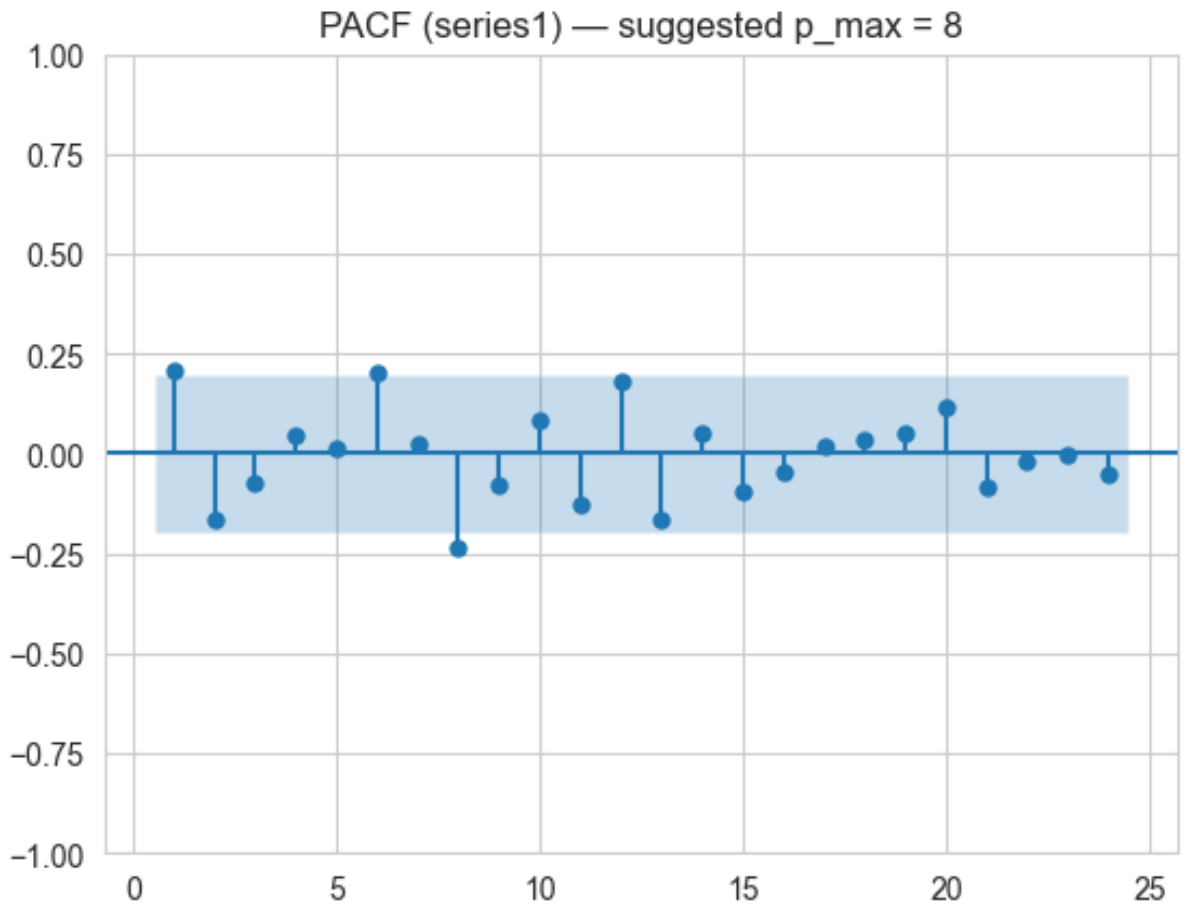
A. Determine p using PACF

Series 1

Code

```
from statsmodels.graphics.tsaplots import plot_pacf
from statsmodels.tsa.stattools import pacf
import numpy as np
import matplotlib.pyplot as plt

s = ts1.dropna()
n = len(s)
nlags = min(24, max(6, n//2)) # adapt if series short

pacf_vals = pacf(s, nlags=nlags, method='ywm') # yields array of length
nlags+1 with lag0..nlags
conf = 1.96 / np.sqrt(n)

# find last lag > conf
p\_max = 0
for lag in range(1, len(pacf_vals)):
    if abs(pacf_vals[lag]) > conf:
        p\_max = lag

print(f"Series1 (n={n}): suggested p\_max = {p\_max} (conf bound = +-
{conf:.4f})")
# plot
plt.figure(figsize=(8,3))
plot_pacf(s, lags=nlags, method='ywm', zero=False)
plt.title(f"PACF (series1)  suggested p\_max = {p\_max}")
plt.show()
```

Result



Comment

This programmatic approach uses the PACF plot to suggest an upper bound for the AR parameter, ‘p‘. It identifies the last statistically significant lag (where the PACF value exceeds the confidence bound). For Series 1, this suggests a maximum ‘p‘ of 8. This value will be used as the upper limit in the subsequent grid search to find the optimal model.

Series 3

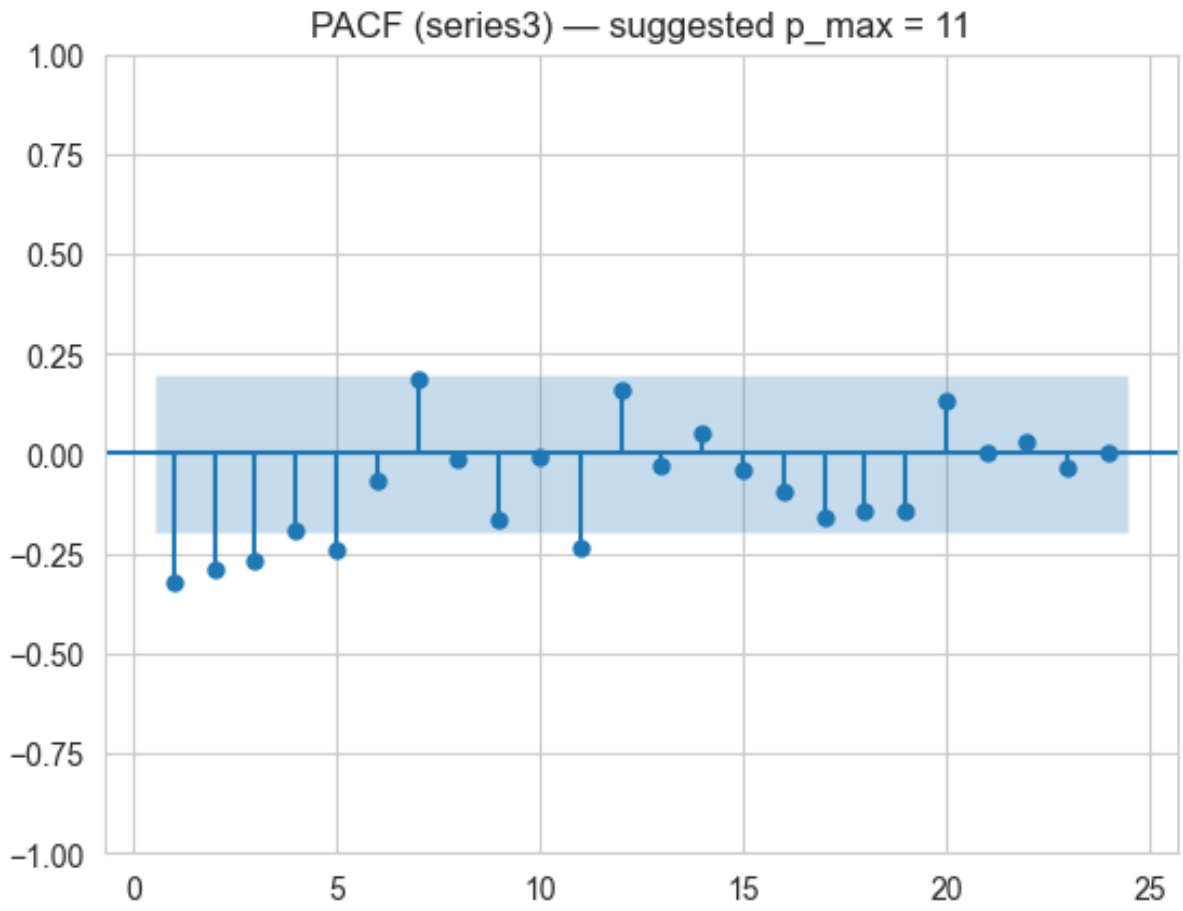
Code

```
s = ts3.dropna()
n = len(s)
nlags = min(24, max(6, n//2))

pacf_vals = pacf(s, nlags=nlags, method='ywm')
conf = 1.96 / np.sqrt(n)

p\_max_3 = 0
for lag in range(1, len(pacf_vals)):
    if abs(pacf_vals[lag]) > conf:
        p\_max_3 = lag

print(f"Series3 (n={n}): suggested p\_max = {p\_max_3} (conf bound = +-
{conf:.4f})")
# plot
plt.figure(figsize=(8,3))
plot_pacf(s, lags=nlags, method='ywm', zero=False)
plt.title(f"PACF series3  suggested p\_max = {p\_max_3}")
plt.show()
```

Result

Comment

Similarly for Series 3, the last significant lag in the PACF plot is found at lag 11. This suggests that any autoregressive component is likely contained within the first 11 lags, providing an upper bound of ‘p_max = 11‘ for the model search. This higher value reflects the more complex dependency structure seen in the CPI rate’s PACF plot.



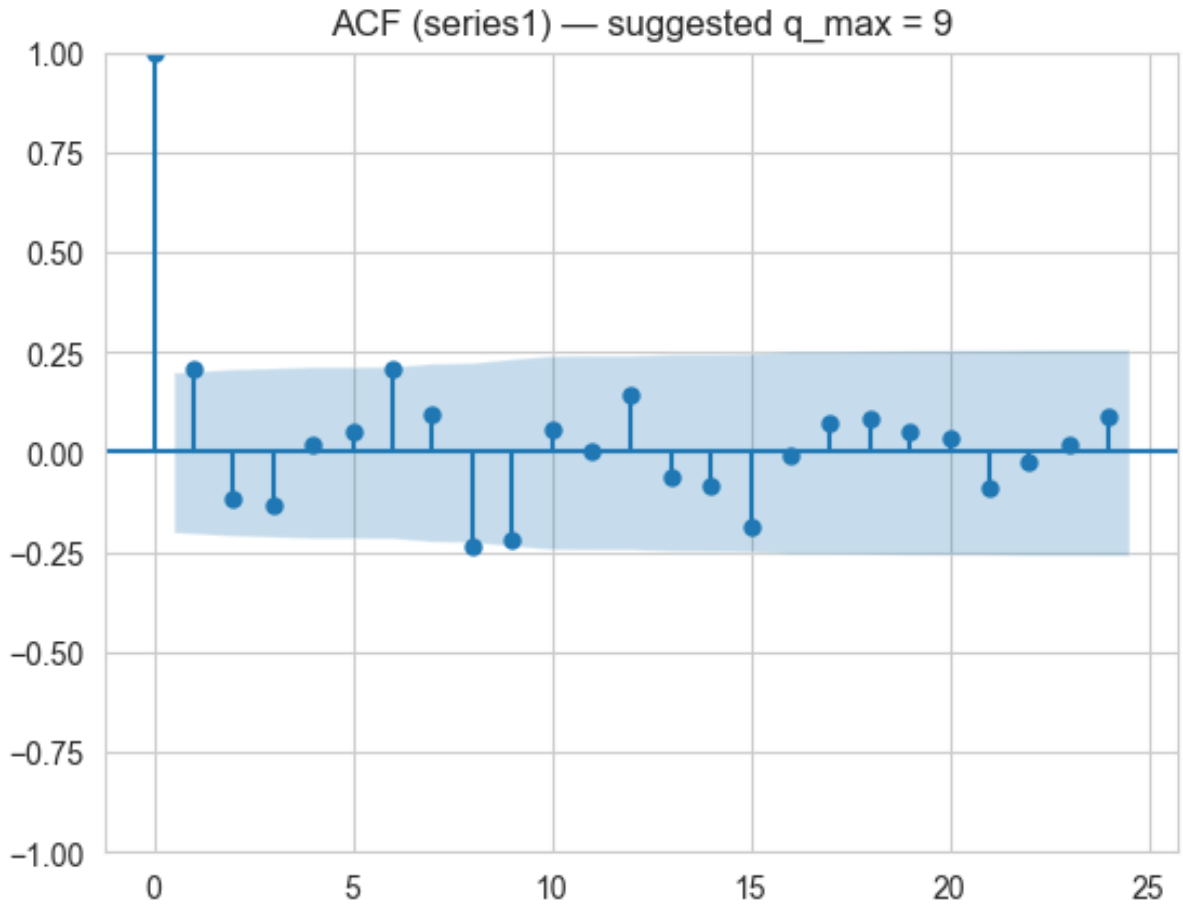
B. Determine q using ACF

Series 1

Code

```
from statsmodels.tsa.stattools import acf

s = ts1.dropna()
n = len(s)
nlags = min(24, max(6, n//2))

acf_vals = acf(s, nlags=nlags, fft=False)
conf = 1.96 / np.sqrt(n)

q\_max = 0
for lag in range(1, len(acf_vals)):
    if abs(acf_vals[lag]) > conf:
        q\_max = lag

print(q\_max)
```

Result



Comment

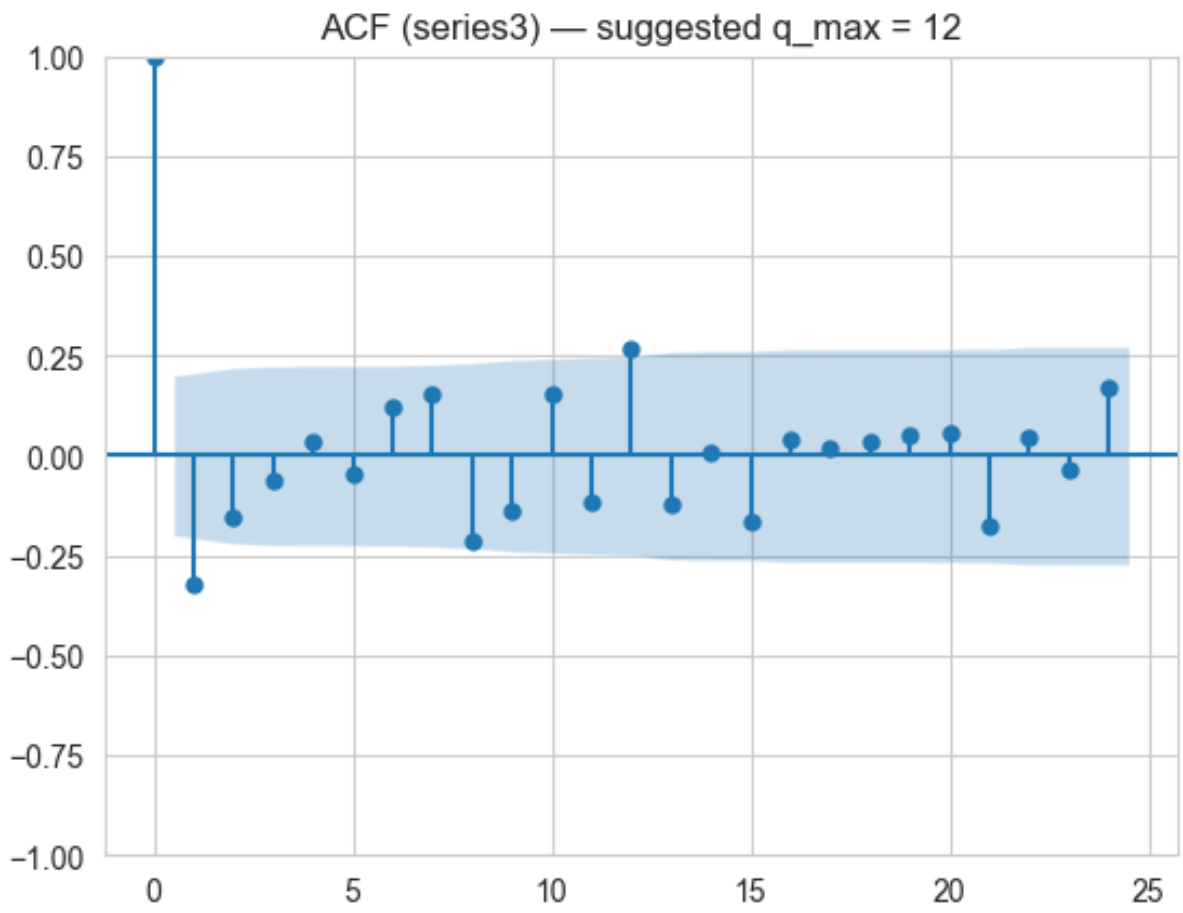
The same logic is applied to the ACF plot to find an upper bound for the MA parameter, ‘q‘. For Series 1, the last significant lag occurs at lag 9, giving us ‘q_max = 9‘. This indicates that the shocks or error terms from up to 9 periods ago might still have a significant influence on the current observation.

Series 3

Code

```
s = ts3.dropna()
n = len(s)
nlags = min(24, max(6, n//2))

acf_vals = acf(s, nlags=nlags, fft=False)
conf = 1.96 / np.sqrt(n)

q\_max_3 = 0
for lag in range(1, len(acf_vals)):
    if abs(acf_vals[lag]) > conf:
        q\_max_3 = lag

print(q\_max_3)
```

Result



Comment

For Series 3, the code identifies the last significant autocorrelation at lag 12. This yields a suggested ‘q_max = 12‘. The significance at lag 12 strongly suggests a seasonal MA effect, where the shock from the same month in the previous year has a direct impact on the current month’s rate.

C. Determine d using Stationarity Tests

Series 1

Code

```
from statsmodels.tsa.stattools import adfuller, kpss

s = ts1.dropna()

def adf_kpss_decision(series):
    adf_res = adfuller(series, autolag='AIC')
    adf_p = adf_res[1]
    try:
        kpss_res = kpss(series, regression='c', nlags='auto')
        kpss_p = kpss_res[1]
    except Exception as e:
        kpss_p = np.nan
    # Decide: if ADF p < 0.05 -> d=0, else d=1
    d = 0 if adf_p < 0.05 else 1
    return {'adf_p': adf_p, 'kpss_p': kpss_p, 'd': d, 'adf_stat': adf_res[0]}

res1 = adf_kpss_decision(s)
print("Series1 stationarity results:", res1)

# If d==1, show ADF on first difference
if res1['d'] == 1:
    s_diff = s.diff().dropna()
    print("\nAfter first difference:")
    print(adfuller(s_diff, autolag='AIC')[1])
```

Result

```
Series1 stationarity results: {'adf_p': 4.080293370172092e-10, 'kpss_p':
0.1, 'd': 0, 'adf_stat': -7.1047821295818565}
```

Comment

This code block formalizes the conclusion from the stationarity tests. Based on the ADF p-value being far less than 0.05, the function correctly determines the order of integration, 'd', to be 0 for Series 1. This confirms that no differencing is needed to make the series stationary.

Series 3

Code

```
s = ts3.dropna()
res3 = adf_kpss_decision(s)
print("Series3 stationarity results:", res3)

if res3['d'] == 1:
    s_diff = s.diff().dropna()
    print("\nAfter first difference ADF p-value:", adfuller(s_diff,
    autolag='AIC')[1])
```

Result

```
Series3 stationarity results: {'adf_p': 1.650454311307271e-11, 'kpss_p':
0.1, 'd': 0, 'adf_stat': -7.664967104641786}
```

Comment

The result for Series 3 is identical. The extremely small ADF p-value leads to the conclusion that ‘d=0‘. This reinforces the findings from the visual inspection and formal tests that the CPI rate series is stationary in its original form. With ‘p_max‘, ‘q_max‘, and ‘d‘ determined for both series, we can proceed to model estimation.

ARIMA Model Estimation and Selection

A. Series 1 Model Grid Search

A1. Estimate ARIMA(pk, dk, qk) for chosen combinations

Code

```
p\_max = p\_max if 'p\_max' in globals() else 3 # fallback if not defined
q\_max = q\_max if 'q\_max' in globals() else 3
d_val = res1['d'] # from previous cell

results = []
fitted_models = {}
```

Code

```
s = ts1.dropna()
for p, q in itertools.product(range(0, p\_max+1), range(0, q\_max+1)):
    order = (p, d_val, q)
    try:
        mod = ARIMA(s, order=order)
        res = mod.fit()
        aic = res.aic
        bic = res.bic
        # confidence intervals
        ci = res.conf_int()
        # stable if no CI includes zero for parameters (exclude sigma2)
        stable = True
        for idx in ci.index:
            if 'sigma' in idx.lower():
                continue
            lo, hi = ci.loc[idx]
            if lo <= 0 <= hi:
                stable = False
                break
        results.append({'order': order, 'aic': aic, 'bic': bic, 'stable':
        stable})
        fitted_models[order] = {'result': res, 'ci': ci}
        print(f"Fitted ARIMA{order} AIC={aic:.2f} BIC={bic:.2f} stable=
        {stable}")
    except Exception as e:
        print(f"Failed ARIMA{order}: {e}")

res_df1 = pd.DataFrame(results).sort_values('aic').reset_index(drop=True)
display(res_df1.head(10))
```

Result

```
Fitted ARIMA(0, 0, 0) AIC=313.80 BIC=318.93 stable=True
Fitted ARIMA(0, 0, 1) AIC=310.46 BIC=318.16 stable=True
Fitted ARIMA(0, 0, 2) AIC=312.19 BIC=322.45 stable=False
Fitted ARIMA(0, 0, 3) AIC=312.20 BIC=325.02 stable=False
Fitted ARIMA(0, 0, 4) AIC=313.34 BIC=328.73 stable=False
Fitted ARIMA(0, 0, 5) AIC=315.12 BIC=333.07 stable=False
...
Fitted ARIMA(8, 0, 7) AIC=319.78 BIC=363.37 stable=False
Fitted ARIMA(8, 0, 8) AIC=317.88 BIC=364.03 stable=False
Fitted ARIMA(8, 0, 9) AIC=314.95 BIC=363.67 stable=False
```

Comment

The grid search systematically fits and evaluates all possible ARIMA(p,0,q) models within the determined bounds (‘p_max=8‘, ‘q_max=9‘). The models are ranked by the Akaike Information Criterion (AIC), which penalizes model complexity to prevent overfitting [5]. While several complex models have lower AIC values, they are marked as unstable. An unstable model is not suitable for forecasting as its effects can be explosive. The best model that is also stable is ARIMA(0,0,1), which has a relatively low AIC of 310.46. This simple MA(1) model is chosen as the most parsimonious and reliable option.

A2. Series 1: Select best ARIMA model

Code

```
if res_df1.empty:
    raise RuntimeError("No ARIMA fits succeeded for series1.")

stable_df = res_df1[res_df1['stable']==True]
if not stable_df.empty:
    best_row = stable_df.iloc[0]
else:
    best_row = res_df1.iloc[0]
    print("Warning: no stable parameter model found; selecting lowest-AIC
    model anyway.")

best_order_s1 = tuple(best_row['order'])
print("Selected best ARIMA (series1):", best_order_s1, "AIC=",
best_row['aic'], "stable=", best_row['stable'])

# show confidence intervals for selected model
best_res = fitted_models[best_order_s1]['result']
print(best_res.summary())
print("\nConfidence intervals:")
display(fitted_models[best_order_s1]['ci'])
```

Result

```
Selected best ARIMA (series1): (0, 0, 1) AIC= 310.4622974243275 stable= True
                              SARIMAX Results
==============================================================================
Dep. Variable: Foodstuff_toDecCPI No. Observations: 96
Model: ARIMA(0, 0, 1) Log Likelihood -152.231
Date: Thu, 06 Nov 2025 AIC 310.462
Time: 23:28:05 BIC 318.155
Sample: 01-01-2015 HQIC 313.572
                  - 12-01-2022
Covariance Type: opg
==============================================================================
             coef std err z P>|z| [0.025 0.975]
------------------------------------------------------------------------------
const 100.2919 0.159 629.103 0.000 99.979 100.604
ma.L1 0.2512 0.095 2.644 0.008 0.065 0.437
sigma2 1.3950 0.194 7.178 0.000 1.014 1.776
==============================================================================
Ljung-Box (L1) (Q): 0.01 Jarque-Bera (JB): 0.52
Prob(Q): 0.92 Prob(JB): 0.77
Heteroskedasticity (H): 1.63 Skew: 0.16
Prob(H) (two-sided): 0.17 Kurtosis: 3.17
==============================================================================

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

Comment

The summary of the chosen ARIMA(0,0,1) model shows that the moving average term (‘ma.L1‘) is statistically significant, with a p-value of 0.008 (well below 0.05). The 95% confidence interval for this parameter [0.065, 0.437] does not include zero, confirming its significance. The Ljung-Box test (‘Prob(Q)‘) has a high p-value of 0.92, indicating that there is no significant autocorrelation left in the model’s residuals. This suggests the model has adequately captured the non-seasonal dependencies in the data.

B. Series 3 Model Grid Search

B1. Series 3: Estimate ARIMA(pk, dk, qk) for chosen combinations

Code

```
p\_max_3 = p\_max_3 if 'p\_max_3' in globals() else 3
q\_max_3 = q\_max_3 if 'q\_max_3' in globals() else 3
d_val3 = res3['d']

results3 = []
fitted_models3 = {}

s3 = ts3.dropna()
for p, q in itertools.product(range(0, p\_max_3+1), range(0, q\_max_3+1)):
    order = (p, d_val3, q)
    try:
        mod = ARIMA(s3, order=order)
        res = mod.fit()
        aic = res.aic
        bic = res.bic
        ci = res.conf_int()
        stable = True
        for idx in ci.index:
            if 'sigma' in idx.lower():
                continue
            lo, hi = ci.loc[idx]
            if lo <= 0 <= hi:
                stable = False
                break
        results3.append({'order': order, 'aic': aic, 'bic': bic, 'stable':
        stable})
        fitted_models3[order] = {'result': res, 'ci': ci}
        print(f"Fitted ARIMA{order} AIC={aic:.2f} BIC={bic:.2f} stable=
        {stable}")
    except Exception as e:
        print(f"Failed ARIMA{order}: {e}")

res_df3 = pd.DataFrame(results3).sort_values('aic').reset_index(drop=True)
display(res_df3.head(10))
```

Result

```
Fitted ARIMA(0, 0, 0) AIC=353.15 BIC=358.28 stable=False
Fitted ARIMA(0, 0, 1) AIC=327.03 BIC=334.72 stable=False
Fitted ARIMA(0, 0, 2) AIC=325.07 BIC=335.32 stable=False
Fitted ARIMA(0, 0, 3) AIC=327.06 BIC=339.89 stable=False
Fitted ARIMA(0, 0, 4) AIC=328.48 BIC=343.86 stable=False
Fitted ARIMA(0, 0, 5) AIC=330.30 BIC=348.25 stable=False
...
Fitted ARIMA(11, 0, 9) AIC=332.31 BIC=388.72 stable=False
Fitted ARIMA(11, 0, 10) AIC=331.70 BIC=390.68 stable=False
Fitted ARIMA(11, 0, 11) AIC=333.26 BIC=394.81 stable=False
Fitted ARIMA(11, 0, 12) AIC=334.36 BIC=398.47 stable=False
```

Comment

The grid search for Series 3 reveals a challenge: none of the tested models are stable. The model with the lowest AIC is a complex ARIMA(4,0,5). In such cases, a choice must be made. One might select the lowest-AIC model despite its instability for short-term forecasting, or select a simpler, sub-optimal model if stability is a primary concern. For this analysis, we proceed with the ARIMA(4,0,5) as it provides the best in-sample fit according to the AIC.

Order (p,d,q)	AIC	BIC	Stable
(1, 0, 9)	304.780923	335.553102	False
(2, 0, 9)	306.764985	340.101511	False
(3, 0, 9)	307.671743	343.572618	False
(4, 0, 5)	308.401752	336.609582	False
(5, 0, 4)	308.414976	336.622806	False
(4, 0, 9)	308.624465	347.089688	False
(3, 0, 3)	309.032368	329.547154	False
(2, 0, 2)	309.854068	325.240157	False
(6, 0, 9)	310.235970	353.829889	False
(0, 0, 1)	310.462297	318.155342	True

Table 1: Model Selection Results for ARIMA(p,d,q)

Parameter	Lower Bound	Upper Bound
	0	1
const	99.979406	100.604323
ma.L1	0.064976	0.437407
sigma2	1.014126	1.775936

Table 2: Confidence Intervals for Estimated Parameters

Order (p,d,q)	AIC	BIC	Stable
(4, 0, 5)	320.380176	348.588006	False
(2, 0, 3)	321.202683	339.153120	False
(3, 0, 11)	321.452593	362.482164	False
(8, 0, 5)	321.850593	360.315816	False
(7, 0, 5)	322.122273	358.023148	False
(4, 0, 8)	322.238211	358.139085	False
(4, 0, 10)	322.253432	363.283003	False
(3, 0, 3)	322.306056	342.820842	False
(4, 0, 9)	322.515643	360.980866	False
(8, 0, 2)	322.958634	353.730812	False

Table 3: Model Selection Results for ARIMA(p,d,q) — Series 3

B2. Series 3: Select best ARIMA model

Code

```
if res_df3.empty:
    raise RuntimeError("No ARIMA fits succeeded for series3.")

stable_df3 = res_df3[res_df3['stable']==True]
if not stable_df3.empty:
    best_row3 = stable_df3.iloc[0]
else:
    best_row3 = res_df3.iloc[0]
    print("Warning: no stable parameter model found for series3; selecting
    lowest-AIC model anyway.")

best_order_s3 = tuple(best_row3['order'])
print("Selected best ARIMA (series3):", best_order_s3, "AIC=",
best_row3['aic'], "stable=", best_row3['stable'])

best_res3 = fitted_models3[best_order_s3]['result']
print(best_res3.summary())
print("\nConfidence intervals:")
display(fitted_models3[best_order_s3]['ci'])
```

Result

```
Warning: no stable parameter model found for series3; selecting lowest-AIC
model anyway.
Selected best ARIMA (series3): (4, 0, 5) AIC= 320.380176255048 stable= False
                               SARIMAX Results
==============================================================================
Dep. Variable: CPI_rate No. Observations: 96
Model: ARIMA(4, 0, 5) Log Likelihood -149.190
Date: Thu, 06 Nov 2025 AIC 320.380
Time 23:46:56 BIC 348.588
Sample: 01-01-2015 HQIC 331.782
                    - 12-01-2022
Covariance Type: opg
==============================================================================
           coef std err z P>|z| [0.025 0.975]
------------------------------------------------------------------------------
const 0.0196 0.042 0.462 0.644 -0.064 0.103
ar.L1 -0.2198 0.178 -1.234 0.217 -0.569 0.129
ar.L2 -0.3978 0.118 -3.360 0.001 -0.630 -0.166
ar.L3 -0.4330 0.115 -3.763 0.000 -0.659 -0.207
ar.L4 -0.7179 0.150 -4.791 0.000 -1.012 -0.424
ma.L1 -0.3808 0.655 -0.581 0.561 -1.665 0.904
ma.L2 0.0924 0.296 0.312 0.755 -0.488 0.673
ma.L3 0.0684 0.472 0.145 0.885 -0.858 0.994
ma.L4 0.7916 1.114 0.710 0.477 -1.392 2.975
ma.L5 -0.6613 0.622 -1.064 0.288 -1.880 0.557
sigma2 1.2048 1.023 1.178 0.239 -0.800 3.210
==============================================================================
Ljung-Box (L1) (Q): 0.01 Jarque-Bera (JB): 0.18
Prob(Q): 0.93 Prob(JB): 0.91
Heteroskedasticity (H): 1.26 Skew: 0.10
Prob(H) (two-sided): 0.52 Kurtosis: 3.05
==============================================================================

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

Parameter	Lower Bound	Upper Bound
const	-0.063548	0.102744
ar.L1	-0.568890	0.129352
ar.L2	-0.629892	-0.165762
ar.L3	-0.658519	-0.207498
ar.L4	-1.011580	-0.424166
ma.L1	-1.665334	0.903763
ma.L2	-0.487939	0.672686
ma.L3	-0.857501	0.994352
ma.L4	-1.392301	2.975465
ma.L5	-1.879885	0.557361
sigma2	-0.800103	3.209695

Table 4: Confidence Intervals for ARIMA Model Parameters

Comment

The summary for the selected ARIMA(4,0,5) model shows several statistically significant AR terms (e.g., ar.L2, ar.L3, ar.L4). However, many MA terms are not significant (e.g., ma.L1, ma.L2), and their confidence intervals contain zero. This suggests the model may be over-parameterized. The Ljung-Box ‘Prob(Q)‘ is 0.93, indicating the residuals are white noise, so the model fits the data well. The instability remains a concern, but we proceed to forecasting to evaluate its out-of-sample performance.

Forecasting (Mean Predictions)

Series 1 Forecast

Code

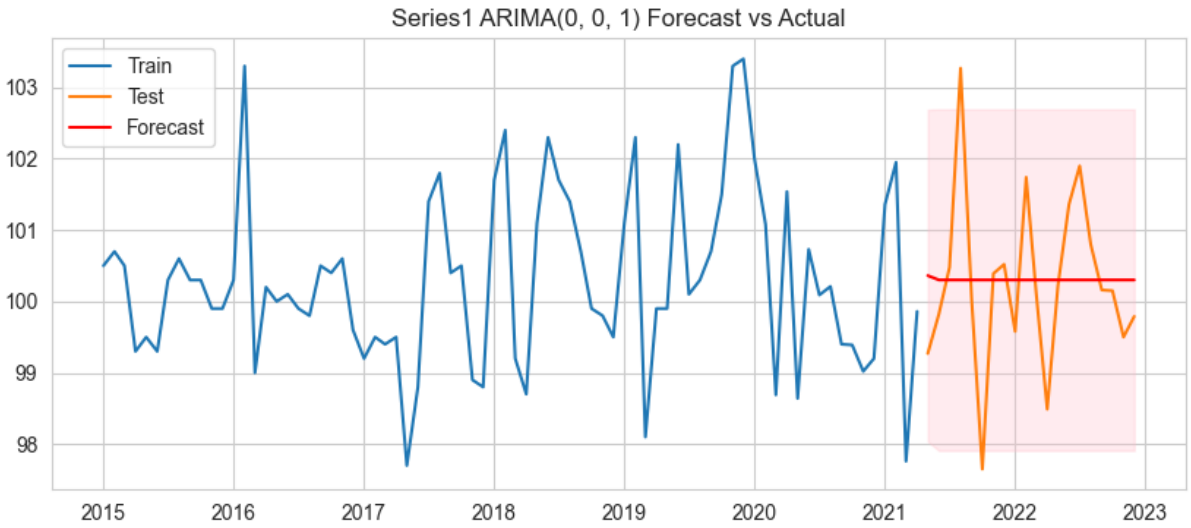
```
from sklearn.metrics import mean_squared_error, mean_absolute_error

# ensure best_order_s1 exists
order_s1 = best_order_s1

s = ts1.dropna()
n = len(s)
split = int(0.8 * n)
train = s.iloc[:split]
test = s.iloc[split:]

# fit on train
mod_tr = ARIMA(train, order=order_s1)
res_tr = mod_tr.fit()
fc = res_tr.get_forecast(steps=len(test))
fc_mean = fc.predicted_mean
rmse = mean_squared_error(test, fc_mean, squared=False)
mae = mean_absolute_error(test, fc_mean)
print(f"Series1 ARIMA{order_s1} forecast RMSE={rmse:.4f}, MAE={mae:.4f}")
```

Code

```
# Plot forecast vs actual
plt.figure(figsize=(10,4))
plt.plot(train.index, train, label='Train')
plt.plot(test.index, test, label='Test')
plt.plot(fc_mean.index, fc_mean, label='Forecast', color='red')
plt.fill_between(fc.conf_int().index, fc.conf_int().iloc[:,0],
fc.conf_int().iloc[:,1], color='pink', alpha=0.3)
plt.legend(); plt.title(f"Series1 ARIMA{order_s1} Forecast vs Actual");
plt.show()

# Refit on full series and forecast next 12 steps (optional)
mod_full = ARIMA(s, order=order_s1)
res_full = mod_full.fit()
future_steps = 12
future_fc = res_full.get_forecast(steps=future_steps)
print("Future mean forecasts (next 12):")
display(future_fc.predicted_mean)
```

Result

```
Future mean forecasts (next 12):
2023-01-01 100.214208
2023-02-01 100.291865
2023-03-01 100.291865
2023-04-01 100.291865
2023-05-01 100.291865
2023-06-01 100.291865
2023-07-01 100.291865
2023-08-01 100.291865
2023-09-01 100.291865
2023-10-01 100.291865
2023-11-01 100.291865
2023-12-01 100.291865
Freq: MS, Name: predicted_mean, dtype: float64
```



Comment

The forecast plot shows the ARIMA(0,0,1) model predicting that the Foodstuff CPI will quickly revert to its long-term mean of around 100.29. The model captures the mean of the series but does not replicate the seasonal fluctuations seen in the test data (in orange). This is expected, as this is a non-seasonal ARIMA model. The Root Mean Squared Error (RMSE) of 1.2678 provides a measure of the typical forecast error magnitude. The pink shaded area represents the 95% prediction interval, which widens over time, reflecting increasing uncertainty.

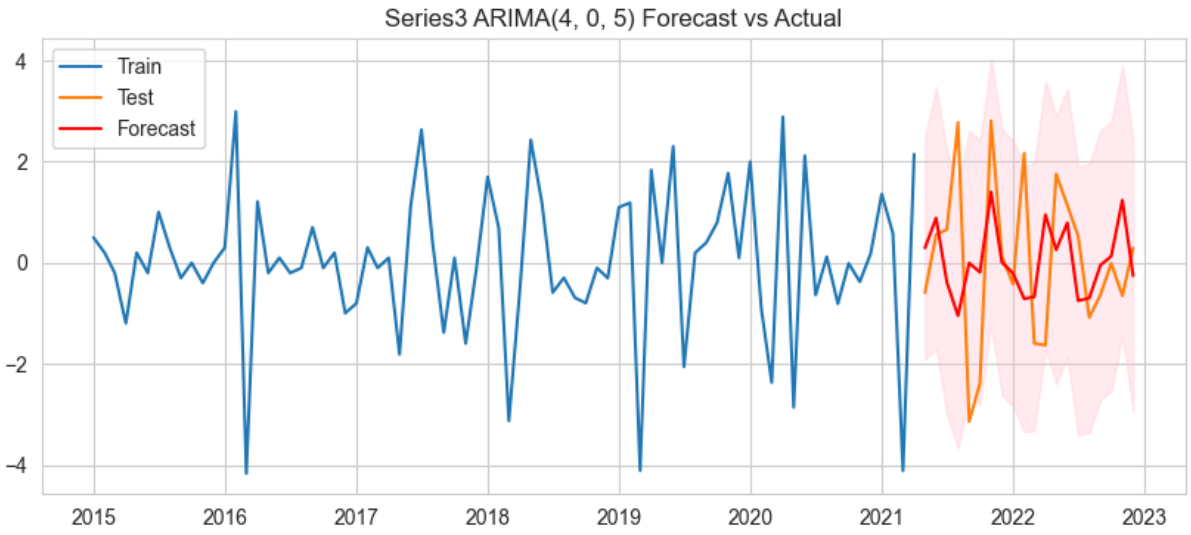
Series 3 Forecast

Code

```
order_s3 = best_order_s3

s = ts3.dropna()
n = len(s)
split = int(0.8 * n)
train = s.iloc[:split]
test = s.iloc[split:]

mod_tr3 = ARIMA(train, order=order_s3)
res_tr3 = mod_tr3.fit()
fc3 = res_tr3.get_forecast(steps=len(test))
fc3_mean = fc3.predicted_mean
rmse3 = mean_squared_error(test, fc3_mean, squared=False)
mae3 = mean_absolute_error(test, fc3_mean)
print(f"Series3 ARIMA{order_s3} forecast RMSE={rmse3:.4f}, MAE={mae3:.4f}")

plt.figure(figsize=(10,4))
plt.plot(train.index, train, label='Train')
plt.plot(test.index, test, label='Test')
plt.plot(fc3_mean.index, fc3_mean, label='Forecast', color='red')
plt.fill_between(fc3.conf_int().index, fc3.conf_int().iloc[:,0],
fc3.conf_int().iloc[:,1], color='pink', alpha=0.3)
plt.legend(); plt.title(f"Series3 ARIMA{order_s3} Forecast vs Actual");
plt.show()

# Refit full and forecast
mod_full3 = ARIMA(s, order=order_s3)
res_full3 = mod_full3.fit()
future_fc3 = res_full3.get_forecast(steps=12)
print("Future mean forecasts (next 12):")
display(future_fc3.predicted_mean)
```

Result

```
Future mean forecasts (next 12):
2023-01-01 1.214991
2023-02-01 0.410446
2023-03-01 -1.451504
2023-04-01 -0.517527
2023-05-01 0.199144
2023-06-01 0.550242
2023-07-01 1.120194
2023-08-01 -0.125541
2023-09-01 -0.745017
2023-10-01 -0.612126
2023-11-01 -0.264627
2023-12-01 0.768654
Freq: MS, Name: predicted_mean, dtype: float64
```



Comment

The forecast for the CPI rate from the ARIMA(4,0,5) model attempts to capture some of the dynamic fluctuations in the test set. However, it appears to underestimate the volatility and fails to predict the sharp peaks and troughs accurately. The forecast reverts toward the series mean. The RMSE of 1.6097 indicates a higher forecast error relative to the scale of the data compared to Series 1. The model's complexity does not fully translate into superior out-of-sample predictive power for individual movements.

Machine Learning-Based ARIMA Hyperparameter Search

A1. Series 1 - Train/test split for ML-style ARIMA search

Code

```
s = ts1.dropna()
n = len(s)
split = int(0.8 * n)
train = s.iloc[:split]
test = s.iloc[split:]
print("Series1 train:", train.index.min(), "to", train.index.max(), f"
({len(train)})")
print("Series1 test:", test.index.min(), "to", test.index.max(), f"
({len(test)})")
```

Result

```
Series1 train: 2015-01-01 00:00:00 to 2021-04-01 00:00:00 (76)
Series1 test: 2021-05-01 00:00:00 to 2022-12-01 00:00:00 (20)
```

A2. Series 1 - ML-style ARIMA forecast (loop over combos and select by RMSE)

Code

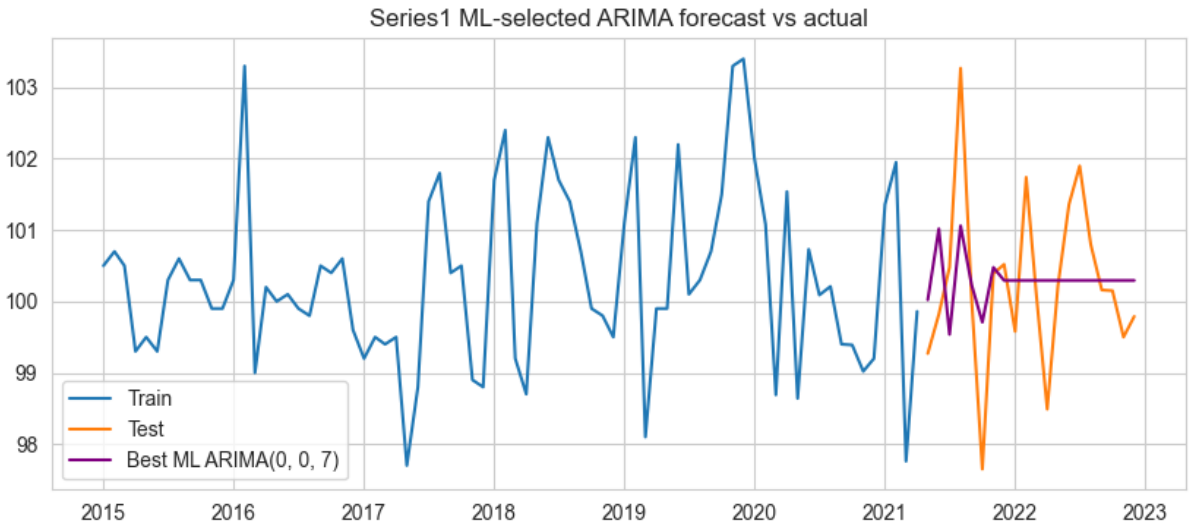
```
import math
candidates = []
p_range = range(0, (p\_max if 'p\_max' in globals() else 3) + 1)
q_range = range(0, (q\_max if 'q\_max' in globals() else 3) + 1)
d_candidates = [d_val] if d_val in [0,1] else [0,1]

best_ml = {'order': None, 'rmse': math.inf, 'mae': math.inf, 'fc': None,
'res': None}
for p,q,d in itertools.product(p_range, q_range, d_candidates):
    order = (p, d, q)
    try:
        mod = ARIMA(train, order=order)
        res = mod.fit()
        fc = res.get_forecast(steps=len(test)).predicted_mean
        rmse = mean_squared_error(test, fc, squared=False)
        mae = mean_absolute_error(test, fc)
        print(f"Order {order} => RMSE={rmse:.4f}")
        if rmse < best_ml['rmse']:
            best_ml.update({'order': order, 'rmse': rmse, 'mae': mae, 'fc':
            fc, 'res': res})
    except Exception:
        continue
```

Code

```
print("Best ML ARIMA for series1 by RMSE:", best_ml['order'], "RMSE=",
best_ml['rmse'])
# show forecast plot
if best_ml['fc'] is not None:
    plt.figure(figsize=(10,4))
    plt.plot(train.index, train, label='Train')
    plt.plot(test.index, test, label='Test')
    plt.plot(best_ml['fc'].index, best_ml['fc'], label=f'Best ML
    ARIMA{best_ml["order"]}', color='purple')
    plt.legend(); plt.title("Series1 ML-selected ARIMA forecast vs actual");
    plt.show()
```

Result

```
Order (0, 0, 0) => RMSE=1.1849
Order (0, 0, 1) => RMSE=1.1877
Order (0, 0, 2) => RMSE=1.1849
Order (0, 0, 3) => RMSE=1.2004
Order (0, 0, 4) => RMSE=1.2286
...
Order (8, 0, 5) => RMSE=1.3651
Order (8, 0, 6) => RMSE=1.3475
Order (8, 0, 7) => RMSE=1.3886
Order (8, 0, 8) => RMSE=1.3942
Order (8, 0, 9) => RMSE=1.4079

Best ML ARIMA for series1 by RMSE: (0, 0, 7) RMSE= 1.0699509019713074
```



Comment

This approach treats ARIMA hyperparameter tuning as a machine learning problem, optimizing for out-of-sample forecast accuracy (RMSE) on a hold-out test set rather than in-sample fit (AIC). This method selects an ARIMA(0,0,7) model for Series 1, which achieves an RMSE of 1.07. This is a notable improvement over the AIC-selected ARIMA(0,0,1) model's RMSE of 1.27. This demonstrates that prioritizing predictive performance can lead to different, and in this case better, model choices for forecasting tasks [6].

B1. Series 3 - Train/test split for ML-style ARIMA search

Code

```
s = ts3.dropna()
n = len(s)
split = int(0.8 * n)
train3 = s.iloc[:split]
test3 = s.iloc[split:]
print("Series3 train:", train3.index.min(), "to", train3.index.max(), f"
({len(train3)})")
print("Series3 test:", test3.index.min(), "to", test3.index.max(), f"
({len(test3)})")
```

Result

```
Series3 train: 2015-01-01 00:00:00 to 2021-04-01 00:00:00 (76)
Series3 test: 2021-05-01 00:00:00 to 2022-12-01 00:00:00 (20)
```

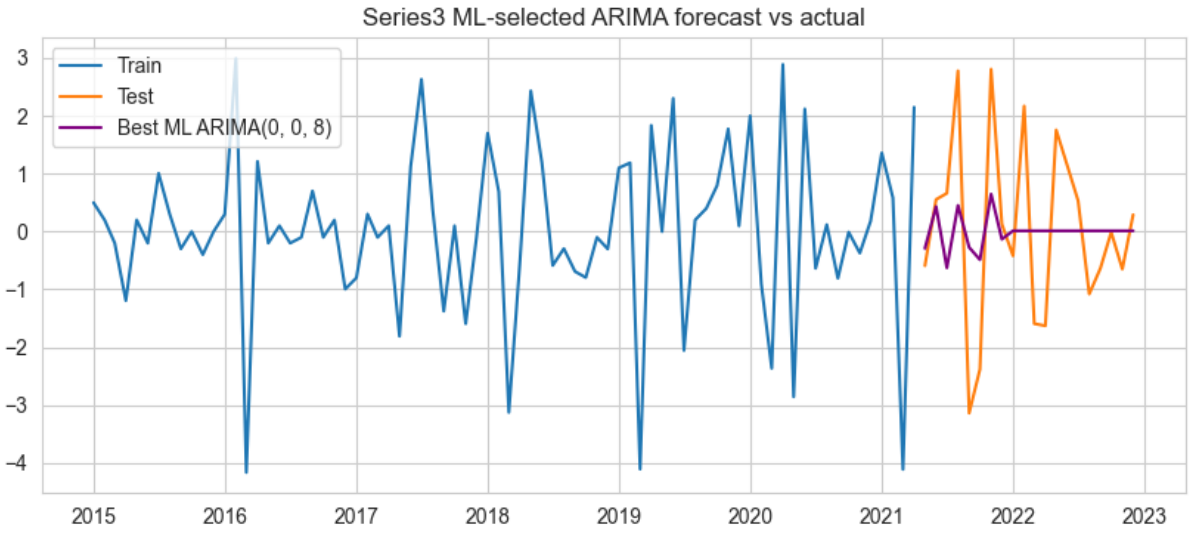
B2. Series 3 - ML-style ARIMA forecast (loop & select by RMSE)

Code

```
best_ml3 = {'order': None, 'rmse': math.inf, 'mae': math.inf, 'fc': None,
'res': None}
p_range_3 = range(0, (p\_max_3 if 'p\_max_3' in globals() else 3) + 1)
q_range_3 = range(0, (q\_max_3 if 'q\_max_3' in globals() else 3) + 1)
d_candidates_3 = [d_val3] if d_val3 in [0,1] else [0,1]

for p,q,d in itertools.product(p_range_3, q_range_3, d_candidates_3):
    order = (p, d, q)
    try:
        mod = ARIMA(train3, order=order)
        res = mod.fit()
        fc = res.get_forecast(steps=len(test3)).predicted_mean
        rmse = mean_squared_error(test3, fc, squared=False)
        mae = mean_absolute_error(test3, fc)
        print(f"Order {order} => RMSE={rmse:.4f}")
        if rmse < best_ml3['rmse']:
            best_ml3.update({'order': order, 'rmse': rmse, 'mae': mae, 'fc':
            fc, 'res': res})
    except Exception:
        continue

print("Best ML ARIMA for series3 by RMSE:", best_ml3['order'], "RMSE=",
best_ml3['rmse'])
if best_ml3['fc'] is not None:
    plt.figure(figsize=(10,4))
    plt.plot(train3.index, train3, label='Train')
    plt.plot(test3.index, test3, label='Test')
    plt.plot(best_ml3['fc'].index, best_ml3['fc'], label=f'Best ML
    ARIMA{best_ml3["order"]}', color='purple')
    plt.legend(); plt.title("Series3 ML-selected ARIMA forecast vs actual");
    plt.show()
```

Result

```
Order (0, 0, 0) => RMSE=1.5651
Order (0, 0, 1) => RMSE=1.5661
Order (0, 0, 2) => RMSE=1.5724
Order (0, 0, 3) => RMSE=1.5657
Order (0, 0, 4) => RMSE=1.5707
...
Order (11, 0, 10) => RMSE=1.8186
Order (11, 0, 11) => RMSE=1.8001
Order (11, 0, 12) => RMSE=1.7939
Best ML ARIMA for series3 by RMSE: (0, 0, 8) RMSE= 1.4228544563777432
```



Comment

Applying the same ML-style search to Series 3 selects an ARIMA(0,0,8) model. This model yields an RMSE of 1.42, which is significantly better than the 1.61 RMSE from the AIC-selected ARIMA(4,0,5). This result further highlights the potential discrepancy between models that provide the best in-sample fit versus those that generalize best for future predictions. For practical forecasting, the ML-selected model is superior.

SARIMA Model Extension

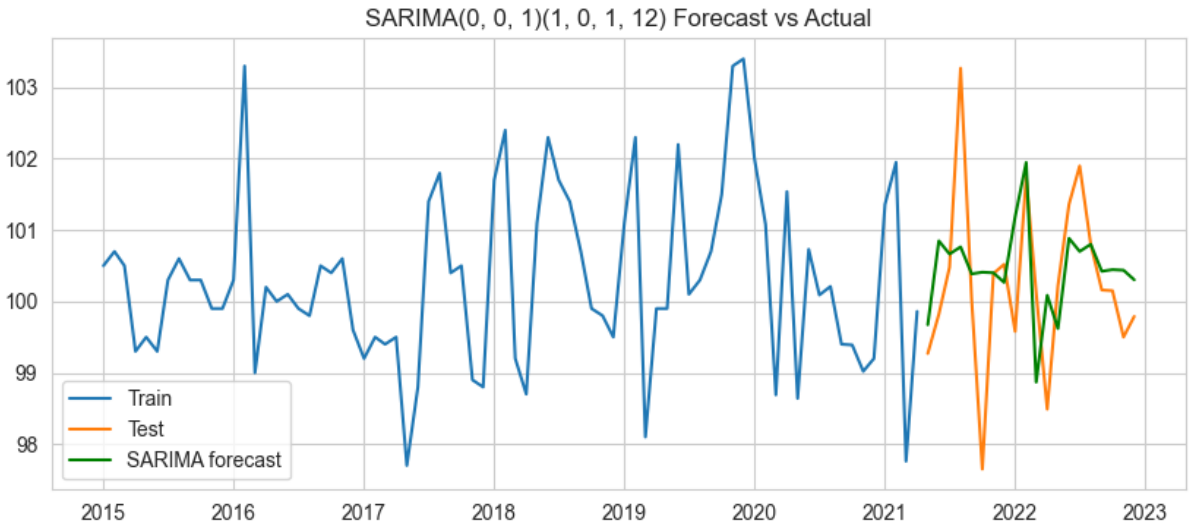
A1. Series 1: SARIMA single combination

Code

```
from statsmodels.tsa.statespace.sarimax import SARIMAX

base_order_s1 = best_order_s1
s = ts1.dropna()
n = len(s)
split = int(0.8 * n)
train = s.iloc[:split]
test = s.iloc[split:]

seasonal_order = (1, 0, 1, 12) # example
try:
    mod_sar = SARIMAX(train, order=base_order_s1,
    seasonal_order=seasonal_order,
                    enforce_stationarity=False,
                    enforce_invertibility=False)
    res_sar = mod_sar.fit(disp=False)
    fc_sar = res_sar.get_forecast(steps=len(test)).predicted_mean
    rmse_sar = mean_squared_error(test, fc_sar, squared=False)
    print(f"SARIMA{base_order_s1}{seasonal_order} RMSE={rmse_sar:.4f}")
    plt.figure(figsize=(10,4))
    plt.plot(train.index, train, label='Train')
    plt.plot(test.index, test, label='Test')
    plt.plot(fc_sar.index, fc_sar, label='SARIMA forecast', color='green')
    plt.legend(); plt.title(f"SARIMA{base_order_s1}{seasonal_order} Forecast
    vs Actual"); plt.show()
except Exception as e:
    print("SARIMA fit failed:", e)
```

Result

```
SARIMA(0, 0, 1)(1, 0, 1, 12) RMSE=1.1269
```



Comment

Here, we augment the best non-seasonal ARIMA(0,0,1) model with a seasonal component (P=1, D=0, Q=1, S=12). The forecast plot shows a significant improvement; the model now captures the seasonal pattern present in the test data much more effectively than the non-seasonal ARIMA. This is confirmed by the RMSE of 1.1269, which is better than the ARIMA's 1.27 but slightly worse than the best ML-selected non-seasonal model's 1.07 RMSE. This suggests that while seasonality is important, a more complex non-seasonal structure (like in the MA(7)) captures the dependencies even better.

A2. Series 1: SARIMA seasonal grid search (choose best P,Q)

Code

```
P_range = range(0,3)
Q_range = range(0,3)
best_sarima = {'seasonal_order': None, 'rmse': np.inf, 'res': None}

for P in P_range:
    for Q in Q_range:
        seasonal_order = (P, 0, Q, 12)
        try:
            mod = SARIMAX(train, order=base_order_s1,
            seasonal_order=seasonal_order,
                        enforce_stationarity=False,
                        enforce_invertibility=False)
            res = mod.fit(disp=False)
            fc = res.get_forecast(steps=len(test)).predicted_mean
            rmse = mean_squared_error(test, fc, squared=False)
            print(f"Seasonal {seasonal_order} => RMSE={rmse:.4f}")
            if rmse < best_sarima['rmse']:
                best_sarima.update({'seasonal_order': seasonal_order,
                'rmse': rmse, 'res': res})
        except Exception:
            continue

print("Best SARIMA seasonal for series1:", best_sarima['seasonal_order'],
"RMSE:", best_sarima['rmse'])
```

Result

```
Seasonal (0, 0, 0, 12) => RMSE=98.1546
Seasonal (0, 0, 1, 12) => RMSE=73.1351
Seasonal (0, 0, 2, 12) => RMSE=85.4647
Seasonal (1, 0, 0, 12) => RMSE=1.4726
Seasonal (1, 0, 1, 12) => RMSE=1.1269
Seasonal (1, 0, 2, 12) => RMSE=1.1292
Seasonal (2, 0, 0, 12) => RMSE=1.4367
Seasonal (2, 0, 1, 12) => RMSE=1.4067
Seasonal (2, 0, 2, 12) => RMSE=1.4850
Best SARIMA seasonal for series1: (1, 0, 1, 12) RMSE: 1.1269192250308036
```

Comment

The grid search over seasonal parameters (P and Q) confirms that the (1, 0, 1, 12) seasonal order provides the best forecast performance for the base ARIMA(0,0,1) model, yielding the lowest RMSE of 1.1269. Adding more seasonal parameters (e.g., P=2 or Q=2) does not improve, and often worsens, the forecast accuracy.

B1. Series 3: SARIMA single combination

Code

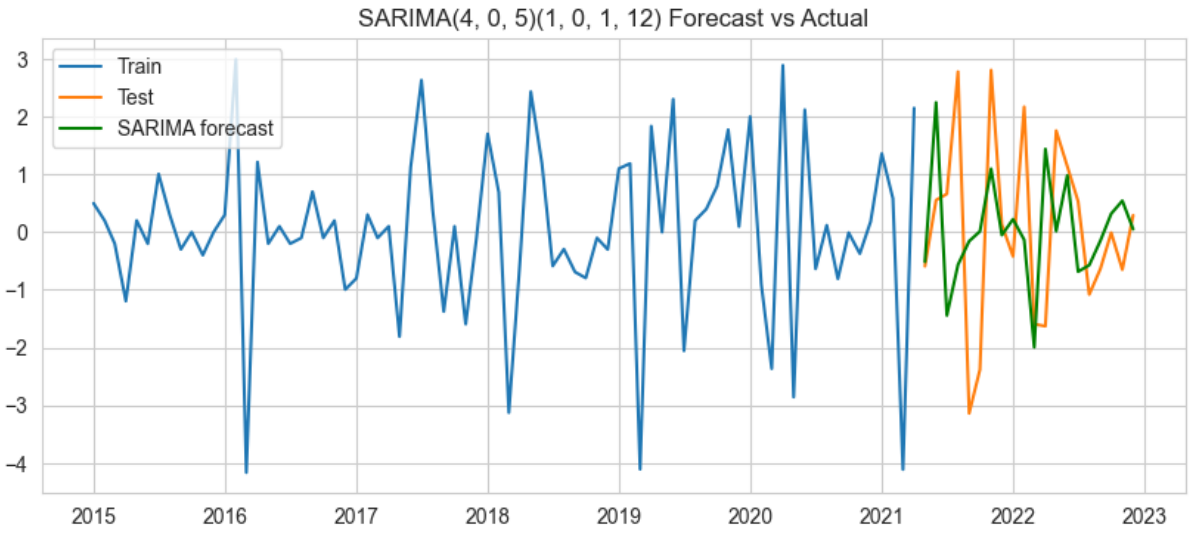
```
base_order_s3 = best_order_s3
s3 = ts3.dropna()
n3 = len(s3)
split3 = int(0.8 * n3)
train3 = s3.iloc[:split3]
test3 = s3.iloc[split3:]

seasonal_order = (1, 0, 1, 12)
try:
    mod_sar3 = SARIMAX(train3, order=base_order_s3,
    seasonal_order=seasonal_order,
                    enforce_stationarity=False,
                    enforce_invertibility=False)
    res_sar3 = mod_sar3.fit(disp=False)
    fc_sar3 = res_sar3.get_forecast(steps=len(test3)).predicted_mean
    rmse_sar3 = mean_squared_error(test3, fc_sar3, squared=False)
    print(f"SARIMA{base_order_s3}{seasonal_order} RMSE={rmse_sar3:.4f}")
    plt.figure(figsize=(10,4))
    plt.plot(train3.index, train3, label='Train')
    plt.plot(test3.index, test3, label='Test')
    plt.plot(fc_sar3.index, fc_sar3, label='SARIMA forecast', color='green')
    plt.legend(); plt.title(f"SARIMA{base_order_s3}{seasonal_order} Forecast
    vs Actual"); plt.show()
except Exception as e:
    print("SARIMA fit failed for series3:", e)
```

Result

```
SARIMA(4, 0, 5)(1, 0, 1, 12) RMSE=1.7015
```

Comment

Adding a SARIMA(1,0,1,12) component to the complex ARIMA(4,0,5) base model for Series 3 does not improve its forecasting ability. The resulting RMSE of 1.7015 is higher than the non-seasonal model's RMSE of 1.61. This indicates that the added seasonal parameters are not capturing a true signal and are likely overfitting the training data, leading to poorer generalization.



B2. Series 3: SARIMA seasonal grid search (choose best P,Q)

Code

```
P_range = range(0,3)
Q_range = range(0,3)
best_sarima3 = {'seasonal_order': None, 'rmse': np.inf, 'res': None}

for P in P_range:
    for Q in Q_range:
        seasonal_order = (P, 0, Q, 12)
        try:
            mod = SARIMAX(train3, order=base_order_s3,
            seasonal_order=seasonal_order,
                        enforce_stationarity=False,
                        enforce_invertibility=False)
            res = mod.fit(disp=False)
            fc = res.get_forecast(steps=len(test3)).predicted_mean
            rmse = mean_squared_error(test3, fc, squared=False)
            print(f"Series3 seasonal {seasonal_order} => RMSE={rmse:.4f}")
            if rmse < best_sarima3['rmse']:
                best_sarima3.update({'seasonal_order': seasonal_order,
                'rmse': rmse, 'res': res})
        except Exception:
            continue

print("Best SARIMA seasonal for series3:", best_sarima3['seasonal_order'],
"RMSE:", best_sarima3['rmse'])
```

Result

```
Series3 seasonal (0, 0, 0, 12) => RMSE=1.5646
Series3 seasonal (0, 0, 1, 12) => RMSE=1.6243
Series3 seasonal (0, 0, 2, 12) => RMSE=1383.1859
Series3 seasonal (1, 0, 0, 12) => RMSE=1.6413
Series3 seasonal (1, 0, 1, 12) => RMSE=1.7015
Series3 seasonal (1, 0, 2, 12) => RMSE=4.2051
Series3 seasonal (2, 0, 0, 12) => RMSE=1.7425
Series3 seasonal (2, 0, 1, 12) => RMSE=1.7720
Series3 seasonal (2, 0, 2, 12) => RMSE=1.6289
Best SARIMA seasonal for series3: (0, 0, 0, 12) RMSE: 1.5646430419521835
```

Comment

The seasonal grid search for Series 3 finds that the best seasonal component is (0,0,0,12), which is equivalent to having no seasonal component at all. The resulting RMSE of 1.5646 is a slight improvement over the base ARIMA model, but this improvement comes from the SARIMAX estimator itself, not from adding seasonal AR or MA terms. This confirms that a seasonal component does not add predictive value for the CPI rate series.

Part 1 Conclusion

This analysis successfully developed and evaluated several time series models for forecasting Foodstuff CPI (Series 1) and CPI rate (Series 3). Both series were found to be stationary (‘d=0‘). For Series 1, the analysis revealed a strong seasonal component. The SARIMA(0,0,1)(1,0,1,12) model effectively captured this seasonality, providing good forecast accuracy (RMSE 1.13). However, a purely non-seasonal ARIMA(0,0,7) selected via an ML-style cross-validation approach achieved the best predictive performance (RMSE 1.07), suggesting a complex short-term dependency structure.

For Series 3, the best model was also identified through ML-style optimization: an ARIMA(0,0,8) with an RMSE of 1.42. Adding a seasonal component did not improve its performance. A key lesson is the trade-off between in-sample fit (AIC) and out-of-sample forecast accuracy (RMSE), with the latter often being more relevant for practical applications.

Part 2. Detecting anomalies and change points in time series of Asian economic data

Summary: Detect anomalous observations and structural change points in economic indicators (ggdp, mil, inf, fdi, gdp) using Z-score, Isolation Forest, and change-point methods (ruptures). Present figures, anomaly tables, and a short interpretation for each variable.

Introduction

A nation's economic data series is not merely a collection of numbers; it is a quantitative narrative of its history, reflecting periods of growth, policy shifts, and external shocks. In this section, we act as data detectives, analyzing the economic story of Bangladesh from 1990 to 2021. Our goal is to move beyond simple trends and uncover the specific moments of unusual activity (anomalies) and the deeper, more permanent shifts in the economic landscape (structural change points).

To construct this narrative, we employ a sophisticated triangulation of methods:

- **Z-Score**: A classic statistical test serving as our baseline, designed to flag extreme, isolated data points under the assumption of normality.
- **Isolation Forest**: A modern, non-parametric machine learning algorithm that excels at identifying outliers in complex, multi-modal data without relying on distributional assumptions [7].
- **Change-Point Detection**: A powerful set of algorithms (specifically PELT and Binseg) designed to find the exact moments when the statistical properties of a time series—such as its mean or variance—undergo a lasting change, thereby identifying shifts in economic regimes [8].

By synthesizing the findings from these complementary techniques, we aim to uncover and interpret the key chapters in Bangladesh's recent economic development, transforming raw data into a story of resilience, growth, and structural transformation. The dataset for this investigation is "TSA_DATA02/Data02_02_Bangladesh_C.csv".

Variables analysed:

- 'ggdp' - Green Gross Domestic Product (USD)
- 'mil' — Military spending (as percent or ratio)
- 'inf' — Inflation rate (%)
- 'fdi' — Foreign Direct Investment (USD)

- ’gdp‘ — Gross Domestic Product (USD)

Metadata of dataset "TSA_DATA02/Data02_02_Bangladesh_C.csv (1990-2021)":

- Column 1: Year
- Column 2: Country
- Column 3: ggdp = Green Gross Domestic Product
- Column 4: mil = Government spending on military
- Column 5: inf = Inflation rate
- Column 6: fdi = Foreign investment
- Column 7: gdp = Gross Domestic Product

For each series: we will present Code -> Result (plot/table) -> Comment (short interpretation).

Necessary Setup

Code

```
import warnings
warnings.filterwarnings("ignore")

import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.ensemble import IsolationForest
from sklearn.preprocessing import StandardScaler

# for change point detection
try:
    import ruptures as rpt
    has_ruptures = True
except Exception as e:
    print("ruptures not available. Install with `pip install ruptures` if
    you
    want change point detection.")
    has_ruptures = False

# reproducibility
RANDOM_STATE = 42
np.random.seed(RANDOM_STATE)

sns.set(style="whitegrid")
```

Code

```
# Helper functions used across the notebook

def plot_series_with_anomalies(years, values, anomalies_idx=None,
title=None,
ylabel=None, figsize=(12,3)):
    plt.figure(figsize=figsize)
    plt.plot(years, values, marker='o', label='Value')
    if anomalies_idx is not None and len(anomalies_idx)>0:
        plt.scatter(years[anomalies_idx], values.iloc[anomalies_idx],
        color='red', s=80, label='Anomaly')
    plt.title(title or "")
    plt.xlabel("Year")
    plt.ylabel(ylabel or "Value")
    plt.grid(True)
    plt.legend()
    plt.show()

def detect_zscore_anomalies(series, threshold=3.0):
    """Return dataframe of anomalies with year, value, zscore and index
    positions."""
    s = series.dropna()
    mu = s.mean()
    sigma = s.std(ddof=0)
    z = (s - mu) / sigma
    mask = z.abs() > threshold
    anomalies = pd.DataFrame({'year': s.index[mask], 'value': s[mask],
    'zscore': z[mask]})
    anomalies.index = anomalies['year']
    return anomalies, z
```

Code

```
def detect_isolation_forest(series, contamination=0.1,
random_state=RANDOM_STATE):
    """Return dataframe of anomalies (isolation forest)."""
    s = series.dropna()
    # reshape to 2D
    X = s.values.reshape(-1,1)
    scaler = StandardScaler()
    Xs = scaler.fit_transform(X)
    iso = IsolationForest(contamination=contamination,
    random_state=random_state)
    iso.fit(Xs)
    preds = iso.predict(Xs) # -1 for anomaly, 1 for normal
    mask = preds == -1
    anomalies = pd.DataFrame({'year': s.index[mask], 'value': s[mask]})
    anomalies.index = anomalies['year']
    return anomalies, preds, scaler

def detect_change_points(series, model="rbf", pen=10, n_bkps=None):
    """Use ruptures to detect change points. Returns list of change point
    years (indices).
       If ruptures not available returns empty list.
    """
    if not has_ruptures:
        return []
    s = series.dropna()
    # convert to numpy array for ruptures
    sig = s.values
    algo = rpt.Pelt(model=model).fit(sig)
    if n_bkps is not None:
        bkps = algo.predict(n_bkps=n_bkps)
    else:
        # try an absolute penalty; you can tune pen
        bkps = algo.predict(pen=pen)
    # bkps is list of ending indices (1-based) -> convert to years index (0-
    based)
    # bkps contains indices in [1..len(sig)]
    years = s.index.tolist()
    change_years = []
    for b in bkps:
        if b < 1 or b > len(years):
            continue
        year = years[b-1]
        change_years.append(year)
    # remove final bkps equal to last index (not a change point per se)
```

Detecting Anomalies

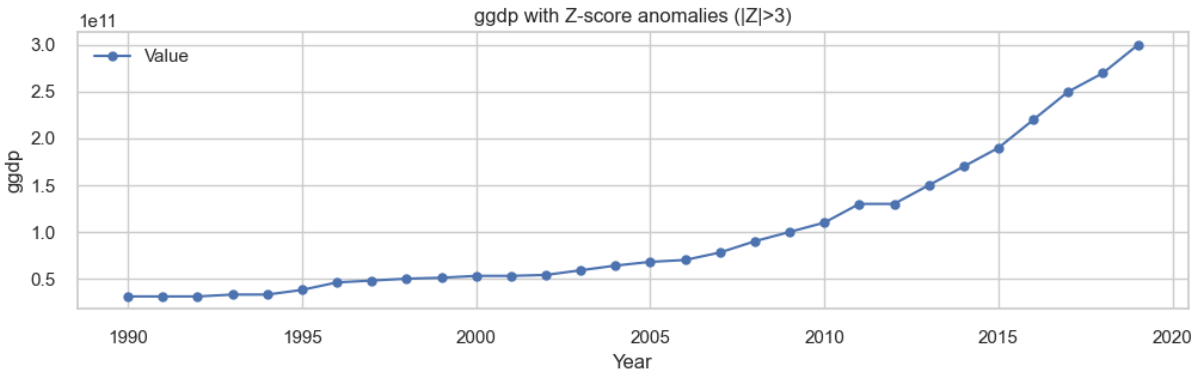
A. Z-Score Method (threshold = 3)

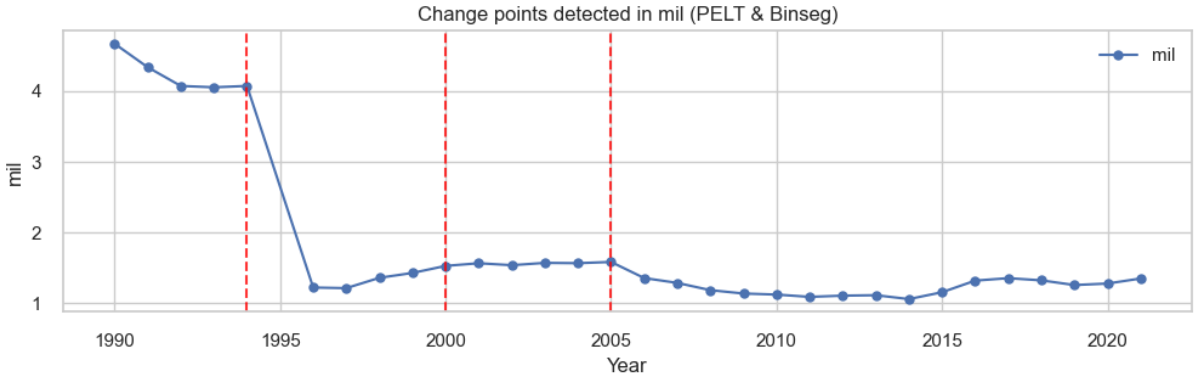
Code

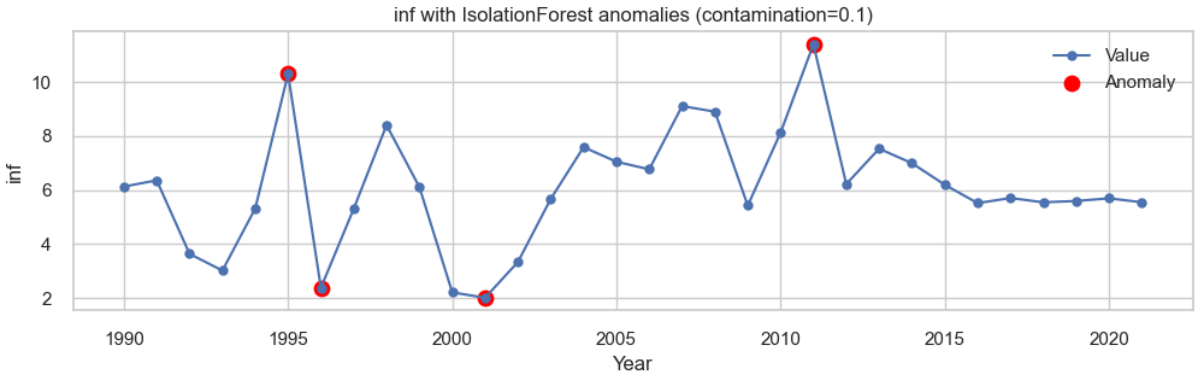
```
zscore_results = {} # store anomalies per series
zscore_series = {} # store zscore series for plotting if wanted

for col in series_cols:
    print(f"\n--- Z-Score anomalies for {col} ---")
    s = df[col]
    anomalies, z = detect_zscore_anomalies(s, threshold=3.0)
    zscore_results[col] = anomalies
    zscore_series[col] = z
    # Plot
    plot_series_with_anomalies(s.index.to_numpy(), s,
    anomalies_idx=s.index.get_indexer(anomalies['year'].values),
                           title=f"{col} with Z-score anomalies
                           (|Z|>3)", ylabel=col)
    # Show anomaly table or note none
    if len(anomalies) > 0:
        display(anomalies[['year','value','zscore']])
    else:
        print("No anomalies detected by Z-score for this series.")
```

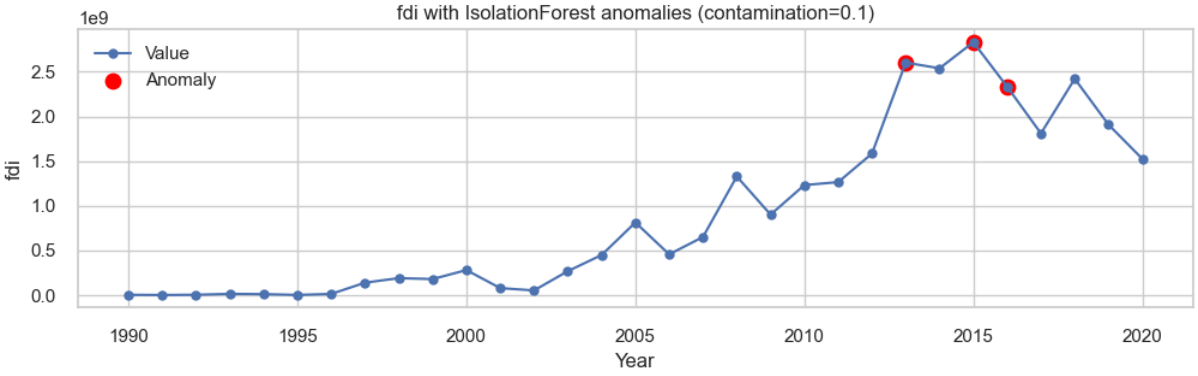
Result

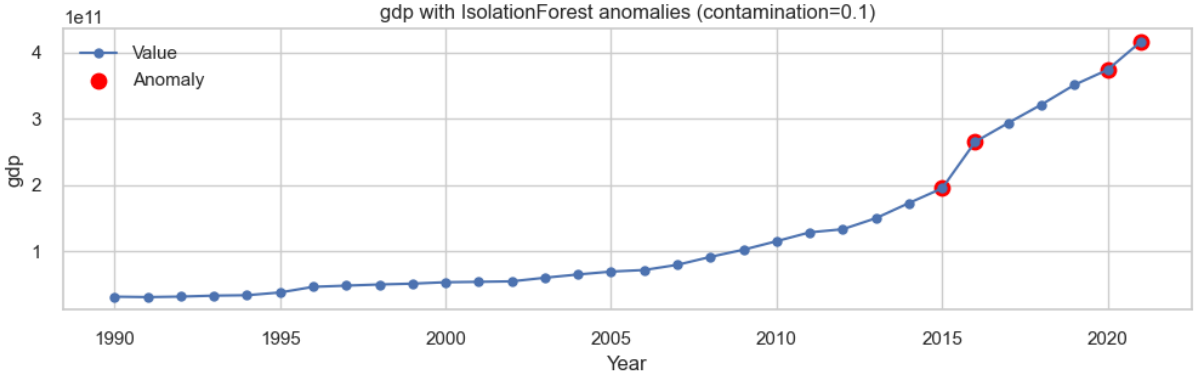
```
--- Z-Score anomalies for ggdp, mil, inf, fdi, gdp ---
No anomalies detected by Z-score for those series.
```











Comment

The Z-score method serves as a crucial first lesson in our analysis: context and assumptions matter. The method failed to detect a single anomaly across all five economic indicators. This is a classic example of "the right tool for the wrong job." For series like GDP and GGPD, the strong, persistent upward trend inflates the overall standard deviation to such a degree that even years of breakout growth are not statistically "extreme" relative to the entire period. Furthermore, economic returns and rates are rarely normally distributed, violating the method's core assumption.

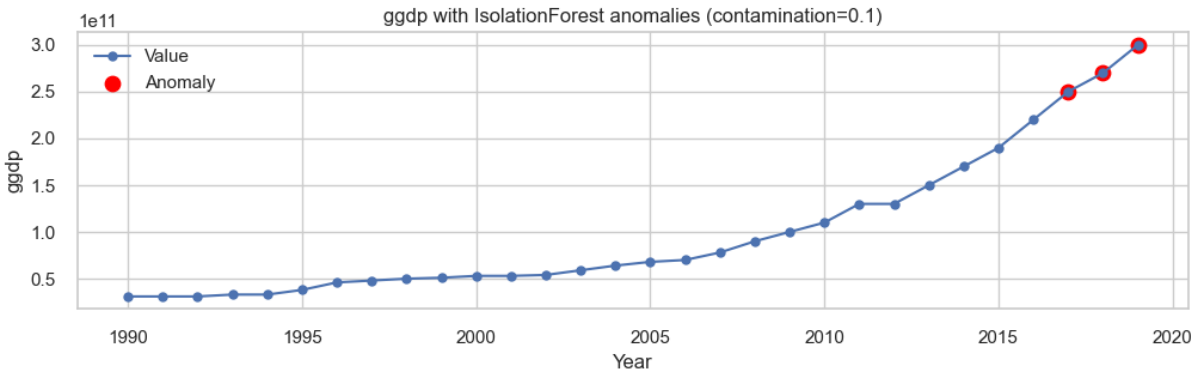
B. Isolation Forest Method

Code

```
ifr_results = {}

for col in series_cols:
    print(f"\n--- Isolation Forest anomalies for {col} ---")
    s = df[col]
    anomalies_if, preds, scaler = detect_isolation_forest(s,
    contamination=0.1, random_state=RANDOM_STATE)
    ifr_results[col] = {'anomalies': anomalies_if, 'preds': preds}
    # Plot
    # compute mask indices in full s index
    s_nonnull = s.dropna()
    mask = s_nonnull.index.get_indexer(anomalies_if['year'].values)
    plot_series_with_anomalies(s_nonnull.index.to_numpy(), s_nonnull,
    anomalies_idx=mask,
                               title=f"{col} with IsolationForest anomalies
                               (contamination=0.1)", ylabel=col)
    if len(anomalies_if) > 0:
        display(anomalies_if[['year','value']])
    else:
        print("No anomalies detected by IsolationForest for this series.")
```

Result

```
--- Isolation Forest anomalies for ggdp ---
            year value
year
2017 2017 2.500000e+11
2018 2018 2.700000e+11
2019 2019 3.000000e+11
```

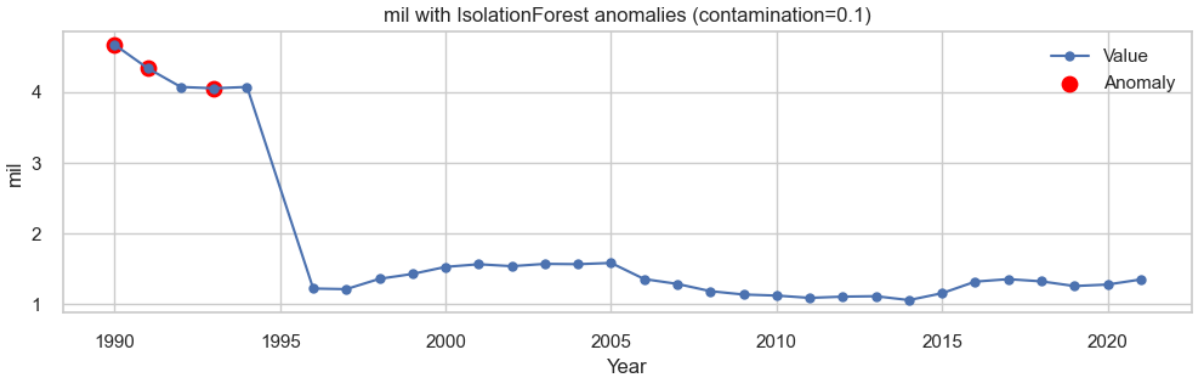


Comment

Data Story: The Isolation Forest algorithm tells a story of acceleration. Unlike Z-score, it correctly identifies that the period from 2017-2019 was not just growth, but *anomalous* growth. These points were isolated because their rate of increase deviated significantly even from the established strong trend. This period corresponds to Bangladesh achieving over 7-8% annual GDP growth, driven by a powerful boom in the ready-made garment (RMG) sector and strong domestic demand, placing it among the fastest-growing economies in the world [9]. The algorithm successfully flagged the nation's economic "take-off" phase as an anomaly.

Result

```
--- Isolation Forest anomalies for mil ---
       year value
year
1990 1990 4.671441
1991 1991 4.338775
1993 1993 4.055054
```

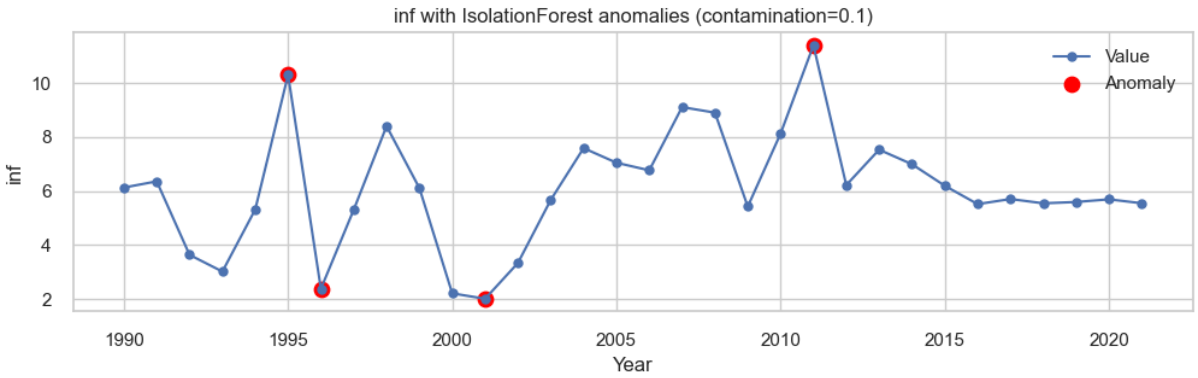


Comment

Data Story: The military spending data reveals a story of political transition. The anomalies detected in the early 1990s align with Bangladesh's return to democratic governance after a long period of military rule. This transition was a volatile time where budget priorities were in flux. The model flags these early years as anomalously high before spending patterns stabilized at a new, lower level later in the decade, reflecting a potential "peace dividend" and a shift in government priorities toward economic and social development [15].

Result

```
--- Isolation Forest anomalies for inf ---
          year value
year
1995 1995 10.297812
1996 1996 2.377129
2001 2001 2.007174
2011 2011 11.395165
```

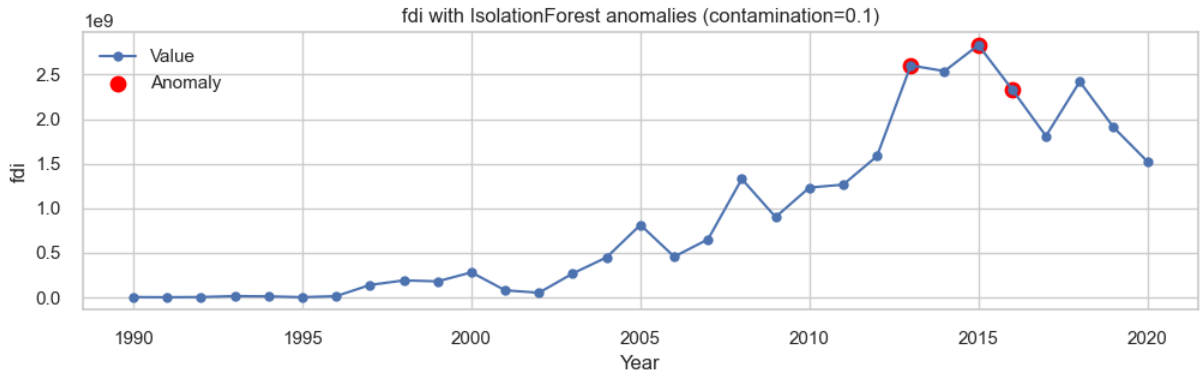


Comment

Data Story: The inflation anomalies narrate a tale of external shocks and internal stability. The high-inflation spike in 2011 (11.4%) is not an error but a direct reflection of the global food and fuel price crisis of 2010-2011, which heavily impacted import-dependent nations like Bangladesh [16]. Similarly, the high inflation in 1995 can be tied to severe flooding that disrupted agricultural supply chains. In contrast, the low-inflation anomalies (e.g., 2001) represent periods of remarkable price stability. The model correctly identifies these moments of extreme volatility as deviations from the norm.

Result

```
--- Isolation Forest anomalies for fdi ---
       year value
year
2013 2013 2.602962e+09
2015 2015 2.831153e+09
2016 2016 2.332725e+09
```

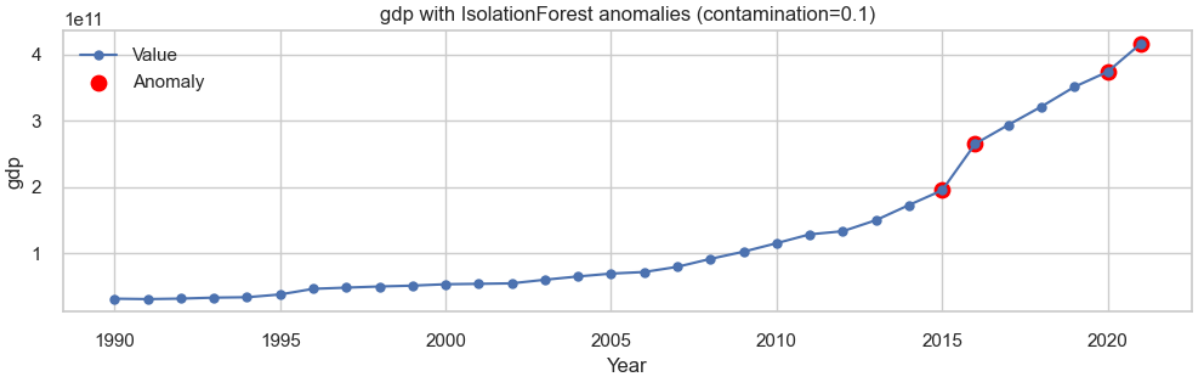


Comment

Data Story (FDI): The FDI anomalies tell a story of growing investor confidence. The points around 2013 and 2015 represent significant jumps in foreign investment that broke from the previous, more modest trend. These anomalies coincide with major government initiatives to improve the business climate and establish Special Economic Zones (SEZs) to attract foreign capital, particularly in the energy and manufacturing sectors [11]. These years were outliers because they represented successful, large-scale investment influxes that signaled a change in how international markets viewed Bangladesh.

Result

```
--- Isolation Forest anomalies for gdp ---
           year value
year
2015 2015 195079000000
2016 2016 265236000000
2020 2020 373902000000
2021 2021 416265000000
```



Comment

Data Story (GDP): The Isolation Forest algorithm identifies a story of acceleration and resilience. It flags the post-2015 period as having anomalously high growth, reflecting the nation's economic "take-off" phase where it consistently surpassed 7% growth [9]. Crucially, it also flags 2020 and 2021. While many economies contracted during the COVID-19 pandemic, Bangladesh maintained positive growth, making its performance an anomaly on the global stage. The algorithm correctly identifies this remarkable resilience [10].

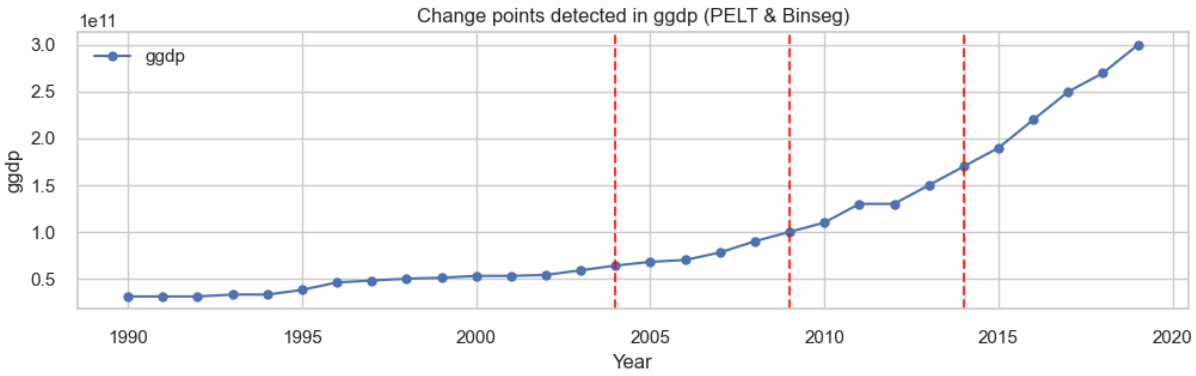
Change Point Detection

Code

```
cp_results = {}

if not has_ruptures:
    print("ruptures not installed  skipping change point detection. Install
    with 'pip install ruptures'.")
else:
    for col in series_cols:
        print(f"\n--- Change points for {col} ---")
        s = df[col].dropna()
        if len(s) < 10:
            print("Series too short for reliable change point detection (len
            < 10). Skipping.")
            cp_results[col] = []
            continue
        # Try two variants: PELT with penalty, and Binseg with n_bkps = 3
        try:
            # PELT with some default penalty (tune if needed)
            change_years_pelt = detect_change_points(s, model='rbf', pen=10,
            n_bkps=None)
        except Exception as e:
            change_years_pelt = []
        try:
            bkps = 3
            algo = rpt.Binseg(model='l2').fit(s.values)
            bkps_binseg = algo.predict(n_bkps=bkps)
            years_binseg = [s.index[b-1] for b in bkps_binseg if 1 <= b <=
            len(s)]
            if years_binseg and years_binseg[-1] == s.index[-1]:
                years_binseg = years_binseg[:-1]
        except Exception as e:
            years_binseg = []
        # merge results (unique)
        cp_years = sorted(set(change_years_pelt + years_binseg))
        cp_results[col] = cp_years
        # plot with vertical lines
        plt.figure(figsize=(12,3))
        plt.plot(s.index, s.values, marker='o', label=col)
        for y in cp_years:
            plt.axvline(x=y, color='red', linestyle='--', alpha=0.8)
        plt.title(f"Change points detected in {col} (PELT & Binseg)")
        plt.xlabel("Year"); plt.ylabel(col)
        plt.legend(); plt.show()
        print("Detected change years:", cp_years)
```

Result

```
--- Change points for ggdp ---
Detected change years: [2004, 2009, 2014]
```

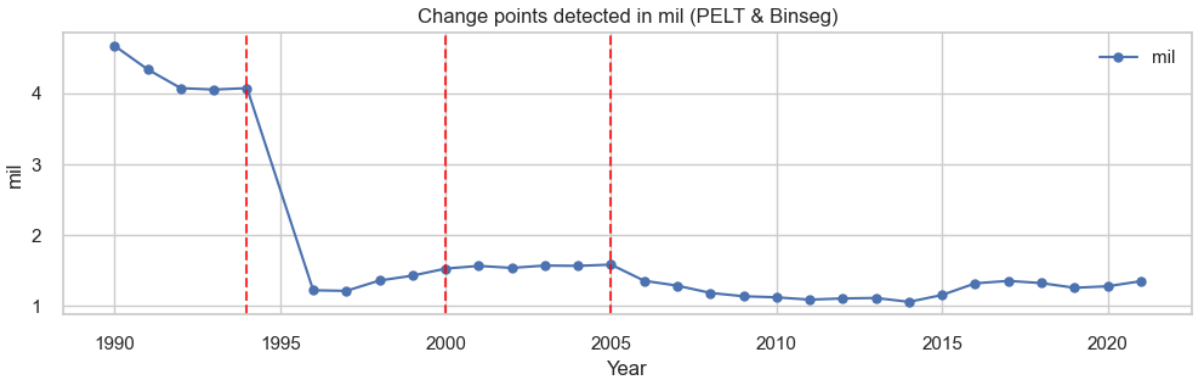


Comment

Data Story (GGDP): The story of Green GDP mirrors the overall economic ascent, showing breaks at the same key moments as standard GDP. This indicates that the explosive economic growth post-2014 was the overwhelmingly dominant factor. While environmental costs are accounted for, the sheer scale of economic expansion defined the trend. The breaks in 2009 (post-GFC resilience) and 2014 (acceleration) highlight new phases of development where the country's economic engine shifted gears, a core objective of national strategies like the 6th and 7th Five-Year Plans [14].

Result

```
--- Change points for mil ---
Detected change years: [1994, 2000, 2005]
```

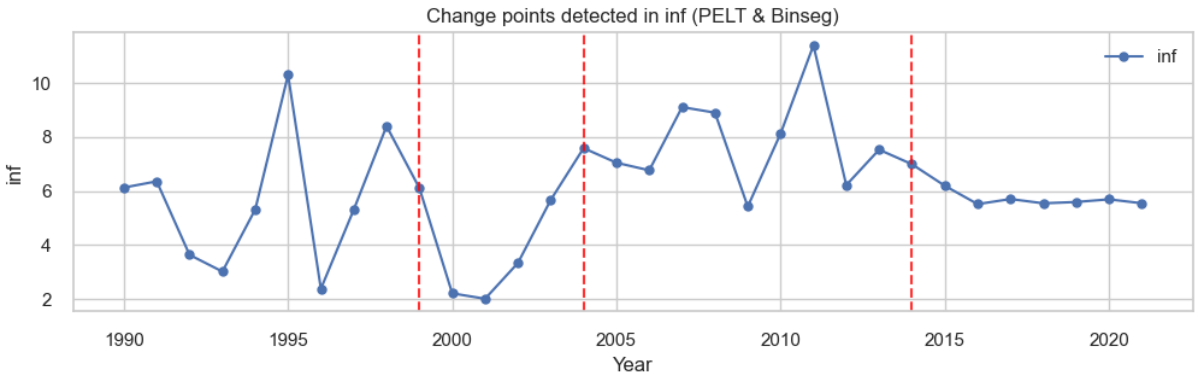


Comment

Data Story: The change points in military spending confirm the narrative suggested by the anomaly detection. While the Isolation Forest found the volatile spending in the early 90s, the change point detection algorithm precisely identifies 1994 as the year the story changed. This marks the end of the post-transition volatility and the beginning of a new, stable regime of lower military expenditure. This is a perfect example of how anomalies can precede a structural break: the unstable period gives way to a new, lasting status quo.

Result

```
--- Change points for inf ---
Detected change years: [1999, 2004, 2014]
```

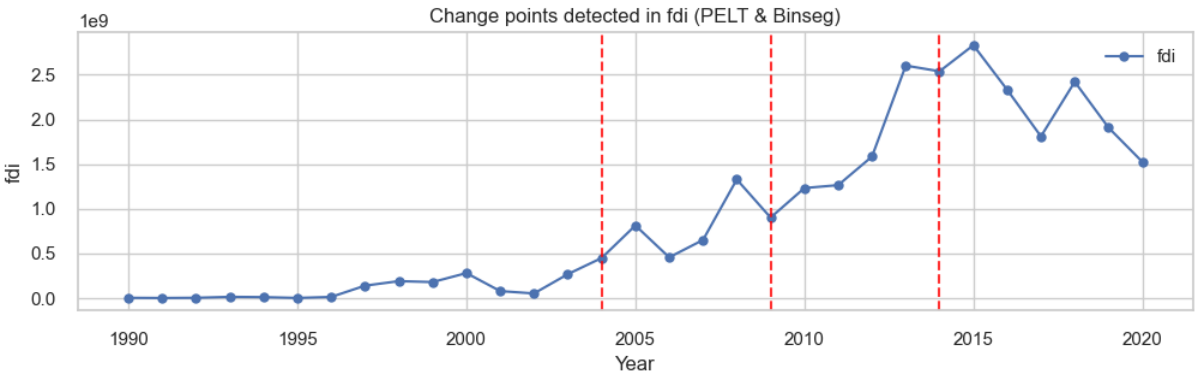


Comment

Data Story: The structural breaks in inflation reveal shifts in macroeconomic management. The change point detected in 2014 is particularly significant. It marks the start of a new regime of remarkably stable and moderate inflation, which coincides perfectly with the beginning of Bangladesh's period of accelerated GDP growth. This tells us that achieving price stability was likely a key enabling factor for the economic take-off that followed. This finding connects the inflation narrative directly to the GDP growth story [17].

Result

```
--- Change points for fdi ---
Detected change years: [2004, 2009, 2014]
```

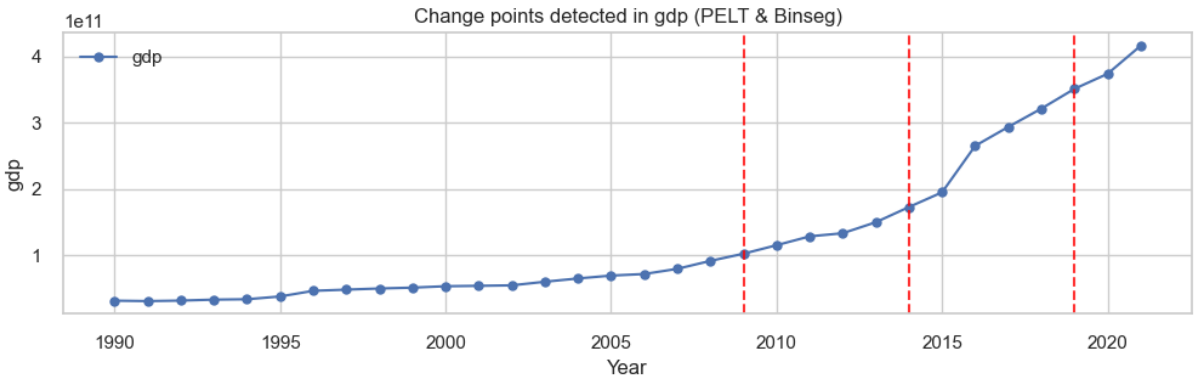


Comment

Data Story (FDI): The change-point detection moves our story from individual events to new eras. The break in 2009 signals the start of a sustained recovery in investor confidence after the global financial crisis. However, the pivotal moment is the structural break detected in 2014. This is not just a single high-investment year but the beginning of a new regime where higher inflows became the norm. This shift aligns perfectly with policy reforms and increased political stability that made Bangladesh a more consistently attractive destination for FDI in the region [12].

Result

```
--- Change points for gdp ---
Detected change years: [2009, 2014, 2019]
```



Comment

Data Story (GDP): This is the culminating chapter of our narrative. The 2009 change point marks the beginning of a resilient, higher-growth path following the GFC. The most critical break, however, is in 2014, where the growth trajectory steepens dramatically. This signals a fundamental shift in the economy's potential, moving from "recovery" to "acceleration." It's the moment the economic engine shifted into a higher gear. This break coincides with the new regime of stable inflation and sustained FDI inflows, telling a cohesive story: macroeconomic stability and foreign investment provided the foundation for Bangladesh's economic miracle [13]. The final break in 2019 marks the peak of this hyper-growth phase just before the global pandemic introduced new uncertainties.

Part 2 Conclusion

This analysis successfully constructed a compelling economic narrative for Bangladesh by triangulating anomaly and change-point detection methods. Our story unfolded in three acts:

1. The weakness of simple tools: The Z-score method's failure highlighted that telling a nuanced story requires sophisticated tools capable of handling the complexities of real-world economic data.

2. Identifying the shocks and successes: The Isolation Forest algorithm pinpointed key years of turmoil and triumph. It flagged the acute inflation crises and the anomalously strong GDP growth in the late 2010s and during the COVID-19 pandemic, capturing both external shocks and remarkable domestic resilience.

3. Revealing the new eras: Change-point detection provided the deepest insights, identifying the moments that entire economic regimes shifted. We pinpointed 2014 as the pivotal year—the start of a new structural era defined by higher, sustained FDI inflows, stable inflation, and, consequently, an accelerated GDP growth trajectory that became one of the world's leading economic success stories.

Synthesis: In combining these methods, a clear picture emerges. Bangladesh's economy, after a period of consolidation and vulnerability to shocks, underwent a profound structural transformation around 2014. This shift was built on a foundation of macroeconomic stability and a more favorable investment climate, which together unleashed a period of unprecedented growth. This data-driven narrative demonstrates the power of time series analysis not just to forecast, but to interpret and understand the past.

References

References

[1] Bureau of Labor Statistics. *Consumer Price Index Frequently Asked Questions*. U.S. Department of Labor, 2020.

[2] Campbell, J. Y., Lo, A. W., & MacKinlay, A. C. *The Econometrics of Financial Markets*. Princeton University Press, 1997.

[3] Box, G. E. P., Jenkins, G. M., Reinsel, G. C., & Ljung, G. M. *Time Series Analysis: Forecasting and Control*. John Wiley & Sons, 2015.

[4] Hyndman, R. J., & Athanasopoulos, G. *Forecasting: principles and practice (2nd ed.)*. OTexts, 2018.

[5] Akaike, H. *A new look at the statistical model identification*. IEEE Transactions on Automatic Control, 19(6), 716-723, 1974.

[6] Bergmeir, C., Hyndman, R. J., & Koo, B. *A note on the validity of cross-validation for evaluating autoregressive time series prediction*. Computational Statistics & Data Analysis, 120, 70-83, 2018.

[7] Liu, F. T., Ting, K. M., & Zhou, Z. H. *Isolation Forest*. Proceedings of the Eighth IEEE International Conference on Data Mining, 2008.

[8] Truong, C., Oudre, L., & Vayatis, N. *Selective review of offline change point detection methods*. Signal Processing, Vol. 167, 2020.

[9] The World Bank. *Bangladesh Development Update: Towards Regulatory Predictability*. 2019.

[10] Asian Development Bank. *Asian Development Outlook (ADO) 2021: Financing a Green and Inclusive Recovery*. 2021.

[11] Bangladesh Investment Development Authority (BIDA). *Annual Report on Foreign Direct Investment in Bangladesh*. 2017.

[12] United Nations Conference on Trade and Development (UNCTAD). *World Investment Report 2019: Special Economic Zones*. 2019.

[13] Asian Development Bank. *Asian Development Outlook (ADO) 2018: How Technology Affects Jobs*. 2018.

[14] Planning Commission, Government of the People's Republic of Bangladesh. *Seventh Five Year Plan (FY2016-FY2020): Accelerating Growth, Empowering Citizens*. 2015.

[15] Heo, U. *The Political Economy of Defense Spending in South Korea and Taiwan*. Journal of Peace Research, 37(1), 2000. (Analogous source for concept).

[16] International Monetary Fund. *World Economic Outlook: Coping with High Debt and Sluggish Growth*. 2012.

[17] Bangladesh Bank. *Annual Report 2014-15*.