**“이 스크립트는 학습된 ARIMA 모델 파라미터를 불러와 새로운 시계열 데이터에서 1-step ahead 잔차를 계산한 뒤, Z-score 기반으로 이상치를 판정하고 시각화하는 역할을 수행합니다.”**

# 4) 모델 초기화 및 가중치 로드

model = ARIMAModel().to(device)

# 저장한 pt 파일을 불러와, 모델에 매핑

state = torch.load(args.model, map\_location=device)

model.load\_state\_dict(state)

model.eval()

“4단계는 ARIMAModel() 객체를 생성하고,  
• .to(device)를 통해 GPU/CPU로 옮깁니다.  
이 모델은 common\_utils.py에 정의된 (p=2, d=1, q=2) 차수의 ARIMA 구조를 갖고 있습니다.  
이후 torch.load(args.model, map\_location=device)를 호출해 .pt 파일(학습된 파라미터)을 불러옵니다.  
• map\_location=device 옵션 덕분에, 원래 GPU 저장 모델이라도 현재 device 환경(GPU/CPU) 임에 맞춰 로드해 줍니다.  
model.load\_state\_dict(state)로 실제 파라미터(φ, θ, μ)를 모델에 복원하고,  
model.eval()로 추론 모드(평가 모드)로 전환합니다.  
→ 이러면 Dropout이나 BatchNorm 같은 학습 전용 레이어가 비활성화됩니다.”

# 5) 잔차 계산

with torch.no\_grad():

# forward()가 호출됨, 1단계앞 예측 후 실제값과의 차이가 잔차

residuals = model(series).squeeze(0)

“다음으로는 ‘잔차’를 계산하는 단계입니다.  
• with torch.no\_grad(): 블록 내부에서는 그래디언트(기울기)를 계산하지 않아서, 메모리 사용량을 줄이고 연산 속도를 높입니다.  
• model(series)를 호출하면, 자동으로 ARIMAModel의 forward() 메서드가 실행됩니다.  
– 내부에서 (p, d, q)=(2,1,2) 차례대로 차분하고, AR∙MA 계산을 거쳐 “예측값”을 만들고,  
– 실제 d차분된 값과 예측값 차이를 빼서 “잔차”를 구합니다.  
• 반환된 결과는 (1, T-p) 형태인데, .squeeze(0)으로 배치 차원(1)을 제거해 1차원 Tensor (T-p,)로 만듭니다.  
이렇게 해서 “각 시계열 시점별 잔차(residuals)”를 얻습니다.”

# 잔차의 총 길이, 너무 짧으면 계산 못함.

res\_len = residuals.numel()

if res\_len == 0:

print("!! 입력 시계열이 너무 짧아 잔차 계산 불가 !!")

sys.exit(1)

“만약 residuals Tensor 길이(numel())가 0이면,  
데이터가 너무 짧아서 ARIMA 모델이 1단계 앞 예측까지 못 하는 경우입니다.  
이럴 때는 “입력 시계열이 너무 짧아 잔차 계산 불가”라는 메시지를 출력한 뒤 종료합니다.”

# 6) Z-score로 이상치 판정

# 잔차를 정규분포의 표준정규(z)값 표준화

z\_scores = zscore(residuals.cpu().numpy(), nan\_policy="omit")

# 3시그마로 이상치 판정

anoms = abs(z\_scores) > 3

“이제 잔차를 기반으로 이상치를 판정합니다.  
• residuals.cpu().numpy()로 GPU Tensor를 CPU 메모리의 NumPy 배열로 변환한 뒤,  
zscore(..., nan\_policy="omit")를 호출해 표준정규분포(z-score) 값으로 바꿉니다.  
• z\_scores 배열에서 abs(z\_scores) > 3인 곳을 True로 표시하면, 3시그마(±3σ)를 벗어난 포인트를 이상치(anoms)로 분류합니다.  
→ 3시그마를 기준으로 삼는 이유는, 통계적으로 정규분포에서 약 99.7%가 ±3σ 구간에 있으므로,  
그 밖으로 벗어난 값은 극단치(이상치)일 확률이 매우 높기 때문입니다.”

# 7) 잔차 길이만큼 타임스탬프 정렬

aligned\_ts = ts\_index[-res\_len:]

“원래 ts\_index는 CSV 전체 길이에 해당하는 DatetimeIndex인데,  
ARIMA 모델이 d차분 후 앞쪽 p개 스텝은 잔차 계산에 쓰이지 않으므로,  
잔차 길이(res\_len) 만큼 뒤쪽 인덱스만 aligned\_ts = ts\_index[-res\_len:]로 잘라냅니다.  
이렇게 하면 aligned\_ts[i]와 residuals[i]가 정확히 대응하게 됩니다.”

# 8) 결과 출력

count = int(anoms.sum())

ratio = count / res\_len \* 100

print(f"전체 {res\_len} 스텝 중 이상치 {count}개 ({ratio:.2f}%)")

print("이상치 타임스탬프 (10개만):")

for t in aligned\_ts[anoms][:10]:

print(" \* ", t)

“이제 이상치 결과를 터미널에 출력합니다.  
• count = int(anoms.sum())는 이상치(True) 개수를 계산하고,  
• ratio = count / res\_len \* 100로 비율(퍼센트)까지 구합니다.  
출력 예시는 “전체 1000 스텝 중 이상치 25개 (2.50%)”처럼 표시됩니다.  
이후 “이상치 타임스탬프 (10개만):”라는 제목으로,  
for t in aligned\_ts[anoms][:10]를 돌면서 최대 10개의 이상치 시점을 한 줄씩 print(" \* ", t) 형태로 보여줍니다.”

# 9) 시각화

plt.figure(figsize=(12, 4))

plt.plot(aligned\_ts, residuals.cpu().numpy(), label="Residuals")

plt.scatter(aligned\_ts[anoms], residuals.cpu().numpy()[anoms],

color="red", label=f"Anomalies ({count})", zorder=5)

plt.title(f"Residuals & Anomalies for {args.tag}")

plt.xlabel("Timestamp")

plt.ylabel("Residual")

plt.legend()

plt.xticks(rotation=30)

plt.tight\_layout()

plt.show()

“마지막으로 Matplotlib을 이용해 잔차 시계열 곡선과 이상치 포인트를 함께 그립니다.

1. plt.figure(figsize=(12, 4))로 가로 12인치, 세로 4인치 크기의 빈 캔버스를 만듭니다.
2. plt.plot(aligned\_ts, residuals.cpu().numpy(), label="Residuals")는 x축에 시간(aligned\_ts), y축에 잔차값을 점·선 그래프로 그립니다.
3. plt.scatter(...)를 호출해, 이상치 인덱스(aligned\_ts[anoms])만 빨간 점(color="red")으로 강조합니다. zorder=5를 주면 다른 레이어보다 위에 표시됩니다.  
   – 레이블은 label=f"Anomalies ({count})"로 붙여, 범례에 이상치 개수가 함께 나타나도록 합니다.
4. plt.title, plt.xlabel, plt.ylabel로 각각 제목과 축 레이블을 설정합니다.
5. plt.legend()로 범례를 표시하고,
6. plt.xticks(rotation=30)로 x축의 시간 레이블을 30도 회전해 겹치지 않게 합니다.
7. plt.tight\_layout()은 레이아웃이 겹치지 않도록 간격을 자동 조정하며,
8. plt.show()를 호출하면 별도 창에 그래프가 나타납니다.  
   → 이렇게 하면 잔차 곡선을 한눈에 볼 수 있고, 이상치 시점은 붉은 점으로 강조되어 직관적으로 확인할 수 있습니다.”