산업용 센서 시계열 데이터 이상 탐지 프로젝트

2025년 딥러닝특론 과제 발표

금동환

목차

개요

프로젝트 목표

산업용 센서 시계열 데이터 이상 탐지

주요 기능

ARIMA 기반 이상 탐지 모델 구축

구성 요소

유틸리티, 학습, 예측 모듈 설계

SWaT 데이터를 이용해 이상 탐지 파이프라인을 구성

학습된 ARIMA 모델로 새로운 시계열의 이상치를 탐지

ARIMA 모델과 SWaT 데이터셋

ARIMA 모델

AR+MA의 조합으로 잔차 기반 이상 탐지에 적합한 모델

- p (AR 차수): 과거 관측치 반영
- d (차분 차수): 차분으로 안정화
- q (MA 차수): 과거 오차 반영

SWaT 데이터셋

산업 제어 시스템용 공개 데이터

- 타임스탬프와 다양한 센서 태그 포함
- 정상/이상 상태 레이블 제공
- 수많은 논문에서 검증된 데이터셋

코드 구조

예측 및 시각화

predict_arima.py

- 잔차 계산 및 표준화
- 이상치 판단 및 분류
- 결과 시각화

학습 스크립트

train_arima.py

- 정상 구간 데이터 학습
- MSE 손실 최소화
- 학습된 파라미터 저장

공통 유틸리티

common_utils.py

- 데이터 로드 및 전처리
- ARIMA 모델 정의
- 학습/예측 디바이스 선택

common_utils.py

load_csv() 함수

시계열 데이터 로드

데이터를 리샘플링 및 보간

ARIMAModel 클래스

PyTorch 기반 ARIMA(p,d,q) 모델 정의

get_device() 함수

CUDA 사용 가능 여부를 확인

GPU 목록을 출력하고 적절한 디바이스를 선택

common_utils.py

```
def load_csv(csv_path: str, tag: str) -> Tuple[torch.Tensor, pd.DatetimeIndex]:
    """
    CSV에서 지정 태그 열을 읽어
    1초 간격으로 리샘플링·보간한 시계열 Tensor와
    DatetimeIndex를 반환.
    """
    # 구분자 자동 감지
    with open(csv_path, 'r', encoding='utf-8') as f:
        hdr = f.readline()
    sep = ';' if hdr.count(';') > hdr.count(',') else ','

    # 모든 열을 문자열로 읽어 Timestamp와 tag 칼럼 추출
    df = pd.read_csv(csv_path, sep=sep, dtype=str, encoding='utf-8')
    cols = df.columns.tolist()
    ts_col = next(c for c in cols if "timestamp" in unicodedata.normalize("NFKC", c).lower())
    if tag not in cols:
        raise ValueError(f"'{tag}' 열을 찾을 수 없습니다. 헤더: {cols}")
    df = df[[ts_col, tag]]

    # Timestamp 전처리, 공백 제거 > datetime 변환 > 인덱스로 설정
    df[ts_col] = df[ts_col].str.strip()
    df[ts_col] = pd.to_datetime(df[ts_col], format=DATE_FMT, dayfirst=True)
    df.set_index(ts_col, inplace=True)

    # 중복 타임스탬프 제거 (중복된 타임스탬프는 첫번째 값만 유지)
    df = df[~df.index.duplicated(keep='first')]

    # sensor 값 문자열 > float32
    df[tag] = df[tag].str.replace(",", ".", regex=False).astype("float32")

    # 결측치 처리, 1초 해상도 재샘플링 + 선형 보간
    series = df[tag].asfreq("1s").interpolate("linear")

    # 텐서 변환 및 인덱스 반환
    tensor = torch.tensor(series.values, dtype=torch.float32)
    return tensor, series.index


def get_device() -> tuple[torch.device, int]:
    """
    CUDA 사용 가능 시 ('cuda', GPU 개수), 아니면 ('cpu', 0) 반환
    """
    print("CUDA available:", torch.cuda.is_available())
    if torch.cuda.is_available():
        count = torch.cuda.device_count()
        [print(f"[{idx}] {torch.cuda.get_device_name(idx)}") for idx in range(count)]
        return torch.device("cuda"), torch.cuda.device_count()
    else:
        print("GPU 미감지: CPU로 실행")
        return torch.device("cpu"), 0
```

```
class ARIMAModel(nn.Module):
    def __init__(self, p: int = 2, d: int = 1, q: int = 2):
        super().__init__()
        self.p, self.d, self.q = p, d, q
        # 학습 가능한 파라미터
        self.phi = nn.Parameter(torch.zeros(p))
        self.theta = nn.Parameter(torch.zeros(q))
        self.mu = nn.Parameter(torch.zeros(1))

    def forward(self, y: torch.Tensor) -> torch.Tensor:
        # y가 1D인 경우 배치 차원 추가
        if y.dim() == 1:
            y = y.unsqueeze(0)

        # 배치 크기(B)와 시퀀스 길이(Tseq)를 가져옴
        B, Tseq = y.shape

        # 모델의 차수 파라미터
        d, p, q = self.d, self.p, self.q

        # 실제 예측 가능한 시퀀스 길이
        T = Tseq - d

        # 잔차를 저장할 버퍼
        eps = y.new_zeros(B, T + q)

        # 차분된 시계열
        yd = y[:, d:] - y[:, :-d]

        # t 에 따라 AR+MA 계산, 잔차 저장
        for t in range(p, T):
            # ar, 과거 p개 시점의 관측치에 대한 가중합
            ar = (self.phi * torch.flip(y[:, d + t - p:d + t], dims=[1])).sum(dim=1)

            # ma, 과거 q개 시점의 잔차에 대한 가중합
            ma = (self.theta * torch.flip(eps[:, t - q:t], dims=[1])).sum(dim=1)

            # 예측값 계산
            pred = self.mu + ar + ma

            # 잔차 계산
            eps[:, t] = yd[:, t] - pred
        # 초기 p 스텝 제외 후 반환
        return eps[:, p:]
```

train_arima.py

데이터 로드

정상 구간 CSV를 읽어 시계열 Tensor를 생성

시계열 데이터를 훈련 가능한 형태로 전처리

모델 초기화

ARIMAModel 객체를 생성

GPU/CPU로 모델을 이동하고 DataParallel로 다중 GPU를 지원

학습 루프

MSE 손실을 최소화하도록 파라미터를 학습

학습 과정 시간을 로깅하여 성능을 모니터링

모델 저장

최종 파라미터를 저장

train_arima.py

```
def main():
    # 1) 인자 파싱
    parser = argparse.ArgumentParser()
    parser.add_argument("--csv", default="train.csv", help="정상 구간 CSV 경로")
    parser.add_argument("--tag", default="LIT101", help="센서 태그명")
    parser.add_argument("--epochs", type=int, default=100, help="전체 Epoch 수")
    parser.add_argument("--lr", type=float, default=1e-2, help="학습률")
    args = parser.parse_args()

    # 2) 디바이스 설정
    device, n_gpu = get_device()
    print(f"학습 디바이스: {device}  (GPU 개수={n_gpu})")

    # 3) 데이터 로드
    try:
        series, _ = load_csv(args.csv, args.tag)
    except Exception as e:
        print(f"데이터 로드 실패: {e}")
        sys.exit(1)
    else:
        # 성공 시에만
        y = series.unsqueeze(0).to(device)  # (1, T)
        if n_gpu > 1:
            y = y.repeat(n_gpu, 1)  # (B=n_gpu, T)

    # 4) 모델 생성
    base_model = ARIMAModel().to(device)
    model = DataParallel(base_model) if n_gpu > 1 else base_model
    optimizer = torch.optim.Adam(model.parameters(), lr=args.lr)

    # 5) 학습 수행
    train_loop(model, y, args.epochs, optimizer)

    # 6) 모델 저장
    Path("models").mkdir(exist_ok=True)
    torch.save(base_model.state_dict(), f"models/arima_{args.tag}.pt")
    print(f"모델 저장 완료: models/arima_{args.tag}.pt")
```

```
def train_loop(model, data, epochs, optimizer):
    """
    • model     :  ARIMA 모델
    • data      : 입력 텐서
    • epochs    : 학습 반복 횟수
    • optimizer: 옵티마이저
    """
    # 로깅을 위한 타이머, 시간 객체
    start_wall = datetime.now()
    start_perf = time.perf_counter()
    print(f"학습 시작: {start_wall:%Y-%m-%d %H:%M:%S}")
    prev_perf = start_perf

    for epoch in trange(1, epochs + 1, desc="Epochs"):
        # 기울기 초기화
        optimizer.zero_grad()
        # forward() 호출 + 손실 계산  # 평균제곱오차
        loss = torch.mean((model(data)) ** 2)
        # 역전파
        loss.backward()
        # 파라미터 업데이트
        optimizer.step()

        # 로깅을 위한 시간 계산
        now_perf = time.perf_counter()
        delta_ms = (now_perf - prev_perf) * 1000
        prev_perf = now_perf

        # 상태 로깅
        print(f"[{epoch:03d}/{epochs}] loss={loss.item():.6f} | Δt={delta_ms:.1f} ms")

    total_time = time.perf_counter() - start_perf
    print(f"학습 종료: {datetime.now():%Y-%m-%d %H:%M:%S} (총 {total_time:.2f}s)")
    return total_time
```

predict_arima.py

모델 로드

학습된 ARIMA 파라미터(.pt)를 로드

잔차 계산

평가용 CSV에서 시계열 Tensor를 생성
잔차를 계산

결과 시각화

잔차 시계열과 이상치를 시각화

이상치 판정

잔차를 Z-score 표준화하고,
Z>3인 포인트를 이상치로 분류

predict_arima.py

```
def main():
    # 1) 인자 파싱
    parser = argparse.ArgumentParser()
    parser.add_argument("--csv", default="test.csv", help="평가용 CSV 파일 경로")
    parser.add_argument("--tag", default="LIT101", help="예측할 센서 태그명")
    parser.add_argument("--model", default="models/arima_LIT101.pt", help="학습된 파라미터(.pt) 경로")
    args = parser.parse_args()

    # 2) 디바이스 확인
    device, _ = get_device()
    print(f"추론 디바이스: {device}")

    # 3) 테스트 데이터 로드 및 전처리
    try:
        series, ts_index = load_csv(args.csv, args.tag)
    except Exception as e:
        print(f"데이터 로드 실패: {e}")
        sys.exit(1)
    else:
        # 성공 시에만
        series = series.to(device)

    # 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()

    # 5) 잔차 계산
    with torch.no_grad():
        # forward()가 호출됨, 1단계앞 예측 후 실제값과의 차이가 잔차
        residuals = model(series).squeeze(0)

    # 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)

    # 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

    # 7) 잔차 길이만큼 타임스탬프 정렬
    aligned_ts = ts_index[-res_len:]

    # 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)

    # 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()
```

진행과정 (계획)

데이터 준비

SWaT 데이터셋 수집 및 전처리

코드 구현

3개의 주요 모듈 개발 및 테스트

모델 학습

정상 데이터로 ARIMA 모델 파라미터 학습

손실 함수 최소화 및 학습 진행 확인

이상 탐지

테스트 데이터에서 이상치 탐지

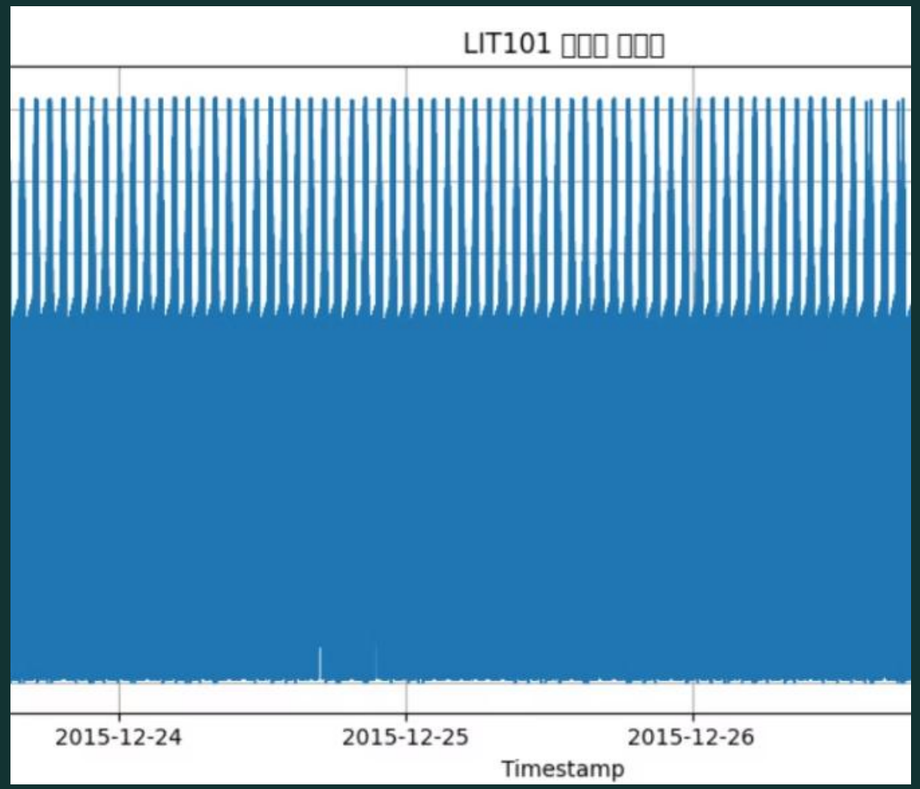
잔차 분석으로 이상 패턴 식별

진행과정 (문제점)

데이터 전처리 문제

SWaT 데이터셋의 불규칙한 타임스탬프와 결측치

리샘플링 시 시간 간격 불일치



메모리 및 성능 이슈

대용량 시계열 데이터 처리 시 메모리 부족 문제 발생

1. 데이터를 배치 단위로 분할
2. 시스템 프리징

진행과정 (해결시도)

배치 단위로 분할 처리

일정 블록으로 순차 처리

메모리 최적화

torch.cuda.empty_cache()

Torch_Arima 모델

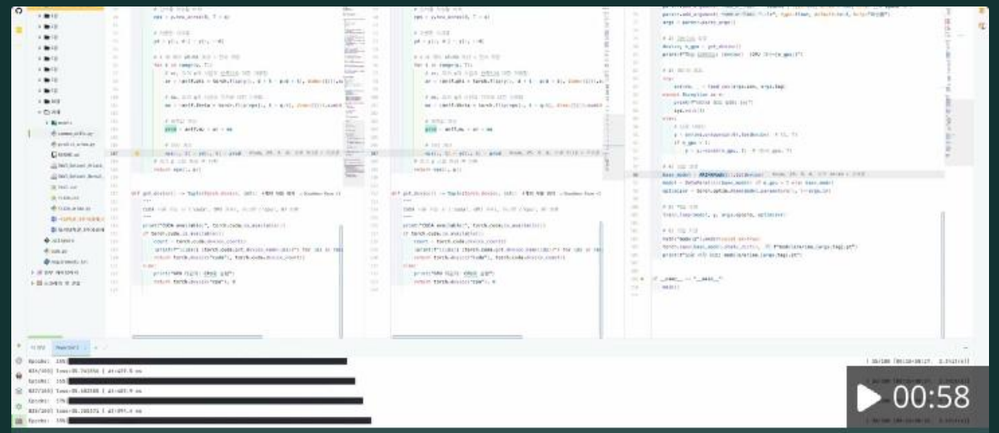
PyTorch 로 구현된 모델로 변경

합성 데이터

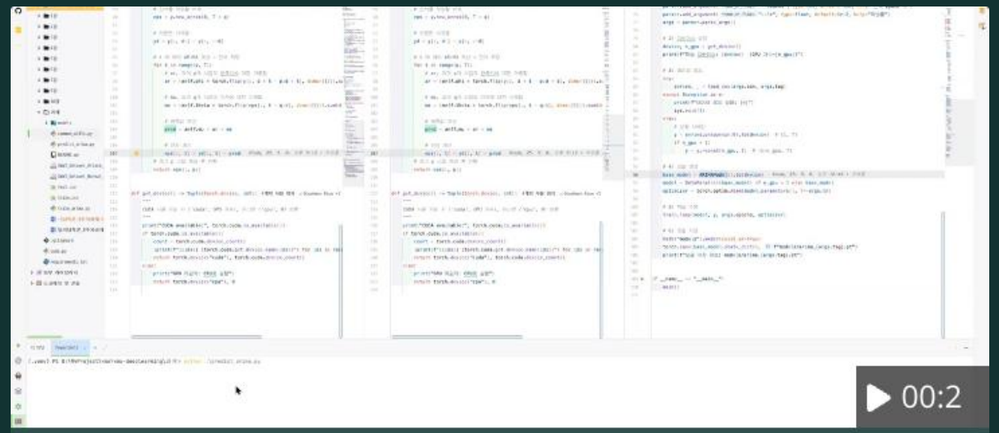
ChatGPT 활용

실행화면

학습



예측



발전방향

실제 데이터 반영

SWaT 데이터 사용

이상징후 예측

이상치 판단 > 이상징후 예측

잔존수명 예측

센서의 잔존수명 예측, 다음 정비 시기 예측

참고자료

ArimaModel

GitHub : torch_arima

SWaT

Secure Water Treatment

Thank you for your attention