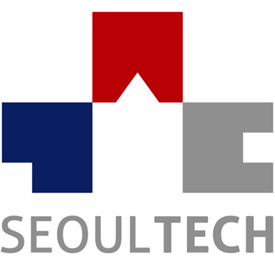
**기능 상세 설계서**

딥러닝 기반 다변량 스팀 사용 이상 감지 및 영향변수의 원인 분석 기능 제작

2023. 11. 28.



서울과학기술대학교 데이터사이언스학과

연구책임자 심재웅

목차

[1. 데이터 이해 4](#_Toc152246879)

[2. 데이터 전처리 5](#_Toc152246880)

[2.1. ‘jr\_progress’ 변수 변환 및 생성 5](#_Toc152246881)

[2.2. 학습 데이터 구축 5](#_Toc152246882)

[2.3. 학습 및 테스트 데이터셋 분할 및 스케일링 7](#_Toc152246883)

[2.4. 각 모델에 적합한 데이터 구조 변경 8](#_Toc152246884)

[2.4.1. Random Forest 8](#_Toc152246885)

[2.4.2. 1D-CNN 9](#_Toc152246886)

[2.4.3. LSTM 10](#_Toc152246887)

[2.4.4. DARNN 10](#_Toc152246888)

[3. 모델 구조 12](#_Toc152246889)

[3.1. Random Forest 12](#_Toc152246890)

[3.1.1. 모델 정의 12](#_Toc152246891)

[3.1.2. 학습 과정 12](#_Toc152246892)

[3.1.3. 성능 평가 12](#_Toc152246893)

[3.2. Convolutional Neural Network model 13](#_Toc152246894)

[3.2.1. 모델 정의 13](#_Toc152246895)

[3.2.2. 학습 및 검증 과정 및 모델 저장 14](#_Toc152246896)

[3.2.3. 테스트 및 성능 평가 15](#_Toc152246897)

[3.3. Long Short-Term Memory model 16](#_Toc152246898)

[3.3.1. 모델 정의 16](#_Toc152246899)

[3.3.2. 학습 및 검증 과정 및 모델 저장 18](#_Toc152246900)

[3.3.3. 테스트 및 성능 평가 19](#_Toc152246901)

[3.4. Dual-Attention model 20](#_Toc152246902)

[3.4.1. 모델 정의 20](#_Toc152246903)

[3.4.2. 모델 학습 과정 25](#_Toc152246904)

[4. 학습 파라미터 26](#_Toc152246905)

[4.1. Random forest 26](#_Toc152246906)

[4.2. 1D CNN 26](#_Toc152246907)

[4.3. LSTM 26](#_Toc152246908)

[4.4. DARNN 26](#_Toc152246909)

[5. 실행 결과 설명 27](#_Toc152246910)

[5.1. 예측 결과 27](#_Toc152246911)

[5.1.1. 결정계수(R2) 27](#_Toc152246912)

[5.1.2. 평균 제곱 오차(MSE) 27](#_Toc152246913)

[5.1.3. 그래프 해석 방법 28](#_Toc152246914)

[5.2. 원인 인자 도출 29](#_Toc152246915)

[5.2.1. Feature Importance 29](#_Toc152246916)

[5.2.2. SHAP 30](#_Toc152246917)

[5.2.3. Attention Map 32](#_Toc152246918)

# 데이터 이해

* 이용 데이터: df\_ext(2023-04-01~2023-08-31,51250385)\_2023-10-17 10-58-30 -seoultec.xlsx (제품 1종에 대한 5달간 센서 데이터)
* 데이터 수집 기간: 2023-04-05 14:59:00 ~ 2023-08-27 03:00:00 (분)

|  |  |  |  |
| --- | --- | --- | --- |
| Column | 의미 | 데이터 타입 | 개수 |
| date | 데이터 수집 시각(단위 : 분) | datetime64[ns] | 66106 |
| tg | Sensor (38개) | float64 |
| output | 제품 생산량 계산값 | float64 |
| ei | 원단위 계산값  (=스팀 사용량/output) | float64 |
| sstable | 원단위 상태 분석값  ~~0 : 좋음~~  1 : 적당  ~~2 : 나쁨~~ | int64 |
| jr | 단위 공정값 / 제품 생산 주기 (생산품 번호) | int64 |
| shift | 작업팀 구분값 | int64 |
| wclass | 작업팀 구분값 | int64 |
| stop | 공정 분석값  0 : 가동  ~~1 : 중지 이벤트 발생~~  ~~2 : 중지 복구~~ | int64 |

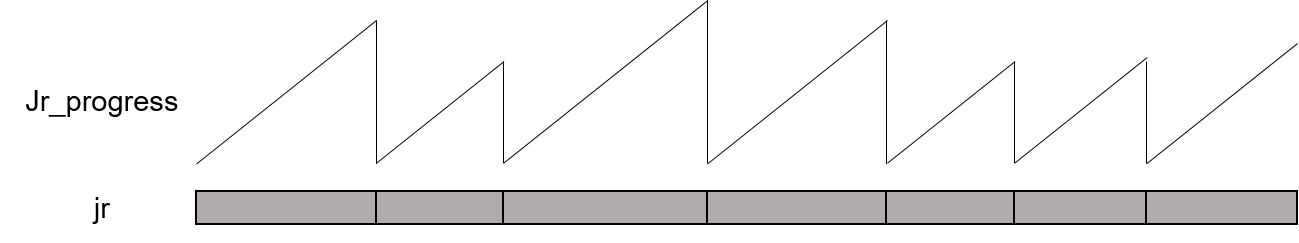
* 시계열 예측 모델 개요
  + 괴거 시점의 시계열 센서값을 통해 미래 시점의 ei값을 추론하는 regression 모델
    - Window: 모델의 input으로 사용될 시계열 센서 데이터의 구간 길이
    - Time gap: 현 시점과 예측 대상이 되는 미래 시점 사이의 구간 길이

스크린샷, 라인, 도표, 평행이(가) 표시된 사진

자동 생성된 설명

# 데이터 전처리

## ‘jr\_progress’ 변수 변환 및 생성



# 각 제품의 생산 경과 시간을 표현하기 위한 변수 ‘jr\_progress’ 추가

df['date'] = pd.to\_datetime(df['date'])

category\_start\_end\_dates = df.groupby('jr')['date'].agg(['min', 'max'])

def calculate\_progress(row):

    start\_date = category\_start\_end\_dates.loc[row['jr'], 'min']

    end\_date = category\_start\_end\_dates.loc[row['jr'], 'max']

    current\_date = row['date']

    # 분모가 0이 되는 경우를 방지합니다.

    if start\_date != end\_date:

        elapsed\_time = (current\_date - start\_date).total\_seconds() / 60.0

        progress = elapsed\_time

        return progress

    else:

        # 카테고리 원소가 단 하루만 존재하는 경우 진행률은 1

        return 1.0

# 'jr\_progress' 컬럼을 계산하여 데이터프레임에 추가

df['jr\_progress'] = df.apply(calculate\_progress, axis=1)

## 활용 변수 정리

* 분석에 사용하지 않을 변수 삭제

# 필요없는 컬럼 삭제

df = df.drop(['sstable', 'jr', 'output', 'stop', 'shift', 'wclass'], axis=1)

* 모델의 input 변수 (X)와 output 변수 (y) 정리
  + Random Forest, 1D-CNN, LSTM

이전의 y 정보를 미래 시점 예측을 위한 학습 데이터로 사용

도표, 라인, 스크린샷, 그래프이(가) 표시된 사진

자동 생성된 설명

def is\_continuous(series):

    return all((series.shift(-1) - series).dropna() == pd.Timedelta(minutes=1))

window\_size = 20 #윈도우 사이즈 설정

time\_gap = 10 #시간 간격 설정

X = []

y = []

#윈도우 사이즈와 지연 예측 시간 고려하여 시계열 데이터로 변환

for i in range(len(df) - window\_size + 1 - time\_gap):

    subset = df.iloc[i:i+window\_size + time\_gap]

    if is\_continuous(subset['date']):

        X.append(subset.iloc[:window\_size])

        y.append(subset.iloc[-1]['ei'])

* + DARNN

이전의 센서값과 보다 이전의 y 정보를 미래 시점 예측을 위한 학습 데이터로 사용

도표, 라인, 그래프, 스크린샷이(가) 표시된 사진

자동 생성된 설명

def is\_continuous(series):

    return all((series.shift(-1) - series).dropna() == pd.Timedelta(minutes=1))

window\_size = 20

time\_gap = 10

X = []

y = []

dec = []

#윈도우 사이즈와 지연 예측 시간 고려하여 시계열 데이터로 변환

for i in range(len(df) - window\_size + 1 - time\_gap):

    subset = df.iloc[i:i+window\_size + time\_gap]

    if is\_continuous(subset['date']):

        X.append(subset.iloc[:window\_size].drop(columns=['ei']))

        y.append(subset.iloc[-1]['ei'])

        dec.append(subset.iloc[:window\_size-1]['ei'])

## 학습 및 테스트 데이터셋 분할 및 스케일링

* 시간대가 끊긴 지점을 기준으로 훈련, 검증 및 테스트 데이터셋 분할

|  |  |  |
| --- | --- | --- |
| 모델 | 머신러닝 모델  (Random forest) | 딥러닝 모델  (1D-CNN, LSTM, DARNN) |
| 데이터 분할 | train / test | train / val / test |
| 분할 비율 | 0.8 / 0.2 | 0.6 / 0.2 / 0.2 |
| 시퀀스 개수 | 49800 / 12440 | 37340 / 12460 / 12440 |

def find\_discontinuous\_points(series):

    discontinuous\_points = []

    for i in range(1, len(series)):

        if series.iloc[i] - series.iloc[i-1] != pd.Timedelta(minutes=1):

            discontinuous\_points.append(i)

    return discontinuous\_points

# 시간이 끊긴 지점을 기준으로 데이터 나누기

break\_points = find\_discontinuous\_points(pd.concat([x['date'] for x in X]))

# 60% 비율에 가장 근접한 끊긴 지점 찾기

val\_ratio = 0.6

best\_point = None

best\_ratio = float('inf')

total\_length = len(X)

for point in break\_points:

    ratio = point / total\_length

    if abs(ratio - val\_ratio) < best\_ratio:

        best\_ratio = abs(ratio - val\_ratio)

        val\_best\_point = point

# 80% 비율에 가장 근접한 끊긴 지점 찾기

best\_point = None

best\_ratio = float('inf')

total\_length = len(X)

test\_ratio = 0.8

for point in break\_points:

    ratio = point / total\_length

    if abs(ratio - test\_ratio) < best\_ratio:

        best\_ratio = abs(ratio - test\_ratio)

        test\_best\_point = point

# 찾은 지점을 기준으로 데이터 나누기

if val\_ratio is not None:

    X\_train = X[:val\_best\_point]

    y\_train = y[:val\_best\_point]

    X\_val = X[val\_best\_point:test\_best\_point]

    y\_val = y[val\_best\_point:test\_best\_point]

    X\_test = X[test\_best\_point:]

    y\_test = y[test\_best\_point:]

X\_train = [x.drop(columns=['date']) for x in X\_train]

X\_val = [x.drop(columns=['date']) for x in X\_val]

X\_test = [x.drop(columns=['date']) for x in X\_test]

X\_train\_array = [x.values for x in X\_train]

X\_val\_array = [x.values for x in X\_val]

X\_test\_array = [x.values for x in X\_test]

* 분할한 데이터셋에 대해 MinMax스케일링 진행

scaler = MinMaxScaler().fit(np.concatenate(X\_train\_array, axis=0))

# X\_train과 X\_test 스케일링

X\_train\_scaled = [scaler.transform(x) for x in X\_train\_array]

X\_val\_scaled = [scaler.transform(x) for x in X\_val\_array]

X\_test\_scaled = [scaler.transform(x) for x in X\_test\_array]

## 각 모델에 적합한 데이터 구조 변경

### Random Forest

* 시간 간격 및 윈도우 사이즈에 따른 윈도우 롤링 후 시계열 정보가 들어있는 데이터셋 구성
  + 주어진 시퀀스 데이터(X)를 지정된 타임스탬프만큼 구간으로 나누고, 각 구간의 데이터를 평탄화하여 하나의 인스턴스로 구성
  + 즉, 특정 타임스탬프만큼 자른 시퀀스 3차원 데이터([인스턴스개수, 윈도우사이즈, 컬럼개수])에서 각 타임스탬프의 변수를 학습에 적용하기 위해 2차원 데이터([인스턴스개수, 윈도우사이즈\*컬럼개수])로 평탄화

def rolling\_window\_sequences\_and\_names(X, window\_size, original\_columns):

    X\_rolled = []

    for sequence in X:

        # Rolling the array and renaming columns

        for start in range(0, sequence.shape[0] - window\_size + 1):

            window = sequence[start:start+window\_size]

            window\_flattened = window.flatten()

            new\_columns = ['t{}\_{}'.format(i, col) for i in range(window\_size) for col in original\_columns]

            rolled\_df = pd.DataFrame([window\_flattened], columns=new\_columns)

            X\_rolled.append(rolled\_df)

    return pd.concat(X\_rolled, axis=0)

original\_columns = X\_train[0].columns

# Rolling windows for X\_train\_scaled and X\_test\_scaled with new column names

X\_train\_df = rolling\_window\_sequences\_and\_names(X\_train\_scaled, window\_size, original\_columns)

X\_test\_df = rolling\_window\_sequences\_and\_names(X\_test\_scaled, window\_size, original\_columns)

### 1D-CNN

* 모델의 인풋 구조에 맞게 데이터 변형
  + 1D-CNN : [batch size, feature, window size]

#파이토치 TensorDataset 생성

X\_train\_tensor = torch.tensor(X\_train\_scaled, dtype=torch.float32)

y\_train\_tensor = torch.tensor(y\_train, dtype=torch.float32)

X\_val\_tensor = torch.tensor(X\_val\_scaled, dtype=torch.float32)

y\_val\_tensor = torch.tensor(y\_val, dtype=torch.float32)

X\_test\_tensor = torch.tensor(X\_test\_scaled, dtype=torch.float32)

y\_test\_tensor = torch.tensor(y\_test, dtype=torch.float32)

#1D-CNN이라 (instance, feature, window) 순으로 정렬

train\_dataset = TensorDataset(X\_train\_tensor.permute(0,2, 1), y\_train\_tensor)

val\_dataset = TensorDataset(X\_val\_tensor.permute(0,2, 1), y\_val\_tensor)

test\_dataset = TensorDataset(X\_test\_tensor.permute(0,2, 1), y\_test\_tensor)

#파이토치 데이터로더

train\_dataloader = DataLoader(train\_dataset, batch\_size=64, shuffle=True)

val\_dataloader = DataLoader(val\_dataset, batch\_size=64, shuffle=False)

test\_dataloader = DataLoader(test\_dataset, batch\_size=2, shuffle=False)

dataiter = iter(test\_dataloader)

images, labels = dataiter.next()

### LSTM

* 모델의 인풋 구조에 맞게 데이터 변형
  + LSTM: [batch size, window size, feature]

#numpy to pytorch tensor

X\_train\_tensor = torch.tensor(X\_train\_scaled, dtype=torch.float32)

y\_train\_tensor = torch.tensor(y\_train, dtype=torch.float32)

X\_val\_tensor = torch.tensor(X\_val\_scaled, dtype=torch.float32)

y\_val\_tensor = torch.tensor(y\_val, dtype=torch.float32)

X\_test\_tensor = torch.tensor(X\_test\_scaled, dtype=torch.float32)

y\_test\_tensor = torch.tensor(y\_test, dtype=torch.float32)

# TensorDataset 생성

train\_dataset = TensorDataset(X\_train\_tensor, y\_train\_tensor)

val\_dataset = TensorDataset(X\_val\_tensor, y\_val\_tensor)

test\_dataset = TensorDataset(X\_test\_tensor, y\_test\_tensor)

#Tensor Dataloader 생성

train\_dataloader = DataLoader(train\_dataset, batch\_size=128, shuffle=True)

val\_dataloader = DataLoader(val\_dataset, batch\_size=128, shuffle=True)

test\_dataloader = DataLoader(test\_dataset, batch\_size=6341, shuffle=False)

dataiter = iter(train\_dataloader)

x\_sample, y\_sample = dataiter.next()

### DARNN

* 모델의 인풋 구조에 맞게 데이터 변형
  + DARNN: ([batch\_enc, batch\_dec], batch\_target)
    - batch\_enc : [batch size, window size, feature]
    - batch\_dec : [batch size, window size -1, 1]
    - batch\_target : [batch size, 1]

class Dataloader(Sequence):

    def \_\_init\_\_(self, enc\_dataset, dec\_dataset, target\_dataset, batch\_size, shuffle=False):

        self.enc, self.dec, self.target = enc\_dataset, dec\_dataset, target\_dataset

        self.batch\_size = batch\_size

        self.shuffle=shuffle

        self.on\_epoch\_end()

    def \_\_len\_\_(self):

        return math.ceil(len(self.enc) / self.batch\_size)

    def \_\_getitem\_\_(self, idx):

        indices = self.indices[idx\*self.batch\_size:(idx+1)\*self.batch\_size]

        batch\_enc = np.array([self.enc[i] for i in indices])

        batch\_dec = np.array([self.dec[i] for i in indices])

        batch\_target = np.array([self.target[i] for i in indices])

        return [batch\_enc, np.expand\_dims(batch\_dec, axis=-1)], batch\_target

train\_loader = Dataloader(enc\_dataset\_train, dec\_dataset\_train, target\_train, batch\_size, shuffle=True)

valid\_loader = Dataloader(enc\_dataset\_val, dec\_dataset\_val, target\_val, batch\_size)

test\_loader = Dataloader(enc\_dataset\_test, dec\_dataset\_test, target\_test, batch\_size)

train\_ds = (

    tf.data.Dataset.from\_tensor\_slices(

        (enc\_dataset\_train, np.expand\_dims(dec\_dataset\_train, axis=-1), target\_train)

    )

    .batch(batch\_size)

    .shuffle(buffer\_size=len(enc\_dataset\_train))

    .prefetch(tf.data.experimental.AUTOTUNE)

)

val\_ds = (

    tf.data.Dataset.from\_tensor\_slices(

        (enc\_dataset\_val, np.expand\_dims(dec\_dataset\_val, axis=-1), target\_val)

    )

    .batch(batch\_size)

    .shuffle(buffer\_size=len(enc\_dataset\_val))

    .prefetch(tf.data.experimental.AUTOTUNE)

)

# 모델 구조

## Random Forest

### 모델 정의

* 랜덤 포레스트란 결정나무를 기본 모델로 사용하는 앙상블 방법

### 학습 과정

* RandomForestRegressor를 사용하여 랜덤 포레스트 회귀 모델 생성
* N\_estimators는 이 모델이 100개의 decision trees를 사용하여 학습하고 예측을 수행하는 것을 의미
* 독립변수 X\_train\_df와 종속변수 y\_train의 관계를 학습
* 학습된 모델을 사용하여 X\_test\_df에 대한 예측을 수행, 결과는 y\_pred에 저장

# 랜덤 포레스트 모델 생성 및 학습

rf\_regressor = RandomForestRegressor(n\_estimators=100) # 하이퍼파라미터

rf\_regressor.fit(X\_train\_df, y\_train)

# 예측

y\_pred = rf\_regressor.predict(X\_test\_df)

### 성능 평가

* 평균 제곱 오차(MSE), 평균 절대 오차(MAE), R² 점수를 계산

# 성능 평가

mse = mean\_squared\_error(y\_test, y\_pred)

print(f"Mean Squared Error(Randomforest): {mse}")

r2 = r2\_score(y\_test, y\_pred)

print(f"R^2 Score(Randomforest): {r2}")

# 성능 지표를 텍스트 파일로 저장

with open( folder\_path + 'model\_performance.txt', 'w') as file:

    file.write(f'r2: {r2}\n')

    file.write(f'mse: {mse}\n')

## Convolutional Neural Network model

### 모델 정의

* 기존 CNN 모델에서 텍스트 및 시계열 데이터를 처리하기 위해 인풋 및 커널의 차원을 축소한 모델
* ‘CNN1DRegressor’ 클래스는 PyTorch의 nn.Module을 상속받아 정의
* 1차원 컨볼루션 레이어와 선형(완전 연결) 레이어를 순차적으로 구성
* 활성화 함수로는 Tanh가 사용되며, 몇몇 레이어 다음에는 드롭아웃이 적용되어 과적합을 방지
* 각 컨볼루션 레이어를 거치면서 채널 수가 다음과 같이 변함(64 🡪 128 🡪 256 🡪 512)
* 압축된 feature map에 FC layer를 적용해 최종적으로 ei값 예측

class CNN1DRegressor(nn.Module):

    def \_\_init\_\_(self):

        super(CNN1DRegressor, self).\_\_init\_\_()

        self.layers = nn.Sequential(

            # input: [batch\_size, 40, 20]

            nn.Conv1d(40, 64, kernel\_size=3),  # output: [batch\_size, 64, 18]

            nn.Tanh(),

            nn.Conv1d(64, 128, kernel\_size=3),  # output: [batch\_size, 128, 16]

            nn.Tanh(),

            nn.Conv1d(128, 256, kernel\_size=3),  # output: [batch\_size, 256, 14]

            nn.Tanh(),

            nn.Conv1d(256, 512, kernel\_size=3),  # output: [batch\_size, 512, 12]

            nn.Tanh(),

            nn.Conv1d(512, 1024, kernel\_size=3),  # output: [batch\_size, 512, 12]

            nn.Tanh(),

            nn.Flatten(),  # output: [batch\_size, 1024\*10]

            nn.Linear(1024\*10, 128),

            nn.Tanh(),

            nn.Dropout(0.4),

            nn.Linear(128, 32),

            nn.Tanh(),

            nn.Dropout(0.4),

            nn.Linear(32, 1)

        )

    def forward(self, x):

        return self.layers(x)

### 학습 및 검증 과정 및 모델 저장

* 학습 단계에서는 모델을 학습 모드로 설정하고, 배치 데이터를 이용해 예측을 수행한 다음, 손실을 계산하고 역전파를 통해 모델의 가중치를 갱신
* 검증 단계에서는 모델을 평가 모드로 설정하고, torch.no\_grad()를 사용하여 gradient계산을 중단하고 검증 데이터셋을 사용하여 모델의 성능을 평가
* 학습이 완료된 후, 가장 좋은 성능을 보인 모델을 저장

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

model = CNN1DRegressor().to(device)

#하이퍼파라미터

criterion = nn.L1Loss()

num\_epochs = 100

optimizer = optim.Adam(model.parameters(), lr=1e-4)

train\_losses = []

val\_losses = []

# 초기 설정

best\_val\_loss = float('inf')  # 초기에는 무한대로 설정

model\_save\_path = './1D\_CNN\_model/best\_model.pth'  # 모델을 저장할 경로

for epoch in range(num\_epochs):

    # Training phase

    model.train()

    train\_loss = 0.0

    for X\_batch, y\_batch in train\_dataloader:

        X\_batch, y\_batch = X\_batch.to(device), y\_batch.to(device)

        outputs = model(X\_batch)

        loss = criterion(outputs.squeeze(), y\_batch.squeeze())

        optimizer.zero\_grad()

        loss.backward()

        optimizer.step()

        train\_loss += loss.item() \* X\_batch.size(0)

    train\_loss /= len(train\_dataloader.dataset)

    train\_losses.append(train\_loss)

    # Validation phase

    model.eval()

    val\_loss = 0.0

    with torch.no\_grad():

        for X\_val, y\_val in val\_dataloader:

            X\_val, y\_val = X\_val.to(device), y\_val.to(device)

            outputs = model(X\_val)

            loss = criterion(outputs.squeeze(), y\_val.squeeze())

            val\_loss += loss.item() \* X\_val.size(0)

        val\_loss /= len(val\_dataloader.dataset)

        val\_losses.append(val\_loss)

        print(f"Epoch [{epoch + 1}/{num\_epochs}], Train Loss: {train\_loss:.6f}, Val Loss: {val\_loss:.6f}")

    # Best model 저장

    if val\_loss < best\_val\_loss:

        print(f"Validation Loss Improved ({best\_val\_loss:.6f} -> {val\_loss:.6f}). Saving model...")

        best\_val\_loss = val\_loss

        torch.save(model.state\_dict(), model\_save\_path)

torch.save(model.state\_dict(), './1D\_CNN\_model/last\_model.pth')

### 테스트 및 성능 평가

* 모델을 평가 모드로 설정한 후, 테스트 데이터셋에 대한 예측을 수행
* 예측된 결과와 실제 레이블을 이용하여 평균 제곱 오차(MSE), 평균 절대 오차(MAE), R² 점수를 계산

# 모델을 평가 모드로 설정

model.load\_state\_dict(torch.load(model\_save\_path))

#테스트 단계

model.eval()

all\_predictions = []

all\_labels = []

with torch.no\_grad():

    for data in test\_dataloader:

        input, labels = data[0].to(device), data[1].to(device)

        outputs = model(input)

        all\_predictions.append(outputs.cpu().numpy())

        all\_labels.append(labels.cpu().numpy())

# Convert to numpy arrays

all\_predictions = np.concatenate(all\_predictions)

all\_labels = np.concatenate(all\_labels)

# Calculate metrics

mse = mean\_squared\_error(all\_labels, all\_predictions)

mae = mean\_absolute\_error(all\_labels, all\_predictions)

r2 = r2\_score(all\_labels, all\_predictions)

print(f"MSE: {mse:.4f}")

print(f"MAE: {mae:.4f}")

print(f"R2 Score: {r2:.4f}")

## Long Short-Term Memory model

### 모델 정의

* LSTM은 장기적인 의존성을 효과적으로 학습하는 순환 신경망(RNN)의 한 유형으로, 긴 시퀀스 데이터에서 정보의 흐름을 조절하고 기억력을 가진 셀로 구성된 모델
* Attention
  + attention score를 추출하기 위한 class로 입력 시퀀스에 대한 가중치를 적용하여 출력을 생성
  + 즉, attention weight가 각 타임스탬프에 대한 중요도를 결정
  + 이 가중치를 확률분포로 변환하여, 각 시퀀스 요소의 중요도를 나타내는 attention score로 사용

# 어텐션 스코어 추출을 위한 추가적인 어텐션 레이어

class Attention(nn.Module):

    def \_\_init\_\_(self, feature\_dim):

        super(Attention, self).\_\_init\_\_()

        self.attention\_weight = nn.Parameter(torch.FloatTensor(feature\_dim, 1))

        self.softmax = nn.Softmax(dim=1)

    def forward(self, x\_in):

        attention\_probs = self.softmax(torch.matmul(x\_in, self.attention\_weight).squeeze(2))

        weighted\_sequence = torch.bmm(x\_in.permute(0, 2, 1), attention\_probs.unsqueeze(2)).squeeze(2)

        return weighted\_sequence, attention\_probs   # 시각화를 위해 어텐션 스코어와 weight 동시에 리턴

* ComplexLSTM
  + LSTM 레이어에 어텐션 레이어를 결합한 LSTM 모델
  + LSTM 레이어는 초기 상태(hidden state, cell state)와 함께 입력을 받아 처리
  + 어텐션 레이어는 LSTM 출력에 적용되어 중요한 정보를 강조
  + 배치 정규화, 드롭아웃, 활성화 함수(Tanh) 및 완전 연결 레이어를 통해 최종 출력을 생성

# 단순 LSTM 코드

class ComplexLSTM(nn.Module):

    def \_\_init\_\_(self, input\_dim, hidden\_dim, num\_layers, output\_dim, dropout\_prob=0.5):

        super(ComplexLSTM, self).\_\_init\_\_()

        self.hidden\_dim = hidden\_dim

        self.num\_layers = num\_layers

        self.lstm = nn.LSTM(input\_dim, hidden\_dim, num\_layers, dropout=dropout\_prob, batch\_first=True)

        self.attention = Attention(hidden\_dim)

        self.batch\_norm = nn.BatchNorm1d(hidden\_dim)

        self.dropout = nn.Dropout(dropout\_prob)

        self.fc1 = nn.Linear(hidden\_dim, hidden\_dim\*2)

        self.fc2 = nn.Linear(hidden\_dim\*2, hidden\_dim)

        self.fc3 = nn.Linear(hidden\_dim, output\_dim)

    def forward(self, x):

        h0 = torch.zeros(self.num\_layers, x.size(0), self.hidden\_dim).to(x.device)

        c0 = torch.zeros(self.num\_layers, x.size(0), self.hidden\_dim).to(x.device)

        out, \_ = self.lstm(x, (h0, c0))

        out, attention\_probs = self.attention(out)

        out = self.batch\_norm(out)

        out = self.dropout(out)

        out = torch.tanh(self.fc1(out))

        out = torch.tanh(self.fc2(out))

        out = self.fc3(out)

        return out, attention\_probs

### 학습 및 검증 과정 및 모델 저장

* 각 epoch에서는 모델을 훈련모드로 설정하여 훈련 데이터셋을 사용하여 모델을 학습하며 손실을 계산하고 역전파를 통해 모델의 가중치 업데이트
* 훈련 후 모델을 평가 모드로 전환하여 검증 데이터셋을 사용해 성능 평가
* 학습이 완료된 후, 가장 좋은 성능을 보인 모델을 저장

#하이퍼파리미터

input\_dim = 40

hidden\_dim = 256

num\_layers = 4

output\_dim = 1

dropout\_prob = 0.4

criterion = nn.L1Loss()

num\_epochs = 100

train\_losses = []

val\_losses = []

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

model = ComplexLSTM(input\_dim, hidden\_dim, num\_layers, output\_dim, dropout\_prob).to(device)

optimizer = optim.Adam(model.parameters(), lr=1e-4)

# 초기 설정

best\_val\_loss = float('inf')  # 초기에는 무한대로 설정

model\_save\_path = 'LSTM/best\_model.pth'  # 모델을 저장할 경로

for epoch in range(num\_epochs):

    # Training phase

    model.train()

    train\_loss = 0.0

    for X\_batch, y\_batch in train\_dataloader:

        X\_batch, y\_batch = X\_batch.to(device), y\_batch.to(device)

        outputs, \_ = model(X\_batch)

        loss = criterion(outputs.squeeze(), y\_batch.squeeze())

        optimizer.zero\_grad()

        loss.backward()

        optimizer.step()

        train\_loss += loss.item() \* X\_batch.size(0)

    train\_loss /= len(train\_dataloader.dataset)

    train\_losses.append(train\_loss)

    # Validation phase

    model.eval()

    val\_loss = 0.0

    with torch.no\_grad():

        for X\_val, y\_val in val\_dataloader:

            X\_val, y\_val = X\_val.to(device), y\_val.to(device)

            outputs, \_ = model(X\_val)

            loss = criterion(outputs.squeeze(), y\_val.squeeze())

            val\_loss += loss.item() \* X\_val.size(0)

        val\_loss /= len(val\_dataloader.dataset)

        val\_losses.append(val\_loss)

        print(f"Epoch [{epoch + 1}/{num\_epochs}], Train Loss: {train\_loss:.6f}, Val Loss: {val\_loss:.6f}")

    # Best model 저장

    if val\_loss < best\_val\_loss:

        print(f"Validation Loss Improved ({best\_val\_loss:.6f} -> {val\_loss:.6f}). Saving model...")

        best\_val\_loss = val\_loss

        torch.save(model.state\_dict(), model\_save\_path)

torch.save(model.state\_dict(), './LSTM/last\_model.pth')

### 테스트 및 성능 평가

* 모델을 평가 모드로 설정한 후, 테스트 데이터셋에 대한 예측을 수행하여 어탠션 맵과 예측값 추출
* 예측된 결과와 실제 레이블을 바탕으로 평균 제곱 오차(MSE), 평균 절대 오차(MAE), R² 점수를 계산
* 어텐션 맵은 모델이 예측을 수행할 때 입력 시퀀스의 어느 부분에 더 많은 중요도를 두었는지 시각화에 이용 가능

# 모델을 평가 모드로 설정

model = ComplexLSTM(input\_dim, hidden\_dim, num\_layers, output\_dim, dropout\_prob).to(device)

model.load\_state\_dict(torch.load(model\_save\_path))

model.eval()

all\_predictions = []

all\_labels = []

all\_attention\_maps = []

#테스트 단계

model.eval()

with torch.no\_grad():

    for X\_batch, y\_batch in test\_dataloader:

        X\_batch, y\_batch = X\_batch.to(device), y\_batch.to(device)

        outputs, attention\_map = model(X\_batch)

        all\_predictions.append(outputs.cpu().numpy())

        all\_labels.append(y\_batch.cpu().numpy())

        all\_attention\_maps.append(attention\_map.cpu().numpy())

# Convert to numpy arrays

all\_predictions = np.concatenate(all\_predictions)

all\_labels = np.concatenate(all\_labels)

# Calculate metrics

mse = mean\_squared\_error(all\_labels, all\_predictions)

mae = mean\_absolute\_error(all\_labels, all\_predictions)

r2 = r2\_score(all\_labels, all\_predictions)

print(f"MSE: {mse:.4f}")

print(f"MAE: {mae:.4f}")

print(f"R2 Score: {r2:.4f}")

# 성능 지표를 텍스트 파일로 저장

with open(folder\_path+'model\_performance.txt', 'w') as file:

    file.write(f'MSE: {mse}\n')

    file.write(f'R2 Score: {r2}\n')

## Dual-Attention model

### 모델 정의

* DARNN (Dual-Stage Attention-Based Recurrent Neural Network)은 시계열 데이터에서 중요한 정보를 효과적으로 추출하기 위해 두 단계의 attention 메커니즘을 적용하는 딥러닝 회귀 모델
* Input attention
  + Encoder에서 사용되는 Input Attention은 예측하고자 하는 변수들 중에서, 예측에 크게 영향을 끼치는 의미 있는 변수들에 weight를 부여하기 위한 attention score 계산
  + 각각 T의 시간 길이를 갖는 n개의 데이터를 사용하고 이를 Encoder에 있는 LSTM에 넣어줘서 hidden state를 추출한 뒤, 각 시간 단계에서 입력 데이터의 각 feature에 대한 중요도를 계산하는 attention 가중치 생성

class InputAttention(Layer):

    def \_\_init\_\_(self, T):

        super(InputAttention, self).\_\_init\_\_(name="input\_attention")

        self.w1 = Dense(T)

        self.w2 = Dense(T)

        self.v = Dense(1)

    def call(self, h\_s, c\_s, x):

        """

        h\_s : hidden\_state (shape = batch,m)

        c\_s : cell\_state (shape = batch,m)

        x : time series encoder inputs (shape = batch,T,n)

        """

        query = tf.concat([h\_s, c\_s], axis=-1)  # batch, m\*2

        query = RepeatVector(x.shape[2])(query)  # batch, n, m\*2

        x\_perm = Permute((2, 1))(x)  # batch, n, T

        score = tf.nn.tanh(self.w1(x\_perm) + self.w2(query))  # batch, n, T

        score = self.v(score)  # batch, n, 1

        score = Permute((2, 1))(score)  # batch,1,n

        attention\_weights = tf.nn.softmax(score)  # t 번째 time step 일 때 각 feature 별 중요도

        return attention\_weights

* Encoder
  + 모든 time step에 대해 계산된 attention weights를 받아 input data와 곱해줘 중요도에 따라 스케일링된 생성

class Encoderlstm(Layer):

def \_\_init\_\_(self, m):

"""

m : feature dimension

h0 : initial hidden state

c0 : initial cell state

"""

super(Encoderlstm, self).\_\_init\_\_(name="encoder\_lstm")

self.lstm = LSTM(m, return\_state=True)

self.initial\_state = None

def call(self, x, training=False):

"""

x : t 번째 input data (shape = batch,1,n)

"""

h\_s, \_, c\_s = self.lstm(x, initial\_state=self.initial\_state)

self.initial\_state = [h\_s, c\_s]

return h\_s, c\_s

def reset\_state(self, h0, c0):

self.initial\_state = [h0, c0]

class Encoder(Layer):

    def \_\_init\_\_(self, T, m):

        super(Encoder, self).\_\_init\_\_(name="encoder")

        self.T = T

        self.input\_att = InputAttention(T)

        self.lstm = Encoderlstm(m)

        self.initial\_state = None

        self.alpha\_t = None

    def call(self, data, h0, c0, n=39, training=False):

        """

        data : encoder data (shape = batch, T, n)

        n : data feature num

        """

        self.lstm.reset\_state(h0=h0, c0=c0)

        alpha\_seq = tf.TensorArray(tf.float32, self.T)

        for t in range(self.T):

            x = Lambda(lambda x: data[:, t, :])(data)

            x = x[:, tf.newaxis, :]  # (batch,1,n)

            h\_s, c\_s = self.lstm(x)

            self.alpha\_t = self.input\_att(h\_s, c\_s, data)  # batch,1,n

            alpha\_seq = alpha\_seq.write(t, self.alpha\_t)

        alpha\_seq = tf.reshape(alpha\_seq.stack(), (-1, self.T, n))  # batch, T, n

        output = tf.multiply(data, alpha\_seq)  # batch, T, n

        return output

* TemporalAttention
  + Attention 메커니즘이 적용된 변수 엑스햇티를 가지고 LSTM에 넣어준 후, 2번째 Attention인 Temporal attention을 적용
  + Encoder의 각 time step에서의 hidden states와 각 time step에서의 Decoder LSTM의 hidden state를 비교하여, 어떤 time steps가 예측에 가장 중요한지를 결정

class TemporalAttention(Layer):

    def \_\_init\_\_(self, m):

        super(TemporalAttention, self).\_\_init\_\_(name="temporal\_attention")

        self.w1 = Dense(m)

        self.w2 = Dense(m)

        self.v = Dense(1)

    def call(self, h\_s, c\_s, enc\_h):

        """

        h\_s : hidden\_state (shape = batch,p)

        c\_s : cell\_state (shape = batch,p)

        enc\_h : time series encoder inputs (shape = batch,T,m)

        """

        query = tf.concat([h\_s, c\_s], axis=-1)  # batch, p\*2

        query = RepeatVector(enc\_h.shape[1])(query)

        score = tf.nn.tanh(self.w1(enc\_h) + self.w2(query))  # batch, T, m

        score = self.v(score)  # batch, T, 1

        attention\_weights = tf.nn.softmax(

            score, axis=1

        )  # encoder hidden state h(i) 의 중요성 (0<=i<=T)

        return attention\_weights

* Decoder
  + 를 예측하기 위해 이전 time stamp의 실제 y값과 TemporalAttention을 통해 생성한 Context Vector를 concatenate하여 해당 값을 인풋으로 하는 LSTM 모델을 사용해 최종 예측을 진행

class Decoderlstm(Layer):

    def \_\_init\_\_(self, p):

        """

        p : feature dimension

        h0 : initial hidden state

        c0 : initial cell state

        """

        super(Decoderlstm, self).\_\_init\_\_(name="decoder\_lstm")

        self.lstm = LSTM(p, return\_state=True)

        self.initial\_state = None

    def call(self, x, training=False):

        """

        x : t 번째 input data (shape = batch,1,n)

        """

        h\_s, \_, c\_s = self.lstm(x, initial\_state=self.initial\_state)

        self.initial\_state = [h\_s, c\_s]

        return h\_s, c\_s

    def reset\_state(self, h0, c0):

        self.initial\_state = [h0, c0]

class Decoder(Layer):

    def \_\_init\_\_(self, T, p, m):

        super(Decoder, self).\_\_init\_\_(name="decoder")

        self.T = T

        self.temp\_att = TemporalAttention(m)

        self.dense = Dense(1)

        self.lstm = Decoderlstm(p)

        self.enc\_lstm\_dim = m

        self.dec\_lstm\_dim = p

        self.context\_v = None

        self.dec\_h\_s = None

        self.beta\_t = None

    def call(self, data, enc\_h, h0=None, c0=None, training=False):

        """

        data : decoder data (shape = batch, T-1, 1)

        enc\_h : encoder hidden state (shape = batch, T, m)

        """

        h\_s = None

        self.lstm.reset\_state(h0=h0, c0=c0)

        self.context\_v = tf.zeros((enc\_h.shape[0], 1, self.enc\_lstm\_dim))  # batch,1,m

        self.dec\_h\_s = tf.zeros((enc\_h.shape[0], self.dec\_lstm\_dim))  # batch, p

        for t in range(self.T - 1):  # 0~T-1

            x = Lambda(lambda x: data[:, t, :])(data)

            x = x[:, tf.newaxis, :]  #  (batch,1,1)

            x = tf.concat([x, self.context\_v], axis=-1)  # batch, 1, m+1

            x = self.dense(x)  # batch,1,1

            h\_s, c\_s = self.lstm(x)  # batch,p

            self.beta\_t = self.temp\_att(h\_s, c\_s, enc\_h)  # batch, T, 1

            self.context\_v = tf.matmul(

                self.beta\_t, enc\_h, transpose\_a=True

            )  # batch,1,m

        return tf.concat(

            [h\_s[:, tf.newaxis, :], self.context\_v], axis=-1

        )  # batch,1,m+p

* DARNN
  + 앞서 구현한 코드들을 종합하여 Dual-Attention RNN 생성

class DARNN(Model):

    def \_\_init\_\_(self, T, m, p):

        super(DARNN, self).\_\_init\_\_(name="DARNN")

        """

        T : 주기 (time series length)

        m : encoder lstm feature length

        p : decoder lstm feature length

        h0 : lstm initial hidden state

        c0 : lstm initial cell state

        """

        self.m = m

        self.encoder = Encoder(T=T, m=m)

        self.decoder = Decoder(T=T, p=p, m=m)

        self.lstm = LSTM(m, return\_sequences=True)

        self.dense1 = Dense(p)

        self.dense2 = Dense(1)

    def call(self, inputs, training=False, mask=None):

        """

        inputs : [enc , dec]

        enc\_data : batch,T,n

        dec\_data : batch,T-1,1

        """

        enc\_data, dec\_data = inputs

        batch = enc\_data.shape[0]

        h0 = tf.zeros((batch, self.m))

        c0 = tf.zeros((batch, self.m))

        enc\_output = self.encoder(

            enc\_data, n=39, h0=h0, c0=c0, training=training

        )  # batch, T, n

        enc\_h = self.lstm(enc\_output)  # batch, T, m

        dec\_output = self.decoder(

            dec\_data, enc\_h, h0=h0, c0=c0, training=training

        )  # batch,1,m+p

        output = self.dense2(self.dense1(dec\_output))

        output = tf.squeeze(output)

        return output

### 모델 학습 과정

* DARNN Class를 사용하여 모델 생성
* 모델의 인풋 데이터는 driving series dataset은 특정 timestep T만큼의 센서 값과 예측하고자 하는 T-1개의 target series ‘ei
* 모델의 아웃풋 데이터는 T 시점에서의 target인 ‘ei’
* 각 epoch에서는 모델을 훈련모드로 설정하여 훈련 데이터셋을 사용하여 모델을 학습하며 손실을 계산하고 역전파를 통해 모델의 가중치 업데이트 & 평가 모드로 전환하여 검증 데이터셋을 사용해 성능 평가

# 학습 파라미터

## Random forest

|  |  |
| --- | --- |
| Random forest | |
| N\_estimators | 100 |

## 1D CNN

|  |  |
| --- | --- |
| 1D CNN | |
| Optimizer | Adam |
| Learning rate | 0.0001 |
| Loss function | L1Loss |
| Epoch | 100 |
| Input channel | 40 |
| Kernel size | 3 |
| Activation Function | Tanh |
| Dropout Rate | 0.4 |

## LSTM

|  |  |
| --- | --- |
| LSTM | |
| Optimizer | Adam |
| Learning rate | 0.0001 |
| Loss function | L1Loss |
| Epoch | 100 |
| Input dim | 40 |
| Hidden dim | 256 |
| Num layers | 4 |
| Output dim | 1 |
| Activation Function | Tanh |
| Dropout rate | 0.4 |

## DARNN

|  |  |
| --- | --- |
| DARNN | |
| Optimizer | Adam |
| Learning rate | 0.001 |
| Loss function | L1Loss |
| Epoch | 200 |
| Encoder dim(Encoder LSTM 레이어의 출력 차원수) | 20 |
| Decoder dim(Decoder LSTM 레이어의 출력 차원수) | 20 |
| Output dim | 1 |
| Activation Function | Tanh |

# 실행 결과 설명

## 예측 결과

### 결정계수(R2)

* 결정 계수(R2)는 회귀 모델의 성능을 평가하는 지표
* 모델이 데이터의 분산을 얼마나 잘 설명하는지를 나타내며, 0에서 1사이의 값을 가짐
* R2가 1에 가까울수룩 모델이 데이터를 잘 설명하고 있다고 해석
* R²가 높다는 것은 모델이 관측된 데이터의 분산을 잘 설명하고 있음을 의미하며, 예측의 정확도가 높음을 의미
  + R² = 1 : 모델이 데이터를 완벽하게 예측
  + R² = 0 : 모델이 평균적인 수준으로만 데이터를 예측
  + R² < 0: 모델의 예측 성능을 신뢰하기 어려움

|  |
| --- |
| =1- |

* + 는 잔차 제곱 합(Residual Sum of Squares)으로, 모델의 예측과 실제 값의 차이의 제곱 합
    - 이 값이 작을수록 모델의 예측이 실제 데이터에 더 가까움을 의미하며 모델의 성능이 더 좋다는 것을 의미
    - =
  + 는 총 제곱 합(Total Sum of Squares)으로, 관측치와 관측치 평균 간의 차이의 제곱 합
    - 데이터가 평균값에서 얼마나 퍼져있는지의 정도인 변동성을 나타내는 값으로 모델이 얼마나 많은 분산을 설명하고 있는지를 의미
    - =

### 평균 제곱 오차(MSE)

* 평균 제곱 오차(MSE)는 모델의 예측값과 실제값의 차이를 제곱하여 평균낸 것으로 회귀 모델에서 일반적으로 많이 사용되는 성능 평가 지표
* 값이 낮을수록 모델의 정확도가 높음을 의미하며, 예측값이 실제값과 완전히 일치할 경우 0.

|  |
| --- |
| MSE = |

### 그래프 해석 방법

* Scatter plot (Ex. CNN model-timegap 10)

도표, 텍스트, 지도이(가) 표시된 사진

자동 생성된 설명

* + X축과 상단 히스토그램은 실제 값을 의미하며 테스트 데이터 셋의 실제 타겟 변수의 값을 의미
  + Y축과 오른쪽 히스토그램은 예측 값을 의미하며 모델에 의해 생성된 예측에 대한 추정 값을 의미
  + 산점도는 각 점이 하나의 데이터 포인트를 의미하며, 예측값이 실제값과 모두 일치할 경우 y = x에 해당하는 대각선에 모든 데이터들이 위치
  + 모델의 예측이 실제 값과 얼마나 잘 일치하는지, 예측의 정확도와 일관성을 시각적으로 파악
  + 분포가 대각선을 따라 좁고 균일할수록 좋은 예측 모델
* Time series prediction plot (Ex. CNN model - timegap 10)

스크린샷, 그래프, 도표, 라인이(가) 표시된 사진

자동 생성된 설명

* + X축("Time")은 시계열 데이터의 각 시점을 의미, Y축("Value")은 시계열 데이터의 실제 값을 의미
  + 파란색 실선("Actual Values")의 경우 실제 시계열 데이터를 나타내며, 빨간색 점선("Predictions")의 경우 모델에 의해 생성된 예측값을 의미
  + 모델이 시간의 흐름에 따라 값을 얼마나 잘 예측하는지를 보여주기 위함이며, 정확도가 높은 예측 모델일 경우 파란색 실선(실제 값)과 빨간색 점선(예측 값)이 서로 밀접하게 일치

## 원인 인자 도출

### Feature Importance

* 각 feature들이 random forest를 구성하는 decision tree들에서 분할에 얼마나 중요한지를 기반으로 계산
* Feature 중요도는 특정 feature를 사용함에 따라 purity 증가량(Gini impurity 또는 Entropy 감소) 따라 결정
* 즉, feature가 분할에 사용될 때 purity가 크게 개선되면, 그 피처는 중요하다고 간주
* 상위 20개를 추출해 이를 시각화
* Random forest에서 도출

#랜덤 포레스트 변수 중요도 시각화

ftr\_importances\_values = rf\_regressor.feature\_importances\_

ftr\_importances = pd.Series(ftr\_importances\_values, *index*=X\_test\_df.columns)

ftr\_top = ftr\_importances.sort\_values(*ascending*=False)[:20]

plt.figure(*figsize*=(8, 6))

bars = sns.barplot(*x*=ftr\_top, *y*=ftr\_top.index)

# 각 막대에 중요도 값을 추가

for idx, val in enumerate(ftr\_top):

    bars.text(val, idx,

              f'{val:.4f}',

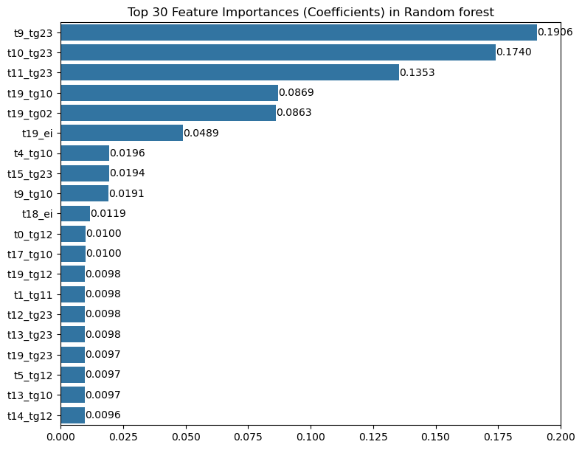
*va*='center', *ha*='left', *color*='black')

plt.tight\_layout()

plt.title("Top 20 Feature Importances (Coefficients) in Random forest")

plt.savefig(folder\_path + 'Random Forest feature plot.png')

plt.show()



### SHAP

* SHAP은 모델의 예측이 어떻게 각 입력 변수의 값에 의해 영향을 받는지를 수치적으로 나타낸 것
* SHAP은 모든 가능한 피처 조합을 고려하여 각 피처의 평균 기여도를 계산하여 피처가 예측 값을 얼마나 증가시키거나 감소시키는지를 나타냄
* 1D-CNN, LSTM에서 도출
* SHAP Summary Plot
  + 사전에 정의한 여러 개의 데이터 포인트에 대한 평균 SHAP value의 분포를 시각화
  + 각 데이터 포인트에서 피처의 SHAP값이 어떻게 변하는지를 나타내며, 피처가 예측에 긍정적인 영향을 미치는지 또는 부정적인 영향을 미치는지에 대한 정보 제공

background\_data = X\_train\_tensor[:10].permute(0,2, 1).to(device)

explainer = shap.DeepExplainer(model, background\_data)

# 테스트 데이터 준비

all\_test\_data = X\_test\_tensor[:10].permute(0,2, 1).to(device)

# SHAP 값 계산

shap\_values = explainer.shap\_values(all\_test\_data)

shap\_values : 각 테스트 데이터 포인트에 대한 특성 별 SHAP value값이 도출

🡪 instance가 10인 test 데이터의 포인트 예측에 대한 각 특성의 기여도를 의미

SHAP value가 양수일 경우: 해당 특성이 모델의 예측 값을 증가시키는 경향이 있음

SHAP value가 음수일 경우: 해당 특성이 모델의 예측 값을 감소시키는 경향이 있음

# shap\_values가 리스트인 경우 numpy 배열로 변환

if isinstance(shap\_values, list):

    shap\_values = np.array(shap\_values[0])

# 시간 단계별로 평균내는 과정

average\_shap\_values\_over\_time = np.mean(shap\_values, *axis*=1)

average\_shap\_values\_over\_time : 각 특성의 모든 시간대의 평균 SHAP value를 계산

🡪 각 인스턴스에 대해 각 특성이 평균적으로 모델의 예측에 얼마나 영향을 미치는지를 나타내는 지표

# 평균낸 SHAP 값의 형태를 (샘플 수, 특성 수)로 변경

shap\_values\_averaged = average\_shap\_values\_over\_time.reshape(shap\_values.shape[0], -1)

# 테스트 데이터를 시간 단계별로 평균냄

all\_test\_data\_averaged = np.mean(all\_test\_data.cpu().numpy(), *axis*=1)

# SHAP 값과 테스트 데이터의 형태가 일치하는지 확인

assert shap\_values\_averaged.shape == all\_test\_data\_averaged.shape

# 시각화

feature\_names = X\_train[0].columns.tolist()  # Pandas DataFrame의 열 이름 사용

shap.summary\_plot(shap\_values\_averaged, all\_test\_data\_averaged, *feature\_names*=feature\_names)

plt.savefig(folder\_path+'/shap summary plot.png')

plt.clf()

shap.summary\_plot : 모델이 각 입력에 대한 평균 SHAP value를 활용하여 각 특성이 예측 결과에 얼마나 영향을 미쳤는지를 시각화, 이 plot에서 각 점은 개별 인스턴스를 나타냄

🡪 각 특성이 결과에 미치는 특성의 전반적인 중요도와 그 영향의 분포 확인 가능

텍스트, 스크린샷, 도표, 번호이(가) 표시된 사진

자동 생성된 설명 텍스트, 스크린샷, 라인, 도표이(가) 표시된 사진

자동 생성된 설명

* + 해석 방법
* X축은 SHAP value를 의미하며 Y축의 경우 서로 다른 feature를 나타냄
* SHAP value가 붉은색인 경우 양수이며 예측 값을 증가시키는 feature라는 것을 의미하고 파란색인 경우 음수이며 예측 값을 감소시키는 feature라는 것을 의미함
* 예를 들어, 오른쪽 summary plot의 tg04의 경우, feature value가 큰 값들의 SHAP value가 크므로 feature와 예측 값이 양의 상관관계를 가진다는 것을 알 수 있고, tg09의 경우 feature value가 큰 값들의 SHAP value가 낮으므로 feature와 예측 값이 음의 상관관계를 이룬다고 해석
* Top Feature Importances Plot by SHAP
  + feature별로 SHAP 값의 평균 절대값을 계산하여 feature의 중요도를 평가하고, 이를 bar chart로 시각화
  + 모델 전반에 걸쳐 각 피처가 모델 예측에 미치는 평균적인 영향을 보여줌

mean\_abs\_shap\_values = np.mean(np.abs(shap\_values), *axis*=(0, 2))

mean\_abs\_shap\_values: 모든 인스턴스에 대한 SHAP값 배열의 절대값의 평균을 계산

🡪 어떤 특성이 일관되게 모델에 중요한 영향을 미치는지를 의미

SHAP value의 절대값이 클 경우: 모델의 예측에 큰 영향을 미치고 있음을 의미

SHAP value의 절대값이 작을 경우: 모델의 예측에 상대적으로 작은 영향을 미치고 있다는 것을 의미

feature\_names = X\_train[0].columns.tolist()

# 특성 중요도와 이름을 DataFrame으로 변환

feature\_importances = pd.DataFrame({

    'feature': feature\_names,

    'importance': mean\_abs\_shap\_values

})

# 중요도에 따라 특성을 정렬

feature\_importances = feature\_importances.sort\_values(*by*='importance', *ascending*=False)

# 중요도에 따라 특성을 정렬하고 상위 10개를 선택

top\_feature\_importances = feature\_importances.sort\_values(*by*='importance', *ascending*=False).head(10)

# 바 차트로 상위 10개 특성 중요도를 시각화

top\_feature\_importances.plot(*kind*='bar', *x*='feature', *y*='importance', *legend*=False)

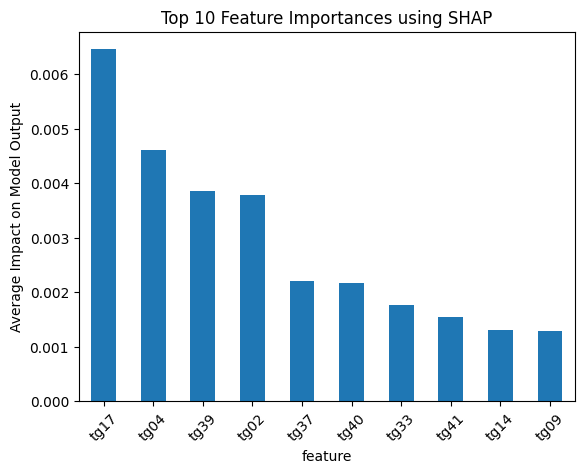
plt.title('Top 10 Feature Importances using SHAP')

plt.ylabel('Average Impact on Model Output')

plt.xticks(*rotation*=45)  # 특성 이름이 길 경우 회전시켜서 라벨이 겹치지 않도록 설정

plt.savefig(folder\_path+'/cumulative shap plot.png')

plt.clf()



* + 해석 방법
* X축은 서로 다른 feature를 의미하고 Y축의 경우 instance의 feature별 shap의 절대값에 대한 평균값을 나타냄
* 각 변수의 SHAP value의 중요도를 의미하는 plot으로 모델에 전반적으로 기여하는 feature에 대해서 알 수 있음
* 예를 들어, 위의 Feature Importances Plot에서 feature가 tg17, tg04, tg39가 모델의 전반적인 예측에 다른 feature에 비해 많이 기여하고 있다고 해석

### Attention Map

* Attention : 예측을 위해 입력되는 여러 센서의 데이터를 처리할 때 모든 센서를 동일하게 취급하는 것이 아닌 예측하고자 하는 아웃풋(ei)과 높은 상관관계를 가지는 센서에 더 많은 가중치를 부여하는 방법
* Input Attention Layer: DARNN에서의 Input attention layer는 모델이 입력 데이터 중 어떤 특징이 중요한지를 판단하는 역할 수행하여 중요한 센서에 더 많은 가중치를 부여하고, 이를 바탕으로 최종 예측 결과에 어떤 센서 값이 어떠한 영향을 미쳤는지를 결정
* Input Attention score : 딥러닝 모델이 입력 센서 데이터의 중요도를 평가하기 위해 Input Attention Layer 를 통해 계산되는 수치
* 센서들의 가중치 : Attention score를 softmax 함수를 통해 정규화 하여 모델이 각 센서의 중요도를 어떻게 인식하는지 수치화
* Attention map : attention score를 활용해 어떤 입력 요소가 중요한 역할을 했는지를 시각화 (단일 인스턴스에 대한 attention score 혹은 여러 인스턴스들에 대한 attention score 값들의 평균 모두 가능)

feature\_names = X\_train[0].columns.tolist()

pred = model(inputs)

alpha = []

for i in range(len(feature\_names)):

    alpha.append(np.mean(model.encoder.alpha\_t[:, 0, i].numpy()))

# 특성 중요도와 이름을 DataFrame으로 변환

feature\_importances = pd.DataFrame({

    'feature': feature\_names,

    'attention\_score': alpha

})

# 중요도에 따라 특성을 정렬

feature\_importances = feature\_importances.sort\_values(*by*='attention\_score', *ascending*=False)

# 중요도에 따라 특성을 정렬하고 상위 10개를 선택

top\_feature\_importances = feature\_importances.sort\_values(*by*='attention\_score', *ascending*=False).head(10)

# 바 차트로 상위 10개 특성 중요도를 시각화

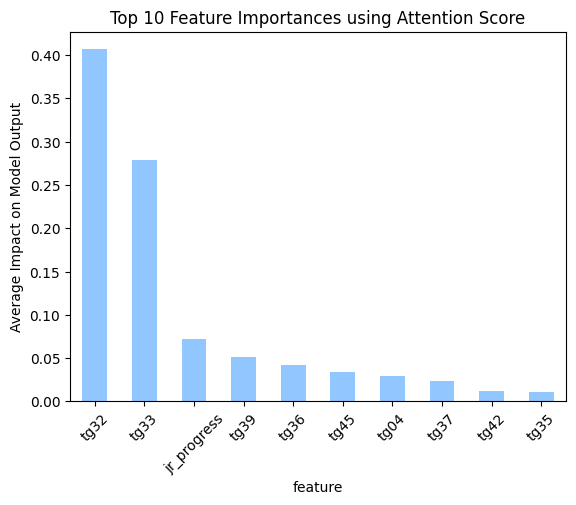
top\_feature\_importances.plot(*kind*='bar', *x*='feature', *y*='attention\_score', *legend*=False)

plt.title('Top 10 Feature Importances using Attention Score')

plt.ylabel('Average Impact on Model Output')

plt.xticks(*rotation*=45)  # 특성 이름이 길 경우 회전시켜서 라벨이 겹치지 않도록 설정

plt.savefig(folder\_path + 'Feature Attention Map.png', *dpi*=300)



* + 해석 방법
* X축은 서로 다른 feature를 의미하고 Y축의 경우 예측한 instance의 attention score(변수의 중요도)의 평균값을 나타냄
* 각 변수의 attention 정도를 알 수 있으며, 모델이 전반적으로 모델 예측에 중요하게 판단하는 feature에 대해서 알 수 있음
* 예를 들어, 위의 Feature Importances Plot에서 feature가 tg32, tg33, jr\_progress가 모델의 전반적인 예측에 다른 feature에 비해 중요도가 높다고 해석