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

- 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)

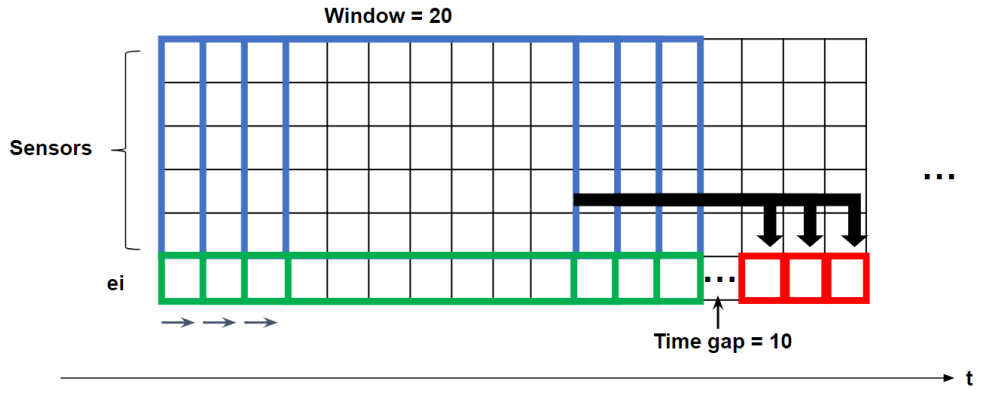
Column	의미	데이터 타입	개수
date	데이터 수집 시각(단위 : 분)	datetime64[ns]	66106
tg	Sensor (38개)	float64	
output	제품 생산량 계산값	float64	
ei	원단위 계산값 (=스팀 사용량/output)	float64	
sstable	원단위 상태 분석값 ~~0 : 좋음~~ 1 : 적당 ~~2 : 나쁨~~	int64	
jr	단위 공정값 / 제품 생산 주기 (생산품 번호)	int64	
shift	작업팀 구분값	int64	
wclass	작업팀 구분값	int64	
stop	공정 분석값 0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

- 시계열 예측 모델 개요

■ 괴거 시점의 시계열 센서값을 통해 미래 시점의 ei값을 추론하는 regression 모델

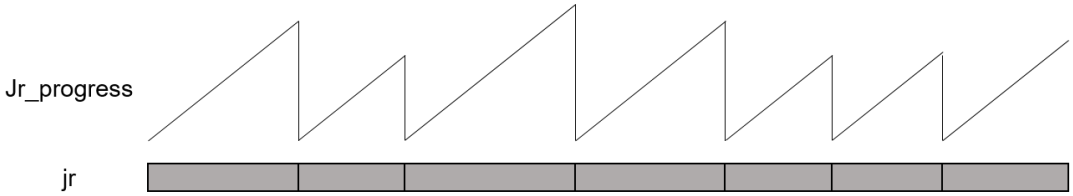
◆ Window: 모델의 input으로 사용될 시계열 센서 데이터의 구간 길이

◆ Time gap: 현 시점과 예측 대상이 되는 미래 시점 사이의 구간 길이



2. 데이터 전처리

2.1. ‘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)
```

2.2. 활용 변수 정리

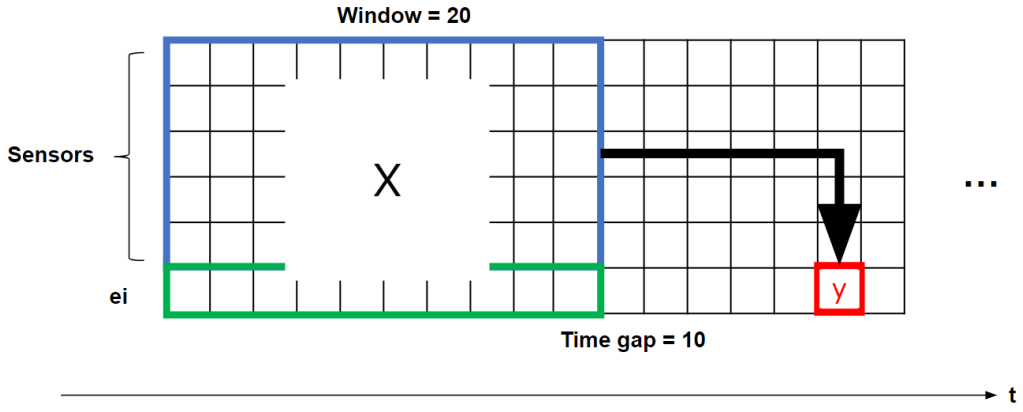
- 분석에 사용하지 않을 변수 삭제

```
# 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

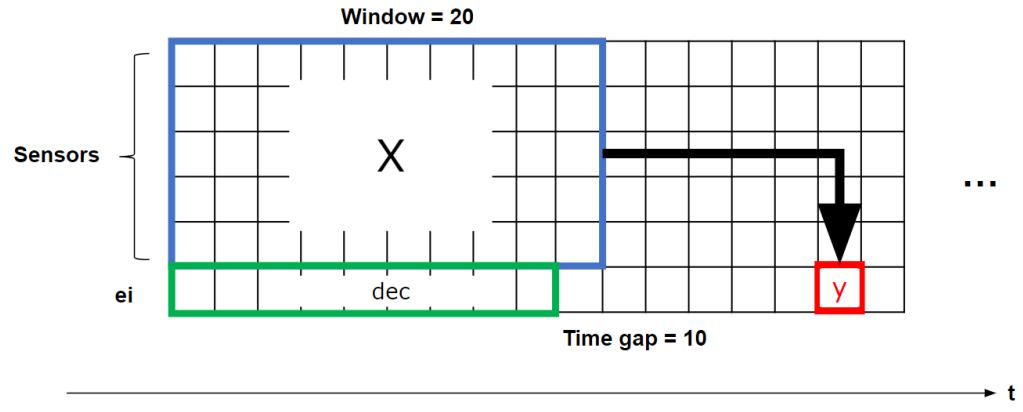
  이전의 y 정보를 미래 시점 예측을 위한 학습 데이터로 사용




```
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



```python
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'])
```

2.3. 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

```python
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_train 과 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]
```

2.4. 각 모델에 적합한 데이터 구조 변경

2.4.1. 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)
```

2.4.2. 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-CNN 이라 (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()
```

2.4.3. 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()
```

2.4.4. 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)
)
```

3. 모델 구조

3.1. Random Forest

3.1.1. 모델 정의

- 랜덤 포레스트란 결정나무를 기본 모델로 사용하는 앙상블 방법

3.1.2. 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)
```

3.1.3. 성능 평가

- 평균 제곱 오차(MSE), 평균 절대 오차(MAE), R^2 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')
```

3.2. Convolutional Neural Network model

3.2.1. 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)
```

3.2.2. 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')
```

3.2.3. 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), R^2 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}")
```

3.3. Long Short-Term Memory model

3.3.1. 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
```

3.3.2. 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')
```

3.3.3. 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), R^2 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')
```

3.4. Dual-Attention model

3.4.1. 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 $\hat{x}_t$ 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

  ■ y_t를 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
```

3.4.2. 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

4. 학습 파라미터

4.1. Random forest

Random forest	
N_estimators	100

4.2. 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

4.3. 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

4.4. 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

5. 실행 결과 설명

5.1. 예측 결과

5.1.1. 결정계수(R^2)

- 결정 계수(R^2)는 회귀 모델의 성능을 평가하는 지표
- 모델이 데이터의 분산을 얼마나 잘 설명하는지를 나타내며, 0에서 1사이의 값을 가짐
- R^2가 1에 가까울수록 모델이 데이터를 잘 설명하고 있다고 해석
- R^2가 높다는 것은 모델이 관측된 데이터의 분산을 잘 설명하고 있음을 의미하며, 예측의 정확도가 높음을 의미
  - $R^2 = 1$: 모델이 데이터를 완벽하게 예측
  - $R^2 = 0$: 모델이 평균적인 수준으로만 데이터를 예측
  - $R^2 < 0$: 모델의 예측 성능을 신뢰하기 어려움

$$R^2 = 1 - \frac{SS_{res}}{SS_{tot}}$$

  - SS_{res}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
    - $SS_{res} = \sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2$
  - SS_{tot}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
    - $SS_{tot} = \sum_{i=1}^{n}(Y_i - \bar{Y}_i)^2$

5.1.2. 평균 제곱 오차(MSE)

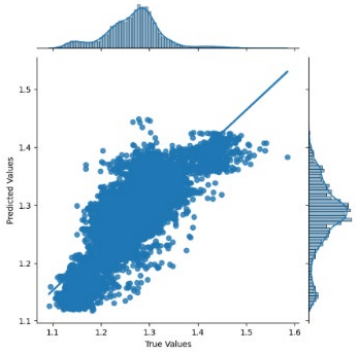
- 평균 제곱 오차(MSE)는 모델의 예측값과 실제값의 차이를 제곱하여 평균낸 것으로 회귀 모델에서 일반적으로 많이 사용되는 성능 평가 지표

- 값이 낮을수록 모델의 정확도가 높음을 의미하며, 예측값이 실제값과 완전히 일치할 경우 0.

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

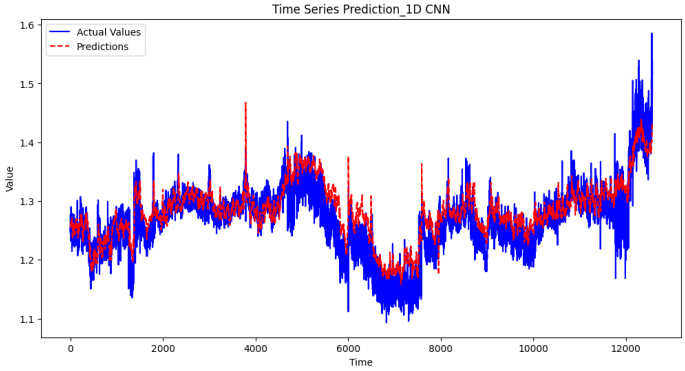
5.1.3. 그래프 해석 방법

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



  - 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)



- 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

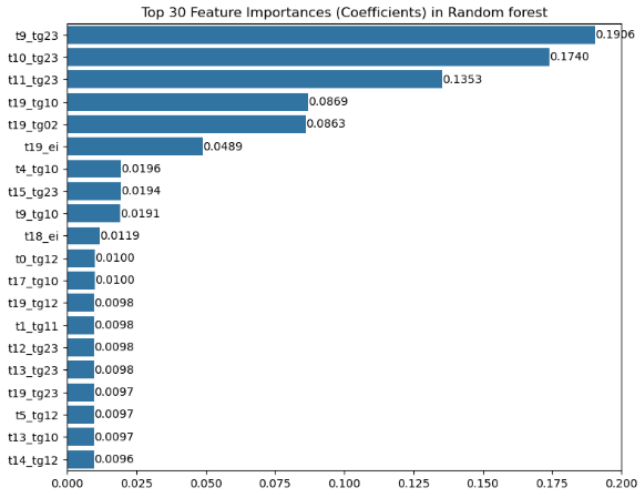
5.2. 원인 인자 도출

5.2.1. 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

```
#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()
```

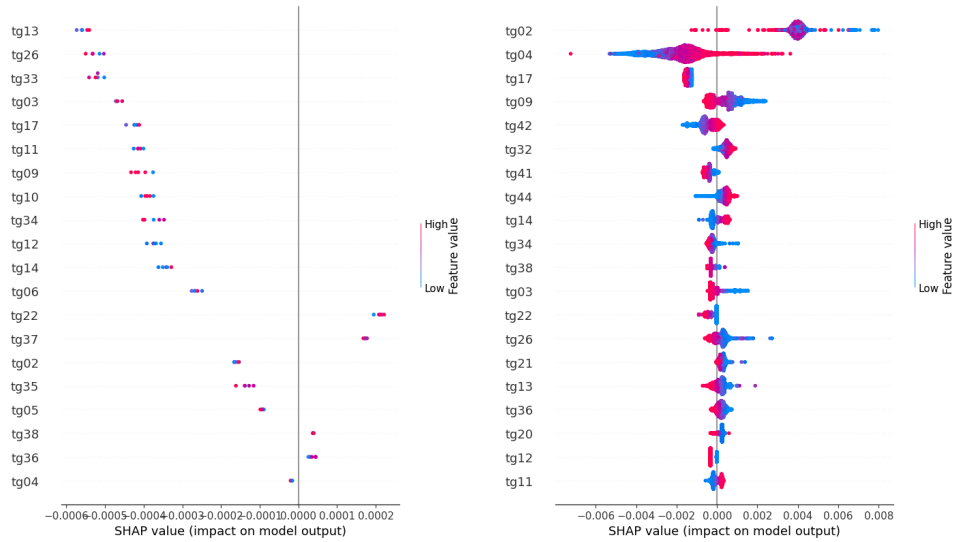


5.2.2. SHAP

- 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 예측값을 얼마나 증가시키거나 감소시키는지를 나타냄
- SHAP Summary Plot
  - ■ 사전에 정의한 몇 개의 데이터 포인트에 대한 SHAP 값의 분포를 시각화
  - ■ 각 데이터 포인트에서 피처의 SHAP 값이 어떻게 변하는지를 나타내며, 피처가 예측에 긍정적인 영향을 미치는지 또는 부정적인 영향을 미치는지에 대한 정보 제공

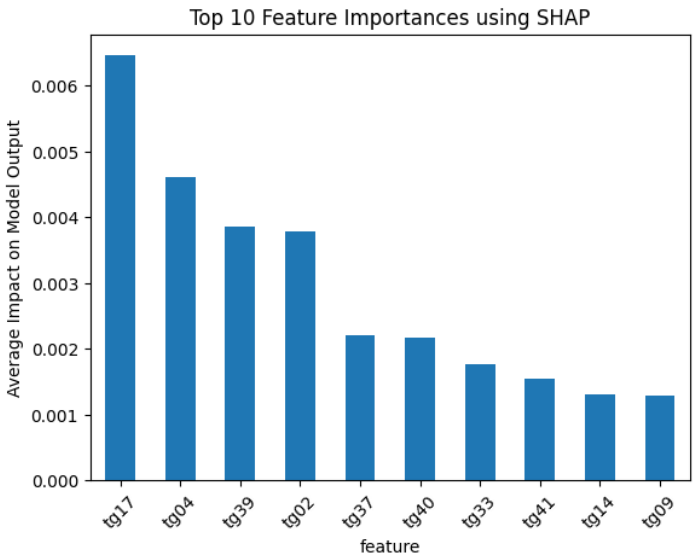
```
background_data = X_train_tensor[:10].permute(0,2, 1).to(device)
explainer = shap.DeepExplainer(model, background_data)

# 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 가 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)

# 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 DataFrame 의 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()
```



- 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

```python
mean_abs_shap_values = np.mean(np.abs(shap_values), axis=(0, 2))

feature_names = X_train[0].columns.tolist()
# 특성 중요도와 이름을 DataFrame 으로 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 10 개를 선택
top_feature_importances = feature_importances.sort_values(by='importance',
ascending=False).head(10)

# 바 차트로 상위 10 개 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()
```



5.2.3. Attention Map

- Attention은 딥러닝 모델에서 예측을 수행할 때 특정 입력 부분에 더 많은 가중치를 할당하는 방식
- Attention map은 attention score를 활용해 어떤 입력 요소가 중요한 역할을 했는지를 시

각화

```
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 DataFrame 으로 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 10 개를 선택
top_feature_importances = feature_importances.sort_values(by='attention_score',
ascending=False).head(10)

# 바 차트로 상위 10 개 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)
```