mmWave Radar 신호를 이용한 낙상 감지 및 예측 시스템 고도화 기술 개발 문서(AI)

1. 개요

visualizer는 ti센서에서 받아온 신호를 자체 프로그램으로 csv 파일 데이터로 만든다. 그 데이터는 전처리되어 visualizer에서 객체의 위치를 나타내고 행동 판단까지 수행한다. AI 부분에서는 센서로부터 input으로 들어오는 csv 파일 데이터를 통해 객체의 위치와 어떤 행동으로 판단했는지를 output으로 낸다. 이 output은 UI에게 input으로 넘겨져서 visualizer에서 그것들을 띄우게 만든다.
우리가 이번 프로젝트에서 맡은 목표는 한 명만 인식 가능하고 낙상인가 아닌가만 인식 가능한 기존 프로그램에서 사람을 5명까지 인식할 수 있고 여러 동작(낙상, 앉기, 걷기 등등..)을 인식 가능하게 만들라는 것이었고 이에 따라 클러스터링을 구현하는 것이 주요한 개발 목표가 되었다.
이 글에서는 AI부분에서 사용하는 데이터의 형식과 전처리 및 클러스터링, 모델 학습 과정을 설명한다. 그리고 구상이나 구현 과정에서 생각했던 문제점들을 나열하고 이를 주요 논문에 나오는 솔루션들로 분석해서 해결책을 생각해서 적었다. 그리고 마지막으로 향후 프로젝트에 무엇을 해야 할 것 같은지에 대해 적었다.

깃허브에 올라온 AI부분 가이드들은 현장실습 전에 적은 내용들이 섞여있기 때문에 맞지 않는 것들이 있다. 따라서 이 문서만 보는 것을 권장한다.

2. 데이터 설명

데이터는 CSV 파일 형식으로 센서로부터 받아와진다.
AI 부분에서는 총 6개의 column을 가진 데이터를 받아와서 판다스의 DataFrame에 저장한다. 아래는 각 컬럼에 대한 설명이다.
frameNum: 55 밀리초 단위의 프레임을 1씩 증가하는 순서로 저장하는데, 몇 번 프레임인지를 여기에서 알 수 있다.
pointNum: 각 프레임에서 센서를 통해 얻어진 point를 나타내는 단위이다. point 여러개가 모여서 움직이는 물체를 나타낸다. 한 frameNum에는 여러 pointNum들이 나타난다.
Range: point가 센서로부터 얼마나 멀리 떨어져 있는지와 관련된 변수
Azimuth, Elevation: 각 point의 위치를 나타내는데 관련된 변수
Doppler: 각 point의 속도와 관련한 변수

SNR: 각 point의 신호의 세기에 관련한 변수

3. 전처리 및 클러스터링 설명

한 프레임에서 나타나는 여러 point들이 있을 것이다. 이 point들은 각각이 합쳐져서 하나의 객체가 되는지, 아니면 3개의 객체를 나타내는지 알 수가 없다. 이를 알아내기 위해 클러스터링 알고리즘을 사용하기로 했는데 우리가 사용하는 데이터는 한 프레임에 몇 개의 클러스터(객체)가 나타날지 알 수 없기 때문에 그 개수를 알려줄 필요가 없다는 점에서 차별화 된 비지도 클러스터링 알고리즘인 DBSCAN을 사용하기로 했다.

이 부분에서 input은 전처리 되지 않은 위의 데이터 설명 부분에서 나온 데이터들이고 output은 각 point들에 이 데이터가 몇 번 클러스터인지 번호가 매겨진 컬럼이 추가된 데이터가 나온다.

클러스터링 과정을 구현한 코드는 우리가 작업했던 깃허브에
src/ai/한_파일에서_군집_골라내기.ipynb 이 경로로 저장되어 있다.

- 코드 설명

이 코드에서는 코드에서 다루는 data 변수에 우리가 수집한 한 csv 데이터를 사용했다. 하지만 데이터 설명에서 소개한 데이터와 같은 형식의 데이터면 모두 사용 가능하다.

센서 오작동으로 인해 DataFrame의 frameNum이 순서대로 증가하지 않거나 1부터 시작하지 않는 경우가 발생했다. 아래는 frameNum을 1부터 1씩 증가시키게 정렬시키는 코드이다.

```
#frameNum 1부터 1씩 증가시키게 정렬
data = pd.read_csv("/content/cloud_2024_12_31_10_59_4.csv")
unique_frame_numbers = {frame: idx + 1 for idx, frame in enumerate(data['frameNum'].unique())}
data['frameNum'] = data['frameNum'].map(unique_frame_numbers)
```

아래는 한 프레임에 여러 클러스터가 나올 수 있으니 그 클러스터를 구분하는 코드다.

```
#한 프레임 내에서 클러스터링
clustered_data = pd.DataFrame(columns = data.columns)
for frame, group in data.groupby("frameNum"):
  group_data = group[["Range", "Azimuth", "Elevation"]]
  scaler = StandardScaler()
  scaled_data = scaler.fit_transform(group_data)

  dbscan = DBSCAN(eps=1, min_samples=3)
  clusters = dbscan.fit_predict(scaled_data)
  group['cluster'] = clusters
  group = group.sort_values(by='cluster')
  clustered_data = pd.concat([clustered_data, group])

clustered_data = clustered_data[clustered_data['cluster'] != -1].reset_index().drop(columns='index')
```

위에서 정렬된 data를 가지고 한 프레임 내의 여러 cluster들을 알아내기 위해 data를 groupby('frameNum')하여 각 프레임마다 cluster 구분을 수행한다. 데이터에서 Doppler, SNR 특징은 포인트마다 너무 큰 차이가 나서 클러스터링에 방해가 되고 클러스터링에는 위치 정보만 이용하는게 좋다고 판단해서 Range, Azimuth, Elevation 3가지 특징만 사용했다. 사

이킷런의 DBSCAN을 사용하기 위해 StandardScaler로 정규화하고 DBSCAN 클러스터링 했다. 매개변수는 휴리스틱한 방식으로 찾았는데 수집한 데이터가 많아지면 그에 맞게 바꿔주길 바란다. 그리고 클러스터된 각 프레임을 정렬해준다. 나중에 global_cluster를 맞춰주는데 필요하기 때문이다(뒤에서 설명). 그리고 빈 데이터프레임에 concat하여 하나씩 붙인다. 그렇게 모든 원래 데이터의 모든 행에 cluster column이 추가된 데이터인 clustered_data를 만든다. 이는 각 point들에 대해 이 데이터가 한 프레임에서 어떤 cluster에 들어가있는지를 나타낸다. 맨 마지막 줄은 이상치로 감지된 클러스터 포인트들을 제외하는 행이다. 나중에는 이 코드를 빼야한다. 왜냐하면 우리의 목표가 이상치로 감지된 포인트들 까지 제대로 레이블링을 해서 쓸 수 있는 데이터로 만들어야 하기 때문이다.

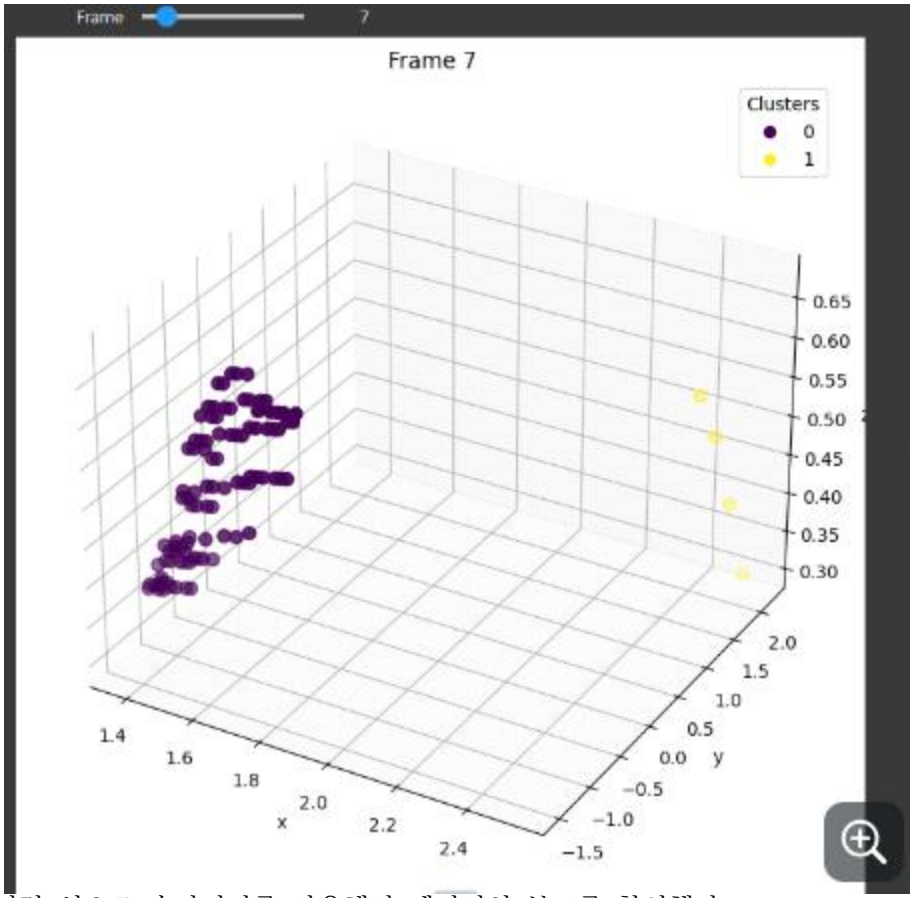
그 후 Range, Elevation, Azimuth 값들을 변형 식을 이용해서 각 pointNum마다 x,y,z 컬럼을 만들었다. 여기서 만들어진 x,y,z 컬럼은 UI부분에서 point들을 띄우는데도 사용하고 클러스터링에서도 사용하고 모델 학습에서도 사용한다.

```
clustered_data['x'] = clustered_data['Range'] * np.cos(clustered_data['Elevation']) * np.cos(clustered_data['Azimuth'])
clustered_data['y'] = clustered_data['Range'] * np.cos(clustered_data['Elevation']) * np.sin(clustered_data['Azimuth'])
clustered_data['z'] = clustered_data['Range'] * np.sin(clustered_data['Elevation'])
```

```
def plot_frame(frame):
  fig = plt.figure(figsize=(8, 8))
  ax = fig.add_subplot(111, projection='3d')
  clustered_data_frame = clustered_data[clustered_data['frameNum'] == frame]
  df_scale = clustered_data_frame[["x", "y", "z", "cluster"]]
  scatter = ax.scatter(
      df_scale['x'],
      df_scale['y'],
      df_scale['z'],
      c=df_scale['cluster'],
      cmap='viridis',
      marker='o',
      s=50)

  legend = ax.legend(*scatter.legend_elements(), title="Clusters")
  ax.add_artist(legend)
  ax.set_xlabel('x')
  ax.set_ylabel('y')
  ax.set_zlabel('z')
  ax.set_title(f'Frame {frame}')
  plt.show()

#슬라이더
frame_slider = widgets.IntSlider(
    value=1,
    min=clustered_data["frameNum"].min(),
    max=clustered_data["frameNum"].max(),

    step=1,
    description='Frame'
)
interact(plot_frame, frame=frame_slider)
```

위는 각 프레임별로 point가 각 프레임에서 어떤 형태로 분포되어 있는지 시각화 하는 코드이

다.



이런 식으로 슬라이더를 사용해서 데이터의 분포를 확인했다.

아래는 이렇게 각 프레임마다 클러스터 된 clustered_data 데이터프레임의 모습이다

	frameNum	pointNum	Range	Azimuth	Elevation	Doppler	SNR	cluster	x	y	z
0	1	0	2.46575	-1.17	0.07	0.06972	9.240000	0.0	0.959661	-2.264781	0.172462
1	1	61	2.59225	-0.63	0.28	0.06972	7.120000	0.0	2.013036	-1.467734	0.716383
2	1	60	2.52900	-0.63	0.25	-0.20888	27.119999	0.0	1.979974	-1.443628	0.625685
3	1	59	2.46575	-0.63	0.25	-0.20888	52.639999	0.0	1.930455	-1.407523	0.610036
4	1	58	2.27625	-0.61	0.17	-0.20888	30.999999	0.0	1.838829	-1.285192	0.385101
...	...	...	...	...	...	...	...	...	...	...	...
3564	43	76	3.79350	0.60	0.12	0.55664	18.120000	1.0	3.108395	2.126568	0.454128
3565	43	77	3.85675	0.60	0.12	0.48720	6.360000	1.0	3.160222	2.162024	0.461700
3566	43	78	3.60400	0.61	0.15	0.55664	26.999999	1.0	2.920841	2.041431	0.538575
3567	43	70	3.66725	0.59	0.15	0.20888	34.639999	1.0	3.013050	2.017404	0.548027
3568	43	101	3.73025	0.68	0.15	0.48720	19.520000	1.0	2.867971	2.319217	0.557

이 데이터 프레임은 같은 frameNum에서의 cluster가 다음 프레임에서의 cluster와 아무 연관이 없다는 문제가 있다. 이 문제를 처리하기 위해 각 프레임마다 클러스터들이 이전 프레임의 어떤 클러스터와 연결되어 있는지 알려주는 알고리즘을 만들기로 했다.

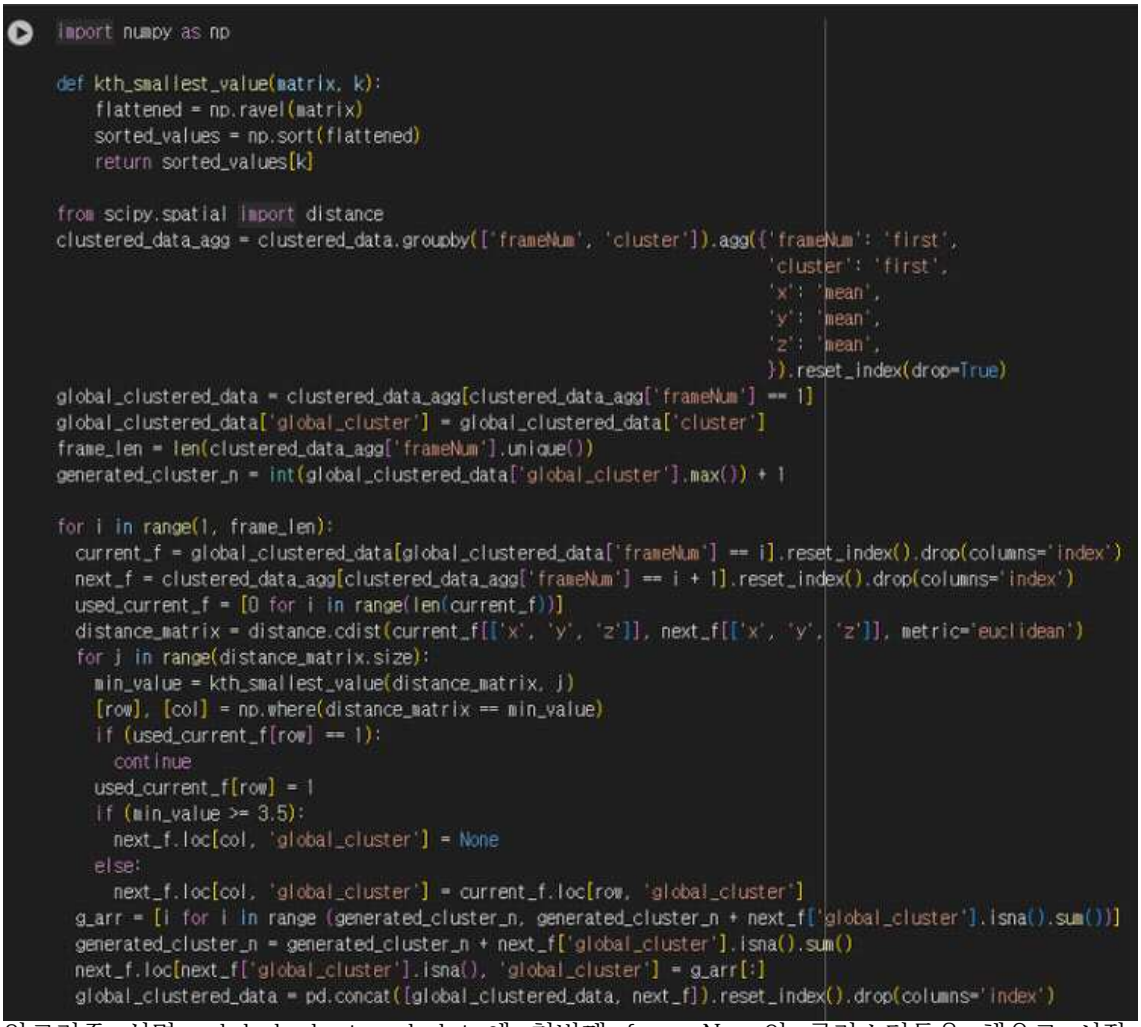
클러스터된 데이터를 ‘frameNum’, 그 안에서 ‘cluster’ 기준으로 나머지 특징들을 평균한다. 그렇게 한 프레임에서 각 클러스터를 대표하는 값들을 가진 행들을 담은

```
clustered_data_agg = clustered_data.groupby(['frameNum', 'cluster']).agg({'frameNum': 'first',
                                                                          'cluster': 'first',
                                                                          'Range': 'mean',
                                                                          'Azimuth': 'mean',
                                                                          'Elevation': 'mean',
                                                                          'Doppler': 'mean',
                                                                          'SNR': 'mean'}).reset_index(drop=True)
```

clustered_data_agg를 만든다

	frameNum	cluster	x	y	z	global_cluster
0	1	0.0	1.705726	-1.601542	0.410367	0.0
1	1	1.0	4.184639	1.469327	0.343905	1.0
2	1	2.0	2.834504	1.657704	0.651839	2.0
3	2	0.0	1.887561	-1.477973	0.454277	0.0
4	2	1.0	4.148217	1.509764	0.703590	1.0
...	...	...	...	...	...	...
89	41	1.0	2.863386	1.968592	0.513841	19.0
90	42	0.0	2.069335	-1.096709	0.559938	17.0
91	42	1.0	3.051533	1.883823	0.552724	19.0
92	43	0.0	2.090437	-1.097339	0.675624	17.0
93	43	1.0	3.042724	2.068884	0.518595	19.0

하고자 하는 것은 원래 clustered_data의 각 행의 클러스터가 이전 프레임의 n번째 클러스터(‘global cluster’ 값이 n)와 이어졌을 때 그 n을 column ‘global_cluster’에 담는 global_clustered_data 데이터프레임을 만드는 것이다. 만약 새로운 클러스터로 생각되는 클러스터가 생성된다면 지금 까지 없던 클러스터 번호를 지금 까지 있던 최대 클러스터 번호에 1을 더해서 할당한다. 원래는 사이킷런이 제공하는 헝가리안 알고리즘 함수를 사용했으나 내부 구현이 어떻게 되어있는지 모르고 설명할 방법도 잘 모르겠어서 직접 구현했다. 아래에 내용이 나와있다.



```
import numpy as np

def kth_smallest_value(matrix, k):
    flattened = np.ravel(matrix)
    sorted_values = np.sort(flattened)
    return sorted_values[k]

from scipy.spatial import distance
clustered_data_agg = clustered_data.groupby(['frameNum', 'cluster']).agg({'frameNum': 'first',
                                                                          'cluster': 'first',
                                                                          'x': 'mean',
                                                                          'y': 'mean',
                                                                          'z': 'mean',
                                                                          }).reset_index(drop=True)
global_clustered_data = clustered_data_agg[clustered_data_agg['frameNum'] == 1]
global_clustered_data['global_cluster'] = global_clustered_data['cluster']
frame_len = len(clustered_data_agg['frameNum'].unique())
generated_cluster_n = int(global_clustered_data['global_cluster'].max()) + 1

for i in range(1, frame_len):
  current_f = global_clustered_data[global_clustered_data['frameNum'] == i].reset_index().drop(columns='index')
  next_f = clustered_data_agg[clustered_data_agg['frameNum'] == i + 1].reset_index().drop(columns='index')
  used_current_f = [0 for i in range(len(current_f))]
  distance_matrix = distance.cdist(current_f[['x', 'y', 'z']], next_f[['x', 'y', 'z']], metric='euclidean')
  for j in range(distance_matrix.size):
    min_value = kth_smallest_value(distance_matrix, j)
    [row], [col] = np.where(distance_matrix == min_value)
    if (used_current_f[row] == 1):
      continue
    used_current_f[row] = 1
    if (min_value >= 3.5):
      next_f.loc[col, 'global_cluster'] = None
    else:
      next_f.loc[col, 'global_cluster'] = current_f.loc[row, 'global_cluster']
  g_arr = [i for i in range (generated_cluster_n, generated_cluster_n + next_f['global_cluster'].isna().sum())]
  generated_cluster_n = generated_cluster_n + next_f['global_cluster'].isna().sum()
  next_f.loc[next_f['global_cluster'].isna(), 'global_cluster'] = g_arr[:]
  global_clustered_data = pd.concat([global_clustered_data, next_f]).reset_index().drop(columns='index')
```

알고리즘 설명: global_clustered_data에 첫번째 frameNum의 클러스터들을 채우고 시작. 반복문 안에서 global_clustered_data의 i번째 frame에 해당하는 행들과 clustered_data_agg의 i + 1번째 frame에 해당하는 행들을 뽑아낸다. 전자는 current_f라는 변수에, 후자는 next_f라는 변수에 저장한다. 이제 next_f라는 변수에 있는 클러스터들에 대해 current_f에 있는 글로벌 클러스터를 연결 시켜주면 된다.

연결을 하는 방법은 예시로 설명하겠다. current_f에 3개의 global_cluster가 각각 1, 2, 3인 클러스터들이 있고 next_f에는 global_cluster가 아직 할당되지 않은 2개의 클러스터가 있다고 하자. 사이킷런의 distance 함수를 써서 current_f의 각 클러스터 중심의 x,y,z와 next_f의 각 클러스터 중심의 x,y,z의 거리 행렬을 구한다. 거기서 거리가 작은 순서대로 할당을 한다. next_f의 두번째 클러스터가 current_f의 global_cluster가 1인 클러스터와 거리 행렬의 값이 가장 작다고 하자. 그러면 next_f의 두번째 클러스터의 global_cluster 값을 1로 하고 다음으로 거리 행렬의 값이 가장 작은 값으로 넘어가서 계속 할당하는 것이다. 그런데 이렇게 하면 next_f의 클러스터들이 current_f의 클러스터보다 많을 때 무조건 원래 있는 next_f의 클러스터에 할당이 되는 문제가 생긴다. 이를 막기 위해 거리 행렬의 값이 일정 이상이면 global_cluster 할당을 하지 못하게 하는 코드를 추가했다. min_value >= 3.5가 이 부분인데 이 값도 휴리스틱하게 찾아서 넣었다. 그리고 global_cluster가 할당 되지 않은 클러스터들은 지금까지 나오지 않은 클러스터 값으로 나오게 하도록 만들었다. 예를 들어 지금까지 나온 클

러스터중 가장 큰 클러스터가 5면 아직 global_cluster가 할당되지 않은 global_cluster를 6부터 할당하는 것이다.

	frameNum	cluster	x	y	z	global_cluster
0	1	0.0	1.705726	-1.601542	0.410367	0.0
1	1	1.0	4.184639	1.469327	0.343905	1.0
2	1	2.0	2.834504	1.657704	0.651839	2.0
3	2	0.0	1.887561	-1.477973	0.454277	0.0
4	2	1.0	4.148217	1.509764	0.703590	1.0
...	...	...	...	...	...	...
89	41	1.0	2.863386	1.968592	0.513841	19.0
90	42	0.0	2.069335	-1.096709	0.559938	17.0
91	42	1.0	3.051533	1.883823	0.552724	19.0
92	43	0.0	2.090437	-1.097339	0.675624	17.0
93	43	1.0	3.042724	2.068884	0.518595	19.0

이렇게 나온 global_cluster 컬럼이 추가된 데이터이다. 이전 프레임에서의 연결되지 않은 클러스터는 계속해서 새로운 번호로 할당되기 때문에 클러스터 번호가 계속해서 늘어날 것이다.

```
clustered_data = pd.merge(clustered_data, global_clustered_data[['cluster','frameNum','global_cluster']], on=['cluster', 'frameNum'], how='left')

clustered_data.drop(columns=['cluster'])
```

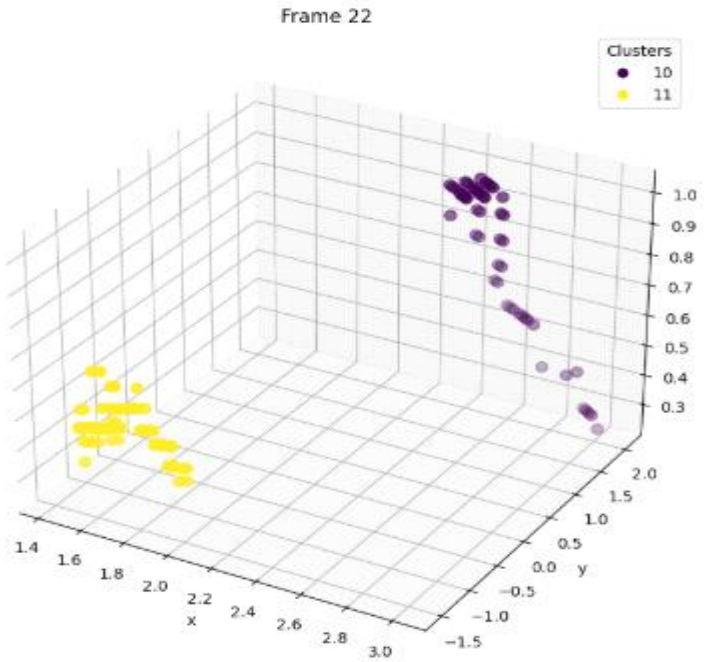
	frameNum	pointNum	Range	Azimuth	Elevation	Doppler	SNR	x	y	z	global_cluster
0	1	0	2.46575	-1.17	0.07	0.06972	9.240000	0.959661	-2.264781	0.172462	0.0
1	1	61	2.59225	-0.63	0.28	0.06972	7.120000	2.013036	-1.467734	0.716383	0.0
2	1	60	2.52900	-0.63	0.25	-0.20888	27.119999	1.979974	-1.443628	0.625685	0.0
3	1	59	2.46575	-0.63	0.25	-0.20888	52.639999	1.930455	-1.407523	0.610036	0.0
4	1	58	2.27625	-0.61	0.17	-0.20888	30.999999	1.838829	-1.285192	0.385101	0.0
...	...	...	...	...	...	...	...	...	...	...	...
3564	43	76	3.79350	0.60	0.12	0.55664	18.120000	3.108395	2.126568	0.454128	19.0
3565	43	77	3.85675	0.60	0.12	0.48720	6.360000	3.160222	2.162024	0.461700	19.0
3566	43	78	3.60400	0.61	0.15	0.55664	26.999999	2.920841	2.041431	0.538575	19.0
3567	43	70	3.66725	0.59	0.15	0.20888	34.639999	3.013050	2.017404	0.548027	19.0
3568	43	101	3.73025	0.68	0.15	0.48720	19.520000	2.867971	2.319217	0.557442	19.0

3569 rows × 11 columns

이 과정을 통해 모든 pointNum들에 global_cluster를 할당해준다.

```python
def plot_frame(frame):
  fig = plt.figure(figsize=(8, 8))
  ax = fig.add_subplot(111, projection='3d')
  clustered_data_frame = clustered_data[clustered_data['frameNum'] == frame]
  df_scale = clustered_data_frame[["x", "y", "z", "global_cluster"]]
  scatter = ax.scatter(
      df_scale['x'],
      df_scale['y'],
      df_scale['z'],
      c=df_scale['global_cluster'],
      cmap='viridis',
      marker='o',
      s=50)

  legend = ax.legend(*scatter.legend_elements(), title="Clusters")
  ax.add_artist(legend)
  ax.set_xlabel('x')
  ax.set_ylabel('y')
  ax.set_zlabel('z')
  ax.set_title(f'Frame {frame}')
  plt.show()
#슬라이더
frame_slider = widgets.IntSlider(
    value=1,
    min=clustered_data["frameNum"].min(),
    max=clustered_data["frameNum"].max(),
    step=1,
    description='Frame'
```

그러면 위와 같은 코드로 global_cluster을 보는 plot을 그릴 수 있다.



이것이 우리가 사용했던 클러스터링 방식이다.

4. 모델 학습 과정 설명

여기서는 위에서 클러스터링 된 한 객체에 대해서 그 객체의 point들을 모델에 학습시키고 평가하는 과정에 대한 코드를 설명한다.

이에 대한 코드는 우리가 작업했던 깃허브에 **src/ai/모델_학습_및_평가.ipynb** 이 경로로 저장되어 있다. 이 코드에서 사용하는 데이터는 어떤 한 프레임에 하나의 객체만 들어있을 때에 대한 데이터이다. 클러스터링을 하고 나면 각 프레임에서 한 객체를 뽑아낼 수 있으니 그 과정을 거치고 나서의 데이터를 넣었다고 생각하면 이는 아무 문제가 없다. 이에 대한 처리는 실제 센서에서 사용하는 코드인 **src/enhancedVisualizer/fall_detection.py**에서 이 코드를 변형하여 구현해놓았다. 어떤 변형이 들어갔는지는 내가 하지 않아서 잘 모르겠지만 그 부분을 맡은 분이 설명해 놓았을 것이다. 코드의 앞부분은 클러스터링 설명 부분에서 데이터를 정렬하고 평균했던 부분과 똑같다. 그래서 그 부분을 다 생략하고 시작하겠다.

```
from sklearn.preprocessing import OneHotEncoder

data_x = [] #(5, 5) 크기의 시계열 데이터를
data_y = [] #원핫인코딩된 4개의 레이블을 담는다
for dir in dirs:
  csvs = os.listdir(path + "/" + dir)
  for csv in csvs:
    if re.search(r".ipynb_checkpoints", csv): #코랩에서 돌리면 계속 숨은 이 확장자 붙은 파일 생겨서 오류생겨서 예외처리
      continue
    csv_data = pd.read_csv(path + "/" + dir + "/" + csv)
    csv_data = csv_data.drop(columns=["Azimuth", "Elevation", "Range"])
    size = 5
    stride = 1
    for i in range (0, len(csv_data) - size + 1, stride):
      if 'fall' in dir:
        data_y.append([1, 0, 0, 0])
      elif 'sit' in dir:
        data_y.append([0, 1, 0, 0])
      elif 'sleep' in dir:
        data_y.append([0, 0, 1, 0])
      elif 'walk' in dir:
        data_y.append([0, 0, 0, 1])
      else:
        continue
      data_x.append(csv_data[i:i+size])
print(data_x[0])

#밑은 넘파이 arr input 아니면 gru가 안돌아가서 그냥 변환해주는거임
for i in range(len(data_x)):
  data_x[i].drop(columns=['frameNum', 'Unnamed: 0'], inplace=True)
  data_x[i] = data_x[i].to_numpy()
data_x = np.array(data_x)
data_y = np.array(data_y)
```

이 코드의 구체적인 설명은 필요없고 data_x에는 길이가 window_size인 연속된 프레임의 x, y, z, Doppler, SNR인 데이터들을 stride를 임의의 값으로 해서 쭉 넣어주면 된다. 그리고 그 각각 데이터에는 원핫 인코딩으로 fall, sit, sleep, walk 4개의 레이블을 data_y에 넣었다. 코드는 여러 엑셀 파일에 있던 데이터를 하나의 엑셀로 묶어서 사용한다고 이상한 부분이 생겼는데 필요 없고 data_x, data_y에 이런 형식으로 데이터가 들어가기만 하면 된다. 그리고 gru모델에 사용할 수 있도록 데이터를 numpy arr로 바꿔준다.

```
##모델 만들기
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import GRU, Dense, Dropout
from tensorflow.keras.optimizers import Adam
from sklearn.model_selection import train_test_split

X_train, X_test, Y_train,Y_test = train_test_split(data_x, data_y,test_size=0.25,random_state=42)
X_train = np.array(X_train)
Y_train = np.array(Y_train)

model = Sequential()
model.add(GRU(32, reset_after=False, return_sequences=True, input_shape=(5, 5)))
model.add(Dropout(0.2))
model.add(GRU(64, reset_after=False, return_sequences=False))
model.add(Dropout(0.3))
model.add(Dense(4, activation='softmax'))

model.compile(loss='categorical_crossentropy', optimizer=Adam(learning_rate=0.001), metrics=['accuracy'])
history = model.fit(X_train, Y_train, epochs=100, batch_size = 16, validation_data=(X_test, Y_test))
```

이렇게 만든 데이터를 학습 데이터 75퍼센트, 평가 데이터 25퍼센트로 나눠서 GRU 모델에 넣었다. 우리가 사용한 데이터는 996개로 약 1000개였다. GRU를 사용한 이유는 센서로 들어오는 데이터를 프레임 별로 끊어서 모델에 넣는 것보다 시계열 데이터로 처리해야 이 동작이 어떤 동작이다 라는 것을 알아 차릴 수 있을 것이라 판단했고 시계열 데이터를 처리하는데 에는 GRU가 정확도와 성능 면에서 좋다고 생각했기 때문이다.

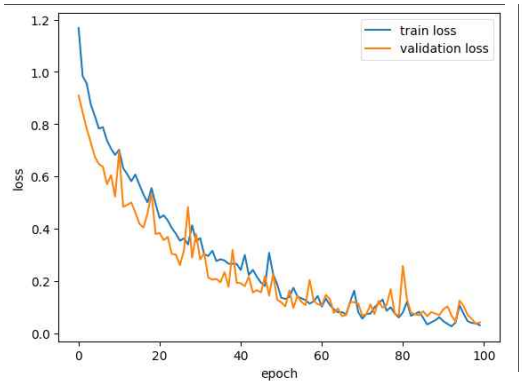
모델의 input으로는 우리가 선택한 (특징 크기 * 데이터의 길이) 크기의 데이터가 들어간다. 여기서는 5*5 크기가 된다. 모델은 32 , 64개의 은닉층과 층에 0.2, 0.3 드랍아웃을 적용했다. output으로는 4개 레이블 각각에 대한 확률값이 softmax 활성함수를 거쳐서 나온다.

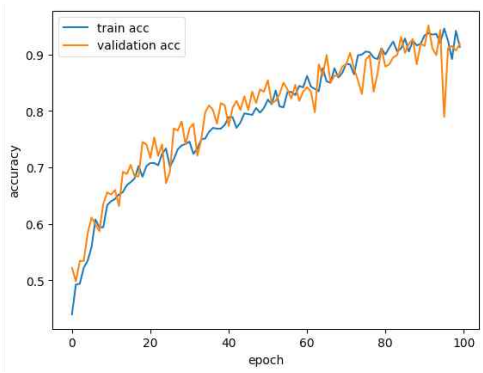
손실 함수는 categorical_crossentropy를 썼고 Adam 옵티마이저를 학습률 0.001로 해서 학습했다. 에포크는 100, batch_size = 16으로 했다. 모든 값들은 휴리스틱한 방식으로 찾아졌다.

```
from sklearn.metrics import f1_score
from sklearn.metrics import recall_score
from sklearn.metrics import precision_score

model.evaluate(x=X_test, y=Y_test, batch_size=16, verbose=1, sample_weight=None, steps=None, callbacks=None)
y_pred = model.predict(X_test)
y_pred = np.argmax(y_pred, axis=1)
y_test = np.argmax(Y_test, axis=1)
print('f1_score: ', f1_score(y_test, y_pred, average='macro'))
print('precision_score: ', precision_score(y_test, y_pred, average='macro'))
print('recall_score: ', recall_score(y_test, y_pred, average='macro'))
```

```
12/12 ━━━━━━━━━━━━━━━━━━━━ 0s 4ms/step - accuracy: 0.9956 - loss: 0.0383
6/6 ━━━━━━━━━━━━━━━━━━━━ 1s 80ms/step
f1_score:  0.989367750547009
precision_score:  0.9927395486496485
recall_score:  0.9861694677871149
```

모델의 성능 지표들에 대해 평가한 것들이다. 매우 높게 나왔지만 한 프레임에 대해 한 물체만 나온다고 가정한 것이라는 점과 데이터가 다양하지 않고 양도 적다는 점에서 모델이 과대적합되서인지 평가지표의 값이 매우 높게 나왔다.





뒤 코드 부분은 pycaret을 이용해서 다른 모델들을 비교해 본 것인데 시계열 데이터를 사용하지 않아서 (5, 5) 데이터를 (25, 1) 길이로 풀어서 그냥 넣는 꼴이다. 때문에 시계열의 길이가 늘어나면 적합하지 않을 것으로 예상되어 사용하지 않았다. 따라서 무시하면 된다.

5. 문제점 및 분석

- 1. 부족한 데이터와 이에 따른 적절하지 않은 데이터의 window size :

각 파일을 읽어서 frameNum 단위로 window size = 5, stride = 2의 5 * 5 시계열 데이터를 추출했다. 이렇게 해서 데이터가 1000개 정도 나왔는데 시계열의 크기를 더 늘리면 데이터가 너무 적어져서 학습이 너무 안될 것 이라고 생각해서 크기를 5로 했다. 하지만 시계열의 크기를 5로 하면 0.275초 정도의 데이터가 모델의 입력으로 들어가는 것이기 때문에 문제가 된다. 이것에 우리가 만든 결과물에서 너무 자세 전환이 빨랐던 이유라고 생각한다. 그래서 최소한 1초 정도의 정보를 담을 수 있게 크기를 최소한 30으로 해야 할 것으로 보인다. 그리고 데이터마다 겹치는 부분이 생기게 데이터를 만드는 것이 잘못된 방법이라고 생각한다. 데이터가 겹치는 부분이 생기면 모델 학습 때 loss를 적게 만들도록 학습 하기만 하는 모델의 특성상 정확도가 높게 나올 수 밖에 없다. 하지만 이는 올바른 모델이 아니다. 우리는 데이터가 적어서 시계열마다 겹치는 부분이 생기도록 해서 많은 데이터를 만들었지만 다른 팀은 데이터마다 겹치는 부분이 없도록 시계열을 만들기를 바란다.

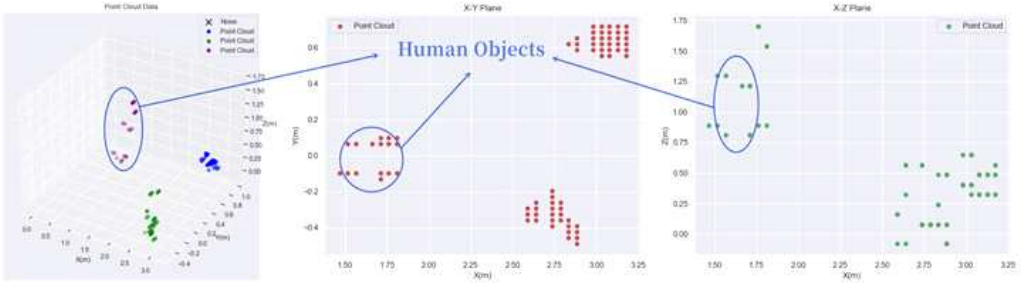
- 2. 학습에 사용하는 특징 컬럼의 적절성 의문 :

학습 때 csv 파일의 형식이 기존 V, x, y, z, Height만 사용하던 것에서 pointNum 단위의 csv 파일로 바꾼 후에 다시 x,y,z로 변환했다. 그렇게 하면 추가적인 SNR, Doppler 컬럼을 사용할 수 있어서 딥러닝 학습 과정에서 더 효과적일 것이라고 생각했다. 하지만 실제로 그런지는 잘 알 수 없다. x,y,z 변환도 센서 회사에서 수학적 검증을 통해 만들어진 데이터인데 우리는 pointNum 자체를 frame 단위로 평균해서 쓰기 때문에 이게 더 성능이 안 좋을 수도 있다. 이를 알아봐야 할 것 같다.

- 3. DBSCAN 직접 구현의 필요성

사이킷런 제공 DBSCAN은 좀더 일반적인 데이터까지 클러스터링 할 수 있게 구현이 되어있는지 직접 구현한 것과 비교해서 속도가 매우 느리다. 200개의 특징 2개의 데이터를 사용했을 때 직접 구현한 것에 비해 속도가 40배가 넘게 차이 난다는 글도 있다. https://zephyrus1111.tistory.com/356 어차피 visualizer에 쓰는 DBSCAN은 어떤 특징을 쓰는지 확실하게 나오기 때문에 그 특징 수랑 데이터 수에 맞게 정확하게 구현하면 속도가 더

빨라질 것이다.



- 4. DBSCAN 구현 과정에서 Z축 (vertical)에 대한 가중치 감소시키기

이 그림을 보면 클러스터에서 x-y 평면에서보다 x-z 평면에서 point들의 일관성이 없는 모습을 볼 수 있다. 따라서

$$D^2(p^i, p^j) = (p_x^i - p_x^j)^2 + (p_y^i - p_y^j)^2 + \alpha(p_z^i - p_z^j)^2$$

이 식처럼 z축에 가중치를 부여해서 유클리드 거리를 구해서 DBSCAN을 구현하는 것이 필요해보인다

- 5. Tracking 알고리즘 개선하기

프레임 별로 클러스터들을 이어주는 방법을 지금은 그냥 연속된 두 프레임에 대해서 클러스터 중심의 유클리디안 거리가 가장 가까운 클러스터끼리 같은 클러스터라고 생각하고 이어지게 했다. 당연히 성능이 좋지 않다.

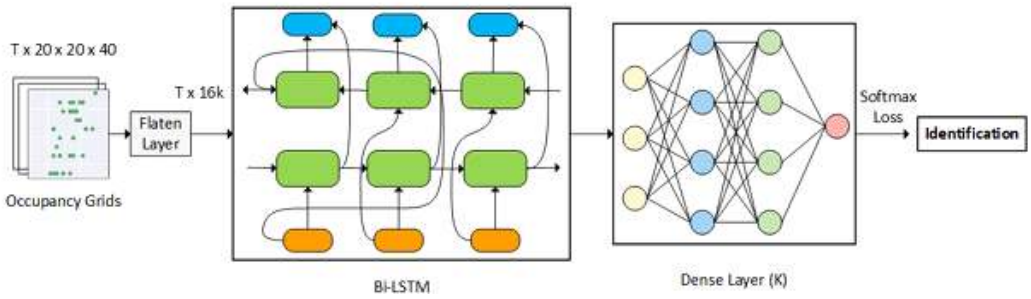
논문[Application of mmWave Rador Sensor for People Identification and Classifition]에서 사용한 프로그램에서는 simplified Global Nearest Neighbour 알고리즘을 써서 트랙을 이어주고 정확한 트랙 예측을 위해서 칼만 필터를 사용했다고 한다. 하지만 논문에서도 이 부분을 그냥 이 알고리즘을 썼다 하고 넘겨버렸다... 그래서 이 부분을 찾아서 구현해보면 좋을 것이다. 이것 외에도 다른 tracking 방법들이 있을 것이다. 다음 팀은 이쪽 부분을 생각해서 만들어주면 좋겠다.

- 6. 외부로부터 들어오는 새로운 물체를 새로운 물체로 인식하기

지금은 외부에서 새로운 물체가 들어올 때 그 물체가 원래 이 안에 있던 물체인지 새로 들어온 물체인지 인식 하는게 알 수가 없게 되어있다. 이런 intruder 판단을 하는 방법에 대한 논문[Application of mmWave Rador Sensor for People Identification and Classifition]에서는 손실 함수에 center loss라는 값을 추가해서 random forest classifier에 넣는 방식으로 해결했다고 한다. 아래는 자세한 설명이다. 설명이 부족하다면 논문을 읽어보는걸 추천한다.

자세한 설명 : detection과 tracking 과정을 통해서 track된 points들이 나왔을 것이다. 만약 각 객체가 경계 박스 안에 있다고 하고 이 안에 그 객체의 점들이 있다고 하면 객체를 각 trajectory frame에서 점유 그리드안에 있다고 할 수 있다. 이 점유 그리드는 예를 들면 키나 중심의 밀도같은 사람의 정보를 담고 있을 것이다. 여기에 사람 몸의 정보같은 내용을 함께 분류기에 입력해서(레이블링을 뜻하는 듯 하다) tracked object의 identity를 결정할 수

있다. 하지만 직접 이 점유 그리드에서 사람의 정보를 추출하는 것은 비효율적이다. 따라서 여기서 양방향 LSTM을 사용한다.



점유 그리드 안의 점들을 flatten해서 양방향 LSTM에 넣는다. LSTM의 결과는 Dense Layer에 들어가고 softmax loss를 통해 classification 결과가 나오게 된다. 여기서 k는 클러스터 개수고 T는 data frame의 개수다. K는 layer size이다.

그런데 out-of-set samples는 intruder로 처리해야 하는데 방금 identification에서 사용한 softmax 층은 다항분포로 밖에 intruder를 예측하지 못한다. scattered features에 대한 softmax loss는 intruder에 구별에 악영향을 미친다, 따라서 center loss를 사용한다.
방법은 아까의 신경망에서 샘플 특징을 뽑아내서 intra-class 거리를 center loss로 최소화하는 것이다.

softmax와 center loss의 조합은 아래 식과 같다.

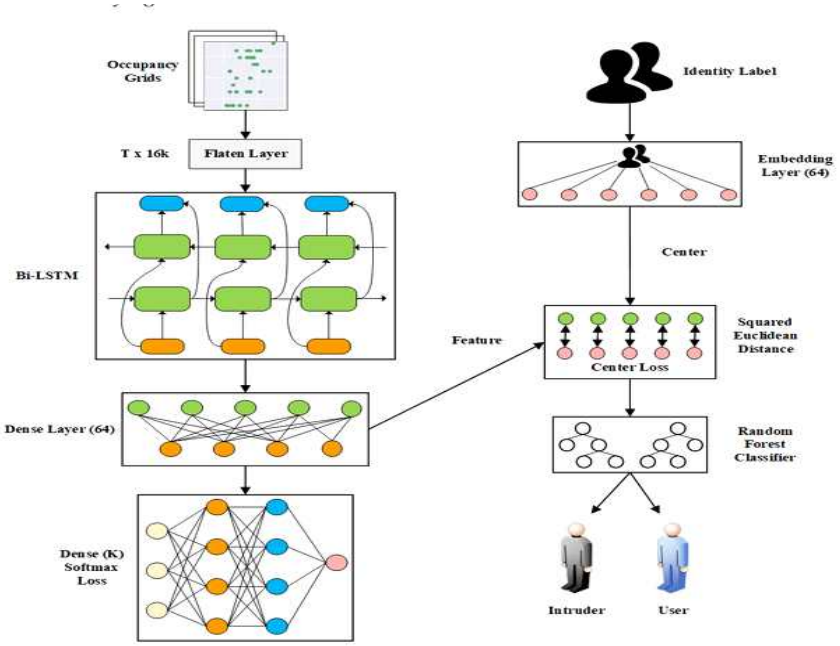
$$\ell = \ell_{softmax} + \lambda \ell_{center}$$

람다는 softmax와 center loss의 균형을 맞춰주는데 여기서는 0~0.1 사이에서 모델을 학습했다고 한다.

$$\ell_{softmax} = -\sum_{i=1}^{m} \log \frac{e^{W_{y_i}^T \mathbf{f}_i} + b_{y_i}}{\sum_{j=1}^{g} e^{W_j^T \mathbf{f}_i} + b_j},$$

$$\ell_{center} = \sum_{i=1}^{m} \|\mathbf{f}_i - \mathbf{c}_{y_i}\|_2^2$$

softmax와 center loss에 대한 식이다. g는 m개 샘플의 학습 데이터중 객체의 개수를 의미하고 f는 특징이다. c는 특징의 중심을 나타내고 W,b는 가중치와 bias이다.



아까 그림에서와 다르게 양방향 LSTM 바로 직후의 Dense layer에서 클러스터 특징을 뽑아낸다. 객체의 ID를 객체의 중심에 레이블링 하기 위해서 임베딩을 사용하는데, inner 클러스터의 임베딩된 특징과 이 뽑아낸 특징의 유클리드 거리를 통해 center loss를 알아낸다. 이를 softmax와 조합한 loss를 이용해서 Random forest classifier에 넣어서 이것이 일정 값보다 크면 intruder로 판단할 것이다. 이렇게 intruder를 알아낼 수 있다.

6. 향후 작업에 대한 방향

[Real-time mmWave Multi-Person Pose Estimation System for Privacy-Aware Windows](https://github.com/AsteriosPar/mmWave_MSc?tab=readme-ov-file)

- 단순화된 GTRACK 알고리즘과 MARS 모델을 결합해서 만든 여러 사람 상세 인식 프로그램이다. 이 프로그램과 같은 동작을 하는 낙상 감지 프로그램을 만드는 것이 목표다.

GTRACK 알고리즘은

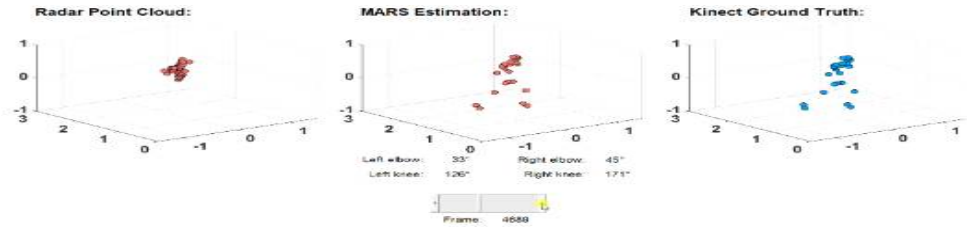
1. 센서 안에서 특정 지역 안에서만 데이터를 처리하도록 지역 밖의 데이터를 DROP함. 그러니까 visualizer에서 네모 박스 안의 point들만 처리한다 그 얘기다.
2. csv에 들어가는 특정 프레임 값을 구하기 위해 이전 프레임의 데이터와 실제 얻어진 이번 프레임의 데이터를 칼만 필터를 사용함. 그렇게 균형이 맞춰진 데이터가 이번 프레임에 실제로 들어감.
3. 클러스터링을 위해 DBSCAN 사용

이 단계를 거쳐서 여러 물체에 대한 인식을 우리가 구현한 방법과 유사하게 구현하여 이미 Visualizer 자체에서 사용되고 있다.

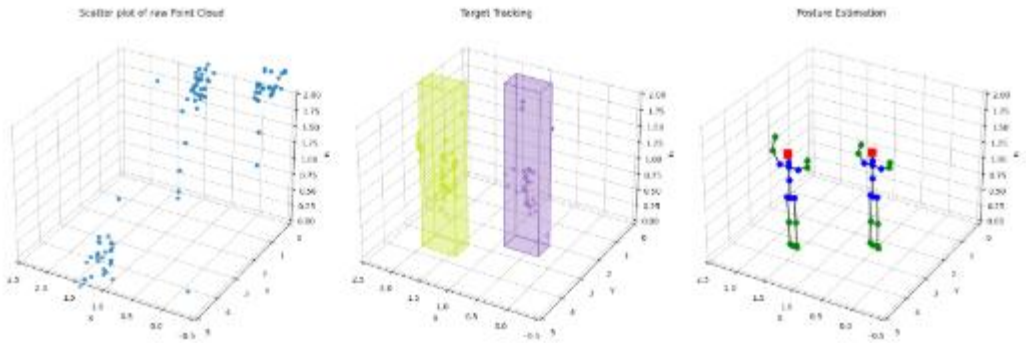
https://dev.ti.com/tirex/explore/node?node=A__AYZwK7t1GX7lsaN.HegOQw__RADAR-

ACADEMY__GwxShWe__LATEST 이 링크에서 더 자세한 내용을 읽을 수 있다.

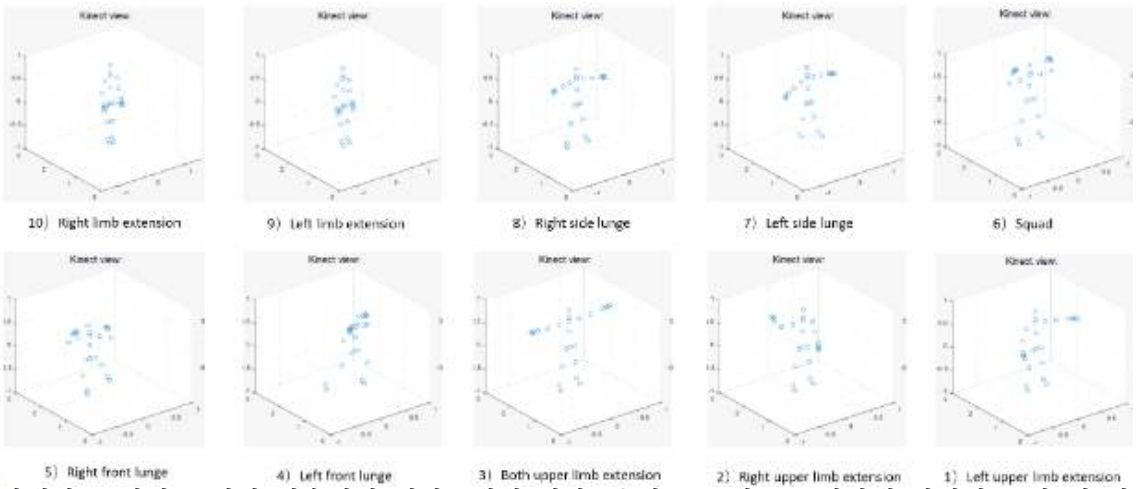
MARS 모델은 센서로 들어온 알 수 없는 형태의 Point Cloud를 Ground Truth 레이블 Point cloud로 학습시켜서 새로 들어오는 데이터를 Estimation Point Cloud로 변환한다. 아래 그림에 첫 번째 사진과 세 번째 사진을 연결시켜서 학습해서 새로운 데이터가 들어오면 두 번째 사진 형태로 나타내게 하는 모델이 MARS 모델이다.



정확한 내용은 논문에 나와있지만 읽어보지 않았다.



그래서 다음 프로젝트의 목표로 보는 것은 지금 센서에서 받는 여러 객체에 대한 point cloud가 들어올 때 위 그림과 같은 형태로 그 point cloud들을 클러스터링 하고 각각에 대한 스켈레톤 shape를 나타내는 것이다. 그럼 학습 다 되어있는 이 모델 쓰면 되는거 아닌가 할 수 있다. 문제는 MARS 모델은



아래의 10가지 동작만 학습되어 있다는데에 있다. 우리 프로젝트는 낙상에 관련된 동작 및 다른 동작들을 추가적으로 학습하여야 하기 때문에 이를 위해 MARS 모델의 구조를 파악하고 필요가 있다. 그리고 위에서 나열한 문제점들을 해결하는 것이 향후 프로젝트의 목표가 될 것이라고 생각한다.