모바일 기기 도난 시 센서 데이터를 이용한 범죄자의 신장 추론

201920871 윤상훈

1. 주제 선정 이유

모바일 기기의 보급이 증가함에 따라, 이들 기기의 도난 사건도 빈번해지고 있습니다. 이러한 상황에서 도난된 기기를 신속히 추적하고 범인을 식별하는 것은 매우 중요합니다. 최근의 기술 발전으로 인해 모바일 기기는 다양한 센서를 탑재하여 사용자의 움직임과 행동을 추적할 수 있게 되었습니다. 기기가 움직일 때 수집되는 가속도, 중력, 회전 속도, 그리고 자세 데이터는 사용자의 체형과 성별을 정보를 추론하는 데 중요한 단서를 제공합니다.

본 프로젝트에서는 기기가 도난당했을 때 기기 내부 센서에서 수집된 데이터를 활용하여 범인의 체형, 그중 신장을 추론하는 방법을 탐구합니다. 제공된 데이터셋은 24명의 다양한 신체 정보를 가진 참가자들이 앉기, 걷기, 계단 오르기 및 내리기 등의 일상적인 동작을 수행하는 동안 수집된 센서 데이터를 포함합니다. 위 데이터 중 걷기와 달리기(조깅)에 대한 데이터를 분석하여 개인의 신체적 특징(신장)을 정확하게 식별할 수 있는지 평가하고, 도난 사건에서 잠재적인 범인을 추적하는 데 활용할 수 있는지를 조사합니다.

하지만 본 주제에는 몇 가지 한계가 존재합니다. 먼저, 기기를 소지하고 있는 사람의 정보를 특정하기에는 표본 크기가 24명으로 한정되어 있어 사용자의 신장을 추론하는 데 한계가 있을 수 있습니다. 또한, 이러한 기술이 오히려 범죄에 악용될 수 있다는 윤리적 문제도 제기될 수 있습니다. 그러나 다양한 센서 데이터를 통합하여 유의미한 결과를 도출할 것이며, 본 핸드폰의 주인만이 이러한 기능을 사용할 수 있도록 하는 등 적절한 방법을 통해 악용에 대한 문제를 해결할 수 있을 것입니다.

이러한 데이터 분석을 통해 도난 사건에서 범죄자 추적 및 식별 방법을 개선하고, 향후 보안 시스템의 발전에 기여할 수 있는 가능성을 모색합니다. 더 나아가, 더 큰 규모의 데이터를 수집하고 다양한 변수를 추가함으로써 이러한 모델을 더욱 발전시킬 수 있을 것입니다.

2. 수행 과정 개요

데이터셋에는 24개의 데이터 주체에 대해 6가지 동작(계단 오르기, 계단 내리기, 앉기, 서기, 걷기, 조깅하기)에 대한 센서 데이터로 구성되어있다. 본 프로젝트에는 걷기와 조깅 동작이 가장 핵심적인 동작이 될 것이다. 수집된 센서 데이터를 통해 걷기 동작인지, 조깅 동작인지를 구분하고 동작에 따른 신장을 구분할 것이다.

이와 같은 방법이 가능한 이유는 일반적으로 보폭은 신장에 비례하기 때문이다[1]. 이번 프로젝트에서는 구체적인 보폭의 길이를 직접 구하는 것이 아닌 서로 다른 보폭에서 발생하는 센서 데이터의 특징에 주목하여 신장을 예측할 것이다. 따라서, 예측 모델은 신장에 따라

---

1) https://www.verywellfit.com/set-pedometer-better-accuracy-3432895
위 문서에서는 여성의 경우 신장(inches)에 0.413, 남성의 경우 키에 0.415를 곱하면 대략적인 보폭을 얻을 수 있다고 한다.

비례하는 선형적인 모습을 관측할 수 있을 것으로 예상된다.

본격적인 분석에 앞서 수월한 분석을 위해 데이터 전처리 혹은 시각화하는 함수 등을 만들고 4개의 서로 다른 타입의 데이터(자세, 중력, 가속도, 회전) 데이터에 대한 비교 분석을 하여 적절한 데이터를 택할 것이다. 다만, 일반적으로 핸드폰 센서를 통해 걸음 수를 분석할 때, 가속도 센서를 사용함으로 가속도 데이터를 사용하는 것은 매우 권장될 것이다. 이러한 점에 유의하여 신장을 구분하기 위한 적절한 데이터를 택할 것이다.

전반적인 학습 과정에서 두 객체에 대한 데이터를 검증을 위해 분리할 것이다.[2] 모델 학습에 대한 모든 과정을 마치고 해당 데이터를 사용하여 결과를 확인하여 모델에 대한 최종 검증을 마칠 것이다.

3. 수행 과정

#데이터 로드

데이터를 가져오는 과정은 깃허브 공식 문서[3]의 주어진 코드를 수정하여 구성하였다.

get_ds_infos 함수

```
def get_ds_infos():
    """...
    dss = pd.read_csv("data_subjects_info.csv")
    print("[INFO] -- Data subjects' information is imported.")
    return dss
```

creat_time_series 함수

```
def creat_time_series(dt_list, act_labels, trial_codes, mode="mag", labeled=True):
    """...
    num_data_cols = len(dt_list) if mode == "mag" else len(dt_list*3)

    if labeled:
        dataset = np.zeros((0, num_data_cols + 7))  # "7" --> [act, code, weight, height, age, gender, trial]
    else:
        dataset = np.zeros((0, num_data_cols))

    ds_list = get_ds_infos()

    print("[INFO] -- Creating Time-Series")
    for sub_id in ds_list["code"]:
        for act_id, act in enumerate(act_labels):
            for trial in trial_codes[act_id]:
                fname = 'A_DeviceMotion_data/' + act + '_' + str(trial) + '/sub_' + str(int(sub_id)) + '.csv'
                raw_data = pd.read_csv(fname)
                raw_data = raw_data.drop(['Unnamed: 0'], axis=1)
                vals = np.zeros((len(raw_data), num_data_cols))
                for x_id, axes in enumerate(dt_list):
                    if mode == "mag":
                        vals[:, x_id] = (raw_data[axes] ** 2).sum(axis=1) ** 0.5
                    else:
                        vals[:, x_id * 3:(x_id + 1) * 3] = raw_data[axes].values
                    vals = vals[:, :num_data_cols]
                if labeled:
                    lbls = np.array([[act_id,
                                      sub_id - 1,
                                      ds_list["weight"][sub_id - 1],
                                      ds_list["height"][sub_id - 1],
                                      ds_list["age"][sub_id - 1],
                                      ds_list["gender"][sub_id - 1],
                                      trial
                                      ]] * len(raw_data))
                    vals = np.concatenate((vals, lbls), axis=1)
                dataset = np.append(dataset, vals, axis=0)
    cols = []
    for axes in dt_list:
        if mode == "raw":
            cols += axes
        else:
            cols += [str(axes[0][:-2])]

    if labeled:
        cols += ["act", "id", "weight", "height", "age", "gender", "trial"]

    dataset = pd.DataFrame(data=dataset, columns=cols)

    # id가 22와 23인 데이터를 제거하고 별도로 저장
    dataset_ver = dataset[(dataset['id'] == 22) | (dataset['id'] == 23)]
    dataset = dataset[(dataset['id'] != 22) & (dataset['id'] != 23)]

    return dataset, dataset_ver
```

set_data_types 함수

```
def set_data_types(data_types=["userAcceleration"]):
    """...
    dt_list = []
    for t in data_types:
        if t != "attitude":
            dt_list.append([t+".x",t+".y",t+".z"])
        else:
            dt_list.append([t+".roll", t+".pitch", t+".yaw"])

    return dt_list
```

dataset load 과정

```
act_labels = ACT_LABELS [3:4]
print("[INFO] -- == dataset_jog ==")
print("[INFO] -- Selected activites: "+str(act_labels))
trial_codes = [TRIAL_CODES[act] for act in act_labels]
dt_list = set_data_types(sdt)
dataset_jog, dataset_ver_jog = creat_time_series(dt_list, act_labels, trial_codes, mode="raw", labeled=True)
print("[INFO] -- Shape of time-Series dataset:"+str(dataset_jog.shape))
print("\n")
```

```
act_labels = ACT_LABELS [2:3]
print("[INFO] -- == dataset_wlk ==")
print("[INFO] -- Selected activites: "+str(act_labels))
trial_codes = [TRIAL_CODES[act] for act in act_labels]
dt_list = set_data_types(sdt)
dataset_wlk, dataset_ver_wlk = creat_time_series(dt_list, act_labels, trial_codes, mode="raw", labeled=True)
print("[INFO] -- Shape of time-Series dataset:"+str(dataset_wlk.shape))
print("\n")
```

2) id가 15.0, 22.0인 데이터

3) https://github.com/mmalekzadeh/motion-sense

creat_time_series 함수에 센서 데이터 타입과 가져오고 싶은 동작의 종류를 인자로 전달하여 호출하면 data_subjects_info.csv 파일에 저장된 사용자의 이름 정보를 읽어와 각 사용자에 대한 센서 정보를 가져옴과 동시에 사용자의 정보를 추가하여 dataset을 구성한다. creat_time_series 함수를 사용하여 모든 데이터를 동작별로 저장하였으며 변수는 dataset_xxx 형식으로 일관성 있게 생성하였다. 이때, 마지막 두 명에 대한 데이터를 분리하여 dataset_ver_xxx 형식으로 저장하도록 하였다.

#데이터 확인

정상적으로 수행되었는지 확인해보기 위해 각 데이터셋을 출력해보았다.

dataset_wlk

act	id	weight	height	age	gender	trial
0.0	0.0	102.0	188.0	46.0	1.0	7.0
0.0	0.0	102.0	188.0	46.0	1.0	7.0
0.0	0.0	102.0	188.0	46.0	1.0	7.0
0.0	0.0	102.0	188.0	46.0	1.0	7.0
0.0	0.0	102.0	188.0	46.0	1.0	7.0
...	...	...	...	...	...	...
0.0	23.0	74.0	173.0	18.0	0.0	15.0
0.0	23.0	74.0	173.0	18.0	0.0	15.0
0.0	23.0	74.0	173.0	18.0	0.0	15.0
0.0	23.0	74.0	173.0	18.0	0.0	15.0
0.0	23.0	74.0	173.0	18.0	0.0	15.0

dataset_ver_wlk

act	id	weight	height	age	gender	trial
0.0	15.0	96.0	172.0	29.0	0.0	7.0
0.0	15.0	96.0	172.0	29.0	0.0	7.0
0.0	15.0	96.0	172.0	29.0	0.0	7.0
0.0	15.0	96.0	172.0	29.0	0.0	7.0
0.0	15.0	96.0	172.0	29.0	0.0	7.0
...	...	...	...	...	...	...
0.0	22.0	68.0	170.0	25.0	0.0	15.0
0.0	22.0	68.0	170.0	25.0	0.0	15.0
0.0	22.0	68.0	170.0	25.0	0.0	15.0
0.0	22.0	68.0	170.0	25.0	0.0	15.0
0.0	22.0	68.0	170.0	25.0	0.0	15.0

dataset_wlk과 달리 dataset_ver_wlk의 경우 분리하였던 15.0, 22.0에 해당하는 id가 담겨있는 것을 확인할 수 있다.

#데이터 전처리 및 결측값 확인 함수 선언

split_dataset_by_id 함수

```python
def split_dataset_by_id(dataset, dataset_name):
    # 각 id별로 데이터를 그룹화하여 딕셔너리에 저장하는 함수
    datasets_by_id = {}

    for id, group in dataset.groupby('id'):
        group_reset = group.reset_index(drop=True)  # 인덱스를 초기화
        datasets_by_id[f'{dataset_name}_{id}'] = group_reset

    return datasets_by_id
```

split_by_trial 함수

```python
def split_by_trial(datasets_by_id):
    # 각 id별로 그룹화된 데이터에서 trial 별로 다시 그룹화하여 딕셔너리에 저장하는 함수
    datasets_by_id_and_trial = {}

    for name, data in datasets_by_id.items():
        id_part = name.split('_')[-1]
        for trial, group in data.groupby('trial'):
            key = f'{name}_trial_{trial}'
            datasets_by_id_and_trial[key] = group.reset_index(drop=True)

    return datasets_by_id_and_trial
```

nan_check 함수

```python
def nan_check(datasets_by_id):
    # 데이터프레임에 결측치가 있는지 확인하는 함수
    all_no_missing = True

    for name, data in datasets_by_id.items():
        missing_values = data.isnull().sum()
        if missing_values.any():
            all_no_missing = False
            print(f"Missing values in {name}:")
            print(missing_values)
            print()

    if all_no_missing:
        print("== No missing value ==")
```

print_lengths_by_id_and_trial 함수

```python
def print_lengths_by_id_and_trial(datasets_by_id_and_trial):
    # 각 id와 trial 별로 데이터프레임의 길이를 출력하는 함수
    ids = []
    trials = []
    lengths = []

    for name, df in datasets_by_id_and_trial.items():
        parts = name.split('_')
        id_part = parts[-3]
        trial_part = parts[-1]
        ids.append(id_part)
        trials.append(trial_part)
        lengths.append(len(df))

    # id, trial, 길이 정보를 문자열로 만들어 출력
    id_str_list = [f"id: {id}" for id in ids]
    trial_str_list = [f"trial: {trial}" for trial in trials]
    length_str_list = [str(length) for length in lengths]

    id_str = " | ".join([s.ljust(12) for s in id_str_list])
    trial_str = " | ".join([s.ljust(12) for s in trial_str_list])
    length_str = " | ".join([s.ljust(12) for s in length_str_list])

    print(id_str)
    print(trial_str)
    print(length_str)

    # 가장 긴 데이터프레임과 짧은 데이터프레임을 찾아 출력
    max_length = max(lengths)
    min_length = min(lengths)
    max_index = lengths.index(max_length)
    min_index = lengths.index(min_length)
    max_id = ids[max_index]
    max_trial = trials[max_index]
    min_id = ids[min_index]
    min_trial = trials[min_index]

    print(f"\nLongest: id: {max_id}, trial: {max_trial} with length {max_length}")
    print(f"Shortest: id: {min_id}, trial: {min_trial} with length {min_length}")
```

split_dataset_by_id 함수 : 각 id 별로 데이터를 그룹화하여 딕셔너리에 저장하는 함수
split_by_trial 함수 : 각 id별로 그룹화된 데이터에서 trial 별로 다시 그룹화하여 딕셔너리에 저장하는 함수
nan_check : id와 trial로 분리된 데이터프레임에 결측치가 있는지 확인하는 함수
print_lengths_by_id_and_trial : 각 id와 trial 별로 데이터프레임의 길이를 출력하는 함수

#전처리한 데이터 확인

위와 같이 함수를 구성하고 실제 수행한 결과는 다음과 같다.

전처리 수행 및 데이터 확인 코드

```
print("wlk dataset")
dataset_wlk_by_id = split_dataset_by_id(dataset_wlk, 'dataset_wlk')
dataset_wlk_by_id_and_trial = split_by_trial(dataset_wlk_by_id)
print_lengths_by_id_and_trial(dataset_wlk_by_id_and_trial)
nan_check(dataset_wlk_by_id_and_trial)

print("\n\njog dataset")
dataset_jog_by_id = split_dataset_by_id(dataset_jog, 'dataset_jog')
dataset_jog_by_id_and_trial = split_by_trial(dataset_jog_by_id)
print_lengths_by_id_and_trial(dataset_jog_by_id_and_trial)
nan_check(dataset_jog_by_id_and_trial)

print("\n\njog_ver dataset")
dataset_ver_wlk_by_id = split_dataset_by_id(dataset_ver_wlk, 'dataset_wlk')
dataset_ver_wlk_by_id_and_trial = split_by_trial(dataset_ver_wlk_by_id)
print_lengths_by_id_and_trial(dataset_ver_wlk_by_id_and_trial)
nan_check(dataset_ver_wlk_by_id_and_trial)

print("\n\njog_ver dataset")
dataset_ver_jog_by_id = split_dataset_by_id(dataset_ver_jog, 'dataset_jog')
dataset_ver_jog_by_id_and_trial = split_by_trial(dataset_ver_jog_by_id)
print_lengths_by_id_and_trial(dataset_ver_jog_by_id_and_trial)
nan_check(dataset_ver_jog_by_id_and_trial)
```

전처리 수행 및 데이터 확인 결과

```
wlk dataset
id: 0.0     | id: 0.0     | id: 0.0     | id: 1.0     | id: 1.0     | id: 1.0     | id: 2.0     | id: 2.0     | id: 2.0
trial: 7.0  | trial: 8.0  | trial: 15.0 | trial: 7.0  | trial: 8.0  | trial: 15.0 | trial: 7.0  | trial: 8.0  | trial: 15.
5439        | 4340        | 1333        | 7681        | 5614        | 6305        | 6410        | 5194        | 3594

Longest: id: 17.0, trial: 7.0 with length 8402
Shortest: id: 0.0, trial: 15.0 with length 1333
== No missing value ==


jog dataset
id: 0.0     | id: 0.0     | id: 1.0     | id: 1.0     | id: 2.0     | id: 2.0     | id: 3.0     | id: 3.0     | id: 4.0
trial: 9.0  | trial: 16.0 | trial: 9.0  | trial: 16.0 | trial: 9.0  | trial: 16.0 | trial: 9.0  | trial: 16.0 | trial: 9.0
4861        | 1567        | 4966        | 1452        | 4753        | 1216        | 5294        | 921         | 3558

Longest: id: 20.0, trial: 9.0 with length 5535
Shortest: id: 4.0, trial: 16.0 with length 765
== No missing value ==


jog_ver dataset
id: 15.0    | id: 15.0    | id: 15.0    | id: 22.0    | id: 22.0    | id: 22.0
trial: 7.0  | trial: 8.0  | trial: 15.0 | trial: 7.0  | trial: 8.0  | trial: 15.0
7366        | 5446        | 2744        | 6505        | 5441        | 3306

Longest: id: 15.0, trial: 7.0 with length 7366
Shortest: id: 15.0, trial: 15.0 with length 2744
== No missing value ==


jog_ver dataset
id: 15.0    | id: 15.0    | id: 22.0    | id: 22.0
trial: 9.0  | trial: 16.0 | trial: 9.0  | trial: 16.0
4814        | 1699        | 4815        | 1147

Longest: id: 22.0, trial: 9.0 with length 4815
Shortest: id: 22.0, trial: 16.0 with length 1147
== No missing value ==
```

위 이미지는 print_lengths_by_id_and_trial 함수와, nan_check 함수를 사용해 출력한 각 데이터프레임 정보 중 일부이다. 현재 데이터는 어떤 동작을 취하는지에 따라 데이터의 길이가 다르며, 무엇보다 개개인의 차이에 의해 똑같은 동작을 수행하여도 그 길이가 다르다.

또한, 각 데이터에 대해 결측치가 있는지 확인해보았으나 결측치는 존재하지 않는 것을 확인할 수 있었다.

#그래프 표출 함수

plot_line 함수

```python
def plot_line(datasets, data_name, id, trial, data_type, x_end=None, x_start=None):
    # 데이터 이름 생성
    name_to_plot = f'{data_name}_{id}_trial_{trial}'

    # 데이터 선택 및 특정 열 선택
    if name_to_plot in datasets:
        columns_to_plot = [col.strip() for col in data_type.split(',')]
        data_to_plot = datasets[name_to_plot][columns_to_plot]



        # 라인 차트 출력
        data_to_plot.plot.line(title=f'Dataset for ID {id} and Trial {trial} in {data_name}', figsize=(14, 4))

        # x_start와 x_end가 지정된 경우 해당 범위의 데이터 선택
        if x_start is not None and x_end is not None:
            plt.xlim(x_start, x_end)
        elif x_end is not None:
            plt.xlim(0, x_end)

        plt.xlabel('Index')
        plt.ylabel('Values')
        plt.show()
    else:
        print(f"ID {id} with Trial {trial} not found in the datasets of type {data_name}.")
```

plot_heatmap 함수

```python
def plot_heatmap(datasets, data_name, id, trial, data_type):
    # 데이터 이름 생성
    name_to_plot = f'{data_name}_{id}_trial_{trial}'

    # 데이터 선택 및 특정 열 선택
    if name_to_plot in datasets:
        columns_to_plot = [col.strip() for col in data_type.split(',')]
        data_to_plot = datasets[name_to_plot][columns_to_plot]

        # 데이터 타입 변환 및 결측값 처리
        data_to_plot = data_to_plot.apply(pd.to_numeric, errors='coerce')
        data_to_plot = data_to_plot.dropna()

        if data_to_plot.empty:
            print(f"No valid numeric data available for {name_to_plot}")
            return

        # 상관계수 계산
        corr = data_to_plot.corr()

        # 히트맵 출력
        fig, ax = plt.subplots(figsize=(10, 8))
        im = ax.imshow(corr.values, cmap='coolwarm')

        # 라벨 선택
        ax.set_xticks(np.arange(len(corr.columns)))
        ax.set_yticks(np.arange(len(corr.columns)))
        ax.set_xticklabels(corr.columns)
        ax.set_yticklabels(corr.columns)

        # 눈금 라벨 회전 및 정렬 설정
        plt.setp(ax.get_xticklabels(), rotation=45, ha="right",
                 rotation_mode="anchor")

        # 데이터 차원별로 텍스트 주석 생성
        for i in range(len(corr.columns)):
            for j in range(len(corr.columns)):
                text = ax.text(j, i, np.around(corr.iloc[i, j], decimals=2),
                               ha="center", va="center", color="black")

        # 컬러바 추가
        cbar = ax.figure.colorbar(im, ax=ax, cmap='coolwarm')
        cbar.ax.set_ylabel("Correlation", rotation=-90, va="bottom")

        plt.title(f'Heatmap for ID {id} and Trial {trial} in {data_name}')
        plt.show()
    else:
        print(f"ID {id} with Trial {trial} not found in the datasets of type {data_name}.")
```

plot_boxplot 함수

```python
def plot_boxplot(datasets, data_name, id, trial, data_type):
    # 데이터 이름 생성
    name_to_plot = f'{data_name}_{id}_trial_{trial}'

    # 데이터 선택 및 특정 열 선택
    if name_to_plot in datasets:
        columns_to_plot = [col.strip() for col in data_type.split(',')]
        data_to_plot = datasets[name_to_plot][columns_to_plot]

        # 데이터 타입 변환 및 결측값 처리
        data_to_plot = data_to_plot.apply(pd.to_numeric, errors="coerce")
        data_to_plot = data_to_plot.dropna()

        if data_to_plot.empty:
            print(f"No valid numeric data available for {name_to_plot}")
            return

        # 박스플롯 출력
        plt.figure(figsize=(8, 4))
        sns.boxplot(data=data_to_plot)
        plt.title(f'Boxplot for ID {id} and Trial {trial} in {data_name}')
        plt.xlabel('Variables')
        plt.ylabel('Values')
        plt.show()
    else:
        print(f"ID {id} with Trial {trial} not found in the datasets of type {data_name}.")
```

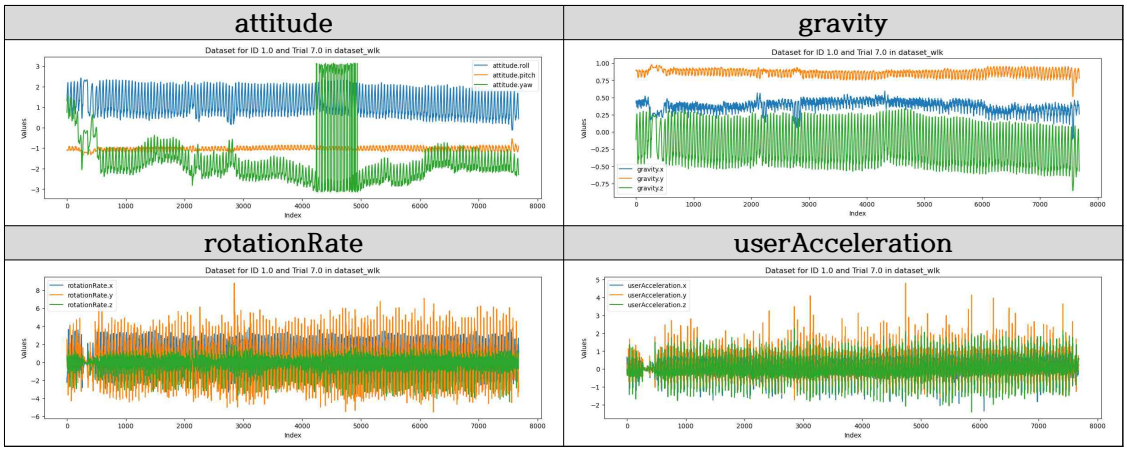
line chart, boxplot, heatmap 시각화 도구를 사용하여 데이터에 대한 분석을 진행하기 위해 각 동작을 함수로 구현하였다. 모든 함수는 출력할 데이터프레임, 데이터 이름, id, trial 그리고 출력할 센서 타입을 입력받으며, 예외적으로 plot_line 함수는 2개의 인자를 더 받는다. plot_line의 경우 x_end와 x_start 인자를 받으며 이를 통해서 시각화하고 싶은 데이터의 길이를 조정할 수 있다. 이때, 두 인자는 입력받지 않을 시 모든 데이터를 출력하며, 첫 번째 인자, 즉 x_end를 입력받으면 시작부터 x_end까지의 데이터를 출력한다.

위와 같이 함수에 대한 구성을 마쳤으므로 본격적으로 wlk 동작에 대한 분석할 것이다. 먼저, 각 데이터 타입 간 비교를 수행한다.

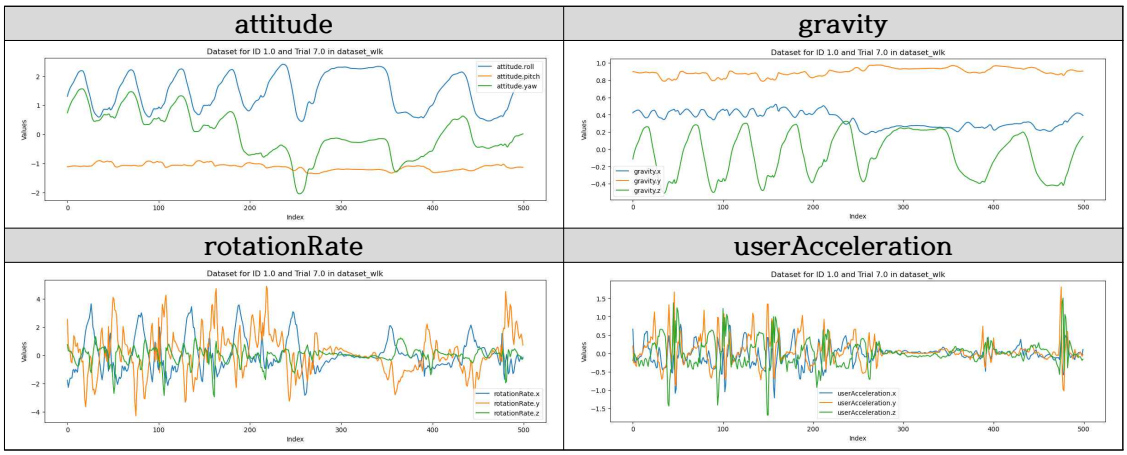
#Datatype 간 비교 (걷기)

데이터 타입 간의 비교를 위해 걷기 동작에서 id 1.0 trail 7.0의 데이터를 시각화하여 분석하였다.

다음은 라인차트의 결과이다.

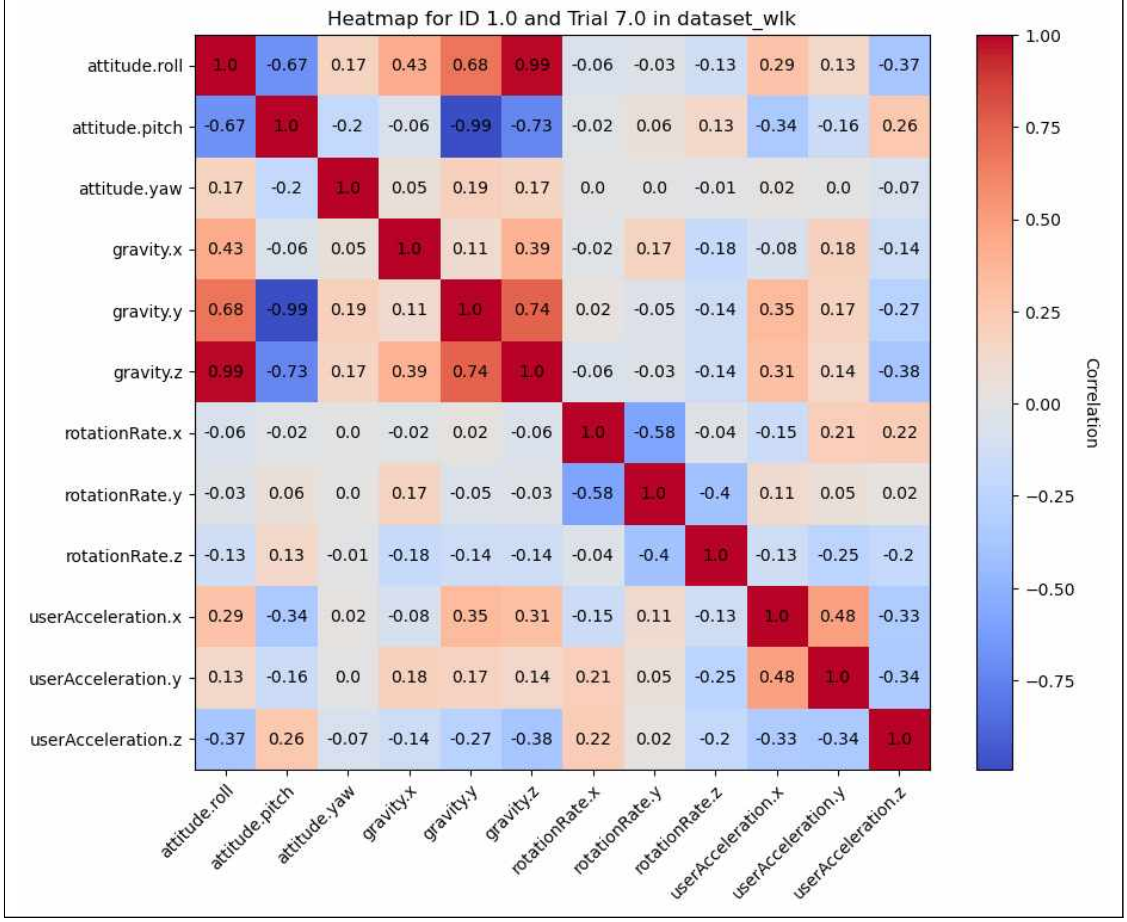


위는 모든 센서 타입별로 데이터를 출력해본 결과이다. 데이터의 길이가 약 8000으로 그래프의 밀도가 높아 구체적인 분석은 힘들다. 모든 센서 타입 데이터들은 진동하는 것 으로 보이나 구체적인 분석이 힘듦으로 0부터 500까지의 데이터만 출력해보았다. 아래는 그 결과이다.



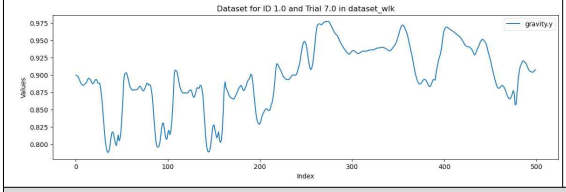
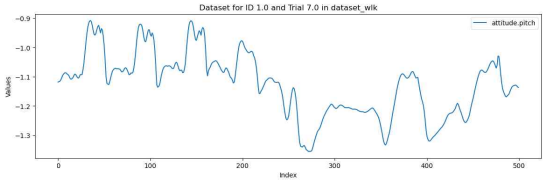
attitude와 gravity 그래프는 어느 정도 눈에 띄는 규칙성을 가진 진동 운동을 하는 것처럼 보인다. 현재 분석 중인 데이터가 걸음걸이에 대한 데이터이므로 attitude와 gravity 그래프에서 주기성이 걸음에서 비롯되었다고 유추할 수 있다. 특히 attitude 데이터의 경우 roll과 yaw 그래프가 매우 비슷한 형태를 가진다. 반면, gravity 데이터는 gravity.z에서만 주기성이 보이며 나머지 데이터는 데이터간 상관관계를 파악하기 힘들다. rotationRate와 userAcceleration 데이터도 규칙적으로 움직이는 것처럼 보이지만 노이즈가 많아 그 규칙성을 확인하기 힘들다. 각 그래프간 데이터간의 상관 관계는 다음 heatmap을 통해 구체적으로 확인해보고자 한다. 다음은 heatmap으로 모든 센서 타입간 상관관계를 확인해본 결과이다.

Heatmap

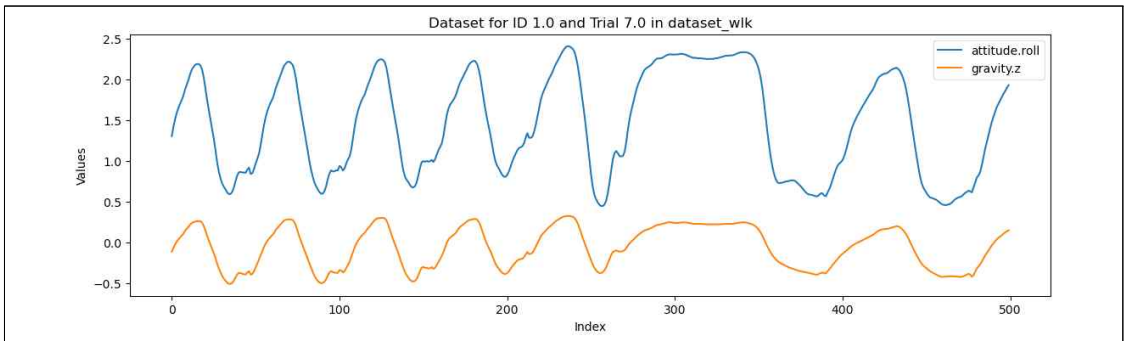


예상과 달리 attitude.roll과 attitude.yaw 데이터는 line chart에서 매우 비슷해 보였으나 0.17로 낮은 상관관계를 보였다. 반대로 attitude.pitch는 -0.67로 상당히 높은 상관관계를 보였다. 또한, attitude.roll과 gravity.z는 0.99로 거의 일치하는 것을 볼 수 있다. 오히려 attitude.pitch는 gravity.y와 -0.99로 마찬가지로 정반대의 상관관계를 보인다.

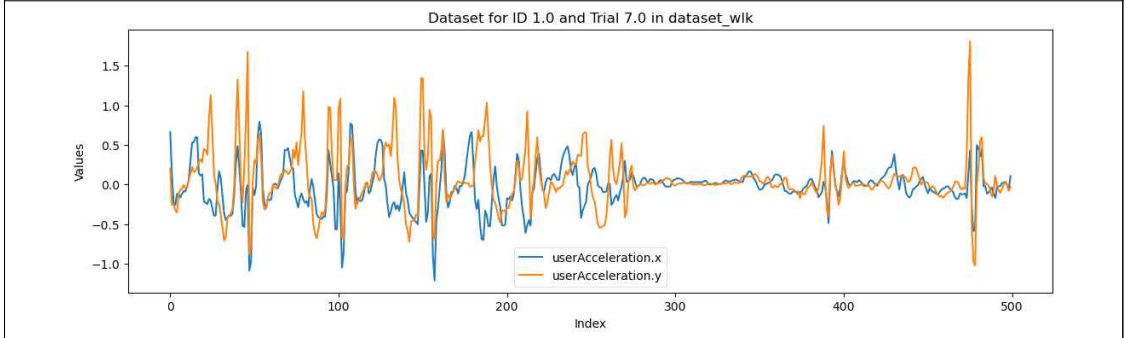
heatmap 결과의 다른 특징은 attitude와 gravity는 어느 정도의 높은 관계성을 보여주나 rotationRate와 userAcceleration과는 전체적으로 매우 낮은 상관관계를 가진다. 또한, rotationRate와 userAcceleration은 둘 사이에도 낮은 상관관계를 가진다. 다만, 각자의 x, y, z 사이에서는 약간의 상관관계가 있는 것을 확인할 수 있다. 이를 눈으로 확인해보기 위해 그래프로 어느 정도의 상관관계를 가진 데이터를 하나의 그래프에 출력하여 확인해보았다.

gravity.y	attitude.pitch
	

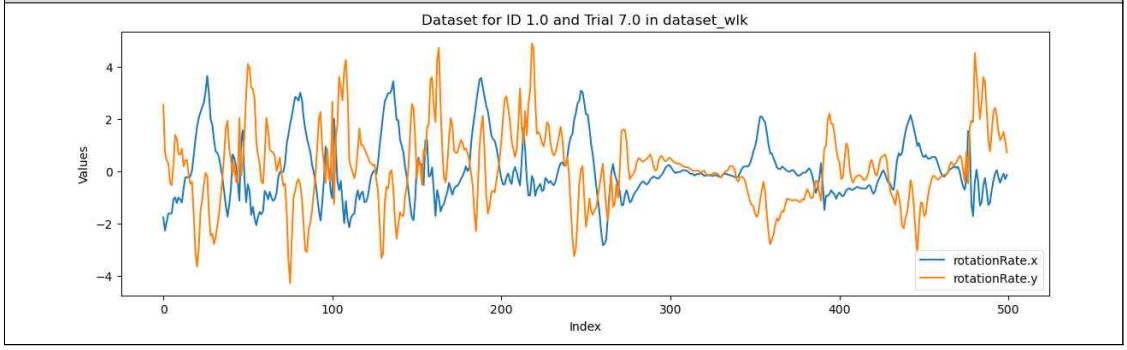
attitude.roll | gravity.z



userAcceleration.x | userAcceleration.y

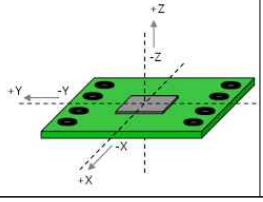
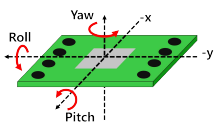


rotationRate.x | rotationRate.y



결과 gravity.y와 attitude.pitch가 확실해 반대의 상관관계를 가진 것을 확인할 수 있었으며 attitude.roll과 gravity.z 또한 높은 상관관계를 가진 것을 볼 수 있다. userAcceleration.x와 userAcceleration.y 그리고 rotaionRate.x와 rotationRate.y 또한 이전 모두 line chart로 표현했을 때 보다 유의미한 상관관계가 있는 것을 확인 가능했다.

gravity.y와 attitude.pitch 그리고 attitude.roll과 gravity.z가 높은 상관관계를 보이는 이유는 다음과 같이 예측할 수 있다.

gravity sensor	attitude sensor
	

좌측 이미지에서 pitch와 roll의 값을 증가시켜 핸드폰의 기울기를 변경할 때 gravity sensor의 y와 x의 값이 각각 pitch와 roll의 움직임에 비례 혹은 반비례할 것이다.

따라서, gravity sensor나 attitude sensor를 택할 경우 attitude.yaw와 gravity.z 값 중 걸음걸이 움직임의 특성을 더욱 잘 담은 센서 타입을 선택하는 것이 좋을 것이다.

일반적으로 핸드폰에 있는 만보기 기능은 가속도계 데이터를 사용하므로 가속도계 데이터를 사용하는 것으로 알려져 있다. 가속도 센서 데이터를 사용하여 걸음걸이를 유추하기 위해 가속도 센서 데이터를 에너지 데이터로 변경하여 데이터를 분석해 보았다.

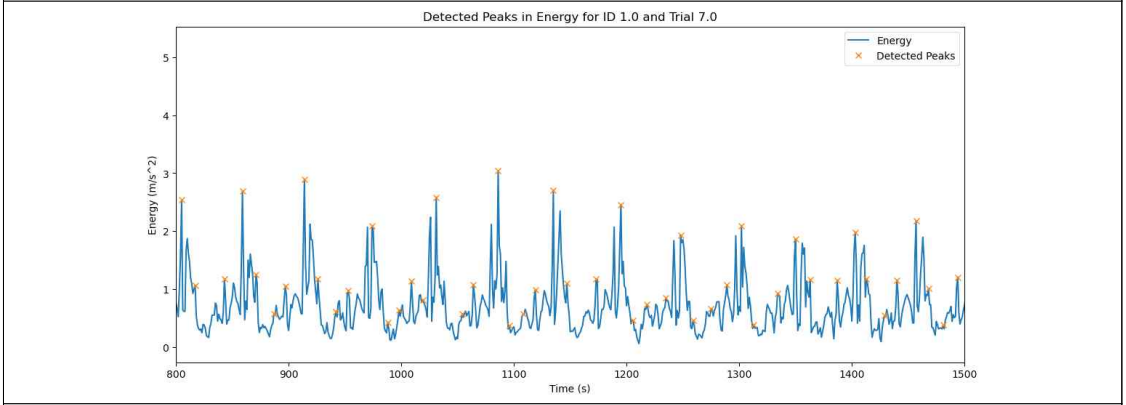
가속도 데이터 분석

```python
# 각 데이터셋에 대해 에너지 계산
def calculate_energy(data):
    data['energy'] = np.sqrt(data['userAcceleration.x']**2 + data['userAcceleration.y']**2 + data['userAcceleration.z']**2)
    return data

corrected_datasets = {}
for name, df in dataset_wlk_by_id_and_trial.items():
    corrected_df = calculate_energy(df)
    corrected_datasets[name] = corrected_df

# 보정된 가속도 데이터와 에너지 플롯 예시 (특정 ID와 Trial에 대해)
id_to_plot = '1.0'
trial_to_plot = '7.0'
name_to_plot = f'dataset_wlk_{id_to_plot}_trial_{trial_to_plot}'
```

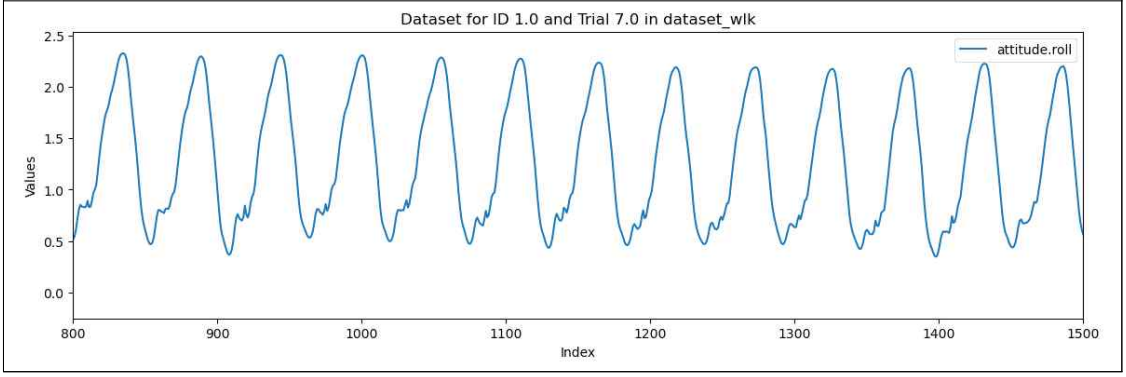
```python
if name_to_plot in corrected_datasets:
    df = corrected_datasets[name_to_plot]

    # 피크 찾기 (걸음걸이 주기 감지)
    peaks, _ = find_peaks(df['energy'], height=0, distance=10)
    plt.figure(figsize=(14, 6))
    plt.plot(df.index, df['energy'], label='Energy')
    plt.plot(df.index[peaks], df['energy'][peaks], "x", label='Detected Peaks')
    plt.xlim(1000, 1500)
    plt.xlabel('Time (s)')
    plt.ylabel('Energy (m/s^2)')
    plt.title(f'Detected Peaks in Energy for ID {id_to_plot} and Trial {trial_to_plot}')
    plt.legend()
    plt.show()

    # 피크 간 간격을 이용한 주기 계산
    step_intervals = np.diff(peaks) * 0.01  # 시간 간격 0.01초를 곱해 실제 시간 간격으로 변환
    average_step_interval = np.mean(step_intervals)
    print(f'Average Step Interval: {average_step_interval:.2f} seconds')
else:
    print(f"ID {id_to_plot} with Trial {trial_to_plot} not found in the datasets.")

plot_line(dataset_wlk_by_id_and_trial, 'dataset_wlk', '1.0', '7.0', "attitude.roll, gravity.z", 1500, 1000)
```

제공된 문서에 따르면 userAcceleration 데이터는 이미 중력 성분을 제거한 데이터이다. 따라서 $E = \sqrt{x^2 + y^2 + z^2}$ 를 통하여 에너지를 계산하고 그래프로 나타내 보았다.

에너지 그래프

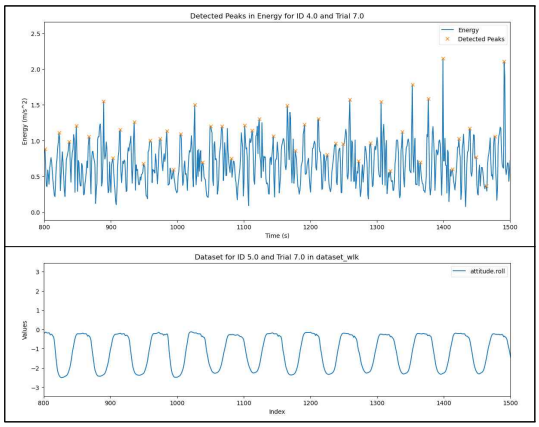
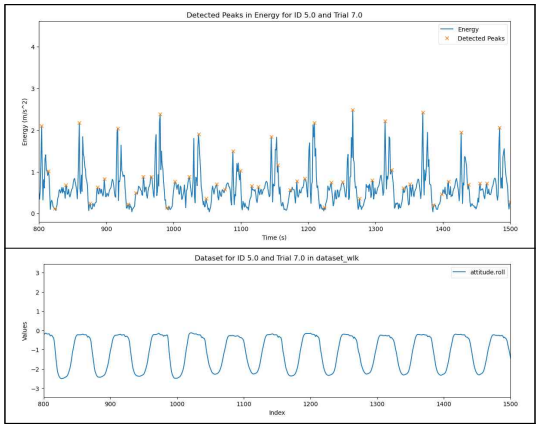
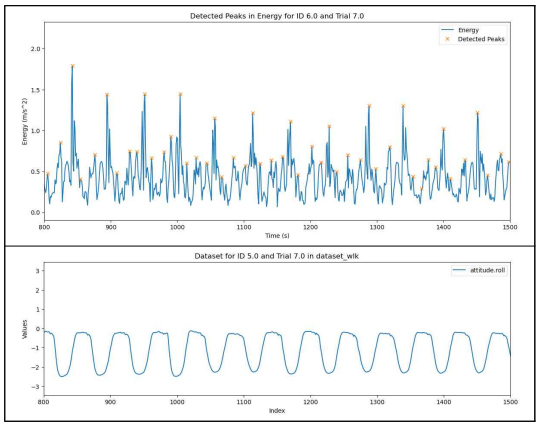
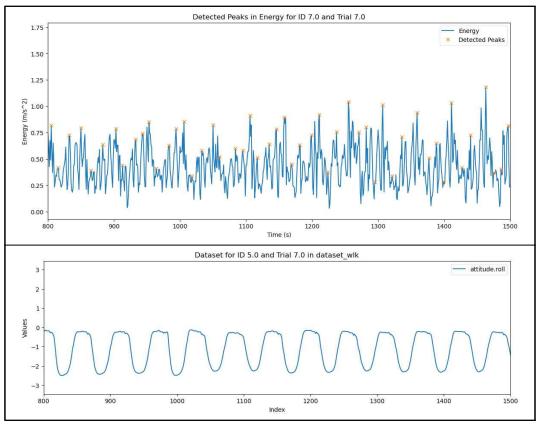


attitude.roll



변환된 에너지 그래프는 여전히 노이즈가 많다. 이 때문에 많은 피크가 감지된 것을 확인할 수 있다. 필터링을 통해 개선의 여지가 있지만 주목할 점은 에너지 그래프의 피크와 attitude.roll의 피크가 매우 유사한 모습을 보인다. 그렇다면 복잡한 과정 없이 attitude.roll 데이터를 사용해서 걸음걸이를 파악해도 충분히 신뢰할 만한 주기를 얻을 수 있을 것이다.

그렇다면 항상 이러한 결과가 나타는지 다른 사람에 대해서도 확인을 해 보았다.

<table>
<tr><th>id : 4.0</th><th>id : 5.0</th></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><th>id : 6.0</th><th>id : 7.0</th></tr>
<tr><td> 
</td><td> 
</td></tr>
</table>

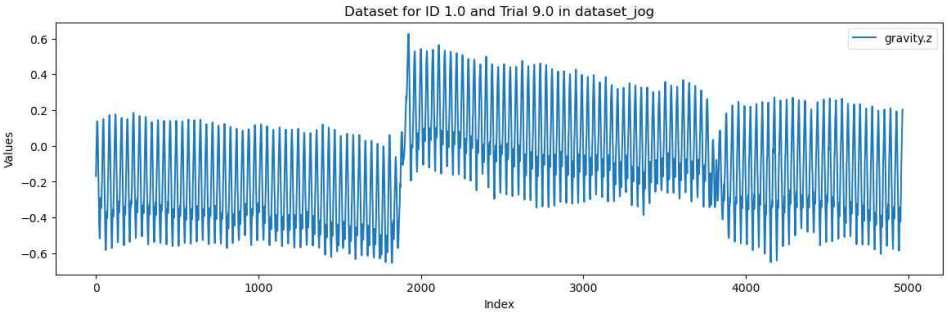
id 5와 6은 매우 뚜렷하게 에너지 그래프와 attitude.roll 그래프와 유사한 것을 볼 수 있다. 하지만, 4와 7은 노이즈가 커 뚜렷하게 보이진 않는다. 하지만, 전반적으로 attitude.roll과 유사한 것을 볼 수 있다.

따라서, attitude.roll과 매우 높은 유사도를 가지는 gravity.z, 혹은 attitude.pitch나 gravity.y를 사용하여 걸음걸이 데이터를 추출하고 신장별 특징을 확인할 것이다.

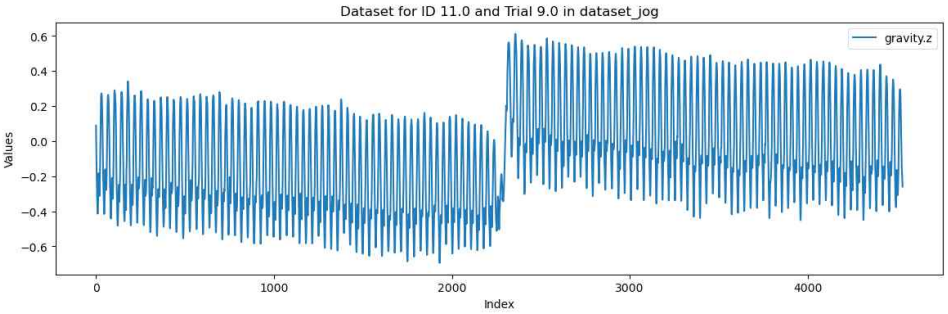
#Datatype 간 비교 (조깅)

걷기 과정에서 분석한 결과를 토대로 조깅에 대한 정보도 시각화하여 분석해보았다.

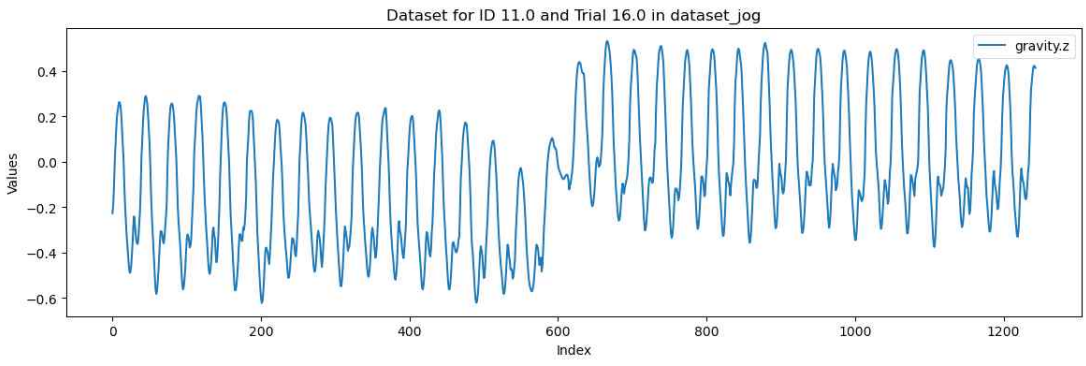
id : 1.0 | trial 9.0



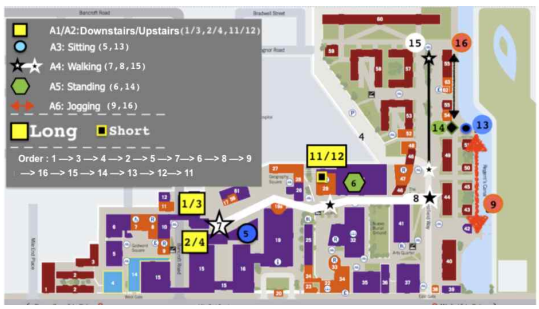
id : 11.0 | trial 9.0

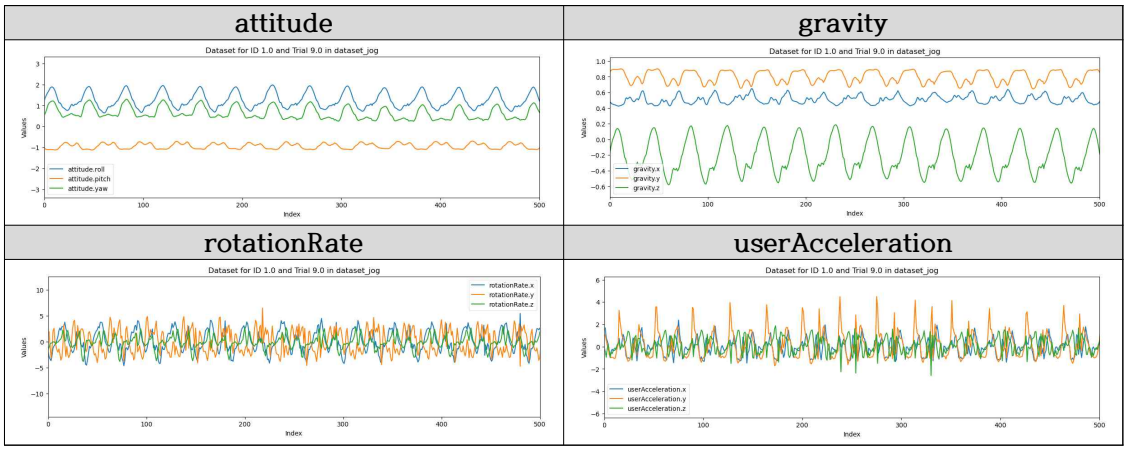


id : 11.0 | trial 16.0

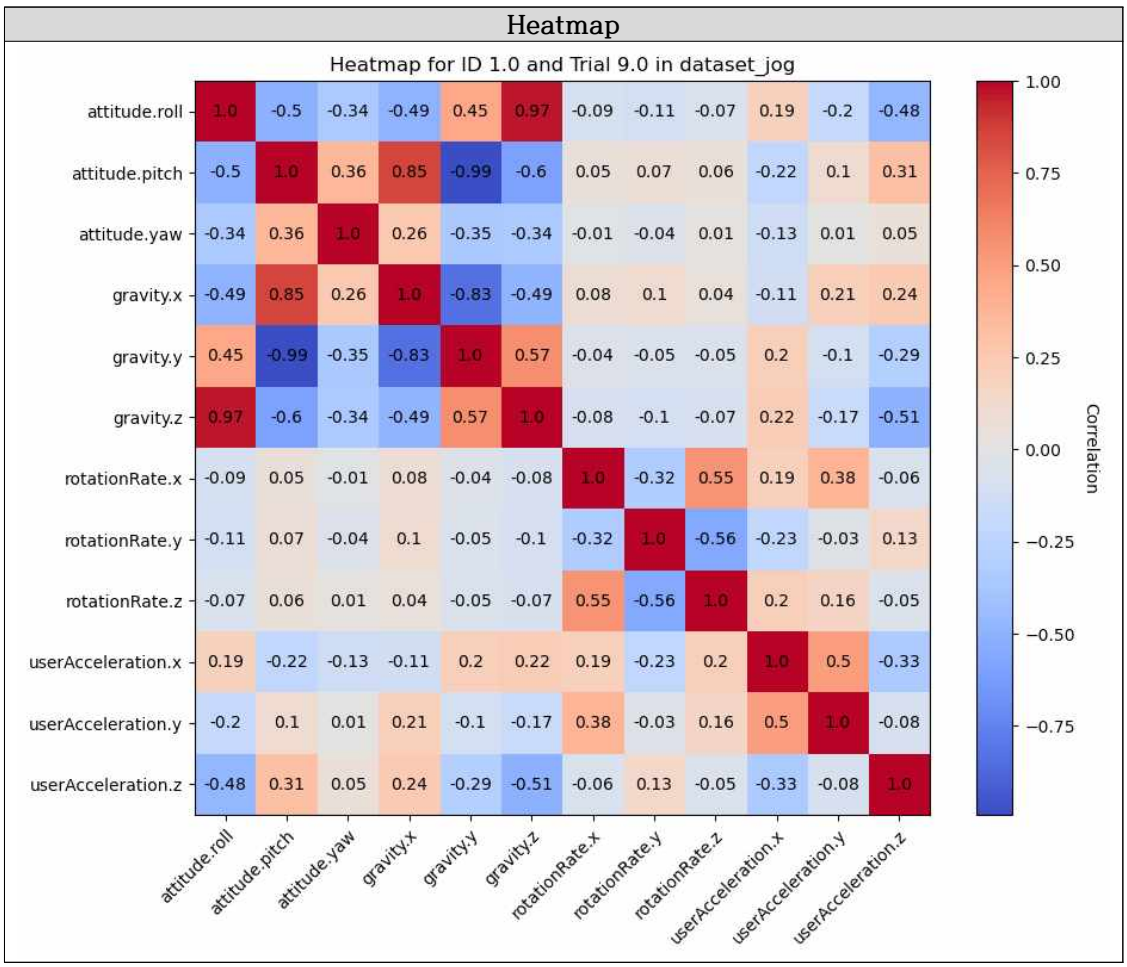


조깅 데이터의 경우 한번 출렁이는 경우가 존재한다. 이유를 찾기 위해 공식 문서를 찾아본 결과 조깅 동작은 왕복 운동을 수행한 것으로 확인할 수 있었다. 조깅에 대한 두 가지 trial 9.0과 16.0 모두 왕복 동작으로 데이터가 출렁이는 모습을 관측할 수 있었다. 이후 데이터를 확대하여 구체적으로 확인해보았다.



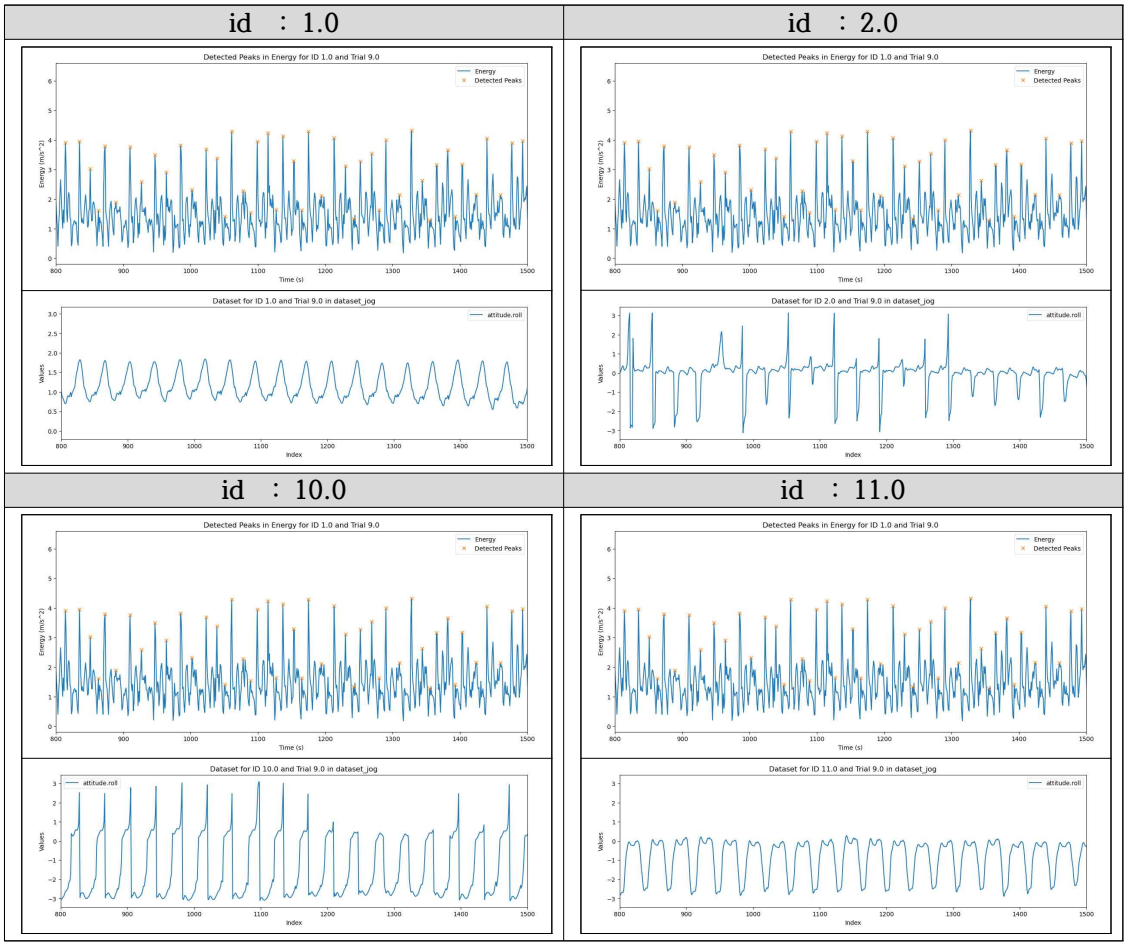


id 1의 trial 9 조깅 운동에 대해 라인차트를 확인해보니 이전 walk와 크게 차이를 보이지 않았다. 따라서 바로 heatmap을 확인해 보았다.

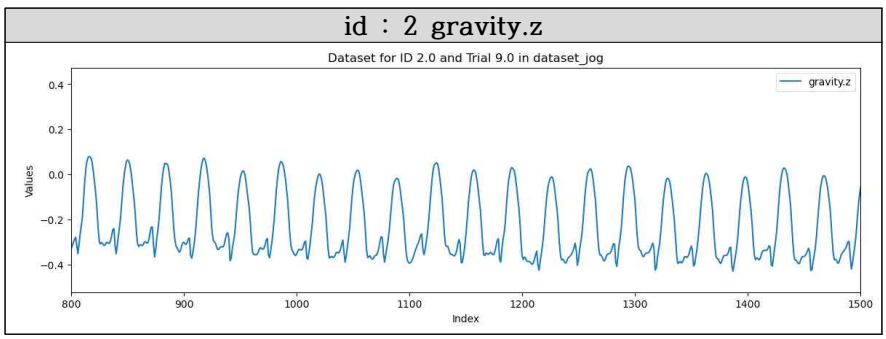


heatmap의 경우도 마찬가지로 이전과 매우 유사하다. 주목할 만한 점은 이전과 마찬가지로 gravity.y와 attitude.pitch 그리고 attitude.roll과 gravity.z가 매우 높은 상관관계를 보이는 점, 그리고 이전과 달리 attitude.pitch와 gravity.x가 -0.06에서 0.85로 상관관계 수치가 매우 높아졌다는 점이다. 하지만, 이전 걷기 동작에서 선택했던 attitude.roll, attitude.pitch,

gravity.y와 gravity.x는 여전히 유효하다. 마지막으로 가속도 센서 데이터를 에너지로 변환하여 확인해본 결과이다.



전반적으로 에너지 그래프와 attitude.roll은 일치하였으나 id 2에서 다소 문제가 될 수 있는 부분을 발견하였다. 데이터에 피크를 일정하게 추출할 수 있어야 걸음걸이를 확인할 수 있으나 이상한 노이즈가 포함되어 있는 것을 확인할 수 있었다. 따라서, 다른 후보군이던 gravity.z를 사용하여 확인해본 결과 다음과 같다.



다행이도 gravity.z의 경우 정상적인 피크를 가지고 있는 것을 확인할 수 있었다. 따라서, gravity.z 데이터를 사용하여 1차적으로 신장별 걸음걸이를 분석할 것이다.

#데이터 가공 및 주기 확인

데이터를 가공하는 과정은 다음과 같다.

1. gravity.z 데이터에서 피크값을 찾고 피크 사이의 간격 즉 주기를 추출한다.
2. gravity.z의 주기를 바탕으로 energy 그래프를 적분한다.
3. 주기별 적분된 데이터의 평균을 내고 평균에 가까운 80%의 데이터를 추출한다.
4. 추출한 80%의 데이터에 대해서 다시 평균을 추출

위와 같은 과정을 통해 가속도 데이터에 대한 필터링 없이 쉽게 에너지에 대한 정보를 추출하고, 추출한 데이터 중 이상치를 제거하여 개개인에 대한 평균적인 걸음걸이를 얻는다.

위 과정을 위해 선언한 함수들은 다음과 같다.

plot_line_peak

```
def plot_line_peak(datasets, data_name, id, trial, data_type, x_end=None, x_start=None):
    # 데이터 이름 생성
    name_to_plot = f'{data_name}_{id}_trial_{trial}'

    # 데이터 선택 및 특정 열 선택
    if name_to_plot in datasets:
        columns_to_plot = [col.strip() for col in data_type.split(',')]
        data_to_plot = datasets[name_to_plot][columns_to_plot]

        # 피크 찾기
        peaks, _ = find_peaks(data_to_plot.iloc[:, 0], distance=10, prominence=0.15, width=1)

        # 라인 차트 출력
        plt.figure(figsize=(14, 4))
        plt.plot(data_to_plot.index, data_to_plot.iloc[:, 0], label='Data')
        plt.plot(data_to_plot.index[peaks], data_to_plot.iloc[peaks, 0], "x", label='Peaks')

        # x_start와 x_end가 지정된 경우 해당 범위의 데이터 선택
        if x_start is not None and x_end is not None:
            plt.xlim(x_start, x_end)
        elif x_end is not None:
            plt.xlim(0, x_end)

        plt.title(f'Dataset for ID {id} and Trial {trial} in {data_name}')
        plt.xlabel('Index')
        plt.ylabel('Values')
        plt.legend()
        plt.show()
    else:
        print(f"ID {id} with Trial {trial} not found in the datasets of type {data_name}.")
```

integrate_energy_over_intervals

```
# 피크 간 적분된 에너지 계산 함수 (gravity.z 사용)
def integrate_energy_over_intervals(data, peaks):
    integrated_values = []
    for start, end in zip(peaks[:-1], peaks[1:]):
        integrated_value = np.trapz(data['energy'][start:end])
        integrated_values.append(integrated_value)
    return np.array(integrated_values)
```

process_and_select_data

```
# 데이터 처리 및 선택 함수
def process_and_select_data(datasets):

    selected_data = {}
    for name, data in datasets.items():
        selected_values = select_closest_to_mean(data)
        selected_data[name] = np.mean(selected_values)
    return selected_data
```

select_closest_to_mean

```
# 평균에 가장 가까운 값을 선택하는 함수
def select_closest_to_mean(data, percentage=0.8):
    mean_value = np.mean(data)
    distances = np.abs(data - mean_value)
    sorted_indices = np.argsort(distances)
    cutoff = int(len(data) * percentage)
    selected_indices = sorted_indices[:cutoff]
    return data[selected_indices]
```

calculate_peak_intervals

```
# 피크 찾기 및 주기 계산
def calculate_peak_intervals(data):
    peak_intervals = {}
    for name, df in data.items():
        peaks, _ = find_peaks(df['gravity.z'], distance=10, prominence=0.15, width=1)
        if len(peaks) > 1:
            intervals = np.diff(peaks)
            intervals = select_closest_to_mean(intervals, percentage=0.8)
            mean_interval = np.mean(intervals)
            peak_intervals[name] = mean_interval
        else:
            peak_intervals[name] = None
    return peak_intervals
```

process_datasets

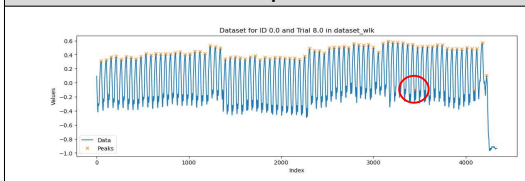
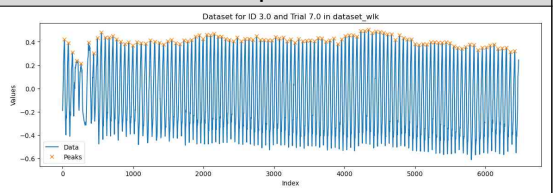
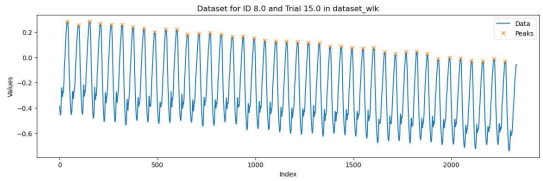
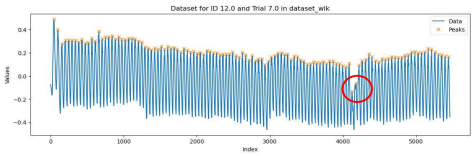
```
# 데이터셋 처리 함수
def process_datasets(datasets):
    processed_data = {}
    for name, df in datasets.items():
        df = calculate_energy(df)
        peaks, _ = find_peaks(df['gravity.z'], distance=10, prominence=0.15, width=1)
        integrated_values = integrate_energy_over_intervals(df, peaks)
        processed_data[name] = integrated_values
    return processed_data
```

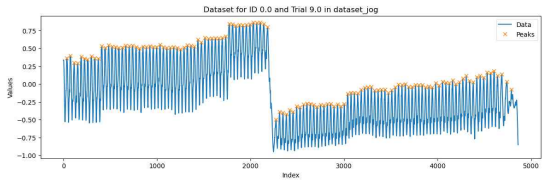
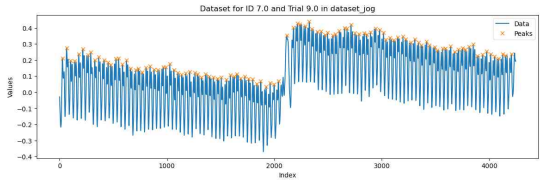
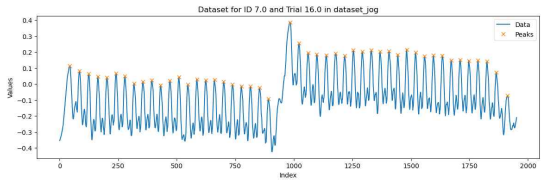
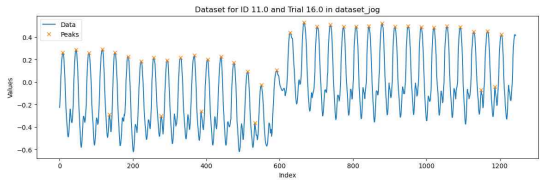
calculate_energy

```
# 에너지 계산 함수
def calculate_energy(data):
    data['energy'] = np.sqrt(data['userAcceleration.x']**2 + data['userAcceleration.y']**2 + data['userAcceleration.z']**2)
    return data
```

plot_line_peak : 이전 라인차트 함수에서 피크를 찍는 것을 추가한 함수
calculate_energy : 가속도 데이터로 에너지를 계산하는 함수
integrate_energy_over_intervals : 피크 위치를 토대로 에너지를 적분하는 함수
process_datasets : gravity.z의 피크를 찾고 적분을 하며, 해당 데이터를 반환하는 함수
select_closest_to_mean : 적분 값 중 평균에 가까운 80%의 데이터만 남기는 함수
process_and_select_data : 남은 80%의 데이터에서 다시 평균을 구하여 반환하는 함수
calculate_peak_intervals : 피크의 간격을 추출하여 평균에 80%에 해당하는 평균을 반환하는 함수

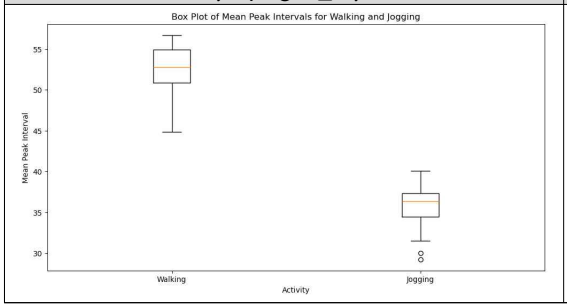
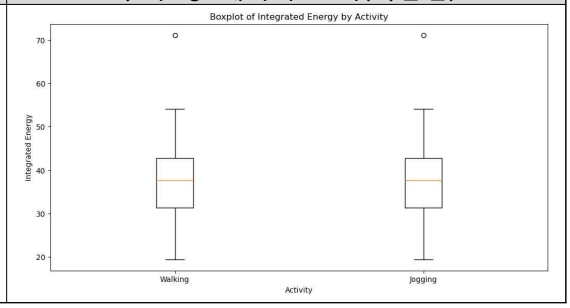
gravity.z 가 정말 주기를 잘 담고 있는지 확인하기 위해 모든 gravity.z 그래프를 출력하여 확인해보았다. 다음은 해당 결과 중 일부이다.

wlk dataset	
id : 0.0 | trial 8.0	id : 3.0 | trial 7.0
	
id : 0.0 | trial 8.0	id : 0.0 | trial 8.0
	

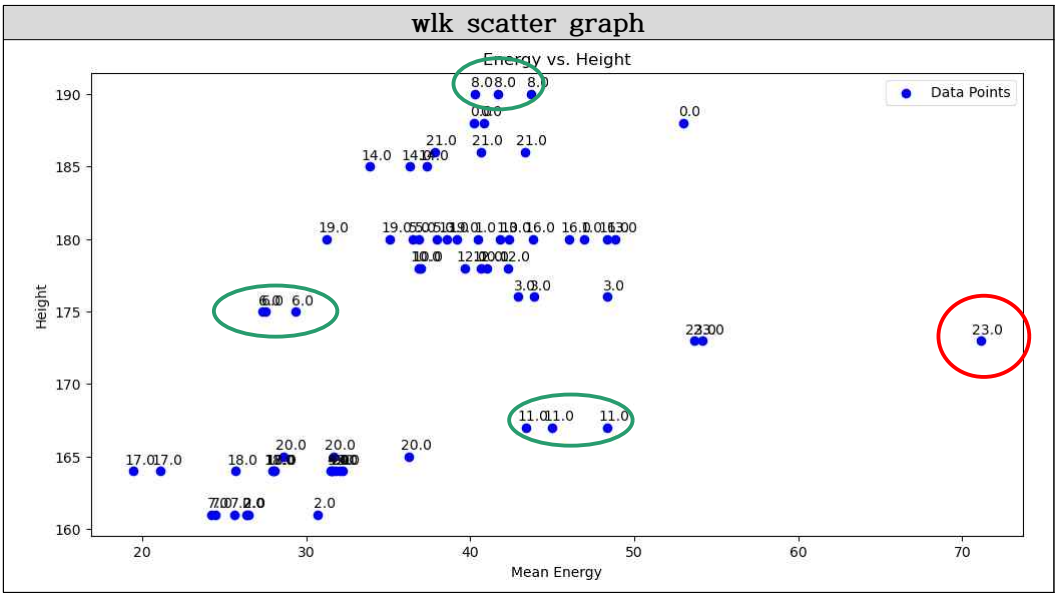
jog dataset	
id : 0.0 | trial 9.0	id : 7.0 | trial 9.0
	
id : 7.0 | trial 16.0	id : 11.0 | trial 16.0
	

피크를 찾을 때, “distance=10, prominence=0.15, width=1”와 같은 옵션을 주었기 때문에 피크를 잘 찾았으나 걷기 데이터에서 일반적이지 않은 위치에서 피크가 존재하는 것을 확인할 수 있다. 하지만, 이러한 데이터는 평균을 내고 80%의 데이터만 택하는 과정에서 제거할 수 있을 것이다. 또한, 조깅 데이터에서 반환점에서 데이터가 출렁일 때 발생하는 오차 또한 제거할 수 있다.

다음은 동작 별 주기 간격의 평균과 적분한 에너지 크기의 평균을 box plot으로 시각화한 결과이다.

주기 당 간격	주기 당 에너지 크기(적분한)
	

위 데이터는 가속도 에너지 데이터를 적분하는 것만으로는 걷기와 뛰기를 구분할 수 없다는 것을 의미한다. 반면, 주기의 간격 데이터는 걷기와 조깅이 겹치지 않고 잘 분리되어 있는 것을 확인할 수 있다.



위 그래프는 적분하여 얻은 결과의 평균을 scatter 그래프로 나타낸 결과이다. 위 결과를 보면 신장이 커질수록 에너지가 커지는 듯한 움직임을 보이긴 하지만, 일부 데이터는 선형적인 움직임을 따르지 않는다. 빨간 원으로 표시한 데이터는 모델을 학습하는 데 있어서 큰 방해가 될 것으로 예상된다. 하지만, 이러한 데이터는 모델 학습 단계에서 제거하여 높은 정확도를 가진 모델을 얻을 수 있을 것이다.

반면, 매우 이상적으로 측정되었음을 알 수 있는 데이터도 존재한다. 초록색 원으로 표시된 데이터들은 같은 id, 즉, 같은 사람으로부터 수집된 데이터가 서로 근접해 있는 것을 확인할 수 있다. 이는, 신장과 에너지가 비례하는 움직임을 떠나서 수집된 데이터가 일관성 있게 측정되었다는 것을 알 수 있다.

#모델 학습(wlk)

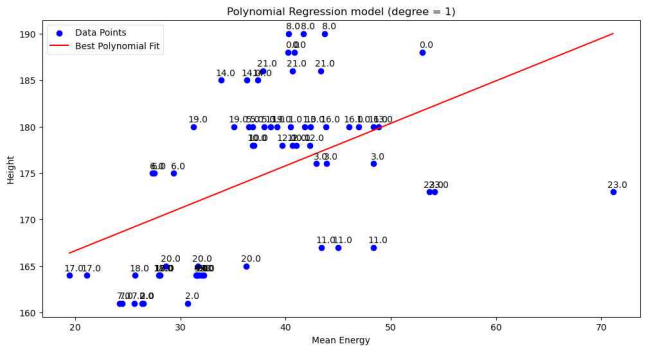
본격적으로 모델을 학습하기에 앞서 걷기 데이터에 대해서만 학습해보았다.

선형 모델 코드

```python
def polynomial_regression_analysis(X, y, ids, degree=1, test_size=0.2, num_iterations=1000):
    # 데이터 스케일링
    scaler = StandardScaler()
    X_scaled = scaler.fit_transform(X.reshape(-1, 1))

    r2_scores = []
    mse_scores = []
    models = []

    for _ in range(num_iterations):
        # 데이터 분할
        X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=test_size, random_state=np.random.randint(10000))

        # 다항 회귀 모델 학습
        polynomial_features = PolynomialFeatures(degree=degree)
        model = make_pipeline(polynomial_features, LinearRegression())
        model.fit(X_train, y_train)
        models.append(model)

        # 예측 및 평가
        y_pred = model.predict(X_test)
        mse = mean_squared_error(y_test, y_pred)
        r2 = r2_score(y_test, y_pred)
        mse_scores.append(mse)
        r2_scores.append(r2)

    # 결과 출력
    best_index = np.argmax(r2_scores)
    best_model = models[best_index]

    print(f'Best R^2 Score: {r2_scores[best_index]:.2f}')
    print(f'Corresponding Mean Squared Error: {mse_scores[best_index]:.2f}')

    # 기존 데이터 산점도 및 예측 곡선 그리기
    plt.figure(figsize=(12, 6))
    plt.scatter(X_scaled[:, 0], y, c='blue', label='Data Points')

    # 각 점에 id 표시
    for i, txt in enumerate(ids):
        plt.annotate(txt, (X_scaled[i, 0], y[i]), textcoords="offset points", xytext=(5, 5), ha='center')

    # 최적 모델의 예측 곡선 그리기
    X_fit = np.linspace(X_scaled.min(), X_scaled.max(), 100)[:, np.newaxis]
    y_fit = best_model.predict(X_fit)
    plt.plot(X_fit, y_fit, color='red', label='Best Polynomial Fit')

    plt.xlabel('Mean Energy')
    plt.ylabel('Height')
    plt.title(f'Polynomial Regression model (degree = {degree})')
    plt.legend()
    plt.show()
```

이후 과정에서 수월한 학습을 위해 함수를 만들어 사용하였다. test 사이즈는 20%, 그리고 반복 횟수는 1000으로 설정하였으나 입력받아 수정가능하도록 하였다. 선형 모델의 차수를 입력받고 모델을 예측한 후 결정계수와 그래프를 출력하도록 하였다. 다음은 1차 선형 모델을 사용한 결과이다.

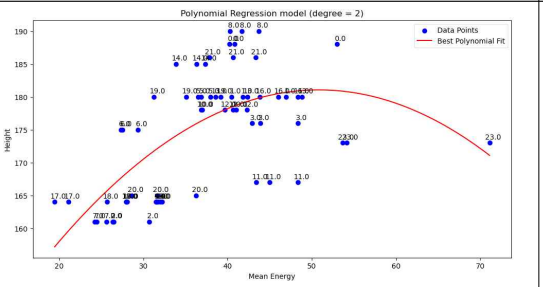
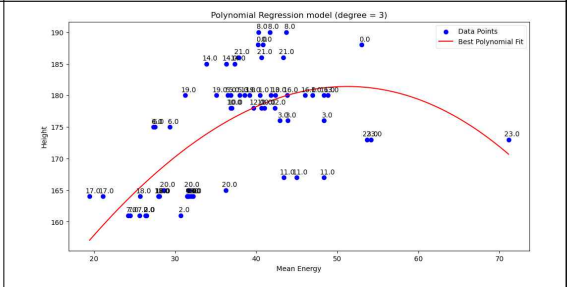
1차 선형 모델




```
Best R^2 Score: 0.59
Corresponding Mean Squared Error: 25.64
```

1000의 반복을 통해 얻은 가장 높은 결정 계수를 가지는 값은 0.59로 매번 랜덤한 값을 넣기 때문에 매번 다른 R^2 점수를 확인할 수 있으나 대략 0.6정도의 결정계수를 가졌다. 위 그래프에 따르면, 현재 그래프에서 결과에 큰 영향을 줄 수 있는 id가 23인 데이터에 의해 선형 모델의 기울기가 실제보다 낮게 나왔을 것으로 예상해 볼 수 있다. 이론적으로 신장에 비례하여 보폭이 넓어지고, 보폭이 넓어질수록 한 걸음당 에너지가 커질 것이므로 선형

모델을 사용하는게 적합하나, 2차, 3차 모델에 대해서도 확인해보았다.

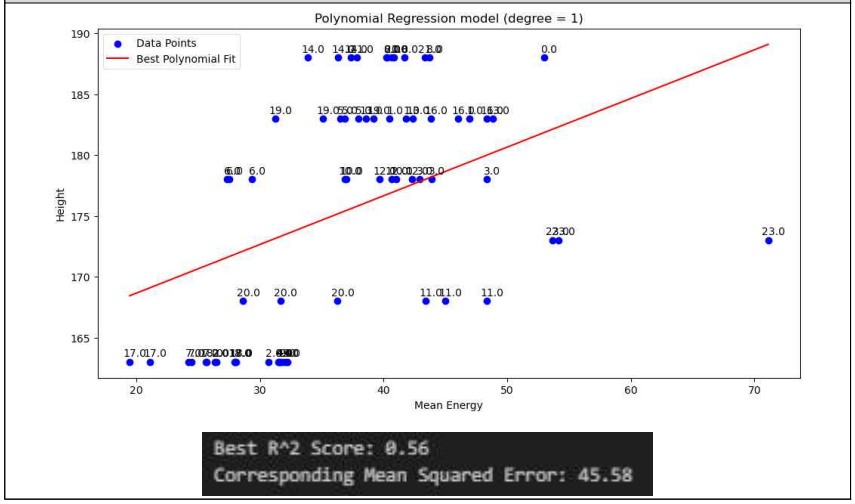
2차 선형 모델	3차 선형 모델
	

다항식을 사용한 결과 2차에서 0.75, 3차에서 0.79의 결정계수를 보였다. 이전보다 높은 결정계수를 보이지만, 이러한 모델은 경험적인 데이터와 일치하지 않으며 과적합되었을 가능성이 높음으로 1차 선형 모델을 사용하는 것이 적합할 것이다.

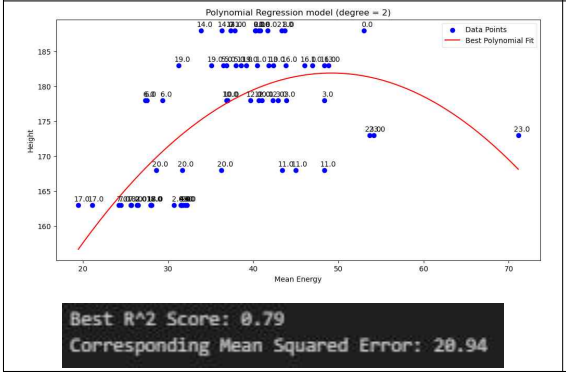
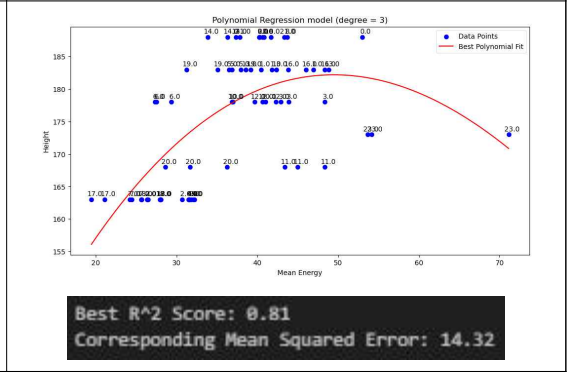
#양자화 적용

이번 과정에서는 신장에 대해 양자화를 수동으로 실시하고 모델을 학습해보았다. 우측과 같이 6단계로 나눠 선형 모델과 다항식 모델을 학습한 결과는 다음과 같다.

```python
def quantize_height(height):
    if height < 165:
        return 163
    elif 165 <= height < 170:
        return 168
    elif 170 <= height < 175:
        return 173
    elif 175 <= height < 180:
        return 178
    elif 180 <= height < 185:
        return 183
    else:
        return 188
```

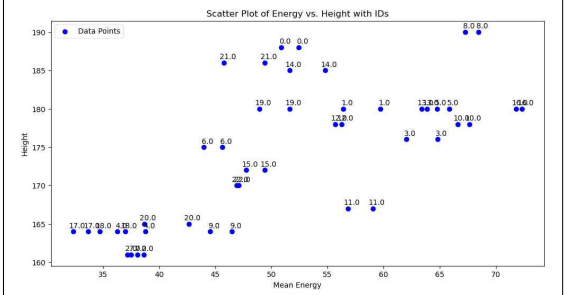
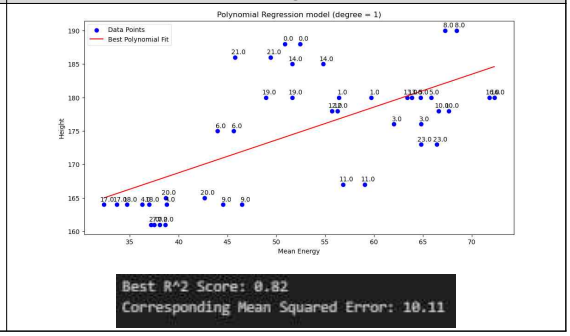
1차 선형 모델


1차 선형 모델은 결정계수가 0.56로 낮아진 것을 확인할 수 있었다. 현재 사용하는 데이터셋의 샘플 수가 적음으로 양자화를 통해 정확도를 향상 시켜보고자 하였으나, 예상과는 달리 오히려 결정계수가 낮아졌다. 2차와 3차 모델에 대해서도 학습을 시켜 추가적인 확인을 해보았다.

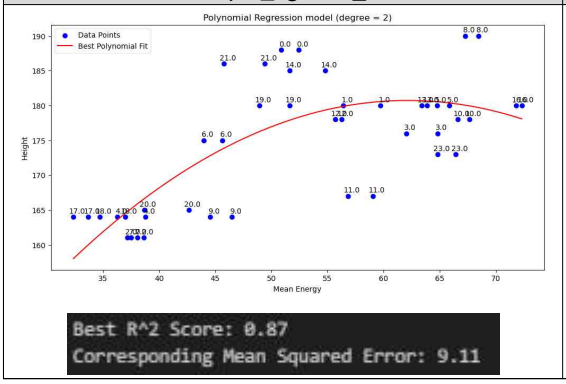
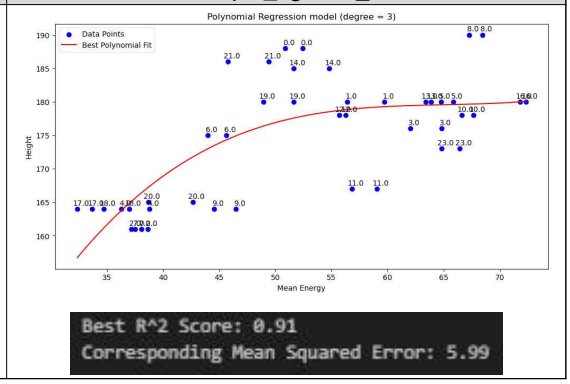
2차 선형 모델	3차 선형 모델
	

2차 선형 모델에서는 0.79, 3차에서는 0.81로 거의 양자화 이전과 큰 차이가 없는 것을 확인할 수 있었다.
결과적으로 걷기 데이터에서는 양자화를 하지 않고, 1차 선형 모델을 사용하는 것이 가장 적합할 것이다.

#모델 학습(jog)

조깅 scatter 그래프	1차 선형 모델
	

조깅 데이터의 경우 육안으로도 걷기 데이터보다 이상치가 적은 것을 확인할 수 있었다.
실제로 1차 선형 모델은 0.90으로 걷기 데이터에 비해 높은 결정계수를 보였다. 이어서, 2차, 3차 선형 모델도 확인해보았다.

2차 선형 모델	3차 선형 모델
	

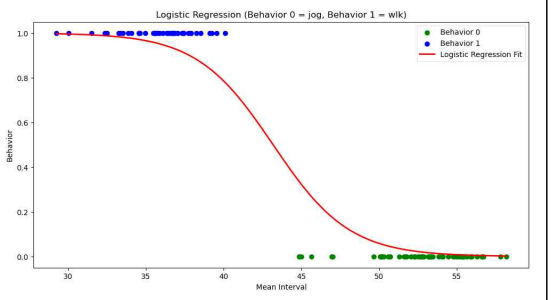
2, 3차 선형 모델에서는 각각 0.87, 0.91로 1차 선형 모델보다 높은 결정계수를 보이긴 하나, 마찬가지로 과적합 되었을 가능성이 높아보인다. 따라서 조깅 데이터셋에서도 1차 선형 모델을 사용할 것이다.

#최종 모델

코드

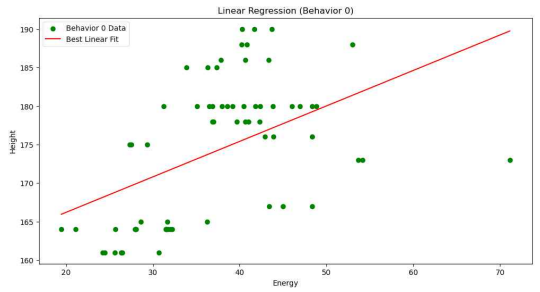
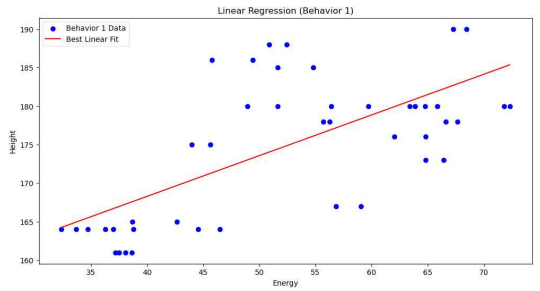
```python
peak_intervals_wlk = calculate_peak_intervals(dataset_wlk_by_id_and_trial)
peak_intervals_jog = calculate_peak_intervals(dataset_jog_by_id_and_trial)

# 주기 및 행동 데이터 준비
X_intervals = np.array(list(peak_intervals_wlk.values()) + list(peak_intervals_jog.values())).reshape(-1, 1)
y_behaviors = np.array([1]*len(peak_intervals_wlk) + [1]*len(peak_intervals_jog))

# 데이터 스케일링
scaler = StandardScaler()
X_intervals_scaled = scaler.fit_transform(X_intervals.reshape(-1, 1))

# Logistic Regression 모델 학습
logistic_model = LogisticRegression()
logistic_model.fit(X_intervals_scaled, y_behaviors)

# 예측 및 결과
behavior_labels = logistic_model.predict(X_intervals_scaled)

# 클러스터링 결과 시각화 (행동 구분)
plt.figure(figsize=(12, 6))
colors = ['blue', 'green']
for behavior in np.unique(behavior_labels):
    behavior_indices = np.where(behavior_labels == behavior)
    X_behavior = X_intervals[behavior_indices]
    y_behavior = y_behaviors[behavior_indices]
    plt.scatter(X_behavior[:, 0], y_behavior, c=colors[behavior-1], label=f'Behavior {behavior}')
plt.xlabel('Mean Interval')
plt.ylabel('Behavior')
plt.title('Logistic Regression Results (Behavior Classification)')
plt.legend()
plt.show()
```

그래프



본격적인 모델 학습을 위해 주기성을 가진 데이터를 추출하여 이를 분석한다면 걷기 동작인지, 뛰기 동작인지 구분할 필요가 있다. 따라서, Logistic Regresstion에서 주기 당 걸음 간격을 사용하여 걷기 동작인지, 뛰기 동작인지를 먼저 구분하고, 이를 토대로 두 데이터를 구분하여 학습할 것이다.
이를 위해 wlk 데이터를 행동 0, jog 데이터를 행동 1로 구분하여 Logistic Regression을 사용하여 분류해본 결과이다. 앞서, 두 데이터의 주기 간격은 서로 겹치지 않고 분리되어있음으로 100%의 정확도로 구분된 것을 확인할 수 있다.

이후, 각 라벨로 분리하여 걷기 데이터와 조깅 데이터를 학습시켜 보았다.

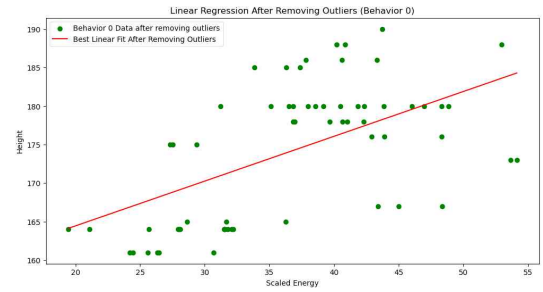
코드	그래프
	 Best R^2 Score for Behavior 0: 0.62 Linear Model for Behavior 0: y = 0.4602 * x + 157.0073  Best R^2 Score for Behavior 1: 0.85 Linear Model for Behavior 1: y = 0.5280 * x + 147.1620

위 데이터는 행동별로 구분된 데이터를 행동에 따라 선형 학습한 결과이다. 걷기 데이터의 경우 0.74, 조깅 데이터의 경우 0.89로 이전과 이전에 개별로 확인해본 결과가 같은 것을 확인할 수 있었다. 조깅 데이터의 경우 나쁘지 않은 결과를 보였으나, 걷기 데이터의 경우 이전에 확인했던 이상치 데이터를 제거하면 더욱 좋은 모델을 얻을 수 있을 것이라 예상하였다. 따라서, 이상치를 제거하고 학습하는 2차 학습을 진행하고자 하였다. 현재는 데이터셋이 적으므로 수동으로 데이터를 제거하는 것이 가능하지만, 데이터셋이 많아지면, 수동으로 데이터를 하나씩 제거할 수 없음으로, 1차적으로 학습한 모델을 토대로 이상치를 제거하고, 2차 학습을 진행하는 방식으로 구현하였다.

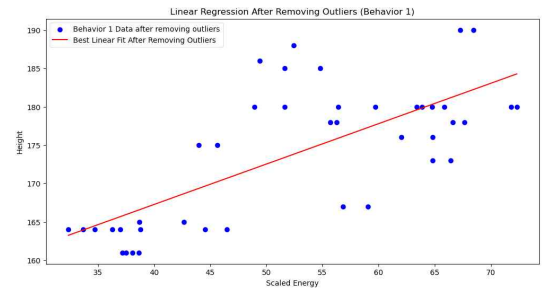
코드	그래프

```python
def remove_outliers(X, y, model, threshold_multiplier=3.5):
    y_pred = model.predict(X)
    residuals = np.abs(y - y_pred)
    threshold = threshold_multiplier * np.std(residuals)
    cleaned_indices = residuals <= threshold
    return X[cleaned_indices], y[cleaned_indices]
```

```python
# 이상치 제거 및 2차 모델 학습
for idx, behavior in enumerate(np.unique(behavior_labels)):
    behavior_indices = np.where(behavior_labels == behavior)
    X_behavior = X_energy[behavior_indices]
    y_behavior = y_energy[behavior_indices]

    # 이전 셀에서 학습한 최적 모델을 사용하여 이상치 제거
    best_model = best_models[idx]
    X_cleaned, y_cleaned = remove_outliers(X_behavior, y_behavior, best_model)

    best_r2_score_cleaned = -np.inf
    best_model_cleaned = None

    for _ in range(num_iterations):
        # 데이터 분할
        X_train, X_test, y_train, y_test = train_test_split(X_cleaned, y_cleaned, test_size=0.2, random_state=np.random.randint(10000))

        # 선형 회귀 모델 학습
        model_cleaned = LinearRegression()
        model_cleaned.fit(X_train, y_train)

        # 예측 및 평가
        y_pred_cleaned = model_cleaned.predict(X_test)
        r2_cleaned = r2_score(y_test, y_pred_cleaned)

        if r2_cleaned > best_r2_score_cleaned:
            best_r2_score_cleaned = r2_cleaned
            best_model_cleaned = model_cleaned

    print(f"Best R^2 Score for Behavior {behavior} after removing outliers: {best_r2_score_cleaned:.2f}")

    # 선형 모델의 기울기 출력
    coef = best_model_cleaned.coef_[0]
    intercept = best_model_cleaned.intercept_
    print(f"Linear Model for Behavior {behavior} after removing outliers: y = {coef:.4f} * x + {intercept:.4f}")

    # 최적 모델의 예측 곡선 그리기
    plt.figure(figsize=(12, 6))
    plt.scatter(X_cleaned, y_cleaned, color=colors[behavior-1], label=f'Behavior {behavior} Data after removing outliers')

    # 최적 모델의 예측 곡선 그리기
    X_fit = np.linspace(X_cleaned.min(), X_cleaned.max(), 100).reshape(-1, 1)
    y_fit = best_model_cleaned.predict(X_fit)
    plt.plot(X_fit, y_fit, color='red', label='Best Linear Fit After Removing Outliers')

    plt.xlabel('Scaled Energy')
    plt.ylabel('Height')
    plt.title(f'Linear Regression After Removing Outliers (Behavior {behavior})')
    plt.legend()
    plt.show()
```



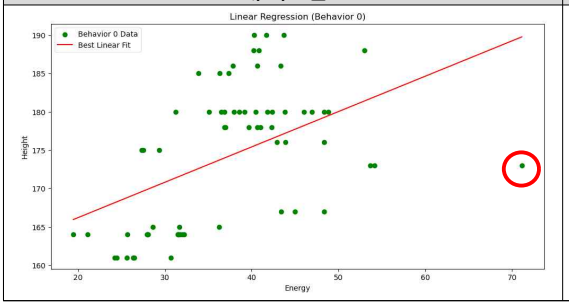

```
Best R^2 Score for Behavior 0 after removing outliers: 0.78
Linear Model for Behavior 0 after removing outliers: y = 0.5813 * x + 152.8256
```

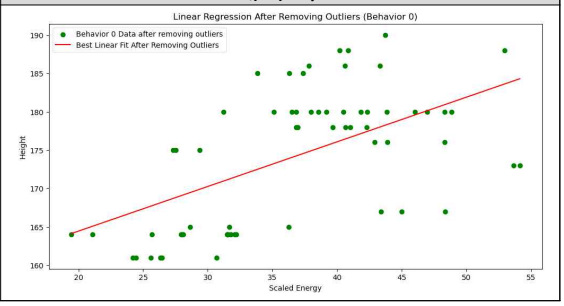



```
Best R^2 Score for Behavior 1 after removing outliers: 0.88
Linear Model for Behavior 1 after removing outliers: y = 0.5260 * x + 146.2282
```

2차 학습에서는 1차로 학습했던 모델을 통해 얻은 예측값과 실제값의 차이를 얻어 차이들의 표준편차를 계산하고 표준편차의 3.5배(여러번의 시도를 통해 선정하였음)를 초과하는 데이터 포인트를 이상치로 간주하고 제거한 후 재학습을 한 것이다. 결과 조깅과 걷기는 각각 결정계수 0.82와 0.91으로 걷기 데이터에서 결정계수가 향상되었으며 두 데이터셋에서 데이터셋에 대한 설명이 가능한 적절한 모델을 얻었다고 판단하였다.
실제로 이상치를 제거한 것과 그렇지 않은 두 데이터를 직접 비교하면 다음과 같다.

제거 전	제거 후





제거 전 빨간 원으로 표시한 모델 추론에 큰 방해가 될 것으로 예상하던 데이터는 필터링을 통해 일부 제거되었으며, 그 외의 데이터는 그대로 유지된 것을 확인할 수 있다. 이를 통해 2차 학습에서는 더욱 정확한 결과를 얻을 수 있을 것이다.

1차 시도와, 2차 시도에서 모델의 변화는 다음과 같다.

걷기	조깅
1차 : $y = 0.4602x + 157.0073$	1차 : $= 0.5282x + 14.1620$
2차: $y = 0.5813x + 152.5256$	2차 : $y = 0.5260x + 146.2282$

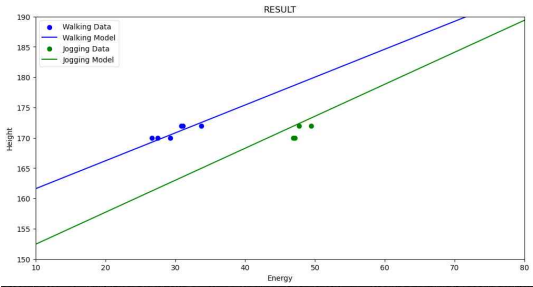
조깅 데이터의 경우 1차와 2차에서의 차이가 거의 근소하나, 걷기 데이터의 경우 유의미하게 기울기가 바뀐 것을 확인할 수 있다. 필터링을 통해 정확한 모델을 얻어 사용할 것이다.

#모델 검증

코드

```
new_processed_datasets_wlk = process_datasets(dataset_ver_wlk_by_id_and_trial)
new_processed_datasets_jog = process_datasets(dataset_ver_jog_by_id_and_trial)
new_selected_data_wlk = process_and_select_data(new_processed_datasets_wlk)
new_selected_data_jog = process_and_select_data(new_processed_datasets_jog)


# 새로운 데이터 준비
X_new = []
y_new = []
ids_new = []

for name, energy in new_selected_data_wlk.items():
    id_value = name.split('_')[2]
    height = dataset_ver_wlk_by_id_and_trial[name]['height'].iloc[0]
    X_new.append(energy)
    y_new.append(height)
    ids_new.append(id_value)

for name, energy in new_selected_data_jog.items():
    id_value = name.split('_')[2]
    height = dataset_ver_jog_by_id_and_trial[name]['height'].iloc[0]
    X_new.append(energy)
    y_new.append(height)
    ids_new.append(id_value)

X_new = np.array(X_new).reshape(-1, 1)
y_new = np.array(y_new)
ids_new = np.array(ids_new)

# 새로운 데이터 결과스피킹 예측 (학습 및 평가 없이)
new_peak_intervals = calculate_peak_intervals({**dataset_ver_wlk_by_id_and_trial, **dataset_ver_jog_by_id_and_trial})
X_new_intervals = np.array(list(new_peak_intervals.values())).reshape(-1, 1)
new_behaviors = logistic_model.predict(scaler.transform(X_new_intervals))

# 새로운 데이터 행동 그룹별 선형 회귀 모델 적용 및 예측 결과 출력
for i, (energy, true_height, behavior, id_value) in enumerate(zip(X_new, y_new, new_behaviors, ids_new)):
    best_model = best_models[behavior - 1]
    pred_height = best_model.predict(energy.reshape(1, -1))[0]

    behavior_label = "Jogging" if behavior == 1 else "Walking"
    print(f"ID: {id_value} | Behavior: {behavior_label} | True Height: {true_height} | Predicted Height: {pred_height:.2f}")
```

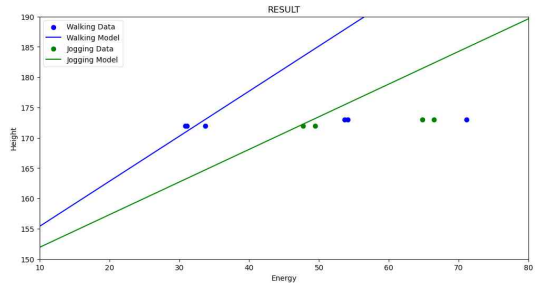
검증 결과




```
ID: 15.0 | Behavior: Walking | True Height: 172.0 | Predicted Height: 171.31
ID: 15.0 | Behavior: Walking | True Height: 172.0 | Predicted Height: 171.20
ID: 15.0 | Behavior: Walking | True Height: 172.0 | Predicted Height: 172.51
ID: 22.0 | Behavior: Walking | True Height: 170.0 | Predicted Height: 169.27
ID: 22.0 | Behavior: Walking | True Height: 170.0 | Predicted Height: 169.66
ID: 22.0 | Behavior: Walking | True Height: 170.0 | Predicted Height: 170.49
ID: 15.0 | Behavior: Jogging | True Height: 172.0 | Predicted Height: 173.25
ID: 15.0 | Behavior: Jogging | True Height: 172.0 | Predicted Height: 172.37
ID: 22.0 | Behavior: Jogging | True Height: 170.0 | Predicted Height: 172.02
ID: 22.0 | Behavior: Jogging | True Height: 170.0 | Predicted Height: 171.93
```

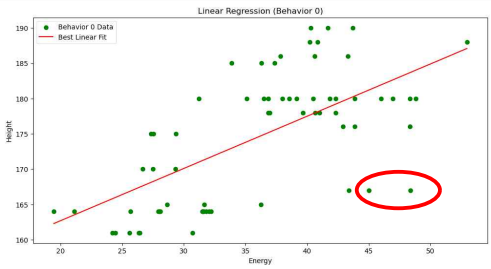
초기 분리해뒀던 데이터를 통해 추론을 진행해 보았다. 결과, id가 22와 23인 데이터 모두 높은 수준으로 키를 예측한 것을 확인할 수 있었다. 하지만, 두 데이터는 우연히 모델에 잘 맞은 것일 수 있으므로 높은 이상치를 가지던 id 23을 22 대신 검증 데이터로 하여 처음부터 다시 진행해 보았다.

검증 결과

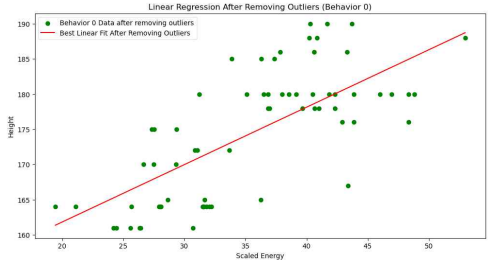



```
ID: 15.0 | Behavior: Walking | True Height: 172.0 | Predicted Height: 171.07
ID: 15.0 | Behavior: Walking | True Height: 172.0 | Predicted Height: 170.90
ID: 15.0 | Behavior: Walking | True Height: 172.0 | Predicted Height: 173.01
ID: 23.0 | Behavior: Walking | True Height: 173.0 | Predicted Height: 200.84
ID: 23.0 | Behavior: Walking | True Height: 173.0 | Predicted Height: 188.21
ID: 23.0 | Behavior: Walking | True Height: 173.0 | Predicted Height: 187.86
ID: 15.0 | Behavior: Jogging | True Height: 172.0 | Predicted Height: 173.17
ID: 15.0 | Behavior: Jogging | True Height: 172.0 | Predicted Height: 172.27
ID: 23.0 | Behavior: Jogging | True Height: 173.0 | Predicted Height: 182.34
ID: 23.0 | Behavior: Jogging | True Height: 173.0 | Predicted Height: 181.47
```

제거 전(걷기)



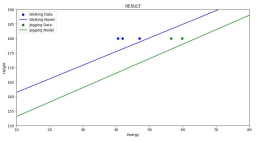
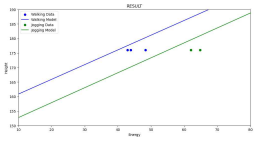
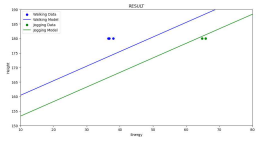
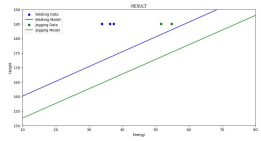
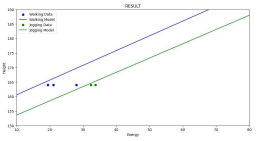
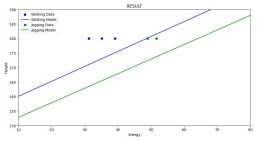
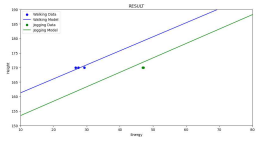
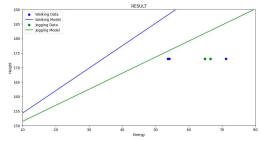
제거 후(걷기)



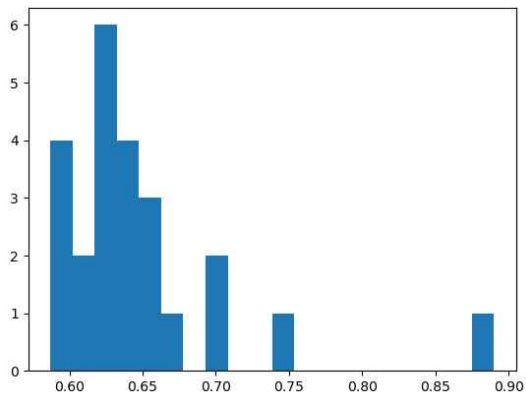
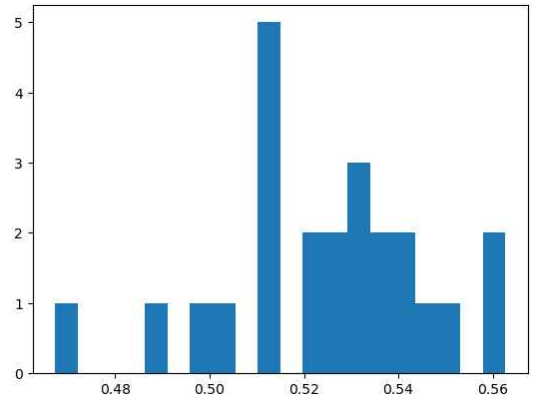
걷기	조깅
1차 : $y = 0.7388x + 147.9311$	1차 : $y = 0.5452x + 146.3989$
2차: $y = 0.8155x + 145.5199$	2차 : $y = 0.5527x + 145.2811$

이상치가 높은 데이터를 사용하니 예측 결과가 키가 173인 사람의 키를 200으로 예측하는 등, 잘못된 예측을 보여주는 것을 확인할 수 있었다. 또한, 조깅의 경우 모델이 크게 달라지지 않았지만, 걷기의 경우 기울기가 크게 변한 것을 볼 수 있다. 이전, 필터링 과정에서 높은 이상치를 보이던 데이터를 필터링을 통해 제거하기 때문에, 어떤 데이터를 검증 데이터로 사용하더라도 모델에 큰 변화가 없을 것이라 예상하였다. 실제로 필터링 결과 이상치가 높은 데이터를 제거하는 것은 유효하였다. 하지만, 이상치에 대해 추론을 정확히 하지 못하는 것을 떠나, 모델 자체가 크게 달라지는 것은 추가적인 검증이 필요해보였다.

따라서, 모델 추론에 사용되는 전반적인 과정을 모아 함수로 선언하여 id가 0번인 데이터부터 23번인 데이터까지 모든 데이터에 대해 하나씩 검증 데이터로 분리하고, 모델을 학습해보았다. 밑은 그 중 일부이다.

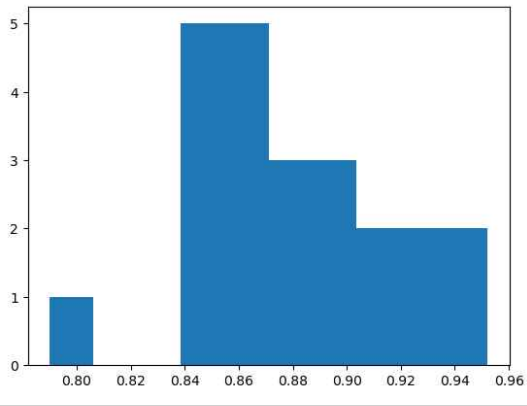
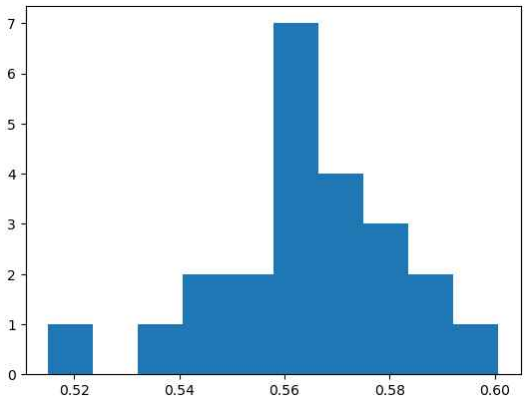
id : 1	id : 3	id : 5	id : 14
			
id : 17	**id : 19**	**id : 22**	**id : 23**
			

전반적으로 검증 데이터로 분리한 데이터가 얼마나 잘못된 값을 보이든 간에 비슷한 모델을 보였으나, id가 23인 모델만 다른 모델과 크게 다른 것을 볼 수 있었다. 이는 23번 데이터 셋이 모델 학습에 큰 영향을 미치는 것이 분명했다. 현재, 24개의 모델의 기울기를 히스토그램으로 표출하면 다음과 같다.

걷기		조깅	
			
평균	0.648	**평균**	0.525
표준편차	0.062	**표준편차**	0.021

걷기 데이터의 경우 특히 기울기가 0.6과 0.65 사이에 밀집되어있으며 조깅 데이터의 경우도 대부분이 0.50에서 0.56 사이에 있는 것을 확인할 수 있다. 전반적으로 대부분의 기울기는 일치하나, 걷기 데이터의 경우 0.88에 해당하는 매우 높은 기울기를 가지는 데이터가 한 개 존재하는 것을 볼 수 있다. 이는 id가 23인 데이터로 마찬가지로 id 23인 데이터를 제외하면

큰 높은 기울기가 나오며 큰 영향을 준다는 것을 알 수 있다. 이를 통해 id 23인 데이터는 모델 학습에 있어서 너무 큰 영향을 줌이 확실함으로 23을 제외하고 모델을 학습하고 확인해보았다.

걷기		조깅	
			
평균	0.880	**평균**	0.565
표준편차	0.037	**표준편차**	0.018

결과, 전체적으로 걷기 데이터는 평균이 증가하였으며, 표준편차 역시 줄어들었다. 23번 데이터가 큰 영향을 미치고 있던 것은 사실이었으며, 실제로 평균은 0.880으로 높게 나오는 것이 옳다는 것을 예상할 수 있다. 따라서 최종적으로 얻은 모델은 기울기가 평균값을 가지는 모델을 사용하는 것으로 선택하는 것이 옳을 것이다. 다음은 최종적으로 선택한 걷기와 조깅의 모델이다.

걷기	조깅
$y = 0.8809x + 143.2086$	$y = 0.5655x + 144.0638$

#결론

최종적으로 걷기와 조깅에 대한 선형 모델을 얻을 수 있었다. 하지만, 위 모델은 많은 개선 여지가 있다. 우선, 더 많은 데이터 셋을 사용하면 더욱 뛰어난 모델을 얻을 수 있을 것이다. 이전, 큰 이상치를 가지며 모델 학습에 큰 영향을 주었던 id가 23번인 데이터는 많은 데이터셋을 사용하여 평균에 가까운 데이터를 많이 수집함으로써 별도의 제거 없이 극복 가능할 것이라 예상할 수 있다. 또한, 이상치 제거, 즉 필터링 과정에서 1차적으로 학습한 모델에서 이상치를 제거하는 것이 아닌, 양자화를 통해 키를 여러 단계로 나누고, 같은 키에서 평균을 구해 이상치를 제거하면 더욱 뛰어난 모델을 얻을 수 있을 것이다. 만약, 충분한 데이터셋이 있다면, 성별에 따른 구분도 가능할 것이다. 두 번째로 센서의 민감도를 높여 더욱 정확한 센서값을 얻을 수 있다면 더욱 신뢰성 있는 에너지 값을 얻을 수 있을 것이다.

다만, 위 과정을 통해서도 해결되지 않는 문제는 있을 것이다. 보폭이라는 것이 일반적으로 신장에 따라 비례하는 것은 사실이나, 구체적인 체형, 그리고 개개인의 차이에 따라 보폭은 크게 좌지우지될 수 있어 잘못된 예측 결과를 얻을 수 있을 것이다. 이러한 한계점은 현재의 모델에서는 극복 불가능하지만, gps와 같은 다른 센서 데이터를 사용하는 등 여러 센서 데이터와 모델들을 종합하여 실제 핸드폰이 분실되었을 때 유용하게 사용할 수 있을 것이다.