```python
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

```
!ls /content/drive/MyDrive/action/data/raw
```

```
0.pkl   11.pkl  13.pkl  15.pkl  17.pkl  1.pkl  3.pkl  5.pkl  7.pkl  9.pkl
10.pkl  12.pkl  14.pkl  16.pkl  18.pkl  2.pkl  4.pkl  6.pkl  8.pkl  action_
```

```
!pip install filterpy
```

```
Collecting filterpy
  Downloading filterpy-1.4.5.zip (177 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 178.0/178.0 kB 4.5 MB/s eta 0
  Preparing metadata (setup.py) ... done
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.1
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/di
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/d
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.1
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/pytho
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-p
Building wheels for collected packages: filterpy
  Building wheel for filterpy (setup.py) ... done
  Created wheel for filterpy: filename=filterpy-1.4.5-py3-none-any.whl size
  Stored in directory: /root/.cache/pip/wheels/0f/0c/ea/218f266af4ad6268975
Successfully built filterpy
Installing collected packages: filterpy
Successfully installed filterpy-1.4.5
```

```
!pip install transforms3d
```

```
Collecting transforms3d
  Downloading transforms3d-0.4.2-py3-none-any.whl.metadata (2.8 kB)
Requirement already satisfied: numpy>=1.15 in /usr/local/lib/python3.10/dis
Downloading transforms3d-0.4.2-py3-none-any.whl (1.4 MB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 1.4/1.4 MB 13.8 MB/s eta 0:00:0
Installing collected packages: transforms3d
Successfully installed transforms3d-0.4.2
```

```
!pip install torch-geometric # install the missing package
```

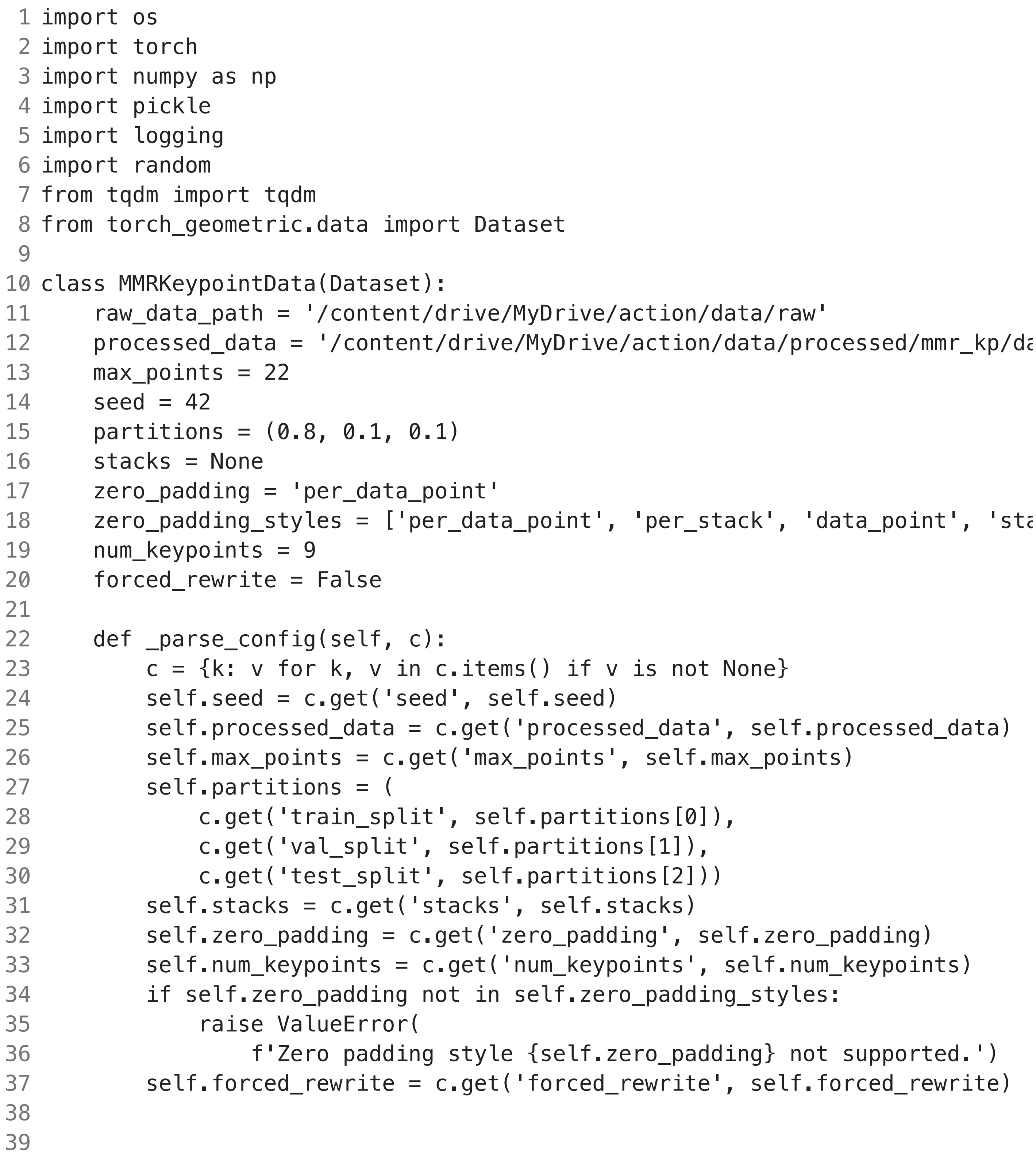
```
Collecting torch-geometric
  Downloading torch_geometric-2.6.1-py3-none-any.whl.metadata (63 kB)
                                              63.1/63.1 kB 1.8 MB/s eta 0:0
Requirement already satisfied: aiohttp in /usr/local/lib/python3.10/dist-pa
Requirement already satisfied: fsspec in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: jinja2 in /usr/local/lib/python3.10/dist-pac
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: psutil>=5.8.0 in /usr/local/lib/python3.10/d
Requirement already satisfied: pyparsing in /usr/local/lib/python3.10/dist-
Requirement already satisfied: requests in /usr/local/lib/python3.10/dist-p
Requirement already satisfied: tqdm in /usr/local/lib/python3.10/dist-packa
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in /usr/local/lib/py
Requirement already satisfied: aiosignal>=1.1.2 in /usr/local/lib/python3.1
Requirement already satisfied: attrs>=17.3.0 in /usr/local/lib/python3.10/d
Requirement already satisfied: frozenlist>=1.1.1 in /usr/local/lib/python3.
Requirement already satisfied: multidict<7.0,>=4.5 in /usr/local/lib/python
Requirement already satisfied: yarl<2.0,>=1.12.0 in /usr/local/lib/python3.
Requirement already satisfied: async-timeout<5.0,>=4.0 in /usr/local/lib/py
Requirement already satisfied: MarkupSafe>=2.0 in /usr/local/lib/python3.10
Requirement already satisfied: charset-normalizer<4,>=2 in /usr/local/lib/p
Requirement already satisfied: idna<4,>=2.5 in /usr/local/lib/python3.10/di
Requirement already satisfied: urllib3<3,>=1.21.1 in /usr/local/lib/python3
Requirement already satisfied: certifi>=2017.4.17 in /usr/local/lib/python3
Requirement already satisfied: typing-extensions>=4.1.0 in /usr/local/lib/p
Requirement already satisfied: propcache>=0.2.0 in /usr/local/lib/python3.1
Downloading torch_geometric-2.6.1-py3-none-any.whl (1.1 MB)
                                          1.1/1.1 MB 19.2 MB/s eta 0:00:0
Installing collected packages: torch-geometric
Successfully installed torch-geometric-2.6.1
```

```
import sys
sys.path.append('/content/drive/MyDrive/action')
```

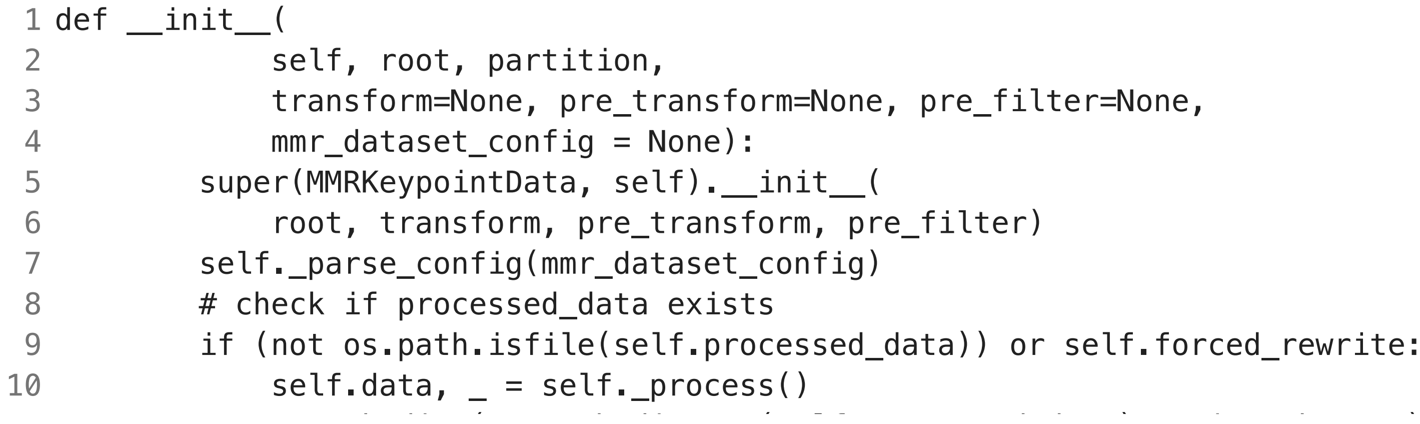
```
import os
import pickle

# Step 3: Specify the path to your folder in Google Drive
folder_path = '/content/drive/MyDrive/action/data/raw'  # Replace with your a

# Step 4: List files in the folder
extracted_files = os.listdir(folder_path)
print("Files in the folder:", extracted_files)

# Step 5: Load each `.pkl` file
pkl_files = [f for f in os.listdir(folder_path) if f.endswith('.pkl')]
data = []

for pkl_file in pkl_files:
    with open(os.path.join(folder_path, pkl_file), 'rb') as file:
        data.append(pickle.load(file))

# Step 6: Check if data is loaded
print(f"Loaded {len(data)} .pkl files.")

```

```
Files in the folder: ['7.pkl', '3.pkl', 'id.json', '10.pkl', '15.pkl', '13.
Loaded 19 .pkl files.
```

1. original code without clustering



```python
import os
import torch
import numpy as np
import pickle
import logging
import random
from tqdm import tqdm
from torch_geometric.data import Dataset

class MMRKeypointData(Dataset):
    raw_data_path = '/content/drive/MyDrive/action/data/raw'
    processed_data = '/content/drive/MyDrive/action/data/processed/mmr_kp/da
    max_points = 22
    seed = 42
    partitions = (0.8, 0.1, 0.1)
    stacks = None
    zero_padding = 'per_data_point'
    zero_padding_styles = ['per_data_point', 'per_stack', 'data_point', 'sta
    num_keypoints = 9
    forced_rewrite = False

    def _parse_config(self, c):
        c = {k: v for k, v in c.items() if v is not None}
        self.seed = c.get('seed', self.seed)
        self.processed_data = c.get('processed_data', self.processed_data)
        self.max_points = c.get('max_points', self.max_points)
        self.partitions = (
            c.get('train_split', self.partitions[0]),
            c.get('val_split', self.partitions[1]),
            c.get('test_split', self.partitions[2]))
        self.stacks = c.get('stacks', self.stacks)
        self.zero_padding = c.get('zero_padding', self.zero_padding)
        self.num_keypoints = c.get('num_keypoints', self.num_keypoints)
        if self.zero_padding not in self.zero_padding_styles:
            raise ValueError(
                f'Zero padding style {self.zero_padding} not supported.')
        self.forced_rewrite = c.get('forced_rewrite', self.forced_rewrite)


```



```python
def __init__(
            self, root, partition,
            transform=None, pre_transform=None, pre_filter=None,
            mmr_dataset_config = None):
        super(MMRKeypointData, self).__init__(
            root, transform, pre_transform, pre_filter)
        self._parse_config(mmr_dataset_config)
        # check if processed_data exists
        if (not os.path.isfile(self.processed_data)) or self.forced_rewrite:
            self.data, _ = self._process()
            os.makedirs(os.path.dirname(self.processed_data), exist_ok=True)
            with open(self.processed_data, 'wb') as f:
                pickle.dump(self.data, f)
        else:
            with open(self.processed_data, 'rb') as f:
                self.data = pickle.load(f)
        total_samples = len(self.data['train']) + len(self.data['val']) + len
        self.data = self.data[partition]
        self.num_samples = len(self.data)
        self.target_dtype = torch.float
        self.info = {
            'num_samples': self.num_samples,
            'num_keypoints': self.num_keypoints,
            'num_classes': None,
            'max_points': self.max_points,
            'stacks': self.stacks,
            'partition': partition,
        }
        logging.info(
            f'Loaded {partition} data with {self.num_samples} samples,'
            f' where the total number of samples is {total_samples}')

    def len(self):
        return self.num_samples

    def get(self, idx):
        data_point = self.data[idx]
        x = data_point['new_x']
        x = torch.tensor(x, dtype=torch.float32)
        y = torch.tensor(data_point['y'], dtype=self.target_dtype)
        return x, y

    @property
    def raw_file_names(self):
        file_names = [i for i in range(19)]
        return [f'{self.raw_data_path}/{i}.pkl' for i in file_names]

    def _process(self):
        data_list = []
        for fn in self.raw_file_names:
            logging.info(f'Loading {fn}')
            with open(fn, 'rb') as f:
                data_slice = pickle.load(f)
            data_list = data_list + data_slice
        num_samples = len(data_list)
        logging.info(f'Loaded {num_samples} data points')

        # stack and pad frames based on config
        data_list = self.transform_keypoints(data_list)
        data_list = self.stack_and_padd_frames(data_list)

        #random shuffle train and val data
        #random shuffle train and val data
        random.seed(self.seed)
        random.shuffle(data_list)

        # get partitions
        train_end = int(self.partitions[0] * num_samples)
        val_end = train_end + int(self.partitions[1] * num_samples)
        train_data = data_list[:train_end]
        val_data = data_list[train_end:val_end]
        test_data = data_list[val_end:]

        data_map = {
            'train': train_data,
            'val': val_data,
            'test': test_data,
        }
        return data_map, num_samples

    def stack_and_padd_frames(self, data_list):
        if self.stacks is None:
            return data_list
        # take multiple frames for each x
        xs = [d['x'] for d in data_list]
        stacked_xs = []
        padded_xs = []
        print("Stacking and padding frames...")
        pbar = tqdm(total=len(xs))

        if self.zero_padding in ['per_data_point', 'data_point']:
            for i in range(len(xs)):
                data_point = []
                for j in range(self.stacks):
                    if i - j >= 0:
                        mydata_slice = xs[i - j]
                        diff = self.max_points - mydata_slice.shape[0]
                        mydata_slice = np.pad(mydata_slice, ((0, max(diff, 0)
                        mydata_slice = mydata_slice[np.random.choice(len(myda
                        data_point.append(mydata_slice)
                    else:
                        data_point.append(np.zeros((self.max_points, 3)))
                padded_xs.append(np.concatenate(data_point, axis=0))
                pbar.update(1)
        elif self.zero_padding in ['per_stack', 'stack']:
            for i in range(len(xs)):
                start = max(0, i - self.stacks)
                stacked_xs.append(np.concatenate(xs[start:i+1], axis=0))
                pbar.update(0.5)
            for x in stacked_xs:
                diff = self.max_points * self.stacks - x.shape[0]
                x = np.pad(x, ((0, max(diff, 0)), (0, 0)), 'constant')
                x = x[np.random.choice(len(x), self.max_points * self.stacks,
                padded_xs.append(x)
                pbar.update(0.5)
        else:
            raise NotImplementedError()
        pbar.close()
        print("Stacking and padding frames done")
        # remap padded_xs to data_list
        new_data_list = [{**d, 'new_x': x} for d, x in zip(data_list, padded_
        return new_data_list

    kp18_names = ['NOSE', 'NECK', 'RIGHT_SHOULDER', 'RIGHT_ELBOW',
                  'RIGHT_WRIST', 'LEFT_SHOULDER', 'LEFT_ELBOW',
                  'LEFT_WRIST', 'RIGHT_HIP', 'RIGHT_KNEE',
                  'RIGHT_ANKLE', 'LEFT_HIP', 'LEFT_KNEE',
                  'LEFT_ANKLE', 'RIGHT_EYE', 'LEFT_EYE',
                  'RIGHT_EAR', 'LEFT_EAR']
    kp9_names = ['RIGHT_SHOULDER', 'RIGHT_ELBOW',
                 'LEFT_SHOULDER', 'LEFT_ELBOW',
                 'RIGHT_HIP', 'RIGHT_KNEE',
                 'LEFT_HIP', 'LEFT_KNEE', 'HEAD']
    head_names = ['NOSE', 'RIGHT_EYE', 'LEFT_EYE', 'RIGHT_EAR', 'LEFT_EAR']
    def transform_keypoints(self, data_list):
        if self.num_keypoints == 18:
            return data_list

        print("Transforming keypoints ...")
        self.kp9_idx = [self.kp18_names.index(n) for n in self.kp9_names[:-1]
        self.head_idx = [self.kp18_names.index(n) for n in self.head_names]
        for data in tqdm(data_list, total=len(data_list)):
            kpts = data['y']
            kpts_new = kpts[self.kp9_idx]
            head = np.mean(kpts[self.head_idx], axis=0)
            kpts_new = np.concatenate((kpts_new, head[None]))
            assert kpts_new.shape == (9, 3)
            data['y'] = kpts_new
        print("Transforming keypoints done")
        return data_list

class MMRActionData(MMRKeypointData):
    processed_data = 'content/drive/MyDrive/action/data/processed/mmr_action/
    def __init__(self, *args, **kwargs):
        self.action_label = np.load('/content/drive/MyDrive/action/data/raw/a
        super().__init__(*args, **kwargs)
        self.info['num_classes'] = len(np.unique(self.action_label))-1 # exce
        self.target_dtype = torch.int64

    def _process(self):
        data_list = []
        for fn in self.raw_file_names:
            logging.info(f'Loading {fn}')
            with open(fn, 'rb') as f:
                data_slice = pickle.load(f)
            data_list = data_list + data_slice
            data_list = data_list + data_slice

        for i, data in enumerate(data_list):
            data['y'] = self.action_label[i]
        data_list = [d for d in data_list if d['y']!=-1]

        data_list = self.stack_and_padd_frames(data_list)
        num_samples = len(data_list)
        logging.info(f'Loaded {num_samples} data points')

        # get partitions
        train_end = int(self.partitions[0] * num_samples)
        val_end = train_end + int(self.partitions[1] * num_samples)
        train_data = data_list[:train_end]
        val_data = data_list[train_end:val_end]
        test_data = data_list[val_end:]

        # #random shuffle train and val data
        random.seed(self.seed)
        random.shuffle(train_data)
        random.shuffle(val_data)

        data_map = {
            'train': train_data,
            'val': val_data,
            'test': test_data,
        }
        return data_map, num_samples

    def stack_and_padd_frames(self, data_list):
        if self.stacks is None:
            return data_list
        # take multiple frames for each x
        xs = [d['x'] for d in data_list]
        stacked_xs = []
        padded_xs = []
        print("Stacking and padding frames...")
        pbar = tqdm(total=len(xs))

        if self.zero_padding in ['per_data_point', 'data_point']:
            for i in range(len(xs)):
                data_point = []
                for j in range(self.stacks):
                    if i - j >= 0 and self.action_label[i] == self.action_lab
                        mydata_slice = xs[i - j]
                        diff = self.max_points - mydata_slice.shape[0]
                        mydata_slice = np.pad(mydata_slice, ((0, max(diff, 0)
                        mydata_slice = mydata_slice[np.random.choice(len(myda
                        data_point.append(mydata_slice)
                    else:
                        data_point.append(np.zeros((self.max_points, 3)))
                padded_xs.append(np.concatenate(data_point, axis=0))
                pbar.update(1)
        elif self.zero_padding in ['per_stack', 'stack']:
            for i in range(len(xs)):
                start = max(0, i - self.stacks)
                while self.action_label[i] != self.action_label[start]:
                    start = start + 1
                stacked_xs.append(np.concatenate(xs[start:i+1], axis=0))
                pbar.update(0.5)
            for x in stacked_xs:
                diff = self.max_points * self.stacks - x.shape[0]
                x = np.pad(x, ((0, max(diff, 0)), (0, 0)), 'constant')
                x = x[np.random.choice(len(x), self.max_points * self.stacks,
                padded_xs.append(x)
                pbar.update(0.5)
        else:
            raise NotImplementedError()
        pbar.close()
        print("Stacking and padding frames done")
        # remap padded_xs to data_list
        new_data_list = [{**d, 'new_x': x} for d, x in zip(data_list, padded_
        return new_data_list


# Testing the MMRActionData class
if __name__ == "__main__":
    # Define root directory and configuration
    root_dir = ''  # Root directory is the current directory
    mmr_dataset_config = {
        'processed_data': '/content/drive/MyDrive/action/data/processed/mmr_a
        'stacks': 5,  # example config, adjust according to needs
        'max_points': 22,
        'num_keypoints': 9,
        'zero_padding': 'per_data_point',
        'seed': 42,
        'forced_rewrite': True # Added line
    }

    # Load train data
    train_dataset = MMRActionData(root=root_dir, partition='train', mmr_datas
    # Load validation data
    val_dataset = MMRActionData(root=root_dir, partition='val', mmr_dataset_c
    # Load test data
    test_dataset = MMRActionData(root=root_dir, partition='test', mmr_dataset

    # Print out the shapes of the train, val, and test data
    print(f"Train data shape: {len(train_dataset)} samples")
    print(f"Validation data shape: {len(val_dataset)} samples")
    print(f"Test data shape: {len(test_dataset)} samples")

    # Optional: inspect a specific sample (e.g., the first one) in the datase
    x_train, y_train = train_dataset.get(0)
    x_val, y_val = val_dataset.get(0)
    x_test, y_test = test_dataset.get(0)

    for data, label in train_dataset:
        print(f"Train data shape: {data.shape}")
        print(f"Train label shape: {label.shape}")

    for data, label in val_dataset:
        print(f"Val data shape: {data.shape}")
        print(f"Val label shape: {label.shape}")

    # print(f"First train sample shape: x={x_train.shape}, y={y_train.shape}"
    # print("First train sample values:")
    # print(f"x_train: {x_train}")
    # print(f"y_train: {y_train}")

    # print(f"First val sample shape: x={x_val.shape}, y={y_val.shape}")
    # print("First val sample values:")
    # print(f"x_val: {x_val}")
    # print(f"y_val: {y_val}")

    # print(f"First test sample shape: x={x_test.shape}, y={y_test.shape}")
    # print("First test sample values:")
    # print(f"x_test: {x_test}")
    # print(f"y_test: {y_test}")

    print(f"First train sample shape: x={x_train.shape}, y={y_train.shape}")
    print(f"First val sample shape: x={x_val.shape}, y={y_val.shape}")
    print(f"First test sample shape: x={x_test.shape}, y={y_test.shape}")

```

```
# import os
# import torch
# import numpy as np
# import pickle
# import logging
# import random
# from tqdm import tqdm
# from torch_geometric.data import
# import pandas as pd

# def Normalize(x, x_min, x_max):
#     """Normalize a value x to a r
#     return (x - x_min) / (x_max -

# class MMRKeypointData(Dataset):
#     raw_data_path = '/content/dri
#     processed_data = '/content/dr
#     max_points = 22
#     seed = 42
#     partitions = (0.8, 0.1, 0.1)
```

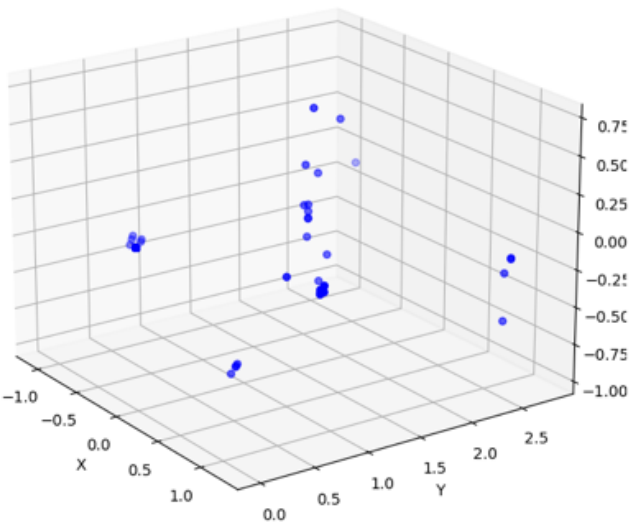
"@property" is not an allowed annotation - allowed values include [@param, @title, @markdown].

```
#     stacks = None
#     zero_padding = 'per_data_point
#     zero_padding_styles = ['per_da
#     num_keypoints = 9
#     forced_rewrite = False

#     def _parse_config(self, c):
#         c = {k: v for k, v in c.it
#         self.seed = c.get('seed',
#         self.processed_data = c.ge
#         self.max_points = c.get('m
#         self.partitions = (
#             c.get('train_split', s
#             c.get('val_split', sel
#             c.get('test_split', se
#         self.stacks = c.get('stack
#         self.zero_padding = c.get(
#         self.num_keypoints = c.get
#         if self.zero_padding not i
#             raise ValueError(
#                 f'Zero padding sty
#         self.forced_rewrite = c.ge

#     def __init__(self, root, parti
#         super(MMRKeypointData, sel
#         self._parse_config(mmr_dat
#         # Check if processed_data
#         if (not os.path.isfile(sel
#             self.data, _ = self._p
#             os.makedirs(os.path.di
#             with open(self.process
#                 pickle.dump(self.d
#         else:
#             with open(self.process
#                 self.data = pickle

#         total_samples = len(self.d
#         self.data = self.data[part
#         self.num_samples = len(sel
#         self.target_dtype = torch.
#         self.info = {
#             'num_samples': self.nu
#             'num_keypoints': self.
#             'num_classes': None,
#             'max_points': self.max
#             'stacks': self.stacks,
#             'partition': partition
#         }
#         logging.info(f'Loaded {par

#     def __len__(self):
```

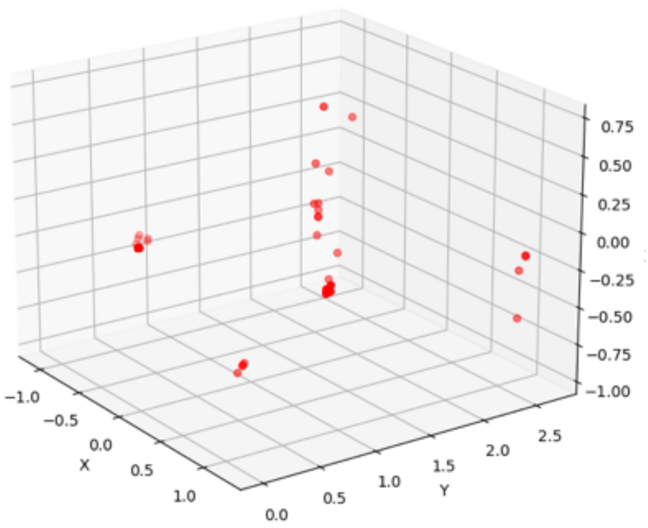
```
#         return self.num_samples

#     def __getitem__(self, idx):
#         data_point = self.data[id
#         x = data_point['new_x']
#         x = torch.tensor(x, dtype=
#         y = torch.tensor(data_poi
#         return x, y

#     @property
#     def raw_file_names(self):
#         file_names = [i for i in
#         return [f'{self.raw_data_

#     def _process(self):
#         data_list = []
#         for fn in self.raw_file_na
#             logging.info(f'Loadin
#             with open(fn, 'rb') a
#                 data_slice = pick
#             data_list += data_sli
#         num_samples = len(data_li
#         logging.info(f'Loaded {nu

#         # Stack and pad frames ba
#         data_list = self.transfor
#         data_list = self.stack_an

#         # Random shuffle train an
#         random.seed(self.seed)
#         random.shuffle(data_list)

#         # Get partitions
#         train_end = int(self.part
#         val_end = train_end + int
#         train_data = data_list[:t
#         val_data = data_list[trai
#         test_data = data_list[val_

#         data_map = {
#             'train': train_data,
#             'val': val_data,
#             'test': test_data,
#         }
#         return data_map, num_samp

#     def stack_and_padd_frames(sel
#         if self.stacks is None:
#             return data_list
#         # Take multiple frames fo
#         xs = [d['x'] for d in dat
#         stacked_xs = []
#         padded_xs = []
#         print("Stacking and paddin
#         pbar = tqdm(total=len(xs)

#         if self.zero_padding in [
#             for i in range(len(xs
#                 data_point = []
#                 for j in range(se
#                     if i - j >= 0
#                         mydata_sl
#                         diff = se
#                         mydata_sl
#                         data_poin
#                     else:
#                         data_poin
#                 padded_xs.append(
#                 pbar.update(1)
#         else:
#             raise NotImplementedE
#         pbar.close()
#         print("Stacking and paddin
#         # Remap padded_xs to data_
#         new_data_list = [{**d, 'ne
#         return new_data_list

#     kp18_names = ['NOSE', 'NECK',
#                   'RIGHT_WRIST',
#                   'LEFT_WRIST', 'R
#                   'RIGHT_ANKLE',
#                   'LEFT_ANKLE', 'R
#                   'RIGHT_EAR', 'LE
#     kp9_names = ['RIGHT_SHOULDER',
#                  'LEFT_SHOULDER',
#                  'RIGHT_HIP', 'RIG
#                  'LEFT_HIP', 'LEFT

#     def transform_keypoints(self,
#         if self.num_keypoints == 1
#             return data_list

#         print("Transforming keypo
#         self.kp9_idx = [self.kp18_
#         for data in tqdm(data_lis
#             kpts = data['y']
#             kpts_new = kpts[self.
#             head = np.mean(kpts[se
#             kpts_new = np.concate
#             assert kpts_new.shape
#             data['y'] = kpts_new
#         print("Transforming keypo
#         return data_list


# class MMRActionData(MMRKeypointDat
#     processed_data = '/content/dri
#     def __init__(self, *args, **kw
#         self.action_label = np.loa
#         super().__init__(*args, **
#         self.info['num_classes'] =
#         self.target_dtype = torch.

#         # Verify labels: Check sha
#         print(f"Action labels shap
#         print(f"Unique action labe

#     def _process(self):
#         data_list = []
#         for fn in self.raw_file_na
#             logging.info(f'Loading
#             with open(fn, 'rb') as
#                 data_slice = pickl
#             data_list += data_slic

#         for i, data in enumerate(d
#             data['y'] = self.actio
#             # Verify label assignm
#             if data['y'] == -1:
#                 print(f"Warning: D

#         data_list = [d for d in da

#         # Normalization step (befo
#         self.normalize_features(da

#         num_samples = len(data_lis
#         logging.info(f'Loaded {num

#         data_list = self.stack_and

#         # Get partitions
#         train_end = int(self.parti
#         val_end = train_end + int(
#         train_data = data_list[:tr
#         val_data = data_list[train
#         test_data = data_list[val_

#         # Random shuffle train and
#         random.seed(self.seed)
#         random.shuffle(train_data)
#         random.shuffle(val_data)
#         data_map = {
#             'train': train_data,
#             'val': val_data,
#             'test': test_data,
#         }
#         return data_map, num_samp

#     def normalize_features(self, c
#         """Normalize intensity fea
#         intensity_values = np.arra
#         intensity_min = intensity_
#         intensity_max = intensity_

#         for data in data_list:
#             data['normalized_inte

#         print(f"Normalized intens

# # Testing the MMRActionData class
# if __name__ == "__main__":
#     # Define root directory and co
#     root_dir = ''  # Root director
#     mmr_dataset_config = {
#         'processed_data': '/conte
#         'stacks': 5,  # Example co
#         'max_points': 22,
#         'num_keypoints': 9,
#         'zero_padding': 'per_data_
#         'seed': 42,
#         'forced_rewrite': True
#     }

#     # Load train data
#     train_dataset = MMRActionData
#     # Load validation data
#     val_dataset = MMRActionData(r
#     # Load test data
#     test_dataset = MMRActionData(

#     # Print out the shapes of the
#     print(f"Train data shape: {le
#     print(f"Validation data shape
#     print(f"Test data shape: {len

```

```
import os
import torch
import numpy as np
import pickle
import logging
import random
from tqdm import tqdm
from torch_geometric.data import Dataset
from sklearn.cluster import DBSCAN  # Import DBSCAN
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

class MMRKeypointData(Dataset):
    raw_data_path = '/content/drive/MyDrive/action/data/raw'  # Updated path
    processed_data = '/content/drive/MyDrive/action/data/processed/mmr_kp/da
    max_points = 22
    seed = 42
    partitions = (0.8, 0.1, 0.1)
    stacks = None
    zero_padding = 'per_data_point'
    zero_padding_styles = ['per_data_point', 'per_stack', 'data_point', 'sta
    num_keypoints = 9
    forced_rewrite = False

    def _parse_config(self, c):
        c = {k: v for k, v in c.items() if v is not None}
        self.seed = c.get('seed', self.seed)
        self.processed_data = c.get('processed_data', self.processed_data)
        self.max_points = c.get('max_points', self.max_points)
        self.partitions = (
            c.get('train_split', self.partitions[0]),
            c.get('val_split', self.partitions[1]),
            c.get('test_split', self.partitions[2]))
        self.stacks = c.get('stacks', self.stacks)
        self.zero_padding = c.get('zero_padding', self.zero_padding)
        self.num_keypoints = c.get('num_keypoints', self.num_keypoints)
        if self.zero_padding not in self.zero_padding_styles:
            raise ValueError(
                f'Zero padding style {self.zero_padding} not supported.')
        self.forced_rewrite = c.get('forced_rewrite', self.forced_rewrite)

    def __init__(
            self, root, partition,
            transform=None, pre_transform=None, pre_filter=None,
            mmr_dataset_config = None):
        super(MMRKeypointData, self).__init__(
            root, transform, pre_transform, pre_filter)
        self._parse_config(mmr_dataset_config)
        # Check if processed_data exists
        if (not os.path.isfile(self.processed_data)) or self.forced_rewrite:
            self.data, _ = self._process()
            # Create directory if it doesn't exist
            os.makedirs(os.path.dirname(self.processed_data), exist_ok=True)
            with open(self.processed_data, 'wb') as f:
                pickle.dump(self.data, f)
        else:
            with open(self.processed_data, 'rb') as f:
                self.data = pickle.load(f)
        total_samples = len(self.data['train']) + len(self.data['val']) + le
        self.data = self.data[partition]
        self.num_samples = len(self.data)
        self.target_dtype = torch.float
        self.info = {
            'num_samples': self.num_samples,
            'num_keypoints': self.num_keypoints,
            'num_classes': None,
            'max_points': self.max_points,
            'stacks': self.stacks,
            'partition': partition,
        }
        logging.info(
            f'Loaded {partition} data with {self.num_samples} samples,'
            f' where the total number of samples is {total_samples}')

    def len(self):
        return self.num_samples

    def get(self, idx):
        data_point = self.data[idx]
        x = data_point['new_x']
        x = torch.tensor(x, dtype=torch.float32)
        y = torch.tensor(data_point['y'], dtype=self.target_dtype)
        return x, y

    @property
    def raw_file_names(self):
        file_names = [i for i in range(19)]
        return [f'{self.raw_data_path}/{i}.pkl' for i in file_names]

    def _process(self):
        data_list = []
        for fn in self.raw_file_names:
            logging.info(f'Loading {fn}')
            with open(fn, 'rb') as f:
                data_slice = pickle.load(f)
            data_list = data_list + data_slice
        num_samples = len(data_list)
        logging.info(f'Loaded {num_samples} data points')

        # Transform keypoints based on config
        data_list = self.transform_keypoints(data_list)

        # Stack and pad frames
        data_list = self.stack_and_padd_frames(data_list)

        # Apply DBSCAN clustering
        data_list = self.apply_dbscan(data_list)

        # Random shuffle train and val data
        random.seed(self.seed)
        random.shuffle(data_list)

        # Get partitions
        train_end = int(self.partitions[0] * num_samples)
        val_end = train_end + int(self.partitions[1] * num_samples)
        train_data = data_list[:train_end]
        val_data = data_list[train_end:val_end]
        test_data = data_list[val_end:]

        data_map = {
            'train': train_data,
            'val': val_data,
            'test': test_data,
        }
        return data_map, num_samples

    def stack_and_padd_frames(self, data_list):
        if self.stacks is None:
            return data_list
        # Take multiple frames for each x
        xs = [d['x'] for d in data_list]
        padded_xs = []
        print("Stacking and padding frames...")
        pbar = tqdm(total=len(xs))

        if self.zero_padding in ['per_data_point', 'data_point']:
            for i in range(len(xs)):
                data_point = []
                for j in range(self.stacks):
                    if i - j >= 0:
                        mydata_slice = xs[i - j]
                        diff = self.max_points - mydata_slice.shape[0]
                        mydata_slice = np.pad(mydata_slice, ((0, max(diff, 0
                        if mydata_slice.shape[0] > self.max_points:
                            idx = np.random.choice(mydata_slice.shape[0], se
                            mydata_slice = mydata_slice[idx]
                        data_point.append(mydata_slice)
                    else:
                        data_point.append(np.zeros((self.max_points, 3)))
                padded_xs.append(np.concatenate(data_point, axis=0))
                pbar.update(1)
        elif self.zero_padding in ['per_stack', 'stack']:
            stacked_xs = []
            for i in range(len(xs)):
                start = max(0, i - self.stacks + 1)
                stacked_xs.append(np.concatenate(xs[start:i+1], axis=0))
                pbar.update(0.5)
            for x in stacked_xs:
                diff = self.max_points * self.stacks - x.shape[0]
                x = np.pad(x, ((0, max(diff, 0)), (0, 0)), 'constant')
                if x.shape[0] > self.max_points * self.stacks:
                    idx = np.random.choice(x.shape[0], self.max_points * sel
                    x = x[idx]
                padded_xs.append(x)
                pbar.update(0.5)
        else:
            raise NotImplementedError()
        pbar.close()
        print("Stacking and padding frames done")
        # Remap padded_xs to data_list
        new_data_list = [{**d, 'new_x': x} for d, x in zip(data_list, padded
        return new_data_list

    # Modified apply_dbscan method
    def apply_dbscan(self, data_list):
        print("Applying DBSCAN clustering...")
        desired_num_points = self.max_points * (self.stacks if self.stacks e
        for data in tqdm(data_list, total=len(data_list)):
            x = data['new_x']  # Shape: [num_points, num_features], e.g., [1
            # Store the data before clustering
            data['new_x_before_dbscan'] = x.copy()
            # Apply DBSCAN clustering
            clustering = DBSCAN(eps=0.5, min_samples=3).fit(x)  # Updated pa
            labels = clustering.labels_
            # Keep only the points that are in clusters (labels != -1)
            mask = labels != -1
            x_filtered = x[mask]
            # Handle cases where x_filtered is empty or has too few/many poi
            num_points = x_filtered.shape[0]
            if num_points == 0:
                # All points are noise; pad with zeros
                x_filtered = np.zeros((desired_num_points, x.shape[1]))
            elif num_points < desired_num_points:
                # Pad with zeros
                diff = desired_num_points - num_points
                x_filtered = np.pad(x_filtered, ((0, diff), (0, 0)), 'consta
            elif num_points > desired_num_points:
                # Randomly sample desired_num_points
                idx = np.random.choice(num_points, desired_num_points, repla
                x_filtered = x_filtered[idx]
            # Else, num_points == desired_num_points; no change needed
            data['new_x'] = x_filtered
        print("DBSCAN clustering applied.")
        return data_list

    kp18_names = ['NOSE', 'NECK', 'RIGHT_SHOULDER', 'RIGHT_ELBOW',
                  'RIGHT_WRIST', 'LEFT_SHOULDER', 'LEFT_ELBOW',
                  'LEFT_WRIST', 'RIGHT_HIP', 'RIGHT_KNEE',
                  'RIGHT_ANKLE', 'LEFT_HIP', 'LEFT_KNEE',
                  'LEFT_ANKLE', 'RIGHT_EYE', 'LEFT_EYE',
                  'RIGHT_EAR', 'LEFT_EAR']
    kp9_names = ['RIGHT_SHOULDER', 'RIGHT_ELBOW',
                 'LEFT_SHOULDER', 'LEFT_ELBOW',
                 'RIGHT_HIP', 'RIGHT_KNEE',
                 'LEFT_HIP', 'LEFT_KNEE', 'HEAD']
    head_names = ['NOSE', 'RIGHT_EYE', 'LEFT_EYE', 'RIGHT_EAR', 'LEFT_EAR']

    def transform_keypoints(self, data_list):
        if self.num_keypoints == 18:
            return data_list

        print("Transforming keypoints ...")
        self.kp9_idx = [self.kp18_names.index(n) for n in self.kp9_names[:-1
        self.head_idx = [self.kp18_names.index(n) for n in self.head_names]
        for data in tqdm(data_list, total=len(data_list)):
            kpts = data['y']
            kpts_new = kpts[self.kp9_idx]
            head = np.mean(kpts[self.head_idx], axis=0)
            kpts_new = np.concatenate((kpts_new, head[None]))
            assert kpts_new.shape == (9, 3)
            data['y'] = kpts_new
        print("Transforming keypoints done")
        return data_list

class MMRActionData(MMRKeypointData):
    processed_data = '/content/drive/MyDrive/action/data/processed/mmr_actio
    def __init__(self, *args, **kwargs):
        self.action_label = np.load('/content/drive/MyDrive/action/data/raw/
        super().__init__(*args, **kwargs)
        self.info['num_classes'] = len(np.unique(self.action_label))-1 # exc
        self.target_dtype = torch.int64

    def _process(self):
        data_list = []
        for fn in self.raw_file_names:
            logging.info(f'Loading {fn}')
            with open(fn, 'rb') as f:
                data_slice = pickle.load(f)
            data_list = data_list + data_slice

        for i, data in enumerate(data_list):
            data['y'] = self.action_label[i]
        data_list = [d for d in data_list if d['y']!=-1]

        data_list = self.stack_and_padd_frames(data_list)

        # Apply DBSCAN clustering
        data_list = self.apply_dbscan(data_list)

        num_samples = len(data_list)
        logging.info(f'Loaded {num_samples} data points')

        # Get partitions
        train_end = int(self.partitions[0] * num_samples)
        val_end = train_end + int(self.partitions[1] * num_samples)
        train_data = data_list[:train_end]
        val_data = data_list[train_end:val_end]
        test_data = data_list[val_end:]

        # Random shuffle train and val data
        random.seed(self.seed)
        random.shuffle(train_data)
        random.shuffle(val_data)

        data_map = {
            'train': train_data,
            'val': val_data,
            'test': test_data,
        }
        return data_map, num_samples

# Testing the MMRActionData class with DBSCAN clustering and visualization
if __name__ == "__main__":
    # Define root directory and configuration
    root_dir = ''  # Root directory is the current directory
    mmr_dataset_config = {
        'processed_data': '/content/drive/MyDrive/action/data/processed/mmr_
        'stacks': 5,  # Example config, adjust according to needs
        'max_points': 22,
        'num_keypoints': 9,
        'zero_padding': 'per_data_point',
        'seed': 42,
        'forced_rewrite': True  # Set to True to process data again
    }

    # Load train data
    train_dataset = MMRActionData(root=root_dir, partition='train', mmr_data
    # Load validation data
    val_dataset = MMRActionData(root=root_dir, partition='val', mmr_dataset_
    # Load test data
    test_dataset = MMRActionData(root=root_dir, partition='test', mmr_datase

    # Print out the shapes of the train, val, and test data
    print(f"Train data shape: {len(train_dataset)} samples")
    print(f"Validation data shape: {len(val_dataset)} samples")
    print(f"Test data shape: {len(test_dataset)} samples")

    # Visualization of 5 random sequences
    # Combine datasets for selection (you can choose from any partition)
    combined_data = train_dataset.data + val_dataset.data + test_dataset.dat
    selected_sequences = random.sample(combined_data, 5)

    for idx, data_point in enumerate(selected_sequences):
        x_before = data_point['new_x_before_dbscan']
        x_after = data_point['new_x']

        fig = plt.figure(figsize=(12, 6))

        # Plot before DBSCAN
        ax1 = fig.add_subplot(121, projection='3d')
        ax1.scatter(x_before[:, 0], x_before[:, 1], x_before[:, 2], c='b', m
        ax1.set_title(f'Sequence {idx+1} Before DBSCAN')
        ax1.set_xlabel('X')
        ax1.set_ylabel('Y')
        ax1.set_zlabel('Z')
        ax1.view_init(elev=20., azim=-35)

        # Plot after DBSCAN
        ax2 = fig.add_subplot(122, projection='3d')
        ax2.scatter(x_after[:, 0], x_after[:, 1], x_after[:, 2], c='r', mark
        ax2.set_title(f'Sequence {idx+1} After DBSCAN')
        ax2.set_xlabel('X')
        ax2.set_ylabel('Y')
        ax2.set_zlabel('Z')
        ax2.view_init(elev=20., azim=-35)

        plt.tight_layout()
        plt.show()
```

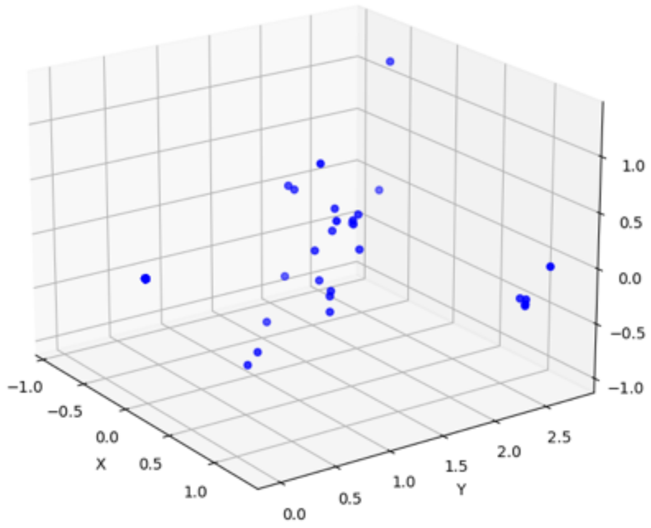
```
Stacking and padding frames...
100%|██████████| 212920/212920 [00:51<00:00, 4161.67it/s]
Stacking and padding frames done
Applying DBSCAN clustering...
100%|██████████| 212920/212920 [05:44<00:00, 617.68it/s]
DBSCAN clustering applied.
Stacking and padding frames...
100%|██████████| 212920/212920 [00:55<00:00, 3841.03it/s]
Stacking and padding frames done
Applying DBSCAN clustering...
100%|██████████| 212920/212920 [05:49<00:00, 609.07it/s]
DBSCAN clustering applied.
Stacking and padding frames...
100%|██████████| 212920/212920 [00:56<00:00, 3759.31it/s]
Stacking and padding frames done
Applying DBSCAN clustering...
100%|██████████| 212920/212920 [05:46<00:00, 615.13it/s]
DBSCAN clustering applied.
Train data shape: 170336 samples
Validation data shape: 21292 samples
Test data shape: 21292 samples
```



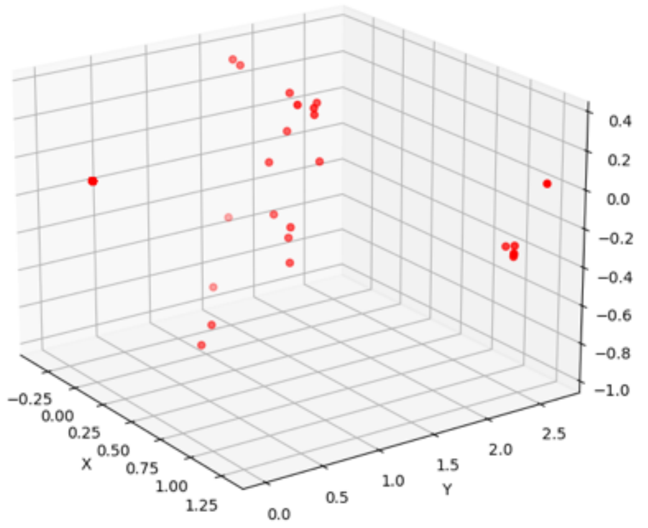
Sequence 2 Before DBSCAN



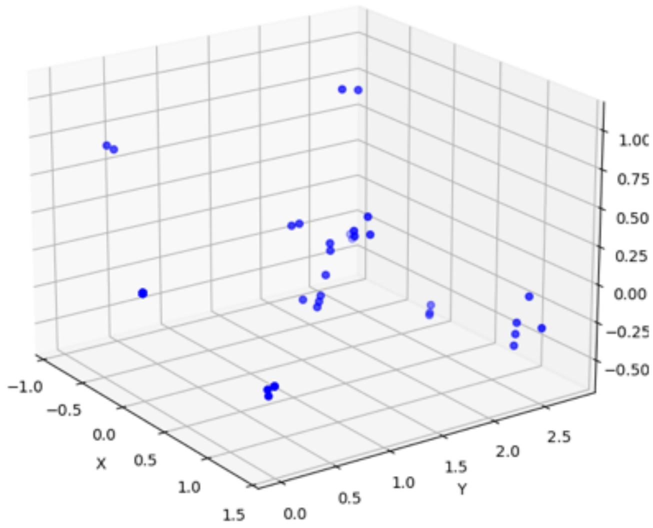
Sequence 2 After DBSCAN



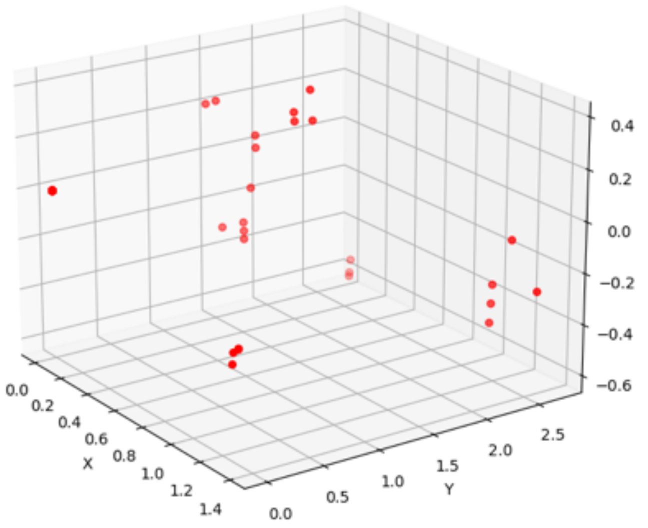
Sequence 3 Before DBSCAN



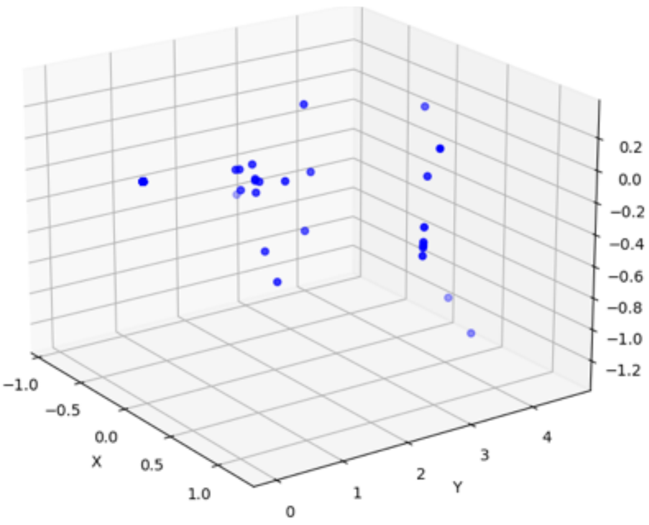
Sequence 3 After DBSCAN



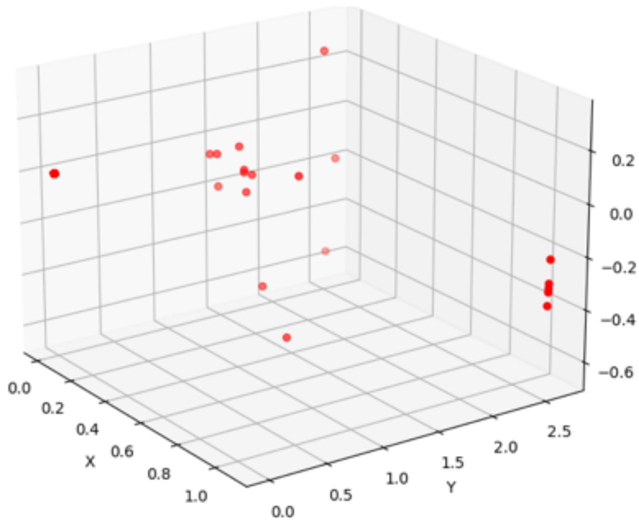
Sequence 4 Before DBSCAN



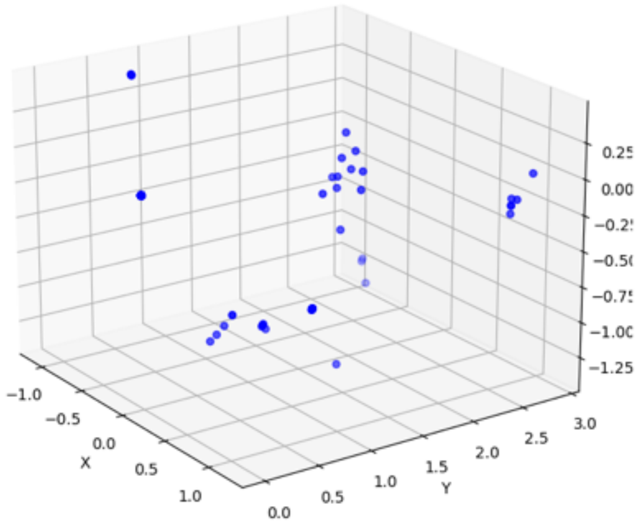
Sequence 4 After DBSCAN

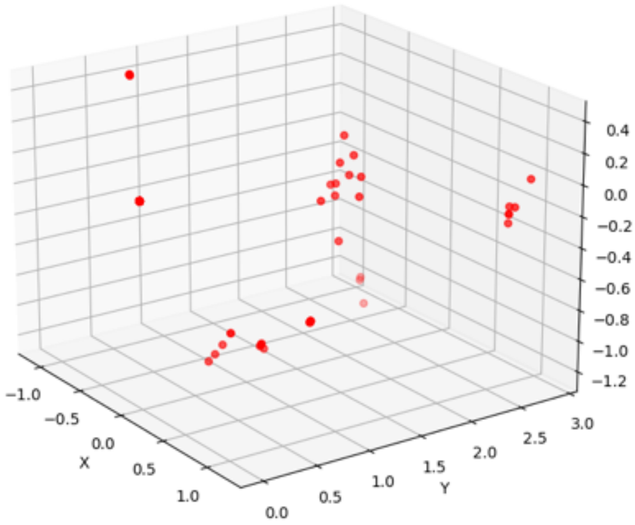


Sequence 5 Before DBSCAN



Sequence 5 After DBSCAN





"@property" is not an allowed annotation - allowed values include [@param, @title, @markdown].

```
# import os
# import torch
# import numpy as np
# import pickle
# import logging
# import random
# from tqdm import tqdm
# from torch_geometric.data import D
# from sklearn.cluster import DBSCAN
# import matplotlib.pyplot as plt
# from mpl_toolkits.mplot3d import A

# class MMRKeypointData(Dataset):
#     raw_data_path = '/content/driv
#     processed_data = '/content/dri
#     max_points = 22
#     seed = 42
#     partitions = (0.8, 0.1, 0.1)
#     stacks = None
#     zero_padding = 'per_data_point
#     zero_padding_styles = ['per_da
#     num_keypoints = 9
#     forced_rewrite = False

#     def _parse_config(self, c):
#         c = {k: v for k, v in c.it
#         self.seed = c.get('seed',
#         self.processed_data = c.ge
#         self.max_points = c.get('m
#         self.partitions = (
#             c.get('train_split', s
#             c.get('val_split', sel
#             c.get('test_split', se
#         self.stacks = c.get('stack
#         self.zero_padding = c.get(
#         self.num_keypoints = c.get
#         if self.zero_padding not i
#             raise ValueError(f'Zer
#         self.forced_rewrite = c.ge

#     def __init__(self, root, part
#                  transform=None, p
#         super(MMRKeypointData, sel
#         if mmr_dataset_config is n
#             self._parse_config(mmr
#         if (not os.path.isfile(sel
#             self.data, _ = self._p
#             os.makedirs(os.path.di
#             with open(self.process
#                 pickle.dump(self.d
#         else:
#             with open(self.process
#                 self.data = pickle

#         total_samples = len(self.d
#         self.data = self.data[part
#         self.num_samples = len(sel
#         self.target_dtype = torch.
#         self.info = {
#             'num_samples': self.nu
#             'num_keypoints': self.
#             'num_classes': None,
#             'max_points': self.max
#             'stacks': self.stacks,
#             'partition': partition
#         }
#         logging.info(
#             f'Loaded {partition} d
#             f' where the total num

#     def len(self):
#         return self.num_samples

#     def get(self, idx):
#         data_point = self.data[idx
#         x = data_point['new_x']
#         x = torch.tensor(x, dtype=
#         y = torch.tensor(data_poin
#         return x, y

#     @property
#     def raw_file_names(self):
#         file_names = [i for i in r
#         return [f'{self.raw_data_p

#     def _process(self):
#         data_list = []
#         for fn in self.raw_file_na
#             logging.info(f'Loading
#             with open(fn, 'rb') as
#                 data_slice = pickl
#             data_list = data_list
#         num_samples = len(data_lis
#         logging.info(f'Loaded {num

#    # Transform keypoints base
#    data_list = self.transform

#    # Stack and pad frames
#    data_list = self.stack_and

#    # Apply DBSCAN clustering
#    data_list = self.apply_dbs

#    # Random shuffle train and
#    random.seed(self.seed)
#    random.shuffle(data_list)

#    # Get partitions
#    train_end = int(self.parti
#    val_end = train_end + int(
#    train_data = data_list[:tr
#    val_data = data_list[train
#    test_data = data_list[val_

#    data_map = {
#      'train': train_data,
#      'val': val_data,
#      'test': test_data,
#    }
#    return data_map, num_sampl

#  def stack_and_padd_frames(self
#    if self.stacks is None:
#      return data_list
#    xs = [d['x'] for d in data
#    padded_xs = []
#    print("Stacking and paddin
#    pbar = tqdm(total=len(xs))

#    if self.zero_padding in ['
#      for i in range(len(xs)
#        data_point = []
#        for j in range(sel
#          if i - j >= 0:
#            mydata_sli
#            diff = sel
#            mydata_sli
#            if mydata_
#              idx =
#              mydata
#            data_point
#          else:
#            data_point
#        padded_xs.append(m
#        pbar.update(1)
#         elif self.zero_padding in
#             stacked_xs = []
#             for i in range(len(xs
#                 start = max(0, i
#                 stacked_xs.append(
#                 pbar.update(0.5)
#             for x in stacked_xs:
#                 diff = self.max_p
#                 x = np.pad(x, ((0,
#                 if x.shape[0] > se
#                     idx = np.rand
#                     x = x[idx]
#                 padded_xs.append(
#                 pbar.update(0.5)
#         else:
#             raise NotImplementedE
#         pbar.close()
#         print("Stacking and paddi
#         new_data_list = [{**d, 'ne
#         return new_data_list

#     def apply_dbscan(self, data_l
#         print("Applying DBSCAN cl
#         desired_num_points = self.
#         for data in tqdm(data_lis
#             x = data['new_x']  #
#             data['new_x_before_db
#             clustering = DBSCAN(e
#             labels = clustering.l
#             mask = labels != -1
#             x_filtered = x[mask]
#             num_points = x_filter
#             if num_points == 0:
#                 x_filtered = np.z
#             elif num_points < des
#                 diff = desired_nu
#                 x_filtered = np.p
#             elif num_points > des
#                 idx = np.random.c
#                 x_filtered = x_fi
#             data['new_x'] = x_fil
#         print("DBSCAN clustering
#         return data_list

#     kp18_names = ['NOSE', 'NECK',
#                   'RIGHT_WRIST',
#                   'LEFT_WRIST', '
#                   'RIGHT_ANKLE',
#                   'LEFT_ANKLE', '
#                   'RIGHT_EAR', 'L
#     kp9_names = ['RIGHT_SHOULDER',
#                  'LEFT_SHOULDER',
#                  'RIGHT_HIP', 'RIG
#                  'LEFT_HIP', 'LEFT
#     head_names = ['NOSE', 'RIGHT_E

#     def transform_keypoints(self,
#         if self.num_keypoints == 1
#             return data_list

#         print("Transforming keypoi
#         self.kp9_idx = [self.kp18_
#         self.head_idx = [self.kp18
#         for data in tqdm(data_list
#             kpts = data['y']
#             kpts_new = kpts[self.k
#             head = np.mean(kpts[se
#             kpts_new = np.concaten
#             assert kpts_new.shape
#             data['y'] = kpts_new
#         print("Transforming keypoi
#         return data_list

# class MMRActionData(MMRKeypointDat
#     processed_data = '/content/dri

#     def __init__(self, *args, **kw
#         self.action_label = np.loa
#         super().__init__(*args, **
#         self.info['num_classes'] =
#         self.target_dtype = torch.

#     def _process(self):
#         data_list = []
#         for fn in self.raw_file_na
#             logging.info(f'Loading
#             with open(fn, 'rb') as
#                 data_slice = pickl
#             data_list = data_list

#         for i, data in enumerate(d
#             data['y'] = self.actio
#         data_list = [d for d in da

#         data_list = self.stack_and
#         data_list = self.apply_dbs

#         num_samples = len(data_lis
#         logging.info(f'Loaded {num

#         train_end = int(self.parti
#         val_end = train_end + int(
#         train_data = data_list[:t
#         val_data = data_list[trai
#         test_data = data_list[val_

#         random.seed(self.seed)
#         random.shuffle(train_data
#         random.shuffle(val_data)

#         data_map = {
#             'train': train_data,
#             'val': val_data,
#             'test': test_data,
#         }
#         return data_map, num_samp

#     # Cluster analysis method
#     def cluster_analysis(self):
#         labels = [data['new_x'] f
#         total_points = len(labels
#         unique_clusters = np.uniq
#         num_clusters = len(unique_

#         print('Total:', total_poi
#         for i in range(num_cluste
#             print('Cluster', i, '
#         print('Noise:', np.sum(la

# # Testing the MMRActionData class
# if __name__ == "__main__":
#     # Define root directory and c
#     root_dir = ''
#     mmr_dataset_config = {
#         'processed_data': '/conte
#         'stacks': 5,
#         'max_points': 22,
#         'num_keypoints': 9,
#         'zero_padding': 'per_data_
#         'seed': 42,
#         'forced_rewrite': True
#     }
# if __name__ == '__main__':
#     X,y = make_moons(100)
#     model = DBSCAN()
#     preds = model.fit_predict(X)
#     # Either low or high values a
#     print(f"Accuracy: {round((sum


#     # Load train, validation, and
#     train_dataset = MMRActionData
#     val_dataset = MMRActionData(r
#     test_dataset = MMRActionData(r

#     # Print out the shapes of the
#     print(f"Train data shape: {len
#     print(f"Validation data shape:
#     print(f"Test data shape: {len(

#     # Perform cluster analysis aft
#     train_dataset.cluster_analysis
#     val_dataset.cluster_analysis()
#     test_dataset.cluster_analysis(

#     # Visualization of 5 random se
#     combined_data = train_dataset.
#     selected_sequences = random.sa

#     for idx, data_point in enumera
#         x_before = data_point['new
#         x_after = data_point['new_

#         fig = plt.figure(figsize=(

#         # Plot before DBSCAN
#         ax1 = fig.add_subplot(121,
#         ax1.scatter(x_before[:, 0]
#         ax1.set_title(f'Sequence {
#         ax1.set_xlabel('X')
#         ax1.set_ylabel('Y')
#         ax1.set_zlabel('Z')
#         ax1.view_init(elev=20., az

#         # Plot after DBSCAN
#         ax2 = fig.add_subplot(122,
#         ax2.scatter(x_after[:, 0],
#         ax2.set_title(f'Sequence {
#         ax2.set_xlabel('X')
#         ax2.set_ylabel('Y')
#         ax2.set_zlabel('Z')
#         ax2.view_init(elev=20., az

#         plt.tight_layout()
#         plt.show()

```

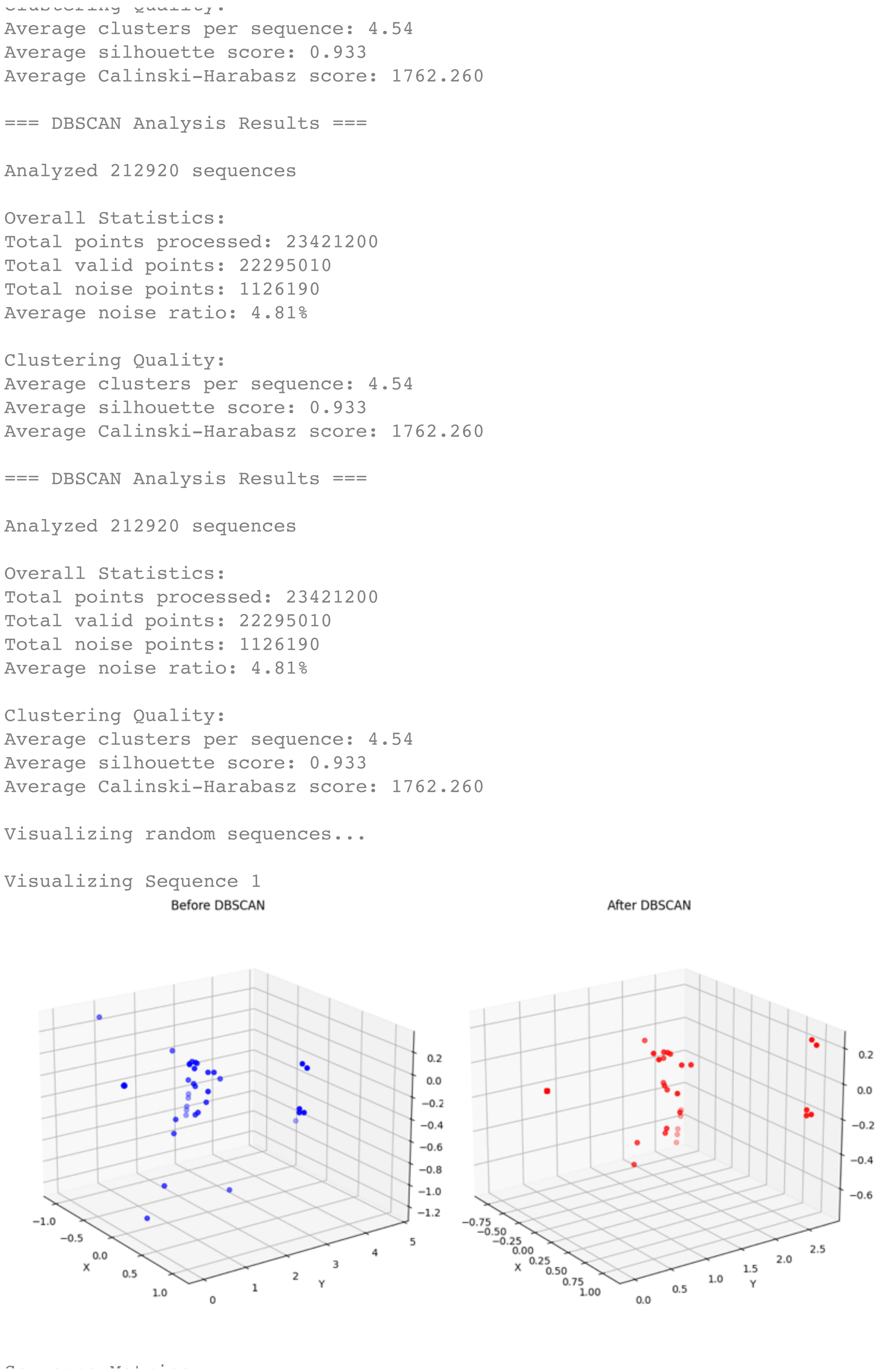
```
import os
import torch
import numpy as np
import pickle
import logging
import random
```

```
from tqdm import tqdm
from torch_geometric.data import Dataset
from sklearn.cluster import DBSCAN
from sklearn.metrics import silhouette_score, calinski_harabasz_score
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import seaborn as sns
import pandas as pd

class MMRActionData(Dataset):
    raw_data_path = '/content/drive/MyDrive/action/data/raw'
    processed_data = '/content/drive/MyDrive/action/data/processed/mmr_actic
    max_points = 22
    seed = 42
    partitions = (0.8, 0.1, 0.1)
    stacks = None
    zero_padding = 'per_data_point'
    zero_padding_styles = ['per_data_point', 'per_stack', 'data_point', 'sta
    num_keypoints = 9
    forced_rewrite = False

    def _parse_config(self, c):
        c = {k: v for k, v in c.items() if v is not None}
        self.seed = c.get('seed', self.seed)
        self.processed_data = c.get('processed_data', self.processed_data)
        self.max_points = c.get('max_points', self.max_points)
        self.partitions = (
            c.get('train_split', self.partitions[0]),
            c.get('val_split', self.partitions[1]),
            c.get('test_split', self.partitions[2]))
        self.stacks = c.get('stacks', self.stacks)
        self.zero_padding = c.get('zero_padding', self.zero_padding)
        self.num_keypoints = c.get('num_keypoints', self.num_keypoints)
        if self.zero_padding not in self.zero_padding_styles:
            raise ValueError(f'Zero padding style {self.zero_padding} not su
        self.forced_rewrite = c.get('forced_rewrite', self.forced_rewrite)

    def __init__(self, root, partition, mmr_dataset_config=None,
                 transform=None, pre_transform=None, pre_filter=None):
        self.partition = partition
        self.metrics = {}  # Store DBSCAN metrics

        # Load action labels before super().__init__
        try:
            self.action_label = np.load(f'{self.raw_data_path}/action_label.
        except FileNotFoundError:
            print(f"Warning: Could not find action_label.npy in {self.raw_da
            self.action_label = None

        if mmr_dataset_config is not None:
            self._parse_config(mmr_dataset_config)

        super(MMRActionData, self).__init__(root, transform, pre_transform,

        if (not os.path.isfile(self.processed_data)) or self.forced_rewrite:
            self.data, _ = self._process()
            os.makedirs(os.path.dirname(self.processed_data), exist_ok=True)
            with open(self.processed_data, 'wb') as f:
                pickle.dump(self.data, f)
        else:
            with open(self.processed_data, 'rb') as f:
                self.data = pickle.load(f)

        total_samples = len(self.data['train']) + len(self.data['val']) + le
        self.data = self.data[partition]
        self.num_samples = len(self.data)
        self.target_dtype = torch.int64

        self.info = {
            'num_samples': self.num_samples,
            'num_keypoints': self.num_keypoints,
            'num_classes': len(np.unique(self.action_label)) - 1 if self.act
            'max_points': self.max_points,
            'stacks': self.stacks,
            'partition': partition,
        }

        logging.info(
            f'Loaded {partition} data with {self.num_samples} samples,'
            f' where the total number of samples is {total_samples}')

    @property
    def raw_file_names(self):
        file_names = [i for i in range(19)]
        return [f'{self.raw_data_path}/{i}.pkl' for i in file_names]

    @property
    def processed_file_names(self):
        return [os.path.basename(self.processed_data)]

    def process(self):
        pass

    def len(self):
        return self.num_samples

    def get(self, idx):
        data_point = self.data[idx]
        x = data_point['new_x']
        x = torch.tensor(x, dtype=torch.float32)
        y = torch.tensor(data_point['y'], dtype=self.target_dtype)
        return x, y
```

```python

    # [Previous code remains the same until _process method]

    def _process(self):
        data_list = []
        for fn in self.raw_file_names:
            logging.info(f'Loading {fn}')
            try:
                with open(fn, 'rb') as f:
                    data_slice = pickle.load(f)
                data_list = data_list + data_slice
            except FileNotFoundError:
                print(f"Warning: Could not find {fn}")
                continue

        num_samples = len(data_list)
        logging.info(f'Loaded {num_samples} data points')

        # First transform keypoints
        data_list = self.transform_keypoints(data_list)

        # Then assign action labels
        if self.action_label is not None:
            for i, data in enumerate(data_list):
                if i < len(self.action_label):
                    data['y'] = self.action_label[i]
                else:
                    print(f"Warning: No action label for data point {i}")
                    data['y'] = -1
            data_list = [d for d in data_list if d['y'] != -1]

        # Stack and pad frames
        data_list = self.stack_and_padd_frames(data_list)

        # Apply DBSCAN clustering
        data_list = self.apply_dbscan(data_list)

        num_samples = len(data_list)
        logging.info(f'Processed {num_samples} data points')

        # Get partitions
        train_end = int(self.partitions[0] * num_samples)
        val_end = train_end + int(self.partitions[1] * num_samples)

        random.seed(self.seed)
        random.shuffle(data_list)

        train_data = data_list[:train_end]
        val_data = data_list[train_end:val_end]
        test_data = data_list[val_end:]

        data_map = {
            'train': train_data,
            'val': val_data,
            'test': test_data,
        }
        return data_map, num_samples

    def transform_keypoints(self, data_list):
        if self.num_keypoints == 18:
            return data_list

        print("Transforming keypoints ...")
        self.kp9_idx = [self.kp18_names.index(n) for n in self.kp9_names[:-1
        self.head_idx = [self.kp18_names.index(n) for n in self.head_names]

        transformed_list = []
        for data in tqdm(data_list, total=len(data_list)):
            try:
                if isinstance(data['y'], (np.ndarray, list)) and len(data['y
                    kpts = np.array(data['y'])
                    kpts_new = kpts[self.kp9_idx]
                    head = np.mean(kpts[self.head_idx], axis=0)
                    kpts_new = np.concatenate((kpts_new, head[None]))

                    if kpts_new.shape == (9, 3):  # Verify correct shape
                        data['y'] = kpts_new
                        transformed_list.append(data)
                    else:
                        print(f"Warning: Skipping data point with incorrect
                else:
                    print(f"Warning: Skipping data point with invalid keypoi
            except Exception as e:
                print(f"Warning: Error transforming keypoints: {str(e)}")
                continue

        print(f"Transformed {len(transformed_list)} keypoints out of {len(da
        return transformed_list

# [Rest of the code remains the same]
    def stack_and_padd_frames(self, data_list):
        if self.stacks is None:
            return data_list
        xs = [d['x'] for d in data_list]
        padded_xs = []
        print("Stacking and padding frames...")
        pbar = tqdm(total=len(xs))

        if self.zero_padding in ['per_data_point', 'data_point']:
            for i in range(len(xs)):
                data_point = []
                for j in range(self.stacks):
                    if i - j >= 0:
                        mydata_slice = xs[i - j]
                        diff = self.max_points - mydata_slice.shape[0]
                        mydata_slice = np.pad(mydata_slice, ((0, max(diff, 0
                        if mydata_slice.shape[0] > self.max_points:
                            idx = np.random.choice(mydata_slice.shape[0], se
                            mydata_slice = mydata_slice[idx]
                        data_point.append(mydata_slice)
                    else:
                        data_point.append(np.zeros((self.max_points, 3)))
                padded_xs.append(np.concatenate(data_point, axis=0))
                pbar.update(1)
        elif self.zero_padding in ['per_stack', 'stack']:
            stacked_xs = []
            for i in range(len(xs)):
                start = max(0, i - self.stacks + 1)
                stacked_xs.append(np.concatenate(xs[start:i+1], axis=0))
                pbar.update(0.5)
            for x in stacked_xs:
                diff = self.max_points * self.stacks - x.shape[0]
                x = np.pad(x, ((0, max(diff, 0)), (0, 0)), 'constant')
                if x.shape[0] > self.max_points * self.stacks:
                    idx = np.random.choice(x.shape[0], self.max_points * sel
                    x = x[idx]
                padded_xs.append(x)
                pbar.update(0.5)
        else:
            raise NotImplementedError()

        pbar.close()
        print("Stacking and padding frames done")
        new_data_list = [{**d, 'new_x': x} for d, x in zip(data_list, padded
        return new_data_list

    def apply_dbscan(self, data_list):
        print("Applying DBSCAN clustering...")
        desired_num_points = self.max_points * (self.stacks if self.stacks e
        all_metrics = []

        for data in tqdm(data_list, total=len(data_list)):
            x = data['new_x']
            data['new_x_before_dbscan'] = x.copy()

            clustering = DBSCAN(eps=0.5, min_samples=3)
            labels = clustering.fit_predict(x)

            # Calculate metrics
            sequence_metrics = self._calculate_sequence_metrics(x, labels)
            all_metrics.append(sequence_metrics)

            # Filter points
            mask = labels != -1
            x_filtered = x[mask]
            num_points = x_filtered.shape[0]

            if num_points == 0:
                x_filtered = np.zeros((desired_num_points, x.shape[1]))
            elif num_points < desired_num_points:
                diff = desired_num_points - num_points
                x_filtered = np.pad(x_filtered, ((0, diff), (0, 0)), 'consta
            elif num_points > desired_num_points:
                idx = np.random.choice(num_points, desired_num_points, repla
                x_filtered = x_filtered[idx]

            data['new_x'] = x_filtered
            data['dbscan_labels'] = labels
            data['metrics'] = sequence_metrics

        self.metrics = self._calculate_overall_metrics(all_metrics)
        print("DBSCAN clustering applied.")
        return data_list

    def _calculate_sequence_metrics(self, x, labels):
        n_clusters = len(set(labels)) - (1 if -1 in labels else 0)
        n_noise = list(labels).count(-1)

        valid_points = labels != -1
        if sum(valid_points) > 1 and len(set(labels[valid_points])) > 1:
            try:
                silhouette = silhouette_score(x[valid_points], labels[valid_
                calinski = calinski_harabasz_score(x[valid_points], labels[v
            except:
                silhouette = calinski = 0
        else:
            silhouette = calinski = 0

        return {
            'n_clusters': n_clusters,
            'n_noise': n_noise,
            'noise_ratio': n_noise / len(x),
            'silhouette_score': silhouette,
            'calinski_score': calinski,
            'total_points': len(x),
            'valid_points': sum(valid_points)
        }

    def _calculate_overall_metrics(self, all_metrics):
        overall = {}
        for key in all_metrics[0].keys():
            if key in ['n_clusters', 'n_noise', 'total_points', 'valid_point
                overall[key] = sum(m[key] for m in all_metrics)
            else:
                overall[key] = np.mean([m[key] for m in all_metrics])

        overall['sequences_analyzed'] = len(all_metrics)
        return overall

    kp18_names = ['NOSE', 'NECK', 'RIGHT_SHOULDER', 'RIGHT_ELBOW',
                  'RIGHT_WRIST', 'LEFT_SHOULDER', 'LEFT_ELBOW',
                  'LEFT_WRIST', 'RIGHT_HIP', 'RIGHT_KNEE',
                  'RIGHT_ANKLE', 'LEFT_HIP', 'LEFT_KNEE',
                  'LEFT_ANKLE', 'RIGHT_EYE', 'LEFT_EYE',
                  'RIGHT_EAR', 'LEFT_EAR']
    kp9_names = ['RIGHT_SHOULDER', 'RIGHT_ELBOW',
                 'LEFT_SHOULDER', 'LEFT_ELBOW',
                 'RIGHT_HIP', 'RIGHT_KNEE',
                 'LEFT_HIP', 'LEFT_KNEE', 'HEAD']
    head_names = ['NOSE', 'RIGHT_EYE', 'LEFT_EYE', 'RIGHT_EAR', 'LEFT_EAR']

    def transform_keypoints(self, data_list):
        if self.num_keypoints == 18:
            return data_list

        print("Transforming keypoints ...")
        self.kp9_idx = [self.kp18_names.index(n) for n in self.kp9_names[:-1
        self.head_idx = [self.kp18_names.index(n) for n in self.head_names]

        for data in tqdm(data_list, total=len(data_list)):
            kpts = data['y']
            kpts_new = kpts[self.kp9_idx]
            head = np.mean(kpts[self.head_idx], axis=0)
            kpts_new = np.concatenate((kpts_new, head[None]))
            assert kpts_new.shape == (9, 3)
            data['y'] = kpts_new

        print("Transforming keypoints done")
        return data_list

    def cluster_analysis(self):
        if not hasattr(self, 'metrics') or not self.metrics:
            print("No clustering metrics available. Run DBSCAN first.")
            return

        print("\n=== DBSCAN Analysis Results ===")
        print(f"\nAnalyzed {self.metrics['sequences_analyzed']} sequences")

        print("\nOverall Statistics:")
        print(f"Total points processed: {self.metrics['total_points']}")
        print(f"Total valid points: {self.metrics['valid_points']}")
        print(f"Total noise points: {self.metrics['n_noise']}")
        print(f"Average noise ratio: {self.metrics['noise_ratio']:.2%}")

        print("\nClustering Quality:")
        print(f"Average clusters per sequence: {self.metrics['n_clusters']/s
        print(f"Average silhouette score: {self.metrics['silhouette_score']:
        print(f"Average Calinski-Harabasz score: {self.metrics['calinski_sco

    def visualize_sequence(self, data_point):
        x_before = data_point['new_x_before_dbscan']
        x_after = data_point['new_x']

        fig = plt.figure(figsize=(12, 6))

        # Plot before DBSCAN
        ax1 = fig.add_subplot(121, projection='3d')
        ax1.scatter(x_before[:, 0], x_before[:, 1], x_before[:, 2], c='b', m
        ax1.set_title('Before DBSCAN')
        ax1.set_xlabel('X')
        ax1.set_ylabel('Y')
        ax1.set_zlabel('Z')
        ax1.view_init(elev=20., azim=-35)

        # Plot after DBSCAN
        ax2 = fig.add_subplot(122, projection='3d')
        ax2.scatter(x_after[:, 0], x_after[:, 1], x_after[:, 2], c='r', mark
        ax2.set_title('After DBSCAN')
        ax2.set_xlabel('X')
        ax2.set_ylabel('Y')
        ax2.set_zlabel('Z')
        ax2.view_init(elev=20., azim=-35)

        plt.tight_layout()
        plt.show()

        # Print sequence metrics if available
        if 'metrics' in data_point:
            print("\nSequence Metrics:")
            for k, v in data_point['metrics'].items():
                print(f"{k}: {v:.3f}" if isinstance(v, float) else f"{k}: {v

if __name__ == "__main__":
    # Define root directory and configuration
    root_dir = ''
    mmr_dataset_config = {
        'processed_data': '/content/drive/MyDrive/action/data/processed/mmr_
        'stacks': 5,
        'max_points': 22,
        'num_keypoints': 9,
        'zero_padding': 'per_data_point',
        'seed': 42,
        'forced_rewrite': True
    }

    try:
        # Create processed directory if it doesn't exist
        os.makedirs(os.path.dirname(mmr_dataset_config['processed_data']), e

        print("Loading datasets...")
        # Load datasets
        train_dataset = MMRActionData(root=root_dir, partition='train',
                                    mmr_dataset_config=mmr_dataset_config)
        val_dataset = MMRActionData(root=root_dir, partition='val',
                                  mmr_dataset_config=mmr_dataset_config)
        test_dataset = MMRActionData(root=root_dir, partition='test',
                                   mmr_dataset_config=mmr_dataset_config)

        # Print dataset sizes
        print(f"\nDataset sizes:")
        print(f"Train data: {len(train_dataset)} samples")
        print(f"Validation data: {len(val_dataset)} samples")
        print(f"Test data: {len(test_dataset)} samples")

        # Perform cluster analysis
        print("\nPerforming cluster analysis...")
        train_dataset.cluster_analysis()
        val_dataset.cluster_analysis()
        test_dataset.cluster_analysis()

        # Visualize random sequences
        print("\nVisualizing random sequences...")
        combined_data = train_dataset.data + val_dataset.data + test_dataset
        selected_sequences = random.sample(combined_data, 5)

        for idx, data_point in enumerate(selected_sequences):
            print(f"\nVisualizing Sequence {idx+1}")
            train_dataset.visualize_sequence(data_point)

    except Exception as e:
        print(f"An error occurred: {str(e)}")
        raise
```

```
Loading datasets...
Transforming keypoints ...
100%|██████████| 545059/545059 [00:21<00:00, 25633.52it/s]
Transforming keypoints done
Stacking and padding frames...
100%|██████████| 212920/212920 [00:50<00:00, 4251.16it/s]
Stacking and padding frames done
Applying DBSCAN clustering...
100%|██████████| 212920/212920 [12:37<00:00, 280.91it/s]
DBSCAN clustering applied.
Transforming keypoints ...
100%|██████████| 545059/545059 [00:13<00:00, 39402.62it/s]
Transforming keypoints done
Stacking and padding frames...
100%|██████████| 212920/212920 [00:49<00:00, 4260.10it/s]
Stacking and padding frames done
```

```
Applying DBSCAN clustering...
100%|██████████| 212920/212920 [12:32<00:00, 282.83it/s]
DBSCAN clustering applied.
Transforming keypoints ...
100%|██████████| 545059/545059 [00:22<00:00, 24101.17it/s]
Transforming keypoints done
Stacking and padding frames...
100%|██████████| 212920/212920 [00:50<00:00, 4249.32it/s]
Stacking and padding frames done
Applying DBSCAN clustering...
100%|██████████| 212920/212920 [12:39<00:00, 280.52it/s]
DBSCAN clustering applied.
Transforming keypoints ...
100%|██████████| 545059/545059 [00:14<00:00, 38779.07it/s]
Transforming keypoints done
Stacking and padding frames...
100%|██████████| 212920/212920 [00:51<00:00, 4171.42it/s]
Stacking and padding frames done
Applying DBSCAN clustering...
100%|██████████| 212920/212920 [12:37<00:00, 280.94it/s]
DBSCAN clustering applied.
Transforming keypoints ...
100%|██████████| 545059/545059 [00:23<00:00, 23145.42it/s]
Transforming keypoints done
Stacking and padding frames...
100%|██████████| 212920/212920 [00:50<00:00, 4250.64it/s]
Stacking and padding frames done
Applying DBSCAN clustering...
100%|██████████| 212920/212920 [12:56<00:00, 274.11it/s]
DBSCAN clustering applied.
Transforming keypoints ...
100%|██████████| 545059/545059 [00:14<00:00, 38660.96it/s]
Transforming keypoints done
Stacking and padding frames...
100%|██████████| 212920/212920 [00:50<00:00, 4202.22it/s]
Stacking and padding frames done
Applying DBSCAN clustering...
100%|██████████| 212920/212920 [13:07<00:00, 270.54it/s]
DBSCAN clustering applied.

Dataset sizes:
Train data: 170336 samples
Validation data: 21292 samples
Test data: 21292 samples

Performing cluster analysis...

=== DBSCAN Analysis Results ===

Analyzed 212920 sequences

Overall Statistics:
Total points processed: 23421200
Total valid points: 22295010
Total noise points: 1126190
Average noise ratio: 4.81%

Clustering Quality:
```



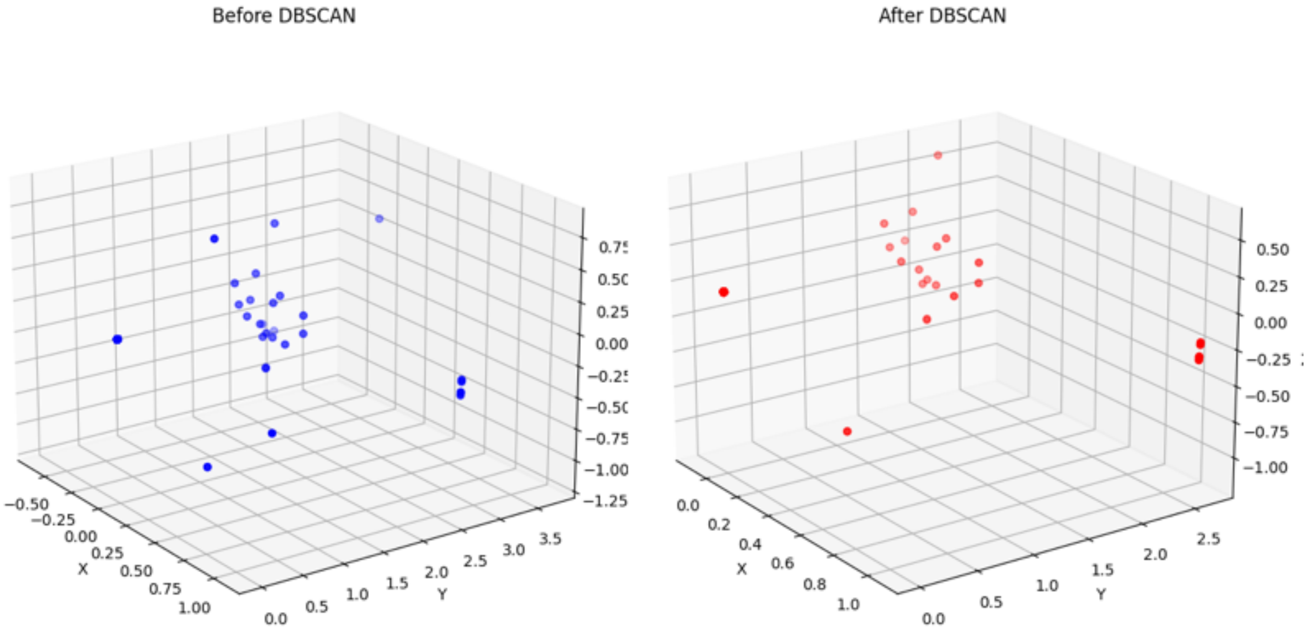
```
Clustering Quality:
Average clusters per sequence: 4.54
Average silhouette score: 0.933
Average Calinski-Harabasz score: 1762.260

=== DBSCAN Analysis Results ===

Analyzed 212920 sequences

Overall Statistics:
Total points processed: 23421200
Total valid points: 22295010
Total noise points: 1126190
Average noise ratio: 4.81%

Clustering Quality:
Average clusters per sequence: 4.54
Average silhouette score: 0.933
Average Calinski-Harabasz score: 1762.260

=== DBSCAN Analysis Results ===

Analyzed 212920 sequences

Overall Statistics:
Total points processed: 23421200
Total valid points: 22295010
Total noise points: 1126190
Average noise ratio: 4.81%

Clustering Quality:
Average clusters per sequence: 4.54
Average silhouette score: 0.933
Average Calinski-Harabasz score: 1762.260

Visualizing random sequences...

Visualizing Sequence 1
```

Before DBSCAN

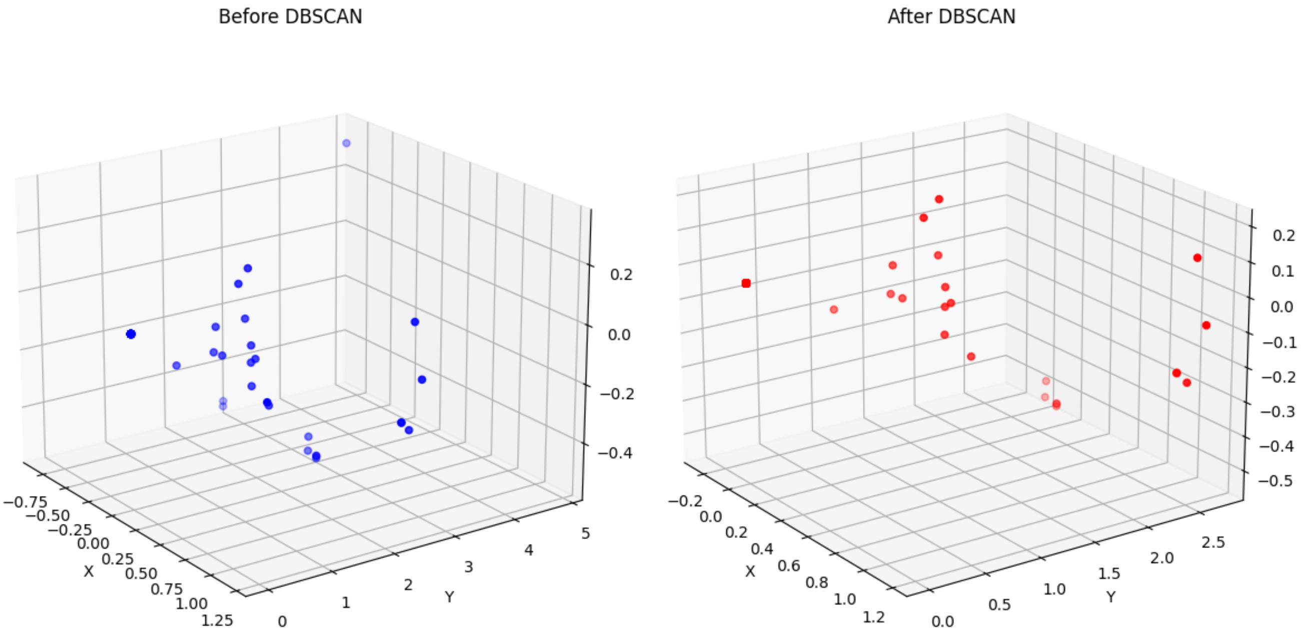
After DBSCAN

```
Sequence Metrics:
```

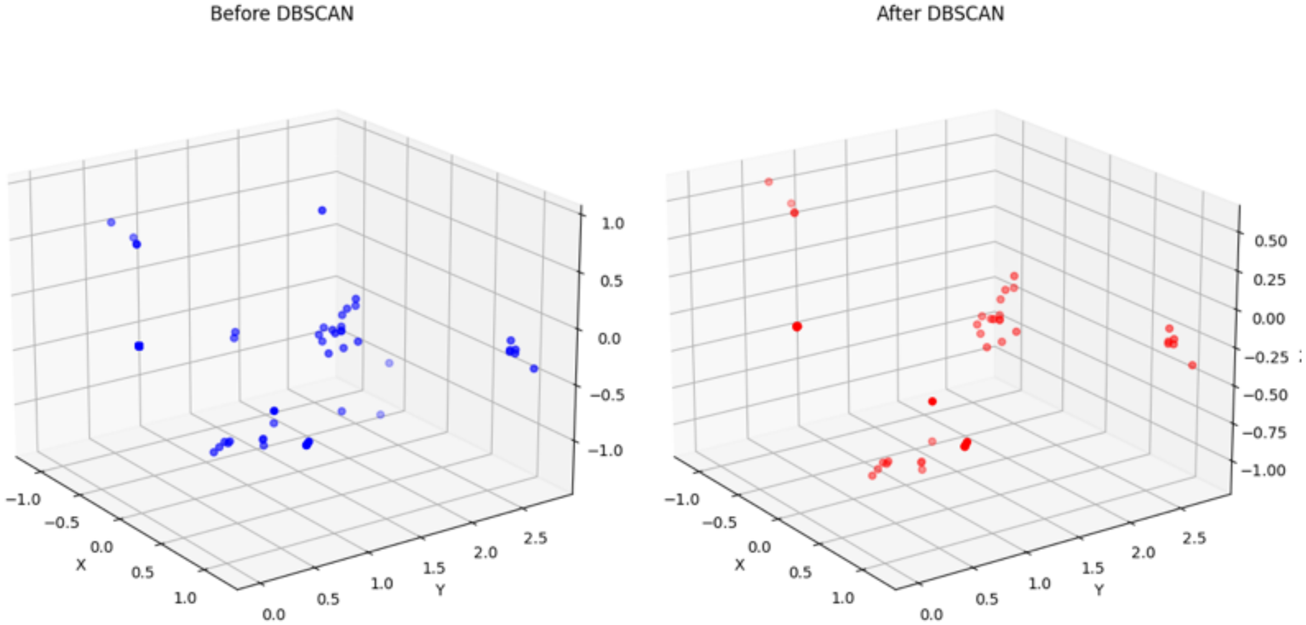
```
Sequence Metrics:
n_clusters: 4
n_noise: 6
noise_ratio: 0.055
silhouette_score: 0.931
calinski_score: 1546.800
total_points: 110
valid_points: 104

Visualizing Sequence 2
```



```
Sequence Metrics:
n_clusters: 4
n_noise: 5
noise_ratio: 0.045
silhouette_score: 0.947
calinski_score: 1729.891
total_points: 110
valid_points: 105

Visualizing Sequence 3
```

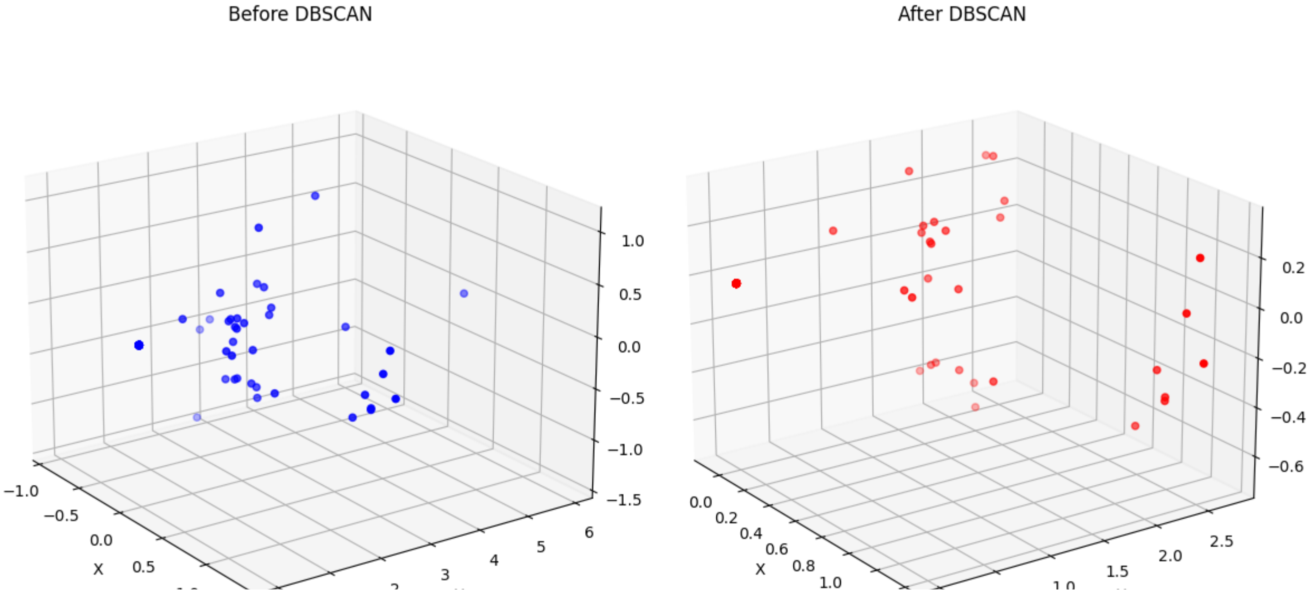


```
Sequence Metrics:
n_clusters: 4
n_noise: 4
noise_ratio: 0.036
silhouette_score: 0.957
calinski_score: 3174.033
total_points: 110
valid_points: 106

Visualizing Sequence 4
```

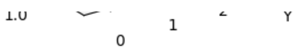

Before DBSCAN

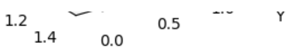
After DBSCAN

```
Sequence Metrics:
n_clusters: 6
n_noise: 6
noise_ratio: 0.055
silhouette_score: 0.935
calinski_score: 1811.647
total_points: 110
valid_points: 104

Visualizing Sequence 5
```


Before DBSCAN

After DBSCAN





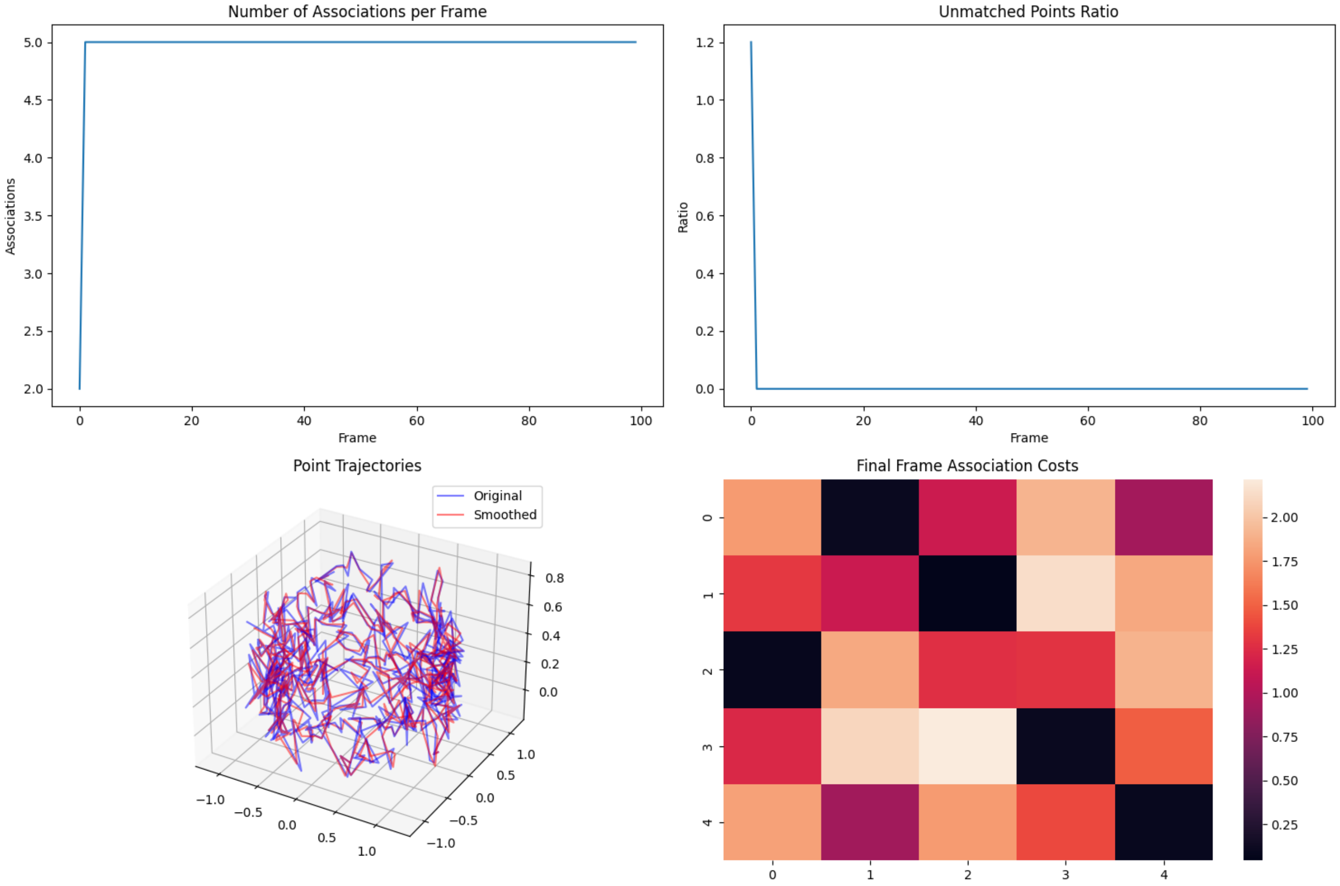
```
Sequence Metrics:
n_clusters: 4
n_noise: 7
noise_ratio: 0.064
silhouette_score: 0.855
calinski_score: 795.872
total_points: 110
valid_points: 103
```

```
!pip install filterpy
```

```
Collecting filterpy
  Downloading filterpy-1.4.5.zip (177 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 178.0/178.0 kB 5.4 MB/s eta 0
  Preparing metadata (setup.py) ... done
Requirement already satisfied: numpy in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: scipy in /usr/local/lib/python3.10/dist-pack
Requirement already satisfied: matplotlib in /usr/local/lib/python3.10/dist
Requirement already satisfied: contourpy>=1.0.1 in /usr/local/lib/python3.1
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/python3.10/di
Requirement already satisfied: fonttools>=4.22.0 in /usr/local/lib/python3.
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib/python3.
Requirement already satisfied: packaging>=20.0 in /usr/local/lib/python3.10
Requirement already satisfied: pillow>=6.2.0 in /usr/local/lib/python3.10/d
Requirement already satisfied: pyparsing>=2.3.1 in /usr/local/lib/python3.1
Requirement already satisfied: python-dateutil>=2.7 in /usr/local/lib/pytho
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.10/dist-p
Building wheels for collected packages: filterpy
  Building wheel for filterpy (setup.py) ... done
  Created wheel for filterpy: filename=filterpy-1.4.5-py3-none-any.whl size
  Stored in directory: /root/.cache/pip/wheels/0f/0c/ea/218f266af4ad6268975
Successfully built filterpy
Installing collected packages: filterpy
Successfully installed filterpy-1.4.5
```

```python
import numpy as np
from filterpy.kalman import KalmanFilter as FPKalmanFilter
import matplotlib.pyplot as plt
from scipy.optimize import linear_sum_assignment
from sklearn.metrics import silhouette_score, calinski_harabasz_score
from tqdm import tqdm
import seaborn as sns

class ComprehensiveKalmanFilter:
    def __init__(self, dim_z=3, dt=1.0, max_cost=10.0):
        """
        Initialize Kalman Filter with Hungarian algorithm

        Parameters:
        -----------
        dim_z : int
            Dimension of measurements (default 3 for x,y,z coordinates)
        dt : float
            Time step between measurements
        max_cost : float
            Maximum cost for point association
        """
        # Previous initialization code remains the same
        self.dim_z = dim_z
        dim_x = dim_z * 3

        self.s_hat = np.zeros(dim_x)
        self.P_hat = np.eye(dim_x) * 100
        self.max_cost = max_cost

        # State transition matrix
        self.F = np.zeros((dim_x, dim_x))
        for i in range(dim_z):
            idx = i * 3
            self.F[idx:idx+3, idx:idx+3] = np.array([
                [1, dt, 0.5*dt**2],
                [0, 1, dt],
                [0, 0, 1]
            ])

        # Measurement matrix
        self.H = np.zeros((dim_z, dim_x))
        for i in range(dim_z):
            self.H[i, i*3] = 1

        # Noise matrices
        self.Q = np.eye(dim_x) * 0.1
        self.Q[0::3, 0::3] *= 0.1
        self.Q[1::3, 1::3] *= 0.2
        self.Q[2::3, 2::3] *= 0.3

        self.R = np.eye(dim_z) * 0.1

        # Analysis storage
        self.K_gain = []
        self.innovation = []
        self.predictions = []
        self.corrections = []
        self.associations = []

    def compute_association_cost(self, predictions, measurements):
        """
        Compute cost matrix for Hungarian algorithm

        Parameters:
        -----------
        predictions : array-like
            Predicted positions (n_points, dim_z)
        measurements : array-like
            Measured positions (n_measurements, dim_z)

        Returns:
        --------
        cost_matrix : array-like
            Matrix of association costs
        """
        n_pred = len(predictions)
        n_meas = len(measurements)

        cost_matrix = np.zeros((n_pred, n_meas))

        for i in range(n_pred):
            for j in range(n_meas):
                cost_matrix[i, j] = np.linalg.norm(predictions[i] - measurem

        return cost_matrix

    def associate_points(self, predictions, measurements):
        """
        Associate predictions with measurements using Hungarian algorithm

        Parameters:
        -----------
        predictions : array-like
            Predicted positions
        measurements : array-like
            Measured positions

        Returns:
        --------
        associations : list of tuples
            List of (prediction_idx, measurement_idx) pairs
        unmatched_predictions : list
            Indices of unmatched predictions
        unmatched_measurements : list
            Indices of unmatched measurements
        """
        cost_matrix = self.compute_association_cost(predictions, measurement

        # Apply Hungarian algorithm
        pred_idx, meas_idx = linear_sum_assignment(cost_matrix)

        # Filter associations based on maximum cost
        valid_associations = []
        unmatched_predictions = set(range(len(predictions)))
        unmatched_measurements = set(range(len(measurements)))

        for p, m in zip(pred_idx, meas_idx):
            if cost_matrix[p, m] <= self.max_cost:
                valid_associations.append((p, m))
                unmatched_predictions.remove(p)
                unmatched_measurements.remove(m)

        return valid_associations, list(unmatched_predictions), list(unmatch

    def smooth_sequence_with_association(self, points):
        """
        Smooth sequence with point association
        """
        n_frames = len(points)
        n_points = points.shape[1] if len(points.shape) > 2 else 1
        smoothed = np.zeros_like(points)
        metrics = {
            'innovations': [],
            'kalman_gains': [],
            'uncertainties': [],
            'associations': [],
            'unmatched_ratio': []
        }

        # Initialize states for all points
        states = [np.zeros(self.dim_z * 3) for _ in range(n_points)]
        covs = [np.eye(self.dim_z * 3) * 100 for _ in range(n_points)]

        for i in range(n_frames):
            current_points = points[i]
            predicted_points = np.array([state[0::3] for state in states])

            # Associate points
            associations, unmatched_pred, unmatched_meas = self.associate_po
                predicted_points, current_points)

            metrics['associations'].append(len(associations))
            metrics['unmatched_ratio'].append(
                (len(unmatched_pred) + len(unmatched_meas)) / n_points)

            # Update matched points
            for pred_idx, meas_idx in associations:
                # Prediction
                states[pred_idx] = self.F @ states[pred_idx]
                covs[pred_idx] = self.F @ covs[pred_idx] @ self.F.T + self.Q

                # Kalman gain
                K = covs[pred_idx] @ self.H.T @ np.linalg.inv(
                    self.H @ covs[pred_idx] @ self.H.T + self.R)

                # Update
                innovation = current_points[meas_idx] - self.H @ states[pred
                states[pred_idx] += K @ innovation
                covs[pred_idx] = (np.eye(len(states[pred_idx])) -
                                K @ self.H) @ covs[pred_idx]

                smoothed[i, pred_idx] = states[pred_idx][0::3]

            # Initialize new tracks for unmatched measurements
            for meas_idx in unmatched_meas:
                if len(unmatched_pred) > 0:
                    pred_idx = unmatched_pred.pop(0)
                    states[pred_idx][0::3] = current_points[meas_idx]
                    states[pred_idx][3:] = 0  # Reset velocity and accelerat
                    covs[pred_idx] = np.eye(self.dim_z * 3) * 100
                    smoothed[i, pred_idx] = current_points[meas_idx]

            metrics['innovations'].append(np.mean([np.linalg.norm(s[0::3] -
                                        current_points[j]) for j, s in enume
            metrics['kalman_gains'].append(np.mean([np.trace(c) for c in cov
            metrics['uncertainties'].append(np.mean([np.trace(c) for c in co

        return smoothed, metrics

    def analyze_association_performance(self, metrics):
        """
        Analyze point association performance
        """
        analysis = {
            'average_associations': np.mean(metrics['associations']),
            'association_stability': np.std(metrics['associations']),
            'average_unmatched_ratio': np.mean(metrics['unmatched_ratio']),
            'max_unmatched_ratio': np.max(metrics['unmatched_ratio'])
        }
        return analysis

    def visualize_associations(self, original, smoothed, metrics):
        """
        Visualize point associations
        """
        fig = plt.figure(figsize=(15, 10))

        # Association count
        ax1 = plt.subplot(221)
        ax1.plot(metrics['associations'])
        ax1.set_title('Number of Associations per Frame')
        ax1.set_xlabel('Frame')
        ax1.set_ylabel('Associations')

        # Unmatched ratio
        ax2 = plt.subplot(222)
        ax2.plot(metrics['unmatched_ratio'])
        ax2.set_title('Unmatched Points Ratio')
        ax2.set_xlabel('Frame')
        ax2.set_ylabel('Ratio')

        # Point trajectories
        ax3 = plt.subplot(223, projection='3d')
        for i in range(min(5, original.shape[1])):  # Plot first 5 points
            ax3.plot(original[:, i, 0], original[:, i, 1], original[:, i, 2]
                    'b-', alpha=0.5, label='Original' if i == 0 else '')
            ax3.plot(smoothed[:, i, 0], smoothed[:, i, 1], smoothed[:, i, 2]
                    'r-', alpha=0.5, label='Smoothed' if i == 0 else '')
        ax3.set_title('Point Trajectories')
        ax3.legend()

        # Association matrix heatmap
        ax4 = plt.subplot(224)
        cost_matrix = self.compute_association_cost(
            original[-1], smoothed[-1])
        sns.heatmap(cost_matrix, ax=ax4)
        ax4.set_title('Final Frame Association Costs')

        plt.tight_layout()
        plt.show()

# Example usage
if __name__ == "__main__":
    # Generate sample data with multiple points
    n_frames = 100
    n_points = 5
    t = np.linspace(0, 2*np.pi, n_frames)

    # Create trajectories for multiple points
    original = np.zeros((n_frames, n_points, 3))
    for i in range(n_points):
        original[:, i] = np.column_stack([
            np.cos(t + i*2*np.pi/n_points),
            np.sin(t + i*2*np.pi/n_points),
            0.1*t
        ]) + np.random.normal(0, 0.1, (n_frames, 3))

    # Initialize and apply filter
    kf = ComprehensiveKalmanFilter(dim_z=3, max_cost=1.0)
    smoothed, metrics = kf.smooth_sequence_with_association(original)

    # Analyze performance
    association_analysis = kf.analyze_association_performance(metrics)
    print("\nAssociation Analysis:")
    for k, v in association_analysis.items():
        print(f"{k}: {v:.6f}")

    # Visualize results
    kf.visualize_associations(original, smoothed, metrics)
```

```
Association Analysis:
average_associations: 4.970000
association_stability: 0.298496
average_unmatched_ratio: 0.012000
max_unmatched_ratio: 1.200000
```



```python
class MMRActionDataEnhanced(MMRActionData):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.kf = ComprehensiveKalmanFilter(dim_z=3, max_cost=1.0)
        self.enhanced_data = None

    def enhance_sequence(self, data_point):
        """Enhance a single sequence using DBSCAN results and Kalman filterin
        x_dbscan = data_point['new_x']  # Shape: [110, 3] (22 points * 5 stac

        # Reshape to sequence format: [frames, keypoints, dims]
        n_frames = self.stacks if self.stacks else 1
        n_points = self.max_points
        x_reshaped = x_dbscan.reshape(n_frames, n_points, 3)

        # Apply Kalman filtering with Hungarian association
        x_enhanced, metrics = self.kf.smooth_sequence_with_association(x_resh

        # Store original and enhanced data
        enhanced_point = {
            'original': data_point['new_x'].copy(),
            'dbscan': x_dbscan,
            'enhanced': x_enhanced.reshape(-1, 3),  # Reshape back to origina
            'kalman_metrics': metrics
        }

        return enhanced_point

    def enhance_dataset(self):
        """Enhance entire dataset"""
        print("Enhancing dataset with Kalman filtering...")
        enhanced_data = []

        for data_point in tqdm(self.data):
            enhanced_point = self.enhance_sequence(data_point)
            enhanced_data.append(enhanced_point)

        self.enhanced_data = enhanced_data
        return enhanced_data

    def analyze_enhancement(self):
        """Analyze enhancement results"""
        if self.enhanced_data is None:
            print("No enhanced data available. Run enhance_dataset first.")
            return

        metrics = {
            'dbscan_metrics': [],
            'kalman_metrics': [],
            'combined_metrics': []
        }

        for data in self.enhanced_data:
            # Calculate improvement metrics
            original = data['original']
            dbscan = data['dbscan']
            enhanced = data['enhanced']

            # DBSCAN improvement
            dbscan_diff = np.linalg.norm(dbscan - original, axis=1)

            # Kalman improvement
            kalman_diff = np.linalg.norm(enhanced - dbscan, axis=1)

            # Overall improvement
            total_diff = np.linalg.norm(enhanced - original, axis=1)

            metrics['dbscan_metrics'].append({
                'mean_improvement': np.mean(dbscan_diff),
                'max_improvement': np.max(dbscan_diff),
                'std_improvement': np.std(dbscan_diff)
            })

            metrics['kalman_metrics'].append({
                'mean_improvement': np.mean(kalman_diff),
                'max_improvement': np.max(kalman_diff),
                'std_improvement': np.std(kalman_diff),
                'association_stats': data['kalman_metrics']
            })

            metrics['combined_metrics'].append({
                'mean_improvement': np.mean(total_diff),
                'max_improvement': np.max(total_diff),
                'std_improvement': np.std(total_diff)
            })

        return metrics

    def visualize_enhancement(self, sequence_idx=0):
        """Visualize enhancement results for a sequence"""
        if self.enhanced_data is None or sequence_idx >= len(self.enhanced_da
            print("Invalid sequence index or no enhanced data available")
            return

        data = self.enhanced_data[sequence_idx]

        fig = plt.figure(figsize=(20, 10))

        # 3D trajectories
        ax1 = fig.add_subplot(231, projection='3d')
        ax1.scatter(data['original'][:, 0],
                   data['original'][:, 1],
                   data['original'][:, 2],
                   c='b', marker='o', label='Original', alpha=0.3)
        ax1.scatter(data['dbscan'][:, 0],
                   data['dbscan'][:, 1],
                   data['dbscan'][:, 2],
                   c='r', marker='o', label='DBSCAN', alpha=0.3)
        ax1.scatter(data['enhanced'][:, 0],
                   data['enhanced'][:, 1],
                   data['enhanced'][:, 2],
                   c='g', marker='o', label='Enhanced', alpha=0.3)
        ax1.set_title('3D Point Trajectories')
        ax1.legend()

        # Point associations
        ax2 = fig.add_subplot(232)
        associations = data['kalman_metrics']['associations']
        associations = data['kalman_metrics']['associations']
        ax2.plot(associations, label='Associations')
        ax2.set_title('Point Associations per Frame')
        ax2.set_xlabel('Frame')
        ax2.set_ylabel('Number of Associations')

        # Improvement metrics
        ax3 = fig.add_subplot(233)
        improvements = [
            np.linalg.norm(data['dbscan'] - data['original'], axis=1).mean(),
            np.linalg.norm(data['enhanced'] - data['dbscan'], axis=1).mean(),
            np.linalg.norm(data['enhanced'] - data['original'], axis=1).mean(
        ]
        ax3.bar(['DBSCAN', 'Kalman', 'Combined'], improvements)
        ax3.set_title('Average Improvements')

        # Velocity profiles
        ax4 = fig.add_subplot(234)
        vel_orig = np.linalg.norm(np.diff(data['original'].reshape(self.stack
        vel_enh = np.linalg.norm(np.diff(data['enhanced'].reshape(self.stacks
        ax4.plot(vel_orig.mean(axis=0), label='Original', alpha=0.5)
        ax4.plot(vel_enh.mean(axis=0), label='Enhanced', alpha=0.5)
        ax4.set_title('Average Velocity Profiles')
        ax4.legend()

        # Uncertainty evolution
        ax5 = fig.add_subplot(235)
        ax5.plot(data['kalman_metrics']['uncertainties'])
        ax5.set_title('State Uncertainty Evolution')
        ax5.set_xlabel('Frame')
        ax5.set_ylabel('Average Uncertainty')

        # Innovation history
        ax6 = fig.add_subplot(236)
        ax6.plot(data['kalman_metrics']['innovations'])
        ax6.set_title('Innovation History')
        ax6.set_xlabel('Frame')
        ax6.set_ylabel('Average Innovation')

        plt.tight_layout()
        plt.show()

# Example usage
if __name__ == "__main__":
    # Initialize dataset with enhancement capabilities
    root_dir = ''
    mmr_dataset_config = {
        'processed_data': '/content/drive/MyDrive/action/data/processed/mmr_a
        'stacks': 5,
        'max_points': 22,
        'num_keypoints': 9,
        'zero_padding': 'per_data_point',
        'seed': 42,
        'forced_rewrite': True
    }

    # Load enhanced dataset
    dataset = MMRActionDataEnhanced(root=root_dir, partition='train',
                                  mmr_dataset_config=mmr_dataset_config)

    # Apply enhancement pipeline
    print("Starting enhancement pipeline...")
    enhanced_data = dataset.enhance_dataset()

    # Analyze results
    print("\nAnalyzing enhancement results...")
    metrics = dataset.analyze_enhancement()

    # Print summary statistics
    print("\nEnhancement Summary:")
    print("\nDBSCAN Improvements:")
    dbscan_means = np.mean([m['mean_improvement'] for m in metrics['dbscan_me
    print(f"Average DBSCAN improvement: {dbscan_means:.3f}")

    print("\nKalman Improvements:")
    kalman_means = np.mean([m['mean_improvement'] for m in metrics['kalman_me
    print(f"Average Kalman improvement: {kalman_means:.3f}")

    print("\nCombined Improvements:")
    combined_means = np.mean([m['mean_improvement'] for m in metrics['combine
    print(f"Average total improvement: {combined_means:.3f}")

    # Visualize results for first few sequences
    print("\nVisualizing enhancement results...")
    for i in range(5):
        print(f"\nSequence {i+1}:")
        dataset.visualize_enhancement(i)
```

```
Transforming keypoints ...
100%|██████████| 545059/545059 [00:13<00:00, 40790.65it/s]
Transforming keypoints done
Stacking and padding frames...
100%|██████████| 212920/212920 [00:48<00:00, 4380.96it/s]
Stacking and padding frames done
Applying DBSCAN clustering...
100%|██████████| 212920/212920 [12:58<00:00, 273.58it/s]
DBSCAN clustering applied.
Transforming keypoints ...
100%|██████████| 545059/545059 [00:14<00:00, 38497.20it/s]
Transforming keypoints done
Stacking and padding frames...
100%|██████████| 212920/212920 [00:48<00:00, 4413.69it/s]
Stacking and padding frames done
Applying DBSCAN clustering...
100%|██████████| 212920/212920 [12:50<00:00, 276.44it/s]
DBSCAN clustering applied.
Starting enhancement pipeline...
Enhancing dataset with Kalman filtering...
  1%|          | 1650/170336 [01:00<1:49:27, 25.68it/s]
```

```
# def analyze_parameter_sensitivity(self, data_point, parameter_ranges):
#     """Analyze sensitivity to different parameters"""
#     results = {
#         'dbscan_eps': [],
#         'dbscan_min_samples': [],
#         'kalman_q': [],
#         'kalman_r': []
#     }

#     # Test DBSCAN parameters
#     for eps in parameter_ranges['eps']:
#         for min_samples in parameter_ranges['min_samples']:
#             dbscan = DBSCAN(eps=eps, min_samples=min_samples)
#             labels = dbscan.fit_predict(data_point['new_x'])

#             metrics = {
#                 'n_clusters': len(set(labels)) - (1 if -1 in labels else 0
#                 'noise_ratio': list(labels).count(-1) / len(labels),
#                 'silhouette': silhouette_score(data_point['new_x'], labels
#                 if len(set(labels)) > 1 else 0
#             }

#             results['dbscan_eps'].append({
#                 'eps': eps,
#                 'min_samples': min_samples,
#                 'metrics': metrics
#             })

#     # Test Kalman parameters
#     for q in parameter_ranges['q']:
```

```
#         for r in parameter_ranges['r']:
#             kf = self.init_kalman(q=q, r=r)
#             smoothed = self.smooth_sequence(data_point['new_x'], kf)

#             metrics = {
#                 'smoothness': np.mean(np.abs(np.diff(smoothed, axis=0))),
#                 'tracking_error': np.mean(np.linalg.norm(
#                     smoothed - data_point['new_x'], axis=1))
#             }

#             results['kalman_q'].append({
#                 'q': q,
#                 'r': r,
#                 'metrics': metrics
#             })

#     return results

# def plot_parameter_sensitivity(self, results):
#     """Plot parameter sensitivity analysis"""
#     fig = plt.figure(figsize=(20, 10))

#     # DBSCAN parameters
#     ax1 = fig.add_subplot(221)
#     eps_values = [r['eps'] for r in results['dbscan_eps']]
#     noise_ratios = [r['metrics']['noise_ratio'] for r in results['dbscan_e
#     ax1.plot(eps_values, noise_ratios)
#     ax1.set_title('DBSCAN eps vs Noise Ratio')
#     ax1.set_xlabel('eps')
#     ax1.set_ylabel('Noise Ratio')

#     ax2 = fig.add_subplot(222)
#     min_samples = [r['min_samples'] for r in results['dbscan_eps']]
#     n_clusters = [r['metrics']['n_clusters'] for r in results['dbscan_eps'
#     ax2.plot(min_samples, n_clusters)
#     ax2.set_title('min_samples vs Number of Clusters')
#     ax2.set_xlabel('min_samples')
#     ax2.set_ylabel('Number of Clusters')

#     # Kalman parameters
#     ax3 = fig.add_subplot(223)
#     q_values = [r['q'] for r in results['kalman_q']]
#     smoothness = [r['metrics']['smoothness'] for r in results['kalman_q']]
#     ax3.plot(q_values, smoothness)
#     ax3.set_title('Process Noise (Q) vs Smoothness')
#     ax3.set_xlabel('Q')
#     ax3.set_ylabel('Smoothness')

#     ax4 = fig.add_subplot(224)
#     r_values = [r['r'] for r in results['kalman_q']]
#     tracking_error = [r['metrics']['tracking_error'] for r in results['kal
#     ax4.plot(r_values, tracking_error)
#     ax4.set_title('Measurement Noise (R) vs Tracking Error')
#     ax4.set_xlabel('R')
#     ax4.set_ylabel('Tracking Error')

#     plt.tight_layout()
```

```
Start coding or generate with AI.
```