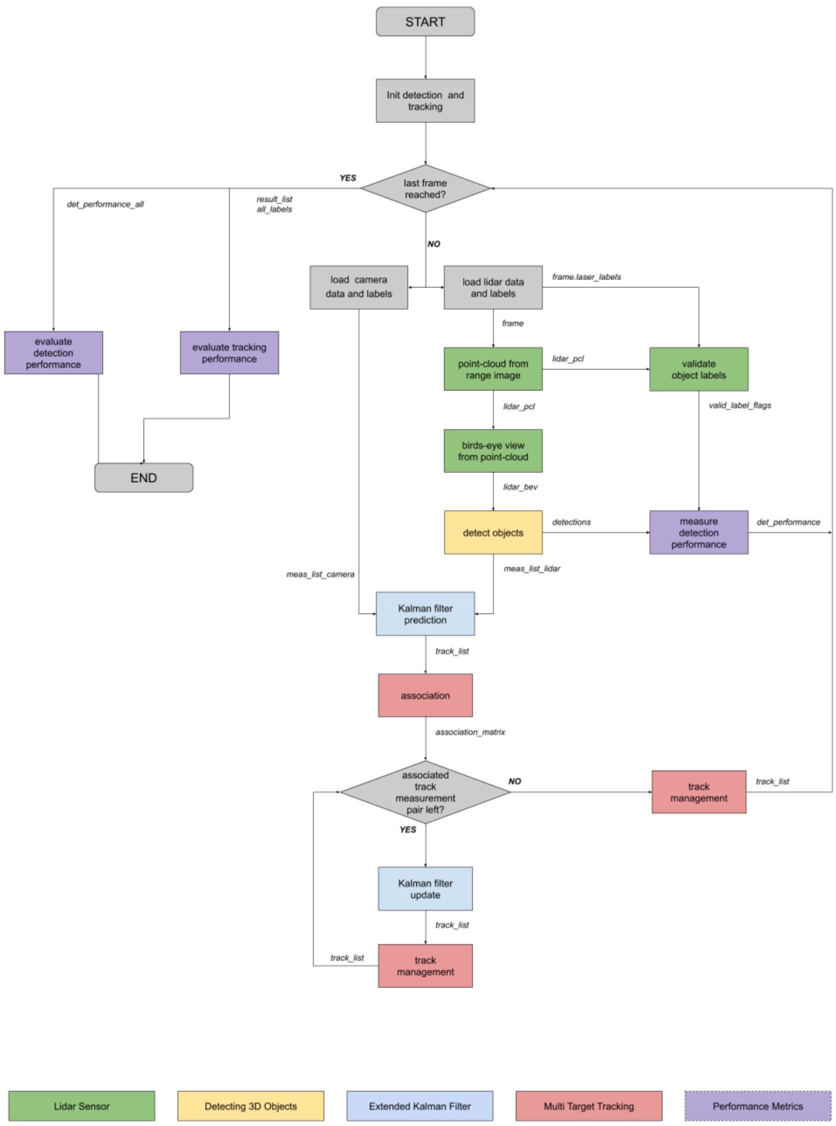
Udacity Self Driving Car Engineer NanoDegree / Mid Term / 3D Object Detection

This mid-Term Project helps to understand the logic of integrating with lidar data and

Computer vision and deep learning. My Project is performed on the Udacity Workspace.

So, This document is now mentioned about local envirment.

The below image shows the logic of this project.



Project File Structure

```
□project
├ □dataset --> contains the Waymo Open Dataset sequences
|
├ □misc
|   ├ evaluation.py --> plot functions for tracking visualization and RMSE calculation
|   ├ helpers.py --> misc. helper functions, e.g. for loading / saving binary files
|   └ objdet_tools.py --> object detection functions without student tasks
|   └ params.py --> parameter file for the tracking part
|
├ □results --> binary files with pre-computed intermediate results
|
├ □student
|   ├ association.py --> data association logic for assigning measurements to tracks incl. student
tasks
|   ├ filter.py --> extended Kalman filter implementation incl. student tasks
|   ├ measurements.py --> sensor and measurement classes for camera and lidar incl. student tasks
|   ├ objdet_detect.py --> model-based object detection incl. student tasks
|   ├ objdet_eval.py --> performance assessment for object detection incl. student tasks
|   ├ objdet_pcl.py --> point-cloud functions, e.g. for birds-eye view incl. student tasks
|   └ trackmanagement.py --> track and track management classes incl. student tasks
|
├ □tools --> external tools
|   ├ □objdet_models --> models for object detection
|   |   |
|   |   ├ □darknet
|   |   |   ├ □config
|   |   |   ├ □models --> darknet / yolo model class and tools
|   |   |   ├ □pretrained --> copy pre-trained model file here
|   |   |   |   └ complex_yolov4_mse_loss.pth
|   |   |   ├ □utils --> various helper functions
|   |   |
|   |   └ □resnet
|   |   |   ├ □models --> fpn_resnet model class and tools
|   |   |   ├ □pretrained --> copy pre-trained model file here
|   |   |   |   └ fpn_resnet_18_epoch_300.pth
|   |   |   ├ □utils --> various helper functions
|   |   |
|   └ □waymo_reader --> functions for light-weight loading of Waymo sequences
|
├ basic_loop.py
├ loop_over_dataset.py
```

Detailed Explainations of under student directory with loop_over_dataset.py

Loop_over_dataset.py : This file loading the each file which contains in student directory.

The code is responsible for iterating over the Waymo dataset, performing object detection, and visualizing the results. It integrates the object detection and tracking functionality from other files and allows you to visualize the processed LiDAR data in different forms, such as range images and BEV images .

Association.py : This file contains the logic for data association, which links detected objects with previous measurements to maintain object tracks over time. It implements methods like `associate` and `gating`, which determine how to match detected objects from frame to frame based on distance metrics like the Mahalanobis distance .

Filter.py : This file implements filtering methods, such as Kalman filters, which predict and update the state of tracked objects based on sensor data. It includes functions like `predict` and `update`, which are used to estimate the next state of an object (e.g., position and velocity) and correct these estimates based on new measurements .

Measurements.py : This file manages sensor data and measurements. It defines classes like `Sensor`, which represents sensors such as LiDAR or cameras, and `Measurement`, which stores the measurements captured by these sensors. This file is vital for handling sensor input during object detection and tracking .

Trackmanagement.py : This file handles the management of object tracks during the tracking phase. It initializes new tracks, deletes outdated tracks, and updates the state of existing tracks based on sensor measurements. It includes logic for managing track life cycles and scoring detections .

Objdet_pcl.py : This file is related to handling and processing point cloud data. Functions like `show_range_image` and `bev_from_pcl` are implemented here to visualize the LiDAR data and convert the 3D point cloud into a bird's-eye view (BEV) image format. This BEV format is critical for object detection using LiDAR .

Objdet_detect.py : This file is responsible for loading and creating deep learning models for object detection in LiDAR point clouds. It defines functions such as `create_model` to initialize the model architecture (e.g., ResNet or Darknet) and `detect_objects` to detect objects within the bird's-eye view image using the trained model .

Objdet_eval.py: This file is responsible for evaluating the performance of object detection. It contains functions like `measure_detection_performance`, which compares ground truth labels with detected objects and computes metrics such as Intersection over Union (IoU), true positives, and false positives. It assesses the quality of the object detection algorithm by calculating precision and recall .

Section 1 : Computer Lidar Point-Cloud from Range Image

Setting of this configuration

'loop_over_dataset.py' to run this section

```
## Select Waymo Open Dataset file and frame numbers
data_filename = 'training_segment-1005081002024129653_5313_150_5333_150_with_camera_labels.tfrecord' #프로젝트 지침 1단계, 범위 이미지 채널 시각화 (ID_S1_EX1)
#data_filename = 'training_segment-1005081002024129653_5313_150_5333_150_with_camera_labels.tfrecord' # Sequence 1
# data_filename = 'training_segment-10072231702153043603_5725_000_5745_000_with_camera_labels.tfrecord' # Sequence 2
# data_filename = 'training_segment-10963653239323173269_1924_000_1944_000_with_camera_labels.tfrecord' # Sequence 3
```

```
## Selective execution and visualization
exec_data = [] ##프로젝트 지침 1단계, 범위 이미지 채널 시각화 (ID_S1_EX1)
#exec_detection = ['bev_from_pcl', 'detect_objects', 'validate_object_labels', 'measure_detection_performance']
exec_detection = [] #프로젝트 지침 1단계, 범위 이미지 채널 시각화 (ID_S1_EX1)
exec_tracking = [] # options are 'perform_tracking',#프로젝트 지침 1단계, 범위 이미지 채널 시각화 (ID_S1_EX1)
#exec_visualization = [] # options are 'show_range_image', 'show_bev', 'show_pcl', 'show_labels_in_image', 'sho
exec_visualization = ['show_range_image'] #프로젝트 지침 1단계, 범위 이미지 채널 시각화 (ID_S1_EX1)
```

'Show_range_image function In the file 'objdet_pcl.py

```
#이 함수는 LiDAR 데이터의 범위(range)와 강도(intensity) 이미지를 시각적으로 표현하는 함수
def show_range_image(frame, lidar_name): #show_range_image 함수,역할: LiDAR 범위(range) 데이터를 
시각화하여 각 포인트의 범위와 강도(intensity)를 보여줌.
    # frame: Waymo 데이터셋의 프레임 객체 ( LiDAR 데이터를 포함하는 정보)
    # idar_name: 처리할 LiDAR 센서의 이름 (예를 들어, 'TOP', 'FRONT' 등 특정 LiDAR 센서를 선택)

    ####### ID_S1_EX1 START #######   라이다 데이터를 시각화
    print("show_range_image - student task")

    # extract lidar data and range image for the roof-mounted lidar
    #LiDAR 데이터 추출 및 압축 해제:
    lidar = waymo_utils.get(frame.lasers, lidar_name)

    ri = dataset_pb2.MatrixFloat() #dataset_pb2.MatrixFloat(): Waymo 데이터 구조의 범위 이미지 데이터를 
                                                              저장하는 객체.

    ri.ParseFromString(zlib.decompress(lidar.ri_return1.range_image_compressed))  # Decompress and 
                                                                                   parse

    ri = np.array(ri.data).reshape(ri.shape.dims)  # Reshape range image
```

```
    # extract the range and the intensity channel from the range image
    #범위 및 강도 채널 추출:
    ri_range = ri[:, :, 0]  # range channel
    #ri[:, :, 0]: LiDAR 데이터의 첫 번째 채널을 사용해 범위 데이터를 추출

    ri_intensity = ri[:, :, 1]  # intensity channel
    #ri[:, :, 1]: 두 번째 채널에서 강도 데이터를 추출

    # set values <0 to zero
    #음수값을 0 으로 설정
    ri_range[ri_range < 0] = 0.0
    ri_intensity[ri_intensity < 0] = 0.0

    # map the range channel onto an 8-bit scale and normalize
    #범위 및 강도 데이터의 8 비트 스케일 변환:
    ri_range = (ri_range - np.min(ri_range)) / (np.max(ri_range) - np.min(ri_range)) * 255
    img_range = ri_range.astype(np.uint8)
    # map the intensity channel onto an 8-bit scale
    #강도 데이터의 8 비트 스케일 변환:
    ri_intensity = np.clip(ri_intensity, 0, 1) * 255

    img_intensity = ri_intensity.astype(np.uint8)

    # stack the range and intensity image vertically
    #범위와 강도 이미지를 수직으로 쌓기:
    img_range_intensity = np.vstack((img_range, img_intensity))

     # 범위 이미지를 전방 x축의 좌측 및 우측에서 +/- 90도로 잘라내기 (1차 제출후 멘토의 코드)

    deg_90_range = int(img_range_intensity.shape[1] / 4)
    center_range = int(img_range_intensity.shape[1] / 2)
    img_range_intensity = img_range_intensity[:, center_range -deg_90_range:center_range +
                                                   deg_90_range]



    return img_range_intensity #return img_range_intensity: 결합된 범위 및 강도 이미지를 반환

    ####### ID_S1_EX1 END #######
```
,

The Result is shown below

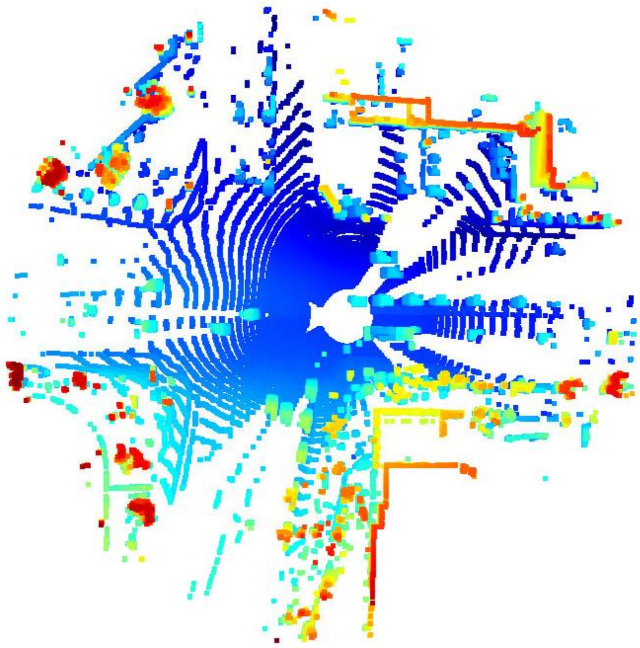
On the (ID_S1_Ex2) Step, the function in the objdet_pcl.py

```
# 이 함수는 Open3D 라이브러리를 사용해 LiDAR 포인트 클라우드를 시각화하는 함수입
def show_pcl(pcl):
    #pcl: LiDAR 포인트 클라우드 데이터를 입력으로 받는 변수. 이 데이터는 3 차원 좌표로 이루어진 포인트들의 집합


    ####### ID_S1_EX2 START #######  라이다 포인트 클라우드 시각화
    print("show_pcl - student task")

    # step 1: initialize open3d with key callback and create window
    vis = o3d.visualization.VisualizerWithKeyCallback()

    vis.create_window()

    # step 2: create instance of open3d point-cloud class
    pcd = o3d.geometry.PointCloud()
          #o3d.geometry.PointCloud(): Open3D 에서 제공하는 포인트 클라우드 객체를 생성합니다. 이 객체는 
3 차원 좌표 데이터를 포함하고, 시각화를 위한 다양한 기능을 제공

    #내부 로직 : o3d.geometry.PointCloud: Open3D 라이브러리를 사용해 3D 포인트 클라우드를 시각화

    # step 3: set points in pcd instance by converting the point-cloud into 3d vectors
    #포인트 클라우드 데이터를 3D 벡터로 변환하여 설정:
    pcd.points = o3d.utility.Vector3dVector(pcl[:, :3])


    # step 4: add the pcd instance to visualization and update geometry
   #포인트 클라우드를 시각화에 추가하고 업데이트
    #내부로직 : vis: 시각화를 위한 Open3D 의 시각화 객체로, 포인트 클라우드를 표시하고 창을 띄움
    vis.add_geometry(pcd)

    vis.update_geometry(pcd)

    # step 5: visualize point cloud and keep window open until right-arrow is pressed
    vis.poll_events() #시각화 창에서 발생하는 사용자 이벤트(키 입력, 마우스 클릭 등)를 처리
    vis.update_renderer() #vis.update_renderer(): 시각화 화면을 다시 그림
    vis.run() #시각화 창을 실행하고, 사용자가 종료할 때까지 창을 유지
    vis.destroy_window()

    ####### ID_S1_EX2 END #######
```

T

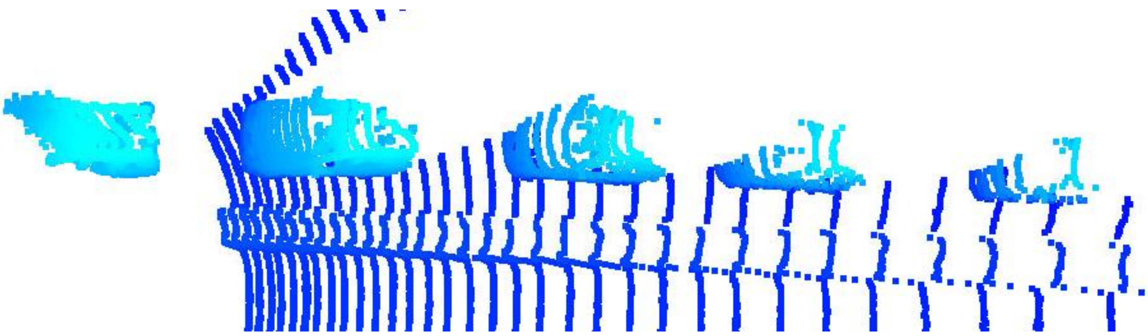
The Result is below

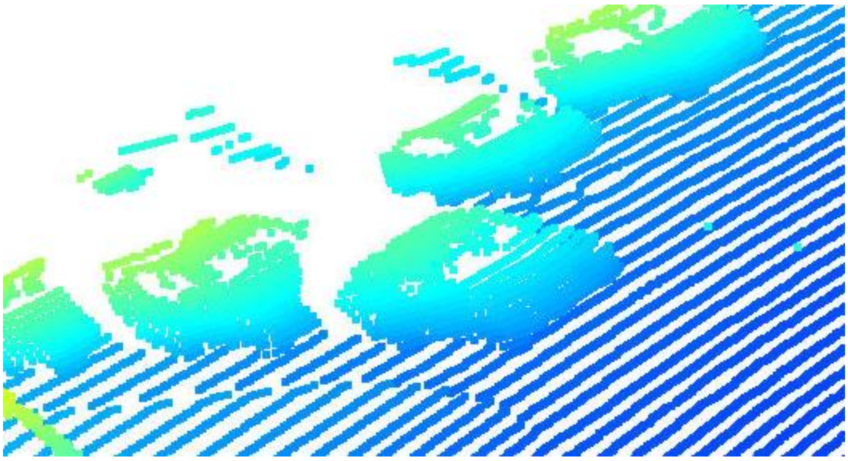
Top View

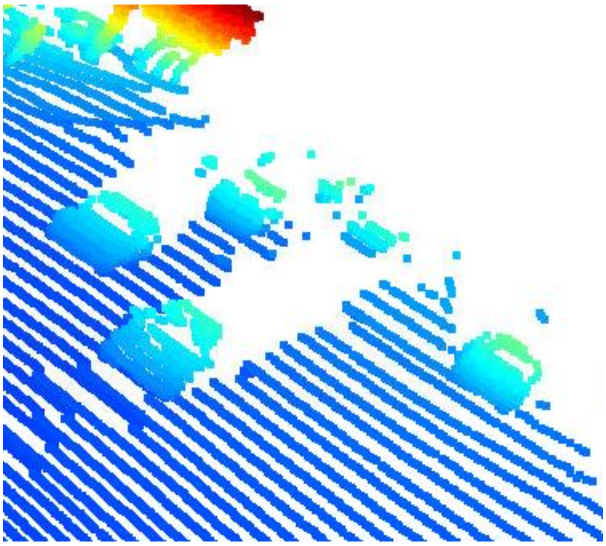


This Top view show the origin position of lidar. It is center of the image.

The Detail View of Lidar Images.







I can recognized the shape of cars. The bonnet and roof and side of the body.

Some of cars shows it’s wheels. But it is not perfectly recognized as car as it is.

It is mixed with many noise. So, it should be implemented with computer vision.

It will be the next steps.

Section 2: Create Birds-Eye View (BEV) from Lidar PCL

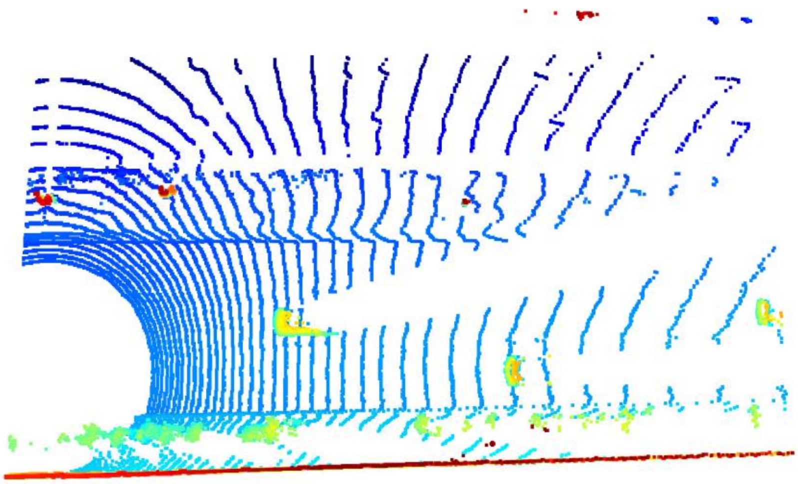
Setting of this configuration

From the file “objdet_pcl.py” , the function bev_from_pcl

```
def bev_from_pcl(lidar_pcl, configs):
    #lidar_pcl: 입력으로 들어오는 LiDAR 포인트 클라우드 데이터
    #configs: BEV 맵을 생성하기 위한 설정 값들이 포함된 객체

    # remove lidar points outside detection area and with too low reflectivity
        #포인트 필터링 ,mask: 설정된 범위 내에 있는 포인트만 필터링하는 마스크.

    mask = np.where((lidar_pcl[:, 0] >= configs.lim_x[0]) & (lidar_pcl[:, 0] <= configs.lim_x[1]) &
                    (lidar_pcl[:, 1] >= configs.lim_y[0]) & (lidar_pcl[:, 1] <= configs.lim_y[1]) &
                    (lidar_pcl[:, 2] >= configs.lim_z[0]) & (lidar_pcl[:, 2] <= configs.lim_z[1]))
    lidar_pcl = lidar_pcl[mask]
    #mask: LiDAR 포인트 데이터 중에서 지정된 범위 내에 있는 포인트만 남기기 위해 마스크를 설정.
    #이 마스크는 X, Y, Z 좌표가 주어진 설정(configs)의 한계 범위 내에 있을 때만 참(True)임.
    #lidar_pcl[mask]: 설정된 범위 내의 포인트만 추출하여 LiDAR 포인트 클라우드로 만듬





    # shift level of ground plane to avoid flipping from 0 to 255 for neighboring pixels
    #지면 레벨 조정
    lidar_pcl[:, 2] = lidar_pcl[:, 2] - configs.lim_z[0]



    # convert sensor coordinates to bev-map coordinates (center is bottom-middle)
    ####### ID_S2_EX1 START #######      BEV 변환을 위한 포인트 클라우드 처리
    print("bev_from_pcl - student task ID_S2_EX1")





    # step 1 : compute bev-map discretization by dividing x-range by the bev-image height (see configs)
    bev_discretization = (configs.lim_x[1] - configs.lim_x[0]) / configs.bev_height
```

```
    # step 2 : create a copy of the lidar pcl and transform all metric x-coordinates into bev-image
coordinates
    #BEV 맵으로 변환, lidar_pcl_cpy[:, 0], lidar_pcl_cpy[:, 1]: 포인트 클라우드의 좌표를 BEV 맵 좌표로 변환.
    lidar_pcl_cpy = np.copy(lidar_pcl)
    lidar_pcl_cpy[:, 0] = np.int_(np.floor(lidar_pcl_cpy[:, 0] / bev_discretization))
    #np.copy(lidar_pcl): LiDAR 포인트 클라우드 데이터를 복사하여 변환에 사용할 복사본을 만듬
    #lidar_pcl_cpy[:, 0]: X 좌표를 BEV 맵의 픽셀 좌표로 변환


    # step 3 : perform the same operation as in step 2 for the y-coordinates but make sure that no
negative bev-coordinates occur
    lidar_pcl_cpy[:, 1] = np.int_(np.floor((lidar_pcl_cpy[:, 1] - configs.lim_y[0]) /
bev_discretization))
    lidar_pcl_cpy[lidar_pcl_cpy[:, 1] < 0, 1] = 0  # avoid negative indices




    # step 4 : visualize point-cloud using the function show_pcl from a previous task
    show_pcl(lidar_pcl_cpy) #섹션 2 과제위해서
```

The Result is below



and in the same function (def bev_from_pcl)

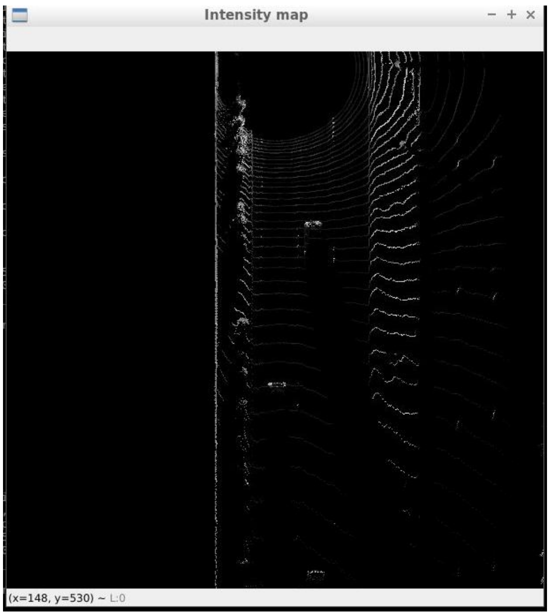
Intensity map, and height map shows by those codes.

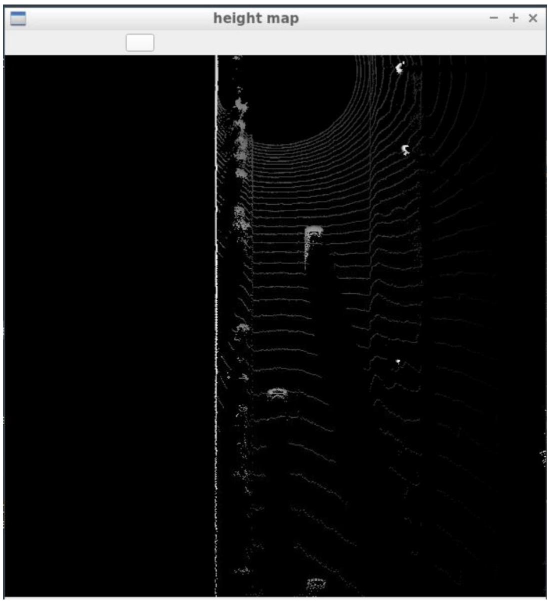
```
## step 1 : create a numpy array filled with zeros which has the same dimensions as the BEV map
    #강도 레이어 생성:
    intensity_map = np.zeros((configs.bev_height + 1, configs.bev_width + 1)) #멘토가 수정한 것(+1 추가)
```

```
    # step 2 : re-arrange elements in lidar_pcl_cpy by sorting first by x, then y, then -z (use
numpy.lexsort)
    #포인트 강도값 정렬 및 처리:
    lidar_pcl_cpy[lidar_pcl_cpy[:,3]>1.0,3] = 1.0  #멘토가 준 것 (한줄 추가) ,# 강도 값을 1.0 으로 제한
    #lidar_pcl_cpy[:,3]>1.0,3] = 1.0: 강도 값이 1 을 넘는 포인트는 1 로 클리핑하여 정규화
    index_intensity = np.lexsort((-lidar_pcl_cpy[:, 3], lidar_pcl_cpy[:, 1], lidar_pcl_cpy[:, 0]))
#기존 [:, 2]였는데 멘토가 [, 3]으로 변경
                    #np.lexsort: X, Y 좌표 순서대로 포인트를 정렬하며, Z 좌표를 기준으로 정렬된 데이터를
가져옴
    lidar_pcl_top = lidar_pcl_cpy[index_intensity] #기존 [idx] -->멘토[index_intensity]

    ## step 3 : extract all points with identical x and y such that only the top-most z-coordinate is
kept (use numpy.unique)
    ##          also, store the number of points per x,y-cell in a variable named "counts" for use in
the next task
    #step3 는 멘토가 완전히 바꿈
    lidar_num, lidar_indices, lidar_count = np.unique(lidar_pcl_cpy[:, 0:2], axis=0, return_index=True,
return_counts=True)
    lidar_pcl_top = lidar_pcl_cpy[lidar_indices]
    #np.unique: X, Y 좌표가 중복되는 포인트 중 가장 위에 있는(Z 가 가장 큰) 포인트만 남김




    ## step 4 : assign the intensity value of each unique entry in lidar_top_pcl to the intensity map
    ##          make sure that the intensity is scaled in such a way that objects of interest (e.g.
vehicles) are clearly visible
    ##          also, make sure that the influence of outliers is mitigated by normalizing intensity on
the difference between the max. and min. value within the point cloud
   #step4 도 멘토가 완전히 바꿈
   #강도 값 맵에 할당:
    intensity_map[np.int_(lidar_pcl_top[:, 0]), np.int_(lidar_pcl_top[:, 1])] = lidar_pcl_top[:, 3] /
(np.amax(lidar_pcl_top[:, 3])-np.amin(lidar_pcl_top[:, 3]))
    #intensity_map: 강도 값을 BEV 맵의 픽셀에 할당. 이때 최대값과 최소값의 차이로 정규화하여 강도 값을
분포시키고, 차량 등의 객체가 뚜렷하게 보이도록 조정


    # step 5: intensity map 시각화 (흑백 이미지로 시각화) 제프리의 코드를 따라함
    img_intensity = (intensity_map * 256).astype(np.uint8) #기존과 동일
    cv2.imshow("Intensity map", img_intensity)  #섹션 2 를 위해서 비주석처리
    cv2.waitKey(0) #멘토 주석처리
    cv2.destroyAllWindows()  # #멘토 주석처리
    ####### ID_S2_EX2 END #######

    # Compute height layer of the BEV map
    ####### ID_S2_EX3 START #######     BEV 맵의 높이(height) 레이어 계산
    print("student task ID_S2_EX3")
    # step 1 : create a numpy array filled with zeros which has the same dimensions as the BEV map
```

```python
    height_map = np.zeros((configs.bev_height + 1, configs.bev_width + 1)) #멘토가 각각 +1 추가
    #height_map: LiDAR 포인트의 높이를 저장할 배열을 생성하고, Z 값에 따라 정규화된 높이를 할당



    ## step 2 : assign the height value of each unique entry in lidar_top_pcl to the height map
    ##          make sure that each entry is normalized on the difference between the upper and lower height defined in the config file
    ##          use the lidar_pcl_top data structure from the previous task to access the pixels of the height_map
    height_map[np.int_(lidar_pcl_top[:, 0]), np.int_(lidar_pcl_top[:, 1])] = lidar_pcl_top[:, 2] / float(np.abs(configs.lim_z[1] - configs.lim_z[0])) #기존과 동일

    #멘토가 주신 코드 (이미지 강도맵구현)
    img_height = height_map * 256
    img_height = img_height.astype(np.uint8)
    cv2.imshow('height map', img_height) #섹션 2 과제출력
    cv2.waitKey(0)
    cv2.destroyAllWindows()
```

The Result is shown below, it shows the normalizing height.





Section 3: Model-based Object Detection in BEV Image

This section purpose is to detect vehicle from an images with using deep learning model and integrated in one window with ladar image. Loop_over_dataset.py configuration is below.

```
exec_data = ['pcl_from_rangeimage', 'load_image']
```

```
exec_detection = ['bev_from_pcl', 'detect_objects']
exec_tracking = []
exec_visualization = ['show_objects_in_bev_labels_in_camera']
exec_list = make_exec_list(exec_detection, exec_tracking, exec_visualization)
```

with objdet_detect.py file has a function (def detect_object) is loaded to show

```
def detect_objects(input_bev_maps, model, configs):

    # deactivate autograd engine during test to reduce memory usage and speed up computations
    with torch.no_grad():
        #with torch.no_grad(): 파이토치에서 autograd 엔진을 비활성화.이는 메모리 사용량을 줄이고,
추론(inference) 중 속도를 높이기 위한 목적으로 사용
        #input_bev_maps: 입력 매개변수, BEV 형태의 LiDAR 데이터
        #model: 사용 중인 객체 감지 모델
        #configs: 설정 값들이 포함된 객체로, 감지할 때 필요한 여러 파라미터가 들어 있음

        # perform inference
        outputs = model(input_bev_maps)
        #model(input_bev_maps): 모델에 입력 BEV 맵을 전달하여 추론을 수행

        # decode model output into target object format
        if 'darknet' in configs.arch:
         # perform post-processing
           output_post = post_processing_v2(outputs, conf_thresh=configs.conf_thresh,
nms_thresh=configs.nms_thresh) # 여기는 그대로 유지
                       #post_processing_v2: 객체 감지 결과를 후처리하는 함수. Non-Maximum
Suppression(NMS)과 같은 후처리를 수행
                       #conf_thresh: 감지된 객체의 신뢰도 임계값
                       #nms_thresh: NMS에서 사용하는 임계값
            detections = [] #감지된 객체를 저장할 리스트
            for sample_i in range(len(output_post)): #output_post 리스트의 각 항목에 대해 반복문을 돌림
                                                   ##sample_i: 인덱스 번호
                if output_post[sample_i] is None:
                    #if output_post[sample_i] is None: 감지 결과가 없을 경우, 다음 루프로
                    continue #ontinue: 현재 루프를 건너뛰고, 다음 루프
                detection = output_post[sample_i]  #detection: output_post에서 감지된 객체 정보를 가져옴
                for obj in detection: #for obj in detection: 감지된 객체들의 각 항목에 대해 반복문을 실행
                    x, y, w, l, im, re, _, _, _ = obj #obj: 하나의 감지된 객체
```

```
                #x, y: 감지된 객체의 중심 좌표
                #w, l: 감지된 객체의 너비와 길이
                #im, re: 물체의 방향을 나타내는 복소수 값
                #im, re: 물체의 방향을 나타내는 복소수 값

                yaw = np.arctan2(im, re)
                #yaw: 감지된 객체의 방향을 나타내는 값. im 과 re 를 사용하여 방향을 계산
                #np.arctan2: 두 값의 아크탄젠트(atan2)를 계산하여 방향을 결정

                detections.append([1, x, y, 0.0, 1.50, w, l, yaw])
                #detections.append: 감지된 객체의 정보를 리스트에 추가.
                #1 은 감지된 객체의 클래스(예: 자동차)를 나타내며, x, y 는 좌표, w, l 은 너비와 길이,
                # yaw 는 방향을 의미



        elif 'fpn_resnet' in configs.arch:
            #elif 'fpn_resnet' in configs.arch: 모델 아키텍처가 'fpn_resnet'일 경우, 이 조건문이 실행

            ####### ID_S3_EX1-5 START #######
            # Apply sigmoid to the 'hm_cen' and 'cen_offset' outputs before decoding --1차 제출후 멘토의
코드를 이렇게 수정해서 사용하기로 함. 직접 torch.sigmoid사용

            outputs["hm_cen"] = torch.sigmoid(outputs["hm_cen"])
            outputs["cen_offset"] = torch.sigmoid(outputs["cen_offset"])

            # Decode the outputs into detections
            detections = decode(outputs['hm_cen'], outputs['cen_offset'], outputs['direction'],
outputs['z_coor'], outputs['dim'], K=configs.K)


            # Apply post-processing to filter the results
            output_post = post_processing(detections, configs) # 여기는 post_processing 함수 호출 시
conf_thresh 와 같은 개별 인자를 전달하는 대신 configs 객체를 통째로 넘기는 방식
            #output_post: 후처리된 결과를 저장하는 변수
            #post_processing(detections, configs): 감지된 객체에 대해 후처리를 수행하는 함수

            output_post = output_post[0][1]
            #output_post[0][1]: 후처리된 객체 중 특정 항목을 선택




            print(type(output_post))  # 결과값의 타입을 출력합니다.
            print(output_post)  # 결과값을 출력합니다.
            detections = output_post
            print("student task ID_S3_EX1-5")
```

```
        ####### ID_S3_EX1-5 END #######

    ####### ID_S3_EX2 START #######
    #######
    # Extract 3d bounding boxes from model response
    print("student task ID_S3_EX2")
    objects = [] #objects: 최종 감지된 객체를 저장할 리스트

    #for det in detections: 감지된 객체들에 대해 반복문을 실행
    for det in detections:
        # step 1 : check whether there are any detections
        if len(det) > 0:
                #if len(det) > 0: 감지된 객체가 하나라도 있을 경우, 조건을 만족

            # step 2 : loop over all detections
                _, bev_x, bev_y, z, h, bev_w, bev_l, yaw = det
                #bev_x, bev_y: 감지된 객체의 BEV 좌표
                #z: 객체의 z 좌표
                #h: 객체의 높이
                #bev_w, bev_l: 객체의 너비와 길이
                #yaw: 객체의 방향
                # step 3 : perform the conversion using the limits for x, y and z set in the configs
structure

                x = bev_y / configs.bev_height * \
                    (configs.lim_x[1] - configs.lim_x[0])
                #x = bev_y / configs.bev_height * (configs.lim_x[1] - configs.lim_x[0])


                y = bev_x / configs.bev_width * \
                    (configs.lim_y[1] - configs.lim_y[0]) + configs.lim_y[0]
                #y = bev_x / configs.bev_width * (configs.lim_y[1] - configs.lim_y[0]) +
configs.lim_y[0]
                #x, y: BEV 좌표를 실제 월드 좌표로 변환하는 계산
               #configs.bev_height 와 configs.bev_width 를 사용해 비율을 맞춤


                z = z + configs.lim_z[0] #z: z 좌표에 제한 값을 추가

                w = bev_w / configs.bev_width * \
                    (configs.lim_y[1] - configs.lim_y[0])
                l = bev_l / configs.bev_height * \
                    (configs.lim_x[1] - configs.lim_x[0])
                #w, l: 객체의 너비와 길이를 실제 월드 좌표로 변환



                if ((x >= configs.lim_x[0]) and (x <= configs.lim_x[1])
```

```
                and (y >= configs.lim_y[0]) and (y <= configs.lim_y[1])
                    and (z >= configs.lim_z[0]) and (z <= configs.lim_z[1])):

                # step 4 : append the current object to the 'objects' array
                objects.append([1, x, y, z, h, w, l, yaw])

    #######
    ####### ID_S3_EX2 END #######
    return objects
```

the image in one window.

Section 4: Performance Evaluation for Object Detection

The goal of this task is to evaluate the performance of the object detection algorithm by pairing ground-truth labels with detected objects. This process helps determine whether an object has been (a) missed (false negative), (b) successfully detected (true positive), or (c) falsely reported (false positive). The geometrical overlap (Intersection over Union, IoU) between the bounding boxes of the labels and detected objects is computed, indicating the percentage of overlap in relation to the bounding box areas. If multiple matches are found, the pair with the highest IoU is kept. False negatives and false positives are then calculated to derive precision and recall.

This model uses the DarkNet architecture from Complex-YOLO, and the data is from the Waymo Dataset without further retraining. After processing all the frames of a sequence, a precision-recall curve is plotted over 100 frames, and precision and recall are computed. Another graph shows the comparison where ground-truth labels are taken as objects.

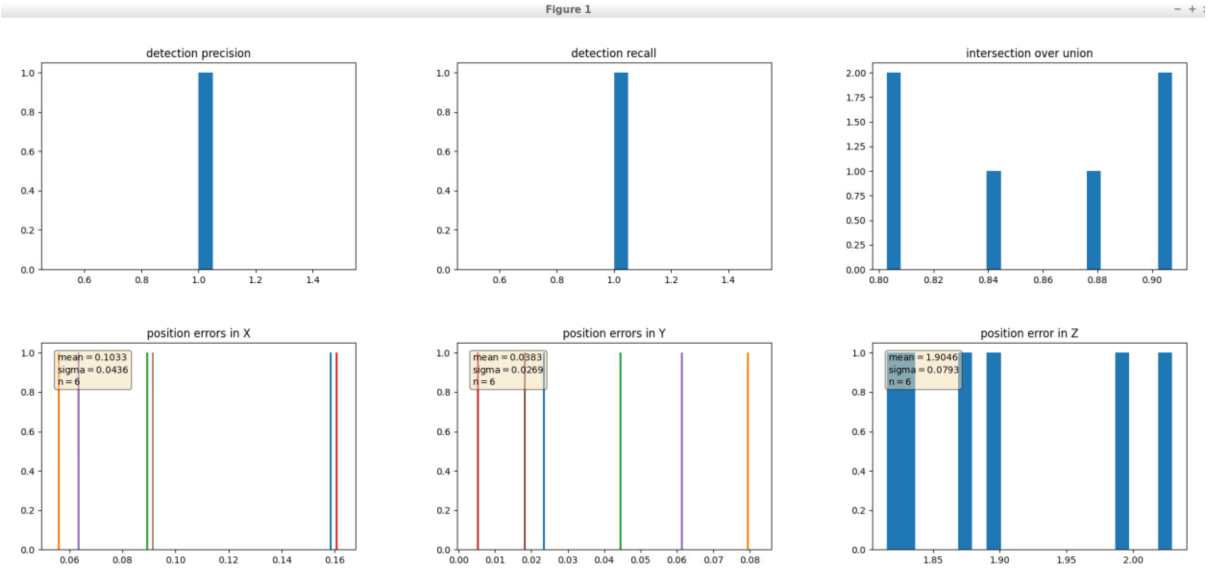
In this section, loop_over_dataset.py with objdet_eval.py is working together and show the results.

In the loop_over_dataset.py file has a configuration like below

```
## Initialize object detection

configs_det = det.load_configs(model_name='darknet')
model_det = det.create_model(configs_det)



exec_data = ['pcl_from_rangeimage'
exec_detection = ['bev_from_pcl', 'detect_objects', 'validate_object_labels',
'measure_detection_performance'] # options are 'bev_from_pcl', 'detect_objects',
exec_tracking = []
exec_visualization = ['show_detection_performance']
exec_list = make_exec_list(exec_detection, exec_tracking, exec_visualization) #make_exec_list lpers.py)
```

The result is shown below (with using loop_over_dataset_4-1.py)

(The Processing is too short term analysis)

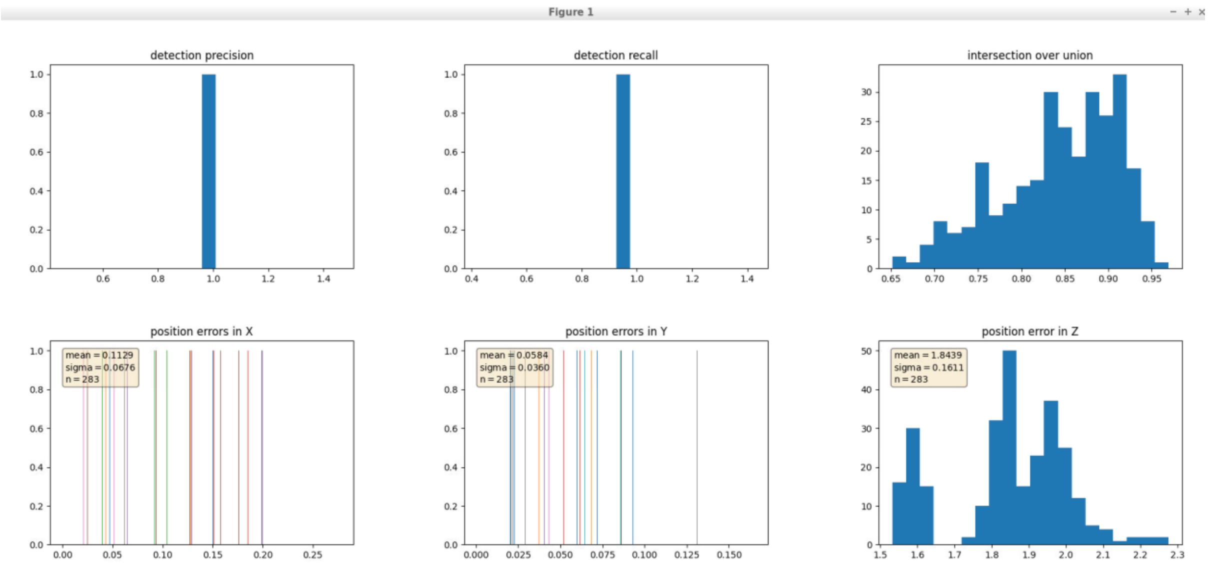


The result is shown below (with using loop_over_dataset_4-3.py)

(The Processing is quite long , it around 40minutes. Until frame #150)

Because it is **model based detection.**

The result is below



The final using the file ‘loop_over_dataset_4-3_second.py’

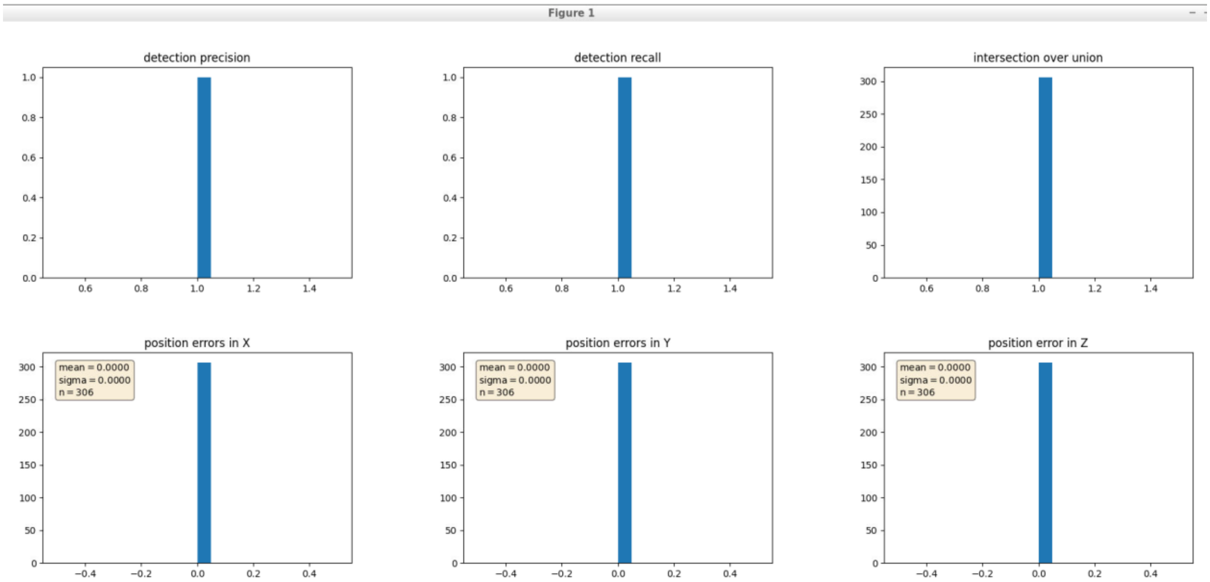
In this run, configs_det.use_labels_as_objects should be set to ‘True’

This setting using groundtruth labels as objects.

(**Ground Truth : already labled by human)**

So, ananlysis is finished within 5minutes.

This should produce precision = 1.0, recall = 1.0.



Reference:

#1.Model-Based Detection:

Detection method: The trained model (e.g., Complex-YOLO) detects objects based on the input data.

Outcome: The model takes inputs such as LiDAR data or images and returns the predicted locations (bounding boxes) of detected objects. The accuracy of these detections depends on the performance of the model. The detected objects are generated based on the learned features from the model.

Purpose: To evaluate the performance of the model, the predicted bounding boxes are compared with ground-truth labels using Intersection over Union (IoU) to measure detection accuracy.

#2. Detection Using Ground-Truth Labels:

Detection method: Instead of using the model's predictions, this approach directly uses the ground-truth data from datasets like the Waymo Dataset to detect objects.

Outcome: Rather than using model predictions, the objects are directly taken from the ground-truth labels. In this case, the actual labeled data is used for evaluation, without involving model inference.

Purpose: Ground-truth detection is used as a baseline to compare the model's performance. When configs_det.use_labels_as_objects = True is set, the actual labels are treated as detected objects for evaluation purposes, skipping the model-based detection.

Summary of Differences:

Model-Based Detection: Uses the model's predictions to detect objects, typically for evaluating the model's detection performance.

Ground-Truth Detection: Uses the actual labels (ground-truth data) for detection, serving as a benchmark for comparison.

Thank you!