1. labeled data로 모델 학습 -> weight 구하기

- labeled data : 따로 분리한 test 제외 900개

2. 위에서 구한 모델 weight로 unlabeled data의 annotation 추론 -> txt파일 생성

In [1]:
```
import os
import numpy as np
import pandas as pd
import pickle
```

In [2]:
```
# for modeling
import torch
torch.manual_seed(0)
import matplotlib.patches as patches
import matplotlib.pyplot as plt
from PIL import Image
import torchvision
from torchvision import transforms, datasets, models
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor
import time
from tqdm import tqdm
import csv
```

In [3]:
```
os.chdir('/home/work/sample-notebooks/train')
```

labeled data(train, test) 파일명 list 로드

In [4]:
```
# train 데이터
with open('./labeled_data/train_img.pkl', 'rb') as file:
    train_img_list = pickle.load(file)

# test 데이터
with open('./labeled_data/test_img.pkl', 'rb') as file:
    test_img_list = pickle.load(file)
```

unlabeled data 파일명 리스트 생성

In [5]:
```
unlabeled_path = './unlabeled_data/'
unlabeled_list = os.listdir(unlabeled_path)
unlabeled_list.sort()
del unlabeled_list[-1]
print(len(unlabeled_list))
unlabeled_list[:5]
```

Out[5]:
```
19000
['sk_ul_000000.jpg',
 'sk_ul_000001.jpg',
 'sk_ul_000002.jpg',
 'sk_ul_000003.jpg',
 'sk_ul_000004.jpg']
```

데이터셋 클래스 정의

In [6]:
```
def generate_box(df_obj, size):  # 객체 하나씩 (한 이미지에 객체 여러개여도 하나씩)
    W = size[0]
    H = size[1]
    xmin = df_obj['xmin']*W
    ymin = df_obj['ymin']*H
    xmax = df_obj['xmax']*W
    ymax = df_obj['ymax']*H

    return [xmin, ymin, xmax, ymax]


def generate_label(df_obj):
    adjust_label = 1

    return int(df_obj['class'] + adjust_label)


def generate_target(file, size):
    df = pd.read_table(file, sep = ' ', header = None, names = ['class','xmin','ymin

    boxes = []
    labels = []
    for obj in range(df.shape[0]):

        boxes.append(generate_box(df.iloc[obj], size))
        labels.append(generate_label(df.iloc[obj]))

    boxes = torch.as_tensor(boxes, dtype = torch.float32)
    labels = torch.as_tensor(labels, dtype = torch.int64)

    target = {}
    target["boxes"] = boxes
    target["labels"] = labels

    return target
```

In [7]:
```
## for labeled data (train, test)
class MaskDataset(object):
    def __init__(self, transforms, path, imgs):
        self.transforms = transforms
        self.path = path  # img path
        self.imgs = imgs   # img 파일명 list

    def __getitem__(self, idx):
        # load image and masks
        file_image = self.imgs[idx]
        file_label = self.imgs[idx][:-3] + 'txt'
        img_path = os.path.join(self.path, file_image)

        if 'test' in self.path:
            label_path = os.path.join('./labeled_data/test_annotations/', file_label)
        else:  # 'train'
            label_path = os.path.join('./labeled_data/train_annotations/', file_label

        img = Image.open(img_path).convert('RGB')
        size = img.size

        # generate label
        target = generate_target(label_path, size)

        if self.transforms is not None:
            img = self.transforms(img)

        return img, target
```

```
    def __len__(self):
        return len(self.imgs)

data_transform = transforms.Compose([      # transforms.Compose : list 내의 작업을 연
    transforms.ToTensor()     # ToTensor : numpy 이미지에서 torch 이미지로 변경
    ])


def collate_fn(batch):
    return tuple(zip(*batch))

dataset = MaskDataset(data_transform, './labeled_data/train_images/', train_img_list)
test_dataset = MaskDataset(data_transform, './labeled_data/test_images/', test_img_lis

data_loader = torch.utils.data.DataLoader(dataset, batch_size = 4, collate_fn = colla
test_data_loader = torch.utils.data.DataLoader(test_dataset, batch_size = 2, collate_
```

In [8]:

```
## for unlabeled data
class MaskDataset(object):
    def __init__(self, transforms, path, imgs):
        self.transforms = transforms
        self.path = path  # img path
        self.imgs = imgs   # img 파일명 list

    def __getitem__(self, idx):
        # load image and masks
        file_image = self.imgs[idx]
        img_path = os.path.join(self.path, file_image)

        img = Image.open(img_path).convert('RGB')

        if self.transforms is not None:
            img = self.transforms(img)

        target = 0
        return img, target

    def __len__(self):
        return len(self.imgs)

data_transform = transforms.Compose([
    transforms.ToTensor()
    ])


def collate_fn(batch):
    return tuple(zip(*batch))

ul_dataset = MaskDataset(data_transform, './unlabeled_data/', unlabeled_list)

ul_data_loader = torch.utils.data.DataLoader(ul_dataset, batch_size = 4, collate_fn =
```

모델 불러오기

In [9]:

```
def get_model_instance_segmentation(num_classes):  # num_classes 는 background 클래스

    model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained = True)
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

    return model
```

전이학습

In [10]:
```
model = get_model_instance_segmentation(8)  # 실제 클래스 개수 : 7 (0~6)

device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
model.to(device)
```

Out[10]:
```
FasterRCNN(
  (transform): GeneralizedRCNNTransform(
      Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
      Resize(min_size=(800,), max_size=1333, mode='bilinear')
  )
  (backbone): BackboneWithFPN(
    (body): IntermediateLayerGetter(
      (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=F
alse)
      (bn1): FrozenBatchNorm2d(64)
      (relu): ReLU(inplace=True)
      (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=F
alse)
      (layer1): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(64)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), b
ias=False)
          (bn2): FrozenBatchNorm2d(64)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(256)
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): FrozenBatchNorm2d(256)
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(64)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), b
ias=False)
          (bn2): FrozenBatchNorm2d(64)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(256)
          (relu): ReLU(inplace=True)
        )
        (2): Bottleneck(
          (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(64)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), b
ias=False)
          (bn2): FrozenBatchNorm2d(64)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(256)
          (relu): ReLU(inplace=True)
        )
      )
      (layer2): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(128)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
```

```
          (bn2): FrozenBatchNorm2d(128)
          (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(512)
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
            (1): FrozenBatchNorm2d(512)
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(128)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
          (bn2): FrozenBatchNorm2d(128)
          (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(512)
          (relu): ReLU(inplace=True)
        )
        (2): Bottleneck(
          (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(128)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
          (bn2): FrozenBatchNorm2d(128)
          (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(512)
          (relu): ReLU(inplace=True)
        )
        (3): Bottleneck(
          (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(128)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
          (bn2): FrozenBatchNorm2d(128)
          (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(512)
          (relu): ReLU(inplace=True)
        )
      )
      (layer3): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(256)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
          (bn2): FrozenBatchNorm2d(256)
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(1024)
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
            (1): FrozenBatchNorm2d(1024)
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(256)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
          (bn2): FrozenBatchNorm2d(256)
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(1024)
          (relu): ReLU(inplace=True)
        )
```

```
        (2): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(256)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
          (bn2): FrozenBatchNorm2d(256)
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(1024)
          (relu): ReLU(inplace=True)
        )
        (3): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(256)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
          (bn2): FrozenBatchNorm2d(256)
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(1024)
          (relu): ReLU(inplace=True)
        )
        (4): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(256)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
          (bn2): FrozenBatchNorm2d(256)
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(1024)
          (relu): ReLU(inplace=True)
        )
        (5): Bottleneck(
          (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(256)
          (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
          (bn2): FrozenBatchNorm2d(256)
          (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(1024)
          (relu): ReLU(inplace=True)
        )
      )
      (layer4): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(512)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1),
bias=False)
          (bn2): FrozenBatchNorm2d(512)
          (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(2048)
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
            (1): FrozenBatchNorm2d(2048)
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(512)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
          (bn2): FrozenBatchNorm2d(512)
          (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(2048)
          (relu): ReLU(inplace=True)
```

```
        )
        (2): Bottleneck(
          (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(512)
          (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1),
bias=False)
          (bn2): FrozenBatchNorm2d(512)
          (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(2048)
          (relu): ReLU(inplace=True)
        )
      )
    )
    (fpn): FeaturePyramidNetwork(
      (inner_blocks): ModuleList(
        (0): Conv2d(256, 256, kernel_size=(1, 1), stride=(1, 1))
        (1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1))
        (2): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1))
        (3): Conv2d(2048, 256, kernel_size=(1, 1), stride=(1, 1))
      )
      (layer_blocks): ModuleList(
        (0): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      )
      (extra_blocks): LastLevelMaxPool()
    )
  )
  (rpn): RegionProposalNetwork(
    (anchor_generator): AnchorGenerator()
    (head): RPNHead(
      (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (cls_logits): Conv2d(256, 3, kernel_size=(1, 1), stride=(1, 1))
      (bbox_pred): Conv2d(256, 12, kernel_size=(1, 1), stride=(1, 1))
    )
  )
  (roi_heads): RoIHeads(
    (box_roi_pool): MultiScaleRoIAlign()
    (box_head): TwoMLPHead(
      (fc6): Linear(in_features=12544, out_features=1024, bias=True)
      (fc7): Linear(in_features=1024, out_features=1024, bias=True)
    )
    (box_predictor): FastRCNNPredictor(
      (cls_score): Linear(in_features=1024, out_features=8, bias=True)
      (bbox_pred): Linear(in_features=1024, out_features=32, bias=True)
    )
  )
)
```

labeled data(train 900개)로 모델 학습

In [11]:

```
num_epochs = 10
params = [p for p in model.parameters() if p.requires_grad]
optimizer = torch.optim.Adam(params, lr = 0.005,  weight_decay = 0.0005)
```

In [46]:

```
# 새로 학습 (start:0)
print('----------------------train start----------------------')
for epoch in range(num_epochs):
    start = time.time()
    model.train()
    i = 0
```

```
    epoch_loss = 0
    for imgs, annotations in data_loader:
        i += 1
        imgs = list(img.to(device) for img in imgs)
        annotations = [{k: v.to(device) for k,v in t.items()} for t in annotations]
        loss_dict = model(imgs, annotations)
        losses = sum(loss for loss in loss_dict.values())

        optimizer.zero_grad()
        losses.backward()
        optimizer.step()
        epoch_loss += losses
    print(f'epoch : {epoch + 1}, Loss : {epoch_loss}, time : {time.time() - start}')
torch.save(model.state_dict(), f'../Miso/weight/model_{num_epochs}.pt')
```

```
----------------------train start----------------------
epoch : 1, Loss : 184.4652099609375, time : 292.696973323822
epoch : 2, Loss : 133.4962158203125, time : 305.45523953437805
epoch : 3, Loss : 116.25265502929688, time : 305.6229157447815
epoch : 4, Loss : 107.01681518554688, time : 305.5948791503906
epoch : 5, Loss : 98.145751953125, time : 305.8539409637451
epoch : 6, Loss : 93.56787109375, time : 306.60982060432434
epoch : 7, Loss : 90.18324279785156, time : 306.8575441837311
epoch : 8, Loss : 86.6170883178711, time : 314.55700159072876
epoch : 9, Loss : 82.99738311767578, time : 308.4084484577179
epoch : 10, Loss : 80.02523040771484, time : 308.6641743183136
```

In [10]:

```
# 학습 과정을 반복 -> Train 함수 생성
def Train(start_epoch, end_epoch):
    print('----------------------train start----------------------')
    # 이전 가중치 불러오기
    model.load_state_dict(torch.load(f'../Miso/weight/model_{start_epoch}.pt'))
    for epoch in range(start_epoch, end_epoch):
        start = time.time()
        model.train()
        i = 0
        epoch_loss = 0
        for imgs, annotations in data_loader:
            i += 1
            imgs = list(img.to(device) for img in imgs)
            annotations = [{k: v.to(device) for k,v in t.items()} for t in annotatic
            loss_dict = model(imgs, annotations)
            losses = sum(loss for loss in loss_dict.values())

            optimizer.zero_grad()
            losses.backward()
            optimizer.step()
            epoch_loss += losses
        print(f'epoch : {epoch + 1}, Loss : {epoch_loss}, time : {time.time() - start
    torch.save(model.state_dict(), f'../Miso/weight/model_{end_epoch}.pt')
```

In [29]:

```
Train(10,40)
```

```
----------------------train start----------------------
epoch : 11, Loss : 78.08599853515625, time : 298.3371398448944
epoch : 12, Loss : 73.4039306640625, time : 309.33442068099976
epoch : 13, Loss : 70.32776641845703, time : 309.09239077568054
epoch : 14, Loss : 69.4464340209961, time : 309.0584738254547
epoch : 15, Loss : 68.42230224609375, time : 309.39530634880066
epoch : 16, Loss : 66.68819427490234, time : 309.711523771286
epoch : 17, Loss : 65.42156982421875, time : 309.3765642642975
epoch : 18, Loss : 64.35672760009766, time : 309.5744860172272
epoch : 19, Loss : 62.35205078125, time : 309.65450716018677
```

```
epoch : 20, Loss : 59.7096061706543, time : 309.80882596969604
epoch : 21, Loss : 59.13330078125, time : 309.62402153015137
epoch : 22, Loss : 58.9998779296875, time : 309.44889283180237
epoch : 23, Loss : 57.543785095214844, time : 309.47259640693665
epoch : 24, Loss : 60.20896530151367, time : 309.87808060646057
epoch : 25, Loss : 59.44280242919922, time : 309.7770435810089
epoch : 26, Loss : 57.681610107421875, time : 309.95213174819946
epoch : 27, Loss : 58.660003662109375, time : 310.0010414123535
epoch : 28, Loss : 57.827945709228516, time : 309.8183982372284
epoch : 29, Loss : 55.76863098144531, time : 309.97965812683105
epoch : 30, Loss : 52.535011291503906, time : 310.4970784187317
epoch : 31, Loss : 51.306514739990234, time : 311.0537631511688
epoch : 32, Loss : 49.14296340942383, time : 311.2446310520172
epoch : 33, Loss : 47.8657112121582, time : 310.87654423713684
epoch : 34, Loss : 48.96051025390625, time : 311.5384712219238
epoch : 35, Loss : 48.83477783203125, time : 312.4387664794922
epoch : 36, Loss : 48.07744216918945, time : 312.50558376312256
epoch : 37, Loss : 48.4630012512207, time : 312.8113830089569
epoch : 38, Loss : 48.66447830200195, time : 312.9918460845947
epoch : 39, Loss : 51.48432540893555, time : 313.1570498943329
epoch : 40, Loss : 50.22679901123047, time : 313.3181531429291
```

In [30]:
```
Train(40,70)
```

```
----------------------train start----------------------
epoch : 41, Loss : 46.886512756347656, time : 312.98372316360474
epoch : 42, Loss : 45.07920455932617, time : 313.6878180503845
epoch : 43, Loss : 44.99940872192383, time : 313.2798149585724
epoch : 44, Loss : 46.298919677734375, time : 313.35889959335327
epoch : 45, Loss : 47.991573333740234, time : 313.4809250831604
epoch : 46, Loss : 48.51558303833008, time : 313.3449685573578
epoch : 47, Loss : 47.643795013427734, time : 313.21340465545654
epoch : 48, Loss : 46.312660217285156, time : 313.36291122436523
epoch : 49, Loss : 46.07572555541992, time : 313.35382986068726
epoch : 50, Loss : 45.76484298706055, time : 313.89527463912964
epoch : 51, Loss : 46.81088638305664, time : 313.49887108802795
epoch : 52, Loss : 45.12034606933594, time : 313.82265424728394
epoch : 53, Loss : 44.5145263671875, time : 314.01485800743103
epoch : 54, Loss : 41.797332763671875, time : 314.00806403160095
epoch : 55, Loss : 42.05780029296875, time : 313.73097109794617
epoch : 56, Loss : 43.16927719116211, time : 313.8496768474579
epoch : 57, Loss : 44.16347885131836, time : 313.9460780620575
epoch : 58, Loss : 45.15998458862305, time : 313.8107635974884
epoch : 59, Loss : 43.57608413696289, time : 314.08036160469055
epoch : 60, Loss : 42.62240219116211, time : 314.68021297454834
epoch : 61, Loss : 42.675716400146484, time : 313.81993651390076
epoch : 62, Loss : 43.08955001831055, time : 314.0584890842438
epoch : 63, Loss : 41.371063232421875, time : 314.1108994483948
epoch : 64, Loss : 41.349578857421875, time : 313.8243193626404
epoch : 65, Loss : 41.28604507446289, time : 313.70141768455505
epoch : 66, Loss : 40.889957427978516, time : 314.12674283981323
epoch : 67, Loss : 40.27585983276367, time : 313.8127603530884
epoch : 68, Loss : 40.802635192871094, time : 313.822603225708
epoch : 69, Loss : 42.174198150634766, time : 313.59251594543457
epoch : 70, Loss : 42.8463020324707, time : 313.87530159950256
```

In [31]:
```
Train(70,100)  # weight/model_100.pt가 성능 좋음
```

```
----------------------train start----------------------
epoch : 71, Loss : 41.44423294067383, time : 314.0329167842865
epoch : 72, Loss : 41.07215118408203, time : 313.7328152656555
epoch : 73, Loss : 42.21272659301758, time : 313.5039610862732
epoch : 74, Loss : 42.957183837890625, time : 312.5663480758667
epoch : 75, Loss : 43.19308090209961, time : 311.84142994880676
```

```
epoch : 76, Loss : 40.83940887451172, time : 311.3276171684265
epoch : 77, Loss : 40.42216110229492, time : 324.3858106136322
epoch : 78, Loss : 39.57860565185547, time : 311.3430845737457
epoch : 79, Loss : 38.749114990234375, time : 312.3819327354431
epoch : 80, Loss : 39.23604202270508, time : 311.1923122406006
epoch : 81, Loss : 39.7637825012207, time : 311.17557168006897
epoch : 82, Loss : 39.48535919189453, time : 311.40045142173767
epoch : 83, Loss : 39.03528594970703, time : 311.3932249546051
epoch : 84, Loss : 38.23387145996094, time : 311.3597774505615
epoch : 85, Loss : 37.65389633178711, time : 311.3727607727051
epoch : 86, Loss : 37.667579650878906, time : 311.49195075035095
epoch : 87, Loss : 37.88323211669922, time : 311.6224546432495
epoch : 88, Loss : 37.414527893066406, time : 311.2522203922272
epoch : 89, Loss : 37.816654205322266, time : 311.19093585014343
epoch : 90, Loss : 37.144447326660156, time : 311.27111625671387
epoch : 91, Loss : 38.285701751708984, time : 311.11113929748535
epoch : 92, Loss : 39.58122253417969, time : 311.13230180740356
epoch : 93, Loss : 39.671722412109375, time : 311.5430042743683
epoch : 94, Loss : 41.34640121459961, time : 311.46081137657166
epoch : 95, Loss : 40.30497741699219, time : 311.0814538002014
epoch : 96, Loss : 38.55747604370117, time : 310.6296863555908
epoch : 97, Loss : 37.35264587402344, time : 310.2290041446686
epoch : 98, Loss : 36.97559356689453, time : 309.8152811527252
epoch : 99, Loss : 36.35737228393555, time : 309.5081362724304
epoch : 100, Loss : 35.499332427978516, time : 319.5265865325928
```

In [38]:
```
Train(100,140)
```

```
----------------------train start----------------------
epoch : 101, Loss : 35.654319763183594, time : 296.6997148990631
epoch : 102, Loss : 35.185333251953125, time : 305.04630160331726
epoch : 103, Loss : 35.73046875, time : 304.64369082450867
epoch : 104, Loss : 37.23564529418945, time : 304.7702009677887
epoch : 105, Loss : 36.554298400878906, time : 304.8173894882202
epoch : 106, Loss : 37.21649169921875, time : 305.03918409347534
epoch : 107, Loss : 36.29158020019531, time : 304.8473255634308
epoch : 108, Loss : 34.52984619140625, time : 304.8507556915283
epoch : 109, Loss : 34.126461029052734, time : 304.9978744983673
epoch : 110, Loss : 34.528141021728516, time : 304.76401376724243
epoch : 111, Loss : 34.87495040893555, time : 304.80209136009216
epoch : 112, Loss : 36.3445930480957, time : 304.89707350730896
epoch : 113, Loss : 36.329654693603516, time : 304.6840515136719
epoch : 114, Loss : 36.42739486694336, time : 304.56305480003357
epoch : 115, Loss : 34.99055862426758, time : 304.960661649704
epoch : 116, Loss : 35.01762390136719, time : 304.90438318252563
epoch : 117, Loss : 34.77829360961914, time : 304.8390271663666
epoch : 118, Loss : 35.224327087402344, time : 304.90428256988525
epoch : 119, Loss : 34.473609924316406, time : 305.12508893013
epoch : 120, Loss : 34.469947814941406, time : 304.933979511261
epoch : 121, Loss : 34.88443374633789, time : 305.13028359413147
epoch : 122, Loss : 36.06177520751953, time : 305.1531512737274
epoch : 123, Loss : 36.98336410522461, time : 304.8476767539978
epoch : 124, Loss : 38.90148162841797, time : 305.0272672176361
epoch : 125, Loss : 36.912532806396484, time : 304.8921477794647
epoch : 126, Loss : 35.91579818725586, time : 304.8470346927643
epoch : 127, Loss : 36.904903411865234, time : 304.9674708843231
epoch : 128, Loss : 36.447959899902344, time : 304.9351942539215
epoch : 129, Loss : 35.97067642211914, time : 304.85839200019836
epoch : 130, Loss : 35.51709747314453, time : 318.1055631637573
epoch : 131, Loss : 35.89180374145508, time : 304.9797818660736
epoch : 132, Loss : 35.38075637817383, time : 304.8183386325836
epoch : 133, Loss : 35.30976486206055, time : 304.98489594459534
epoch : 134, Loss : 36.277774810791016, time : 304.8892915248871
epoch : 135, Loss : 37.57420349121094, time : 305.2287263870239
```

```
epoch : 136, Loss : 37.0010871887207, time : 305.1114556789398
epoch : 138, Loss : 34.92347717285156, time : 305.32048439979553
epoch : 139, Loss : 34.03165054321289, time : 305.7618935108185
epoch : 140, Loss : 34.27888107299805, time : 305.4651589393616
```

In [39]:
```
Train(140,170)
```

```
----------------------train start-----------------------
epoch : 141, Loss : 34.311866760253906, time : 304.9764678478241
epoch : 142, Loss : 34.66252899169922, time : 305.83146715164185
epoch : 143, Loss : 35.58416748046875, time : 305.1754677295685
epoch : 144, Loss : 35.663822174072266, time : 305.3140480518341
epoch : 145, Loss : 34.394718170166016, time : 305.1913990974426
epoch : 146, Loss : 35.37107467651367, time : 305.4824016094208
epoch : 147, Loss : 36.77742385864258, time : 305.49629640579224
epoch : 148, Loss : 34.299163818359375, time : 305.75249576568604
epoch : 149, Loss : 33.95438003540039, time : 305.60509991645813
epoch : 150, Loss : 33.88557815551758, time : 314.51961946487427
epoch : 151, Loss : 33.50310516357422, time : 313.3665699958801
epoch : 152, Loss : 34.11591339111328, time : 305.5282037258148
epoch : 153, Loss : 33.62476348876953, time : 305.458616733551
epoch : 154, Loss : 32.59880447387695, time : 305.3469545841217
epoch : 155, Loss : 33.43467712402344, time : 307.69419288635254
epoch : 156, Loss : 32.6848030090332, time : 306.1480026245117
epoch : 157, Loss : 33.13824462890625, time : 305.49190855026245
epoch : 158, Loss : 33.933433532714844, time : 305.6342885494232
epoch : 159, Loss : 34.814117431640625, time : 305.47767972946167
epoch : 160, Loss : 35.94673156738281, time : 305.994357585907
epoch : 161, Loss : 35.238731384277344, time : 318.0277473926544
epoch : 162, Loss : 34.63956832885742, time : 311.3616302013397
epoch : 163, Loss : 35.70078659057617, time : 305.57619881629944
epoch : 164, Loss : 35.21100997924805, time : 307.9537501335144
epoch : 165, Loss : 34.77338409423828, time : 305.6760790348053
epoch : 166, Loss : 34.150936126708984, time : 305.70644998550415
epoch : 167, Loss : 33.712989807128906, time : 305.58053755760193
epoch : 168, Loss : 34.40760040283203, time : 306.36349534988403
epoch : 169, Loss : 34.60575485229492, time : 305.7107946872711
epoch : 170, Loss : 34.21036148071289, time : 305.6560573577881
```

In [40]:
```
Train(170,200)
```

```
----------------------train start-----------------------
epoch : 171, Loss : 33.48416519165039, time : 305.67923951148987
epoch : 172, Loss : 34.270172119140625, time : 305.59146785736084
epoch : 173, Loss : 35.98695373535156, time : 305.4382891654968
epoch : 174, Loss : 35.328941345214844, time : 308.4758508205414
epoch : 175, Loss : 35.657936096191406, time : 307.8103082180023
epoch : 176, Loss : 33.6329345703125, time : 305.9856741428375
epoch : 177, Loss : 33.09095764160156, time : 306.6250960826874
epoch : 178, Loss : 32.7260627746582, time : 306.9916408061981
epoch : 179, Loss : 32.34452438354492, time : 306.37062525749207
epoch : 180, Loss : 32.746883392333984, time : 305.35312509536743
epoch : 181, Loss : 33.69898223876953, time : 305.73690009117126
epoch : 182, Loss : 32.02189636230469, time : 305.9854555130005
epoch : 183, Loss : 31.623329162597656, time : 305.60191106796265
epoch : 184, Loss : 32.82634353637695, time : 306.0196177959442
epoch : 185, Loss : 33.09264373779297, time : 305.91721081733704
epoch : 186, Loss : 33.3814697265625, time : 305.82724714279175
epoch : 187, Loss : 34.27228546142578, time : 305.87115454673767
epoch : 188, Loss : 35.410221099853516, time : 305.91414642333984
epoch : 189, Loss : 35.8953971862793, time : 305.63165187835693
epoch : 190, Loss : 34.83549118041992, time : 310.95782828330994
epoch : 191, Loss : 34.147247314453125, time : 306.73854804039
epoch : 192, Loss : 32.12699508666992, time : 320.55077719688416
```

```
epoch : 193, Loss : 32.13589859008789, time : 305.3343138694763
epoch : 194, Loss : 31.81453514099121, time : 305.98097825050354
epoch : 195, Loss : 32.720481872558594, time : 306.11534547805786
epoch : 196, Loss : 32.20090866088867, time : 305.9428324699402
epoch : 197, Loss : 32.64332962036133, time : 306.3147692680359
epoch : 198, Loss : 32.4322395324707, time : 306.12068247795105
epoch : 199, Loss : 31.82786750793457, time : 305.97158646583557
epoch : 200, Loss : 31.562959671020508, time : 305.9004907608032
```

모델 weight 로드

In [12]:
```
model.load_state_dict(torch.load(f'../Miso/weight/model_100.pt'))
```

Out[12]:
```
<All keys matched successfully>
```

예측

In [13]:
```
def make_prediction(model, img, threshold):
    model.eval()
    preds = model(img)
    for id in range(len(preds)):
        idx_list = []

        for idx, score in enumerate(preds[id]['scores']):  # 한 이미지 내에 검출된 객
            if score > threshold:  # 신뢰도가 threshold보다 높은것만 idx_list에 저장
                idx_list.append(idx)

        # thr보다 높은 객체들만 boxes, labels, scores 정보 추출해서 덮어쓰기로 저장
        preds[id]['boxes'] = preds[id]['boxes'][idx_list]
        preds[id]['labels'] = preds[id]['labels'][idx_list]-1  # label : 0-6
        preds[id]['scores'] = preds[id]['scores'][idx_list]

    return preds
```

labeled data로 학습한 모델 결과

In [14]:
```
with torch.no_grad():
    for imgs, annotations in test_data_loader:
        im = imgs
        imgs = list(img.to(device) for img in imgs)

        pred = make_prediction(model, imgs, 0.2) # threshold
        print(pred)
        break   # test_data_loader의 첫 번째 배치만 결과 출력
```

```
[{'boxes': tensor([[ 703.4944,   59.9619,  725.3139,   99.6871],
        [ 774.2667,   55.1289,  796.3938,   99.5487],
        [ 880.0817,  209.7026, 1001.0000,  321.6533],
        [ 554.2569,   20.5485,  590.6553,   49.1402],
        [ 312.9951,  132.5156,  339.6205,  185.6905],
        [ 775.7050,   69.8275,  794.6923,  108.3002],
        [ 449.7556,   18.2737,  483.4883,   32.0022],
        [ 752.9364,   19.0609,  782.9688,   44.3429],
        [ 956.2925,   91.0808,  991.5442,  129.3998],
        [ 485.5267,   18.3976,  520.0612,   33.1010],
        [ 769.5856,   21.2229,  811.9795,   49.7759],
        [ 312.1575,  145.5819,  337.2722,  194.1991],
        [ 709.4665,  296.4266,  855.6778,  434.6549],
        [ 822.0977,   48.4249,  878.2283,   94.5753],
        [ 866.3963,   40.7793,  886.5568,   65.0910],
        [ 796.1738,   42.0485,  839.4514,   79.5353],
```

```
        [ 705.5055,   74.3117,  722.3051,  102.3828],
        [ 868.8712,   32.0100,  946.5863,   65.4043],
        [ 827.4374,   80.7260,  838.3604,  101.3921],
        [ 477.8173,   19.3210,  488.2294,   35.6006],
        [ 514.8702,   18.9247,  527.0136,   33.8235],
        [ 557.6417,   19.4193,  587.9046,   31.1005],
        [ 515.4282,   19.2760,  526.1896,   35.8974],
        [ 773.1207,   47.7967,  792.6917,   86.2238],
        [ 777.9705,   23.8157,  813.5848,   56.7420],
        [ 816.8560,   52.9844,  830.9999,   89.8972],
        [ 761.9490,   22.5953,  779.9617,   46.7803],
        [ 516.5008,   18.8144,  526.4512,   33.2618],
        [ 869.4634,   33.0137,  946.2191,   66.4214],
        [ 764.3045,   24.4697,  776.0478,   48.1999],
        [ 774.7969,   57.6384,  790.8819,   79.0378],
        [ 553.9160,   18.9423,  578.7875,   28.8368]], device='cuda:0'), 'labels': ten
sor([0, 0, 1, 1, 0, 6, 1, 1, 5, 1, 1, 6, 1, 1, 1, 1, 6, 1, 6, 1, 1, 1, 6, 0,
        3, 0, 1, 0, 3, 0, 6, 1], device='cuda:0'), 'scores': tensor([0.9992, 0.9981,
0.9974, 0.9972, 0.9970, 0.9969, 0.9951, 0.9930, 0.9924,
        0.9920, 0.9903, 0.9899, 0.9897, 0.9854, 0.9786, 0.9756, 0.9632, 0.9591,
        0.9361, 0.7834, 0.7581, 0.6922, 0.6107, 0.5900, 0.4690, 0.4012, 0.3918,
        0.3524, 0.3431, 0.3259, 0.2421, 0.2239], device='cuda:0')}, {'boxes': tensor
([[5.4197e+02, 1.0577e+02, 5.6161e+02, 1.1947e+02],
        [7.1845e+02, 2.1191e+02, 7.3863e+02, 2.7065e+02],
        [5.2045e+02, 9.2578e+01, 5.3110e+02, 1.0260e+02],
        [2.9218e+02, 3.7259e+02, 4.3478e+02, 5.0995e+02],
        [5.5617e+02, 4.6917e+02, 7.5163e+02, 6.1100e+02],
        [8.7849e+02, 4.2869e+02, 1.0010e+03, 5.3517e+02],
        [4.9921e+02, 2.9471e+02, 6.0934e+02, 4.3544e+02],
        [7.4715e+02, 1.5721e+02, 7.5870e+02, 1.8373e+02],
        [5.9285e+02, 1.2217e+02, 6.1037e+02, 1.4023e+02],
        [6.3336e+02, 1.3224e+02, 6.4443e+02, 1.5227e+02],
        [5.2902e+02, 9.2152e+01, 5.3680e+02, 1.0276e+02],
        [6.2791e+02, 1.4445e+02, 6.4024e+02, 1.7181e+02],
        [5.6576e+02, 1.1777e+02, 5.9077e+02, 1.4072e+02],
        [5.1024e+02, 1.2745e+02, 5.1937e+02, 1.4700e+02],
        [5.7948e+02, 1.3330e+02, 6.0562e+02, 1.5745e+02],
        [5.0977e+02, 1.3413e+02, 5.1960e+02, 1.4995e+02],
        [7.4157e+02, 1.5558e+02, 7.5359e+02, 1.8599e+02],
        [5.3116e+02, 9.6498e+01, 5.4247e+02, 1.0877e+02],
        [9.1798e+02, 1.9529e+02, 9.6938e+02, 2.3538e+02],
        [5.4863e+02, 1.1975e+02, 5.6568e+02, 1.3997e+02],
        [4.0260e+02, 1.5022e+02, 4.5583e+02, 1.9512e+02],
        [5.8672e+02, 1.3099e+02, 6.0102e+02, 1.5407e+02],
        [5.3045e+02, 9.3831e+01, 5.4060e+02, 1.0556e+02],
        [5.1360e+02, 9.9019e+01, 5.2321e+02, 1.1653e+02],
        [7.4070e+02, 1.5759e+02, 7.6035e+02, 1.8620e+02],
        [4.0116e+02, 1.4932e+02, 4.5792e+02, 1.9712e+02],
        [6.4023e+02, 1.3920e+02, 6.4917e+02, 1.6463e+02],
        [5.7531e+02, 1.2662e+02, 5.9121e+02, 1.4342e+02],
        [4.8444e+02, 1.0668e+02, 5.0660e+02, 1.3146e+02],
        [5.2471e+02, 9.2283e+01, 5.3402e+02, 1.0279e+02],
        [5.1279e+02, 9.8224e+01, 5.2363e+02, 1.1600e+02],
        [5.7827e+02, 1.2870e+02, 5.9041e+02, 1.4917e+02],
        [6.0908e+02, 1.2017e+02, 6.1664e+02, 1.3806e+02],
        [6.2300e+02, 1.4647e+02, 6.4808e+02, 1.7066e+02],
        [5.1409e+02, 9.7205e+01, 5.2202e+02, 1.1507e+02],
        [5.3196e+02, 9.9069e+01, 5.4291e+02, 1.1621e+02],
        [9.5719e-02, 7.0396e+01, 4.2164e+01, 1.0039e+02],
        [4.8407e+02, 1.0628e+02, 5.0562e+02, 1.3074e+02],
        [6.3641e+02, 1.3377e+02, 6.4692e+02, 1.5751e+02],
        [5.6980e+02, 1.1649e+02, 5.8292e+02, 1.4202e+02],
        [5.3168e+02, 9.9477e+01, 5.4342e+02, 1.1550e+02],
        [5.6634e+02, 1.1749e+02, 5.8834e+02, 1.4099e+02],
        [5.4163e+02, 1.0269e+02, 5.5506e+02, 1.1081e+02],
```

```
        [9.1896e+02, 1.9377e+02, 9.6966e+02, 2.3440e+02],
        [4.8400e+02, 1.0724e+02, 5.0653e+02, 1.3046e+02]], device='cuda:0'), 'labels':
tensor([1, 0, 1, 1, 1, 1, 3, 0, 1, 0, 1, 0, 1, 0, 5, 6, 0, 1, 4, 5, 5, 0, 1, 5,
        6, 3, 0, 1, 5, 1, 1, 0, 0, 6, 0, 5, 1, 4, 0, 0, 1, 5, 1, 1, 2],
       device='cuda:0'), 'scores': tensor([0.9992, 0.9989, 0.9988, 0.9983, 0.9976, 0.9
975, 0.9963, 0.9800, 0.9665,
        0.9583, 0.9436, 0.9278, 0.8598, 0.8566, 0.8348, 0.8211, 0.7984, 0.7879,
        0.7634, 0.7424, 0.7242, 0.7047, 0.6909, 0.6675, 0.6459, 0.5962, 0.5890,
        0.5556, 0.5426, 0.5083, 0.4890, 0.4603, 0.3843, 0.3843, 0.3591, 0.3570,
        0.3441, 0.2920, 0.2752, 0.2595, 0.2593, 0.2269, 0.2242, 0.2215, 0.2061],
       device='cuda:0')}]
```

In [15]:

```
def plot_image_from_output(img, annotation):
    img = img.cpu().permute(1,2,0)  # 0위치:2에 있던게 옴, 1위치:0, 2위치:1 으로 차원

    fig, ax = plt.subplots(1)
    ax.imshow(img)  # (M,N,3) : M=row, N=col

    for idx in range(len(annotation["boxes"])):
        xmin, ymin, xmax, ymax = annotation["boxes"][idx]
        xmin = xmin.cpu().data.numpy()
        ymin = ymin.cpu().data.numpy()
        xmax = xmax.cpu().data.numpy()
        ymax = ymax.cpu().data.numpy()

        #print("xmin,ymin,xmax,ymax :", xmin,ymin,xmax,ymax)
        rect = patches.Rectangle((xmin,ymin),(xmax-xmin),(ymax-ymin), linewidth=1, ed

        ax.add_patch(rect)
    plt.show()
```

In [16]:

```
_idx = 0
print("Target :", annotations[_idx]['labels'])  # Ground Truth 값 (-1 해야됨)
plot_image_from_output(imgs[_idx], annotations[_idx])  # imgs : test_data_loader에서 b

print("Prediction :", pred[_idx]['labels'])
plot_image_from_output(imgs[_idx], pred[_idx])
```

```
Target : tensor([1, 1, 1, 1, 1, 2, 2, 7, 7, 7, 2, 2, 2, 2, 2, 2, 2, 2, 7, 7, 2, 7])
```


```
Prediction : tensor([0, 0, 1, 1, 0, 6, 1, 1, 5, 1, 1, 6, 1, 1, 1, 1, 6, 1, 6, 1, 1, 1,
6, 0,
        3, 0, 1, 0, 3, 0, 6, 1], device='cuda:0')
```

Unlabeled data(19000개)의 annotation 추론 -> txt파일로 저장

In [17]:
```
# labeled data로 학습시킨 모델 weight 로드
model.load_state_dict(torch.load(f'../Miso/weight/model_100.pt'))
```

Out[17]:
```
<All keys matched successfully>
```

In [18]:
```
from PIL import ImageFile
ImageFile.LOAD_TRUNCATED_IMAGES = True
```

In [14]:
```
pred_list = []
with torch.no_grad():
    for imgs, annotations in tqdm(ul_data_loader):
        imgs = list(img.to(device) for img in imgs)

        pred = make_prediction(model, imgs, 0.2) # threshold   # imgs: [C, H, W]
        pred_list.append(pred)
```

```
100%|██████████| 4750/4750 [1:04:41<00:00,  1.22it/s]
```

In [16]:
```
df_size = pd.DataFrame(columns=['H','W'])
h = []
w = []

for file_name in tqdm(unlabeled_list):
    ul_img_path = './unlabeled_data/' + file_name
    img = Image.open(ul_img_path) #.convert('RGB')

    h.append(img.size[1])   # H
    w.append(img.size[0])   # W

df_size['H'] = h
df_size['W'] = w
df_size
```

```
100%|██████████| 19000/19000 [00:08<00:00, 2329.22it/s]
```

Out[16]:

	H	W
0	810	1920
1	611	1001
2	611	1001

	H	W
3	611	1001
4	611	1001
...	...	...
18995	1350	982
18996	810	1920
18997	1350	982
18998	1350	982
18999	800	1012

19000 rows × 2 columns

In [17]:
```
import csv
small_num = 0; img_num = 0

for prd in tqdm(pred_list): # pred_list길이: 4750, prd길이: 4
    for pr in prd:  # prd길이: 4
        ul_txt = unlabeled_list[img_num][:-3] + 'txt'
        W = df_size.iloc[img_num]['W']
        H = df_size.iloc[img_num]['H']
        obj_num = len(pr['boxes'])

        annots_list = []

        for i in range(obj_num):
            annot = []
            # bbox
            xmin = float(pr['boxes'][i][0])/W
            ymin = float(pr['boxes'][i][1])/H
            xmax = float(pr['boxes'][i][2])/W
            ymax = float(pr['boxes'][i][3])/H

            if (xmax - xmin)*(ymax - ymin) > 0.0001:   # 조건: normalized area of bbo
                annot.append(xmin)
                annot.append(ymin)
                annot.append(xmax)
                annot.append(ymax)
            else:
                small_num += 1
                continue

            # class_id
            annot.insert(0, int(pr['labels'][i]))

            ## append to annots_list
            annots_list.append(annot)

        ## save annots_list to txt file  # bbox를 (0-1사이 비율)로 저장
        with open('./unlabeled_annots_rate/' + ul_txt, 'w', newline = '') as f:
            write = csv.writer(f)
            write.writerows(annots_list)

        img_num += 1
print(small_num)
```

100%|██████████| 4750/4750 [29:12<00:00, 2.71it/s]

```
216
```

In [19]:
```
# 개수 확인
ul_txt_list = os.listdir('./unlabeled_annots_rate/')
len(ul_txt_list)
```

Out[19]:
```
19000
```

예측 결과 평가

(labeled data에서 분리한 100개 test 데이터)

In [ ]:
```
from tqdm import tqdm

labels = []
preds_adj_all = []
annot_all = []

for im, annot in tqdm(test_data_loader, position = 0, leave = True):
    im = list(img.to(device) for img in im)

    for t in annot:
        for i in range(len(t['labels'])):
            t['labels'][i] = t['labels'][i] - 1
        labels += t['labels']

    with torch.no_grad():
        preds_adj = make_prediction(model, im, 0)
        preds_adj = [{k: v.to(torch.device('cpu')) for k, v in t.items()} for t in p
        preds_adj_all.append(preds_adj)
        annot_all.append(annot)
```

In [44]:
```
import utils_ObjectDetection as utils

sample_metrics = []
for batch_i in range(len(preds_adj_all)):
    sample_metrics += utils.get_batch_statistics(preds_adj_all[batch_i],
                                                 annot_all[batch_i], iou_threshold = (
true_positives, pred_scores, pred_labels = [torch.cat(x,0) for x in list(zip(*sample_

#precision, recall, AP, f1, ap_class = utils.ap_per_class(true_positives, pred_scores,
_, _, AP, _, _ = utils.ap_per_class(true_positives, pred_scores, pred_labels, torch.t
mAP = torch.mean(AP)

print(f'mAP : {mAP}')
print(f'AP : {AP}')
# threshold = 0.6 : mAP : 0.574
# threshold = 0.5 : mAP : 0.586
# threshold = 0.4 : mAP : 0.591
# threshold = 0.3 : mAP : 0.597
# threshold = 0.1 : mAP : 0.616
# threshold = 0.0 : mAP : 0.621
```

```
mAP : 0.6208213365001757
AP : tensor([0.5808, 0.7780, 0.6428, 0.6583, 0.7625, 0.3822, 0.5412],
       dtype=torch.float64)
```

In [ ]:

In [ ]:

In [ ]:

In [ ]: