```
#In case google denies access for downloading the file, you can try by installing the l
pip install -U --no-cache-dir gdown --pre
```

```
  File "<ipython-input-2-336916444db0>", line 2
    pip install -U --no-cache-dir gdown --pre
                ^
SyntaxError: invalid syntax
```

SEARCH STACK OVERFLOW

```
#Download the dataset
#Shared file link https://drive.google.com/file/d/1WfdEh2CyoV-cSNFyncKvLYhmD0rv2fFN/vie
#Alternatively you can download to your local computer an upload back to colab

!gdown --id '1WfdEh2CyoV-cSNFyncKvLYhmD0rv2fFN' --output acid_sub
```

```
Downloading...
From: https://drive.google.com/uc?id=1WfdEh2CyoV-cSNFyncKvLYhmD0rv2fFN
To: /content/acid_sub
1.25GB [00:10, 124MB/s]
```

# Object Detection tutorial

For this tutorial, we will be finetuning a pre-trained Faster R-CNN model in the ACID dataset with 3 classes. It contains 3929 images with 8000+ instances of construction equipment, and we will use it to illustrate how to use the new features in torchvision in order to train an object detection model on a custom dataset.

First, we need to install `pycocotools`. This library will be used for computing the evaluation metrics following the COCO metric for intersection over union.

```
%%shell

pip install cython
# Install pycocotools, the version by default in Colab
# has a bug fixed in https://github.com/cocodataset/cocoapi/pull/354
pip install -U 'git+https://github.com/cocodataset/cocoapi.git#subdirectory=PythonAPI'
```

```
Requirement already satisfied: cython in /usr/local/lib/python3.7/dist-packages (0
Collecting git+https://github.com/cocodataset/cocoapi.git#subdirectory=PythonAPI
  Cloning https://github.com/cocodataset/cocoapi.git to /tmp/pip-req-build-_0c2mak
  Running command git clone -q https://github.com/cocodataset/cocoapi.git /tmp/pip
Requirement already satisfied, skipping upgrade: setuptools>=18.0 in /usr/local/li
Requirement already satisfied, skipping upgrade: cython>=0.27.3 in /usr/local/lib/
Requirement already satisfied, skipping upgrade: matplotlib>=2.1.0 in /usr/local/l
Requirement already satisfied, skipping upgrade: cycler>=0.10 in /usr/local/lib/py
Requirement already satisfied, skipping upgrade: python-dateutil>=2.1 in /usr/loca
Requirement already satisfied, skipping upgrade: kiwisolver>=1.0.1 in /usr/local/l
Requirement already satisfied, skipping upgrade: numpy>=1.11 in /usr/local/lib/pyt
Requirement already satisfied, skipping upgrade: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,
Requirement already satisfied, skipping upgrade: six in /usr/local/lib/python3.7/d
Building wheels for collected packages: pycocotools
  Building wheel for pycocotools (setup.py) ... done
  Created wheel for pycocotools: filename=pycocotools-2.0-cp37-cp37m-linux_x86_64.
```

```
  Stored in directory: /tmp/pip-ephem-wheel-cache-vpdmakm0/wheels/90/51/41/646daf4
Successfully built pycocotools
Installing collected packages: pycocotools
  Found existing installation: pycocotools 2.0.2
    Uninstalling pycocotools-2.0.2:
      Successfully uninstalled pycocotools-2.0.2
Successfully installed pycocotools-2.0
```

## Defining the Dataset

The [torchvision reference scripts for training object detection, instance segmentation and person keypoint detection] allows for easily supporting adding new custom datasets. The dataset should inherit from the standard `torch.utils.data.Dataset` class, and implement `__len__` and `__getitem__`.

The only specificity that we require is that the dataset `__getitem__` should return:

- image: a PIL Image of size (H, W)
- target: a dict containing the following fields
  - `boxes` (`FloatTensor[N, 4]`): the coordinates of the `N` bounding boxes in `[x0, y0, x1, y1]` format, ranging from `0` to `W` and `0` to `H`
  - `labels` (`Int64Tensor[N]`): the label for each bounding box
  - `image_id` (`Int64Tensor[1]`): an image identifier. It should be unique between all the images in the dataset, and is used during evaluation
  - `area` (`Tensor[N]`): The area of the bounding box. This is used during evaluation with the COCO metric, to separate the metric scores between small, medium and large boxes.
  - `iscrowd` (`UInt8Tensor[N]`): instances with `iscrowd=True` will be ignored during evaluation.
  - (optionally) `masks` (`UInt8Tensor[N, H, W]`): The segmentation masks for each one of the objects
  - (optionally) `keypoints` (`FloatTensor[N, K, 3]`): For each one of the `N` objects, it contains the `K` keypoints in `[x, y, visibility]` format, defining the object. `visibility=0` means that the keypoint is not visible. Note that for data augmentation, the notion of flipping a keypoint is dependent on the data representation, and you should probably adapt `references/detection/transforms.py` for your new keypoint representation

If your model returns the above methods, they will make it work for both training and evaluation, and will use the evaluation scripts from pycocotools.

Additionally, if you want to use aspect ratio grouping during training (so that each batch only contains images with similar aspect ratio), then it is recommended to also implement a `get_height_and_width` method, which returns the height and the width of the image. If this method is not provided, we query all elements of the dataset via `__getitem__` , which loads the image in memory and is slower than if a custom method is provided.

## Writing a custom dataset for ACID

Let's write a dataset for the ACID dataset.

First, let's extract the data, present in a zip file

```
!unzip acid_sub
```

```
  inflating: 3classes/08686.jpg
  inflating: 3classes/08688.jpg
  inflating: 3classes/08689.jpg
  inflating: 3classes/08690.jpg
  inflating: 3classes/08695.jpg
  inflating: 3classes/08696.jpg
  inflating: 3classes/08699.jpg
  inflating: 3classes/08700.jpg
  inflating: 3classes/08701.jpg
  inflating: 3classes/08703.jpg
  inflating: 3classes/08705.jpg
  inflating: 3classes/08707.jpg
  inflating: 3classes/08713.jpg
  inflating: 3classes/08714.jpg
  inflating: 3classes/08717.jpg

  inflating: 3classes/08720.jpg
  inflating: 3classes/08722.jpg
  inflating: 3classes/08723.jpg
  inflating: 3classes/08725.jpg
  inflating: 3classes/08726.jpg
  inflating: 3classes/08727.jpg
  inflating: 3classes/08728.jpg
  inflating: 3classes/08731.jpg
  inflating: 3classes/08733.jpg
  inflating: 3classes/08737.jpg
  inflating: 3classes/08745.jpg
  inflating: 3classes/08746.jpg
  inflating: 3classes/08758.jpg
  inflating: 3classes/08759.jpg
  inflating: 3classes/08760.jpg
  inflating: 3classes/08763.jpg
  inflating: 3classes/08768.jpg
  inflating: 3classes/08770.jpg
  inflating: 3classes/08772.jpg
  inflating: 3classes/08773.jpg
  inflating: 3classes/08777.jpg
  inflating: 3classes/08778.jpg
  inflating: 3classes/08788.jpg
  inflating: 3classes/08791.jpg
  inflating: 3classes/08793.jpg
  inflating: 3classes/08795.jpg
  inflating: 3classes/08796.jpg
  inflating: 3classes/08797.jpg
  inflating: 3classes/08799.jpg
  inflating: 3classes/08805.jpg
  inflating: 3classes/08810.jpg
  inflating: 3classes/08811.jpg
  inflating: 3classes/08812.jpg
  inflating: 3classes/08816.jpg
  inflating: 3classes/08818.jpg
  inflating: 3classes/08828.jpg
  inflating: 3classes/08831.jpg
  inflating: 3classes/08832.jpg
  inflating: 3classes/08833.jpg
  inflating: 3classes/08836.jpg
  inflating: 3classes/08840.jpg
  inflating: 3classes/08841.jpg
  inflating: 3classes/08842.jpg
```

```
 inflating: 3classes/00042.jpg
 inflating: 3classes/08846.jpg
```

Let's have a look at the dataset and how it is layed down.

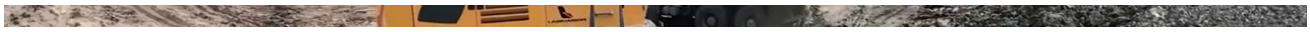
The data is structured as follows

```
 3classes/

     00001.jpg
     00002.jpg
     ....
     ....
     3classes.csv
     3classes.json
```

Here is one example of an image in the dataset

```
from PIL import Image
import os
path = '/content/3classes/'
Image.open(os.path.join(path,'10031.jpg'))
#Image.open(path)
```


So each image has a corresponding bounding box. Let's write a `torch.utils.data.Dataset` class for this dataset.

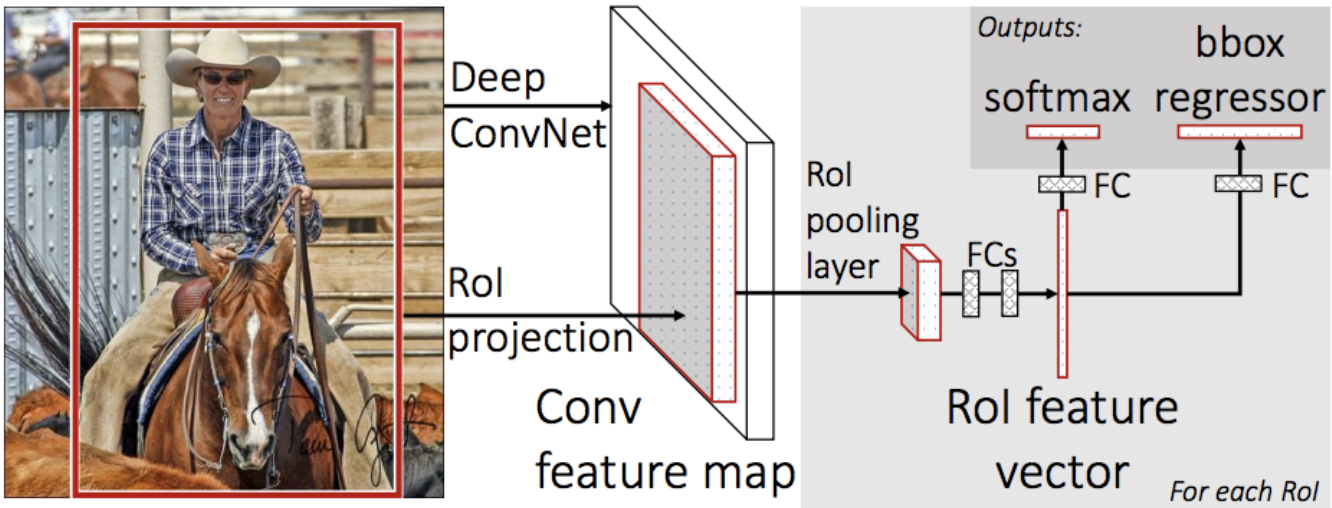


```
import os
import numpy as np
import torch
import torch.utils.data
from PIL import Image
from pycocotools.coco import COCO


class ACID_dataset(torch.utils.data.Dataset):
    def __init__(self, root, annotation, transforms=None):
        self.root = root
        self.transforms = transforms
        self.coco = COCO(annotation)
        self.ids = list(sorted(self.coco.imgs.keys()))


    def __getitem__(self, idx):
        # Own coco file
        coco = self.coco
        # Image ID
        img_id = self.ids[idx]
        # List: get annotation id from coco
        ann_ids = coco.getAnnIds(imgIds=img_id)

        # Dictionary: target coco_annotation file for an image
        coco_annotation = coco.loadAnns(ann_ids)


        # path for input image
        path = coco.loadImgs(img_id)[0]['file_name']
        # open the input image
        img = Image.open(os.path.join(self.root, path))

        # get bounding box coordinates for each object
        #num_objs = len(obj_ids)
        num_objs = len(coco_annotation)

        boxes = []
        cat_ids = []
        for i in range(num_objs):
            xmin = coco_annotation[i]['bbox'][0]
            ymin = coco_annotation[i]['bbox'][1]
            xmax = xmin + coco_annotation[i]['bbox'][2]
            ymax = ymin + coco_annotation[i]['bbox'][3]
            boxes.append([xmin, ymin, xmax, ymax])
            cat_id = coco_annotation[i]["category_id"]
            cat_ids.append(cat_id)
        boxes = torch.as_tensor(boxes, dtype=torch.float32)
```

```
        labels = torch.as_tensor(cat_ids, dtype=torch.int64)

        image_id = torch.tensor([idx])
        area = (boxes[:, 3] - boxes[:, 1]) * (boxes[:, 2] - boxes[:, 0])
        # suppose all instances are not crowd
        iscrowd = torch.zeros((num_objs,), dtype=torch.int64)

        target = {}
        target["boxes"] = boxes
        target["labels"] = labels
        target["image_id"] = image_id
        target["area"] = area
        target["iscrowd"] = iscrowd

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

        return img, target

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

## Defining your model

In this tutorial, we will be using Faster R-CNN. Faster R-CNN is a model that predicts both bounding boxes and class scores for potential objects in the image.




There are two common situations where one might want to modify one of the available models in torchvision modelzoo. The first is when we want to start from a pre-trained model, and just finetune the last layer. The other is when we want to replace the backbone of the model with a different one (for faster predictions, for example).

Let's go see how we would do one or another in the following sections.

### 1 - Finetuning from a pretrained model

Let's suppose that you want to start from a model pre-trained on COCO and want to finetune it for your particular classes. Here is a possible way of doing it:

```
import torchvision
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor

# load a model pre-trained pre-trained on COCO
model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)

# replace the classifier with a new one, that has
# num_classes which is user-defined
num_classes = 2  # 1 class (person) + background
# get number of input features for the classifier
in_features = model.roi_heads.box_predictor.cls_score.in_features
# replace the pre-trained head with a new one
model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)
```

## 2 - Modifying the model to add a different backbone

Another common situation arises when the user wants to replace the backbone of a detection model with a different one. For example, the current default backbone (ResNet-50) might be too big for some applications, and smaller models might be necessary.

Here is how we would go into leveraging the functions provided by torchvision to modify a backbone.

```
import torchvision
from torchvision.models.detection import FasterRCNN
from torchvision.models.detection.rpn import AnchorGenerator

# load a pre-trained model for classification and return
# only the features
backbone = torchvision.models.mobilenet_v2(pretrained=True).features
# FasterRCNN needs to know the number of
# output channels in a backbone. For mobilenet_v2, it's 1280
# so we need to add it here
backbone.out_channels = 1280

# let's make the RPN generate 5 x 3 anchors per spatial
# location, with 5 different sizes and 3 different aspect
# ratios. We have a Tuple[Tuple[int]] because each feature
# map could potentially have different sizes and
# aspect ratios
anchor_generator = AnchorGenerator(sizes=((32, 64, 128, 256, 512),),
                                   aspect_ratios=((0.5, 1.0, 2.0),))

# let's define what are the feature maps that we will
# use to perform the region of interest cropping, as well as
# the size of the crop after rescaling.
# if your backbone returns a Tensor, featmap_names is expected to
# be [0]. More generally, the backbone should return an
# OrderedDict[Tensor], and in featmap_names you can choose which
# feature maps to use.
```

```
 roi_pooler = torchvision.ops.MultiScaleRoIAlign(featmap_names=[0],
                                                 output_size=7,
                                                 sampling_ratio=2)

 # put the pieces together inside a FasterRCNN model
 model = FasterRCNN(backbone,
                    num_classes=2,
                    rpn_anchor_generator=anchor_generator,
                    box_roi_pool=roi_pooler)
```

### An object_detection model for Construction Equipment Detection

In our case, we want to fine-tune from a pre-trained model, given that our dataset is very small. So we will be following approach number 1.

```
import torchvision
from torchvision.models.detection.faster_rcnn import FastRCNNPredictor


def get_object_detection_model(num_classes):
    # load an object_detection model pre-trained on COCO
    model = torchvision.models.detection.fasterrcnn_resnet50_fpn(pretrained=True)
    # get the number of input features for the classifier
    in_features = model.roi_heads.box_predictor.cls_score.in_features
    # replace the pre-trained head with a new one
    model.roi_heads.box_predictor = FastRCNNPredictor(in_features, num_classes)

    return model
```

That's it, this will make model be ready to be trained and evaluated on our custom dataset.

## Training and evaluation functions

In `references/detection/`, we have a number of helper functions to simplify training and evaluating detection models. Here, we will use `references/detection/engine.py`, `references/detection/utils.py` and `references/detection/transforms.py`.

Let's copy those files (and their dependencies) in here so that they are available in the notebook

```
%%shell

# Download TorchVision repo to use some files from
# references/detection
git clone https://github.com/pytorch/vision.git
cd vision
git checkout v0.3.0

cp references/detection/utils.py ../
cp references/detection/transforms.py ../
cp references/detection/coco_eval.py ../
cp references/detection/engine.py ../
cp references/detection/coco_utils.py ../
```

```
cp references/detection/coco_utils.py ../
```

```
Cloning into 'vision'...
remote: Enumerating objects: 26939, done.
remote: Counting objects: 100% (4548/4548), done.
remote: Compressing objects: 100% (1159/1159), done.
remote: Total 26939 (delta 3508), reused 4261 (delta 3305), pack-reused 22391
Receiving objects: 100% (26939/26939), 35.03 MiB | 34.46 MiB/s, done.
Resolving deltas: 100% (20082/20082), done.
Note: checking out 'v0.3.0'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -b with the checkout command again. Example:

  git checkout -b <new-branch-name>

HEAD is now at be376084 version check against PyTorch's CUDA version
```

Let's write some helper functions for data augmentation / transformation, which leverages the functions in `refereces/detection` that we have just copied:

```
from engine import train_one_epoch, evaluate
import utils
import transforms as T


def get_transform(train):
    transforms = []
    # converts the image, a PIL image, into a PyTorch Tensor
    transforms.append(T.ToTensor())
    if train:
        # during training, randomly flip the training images
        # and ground-truth for data augmentation
        transforms.append(T.RandomHorizontalFlip(0.5))
    return T.Compose(transforms)
```

Note that we do not need to add a mean/std normalization nor image rescaling in the data transforms, as those are handled internally by the Faster R-CNN model.

## Putting everything together

We now have the dataset class, the models and the data transforms. Let's instantiate them

```
# use our dataset and defined transformations
annotation_root = '/content/3classes/3classes.json'
file_root = '/content/3classes'

dataset = ACID_dataset(root =file_root, annotation = annotation_root, transforms = get_
dataset_val = ACID_dataset(root =file_root, annotation = annotation_root, transforms =g
```

```
# split the dataset in train and test set [use 20% for testing]
torch.manual_seed(1)
indices = torch.randperm(len(dataset)).tolist()
print(len(indices))
val_size = int(0.2*len(indices))
print(val_size)
dataset = torch.utils.data.Subset(dataset, indices[:-val_size])
dataset_val = torch.utils.data.Subset(dataset_val, indices[-val_size:])


# define training and validation data loaders
data_loader = torch.utils.data.DataLoader(
    dataset, batch_size=2, shuffle=True, num_workers=4,
    collate_fn=utils.collate_fn)

data_loader_val = torch.utils.data.DataLoader(
    dataset_val, batch_size=1, shuffle=False, num_workers=4,
    collate_fn=utils.collate_fn)
```

```
loading annotations into memory...
Done (t=0.03s)
creating index...
index created!
loading annotations into memory...
Done (t=0.02s)
creating index...
index created!
3929
785
/usr/local/lib/python3.7/dist-packages/torch/utils/data/dataloader.py:477: UserWar
  cpuset_checked))
```

Now let's instantiate the model and the optimizer

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

# our dataset has 4 classes only - background and 3 types of equipment
num_classes = 4

# get the model using our helper function
model = get_object_detection_model(num_classes)
# move model to the right device
model.to(device)

# construct an optimizer
params = [p for p in model.parameters() if p.requires_grad]
optimizer = torch.optim.SGD(params, lr=0.005,
                            momentum=0.9, weight_decay=0.0005)

# and a learning rate scheduler which decreases the learning rate by
# 10x every 3 epochs
lr_scheduler = torch.optim.lr_scheduler.StepLR(optimizer,
                                               step_size=3,
                                               gamma=0.1)
#Let us visualize the model architecture
print(model)
```

```
Downloading: "https://download.pytorch.org/models/fasterrcnn_resnet50_fpn_coco-258
100%                                160M/160M [00:02<00:00, 66.8MB/s]

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), bi
      (bn1): FrozenBatchNorm2d(64, eps=0.0)
      (relu): ReLU(inplace=True)
      (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mo
      (layer1): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(64, eps=0.0)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
          (bn2): FrozenBatchNorm2d(64, eps=0.0)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(256, eps=0.0)
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
            (1): FrozenBatchNorm2d(256, eps=0.0)
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(64, eps=0.0)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
          (bn2): FrozenBatchNorm2d(64, eps=0.0)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(256, eps=0.0)
          (relu): ReLU(inplace=True)
        )
        (2): Bottleneck(
          (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(64, eps=0.0)
          (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1
          (bn2): FrozenBatchNorm2d(64, eps=0.0)
          (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(256, eps=0.0)
          (relu): ReLU(inplace=True)
        )
      )
      (layer2): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(128, eps=0.0)
          (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1,
          (bn2): FrozenBatchNorm2d(128, eps=0.0)
          (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn3): FrozenBatchNorm2d(512, eps=0.0)
          (relu): ReLU(inplace=True)
          (downsample): Sequential(
            (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
            (1): FrozenBatchNorm2d(512, eps=0.0)
          )
        )
        (1): Bottleneck(
          (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
          (bn1): FrozenBatchNorm2d(128, eps=0.0)
```

```
      (bn1): FrozenBatchNorm2d(128, eps=0.0)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
      (bn2): FrozenBatchNorm2d(128, eps=0.0)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): FrozenBatchNorm2d(512, eps=0.0)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): FrozenBatchNorm2d(128, eps=0.0)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
      (bn2): FrozenBatchNorm2d(128, eps=0.0)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): FrozenBatchNorm2d(512, eps=0.0)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): FrozenBatchNorm2d(128, eps=0.0)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
      (bn2): FrozenBatchNorm2d(128, eps=0.0)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): FrozenBatchNorm2d(512, eps=0.0)
      (relu): ReLU(inplace=True)
    )
  )
  (layer3): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): FrozenBatchNorm2d(256, eps=0.0)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1,
      (bn2): FrozenBatchNorm2d(256, eps=0.0)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): FrozenBatchNorm2d(1024, eps=0.0)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): FrozenBatchNorm2d(1024, eps=0.0)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): FrozenBatchNorm2d(256, eps=0.0)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
      (bn2): FrozenBatchNorm2d(256, eps=0.0)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): FrozenBatchNorm2d(1024, eps=0.0)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): FrozenBatchNorm2d(256, eps=0.0)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
      (bn2): FrozenBatchNorm2d(256, eps=0.0)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): FrozenBatchNorm2d(1024, eps=0.0)
      (relu): ReLU(inplace=True)
    )
    (3): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): FrozenBatchNorm2d(256, eps=0.0)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
      (bn2): FrozenBatchNorm2d(256, eps=0.0)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): FrozenBatchNorm2d(1024, eps=0.0)
      (relu): ReLU(inplace=True)
```

```
      (relu): ReLU(inplace=True)
    )
    (4): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): FrozenBatchNorm2d(256, eps=0.0)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
      (bn2): FrozenBatchNorm2d(256, eps=0.0)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): FrozenBatchNorm2d(1024, eps=0.0)
      (relu): ReLU(inplace=True)
    )
    (5): Bottleneck(
      (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): FrozenBatchNorm2d(256, eps=0.0)
      (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1,
      (bn2): FrozenBatchNorm2d(256, eps=0.0)
      (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): FrozenBatchNorm2d(1024, eps=0.0)
      (relu): ReLU(inplace=True)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): FrozenBatchNorm2d(512, eps=0.0)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1,
      (bn2): FrozenBatchNorm2d(512, eps=0.0)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): FrozenBatchNorm2d(2048, eps=0.0)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): FrozenBatchNorm2d(2048, eps=0.0)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): FrozenBatchNorm2d(512, eps=0.0)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
      (bn2): FrozenBatchNorm2d(512, eps=0.0)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): FrozenBatchNorm2d(2048, eps=0.0)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn1): FrozenBatchNorm2d(512, eps=0.0)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1,
      (bn2): FrozenBatchNorm2d(512, eps=0.0)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False
      (bn3): FrozenBatchNorm2d(2048, eps=0.0)
      (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(featmap_names=['0', '1', '2', '3'], output_
        (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=4, bias=True)
          (bbox_pred): Linear(in_features=1024, out_features=16, bias=True)
        )
      )
    )
```

And now let's train the model for 10 epochs, evaluating at the end of every epoch.

```
! ls /content/3classes

# let's train it for 10 epochs
num_epochs = 1

for epoch in range(num_epochs):
    # train for one epoch, printing every 10 iterations
    train_one_epoch(model, optimizer, data_loader, device, epoch, print_freq=10)
    # update the learning rate
    lr_scheduler.step()
    # evaluate on the test dataset
    evaluate(model, data_loader_val, device=device)
```

```
 00001.jpg  01527.jpg  02935.jpg  04431.jpg  05853.jpg  07294.jpg  08689.jpg
 00002.jpg  01529.jpg  02940.jpg  04433.jpg  05854.jpg  07296.jpg  08690.jpg
 00010.jpg  01531.jpg  02941.jpg  04435.jpg  05859.jpg  07297.jpg  08695.jpg
 00011.jpg  01534.jpg  02946.jpg  04437.jpg  05861.jpg  07300.jpg  08696.jpg
 00012.jpg  01536.jpg  02948.jpg  04438.jpg  05862.jpg  07306.jpg  08699.jpg
 00013.jpg  01537.jpg  02949.jpg  04439.jpg  05869.jpg  07307.jpg  08700.jpg
 00018.jpg  01538.jpg  02950.jpg  04440.jpg  05870.jpg  07308.jpg  08701.jpg
 00021.jpg  01539.jpg  02951.jpg  04444.jpg  05871.jpg  07309.jpg  08703.jpg
 00022.jpg  01542.jpg  02952.jpg  04448.jpg  05876.jpg  07313.jpg  08705.jpg
```

```
00025.jpg  01544.jpg  02959.jpg  04450.jpg  05881.jpg  07315.jpg  08707.jpg
00028.jpg  01546.jpg  02962.jpg  04451.jpg  05883.jpg  07318.jpg  08713.jpg
00030.jpg  01549.jpg  02964.jpg  04452.jpg  05890.jpg  07319.jpg  08714.jpg
00031.jpg  01551.jpg  02972.jpg  04458.jpg  05891.jpg  07321.jpg  08717.jpg
00034.jpg  01556.jpg  02979.jpg  04460.jpg  05893.jpg  07323.jpg  08720.jpg
00037.jpg  01561.jpg  02983.jpg  04461.jpg  05894.jpg  07325.jpg  08722.jpg
00038.jpg  01562.jpg  02984.jpg  04465.jpg  05896.jpg  07326.jpg  08723.jpg
00041.jpg  01564.jpg  02986.jpg  04468.jpg  05897.jpg  07329.jpg  08725.jpg
00042.jpg  01565.jpg  02989.jpg  04471.jpg  05902.jpg  07331.jpg  08726.jpg
00043.jpg  01566.jpg  02992.jpg  04475.jpg  05903.jpg  07332.jpg  08727.jpg
00047.jpg  01568.jpg  02996.jpg  04477.jpg  05904.jpg  07333.jpg  08728.jpg
00051.jpg  01570.jpg  02997.jpg  04482.jpg  05906.jpg  07335.jpg  08731.jpg
00053.jpg  01574.jpg  03000.jpg  04484.jpg  05907.jpg  07337.jpg  08733.jpg
00059.jpg  01576.jpg  03003.jpg  04485.jpg  05911.jpg  07340.jpg  08737.jpg
00066.jpg  01577.jpg  03006.jpg  04486.jpg  05918.jpg  07345.jpg  08745.jpg
00076.jpg  01579.jpg  03008.jpg  04487.jpg  05921.jpg  07346.jpg  08746.jpg
00080.jpg  01580.jpg  03010.jpg  04491.jpg  05922.jpg  07349.jpg  08758.jpg
00081.jpg  01583.jpg  03011.jpg  04493.jpg  05926.jpg  07352.jpg  08759.jpg
00082.jpg  01585.jpg  03014.jpg  04494.jpg  05933.jpg  07355.jpg  08760.jpg
00085.jpg  01587.jpg  03015.jpg  04496.jpg  05937.jpg  07358.jpg  08763.jpg
00088.jpg  01588.jpg  03018.jpg  04497.jpg  05944.jpg  07361.jpg  08768.jpg
00090.jpg  01590.jpg  03021.jpg  04500.jpg  05945.jpg  07365.jpg  08770.jpg
00092.jpg  01591.jpg  03022.jpg  04503.jpg  05950.jpg  07366.jpg  08772.jpg
00093.jpg  01592.jpg  03025.jpg  04504.jpg  05954.jpg  07367.jpg  08773.jpg
00094.jpg  01596.jpg  03026.jpg  04506.jpg  05963.jpg  07368.jpg  08777.jpg
00095.jpg  01597.jpg  03027.jpg  04509.jpg  05964.jpg  07369.jpg  08778.jpg
00098.jpg  01598.jpg  03032.jpg  04512.jpg  05966.jpg  07373.jpg  08788.jpg
00099.jpg  01602.jpg  03033.jpg  04517.jpg  05967.jpg  07379.jpg  08791.jpg
00100.jpg  01606.jpg  03035.jpg  04523.jpg  05968.jpg  07380.jpg  08793.jpg
00105.jpg  01607.jpg  03036.jpg  04524.jpg  05969.jpg  07382.jpg  08795.jpg
00108.jpg  01608.jpg  03037.jpg  04527.jpg  05974.jpg  07385.jpg  08796.jpg
00109.jpg  01610.jpg  03040.jpg  04529.jpg  05977.jpg  07386.jpg  08797.jpg
00118.jpg  01615.jpg  03054.jpg  04530.jpg  05978.jpg  07387.jpg  08799.jpg
00119.jpg  01620.jpg  03055.jpg  04535.jpg  05979.jpg  07391.jpg  08805.jpg
00126.jpg  01626.jpg  03058.jpg  04536.jpg  05980.jpg  07392.jpg  08810.jpg
00127.jpg  01627.jpg  03063.jpg  04538.jpg  05981.jpg  07395.jpg  08811.jpg
00130.jpg  01631.jpg  03068.jpg  04540.jpg  05983.jpg  07397.jpg  08812.jpg
00133.jpg  01632.jpg  03071.jpg  04541.jpg  05984.jpg  07398.jpg  08816.jpg
00134.jpg  01638.jpg  03072.jpg  04543.jpg  05988.jpg  07399.jpg  08818.jpg
00135.jpg  01639.jpg  03073.jpg  04545.jpg  05989.jpg  07407.jpg  08828.jpg
00140.jpg  01640.jpg  03075.jpg  04546.jpg  05992.jpg  07409.jpg  08831.jpg
00141.jpg  01646.jpg  03076.jpg  04549.jpg  05993.jpg  07412.jpg  08832.jpg
00148.jpg  01650.jpg  03078.jpg  04550.jpg  05994.jpg  07415.jpg  08833.jpg
00152.jpg  01652.jpg  03080.jpg  04552.jpg  05998.jpg  07418.jpg  08836.jpg
00153.jpg  01654.jpg  03082.jpg  04553.jpg  05999.jpg  07423.jpg  08840.jpg
00157.jpg  01656.jpg  03084.jpg  04554.jpg  06000.jpg  07425.jpg  08841.jpg
00158.jpg  01660.jpg  03089.jpg  04556.jpg  06002.jpg  07426.jpg  08842.jpg
00159.jpg  01663.jpg  03092.jpg  04559.jpg  06003.jpg  07427.jpg  08846.jpg
00161.jpg  01665.jpg  03099.jpg  04562.jpg  06004.jpg  07428.jpg  08849.jpg
00164.jpg  01666.jpg  03106.jpg  04563.jpg  06008.jpg  07431.jpg  08851.jpg
```

Now that training has finished, let's have a look at what it actually predicts in a test image

```
# pick one image from the test set
img, _ = dataset_val[5]
# put the model in evaluation mode
model.eval()
with torch.no_grad():
    prediction = model([img.to(device)])
```

Printing the prediction shows that we have a list of dictionaries. Each element of the list corresponds to a different image. As we have a single image, there is a single dictionary in the list. The dictionary contains the predictions for the image we passed. In this case, we can see that it contains `boxes`,

```
prediction
```

```
[{'boxes': tensor([[ 847.1060,  238.2487, 1259.9624,  544.6272],
          [  67.4519,  218.1165,  828.6486,  588.8081],
          [ 415.7662,  222.1267, 1248.2137,  561.9327],
          [ 158.3181,  204.8467,  793.3710,  589.0566],
          [  16.0116,  308.3200,  484.0447,  584.0211]], device='cuda:0'),
  'labels': tensor([2, 2, 2, 3, 2], device='cuda:0'),
  'scores': tensor([0.9667, 0.6619, 0.1202, 0.1194, 0.0722], device='cuda:0')}]
```

# Visualization of the predictions

Let's inspect an image and visualize the top predicted bounding box.

```
from google.colab.patches import cv2_imshow
from PIL import Image
import matplotlib.pyplot as plt
import torch
import torchvision.transforms as T
import torchvision
import numpy as np
import json
import cv2
import random
import warnings
warnings.filterwarnings('ignore')

# set to evaluation mode
model.eval()

file = open('3classes/3classes.json', 'r')
data = json.loads(file.read())
CLASS_NAMES = []
for category in data['categories']:
  CLASS_NAMES.append(category['name'])
print(CLASS_NAMES)


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

def get_prediction(img_path, confidence):
    """
    get_prediction
      parameters:
        - img_path - path of the input image
        - confidence - threshold to keep the prediction or not
      method:
        - Image is obtained from the image path
        - the image is converted to image tensor using PyTorch's Transforms
        - image is passed through the model to get the predictions
```

```
        - classes and bounding boxes are obtained from the model and soft masks are mad
          ie: eg. segment of cat is made 1 and rest of the image is made 0

    """
    img = Image.open(img_path)
    transform = T.Compose([T.ToTensor()])
    img = transform(img)

    img = img.to(device)
    pred = model([img])
    pred_score = list(pred[0]['scores'].detach().cpu().numpy())
    pred_t = [pred_score.index(x) for x in pred_score if x>confidence][-1]
    pred_class = [CLASS_NAMES[i] for i in list(pred[0]['labels'].cpu().numpy())]
    pred_boxes = [[(i[0], i[1]), (i[2], i[3])] for i in list(pred[0]['boxes'].detach().
    pred_boxes = pred_boxes[:pred_t+1]
    pred_class = pred_class[:pred_t+1]
    return pred_boxes, pred_class

def detect_object(img_path, confidence=0.5, rect_th=2, text_size=1, text_th=2):
    """
    detect_object
      parameters:
        - img_path - path to input image
        - confidence- confidence to keep the prediction or not
        - rect_th - rect thickness
        - text_size
        - text_th - text thickness
      method:
        - prediction is obtained by get_prediction
        - each box is given a color
        - final output is displayed
    """
    boxes, pred_cls = get_prediction(img_path, confidence)
    img = cv2.imread(img_path)
    img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
    for i in range(len(boxes)):
      random.seed(32)
      cv2.rectangle(img, boxes[i][0], boxes[i][1],color=(0,255,0), thickness=rect_th)
      cv2.putText(img,pred_cls[i], boxes[i][0], cv2.FONT_HERSHEY_SIMPLEX, text_size, (2
    return img

     ['None', 'dozer', 'dump_truck', 'excavator']


#Try with one image from the dataset
img_path = '3classes/10052.jpg'
img = detect_object(img_path, confidence=0.7)
img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
cv2_imshow(img)
img_path1 = '3classes/10071.jpg'
img1 = detect_object(img_path1, confidence=0.7)
img1 = cv2.cvtColor(img1, cv2.COLOR_BGR2RGB)
cv2_imshow(img1)
```


dozer
dump_truck
excavator
dump_truck
dump_truck

Looks pretty good!

## Wrapping up

In this tutorial, you have learned how to create your own training pipeline for object detection models, on a custom dataset. For that, you wrote a `torch.utils.data.Dataset` class that returns the images and the ground truth boxes. You also leveraged a Faster R-CNN model pre-trained on COCO train2017 in order to perform transfer learning on this new dataset.

For a more complete example, which includes multi-machine / multi-gpu training, check `references/detection/train.py`, which is present in the [torchvision GitHub repo](#).

Reference: This handson was prepared following this [PyTorch](#) tutorial

# Give your judgement

Now its time for you to evaluate the model performance and comment on the following scenarios

1.Do the following hyper parameters affect the accuracy, precision and recall? If so, then explain how.

```
i. learning rates
ii. Optimizer (Can try with SGD, Adam, RMSProp)
iii. batch size
iv. Number of epochs
```

2.Try the model with another backbone and comment on your observations.

# Calculate accuracy, and draw precision-recall Curve

```
from __future__ import division
import warnings
import numpy as np
import matplotlib.pyplot as plt

def precision(tp, fn, fp, tn):
    with np.errstate(divide='ignore', invalid='ignore'):
        return tp/(tp + fp)


def recall(tp, fn, fp, tn):
    with np.errstate(divide='ignore', invalid='ignore'):
        return tp/(tp + fn)
```

```
def precision_gain(tp, fn, fp, tn):
    """Calculates Precision Gain from the contingency table
    This function calculates Precision Gain from the entries of the contingency
    table: number of true positives (TP), false negatives (FN), false positives
    (FP), and true negatives (TN). More information on Precision-Recall-Gain
    curves and how to cite this work is available at
    http://www.cs.bris.ac.uk/~flach/PRGcurves/.
    """
    n_pos = tp + fn
    n_neg = fp + tn
    with np.errstate(divide='ignore', invalid='ignore'):
        prec_gain = 1. - (n_pos/n_neg) * (fp/tp)
    if np.alen(prec_gain) > 1:
        prec_gain[tn + fn == 0] = 0
    elif tn + fn == 0:
        prec_gain = 0
    return prec_gain

def recall_gain(tp, fn, fp, tn):
    """Calculates Recall Gain from the contingency table
    This function calculates Recall Gain from the entries of the contingency
    table: number of true positives (TP), false negatives (FN), false positives
    (FP), and true negatives (TN). More information on Precision-Recall-Gain
    curves and how to cite this work is available at
    http://www.cs.bris.ac.uk/~flach/PRGcurves/.
    Args:
        tp (float) or ([float]): True Positives
        fn (float) or ([float]): False Negatives
        fp (float) or ([float]): False Positives
        tn (float) or ([float]): True Negatives
    Returns:
        (float) or ([float])
    """
    n_pos = tp + fn
    n_neg = fp + tn
    with np.errstate(divide='ignore', invalid='ignore'):
        rg = 1. - (n_pos/n_neg) * (fn/tp)
    if np.alen(rg) > 1:
        rg[tn + fn == 0] = 1
    elif tn + fn == 0:
        rg = 1
    return rg

def create_segments(labels, pos_scores, neg_scores):
    n = np.alen(labels)
    # reorder labels and pos_scores by decreasing pos_scores, using increasing neg_scor
    new_order = np.lexsort((neg_scores, -pos_scores))
    labels = labels[new_order]
    pos_scores = pos_scores[new_order]
    neg_scores = neg_scores[new_order]
    # create a table of segments
    segments = {'pos_score': np.zeros(n), 'neg_score': np.zeros(n),
                'pos_count': np.zeros(n), 'neg_count': np.zeros(n)}
    j = -1
    for i, label in enumerate(labels):
        if ((i == 0) or (pos_scores[i-1] != pos_scores[i])
```

```
                    or (neg_scores[i-1] != neg_scores[i])):
                j += 1
                segments['pos_score'][j] = pos_scores[i]
                segments['neg_score'][j] = neg_scores[i]
            if label == 0:
                segments['neg_count'][j] += 1
            else:
                segments['pos_count'][j] += 1
        segments['pos_score'] = segments['pos_score'][0:j+1]
        segments['neg_score'] = segments['neg_score'][0:j+1]
        segments['pos_count'] = segments['pos_count'][0:j+1]
        segments['neg_count'] = segments['neg_count'][0:j+1]
        return segments

    def get_point(points, index):
        keys = points.keys()
        point = np.zeros(np.alen(keys))
        key_indices = dict()
        for i, key in enumerate(keys):
            point[i] = points[key][index]
            key_indices[key] = i
        return [point, key_indices]


    def insert_point(new_point, key_indices, points, precision_gain=0,
            recall_gain=0, is_crossing=0):
        for key in key_indices.keys():
            points[key] = np.insert(points[key], 0, new_point[key_indices[key]])
        points['precision_gain'][0] = precision_gain
        points['recall_gain'][0] = recall_gain
        points['is_crossing'][0] = is_crossing
        new_order = np.lexsort((-points['precision_gain'],points['recall_gain']))
        for key in points.keys():
            points[key] = points[key][new_order]
        return points


    def _create_crossing_points(points, n_pos, n_neg):
        n = n_pos+n_neg
        points['is_crossing'] = np.zeros(np.alen(points['pos_score']))
        # introduce a crossing point at the crossing through the y-axis
        j = np.amin(np.where(points['recall_gain'] >= 0)[0])
        if points['recall_gain'][j] > 0:  # otherwise there is a point on the boundary and
            [point_1, key_indices_1] = get_point(points, j)
            [point_2, key_indices_2] = get_point(points, j-1)
            delta = point_1 - point_2
            if delta[key_indices_1['TP']] > 0:
                alpha = (n_pos*n_pos/n - points['TP'][j-1]) / delta[key_indices_1['TP']]
            else:
                alpha = 0.5

            with warnings.catch_warnings():
                warnings.simplefilter("ignore")
                new_point = point_2 + alpha*delta

            new_prec_gain = precision_gain(new_point[key_indices_1['TP']], new_point[key_in
                                           new_point[key_indices_1['FP']], new_point[key_in
            points = insert_point(new_point, key_indices_1, points,
```

```
                                precision_gain=new_prec_gain, is_crossing=1)

        # now introduce crossing points at the crossings through the non-negative part of t
        x = points['recall_gain']
        y = points['precision_gain']
        temp_y_0 = np.append(y, 0)
        temp_0_y = np.append(0, y)
        temp_1_x = np.append(1, x)
        with np.errstate(invalid='ignore'):
            indices = np.where(np.logical_and((temp_y_0 * temp_0_y < 0), (temp_1_x >= 0)))[
        for i in indices:
            cross_x = x[i-1] + (-y[i-1]) / (y[i] - y[i-1]) * (x[i] - x[i-1])
            [point_1, key_indices_1] = get_point(points, i)
            [point_2, key_indices_2] = get_point(points, i-1)
            delta = point_1 - point_2
            if delta[key_indices_1['TP']] > 0:
                alpha = (n_pos * n_pos / (n - n_neg * cross_x) - points['TP'][i-1]) / delta
            else:
                alpha = (n_neg / n_pos * points['TP'][i-1] - points['FP'][i-1]) / delta[key

            with warnings.catch_warnings():
                warnings.simplefilter("ignore")
                new_point = point_2 + alpha*delta

            new_rec_gain = recall_gain(new_point[key_indices_1['TP']], new_point[key_indice
                                       new_point[key_indices_1['FP']], new_point[key_indice
            points = insert_point(new_point, key_indices_1, points,
                                  recall_gain=new_rec_gain, is_crossing=1)
            i += 1
            indices += 1
            x = points['recall_gain']
            y = points['precision_gain']
            temp_y_0 = np.append(y, 0)
            temp_0_y = np.append(0, y)
            temp_1_x = np.append(1, x)
        return points

    def create_prg_curve(labels, pos_scores, neg_scores=[]):
        """Precision-Recall-Gain curve
        This function creates the Precision-Recall-Gain curve from the vector of
        labels and vector of scores where higher score indicates a higher
        probability to be positive. More information on Precision-Recall-Gain
        curves and how to cite this work is available at
        http://www.cs.bris.ac.uk/~flach/PRGcurves/.
        """
        create_crossing_points = True # do it always because calc_auprg otherwise gives the
        if np.alen(neg_scores) == 0:
            neg_scores = -pos_scores
        n = np.alen(labels)
        n_pos = np.sum(labels)
        n_neg = n - n_pos
        # convert negative labels into 0s
        labels = 1 * (labels == 1)
        segments = create_segments(labels, pos_scores, neg_scores)
        # calculate recall gains and precision gains for all thresholds
        points = dict()
        points['pos_score'] = np.insert(segments['pos_score'], 0, np.inf)
        points['neg_score'] = np.insert(segments['neg_score'], 0, -np.inf)
```

```
        points['TP'] = np.insert(np.cumsum(segments['pos_count']), 0, 0)
        points['FP'] = np.insert(np.cumsum(segments['neg_count']), 0, 0)
        points['FN'] = n_pos - points['TP']
        points['TN'] = n_neg - points['FP']
        points['precision'] = precision(points['TP'], points['FN'], points['FP'], points['T
        points['recall'] = recall(points['TP'], points['FN'], points['FP'], points['TN'])
        points['precision_gain'] = precision_gain(points['TP'], points['FN'], points['FP'],
        points['recall_gain'] = recall_gain(points['TP'], points['FN'], points['FP'], point
        if create_crossing_points == True:
            points = _create_crossing_points(points, n_pos, n_neg)
        else:
            points['pos_score'] = points['pos_score'][1:]
            points['neg_score'] = points['neg_score'][1:]
            points['TP'] = points['TP'][1:]
            points['FP'] = points['FP'][1:]
            points['FN'] = points['FN'][1:]
            points['TN'] = points['TN'][1:]
            points['precision_gain'] = points['precision_gain'][1:]
            points['recall_gain'] = points['recall_gain'][1:]
        with np.errstate(invalid='ignore'):
            points['in_unit_square'] = np.logical_and(points['recall_gain'] >= 0,
                                                      points['precision_gain'] >= 0)
        return points

    def calc_auprg(prg_curve):
        """Calculate area under the Precision-Recall-Gain curve
        This function calculates the area under the Precision-Recall-Gain curve
        from the results of the function create_prg_curve. More information on
        Precision-Recall-Gain curves and how to cite this work is available at
        http://www.cs.bris.ac.uk/~flach/PRGcurves/.
        """
        area = 0
        recall_gain = prg_curve['recall_gain']
        precision_gain = prg_curve['precision_gain']
        for i in range(1,len(recall_gain)):
            if (not np.isnan(recall_gain[i-1])) and (recall_gain[i-1]>=0):
                width = recall_gain[i]-recall_gain[i-1]
                height = (precision_gain[i]+precision_gain[i-1])/2
                area += width*height
        return(area)


    # from
    def convex_hull(points):
        """Computes the convex hull of a set of 2D points.
        Input: an iterable sequence of (x, y) pairs representing the points.
        Output: a list of vertices of the convex hull in counter-clockwise order,
          starting from the vertex with the lexicographically smallest coordinates.
        Implements Andrew's monotone chain algorithm. O(n log n) complexity.
        Source code from:
        https://en.wikibooks.org/wiki/Algorithm_Implementation/Geometry/Convex_hull/Monoton
        """

        # Sort the points lexicographically (tuples are compared lexicographically).
        # Remove duplicates to detect the case we have just one unique point.
        points = sorted(set(points))

        # Boring case: no points or a single point, possibly repeated multiple times.
        if len(points) <= 1:
```

```
    if len(points) <= 1:
        return points

    # 2D cross product of OA and OB vectors, i.e. z-component of their 3D cross product
    # Returns a positive value, if OAB makes a counter-clockwise turn,
    # negative for clockwise turn, and zero if the points are collinear.
    def cross(o, a, b):
        return (a[0] - o[0]) * (b[1] - o[1]) - (a[1] - o[1]) * (b[0] - o[0])

    # Build upper hull
    upper = []
    for p in reversed(points):
        while len(upper) >= 2 and cross(upper[-2], upper[-1], p) <= 0:
            upper.pop()
        upper.append(p)

    return upper


def plot_prg(prg_curve,show_convex_hull=True,show_f_calibrated_scores=False):
    """Plot the Precision-Recall-Gain curve
    This function plots the Precision-Recall-Gain curve resulting from the
    function create_prg_curve using ggplot. More information on
    Precision-Recall-Gain curves and how to cite this work is available at
    http://www.cs.bris.ac.uk/~flach/PRGcurves/.
    @param prg_curve the data structure resulting from the function create_prg_curve
    @param show_convex_hull whether to show the convex hull (default: TRUE)
    @param show_f_calibrated_scores whether to show the F-calibrated scores (default:TR
    @return the ggplot object which can be plotted using print()
    @details This function plots the Precision-Recall-Gain curve, indicating
        for each point whether it is a crossing-point or not (see help on
        create_prg_curve). By default, only the part of the curve
        within the unit square [0,1]x[0,1] is plotted.
    @examples
        labels = c(1,1,1,0,1,1,1,1,1,1,0,1,1,1,0,1,0,0,1,0,0,0,1,0,1)
        scores = (25:1)/25
        plot_prg(create_prg_curve(labels,scores))
    """
    pg = prg_curve['precision_gain']
    rg = prg_curve['recall_gain']

    fig = plt.figure(figsize=(6,5))
    plt.clf()
    plt.axes(frameon=False)
    ax = fig.gca()
    ax.set_xticks(np.arange(0,1.25,0.25))
    ax.set_yticks(np.arange(0,1.25,0.25))
    ax.grid(b=True)
    ax.set_xlim((-0.05,1.02))
    ax.set_ylim((-0.05,1.02))
    ax.set_aspect('equal')
    # Plot vertical and horizontal lines crossing the 0 axis
    plt.axvline(x=0, ymin=-0.05, ymax=1, color='k')
    plt.axhline(y=0, xmin=-0.05, xmax=1, color='k')
    plt.axvline(x=1, ymin=0, ymax=1, color='k')
    plt.axhline(y=1, xmin=0, xmax=1, color='k')
    # Plot cyan lines
    indices = np.arange(np.argmax(prg_curve['in_unit_square']) - 1,
                        len(prg_curve['in_unit_square']))
```