Note for TA/grader:

I feel extremely difficult for me to run the code on Google Colab. Several items prevent me from doing so:

- The project requires specifically Python 3.7 and pytorch=1.9.1+cu102. However, I tried for a long time to install this specific pytorch version but failed. No matter how I changed the python version or install the older version of pytorch, the environment on Google Colab is always cuda 12.2 with pytorch 2.0.
- I cannot install GLIP and maskrcnn_benchmark on Google Colab, as they need to be manually built (e.g., using python setup.py build develop).

Therefore, I cannot run the code on Google Colab. Instead, I will only show the corresponding code snippets regarding every part of the project as below in the Google Colab. The actual experiments are run on my own machines with NVIDIA 1080/2080 GPUs. I would take a screenshot of my experimental results, and then copy the screenshot images to the Google Colab below. Even building the environment on my own machines, it is still difficult for me to work on this project. (But finally I made it!) My major difficulities include:

- The given dataset in the official repository on GitHub is actually not incomplete. The validation data is missing. Therefore, I find the full datasets from another repository https://github.com/DengPingFan/PraNet.
- The maskrcnn_benchmark/layers folder in the official repository actually missed an important file "_utils.py". I previously thought that it is because my installed pytorch/maskrcnn_benchmark packages are in the wrong version. But finally I find out that adding the _utils.py back to the directory would fix this issue after some time and effort.
- When building the maskrcnn_benchmark package from source code, we should specify CUDA_HOME with the path of CUDA 10.2. Otherwise, the maskrcnn_benchmark cannot use GPU when running the code.

Introduction

- Background of the problem

    - what type of problem: This paper focuses on medical image understanding with pretrained vision language models. The medical image understanding tasks investigated in the paper are mainly referring object detection tasks.
    - what is the importance/meaning of solving the problem: Currently vision language models have shown to be extremely powerful in domains like natural images and other image domains, so they have large potential for benefiting the medical image domain.
    - what is the difficulty of the problem: The medical image domain requires a very high level of expert knowledge, so it is very challenging to effectively transfer the pretrained knowledge in these vision language models to the medical image domain.
    - the state of the art methods and effectiveness: The state-of-the-art solution is to use knowledge transfer techniques to transfer the knowledge from the pretrained vision language models to the specific medical image domain. However, since the medical image domain has much expert knowledge specific in the domain, these knowledge transfer techniques are not that effective.
- Paper explanation
    - what did the paper propose: This paper proposes approaches of automatic generation of medical prompts for the pretrained vision language models.
    - what is the innovations of the method: The authors conduct a comprehensive study on different strategies of generating the medical prompts, giving a rough guideline of what kinds of prompts are most beneficial for the task.
    - how well the proposed method work (in its own metrics): The proposed method outperforms all existing methods in medical image understanding tasks like object detection.
    - what is the contribution to the reasearch regime (referring the Background above, how important the paper is to the problem): This paper presents an effective way of generating medical prompts, and successfully utilizing the generated prompts to use the knowledge from pretrained vision language models for medical image understanding. It is the first of accomplishing this goal effectively.

Scope of Reproducibility:

List hypotheses from the paper you will test and the corresponding experiments you will run.

1. Hypothesis 1: Using the Masked Language Model (MLM) technology, we can realize auto-prompt generation.
2. Hypothesis 2: Using the automatically generated prompts, the knowledge transferability from pretrained vision language model (GLIP used in this paper) to the target medical image domain can be enhanced.

Methodology

This methodology is the core of your project. It consists of run-able codes with necessary annotations to show the expeiment you executed for testing the hypotheses.

The methodology at least contains two subsections **data** and **model** in your experiment.

Data

Data includes raw data (MIMIC III tables), descriptive statistics (our homework questions), and data processing (feature engineering).

- Source of the data: The data is downloaded from https://drive.google.com/file/d/10ISx1yXxfE20nKq6UqquUAD5Egk3hyqi/view?usp=sharing and https://drive.google.com/file/d/1Y2z7FD5p5y31vkZwQQomXFRB0HutHyao/view. Since the officially released dataset (the prior link) is incomplete. The validation data is missing.
- Statistics: include basic descriptive statistics of the dataset like size, cross validation split, label distribution, etc.
- Data process: how do you munipulate the data, e.g., change the class labels, split the dataset to train/valid/test, refining the dataset.
- Illustration: printing results, plotting figures for illustration.

```
from  maskrcnn_benchmark.data.datasets  import  CocoDetection
import  os
import  os.path
import  math
from  PIL  import  Image,  ImageDraw

import  random
```

```
import numpy as np

import torch
import torchvision
import torch.utils.data as data
from maskrcnn_benchmark.data.datasets.coco import COCODataset

from maskrcnn_benchmark.structures.bounding_box import BoxList
from maskrcnn_benchmark.structures.segmentation_mask import SegmentationMask
from maskrcnn_benchmark.structures.keypoint import PersonKeypoints
from maskrcnn_benchmark.config import cfg
import pdb


def pil_loader(path, retry=5):
        ri = 0
        while ri < retry:
                try:
                        with open(path, 'rb') as f:
                                img = Image.open(f)
                                return img.convert('RGB')
                except:
                        ri += 1


def rgb2id(color):
        if isinstance(color, np.ndarray) and len(color.shape) == 3:
                if color.dtype == np.uint8:
                        color = color.astype(np.int32)
                return color[:, :, 0] + 256 * color[:, :, 1] + 256 * 256 * color[:, :, 2]
        return int(color[0] + 256 * color[1] + 256 * 256 * color[2])


class CocoDetection(data.Dataset):
        def __init__(self, root, annFile, transform=None, target_transform=None):
                from pycocotools.coco import COCO
                self.root = root
                self.coco = COCO(annFile)
```

```
        self.ids = list(self.coco.imgs.keys())
        self.transform = transform
        self.target_transform = target_transform

    def __getitem__(self, index, return_meta=False):
        coco = self.coco
        img_id = self.ids[index]
        if isinstance(img_id, str):
            img_id = [img_id]
        ann_ids = coco.getAnnIds(imgIds=img_id)
        target = coco.loadAnns(ann_ids)

        meta = coco.loadImgs(img_id)[0]
        path = meta['file_name']
        img = pil_loader(os.path.join(self.root, path))

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

        if self.target_transform is not None:
            target = self.target_transform(target)

        if return_meta:
            return img, target, meta
        else:
            return img, target, os.path.join(self.root, path)

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

    def __repr__(self):
        fmt_str = 'Dataset ' + self.__class__.__name__ + '\n'
        fmt_str += '    Number of datapoints: {}\n'.format(self.__len__())
        fmt_str += '    Root Location: {}\n'.format(self.root)
        tmp = '    Transforms (if any): '
        fmt_str += '{0}{1}\n'.format(tmp, self.transform.__repr__().replace('\n', '\n' + ' ' * len(tmp)))
        tmp = '    Target Transforms (if any): '
```

```
                fmt_str  +=  '{0}{1}'.format(tmp,  self.target_transform.__repr__().replace('\n',  '\n'  +  '  '  *  len(tmp)))
                return  fmt_str


class  VqaCollator(object):
        def  __init__(self,  size_divisible=0):
                self.size_divisible  =  size_divisible

        def  __call__(self,  batch):
                transposed_batch  =  list(zip(*batch))
                images  =  transposed_batch[0]
                targets  =  transposed_batch[1]
                paths  =  transposed_batch[2]
                return  images,  targets,  paths

def  make_dataloader(root,  annFile,  transforms,  **args):
        print(root,  annFile,  "root!!!!!!!!!!!!!!!")
        dataset  =  CocoDetection(root,  annFile,  transforms)
        collate_batch  =  VqaCollator()
        data_loader  =  torch.utils.data.DataLoader(
                dataset,
                num_workers=8,
                collate_fn=collate_batch
        )

        return  data_loader
```

Model

The model includes the model definitation which usually is a class, model training, and other necessary parts.

- Model architecture: layer number/size/type, activation function, etc
- Training objectives: loss function, optimizer, weight of each loss term, etc
- Others: whether the model is pretrained, Monte Carlo simulation for uncertainty analysis, etc

- The code of model should have classes of the model, functions of model training, model validation, etc.
- If your model training is done outside of this notebook, please upload the trained model here and develop a function to load and test it.

```
import torch
from torch import nn
import torch.nn.functional as F

from maskrcnn_benchmark.structures.image_list import to_image_list
from maskrcnn_benchmark.structures.bounding_box import BoxList
from maskrcnn_benchmark.structures.boxlist_ops import cat_boxlist

from ..backbone import build_backbone
from ..rpn import build_rpn
from ..roi_heads import build_roi_heads

from ..language_backbone import build_language_backbone
from transformers import AutoTokenizer

import random
import timeit
import pdb
from copy import deepcopy


def random_word(input_ids, mask_token_id, vocabs, padding_token_id, greenlight_map):
        output_label = deepcopy(input_ids)
        for j in range(input_ids.size(0)):
                for i in range(input_ids.size(1)):
                        prob = random.random()
                        ratio = 0.15
                        if greenlight_map is not None and greenlight_map[j,i] == -1:
                                output_label[j,i] = -100
                                continue

                        if (not input_ids[j,i] == padding_token_id) and prob < ratio:
                                prob /= ratio
```

```
                                if  prob  <  0.8:
                                        input_ids[j,i]  =  mask_token_id

                                elif  prob  <  0.9:
                                        input_ids[j,i]  =  random.choice(vocabs)

                        else:
                                output_label[j,i]  =  -100

                        if  greenlight_map  is  not  None  and  greenlight_map[j,i]  !=  1:
                                output_label[j,i]  =  -100
        return  input_ids,  output_label


class  GeneralizedVLRCNN(nn.Module):
        def  __init__(self,  cfg):
                super(GeneralizedVLRCNN,  self).__init__()
                self.cfg  =  cfg
                self.backbone  =  build_backbone(cfg)

                if  cfg.MODEL.LANGUAGE_BACKBONE.TOKENIZER_TYPE  ==  "clip":
                        from  transformers  import  CLIPTokenizerFast
                        if  cfg.MODEL.DYHEAD.FUSE_CONFIG.MLM_LOSS:
                                print("Reuse  token  'ðŁĵij</w>'  (token_id  =  49404)  for  mask  token!")
                                self.tokenizer  =  CLIPTokenizerFast.from_pretrained("openai/clip-vit-base-patch32",
                                                                                     from_slow=True,  mask_token='ðŁĵij</w>')
                        else:
                                self.tokenizer  =  CLIPTokenizerFast.from_pretrained("openai/clip-vit-base-patch32",
                                                                                     from_slow=True)
                else:
                        self.tokenizer  =  AutoTokenizer.from_pretrained(cfg.MODEL.LANGUAGE_BACKBONE.TOKENIZER_TYPE)
                self.tokenizer_vocab  =  self.tokenizer.get_vocab()
                self.tokenizer_vocab_ids  =  [item  for  key,  item  in  self.tokenizer_vocab.items()]

                self.language_backbone  =  build_language_backbone(cfg)

                self.rpn  =  build_rpn(cfg)
```

```
        self.roi_heads = build_roi_heads(cfg)
        self.DEBUG = cfg.MODEL.DEBUG

        self.freeze_backbone = cfg.MODEL.BACKBONE.FREEZE
        self.freeze_fpn = cfg.MODEL.FPN.FREEZE
        self.freeze_rpn = cfg.MODEL.RPN.FREEZE
        self.add_linear_layer = cfg.MODEL.DYHEAD.FUSE_CONFIG.ADD_LINEAR_LAYER

        self.force_boxes = cfg.MODEL.RPN.FORCE_BOXES

        if cfg.MODEL.LINEAR_PROB:
            assert cfg.MODEL.BACKBONE.FREEZE, "For linear probing, backbone should be frozen!"
            if hasattr(self.backbone, 'fpn'):
                assert cfg.MODEL.FPN.FREEZE, "For linear probing, FPN should be frozen!"
        self.linear_prob = cfg.MODEL.LINEAR_PROB
        self.freeze_cls_logits = cfg.MODEL.DYHEAD.FUSE_CONFIG.USE_DOT_PRODUCT_TOKEN_LOSS
        if cfg.MODEL.DYHEAD.FUSE_CONFIG.USE_DOT_PRODUCT_TOKEN_LOSS:
            if hasattr(self.rpn.head, 'cls_logits'):
                for p in self.rpn.head.cls_logits.parameters():
                    p.requires_grad = False

        self.freeze_language_backbone = self.cfg.MODEL.LANGUAGE_BACKBONE.FREEZE
        if self.cfg.MODEL.LANGUAGE_BACKBONE.FREEZE:
            for p in self.language_backbone.parameters():
                p.requires_grad = False

        self.use_mlm_loss = cfg.MODEL.DYHEAD.FUSE_CONFIG.MLM_LOSS
        self.mlm_loss_for_only_positives = cfg.MODEL.DYHEAD.FUSE_CONFIG.MLM_LOSS_FOR_ONLY_POSITIVES

    def train(self, mode=True):
        super(GeneralizedVLRCNN, self).train(mode)
        if self.freeze_backbone:
            self.backbone.body.eval()
            for p in self.backbone.body.parameters():
                p.requires_grad = False
        if self.freeze_fpn:
            self.backbone.fpn.eval()
```

```
                for p in self.backbone.fpn.parameters():
                    p.requires_grad = False
        if self.freeze_rpn:
            if hasattr(self.rpn, 'head'):
                self.rpn.head.eval()
            for p in self.rpn.parameters():
                p.requires_grad = False
        if self.linear_prob:
            if self.rpn is not None:
                for key, value in self.rpn.named_parameters():
                    if not ('bbox_pred' in key or 'cls_logits' in key or 'centerness' in key or 'cosine_scale'
                        value.requires_grad = False
            if self.roi_heads is not None:
                for key, value in self.roi_heads.named_parameters():
                    if not ('bbox_pred' in key or 'cls_logits' in key or 'centerness' in key or 'cosine_scale'
                        value.requires_grad = False
        if self.freeze_cls_logits:
            if hasattr(self.rpn.head, 'cls_logits'):
                self.rpn.head.cls_logits.eval()
                for p in self.rpn.head.cls_logits.parameters():
                    p.requires_grad = False
        if self.add_linear_layer:
            if self.rpn is not None:
                for key, p in self.rpn.named_parameters():
                    if 'tunable_linear' in key:
                        p.requires_grad = True

        if self.freeze_language_backbone:
            self.language_backbone.eval()
            for p in self.language_backbone.parameters():
                p.requires_grad = False

    def forward(self,
            images,
            targets=None,
            captions=None,
            positive_map=None,
```

```
        greenlight_map=None):
        if self.training and targets is None:
                raise ValueError("In training mode, targets should be passed")
        images = to_image_list(images)
        device = images.tensors.device

        language_dict_features = {}
        if captions is not None:

                tokenized = self.tokenizer.batch_encode_plus(captions,
                                                             max_length=self.cfg.MODEL.LANGUAGE_BACKBONE.MAX_QUERY_LEN,
                                                             padding='max_length' if self.cfg.MODEL.LANGUAGE_BACKBONE.PAD_MAX else
                                                             return_special_tokens_mask=True,
                                                             return_tensors='pt',
                                                             truncation=True).to(device)
                if self.use_mlm_loss:
                        if not self.mlm_loss_for_only_positives:
                                greenlight_map = None
                        input_ids, mlm_labels = random_word(
                                input_ids=tokenized.input_ids,
                                mask_token_id=self.tokenizer.mask_token_id,
                                vocabs=self.tokenizer_vocab_ids,
                                padding_token_id=self.tokenizer.pad_token_id,
                                greenlight_map=greenlight_map)
                else:
                        input_ids = tokenized.input_ids
                        mlm_labels = None

                tokenizer_input = {"input_ids": input_ids,
                                                "attention_mask": tokenized.attention_mask}

                if self.cfg.MODEL.LANGUAGE_BACKBONE.FREEZE:
                        with torch.no_grad():
                                language_dict_features = self.language_backbone(tokenizer_input)
                else:
                        language_dict_features = self.language_backbone(tokenizer_input)
```

```
        if self.cfg.DATASETS.ONE_HOT:
            new_masks = torch.zeros_like(language_dict_features['masks'],
                                         device=language_dict_features['masks'].device)
            new_masks[:, :self.cfg.MODEL.DYHEAD.NUM_CLASSES] = 1
            language_dict_features['masks'] = new_masks

        if self.cfg.MODEL.LANGUAGE_BACKBONE.MASK_SPECIAL:
            language_dict_features["masks"] = 1 - tokenized.special_tokens_mask

        language_dict_features["mlm_labels"] = mlm_labels

    swint_feature_c4 = None
    if 'vl' in self.cfg.MODEL.SWINT.VERSION:
        inputs = {"img": images.tensors, "lang": language_dict_features}
        visual_features, language_dict_features, swint_feature_c4 = self.backbone(inputs)
    else:
        visual_features = self.backbone(images.tensors)

    if targets:
        targets = [target.to(device)
                   for target in targets if target is not None]

    if self.force_boxes:
        proposals = []
        for t in targets:
            tb = t.copy_with_fields(["labels"])
            tb.add_field("scores", torch.ones(tb.bbox.shape[0], dtype=torch.bool, device=tb.bbox.device))
            proposals.append(tb)
        if self.cfg.MODEL.RPN.RETURN_FUSED_FEATURES:
            _, proposal_losses, fused_visual_features = self.rpn(
                    images, visual_features, targets, language_dict_features,
                    positive_map, captions, swint_feature_c4)
        elif self.training:
            null_loss = 0
            for key, param in self.rpn.named_parameters():
                null_loss += 0.0 * param.sum()
            proposal_losses = {('rpn_null_loss', null_loss)}
```

```
else:
        proposals, proposal_losses, fused_visual_features = self.rpn(images, visual_features, targets, language_dict_featu
                                                                     captions, swint_feature_c4)
if self.roi_heads:
        if self.cfg.MODEL.ROI_MASK_HEAD.PREDICTOR.startswith("VL"):
                if self.training:
                        assert len(targets) == 1 and len(targets[0]) == len(positive_map), "shape match assert for ma
                        targets[0].add_field("positive_map", positive_map)
        if self.cfg.MODEL.RPN.RETURN_FUSED_FEATURES:
                x, result, detector_losses = self.roi_heads(
                        fused_visual_features, proposals, targets,
                        language_dict_features=language_dict_features,
                        positive_map_label_to_token=positive_map if not self.training else None
                )
        else:
                x, result, detector_losses = self.roi_heads(
                        visual_features, proposals, targets,
                        language_dict_features=language_dict_features,
                        positive_map_label_to_token=positive_map if not self.training else None
                )
else:
        x = visual_features
        result = proposals
        detector_losses = {}

if self.training:
        losses = {}
        losses.update(detector_losses)
        losses.update(proposal_losses)
        return losses

return result
```

Results

In this section, you should finish training your model training or loading your trained model. That is a great experiment! You should share the results with others with necessary metrics and figures.

Please test and report results for all experiments that you run with:

- specific numbers (accuracy, AUC, RMSE, etc)
- figures (loss shrinkage, outputs from GAN, annotation or label of sample pictures, etc)

```
def  evaluate(self):
        """
        Run  per  image  evaluation  on  given  images  and  store  results
        (a  list  of  dict)  in  self.eval_imgs.
        """

        self.params.img_ids  =  list(np.unique(self.params.img_ids))

        if  self.params.use_cats:
                cat_ids  =  self.params.cat_ids
```