```
import torch.nn.functional as F
from torch.utils.data.dataloader import DataLoader
from torch.optim import AdamW
import pandas as pd
from tqdm import tqdm
import numpy as np
import logging
from sklearn.metrics import precision_recall_fscore_support
import random
import os
import sys
from transformers import AutoConfig, AutoModel, AutoTokenizer, PreTrainedModel
from torch import nn
import torch
from transformers import get_linear_schedule_with_warmup, Trainer,
TrainingArguments, HfArgumentParser
from datasets import load_dataset
from collections.abc import Mapping
from typing import TYPE_CHECKING, Any, Callable, Dict, List, Optional, Tuple,
Union
import itertools


# set up logging info
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)


def set_seed(seed):
    """ Set all seeds to make results reproducible """
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmark = False
    np.random.seed(seed)
    random.seed(seed)
    os.environ['PYTHONHASHSEED'] = str(seed)


class Training_args:
    def __init__(self,
            model_name,
            bs: int=16,
            train_epoch: int=20,
            gradient_accumulation_step: int=1,
```

```
            output_dir: str='models/sup/',
            lr: float=3e-5,
            device: str='cuda',
            warmup_proportion: float=0.1,
        temp: float=0.05,
            ):
        self.model_name = model_name
        self.bs = bs
        self.train_epoch = train_epoch
        self.gradient_accummulation_step = gradient_accumulation_step
        self.output_dir = output_dir
        self.lr = lr
        self.device = device
        self.warmup_proportion = warmup_proportion
        self.temp = temp
        # self.infer_bs = infer_bs
        # self.eval_method = eval_method
        # self.eval_split = eval_split


class Set:
    def __init__(self, data_path=None):
        self.data_path = data_path
        self.embedding = None
        if self.data_path:
            dataset = load_dataset('csv', data_files=self.data_path)
            key = list(dataset.keys())[0]
            self.dataset = dataset[key]

    def get_embedding(self, tokenizer, model):
        logger.info("**** Running model to get embeddings ****")
        model.to('cuda')
        dataloader = DataLoader(self.dataset, batch_size=8, shuffle=False)
        embedding_ls = []
        for step, data in enumerate(tqdm(dataloader)):
            inputs = tokenizer(data['utterance'], truncation=True, padding=True,
return_tensors='pt')
            inputs.to('cuda')
            # emb = model(**inputs, output_hidden_states=True,
return_dict=True).pooler_output.clone().detach().to('cpu')
            emb = model(**inputs, output_hidden_states=True,
return_dict=True).last_hidden_state[:, 0, :].clone().detach().to('cpu')
            embedding_ls.append( emb )
        self.embedding = torch.cat(embedding_ls)
```

```
    def intersect(self, set2, tokenizer, model, top_k=None, debug=False):
        if self.embedding is None or set2.embedding is None:
            if self.embedding is None:
                self.get_embedding(tokenizer, model)
            if set2.embedding is None:
                set2.get_embedding(tokenizer, model)
        # do intersection
        # the only method offered is to measure similarity using cosine
similarity
        cossim_mat = F.normalize(self.embedding) @
F.normalize(set2.embedding).t()
        if isinstance(top_k, int):
            cossim_k, idx_set2 = torch.topk(cossim_mat, top_k, dim=1)
            cossim = cossim_k.mean(dim=1)
        elif top_k == 'all':
            cossim = cossim_mat.mean(dim=1)
        idx_to_set2 = np.argsort(-cossim).tolist()
        if debug==True:
            return (idx_to_set2, cossim)
        return idx_to_set2

    def difference(self, set2, tokenizer, model, top_k=None, debug=False):
        if self.embedding is None or set2.embedding is None:
            if self.embedding is None:
                self.get_embedding(tokenizer, model)
            if set2.embedding is None:
                set2.get_embedding(tokenizer, model)
        # do difference
        # the only method offered here is to measure similarity using cosine
similarity
        cossim_mat = - ( F.normalize(self.embedding) @
F.normalize(set2.embedding).t() )
        if isinstance(top_k, int):
            cossim_k, idx_set2 = torch.topk(cossim_mat, top_k, dim=1)
            cossim = cossim_k.mean(dim=1)
        elif top_k == 'all':
            cossim = cossim_mat.mean(dim=1)
        idx_to_set2 = np.argsort(-cossim).tolist()
        if debug==True:
            return (idx_to_set2, cossim)
        return idx_to_set2



class SetCSETrainer:
```

```
    def __init__(self, training_args, local_files_only=False):
        self.args = training_args
        self.local_files_only=local_files_only
        self.tokenizer = AutoTokenizer.from_pretrained(self.args.model_name,
local_files_only=self.local_files_only)
        self.model = AutoModel.from_pretrained(self.args.model_name,
local_files_only=self.local_files_only)

    def interset_loss(self, embeddings_ls):
        temp = self.args.temp
        comb_emb = list(itertools.combinations(embeddings_ls, 2))
        loss = 0.0
        for emb_a, emb_b in comb_emb:
            emb_a = F.normalize(emb_a)
            emb_b = F.normalize(emb_b)
            mat = torch.mm(emb_a, emb_b.T)
            # loss += torch.logsumexp( mat / temp, (0,1) )
            # loss += torch.logsumexp( mat / temp, 1 ).mean()
            loss += torch.mean( mat / temp, dim=(0,1) )
        return loss

    def SetCSEtrain(self, set_ls):
        """
        This function will cover the whole training process.
        """
        logging.basicConfig(level=logging.DEBUG)
        logger = logging.getLogger(__name__)

        # determine bs size and prepare dataloader
        n = len(set_ls)
        size_ls = []
        bs_ls = []
        for s in set_ls:
            size_ls.append(len(s.dataset))
        bs_ls = [int( x*self.args.bs/sum(size_ls) ) for x in size_ls]
        dataloader_ls = []
        step_ls = []
        for i in range(n):
            dataloader = DataLoader(set_ls[i].dataset, batch_size=bs_ls[i],
shuffle=True)
            dataloader_ls.append(dataloader)
            step_ls.append(len(dataloader))

        # compute the training steps
```

```
        train_steps = int( size_ls[0] * self.args.train_epoch / bs_ls[0] /
self.args.gradient_accummulation_step )
        warmup_steps = int( train_steps * self.args.warmup_proportion )

        # prepare optimizer and scheduler
        param_optimizer = list(self.model.named_parameters())
        no_decay = ['bias','LayerNorm.weight']
        optimizer_grouped_parameters = [
            {'params': [p for n, p in param_optimizer if not any(nd in n for nd
in no_decay)], 'weight_decay': 0.01},
            {'params': [p for n, p in param_optimizer if any(nd in n for nd in
no_decay)], 'weight_decay': 0.0}]
        optimizer = AdamW(optimizer_grouped_parameters, lr=self.args.lr, eps=1e-
8)
        scheduler = get_linear_schedule_with_warmup(
                optimizer,
                num_warmup_steps=warmup_steps,
                num_training_steps=train_steps)

        # train!
        logger.info("***** Running training *****")
        logger.info("  Num Steps = %d", train_steps)

        self.model.to(self.args.device)
        self.model.train()
        running_loss = []
        for epoch in range(self.args.train_epoch):
            epoch_loss = 0.0
            iter_ls = []
            for dl in dataloader_ls:
                iter_ls.append(iter(dl))

            for step in range(max(step_ls)):
                embeddings_ls = []
                for it in iter_ls:
                    try:
                        data = next(it)
                        inputs = self.tokenizer(data['utterance'],
truncation=True, padding=True, return_tensors='pt')
                        inputs.to(self.args.device)
                        embeddings_ls.append(self.model(**inputs).last_hidden_sta
te[:, 0, :])
                    except StopIteration:
                        pass
```

```
            loss = self.interset_loss(embeddings_ls)
            loss.backward()

            if (step+1) % self.args.gradient_accummulation_step == 0:
                optimizer.step()
                scheduler.step()
                self.model.zero_grad()

            running_loss.append(loss.clone().detach().to('cpu').numpy())
            epoch_loss += float(loss)

        logger.info(f'The loss of {epoch}-th epoch is {epoch_loss}')

    # save model
    self.model.save_pretrained(self.args.output_dir)

    logger.info("Training completed and model successfully saved!")
```