```
from transformers import AutoTokenizer, AutoModel, TrainingArguments, Trainer
import pandas as pd
from datasets import load_dataset
from transformers import DataCollatorWithPadding
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data.dataloader import DataLoader
from torch.optim import AdamW
from transformers import get_linear_schedule_with_warmup, Trainer,
TrainingArguments, HfArgumentParser
import logging
from sklearn import preprocessing
from tqdm import tqdm
import torch
from sklearn.metrics import precision_recall_fscore_support
import numpy as np
import string
import random
import argparse
import os


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)

def id_generator(size=6):
    return ''.join(random.choice(string.ascii_uppercase + string.digits) for _ in
range(size))

class Training_args:
    def __init__(self,
            model_name,
            bs: int=16,
            infer_bs: int=32,
            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,
            n_intent: int=3,
            ):
        self.model_name = model_name
        self.bs = bs
        self.infer_bs = infer_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.n_intent = 3

class clf_model(nn.Module):
    def __init__(self, model_name, n_label):
        super().__init__()
        # a manual fix for loading Huggingface model locally
        if 'MCSE' in model_name:
            self.embedding_model = AutoModel.from_pretrained(model_name,
local_files_only=True)
        else:
            self.embedding_model = AutoModel.from_pretrained(model_name)
        n_emb = self.embedding_model.pooler.dense.out_features
        self.fc = nn.Linear(n_emb, n_label)

    def forward(self, inputs):
        x = self.embedding_model(**inputs).last_hidden_state[:, 0, :]
        x = self.fc(x)
        return x

class CLF_trainer:
    def __init__(self, training_args):
        self.args = training_args
        self.tokenizer = AutoTokenizer.from_pretrained(self.args.model_name)
        self.model = clf_model(self.args.model_name, self.args.n_intent)
        self.label_enc = preprocessing.LabelEncoder()

    def compute_loss(self, outputs, targets):
        criterion = nn.CrossEntropyLoss()
        loss = criterion(outputs, targets)
        return loss

    def train(self, dataset):
```

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

        dataloader = DataLoader(dataset['train'], batch_size=self.args.bs,
shuffle=True)
        size = len(dataset['train'])

        # compute the training steps
        train_steps = int( size * self.args.train_epoch / self.args.bs /
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
            for step, data in enumerate(tqdm(dataloader)):
                inputs = self.tokenizer(data['utterance'], truncation=True,
padding=True, return_tensors='pt')
                inputs.to(self.args.device)
                outputs = self.model.forward(inputs)
                targets = self.label_enc.fit_transform(data['label'])
                targets = torch.as_tensor(targets).to(self.args.device)
                loss = self.compute_loss(outputs, targets)
```

```
                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!")


    def evaluate_topn(self, dataset, df_test):
        # set up logger
        logging.basicConfig(level=logging.INFO)
        logger = logging.getLogger(__name__)

        dataloader = DataLoader(dataset['test'], batch_size=self.args.infer_bs,
shuffle=False)
        size = len(dataset['test'])

        # compute the training steps
        test_steps = int( size / self.args.infer_bs /
self.args.gradient_accummulation_step )

        # evaluate!
        logger.info("***** Running evaluation *****")
        logger.info("  Num Steps = %d", test_steps)

        self.model.to(self.args.device)

        logits_ls = []
        for step, data in enumerate(tqdm(dataloader)):
            inputs = self.tokenizer(data['utterance'], truncation=True,
padding=True, return_tensors='pt')
            inputs.to(self.args.device)
            logits =
self.model.forward(inputs).softmax(dim=1).clone().detach().to('cpu')
            logits_ls.append(logits)
```

```
        logits = torch.cat(logits_ls)
        labels = list(self.label_enc.classes_)
        n_count =
df_test.label.value_counts()[list(self.label_enc.classes_)].to_list()

        dfi_ls = []
        for i in range(len(n_count)):
            idx = torch.topk(logits[:,i], n_count[i]).indices.tolist()
            dfi = df_test.iloc[idx]
            dfi['pred'] = labels[i]
            dfi_ls.append(dfi)
        df_res = pd.concat(dfi_ls)
        res = precision_recall_fscore_support(df_res['label'].to_list(),
df_res['pred'].to_list(), average='weighted')
        acc = df_res[df_res['label']==df_res['pred']].shape[0]/df_res.shape[0]
        return (acc,) + res[:3]

    def evaluate(self, dataset, df_test):
        # set up logger
        logging.basicConfig(level=logging.INFO)
        logger = logging.getLogger(__name__)

        dataloader = DataLoader(dataset['test'], batch_size=self.args.infer_bs,
shuffle=False)
        size = len(dataset['test'])

        # compute the training steps
        test_steps = int( size / self.args.infer_bs /
self.args.gradient_accummulation_step )

        # evaluate!
        logger.info("***** Running evaluation *****")
        logger.info("  Num Steps = %d", test_steps)

        self.model.to(self.args.device)

        logits_ls = []
        for step, data in enumerate(tqdm(dataloader)):
            inputs = self.tokenizer(data['utterance'], truncation=True,
padding=True, return_tensors='pt')
            inputs.to(self.args.device)
            logits =
self.model.forward(inputs).softmax(dim=1).clone().detach().to('cpu')
            logits_ls.append(logits)
        logits = torch.cat(logits_ls)
```

```
        pred = list(self.label_enc.inverse_transform(logits.argmax(dim=1)))
        df_test['pred'] = pred
        res = precision_recall_fscore_support(df_test['label'].to_list(),
df_test['pred'].to_list(), average='weighted')
        acc =
df_test[df_test['label']==df_test['pred']].shape[0]/df_test.shape[0]
        return (acc,) + res[:3]

    def pred(self, text):
        self.model.to(self.args.device)
        inputs = self.tokenizer(text, truncation=True, padding=True,
return_tensors='pt')
        inputs.to(self.args.device)
        logits =
self.model.forward(inputs).softmax(dim=1).clone().detach().to('cpu')
        print(logits)

def evaluate(model_name, data_path, n_intent=3, n_sample=20, train_epoch=20,
temp_code='abcd'):
    df = pd.read_csv(data_path)
    intents = list(df.label.value_counts().index)
    intents = intents[:n_intent]
    pool = []
    dfi_ls = []
    for intent in intents:
        dfi = df[df['label']==intent].sample(frac=1)
        sample = dfi.iloc[:n_sample]
        dfi_ls.append(sample)
        pool.append(dfi.iloc[n_sample:])
    train = pd.concat(dfi_ls).sample(frac=1)
    test = pd.concat(pool).sample(frac=1)
    train.to_csv(f'data/sup_clf/temp_{temp_code}/train.csv',index=False)
    test.to_csv(f'data/sup_clf/temp_{temp_code}/test.csv',index=False)
    dataset = load_dataset('csv',
data_files={'train':f'data/sup_clf/temp_{temp_code}/train.csv',
'test':f'data/sup_clf/temp_{temp_code}/test.csv'})
    df_test = pd.read_csv(f'data/sup_clf/temp_{temp_code}/test.csv')

    args = Training_args(model_name, bs=16, train_epoch=train_epoch)
    trainer = CLF_trainer(args)
    trainer.train(dataset)
    res = trainer.evaluate_topn(dataset, df_test)
    return res
```

```
torch.cuda.empty_cache()

# Step 1. Read arguments:
parser = argparse.ArgumentParser()
parser.add_argument("-mn", "--model_name", default='princeton-nlp/sup-simcse-
bert-base-uncased', help="Name of the model for evaluation")
parser.add_argument("-dp", "--data_path", default='data/fb/en_all.csv',
help="Dataset for evaluation")
parser.add_argument("-nr", "--n_rep", default=5, help="Number to times to repeat
each experiment.")
parser.add_argument("-te", "--train_epoch", default=60, help="Number of training
epoch.")
parser.add_argument("-ni", "--n_intent", default=3, help="Number of sample set.")
parser.add_argument("-ns", "--n_sample", default=20, help="Sample set size.")
args = parser.parse_args()

data_path = args.data_path
df = pd.read_csv(data_path)
# ['data/fb/en_all.csv', 'data/agnews/title_3000.csv',
'data/agnews/description_3000.csv', 'data/FPB/all.csv']
# ["princeton-nlp/sup-simcse-bert-base-uncased", "princeton-nlp/sup-simcse-
roberta-base","voidism/diffcse-bert-base-uncased-sts","voidism/diffcse-roberta-
base-sts"]

n_rep = int(args.n_rep)
n_sample = int(args.n_sample)
n_intent = int(args.n_intent)
top_k = n_sample
train_epoch = int(args.train_epoch)
model_name = args.model_name



# Step 2. Make data and model temp directory:
temp_code = id_generator(4)
data_tempdir = 'data/sup_clf/temp_' + temp_code
if not os.path.exists(data_tempdir):
    os.makedirs(data_tempdir)


# Step 3. Do evaluations
set_seed(43)

res = np.zeros([n_rep, 4])
for i in range(n_rep):
```

```
    res[i] = evaluate(model_name=model_name, data_path=data_path,
n_sample=n_sample, n_intent=n_intent, train_epoch=train_epoch,
temp_code=temp_code)
    print(f'Unsup finished: {i+1}!')

print('supervised classification:')
res = pd.DataFrame(res, columns=['acc', 'precision', 'recall', 'F1'])
print(res)
res = res.mean().to_frame('Sup_CLF').T
print(res)
```