Table of Contents

1 实验内容

谣言识别:

- 将谣言识别验证集上的准确率（第四次课程 PPT 第 82 页）提升至 88.0% 以上。方法不限，可使用本课程或课程外的方法，也可使用预训练模型和任何 package。

- 在代码中实现 K-Fold 交叉验证（k=5）;

2 数据处理

2.1 微博谣言数据集介绍

数据集链接：微博谣言数据集

整个数据集共包含从 2009 年 9 月 4 日至 2017 年 6 月 12 日的 31669 条谣言，同时，还额外存放了每条留言的评论与转发信息，本作业仅用到了 CED_dataset 中的 3377 条谣言，且只使用谣言本身的信息，对于其评论和转发信息并没有加以使用。

2.2 数据集划分

本作业对 CED_dataset 的 3377 条留言数据集划分成了 train、val、test 三个部分，比例为 2:1:1，同时，为了避免划分时出现类别不平衡的问题，本作业通过代码严格限制了谣言非谣言的比例要在 [2/3，3/2] 之间，代码见Appendix.A

3 实验方法

3.1 GloVe + fc 层（仅使用当前数据集）

本方法先采用 GloVe 的预训练方法，在当前数据集上进行预训练，GloVe 是基于 co-occurrence matrix 的预训练方法，将词共现建模为回归问题，即回归其共现次数的 log，以此方法学习到一个不错的 embeddding 矩阵。本文采用了两种方法去作预训练，一种是用共现次数本身的 log 作为回归目标，一种是用共现次数 * 1/与中心词距离的 log 作为回归目标。

在进行谣言预测训练时，模型加载 GloVe 的 center 和 context 两个 embedding 矩阵，并利用共现矩阵的对称性，将二者进行相加**（本质上是两个初始化不同的模型的集成）**，再接以 fc 层、GAP 层、fc 层进行分类。　　为了进一步提升模型性能，本文训练了 output 维度为 16、32、64、96 的 GloVe embedding 矩阵，并相应的接上不同的 fc 层；最终，用一个可学习的 linear 层对所有模型进行加权集成，得到的性能如表 1所示：

其中，dw 表示 distence weight，train&val 表示同时用 train 和 val 数据集用来训练模型。网络结构代码见Appendix.B，GloVe 代码见提交的 rumo_glove.ipynb 或个人 github 仓库GloVe 仓库地址

	single(best)	ensemble	train&val ensemble
w/o dw	87.35	86.64	87.71
w dw	87.23	86.87	88.29

Table 1: GloVe + fc 的 test acc

3.2 Word2Vec + fc + cross-val

本方法采用其他大规模中文数据集训练得到的 word2vec embedding 矩阵，用交叉验证比较其得分，从而选择加载哪个预训练的 word2vec embedding。其中，大语料库预训练的 word2vec 来自于大语料库预训练的 word2vec 仓库。这里选用了用微博、搜狗、百度百科、多个大语料库合并这四种语料库训练的 word2vec 矩阵作为预训练模型。

对于分词器和词表的问题，这里分词器依旧采用 jieba 的 HMM 进行分词，词表也依旧采用之前的词表，每个 token 的 embedding 是通过查询 word2vec 的词表得到的，所以也会有一小部分词无法通过查询得到，则是随机初始化得到，查询情况大约 90% 的词都可以通过查询得到，仅 10% 不到的词是随机初始化得到。　　接着，本工作将训练集作同样的 5-fold 划分，比较这四者的交叉验证得分，代码如Appendix.C或者提交的 big_corpus_word2vec.ipynb 文件所示，具体结果如表 2所示（dropout 均采用 0.5）

	微博	搜狗	百度百科	多个大语料库合并
5-fold	87.20	86.67	86.84	87.26

Table 2: 交叉验证平均得分

接着，这里使用了最好的多语料库合并的模型，得到如表 3所示结果：

	train	train&val
acc	86.99	87.59

Table 3: 多个语料库合并的 word2vec 的性能

其中，train&val 代表同时使用训练集和验证集进行训练。

3.3 预训练的 BERT 用来作 embedding

word2vec 以及 GloVe 有一个共同的弊端，即对于多义词无法得到多个语义向量，在 ELMo 中采用双向 RNN 使用双向语言模型作预训练一定程度上解决了这一问题，但 ELMo 是 task-specific 的，而后续的 GPT 和 BERT 解决了这一问题，它们都可以通过 token 的设计用于诸多的下游任务，一定程度上是 task-agnostic 的。本作业使用了两个开源仓库：bert 开源仓库 1和专门针对中文特点使用 whold word mask（wwm）的bert+wwm 仓库。

这里分词器和词表均采用 BERT 给定的，同时，本作业计算了过滤标点与否的分词后的最大词数，过滤标点的最大词数为 135，不过滤标点的最大词数为 171。

因此，有两个关键的超参数，就是 max_len 和是否过滤标点 filtered, 它决定了模型是否能够将文本看全、一些特殊标点包含的情感信息等是否看到，所以这里对二者进行了讨论，具体结果如表 4所示：

	100+filtered	**150+filtered**	**150+w/o filtered**
BERT	89.83	90.90	**91.96**
BERT-wwm	89.91	91.02	91.37

Table 4: 对 max—len 以及是否过滤标点的讨论

对于 BERT 迁移到谣言预测下游任务的方式，这里对 linear-probing 和 fine-tuning 两种方式均进行了实验，结果如表 5所示：**(这里均采用 150+filtered 的设定)**

	linear_probing	**fine-tuning**
BERT	71.51	90.90
BERT-wwm	71.51	91.02

Table 5: linear-probing & fine-tuning

4 最佳性能：val acc: 92.32%, test acc: 91.96%

Appendix

A 数据集划分类别控制代码

```
# 测试一下划分情况
while True:
    tag = True
    for key, indices in split.items():
        count = [0, 0]
        for idx in indices:
            count[labels[idx]] += 1
        # 如果类别过于不平衡，则重新随机化
        cls_ratio = count[0]/count[1]
        ratio_threshold = 1.5
        if cls_ratio > ratio_threshold or cls_ratio < 1/ratio_threshold:
            indices = np.random.permutation(len(bow))
            tag = False
            break
        print(key, '非谣言有{}条，谣言有{}条'.format(count[0], count[1]))
        if key==2:
            tag = True
    if tag:
        break
```

B GloVe 的模型结构与模型集成

```
def init_weights(m,mode='zero'):
    if type(m) == nn.Embedding:
        if mode =='xavier':
            nn.init.xavier_normal_(m.weight)
        else:
            nn.init.zeros_(m.weight)

class GloVe(nn.Module):
    def __init__(self,vocab_size,embedding_size) -> None:
        super(GloVe,self).__init__()
        self.context = nn.Embedding(vocab_size,embedding_size)
        self.center = nn.Embedding(vocab_size,embedding_size)
        self.context_bias = nn.Embedding(vocab_size,1)
        self.center_bias = nn.Embedding(vocab_size,1)
        init_weights(self.context,mode='xavier')
        init_weights(self.center,mode='xavier')
        init_weights(self.center_bias)
        init_weights(self.context_bias)


    def forward(self,center, all_contexts): # input.shape: B,N,vocab_size
        bs,max_len = all_contexts.shape
        contexts = self.context(all_contexts) # shape (B,N_context,embedding_size)
        centers = self.center(center) # shape (B,1,embedding_size)
        contexts_bias = self.context_bias(all_contexts) # shape (B,N_context,1)
        centers_bias = self.center_bias(center) # shape (B,1,1)
        similarity = centers @ contexts.transpose(1,2) # shape:[B,1,N_context]
        similarity = similarity.reshape(bs,max_len)
```

```
        centers_bias = centers_bias.reshape(bs,1)
        contexts_bias = contexts_bias.reshape(bs,max_len)
        output = contexts_bias + similarity + centers_bias
        return output

class Pretrainedembs_MLP(nn.Module):
    def __init__(self,embs_size,vocab_size,pad_index,dropout,use_distance_weight=False):
        super().__init__()
        self.word_embs = nn.Embedding(vocab_size,embs_size)
        self.mlp = nn.Sequential(
            # nn.Conv1d(embs_size,embs_size,1,1,0),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Conv1d(embs_size,2,1,1,0)
        )
        self.pad_index = pad_index
        self.GAP = nn.AdaptiveAvgPool1d(1)
        if not use_distance_weight:
            model_path = f'./model_vocab_glove_for_mlp/model_{embs_size}.pth'
            with open(model_path,'rb') as f:
                glove = torch.load(f)
            self.word_embs.weight.data[:-1,:] = glove.context.weight.data +\
            glove.center.weight.data
        else:
            glove_path = f'./model_vocab_glove_for_mlp_dw/model_{embs_size}.pth'
            with open(glove_path,'rb') as f:
                glove = torch.load(f)
            self.word_embs.weight.data[:-1,:] = glove.context.weight.data + \
            glove.center.weight.data

    def forward(self,x,log_softmax=True):
        bs = x.shape[0]
        pad_mask = x!=self.pad_index
        embed = self.word_embs(x)
        embed = embed.permute(0,2,1).contiguous()
        cls = self.GAP(self.mlp(embed)*pad_mask[:,None,:]).view(bs,2)
        if log_softmax:
            cls = F.log_softmax(cls,dim=1)
        return cls
class Model_Ensemble(nn.Module):
    def __init__(self,models):
        super().__init__()
        self.model_lst = models
        self.linear_weight = nn.Linear(2*len(self.model_lst),2,bias=False)
    def forward(self,x):
        bs = x.shape[0]
        scores = []
        for model in self.model_lst:
            model.eval()
            scores.append(model(x,log_softmax=False))
        finale_score = self.linear_weight(torch.cat(scores,dim=1))
        return finale_score
```

C Word2Vec 的模型以及交叉验证

```
# 导入word2vec文件
```

```
word2vec_dir = '/media/charon/ubuntu_data/cn_pretrained_model'
file_lst = os.listdir(word2vec_dir)
filtered_file_lst = [file for file in file_lst if Path(file).suffix != '.py']
word2vec_param_lst = []
for filename in filtered_file_lst:
    count = 0
    with open(os.path.join(word2vec_dir,filename),'r') as f:
        for idx,line in enumerate(f):
            if not idx:
                num_words,num_embed = [int(e) for e in line.split(' ')]
                word2vec_param = torch.randn((len(vocab),num_embed),dtype=torch.float32)
                continue

            l = line.split(' ')
            word,vec = l[0],[float(e) for e in l[1:-1]]
            if word in vocab.keys():
                idx = vocab[word]
                word2vec_param[idx,:] = torch.tensor(vec,dtype=torch.float32)
                count += 1
    print(f'pretrained model filename {filename}')
    print('num of words not in the pre-training corpus : ',len(vocab)-count)
    print('num of words in the pre-training corpus : ',count)
    word2vec_param_lst.append(word2vec_param)

class Word2Vec_Cls(nn.Module):
    def __init__(self,vocab_size,pad_index,dropout,word2vec_param):
        super().__init__()
        vec_size = word2vec_param.shape[1]
        self.word_embs = nn.Embedding(vocab_size,vec_size)
        self.mlp = nn.Sequential(
            nn.LayerNorm(vec_size),
            nn.ReLU(),
            nn.Dropout(dropout),
            nn.Linear(vec_size,2)
        )
        self.pad_index = pad_index
        self.GAP = nn.AdaptiveAvgPool1d(1)
        self.word_embs.weight.data[:-1,:] = word2vec_param
        self.word_embs.weight.data[-1:-2,:] = torch.zeros((1,vec_size),dtype=torch.float32)

    def forward(self,x,log_softmax=True):
        bs = x.shape[0]
        embed = self.word_embs(x)
        cls = self.GAP(self.mlp(embed).permute(0,2,1).contiguous()).view(bs,2)
        if log_softmax:
            cls = F.log_softmax(cls,dim=1)
        return cls

class K_Fold:
    def __init__(self,data,split,k=5):
        assert k>1
        train_indices = split['train']
        self.k = k
        self.train_indices,self.val_indices = self.split_train(data,train_indices,self.k)

    def __call__(self,data,model_lst):
        scores = []

        for model in model_lst:
```

```
            initial_model = deepcopy(model)
            temp_scores = []
            for i in range(self.k):
                new_split = {'train':self.train_indices[i],'vali':self.val_indices[i]}
                train_loader,vali_loader = self.get_loader(data,new_split,batch_size=64)
                model = deepcopy(initial_model)
                temp_scores.append(self.train_val(train_loader,vali_loader,model))
            scores.append(temp_scores)
        return scores

    @staticmethod
    def split_train(data,train_indices,k):
        num_data = len(train_indices)
        random.shuffle(train_indices)
        shuffle_indices = deepcopy(train_indices).tolist()
        train_indices = []
        val_indices = []
        num_per_fold = num_data // k
        for start in range(0,num_data-num_per_fold+1,num_per_fold):
            whole_indices = deepcopy(shuffle_indices)
            val_temp = whole_indices[start:start+num_per_fold]
            for each in val_temp:
                whole_indices.remove(each)
            train_indices.append(whole_indices)
            val_indices.append(val_temp)
        return train_indices,val_indices


    @staticmethod
    def get_loader(data, split, batch_size=64):
        class MyDataset(Dataset):
        # Transformer的输入句子的表示形式会不定长，要矩阵化存储需要指定一个max_len
        # 不到的去做padding，超过的做truncationœ
            def __init__(self, data, split):
                super().__init__()
                self.vocab = data['vocab']
                self.pad_index = len(self.vocab.keys()) if '<pad>' not in self.vocab.keys()\
                else self.vocab['<pad>']
                self.max_len = data.get('max_len', 30)
                self.make_dataset(data, split)

            def make_dataset(self, data, split):
                # Data是包含了整个数据集的数据
                # 而我们只需要训练集/验证集/测试集的数据
                # 我们按照划分基准split里面的下标来确定加载哪部分的数据
                self.dataset = []
                for idx in split:
                    this_sentence_id = data['sentences_id'][idx]
                    item = [
                        torch.LongTensor(self.pad_data(this_sentence_id)),
                        torch.LongTensor([data['labels'][idx]])
                    ]
                    self.dataset.append(item)

            def pad_data(self, seq):
                # 让序列长度最长只有max_len，不足就补pad，超过就截断
                if len(seq) < self.max_len:
                    seq += [self.pad_index] * (self.max_len - len(seq))

```

```
                else:
                    seq = seq[:self.max_len]
                return seq

            def get_pad_index(self):
                return self.pad_index

            def __getitem__(self, ix):
                # ix大于等于0，小于len(self.dataset)
                return self.dataset[ix]

            def __len__(self):
                # 一共有多少数据
                return len(self.dataset)
        # split.keys() 包括 'train', 'vali'
        # 所以此函数是为了拿到训练集，验证集的数据加载器
        loader = []
        for mode in split.keys():
            # split[mode]指定了要取data的哪些数据
            dataset = MyDataset(data, split[mode])
            # Dataloader可帮助我们一次性取batch_size个样本出来
            loader.append(
                DataLoader(dataset,
                        batch_size = batch_size,
                        shuffle = True if mode=='train' else False)
            )
        return loader

    @staticmethod
    def train_val(train_loader,vali_loader,model):
        # 参数
        num_epochs = 50
        learning_rate = 0.005
        batch_size = 128
        vocab_size = len(vocab)+1 # +1是因为<pad> token

        data['max_len'] = 100


        # 运行的设备
        if torch.cuda.is_available():
            device = torch.device('cuda')
        else:
            device = torch.device('cpu')

        # 损失函数 —— 交叉熵
        crit = torch.nn.NLLLoss()
        # 优化方法
        optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
        best_val_acc = 0.
        for epoch in range(num_epochs):
            train_loss, train_acc = training(model, train_loader, crit, optimizer, device)
            # 验证
            vali_loss, vali_acc = evaluate(model, vali_loader, crit, device)
            best_val_acc = max(best_val_acc,vali_acc)
        return best_val_acc

model_lst = []
for word2vec_param in word2vec_param_lst:
```

```
    model = Word2Vec_Cls(vocab_size,pad_index = train_loader.dataset.get_pad_index(),\
    dropout=0.5,word2vec_param=word2vec_param)
    model_lst.append(model)

k_fold = K_Fold(data,split,k=5)
scores = k_fold(data,model_lst)
for score in scores:
    print(score,end='  ')
    print(f'mean score: {np.array(score).mean()}')
```