题目 2:

mbti 性格预测

目录

一、数据预处理

1.1 数据探索

由于数据中的特征比较少，只有用户在论坛的评论文本，在这里本文构建了一个关于每条评论平均长度的特征 comment_length，计算方法是取用户所发的五十条评论的平均长度（由于每个用户的“|||”数量是一样的，故在计算时没有把它去掉，代码如下：

```
train['comment_lengtht'] = train['posts'].apply(lambda x: len(x.split()))/50
```

用小提琴图展示各个性格的用户的‘comment_length’特征的信息，如图 1。可以看出，性格为‘ISFJ’（内倾感觉情感判断）的人的评论长度跨度最大，他们中既有热衷于发表长篇大论的“活跃家”，也有习惯于只言片语的“沉默者”；并且，总体而言，可以发现在判断方式上属于“情感”（F)的人相比于属于“思维”（T)的人发表的评论长度更长。

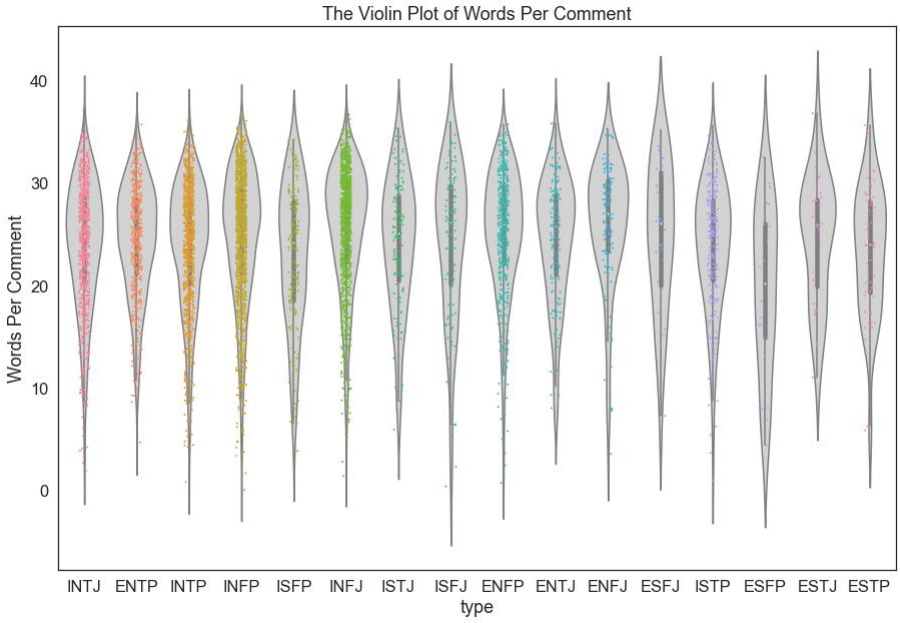

图 1 各性格关于评论长度的小提琴图

1.2 数据预处理

由于计算机是不能直接从文本字符串中发现规律的。所以我们需要将字符串编码为计算机可以理解的数字，在本报告中，我们分别采用了两种方法：TF-IDF 和 Word2cv。

1.2.1 TF-IDF

首先，我们对 TF-IDF 概念进行一个简单介绍。TF-IDF 是一种加权技术。采用一种统计方法，根据字词在文本中出现的次数和在整个语料中出现的文档频率来计算一个字词在整个

语料中的重要程度。

- 词频（TF） = 某个词在文章中的出现次数 / 文章总词数 或者 词频（TF） = 某个词在文章中的出现次数 / 拥有最高词频的词的次数
- 逆文档频率（IDF） = log（语料库的文档总数/包含该词的文档总数+1）

IDF 表示计算倒文本频率。文本频率是指某个关键词在整个语料所有文章中出现的次数。倒文本频率是文本频率的倒数，主要用于降低所有文档中一些常见却对文档影响不大的词语的作用。IDF 反应了一个词在所有文本中出现的频率，如果一个词在很多的文本中出现，那么它的 IDF 值应该低，而反过来如果一个词在比较少的文本中出现，那么它的 IDF 值应该高。

- TF-IDF = 词频（TF) * 逆文档频率（IDF）

TF-IDF 能过滤掉一些常见的却无关紧要的词语，同时保留影响整个文本的重要词语，常用于提取关键词作为分类或聚类问题的输入。但同时也有会丢失文本上下文之间联系的缺点。

1.2.2 word2cv

在 word2vec 出现之前，自然语言处理经常把字词转为 one-hot 编码类型的词向量，这种方式虽然非常简单易懂，但是数据稀疏性非常高，维度很多，很容易造成维度灾难，尤其是在深度学习中；其次这种词向量中任意两个词之间都是孤立的，存在语义鸿沟（这样就不能体现词与词之间的关系）。Hinton 大神提出的 Distributional Representation 很好的解决了 one-hot 编码的主要缺点。解决了语义之间的鸿沟，可以通过计算向量之间的距离来体现词与词之间的关系。Distributional Representation 词向量是密集的。word2vec 是一个用来训练 Distributional Representation 类型的词向量的一种工具。

word2vec 主要分为 CBOW（Continuous Bag of Words）和 Skip-Gram 两种模式。CBOW 是从原始语句推测目标字词；而 Skip-Gram 正好相反，是从目标字词推测出原始语句。CBOW 对小型数据库比较合适，而 Skip-Gram 在大型语料中表现更好。

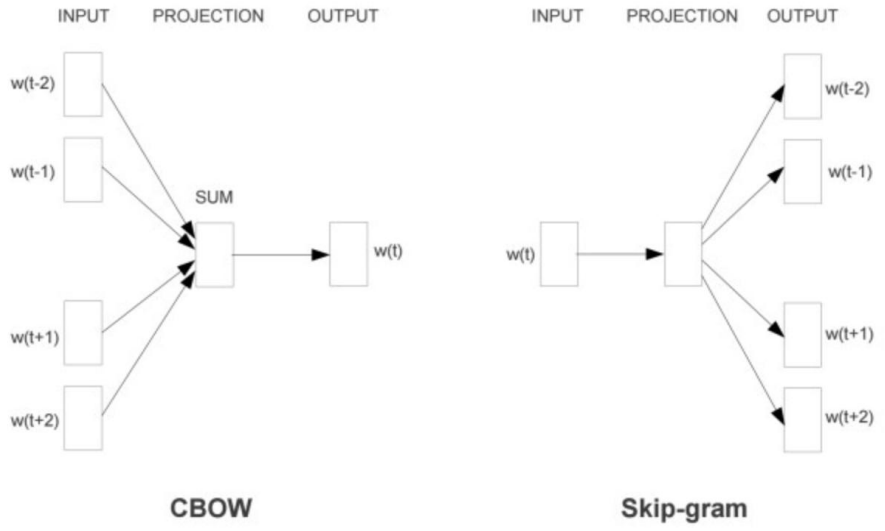

图 2 CBOW 与 Skip-gram

例如，对同样一个句子：Hangzhou is a nice city。我们要构造一个语境与目标词汇的映射关系，其实就是 input 与 label 的关系。

假设滑窗尺寸为 1，则

- CBOW 可以制造的映射关系为：[Hangzhou,a]—>is，[is,nice]—>a，[a,city]—>nice
- Skip-Gram 可以制造的映射关系为(is,Hangzhou)，(is,a)，(a,is)， (a,nice)，(nice,a)，(nice,city)

1.2.3 具体处理过程

首先，定义一个数据处理函数，去除了各种标点符号，HTML 标签，进行文本的小写化。

```
[55]: ##数据处理函数
      def cleanReview(subject):
          beau = BeautifulSoup(subject)
          newSubject = beau.get_text()
          newSubject = newSubject.replace("\\", "").replace("\'", "").replace('/', '').replace('"', '').replace(',', '').replace('.', '').repla
          newSubject = newSubject.strip().split(" ")
          newSubject = [word.lower() for word in newSubject]
          newSubject = " ".join(newSubject)

          return newSubject
```

对于 TF-IDF 的计算，我直接采用了 sklearn.feature_extraction.text 中的 TfidfVectorizer 函数，然后得到了文档词语矩阵，这将作为我们后续机器学习模型的输入。

对于 word2cv，使用 gensim 中的 word2vec API 来训练模型，设置词向量维度为 1000，迭代次数 iter 为 8，采用 skip-gram 模型，模型保存为 bin 格式，在后续深度学习模型中使用。

此外，通过查阅资料，我设置了一些停用词，如“is”、“am”等。

二、模型建立

2.1 机器学习模型

首先，我们考虑采用机器学习模型来预测人的性格，使用的是 TF-IDF 处理后的文本特征作为输入，分别使用了逻辑回归 logisticregression、XGBoost 分类模型来进行预测。采用网格搜索进行调参后，得到

- 逻辑回归最优参数为 C=1,其中 C 是正则化相关的参数。
- XGBoost 分类模型的参数为 max_depth=10,n_estimators=200，learning_rate=0.3

```
Out[186]:
XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints='',
              learning_rate=0.300000012, max_delta_step=0, max_depth=10,
              min_child_weight=1, missing=nan, monotone_constraints='()',
              n_estimators=200, n_jobs=12, num_parallel_tree=1,
              objective='multi:softprob', random_state=0, reg_alpha=0,
              reg_lambda=1, scale_pos_weight=None, subsample=1,
              tree_method='exact', use_label_encoder=False,
              validate_parameters=1, verbosity=None)
```

- 得到他们在训练集和测试集上的表现（使用的指标为 score，表示预测准确率）为

	训练集	测试集
logisticregression	0.7796	0.6357
XGBoostClassification	0.9684	0.7145

2.2 深度学习模型

采用 word2cv 处理后的文本词向量作为输入，训练集按照 0.9 的比例划分为训练集和验证集。训练集的构建过程为

- 1）将数据加载进来，将句子分割成词表示，并去除低频词和停用词。

- 2）将词映射成索引表示，构建词汇-索引映射表，并保存成 json 的数据格式，之后做 inference 时可以用到。（注意，有的词可能不在 word2vec 的预训练词向量中，这种词直接用 UNK 表示）

- 3）从预训练的词向量模型中读取出词向量，作为初始化值输入到模型中。

- 4）将数据集分割成训练集和测试集

词向量的长度 embedding size=1000，采用 Bi-LSTM 进行训练，网络结构为一个两层的 LSTM 层，中间的隐藏层参数为（256，256），模型其余参数为

参数	Learning_rate	epoch	batch_size	dropoutKeepProb
	0.001	80	64	0.5

得到的训练过程中偏差 bias 的梯度曲线为

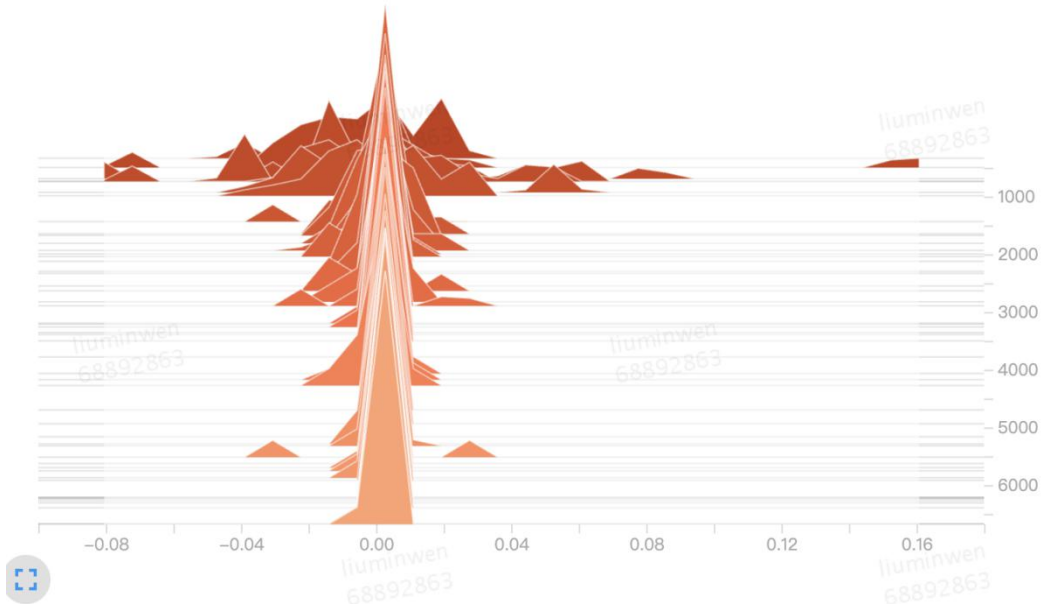

图 3 Bias‘s grad hist

训练的 loss 曲线为

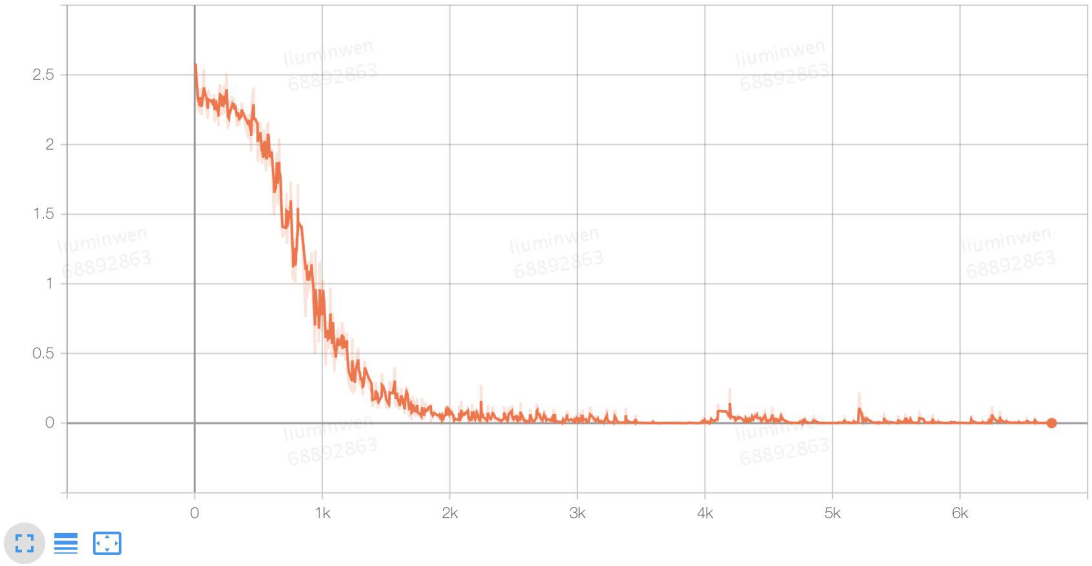

图 4 loss 曲线

可以看到，大概在两千步迭代后，loss 就已经下降到一个比较低的水平。得到在训练集上、验证集、测试集上的 acc、recall、precision、f_beta 值（注：对于多分类问题的这些指标，我们把各个 label 当做一个二分类问题计算出这些指标值，然后对于这 16 个 label 的这些指标值求平均，就作为多分类问题的指标值。）为

	acc	recall	precision	F_beta
训练集	0.999	0.813	0.781	0.779
验证集	0.625	0.348	0.305	0.315
测试集	0.589	0.321	0.310	0.308

三、模型分析

3.1 结果分析

由上面的结果可知，xgboost 模型的表现最好。logist 模型的训练速度很快，需要设置的超参数少，经过调参得到的提升有限，最好的结果仅为训练集上 0.8 不到的准确率，测试集上 0.63 左右；而 xgboost 模型的训练速度相较更久，调参前容易过拟合，调参相对复杂，最后得到的结果虽然仍有过拟合的嫌疑，但测试集上达到了 0.7145 的准确率。而采用的深度学习模型，其过拟合现象异常严重，在实验中，我尝试改进结果，但是即使设置早停 early-stopping 也不能明显改善，通常而言，深度学习模型不会效果如此之差，考虑到在 BI-LSTM 模型的输入我们采用的是 word2cv 而非 TF-IDF 且训练集数据量不大，应该是在本例中采用 word2cv 进行文本处理不是一个最佳的方法。

得到的最佳模型 xgb 的混淆矩阵,如图 5。

confusion_matrix(true,prediction,labels=['ISTJ','ISFJ','INFJ','INTJ','ISTP','ISFP','INFP','INTP','ESTJ','ESFJ','ENFJ','ENTJ','ESTP','ESFP','ENFP','ENTP'])

由图可知，如第八列恒为 0 可知，测试集中没有人被预测为'INTP'，（第八对应的 label 为 INTP'），且有第 6、7 列的尤其多，可知模型预测时倾向于预测结果为'ISFP'、'INFP'，这和样本的不平衡问题有关。

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	3	0	9	18	0	0	24	13	0	0	0	0	0	0	5	2
1	0	9	9	2	0	1	20	6	0	0	0	1	0	0	1	0
2	0	0	318	16	1	0	77	20	0	0	0	0	0	0	4	6
3	0	0	32	229	0	0	47	43	0	0	0	0	0	0	5	8
4	0	1	9	8	29	0	21	31	0	0	0	0	0	0	2	1
5	0	0	8	3	2	12	51	9	0	0	0	0	0	0	5	2
6	0	1	21	8	0	1	497	27	0	0	1	1	0	0	4	7
7	0	0	22	15	1	0	38	319	0	0	0	0	0	0	3	5
8	0	0	0	0	0	0	6	0	0	0	0	0	0	0	1	0
9	0	0	1	0	0	0	5	4	0	0	0	0	0	0	1	0
10	0	0	13	4	0	0	20	6	0	0	6	0	0	0	2	0
11	0	0	5	20	0	0	14	21	0	0	2	9	0	0	1	5
12	0	0	8	4	0	0	8	8	0	0	0	0	0	0	0	1
13	0	0	3	3	0	1	5	1	0	0	0	1	0	0	4	1
14	0	1	15	14	2	0	40	7	0	0	0	0	0	0	100	5
15	0	0	14	14	0	0	22	43	0	0	0	0	0	0	5	105

图 5 验证集上结果的混淆矩阵

3.2 模型对比、总结与展望

首先在文本处理上，TF-IDF 作用主要在于提取出关键词，相比 word2cv 它更具有可解释性，但是其不能保留上下文的联系性；word2cv 则保留了上下文的联系，常用于计算相似度等，但是对于 embedding 的维度选择也存在一些技巧，如本报告中，由于维度选择过大，是的后续模型的过拟合风险大大增高，并且在本题中，由于我们的目的其实并不是要判断上下文或计算相似度，而是判断文本背后的人物性格，且数据量不算大，那么对于本题而言，使用 TF-IDF 作文本处理是更好的。

由以上结果和分析可知，线性模型的训练速度快，但“天花板”较低；XGBoost 模型的优点为“天花板”高，经过调参可达到较高的准确度，效果更好。

而深度学习模型，它的“天花板”理论上更高，但是复杂的参数选择和网络结构设计也让模型的改进存在难点，且本题中的数据量不算大，生成词向量时的词库未必准确，使得模型难以达到一个理想的结果。

总体而言，模型的结果仍存在改进空间，在测试集上的表现仍然有较大的提升空间，在

后续中存在一些改进的方向：

- 调小 embedding size，在起初，由于考虑到 50 条文本的长度较长，将维度设为了 1000，但其实每条文本的平均长度不到 30，虽然每个用户的数据一口气输入 50 条评论，但实际上每条文本的地位是等效的；且过大的 size 使得词向量过于稀疏，在数据量不够的情况下，使得过拟合的风险大大增加。
- 虽然采用 Bi-LSTM 模型这样一个双向结构能够考虑上下文前后双向的信息，但仔细想，在本题中实际上对前后文理解的要求并没有那么高，我们更应该关注的是句子中能够体现性格的重要信息，可以考虑在后续引入 attention 机制（注意力机制）。

四、附录

4.1 代码

4.1.1 机器学习

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Sat Jun 26 21:00:37 2021

@author: liuminwen
"""

import pandas as pd
import numpy as np
train=pd.read_csv('/Users/liuminwen/Desktop/Big Project/personality/mbti_train.csv')
test=pd.read_csv('/Users/liuminwen/Desktop/Big Project/personality/mbti_test.csv')
train['label']=np.arange(6000)
type_dict={'ISTJ':1,'ISFJ':2,'INFJ':3,'INTJ':4,'ISTP':5,'ISFP':6,'INFP':7,'INTP':8,'ESTJ':9,'ESFJ':10,'
ENFJ':11,'ENTJ':12,'ESTP':13,'ESFP':14,'ENFP':15,'ENTP':16}
for i in range(len(train.iloc[:,1])):
    train['label'][i]=type_dict[train.iloc[i,1]]
test['label']=np.arange(2675)
for i in range(len(test.iloc[:,1])):
    test['label'][i]=type_dict[test.iloc[i,1]]

#%%
#数据探索
#评论的长度
train['words_per_comment'] = train['posts'].apply(lambda x: len(x.split()))/50
train['posts'] = train['posts'].apply(lambda x:x.lower())
```

```
train.head()

#展示一下
import seaborn as sns
import matplotlib.pyplot as plt
#画布设置及尺寸
sns.set(style='white', font_scale=1.5)
plt.figure(figsize=(15, 10))
#绘制小提琴图
sns.violinplot(x='type',
                    y='words_per_comment',
                    data=train,
                    color='lightgray')
#绘制分类三点图，叠加到小提琴图图层上方
sns.stripplot(x='type',
                    y='words_per_comment',
                    data=train,
                    size=2,
                    jitter=True)
#标题及 y 轴名
plt.title('The Violin Plot of Words Per Comment', size=18)
plt.ylabel('Words Per Comment')
#显示
plt.show()

#%%
#文本向量化
X_train=train['posts']
test=test.rename(columns={'post':'posts'})
X_test=test['posts']
from sklearn.feature_extraction.text import TfidfVectorizer
tfidf = TfidfVectorizer(stop_words='english')
X_train = tfidf.fit_transform(X_train)
X_test = tfidf.transform(X_test)

#训练模型
y_train=train['type']
y_test=test['type']
from sklearn.linear_model import LogisticRegression
model1 = LogisticRegression()
model1.fit(X_train, y_train)
model1.score(X_train, y_train)
model1.score(X_test, y_test)
```

```
#
from sklearn.linear_model import SGDClassifier
model2 = SGDClassifier()
model2.fit(X_train, y_train)
model2.score(X_train, y_train)
model2.score(X_test, y_test)

#
from sklearn.linear_model import Perceptron
model3 = Perceptron()
model3.fit(X_train, y_train)
model3.score(X_train, y_train)
model3.score(X_test, y_test)



#%%
df1=pd.DataFrame(train['posts'].apply(lambda x:x.split('|||')))
df2=pd.DataFrame()
df2 = pd.DataFrame(df1.posts.values.tolist())

#%%
from sklearn.model_selection import StratifiedKFold #交叉验证
from sklearn.model_selection import GridSearchCV #网格搜索



param_grid_1={'C':[0.1,0.5,0.8,1]}
grid_search_1 = GridSearchCV(model1,param_grid_1,cv = 5)
grid_search_1.fit(X_train,y_train)
model1 = LogisticRegression(C=1)
model1.fit(X_train, y_train)
model1.score(X_train, y_train)
model1.score(X_test, y_test)
pred_1=model1.predict(X_test)

param_grid_2={'alpha':[0.0001,0.001,0.01,0.05],'early_stopping':[True,False],'max_iter':[1000,2000,
5000]}
grid_search_2 = GridSearchCV(model2,param_grid_2,scoring = 'roc_auc',n_jobs = -1,cv = 5)
grid_search_2.fit(X_train,y_train)

#%%
```

```
from xgboost import XGBClassifier
model4=XGBClassifier(objective='multi:softmax')
param_grid_4={'max_depth':[5,6,10,20,50],'n_estimators':[20,50,100,200]}
grid_search_4 = GridSearchCV(model4,param_grid_4,cv = 5)
grid_search_4.fit(X_train,y_train)

model4=grid_search_4.best_estimator
model4.fit(X_train,y_train)
model4.score(X_train, y_train)
pred=model4.predict(X_test)
model4.score(X_test, y_test)

#%%
#预测集上的对比
from sklearn.metrics import confusion_matrix
result1=confusion_matrix(y_test,pred_1,labels=['ISTJ','ISFJ','INFJ','INTJ','ISTP','ISFP','INFP','INT
P','ESTJ','ESFJ','ENFJ','ENTJ','ESTP','ESFP','ENFP','ENTP'])
```

4.1.2 word2cv & Bi-LSTM

```
import pandas as pd
import numpy as np
from bs4 import BeautifulSoup
##数据处理函数
def cleanReview(subject):
        beau = BeautifulSoup(subject)
        newSubject = beau.get_text()
        newSubject = newSubject.replace("\\", "").replace("\'", "").replace('/', '').replace('"', '').replace(',',
'').replace('.', '').replace('?', '').replace('(', '').replace(')', '').replace('|||', '')
        newSubject = newSubject.strip().split(" ")
        newSubject = [word.lower() for word in newSubject]
        newSubject = " ".join(newSubject)

        return newSubject
train_x= train['posts'].apply(cleanReview)
df=pd.concat([train_x, train['type']], axis=0)
df.to_csv("./wordEmbdiing.txt", index=False)
import logging
import gensim
from gensim.models import word2vec

# 设置输出日志
logging.basicConfig(format='%(asctime)s : %(levelname)s : %(message)s', level=logging.INFO)
```

```
# 直接用gemsim提供的API去读取txt文件，读取文件的API有LineSentence 和 Text8Corpus, PathLineSentences 等。
sentences = word2vec.LineSentence("./wordEmbdiing.txt")

# 训练模型，词向量的长度设置为1000， 迭代次数为8，采用skip-gram模型，模型保存为bin格式
model = gensim.models.Word2Vec(sentences, size=1000, sg=1, iter=8)
model.wv.save_word2vec_format("./word2Vec" + ".bin", binary=True)

# 加载bin格式的模型
wordVec = gensim.models.KeyedVectors.load_word2vec_format("word2Vec.bin", binary=True)
import os
import csv
import time
import datetime
import random
import json

import warnings
from collections import Counter
from math import sqrt

import gensim
import pandas as pd
import numpy as np
import tensorflow as tf
from sklearn.metrics import roc_auc_score, accuracy_score, precision_score, recall_score
warnings.filterwarnings("ignore")
# 配置参数

class TrainingConfig(object):
    epoches = 80
    evaluateEvery = 100
    checkpointEvery = 100
    learningRate = 0.001

class ModelConfig(object):
    embeddingSize = 1000

    hiddenSizes = [256, 256]  # 单层LSTM结构的神经元个数

    dropoutKeepProb = 0.5
    l2RegLambda = 0.0
```

```
class Config(object):
    sequenceLength = 200    # 取了所有序列长度的均值
    batchSize = 64

    dataSource = "./mbti_train.csv"

    stopWordSource = "./english.txt"

    numClasses = 16    # 二分类设置为 1，多分类设置为类别的数目

    rate = 0.9    # 训练集的比例

    training = TrainingConfig()

    model = ModelConfig()


# 实例化配置参数对象
config = Config()
# 数据预处理的类，生成训练集和测试集

class Dataset(object):
    def __init__(self, config):
        self.config = config
        self._dataSource = config.dataSource
        self._stopWordSource = config.stopWordSource

        self._sequenceLength = config.sequenceLength    # 每条输入的序列处理为定长
        self._embeddingSize = config.model.embeddingSize
        self._batchSize = config.batchSize
        self._rate = config.rate

        self._stopWordDict = {}

        self.trainReviews = []
        self.trainLabels = []

        self.evalReviews = []
        self.evalLabels = []

        self.wordEmbedding =None

        self.labelList = []
```

```
    def _readData(self, filePath):
        """
        从 csv 文件中读取数据集
        """

        df = pd.read_csv(filePath)

        if self.config.numClasses == 1:
            labels = df["sentiment"].tolist()
        elif self.config.numClasses > 1:
            labels = df["type"].tolist()

        review = df["posts"].tolist()
        reviews = [line.strip().split() for line in review]

        return reviews, labels

    def _labelToIndex(self, labels, label2idx):
        """
        将标签转换成索引表示
        """
        labelIds = [label2idx[label] for label in labels]
        return labelIds

    def _wordToIndex(self, reviews, word2idx):
        """
        将词转换成索引
        """
        reviewIds = [[word2idx.get(item, word2idx["UNK"]) for item in review] for review in reviews]
        return reviewIds

    def _genTrainEvalData(self, x, y, word2idx, rate):
        """
        生成训练集和验证集
        """
        reviews = []
        for review in x:
            if len(review) >= self._sequenceLength:
                reviews.append(review[:self._sequenceLength])
            else:
                reviews.append(review + [word2idx["PAD"]] * (self._sequenceLength - len(review)))
```

```
        trainIndex = int(len(x) * rate)

        trainReviews = np.asarray(reviews[:trainIndex], dtype="int64")
        trainLabels = np.array(y[:trainIndex], dtype="float32")

        evalReviews = np.asarray(reviews[trainIndex:], dtype="int64")
        evalLabels = np.array(y[trainIndex:], dtype="float32")

        return trainReviews, trainLabels, evalReviews, evalLabels

    def _genVocabulary(self, reviews, labels):
        """
        生成词向量和词汇-索引映射字典，可以用全数据集
        """

        allWords = [word for review in reviews for word in review]

        # 去掉停用词
        subWords = [word for word in allWords if word not in self.stopWordDict]

        wordCount = Counter(subWords)  # 统计词频
        sortWordCount = sorted(wordCount.items(), key=lambda x: x[1], reverse=True)

        # 去除低频词
        words = [item[0] for item in sortWordCount if item[1] >= 5]

        vocab, wordEmbedding = self._getWordEmbedding(words)
        self.wordEmbedding = wordEmbedding

        word2idx = dict(zip(vocab, list(range(len(vocab)))))

        uniqueLabel = list(set(labels))
        label2idx = dict(zip(uniqueLabel, list(range(len(uniqueLabel)))))
        self.labelList = list(range(len(uniqueLabel)))

        # 将词汇-索引映射表保存为 json 数据，之后做 inference 时直接加载来处理数据
        with open("./wordJson/word2idx.json", "w", encoding="utf-8") as f:
            json.dump(word2idx, f)

        with open("./wordJson/label2idx.json", "w", encoding="utf-8") as f:
            json.dump(label2idx, f)

        return word2idx, label2idx
```

```
    def _getWordEmbedding(self, words):
        """
        按照我们的数据集中的单词取出预训练好的 word2vec 中的词向量
        """

        wordVec = gensim.models.KeyedVectors.load_word2vec_format("./word2Vec.bin",
binary=True)
        vocab = []
        wordEmbedding = []

        # 添加 "pad" 和 "UNK",
        vocab.append("PAD")
        vocab.append("UNK")
        wordEmbedding.append(np.zeros(self._embeddingSize))
        wordEmbedding.append(np.random.randn(self._embeddingSize))

        for word in words:
            try:
                vector = wordVec.wv[word]
                vocab.append(word)
                wordEmbedding.append(vector)
            except:
                print(word + "不存在于词向量中")

        return vocab, np.array(wordEmbedding)

    def _readStopWord(self, stopWordPath):
        """
        读取停用词
        """

        with open(stopWordPath, "r") as f:
            stopWords = f.read()
            stopWordList = stopWords.splitlines()
            # 将停用词用列表的形式生成，之后查找停用词时会比较快
            self.stopWordDict = dict(zip(stopWordList, list(range(len(stopWordList)))))

    def dataGen(self):
        """
        初始化训练集和验证集
        """

        # 初始化停用词
        self._readStopWord(self._stopWordSource)
```

```
        # 初始化数据集
        reviews, labels = self._readData(self._dataSource)

        # 初始化词汇-索引映射表和词向量矩阵
        word2idx, label2idx = self._genVocabulary(reviews, labels)

        # 将标签和句子数值化
        labelIds = self._labelToIndex(labels, label2idx)
        reviewIds = self._wordToIndex(reviews, word2idx)

        # 初始化训练集和测试集
        trainReviews, trainLabels, evalReviews, evalLabels = self._genTrainEvalData(reviewIds,
labelIds, word2idx, self._rate)
        self.trainReviews = trainReviews
        self.trainLabels = trainLabels

        self.evalReviews = evalReviews
        self.evalLabels = evalLabels


data = Dataset(config)
data.dataGen()
# 输出 batch 数据集

def nextBatch(x, y, batchSize):
        """
        生成 batch 数据集，用生成器的方式输出
        """

        perm = np.arange(len(x))
        np.random.shuffle(perm)
        x = x[perm]
        y = y[perm]

        numBatches = len(x) // batchSize

        for i in range(numBatches):
            start = i * batchSize
            end = start + batchSize
            batchX = np.array(x[start: end], dtype="int64")
            batchY = np.array(y[start: end], dtype="float32")

            yield batchX, batchY
```

```
# 构建模型
class BiLSTM(object):
    """
    Bi-LSTM 用于文本分类
    """
    def __init__(self, config, wordEmbedding):

        # 定义模型的输入
        self.inputX = tf.placeholder(tf.int32, [None, config.sequenceLength], name="inputX")
        self.inputY = tf.placeholder(tf.int32, [None], name="inputY")

        self.dropoutKeepProb = tf.placeholder(tf.float32, name="dropoutKeepProb")

        # 定义 l2 损失
        l2Loss = tf.constant(0.0)

        # 词嵌入层
        with tf.name_scope("embedding"):

            # 利用预训练的词向量初始化词嵌入矩阵
            self.W = tf.Variable(tf.cast(wordEmbedding, dtype=tf.float32,
name="word2vec") ,name="W")
            # 利用词嵌入矩阵将输入的数据中的词转换成词向量，维度[batch_size,
sequence_length, embedding_size]
            self.embeddedWords = tf.nn.embedding_lookup(self.W, self.inputX)

        # 定义两层双向 LSTM 的模型结构
        with tf.name_scope("Bi-LSTM"):

            for idx, hiddenSize in enumerate(config.model.hiddenSizes):
                with tf.name_scope("Bi-LSTM" + str(idx)):
                    # 定义前向 LSTM 结构
                    lstmFwCell =
tf.nn.rnn_cell.DropoutWrapper(tf.nn.rnn_cell.LSTMCell(num_units=hiddenSize,
state_is_tuple=True),

output_keep_prob=self.dropoutKeepProb)
                    # 定义反向 LSTM 结构
                    lstmBwCell =
tf.nn.rnn_cell.DropoutWrapper(tf.nn.rnn_cell.LSTMCell(num_units=hiddenSize,
state_is_tuple=True),

output_keep_prob=self.dropoutKeepProb)
```

```
                    # 采用动态 rnn，可以动态的输入序列的长度，若没有输入，则取序列的全长
                    # outputs 是一个元祖(output_fw, output_bw), 其中两个元素的维度都是[batch_size, max_time, hidden_size],fw 和 bw 的 hidden_size 一样
                    # self.current_state 是最终的状态，二元组(state_fw, state_bw)，state_fw=[batch_size, s]，s 是一个元祖(h, c)
                    outputs, self.current_state = tf.nn.bidirectional_dynamic_rnn(lstmFwCell, lstmBwCell,

self.embeddedWords, dtype=tf.float32,

scope="bi-lstm" + str(idx))

                    # 对 outputs 中的 fw 和 bw 的结果拼接 [batch_size, time_step, hidden_size * 2]
                    self.embeddedWords = tf.concat(outputs, 2)

        # 去除最后时间步的输出作为全连接的输入
        finalOutput = self.embeddedWords[:, 0, :]

        outputSize = config.model.hiddenSizes[-1] * 2  # 因为是双向 LSTM，最终的输出值是 fw 和 bw 的拼接，因此要乘以 2
        output = tf.reshape(finalOutput, [-1, outputSize])  # reshape 成全连接层的输入维度

        # 全连接层的输出
        with tf.name_scope("output"):
            outputW = tf.get_variable(
                "outputW",
                shape=[outputSize, config.numClasses],
                initializer=tf.contrib.layers.xavier_initializer())

            outputB= tf.Variable(tf.constant(0.1, shape=[config.numClasses]), name="outputB")
            l2Loss += tf.nn.l2_loss(outputW)
            l2Loss += tf.nn.l2_loss(outputB)
            self.logits = tf.nn.xw_plus_b(output, outputW, outputB, name="logits")
            if config.numClasses == 1:
                self.predictions = tf.cast(tf.greater_equal(self.logits, 0.0), tf.float32, name="predictions")
            elif config.numClasses > 1:
                self.predictions = tf.argmax(self.logits, axis=-1, name="predictions")

        # 计算二元交叉熵损失
        with tf.name_scope("loss"):
```

```
                if config.numClasses == 1:
                    losses = tf.nn.sigmoid_cross_entropy_with_logits(logits=self.logits,
labels=tf.cast(tf.reshape(self.inputY, [-1, 1]),

dtype=tf.float32))
                elif config.numClasses > 1:
                    losses = tf.nn.sparse_softmax_cross_entropy_with_logits(logits=self.logits,
labels=self.inputY)

                self.loss = tf.reduce_mean(losses) + config.model.l2RegLambda * l2Loss
"""
定义各类性能指标
"""

def mean(item: list) -> float:
    """
    计算列表中元素的平均值
    :param item: 列表对象
    :return:
    """
    res = sum(item) / len(item) if len(item) > 0 else 0
    return res


def accuracy(pred_y, true_y):
    """
    计算二类和多类的准确率
    :param pred_y: 预测结果
    :param true_y: 真实结果
    :return:
    """
    if isinstance(pred_y[0], list):
        pred_y = [item[0] for item in pred_y]
    corr = 0
    for i in range(len(pred_y)):
        if pred_y[i] == true_y[i]:
            corr += 1
    acc = corr / len(pred_y) if len(pred_y) > 0 else 0
    return acc


def binary_precision(pred_y, true_y, positive=1):
    """
```

```
    二类的精确率计算
    :param pred_y: 预测结果
    :param true_y: 真实结果
    :param positive: 正例的索引表示
    :return:
    """
    corr = 0
    pred_corr = 0
    for i in range(len(pred_y)):
        if pred_y[i] == positive:
            pred_corr += 1
            if pred_y[i] == true_y[i]:
                corr += 1

    prec = corr / pred_corr if pred_corr > 0 else 0
    return prec


def binary_recall(pred_y, true_y, positive=1):
    """
    二类的召回率
    :param pred_y: 预测结果
    :param true_y: 真实结果
    :param positive: 正例的索引表示
    :return:
    """
    corr = 0
    true_corr = 0
    for i in range(len(pred_y)):
        if true_y[i] == positive:
            true_corr += 1
            if pred_y[i] == true_y[i]:
                corr += 1

    rec = corr / true_corr if true_corr > 0 else 0
    return rec


def binary_f_beta(pred_y, true_y, beta=1.0, positive=1):
    """
    二类的 f beta 值
    :param pred_y: 预测结果
    :param true_y: 真实结果
    :param beta: beta 值
```

```
    :param positive: 正例的索引表示
    :return:
    """
    precision = binary_precision(pred_y, true_y, positive)
    recall = binary_recall(pred_y, true_y, positive)
    try:
        f_b = (1 + beta * beta) * precision * recall / (beta * beta * precision + recall)
    except:
        f_b = 0
    return f_b


def multi_precision(pred_y, true_y, labels):
    """
    多类的精确率
    :param pred_y: 预测结果
    :param true_y: 真实结果
    :param labels: 标签列表
    :return:
    """
    if isinstance(pred_y[0], list):
        pred_y = [item[0] for item in pred_y]

    precisions = [binary_precision(pred_y, true_y, label) for label in labels]
    prec = mean(precisions)
    return prec


def multi_recall(pred_y, true_y, labels):
    """
    多类的召回率
    :param pred_y: 预测结果
    :param true_y: 真实结果
    :param labels: 标签列表
    :return:
    """
    if isinstance(pred_y[0], list):
        pred_y = [item[0] for item in pred_y]

    recalls = [binary_recall(pred_y, true_y, label) for label in labels]
    rec = mean(recalls)
    return rec
```

```
def multi_f_beta(pred_y, true_y, labels, beta=1.0):
    """
    多类的 f beta 值
    :param pred_y: 预测结果
    :param true_y: 真实结果
    :param labels: 标签列表
    :param beta: beta 值
    :return:
    """
    if isinstance(pred_y[0], list):
        pred_y = [item[0] for item in pred_y]

    f_betas = [binary_f_beta(pred_y, true_y, beta, label) for label in labels]
    f_beta = mean(f_betas)
    return f_beta


def get_binary_metrics(pred_y, true_y, f_beta=1.0):
    """
    得到二分类的性能指标
    :param pred_y:
    :param true_y:
    :param f_beta:
    :return:
    """
    acc = accuracy(pred_y, true_y)
    recall = binary_recall(pred_y, true_y)
    precision = binary_precision(pred_y, true_y)
    f_beta = binary_f_beta(pred_y, true_y, f_beta)
    return acc, recall, precision, f_beta


def get_multi_metrics(pred_y, true_y, labels, f_beta=1.0):
    """
    得到多分类的性能指标
    :param pred_y:
    :param true_y:
    :param labels:
    :param f_beta:
    :return:
    """
    acc = accuracy(pred_y, true_y)
    recall = multi_recall(pred_y, true_y, labels)
    precision = multi_precision(pred_y, true_y, labels)
```

```
f_beta = multi_f_beta(pred_y, true_y, labels, f_beta)
return acc, recall, precision, f_beta
# 训练模型

# 生成训练集和验证集
trainReviews = data.trainReviews
trainLabels = data.trainLabels
evalReviews = data.evalReviews
evalLabels = data.evalLabels

wordEmbedding = data.wordEmbedding
labelList = data.labelList

# 定义计算图
with tf.Graph().as_default():

    session_conf = tf.ConfigProto(allow_soft_placement=True, log_device_placement=False)
    session_conf.gpu_options.allow_growth=True
    session_conf.gpu_options.per_process_gpu_memory_fraction = 0.9  # 配置 gpu 占用率

    sess = tf.Session(config=session_conf)

    # 定义会话
    with sess.as_default():
        lstm = BiLSTM(config, wordEmbedding)

        globalStep = tf.Variable(0, name="globalStep", trainable=False)
        # 定义优化函数，传入学习速率参数
        optimizer = tf.train.AdamOptimizer(config.training.learningRate)
        # 计算梯度,得到梯度和变量
        gradsAndVars = optimizer.compute_gradients(lstm.loss)
        # 将梯度应用到变量下，生成训练器
        trainOp = optimizer.apply_gradients(gradsAndVars, global_step=globalStep)

        # 用 summary 绘制 tensorBoard
        gradSummaries = []
        for g, v in gradsAndVars:
            if g is not None:
                tf.summary.histogram("{}/grad/hist".format(v.name), g)
                tf.summary.scalar("{}/grad/sparsity".format(v.name), tf.nn.zero_fraction(g))

        outDir = os.path.abspath(os.path.join(os.path.curdir, "summarys"))
        print("Writing to {}\n".format(outDir))
```

```
        lossSummary = tf.summary.scalar("loss", lstm.loss)
        summaryOp = tf.summary.merge_all()

        trainSummaryDir = os.path.join(outDir, "train")
        trainSummaryWriter = tf.summary.FileWriter(trainSummaryDir, sess.graph)

        evalSummaryDir = os.path.join(outDir, "eval")
        evalSummaryWriter = tf.summary.FileWriter(evalSummaryDir, sess.graph)


        # 初始化所有变量
        saver = tf.train.Saver(tf.global_variables(), max_to_keep=5)

        # 保存模型的一种方式，保存为 pb 文件
        savedModelPath = "./Bi-LSTM/savedModel"
        if os.path.exists(savedModelPath):
            os.rmdir(savedModelPath)
        builder = tf.saved_model.builder.SavedModelBuilder(savedModelPath)

        sess.run(tf.global_variables_initializer())

        def trainStep(batchX, batchY):
            """
            训练函数
            """
            feed_dict = {
              lstm.inputX: batchX,
              lstm.inputY: batchY,
              lstm.dropoutKeepProb: config.model.dropoutKeepProb
            }
            _, summary, step, loss, predictions = sess.run(
                [trainOp, summaryOp, globalStep, lstm.loss, lstm.predictions],
                feed_dict)

            timeStr = datetime.datetime.now().isoformat()

            if config.numClasses == 1:
                acc, recall, prec, f_beta = get_binary_metrics(pred_y=predictions,
true_y=batchY)

            elif config.numClasses > 1:
                acc, recall, prec, f_beta = get_multi_metrics(pred_y=predictions,
true_y=batchY,
```

```
labels=labelList)

                trainSummaryWriter.add_summary(summary, step)

                return loss, acc, prec, recall, f_beta

            def devStep(batchX, batchY):
                """
                验证函数
                """
                feed_dict = {
                  lstm.inputX: batchX,
                  lstm.inputY: batchY,
                  lstm.dropoutKeepProb: 1.0
                }
                summary, step, loss, predictions = sess.run(
                    [summaryOp, globalStep, lstm.loss, lstm.predictions],
                    feed_dict)

                if config.numClasses == 1:

                    acc, precision, recall, f_beta = get_binary_metrics(pred_y=predictions,
true_y=batchY)
                elif config.numClasses > 1:
                    acc, precision, recall, f_beta = get_multi_metrics(pred_y=predictions,
true_y=batchY, labels=labelList)

                evalSummaryWriter.add_summary(summary, step)

                return loss, acc, precision, recall, f_beta

            for i in range(config.training.epoches):
                # 训练模型
                print("start training model")
                for batchTrain in nextBatch(trainReviews, trainLabels, config.batchSize):
                    loss, acc, prec, recall, f_beta = trainStep(batchTrain[0], batchTrain[1])

                    currentStep = tf.train.global_step(sess, globalStep)
                    print("train: step: {}, loss: {}, acc: {}, recall: {}, precision: {}, f_beta:
{}".format(
                        currentStep, loss, acc, recall, prec, f_beta))
                    if currentStep % config.training.evaluateEvery == 0:
                        print("\nEvaluation:")
```

```
                losses = []
                accs = []
                f_betas = []
                precisions = []
                recalls = []

                for batchEval in nextBatch(evalReviews, evalLabels, config.batchSize):
                    loss, acc, precision, recall, f_beta = devStep(batchEval[0], batchEval[1])
                    losses.append(loss)
                    accs.append(acc)
                    f_betas.append(f_beta)
                    precisions.append(precision)
                    recalls.append(recall)

                time_str = datetime.datetime.now().isoformat()
                print("{}, step: {}, loss: {}, acc: {},precision: {}, recall: {}, f_beta: {}".format(time_str, currentStep, mean(losses),

mean(accs), mean(precisions),

mean(recalls), mean(f_betas)))

            if currentStep % config.training.checkpointEvery == 0:
                # 保存模型的另一种方法，保存 checkpoint 文件
                path = saver.save(sess, "./Bi-LSTM/model/my-model", global_step=currentStep)
                print("Saved model checkpoint to {}\n".format(path))

        inputs = {"inputX": tf.saved_model.utils.build_tensor_info(lstm.inputX),
                  "keepProb": tf.saved_model.utils.build_tensor_info(lstm.dropoutKeepProb)}

        outputs = {"predictions": tf.saved_model.utils.build_tensor_info(lstm.predictions)}

        prediction_signature = tf.saved_model.signature_def_utils.build_signature_def(inputs=inputs,outputs=outputs, method_name=tf.saved_model.signature_constants.PREDICT_METHOD_NAME)
        legacy_init_op = tf.group(tf.tables_initializer(), name="legacy_init_op")
        builder.add_meta_graph_and_variables(sess, [tf.saved_model.tag_constants.SERVING],
                                             signature_def_map={"predict": prediction_signature}, legacy_init_op=legacy_init_op)
```

```
            builder.save()
    x = "this movie is full of references like mad max ii the wild one and many others the ladybug´s face it´s a clear reference or tribute to peter lorre this movie is a masterpiece we´ll talk much more about in the future"

    # 注：下面两个词典要保证和当前加载的模型对应的词典是一致的
    with open("./wordJson/word2idx.json", "r", encoding="utf-8") as f:
        word2idx = json.load(f)

    with open("./wordJson/label2idx.json", "r", encoding="utf-8") as f:
        label2idx = json.load(f)
    idx2label = {value: key for key, value in label2idx.items()}

    xIds = [word2idx.get(item, word2idx["UNK"]) for item in x.split(" ")]
    if len(xIds) >= config.sequenceLength:
        xIds = xIds[:config.sequenceLength]
    else:
        xIds = xIds + [word2idx["PAD"]] * (config.sequenceLength - len(xIds))

    graph = tf.Graph()
    with graph.as_default():
        gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=0.333)
        session_conf = tf.ConfigProto(allow_soft_placement=True, log_device_placement=False, gpu_options=gpu_options)
        sess = tf.Session(config=session_conf)

        with sess.as_default():
            checkpoint_file = tf.train.latest_checkpoint("./Bi-LSTM/model/")
            saver = tf.train.import_meta_graph("{}.meta".format(checkpoint_file))
            saver.restore(sess, checkpoint_file)

            # 获得需要喂给模型的参数，输出的结果依赖的输入值
            inputX = graph.get_operation_by_name("inputX").outputs[0]
            dropoutKeepProb = graph.get_operation_by_name("dropoutKeepProb").outputs[0]

            # 获得输出的结果
            predictions = graph.get_tensor_by_name("output/predictions:0")

            pred = sess.run(predictions, feed_dict={inputX: [xIds], dropoutKeepProb: 1.0})

    pred = [idx2label[item] for item in pred]
    print(pred)
```

4.2 模型与其它文档

本报告中所用的停用词即模型的训练结果（loss 图表等），以及训练出来的模型已上传 github。网址：https://github.com/liuminwen/use-text-to-predict-mbti