核心程序及源代码

# 1. AttriRank算法程序(AttriRank.py)

# -\*- coding: utf-8 -\*-

import numpy as np

from scipy.sparse import csr\_matrix

from sklearn import preprocessing

from sklearn.metrics.pairwise import (rbf\_kernel, cosine\_similarity,

sigmoid\_kernel, euclidean\_distances)

from collections import defaultdict

class AttriRank(object):

convergenceThreshold = 1e-10

Matrix = False

track = False

scores = {}

print\_every = 1000

track\_scores = {}

def \_\_init\_\_(self, graph, featureMatrix, itermax=100000,

weighted=True, nodeCount=None):

"""

Standardize input features and set the basic parameters

graph: [[node\_from, node\_to], ...]

featureMatrix: N \* d matrix; i-th node's feature is the i-th row

itermax: maximum iterations

weighted: transition Matrix weighted by number of links

"""

self.graph = np.array(graph)

self.featMat = preprocessing.scale(np.array(featureMatrix) / 100.0)

self.featCount = self.featMat.shape[1]

if nodeCount is None:

self.nodeCount = graph.max() + 1

else:

self.nodeCount = nodeCount

self.iterationMax = itermax

self.weighted = weighted

def ResetProbVec(self, kernel='rbf\_ap'):

"""

Calculate the reset probability vector with assigned kernel

rbf: Radial basis function

cos: (cosine similarity + 1) / 2.0

euc: 1.0 / (1 + euclidean distances)

sigmoid: (tanh(gamma <X\_i, X\_j>) + 1) / 2.0

rbf\_ap: Taylor-expansion approximated Radial basis function

"""

if kernel == 'rbf':

RBF = rbf\_kernel(self.featMat, gamma=1.0 / self.featCount)

RBF = RBF.sum(axis=0)

resetProbVec = RBF / np.sum(RBF)

elif kernel == 'cos':

Cos = (cosine\_similarity(self.featMat) + 1) / 2.0

Cos = Cos.sum(axis=0)

resetProbVec = Cos / np.sum(Cos)

elif kernel == 'euc':

Euc = 1.0 / (euclidean\_distances(self.featMat) + 1)

Euc = Euc.sum(axis=0)

resetProbVec = Euc / np.sum(Euc)

elif kernel == 'sigmoid':

gamma = 1.0 / self.featCount

Sig = sigmoid\_kernel(self.featMat, coef0=0, gamma=gamma)

Sig = (Sig + 1.0) / 2.0

Sig = Sig.sum(axis=0)

resetProbVec = Sig / np.sum(Sig)

elif kernel == 'rbf\_ap':

parameter = 1.0 / self.featCount

# w

lengths = np.einsum("ij, ij -> i", self.featMat, self.featMat)

expNormVector = np.exp(- parameter \* lengths)

# y

f\_normVec = np.einsum("i, ij -> j", expNormVector, self.featMat)

featureNormVector = f\_normVec \* (2.0 \* parameter)

# Z

outerMat = np.einsum("i, ij, ik -> jk", expNormVector,

self.featMat, self.featMat)

featureOuterNorm = outerMat \* (2.0 \* parameter \*\* 2)

# r'

first = expNormVector \* np.sum(expNormVector)

second = np.einsum("i, j, ij -> i", expNormVector,

featureNormVector, self.featMat)

third = np.einsum("i, jk, ij, ik -> i", expNormVector,

featureOuterNorm, self.featMat, self.featMat)

resetProbVec = first + second + third

# r

resetProbVec /= np.sum(resetProbVec)

self.resetProbVec = resetProbVec

def ResetProbMat(self):

"""Calculate the Q transition Matrix with RBF kernel"""

parameter = 1.0 / self.featCount

RBF = rbf\_kernel(self.featMat, gamma=parameter)

self.resetProbMat = RBF / RBF.sum(axis=0)

def TransMat(self):

"""Construct transition matrix"""#构建转移矩阵

links = defaultdict(int)

for nodefrom, nodeto in self.graph:

if self.weighted:

links[(nodefrom, nodeto)] += 1.0

else:

links[(nodefrom, nodeto)] = 1.0

entryList = list()

rowList = list()

columnList = list()

print()

for key, val in links.items():

entryList.append(val)

columnList.append(key[0])

rowList.append(key[1])

# transition matrix 过渡矩阵

traMat = csr\_matrix((entryList, (rowList, columnList)),

shape=(self.nodeCount, self.nodeCount))

self.transMat = traMat.multiply(csr\_matrix(1.0 / traMat.sum(axis=0)))

# find dangling nodes

col\_sum = np.array(traMat.sum(axis=0))[0]

self.dangVec = np.arange(col\_sum.shape[0])[col\_sum == 0]

def runPageRank(self, damp=0.85, do=True, doTrans=True, kernel='rbf\_ap'):

"""

do: whether to compute the reset probability vector

doTrans: whether to compute the transition matrix

"""

if doTrans:

self.TransMat()

print("\tGenerate transition matrix")

if do:

if self.Matrix:

self.ResetProbMat()

print("\tGenerate matrix Q")

else:

print("\tGenerate reset probability vector")

self.ResetProbVec(kernel=kernel)

if damp == 0:

scoreVector = self.resetProbVec

return scoreVector

# record the scores of each update

self.track\_scores[damp] = []

scoreVector = np.ones(self.nodeCount) / self.nodeCount

for iteration in range(self.iterationMax):

leak\_scores = np.sum(scoreVector[self.dangVec])

dangScore = leak\_scores \* self.resetProbVec

if self.Matrix:

teleport\_prob = self.resetProbMat.dot(scoreVector)

else:

teleport\_prob = self.resetProbVec

newScoreVector = (1.0 - damp) \* teleport\_prob + \

damp \* (self.transMat.dot(scoreVector) + dangScore)

error = np.linalg.norm(newScoreVector - scoreVector)

if error < self.convergenceThreshold:

break

scoreVector = newScoreVector

if self.track:

self.track\_scores[damp].append(scoreVector)

return scoreVector

def TotalRank(self, alpha=1, beta=1, kernel='rbf\_ap'):

"""

Implementation of TotalRank with beta distribution as the prior

(alpha, beta): parameters for the beta distribution

"""

print("\tGenerate transition matrix and reset probability vector")

self.TransMat()

self.ResetProbVec(kernel=kernel)

rho\_t = self.resetProbVec \* beta / (alpha + beta)

pi\_t = self.resetProbVec \* beta / (alpha + beta)

for iteration in range(self.iterationMax):

dangScore = np.sum(rho\_t[self.dangVec]) \* self.resetProbVec

P\_rho = (self.transMat.dot(rho\_t) + dangScore)

rho\_next = P\_rho \* (iteration + alpha) / (iteration+1+alpha+beta)

pi\_t += rho\_next

error = np.linalg.norm(rho\_next)

if iteration % self.print\_every == (self.print\_every - 1):

print("\tIteration %d:\t%.10f" % (iteration + 1, error))

if error < self.convergenceThreshold:

break

rho\_t = rho\_next

if self.track:

self.track\_scores['total'].append(pi\_t)

return pi\_t

def runModel(self, factors=[0.85], Matrix=False, track=False,

TotalRank=False, alpha=1, beta=1, print\_every=1000,

kernel='rbf\_ap'):

"""

Give a list of damping factors to work with

return a dict: key=(damp factor); value=(scores of each node)

Matrix: use the exact Q or approximated r (True for Q)

track: record the score vector at each iteration during updating

"""

self.Matrix = Matrix

self.track = track

self.print\_every = print\_every

scores = {}

if TotalRank:

print("Run AttriRank with prior...")

scores['total'] = list(self.TotalRank(alpha=alpha, beta=beta))

else:

do = True

doTrans = True

for dampFac in factors:

print("Run AttriRank, damp:", dampFac)

score\_vec = self.runPageRank(dampFac, do=do, doTrans=doTrans,

kernel=kernel)

# already have reset vector and transition matrix

do = False

doTrans = False

scores[str(dampFac)] = list(score\_vec)

print("\tDone.")

self.scores = scores

return scores

# 2. AttriRank算法的调用程序（main.py）

"""

Implementation of AttriRank.

Author: Yi-An Lai

For more details, refer to the paper:

Unsupervised Ranking using Graph Structures and Node Attributes

Chin-Chi Hsu, Yi-An Lai, Wen-Hao Chen, Ming-Han Feng, and Shou-De Lin

Web Search and Data Mining (WSDM), 2017

"""

import argparse

import numpy as np

import pandas as pd

from AttriRank import AttriRank

def parse\_args():

'''

Parses AttriRank arguments.

'''

parser = argparse.ArgumentParser(description="Run AttriRank.")

parser.add\_argument('--inputgraph', nargs='?',

default='D:\\lxw\\AttriRank-master\\sample\\graph.edgelist',

help='Input graph path')

parser.add\_argument('--inputfeature', nargs='?',

default='D:\\lxw\\AttriRank-master\\sample\\graph.feature',

help='Input feature path')

parser.add\_argument('--output', nargs='?', default='graph.rankscore0.85',

help='Output rankscore path')#输出到当前路径

parser.add\_argument('--kernel', default='rbf\_ap',

help='Kernel: rbf\_ap, rbf, cos, euc, sigmoid')

parser.add\_argument('--damp', nargs='\*', default=[0.85], type=float,

help='damping parameters')#阻尼系数

parser.add\_argument('--totalrank', dest='totalrank', action='store\_true',

help='Use TotalRank or not. Default is False.')

parser.set\_defaults(totalrank=False)

parser.add\_argument('--alpha', type=float, default=1.0,

help='alpha of beta distribution. Default is 1.0.')

parser.add\_argument('--beta', type=float, default=1.0,

help='beta of beta distribution. Default is 1.0.')

parser.add\_argument('--matrix', dest='matrix', action='store\_true',

help='Using original Q matrix. Default is False.')

parser.set\_defaults(matrix=False)

parser.add\_argument('--print\_every', type=int, default=1000,

help='Print TotalRank process. Default is 1000.')

parser.add\_argument('--itermax', type=int, default=100000,

help='Number of max iterations. Default is 100000.')

parser.add\_argument('--weighted', dest='weighted', action='store\_true',

help='Specifying (un)weighted. Default is unweighted.')

parser.set\_defaults(weighted=False)

parser.add\_argument('--undirected', dest='directed', action='store\_false',

help='Graph is (un)directed. Default is directed.')

parser.set\_defaults(directed=True)

return parser.parse\_args()

def load\_graph(filename):

"""Read the graph into numpy array"""

return pd.read\_csv(filename, sep=' ', header=None).values

def load\_features(filename):

"""Read the features into numpy array, first column as index"""

return pd.read\_csv(filename, header=None).set\_index(0).values

def main(args):

"""

Pipeline for unsupervised ranking using graph and node features

"""

graph = load\_graph(args.inputgraph)

feat = load\_features(args.inputfeature)

N = len(feat)

if not args.directed:

graph = np.concatenate((graph, graph[:, [1, 0]]))

AR = AttriRank(graph, feat, itermax=args.itermax, weighted=args.weighted,

nodeCount=N)

scores = AR.runModel(factors=args.damp, kernel=args.kernel,

Matrix=args.matrix, TotalRank=args.totalrank,

alpha=args.alpha, beta=args.beta,

print\_every=args.print\_every)

df = pd.DataFrame(data=scores)

df.to\_csv(args.output, float\_format='%.16f', index\_label='node\_id')

args = parse\_args()

main(args)

# 3. LDA主题模型程序（LDA模型.py）

from gensim import corpora, models

import pandas as pd

# Global Dictionary

stopwords = [line.strip() for line in open('D:\\lxw\\第一篇小论文\\论文程序\\LDA主题模型\\中文停用词表.txt', encoding='utf-8').readlines()]#加载停用词

punctuation = [line.strip() for line in open('D:\\lxw\\第一篇小论文\\论文程序\\LDA主题模型\\标点符号.txt', encoding='utf-8').readlines()]#标点符号

words\_nature = ('n', 'nr', 'ns', 'nt', 'eng', 'v', 'd') # 可用的词性

def remove\_punctuation(ls): # 去除标点符号

return [word for word in ls if word not in punctuation]

def remove\_stopwords(ls): # 去除停用词

return [word for word in ls if word not in stopwords]

def remove\_word(ls): # 去除停用词

return [word for word in ls if len(word) != 1]

import pkuseg

import csv

lexicon = [line.strip() for line in open('D:\\lxw\\第一篇小论文\\论文程序\\LDA主题模型\\自定义词典.txt', encoding='utf-8').readlines()] #希望分词时用户词典中的词固定不分开

seg = pkuseg.pkuseg(model\_name='medicine',user\_dict=lexicon) #加载模型，给定用户词典

with open('D:\\lxw\\第三篇小论文\\最终数据.csv','r',encoding='gb18030') as csvfile:

reader = csv.reader(csvfile)

column6 = [row[5] for row in csvfile]#读取第6列的数据

words\_ls = []

for text in column6:

print('text',text)

words = remove\_word(remove\_stopwords(remove\_punctuation((seg.cut(text))))) # 进行分词

print(words)

words\_ls.append(words)

print("words\_ls",words\_ls)

# 生成语料词典

dictionary = corpora.Dictionary(words\_ls)

print(dictionary)

# 生成稀疏向量集

corpus = [dictionary.doc2bow(words) for words in words\_ls]

# LDA模型，num\_topics设置聚类数，即最终主题的数量

lda = models.ldamodel.LdaModel(corpus=corpus, id2word=dictionary, num\_topics=6)

# print(lda)

# 展示每个主题的前20的词语

for topic in lda.print\_topics(num\_words=20):

print(topic)

# for i in range(6):

# a = len(lda.print\_topic(i))

# print('len',a)

# # 推断每个语料库中的主题类别

print('推断：')

for e, values in enumerate(lda.inference(corpus)[0]):

topic\_val = 0

topic\_id = 0

for tid, val in enumerate(values):

if val > topic\_val:

topic\_val = val

topic\_id = tid

print(topic\_id, '->', column6[e])

import pandas as pd

data = pd.DataFrame(lda.print\_topics())

data.to\_csv('D:\\lxw\\主题.txt', sep='\t', index=0, header=0)

print('推断：')

# 1. 创建文件对象

f = open('D:\\lxw\\主题.csv','w',encoding='utf-8',newline='')

# 2. 基于文件对象构建 csv写入对象

csv\_writer = csv.writer(f)

# 3. 构建列表头

csv\_writer.writerow(["id"])

for e, values in enumerate(lda.inference(corpus)[0]):

topic\_val = 0

topic\_id = 0

for tid, val in enumerate(values):

if val > topic\_val:

topic\_val = val

topic\_id = tid

print(topic\_id)

print(type(topic\_id))

# 4. 写入csv文件内容

csv\_writer.writerow([topic\_id])

f.close()

print('处理完成')

# 4. 主题-困惑度程序（困惑度.py）

def ldamodel(num\_topics):

from gensim import corpora, models

# Global Dictionary

stopwords = [line.strip() for line in open('D:\\lxw\\第一篇小论文\\论文程序\\LDA主题模型\\中文停用词表.txt', encoding='utf-8').readlines()] # 加载停用词

punctuation = [line.strip() for line in open('D:\\lxw\\第一篇小论文\\论文程序\\LDA主题模型\\标点符号.txt', encoding='utf-8').readlines()]#标点符号

words\_nature = ('n', 'nr', 'ns', 'nt', 'eng', 'v', 'd') # 可用的词性

def remove\_punctuation(ls): # 去除标点符号

return [word for word in ls if word not in punctuation]

def remove\_stopwords(ls): # 去除停用词

return [word for word in ls if word not in stopwords]

def remove\_word(ls): # 去除停用词

return [word for word in ls if len(word) != 1]

import csv

import pkuseg

lexicon = [line.strip() for line in open('D:\\lxw\\第一篇小论文\\论文程序\\LDA主题模型\\自定义词典.txt', encoding='utf-8').readlines()] #希望分词时用户词典中的词固定不分开

seg = pkuseg.pkuseg(model\_name='medicine', user\_dict=lexicon) # 加载模型，给定用户词典

with open('D:\\lxw\\第一篇小论文\\论文数据\\最终数据.csv', 'r',encoding='gb18030') as csvfile:

reader = csv.reader(csvfile)

column6 = [row[5] for row in reader] # 读取第7列的数据

words\_ls = []

for text in column6:

words = remove\_word(remove\_stopwords(remove\_punctuation((seg.cut(text)))))

words\_ls.append(words)

dictionary = corpora.Dictionary(words\_ls)

corpus = [dictionary.doc2bow(text) for text in

words\_ls] # corpus里面的存储格式（0, 1), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1)

corpora.MmCorpus.serialize('corpus.mm', corpus)

lda = models.LdaModel(corpus=corpus, id2word=dictionary, random\_state=1,

num\_topics=num\_topics) # random\_state 等价于随机种子的random.seed()，使每次产生的主题一致

topic\_list = lda.print\_topics(num\_topics, 10)

# print("主题的单词分布为：\n")

# for topic in topic\_list:

# print(topic)

return lda, dictionary

import math

def perplexity(ldamodel, testset, dictionary, size\_dictionary, num\_topics):

print('the info of this ldamodel: \n')

print('num of topics: %s' % num\_topics)

prep = 0.0

prob\_doc\_sum = 0.0

topic\_word\_list = []

for topic\_id in range(num\_topics):

topic\_word = ldamodel.show\_topic(topic\_id, size\_dictionary)

dic = {}

for word, probability in topic\_word:

dic[word] = probability

topic\_word\_list.append(dic)

doc\_topics\_ist = []

for doc in testset:

doc\_topics\_ist.append(ldamodel.get\_document\_topics(doc, minimum\_probability=0))

testset\_word\_num = 0

for i in range(len(testset)):

prob\_doc = 0.0 # the probablity of the doc

doc = testset[i]

doc\_word\_num = 0

for word\_id, num in dict(doc).items():

prob\_word = 0.0

doc\_word\_num += num

word = dictionary[word\_id]

for topic\_id in range(num\_topics):

# cal p(w) : p(w) = sumz(p(z)\*p(w|z))

prob\_topic = doc\_topics\_ist[i][topic\_id][1]

prob\_topic\_word = topic\_word\_list[topic\_id][word]

prob\_word += prob\_topic \* prob\_topic\_word

prob\_doc += math.log(prob\_word) # p(d) = sum(log(p(w)))

prob\_doc\_sum += prob\_doc

testset\_word\_num += doc\_word\_num

prep = math.exp(-prob\_doc\_sum / testset\_word\_num) # perplexity = exp(-sum(p(d)/sum(Nd))

print("模型困惑度的值为 : %s" % prep)

return prep

from gensim import corpora, models

import matplotlib.pyplot as plt

def graph\_draw(topic, perplex): # 做主题数与困惑度的折线图

x = topic

y = perplex

plt.plot(x, y, color="black", linewidth=1)

plt.xlabel("Number of Topic")

plt.ylabel("Perplexity")

plt.show()

if \_\_name\_\_ == '\_\_main\_\_':

for i in range(5, 6, 1): # 每5篇文档中抽取1篇（这里只是为了调试最优结果，可以直接设定不循环）

print("抽样为" + str(i) + "时的perplexity")

a = range(1, 20, 5) # 主题个数20个主题以5为间隔

p = []

for num\_topics in a:

lda, dictionary = ldamodel(num\_topics)

corpus = corpora.MmCorpus('corpus.mm')

testset = []

for c in range(int(corpus.num\_docs / i)):#5篇里面抽取1篇作为测试集，一般20%-30%

testset.append(corpus[c \* i])

prep = perplexity(lda, testset, dictionary, len(dictionary.keys()), num\_topics)

p.append(prep)

graph\_draw(a, p)

# 5. 情感极性分析程序（情感极性分析.py）

# \_\*\_ coding:utf-8 \_\*\_

import pandas as pd

import pkuseg

from collections import defaultdict

import os

import re

import jieba

import codecs

import pandas as pd

def get\_label\_data(path):

"""

根据情感词汇本体返回相应的标签

"""

sem\_data = pd.read\_excel(path)

for i in range(sem\_data.shape[0]):

score = 0

if sem\_data.iloc[i, 6] == 2:

sem\_data.iat[i, 6] = -1

if sem\_data.iloc[i, 9] == 2:

sem\_data.iat[i, 9] = -1

# 是否需要辅助情感分类，目前先不要啦

# if sem\_data.iloc[i,8] >= 0:

# score += sem\_data.iloc[i, 8] \* sem\_data.iloc[i, 9]

# print(score)

# 增加每个词语的情感强度值

score += sem\_data.iloc[i, 5] \* sem\_data.iloc[i, 6]

sem\_data.iat[i, -1] = score

# 定义情感字典

match\_dict = {

"joy": ["PA", "PE"],

"surprise": ["PC"],

"anger": ["NA"],

"sadness": ["NB", "NJ", "NH", "PF"],

"fear": ["NI", "NC", "NG"],

"disgust": ["ND", "NE", "NN", "NK", "NL"],

}

# 获取情感类别

label\_dict = {}

keys = match\_dict.keys()

for k in keys:

word\_dict = {}

for i in range(sem\_data.shape[0]):

label = sem\_data.iloc[i, 4]

if label in match\_dict[k]:

word\_dict[sem\_data.iloc[i, 0]] = sem\_data.iloc[i, -1]

label\_dict[k] = word\_dict

return label\_dict

path = "D:\\xyw\\中文情感词汇本体.xlsx"

label\_dict\_7\_class = get\_label\_data(path)

def seg\_word(sentence):

"""使用jieba对文档分词"""

seg\_list = jieba.cut(sentence)

seg\_result = []

for w in seg\_list:

seg\_result.append(w)

# 读取停用词文件

stopwords = set()

fr = codecs.open('D:\\xyw\\中文停用词表.txt', 'r', encoding='utf-8')

for word in fr:

stopwords.add(word.strip())

fr.close()

# 去除停用词

return list(filter(lambda x: x not in stopwords, seg\_result))

def classify\_words(word\_dict):

"""词语分类,找出情感词、否定词、程度副词"""

# 读取情感字典文件

sen\_file = open('D:\\xyw\\BosonNLP\_sentiment\_score.txt', 'r+', encoding='utf-8')

# 获取字典文件内容

sen\_list = sen\_file.readlines()

# 创建情感字典

sen\_dict = defaultdict()

# 读取字典文件每一行内容，将其转换为字典对象，key为情感词，value为对应的分值

for s in sen\_list:

# 每一行内容根据空格分割，索引0是情感词，索引01是情感分值

s\_split=s.split()

if len(s\_split)==2:

sen\_dict[s.split(' ')[0]] = s.split(' ')[1]

# 读取否定词文件

not\_word\_file = open('D:\\xyw\\否定词.txt', 'r+', encoding='utf-8')

# 由于否定词只有词，没有分值，使用list即可

not\_word\_list = not\_word\_file.readlines()

# 读取程度副词文件

degree\_file = open('D:\\xyw\\程度副词.txt', 'r+', encoding='utf-8')

degree\_list = degree\_file.readlines()

degree\_dic = defaultdict()

# 程度副词与情感词处理方式一样，转为程度副词字典对象，key为程度副词，value为对应的程度值

for d in degree\_list:

# print(d)

degree\_dic[d.split(' ')[0]] = d.split(' ')[1]

# 分类结果，词语的index作为key,词语的分值作为value，否定词分值设为-1

sen\_word = dict()

not\_word = dict()

degree\_word = dict()

# 分类

for word in word\_dict.keys():

if word in sen\_dict.keys() and word not in not\_word\_list and word not in degree\_dic.keys():

# 找出分词结果中在情感字典中的词

sen\_word[word\_dict[word]] = sen\_dict[word]

elif word in not\_word\_list and word not in degree\_dic.keys():

# 分词结果中在否定词列表中的词

not\_word[word\_dict[word]] = -1

elif word in degree\_dic.keys():

# 分词结果中在程度副词中的词

degree\_word[word\_dict[word]] = degree\_dic[word]

degree\_file.close()

not\_word\_file.close()

# 将分类结果返回

return sen\_word, not\_word, degree\_word

def list\_to\_dict(word\_list):

"""将分词后的列表转为字典，key为单词，value为单词在列表中的索引，索引相当于词语在文档中出现的位置"""

data = {}

for x in range(0, len(word\_list)):

data[word\_list[x]] = x

return data

def get\_init\_weight(sen\_word, not\_word, degree\_word):

# 权重初始化为1

W = 1

# 将情感字典的key转为list

sen\_word\_index\_list = list(sen\_word.keys())

if len(sen\_word\_index\_list) == 0:

return W

# 获取第一个情感词的下标，遍历从0到此位置之间的所有词，找出程度词和否定词

for i in range(0, sen\_word\_index\_list[0]):

if i in not\_word.keys():

W \*= -1

elif i in degree\_word.keys():

# 更新权重，如果有程度副词，分值乘以程度副词的程度分值

W \*= float(degree\_word[i])

return W

def socre\_sentiment(sen\_word, not\_word, degree\_word, seg\_result):

"""计算得分"""

# 权重初始化为1

W = 1

score = 0

# 情感词下标初始化

sentiment\_index = -1

# 情感词的位置下标集合

sentiment\_index\_list = list(sen\_word.keys())

# 遍历分词结果(遍历分词结果是为了定位两个情感词之间的程度副词和否定词)

for i in range(0, len(seg\_result)):

# 如果是情感词（根据下标是否在情感词分类结果中判断）

if i in sen\_word.keys():

# 权重\*情感词得分

score += W \* float(sen\_word[i])

# 情感词下标加1，获取下一个情感词的位置

sentiment\_index += 1

if sentiment\_index < len(sentiment\_index\_list) - 1:

# 判断当前的情感词与下一个情感词之间是否有程度副词或否定词

for j in range(sentiment\_index\_list[sentiment\_index], sentiment\_index\_list[sentiment\_index + 1]):

# 更新权重，如果有否定词，取反

if j in not\_word.keys():

W \*= -1

elif j in degree\_word.keys():

# 更新权重，如果有程度副词，分值乘以程度副词的程度分值

W \*= float(degree\_word[j])

# 定位到下一个情感词

if sentiment\_index < len(sentiment\_index\_list) - 1:

i = sentiment\_index\_list[sentiment\_index + 1]

return score

# 计算得分

def setiment\_score(sententce):

# 1.对文档分词

seg\_list = seg\_word(sententce)

# 2.将分词结果列表转为dic，然后找出情感词、否定词、程度副词

sen\_word, not\_word, degree\_word = classify\_words(list\_to\_dict(seg\_list))

# 3.计算得分

score = socre\_sentiment(sen\_word, not\_word, degree\_word, seg\_list)

return score

import csv

f = open('D:\\lxw\\信息情感.csv', 'w', encoding='utf-8', newline='')

# 2. 基于文件对象构建 csv写入对象

csv\_writer = csv.writer(f)

# 3. 构建列表头

csv\_writer.writerow(["score"])

with open('D:\\lxw\\\\第二次数据\\情绪分析.csv','r',encoding='gb18030',errors='ignore') as csvfile:

reader = csv.reader(csvfile)

column7 = [row[6] for row in reader]#读取第6列的数据

for sentence in column7:

print('sentence',sentence)

a = [setiment\_score(sentence)]

print(a)

csv\_writer.writerow(a)

f.close()

print('处理完成')

# 6. 网络效率程序（网络效率.py）

import networkx as nx

import pandas as pd

import numpy as np

import csv

import re

g = nx.DiGraph()

node\_filename = "D:\\lxw\\第一篇小论文\\关键用户识别\\node.csv"

edge\_filename = "D:\\lxw\\第一篇小论文\\关键用户识别\\edge.csv"

csv\_node=open(node\_filename)

csv\_edge=open(edge\_filename)

node\_line = csv\_node.readline() # 逐行读取点文件

while node\_line:

temp\_line = re.split(',', node\_line)

# print(temp\_line)

g.add\_node(temp\_line[0])

node\_line = csv\_node.readline()

print('node\_num',g.number\_of\_nodes())

print(g.nodes)

print(type(g.nodes))

edge\_line = csv\_edge.readline() # 逐行读取边文件

while edge\_line:

temp\_line = re.split(",", edge\_line) # 把边文件中的每一行用","分隔开

temp\_line[1] = temp\_line[1].replace('\n', '').replace('\r', '') # 去除文本内容中的换行符

# print(temp\_line)

g.add\_edge(temp\_line[0], temp\_line[1]) # 向网络中添加边

edge\_line = csv\_edge.readline()

print('edge\_num',g.number\_of\_edges())

step =0

results = []

node\_score =pd.read\_excel(r'D:\\lxw\\第一篇小论文\\关键用户识别\\节点排序.xlsx') #默认读取第一个sheet,sheet\_name=2

print(node\_score['node\_Id4'])

for i in node\_score['node\_Id4']:

i=str(i)

print('i',i)

g.remove\_node(i)

print('step',step)

N = g.number\_of\_nodes()

print('N',N)

print('edge\_num',g.number\_of\_edges())

sumeff =0

for u in g.nodes(): # 遍历流量图F的每个点

path = nx.shortest\_path\_length(g,source=u) # 在网络G中计算从u开始到其他所有节点（注意包含自身）的最短路径长度。如果两个点之间没有路径，那path里也不会存储这个目标节点（比前面的代码又省了判断是否has\_path的过程）

for v in path.keys(): # path是一个字典，里面存了所有目的地节点到u的最短路径长度

if u != v: # 如果起终点不同才累加计算效率

sumeff += 1 / path[v]

result = (1 / (N \* (N - 1))) \* sumeff # 计算网络剩余效率

print('result',result)

results.append(result)

step = step + 1

if result == 0:

break

print('results',results)

f = open('C:\\Users\\Administrator\\Desktop\\网络效率.csv', 'w', encoding='gb18030', newline='')

csv\_writer = csv.writer(f)

for i in results:

i=list(i)

csv\_writer.writerow(i)

print('处理完成')