Homework 2

顾淳 19307110344

In [1]:

```
import nltk
from nltk import sent_tokenize, word_tokenize, ngrams, FreqDist, ConditionalFreqDist
import matplotlib.pyplot as plt
from collections import defaultdict
import numpy as np
import json
import re
import random
import math
import time
import pickle
import pandas as pd
import copy
import os
```

Task 1

Train word embeddings using SGNS: Use our enwiki_20220201.json as training data.

Preprocessing

In [2]:

```
np.random.seed(0)
wiki_data = []
with open("enwiki_20220201.json", "r") as f:
    for each_line in f:
        record = json.loads(each_line)
        wiki_data.append(record)
l = len(wiki_data)
```

In [3]:

```
def preprocess(sentence):
    # 将句内除字母、数字、空格外的所有字符替换为空格
    res = re.sub(r'[^\w\s]', ' ', sentence)
    return res.lower()
```

对每篇文章进行分词

In [5]:

```
words = [word_tokenize(preprocess(text['text'])) for text in wiki_data]
```

定义 dataset 类

In [6]:

```python
class dataset:
    def __init__(self,words,**kwargs):
        # 对数据集进行采样，随机丢弃出现频率高的单词
        self.words=dataset.downsample(words)
        self.vocab=FreqDist([word for text in self.words for word in text])
        self.vocab['<unknown>']=0
        # 序号->单词 映射，频率
        self.index2word,self.frequency=np.array(list(self.vocab.items())).T
        self.frequency=self.frequency.astype(np.float32)
        self.frequency/=self.frequency.sum()
        self.indexes=np.arange(len(self.index2word))
        # 单词->序号 映射
        self.word2index={word:i for i,word in enumerate(self.index2word)}
        # 获得中心词，上下文，负采样
        self.centers,self.contexts,self.negatives=self.get_center_context(**kwargs)
    @staticmethod
    def downsample(words,t=1e-4):
        '''
        下采样高频词
        '''
        vocab = FreqDist([word for sent in words for word in sent])
        N = vocab.N()
        # 按照概率，随机丢弃高频词
        def drop(word):
            return random.random()<max(1-math.sqrt(t/(vocab[word]/N)),0)
        return [[word for word in text if not drop(word)] for text in words]

    def get_center_context(self, **kwargs):
        """
        获得中心词、上下文、负采样
        """
        window=kwargs.get('window',2)
        alpha=kwargs.get('alpha',0.75)
        k=kwargs.get('k',5)
        centers=[]
        contexts=[]
        # 从数据集中获得中心词和上下文
        for sent in self.words:
            for i in range(window,len(sent)-window):
                center=self.word2index[sent[i]]
                centers.append(center)
                context = sent[i-window:i]+sent[i+1:i+window+1]
                contexts.append([self.word2index[word] for word in context])
        centers=np.array(centers)
        contexts=np.array(contexts)
        weights=np.array([self.vocab[word] for word in self.index2word])**alpha
        weights = weights/np.sum(weights)
        # 负采样
        negatives=self.negative_sample(contexts,weights,k)
        not_na=(negatives.isna().sum(axis=1)==0).values
        negatives=negatives.values
        return centers[not_na],contexts[not_na],negatives[not_na].astype(int)

    def negative_sample(self,contexts,weights,k):
        """
        负采样
        """
        # 随机生成负采样样本
        neg_sample=np.random.choice(self.indexes,size=(len(contexts),len(contexts[0])*k),p=weights)
```

```
        a=contexts[:,None].repeat(len(contexts[0])*k,axis=1)
        b=neg_sample[:,:,None]
        # 获得与上下文有重叠的部分（需要被取代）
        condition=(~(a==b)).prod(axis=-1)
        new=np.where(condition,neg_sample,np.nan)
        df=pd.DataFrame(new)
        # 用前项或后项填充重叠部分
        df.fillna(method='bfill',axis=1,inplace=True)
        df.fillna(method='ffill',axis=1,inplace=True)
        return df

    def __getitem__(self,i):
        """
        从数据集中获取中心词、上下文、负采样
        """
        centers=self.centers[i]
        context_neg=np.concatenate([self.contexts[i],self.negatives[i]],axis=1)
        # 上下文标签为 1，负采样标签为 0
        labels=np.concatenate([np.ones_like(self.contexts[i]),np.zeros_like(self.negatives[i])],axis
        return centers,context_neg,labels

    def __len__(self):
        """
        数据集大小
        """
        return len(self.centers)
```

定义 batch_loader 类

用于批量读取数据

In [7]:

```
class batch_loader:
    def __init__(self,dataset,batch=16,shuffle=True):
        self.dataset=dataset
        self.batch=batch
        self.i=0
        self.N=len(dataset)
        # 随机取样
        self.shuffle=shuffle

    def __iter__(self):
        return self

    def __next__(self):
        if self.i>self.N:
            self.i=0
            #raise StopIteration
        self.i+=self.batch
        # 随机取样
        if self.shuffle:
            x=random.randint(1,self.N-self.batch)
        else:
            x=self.i
        return self.dataset[x:x+self.batch]
```

定义 skip-gram 模型

In [8]:

```
class mymodel:
    def __init__(self, embed_dim, vocab_size):
        self.dim = embed_dim
        self.vocab_size = vocab_size
        # 中心词矩阵
        self.embed_v = np.random.randn(self.vocab_size, self.dim)
        # 上下文矩阵
        self.embed_w = np.random.randn(self.vocab_size, self.dim)

    def __call__(self, center, context, label):
        self.center = center
        self.context = context
        # 从中心词、上下文矩阵中取出对应的词向量
        self.v = v = np.expand_dims(self.embed_v[center], 1)
        self.w = w = self.embed_w[context]
        # 中心词向量和上下文向量点乘
        pred = (v * w).sum(axis=-1)
        # 计算 sigmoid
        logit = sigmoid(pred)
        # 损失函数
        loss = -np.mean((label * np.log(logit) + (1 - label) * np.log(1 - logit)))
        # 计算中心词和上下文矩阵的梯度，便于后续梯度下降
        self.dv = -(w * np.expand_dims(label * (1 - logit) - (1 - label) * logit, -1)).sum(axis=1,
        self.dw = -(v * np.expand_dims(label * (1 - logit) - (1 - label) * logit, -1))
        return loss

    def step(self, lr=1e-5):
        """
        反向传播更新参数
        """
        # 梯度下降
        self.embed_v[self.center] -= self.dv.reshape(-1, self.dv.shape[-1]) * lr
        self.embed_w[self.context.reshape(-1)] -= self.dw.reshape(-1, self.dw.shape[-1]) * lr

    def save(self, path):
        np.save(path + '.npy',
                {'dim': self.dim, 'vocab_size': self.vocab_size, 'embed_v': self.embed_v, 'embed_w':

    def load(self, path):
        dic = np.load(path + '.npy', allow_pickle=True).item()
        self.dim = dic['dim']
        self.vocab_size = dic['vocab_size']
        self.embed_v = dic['embed_v']
        self.embed_w = dic['embed_w']

def sigmoid(x):
    """
    sigmoid 函数
    """
    s = 1 / (1 + np.exp(-x)+1e-8)
    return s

def onehot(x, size):
    """
    将序号转换为onehot形式（未使用，由于计算量太大，效率很低）
    """
    if len(x.shape)==2:
        return (np.arange(size)==x[:, :, None]).astype(int)
    elif len(x.shape)==1:
```

```
        return (np.arange(size)==x[:,None]).astype(int)
```

定义训练函数（由于数据集极大，故不设置验证集）

In [9]:

```
def train(kwargs):
    """
    训练函数
    """
    # 如果改变 windows, k, alpha 等超参数，则需要更新训练集
    if kwargs.get('change',False):
        train_set.centers,train_set.contexts,train_set.negatives=train_set.get_center_context(**kwar
    # 获得 batchloader
    train_loader=batch_loader(train_set,kwargs['batch'],shuffle=kwargs.get('shuffle',False))
    # 获得模型
    m=mymodel(kwargs['embed_dim'],len(train_set.vocab))
    if os.path.exists(kwargs['exp_name']) and kwargs.get('load',False):
        print('loading weights...')
        m.load(kwargs['exp_name'])
    i=0
    lr=kwargs['lr']
    best_loss=10000
    best_iter=0
    for data in train_loader:
        i+=1
        # 前向传播，计算 loss
        loss=m(*data)
        # 反向传播更新参数
        m.step(lr=lr)
        if i%20000==0 and kwargs.get('print',True):
            print('Exp:%s, Iteration:%06d, Loss:%.3f'%(kwargs['exp_name'],i,loss))
        # learning rate decay
        if i%kwargs['lr_decay_step']==0:
            lr/=10
            if kwargs.get('print',True):
                print('lr change to %f'%lr)
        # 保存最佳 loss，用于后续比较
        if best_loss>loss:
            best_loss=np.copy(loss)
            best_iter=i
        if kwargs.get('early_stop',None) and kwargs['early_stop']<i:
            break
    m.save(kwargs['exp_name'])
    print('Exp:%s, best iter:%d, best loss:%f'%(kwargs['exp_name'],best_iter,best_loss))
    return {'loss':loss,'model':m}
```

初始化训练集

In [10]:

```
j=0
results={}
kwargs={
    'exp_name':'%03d'%j,   # 实验名称
    'embed_dim':50,     # 词向量长度
    'window':2,      # 上下文窗口大小
    'alpha':0.75,    # 负采样时，用于确定每个单词的权重
    'k':5,      # 每个 context 采样 5 个负采样
    'lr':0.001,   # learning rate
    'batch':1024,   # 每步 16个中心词
    'lr_decay_step':20000,
    'early_stop':1000000, # 训练 100万步后停止
    'shuffle':True,
    'print':True
}
train_set=dataset(words,**kwargs)
```

查看训练集大小

In [11]:

```
# 单词数，训练集大小
train_set.vocab.B(),len(train_set)
```

Out[11]:

(357306, 24220900)

进行实验选择各超参数

对比词向量长度分别为30，50，100，200时，模型的表现。

实验发现当词向量长度为 200 时，效果最好。

（由于实验耗时长，耗内存高，所以分很多次进行，未保存结果）

In [ ]:

```
j=0
embed_dims=[50,100,200,300]
for embed_dim in embed_dims:
    kwargs={
        'exp_name':'%03d'%j,
        'embed_dim':embed_dim,
        'window':2,
        'alpha':0.75,
        'k':5,
        'lr':0.001,
        'batch':16,
        'lr_decay_step':200000,
        'early_stop':1000000,
        'change':True,
        'shuffle':True,
        'print':True
    }
    print('\n\nEmbed_dim =',embed_dim)
    results[kwargs['exp_name']]={'train':train(kwargs),'kwargs':kwargs}
    j+=1
```

对比上下文窗口大小分别为3，5，7时，模型的表现。

实验发现当上下文窗口大小为 5 时，效果最好。

In [ ]:

```
windows=[1,2,3]
for window in windows:
    kwargs={
        'exp_name':'%03d'%j,
        'embed_dim':200,
        'window':window,
        'alpha':0.75,
        'k':5,
        'lr':0.001,
        'batch':16,
        'lr_decay_step':200000,
        'early_stop':1000000,
        'change':True,
        'shuffle':True,
        'print':True
    }
    print('\n\nwindow =',window)
    results[kwargs['exp_name']]={'train':train(kwargs),'kwargs':kwargs}
    j+=1
```

对比 alpha 大小分别为 0.5，0.75，1时，模型的表现。

实验发现当 alpha 大小为 0.75 时，效果最好。

In [ ]:

```
alphas=[0.5,0.75,1]
for alpha in alphas:
    kwargs={
        'exp_name':'%03d'%j,
        'embed_dim':200,
        'window':2,
        'alpha':alpha,
        'k':5,
        'lr':0.001,
        'batch':16,
        'lr_decay_step':200000,
        'early_stop':1000000,
        'change':True,
        'shuffle':True,
        'print':True
    }
    print('\n\nalpha =',alpha)
    results[kwargs['exp_name']]={'train':train(kwargs),'kwargs':kwargs}
    j+=1
```

对比 k （每个上下文采样 k 个负样本）大小分别为3，5，7时，模型的表现。

实验发现当 k 大小为 5 时，效果最好。

In [ ]:

```
ks=[3,5,7]
for k in ks:
    kwargs={
        'exp_name':'%03d'%j,
        'embed_dim':50,
        'window':2,
        'alpha':0.75,
        'k':k,
        'lr':0.001,
        'batch':16,
        'lr_decay_step':200000,
        'early_stop':1000000,
        'change':True,
        'shuffle':True,
        'print':True
    }
    results[kwargs['exp_name']]={'train':train(kwargs),'kwargs':kwargs}
    j+=1
```

对比初始 learning rate 大小分别为 1e-2，1e-3，1e-4, 1e-5 时，模型的表现。

实验发现当初始 learning rate 大小为 1e-2 时，效果最好。

In [ ]:

```
lrs=[1e-2,1e-3,1e-4,1e-5]
for lr in lrs:
    kwargs={
        'exp_name':'%03d'%j,
        'embed_dim':50,
        'window':2,
        'alpha':0.75,
        'k':5,
        'lr':lr,
        'batch':16,
        'lr_decay_step':400000,
        'early_stop':2000000,
        'shuffle':True,
        'print':True
    }
    results[kwargs['exp_name']]={'train':train(kwargs),'kwargs':kwargs}
    j+=1
```

用最好的一组超参数训练，得到后续要使用的模型

In [12]:

```
j=90
kwargs={
        'exp_name':'%03d'%j,
        'embed_dim':200,
        'window':5,
        'alpha':0.75,
        'k':5,
        'lr':1e-2,
        'batch':16,
        'lr_decay_step':2000000,
        'early_stop':10000000,
        'shuffle':True,
        'print':True
    }
```

In [ ]:

```
results[kwargs['exp_name']]={'train':train(kwargs),'kwargs':kwargs}
```

In [18]:

```
model=results['090']['train']['model']
```

Task 2

Find similar/dissimilar word pairs: Randomly generate 100, 1000, and 10000-word pairs from the vocabularies. For each set, print 5 closest word pairs and 5 furthest word pairs

定义随机获取词对，通过 cos 计算相似度的函数

In [13]:

```
def generate_pairs(dataset,n=100):
    """
    随机获取词对
    """
    # 随机获取词对
    c=np.random.choice(dataset.indexes,size=(n,2),p=dataset.frequency)
    # 去除重复词对（两个词相同的词对）
    duplicate=c[:,0]==c[:,1]
    s=duplicate.sum()
    while s:
        a=np.random.choice(dataset.indexes,size=(s,2),p=dataset.frequency)
        c[duplicate]=a
        duplicate=c[:,0]==c[:,1]
        s=duplicate.sum()
    return c

def compute_similarity(pairs,model):
    """
    根据词对计算 cos 相似度
    """
    data=model.embed_v[pairs]
    x=data[:,0]
    y=data[:,1]
    cos=np.abs((x*y).sum(axis=1))/(np.linalg.norm(x,axis=1)*np.linalg.norm(y,axis=1))
    return cos
```

获取词对，并计算相似度进行排序，获得最相似、最不相似的 5 个词

In [14]:

```
ns=[100,1000,10000]
results2=defaultdict(dict)
for n in ns:
    # 随机获得词对
    pairs=generate_pairs(train_set,n=100)
    # 计算相似度
    similarity=compute_similarity(pairs,model)
    # 排序获得最相似、最不相似的 5 个词
    furthest5=pairs[similarity.argsort()][:5]
    closest5=pairs[similarity.argsort()][-5:]
    # 保存
    results2[n]['furthest5']=train_set.index2word[furthest5]
    results2[n]['closest5']=train_set.index2word[closest5]
```

展示最相似、最不相似的 5 个词

In [15]:

```
ns=[100,1000,10000]
for n in ns:
    print('\nRandomly select %d pairs'%n)
    print('\tThe furthest5:',end='\t')
    for pair in results2[n]['furthest5']:
        print('%s,%s'%(pair[0],pair[1]),end='\t')
    print('\n\tThe closest5:',end='\t')
    for pair in results2[n]['closest5']:
        print('%s,%s'%(pair[0],pair[1]),end='\t')
```

```
Randomly select 100 pairs
        The furthest5:  gon,removing    integration,masses      united,doors    esca
pes,their       perfect,sen
        The closest5:   need,was        bäckmann,given  warren,rescue   announced,2
following,uncle
Randomly select 1000 pairs
        The furthest5:  foreword,agenda favourite,king  during,brigade  attacks,expa
nded    deposited,winnipeg
        The closest5:   mills,regular   related,allowed texas,confederate       fire
d,new   with,parent
Randomly select 10000 pairs
        The furthest5:  bursting,13     december,developers     dramatic,raban  goy
a,pliny commented,to
        The closest5:   notable,receive ceding,produce  writing,socialist       plac
ed,saying       patrick,anti
```

Task 3

Present a document as an embedding

定义获得 document embeding 的函数

In [70]:

```
# 对文章所有词的词向量取平均获得文章向量
def doc_embed_all_words(text,model,train_set):
    words_idx=np.array([train_set.word2index.get(word,train_set.word2index['<unknown>']) for word i
    return model.embed_v[words_idx].mean(axis=0)

# 对文章前n个词的词向量取平均获得文章向量
def doc_embed_first_n(text,model,train_set,n=100):
    words_idx=np.array([train_set.word2index.get(word,train_set.word2index['<unknown>']) for word i
    return model.embed_v[words_idx].mean(axis=0)
```

获取文章向量

In [71]:

```
def get_all_doc_embed(texts,model,train_set,method=doc_embed_all_words,**kwargs):
    """
    根据之前定义的方法获得文档向量
    """
    embeds=[]
    for text in texts:
        embeds.append(method(text,model,train_set,**kwargs))
    embeds=np.array(embeds)
    return embeds
```

In [59]:

```
# 用文档所有词向量求平均获得文章向量
doc_embeds_1=get_all_doc_embed(words,model,train_set,method=doc_embed_all_words)
```

In [66]:

```
# 用文章前 100 个词向量求平均得到结果
doc_embeds_2=get_all_doc_embed(words,model,train_set,method=doc_embed_first_n,n=1000)
```

使用 Doc2Vec 库函数

In [72]:

```
from gensim.models.doc2vec import TaggedDocument
from gensim.models import Doc2Vec
data=[TaggedDocument(sent,[i]) for i,sent in enumerate(words)]
```

In [ ]:

```
doc_model=Doc2Vec(vector_size=40,min_count=1,epochs=30)
doc_model.build_vocab(data)
doc_model.train(data,total_examples=doc_model.corpus_count,epochs=doc_model.epochs)
X_doc2vec=np.array([doc_model.infer_vector(words[i]) for i in range(10000)])
```

k 均值聚类

In [18]:

```
def k_means(doc_embeds,k=10):
    """
    对所有 document 进行 k-means 聚类
    """
    # 初始化 k 个类中心为所有文档中的任意 k 个
    np.random.seed(2000)
    idx=np.random.randint(doc_embeds.shape[0],size=k)
    #idx=np.linspace(500,9500,10).astype(int)
    centers=doc_embeds[idx]
    centers_pre=np.zeros_like(centers)
    embeds=doc_embeds[:,None,:].repeat(k,axis=1)
    df=pd.DataFrame(doc_embeds)
    i=0
    while (centers!=centers_pre).sum()>0:
        i+=1
        centers_pre=np.copy(centers)
        # 对每个文档分类到其距离最近的类中
        classify=np.linalg.norm(embeds-centers,axis=-1).argsort(axis=-1)[:,0]
        # 重新计算类中心
        group_mean=df.groupby(classify).mean()
        index=group_mean.index.values
        values=group_mean.values
        centers[index]=values
        if i%1==0:
            print('Iter:%d, loss:'%i,np.square(centers-centers_pre).sum())
            print('Number of each class:',np.bincount(classify))
    print('\nIt takes %d iterations.'%i)
    print('Number of each class:',np.bincount(classify))
    return classify
```

In [60]:

```
classify_1=k_means(doc_embeds_1)
```

```
Iter:1, loss: 0.3498395699465528
Number of each class: [5289    6   12   22 2467    2   35    3 2155    9]
Iter:2, loss: 0.16045727887223674
Number of each class: [4295    6   17   71 3533    1   76    3 1982   16]
Iter:3, loss: 0.011810366685253848
Number of each class: [3875    5   25   82 3744    1   75    3 2161   29]
Iter:4, loss: 0.006489677283829474
Number of each class: [3539    5   26   93 3805    1   74    3 2377   77]
Iter:5, loss: 0.0033172994774527138
Number of each class: [3282    5   24  120 3787    1   73    3 2532  173]
Iter:6, loss: 0.0013838136684537286
Number of each class: [3086    5   22  157 3753    1   73    3 2597  303]
Iter:7, loss: 0.0007706459600869113
Number of each class: [2962    5   19  189 3720    1   73    3 2617  411]
Iter:8, loss: 0.00019228082452139293
Number of each class: [2873    5   19  215 3699    1   73    3 2606  506]
Iter:9, loss: 9.311336742646886e-05
Number of each class: [2809    5   19  230 3686    1   73    3 2596  578]
Iter:10, loss: 5.619336213308274 6e-05
Number of each class: [2771    5   19  242 3680    1   73    3 2564  642]
Iter:11, loss: 5.63974339526842e-05
Number of each class: [2745    5   19  262 3666    1   73    3 2534  692]
Iter:12, loss: 2.402662186278652e-05
Number of each class: [2726    5   19  268 3664    1   73    3 2498  743]
Iter:13, loss: 1.977887882423848e-05
Number of each class: [2717    5   19  272 3646    1   73    3 2466  798]
Iter:14, loss: 1.3441017218421053e-05
Number of each class: [2709    5   19  273 3630    1   73    3 2426  861]
Iter:15, loss: 1.0336724195692252e-05
Number of each class: [2699    5   19  274 3611    1   73    3 2396  919]
Iter:16, loss: 7.0788263450118916e-06
Number of each class: [2695    5   19  273 3594    1   73    3 2372  965]
Iter:17, loss: 6.36777882634842e-06
Number of each class: [2692    5   19  273 3575    1   73    3 2341 1018]
Iter:18, loss: 6.964465540128823e-06
Number of each class: [2691    5   19  273 3558    1   73    3 2301 1076]
Iter:19, loss: 7.108834216583701e-05
Number of each class: [2685    5   19  272 3545    1   72    3 2259 1139]
Iter:20, loss: 8.682509552446782e-06
Number of each class: [2680    5   19  272 3534    1   72    3 2204 1210]
Iter:21, loss: 1.1870652756443936e-05
Number of each class: [2672    5   19  273 3515    1   72    3 2145 1295]
Iter:22, loss: 1.2393159360871482e-05
Number of each class: [2657    5   19  273 3492    1   72    3 2081 1397]
Iter:23, loss: 1.0771800296668545e-05
Number of each class: [2649    5   19  273 3458    1   72    3 2018 1502]
Iter:24, loss: 8.587412596474948e-06
Number of each class: [2654    5   19  273 3430    1   72    3 1960 1583]
Iter:25, loss: 9.88525785479242e-06
Number of each class: [2651    5   19  273 3401    1   72    3 1912 1663]
Iter:26, loss: 5.813744242633067e-06
Number of each class: [2651    5   19  274 3384    1   72    3 1866 1725]
Iter:27, loss: 5.546868293646828e-06
Number of each class: [2654    5   19  275 3363    1   72    3 1827 1781]
Iter:28, loss: 3.6921311041451674e-06
```

```
Number of each class: [2657    5   19  276 3348    1   72    3 1789 1830]
Iter:29, loss: 4.479387505053959e-06
Number of each class: [2656    5   19  277 3329    1   72    3 1759 1879]
Iter:30, loss: 3.1139024114368783e-06
Number of each class: [2658    5   19  277 3317    1   72    3 1725 1923]
Iter:31, loss: 4.345003862182406e-06
Number of each class: [2663    5   19  278 3303    1   72    3 1698 1958]
Iter:32, loss: 8.398365829782175e-07
Number of each class: [2665    5   19  278 3291    1   72    3 1685 1981]
Iter:33, loss: 7.336089961864778e-07
Number of each class: [2664    5   19  278 3280    1   72    3 1674 2004]
Iter:34, loss: 4.5445961709948836e-07
Number of each class: [2668    5   19  278 3275    1   72    3 1663 2016]
Iter:35, loss: 4.230289241027597e-07
Number of each class: [2671    5   19  278 3271    1   72    3 1655 2025]
Iter:36, loss: 4.455196611701214e-07
Number of each class: [2674    5   19  277 3270    1   72    3 1650 2029]
Iter:37, loss: 1.5075224475948046e-07
Number of each class: [2674    5   19  277 3267    1   72    3 1646 2036]
Iter:38, loss: 4.3456203197562875e-08
Number of each class: [2674    5   19  277 3265    1   72    3 1646 2038]
Iter:39, loss: 6.013506632073895e-08
Number of each class: [2674    5   19  277 3262    1   72    3 1646 2041]
Iter:40, loss: 5.318946967596442e-08
Number of each class: [2674    5   19  277 3261    1   72    3 1646 2042]
Iter:41, loss: 0.0
Number of each class: [2674    5   19  277 3261    1   72    3 1646 2042]

It takes 41 iterations.
Number of each class: [2674    5   19  277 3261    1   72    3 1646 2042]
```

In [67]:

```
classify_2=k_means(doc_embeds_2)
```

```
Iter:1, loss: 0.4939831036889345
Number of each class: [    2    2    5  381 1235    2  328    3 7714  328]
Iter:2, loss: 0.17366495003118695
Number of each class: [    3    2    9 1149 2103    1  799    3 4905 1026]
Iter:3, loss: 0.010332248563642264
Number of each class: [    3    2   19 1560 2691    1  960    3 3165 1596]
Iter:4, loss: 0.0025682654714530738
Number of each class: [    3    2   27 1695 2847    1  956    3 2695 1771]
Iter:5, loss: 0.0011962701229559881
Number of each class: [    3    2   34 1798 2879    1  902    3 2574 1804]
Iter:6, loss: 0.0008391821441718444
Number of each class: [    3    2   38 1906 2889    1  846    3 2542 1770]
Iter:7, loss: 0.0010610205065207249
Number of each class: [    3    2   47 2030 2877    1  751    3 2550 1736]
Iter:8, loss: 0.0017132236605495444
Number of each class: [    3    2   70 2139 2869    1  646    3 2562 1705]
Iter:9, loss: 0.0027204337972718613
Number of each class: [    3    2  124 2246 2817    1  565    3 2565 1674]
Iter:10, loss: 0.0009871507013576411
Number of each class: [    3    2  185 2345 2771    1  501    3 2543 1646]
Iter:11, loss: 0.0008890818523824271
Number of each class: [    3    2  258 2410 2698    1  458    3 2542 1625]
Iter:12, loss: 0.0005656311447205981
Number of each class: [    3    2  357 2470 2598    1  420    3 2533 1613]
Iter:13, loss: 0.00037936081887889235
Number of each class: [    3    2  441 2511 2500    1  372    3 2553 1614]
Iter:14, loss: 0.00029875734238803413
Number of each class: [    3    2  521 2567 2427    1  328    3 2565 1583]
Iter:15, loss: 0.00036942381246555546
Number of each class: [    3    2  588 2636 2360    1  283    3 2578 1546]
Iter:16, loss: 0.00022015441483262053
Number of each class: [    3    2  652 2700 2308    1  254    3 2574 1503]
Iter:17, loss: 0.00011513869903747775
Number of each class: [    3    2  701 2769 2277    1  239    3 2578 1427]
Iter:18, loss: 0.00012084790657778982
Number of each class: [    3    2  739 2810 2253    1  223    3 2580 1386]
Iter:19, loss: 9.42937043972157e-05
Number of each class: [    3    2  769 2835 2249    1  211    3 2577 1350]
Iter:20, loss: 8.635309355389844e-05
Number of each class: [    3    2  794 2864 2241    1  200    3 2567 1325]
Iter:21, loss: 1.8365072760644953e-05
Number of each class: [    3    2  819 2880 2233    1  197    3 2560 1302]
Iter:22, loss: 2.1152501461123612e-05
Number of each class: [    3    2  835 2895 2233    1  194    3 2556 1278]
Iter:23, loss: 1.612868517313506e-05
Number of each class: [    3    2  852 2916 2232    1  192    3 2549 1250]
Iter:24, loss: 1.4662128200675175e-05
Number of each class: [    3    2  860 2932 2233    1  190    3 2546 1230]
Iter:25, loss: 4.935983275925065e-06
Number of each class: [    3    2  871 2939 2227    1  190    3 2550 1214]
Iter:26, loss: 4.505335053608208e-06
Number of each class: [    3    2  881 2947 2222    1  190    3 2552 1199]
Iter:27, loss: 4.9958559390557376e-06
Number of each class: [    3    2  890 2959 2220    1  190    3 2555 1177]
Iter:28, loss: 3.6586674659530783e-06
```

```
Number of each class: [   3    2  900 2966 2221    1  190    3 2550 1164]
Iter:29, loss: 3.208060835319429e-06
Number of each class: [   3    2  908 2977 2217    1  190    3 2544 1155]
Iter:30, loss: 1.1726722194127589e-05
Number of each class: [   3    2  917 2987 2212    1  189    3 2542 1144]
Iter:31, loss: 1.4906407006888673e-06
Number of each class: [   3    2  920 2992 2208    1  189    3 2545 1137]
Iter:32, loss: 8.821716577888327e-07
Number of each class: [   3    2  922 3001 2208    1  189    3 2540 1131]
Iter:33, loss: 1.001297045086553e-06
Number of each class: [   3    2  926 3003 2208    1  189    3 2540 1125]
Iter:34, loss: 9.790091925966224e-07
Number of each class: [   3    2  928 3006 2207    1  189    3 2541 1120]
Iter:35, loss: 5.012437406815541e-07
Number of each class: [   3    2  930 3009 2207    1  189    3 2540 1116]
Iter:36, loss: 4.608229073711711e-07
Number of each class: [   3    2  931 3011 2206    1  189    3 2541 1113]
Iter:37, loss: 5.03121737943701e-07
Number of each class: [   3    2  932 3014 2205    1  189    3 2542 1109]
Iter:38, loss: 5.017769188584416e-07
Number of each class: [   3    2  933 3017 2205    1  189    3 2541 1106]
Iter:39, loss: 7.593071953445887e-08
Number of each class: [   3    2  934 3018 2204    1  189    3 2541 1105]
Iter:40, loss: 1.5448761152851067e-07
Number of each class: [   3    2  934 3020 2204    1  189    3 2541 1103]
Iter:41, loss: 4.146929992157005e-07
Number of each class: [   3    2  934 3024 2203    1  189    3 2541 1100]
Iter:42, loss: 3.204419079166546e-07
Number of each class: [   3    2  934 3024 2205    1  189    3 2541 1098]
Iter:43, loss: 5.584126213140717e-07
Number of each class: [   3    2  935 3022 2205    1  189    3 2544 1096]
Iter:44, loss: 4.143591898689499e-07
Number of each class: [   3    2  937 3024 2206    1  189    3 2541 1094]
Iter:45, loss: 2.8248028354380203e-07
Number of each class: [   3    2  939 3023 2205    1  189    3 2542 1093]
Iter:46, loss: 3.8169240509512123e-07
Number of each class: [   3    2  942 3023 2206    1  189    3 2540 1091]
Iter:47, loss: 3.313607560840978e-07
Number of each class: [   3    2  944 3025 2206    1  189    3 2537 1090]
Iter:48, loss: 2.4096284260629787e-07
Number of each class: [   3    2  946 3025 2206    1  189    3 2535 1090]
Iter:49, loss: 2.410872310883924e-07
Number of each class: [   3    2  947 3026 2205    1  189    3 2535 1089]
Iter:50, loss: 2.8370534488853789e-07
Number of each class: [   3    2  947 3026 2205    1  189    3 2535 1089]
Iter:51, loss: 2.2762178613687122e-08
Number of each class: [   3    2  947 3025 2205    1  189    3 2536 1089]
Iter:52, loss: 4.295149969255425e-08
Number of each class: [   3    2  947 3026 2205    1  189    3 2535 1089]
Iter:53, loss: 7.309344483180502e-08
Number of each class: [   3    2  948 3026 2205    1  189    3 2534 1089]
Iter:54, loss: 0.0
Number of each class: [   3    2  948 3026 2205    1  189    3 2534 1089]

It takes 54 iterations.
Number of each class: [   3    2  948 3026 2205    1  189    3 2534 1089]
```

In [88]:

```
classify_3=k_means(X_doc2vec)
```

```
Iter:1, loss: 1001.4569
Number of each class: [1861  555  820  736 1108  229  882 1356 1711  742]
Iter:2, loss: 66.57652
Number of each class: [1784  543  773  765 1255  402 1080 1189 1314  895]
Iter:3, loss: 28.537455
Number of each class: [1784  558  920  748 1263  479 1104 1072 1195  877]
Iter:4, loss: 17.410084
Number of each class: [1723  604 1063  794 1270  508 1111 1022 1113  792]
Iter:5, loss: 17.89223
Number of each class: [1661  662 1219  843 1313  520 1121  959 1034  668]
Iter:6, loss: 21.506447
Number of each class: [1643  728 1352  874 1362  546 1127  885  945  538]
Iter:7, loss: 15.626589
Number of each class: [1630  823 1443  877 1418  563 1152  806  819  469]
Iter:8, loss: 8.184514
Number of each class: [1621  884 1504  859 1449  614 1180  729  710  450]
Iter:9, loss: 6.3854303
Number of each class: [1606  905 1566  838 1444  655 1205  680  654  447]
Iter:10, loss: 5.884942
Number of each class: [1588  878 1631  807 1447  682 1233  648  639  447]
Iter:11, loss: 4.9466076
Number of each class: [1562  881 1664  776 1446  703 1265  623  634  446]
Iter:12, loss: 3.1098433
Number of each class: [1554  868 1673  769 1433  717 1286  608  647  445]
Iter:13, loss: 2.8488734
Number of each class: [1554  863 1673  767 1416  720 1286  599  677  445]
Iter:14, loss: 2.1852548
Number of each class: [1545  882 1677  772 1392  720 1269  593  703  447]
Iter:15, loss: 1.4230978
Number of each class: [1539  903 1671  786 1382  712 1266  591  704  446]
Iter:16, loss: 0.71358573
Number of each class: [1532  911 1676  795 1370  707 1265  591  706  447]
Iter:17, loss: 0.33653718
Number of each class: [1525  930 1678  802 1368  699 1259  590  702  447]
Iter:18, loss: 0.32768932
Number of each class: [1527  933 1674  805 1370  699 1248  589  707  448]
Iter:19, loss: 0.17312151
Number of each class: [1527  934 1671  809 1372  699 1242  588  710  448]
Iter:20, loss: 0.07972757
Number of each class: [1526  923 1670  810 1375  701 1243  586  719  447]
Iter:21, loss: 0.101482585
Number of each class: [1525  915 1666  809 1374  705 1245  583  731  447]
Iter:22, loss: 0.0863325
Number of each class: [1524  907 1664  809 1373  708 1244  581  743  447]
Iter:23, loss: 0.11065065
Number of each class: [1524  901 1659  807 1373  709 1241  580  759  447]
Iter:24, loss: 0.052730452
Number of each class: [1523  896 1659  807 1374  707 1236  580  771  447]
Iter:25, loss: 0.03906338
Number of each class: [1526  891 1655  808 1371  707 1238  580  777  447]
Iter:26, loss: 0.036833197
Number of each class: [1528  890 1653  807 1369  704 1239  580  783  447]
Iter:27, loss: 0.064244635
Number of each class: [1532  883 1652  806 1369  699 1241  580  791  447]
Iter:28, loss: 0.075732365
```

```
Number of each class: [1532  881 1653  808 1368  696 1243  581  791  447]
Iter:29, loss: 0.032838404
Number of each class: [1531  883 1651  809 1367  694 1241  581  796  447]
Iter:30, loss: 0.040433146
Number of each class: [1532  879 1647  809 1367  693 1243  581  802  447]
Iter:31, loss: 0.012260179
Number of each class: [1532  878 1649  808 1366  692 1243  581  804  447]
Iter:32, loss: 0.009210782
Number of each class: [1531  874 1652  807 1365  693 1242  581  808  447]
Iter:33, loss: 0.022318644
Number of each class: [1532  869 1653  807 1366  694 1241  581  810  447]
Iter:34, loss: 0.010759069
Number of each class: [1531  865 1652  807 1368  695 1242  581  812  447]
Iter:35, loss: 0.017834313
Number of each class: [1532  862 1652  805 1368  696 1243  581  814  447]
Iter:36, loss: 0.02819138
Number of each class: [1531  855 1652  806 1367  697 1243  581  821  447]
Iter:37, loss: 0.02732075
Number of each class: [1531  851 1650  807 1368  696 1242  581  827  447]
Iter:38, loss: 0.01099537
Number of each class: [1531  848 1649  807 1368  696 1242  581  831  447]
Iter:39, loss: 0.020491006
Number of each class: [1531  844 1649  807 1368  696 1245  581  833  446]
Iter:40, loss: 0.023278609
Number of each class: [1530  839 1649  807 1368  695 1246  581  839  446]
Iter:41, loss: 0.024833445
Number of each class: [1529  839 1650  805 1366  695 1246  581  843  446]
Iter:42, loss: 0.011854062
Number of each class: [1529  842 1646  805 1365  695 1247  581  844  446]
Iter:43, loss: 0.015463324
Number of each class: [1528  849 1643  803 1364  695 1248  581  843  446]
Iter:44, loss: 0.0017917256
Number of each class: [1528  851 1642  803 1364  694 1248  581  843  446]
Iter:45, loss: 0.0017002099
Number of each class: [1528  851 1642  803 1364  693 1248  581  844  446]
Iter:46, loss: 0.0020615468
Number of each class: [1528  851 1642  803 1364  692 1248  581  845  446]
Iter:47, loss: 0.0023697712
Number of each class: [1528  853 1642  802 1364  691 1248  581  845  446]
Iter:48, loss: 0.0
Number of each class: [1528  853 1642  802 1364  691 1248  581  845  446]

It takes 48 iterations.
Number of each class: [1528  853 1642  802 1364  691 1248  581  845  446]
```

定义评估方法

In [19]:

```
from IPython.display import display

def get_confusion(classify,label):
    """
    评估聚类效果，获得 Confusion Matrix
    """
    cu=classify[:,None]==classify[None,:]
    lei=label[:,None]==label[None,:]
    # 减去和自己的比较
    tp=(cu&lei).sum()-len(classify)
    fn=((~cu)&lei).sum()
    fp=(cu&(~lei)).sum()
    tn=((~cu)&(~lei)).sum()
    return pd.DataFrame([[tp,fn],[fp,tn]],index=['同类','非同类'],columns=['同簇','非同簇'])

def micro_f1(classify,label,beta=1):
    """
    计算 micro F1-scroe
    """
    confusion=get_confusion(classify,label)
    precision=confusion.iloc[0,0]/confusion.iloc[:,0].sum()
    recall=confusion.iloc[0,0]/confusion.iloc[0,:].sum()
    f1=(1+beta**2)*(precision*recall)/(beta**2*precision+recall)
    display(confusion)
    print('Precision:%.2f%% ,Recall:%.2f%% ,Micro F1:%.2f%%'%(precision*100,recall*100,f1*100))
```

In [20]:

```
label=pd.DataFrame(wiki_data,columns=['label'])['label'].values
```

比较结果

1.使用文档的全部单词向量的平均值作为文档向量

In [92]:

```
confusion=micro_f1(classify_1,label)
```

	同簇	非同簇
同类	16587736	9390112
非同类	8148050	65864102

```
Precision:67.06% ,Recall:63.85% ,Micro F1:65.42%
```

2.使用文档的前100个单词向量的平均值作为文档向量

In [91]:

```
confusion=micro_f1(classify_2,label)
```

	同簇	非同簇
同类	12145120	13832728
非同类	10405106	63607046

```
Precision:53.86% ,Recall:46.75% ,Micro F1:50.05%
```

3.使用Doc2Vec模型计算文档向量（结果大约为 F1:40%）

In [ ]:

```
confusion=micro_f1(classify_3,label)
```

可见用所有单词向量的平均作为文档向量有着最好的效果

用文档中最常出现的n个单词（去除出现频率大于100的常用词），按照其出现频率加权求和得到文档向量

In [81]:

```
def get_doc_embed(text,model,train_set):
    vocab=FreqDist(text)
    word_list=vocab.most_common(1000)
    f=np.zeros(200)
    m=0
    for word,k in word_list:
        idx=train_set.word2index.get(word,train_set.word2index['<unknown>'])
        f+=model.embed_v[idx]*k
        m+=k
    f/=m
    return f
```

In [82]:

```
doc_embeds_4=get_all_doc_embed(words,model,train_set,method=get_doc_embed)
```

In [83]:

```
classify_4=k_means(doc_embeds_4)
```

```
Iter:1, loss: 0.199961184359514
Number of each class: [ 121  968 2656    5    2    1 1979    1 4266    1]
Iter:2, loss: 0.0658144214839575
Number of each class: [ 398 2350 2241   12    3    1 2035    1 2958    1]
Iter:3, loss: 0.012840927859489941
Number of each class: [ 619 2841 1984   14    4    1 2078    1 2457    1]
Iter:4, loss: 0.00048484451711700396
Number of each class: [ 806 3009 1815   15    4    1 2150    1 2198    1]
Iter:5, loss: 0.00015685055819303877
Number of each class: [1041 2977 1651   15    4    1 2262    1 2047    1]
Iter:6, loss: 0.00011352654767406693
Number of each class: [1308 2875 1472   15    4    1 2357    1 1966    1]
Iter:7, loss: 9.0873501334931e-05
Number of each class: [1566 2766 1307   15    4    1 2433    1 1906    1]
Iter:8, loss: 5.611973273702895e-05
Number of each class: [1772 2677 1202   15    4    1 2480    1 1847    1]
Iter:9, loss: 5.271255198919269e-05
Number of each class: [1941 2613 1089   15    4    1 2527    1 1808    1]
Iter:10, loss: 4.9236237804297956e-05
Number of each class: [2094 2569  982   15    4    1 2555    1 1778    1]
Iter:11, loss: 4.595641872007373e-05
Number of each class: [2205 2543  881   15    4    1 2590    1 1759    1]
Iter:12, loss: 5.0103759338445134e-05
Number of each class: [2296 2524  800   15    4    1 2625    1 1733    1]
Iter:13, loss: 7.310945270107658e-05
Number of each class: [2363 2523  710   15    4    1 2657    1 1725    1]
Iter:14, loss: 5.4755895809431395e-05
Number of each class: [2412 2529  650   15    4    1 2677    1 1710    1]
Iter:15, loss: 6.347824394550093e-05
Number of each class: [2447 2543  594   15    4    1 2695    1 1699    1]
Iter:16, loss: 3.59859155452393e-05
Number of each class: [2463 2560  563   15    4    1 2707    1 1685    1]
Iter:17, loss: 2.2768246315088913e-05
Number of each class: [2468 2580  536   15    4    1 2717    1 1677    1]
Iter:18, loss: 1.3131107559538545e-05
Number of each class: [2457 2605  523   15    4    1 2721    1 1672    1]
Iter:19, loss: 1.7845249609338196e-05
Number of each class: [2454 2628  512   15    4    1 2724    1 1660    1]
Iter:20, loss: 1.1028985522337571e-05
Number of each class: [2452 2645  501   15    4    1 2729    1 1651    1]
Iter:21, loss: 1.4132518696444194e-05
Number of each class: [2464 2654  490   15    4    1 2737    1 1633    1]
Iter:22, loss: 3.052373460031008e-06
Number of each class: [2469 2670  489   15    4    1 2741    1 1609    1]
Iter:23, loss: 4.227598733891307e-06
Number of each class: [2467 2688  487   15    4    1 2743    1 1593    1]
Iter:24, loss: 1.6387396554113427e-06
Number of each class: [2470 2702  487   15    4    1 2738    1 1581    1]
Iter:25, loss: 1.670220578125822e-06
Number of each class: [2473 2713  489   15    4    1 2735    1 1568    1]
Iter:26, loss: 3.89327367326177e-06
Number of each class: [2477 2725  494   15    4    1 2731    1 1551    1]
Iter:27, loss: 6.319025654059487e-06
Number of each class: [2478 2743  497   15    4    1 2725    1 1535    1]
Iter:28, loss: 7.770519038756573e-06
```

```
Number of each class: [2476 2757  505   15    4    1 2722    1 1518    1]
Iter:29, loss: 1.959424810221257e-05
Number of each class: [2475 2770  521   15    4    1 2720    1 1492    1]
Iter:30, loss: 1.4883111647619336e-05
Number of each class: [2469 2783  536   15    4    1 2720    1 1470    1]
Iter:31, loss: 7.701183034220868e-06
Number of each class: [2472 2795  547   15    4    1 2723    1 1441    1]
Iter:32, loss: 1.4480651365634573e-05
Number of each class: [2474 2804  563   15    4    1 2726    1 1411    1]
Iter:33, loss: 1.4250709789021626e-05
Number of each class: [2469 2817  581   15    4    1 2730    1 1381    1]
Iter:34, loss: 1.2484164859095248e-05
Number of each class: [2473 2820  598   15    4    1 2736    1 1351    1]
Iter:35, loss: 5.265541645323446e-06
Number of each class: [2468 2831  607   15    4    1 2744    1 1328    1]
Iter:36, loss: 6.414084416453196e-06
Number of each class: [2472 2841  614   15    4    1 2748    1 1303    1]
Iter:37, loss: 5.005748787358601e-06
Number of each class: [2478 2851  621   15    4    1 2748    1 1280    1]
Iter:38, loss: 1.9962767839624874e-06
Number of each class: [2469 2872  623   15    4    1 2746    1 1268    1]
Iter:39, loss: 2.7734366548216573e-06
Number of each class: [2472 2887  628   15    4    1 2744    1 1247    1]
Iter:40, loss: 1.9908711506435655e-06
Number of each class: [2481 2896  629   15    4    1 2743    1 1229    1]
Iter:41, loss: 2.267534420056685e-06
Number of each class: [2482 2909  630   15    4    1 2744    1 1213    1]
Iter:42, loss: 2.243174813027351e-06
Number of each class: [2482 2926  630   15    4    1 2744    1 1196    1]
Iter:43, loss: 1.8249300190012928e-06
Number of each class: [2479 2944  632   15    4    1 2741    1 1182    1]
Iter:44, loss: 1.9067132805419677e-06
Number of each class: [2479 2960  633   15    4    1 2744    1 1162    1]
Iter:45, loss: 1.286593081615536e-06
Number of each class: [2483 2971  633   15    4    1 2739    1 1152    1]
Iter:46, loss: 1.7065043831529758e-06
Number of each class: [2490 2978  634   15    4    1 2739    1 1137    1]
Iter:47, loss: 1.4747126356278655e-06
Number of each class: [2493 2989  633   15    4    1 2738    1 1125    1]
Iter:48, loss: 2.1902137917375403e-06
Number of each class: [2494 3001  633   15    4    1 2743    1 1107    1]
Iter:49, loss: 1.7457486626816174e-06
Number of each class: [2493 3012  633   15    4    1 2746    1 1094    1]
Iter:50, loss: 2.695163898021186e-06
Number of each class: [2486 3026  636   15    4    1 2753    1 1077    1]
Iter:51, loss: 2.5469672347195086e-06
Number of each class: [2480 3038  639   15    4    1 2757    1 1064    1]
Iter:52, loss: 2.225406108687684e-06
Number of each class: [2477 3049  640   15    4    1 2760    1 1052    1]
Iter:53, loss: 3.741461463258523e-06
Number of each class: [2479 3057  643   15    4    1 2757    1 1042    1]
Iter:54, loss: 2.7096654632658233e-06
Number of each class: [2485 3067  644   15    4    1 2755    1 1027    1]
Iter:55, loss: 2.5013325861014423e-06
Number of each class: [2492 3074  646   15    4    1 2756    1 1010    1]
Iter:56, loss: 3.040069724480457e-06
Number of each class: [2505 3080  648   15    4    1 2759    1  986    1]
Iter:57, loss: 4.565552105094942e-06
Number of each class: [2526 3082  650   15    4    1 2760    1  960    1]
Iter:58, loss: 3.0668632252858433e-06
Number of each class: [2539 3084  651   15    4    1 2761    1  943    1]
```

```
Iter:59, loss: 3.34830795835699e-06
Number of each class: [2560 3084  652   15    4    1 2761    1  921    1]
Iter:60, loss: 1.913751155679624e-06
Number of each class: [2570 3087  652   15    4    1 2761    1  908    1]
Iter:61, loss: 2.08414963393351e-06
Number of each class: [2582 3091  652   15    4    1 2759    1  894    1]
Iter:62, loss: 3.871983678882326e-06
Number of each class: [2595 3095  654   15    4    1 2760    1  874    1]
Iter:63, loss: 5.532322190447935e-06
Number of each class: [2615 3097  655   15    4    1 2761    1  850    1]
Iter:64, loss: 6.880269036492535e-06
Number of each class: [2638 3101  655   15    4    1 2763    1  821    1]
Iter:65, loss: 3.944169795049064e-06
Number of each class: [2646 3112  655   15    4    1 2762    1  803    1]
Iter:66, loss: 5.2464695350073554e-06
Number of each class: [2670 3114  654   15    4    1 2759    1  781    1]
Iter:67, loss: 3.8061367127532427e-06
Number of each class: [2682 3123  654   15    4    1 2757    1  762    1]
Iter:68, loss: 7.33056321453006e-06
Number of each class: [2706 3135  653   15    4    1 2748    1  736    1]
Iter:69, loss: 7.2331253276816e-06
Number of each class: [2724 3145  653   15    4    1 2744    1  712    1]
Iter:70, loss: 1.1120166585077908e-05
Number of each class: [2739 3159  654   15    4    1 2739    1  687    1]
Iter:71, loss: 6.838151575882184e-06
Number of each class: [2736 3183  653   15    4    1 2741    1  665    1]
Iter:72, loss: 5.004069487088813e-06
Number of each class: [2741 3202  652   15    4    1 2734    1  649    1]
Iter:73, loss: 4.087278722399844e-06
Number of each class: [2734 3227  651   15    4    1 2731    1  635    1]
Iter:74, loss: 2.244928010657871e-06
Number of each class: [2732 3241  650   15    4    1 2727    1  628    1]


Iter:75, loss: 1.1918277529650543e-06
Number of each class: [2728 3254  650   15    4    1 2722    1  624    1]
Iter:76, loss: 1.1929087294201108e-06
Number of each class: [2728 3264  649   15    4    1 2715    1  622    1]
Iter:77, loss: 5.293535338530317e-07
Number of each class: [2730 3269  649   15    4    1 2711    1  619    1]
Iter:78, loss: 5.242889052860679e-07
Number of each class: [2726 3276  649   15    4    1 2710    1  617    1]
Iter:79, loss: 1.2933719796119964e-06
Number of each class: [2730 3279  649   15    4    1 2709    1  611    1]
Iter:80, loss: 8.30813808198893e-07
Number of each class: [2735 3279  648   15    4    1 2708    1  608    1]
Iter:81, loss: 2.41229441489033e-07
Number of each class: [2736 3281  649   15    4    1 2705    1  607    1]
Iter:82, loss: 2.2967007992967084e-07
Number of each class: [2737 3282  649   15    4    1 2704    1  606    1]
Iter:83, loss: 1.6175109867998626e-07
Number of each class: [2738 3282  649   15    4    1 2704    1  605    1]
Iter:84, loss: 0.0
Number of each class: [2738 3282  649   15    4    1 2704    1  605    1]

It takes 84 iterations.
Number of each class: [2738 3282  649   15    4    1 2704    1  605    1]
```

In [84]:

```
confusion=micro_f1(classify_4,label)
```

	同簇	非同簇
同类	17487038	8490810
非同类	8870216	65141936

```
Precision:66.35% ,Recall:67.32% ,Micro F1:66.83%
```

可以看见，此方法优于所有词向量取均值。

Task 4

Use t-SNE to project these vectors into 2-d and plot them out for each of the above choices.

计算向量距离矩阵

In [93]:

```
def cal_dist(data,n=200):
    """
    根据向量矩阵计算距离矩阵
    """
    N=data.shape[0]
    dist=np.zeros((N,N))
    # 防止爆内存，分批计算
    for i in range(0,N,n):
        dist[i:i+n,:]=np.linalg.norm(data[None,:].repeat(n,axis=0)-data[i:i+n][:,None],axis=-1)
        #print(i)
    return dist
```

计算高维向量概率/熵

In [94]:

```
def calc_p_and_entropy(dist,beta):
    """
    计算高维向量概率/熵
    """
    n=dist.shape[0]
    p = np.exp(-np.square(dist) * beta[:,None])
    # 防止数字下溢，现将对角线设为 0
    p[range(n),range(n)]=0
    p_sum = p.sum(axis=1,keepdims=True)
    # 防止取 log 时对角线上为 0
    p[range(n),range(n)]=1e-20
    p/=p_sum
    p[p<1e-20]+=1e-20
    # 计算熵
    log_entropy_matrix = -(p*np.log(p))
    #print(log_entropy_matrix)
    #log_entropy = log_entropy_matrix.sum(axis=1)-log_entropy_matrix[range(n),range(n)]
    log_entropy = log_entropy_matrix.sum(axis=1)
    p[range(n),range(n)]=0
    return p,log_entropy
```

二分法搜索 beta 值

ref:[https://www.bilibili.com/video/BV1cU4y1w74A (https://www.bilibili.com/video/BV1cU4y1w74A)](https://www.bilibili.com/video/BV1cU4y1w74A)

In [104]:

```
def binary_search(dist,init_beta,perplexity,threshold=1e-4,max_iter=50):
    """
    二分法搜索最佳 beta 值
    """
    print("寻找最佳 beta...")
    n=dist.shape[0]
    # 初始化 beta 上下限
    beta_max = np.array([np.inf]*n,dtype=np.float32)
    beta_min = np.array([-np.inf]*n,dtype=np.float32)
    beta = np.array([init_beta]*n,dtype=np.float32)
    # 计算高维向量概率/熵
    P,log_entropy=calc_p_and_entropy(dist,beta)
    # 计算与设定困惑度的差值
    diff = log_entropy - perplexity
    i=0
    while np.abs(diff).max()> threshold and i<max_iter:
        # 更新上下限
        beta_min[diff>0]=beta[diff>0]
        beta_max[diff<=0]=beta[diff<=0]
        # 交叉熵比期望值大，增大beta
        beta[(diff>0)&(beta_max==np.inf)]*=2.
        beta[(diff>0)&(beta_max!=np.inf)]=(beta[(diff>0)&(beta_max!=np.inf)]+beta_max[(diff>0)&(be
        # 交叉熵比期望值小， 减少beta
        beta[(diff<=0)&(beta_min==-np.inf)]/=2.
        beta[(diff<=0)&(beta_min!=-np.inf)]=(beta[(diff<=0)&(beta_min!=-np.inf)]+beta_min[(diff<=(
        # 重新计算
        p,log_entropy=calc_p_and_entropy(dist,beta)
        diff = log_entropy - perplexity
        print('iter %d'%(i+1),',max difference of log-entropy:%.6f'%np.abs(diff).max())
        i+=1
    # 返回最优的 beta 以及所对应的 P
    return p, beta
```

计算高维联合概率

In [96]:

```
def p_joint(data, init_beta=1,perplexity=5):
    """
    计算高维联合概率
    """
    N=data.shape[0]
    # 计算距离
    dist=cal_dist(data,n=200)
    # 二分法获得最佳 beta
    p,beta=binary_search(dist,init_beta,perplexity)
    p_join = (p + p.T)/2
    p_join /= p_join.sum()
    p_join[range(N),range(N)]=1e-10
    print("Mean value of beta: %f" % np.mean(beta))
    return p_join
```

计算低维联合概率：t 分布

In [97]:

```
def q_tsne(dist):
    """
    计算低维联合概率：t分布
    """
    N = dist.shape[0]
    tmp=(1+np.square(dist))**-1
    tmp[range(N),range(N)]=0
    # 归一化
    q=tmp/tmp.sum()
    # 设对角线为非 0 值，方便后面计算
    q[range(N),range(N)]=1
    return q
```

定义画图函数

In [98]:

```
def draw_pic(data,labs,name = '1.jpg'):
    """
    画图
    """
    plt.cla()
    unque_labs = np.unique(labs)
    colors = [plt.cm.Spectral(each) for each in np.linspace(0, 1,len(unque_labs))]
    p=[]
    legends = []
    for i in range(len(unque_labs)):
        index = np.where(labs==unque_labs[i])
        pi = plt.scatter(data[index, 0], data[index, 1], c =[colors[i]] )
        p.append(pi)
        legends.append(unque_labs[i])

    plt.legend(p, legends)
    #plt.savefig(name)
    plt.show()
```

利用 T-SNE 将高维向量投影到 2 维

In [110]:

```
def tsen(data,dim,init_beta,target_perplexity,plot=False,p=None):
    """
    计算 tsne
    data:文档向量
    dim:低维向量维度
    init_beta:初始化beta值
    target_perplexity:目标困惑度
    """
    N,D = data.shape
    # 随机初始化低维数据
    y = np.random.randn(N, dim)
    # 计算高维向量的联合概率
    print("1.计算高维向量的联合概率")
    if p is None:
        p = p_joint(data,init_beta,target_perplexity)
    # 开始进行迭代训练
    # 训练相关参数, 用 Adam 算法迭代
    print("2.迭代计算低维向量的联合概率")
    max_iter = 30
    lr=300
    beta1=0.9
    beta2=0.999
    eps=1e-20
    m=np.zeros_like(y)
    v=np.zeros_like(y)
    for m_iter in range(max_iter):
        # 低维距离
        dist_y=cal_dist(y)
        # 低维联合概率
        q= q_tsne(dist_y)
        # 计算梯度
        y_minus=y[:,None].repeat(N,axis=1)-y[None,:].repeat(N,axis=0)
        dy=4*((p-q)[:,:,None]*y_minus*(1+dist_y**2)[:,:,None]**-1).sum(axis=1)
        # Adam 优化器
        m=(1-beta1)*m+beta1*dy
        v=(1-beta2)*v+beta2*dy**2
        m_hat=m/(1-beta1**(m_iter+1))
        v_hat=v/(1-beta2**(m_iter+1))
        y-=lr*m_hat/(np.sqrt(v_hat)+eps)
        #y-=lr*dy
        # 损失函数
        if (m_iter + 1) % 1 == 0:
            c=p * np.log(p / q)
            loss=c.sum()-c[range(N),range(N)].sum()
            print("Iteration %d: ,loss: %f" % (m_iter + 1, loss))
            if loss<1e-2:
                break

    return y
```

用 t-SNE 映射到2维

选择 perplexity 为 8

In [112]:

```
y=tsen(doc_embeds_1,2,1,8)
```

```
1.计算高维向量的联合概率
寻找最佳 beta...
iter 1 ,max difference of log-entropy:1.207288
iter 2 ,max difference of log-entropy:1.200100
iter 3 ,max difference of log-entropy:1.177830
iter 4 ,max difference of log-entropy:7.081110
iter 5 ,max difference of log-entropy:6.834616
iter 6 ,max difference of log-entropy:7.741193
iter 7 ,max difference of log-entropy:8242295340.724341
iter 8 ,max difference of log-entropy:7.485889
iter 9 ,max difference of log-entropy:1.776096
iter 10 ,max difference of log-entropy:0.632702
iter 11 ,max difference of log-entropy:0.259153
iter 12 ,max difference of log-entropy:0.142855
iter 13 ,max difference of log-entropy:0.068369
iter 14 ,max difference of log-entropy:0.033078
iter 15 ,max difference of log-entropy:0.016036
iter 16 ,max difference of log-entropy:0.007737
iter 17 ,max difference of log-entropy:0.003991
iter 18 ,max difference of log-entropy:0.001751
iter 19 ,max difference of log-entropy:0.000951
iter 20 ,max difference of log-entropy:0.000447
iter 21 ,max difference of log-entropy:0.000221
iter 22 ,max difference of log-entropy:0.000134
iter 23 ,max difference of log-entropy:0.000067
Mean value of beta: 122.242249
2.迭代计算低维向量的联合概率
Iteration 1: ,loss: 1.050600
Iteration 2: ,loss: 6.663622
Iteration 3: ,loss: 6.123033
Iteration 4: ,loss: 3.869120
Iteration 5: ,loss: 2.789792
Iteration 6: ,loss: 2.524771
Iteration 7: ,loss: 2.474798
Iteration 8: ,loss: 2.459945
Iteration 9: ,loss: 2.452864
Iteration 10: ,loss: 2.469265
Iteration 11: ,loss: 2.470422
Iteration 12: ,loss: 2.485549
Iteration 13: ,loss: 2.499139
Iteration 14: ,loss: 2.506209
Iteration 15: ,loss: 2.515930
Iteration 16: ,loss: 2.515370
Iteration 17: ,loss: 2.527019
Iteration 18: ,loss: 2.541587
Iteration 19: ,loss: 2.534746
Iteration 20: ,loss: 2.548401
Iteration 21: ,loss: 2.562620
Iteration 22: ,loss: 2.541740
Iteration 23: ,loss: 2.584747
Iteration 24: ,loss: 2.564516
Iteration 25: ,loss: 2.577333
Iteration 26: ,loss: 2.594686
Iteration 27: ,loss: 2.588397
Iteration 28: ,loss: 2.595040
```

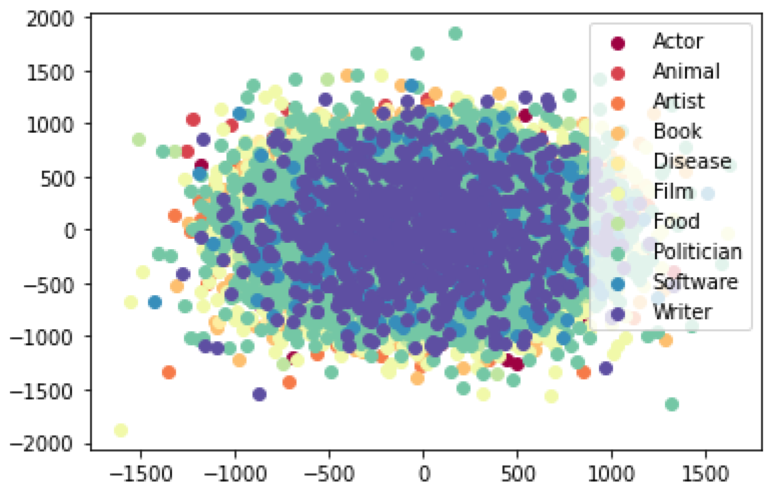
```
Iteration 29: ,loss: 2.585675
Iteration 30: ,loss: 2.605602
```

画图得到结果

效果并不好

In [113]:

```
draw_pic(y,label)
```



改用库函数

能看到同类数据点有明显地聚集

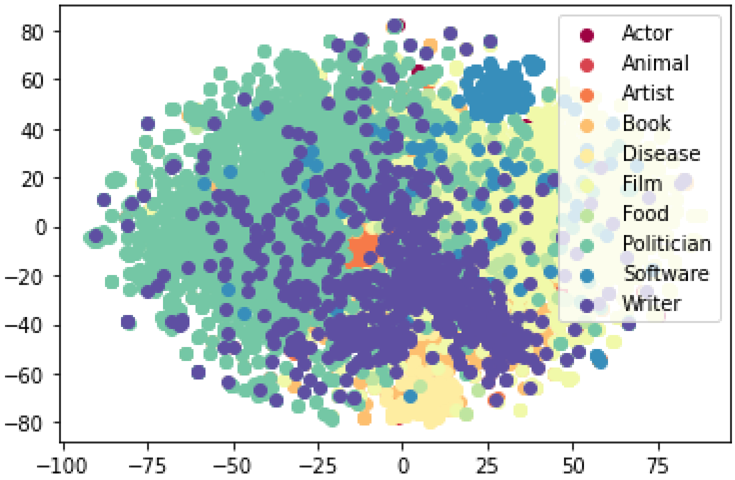
In [114]:

```
from sklearn.manifold import TSNE
tsne_ = TSNE(n_components=2,init='pca',random_state=33,perplexity=5)
X_tsne=tsne_.fit_transform(doc_embeds_1)
```

```
D:\Program Files (x86)\Anaconda\envs\nlp\lib\site-packages\sklearn\manifold\_t_sne.p
y:790: FutureWarning: The default learning rate in TSNE will change from 200.0 to 'a
uto' in 1.2.
  warnings.warn(
D:\Program Files (x86)\Anaconda\envs\nlp\lib\site-packages\sklearn\manifold\_t_sne.p
y:982: FutureWarning: The PCA initialization in TSNE will change to have the standar
d deviation of PC1 equal to 1e-4 in 1.2. This will ensure better convergence.
  warnings.warn(
```

In [115]:

```
draw_pic(X_tsne,label)
```



In [ ]: