1. config.py

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

'''

Filename : config.py

Function : All the file path for input and output, users could change them for usage.

All the parameter settings used in KR-ERE, users could change any as they need.

'''

import torch

import os

def check\_path(path, raise\_error=True):

if not os.path.exists(path):

if raise\_error:

print("error : Path %s does not exist!" % path)

exit(1)

else:

print("warning : Path %s does not exist!" % path)

print("info : Creating path %s." % path)

os.makedirs(path)

print("info : Successfully making dir!")

class Config():

def \_\_init\_\_(self):

# Input&Output data arguments

self.ori\_train\_path = './data/original/TrainingSet.txt'

self.ori\_dev\_path = './data/original/DevelopmentSet.txt'

self.ori\_test\_path = './data/original/TestSet.txt'

self.intra\_path = './data/process/intra\_ins/'

self.inter\_path = './data/process/inter\_ins/'

self.train\_ins\_path = 'TrainingSet.instance'

self.dev\_ins\_path = 'DevelopmentSet.instance'

self.test\_ins\_path = 'TestSet.instance'

self.clean\_data = 'data.pkl'

self.word\_vec\_path = 'word2vec.vec'

self.entity\_index\_path = './data/knowledge\_base/entity2id.txt'

self.relation\_index\_path = './data/knowledge\_base/relation2id.txt'

self.triple\_path = './data/knowledge\_base/train.txt'

self.entity\_vec\_path = './data/knowledge\_base/entity2vec.bern'

self.relation\_vec\_path = './data/knowledge\_base/relation2vec.bern'

self.result\_path = './result/'

self.intra = 'intra/'

self.inter = 'inter/'

self.document = 'merge/'

self.pretrain\_model\_path = 'pretrain\_model.param'

self.model\_save\_path0 = './result/model/'

self.model\_save\_path = self.model\_save\_path0 + 'KR\_ERE\_model.param'

# Hyper-parameter settings

self.word\_vec\_dim = 100

self.kg\_vec\_dim = 100

self.intra\_lr = 0.0001

self.inter\_lr = 0.0002

self.class\_number = 2

self.epoch\_number = 30

self.batch\_size = 20

self.convolution\_dim = 100

self.kernel\_size = [1,2,3,4,5]

self.is\_document = True

# Check Path

self.CheckPath()

def CheckPath(self):

# Check files

check\_path(self.ori\_train\_path)

check\_path(self.ori\_dev\_path)

check\_path(self.ori\_test\_path)

check\_path(self.ori\_train\_path)

check\_path(self.entity\_index\_path)

check\_path(self.relation\_index\_path)

check\_path(self.triple\_path)

# Check dirs

check\_path(self.intra\_path, raise\_error=False)

check\_path(self.inter\_path, raise\_error=False)

check\_path(self.result\_path + self.intra, raise\_error=False)

check\_path(self.result\_path + self.inter, raise\_error=False)

check\_path(self.result\_path + self.document, raise\_error=False)

check\_path(self.model\_save\_path0, raise\_error=False)

2. extract\_instance.py

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

'''

Filename : extracct\_instance.py

Function : From document level data into sentence level instance

Extract instances from the given data, instances could be divided into 2 kinds:

1. intra-sentence instances: entities in the same sentence, which we extract their co-existed sentence.

2. inter-sentence instances: entities in different sentences, which we extract two sentences and connect them together.

'''

import pickle

import os

import numpy as np

from collections import defaultdict,OrderedDict

import re

import copy

from config import Config

ent\_repl\_dic = {}

Induced = []

Title = OrderedDict()

def norm\_sentence(sentence):

sentence = sentence + ' '

sentence = sentence.replace("'s ",' ')

sentence = sentence.replace("'",' ')

sentence = sentence.replace('"',' ')

sentence = sentence.replace(', ',' , ')

sentence = sentence.replace(': ',' : ')

sentence = sentence.replace('! ',' ! ')

sentence = sentence.replace('? ',' ? ')

sentence = sentence.replace('. ',' . ')

sentence = sentence.replace('; ',' ; ')

sentence = sentence.replace('-',' - ')

sentence = re.sub('[\(\)\[\]\{\}]',' ',sentence)

sentence = re.sub('\b[0-9]+\b','num',sentence)

sentence = re.sub('\s+',' ',sentence)

for \_ in range (5):

sentence = sentence.replace(' ',' ')

return sentence.strip()

def generate\_instance(article,entities):

Intra\_sent = []

Inter\_sent\_raw = []

del\_overlap = {} #store the shortest length of the two entities, and only save the shortest instance.

index\_e1 = -1

for ent\_1 in entities:

index\_e1 += 1

if ent\_1[4] != 'Chemical': #optional

continue

index\_e2 = -1

for ent\_2 in entities:

index\_e2 += 1

if ent\_2[4] != 'Disease': #optional

continue

if ent\_1[1]>=ent\_2[2]:

first\_ent, last\_ent = ent\_2, ent\_1

else:

first\_ent, last\_ent = ent\_1, ent\_2

sent\_list = [i+1 for i in range(first\_ent[2],last\_ent[1]-1) if article[i:i+2] == '. ']

len\_e1\_e2 = len ( article [first\_ent[2]:last\_ent[1]].strip().split(' ') ) #optinal, to judge and save the shortest length instance

if len(sent\_list) > 2: #optional

continue

start = 0

end = len(article)

for i in range(first\_ent[1],1,-1):

if article[i-2:i]=='. ':

start = i

break

for i in range(last\_ent[2],len(article)-2):

if article[i:i+2]=='. ':

end = i + 1

break

if not len (sent\_list):

#intra sentence

sentence\_part1 = norm\_sentence(article[start:first\_ent[1]])

first\_ent\_name = article[first\_ent[1]:first\_ent[2]]

sentence\_part2 = norm\_sentence(article[first\_ent[2]:last\_ent[1]])

last\_ent\_name = article[last\_ent[1]:last\_ent[2]]

sentence\_part3 = norm\_sentence(article[last\_ent[2]:end])

first\_ent\_start = len(sentence\_part1.split(' '))

first\_ent\_end = first\_ent\_start + len(first\_ent\_name.split(' '))

last\_ent\_start = len((' '.join([sentence\_part1,first\_ent\_name,sentence\_part2])).split(' '))

last\_ent\_end = last\_ent\_start + len(last\_ent\_name.split(' '))

sentence = ' '.join([sentence\_part1,first\_ent\_name,sentence\_part2,last\_ent\_name,sentence\_part3])

if first\_ent[4] == 'Chemical':

pos\_chem\_start, pos\_chem\_end, pos\_dis\_start, pos\_dis\_end = first\_ent\_start, first\_ent\_end, last\_ent\_start, last\_ent\_end

else:

pos\_chem\_start, pos\_chem\_end, pos\_dis\_start, pos\_dis\_end = last\_ent\_start, last\_ent\_end, first\_ent\_start, first\_ent\_end

Intra\_sent.append([ent\_1[0], ent\_1[5],str(pos\_chem\_start),str(pos\_chem\_end),ent\_2[5],str(pos\_dis\_start),str(pos\_dis\_end),sentence])

del\_overlap[ent\_1[5]+'\_'+ent\_2[5]] = 0

else :

#inter sentence

sent1\_part1 = norm\_sentence(article[start:first\_ent[1]])

sent1\_part2 = norm\_sentence(article[first\_ent[2]:sent\_list[0]])

sent2\_part1 = norm\_sentence(article[sent\_list[-1]+1:last\_ent[1]])

sent2\_part2 = norm\_sentence(article[last\_ent[2]:end])

first\_ent\_name = article[first\_ent[1]:first\_ent[2]]

last\_ent\_name = article[last\_ent[1]:last\_ent[2]]

first\_ent\_start = len(sent1\_part1.split(' '))

first\_ent\_end = first\_ent\_start + len(first\_ent\_name.split(' '))

last\_ent\_start = len(' '.join([sent1\_part1,first\_ent\_name,sent1\_part2,sent2\_part1]))

last\_ent\_end = last\_ent\_start + len (last\_ent\_name.split(' '))

sentence = ' '.join([sent1\_part1,first\_ent\_name,sent1\_part2,sent2\_part1,last\_ent\_name,sent2\_part2])

if first\_ent[4] == 'Chemical':

pos\_chem\_start, pos\_chem\_end, pos\_dis\_start, pos\_dis\_end = first\_ent\_start, first\_ent\_end, last\_ent\_start, last\_ent\_end

else:

pos\_chem\_start, pos\_chem\_end, pos\_dis\_start, pos\_dis\_end = last\_ent\_start, last\_ent\_end, first\_ent\_start, first\_ent\_end

Inter\_sent\_raw.append([ent\_1[0], ent\_1[5],str(pos\_chem\_start),str(pos\_chem\_end),ent\_2[5],str(pos\_dis\_start),str(pos\_dis\_end),sentence,len\_e1\_e2])

if len\_e1\_e2 <= del\_overlap.get(ent\_1[5]+'\_'+ent\_2[5],len(article)):

del\_overlap[ent\_1[5]+'\_'+ent\_2[5]] = len\_e1\_e2

Inter\_sent = []

for instance in Inter\_sent\_raw:

if del\_overlap[instance[1]+'\_'+instance[4]] >= instance[-1]:

Inter\_sent.append(instance[:-1])

return Intra\_sent, Inter\_sent

def load\_original\_data (data\_path):

Intra\_Ins = []

Inter\_Ins = []

label = defaultdict(lambda:'UN')

sum = 0

with open (data\_path,'r') as f:

line = f.readline()

while not line == '':

item = line.strip().split('|')

Article = item[2]

title\_length = len(Article)

title = Article

line = f.readline()

item = line.strip().split('|')

Article = Article + ' ' + item[2]

line = f.readline()

entities = []

while not line == '\n':

item = line.strip().split('\t')

if item[1] == 'CID':

label[item[0]+'\_'+item[2]+'\_'+item[3]] = 'CID'

sum += 1

else:

if len(item) >= 6 and item[5][0] in ['C','D'] and item[5][1] != 'H':

#print (item)

item[1],item[2]=int(item[1]),int(item[2])

if not Article[max(item[1]-1,0)] =='(' or not Article[min(item[2],len(Article)-1)] == ')':

#statistic the inudced words.

if not item[2] == len(Article) and Article[item[2]] == '-':

induce = ''

for fig in Article[item[2]+1:]:

if not fig.isalpha():

break

induce = induce + fig

if induce not in Induced:

Induced.append(induce)

item[3] = item[3].lower()

item[5] = item[5].split('|')[0]

entities.append(item)

line = f.readline()

Title[item[0]] = title

#here we add the filter of hyper.

intra, inter = generate\_instance(Article,entities)#,ex\_word\_dic)

Intra\_Ins.extend(intra)

Inter\_Ins.extend(inter)

line = f.readline()

return Intra\_Ins,Inter\_Ins,label,sum

def out\_ins\_data (ins, label, sum, ins\_path):

'''

if not os.path.isdir(ins\_path):

os.makedirs(ins\_path)

'''

ins\_out = open(ins\_path,'w')

ins\_out.write(str(sum) + '\n')

for item in ins:

ins\_out.write (label[item[0]+'\_'+item[1]+'\_'+item[4]] + '\t' + '\_'.join(item[:7]) + '\t' + item[-1] + '\n')

ins\_out.close()

def out\_title (ID\_path,title\_path):

out\_ID = open(ID\_path,'w')

out\_tit = open(title\_path,'w')

for key in Title.keys():

out\_ID.write(key +'\n')

out\_tit.write(Title[key] + '\n')

out\_ID.close()

out\_tit.close()

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

# intra-sentence level data

config = Config()

CDR\_train\_data\_path = config.ori\_train\_path

CDR\_dev\_data\_path = config.ori\_dev\_path

CDR\_test\_data\_path = config.ori\_test\_path

CDR\_train\_intra\_ins\_path = config.intra\_path + config.train\_ins\_path

CDR\_dev\_intra\_ins\_path = config.intra\_path + config.dev\_ins\_path

CDR\_test\_intra\_ins\_path = config.intra\_path + config.test\_ins\_path

CDR\_train\_inter\_ins\_path = config.inter\_path + config.train\_ins\_path

CDR\_dev\_inter\_ins\_path = config.inter\_path + config.dev\_ins\_path

CDR\_test\_inter\_ins\_path = config.inter\_path + config.test\_ins\_path

ID\_path = 'title\_ID.txt'

title\_path = 'title\_seq.txt'

print("load train data...")

train\_intra, train\_inter, train\_label,sum = load\_original\_data (CDR\_train\_data\_path)

print("output train data...")

out\_ins\_data (ins = train\_intra,

label = train\_label,

sum = sum,

ins\_path = CDR\_train\_intra\_ins\_path)

out\_ins\_data (ins = train\_inter,

label = train\_label,

sum = sum,

ins\_path = CDR\_train\_inter\_ins\_path)

print("load develop data...")

dev\_intra, dev\_inter, dev\_label,sum = load\_original\_data (CDR\_dev\_data\_path)

print("output dev data...")

out\_ins\_data (ins = dev\_intra,

label = dev\_label,

sum = sum,

ins\_path = CDR\_dev\_intra\_ins\_path)

out\_ins\_data (ins = dev\_inter,

label = dev\_label,

sum = sum,

ins\_path = CDR\_dev\_inter\_ins\_path)

print("load test data...")

test\_intra, test\_inter, test\_label,sum = load\_original\_data (CDR\_test\_data\_path)

print("output test data...")

out\_ins\_data (ins = test\_intra,

label = test\_label,

sum = sum,

ins\_path = CDR\_test\_intra\_ins\_path)

out\_ins\_data (ins = test\_inter,

label = test\_label,

sum = sum,

ins\_path = CDR\_test\_inter\_ins\_path)

print ("output title...")

out\_title (ID\_path,title\_path)

out\_dic = open('./Induced.txt','w')

for item in Induced:

out\_dic.write(item+'\n')

out\_dic.close()

3. process\_data.py

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

'''

Filename : process\_data.py

Function : Add knowledge bases into entity pair instances

Pickle all the training/development/test instances into pkl

'''

import pickle

import os

import numpy as np

from config import Config

np.random.seed(1337)

config = Config()

wordNum = 0

word\_index = {}

entity\_index = {}

relation\_index ={}

entity\_rel = {}#defaultdict(lambda:3)

entity\_vecs = []

relation\_vecs = []

def entity2ID(entity\_path):

#load the id of entities in knowledge base

with open (entity\_path,'r') as f:

for line in f:

line = line.strip().split('\t')

entities = line[0].strip().split('|')

for entity in entities:

entity\_index[entity] = int(line[1])

def relation2ID(relation\_path):

#load the id of the relation in knowledge base

with open (relation\_path,'r') as f:

for line in f:

line = line.strip().split('\t')

relation\_index[line[0]] = int(line[1])

if 'NULL' not in relation\_index:

relation\_index['NULL'] = len(relation\_index)

def load\_kgfile(kg\_path):

#load the chem-dis relation in knowledge base, the format is "C000xxx D000xxx NONE"

with open (kg\_path,'r') as f:

for line in f:

line = line.strip().split('\t')

entities\_1 = line[0].strip().split('|')

entities\_2 = line[1].strip().split('|')

for entity\_1 in entities\_1:

for entity\_2 in entities\_2:

entity\_rel[entity\_1+entity\_2] = relation\_index[line[2]]

def load\_entity (file\_path):

with open (file\_path,'r') as f:

for line in f:

line = line.strip().split('\t')

line = np.array([float(num) for num in line])

entity\_vecs.append(line)

def load\_relation (file\_path):

with open (file\_path,'r') as f:

for line in f:

line = line.strip().split('\t')

line = np.array([float(num) for num in line])

relation\_vecs.append(line)

none\_rel = np.random.uniform(-0.25, 0.25, size= 100)

relation\_vecs.append(none\_rel)

def get\_triples (Entities):

triples = []

for entity in Entities:

if entity[1] not in entity\_index:

entity\_index[entity[1]] = len (entity\_index)

entity\_vecs.append(np.random.uniform(-0.25,0.25,size=config.word\_vec\_dim))

if entity[4] not in entity\_index.keys():

entity\_index[entity[4]] = len (entity\_index)

entity\_vecs.append(np.random.uniform(-0.25,0.25,size=config.word\_vec\_dim))

if entity[1]+entity[4] not in entity\_rel:

entity\_rel[entity[1]+entity[4]] = relation\_index['NULL']

triples.append([entity\_index[entity[1]],entity\_index[entity[4]], entity\_rel[entity[1]+entity[4]]])

return triples

def Sentence2ID(sentences):

#to make the word in sentences into ID

global wordNum

sentencesID = []

for sentence in sentences:

sentenceid = []

for word in sentence:

if word not in word\_index.keys():

word\_index[word] = wordNum

wordNum += 1

sentenceid.append(word\_index[word])

sentencesID.append(sentenceid)

return sentencesID

def load\_file(file\_path):

Label = []

Entities = []

Sentence = []

with open(file\_path,'r')as f:

gold\_num = int(f.readline())

for line in f:

line =line.strip().split('\t')

Label.append(line[0]=='CID')

entity = line[1].strip().split('\_')

#means articleID; ent1 ID; ent1's words start position;end position;ent2 ID; ent2 start position;end position

#the sentence could be the sdp or sequence

Entities.append ([entity[0],entity[1],int(entity[2]),int(entity[3]),entity[4],int(entity[5]),int(entity[6])])

Sentence.append (line[2].lower().strip().split(' '))

return Label, Entities, Sentence, gold\_num

def process\_data(train\_Label,train\_Entities,train\_Sentence,train\_sum,\

dev\_Label,dev\_Entities,dev\_Sentence,dev\_sum,\

test\_Label,test\_Entities,test\_Sentence,test\_sum,\

output\_path,data\_path,word\_vec\_path = ''):

max\_sen\_len = 0

#MAX\_SEQUENCE\_LENGTH=10

# calcuate the max length of doc, sentence and word

for data in (train\_Sentence, dev\_Sentence, test\_Sentence):

for item in data:

# print i

max\_sen\_len = max(max\_sen\_len, len(item))

print ('the max sentence length is : ', max\_sen\_len)

train\_SentenceID = Sentence2ID (train\_Sentence)

dev\_SentenceID = Sentence2ID (dev\_Sentence)

test\_SentenceID = Sentence2ID (test\_Sentence)

print('Found %s unique tokens.' % wordNum)

dict\_writer =open(output\_path+'/words.vocab','w')

word\_index\_sorted = sorted(word\_index.items(), key=lambda x: x[1])

for k, v in word\_index\_sorted:

dict\_writer.write(k + ' ' + str(v) + '\n')

dict\_writer.close()

# load word embeddings...

embeddings\_index = {}

EMBEDDING\_DIM = 100

if word\_vec\_path:

f = open(word\_vec\_path,'w')

f.readline()

for line in f:

values = line.split()

word = values[0].lower()

coefs = np.asarray(values[1:], dtype='float32')

embeddings\_index[word] = coefs

f.close()

print('Total %s word vectors in Glove .' % len(embeddings\_index))

else:

print('No pretrained word2vec, all embedding will be initialized randomly')

embedding\_matrix = np.random.uniform(-0.25, 0.25, size=(wordNum + 1, EMBEDDING\_DIM))

embedding\_matrix[0] = np.zeros(shape=EMBEDDING\_DIM, dtype='float32')

number = 0

vocab\_oup =open(output\_path+'/miss.vocab','w')

for word in word\_index.keys():

embedding\_vector = embeddings\_index.get(word)

if embedding\_vector is not None:

# words not found in embedding index will be all-zeros.

embedding\_matrix[word\_index[word]] = embedding\_vector

else:

number += 1

vocab\_oup.write(word+'\n')

print ('total {} word not find in the word embeddings'.format(number))

vocab\_oup.close()

# add the relation entities:

train\_Triples = get\_triples (train\_Entities)

dev\_Triples = get\_triples (dev\_Entities)

test\_Triples = get\_triples (test\_Entities)

entity\_matrix = np.asarray(entity\_vecs,dtype = 'float32')

relation\_matirx = np.asarray(relation\_vecs,dtype= 'float32')

print ('dump the file ...')

pickle.dump([train\_SentenceID,train\_Label,train\_Entities,train\_Triples,train\_sum,\

dev\_SentenceID,dev\_Label,dev\_Entities,dev\_Triples,dev\_sum,\

test\_SentenceID,test\_Label,test\_Entities,test\_Triples,test\_sum,\

embedding\_matrix,entity\_matrix,relation\_matirx], open(output\_path+'/data.pkl', 'wb'))

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

# intra-sentence level data

train\_data\_path = config.intra\_path + config.train\_ins\_path

dev\_data\_path = config.intra\_path + config.dev\_ins\_path

test\_data\_path = config.intra\_path + config.test\_ins\_path

print("load entity index...")

entity2ID (config.entity\_index\_path)

print("load relation index...")

relation2ID (config.relation\_index\_path)

print("load kgfile...")

load\_kgfile (config.triple\_path)

print("load entity vectors in kb...")

load\_entity(config.entity\_vec\_path)

print("load relation vectors in kb...")

load\_relation(config.relation\_vec\_path)

print ('load intra instances')

train\_Label,train\_Entities,train\_Sentence,train\_sum = load\_file(train\_data\_path)

dev\_Label,dev\_Entities,dev\_Sentence,dev\_sum = load\_file(dev\_data\_path)

test\_Label,test\_Entities,test\_Sentence,test\_sum = load\_file(test\_data\_path)

output\_path = config.intra\_path

print ('dump intra instances')

process\_data(train\_Label,train\_Entities,train\_Sentence,train\_sum,\

dev\_Label,dev\_Entities,dev\_Sentence,dev\_sum,\

test\_Label,test\_Entities,test\_Sentence,test\_sum,\

output\_path, config.test\_ins\_path,config.word\_vec\_path)

#inter-sentence level

train\_data\_path = config.inter\_path + config.train\_ins\_path

dev\_data\_path = config.inter\_path + config.dev\_ins\_path

test\_data\_path = config.inter\_path + config.test\_ins\_path

print ('load inter instances')

train\_Label,train\_Entities,train\_Sentence,train\_sum = load\_file(train\_data\_path)

dev\_Label,dev\_Entities,dev\_Sentence,dev\_sum = load\_file(dev\_data\_path)

test\_Label,test\_Entities,test\_Sentence,test\_sum = load\_file(test\_data\_path)

output\_path = config.inter\_path

print ('dump inter instances')

process\_data(train\_Label,train\_Entities,train\_Sentence,train\_sum,\

dev\_Label,dev\_Entities,dev\_Sentence,dev\_sum,\

test\_Label,test\_Entities,test\_Sentence,test\_sum,\

output\_path, config.test\_ins\_path,config.word\_vec\_path)

4. main.py

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

'''

Filename : main.py

Function : Train and test the model

1. Train and test model on intra-sentence instances

2. Train and test model on inter-sentence instances

3. Merge the best results (on development set) of intra- and inter-sentence instances

'''

import os

import pickle

import numpy as np

import torch

import torch.nn as nn

import torch.optim as optim

from torch.autograd import Variable

from merge\_result import merge\_intra\_inter

import torch.nn.functional as F

from config import Config

from KR\_ERE\_model import KR\_ERE

torch.manual\_seed(1337)

torch.cuda.manual\_seed(1337)

mySeed = np.random.RandomState(1234)

config = Config()

def get\_property(max\_len,wordEmbed,entityEmbed,relationEmbed,sentence,ent\_feat,triple):

sent\_id = ent\_feat[0]

e1\_id = ent\_feat[1]

e1\_start\_pos = ent\_feat[2]

e1\_end\_pos = ent\_feat[3]

e2\_id = ent\_feat[4]

e2\_start\_pos = ent\_feat[5]

e2\_end\_pos = ent\_feat[6]

len\_e1 = e1\_end\_pos - e1\_start\_pos

len\_e2 = e2\_end\_pos - e2\_start\_pos

######################generate the sequence##################################

e1 = sentence[e1\_start\_pos:e1\_end\_pos]

e2 = sentence[e2\_start\_pos:e2\_end\_pos]

pos = sorted([(e1\_start\_pos,-1),(e1\_end\_pos,-1),(e2\_start\_pos,-2),(e2\_end\_pos,-2)],key=lambda p : p[0])

sentence = sentence[:pos[0][0]]+[pos[0][1]]+sentence[pos[1][0]:pos[2][0]]+[pos[2][1]]+sentence[pos[3][0]:]

n = len(sentence)

e1vectors = torch.cat([wordEmbed(Variable(torch.LongTensor([int(e)]).cuda())).view(100,1) for e in e1],1)

e2vectors = torch.cat([wordEmbed(Variable(torch.LongTensor([int(e)]).cuda())).view(100,1) for e in e2],1)

e1vector = torch.sum(e1vectors,1)/len\_e1

e2vector = torch.sum(e2vectors,1)/len\_e2

wordVectorLength = len(e2vector)

#wordsembeddings:

if sentence[0] == -1:

words = e1vector.view(wordVectorLength,1)

elif sentence[0] == -2:

words = e2vector.view(wordVectorLength,1)

else:

words = wordEmbed(Variable(torch.LongTensor([sentence[0]]).cuda())).view(wordVectorLength,1)

for word in sentence[1:]:

if word == -1:

words = torch.cat([words, e1vector.view(wordVectorLength,1)],1)

elif word == -2:

words = torch.cat([words, e2vector.view(wordVectorLength,1)],1)

else:

words = torch.cat([words, wordEmbed(Variable(torch.LongTensor([word]).cuda())).view(wordVectorLength,1)],1)

wordswithPos = words

#########################generate the kb################################

E1 = entityEmbed(Variable(torch.LongTensor([triple[0]]).cuda()))

E2 = entityEmbed(Variable(torch.LongTensor([triple[1]]).cuda()))

relation = relationEmbed(Variable(torch.LongTensor([triple[2]]).cuda()))

return sent\_id, wordswithPos, e1\_id,e2\_id,E1,E2,e1vectors,e2vectors,len\_e1,len\_e2,relation, n

def train\_and\_test(data\_path,save\_path,is\_intra,func):

#####################load the data###########################

train\_Sentence,train\_Label,train\_Entities,train\_Triples,train\_gold\_sum,\

dev\_Sentence,dev\_Label,dev\_Entities,dev\_Triples,dev\_gold\_sum,\

test\_Sentence,test\_Label,test\_Entities,test\_Triples,test\_gold\_sum,\

word2vector,entity2vector,relation2vector = pickle.load(open(data\_path,'rb'),encoding = 'iso-8859-1') #all the data for model training

#####################get he feature###########################sentence length

max\_len = max(max([len(sentence) for sentence in train\_Sentence]),\

max([len(sentence) for sentence in dev\_Sentence]),\

max([len(sentence) for sentence in test\_Sentence]))

max\_word\_num = len(word2vector) #the number of the words

embedding\_dim = len(word2vector[0])

embedding\_dim\_kg = len(entity2vector[0])

print ("max sentence length: {}".format(max\_len))

print ("unique word number: {}".format(max\_word\_num))

print ("word embedding dim: {}".format(embedding\_dim))

print ("knowledge embedding dim: {}".format(embedding\_dim\_kg))

#################prepare the embedding layer#################

word2vector = nn.Parameter(torch.FloatTensor(word2vector).cuda())

wordEmbed = nn.Embedding(max\_word\_num,embedding\_dim)

wordEmbed.weight = word2vector

entitynum = len(entity2vector)

entity2vector = nn.Parameter(torch.FloatTensor(entity2vector).cuda())

entityEmbed = nn.Embedding(entitynum,embedding\_dim\_kg)

entityEmbed.weight = entity2vector

relationnum = len(relation2vector)

relation2vector = nn.Parameter(torch.FloatTensor(relation2vector).cuda())

relationEmbed = nn.Embedding(relationnum,embedding\_dim\_kg)

relationEmbed.weight = relation2vector

######################prepare the data#######################

print ('generate the train data...')

train\_set = []

for i in range(len(train\_Sentence)):

sampleTuple = get\_property(max\_len = max\_len,

wordEmbed = wordEmbed,

entityEmbed = entityEmbed,

relationEmbed = relationEmbed,

sentence = train\_Sentence[i],

ent\_feat = train\_Entities[i],

triple =train\_Triples[i])

train\_set.append(sampleTuple)

print ('generate the development data...')

dev\_set = []

for i in range(len(dev\_Sentence)):

sampleTuple = get\_property(max\_len = max\_len,

wordEmbed = wordEmbed,

entityEmbed = entityEmbed,

relationEmbed = relationEmbed,

sentence = dev\_Sentence[i],

ent\_feat = dev\_Entities[i],

triple =dev\_Triples[i])

dev\_set.append(sampleTuple)

print ('generate the test data...')

test\_set = []

for i in range(len(test\_Sentence)):

sampleTuple = get\_property(max\_len = max\_len,

wordEmbed = wordEmbed,

entityEmbed = entityEmbed,

relationEmbed = relationEmbed,

sentence = test\_Sentence[i],

ent\_feat = test\_Entities[i],

triple =test\_Triples[i])

test\_set.append(sampleTuple)

#########################Model###############################

model = func(wordEmbed,entityEmbed,relationEmbed)

print("model fitting - {}".format(model.name))

prediction = model.train\_fit(trainset = train\_set,

trainLabel = train\_Label,

train\_gold\_num = train\_gold\_sum,

valset = dev\_set,

valLabel = dev\_Label,

val\_gold\_num = dev\_gold\_sum,

testset = test\_set,

testLabel = test\_Label,

test\_gold\_num = test\_gold\_sum,

resultOutput = save\_path,

is\_intra = is\_intra,

pretrain\_path = config.pretrain\_model\_path)

return prediction

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

##############intra sentence level########################

data\_path = config.intra\_path + config.clean\_data

save\_path = config.result\_path + config.intra

#####train and predict########

intra\_prediction = train\_and\_test(data\_path = data\_path,

save\_path = save\_path,

is\_intra = True,

func = KR\_ERE)

if config.is\_document:

##############inter sentence level########################

data\_path = config.inter\_path + config.clean\_data

ctd\_path = config.result\_path + config.inter

#####train and predict########

inter\_prediction = train\_and\_test(data\_path = data\_path,

save\_path = save\_path,

is\_intra = False,

func = KR\_ERE)

merge\_intra\_inter(intra\_path = intra\_prediction,

inter\_path = inter\_prediction,

save\_path = config.result\_path + config.document)

5. KR\_ERE\_model.py

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

'''

Filename : KR\_ERE\_model.py

Function : Desigh the KR\_ERE model used in our software

train\_fit: Train the model on training set

test\_eval: Test the model on development/test set

'''

import numpy as np

import os

import random

import torch

import torch.nn as nn

import torch.optim as optim

from torch.autograd import Variable

import torch.nn.functional as F

from collections import defaultdict

import time

import argparse

from config import Config

from doc\_level\_evaluation import evaluate\_score

mySeed = np.random.RandomState(1234)

class KR\_ERE(nn.Module):

def \_\_init\_\_(self, wordEmbed,entityEmbed,relationEmbed):

super(KR\_ERE,self).\_\_init\_\_()

self.name = 'KR\_ERE\_model'

self.wordEmbed = wordEmbed

self.entityEmbed = entityEmbed

self.relationEmbed = relationEmbed

self.config = Config()

self.batchSize = self.config.batch\_size

self.wordVectorLength = self.config.word\_vec\_dim

self.vectorLength = self.config.word\_vec\_dim

self.entityLength = self.config.kg\_vec\_dim

self.classNumber = self.config.class\_number

self.numEpoches = self.config.epoch\_number

self.convdim = self.config.convolution\_dim

self.dropout = nn.Dropout(p=0.5)

self.kernel\_size = self.config.kernel\_size

self.convstc = nn.ModuleList([nn.Conv1d(self.vectorLength, self.convdim, K, padding=int((K-1)/2), bias = True) for K in self.kernel\_size]).cuda()

self.convsc = nn.ModuleList([nn.Conv1d(self.vectorLength, self.convdim, K, padding=int((K-1)/2), bias = True) for K in self.kernel\_size]).cuda()

self.convstd = nn.ModuleList([nn.Conv1d(self.vectorLength, self.convdim, K, padding=int((K-1)/2), bias = True) for K in self.kernel\_size]).cuda()

self.convsd = nn.ModuleList([nn.Conv1d(self.vectorLength, self.convdim, K, padding=int((K-1)/2), bias = True) for K in self.kernel\_size]).cuda()

self.chemical\_W = nn.Parameter(torch.FloatTensor(mySeed.uniform(-0.01, 0.01, (self.convdim, self.entityLength ))).cuda (), requires\_grad = True)

self.chemical\_b = nn.Parameter(torch.FloatTensor(mySeed.uniform(-0.01, 0.01, (self.convdim, 1 ))).cuda (), requires\_grad = True)

self.disease\_W = nn.Parameter(torch.FloatTensor(mySeed.uniform(-0.01, 0.01, (self.convdim, self.entityLength ))).cuda (), requires\_grad = True)

self.disease\_b = nn.Parameter(torch.FloatTensor(mySeed.uniform(-0.01, 0.01, (self.convdim, 1 ))).cuda (), requires\_grad = True)

self.LinearLayer\_W = nn.Parameter(torch.FloatTensor(mySeed.uniform(-0.01, 0.01, (self.convdim, self.convdim \* 2 \* len(self.kernel\_size) ))).cuda (), requires\_grad = True)

self.LinearLayer\_b = nn.Parameter(torch.FloatTensor(mySeed.uniform(-0.01, 0.01, (self.convdim, 1 ))).cuda (), requires\_grad = True)

self.attention\_W = nn.Parameter(torch.FloatTensor(mySeed.uniform(-0.01, 0.01, (self.entityLength, self.convdim ))).cuda (), requires\_grad = True)

self.attention\_b = nn.Parameter(torch.FloatTensor(mySeed.uniform(-0.01, 0.01, (self.entityLength, 1))).cuda (), requires\_grad = True)

self.softmaxLayer\_W = nn.Parameter(torch.FloatTensor(mySeed.uniform(-0.01, 0.01, ( self.classNumber, self.convdim ))).cuda(), requires\_grad=True)

self.softmaxLayer\_b = nn.Parameter(torch.FloatTensor(mySeed.uniform(-0.01, 0.01, ( self.classNumber,1 ))).cuda(), requires\_grad=True)

self.softmax = torch.nn.Softmax(dim = 1)

self.loss\_function = torch.nn.NLLLoss()

def forward(self, contxtWords, e1,e2, e1vs,e2vs,e1v,e2v, relation, senlength,is\_train):

softmaxLayer\_W = self.softmaxLayer\_W

softmaxLayer\_b = self.softmaxLayer\_b

vectorLength = self.vectorLength

#generate entity expression

E1 = torch.mm(self.chemical\_W, e1.view(self.wordVectorLength,1)) + self.chemical\_b

E2 = torch.mm(self.disease\_W, e2.view(self.wordVectorLength,1)) + self.disease\_b

contxt\_chem = []

contxt\_dis = []

gate\_chem = []

gate\_dis = []

for i,conv in enumerate(self.convstc):

if i%2:

contxt\_chem.append(torch.tanh(conv(torch.cat([contxtWords.view(1,vectorLength,senlength),Variable(torch.zeros(1,vectorLength,1).cuda())],2))))

else:

contxt\_chem.append(torch.tanh(conv(contxtWords.view(1,vectorLength,senlength))))

for i,conv in enumerate(self.convstd):

if i%2:

contxt\_dis.append(torch.tanh(conv(torch.cat([contxtWords.view(1,vectorLength,senlength),Variable(torch.zeros(1,vectorLength,1).cuda())],2))))

else:

contxt\_dis.append(torch.tanh(conv(contxtWords.view(1,vectorLength,senlength))))

for i,conv in enumerate(self.convsc):

if i%2:

gate\_chem.append(torch.relu(conv(torch.cat([contxtWords.view(1,vectorLength,senlength),Variable(torch.zeros(1,vectorLength,1).cuda())],2))+ E1.view(1,self.convdim,1)))#

else:

gate\_chem.append(torch.relu(conv(contxtWords.view(1,vectorLength,senlength)) + E1.view(1,self.convdim,1)))#

for i,conv in enumerate(self.convsd):

if i%2:

gate\_dis.append(torch.relu(conv(torch.cat([contxtWords.view(1,vectorLength,senlength),Variable(torch.zeros(1,vectorLength,1).cuda())],2))+ E2.view(1,self.convdim,1)))#

else:

gate\_dis.append(torch.relu(conv(contxtWords.view(1,vectorLength,senlength)) + E2.view(1,self.convdim,1)))#

contxtWords\_chem = [(i\*j).squeeze(0) for i, j in zip(contxt\_chem, gate\_chem)]

contxtWords\_dis = [(i\*j).squeeze(0) for i, j in zip(contxt\_dis, gate\_dis)]

contxtWords0\_chem = []

contxtWords0\_dis = []

for contxt\_chem,contxt\_dis in zip(contxtWords\_chem,contxtWords\_dis):

att = self.softmax( torch.mm(relation.view(1,self.entityLength), torch.tanh(torch.mm(self.attention\_W,contxt\_chem) + self.attention\_b)) )

contxtWords0\_chem.append(torch.mm(att,contxt\_chem.transpose(0,1)).view(self.convdim,1))

att = self.softmax( torch.mm(relation.view(1,self.entityLength), torch.tanh(torch.mm(self.attention\_W,contxt\_dis) + self.attention\_b)) )

contxtWords0\_dis.append(torch.mm(att,contxt\_dis.transpose(0,1)).view(self.convdim,1))

contxtWords0\_chem = torch.cat(contxtWords0\_chem,0)

contxtWords0\_dis = torch.cat(contxtWords0\_dis,0)

contxtWords0 = torch.cat([contxtWords0\_chem,contxtWords0\_dis],0)

linearLayerOut = torch.relu(torch.mm(self.LinearLayer\_W,self.dropout(contxtWords0)) + self.LinearLayer\_b)

finallinearLayerOut = torch.mm(softmaxLayer\_W,linearLayerOut) + softmaxLayer\_b

return finallinearLayerOut

def load\_pretrain\_parameters (self, parameters\_path = None):

if parameters\_path != None:

checkpoint = torch.load(parameters\_path)

self.load\_state\_dict(checkpoint)

def train\_fit(self, trainset,trainLabel,train\_gold\_num,valset,valLabel,val\_gold\_num,testset,testLabel,test\_gold\_num,resultOutput, is\_intra = True,pretrain\_path = ''):

F1 = 0

indicates=list(range(len(trainset)))

trainsetSize = len(trainset)

if pretrain\_path:

self.load\_pretrain\_parameters(pretrain\_path)

if is\_intra:

learn\_rate = self.config.intra\_lr

else:

learn\_rate = self.config.inter\_lr

optimizer = optim.Adam(self.parameters(), lr = learn\_rate)

for epoch\_idx in range (self.numEpoches):

mySeed.shuffle(indicates)

total\_loss = Variable(torch.FloatTensor([0]).cuda(), requires\_grad=True)

sum\_loss= 0.0

print("=====================================================================")

print("epoch " + str(epoch\_idx) + ", trainSize: " + str(trainsetSize))

count = 0

correct = 0

tp = 0

tp\_fp = 0

predict\_dic = []

time0 = time.time()

self.train()

for i in range(len(indicates)):

sentid, sentwords,e1id,e2id,e1,e2,e1vs,e2vs,e1v,e2v,relation, senlength= trainset[indicates[i]]

finallinearLayerOut = self.forward(

sentwords,

e1,

e2,

e1vs,

e2vs,

e1v,

e2v,

relation,

senlength,

True

)

log\_prob = F.log\_softmax(finallinearLayerOut.view(1, self.classNumber),dim = 1)

loss = self.loss\_function(log\_prob, Variable(torch.LongTensor([trainLabel[indicates[i]]]).cuda()))

classification = self.softmax(finallinearLayerOut.view(1, self.classNumber))

total\_loss = torch.add(total\_loss, loss)

predict = np.argmax(classification.cpu().data.numpy())

prediction = [sentid,e1id,e2id,predict]

if predict == trainLabel[indicates[i]]:

correct += 1.0

count += 1

if predict and prediction not in predict\_dic:

predict\_dic.append (prediction)

tp\_fp += 1

if predict == trainLabel[indicates[i]]:

tp += 1

####################Update#######################

if count % self.batchSize == 0:

total\_loss = total\_loss/self.batchSize

total\_loss.backward(retain\_graph=True)

optimizer.step()

optimizer.zero\_grad()

total\_loss = Variable(torch.FloatTensor([0]).cuda(),requires\_grad = True)

optimizer.step()

optimizer.zero\_grad()

self.eval()

resultStream = open(resultOutput + "vresult\_" + str(epoch\_idx) + ".txt", 'w')

probPath = resultOutput + "vprob\_" + str(epoch\_idx) + ".txt"

VP,VR,VF = self.test\_eval(valset,valLabel, val\_gold\_num,resultStream, probPath)

resultStream = open(resultOutput + "tresult\_" + str(epoch\_idx) + ".txt", 'w')

probPath = resultOutput + "tprob\_" + str(epoch\_idx) + ".txt"

TP,TR,TF = self.test\_eval(testset,testLabel, test\_gold\_num, resultStream, probPath)

resultStream.close()

if VF >= F1:

F1 = VF

file\_path = resultOutput + "tresult\_" + str(epoch\_idx) + ".txt"

torch.save(self.state\_dict(),self.config.model\_save\_path)

####################Update#######################

train\_P = tp/max(tp\_fp,1)

train\_R = tp/train\_gold\_num

train\_F = 2\*train\_P\*train\_R/max(0.0001,(train\_P+train\_R))

time1 = time.time()

print (tp,tp\_fp,train\_gold\_num)

print("train P: ", train\_P, " R: ", train\_R , " F1: ", train\_F)

print("val P: ", VP, " R: ", VR , " F1: ", VF)

print("test P: ", TP, " R: ", TR , " F1: ", TF)

print("Iteration", epoch\_idx, "Loss", total\_loss.cpu().data.numpy()[0] / self.batchSize, "train Acc: ", float(correct / count) , "time: ", str(time1 - time0))

return file\_path

def test\_eval(self, testset, testLabel,gold\_correct,resultStream, probPath):

time0 = time.time()

probs = []

predict\_dic = []

correct = 0

count = 0

for i in range(len(testset)):

sentid, sentwords, e1id,e2id,e1,e2,e1vs,e2vs,e1v,e2v,relation,senlength = testset[i]

finallinearLayerOut = self.forward(

sentwords,

e1,

e2,

e1vs,

e2vs,

e1v,

e2v,

relation,

senlength,

False

)

classification = self.softmax(finallinearLayerOut.view(1, self.classNumber))

prob = classification.cpu().data.numpy().reshape(self.classNumber)

predict = np.argmax(prob)

probs.append(prob)

prediction = [sentid,e1id,e2id,predict]

if predict and (prediction not in predict\_dic):

resultStream.write("\t".join([sentid, "CID", e1id, e2id]) + "\n")

predict\_dic.append(prediction)

if predict and testLabel[i]:

correct += 1

count += 1

P = correct/max(count,1)

R = correct/gold\_correct

F = 2\*P\*R/max(0.0001,(P+R))

if probPath:

np.savetxt(probPath, probs, '%.5f',delimiter=' ')

time1 = time.time()

print("test time : ", str(time1 - time0))

return P,R,F

6. merge\_result.py

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

'''

Filename : merge\_result.py

Function : merge the intra-sentence result and inter-sentence result as final results

Note that we choose the epoch where the development results are the best, to combine the results of development set and test set.

'''

import numpy as numpy

import os

instance = []

probability = []

def merge(result\_path):

with open (result\_path,'r') as f:

for line in f:

line = line.strip().split('\t')

if line not in instance:

instance.append(line)

probability.append('0.5')

def out\_result(merge\_path):

out = open (merge\_path,'w')

for i in range(len(instance)):

ins = '\t'.join(instance[i])

out.write(ins+'\t'+probability[i]+'\n')

out.close()

def merge\_intra\_inter(intra\_path,inter\_path,save\_path):

intra\_epoch = intra\_path.split('.')[0].split('\_')[-1]

inter\_epoch = inter\_path.split('.')[0].split('\_')[-1]

merge\_path = 'tmerge\_result\_' + intra\_epoch +'\_'+ inter\_epoch+'.txt'

merge(intra\_path)

merge(inter\_path)

out\_result(merge\_path)