|  |
| --- |
| **def** \_\_iter\_\_(self):  **for** fname **in** os.listdir(self.dirname):  **for** line **in** open(os.path.join( \  self.dirname, fname),\  errors="ignore"):  **if** self.caseless:  line = line[0].lower()+line[1:]   ori\_line = line  line = line.strip('\n').split('\t') text = line[1]  raw\_label = line[2].strip(':: ')   **if** self.split\_line:  **if** self.split\_method == \  'Twitter':  tknz = TweetTokenizer()  text = tknz.tokenize(text)  **elif** self.split\_method == \  'space':  text = text.split(' ')  **elif not** self.parser **is None**:  text = \  text.split(self.parser)   **if not** self.split\_line:  text = ori\_line **if** self.w2v:  text = list(map(self.\_\_w2v\_\_,\  text))   **if** self.matlabel:  raw\_label = \  self.label\_dict[raw\_label]   **if** self.tensor\_out:  temp = np.array(text)  text = np.zeros((1, \  self.max\_length, \  self.vec\_length))  text[0, :len(temp), :] = temp  raw\_label = \  np.array(raw\_label) \  .reshape((1, 6))   **if** self.label:  **yield** text, raw\_label  **else**:  **yield** text |

Data pipeline:

Data set:

Preprocess: remove stop words:

|  |
| --- |
| **def \_\_pre\_process\_\_**(text):  text = re.sub(r'@\w\* ', '', text)  text = re.sub(r'&\w\*;', '', text)  text = re.sub(r'&#\w\*;', '', text)  text = re.sub(r'#\w\*', '', text)  text = re.sub(r'@', '', text)  text = re.sub(r'[,.;"]+', '', text) |

Data split:

Split data into training set (60%), evaluation set (20%), test set (20%):

|  |
| --- |
| **def data\_split**(sentences):  path = './data\_set/train/'  **if not** os.path.exists(path):  os.mkdir(path)  #...... check path avaliability  line\_count = 0  **for** \_ **in** sentences:  line\_count += 1   count = 0  **for** line, label **in** sentences:  **if** count / line\_count < 0.6:  **if not \**  os.path.exists(path):  os.mkdir(path)  **with \**  open(‘train file’) **as** f:  f.write(line)  **elif** count / line\_count < 0.8:  **with** open(‘eval’) **as** f:  f.write(line)  # f.write('\n')  **else**:  **with** open(‘test’) **as** f:  f.write(line)  # f.write('\n')   count += 1 |

Data set next batch:

Due to the limit of our computer, it is not possible to use full data set in a batch, so we write a batch generator to complete task. At first we wanted to write a batch generator which can implement multi thread mechanism so that it can decrease the overall training time, however, it does not work, so we write a normal generator called “next\_batch\_stupid”

To generate next batch for lstm training

|  |
| --- |
| **def next\_batch\_stupid**(self, batch\_size):  **try**:  batch\_x, batch\_y = \  next(self.gen)  **except** StopIteration **as** e:  self.gen = \  self.sentences.\_\_iter\_\_()  batch\_x, batch\_y = \  next(self.gen)   **for** i **in** range(batch\_size - 1):  **try**:  new\_x, new\_y = \  next(self.gen)  **except** StopIteration **as** e:  self.gen = \  self.sentences.\_\_iter\_\_()  new\_x, new\_y = \  next(self.gen)   batch\_x =np.append(batch\_x,\  new\_x, axis=0)  batch\_y =np.append(batch\_y,\  new\_y, axis=0)   **return** batch\_x, batch\_y |

Word embed:

We used genism library to train and look up word to vec algorithm:

|  |
| --- |
| # load sentences to varibables sentences=Sentences(dirname='data’,\  split\_line=**True**, \  split\_method='Twitter')   # train the model model=gensim.models.Word2Vec(sentences,  size=100, min\_count=1,\  workers=10, iter=2000) |

LSTM:

Set up LSTM model (1 layer):

|  |
| --- |
| **def RNN**(x, weights, biases):  x = tf.transpose(x, [1, 0, 2])  x = tf.reshape(x, [-1, n\_input])  x = tf.split(0, n\_steps, x)   lstm\_cell = rnn\_cell.LSTMCell(n\_hidden, \  forget\_bias=1.0)   # Get lstm cell output  outputs, states = \  rnn.rnn(lstm\_cell, x, \  dtype=tf.float32)   **if not** soft\_layer:  # Linear activation, using rnn inner loop last output  **return** tf.matmul(outputs[-1], weights['out']) + biases['out']  **else**:  # Linear activation and softmax output   line\_out=tf.matmul(outputs[-1], \  weights['out']) + biases['out']  **return** tf.nn.softmax(line\_out) |

Set up LSTM (3 layer):

|  |
| --- |
| **def RNN**(x, weights, biases):  x = tf.transpose(x, [1, 0, 2])  x = tf.reshape(x, [-1, n\_input])  x = tf.split(0, n\_steps, x)   lstm\_cell = rnn\_cell.BasicLSTMCell(n\_hidden, \  forget\_bias=0.0,state\_is\_tuple=**True**)  **if** keep\_prob<1:  lstm\_cell=tf.nn.rnn\_cell.DropoutWrapper(  lstm\_cell,output\_keep\_prob=keep\_prob  )  cell=tf.nn.rnn\_cell.MultiRNNCell([lstm\_cell]\*num\_layers,state\_is\_tuple=**True**)   # Get lstm cell output  outputs, states = rnn.rnn(cell, x, dtype=tf.float32)   ## try  line\_out = tf.matmul(outputs[-1], weights['out']) + biases['out']  **return** tf.nn.softmax(line\_out) |

Feature Reduction:

Least Representationive Words:

Find out the least representative words (high appearance but help little in emotion expression)

|  |
| --- |
| **def find\_least\_repres**(dic\_c,dic\_common):  #----------------print the least representative words-----------------#  keys=list(dic\_common.keys())  values=list(dic\_common.values())  sumtotal=0  **for** i **in** range(len(values)):  sumtotal+=values[i]  dic\_var={}  dic\_common\_sorted=sorted(dic\_common.items(),key=**lambda** dic\_common:dic\_common[1],reverse=**True**)  l=len(dic\_common\_sorted)  var\_m=[]  dic\_final={}  **for** i **in** range(len(dic\_common)):  pm=[]  **for** j **in** range(len(dic\_c)):  **if** keys[i] **in** dic\_c[j]:  pm.append(dic\_c[j][keys[i]]/values[i])  **else**:  pm.append(0)  pmarray1=np.array(pm)  sum1=pmarray1.sum()  pmarray2=pmarray1\*pmarray1  sum2=pmarray2.sum()  mean=sum1/(len(dic\_c))  var=sum2/(len(dic\_c))-mean\*\*2  word=keys[i]  dic\_var[word]=var  #var\_m.append(var)  score=-np.log((values[i]/sumtotal))\*var  dic\_final[word]=score  dic\_final\_sorted=sorted(dic\_final.items(),key=**lambda** dic\_final:dic\_final[1])  print('-----------------------------')  print('print the least representative words')  count=0  # print(dic\_common\_sorted[:10])  **for** i **in** range(100):  sys.stdout.write('\" '+dic\_final\_sorted[i][0]+' \"' + ',') |

Most Representative Words:

|  |
| --- |
| **def find\_most\_repres**(dic,dic\_common,dic\_label):  #------------print the most representative words-------------#   **for** j **in** range(len(dic)):  dic\_p={} #dic\_p is the dictionary that is to be sorted  dic\_count\_sorted= dic[j] #dic\_count\_sorted is a list which contains sorted list of every category with number of occurrence  l=len(dic\_count\_sorted)  medium=int(l/12)  **for** k **in** range(len(dic\_count\_sorted)):  occurrence\_p=float(dic\_count\_sorted[k][1]/dic\_common[dic\_count\_sorted[k][0]])  word=dic\_count\_sorted[k][0]  dic\_p[word]=occurrence\_p  # dic\_tobesort=sorted(dic\_tobesort, key=lambda dic\_tobesort: dic\_tobesort[1], reverse=True)  dic\_prob\_sorted=sorted(dic\_p.items(),key=**lambda** dic\_p:dic\_p[1],reverse=**True**)  #print(dic\_prob\_sorted)  print('-----------------------------------')  print('most representative words for '+dic\_label[j])  count=0  **for** i **in** range(len(dic\_prob\_sorted)):  word=dic\_prob\_sorted[i][0]  **for** j **in** range(len(dic\_count\_sorted)):  **if** word==dic\_count\_sorted[j][0]:  number=dic\_count\_sorted[j][1]  **if** (number>=dic\_count\_sorted[medium][1]):  print(dic\_prob\_sorted[i][0])  count+=1  **if** count>9:  **break** |