Text Similarity

Group 1: Lee, Sam, Alvin, Miracle
Teacher: Wang Peng

Abstract

Text Similarity plays a significant role in the fields of web science and natural language process. Currently, there are many approaches to measure the similarity between two texts, such as TF/IDF + Cosine Similarity, Jaccard Similarity, Simhash algorithm, word movers' distance method, Cosine Similarity and Knowledge-based Similarity. Some tools created by some advanced universities can also be used to reach our aims. Although there are some limitations for each method, their advantages still play an important role in various fields. This paper, we try to implement five kinds of methods to reach our aim of text similarity and analyze their features briefly, which can lay the foundation for future study.

Key words: text similarity, cosine Similarity, word movers' distance

1. Problem definition

Text similarity has to determine how ‘close’ two pieces of text are both in surface closeness [lexical similarity] and meaning [semantic similarity].

For instance, how similar are the phrases “the cat ate the mouse” with “the mouse ate the cat food” by just looking at the words?

- On the surface, if you consider only word level similarity, these two phrases appear very similar as 3 of the 4 unique words are an exact overlap. It typically does not take into account the actual meaning behind words or the entire phrase in context.
- Instead of doing a word for word comparison, we also need to pay attention to context in order to capture more of the semantics. To consider semantic similarity we need to focus on phrase/paragraph levels (or lexical chain level) where a piece of text is broken into a relevant group of related words prior to computing similarity. We know that while the words significantly overlap, these two phrases actually have different meaning.

1.1 Problem Description

Given two text documents (such as news reports) D1 and D2

How to calculate the similarity between D1 and D2

1.2 Problem Analyze

The core problem is that what is similarity between two texts.

Solutions:

- Jaccard Similarity
- TF/IDF + Cosine Similarity

- Word Embedding + Cosine Similarity
- Knowledge-based Similarity

1.3 Related works

The big idea is that you represent documents as vectors of features, and compare documents by measuring the distance between these features. There are multiple ways to compute features that capture the semantics of documents and multiple algorithms to capture dependency structure of documents to focus on meanings of documents.

Supervised training can help sentence embeddings learn the meaning of a sentence more directly.

2. Theories of measurements

2.1 Jaccard Similarity

Given two sets A, B, Jaccard coefficient is defined as the ratio of the size of the intersection of A and B to the size of the union of A and B.

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} = \frac{|A \cap B|}{|A| + |B| - |A \cap B|}$$

2.2 TF/IDF

Tf-idf stands for term frequency-inverse document frequency, and the tf-idf weight is a weight often used in information retrieval and text mining. This weight is a statistical measure used to evaluate how important a word is to a document in a collection or corpus. The importance increases proportionally to the number of times a word appears in the document but is offset by the frequency of the word in the

corpus. Variations of the tf-idf weighting scheme are often used by search engines as a central tool in scoring and ranking a document's relevance given a user query.

Typically, the tf-idf weight is composed by two terms: the first computes the normalized Term Frequency (TF), aka. the number of times a word appears in a document, divided by the total number of words in that document; the second term is the Inverse Document Frequency (IDF), computed as the logarithm of the number of the documents in the corpus divided by the number of documents where the specific term appears.

- TF: Term Frequency, which measures how frequently a term occurs in a document. Since every document is different in length, it is possible that a term would appear much more times in long documents than shorter ones. Thus, the term frequency is often divided by the document length (aka. the total number of terms in the document) as a way of normalization:

  TF(t) = (Number of times term t appears in a document) / (Total number of terms in the document).
- IDF: Inverse Document Frequency, which measures how important a term is. While computing TF, all terms are considered equally important. However it is known that certain terms, such as "is", "of", and "that", may appear a lot of times but have little importance. Thus we need to weigh down the frequent terms while scale up the rare ones, by computing the following:

  IDF(t) = log_e(Total number of documents / Number of documents with term t in it).

2.3 Simhash algorithm

The main idea is to reduce the dimension, map the high-dimensional feature vector into a low-dimensional feature vector, and determine whether the article is repeated or highly approximated by the Hamming Distance of the two vectors.

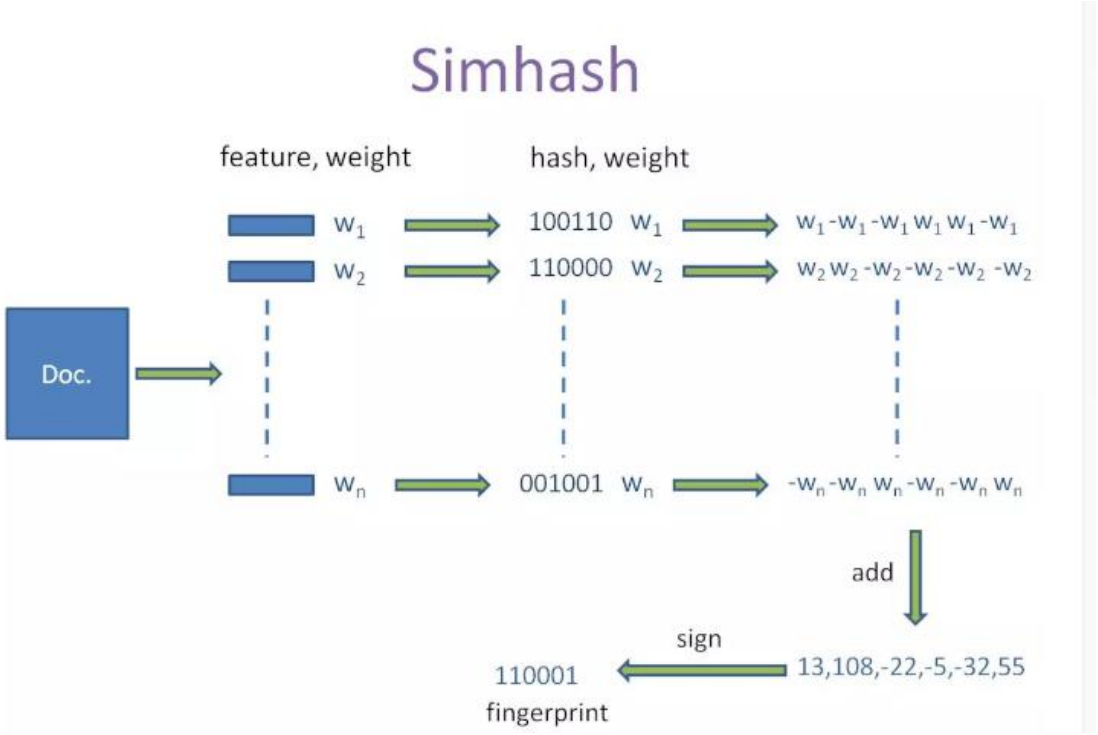


Figure 1: The structure of Simhash

Procedure：

- Given a text, the feature vector can be a word in the text, and its weight can be the number of times the word appears.
- Calculating the hash value of each feature vector through the hash function.
- Based on the hash value, all feature vectors are weighted.
- The weighting results of the above respective feature vectors are accumulated to become only one sequence string.

Dimensionality reduction : for the accumulated result, if it is greater than 0, it is set to 1, otherwise it is set to 0, so that the simhash value of the statement is obtained.

2.4 Word Embedding + Cosine Similarity

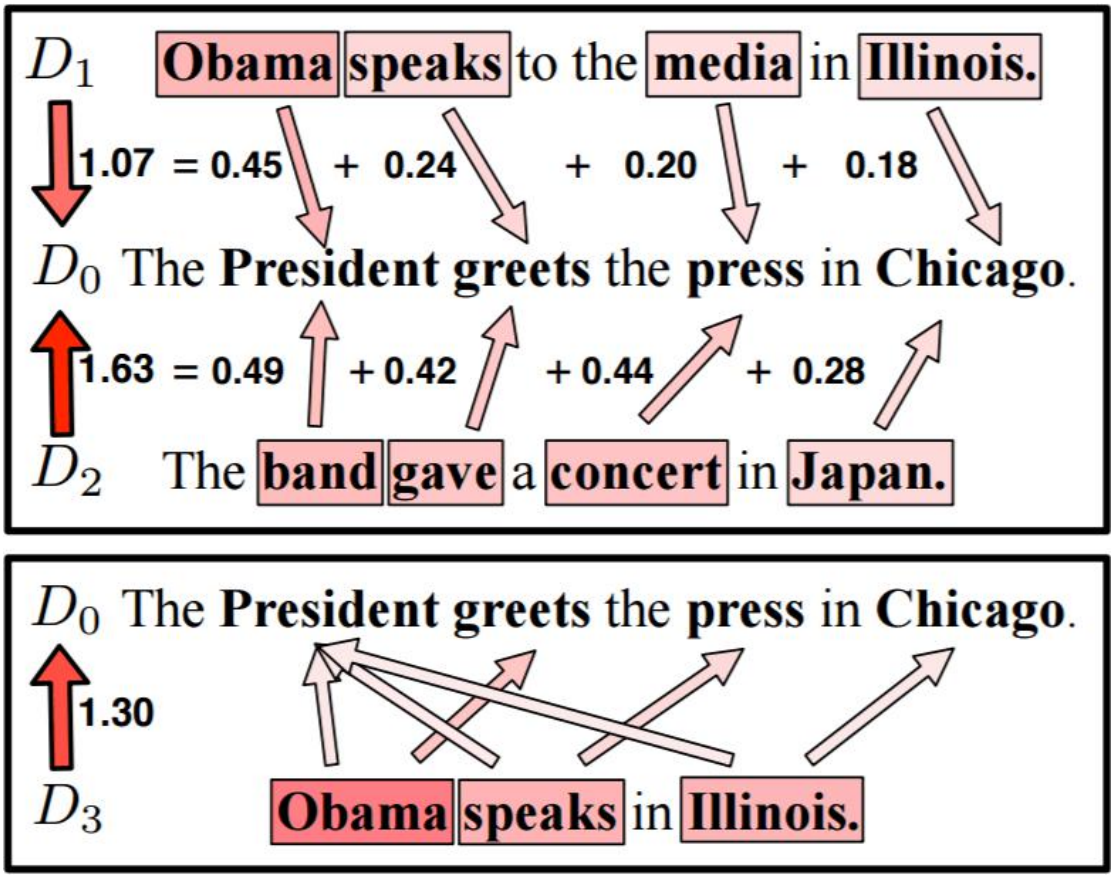
In the last homework, our team has implemented this method on word level. To compare two sentences, we need to transfer word vector to sentence vector. The simplest way is to weighted sum all the word vector appeared in the document and compute a mean value as the sentence vector. However, this method ignored the sequence of words, which represents significant sematic meaning. There are some

other ways like Recursive Neural Network, RNN and CNN. Those are too hard to implement in a short time.

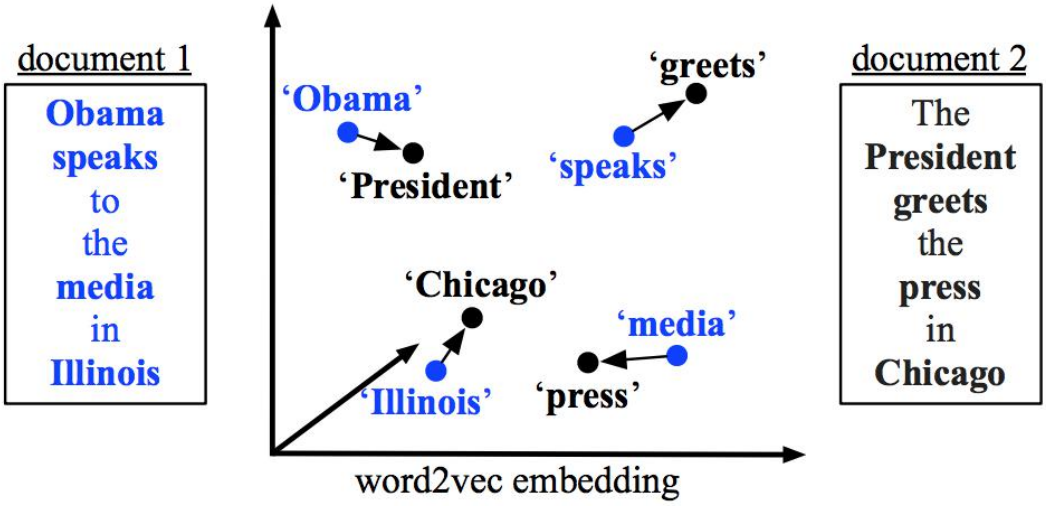
From other aspect, cosine similarity calculates similarity by measuring the cosine of angle between two vectors. Pre-trained method (such as Glove) matches up with cosine similarity can get a good sematic performance. However, as the figure shows, some methods can replace cosine similarity and achieve a better performance.

- **Jaccard Similarity** ☹☹☹
- Different embeddings+ **K-means** ☹☹
- Different embeddings+ **Cosine Similarity** ☹
- **Word2Vec + Smooth Inverse Frequency + Cosine Similarity** 😊
- Different embeddings+**LSI + Cosine Similarity** ☹
- Different embeddings+ **LDA + Jensen-Shannon distance** 😊
- Different embeddings+ **Word Mover Distance** 😊😊
- Different embeddings+ **Variational Auto Encoder (VAE)** 😊 😊
- Different embeddings+ **Universal sentence encoder** 😊😊
- Different embeddings+ **Siamese Manhattan LSTM** 😊😊😊
- **Knowledge-based Measures** ❤

In this part, we choose to implement embedding + word mover distance. Word mover distance takes account of the words' similarities in word embedding space. It uses the word embeddings of the words in two texts to measure the minimum distance that the words in one text need to "travel" in semantic space to reach the words in the other text.



The earth mover's distance (EMD) is a measure of the distance between two probability distributions over a region D (known as the Wasserstein metric). Informally, if the distributions are interpreted as two different ways of piling up a certain amount of dirt over the region D, the EMD is the minimum cost of turning one pile into the other; where the cost is assumed to be amount of dirt moved times the distance by which it is moved.



2.5 Knowledge-Based Similarity

In this part, we will talk about knowledge-based measurement of two document. Knowledge-based measures quantify semantic relatedness of words using a semantic network. The most frequently used network is called WordNet, which is a large lexical database for English. The figure below shows a subgraph of WordNet.

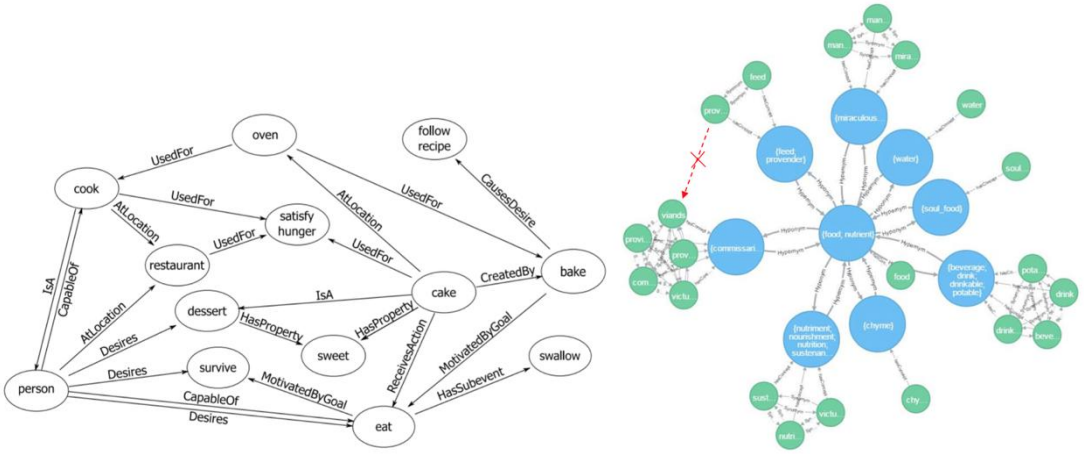


Figure x: WordNet graph(right) compared with ConceptNet graph.

Here we consider the problem of embedding entities and relationships of multi-relational data in low-dimensional vector spaces, its objective is to propose a canonical model which is easy to train, contains a reduced number of parameters and can scale up to very large databases. In Python NLTK module, it provides us with the function to calculate the similarity of two word. To compute sentence similarities or document similarities, we apply the formula from (Mihalcea & al, 2006), that is:

$$\mathrm{Sim}(T_1, T_2) = \frac{1}{2}\left(\frac{\sum_{w \in T_1} \mathrm{maxSim}(w, T_2) \cdot idf(w)}{\sum_{w \in T_1} idf(w)} + \frac{\sum_{w \in T_2} \mathrm{maxSim}(w, T_1) \cdot idf(w)}{\sum_{w \in T_2} idf(w)}\right),$$

T1 and T2 is the text units. Function maxSim is to compute the maximum value of word /omega and each word in another text unit. At the same time, by using idf value to weigh the similarity value.

3. Experiments

3.1 Algorithms Implement

3.1.1 Jaccard

```
work5.py
# encoding=utf-8
import jieba
def Jaccrad(model, reference):
    terms_reference = jieba.cut(reference)  # 默认精准模式
#   print("Default Mode: " + "/ ".join(terms_reference))  # 精确模式
    terms_model = jieba.cut(model)
#   print("Default Mode: " + "/ ".join(terms_model))  # 精确模式
    grams_reference = set(terms_reference)  # 去重；如果不需要就改为list
    grams_model = set(terms_model)
    temp = 0
    for i in grams_reference:
        if i in grams_model:
            temp = temp + 1# 交集
    fenmu = len(grams_model) + len(grams_reference) - temp  # 并集
    jaccard_coefficient = temp / float(fenmu)
    return jaccard_coefficient
file_handle =open('C:/Users/rcc/Desktop/test1.txt', mode='r')
a = file_handle.read()
file_handle1 =open('C:/Users/rcc/Desktop/test2.txt', mode='r')
b = file_handle1.read()
file_handle2 =open('C:/Users/rcc/Desktop/test3.txt', mode='r')
c = file_handle2.read()
jaccard_coefficient1 = Jaccrad(a, b)
jaccard_coefficient2 = Jaccrad(a, c)
print(jaccard_coefficient1)
print(jaccard_coefficient2)
```

```
Debug: work5
Debugger  Console
Loading model cost 0.380 seconds.
Prefix dict has been built succesfully.
0.469387755102
0.0717488789238
import sys; print('Python %s on %s' % (sys.version, sys.platform))
Python 2.7.13 |Continuum Analytics, Inc.| (default, May 11 2017, 13:17:26) [MSC v.1500 64 bit (AMD64)] on win32
>>>
```

Figure x: This is the code and result of Jaccard experiment.

3.1.2 TF-IDF

In this part, we manually implement the calculation of TF-IDF algorithm. Detailed process can be seen in the code.

In [1]:
```
import re
import numpy as np
import gensim
```

In [2]:
```
punctuation = '!,;:?"\(\)%\'1234567890$.\[\]'
docList=[]
for i in range(9):
    doc=[]
    file=open('doc/'+str(i)+'.txt')
    para=file.readlines()
    for line in para:
    #    clear punctuation
        temp=re.sub(r'[{}]+'.format(punctuation),'',line)
        temp=temp.strip().lower().split(' ')
        doc.append(temp)
    wordList=[]
    for i in doc:
        for j in i:
            if j:
                wordList.append(j)
    docList.append(wordList)
```

In [3]:
```
docNum=len(docList)
term_df = dict()
for doc in docList:
    for term in set(doc):
        if term not in term_df:
            term_df[term]=1.0
        else:
            term_df[term]+=1.0
```

In [4]:
```
from math import log10
for term in term_df:
    # term_df record the idf value of each words
    term_df[term] = log10(docNum/term_df[term])
```

3.1.3 Simhash Algorithm

In this part, we implement the Simhash Algorithm.

```
import math
import jieba
import jieba.analyse

class SimHash(object):

    def __init__(self):
        pass

    def getBinStr(self, source):
        if source == "":
            return 0
        else:
            x = ord(source[0]) << 7
            m = 1000003
            mask = 2 ** 128 - 1
            for c in source:
                x = ((x * m) ^ ord(c)) & mask
            x ^= len(source)
            if x == -1:
                x = -2
            x = bin(x).replace('0b', '').zfill(64)[-64:]
            print(source, x)

            return str(x)


    def getWeight(self, source):
        # fake weight with keyword
        return ord(source)
    def unwrap_weight(self, arr):
        ret = ""
        for item in arr:
            tmp = 0
            if int(item) > 0:
                tmp = 1
            ret += str(tmp)
        return ret
```

```
def simHash(self, rawstr):
    seg = jieba.cut(rawstr, cut_all=True)
    keywords = jieba.analyse.extract_tags("|".join(seg), topK=100, withWeight=True)
    print(keywords)
    ret = []
    for keyword, weight in keywords:
        binstr = self.getBinStr(keyword)
        keylist = []
        for c in binstr:
            weight = math.ceil(weight)
            if c == "1":
                keylist.append(int(weight))
            else:
                keylist.append(-int(weight))
        ret.append(keylist)
    # 对列表进行"降维"
    rows = len(ret)
    cols = len(ret[0])
    result = []
    for i in range(cols):
        tmp = 0
        for j in range(rows):
            tmp += int(ret[j][i])
        if tmp > 0:
            tmp = "1"
        elif tmp <= 0:
            tmp = "0"
        result.append(tmp)
    return "".join(result)

def getDistince(self, hashstr1, hashstr2):
    length = 0
    for index, char in enumerate(hashstr1):
        if char == hashstr2[index]:
            continue
        else:
            length += 1
    return length


if __name__ == "__main__":
    simhash = SimHash()
    # s1 = "我们喜欢吃阿克苏苹果，你呢"
    # s2 = "我们爱阿克苏苹，你呢"
    with open("1.txt", "r") as file:
        s1 = "".join(file.readlines())
        file.close()
    with open("2.txt", "r") as file:
        s2 = "".join(file.readlines())
        file.close()
    with open("3.txt", "r") as file:
        s3 = "".join(file.readlines())
        file.close()
    # s1 = "this is just test for simhash, here is the difference"
    # s2 = "this is a test for simhash, here is the difference"
    # print(simhash.getBinStr(s1))
    # print(simhash.getBinStr(s2))
    hash1 = simhash.simHash(s1)
    hash2 = simhash.simHash(s2)
    hash3 = simhash.simHash(s3)
    print("\n")
    print("网易Facebook：", hash1)
    print("新浪Facebook：", hash2)
    print("新华社三农：", hash3)
    distince1 = simhash.getDistince(hash1, hash2)
    distince2 = simhash.getDistince(hash1, hash3)
    # value = math.sqrt(len(s1)**2 + len(s2)**2)
    value1 = 6
    value = 6
    print("网易Facebook和新浪Facebook：海明距离：", distince1, "判定距离：", value1, "是否相似：", distince1<=value)
    print("网易Facebook和新华社三农：海明距离：", distince2, "判定距离：", value, "是否相似：", distince2 <= value)
```

```
网易Facebook： 00100011010111011101011101011000100001110011100010110100000101111
新浪Facebook： 00100111010111101101011100111000110001110111000001010100000101010
新华社三农： 00100101100011001110111101011111110100100000011000011111101100101
网易Facebook和新浪Facebook：海明距离： 13 判定距离： 6 是否相似： False
网易Facebook和新华社三农：海明距离： 29 判定距离： 6 是否相似： False
```

3.2 Word2Vec + Word Mover Distance

Rather than using cosine similarity, we decide to use word mover distance to compute similarity, both of which is based on word vector.

In the experiment, the word vector is loaded from GoogleNews-vectors-negative300, which is nearly 4G and obtain large scale words. Gensim module can help us to load this word vector.

In our experiment, I do not apply better methods to compose a sentence vector. The test object can only be the key words of a sentence in case that the words is not in the word vector. When facing the problem of text, I choose the key words by using TF-IDF.

Then we compute the word mover distance by using pulp module. The detailed process is in the figure.

In [20]:
```
def tokens_to_fracdict(tokens):
    cntdict = defaultdict(lambda : 0)
    for token in tokens:
        cntdict[token] += 1
    totalcnt = sum(cntdict.values())
    return {token: float(cnt)/totalcnt for token, cnt in cntdict.items()}
```

In [21]:
```
def word_mover_distance_probspec(first_sent_tokens, second_sent_tokens, wvmodel, lpFile=None):
    all_tokens = list(set(first_sent_tokens+second_sent_tokens))
    wordvecs = {token: wvmodel[token] for token in all_tokens}

    first_sent_buckets = tokens_to_fracdict(first_sent_tokens)
    second_sent_buckets = tokens_to_fracdict(second_sent_tokens)

    T = pulp.LpVariable.dicts('T_matrix', list(product(all_tokens, all_tokens)), lowBound=0)

    prob = pulp.LpProblem('WMD', sense=pulp.LpMinimize)
    prob += pulp.lpSum([T[token1, token2]*euclidean(wordvecs[token1], wordvecs[token2])
                        for token1, token2 in product(all_tokens, all_tokens)])
    for token2 in second_sent_buckets:
        prob += pulp.lpSum([T[token1, token2] for token1 in first_sent_buckets])==second_sent_buckets[token2]
    for token1 in first_sent_buckets:
        prob += pulp.lpSum([T[token1, token2] for token2 in second_sent_buckets])==first_sent_buckets[token1]

    if lpFile!=None:
        prob.writeLP(lpFile)

    prob.solve()

    return prob
```

In [22]:
```
def word_mover_distance(first_sent_tokens, second_sent_tokens, wvmodel, lpFile=None):
    prob = word_mover_distance_probspec(first_sent_tokens, second_sent_tokens, wvmodel, lpFile=lpFile)
    return pulp.value(prob.objective)
```

In [24]:
```
prob = word_mover_distance_probspec(['educator', 'expert', 'teaching'], ['student', 'love', 'study','math'], wvmodel)
print(pulp.value(prob.objective))
for v in prob.variables():
    if v.varValue!=0:
        print(v.name, '=', v.varValue)
```

```
3.354862110125402
T_matrix_('educator',_'love') = 0.083333333
T_matrix_('educator',_'student') = 0.25
T_matrix_('expert',_'love') = 0.083333333
T_matrix_('expert',_'study') = 0.25
T_matrix_('teaching',_'love') = 0.083333333
T_matrix_('teaching',_'math') = 0.25
```

4.4 WordNet

In this part, we use the Python NLTK module to get the word similarity, and implement the formula mentioned above which is to compute sentence similarity based on word similarity.

Firstly, we extract each word from the document together with clear of punctuation. Then we compute the idf value of each word based on the idf value formula and establish a dictionary to install the value.

This is the implement code:

In [2]:
```
punctuation = '!,;:?"\(\)%\'1234567890$.\[\]'
docList=[]
for i in range(9):
    doc=[]
    file=open('doc/'+str(i)+'.txt')
    para=file.readlines()
    for line in para:
    #    clear punctuation
        temp=re.sub(r'[{}]+'.format(punctuation),'',line)
        temp=temp.strip().lower().split(' ')
        doc.append(temp)
    wordList=[]
    for i in doc:
        for j in i:
            wordList.append(j)
    docList.append(wordList)
```

In [3]:
```
docNum=len(docList)
term_df = dict()
for doc in docList:
    for term in set(doc):
        if term not in term_df:
            term_df[term]=1.0
        else:
            term_df[term]+=1.0
```

In [4]:
```
from math import log10
for term in term_df:
    # term_df record the idf value of each words
    term_df[term] = log10(docNum/term_df[term])
```

The next step is to search the maximum similarity value of a word and a document. Further, do some simple operation to get the similarity of two document. Note that, we need to normalize the final value to 0-1 by dividing 10, since that I manually set the highest similarity value is 10.

```
In [5]: def maxSim(word,docNum):
            wordSimilarityValue=dict()
            word1=wn.synsets(word, pos=wn.NOUN)
            if word1:
                for i in set(docList[docNum]):
                    #   come across a problem here
                    word2=wn.synsets(i, pos=wn.NOUN)
                    if word2:
                        value=(word1[0].res_similarity(word2[0], brown_ic))*term_df[word]
                        if value>10:
                            value=10
                        wordSimilarityValue[i]=value
                    else:
                        wordSimilarityValue[i]=0
                maxSim=max(wordSimilarityValue.items(),key=lambda x:x[-1])
                return maxSim
            else:
                return (0,0)

In [6]: def doc2docSim(docNum1,docNum2):
            idf_val=0.0
            sumSim=0.0
            for i in docList[docNum1]:
                sumSim=sumSim+maxSim(i,docNum2)[1]
                idf_val=idf_val+term_df[i]
            return sumSim/idf_val

In [9]: def documentSimilarity(docNum1,docNum2):
        #     document similarity
            res=(doc2docSim(docNum1,docNum2)+doc2docSim(docNum2,docNum1))/2
        #     normalized
            return res/10

In [10]: documentSimilarity(0,8)
Out[10]: 0.402740386978241
```

4. Conclusion

This time, our group has done all the solution as we do last time. We manually implement TF-IDF algorithm to compute the importance of each word in the document.

In the experiment related embedding, we use word mover distance (WMD) to replace the cosine angle. In order to represent a document, we choose top 20 words with highest TF-IDF value. Then we compute the WMD value of keys, together with similarity of key words list.

In the last experiment, we use wordnet provided by Princeton University as a tool to calculate the word similarity. Then we manually apply the formula from (Mihalcea & al, 2006), to calculate the document similarity.

Reference

1. Kusner, M., Sun, Y., Kolkin, N., & Weinberger, K. (2015, June). From word embeddings to document distances. In *International conference on machine learning* (pp. 957-966).
2. Miller, G. A. (1995). WordNet: a lexical database for English. *Communications of the ACM*, *38*(11), 39-41.
3. Mihalcea, R., Corley, C., & Strapparava, C. (2006, July). Corpus-based and knowledge-based measures of text semantic similarity. In *Aaai* (Vol. 6, No. 2006, pp. 775-780).