1 잠재의미분석 (LSA)

1.1 실습 템플릿

```
class LSA :
  def __init__(self, doc_ls, topic_num):
    pass

  # tdm matrix 생성
  def TDM(self, doc_ls):
    pass

  # tdm matrix 특이값 분해(SVD)
  # U, s, Vt로 분해
  def SVD(self, tdm):
    pass

  # 토픽별 주요 키워드 출력
  def TopicModeling(self) :
    pass

  # 단어 벡터 행렬 생성 dot(U,s)
  def TermVectorMatrix(self, u, s):
    pass

  # 문서 벡터 행렬 생성 dot(s,Vt).T
  def DocVectorMatrix(self, s, vt):
    pass

  # 키워드를 입력했을 때 단어 벡터 반환
  def GetTermVector(self, term):
    pass

  # 문서를 입력했을 때 문서 벡터 반환
  def GetDocVector(self, doc):
    pass

  # 단어-문서 벡터 행렬 생성
  def TermDocVectorMatrix(self, u, s, vt):
    pass

  # 단어 벡터 행렬에서 단어 간 코사인 유사도 측정하여 행렬형태로 반환
```

```
  def TermSimilarityMatrix(self, term_vec_matrix):
    pass

  # 두개 단어를 입력했을 때 코사인 유사도 반환
  def GetTermSimilarity(self, term1, term2):
    pass

  # 문서 벡터 행렬에서 문서 간 코사인 유사도 측정하여 행렬형태로 반환
  def DocSimilarityMartrix(self, doc_vec_matrix):
    pass

  # 두개 문서를 입력했을 때 코사인 유사도 반환
  def GetDocSimilarity(self, doc1, doc2):
    pass
```

```
doc_ls = ['바나나 사과 포도 포도',
          '사과 포도',
          '포도 바나나',
          '짜장면 짬뽕 탕수육',
          '볶음밥 탕수육',
          '짜장면 짬뽕',
          '라면 스시',
          '스시',
          '가츠동 스시 소바',
          '된장찌개 김치찌개 김치',
          '김치 된장',
          '비빔밥 김치'
          ]

lsa = LSA(doc_ls, 3)
lsa.TopicModeling()
lsa.GetTermSimilarity('사과','바나나')
lsa.GetTermSimilarity('사과','짜장면')
lsa.GetDocSimilarity('사과 포도', '포도 바나나')
lsa.GetDocSimilarity('사과 포도', '라면 스시')
```

1.2 실습 예제 코드

```
import numpy as np
from collections import defaultdict
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.decomposition import randomized_svd

class LSA :
  def __init__(self, doc_ls, topic_num):
```

```
    self.doc_ls = doc_ls
    self.topic_num = topic_num
    self.term2idx, self.idx2term = self.toIdxDict(' '.join(doc_ls).split())
    self.doc2idx, self.idx2doc = self.toIdxDict(doc_ls)

    self.tdm = self.TDM(doc_ls)
    self.U, self.s, self.VT = self.SVD(self.tdm)

    #print(self.s)
    #print(self.U[:,:1].round(2))

    self.term_mat = self.TermVectorMatrix(self.U, self.s, topic_num)
    self.doc_mat = self.DocVectorMatrix(self.s, self.VT, topic_num)
    self.term_doc_mat = self.TermDocVectorMatrix(self.U, self.s, self.VT, topic_num)

    #print(self.term2idx.keys())
    #print(self.term_mat.round(2))

    self.term_sim = self.TermSimilarityMatrix(self.term_mat)
    self.doc_sim = self.DocSimilarityMartrix(self.doc_mat)

  # 리스트내 값을 index로 변환하는 dict과
  # index를 리스트내 값으로 변환하는 dict
  def toIdxDict(self, ls) :
    any2idx = defaultdict(lambda : len(any2idx))
    idx2any = defaultdict()

    for item in ls:
        any2idx[item]
        idx2any[any2idx[item]] = item

    return any2idx, idx2any

  def TDM(self, doc_ls):
    # 행(토큰크기), 열(문서갯수)로 TDM 생성
    tdm = np.zeros([len(self.term2idx.keys()), len(doc_ls)])

    for doc_idx, doc in enumerate(doc_ls) :
      for term in doc.split() :
        #등장한 단어를 dictionary에서 위치를 탐색하여 빈도수 세기
        tdm[self.term2idx[term], doc_idx] += 1

    return tdm

  # 특이값 분해
  def SVD(self, tdm):
    U, s, VT = randomized_svd(X,
                              n_components=15,
                              n_iter=5,
                              random_state=None)

```

```
    #U, s, VT = np.linalg.svd(tdm)
    return U, s, VT

  # 토픽별 주요 키워드 출력
  def TopicModeling(self, count = 3) :
    topic_num = self.topic_num

    for i in range(topic_num) :
      score = self.U[:,i:i+1].T
      sorted_index = np.flip(np.argsort(-score),0)

      a = []
      for j in sorted_index[0,: count] :
        a.append((self.idx2term[j], score[0,j].round(3)))

      print("Topic {} - {}".format(i+1,a ))

  def vectorSimilarity(self, matrix) :
    similarity = np.zeros([matrix.shape[1], matrix.shape[1]])

    for i in range(matrix.shape[1]) :
      for j in range(matrix.shape[1]) :
        similarity[i,j] =  cosine_similarity(matrix[:,i].T, matrix[:,j].T)

    return similarity

  # 키워드를 입력했을 때 단어 벡터 반환
  def GetTermVector(self, term):
    vec = self.term_mat[self.term2idx[term]:self.term2idx[term]+1,:]
    print('{} = {}'.format(term, vec))
    return vec

  # 문서를 입력했을 때 문서 벡터 반환
  def GetDocVector(self, doc):
    vec = self.doc_mat.T[self.doc2idx[doc]:self.doc2idx[doc]+1,:]
    print('{} = {}'.format(doc, vec))
    return vec

  def Compression(self, round_num=0) :
    print(self.tdm)
    print(self.term_doc_mat.round(round_num))

  def TermVectorMatrix(self, u, s, topic_num):
    term_mat = np.matrix(u[:, :topic_num])# * np.diag(s[:topic_num])
    return term_mat

  def DocVectorMatrix(self, s, vt, topic_num):
    doc_mat = np.diag(s[:topic_num]) * np.matrix(vt[:topic_num,:])
    return doc_mat

  def TermDocVectorMatrix(self, u, s, vt, topic_num):
```

```
    term_doc_mat = np.matrix(u[:, :topic_num]) * np.diag(s[:topic_num]) * np.matrix(vt[:topic_num,:])
    return term_doc_mat

  def TermSimilarityMatrix(self, termVectorMatrix):
    return self.vectorSimilarity(termVectorMatrix.T)

  def GetTermSimilarity(self, term1, term2):
    sim = self.term_sim[self.term2idx[term1], self.term2idx[term2]]
    print("({},{}) term similarity = {}".format(term1, term2, sim))
    return sim

  def DocSimilarityMartrix(self,docVectorMatrix):
    return self.vectorSimilarity(docVectorMatrix)

  def GetDocSimilarity(self, doc1, doc2):
    sim = self.doc_sim[self.doc2idx[doc1], self.doc2idx[doc2]]
    print("('{}','{}') doc similarity = {}".format(doc1, doc2, sim))
    return sim
```

```
doc_ls = ['바나나 사과 포도 포도 짜장면',
          '사과 포도',
          '포도 바나나',
          '짜장면 짬뽕 탕수육',
          '볶음밥 탕수육',
          '짜장면 짬뽕',
          '라면 스시',
          '스시 짜장면',
          '가츠동 스시 소바',
          '된장찌개 김치찌개 김치',
          '김치 된장 짜장면',
          '비빔밥 김치'
          ]

lsa = LSA(doc_ls, 4)
X = lsa.TDM(doc_ls)
print('== 토픽 모델링 ==')
lsa.TopicModeling(4)
print('₩n== 단어 벡터 ==')
lsa.GetTermVector('사과')
lsa.GetTermVector('짜장면')
print('₩n== 단어 유사도 ==')
lsa.GetTermSimilarity('사과','바나나')
lsa.GetTermSimilarity('사과','짜장면')
lsa.GetTermSimilarity('포도','짜장면')
lsa.GetTermSimilarity('사과','스시')
print('₩n== 문서 벡터 ==')
lsa.GetDocVector('사과 포도')
lsa.GetDocVector('짜장면 짬뽕')
print('₩n== 문서 유사도 ==')
```

```
lsa.GetDocSimilarity('사과 포도', '포도 바나나')
lsa.GetDocSimilarity('사과 포도', '라면 스시')
print('₩n== 토픽 차원수로 압축 ==')
lsa.Compression(0)
```

```
== 토픽 모델링 ==
Topic 1 - [('포도', 0.697), ('짜장면', 0.486), ('사과', 0.348), ('바나나', 0.348)]
Topic 2 - [('짜장면', 0.584), ('짬뽕', 0.356), ('김치', 0.337), ('스시', 0.256)]
Topic 3 - [('김치', 0.611), ('된장찌개', 0.264), ('김치찌개', 0.264), ('비빔밥', 0.185)]
Topic 4 - [('스시', 0.552), ('김치', 0.371), ('가츠동', 0.277), ('소바', 0.277)]

== 단어 벡터 ==
사과 = [[ 0.34843127 -0.19370961  0.01592593  0.03744775]]
짜장면 = [[ 0.48563449  0.58415588 -0.07468389 -0.18737521]]

== 단어 유사도 ==
(사과,바나나) term similarity = 1.0
(사과,짜장면) term similarity = 0.1519133521480699
(포도,짜장면) term similarity = 0.15191335214807006
(사과,스시) term similarity = -0.04097825202732752

== 문서 벡터 ==
사과 포도 = [[ 1.04529381 -0.58112882  0.04777778  0.11234324]]
짜장면 짬뽕 = [[ 0.61011838  0.93971158 -0.17760018 -0.53682795]]

== 문서 유사도 ==
('사과 포도','포도 바나나') doc similarity = 0.9999999999999998
('사과 포도','라면 스시') doc similarity = -0.03850688211350069

== 토픽 차원수로 압축 ==
[[1. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [2. 1. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
 [1. 0. 0. 1. 0. 1. 0. 1. 0. 0. 1. 0.]
 [0. 0. 0. 1. 0. 1. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 1. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 1. 1. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 1. 1.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
```

```
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
 [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]]
[[ 1.  0.  0. -0. -0.  0. -0.  0. -0. -0.  0. -0.]
 [ 1.  0.  0. -0. -0.  0. -0.  0. -0. -0.  0. -0.]
 [ 2.  1.  1. -0. -0.  0. -0.  0. -0. -0.  0. -0.]
 [ 1.  0.  0.  1.  0.  1.  0.  1.  0.  0.  1.  0.]
 [ 0. -0. -0.  1.  0.  1. -0.  0. -0. -0.  0. -0.]
 [ 0. -0. -0.  1.  0.  0. -0.  0. -0. -0.  0. -0.]
 [-0. -0. -0.  0.  0.  0. -0.  0. -0. -0.  0. -0.]
 [-0. -0. -0. -0. -0. -0.  0.  0.  0. -0. -0. -0.]
 [ 0. -0. -0.  0. -0.  0.  1.  1.  1. -0.  0. -0.]
 [-0. -0. -0. -0. -0. -0.  0.  0.  1. -0. -0. -0.]
 [-0. -0. -0. -0. -0. -0.  0.  0.  1. -0. -0. -0.]
 [-0. -0. -0. -0. -0. -0. -0. -0. -0.  0.  0.  0.]
 [-0. -0. -0. -0. -0. -0. -0. -0. -0.  0.  0.  0.]
 [ 0. -0. -0. -0. -0.  0. -0.  0. -0.  1.  1.  1.]
 [ 0. -0. -0.  0.  0.  0. -0.  0. -0.  0.  0.  0.]
 [-0. -0. -0. -0. -0. -0. -0. -0. -0.  0.  0.  0.]]
```

1.3 sklearn LSA 구현

```
doc_ls = ['바나나 사과 포도 포도',
          '사과 포도',
          '포도 바나나',
          '짜장면 짬뽕 탕수육',
          '볶음밥 탕수육',
          '짜장면 짬뽕',
          '라면 스시',
          '스시',
          '가츠동 스시 소바',
          '된장찌개 김치찌개 김치',
          '김치 된장',
          '비빔밥 김치'
          ]
```

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(
max_features= 1000, # 상위 1,000개의 단어를 보존
max_df = 0.5,
```

```
smooth_idf=True)

X = vectorizer.fit_transform(doc_ls)

from sklearn.decomposition import TruncatedSVD
svd_model = TruncatedSVD(n_components=4, algorithm='randomized', n_iter=100)
svd_model.fit(X)

np.shape(svd_model.components_)

terms = vectorizer.get_feature_names() # 단어 집합. 1,000개의 단어가 저장됨.

def get_topics(components, feature_names, n=3):
    for idx, topic in enumerate(components):
        print("Topic %d:" % (idx+1), [(feature_names[i], topic[i].round(5)) for i in topic.argsort()[:-n - 1:-1]])
get_topics(svd_model.components_,terms)
```

```
Topic 1: [('포도', 0.78069), ('사과', 0.44189), ('바나나', 0.44189)]
Topic 2: [('스시', 0.8864), ('라면', 0.33189), ('가츠동', 0.2282)]
Topic 3: [('짬뽕', 0.6258), ('짜장면', 0.6258), ('탕수육', 0.43614)]
Topic 4: [('김치', 0.76421), ('된장', 0.37119), ('비빔밥', 0.37119)]
```

2 LDA 실습

2.1 실습 템플릿

```
class LDA :
  def __init__(self, doc_ls, topic_num, alpha = 0.1, beta = 0.001):
    self.alpha = alpha
    self.beta = beta
    self.k = topic_num

  def RandomlyAssignTopic(self, doc_ls):
    pass


  def IterateAssignTopic(self) :
    pass


  # 토픽별 주요 키워드 출력
  def TopicModeling(self) :
```

```
    pass

```

2.2 실습 예제 코드

```
import nltk
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')
```

```
[nltk_data] Downloading package punkt to /root/nltk_data...
[nltk_data]   Package punkt is already up-to-date!
[nltk_data] Downloading package stopwords to /root/nltk_data...
[nltk_data]   Package stopwords is already up-to-date!
[nltk_data] Downloading package wordnet to /root/nltk_data...
[nltk_data]   Unzipping corpora/wordnet.zip.
True
```

```
import random
import numpy as np
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
from nltk.corpus import stopwords
from collections import defaultdict



class LDA :
  def __init__(self, docs, topic_num, alpha = 0.1, beta = 0.001):
    self.alpha = alpha
    self.beta = beta
    self.k = topic_num
    self.docs = docs


  def RandomlyAssignTopic(self, docs) :
    dic = defaultdict(dict)
    t2i = defaultdict(lambda : len(t2i))
    i2t = defaultdict()
    d = 0
    w = 0

    wnl = WordNetLemmatizer()
```

```
    stopword = stopwords.words('english')
    stopword.append(',')

    # 임의의 토픽을 할당
    for tokens in [word_tokenize(doc) for doc in docs] :
      for token in [wnl.lemmatize(token.lower()) for token in tokens if token not in stopword] :
        i2t[t2i[token]] = token
        dic[(d, t2i[token], w)] = random.randint(0,self.k-1)
        w += 1
      d += 1

    #print(dic)
    return dic, t2i, i2t


  def CountDocTopic(self) :
    docs = np.zeros((self.k, len(self.docs)))
    terms = np.zeros((self.k, len(self.t2i.keys())))


    #문서별 토큰별 빈도수 계산
    for (d, n, w) in self.term_topic.keys() :
      docs[self.term_topic[(d, n, w)], d] += 1 + self.alpha
      terms[self.term_topic[(d, n, w)], d] += 1 + self.beta
    #print(doc_m)

    #비어있는 값는 값에 alpha, beta 설정
    docs = np.where(docs==0.0, self.alpha, docs)
    terms = np.where(terms==0.0, self.beta, terms)

    return docs, terms


  def IterateAssignTopic(self, docs, terms) :
    #한개의 단어씩 주제 배정
    prev = {}

    while prev != self.term_topic:
      for (d, n, w) in self.term_topic.keys() :
        topic = [0, 0]

        docs[self.term_topic[(d, n, w)], d] -= (1 + self.alpha)
        terms[self.term_topic[(d, n, w)], n] -= (1 + self.beta)

        #print(doc_m)
        prev = self.term_topic

        for t in range(self.k) :
          p_t_d = docs[t, d]/docs[:,d:d+1].sum()
          p_w_t = terms[t, n]/terms[t:t+1,:].sum()
          prob = p_t_d * p_w_t
```

```
77
78            if topic[1] < prob :
79              topic = [t, prob]
80
81          #확률이 가장 큰 토픽을 할당
82          self.term_topic[(d, n, w)] = topic[0]
83          docs[self.term_topic[(d, n, w)], d] += (1 + self.alpha)
84          terms[self.term_topic[(d, n, w)], t] += (1 + self.beta)
85
86          #print(self.term_topic)
87
88      return terms
89
90
91    # 토픽별 주요 키워드 출력
92    def TopicModeling(self, count=5) :
93      self.term_topic, self.t2i, self.i2t = self.RandomlyAssignTopic(self.docs)
94      docs, terms = self.CountDocTopic()
95      terms = self.IterateAssignTopic(docs, terms)
96
97      score = terms / terms.sum(axis=1, keepdims=True)
98
99      for i in range(self.k) :
100       print("WnTopic {}".format(i+1))
101       sorted_index = np.flip(np.argsort(score[i]),0)[:count]
102       for j in sorted_index :
103         print("({}={})".format(self.i2t[j], score[i,j].round(3)), end = ' ')
104
105
```

```
1 doc_ls = ["Cute kitty",
2 "Eat rice or cake",
3 "Kitty and hamster",
4 "Eat bread",
5 "Rice, bread and cake",
6 "Cute hamster eats bread and cake"]
7
8 lda = LDA(doc_ls, 2)
9 lda.TopicModeling()
```

2.3 pyLDAvis 를 이용한 LDA 시각화

```
!pip install pyLDAvis
```

```
import pandas as pd
from sklearn.datasets import fetch_20newsgroups
dataset = fetch_20newsgroups(shuffle=True, random_state=1, remove=('headers', 'footers', 'quotes'))
documents = dataset.data
len(documents)
```

```
news_df = pd.DataFrame({'document':documents})
# 특수 문자 제거
news_df['clean_doc'] = news_df['document'].str.replace("[^a-zA-Z]", " ")
# 길이가 3이하인 단어는 제거 (길이가 짧은 단어 제거)
news_df['clean_doc'] = news_df['clean_doc'].apply(lambda x: ' '.join([w for w in x.split() if len(w)>3]))
# 전체 단어에 대한 소문자 변환
news_df['clean_doc'] = news_df['clean_doc'].apply(lambda x: x.lower())
```

```
import nltk
nltk.download('stopwords')
from nltk.corpus import stopwords
stop_words = stopwords.words('english') # NLTK로부터 불용어를 받아옵니다.
tokenized_doc = news_df['clean_doc'].apply(lambda x: x.split()) # 토큰화
tokenized_doc = tokenized_doc.apply(lambda x: [item for item in x if item not in stop_words])
```

```
from gensim import corpora
dictionary = corpora.Dictionary(tokenized_doc)
corpus = [dictionary.doc2bow(text) for text in tokenized_doc]
print(corpus[1]) # 수행된 결과에서 두번째 뉴스 출력. 첫번째 문서의 인덱스는 0
```

```
import gensim
NUM_TOPICS = 20 #20개의 토픽, k=20
ldamodel = gensim.models.ldamodel.LdaModel(corpus, num_topics = NUM_TOPICS, id2word=dictionary, passes=15)
topics = ldamodel.print_topics(num_words=4)
for topic in topics:
```

```
    print(topic)
```

```
import pyLDAvis.gensim
pyLDAvis.enable_notebook()
vis = pyLDAvis.gensim.prepare(ldamodel, corpus, dictionary)
pyLDAvis.display(vis)
```

```
for i, topic_list in enumerate(ldamodel[corpus]):
    if i==5:
        break
    print(i,'번째 문서의 topic 비율은',topic_list)
```

```
def make_topictable_per_doc(ldamodel, corpus, texts):
    topic_table = pd.DataFrame()

    # 몇 번째 문서인지를 의미하는 문서 번호와 해당 문서의 토픽 비중을 한 줄씩 꺼내온다.
    for i, topic_list in enumerate(ldamodel[corpus]):
        doc = topic_list[0] if ldamodel.per_word_topics else topic_list
        doc = sorted(doc, key=lambda x: (x[1]), reverse=True)
        # 각 문서에 대해서 비중이 높은 토픽순으로 토픽을 정렬한다.
        # EX) 정렬 전 0번 문서 : (2번 토픽, 48.5%), (8번 토픽, 25%), (10번 토픽, 5%), (12번 토픽, 21.5%),
        # Ex) 정렬 후 0번 문서 : (2번 토픽, 48.5%), (8번 토픽, 25%), (12번 토픽, 21.5%), (10번 토픽, 5%)
        # 48 > 25 > 21 > 5 순으로 정렬이 된 것.

        # 모든 문서에 대해서 각각 아래를 수행
        for j, (topic_num, prop_topic) in enumerate(doc): #  몇 번 토픽인지와 비중을 나눠서 저장한다.
            if j == 0:  # 정렬을 한 상태이므로 가장 앞에 있는 것이 가장 비중이 높은 토픽
                topic_table = topic_table.append(pd.Series([int(topic_num), round(prop_topic,4), topic_list]), ignore_index=True)
                # 가장 비중이 높은 토픽과, 가장 비중이 높은 토픽의 비중과, 전체 토픽의 비중을 저장한다.
            else:
                break
    return(topic_table)
```

```
topictable = make_topictable_per_doc(ldamodel, corpus, tokenized_doc)
topictable = topictable.reset_index() # 문서 번호을 의미하는 열(column)로 사용하기 위해서 인덱스 열을 하나 더 만든다.
topictable.columns = ['문서 번호', '가장 비중이 높은 토픽', '가장 높은 토픽의 비중', '각 토픽의 비중']
topictable[:10]
```

	문서 번호	가장 비중이 높은 토픽	가장 높은 토픽의 비중	각 토픽의 비중
0	0	17.0	0.6845	[(8, 0.30094197), (17, 0.6845419)]
1	1	8.0	0.5632	[(2, 0.029335152), (8, 0.5632338), (10, 0.1083...
2	2	8.0	0.6275	[(4, 0.018661937), (8, 0.62747276), (16, 0.045...
3	3	19.0	0.2927	[(1, 0.044962943), (8, 0.14445594), (9, 0.0294...
4	4	10.0	0.6536	[(2, 0.082709715), (5, 0.23224762), (10, 0.653...
5	5	8.0	0.6517	[(0, 0.105608456), (3, 0.15081134), (8, 0.6517...
6	6	17.0	0.7437	[(2, 0.013339129), (7, 0.012894245), (8, 0.014...
7	7	8.0	0.4894	[(8, 0.48941994), (10, 0.12249057), (13, 0.024...
8	8	3.0	0.3653	[(0, 0.2194796), (3, 0.36532196), (8, 0.220569...
9	9	19.0	0.4796	[(0, 0.038544882), (4, 0.04621158), (6, 0.0463...