1. The dataset selected is enwikivoyage-20180801-pages-meta-current.xml.bz2, download from https://dumps.wikimedia.org/enwikivoyage/ and enwikinews-20180801-pages-articles-multistream.xml.bz2 from https://dumps.wikimedia.org/enwikinews/. The file is 120 MB and 44.5MB, and should be extracted by WikiExtractor.

```
In [7]: import findspark
        findspark.init()
        from pyspark.sql import SparkSession
        from pyspark import SparkContext
        from pyspark.sql import SQLContext
        from pyspark.ml.feature import Tokenizer, RegexTokenizer
        from pyspark.sql.functions import col, udf
        from pyspark.sql.types import *
        from pyspark.ml.feature import StopWordsRemover
        from pyspark.ml.feature import NGram
        from pyspark.ml.feature import Word2Vec
        from pyspark.ml.feature import CountVectorizer
        from pyspark.ml.feature import HashingTF, IDF
        spark = SparkSession.builder.appName("clustering").getOrCreate()
        import pandas as pd
```

```
In [8]: df0 = spark.read.format('json').load('/home/hz2558/wiki_00')
        df1 = spark.read.format('json').load('/home/hz2558/wiki_01')
        pd_final = df1.toPandas().append(df0.toPandas())

        sc = spark.sparkContext
        sqlContest = SQLContext(sc)
        df = sqlContest.createDataFrame(pd_final)
```

```
In [16]: regexTokenizer = RegexTokenizer(inputCol="text", outputCol="words", pattern="[^A-Za-z]+", /
                                         toLowercase=True)
         tokenized_data = regexTokenizer.transform(df)
         stopWordsRemover = StopWordsRemover(inputCol="words", outputCol="filtered_words")
         filtered_data = stopWordsRemover.transform(tokenized_data)
         hashingTF = HashingTF(inputCol="filtered_words", outputCol="raw_features", numFeatures=20)
         featurizedData = hashingTF.transform(filtered_data)
         idf= IDF(inputCol="raw_features", outputCol="features")
         idfModel = idf.fit(featurizedData)
         featurized_data = idfModel.transform(featurizedData)
         data = featurized_data.drop('id','text','title','url','words','filtered_words','raw_features')
```

We first need to process dataset. We choose two files from different dataset. Then we use pandas' function to merge the dataset. Then we will token and filter the data, and then convert the data frames to feature vectors, then normalize the feature vector with the document frequencies. Finally, we use TF-IDF to get the featurized data. Then we can deal with the featurized vector and do clustering.

K-means:

```
In [20]: from pyspark.ml.clustering import KMeans
kmeans = KMeans(featuresCol='features', predictionCol='prediction', k=2, maxIter=20, seed=1)
model = kmeans.fit(data)
result = model.transform(data)
wssse = model.computeCost(data)
print("Within Set Sum of Squared Errors = " + str(wssse))
centers = model.clusterCenters()
print("Cluster Centers:")
for center in centers:
    print(center)

print('\n')
pd_result = result.drop('label','features').toPandas()
sum1 = pd_result[0:700].apply(lambda x: x.sum(), axis=0)
sum2 = pd_result[700:1270].apply(lambda x: x.sum(), axis=0)

d = {'predict_0': [0,0],'predict_1':[0,0], 'total':[700, 570],
    'Title'  : ["truth=0", "truth=1"]}
confusion_matrix = pd.DataFrame(d)
confusion_matrix.index = d["Title"]
del confusion_matrix["Title"]

confusion_matrix['predict_0'][0] = 700-sum1
confusion_matrix['predict_1'][0] = sum1
confusion_matrix['predict_0'][1] = 570-sum2
confusion_matrix['predict_1'][1] = sum2
accuracy = (920-sum1+sum2)/1270
print(accuracy,'\n')
confusion
```

```
Within Set Sum of Squared Errors = 3719.6484232729395
Cluster Centers:
[1.41212113 1.12848131 1.20022036 1.36905056 1.37112747 0.91710821
 1.31798469 1.11116542 1.24159345 0.84476023 1.3690782  0.90955896
 1.48364542 0.81802587 1.12855602 0.94117039 1.54094246 1.3803233
 1.3237269  1.43767311]
[0.4110819  0.35480866 0.35002986 0.4369304  0.42484339 0.28804524
 0.41650936 0.36108054 0.39851917 0.25899716 0.40798446 0.30604195
 0.45086866 0.2563696  0.35324833 0.31650382 0.40609945 0.39749072
 0.42795263 0.42144812]


prediction    0.651181
dtype: float64
```

	predict_0	predict_1	total
truth=0	91	609	700
truth=1	54	516	570

With K-means, we divide the merged the dataset into two clusters.
The within set sum of squared errors is around 3700, which shows that the two dataset is clustered properly.
The datasets we first imported are 700 and 570 items separately. Then we sum the result with result[0:700] and result[700:1270] separately. Then we compared it with the predictions. For example, if A = [1:30, 0:20], predict_A = [1:20, 0:30]. Then the number of "1" is sum(A) = 30, and the "0" is 50 – 30 = 20; the predict_1 is 20, and predict_0 is 30. So the accuracy is (number of correct "1" + number of correct "1") /total.

Bisecting Kmeans

```
from pyspark.ml.clustering import BisectingKMeans
bkmeans = BisectingKMeans().setK(2).setSeed(1)
model = bkmeans.fit(data)
result = model.transform(data)
cost = model.computeCost(data)
print("Within Set Sum of Squared Errors = " + str(cost))


print("Cluster Centers: ")
centers = model.clusterCenters()
for center in centers:
    print(center)

print('\n')
pd_result = result.drop('label','features').toPandas()
sum1 = pd_result[0:700].apply(lambda x: x.sum(), axis=0)
sum2 = pd_result[700:1270].apply(lambda x: x.sum(), axis=0)

d = {'predict_0': [0,0],'predict_1':[0,0], 'total':[700, 570],
    'Title'  : ["truth=0", "truth=1"]}
confusion = pd.DataFrame(d)
confusion.index = d["Title"]
del confusion["Title"]

confusion['predict_0'][0] = 700-sum1
confusion['predict_1'][0] = sum1
confusion['predict_0'][1] = 570-sum2
confusion['predict_1'][1] = sum2
accuracy = (920-sum1+sum2)/1270
print(accuracy,'\n')
confusion
```

```
Within Set Sum of Squared Errors = 3719.6484232729395
Cluster Centers:
[0.4110819  0.35480866 0.35002986 0.4369304  0.42484339 0.28804524
 0.41650936 0.36108054 0.39851917 0.25899716 0.40798446 0.30604195
 0.45086866 0.2563696  0.35324833 0.31650382 0.40609945 0.39749072
 0.42795263 0.42144812]
[1.41212113 1.12848131 1.20022036 1.36905056 1.37112747 0.91710821
 1.31798469 1.11116542 1.24159345 0.84476023 1.3690782  0.90955896
 1.48364542 0.81802587 1.12855602 0.94117039 1.54094246 1.3803233
 1.3237269  1.43767311]


prediction    0.695276
dtype: float64
```

	predict_0	predict_1	total
truth=0	609	91	700
truth=1	516	54	570

With Bisecting Kmeans, we divide the merged the dataset into two clusters.
The within set sum of squared errors is around 3700, which shows that the two dataset is clustered properly.
We use the method shown above to show the confusion matrix and accuracy.

Gaussian Mixture

In [24]:
```
from pyspark.ml.clustering import GaussianMixture

gmm = GaussianMixture().setK(2).setSeed(1)
model = gmm.fit(data)

result = model.transform(data)
summary = model.summary
print('\n')

pd_result = result.select('prediction').toPandas()
sum1 = pd_result[0:700].apply(lambda x: x.sum(), axis=0)
sum2 = pd_result[700:1270].apply(lambda x: x.sum(), axis=0)

d = {'predict_0': [0,0],'predict_1':[0,0], 'total':[700, 570],
    'Title'  : ["truth=0", "truth=1"]}
confusion_matrix = pd.DataFrame(d)
confusion_matrix.index = d["Title"]
del confusion_matrix["Title"]

confusion['predict_0'][0] = 700-sum1
confusion['predict_1'][0] = sum1
confusion['predict_0'][1] = 570-sum2
confusion['predict_1'][1] = sum2
accuracy = (920-sum1+sum2)/1270
print(accuracy,'\n')
confusion
```

```
prediction    0.682677
dtype: float64
```

Out[24]:

	predict_0	predict_1	total
truth=0	194	506	700
truth=1	117	453	570

Performance:
Within all three algorithms, the prediction is not very well, and the Gaussian Mixture seems to have a better accuracy.
As can be seen, the accuracy is not very great. I think there are two reasons: the first is that the dataset we select is not labeled perfectly; secondly, the data featurized is not properly.

2. TCP

In [1]:
```python
import findspark
findspark.init()
import socket
import sys
import requests
import requests_oauthlib
import json
from tweepy import Stream
from tweepy import OAuthHandler
from tweepy.streaming import StreamListener
```

In [2]:
```python
CONSUMER_KEY = "R3iYOM47n2TD4uNNIafuO6pM3"
CONSUMER_SECRET = "WDbV7DAED63sdveK0rezpBnTvS7N08YcAUvM0Zx7sARuyAA0Aj"
ACCESS_TOKEN = "1051887802391154689-rVPND8LnALxFyxQlBDi8IjoPid23aa"
ACCESS_TOKEN_SECRET = "3YSP9XKIu5bixbW6LxkfnbKOWt2Cx4tlz7aYHSS5YMoyN"
```

In [3]:
```python
class listener(StreamListener):
    def on_data(self, data):
        print(data)
        return (True)
    def on_error(self, status):
        print(status)
```

In [5]:
```python
my_auth = requests_oauthlib.OAuth1(CONSUMER_KEY, CONSUMER_SECRET, ACCESS_TOKEN, ACCESS_TOKEN_SECRET)
def get_tweets():
    url = 'https://stream.twitter.com/1.1/statuses/filter.json'
    query_data = [('language','en'),('track','the','love', 'dog', 'trump','movie', 'music', 'the', 'to', 'for', 'a'
    query_url = url + '?' + '&'.join([str(t[0]) + '=' + str(t[1]) for t in query_data])
    response = requests.get(query_url, auth=my_auth, stream=True)
    print(query_url, response)
    return response
```

In [6]:
```python
def send_tweets_to_spark(http_resp, tcp_connection):
    for line in http_resp.iter_lines():
        try:
            full_tweet = json.loads(line)
            tweet_text = full_tweet['text']
            print(tweet_text)
            tcp_connection.send((tweet_text + '\n').encode())
        except:
            e = sys.exc_info()[0]
            print(e)
```

In [ ]:
```python
TCP_IP = 'localhost'
TCP_PORT = 9009
conn = None
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((TCP_IP, TCP_PORT))
s.listen(1)
print("Waiting for TCP connection...")
conn, addr = s.accept()
print("Connected...Starting getting tweets.")
resp = get_tweets()
send_tweets_to_spark(resp, conn)
```

```
Waiting for TCP connection...
Connected...Starting getting tweets.
```

In [1]:
```python
import findspark
findspark.init()
from pyspark import SparkConf, SparkContext
from pyspark.streaming import StreamingContext
from pyspark.sql import Row,SQLContext
import sys
import requests
```

In [2]:
```python
conf = SparkConf()
conf.setMaster('local[2]')
conf.setAppName("TwitterStreamApp1")
sc = SparkContext(conf=conf)
sc.setLogLevel("Error")
ssc = StreamingContext(sc, 1)
ssc.checkpoint("checkpoint_TwitterApp")
dataStream = ssc.socketTextStream("localhost",9009)

def sum_tags_count(new_values,total_sum):
    return sum(new_values)+(total_sum or 0)
def get_sql_context_instance(spark_context):
    if('sqlContextSingletonInstance' not in globals()):
        globals()['sqlContextSingletonInstance'] = SQLContext(spark_context)
    return globals()['sqlContextSingletonInstance']
def process_rdd(time,rdd):
    print("-----------%s-----------"%str(time))
    try:
        sql_context=get_sql_context_instance(rdd.context)
        row_rdd=rdd.map(lambda w:Row(hashtag=w[0],hashtag_count=w[1]))
        hashtags_df = sql_context.createDataFrame(row_rdd)
        hashtags_df.registerTempTable("hashtags")
        hashtag_counts_df = sql_context.sql("select hashtag, hashtag_count from hashtags where\
                                            hashtag in ('#movie', '#Trump','#game', '#girl', \
                                            '#art', '#dog', '#Dota','#cs', '#music', '#class')")
        hashtag_counts_df.show()
    except:
        e=sys.exc_info()[0]
        print(e)
```

In [7]:
```python
import findspark
findspark.init()
from pyspark.sql import SparkSession
from pyspark import SparkContext
from pyspark.sql import SQLContext
from pyspark.ml.feature import Tokenizer, RegexTokenizer
from pyspark.sql.functions import col, udf
from pyspark.sql.types import *
from pyspark.ml.feature import StopWordsRemover
from pyspark.ml.feature import NGram
from pyspark.ml.feature import Word2Vec
from pyspark.ml.feature import CountVectorizer
from pyspark.ml.feature import HashingTF, IDF
spark = SparkSession.builder.appName("clustering").getOrCreate()
import pandas as pd
```

In [8]:
```python
df0 = spark.read.format('json').load('/home/hz2558/wiki_00')
df1 = spark.read.format('json').load('/home/hz2558/wiki_01')
pd_final = df1.toPandas().append(df0.toPandas())

sc = spark.sparkContext
sqlContest = SQLContext(sc)
df = sqlContest.createDataFrame(pd_final)
```

In [16]:
```python
regexTokenizer = RegexTokenizer(inputCol="text", outputCol="words", pattern="[^A-Za-z]+", /
                                toLowercase=True)
tokenized_data = regexTokenizer.transform(df)
stopWordsRemover = StopWordsRemover(inputCol="words", outputCol="filtered_words")
filtered_data = stopWordsRemover.transform(tokenized_data)
hashingTF = HashingTF(inputCol="filtered_words", outputCol="raw_features", numFeatures=20)
featurizedData = hashingTF.transform(filtered_data)
idf= IDF(inputCol="raw_features", outputCol="features")
idfModel = idf.fit(featurizedData)
featurized_data = idfModel.transform(featurizedData)
data = featurized_data.drop('id','text','title','url','words','filtered_words','raw_features')
```

In [ ]:
```python
words = dataStream.flatMap(lambda line: line.split(" "))
hashtags = words.filter(lambda w: '#' in w).map(lambda x:(x, 1))
tags_totals = hashtags.updateStateByKey(sum_tags_count)
tags_totals.foreachRDD(process_rdd)
ssc.start()
ssc.awaitTermination()
```

```
-----------2018-10-20 18:30:46-----------
+-------+------------+
|hashtag|hashtag_count|
+-------+------------+
+-------+------------+
```

```
-----------2018-10-20 21:56:15-----------
+-------+------------+
|hashtag|hashtag_count|
+-------+------------+
|   #art|          47|
| #music|          51|
|   #dog|          24|
| #movie|           6|
| #class|           1|
| #Trump|         219|
|  #girl|           2|
|  #game|           5|
+-------+------------+
```

The keywords we track is [the, love, dog, trump, movie, music, the, to for, a]
The tags we set is [movie, Trump, dog, music, art, class, girl, game, cs, Dota].
After 3h data processing, the result, shown that: the Trump topic is the most popular, which means more tweeters are concerning about it
And there are almost no people concern about cs and Dota topic.

3.

```
In [3]: import findspark
        findspark.init()
        from pyspark.sql import SparkSession
        from pyspark import SparkContext
        from pyspark.sql import SQLContext
        from pyspark.ml.feature import Tokenizer, RegexTokenizer
        from pyspark.sql.functions import col, udf
        from pyspark.sql.types import *
        from pyspark.ml.feature import StopWordsRemover
        from pyspark.ml.feature import NGram
        from pyspark.ml.feature import Word2Vec
        from pyspark.ml.feature import CountVectorizer
        from pyspark.ml.feature import HashingTF, IDF
        import pandas as pd
        from pyspark.sql.functions import lit
```

```
In [4]: spark = SparkSession.builder.appName("graph").getOrCreate()
        sqlContext = SQLContext(spark)
        df0 = spark.read.format('json').load('/home/hz2558/wiki_00')
        df1 = spark.read.format('json').load('/home/hz2558/wiki_01')

        pd0 = df0.toPandas()
        pd1 = df1.toPandas()
        pd_final = pd1.append(pd0)

        sc = spark.sparkContext
        sqlContest = SQLContext(sc)
        df = sqlContest.createDataFrame(pd_final)

        regexTokenizer = RegexTokenizer(inputCol="text", outputCol="words", pattern="[^A-Za-z]+", /
                                        toLowercase=True)
        tokenized_data = regexTokenizer.transform(df)

        stopWordsRemover = StopWordsRemover(inputCol="words", outputCol="filtered_words")
        filtered_data = stopWordsRemover.transform(tokenized_data)

        hashingTF = HashingTF(inputCol="filtered_words", outputCol="raw_features", numFeatures=20)
        featurized_data = hashingTF.transform(filtered_data)

        idf= IDF(inputCol="raw_features", outputCol="features")
        idfModel = idf.fit(featurized_data)
        featurized_data = idfModel.transform(featurized_data)
        data = featurized_data.select('id','features')
```

```
In [5]: import  pyspark.sql.functions as psf
        from pyspark.sql.types import DoubleType
        from math import sqrt

        dot_udf = psf.udf(lambda x,y:float(x.dot(y)/sqrt(x.dot(x) * y.dot(y))),DoubleType())
        s = data.alias("data1").join(data.alias("data2"),psf.col("data1.id") != psf.col("data2.id"))/
        .select(
                psf.col("data1.id").alias("data1"),
                psf.col("data2.id").alias("data2"),
                dot_udf("data1.features","data2.features").alias("similarity")).sort("data1","data2")
```

In the Q3, we also use IDF to feature the data imported.
Then we will use cosine to calculate the similarity. As we know, cos(u, v) = dot(u, v)/(|u||v|). When the two feature vector's cosine is closer 1, it means that the angle between two vectors is smaller, which means the difference between two vectors is smaller.
The vectors we select is data1, data2 from dataset where data1 != data2. The thing we need to pay attention is that we don't need the calculate the similarity between one vector and itself.

In [6]:
```
edges = s.toPandas()
edges
```

Out[6]:

	data1	data2	similarity
0	1002	1003	0.484693
1	1002	1027	0.500629
2	1002	1037	0.424129
3	1002	1038	0.594568
4	1002	1041	0.566332
5	1002	1053	0.417231
6	1002	1054	0.504772
7	1002	1057	0.659068
8	1002	1063	0.428603
9	1002	1073	0.410321
10	1002	1083	0.588974
11	1002	1085	0.478652
12	1002	1087	0.575392
13	1002	1089	0.396870
14	1002	1095	0.591747
15	1002	1103	0.466609
16	1002	1107	0.371832
17	1002	1116	0.484639
18	1002	1117	0.526477
19	1002	1126	0.552883
20	1002	1141	0.364078
21	1002	1153	0.467767
22	1002	1165	0.496586

In [7]:
```
node = data.toPandas()
node
```

Out[7]:

	id	features
0	3	(0.0, 0.07344245499209183, 0.0, 0.0, 0.0, 0.0,...
1	736	(0.44065472995255095, 0.22032736497627547, 0.4...
2	741	(0.22032736497627547, 0.07344245499209183, 0.6...
3	743	(0.8078670049130101, 0.5875396399367346, 0.270...
4	764	(0.14688490998418366, 0.0, 0.0, 0.0, 0.0751373...
5	779	(0.36721227496045916, 0.2937698199683673, 0.33...
6	783	(0.2937698199683673, 0.07344245499209183, 0.20...
7	797	(0.07344245499209183, 0.0, 0.0675327712668638,...
8	798	(0.14688490998418366, 0.5140971849446428, 0.0,...
9	807	(0.0, 0.14688490998418366, 0.1350655425337276,...
10	813	(0.07344245499209183, 0.2937698199683673, 0.0,...
11	814	(0.5140971849446428, 0.22032736497627547, 0.13...
12	817	(0.7344245499209183, 0.14688490998418366, 0.60...
13	820	(0.36721227496045916, 1.0281943698892857, 0.33...
14	822	(0.6609820949288264, 0.2937698199683673, 0.337...
15	841	(0.36721227496045916, 0.5140971849446428, 0.27...
16	848	(0.5140971849446428, 0.14688490998418366, 0.06...
17	849	(0.22032736497627547, 0.22032736497627547, 0.5...
18	856	(0.07344245499209183, 0.0, 0.0, 0.080239422535...
19	862	(0.2937698199683673, 0.22032736497627547, 0.13...
20	865	(0.2937698199683673, 0.2937698199683673, 0.270...
21	871	(0.9547519148971938, 0.36721227496045916, 0.60...
22	876	(0.14688490998418366, 0.07344245499209183, 0.0...

In [8]:
```
file1="/home/hz2558/vfile.csv"
node.to_csv(path_or_buf=file1, header=False, sep=",")
file2 = "/home/hz2558/efile.csv"
edges.to_csv(path_or_buf=file2,header = False,sep=",")
```

we will import edges as edge file, and the node as vertice file.

```
>>> import findspark
>>> findspark.init()
>>> from pyspark.sql import SparkSession
>>> from pyspark import SparkContext
>>> SparkSession.builder.appName("Graph").getOrCreate()
<pyspark.sql.session.SparkSession object at 0x7f1880022b00>
>>> spark = SparkSession.builder.appName("Graph").getOrCreate()
>>> V = spark.read.csv("/home/hz2558/vfile.csv")
2018-10-20 00:45:04 WARN  ObjectStore:568 - Failed to get database global_temp, returning NoS
uchObjectException
>>> e = E.select(E._c1,E._c2,E._c3.cast("float")).selectExpr("_c1 as src","_c2 as dst","_c3 as simil
arity")
>>> from graphframes import *
>>> e = GraphFrame(v, e).edges.filter("similarity > 0.8")
>>> g = GraphFrame(v, e)
```

First, we will initialize the graph with the edge file and vertices file we get from Q1. Then, the similarity threshold we set is 0.8, so we filter the edges with similarity below 0.8 using edges = g.edges.filter("similarity > 0.8")

Vertices

```
>>> g.vertices.show()
+---+--------------------+
| id|            features|
+---+--------------------+
|  3|(20,[1,7,17],[0.0...|
|736|(20,[0,1,2,3,4,5,...|
|741|(20,[0,1,2,3,4,5,...|
|743|(20,[0,1,2,3,4,5,...|
|764|(20,[0,4,6,9,11,1...|
|779|(20,[0,1,2,4,5,6,...|
|783|(20,[0,1,2,3,4,5,...|
|797|(20,[0,2,3,4,5,6,...|
|798|(20,[0,1,5,6,7,8,...|
|807|(20,[1,2,4,6,7,10...|
|813|(20,[0,1,3,4,5,6,...|
|814|(20,[0,1,2,3,4,5,...|
|817|(20,[0,1,2,3,4,5,...|
|820|(20,[0,1,2,3,4,5,...|
|822|(20,[0,1,2,3,4,5,...|
|841|(20,[0,1,2,3,4,5,...|
|848|(20,[0,1,2,4,5,8,...|
|849|(20,[0,1,2,3,4,5,...|
|856|(20,[0,3,4,5,6,8,...|
|862|(20,[0,1,2,3,4,5,...|
+---+--------------------+
only showing top 20 rows
```

Edges:

```
>>> g.edges.show()
+----+----+----------+
| src| dst|similarity|
+----+----+----------+
|1002|5927| 0.8364375|
|1003|1168| 0.8392109|
|1003|1587| 0.8404078|
|1003|1836|0.84692985|
|1003|1905| 0.8390833|
|1003|2300|0.86443603|
|1003|2521|0.83688617|
|1003|2557| 0.8254175|
|1003|2639|0.90549123|
|1003|2962|0.86111134|
|1003|3024| 0.8605593|
|1003|3199| 0.8316683|
|1003|3209|0.82829756|
|1003|3287|0.82308304|
|1003|3304|0.82253605|
|1003|3407| 0.8893002|
|1003|3521|0.82845545|
|1003|3531|0.83176696|
|1003|3693|0.82349986|
|1003|3694|0.83491087|
+----+----+----------+
only showing top 20 rows
```

Indegrees:

```
>>> vertexInDegrees = g.inDegrees
>>> vertexInDegrees.show()
+----+--------+
|  id|inDegree|
+----+--------+
|4821|     540|
|6194|     602|
|3414|     312|
|5645|     698|
|2294|     501|
|7273|     145|
|7362|     468|
|2756|     403|
|3858|     137|
|5149|     238|
|3826|     125|
|7655|      24|
|1265|     777|
|5316|     782|
|6900|     704|
|7056|     456|
|3949|     594|
|2898|     207|
|5297|     635|
|1953|     774|
+----+--------+
only showing top 20 rows
```

PageRank:

```
>>> pagerank.vertices.select("id", "pagerank").show()
+----+-------------------+
|  id|           pagerank|
+----+-------------------+
|6806| 0.5572868370610826|
|4631| 1.3372911478092304|
|2521| 0.9674671831487004|
|3417| 0.6283629744997401|
|1925|0.17248499987140542|
|6539| 0.4380495552274247|
|4439| 1.0666850032625628|
|3749| 0.9534939260201519|
|7216| 0.3933974956141231|
|7773|  1.252219298591858|
|7280| 1.8032407028354935|
|4088| 1.8638550948503734|
|4198| 1.7959968295074444|
|1238|  0.866118889148295|
|4331| 1.4843086646533585|
|1255| 1.7089559993971466|
|1631| 1.9406903465364975|
|1965| 0.2655632938620651|
|3960| 0.5354272310602128|
|5651| 1.9404992440634001|
+----+-------------------+
only showing top 20 rows
```

The pagerank is in the rank from 0.5 to 1.9. If the node's pagerank is high, it means that the node's similar to more data.

Source, Distance and Weight

```
>>> pagerank.edges.select("src", "dst", "weight").show()
+----+----+--------------------+
| src| dst|              weight|
+----+----+--------------------+
|2294|4951|0.001996007984031936|
|2294|5793|0.001996007984031936|
|2294|7470|0.001996007984031936|
|4821|2765|0.001851851851851852|
|4821|5926|0.001851851851851852|
|4821|7833|0.001851851851851852|
|4821|5639|0.001851851851851852|
|5645|2294|0.001432664756446...|
|5645|5686|0.001432664756446...|
|5645|6740|0.001432664756446...|
|6194|5327|0.001661129568106...|
|7273| 903|0.006896551724137931|
|2756|7361|0.002481389578163...|
|2756|6641|0.002481389578163...|
|2756|6415|0.002481389578163...|
|7362|7219|0.002136752136752137|
|7362|5263|0.002136752136752137|
|1265|4198|0.001287001287001287|
|1265|7669|0.001287001287001287|
|1265|2406|0.001287001287001287|
+----+----+--------------------+
only showing top 20 rows
```

Connected Componets

```
>>> connected.select("id", "component").show()
+---+-------------+
| id|    component|
+---+-------------+
|  3| 154618822657|
|736|            0|
|741|            0|
|743|            0|
|764|1477468749830|
|779|            0|
|783|            0|
|797|            0|
|798|            0|
|807|            0|
|813|            0|
|814|            0|
|817|            0|
|820|            0|
|822|            0|
|841|            0|
|848|            0|
|849|            0|
|856|            0|
|862|            0|
+---+-------------+
only showing top 20 rows
```

In the result above, we can see that many nodes are in the same connected components This means that the connectivity of the graph is great. For we set the similarity threshold is 0.8, which is high enough do judge the similarity. And the graph is connected well, it means the data in the dataset is similar to each other.

Triangle Counts.

id	count
2294	81399
3414	34993
4821	92953
5645	139839
6194	112653
7273	8146
2756	53809
3826	5730
3858	7074
5149	21527
7362	72779
7655	192
1265	160592
2898	15360
3949	108476
5316	165813
6900	143106
7056	67450
1953	164037
4236	5342

only showing top 20 rows

We found that the dataset' similarity is great.
As shown in the connected components, the subtopic will be separated from the main component, which means the subtopic is not very similar to the main.