Homework 1: Finding Similar Items: Textually Similar Documents

Description:

You are to implement the stages of finding textually similar documents based on Jaccard similarity using the shingling, minhashing, and locality-sensitive hashing (LSH) techniques and corresponding algorithms. The implementation can be done using any big data processing framework, such as Apache Spark, Apache Flink, or no framework, e.g., in Java, Python, etc. To test and evaluate your implementation, write a program that uses your implementation to find similar documents in a corpus of 5-10 or more documents, such as web pages or emails.

The stages should be implemented as a collection of classes, modules, functions, or procedures depending on the framework and the language of your choice. Below, we describe sample classes implementing different stages of finding textually similar documents. You do not have to develop the exact same classes and data types described below. Feel free to use data structures that suit you best.

- A class Shingling that constructs k–shingles of a given length k (e.g., 10) from a given document, computes a hash value for each unique shingle and represents the document in the form of an ordered set of its hashed k-shingles.
- A class CompareSets computes the Jaccard similarity of two sets of integers – two sets of hashed shingles.
- A class MinHashing that builds a minHash signature (in the form of a vector or a set) of a given length n from a given set of integers (a set of hashed shingles).
- A class CompareSignatures estimates the similarity of two integer vectors – minhash signatures – as a fraction of components in which they agree. (Optional task for extra 2 bonus points) A class LSH that implements the LSH technique: given a collection of minhash signatures (integer vectors) and a similarity threshold t, the LSH class (using banding and hashing) finds candidate pairs of signatures agreeing on at least a fraction t of their components. To test and evaluate your implementation's scalability (the execution time versus the size of the input dataset), write a program that uses your classes to find similar documents in a corpus of 5-10 documents. Choose a similarity threshold s (e.g., 0,8) that states that two documents are similar if the Jaccard similarity of their shingle sets is at least s.

Program Introduction

I extracted five emails to form test data for our program. I create all of the methods mentioned above as functions below. For the report, I have explained each algorithm in the beginning of them.

Libararies

Here are some libraries that I used in the program.

- Using "numpy" as our data format.
- Using "itertools" for iteration in the function "LSH".

- Using "collections" to computes and merges the same elements in one list in the function "LSH".
- Using "time" to collect program run time.

In [ ]:
```
import numpy as np
import pandas as pd
import itertools
from collections import Counter
import time
```

Shingling

Shingling is used to transform document into a set that contains tokens in the document. For example, a document "using numpy" will be divide into tokens {us si in ng gn .. py} with k-shingling size of 2

Function "shingling" in our program will first replace all the spaces and then divide the document into tokens by given value k.

Function "shinglingHash" transform sets that created from shingling into hash by using naive hash function in Python.

In [ ]:
```
def shingling(doc, k):
    doc = doc.replace(" ","")
    docLength = len(doc)
    shingles = {doc[i:i + k] for i in range(docLength - k + 1)}
    return shingles

def shinglingHash(doc, k):
    shingles = shingling(doc, k)
    hashBoundary=((2**32)-1)
    hashes = {hash(i)%hashBoundary for i in shingles}
    return hashes

print(shingling("just a test", 2))
shinglingHash("just a test", 2)
```

{'te', 'at', 'es', 'ta', 'us', 'ju', 'st'}

Out[ ]:
```
{39355882,
 522333644,
 1754651990,
 1876116936,
 2839603417,
 3818988377,
 3925642513}
```

Jaccard similarity

The Jaccard similarity of two sets, C1 and C2, is the fraction of common items, i.e., the fraction of their intersection – size of their intersection divided by the size of their union: sim(C1, C2) = |C1∩C2| / |C1∪C2|

In the program, I use function "intersection" and "union" to get the results of Jaccard similarity.

In [ ]:
```
def compareSets(doc1, doc2):
    intersection = len(doc1.intersection(doc2))
```

```
    union = len(doc1.union(doc2))
    return intersection/union


doc1 = shinglingHash("I'm okey, it's just a test", 2)
doc2 = shinglingHash("Are you okey, it's not a test", 2)

print(doc1)
compareSets(doc1, doc2)
```

```
{3925642513, 2150803989, 1173930389, 285524761, 3495883174, 3768853930, 3923407802, 1369153470, 1876116936, 522333644, 2279319117, 1754651990, 3099144920, 3818988377, 2839603417, 39355882, 3066820973, 1938689911, 3398044927}
```

Out[ ]: 0.48148148148148145

Min-hash

Then follow the homework discription, I implement the MinHashing algorithm. I have made a minHashing function that takes a set of shingles and returns a signature. Details of our implementation:

- First, create original chart (the chart that formed through shingling set that with 0 and 1).
- Second, create a random vector for original chart to transform to a min-hash chart.
- Third, find that first col that contains 1 to form a signature chart.

In [ ]:

```
def minHashing(shinglingList, K=100):
    union = set().union(*shinglingList)
    oriChart = np.array([[int(e in s) for s in shinglingList] for e in union])
    sigChart = []
    for i in range(K):
      randomSet = np.random.RandomState(seed=i).permutation(oriChart.shape[0])
      perSigChart = np.take(oriChart, randomSet,axis=0)
      sigCharRow = []
      for col in perSigChart.T:
        sigCharRow.append([np.where(col == 1)][0][0][0])
      sigChart.append(sigCharRow)
    return np.array(sigChart)

def compareSignatures(sigChart):
  return (np.sum(sigChart[:,0] == sigChart[:,1])) / len(sigChart[:,0])

testminHashing = minHashing([doc1, doc2])
print(testminHashing)
print(compareSignatures(testminHashing))
```

```
[[0 1]
 [0 0]
 [4 0]
 [0 0]
 [0 0]
 [0 0]
 [0 1]
 [0 1]
 [0 0]
 [2 0]
 [0 0]
 [1 0]
 [0 0]
 [1 0]
 [0 0]
 [0 1]
```

```
[0 1]
[2 0]
[0 0]
[0 0]
[0 0]
[0 2]
[0 0]
[0 0]
[0 1]
[0 1]
[0 0]
[0 0]
[1 0]
[0 0]
[2 0]
[1 0]
[0 0]
[1 0]
[0 0]
[2 0]
[0 0]
[0 0]
[3 0]
[0 0]
[0 1]
[0 0]
[1 0]
[1 0]
[2 0]
[1 0]
[0 0]
[2 0]
[0 0]
[1 0]
[0 0]
[0 0]
[0 0]
[0 1]
[1 0]
[1 0]
[1 0]
[0 0]
[0 0]
[0 0]
[0 0]
[1 0]
[0 1]
[1 0]
[1 0]
[0 0]
[0 1]
[0 0]
[0 0]
[0 0]
[0 0]
[2 0]
[0 0]
[0 0]
[0 0]
[0 1]
[0 1]
[0 1]
[0 0]
[2 0]
[0 0]
[0 1]
[1 0]
[1 0]
[0 1]
```

```
 [0 1]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [1 0]
 [0 1]
 [1 0]
 [0 0]
 [0 1]
 [0 0]
 [0 1]
 [0 0]
 [1 0]
 [0 0]]
0.49
```

Another min-hash function is to finds the signature matrix without constructing the characteristic matrix, but use a hash function on the form of $ax + b mod N$.

To compare min-hash results ,function "compareHashSets" estimates the similarity of two integer vectors.

In [ ]:

```
def minHashingwithHash(shinglingList, K=100):
    maxID = (2**31)-1
    c = (2**32)-1
    a = np.random.RandomState(seed=1).randint(low=0, high=maxID, size=K, dtype=np.in
    b = np.random.RandomState(seed=2).randint(low=0, high=maxID, size=K, dtype=np.in
    sigHashChart = []
    for i in range(K):
        perSigHashChart = []
        for j in shinglingList:
            perSigHashChart = np.hstack((perSigHashChart,np.min([(a[i] * e + b[i]) %
        sigHashChart.append(perSigHashChart)

    return np.array(sigHashChart)

def compareHashSets(document1, document2):
    return (np.sum(document1 == document2)) / len(document1)

testminHashingWithHash = minHashingwithHash([doc1, doc2])
print(testminHashingWithHash.T)
print(compareHashSets(testminHashingWithHash[:,0],testminHashingWithHash[:,1]))
```

```
[[7.35178300e+06 1.59259731e+08 1.34933799e+08 4.10128996e+08
  1.42226736e+08 3.25538366e+08 3.74523254e+08 2.17887700e+07
  5.04125380e+07 5.55232479e+08 5.27062200e+06 1.12472500e+06
  4.41712986e+08 8.63069658e+08 3.84970300e+07 2.26157102e+08
  3.10759117e+08 1.30022943e+08 1.84466459e+08 1.94876024e+08
  6.20451770e+07 6.40417135e+08 1.03082552e+08 6.88109252e+08
  4.78131967e+08 1.39912656e+08 1.36770146e+08 1.72775804e+08
  5.17270804e+08 2.46990300e+07 7.91173962e+08 2.48979347e+08
  9.16391700e+06 2.99004801e+08 5.68057900e+06 1.08943247e+08
  1.30512205e+08 2.40963100e+07 2.01449450e+07 6.46345110e+07
  6.48986795e+08 8.56120240e+07 3.80111149e+08 2.54544362e+08
  3.93920177e+08 2.70735910e+07 1.74197331e+08 2.99918197e+08
  4.49622137e+08 1.77615295e+08 1.56119444e+08 1.29113500e+06
  1.84237280e+07 1.09266555e+08 1.77714339e+08 2.43794167e+08
  2.55352077e+08 4.34992244e+08 4.20050655e+08 3.34379782e+08
  7.83623560e+07 9.74568520e+07 1.50949110e+08 1.90106659e+08
  1.67672303e+08 1.00154973e+08 2.98889051e+08 1.19564550e+08
  3.67331540e+07 8.43808200e+06 9.53452012e+08 1.46044890e+07
  2.30425929e+08 7.26164200e+06 1.92115887e+08 2.31302774e+08
  3.12789928e+08 1.43205763e+08 7.42219170e+07 5.58929000e+06
  6.85837742e+08 1.23138840e+07 3.82473670e+07 1.94596150e+07
```

```
 2.49180770e+08 2.05766027e+08 5.78646110e+07 3.60213982e+08
 4.53457707e+08 1.85899446e+08 2.47062340e+07 1.52768390e+07
 2.44518589e+08 4.06071736e+08 7.22415000e+06 1.79132373e+08
 2.62570580e+07 9.49398770e+07 2.20084746e+08 2.43113846e+08]
[7.35178300e+06 4.66944414e+08 5.61989516e+08 3.66195556e+08
 1.72722875e+08 3.25538366e+08 3.39325599e+08 2.17887700e+07
 5.04125380e+07 4.08242910e+07 5.68615260e+07 1.12472500e+06
 3.14565260e+07 1.71307816e+08 3.84970300e+07 3.75716955e+08
 3.10759117e+08 1.30022943e+08 2.38114940e+07 1.94876024e+08
 2.38847027e+08 3.80472610e+07 4.98204056e+08 8.22778720e+07
 5.06501656e+08 3.72429582e+08 1.38788436e+08 1.17179039e+08
 3.20991551e+08 2.46990300e+07 3.02395332e+08 2.48979347e+08
 9.16391700e+06 9.31880730e+07 5.68057900e+06 1.24690121e+08
 5.52797820e+07 1.31953316e+08 2.01449450e+07 2.62826070e+08
 1.96196786e+08 9.70686220e+07 4.42002679e+08 6.26114672e+08
 3.93920177e+08 7.79052100e+06 1.74197331e+08 2.99918197e+08
 5.86431500e+06 1.77615295e+08 1.56119444e+08 1.95416117e+08
 1.84237280e+07 1.09266555e+08 3.20376801e+08 3.72291161e+08
 4.75269810e+07 5.52552530e+07 1.80003602e+08 3.34379782e+08
 7.83623560e+07 9.74568520e+07 1.50949110e+08 1.90106659e+08
 1.71428223e+08 1.64145948e+08 8.15202579e+08 1.01827379e+08
 5.95150610e+07 2.39281199e+08 9.53452012e+08 1.46044890e+07
 1.00134503e+09 1.16093710e+07 4.98081310e+07 3.62298785e+08
 3.12789928e+08 1.43205763e+08 7.42219170e+07 5.58929000e+06
 9.14226960e+07 9.21922720e+07 3.82473670e+07 1.94596150e+07
 6.25317750e+07 2.05766027e+08 1.53472544e+08 3.60213982e+08
 4.53457707e+08 1.85899446e+08 5.98811890e+07 9.29149530e+07
 8.08636580e+07 1.63801423e+08 8.37792840e+07 1.79132373e+08
 9.80912810e+07 9.49398770e+07 5.40690200e+07 7.88166280e+07]]
0.4
```

Locality-Sensitive Hashing

A class LSH that implements the LSH technique: given a collection of minhash signatures (integer vectors) and a similarity threshold t, the LSH class (using banding and hashing) finds candidate pairs of signatures agreeing on at least a fraction t of their components. Detail of our implementation:

- Develop an emty sets for buckets.
- Add index into buckets after hashing with a group of signature.
- For each buckets, if it has more than one index in, are added into relevent buckets.
- Choose the index group that has the value bigger then threshold to add into results.

In [ ]:

```
def lhs(signatures, t, bandsNum = 5, bucketsNum = 5):

    buckets = [set() for x in range(0, bucketsNum)]
    bandsTmp = np.linspace(0, len(signatures[0]), bandsNum)
    bands = bandsTmp.astype(int).tolist()

    for index, signature in enumerate(signatures):
        for i in range(0, bandsNum - 1):
            band = "".join(str(x) for x in signature[bands[i]:bands[i + 1]])
            bucket = hash(band) % bucketsNum
            buckets[bucket].add("%s" % index)

    relevant_buckets = list()
    for x in buckets:
        if len(x) >= 2:
            relevant_buckets.append(x)

    relevant_pairs = list()
    for bucket in relevant_buckets:
        pairs = list()
```

```
        for x in itertools.combinations(bucket, 2):
            if x[0] != x[1]:
                pairs.append(x)
        relevant_pairs += pairs

    count = Counter(relevant_pairs)

    indices = list()
    for index, x in enumerate(count.values()):
        if (x/(bandsNum-1)) >= t:
            indices.append(index)

    candidate_pairs = list()
    for index, pair in enumerate(count.keys()):
        if index in indices:
            candidate_pairs.append(pair)

    return candidate_pairs

print(lhs(testminHashingWithHash.T, 0.5))
```

```
[('0', '1')]
```

Testing

I extract five emails as our data to test these functions.

- Using shinging function to create sets of these documents. Then compare these shingling sets with function compareSets.
- Using minHash function to form a signature sets. Get the results using compareMinHashsets.
- Compare the time cost by using shingling with using min-hash.
- Using LSH to get reletive doc with different threshold.

In [ ]:

```
docList = list()
for i in range(1,6):
    docNum = ("%s" % i)
    docName = "doc" + docNum + ".txt"
    with open(docName, "r") as f:
        docList.append(f.read())
    f.close

docListShingling = list()
for i in range(5):
    docListShingling.append(shinglingHash(docList[i],2))

print('1-2', compareSets(docListShingling[0],docListShingling[1]))
print('1-3', compareSets(docListShingling[0],docListShingling[2]))
print('1-4', compareSets(docListShingling[0],docListShingling[3]))
print('1-5', compareSets(docListShingling[0],docListShingling[4]))
print('2-3', compareSets(docListShingling[1],docListShingling[2]))
print('2-4', compareSets(docListShingling[1],docListShingling[3]))
print('2-5', compareSets(docListShingling[1],docListShingling[4]))
print('3-4', compareSets(docListShingling[2],docListShingling[3]))
print('3-5', compareSets(docListShingling[2],docListShingling[4]))
print('4-5', compareSets(docListShingling[3],docListShingling[4]))

docMinHash = minHashingwithHash(docListShingling)
print(compareHashSets(docMinHash[:,0],docMinHash[:,1]))
print(compareHashSets(docMinHash[:,2],docMinHash[:,4]))
```

```
start = time.time()
for i in range(200):
    compareSets(docListShingling[0],docListShingling[1])
print("without min-hash: %s seconds" % (time.time() - start))

start = time.time()
for i in range(200):
    compareHashSets(docMinHash[:,0],docMinHash[:,1])
print("with min-hash: %s seconds" % (time.time() - start))

print(lhs(docMinHash.T, 0.8))
print(lhs(docMinHash.T, 0.6))
print(lhs(docMinHash.T, 0.4))
```

```
1-2 0.5620767494356659
1-3 0.291005291005291
1-4 0.3339920948616601
1-5 0.2793522267206478
2-3 0.2813688212927757
2-4 0.30168776371308015
2-5 0.25877192982456143
3-4 0.3630705394190871
3-5 0.31759656652360513
4-5 0.36855036855036855
0.57
0.28
without min-hash: 0.003000974655151367 seconds
with min-hash: 0.0009980201721191406 seconds
[]
[('3', '0'), ('3', '1'), ('0', '1')]
[('3', '0'), ('3', '1'), ('0', '1'), ('4', '2')]
```

Conclusion of the Testing Result

- By using the funcition shingling, program can extract tokens from each doc for further action (comprare). And the comparation results that created by using these tokens are close to the real similirity relations between these five emails.
- The comparation of using min-hash function is close to the result of comparation by shingling as you can see on the top. At the same time, the time cost in min-hash function is less than only use shingling.
- By using different value of threshold t, the result of finding relevent doc by using LHS function is showing on the top. We can see that with the threshold value getting bigger, the relative doc getting fewer.