Homework1-Sherly_Martignano

November 12, 2019

1 ID2222 - Homework 1

Authors: Sherly Sherly and Anna Martignano

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 the framework and the language of your choice. Below, we give a description of sample classes that implement different stages of finding textually similar documents. You do not have to develop the exact same classes and data types as described below. Feel free to use data structures that suit you best.

1. 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.
2. A class CompareSets that computes the Jaccard similarity of two sets of integers – two sets of hashed shingles.
3. 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).
4. A class CompareSignatures that estimates similarity of two integer vectors – minhash signatures – as a fraction of components, in which they agree.
5. (Optional task for extra 2 bonus) 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 all candidate pairs of signatures that agree on at least fraction t of their components.

To test and evaluate scalability (the execution time versus the size of input dataset) of your implementation, 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.

[1]:
```python
# import cell
import os
import string
import binascii
import time
import random
import re
import numpy as np

from collections import defaultdict
```

[2]:

```
def load_conrad():
    path = "conradbooks"
    file_list = os.listdir(path)

    data = {}

    for file in file_list:
        with open(os.path.join(path, file), 'rb') as f:
            data[file.split(".")[0]] = f.read().decode('utf-8',␣
 ↪errors='replace')

    return data
```

[3]:

```
dataset = load_conrad()
```

[4]:

```
class Shingling:
    def __init__(self, k):
        self.k = k
        self.docs_shingles = {}
        self.doc_names = []

    def _clean(self, doc):
        """
        Some rules for cleaning the text:
        https://www.cs.utah.edu/~jeffp/teaching/cs5955/L4-Jaccard+Shingle.pdf
        """
        doc = doc.lower().replace('\n', ' ')
        doc = re.sub('[^A-Za-z\d\s]', '', doc)
        doc = re.sub(' +', ' ', doc)
        doc = doc.replace(" ", "_")
        return doc

    def _tokenize(self, doc):
        """
        Construct the shingles based on k-characters
        """
        sh = set()
        if len(doc) >= self.k:
            for idx, token in enumerate(doc):
                if idx + self.k <= len(doc):
                    sh.add(self._hash(doc[idx:idx + self.k]))
        return sh

    def _hash(self, shingle):
        """
        Compute hash values for the shingle
        """
        return binascii.crc32(shingle.encode("utf-8")) & 0xffffffff

    def generate_shingles(self, doc):
        doc = self._clean(doc)
        shingles = self._tokenize(doc)

        return shingles

    def generate_shingles_for_docs(self, docs):
        """
        Takes in docs in the form of a dict of {"docID": "doc string"}
        """
        print("Shingling {} articles...".format(len(docs)))

        t0 = time.time()
        for k, v in docs.items():
            self.doc_names.append(k)
            d = self._clean(v)
            d = self._tokenize(d)

            self.docs_shingles[k] = d

        print ('\nShingling took %.2f sec.' % (time.time() - t0))

    @staticmethod
    def compare_sets(s1, s2):
        """
        Compute Jaccard Similarity
        n(intersection) / n(union)
        """
        # add in some checks
        if(s1 == set() or s2 == set()):
            print("Warning: at least one of the two set is empty\n")
            return 0
        else:
            jacc_sim = (len(s1.intersection(s2)) / float(len(s1.union(s2))))
            return jacc_sim
```

[5]:

```
shing = Shingling(7)
```

[6]:

```
shing.generate_shingles_for_docs(dataset)
```

```
Shingling 10 articles...

Shingling took 3.14 sec.
```

[7]:
```
shing.compare_sets(shing.docs_shingles[shing.doc_names[1]],
                   shing.docs_shingles[shing.doc_names[2]])
```

[7]:
```
0.1839672697052995
```

1.0.1 MinHashing

[8]:
```
class MinHashing:

    def __init__(self, n, max_shingle_ID = 2**32-1):
        self.n = n # number of hashes
        self.max_shingle_ID = max_shingle_ID # the max number
        self.next_prime = 4294967311 # the next prime number after max shingle␣
 ↪ID
        self.coeffs_A = self.generate_coeffs()
        self.coeffs_B = self.generate_coeffs()
        self.docs_minhash_signatures = {}

    def generate_coeffs(self):
        """
        Create a list of 'n' unique random values.
        """
        coeffs_list = []

        for _ in range(self.n):
            # TODO: check if it a good idea to have 0 for coeff A
            rand_idx = random.randint(0, self.max_shingle_ID)

            # Ensure that each random number is unique.
            while rand_idx in coeffs_list:
                rand_idx = random.randint(0, self.max_shingle_ID)

            coeffs_list.append(rand_idx)

        return coeffs_list

    def _minHash_function(self, pos, x):
        """
        Return a hash in the form of (ax+b) % prime
        """
        return (self.coeffs_A[pos] * x + self.coeffs_B[pos]) % self.next_prime

    def generate_signature(self, shingle_set):
        """
        Given a shingle set of IDs, generate the hashes and compute the minimum␣
 ↪hash
        """
        signature = []

        for i in range(self.n):
            signature.append(min(map(lambda x: self._minHash_function(i,x),␣
 ↪shingle_set)))

        return signature

    def generate_doc_signatures(self, shingles):
        print("Generating MinHash signatures for documents..")
        t0 = time.time()

        for k, v in shingles.items():
            self.docs_minhash_signatures[k] = self.generate_signature(v)

        print ('\n Generating Signatures for ' + str(len(shingles)) + ' docs␣
 ↪took %.2f sec.' % (time.time() - t0))

    @staticmethod
    def compare_signatures(s1, s2):
        if not len(s1) == len(s2):
            print("Unequal length of Signature")

        equality = 0
        signature_len = len(s1)
        for x, y in zip(s1, s2):
            if(x == y):
                equality += 1
        return equality / float(signature_len)
```

[9]:
```
minhash = MinHashing(200)
```

[10]:
```
minhash.compare_signatures(minhash.generate_signature(shing.docs_shingles[shing.
 ↪doc_names[1]]),
                   minhash.generate_signature(shing.docs_shingles[shing.
 ↪doc_names[2]]))
```

[10]:
```
0.19
```

[11]:
```
minhash.generate_doc_signatures(shing.docs_shingles)
```

```
Generating MinHash signatures for documents..

 Generating Signatures for 10 docs took 157.67 sec.
```

1.0.2 LSH

Partition into Bands - Divide matrix M into b bands of r rows. - For each band, hash its portion of each column to a hash table with k buckets. - Make k as large as possible. - Candidate column pairs are those that hash to the same bucket for a number of bands with regards to the threshold set.

[12]:
```
class LSH:

    def __init__(self, band_size, row_size, threshold):
        self.band_size = band_size
        self.threshold = threshold
        self.row_size = row_size
        self.docs_lsh = {}
        self.candidate_pairs = defaultdict(set)

    def get_lsh(self, signature):
        lsh = []
        for i in range(self.band_size):
            lsh.append(hash(tuple(signature[i*self.row_size:(i*self.
 ↪row_size+self.row_size)])) % 4294967311)
        return lsh

    def get_lsh_for_docs(self, signatures):
        print("Generating LSH signatures for documents..")
        t0 = time.time()
        for k, v in signatures.items():
            self.docs_lsh[k] = self.get_lsh(v)
        print ('\n Generating LSH for ' + str(len(signatures)) + ' docs took %.
 ↪2f sec.' % (time.time() - t0))


    def generate_candidate_pairs(self):
        """
        t: the fraction of components that pair of signatures agrees on
        """
        print("Generating Candidate Pairs for documents..")
        t0 = time.time()

        all_docs = list(self.docs_lsh.values())
        all_names = list(self.docs_lsh.keys())

        # Minimum number of bands that should has overlap
        # hash according to the threshold set
        threshold = self.threshold * self.band_size

        # Stores the intermediate number of band overlaps
        pairs = defaultdict(lambda: defaultdict(float))
        for idx, s1 in enumerate(all_docs):
            s1_name = all_names[idx]

            # Sliding count to perform comparison
            for curr_iter, s2 in enumerate(all_docs[idx + 1:]):
                s2_name = all_names[curr_iter + idx + 1]

                for x, y in zip(s1, s2):
                    if(x == y):
                        if not pairs[s1_name][s2_name]:
                            pairs[s1_name][s2_name] = 1
                        else:
                            pairs[s1_name][s2_name] += 1

                        if not pairs[s2_name][s1_name]:
                            pairs[s2_name][s1_name] = 1
                        else:
                            pairs[s2_name][s1_name] += 1

                # Store pairs that is above the threshold as candidate pairs
                if pairs[s1_name][s2_name] > threshold:
                    self.candidate_pairs[s1_name].add(s2_name)
                    self.candidate_pairs[s2_name].add(s1_name)


        print ('\n Generating Candidate Pairs for ' + str(len(all_docs)) + '␣
↪docs took %.2f sec.' % (time.time() - t0))
```

[32]:
```
lshh = LSH(100, 2, 0.08)
```

[33]:
```
lshh.get_lsh_for_docs(minhash.docs_minhash_signatures)
```

[34]:
```
lshh.generate_candidate_pairs()
```

```
Generating MinHash signatures for documents..

 Generating Signatures for 10 docs took 0.00 sec.
```

[35]:
```
lshh.candidate_pairs
```

[35]:
```
defaultdict(set,
            {'AmyFoster': {'The Secret Sharer'},
             'The Secret Sharer': {'AmyFoster'},
             'ChanceATaleInTwoParts': {'TheArrowofGold'},
             'TheArrowofGold': {'ChanceATaleInTwoParts'}})
```

1.1 Scalability

To test and evaluate scalability (the execution time versus the size of input dataset) of your implementation, 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.

[36]:

```
def generate_sim_matrix(vectors, doc_names):
    print("Calculating the Similarity for all documents")
    dataset_size = len(vectors)

    simMatrix = np.zeros(dataset_size * dataset_size).
 ↪reshape(dataset_size,dataset_size)
    t0 = time.time()

    for j in range(0, len(doc_names)):
        s1 = vectors[doc_names[j]]
        for k in range(j, len(doc_names)):
            s2 = vectors[doc_names[k]]
            if(s1 == set() or s2 == set()):
                print("Warning: at least one of the two set is empty\n")
                sim = 0
            elif j == k:
                sim = 0
            else:
                sim = (len(set(s1).intersection(set(s2))) / float(len(set(s1).
 ↪union(set(s2)))))
            simMatrix[j, k] = sim
            simMatrix[k, j] = sim

    print('\nSimilarity for ' + str(len(doc_names)) + ' docs took %.2f sec.' %␣
 ↪(time.time() - t0))
    np.set_printoptions(precision=3)
    print('\nSimilarity Matrix\n ' + str(simMatrix))

    return simMatrix

def retrieve_documents(sim_matrix, threshold=0.2, doc_names=None):
    sim_doc_indices = np.argwhere(sim_matrix > threshold)

    sim_docs = defaultdict(list)
    for pair in sim_doc_indices:
        if doc_names:
            sim_docs[doc_names[pair[0]]].append(doc_names[pair[1]])
        else:
            sim_docs[pair[0]].append(pair[1])

    return sim_docs
```

[44]:

```python
def retrieve_similar_docs_by_shingles(dataset, n=7, sim_score=0.2):
    print("Generating Shingles for documents..")
    t0 = time.time()

    s = Shingling(n)
    s.generate_shingles_for_docs(dataset)
    sim_matrix = generate_sim_matrix(s.docs_shingles, s.doc_names)
    sim_docs = retrieve_documents(sim_matrix, sim_score, s.doc_names)

    print("Similar documents are: \n")
    print(sim_docs)
    print ('\n Retrieving similar documents for ' + str(len(dataset)) + ' docs␣
 ↪took %.2f sec.' % (time.time() - t0))
    return sim_docs
```

[46]:

```python
retrieve_similar_docs_by_shingles(dataset, sim_score=0.25)
```

```
Generating Shingles for documents..
Shingling 10 articles...

Shingling took 3.09 sec.
Calculating the Similarity for all documents

Similarity for 10 docs took 2.67 sec.

Similarity Matrix
 [[0.    0.175 0.232 0.243 0.245 0.223 0.187 0.246 0.232 0.236]
 [0.175 0.    0.184 0.137 0.183 0.213 0.233 0.148 0.196 0.165]
 [0.232 0.184 0.    0.237 0.238 0.229 0.199 0.244 0.232 0.243]
 [0.243 0.137 0.237 0.    0.243 0.21  0.158 0.3   0.221 0.248]
 [0.245 0.183 0.238 0.243 0.    0.234 0.2   0.241 0.235 0.241]
 [0.223 0.213 0.229 0.21  0.234 0.    0.227 0.217 0.236 0.22 ]
 [0.187 0.233 0.199 0.158 0.2   0.227 0.    0.169 0.212 0.182]
 [0.246 0.148 0.244 0.3   0.241 0.217 0.169 0.    0.229 0.248]
 [0.232 0.196 0.232 0.221 0.235 0.236 0.212 0.229 0.    0.226]
 [0.236 0.165 0.243 0.248 0.241 0.22  0.182 0.248 0.226 0.   ]]
Similar documents are:

defaultdict(<class 'list'>, {'ChanceATaleInTwoParts': ['TheArrowofGold'],
'TheArrowofGold': ['ChanceATaleInTwoParts']})

 Retrieving similar documents for 10 docs took 5.77 sec.
```

[46]:

```
defaultdict(list,
            {'ChanceATaleInTwoParts': ['TheArrowofGold'],
             'TheArrowofGold': ['ChanceATaleInTwoParts']})
```

[39]:

```python
def retrieve_similar_docs_by_minhash(dataset, n=7, sim_score=0.15, k=200):
    print("Generating Minhash for documents..")
    t0 = time.time()

    s = Shingling(n)
    s.generate_shingles_for_docs(dataset)
    m = MinHashing(k)
    m.generate_doc_signatures(s.docs_shingles)

    sim_matrix = generate_sim_matrix(m.docs_minhash_signatures, s.doc_names)
    sim_docs = retrieve_documents(sim_matrix, sim_score, s.doc_names)

    print("Similar documents are: \n")
    print(sim_docs)
    print ('\n Retrieving similar documents for ' + str(len(dataset)) + ' docs␣
 ↪took %.2f sec.' % (time.time() - t0))
    return sim_docs
```

[42]:

```python
retrieve_similar_docs_by_minhash(dataset, sim_score=0.15)
```

```
Generating Minhash for documents..
Shingling 10 articles...

Shingling took 2.62 sec.
Generating MinHash signatures for documents..

 Generating Signatures for 10 docs took 159.83 sec.
Calculating the Similarity for all documents

Similarity for 10 docs took 0.00 sec.

Similarity Matrix
 [[0.    0.096 0.12  0.16  0.157 0.143 0.099 0.13  0.13  0.153]
 [0.096 0.    0.078 0.073 0.093 0.13  0.13  0.064 0.117 0.067]
 [0.12  0.078 0.    0.121 0.134 0.13  0.096 0.14  0.117 0.134]
 [0.16  0.073 0.121 0.    0.147 0.13  0.084 0.143 0.124 0.147]
 [0.157 0.093 0.134 0.147 0.    0.153 0.096 0.143 0.121 0.131]
 [0.143 0.13  0.13  0.13  0.153 0.    0.12  0.146 0.143 0.124]
 [0.099 0.13  0.096 0.084 0.096 0.12  0.    0.078 0.096 0.075]
 [0.13  0.064 0.14  0.143 0.143 0.146 0.078 0.    0.114 0.134]
 [0.13  0.117 0.117 0.124 0.121 0.143 0.096 0.114 0.    0.121]
 [0.153 0.067 0.134 0.147 0.131 0.124 0.075 0.134 0.121 0.   ]]
Similar documents are:

defaultdict(<class 'list'>, {"Almayer'sFolly": ['ChanceATaleInTwoParts',
'EndOfTheTether', 'TheMirrorOfTheSee'], 'ChanceATaleInTwoParts':
["Almayer'sFolly"], 'EndOfTheTether': ["Almayer'sFolly", 'Falk'], 'Falk':
['EndOfTheTether'], 'TheMirrorOfTheSee': ["Almayer'sFolly"]})
```

```
 Retrieving similar documents for 10 docs took 162.45 sec.
```

[42]:
```
defaultdict(list,
            {"Almayer'sFolly": ['ChanceATaleInTwoParts',
              'EndOfTheTether',
              'TheMirrorOfTheSee'],
             'ChanceATaleInTwoParts': ["Almayer'sFolly"],
             'EndOfTheTether': ["Almayer'sFolly", 'Falk'],
             'Falk': ['EndOfTheTether'],
             'TheMirrorOfTheSee': ["Almayer'sFolly"]})
```

[47]:
```
def retrieve_similar_docs_by_lsh(dataset, n=7, k=200, b=100, r=2, threshold=0.
 ↪1):
    print("Generating LSH for documents..")
    t0 = time.time()

    s = Shingling(n)
    s.generate_shingles_for_docs(dataset)
    m = MinHashing(k)
    m.generate_doc_signatures(s.docs_shingles)
    lsh = LSH(b, r, threshold)
    lsh.get_lsh_for_docs(m.docs_minhash_signatures)
    lsh.generate_candidate_pairs()

    print("Similar documents are: \n")
    print(lsh.candidate_pairs)
    print ('\n Retrieving similar documents for ' + str(len(dataset)) + ' docs␣
 ↪took %.2f sec.' % (time.time() - t0))
    return lsh.candidate_pairs
```

[48]:
```
retrieve_similar_docs_by_lsh(dataset, threshold=0.08)
```

```
Generating LSH for documents..
Shingling 10 articles...

Shingling took 2.79 sec.
Generating MinHash signatures for documents..

 Generating Signatures for 10 docs took 148.84 sec.
Generating MinHash signatures for documents..

 Generating Signatures for 10 docs took 0.00 sec.
Similar documents are:

defaultdict(<class 'set'>, {'ChanceATaleInTwoParts': {'TheArrowofGold'},
'TheArrowofGold': {'ChanceATaleInTwoParts'}})

 Retrieving similar documents for 10 docs took 151.63 sec.
```

[48]:
```
defaultdict(set,
            {'ChanceATaleInTwoParts': {'TheArrowofGold'},
             'TheArrowofGold': {'ChanceATaleInTwoParts'}})
```

[ ]:
```

```