AI506: Data Mining and Search (Spring 2020)

Homework 1: Locality Sensitive Hashing

Student ID: 20194331

Name: Sungnyun Kim 김성년

1 Min-Hashing

1.1 Shingling

1. `get_shingles`

```python
def get_shingles(documents):
    ########################################################################################
    # Programming 1.1  [10pt]                                                              #
    #  Implement 'get_shingles' function to get 1-singles from the preprocessed documents  #
    #  You should especially be take care of your algorithm's computational efficiency    #
    #                                                                                      #
    #  Parameters:                                                                         #
    #    documents (dict)                                                                  #
    #                                                                                      #
    #  Returns:                                                                            #
    #    shingles (set) set of tuples where each element is a k-shingle                   #
    #    ex) shingles = {('its', 'hard', 'to', 'say', 'whether'),                          #
    #                    ('known', 'bugs', 'in', 'the', 'warning') ...}                    #
    ########################################################################################
    shingles = set()
    for doc in documents:
        doc_split = doc.split()
        for i in range(len(doc_split) - (K-1)):
            shingles.add(tuple(doc_split[i:i+K]))

    return shingles
```

>>> Test Result

```python
start = time.time()
shingles = get_shingles(documents)
end = time.time()

# Check whether your implementation is correct [5pt]
if len(shingles) == 1766049:
    pass_test1_1_1 = True
    print('Test1 passed')

    # Check whether your implementation is efficient enough [5pt]
    # With 4-lines of my implementations, it took 4.8 seconds with i7-8700 cpu
    if (end - start) < 20:
        pass_test1_1_2 = True
        print('Test2 passed')
```

```
Test1 passed
Test2 passed
```

2. build_doc_to_shingle_dictionary

```python
def build_doc_to_shingle_dictionary(documents, shingles):
    ##########################################################################################
    # Programming 1.2 [5pt]
    #  Implement 'build_doc_to_shingle_dictionary' function to convert documents into shingle
    #   You need to construct and utilize a shingle2idx dictionary that maps each shingle int
    #
    #  Parameters:
    #    documents (dict)
    #    shingles (set)
    #
    #  Returns:
    #    doc_to_shingles (dict)
    #        key: index of the documents
    #        value: list of the shingle indexes
    #        ex) doc_to_shingles = {0: [1705196, 422880, 491967, ...],
    #                               1: [863922, 1381606, 1524066, ...],
    #                               ... }
    ##########################################################################################
    doc_to_shingles = {}
    shingle2idx = {}
    for idx, shingle in enumerate(shingles):
        shingle2idx[shingle] = idx
    for idx, doc in enumerate(documents):
        shingle_list = [shingle2idx[s] for s in get_shingles([doc])]
        doc_to_shingles[idx] = shingle_list

    return doc_to_shingles
```

>>> Test Result

```python
doc_to_shingles = build_doc_to_shingle_dictionary(documents, shingles)

# Check whether your implementation is correct [5pt]
if len(doc_to_shingles) == 10882 and len(doc_to_shingles[0]) == 84:
    pass_test1_2 = True
    print('Test passed')
```

```
Test passed
```

1.2. Min-Hashing

1.2.1. Computing MinHash signatures

1. Calculate minhash signature

$$\min_{x\in S_1} h_1(x) = \min\{h_1(2), h_1(5)\} = \min\{5,5\} = 5$$
$$\min_{x\in S_1} h_2(x) = \min\{h_2(2), h_2(5)\} = \min\{2,5\} = 2$$
$$\min_{x\in S_1} h_3(x) = \min\{h_3(2), h_3(5)\} = \min\{0,3\} = 0$$

Similarly,

$$[\min_{x\in S_2} h_1, \min_{x\in S_2} h_2, \min_{x\in S_2} h_3] = [1,2,1]$$
$$[\min_{x\in S_3} h_1, \min_{x\in S_3} h_2, \min_{x\in S_3} h_3] = [1,2,4]$$
$$[\min_{x\in S_4} h_1, \min_{x\in S_4} h_2, \min_{x\in S_4} h_3] = [1,2,0]$$

Signature:

S_1	S_2	S_3	S_4
5	1	1	1
2	2	2	2
0	1	4	0

2. Calculate true and estimated Jaccard similarities.

	(S_1,S_2)	(S_1,S_3)	(S_1,S_4)	(S_2,S_3)	(S_2,S_4)	(S_3,S_4)
true	0	0	1/4	0	1/4	1/4
estimated	1/3	1/3	2/3	2/3	2/3	2/3

1.2.2. Implementation

1. `jaccard_similarity`

```
def jaccard_similarity(s1, s2):
    ##################################################################################
    # Programming 2.2 [5pt]                                                          #
    #   Implement the jaccard similarity algorithm to get the similarity of two sets #
    #                                                                                #
    # Parameters                                                                     #
    #    s1 (set)                                                                    #
    #    s2 (set)                                                                    #
    # Returns                                                                        #
    #    similarity (float)                                                          #
    ##################################################################################
    similarity = len(s1&s2) / len(s1|s2)
    return similarity
```

>>> Test Result

```
s1 = {1, 3, 4}
s2 = {3, 4, 6}

if (jaccard_similarity(s1, s2) - 0.5) < 1e-3:
    pass_test2_1 = True
    print('Test passed')
```

```
Test passed
```

2. min_hash

I modified `Hash` class in order to parallelize the hash function calculation.

```
class Hash():
    def __init__(self, M, N):
        self.M = M
        self.N = N
        self.p = generate_prime_numbers(M, N)

        self.a = np.random.choice(9999, M)
        self.b = np.random.choice(9999, M)

    def __call__(self, x):
        return np.mod(np.mod((self.a * x + self.b), self.p), self.N)

    def __len__(self):
        return M

#primes = generate_prime_numbers(M, N)
hash_functions = Hash(M, N)
```

```
def min_hash(doc_to_shingles, hash_functions):
    ###########################################################################################
    # Programming 2.3 [20pt]                                                                  #
    #   Implement the min-hash algorithm to create the signatures for the documents           #
    #   It would take about ~10 minutes to finish computation,                                #
    #      while would take ~20 seconds if you parallelize your hash functions                #
    #                                                                                         #
    #  Parameters                                                                             #
    #     doc_to_shingles: (dict) dictionary that maps each document to the list of shingles  #
    #     hash_functions: [list] list of hash functions                                       #
    #  Returns                                                                                #
    #     signatures (np.array) numpy array of size (M, C) where C is the number of documents #
    #                                                                                         #
    ###########################################################################################

    C = len(doc_to_shingles)
    M = len(hash_functions)
    signatures = np.array(np.ones((M, C)) * 999999999999, dtype = np.int)

    for doc_id in range(C):
        shingles = doc_to_shingles[doc_id]
        for shingle in shingles:
            hash_shingle = hash_functions(shingle)
            signatures[:,doc_id] = np.where(hash_shingle < signatures[:,doc_id], hash_shingle, signatures[:,doc_id])

    return signatures
```

>>> Test Result

```
start = time.time()
signatures = min_hash(doc_to_shingles, hash_functions)
end = time.time()

diff_list = compare(signatures, doc_to_shingles)

# Check whether your implementation is correct [20pt]
# Average difference of document's jaccard similarity betwe
# With 10 random seeds, difference was around 1e-5 ~ 1e-6%
if np.mean(diff_list) < 0.01:
    pass_test2_2 = True
    print('Test passed')
```

```
100%|██████████| 10000/10000 [00:04<00:00, 2437.12it/s]
Test passed
```

2 Locality Sensitive Hashing (LSH)

2.1 Locality Sensitive Hashing

1. lsh

```
def lsh(signatures, b, r):
    ##################################################################################################
    # Programming 3.1 [20pt]                                                                         #
    #   Implement the min-hash based LSH algorithm to find the candidate pairs of the similar documents.    #
    #   In the implementation, use python's dictionary to make your hash table,                      #
    #    where each column is hashed into a bucket.                                                   #
    #   Convert each column vector (within a band) into the tuple and use it as a key of the dictionary.    #
    #                                                                                                #
    #  Parameters                                                                                    #
    #     signatures: (np.array) numpy array of size (M, C) where                                    #
    #                 M is the number of min-hash functions, C is the number of documents            #
    #     b: (int) the number of bands                                                               #
    #     r: (int) the number of rows per each band                                                  #
    #                                                                                                #
    #  Requirements                                                                                  #
    #     1) M should be equivalent to b * r                                                         #
    #                                                                                                #
    #  Returns                                                                                       #
    #     candidatePairs (Set[Tuple[int, int]]) set of the pairs of indexes of candidate document pairs    #
    #                                                                                                #
    ##################################################################################################
    M = signatures.shape[0]  # The number of min-hash functions
    C = signatures.shape[1]  # The number of documents

    assert M == b * r

    candidatePairs = set()

    # TODO: Write down your code here
    for num_b in range(b):
        bucket = {}
        bands = signatures[num_b*r:(num_b+1)*r]
        for col in range(C):
            if tuple(bands[:,col]) in bucket.keys():
                bucket[tuple(bands[:,col])].append(col)
            else:
                bucket[tuple(bands[:,col])] = [col]

        for value in bucket.values():
            if len(value) >= 2:
                combi = combinations(value, 2)
                candidatePairs.update(list(combi))
                #import ipdb; ipdb.set_trace()
    ### Implementation End ###

    return candidatePairs
```

>>> Test Result

```python
# You can test your implementation here
b = 10
n = 0
tmpPairs = list(lsh(signatures, b, M // b))
print(f"b={b}")
print(f"# of candidate pairs = {len(tmpPairs)}")
samplePair = tmpPairs[n]
shingle1, shingle2 = set(doc_to_shingles[samplePair[0]]), set(doc_to_shingles[samplePair[1]])
print(f"{n}th sample pair: {samplePair}")
print(f"Jaccard similarity: {jaccard_similarity(shingle1, shingle2)}")
print('-------------')
print(documents[samplePair[0]])
print('-------------')
print(documents[samplePair[1]])
print('-------------')
```

```
b=10
# of candidate pairs = 162
0th sample pair: (1658, 5780)
Jaccard similarity: 0.8108108108108109
-------------
from paynecrldeccom andrew payne messageid organization dec cambridge research lab date tue apr gmt does anyone know if a source for the mode
ms ideally something that is geared toward hobbyists small quantity mail order etc for years weve been buying them from a distributor marshal
e dropped to the point where we can no longer afford to offer this service and all of the distributors ive checked have some crazy minimum or
still interested in building pmp kits any suggestions andrew c payne dec cambridge research lab
-------------
does anyone know if a source for the modem chips as used in the baycom and my pmp modems ideally something that is geared toward hobbyists sm
en buying them from a distributor marshall by the hundreds for pmp kits but orders have dropped to the point where we can no longer afford to
rs ive checked have some crazy minimum order or so id like to find a source for those still interested in building pmp kits any suggestions
-------------
```

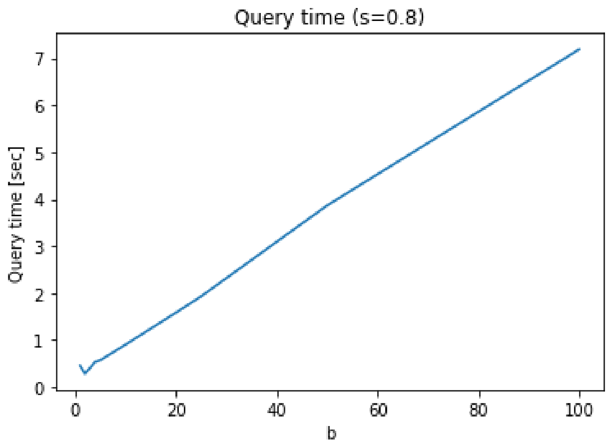
2.2 Analysis

1. `query_analysis`

```python
def query_analysis(signatures, b, s, numConditionPositives):
    ###########################################################################################################
    # Programming 3.2 [10pt]                                                                                  #
    #   Calculate the query time, precision, recall, and F1 score for the given configuration                 #
    #                                                                                                         #
    #  Parameters                                                                                             #
    #     signatures: (np.array) numpy array of size (M, C) where                                             #
    #                 M is the number of min-hash functions, C is the number of documents                     #
    #     b: (int) the number of bands                                                                        #
    #     s: (float) similarity threshold for checking condition positives                                    #
    #     numConditionPositives: (int) the number of condition positives                                      #
    #                                                                                                         #
    #  Requirements                                                                                           #
    #     1) b should be the divisor of M                                                                     #
    #     2) 0 <= s <= 1                                                                                      #
    #                                                                                                         #
    #  Returns                                                                                                #
    #     query time: (float) the execution time of the codes which find the similar document candidate pairs #
    #     precision: (float)                                                                                  #
    #     recall: (float)                                                                                     #
    #     f1: (float) F1-Score                                                                                #
    #                                                                                                         #
    ###########################################################################################################
    M = signatures.shape[0]  # The number of min-hash functions
    assert M % b == 0

    # TODO: Write down your code here
    TP = 0
    t = time.time()
    candidatePairs = lsh(signatures, b, M // b)
    query_time = time.time() - t

    for pair in candidatePairs:
        shingle1, shingle2 = set(doc_to_shingles[pair[0]]), set(doc_to_shingles[pair[1]])
        if jaccard_similarity(shingle1, shingle2) >= s:
            TP += 1

    precision = TP / len(candidatePairs)
    recall = TP / numConditionPositives
    f1 = 2 * precision * recall / (precision + recall)
    ### Implementation End ###

    return query_time, precision, recall, f1
```

2. query time graph

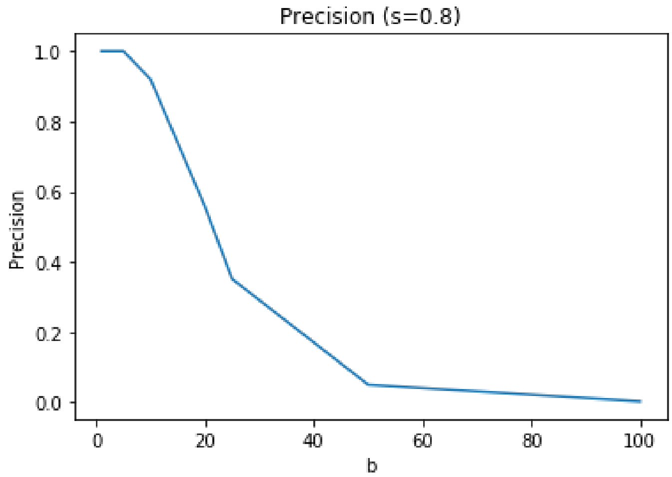
The query time linearly increases with b. This is because the larger b is, the shorter each band is. Then this means each band is more likely to be in the same bucket, which is being a candidate pair. Increased candidate pairs cause linearly increased query time. Therefore, query time has time complexity of O(b).

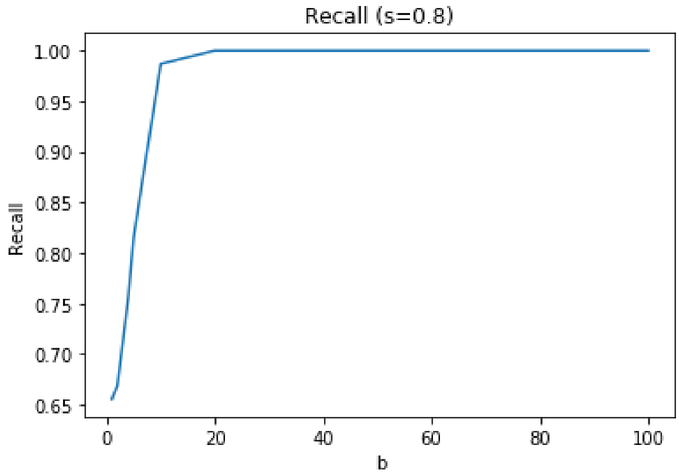


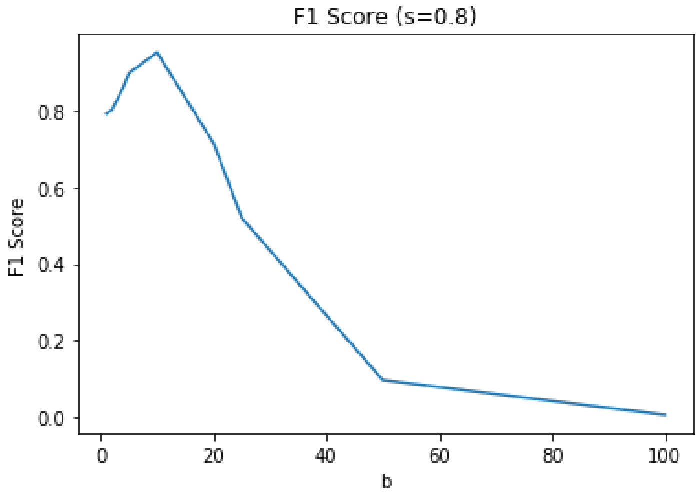
3. precision, recall, and f1-score graph

Precision decreases with b, while recall increases with b. This is obvious because as b increases, the number of candidate pairs increase. Therefore, positive pairs that are assumed to be similar increase. For precision, there is FP in denominator, and FP increases as there are more positive pairs. So precision decreases. For recall, there is FN in denominator, and FN decreases as positive pairs increase. Therefore, recall increases.

In precision measure, the best b value is 1, but in recall measure, the best b value is 20~100. By the way, by looking at the f1-score, we can be between that trade-off. The best b, in terms of f1-score measure, is 10.


Precision (s=0.8)
Precision
1.0
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
b


Recall (s=0.8)
Recall
1.00
0.95
0.90
0.85
0.80
0.75
0.70
0.65
0
20
40
60
80
100
b


F1 Score (s=0.8)
F1 Score
0.8
0.6
0.4
0.2
0.0
0
20
40
60
80
100
b