

AI506: DATA MINING AND SEARCH (SPRING 2020)

Homework 1: Locality Sensitive Hashing

20203221 민향숙

1 Min-Hashing [50 points]

The starter code is provided for you to implement the Min Hashing algorithm. Fill in the blanks in the code marked as programming assignments.

1.1 Shingling [15 points]

In this section, you will implement the shingling algorithm to convert the document into the characteristic matrix. You will use a *word* as a token for your shingle, not a *character*. However, since storing the whole characteristic matrix in the form of a dense matrix is expensive in terms of space, your implementation should store the characteristic matrix in the form of a dictionary.

Note that implementation of shingling is divided into 2-steps just for the readability of the algorithm.

1. (10 points) Implement the `get_shingles` to get 1-shingles from the documents. You should also take care of your implementation's computational efficiency.

[CODE]

```
def get_shingles(documents):  
    |  
    shingle = []  
    for doc in documents:  
        words = doc.split()  
        s = zip(*[words[i:] for i in range(K)])  
        shingle.extend(list(s))  
    shingles = set(shingle)  
  
    return shingles
```

[RESULT]

```
start = time.time()  
shingles = get_shingles(documents)  
end = time.time()  
|  
if len(shingles) == 1766049:  
    pass_test1_1_1 = True  
    print("Test1 passed")  
  
    if (end-start) < 20:  
        pass_test1_1_2 = True  
        print("Test2 passed")  
  
Output was trimmed for performance reasons.  
To see the full output set the setting "python.dataS  
...  
Test1 passed  
Test2 passed
```

I first made a list *shingle* and use extend function to store each of shingles. Next changed the list *shingle* into the set *shingles*.

2. (5 points) Implement the `build_doc_to_shingle_dictionary` to build the dictionary that maps each document to the list of 1-shingles in the document.

[CODE]

```
def build_doc_to_shingle_dictionary(documents, shingles):
    |
    doc_to_shingles = {}
    shingle2idx = {}

    for k,v in enumerate(shingles):
        shingle2idx[v] = k

    for i in range(len(documents)):
        words = documents[i].split()
        s = zip(*[words[j:] for j in range(K)])
        doc_to_shingles[i] = list(set([shingle2idx[k] for k in s]))

    return doc_to_shingles
```

[RESULT]

```
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
```

First, I tried to make a dictionary *single2idx* which has a key as the list of shingles, and a value as a document number. Secondly, I made another dictionary *doc_to_shingles* which represents documents as ids. And ids in each document are not duplicated.

1.2 Min-Hashing [35 points]

In this section, you will take a time to understand the Min Hash algorithm.

1.2.1 (Written) Computing MinHash signatures [10 points]

First, to review the algorithm, calculate by hand the minhash signatures using a simple characteristic matrix.

Element	S ₁	S ₂	S ₃	S ₄
0	0	1	0	1
1	0	1	0	0
2	1	0	0	1
3	0	0	1	0
4	0	0	1	1
5	1	0	0	0

Characteristic matrix for Hw 1.2.1

1. (5 points) Calculate the minhash signatures of each column of the matrix given above using the following 3 hash functions:
 $h_1(x) = 2x+1 \bmod 6$; $h_2(x) = 3x+2 \bmod 6$; $h_3(x) = 5x+2 \bmod 6$.

Element	S ₁	S ₂	S ₃	S ₄	$2x+1 \bmod 6$	$3x+2 \bmod 6$	$5x+2 \bmod 6$
0	0	1	0	1	1	2	2
1	0	1	0	0	3	5	1
2	1	0	0	1	5	2	0
3	0	0	1	0	1	5	5
4	0	0	1	1	3	2	4
5	1	0	0	0	5	5	3

■ Signature Matrix				
	S1	S2	S3	S4
H1	5	1	1	1
H2	2	2	2	2
H3	0	1	4	0

2. (5 points) Calculate the true and estimated Jaccard similarities of the six pairs of the columns.

(1) True Jaccard similarity

	S1 = {2,5}	S2 = {0,1}	S3 = {3,4}	S4 = {0,2,4}
S1	1	$S1 \cup S2 = \{0,1,2,5\}$ $S1 \cap S2 = \{\emptyset\}$ $\text{Sim}(S1, S2) = 0$	$S1 \cup S3 = \{2,3,4,5\}$ $S1 \cap S3 = \{\emptyset\}$ $\text{Sim}(S1, S3) = 0$	$S1 \cup S4 = \{0,2,4,5\}$ $S1 \cap S4 = \{2\}$ $\text{Sim}(S1, S4) = 1/4$
S2		1	$S2 \cup S3 = \{0,1,3,4\}$ $S2 \cap S3 = \{\emptyset\}$ $\text{Sim}(S2, S3) = 0$	$S2 \cup S4 = \{0,1,2,4\}$ $S2 \cap S4 = \{0\}$ $\text{Sim}(S2, S4) = 1/4$
S3			1	$S3 \cup S4 = \{0,2,3,4\}$ $S3 \cap S4 = \{\emptyset\}$ $\text{Sim}(S3, S4) = 1/4$
S4				1

(2) Estimated Jaccard similarity

	S1 = [5,2,0]	S2 = [1,2,1]	S3 = [1,2,4]	S4 = [1,2,0]
S1	1	Second is same $\text{Sim}(S1, S2) = 1/3$	Second is same. $\text{Sim}(S1, S3) = 1/3$	Second and Third is same. $\text{Sim}(S1, S4) = 2/3$
S2		1	First and Second is same. $\text{Sim}(S2, S3) = 2/3$	First and Second is same. $\text{Sim}(S2, S4) = 2/3$
S3			1	First and Second is same. $\text{Sim}(S1, S4) = 2/3$
S4				1

1.2.2 Implementation [25 points]

Now, implement the min-hashing algorithm to convert the characteristic matrix into the signatures.

1. (5 points) Implement the `jaccard_similarity` to compute the Jaccard similarity of two given sets.
2. (20 points) Implement the `min_hash` to create the signature for the documents. It would take about 10 minutes to compute signature in iterative manner, while would take about 20 seconds if parallelized. We will not evaluate for the computational efficiency.

I changed *Hash function* a little bit. For `__call__` part, originally it gets scalar x as a parameter, instead of scalar, I put a vector V in the parameter to compute the function much faster. Finally *Hash function* returns a minimum value of each document results. Thus, *function min_hash* gets signatures by calculating documents one by one for one hash function.

[CODE]

```
M = 100 # Number of Hash functions to use
N = len(shingles)

# First we will create M universal hashing functions
# You can also modify or implement your own hash functions for implementing min_hash function

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

        self.a = np.random.randint(9999)
        self.b = np.random.randint(9999)

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

primes = generate_prime_numbers(M, N)
hash_functions = [Hash(M, N, p) for p in primes]
```

[CODE]

```
def jaccard_similarity(s1, s2):
    intersection = s1.intersection(s2)
    union = s1.union(s2)
    similarity = len(intersection) / len(union)
    return similarity
```

[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
```

```
def min_hash(doc_to_shingles, hash_functions):
    C = len(doc_to_shingles)
    M = len(hash_functions)
    signatures = np.array(np.ones((M, C)) * np.inf, dtype = np.int)

    for i in tqdm(range(len(hash_functions))):
        signatures[i] = list(map(hash_functions[i], list(doc_to_shingles.values())))
    return signatures
```

[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 between characteristic matrix and
# signatures should be at most 1%
# 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%|██████████| 100/100 [00:25<00:00, 3.98it/s]
100%|██████████| 10000/10000 [00:04<00:00, 2209.51it/s]Test passed

1.3 Notes

You may encounter some subtleties when it comes to implementation, please come up with your own design and/or contact Hojoon Lee (joonleesky@kaist.ac.kr) for discussion. Any ideas can be taken into consideration when grading if they are written in the *readme* file.

2 Locality Sensitive Hashing (LSH) [50 points]

The starter code is provided for you to implement the LSH algorithm and analyze it. Fill in the blanks in the code marked as Programming assignments.

2.1 Locality Sensitive Hashing [20 points]

In this section, you will implement the min-hash based LSH algorithm. It receives signature matrix and b (i.e., the number of bands) and r (i.e., the number of rows in each band) as input parameters, and return the candidate pairs of the similar documents.

1. (20 points) Implement the `lsh` function [programming 3.1] to find all candidate pairs of the similar documents using LSH algorithm.

I made a hashtable that works as a bucket, and stored the signature as a key and the document number as a value. So, if the documents have same signature they are stored in the same bucket. In addition, when I chose candidate pairs, I used combinations function from itertools if a bucket has more than two documents.

[CODE]

```
def lsh(signatures, b, r):
    |
    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 i in range(0, M, r):
        hashtables = defaultdict(set)
        for v, k in enumerate(signatures[i:(i+r),:].transpose()):
            hashtables[k.tobytes()].add(v)
        for _, v in hashtables.items():
            if len(v) == 2:
                candidatePairs.add(tuple(sorted(v)))
            elif len(v) > 2:
                candidatePairs = candidatePairs.union(set(combinations(sorted(v), 2)))

    ### Implementation End ###

    return candidatePairs
```

[RESULT]

```
# 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 = 163
0th sample pair: (1658, 5780)
Jaccard similarity: 0.8108108108108109
-----
from paynecrldccom andrew payne messageid organization dec cambridge research lab date tue apr gmt does
anyone know if a source for the modem chips as used in the baycom and my pmp modems ideally something th
at is geared toward hobbyists small quantity mail order etc for years weve been buying them from a distri
butor marshall by the hundreds for pmp kits but orders have dropped to the point where we can no longer a
fford to offer this service and all of the distributors ive checked have some crazy minimum order or so i
d like to find a source for those still interested in building pmp kits any suggestions andrew c payne de
c cambridge research lab
-----
does anyone know if a source for the modem chips as used in the baycom and my pmp modems ideally somethin
g that is geared toward hobbyists small quantity mail order etc for years weve been buying them from a di
stributor marshall by the hundreds for pmp kits but orders have dropped to the point where we can no long
er afford to offer this service and all of the distributors 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
-----
```

Notes: 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.

2.2 Analysis [30 points]

In this section, you will analyze the LSH algorithm in terms of query speed and the various measures of relevance. In this assignment, we fixed M (i.e. the number of min-hash functions) into 100, b to be the divisor of M : b 1, 2, 4, 5, 10, 20, 25, 50, 100, and $s = 0.8$ (i.e. the similarity threshold for checking condition positives).

1. (10 points) Implement the `query_analysis` function [programming 3.2] to compute the query time, precision, recall, and F1 score as b and r change.

[CODE]

```
def query_analysis(signatures, b, s, numConditionPositives):
    |
    M = signatures.shape[0] # The number of min-hash functions
    assert M % b == 0

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

    TruePositives = 0
    for d1, d2 in candidatePairs:
        shingle1, shingle2 = set(doc_to_shingles[d1]), set(doc_to_shingles[d2])
        true_sim = jaccard_similarity(shingle1, shingle2)
        if true_sim >= s :
            TruePositives += 1
    FalsePositives = len(candidatePairs) - TruePositives
    FalseNegatives = numConditionPositives - TruePositives
    TrueNegatives = len(documents) - TruePositives - FalsePositives - FalseNegatives

    precision = TruePositives / (TruePositives + FalsePositives)
    recall = TruePositives / (TruePositives + FalseNegatives)
    f1 = 2 * precision * recall / (precision + recall)
    ## Implementation End ###
    return query_time, precision, recall, f1
```

[RESULT]

```
b_list = find_divisors(M)

query_time_list = list()
precision_list = list()
recall_list = list()
f1_list = list()

for b in tqdm(b_list):
    query_time, precision, recall, f1 = query_analysis(signatures, b, s, numConditionPositives)

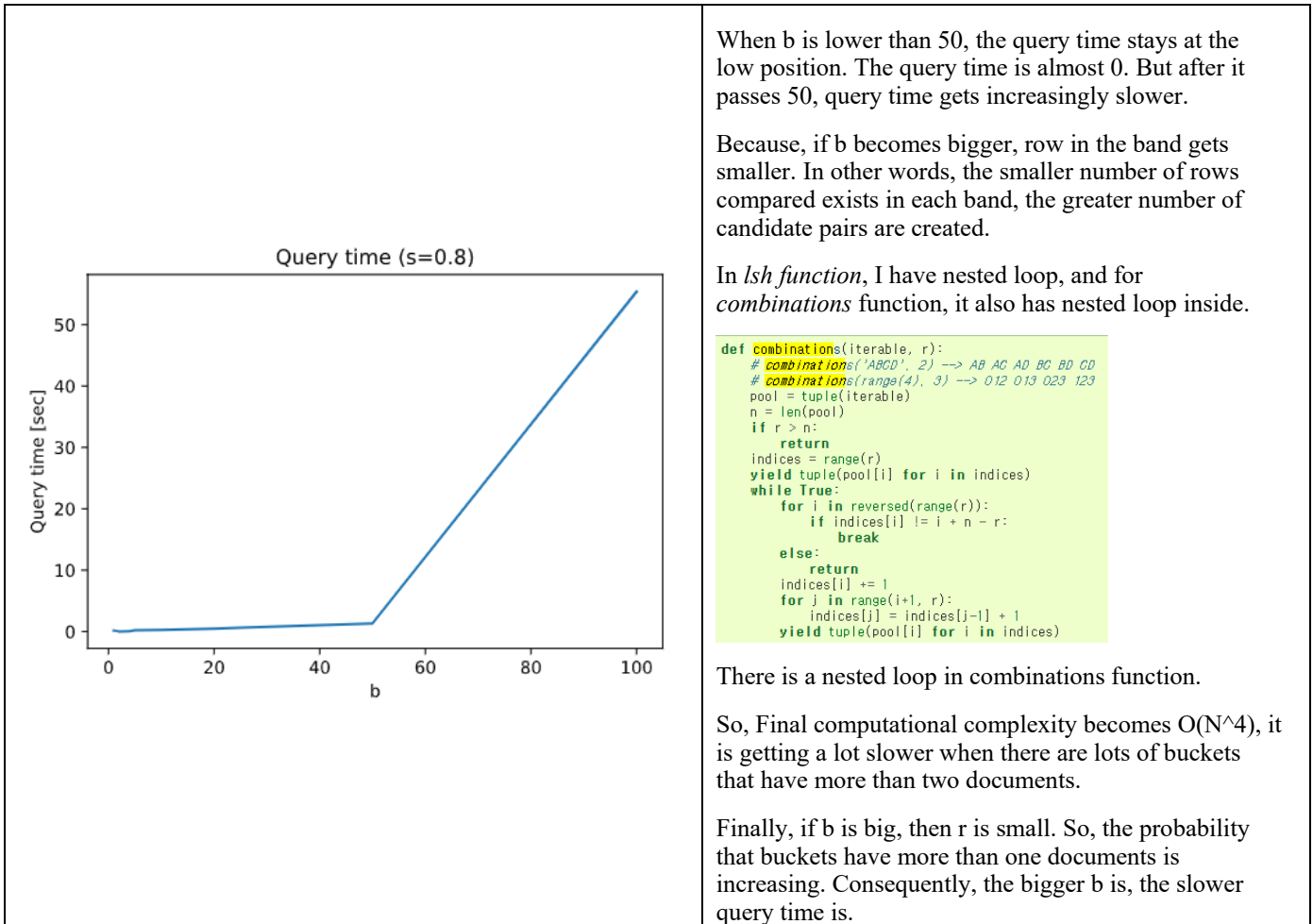
    query_time_list.append(query_time)
    precision_list.append(precision)
    recall_list.append(recall)
    f1_list.append(f1)

100%|██████████| 9/9 [01:14<00:00, 8.30s/it]

print("b: ", b_list)
print("Query times: ", query_time_list)
print("Precisions: ", precision_list)
print("Recalls: ", recall_list)
print("F1 scores: ", f1_list)

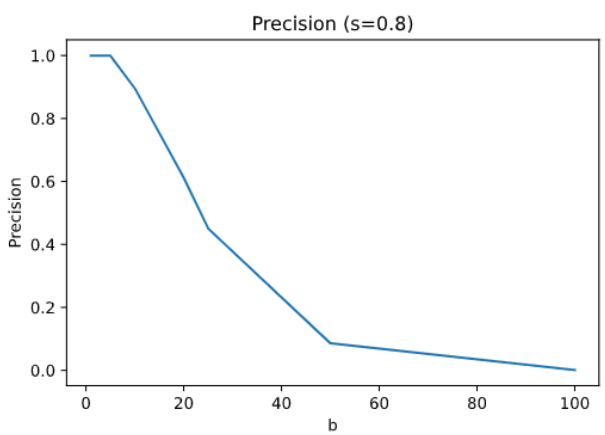
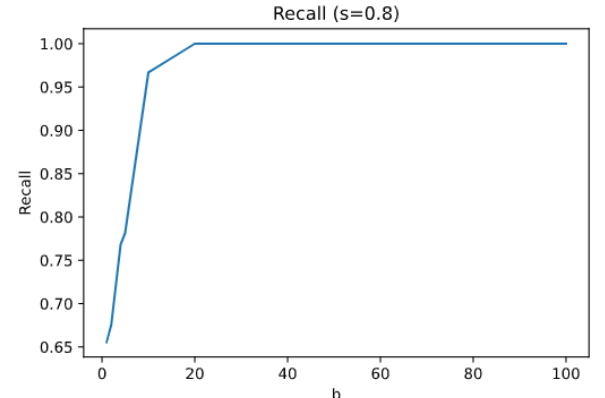
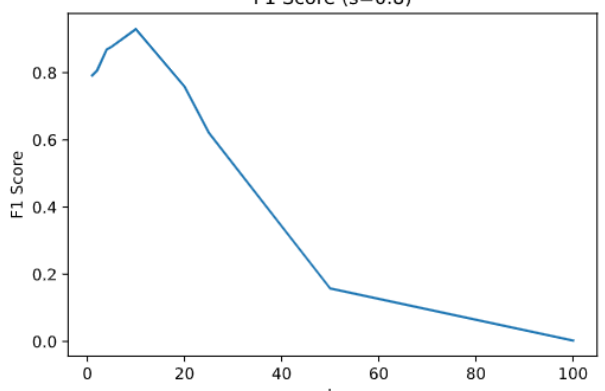
b: [1, 2, 4, 5, 10, 20, 25, 50, 100]
Query times: [0.1825101375579834, 0.023936033248901367, 0.04288506507873535, 0.2214043140411377, 0.24338
316917419434, 0.4587719440460205, 0.6333084106445312, 1.3214657306671143, 55.39484906196594]
Precisions: [1.0, 1.0, 1.0, 1.0, 1.0, 0.8957055214723927, 0.611336032388664, 0.4507462686567164, 0.0856494611
4577425, 0.0012397678103729978]
Recalls: [0.6556291390728477, 0.6754966887417219, 0.7682119205298014, 0.7814569536423841, 0.966887417218
5431, 1.0, 1.0, 1.0, 1.0]
F1 scores: [0.792, 0.8063241106719368, 0.8689138576779027, 0.8773234200743495, 0.9299363057324841, 0.758
7939698492463, 0.6213991769547325, 0.15778474399164052, 0.0024764653786859972]
```


2. (10 points) Attach your query time graph. How does the query speed change as b changes? Explain the results in terms of the computational complexity.



3. (10 points) Attach your precision, recall, and f1-score graphs. How does each measure change as b changes? Which b gives the best result in terms of each measure? Explain the results in detail.

When b is 10, it gives the best results. Because when b is 10, it has the highest F1 score. For detail, I wrote down next to each graph.

 <p>Precision (s=0.8)</p> <table border="1"> <thead> <tr> <th>b</th> <th>Precision</th> </tr> </thead> <tbody> <tr><td>1</td><td>1.00</td></tr> <tr><td>5</td><td>1.00</td></tr> <tr><td>10</td><td>0.90</td></tr> <tr><td>20</td><td>0.65</td></tr> <tr><td>25</td><td>0.45</td></tr> <tr><td>50</td><td>0.10</td></tr> <tr><td>100</td><td>0.02</td></tr> </tbody> </table>	b	Precision	1	1.00	5	1.00	10	0.90	20	0.65	25	0.45	50	0.10	100	0.02	<p>Precision is the number of actual true results divided by the number of results which are predicted positive. Thus, high precision means that it has high performance to find the actual true result. When b is small which means each band has lots of rows, it can specifically compare documents. For example, if there are 100 hash functions and band is 5, there are 20 rows in each band. 20 rows become the decision maker that decides whether documents are similar or not. That is to say, the more rows to compare exist, the less FalsePositives appear.</p> <p>The graph shows that when b is 1, precision is almost 1.</p> <p>However, high precision doesn't mean it absolutely has a good performance. Because, as I mentioned, there will be lots of FalsePositives when b is high. So, we need to take a look at precision and recall together.</p>
b	Precision																
1	1.00																
5	1.00																
10	0.90																
20	0.65																
25	0.45																
50	0.10																
100	0.02																
 <p>Recall (s=0.8)</p> <table border="1"> <thead> <tr> <th>b</th> <th>Recall</th> </tr> </thead> <tbody> <tr><td>1</td><td>0.65</td></tr> <tr><td>5</td><td>0.75</td></tr> <tr><td>10</td><td>0.95</td></tr> <tr><td>20</td><td>1.00</td></tr> <tr><td>100</td><td>1.00</td></tr> </tbody> </table>	b	Recall	1	0.65	5	0.75	10	0.95	20	1.00	100	1.00	<p>Recall is the number of predicted positive results divided by the number of actual true results. High recall means it has good ability to find actual True results.</p> <p>Thus, when b is getting larger which means each band has less rows, it can easily capture similar documents. For example, if b is equal to the number of rows, it only needs to compare one rows, which means that it is easy to be contained in a bucket. That is to say, the less rows to compare exist, the more FalseNeagtives appear.</p> <p>The graph shows that when b is over 20, recall is almost 1.</p> <p>However, as I mentioned in Precision section, high recall does not mean it is absolutely good. We should consider recall and precision together.</p>				
b	Recall																
1	0.65																
5	0.75																
10	0.95																
20	1.00																
100	1.00																
 <p>F1 Score (s=0.8)</p> <table border="1"> <thead> <tr> <th>b</th> <th>F1 Score</th> </tr> </thead> <tbody> <tr><td>1</td><td>0.80</td></tr> <tr><td>5</td><td>0.85</td></tr> <tr><td>10</td><td>0.90</td></tr> <tr><td>20</td><td>0.75</td></tr> <tr><td>25</td><td>0.60</td></tr> <tr><td>50</td><td>0.15</td></tr> <tr><td>100</td><td>0.02</td></tr> </tbody> </table>	b	F1 Score	1	0.80	5	0.85	10	0.90	20	0.75	25	0.60	50	0.15	100	0.02	<p>The Formula of F1 Score is a harmonic mean of recall and precision. It means that F1 score shows a balance point between precision and recall. It means that recall and precision have a good balance at the highest point. Consequently, when b is about 10, F1 score reaches highest point. has a good performance.</p> <p>In other words, when b is 10 it gives the best performance.</p>
b	F1 Score																
1	0.80																
5	0.85																
10	0.90																
20	0.75																
25	0.60																
50	0.15																
100	0.02																

2.3 Notes

- You may encounter some subtleties when it comes to implementation, please come up with your own design and/or contact Inkyu Park (inkyuhak@kaist.ac.kr) for discussion. Any ideas can be taken into consideration when grading if they are written in the *readme* file.