Clare Xinyu Xu
Z5175081
COMP9313 Project 1

C2LSH

Screenshot of My Code

```
1  ## import modules here
2  from pyspark import SparkContext, SparkConf
3  import pickle
4
5  ########## Question 1 ##########
6  # do not change the heading of the function
7  offset=0
8  def count(hashes_1, hashes_2, offset):
9      rdd_1 = hashes_1.map(lambda x: (x[0],[abs(a-b) for a,b in zip(x[1], hashes_2)]))
10     #if satisfy the condition, return the data list, else is null
11     rdd_2 = rdd_1.map(lambda x:(x[0], sum([1 for c in x[1] if c <= offset])))
12     #retrun (key,count)
13     return rdd_2
14
15 def c2lsh(data_hashes, query_hashes, alpha_m, beta_n):
16     offset=0
17     while True:
18         rdd = count(data_hashes,query_hashes,offset)
19         rdd_3 = rdd.filter(lambda x: x[1] >= alpha_m).keys()
20         if rdd_3.count() < beta_n:
21             offset = offset + 1
22         else:
23             break
24     return rdd_3
```

1. Implementation details of c2lsh ()

The algorithm's idea is based on the pseudo-code of C2LSH() in the lecture notes provided in week5. We can treat the for loop in the canGen function as a whole, which means that we can apply the function on rdd instead of separating them into the small part.

In that case, what rdd_1 do is to return an rdd that x [0]-id is the key and absolute value of the difference of each hash1 in data_hashes and hash2 in query_hashes is value. Zip (x[1], hashes2) creates a combination of the pair of data_hashes and query_hashes, and a,b here refers to the hash1 and hash2 in those two hashes.

Followed by rdd_2, we select the absolute value conducted by rdd_1, which satisfies the conditions that less or equal to offset. Since the counter starts from 0 and increases by one every time, we set each one that satisfies the offset conditions to 1 and null if not and sum the counter (which is c here in the code) up. Rdd_2 returns an rdd with the x[0]-id as a key and the sum of the counter as a value.

Therefore, rdd function in c2LSH () performs the count function and return the result the same as each implementation of rdd_2 with the related input. We will not conduct the count function until it meets the conditions that the total number of candidates more significant or equal to alpha_m. Therefore, rdd_3 works on this condition, performing to conduct the filter and return the satisfied keys (id) only in the rdd type.

Line 20 in the screenshot compares the number/length of the id with the minimum

number of candidates to be returned, beta_n. The offset will be increased if we meet the conditions; otherwise, the function will break. Thus, C2LSH() return rdd_3, which is an rdd data type, includes a list of satisfies keys(id) value.

2. Show the evaluation result of your implementation using your own test cases.

```
from pyspark import SparkContext, SparkConf
from time import time
import submission
import random


def createSC():
    conf = SparkConf()
    conf.setMaster("local[*]")
    conf.setAppName("C2LSH")
    sc = SparkContext(conf=conf)
    return sc


random.seed(1)

query_hashes = [random.randint(0, 999) for i in range(100)]
data = []
for i in range(1000):
    random.seed(6666-i)
    data.append((i, [random.randint(0, 999) for j in range(100)]))
alpha_m = 20
beta_n = 10

sc = createSC()
data_hashes = sc.parallelize([(item[0], item[1]) for item in data])
start_time = time()
res = submission.c2lsh(data_hashes, query_hashes, alpha_m, beta_n).collect()
end_time = time()
sc.stop()

print('running time:', end_time - start_time)
print('Number of candidate: ', len(res))
print('set of candidate: ', set(res))
```

Generate my own test with the size of 100 that contains the value from 0 to 999.

Similarly, creating the data with key from 0 to 999 and for each key has the size number of values which the amount is 100.

Setting alpha_m into 20 and beta_n into 10.

Here is my result:

```
data = []
for i in range(1000):
    random.seed(6666-i)
    data.append((i, [random.randint(0, 999) for j in range(100)]))
data
```

Clare Xinyu Xu
Z5175081

3. What did you do to improve the efficiency of your implementation?

Reduce the # of transformations:

I combined the calculation of the count and sum to calculate the counter with the offset into one rdd in the rdd_2 to decrease the number of transformations. Also, the application of zip function and absolute value calculation avoid the excessive number of transformations. In rdd_3, we can straight use keys () to abstract the candidate id into an rdd. Therefore, the total number of rdds has been decreased to improve efficiency.

Reduce the # of shuffling/wide transformation

There is no wide transformation in my implementation which all the map and filter will be in one stage.