COMP9313 Project one Report

1. **Implementation details of your c2lsh(). Explain how your major transform function works.**

First of all, through the understanding of the assessment, Professor's purpose of this project is to help us to understand and apply RDD, and combine it with C2LSH (func) taught in week 5 to carry out a series of operations to obtain the required content. Sc. Take and Collect e.g. functions cannot be used in order not to dismantle RDD, but to allow all programming to only operate within RDD. Meanwhile, according to the attributes of RDD to improve the efficiency of data processing.

The way I thought about it was that since transformations to RDD are lazy-loaded, that is, they don't directly compute the results. Instead, they simply remember the transformation actions applied to the underlying dataset (such as a file). Only when an Action that returns a result occurs will these actions actually run. This design will make Spark run more efficiently.

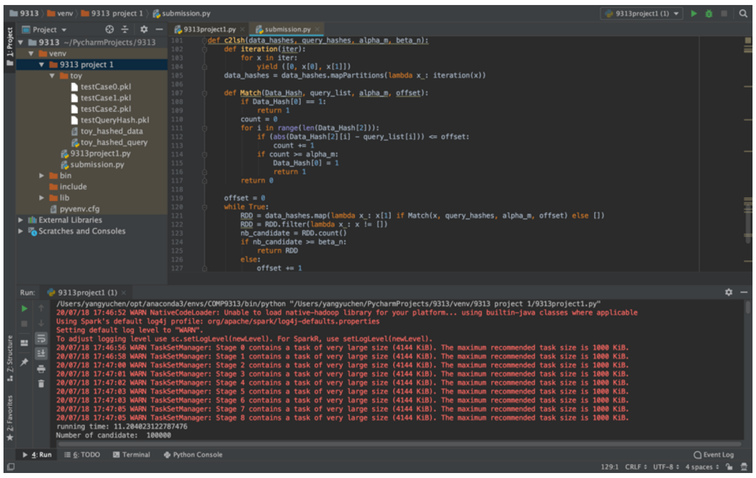
So for efficiency, I'm going to try to avoid using action and focus on calling an existing RDD to generate a new ONE, and then fetching what I need.

C2LSH (func) import is a RDD, one list and two constraint conditions, to understand the structure of RDD, found RDD is composed of ID and data_hash list, and I need to do is comparing query_hash elements in the list whether with data_hash list, if do not conform to, used to offset to get the offset and store offsets, again to find the difference to traverse the entire list to find the right conditions of the element

So my basic code works like this, using lambda conditions to grab a list in RDD, compare elements in query_hash list, grab elements that meet the offset criteria, compare them with Alpha_m, and find the number of matches. If the alpha_m number does not match, increase/decrease the offset and iterate. Filter the new RDD until the number meets the alpha_m condition, otherwise discard the list. Count the number of conditions in RDD until the second constraint is met.

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

I'm using a 2.7ghz dual-core Core I5 processor with 8GB of ram and a 256GB MAC. When I run my current code on Pycharm, A 32-bit HashCode that runs 200,000 data and picks 10,000 candidates at 11.2S. The runtime is shown in the figure:



```python
def c2lsh(data_hashes, query_hashes, alpha_m, beta_n):
    def iteration(iter):
        for x in iter:
            yield ([0, x[0], x[1]])
    data_hashes = data_hashes.mapPartitions(lambda x_: iteration(x))

    def Match(Data_Hash, query_list, alpha_m, offset):
        if Data_Hash[0] == 1:
            return 1
        count = 0
        for i in range(len(Data_Hash[2])):
            if (abs(Data_Hash[2][i] - query_list[i])) <= offset:
                count += 1
            if count >= alpha_m:
                Data_Hash[0] = 1
                return 1
        return 0

    offset = 0
    while True:
        RDD = data_hashes.map(lambda x_: x[1] if Match(x, query_hashes, alpha_m, offset) else [])
        RDD = RDD.filter(lambda x_: x != [])
        nb_candidate = RDD.count()
        if nb_candidate >= beta_n:
            return RDD
        else:
            offset += 1
```

```
/Users/yangyuchen/opt/anaconda3/envs/COMP9313/bin/python "/Users/yangyuchen/PycharmProjects/9313/venv/9313 project 1/9313project1.py"
20/07/18 17:46:52 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
20/07/18 17:46:56 WARN TaskSetManager: Stage 0 contains a task of very large size (4144 KiB). The maximum recommended task size is 1000 KiB.
20/07/18 17:46:58 WARN TaskSetManager: Stage 1 contains a task of very large size (4144 KiB). The maximum recommended task size is 1000 KiB.
20/07/18 17:47:00 WARN TaskSetManager: Stage 2 contains a task of very large size (4144 KiB). The maximum recommended task size is 1000 KiB.
20/07/18 17:47:01 WARN TaskSetManager: Stage 3 contains a task of very large size (4144 KiB). The maximum recommended task size is 1000 KiB.
20/07/18 17:47:02 WARN TaskSetManager: Stage 4 contains a task of very large size (4144 KiB). The maximum recommended task size is 1000 KiB.
20/07/18 17:47:03 WARN TaskSetManager: Stage 5 contains a task of very large size (4144 KiB). The maximum recommended task size is 1000 KiB.
20/07/18 17:47:03 WARN TaskSetManager: Stage 6 contains a task of very large size (4144 KiB). The maximum recommended task size is 1000 KiB.
20/07/18 17:47:05 WARN TaskSetManager: Stage 7 contains a task of very large size (4144 KiB). The maximum recommended task size is 1000 KiB.
20/07/18 17:47:05 WARN TaskSetManager: Stage 8 contains a task of very large size (4144 KiB). The maximum recommended task size is 1000 KiB.
running time: 11.204023122787476
Number of candidate:  100000
```

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

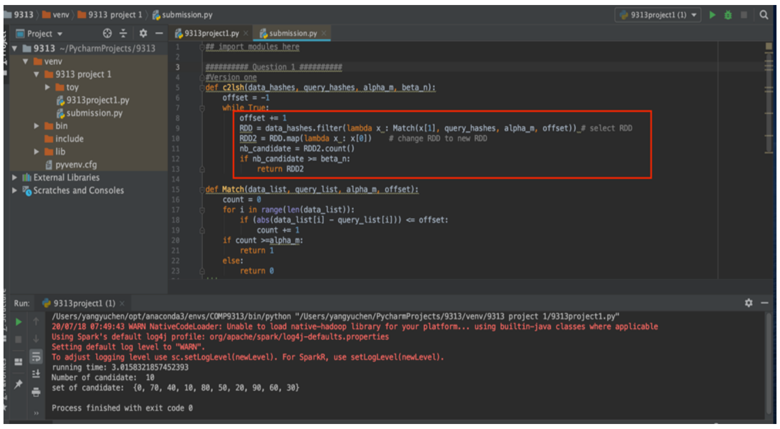
My actions on RDD focus on common action transformations and actions. On how to improve the efficiency of my code, I am currently working on 4 versions:

1） Version 1 — According to the Psedudo code to complete the program

The first version is the establishment of the initial version, using two transformation functions, fliter(func) and map(func).The filter(func) is mainly used to filter RDD and select eligible RDD. Considering the limitation of the conditional RDD of the final output (collect cannot be used to transform RDD), I used map to iterate through the ID collection of eligible RDD.

Finally, the betA_N condition in the title is used to check the output condition. If it is not consistent, the data will be re-entered into the loop, and the qualified RDD will be searched through comparison with query_hashes list and offset increase until success.

The time to pass the Project sample test is 3.016S.



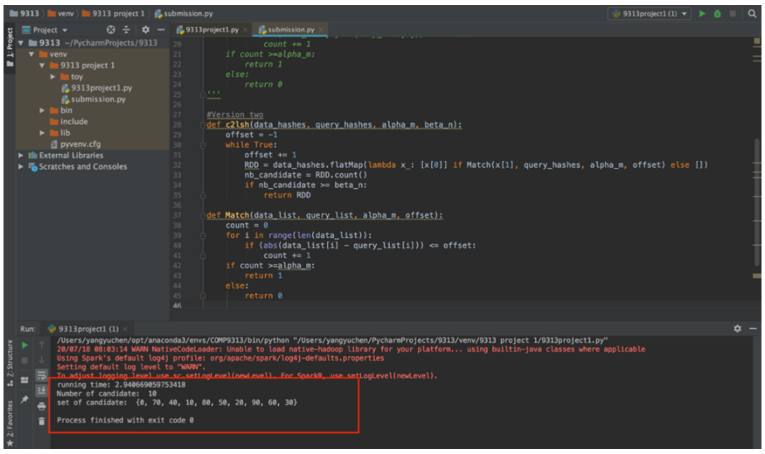
```python
## import modules here

########## Question 1 ##########
#Version one
def c2lsh(data_hashes, query_hashes, alpha_m, beta_n):
    offset = -1
    while True:
        offset += 1
        RDD = data_hashes.filter(lambda x_: Match(x[1], query_hashes, alpha_m, offset))_# select RDD
        RDD2 = RDD.map(lambda x_: x[0])    # change RDD to new RDD
        nb_candidate = RDD2.count()
        if nb_candidate >= beta_n:
            return RDD2

def Match(data_list, query_list, alpha_m, offset):
    count = 0
    for i in range(len(data_list)):
        if (abs(data_list[i] - query_list[i])) <= offset:
            count += 1
    if count >=alpha_m:
        return 1
    else:
        return 0
```

```
/Users/yangyuchen/opt/anaconda3/envs/COMP9313/bin/python "/Users/yangyuchen/PycharmProjects/9313/venv/9313 project 1/9313project1.py"
20/07/18 07:49:43 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
running time: 3.0158321857452393
Number of candidate:  10
set of candidate:  {0, 70, 40, 10, 80, 50, 20, 90, 60, 30}

Process finished with exit code 0
```

2） Version 2 —Change map and filter these two transformation function to one flatMap transformation function.

After finishing the first version, I wondered whether upgrading fliter(func) and map(func) to flatMap(func) would improve its efficiency. Because flatMap(func) is a map-like function and each input element can be mapped to 0 or more output elements. Unlike the output of a map(func), it returns a sequence instead of a single element. Fortunately, this does not affect the calculation of the number of qualified RDD. I can still use count(func) as the action function, to count the qualified RDD.

So I made the second version, and by comparing it to the first version, I found that it really made the whole code more efficient. The time to pass the Project sample test is 2.940S.



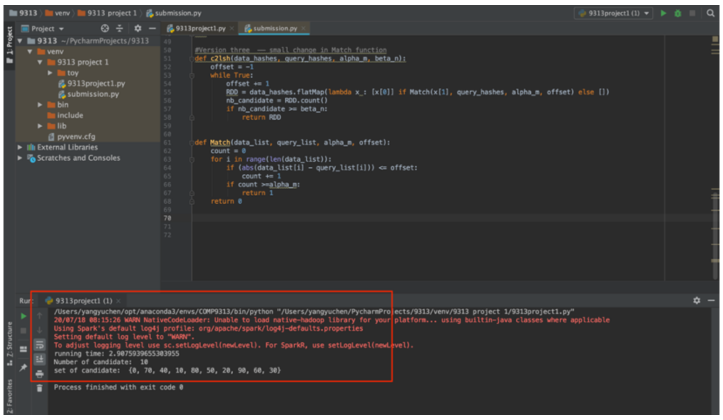
```python
            count += 1
    if count >=alpha_m:
        return 1
    else:
        return 0
'''

#Version two
def c2lsh(data_hashes, query_hashes, alpha_m, beta_n):
    offset = -1
    while True:
        offset += 1
        RDD = data_hashes.flatMap(lambda x_: [x[0]] if Match(x[1], query_hashes, alpha_m, offset) else [])
        nb_candidate = RDD.count()
        if nb_candidate >= beta_n:
            return RDD

def Match(data_list, query_list, alpha_m, offset):
    count = 0
    for i in range(len(data_list)):
        if (abs(data_list[i] - query_list[i])) <= offset:
            count += 1
    if count >=alpha_m:
        return 1
    else:
        return 0
```

```
/Users/yangyuchen/opt/anaconda3/envs/COMP9313/bin/python "/Users/yangyuchen/PycharmProjects/9313/venv/9313 project 1/9313project1.py"
20/07/18 08:03:14 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
running time: 2.940669059753418
Number of candidate:  10
set of candidate:  {0, 70, 40, 10, 80, 50, 20, 90, 60, 30}

Process finished with exit code 0
```

3） Version 3 — Changed the position of condition

I found that when I traversed the RDD whether the data list matches the Query List, I placed the judgment condition(if) at the end of the loop, not inside the loop. In this way, when the program finds the solution, it cannot stop in time, and it needs to complete the traversal before it can be judged. However, if I put it inside the loop, each iteration would require o(1) time complexity judgment, and I was not sure if I could improve the efficiency, so I tried version 3.

Experiments have shown that this slight but effective change reduces the time from 2.9406 second to 2.9075 second. Although taking the data of example as an example, such a change is very small, I believe it will be a huge change in the face of big data.

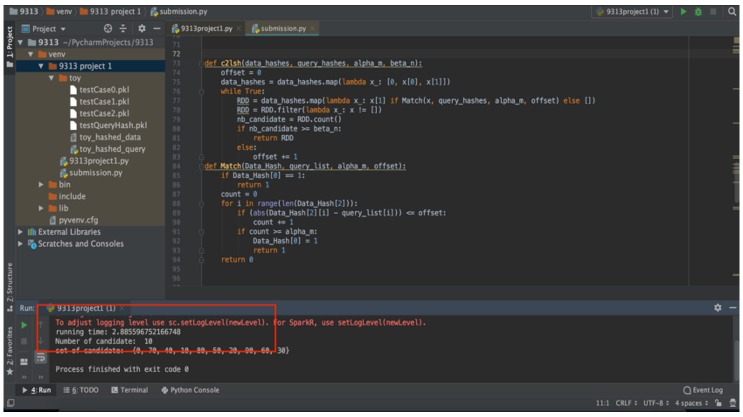


```python
#Version three  — small change in Match function
def c2lsh(data_hashes, query_hashes, alpha_m, beta_n):
    offset = -1
    while True:
        offset += 1
        RDD = data_hashes.flatMap(lambda x_: [x[0]] if Match(x[1], query_hashes, alpha_m, offset) else [])
        nb_candidate = RDD.count()
        if nb_candidate >= beta_n:
            return RDD


def Match(data_list, query_list, alpha_m, offset):
    count = 0
    for i in range(len(data_list)):
        if (abs(data_list[i] - query_list[i])) <= offset:
            count += 1
        if count >=alpha_m:
            return 1
    return 0
```

```
/Users/yangyuchen/opt/anaconda3/envs/COMP9313/bin/python "/Users/yangyuchen/PycharmProjects/9313/venv/9313 project 1/9313project1.py"
20/07/18 08:15:26 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
running time: 2.9075939655303955
Number of candidate:  10
set of candidate:  {0, 70, 40, 10, 80, 50, 20, 90, 60, 30}

Process finished with exit code 0
```

4） Version 4 — Ignore these elements which have been tested with matched

I found that although flatmap's function was efficient, I still needed to traverse all the traversed lists at a time, so I tried adding an element to RDD. I make a mark when the list data already meets the offset criteria. The next traversal does not traverse it. It changed the time from 2.9075 second to 2.8856 second.

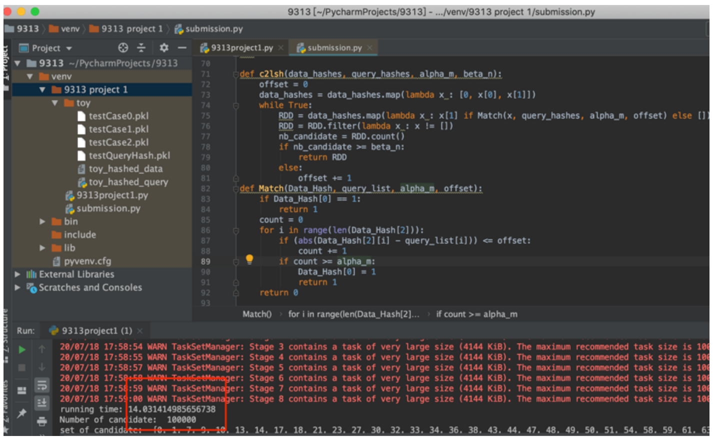


```python
def c2lsh(data_hashes, query_hashes, alpha_m, beta_n):
    offset = 0
    data_hashes = data_hashes.map(lambda x_: [0, x[0], x[1]])
    while True:
        RDD = data_hashes.map(lambda x_: x[1] if Match(x, query_hashes, alpha_m, offset) else [])
        RDD = RDD.filter(lambda x_: x != [])
        nb_candidate = RDD.count()
        if nb_candidate >= beta_n:
            return RDD
        else:
            offset += 1
def Match(Data_Hash, query_list, alpha_m, offset):
    if Data_Hash[0] == 1:
        return 1
    count = 0
    for i in range(len(Data_Hash[2])):
        if (abs(Data_Hash[2][i] - query_list[i])) <= offset:
            count += 1
        if count >= alpha_m:
            Data_Hash[0] = 1
            return 1
    return 0
```

```
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
running time: 2.885596752166748
Number of candidate:  10
set of candidate:  {0, 70, 40, 10, 80, 50, 20, 90, 60, 30}

Process finished with exit code 0
```

5） Version 5 — Use MapPartitions function instead of Map

When I first made this substitution, I found that the running time difference between the two versions was not big when I used the professor's Project sample. However, once big data is involved, for example, when I need to run 200,000 data, version4 takes 14 seconds which pic showed as followed, while Version5 only takes 11.2 seconds, which obviously expands the running time. Therefore, mapPartition(func) as data repartition can significantly improve the efficiency of the algorithm.



```python
def c2lsh(data_hashes, query_hashes, alpha_m, beta_n):
    offset = 0
    data_hashes = data_hashes.map(lambda x_: [0, x[0], x[1]])
    while True:
        RDD = data_hashes.map(lambda x_: x[1] if Match(x, query_hashes, alpha_m, offset) else [])
        RDD = RDD.filter(lambda x_: x != [])
        nb_candidate = RDD.count()
        if nb_candidate >= beta_n:
            return RDD
        else:
            offset += 1
def Match(Data_Hash, query_list, alpha_m, offset):
    if Data_Hash[0] == 1:
        return 1
    count = 0
    for i in range(len(Data_Hash[2])):
        if (abs(Data_Hash[2][i] - query_list[i])) <= offset:
            count += 1
        if count >= alpha_m:
            Data_Hash[0] = 1
            return 1
    return 0
```

```
20/07/18 17:58:54 WARN TaskSetManager: Stage 3 contains a task of very large size (4144 KiB). The maximum recommended task size is 10
20/07/18 17:58:55 WARN TaskSetManager: Stage 4 contains a task of very large size (4144 KiB). The maximum recommended task size is 10
20/07/18 17:58:57 WARN TaskSetManager: Stage 5 contains a task of very large size (4144 KiB). The maximum recommended task size is 10
20/07/18 17:58:58 WARN TaskSetManager: Stage 6 contains a task of very large size (4144 KiB). The maximum recommended task size is 10
20/07/18 17:58:59 WARN TaskSetManager: Stage 7 contains a task of very large size (4144 KiB). The maximum recommended task size is 10
20/07/18 17:59:00 WARN TaskSetManager: Stage 8 contains a task of very large size (4144 KiB). The maximum recommended task size is 10
running time: 14.031414985656738
Number of candidate:  100000
set of candidate:  {0, 1, 7, 9, 10, 13, 14, 17, 18, 21, 23, 27, 30, 32, 33, 34, 36, 38, 43, 44, 47, 48, 49, 50, 51, 54, 58, 59, 61, 6
```

Conclusion

In fact, I also tried using union and distinct functions between RDD, But they were not particularly effective. The overall feeling of improving is that The size of data is too small and CPU is not enough. The current changing is all based on theoretical analysis It is really a big data operation, the time difference between the results and a big CPU will be larger.

Personally, I have read a lot, But I do not know RDD deeply enough. flatmap(func) and mappartition (func) can be clearly shown to be effective.

I do think the current Adjustment has not truly changed the efficiency of the algorithm. For real big data, It should be done according to the offset, data form and enviroment to improve the algorithm.