💯

Mining Big Data Assignment 4

`Exercise 2 Frequent Itemsets:`

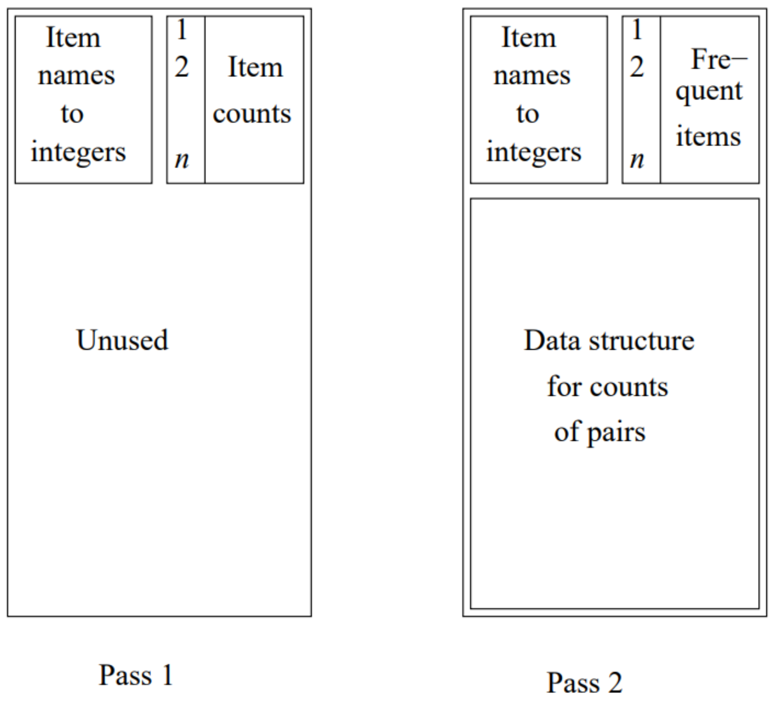
`Exercise 2.1:`

- SIMPLE RANDOMIZED ALGOTIRHM IMPLEMENTATION:

- What is simple randomized algorithm and what is A-priori algorithm:

The simple randomized algorithm is useful for finding most frequent itemsets if the main memory is too small to hold the data. The algorithm is nothing but using a random subset of the baskets , instead of using the entire file of baskets. The support threshold will be adjusted according to the sample size. For example: if the support threshold for the full dataset is `s`, and we choose a sample of `1%` of the baskets, then we should examine the sample for itemsets that appear in at least `s/100`of the baskets. The most important thing for randomized algorithm is to pick samples. If we can be sure that the baskets appear in random order in the file, then we can just pick first `m` baskets. For example, if we want the sample size to be 1% and the full dataset has `10000` basket, then we only need to pick first `100` baskets. However, if the baskets does not appear in random order in the file, we have to randomly select baskets. One way of doing that is to set a fixed probability `p` and we can read the entire dataset and for each basket , select that basket for the sample with `P`.

The A-priori algorithm can be used to find all the frequent itemsets. For now, let us concentrate on finding the frequent 2-itemsers only. The A-priori consists of two passes and two tables can be generated during each passes:



First pass of A-priori:

In the first pass, we first generate two tables(arrays). the first table contains all the item, the second table contains the occurrences of each item. For example, if items are {1,2,3,4,5,6} and their occurrences in all the baskets are {5,2,7,4,7,8}, then the first table: [1,2,3,4,5,6], and second table:[5,2,7,4,7,8]. By using two tables, memory can be saved since the contents in each table can be directly accessed through the index.

Between the Passes of A-Priori:

After the first pass, we examine the counts of the items to determine which of them are frequent as singletons.

The Second Pass of A-Priori:

During the second pass, we count all the pairs that consist of two frequent items. We know that a pair cannot be frequent unless both its members are frequent. And the space required on the second pass is `2m^2` bytes, rather than `2n^2` bytes, if we use the triangular matrix method for counting.

Coding part:

```
def randomized_algo(perct, min_support, transaction_list):
    num_samples = round(len(transaction_list) * perct/100)
    sample_set  = random.sample(transaction_list,num_samples)
    freq_itemset = Apriori_algo(min_support , sample_set)

    return(freq_itemset,ran_sample )
```

> The code above demonstrated the implementation of simple randomized algorithm.

> 1. Calculate the number of samples we need based on the percentage given. For example: if the total number of data set is 10000, the sample size should be 1000 given that percentage = 1%.
> 2. Use random sample method to randomly draw samples from original dataset.
> 3. Pass the min support threshold and sample set into A-priori algorithm to get frequent itemset .

```
def Apriori_algo(min_support, transaction_list ):
    result_count = dict(Counter(i for sub in transaction_list for i in set(sub)))
    frequent_1_itemset = [key for key, value in result_count.items() if value >=min_support]
    freq_itemset = frequent_1_itemset

    freq_itemset = freq_itemset_finder(freq_itemset=freq_itemset, transaction_set=transaction_list, min_support=min_support)
    return(freq_itemset)
```

> The code above demonstrated the implementation of A-priori algorithm.
>
> 1. Count all the singleton items along with their support values. The item and its respective support value will be stored as key-value pair in a dictionary. For example, {1,2,3,2,5, 2, 6} is a subset , then the singleton 1 has support value 1, singleton 2 has support value 3 ….
> 2. For each singleton item, check if its support value > min support threshold. If it does, keep it in a list.
> 3. Pass all the singletons which have support value greater than support threshold into frequent itemset finder function to generate all the frequent pairs.

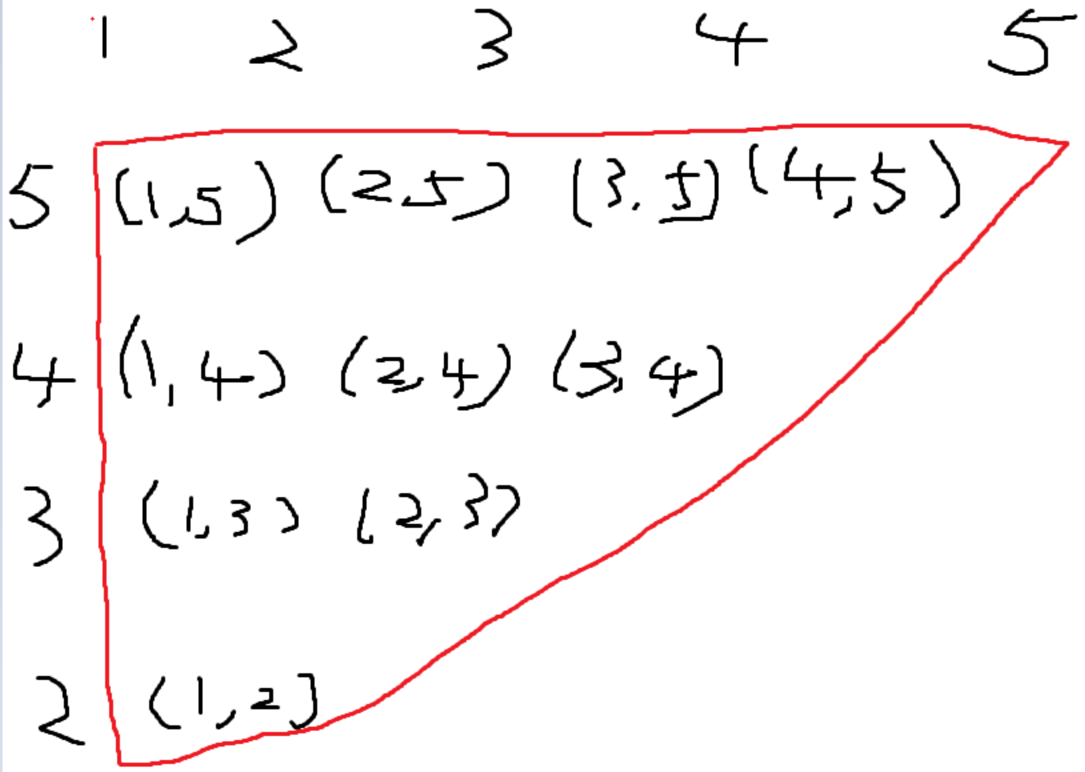
```
def pair_items(freq_itemset):
    item_set = []
    freq_itemset_1 = freq_itemset[::-1]

    for i in range(len(freq_itemset)):
        for j in range(len(freq_itemset_1)-1):
            if freq_itemset[i] != freq_itemset_1[j]:
                item_set.append(tuple([freq_itemset[i], freq_itemset_1[j]]))
            else:
                break
    return item_set
```

```
def freq_itemset_finder(freq_itemset, transaction_set, min_support):
    item_set=[]

    item_set = pair_items(freq_itemset)
    item_set= list(set(item_set))
    new_freq_itemset = []
    for item in item_set:
        j=0
        for outer in transaction_set:
            if(set(item).issubset(outer)):
                j=j+1
        if(j>min_support):
            new_freq_itemset.append([item,j])

    return(new_freq_itemset)
```



> The code above demonstrated the implementation of frequent itemset finder method
>
> 1. Pair each singleton items. For example: {1,2,3,4,5} can be paired as: (1,2), (1,3), (1,4), (1,5), (2,3), (2,4), (2,5), (3,4), (3,5) this essentially generate all the candidate set. The pair items method used triangular matrix method. In this way, space required can be reduced from `2n^2` bytes to `2m^2` bytes.
> 2. For each pair in the candidate set, count the occurrences of it in all the sample sets. If the occurrences(support value) is greater than min support value, then we keep it in a list. The pair gets kept in the list are the frequent itemset.

`Exercise 2.2:`

- SON ALGORITHM IMPLEMENTATION:

- What is SON algorithm:

The SON algorithm lends itself well to a parallel-computing environment. Each of the chunks can be processed in parallel, and the frequent itemsets from each chunk combined to form the candidates.

First Map Function:

A portion of baskets will be passed to map, and the map function just simply count the occurrences of each item in those baskets, if the occurrence(support value) is greater than min support value, then the item(key) will be passed to reduce function. Since we're only dealing with a portion of baskets, the support threshold should be adjusted accordingly.

First Reduce Function:

In the reduce function, all the frequent singletons passed from map will be paired to generate candidate itemsets.

Second Map Function:

The Map tasks for the second Map function take all the output from the first Reduce Function (the candidate itemsets) and a portion of the input data file. Then the map function simply count the number of occurrences of each of the candidate itemsets among the baskets in the portion of the dataset that it was assigned. Then, each candidate itemset and its corresponding occurrence number(support value) will be passed to the second reduce function.

Second Reduce Function:

In this reduce function, the support value of each candidate itemset gets summed up and those
itemsets whose sum of support value is at least `s` (min support value) are frequent in the whole dataset. Itemsets that do not have total support at least s are not transmitted to the output of the Reduce task.

Coding part:

```
def chunks(lst, n):
    for i in range(0, len(lst), n):
        yield lst[i:i + n]

def divide_baskets(transaction_list, num_in_baskets):
    new_transaction_list = list(chunks(transaction_list, num_in_baskets))
    num_chunks = (len(new_transaction_list))
    return new_transaction_list, num_chunks
```

```
def SON_algo(min_support_perct, new_transaction_list, num_chunks):
    frequent_1_itemset = []
    min_sup = (1/num_chunks * min_support_perct)
    for chunk in new_transaction_list:
        result_count = dict(Counter(i for sub in chunk for i in set(sub)))
        for key, value in result_count.items():
            if value >= min_sup and key not in frequent_1_itemset:
                frequent_1_itemset.append(key)

    freq_itemset = frequent_1_itemset
    # print(freq_itemset)
    freq_itemset = freq_itemset_finder(freq_itemset=freq_itemset, transaction_set=new_transaction_list, min_support_perct=min_support_perct

    return(freq_itemset)
```

> The code above demonstrated the implementation of the SON algorithm
>
> 1. The original dataset will firstly be passed to the divide baskets method. The return value of divided baskets will be several chunks of baskets. This process is aimed at simulating parallel computing.
> 2. The chunks of baskets, the number of chunks, and min support will then be passed to the SON algorithm to generate frequent itemsets. The min support value for each chunk will be adjusted to (1/number of chunks) * min support.

> 3. The support value for each item that appears in each chunk is then calculated and compared with the adjusted support threshold. If the item's support value is greater than the support threshold, it gets kept.
>
> The above procedures correspond to the first map task
>
> 4. The frequent singleton sets will then be passed to the frequent itemset finder method to generate frequent pair sets. (pass first map task outputs to the first reducer)

```
def freq_itemset_finder(freq_itemset, transaction_set, min_support_perct):
    item_set=[]

    item_set = pair_items(freq_itemset)
    item_set= list(set(item_set))
    new_freq_itemset = {}
    result = []
    for chunk in transaction_set:
        for item in item_set:
            j=0
            for outer in chunk:
                if(set(item).issubset(outer)): # if the item set is a subset of transction set
                    j=j+1
            if item in new_freq_itemset:
                new_freq_itemset[item] = new_freq_itemset[item]+j
            else:
                new_freq_itemset[item] = j

    for key, value in new_freq_itemset.items():
        if value >= min_support_perct:
            result.append([key,value])

    return(result)
```

> The code above demonstrated the implementation of a frequent itemset finder.
>
> 1. Frequent singleton item sets will be passed to the pair items method which uses the triangular matrix to generate all candidate pair sets.
> 2. The support value of each candidate pair will then be calculated by comparing all the items in each chunk. If the candidate pair is a subset of a basket in one chunk, then its support value will increment.
>
> The above procedures correspond to second map task
>
> 3. After scanning all the chunks, the support value of each candidate pair gets added up. Those pairs whose sum of values is at least s(min support value) are frequent in the whole dataset
>
> The above procedure corresponds to second reducer task

`Exercise 2.3:`

- Observations:

Since both T10I4D100K and T40I10D100K datasets have a total of 100,000 baskets with various items inside, the time taken for running the SON algorithm would be a bit longer than running the simple random algorithm. However, for the rest of the datasets, the items that appeared in each basket are very similar, hence the time taken for the SON algorithm is very close to the simple

random algorithm. Since the simple random algorithm only takes a portion of baskets and pretends it to be the entire dataset, false positive and false negative may be introduced. The SON algorithm can eliminate false positives and false negatives since the candidate itemsets are examined against the whole dataset.
To further verify the simple random algorithm has false positive and false negative, let us take T40I10D100K as an example.

```
T40I10D100K :

SIMPLE RANDOMIZED ALGOTIRHM min support threshold = 6000 and sample size = 1%
Results:
[[('368', '510'), 62], [('368', '829'), 64], [('368', '489'), 65], [('368', '722'), 63], [('368', '682'), 68]]

SON ALGORITHM min support threshold = 6000

Results:
[[('368', '529'), 7500], [('489', '368'), 6120], [('368', '217'), 6125], [('368', '829'), 6957], [('682', '368'), 6130]]
```

> The results shown above are the outcome of the dataset of T40I10D100K processed in a simple random algorithm with sample size = 1% and min support threshold = 6000 and SON algorithm with min support threshold = 6000.
>
> We can see that the frequent 2-itemsets (368, 510) and (368, 722) have been determined in a simple sample algorithm, but not in the SON algorithm. Moreover, frequent 2-itemsets (368,529) and (368,217) have been determined in the SON algorithm, but not in the simple random algorithm. Hence, we conclude that false positive and false negative has occurred in the simple random algorithm.

`Exercise 2.4:`

- Comparation on individual datasets:

- T10I4D100K:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/b6f1c2c5-d78e-439c-b445-db8ba99f9930/output_T10I4D100K.dat.txt

> The T10I4D100K dataset has a total of 100,000 baskets with various lengths, which denotes this is a sparse dataset. By setting a fixed min support threshold of 1200 and varying the sample size to be 1%, 2%, 5%, and 10%, the number of frequent 2-itemsets is 11, 7, 6, and 1 respectively for the simple sample algorithm. Only 1 frequent 2-itemsets gets picked out for the SON algorithm. Hence, we observed that as the sample size increases, the number of frequent 2-itemsets decreases. The frequent 2-itemsets found approaches to true value when the sample size is large enough. Another observation is that the time taken for the simple sample algorithm will increase as the sample size increases. For instance: 1% sample size takes about 25sec and 2% sample size takes about 48 sec.

- T40I10D100K:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/a58b85f5-c65d-4be4-a1b6-e74392498f79/output_T40I10D100K.dat.txt

> The T40I10D100K dataset has a total of 100,000 baskets of almost the same lengths, which denotes this is a dense dataset. By setting a fixed min support threshold of 6000 and varying the sample size to be 1%, 2%, 5%, and 10%, the number of frequent 2-itemsets is 5, 1, 4, and 5 respectively for the simple sample algorithm. 5 frequent 2-itemsets were picked out for the SON algorithm. Hence, we observed that as the sample size increases, the number of frequent 2-itemsets doesn't vary much. Still, the frequent 2-itemsets found approaches to true value when the sample size is large enough. Another observation is that the time taken for the simple sample algorithm will increase as the sample size increases. For instance: 1% sample size takes about 33sec and 2% sample size takes about 75 sec.

- chess:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/b399bf83-a6ee-487a-8f44-a6d326a9d2c8/output_chess.dat.txt

> The chess dataset has a total of 3,196 baskets of the same lengths, which denotes this is a dense dataset. Unlike the previous datasets, this dataset has almost identical items in each basket. By setting a fixed min support threshold of 3000 and varying the sample size to be 1%, 2%, 5%, and 10%, the number of frequent 2-itemsets is 52, 39, 33, and 49 respectively for the simple sample algorithm. 38 frequent 2-itemsets were picked out for the SON algorithm. Hence, we observed that as the sample size increases, the number of frequent 2-itemsets varies a lot. The frequent 2-itemsets found cannot approach the true value when the sample size gets larger. The true value can only be found when the sample size is equal to the original dataset's size. Another observation is that a large number of frequent 2-itemsets have been picked out even though a high support threshold value has been set. This is simply due to identical items repeatedly appearing in every basket.

- connect:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/2645b6ff-9c90-4fa2-9737-6db2f07c9255/output_connect.dat.txt

> The connect dataset has a total of 67557 baskets of the same lengths, which denotes this is a dense dataset. Just like the chess dataset, this dataset has almost identical items in each basket. By setting a fixed min support threshold of 67000 and varying the sample size to be 1%, 2%, 5%, and 10%, the number of frequent 2-itemsets is 10, 7, 8, and 11 respectively for a simple sample algorithm. 6 frequent 2-itemsets were picked out for the SON algorithm. Hence, we observed that simply increasing the sample size cannot achieve true values for frequent 2-itemsets. The true value can only be found when

> the sample size is equal to the original dataset's size. Another observation is as we increase the min support value to be very close to dataset size, the number of frequent 2-itemsets doesn't get so large as chess dataset.

- mushroom:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/8222fbaa-e16d-4552-a39c-c0737772df8f/output_mushroom.dat.txt

> The mushroom dataset has a total of 8124 baskets with mostly the same lengths, which denotes this is a dense dataset. By browsing this dataset, we found a lot of repeated items. By setting a fixed min support threshold of 6000 and varying the sample size to be 1%, 2%, 5%, and 10%, the number of frequent 2-itemsets is 9, 10, 10, and 10 respectively for the simple sample algorithm. 10 frequent 2-itemsets were picked out for the SON algorithm. Hence, we observed that as the sample size increases, the number of frequent 2-itemsets increases and would eventually approaches to true value..

- pumsb:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/ca59f989-77ae-4609-88e4-9421d6c9601f/output_pumsb.dat.txt

> The pumsb dataset has a total of 49046 baskets with the same long lengths, which denotes this is a dense dataset. By browsing this dataset, we found a lot of repeated items. By setting a fixed min support threshold of 48000 and varying the sample size to be 1%, 2%, 5%, and 10%, the number of frequent 2-itemsets is 8, 6, 6, and 7 respectively for a simple sample algorithm. 7 frequent 2-itemsets were picked out for the SON algorithm. Hence, we observed that as the sample size increases, the number of frequent 2-itemsets doesn't vary much. The true value can be achieved when the sample size is set to be 10%.

- pumsb star:

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/364b28ef-ceb4-4d7a-ad4d-dad3256d7c1a/output_pumsb_star.dat.txt

> The pumsb star dataset has a total of 49046 baskets with mostly the same long lengths, which denotes this is a dense dataset. By browsing this dataset, we found a lot of repeated items. By setting a fixed min support threshold of 35000 and varying the sample size to be 1%, 2%, 5%, and 10%, the number

> of frequent 2-itemsets is 4, 10, 5, and 5 respectively for a simple sample algorithm. 5 frequent 2-itemsets were picked out for the SON algorithm. Hence, we observed that as the sample size increases, the number of frequent 2-itemsets varies and the true value cannot be achieved when the sample size is 10%. Another observation is that none of the frequent 2-itemsets can be found when the threshold is set to be 48000. This is different than the pumsb dataset, which means although the size of datasets is the same for both pumsb and pumsb star, less frequent itemsets appeared in the pumsb star dataset.

- Overall observations:

1. A dataset that has baskets of almost the same lengths is treated as a dense dataset. By browsing these 7 datasets, other than T10I4D100K and T40I10D100K datasets, a lot of repeated items can be found within their baskets. This gives us a sense that a huge number of frequent 2-itemsets will appear in those datasets. Therefore, setting a high support threshold can eliminate some potential candidate frequent 2-itemsets and reduce the time taken for both simple sample and SON algorithms. Since a simple sample algorithm only takes a portion of baskets and pretends it to be the entire dataset, the sample size would have a great effect on the results of frequent 2-itemsets. For most of the datasets, increasing the sample size can help us get more accurate results, but the time taken would also increase. Since the SON algorithm distributes datasets into several chunks and does parallel computing, the results will always be accurate.
2. Since a simple sample algorithm cannot either produce all the itemsets that are frequent in the whole dataset nor will it produce only itemsets that are frequent in the whole, false positive and false negative will be introduced. An itemset that is frequent in the whole but not in the sample is a false negative, while an itemset that is frequent in the sample but not the whole is a false positive.

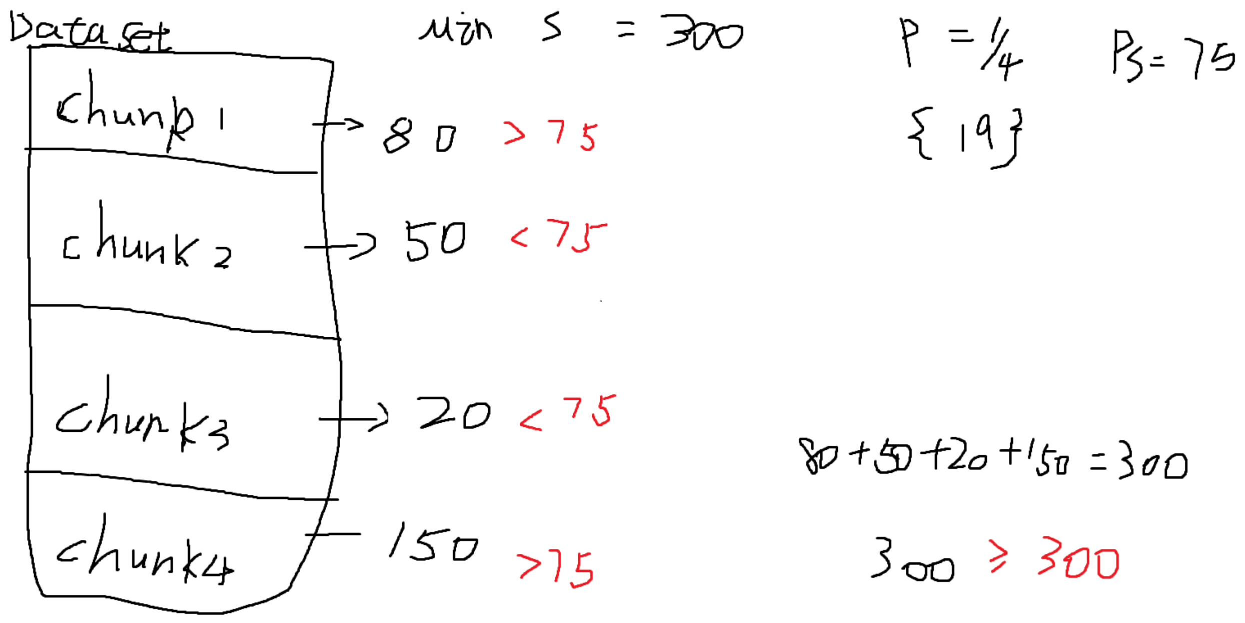
- One improvement can be introduced to eliminate false positives for simple sample algorithm:

Making a pass through the full dataset and counting all the itemsets that were identified as frequent in the sample. Retain as frequent only those itemsets that were frequent in the sample and
also frequent in the whole.

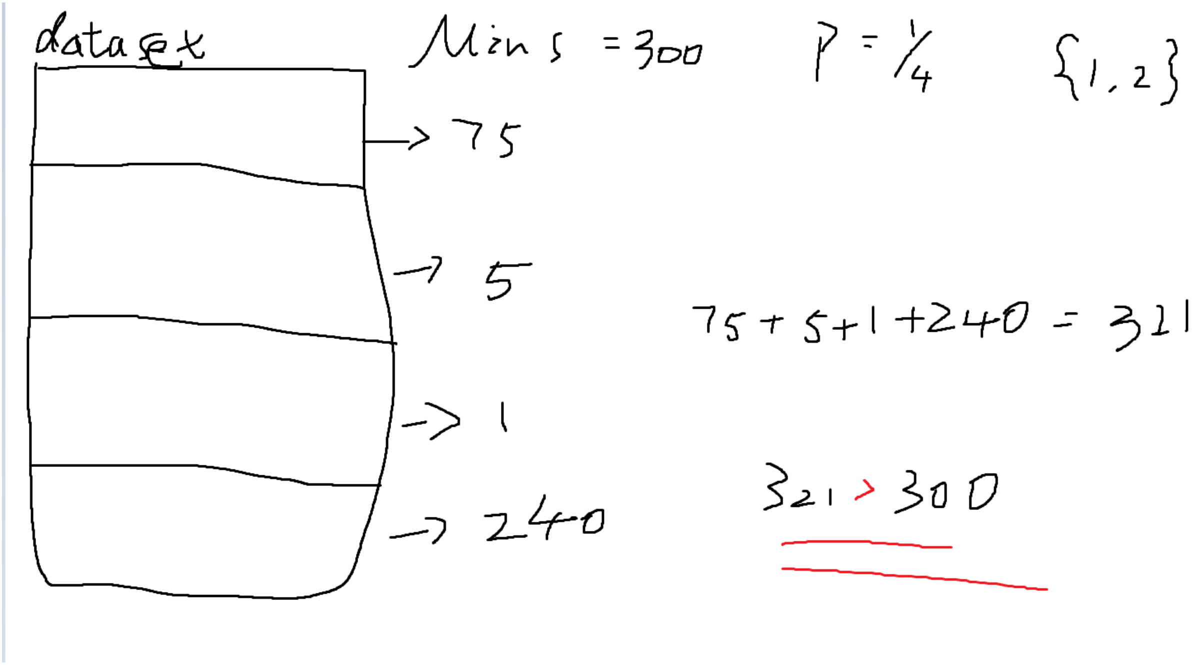
- One improvement can be introduced to reduce the number of false negatives:

Instead of using `ps` as the min support threshold for sample, we can use `0.9ps`.

Having a lower threshold means that more itemsets of each size will have to be counted, so the main-memory requirement rises. On the other hand, if there is enough main memory, then we shall identify as having the support of at least 0.9ps in the sample almost all those itemsets that have support of at least s is the whole.



The figure above demonstrated the reason why the SON algorithm doesn’t have any false positives or false negatives. Take the example above, the all dataset has been divided into 4 chunks, the min support value for the whole dataset is 300, the p = 1/number of chunks = 1/4, therefore the ps = 75. If one singleton item (19) has occurred 80 times in the first chunk, 50 times in the second chunk, 20 times in the third chunk, and 150 times in the fourth chunk, then we concluded that (19) is a frequent singleton item. The reason is that it has larger support values in the first and fourth chunks. To verify our assumption, we can add all the support values together and it equals the min support value, which means this singleton item is indeed a frequent item. If one item is indeed a frequent item, its overall support value will remind the same. Therefore, if all the support value is less than ps in the first three chunks, the last chunk will definitely have a support value greater than ps.



In the SON algorithm, we count the support value for candidate 2-itemsets in each chunk and compare the support value with the global min support value. For example, even though the support value for {1,2} are only 5 and 1 in the second and third chunks, the summed up support value is greater than the global min support value which means, {1,2} is a frequent 2-itemset.

- Time and Space complexity analyse:

simple random algorithm:

Although the simple random algorithm only executed on a portion of overall data, the frequent itemsets are calculated based on A-priori algorithm. Therefore, both time and space complexity is O(2^d), where d is total number of unique items in transaction dataset. Practically its complexity can be significantly reduced using pruning process in intermediate steps and using some optimizations techniques like usage of hash tress for calculating support values of candidates.

If we consider space complexity, then at first step you need to store dC1 candidates, in 2nd dC2 , in 3rd it will be dC3, till you reach to last step where you need to store dCd items. So if we sum up above terms, it will be

dC1+dC2+...+dCd=2^d−1

Which is O(2^d)

`Exercise 3 Advertising:`

`Part 1`

- **Greedy Algorithm**

A greedy algorithm is an algorithmic strategy that makes the best optimal choice at each small stage with the goal of this eventually leading to a globally optimum solution. This means that the algorithm picks the best solution at the moment without regard for consequences. It picks the best immediate output, but does not consider the big picture, hence it is considered greedy.

A wants x , budget = 4 $

B wants x or y , budget = 4 $

Q $\underline{x\ x\ x\ x\ y\ y\ y\ y}$ (x costs 1 $, y costs 1 $)

A first

A $\underline{x\ x\ x\ x}$ B $\underline{y\ y\ y\ y}$

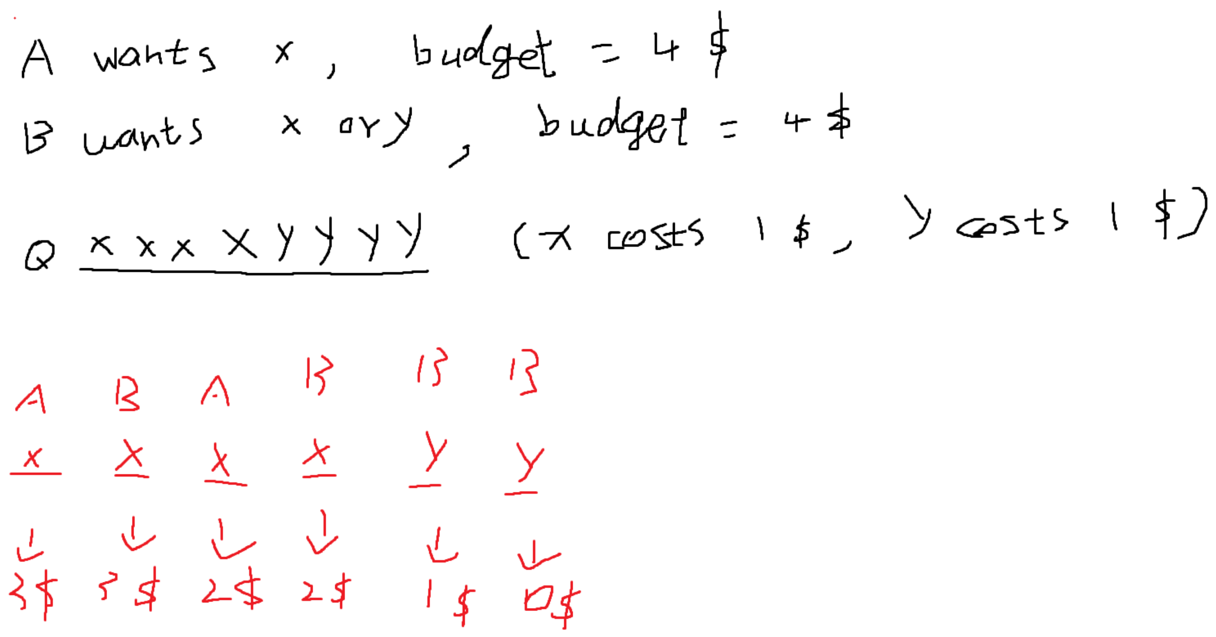
B first

B $\underline{x\ x\ x\ x}$ A $\underline{\qquad\qquad}$

The figure above demonstrated how the greedy algorithm works.
A wants product x and B wants products x or y. Each person has 4$ and each item costs 1 $ to purchase. If the queries arrive as four x products followed by four y products, if A bids first, 4 x products can be purchased by A, and following 4 y products can be purchased by B, then all the products can all be purchased, and result in an optimal solution. However, if B bids first, all 4 x products will be purchased by B and no products can be purchased by A since A doesn’t want any y product, this results in the worst solution. Both situations can happen in a greedy algorithm since the greedy algorithm only cares about the best solution at the moment without regard for consequences.

- **The Balance Algorithm**

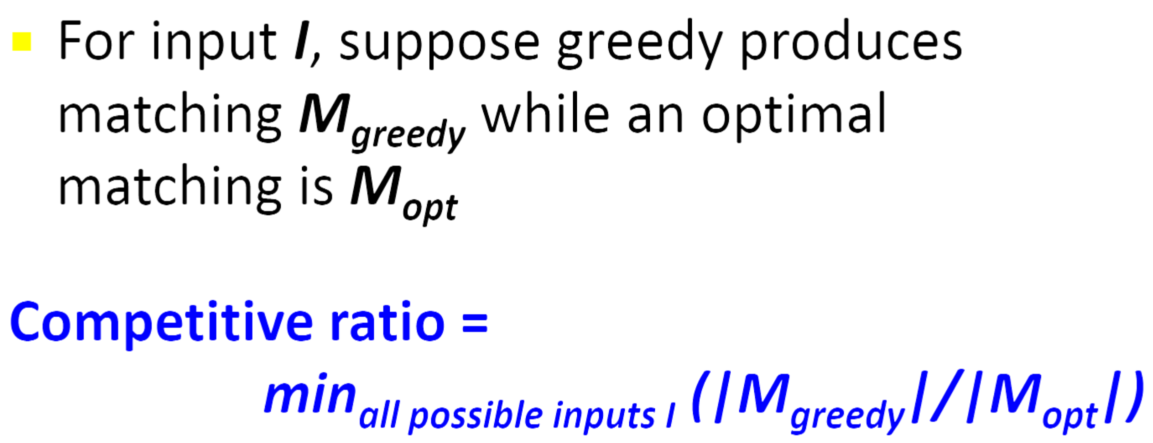
Balance algorithm is an improved algorithm derived from greedy algorithm which offers better competitive ratio.



A wants x , budget = 4 $
B wants x or y , budget = 4 $
Q x x x x y y y y (x costs 1 $, y costs 1 $)
A B A B B B
x x x x y y
3$ 3$ 2$ 2$ 1$ 0$

The figure above demonstrated how the balance algorithm works.
Consider the same situation as in the previous example. The balance algorithm assigns the first product to either A or B because both A and B bid on x and they have the same reminding budgets. Let’s say the first x is assigned to A, then the second x must be assigned to B since A has less remaining budget than B. The first y will be assigned to B because B still has a budget remaining and is a bidder for y. The third and fourth y will not be assigned to anyone since B is out of budget and A did not bid.

- **The Competitive Ratio**



- For input I, suppose greedy produces matching M_{greedy} while an optimal matching is M_{opt}

Competitive ratio =

$$\min_{\text{all possible inputs } I} \left(|M_{greedy}|/|M_{opt}|\right)$$

As we see from the previous two examples, an on-line algorithm need not give as good a result as the best off-line algorithm for the same problem. The off-line algorithm can achieve the best result by assigning four x to A and four y to B. Therefore, we expect there is a constant c which is less than 1, such that on any input, the result of a particular on-line algorithm is at least c times the result of the optimum off-line algorithm. If this constant c exists, it is called the competitive ratio. For the greedy algorithm example, the optimal solution is all the products can be purchased by A and B, and the worst solution is only four x products were purchased by B and none for A, therefore, the competitive ratio is 1/2, denoting that greedy algorithm is 1/2 as good as the optimum algorithm. Same for the balance algorithm, the competitive ratio of 3/4 denotes that the balance algorithm is 3/4 as good as the optimum algorithm.

`Part 2`

1. Since A only bids on x, B can bid on x and y, and C can bid on x and y and z, the worst solution will occur when A is not the first bidder or when C bids before B. For example, If B bids first followed by A then C, then two x will be assigned to B, A gets nothing and two z will be assigned to C. If A bids first followed by C then B, then two x will be assigned to A, two y will be assigned to C and B gets nothing. The optimal solution will occur Therefore, the greedy algorithm will assign at least 4 of 6 queries.
2. Queries y y z z would allow the greedy algorithm to assign as few as half the queries that the optimal offline algorithm would assign to that sequence. For example, the offline algorithm would assign two y to B and assign two z to C. The greedy algorithm would assign two y to C and nothing for A and B if C bids before B.