INF 553(Spring 2016)

Assignment 2: Frequent Itemsets

In this assignment you will be implementing 3 different algorithms to find frequent itemsets namely PCY, Multi-Stage, Toivonen. The transactions will be given as an input file. The goal of the assignment is to make you understand the algorithms better by coding them.

Write your own code!

For this assignment to be an effective learning experience, you must write your own code! I emphasize this point because you will be able to find Python implementations of most or perhaps even all of the required functions on the web. Please do not look for or at any such code! **Do not share code with other students in the class!!**

Here's why:

- The most obvious reason is that it will be a huge temptation to cheat: if you include code written by anyone else in your solution to the assignment, you will be cheating. As mentioned in the syllabus, this is a very serious offense, and may lead to you failing the class.
- However, even if you do not directly include any code you look at in your solution, it surely will influence your coding. Put another way, it will short-circuit the process of you figuring out how to solve the problem, and will thus decrease how much you learn.

So, just don't look on the web for any code relevant to these problems. Don't do it.

Submission Details

For each problem, you will turn in a python programs. To allow you to test your programs, sample data will be provided in the data folder, and the corresponding solutions will be provided for each problem in the solutions folder.

You need to turn in the following python files.

1. Program that implements PCY algorithm. : <firstname>_<lastname>_pcy.py
2. Program that implements Multi-Stage : <firstname>_<lastname>_multistage.py
3. Program that implements Toivonen: <firstname>_<lastname>_toivonen.py

Problem 1: PCY Algorithm

Implement PCY algorithm using a single hash and print all frequent itemsets. You can use a hashing function of your choice.

Input Parameters:

1. Input.txt: This is the input file containing all transactions. Each line corresponds to a transaction. Each transaction has items that are comma separated. Use input.txt and input2.txt to test this algorithm.
2. Support: Integer that defines the minimum count to qualify as a frequent itemset.
3. Bucket size: This is the size of the hash table.

Output: The output needs to contain the frequent itemsets generated in each pass sorted lexicographically. It should also contain information about the memory usage during each pass over the data along with the hash buckets and their counts (Assume each count takes 4 bytes).

Note: In PCY the counts for pairs are stored in form of Triples (e.g. count of pair (“a”,”b”) is stored as (“a”, ”b”, <count>) and it uses 12 bytes.). Similarly, frequent itemsets of size 3 will be stored in form of quadruples taking 16 bytes and so on. The counts in the buckets can vary depending on the hashing function used. So do not try to match this with the output files provided. Also keep in mind that while finding frequent pairs you should hash all the possible pairs but in finding frequent itemsets of **size >= 3**, as you know the frequent itemsets of **size-1** you can use those and hash only those itemsets whose all possible subsets of **size-1** are frequent.

Two sample solutions are shown below.
Sample 1:

Executing code: **python <lastname>_<firstname>_pcy.py input2.txt 3 20**

```
memory for item counts: 48
memory for hash table counts for size 2 itemsets: 80
{0: 0, 1: 0, 2: 0, 3: 0, 4: 1, 5: 3, 6: 2, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 1, 14: 2, 15: 3, 16: 1, 17: 0, 18: 0, 19: 0}
frequent itemsets of size 1: ['e']
```

Here for the 1st pass we show the memory used for the item counts, hash table counts, hash_buckets with their counts and frequent itemsets found respectively. As there are no frequent itemsets of size 2, we do not output anything for subsequent passes.

Sample 2:

Executing code: **python <lastname>_<firstname>_pcy.py input.txt 5 20**

```
memory for item counts: 96
memory for hash table counts for size 2 itemsets: 80
{0: 0, 1: 0, 2: 0, 3: 7, 4: 4, 5: 10, 6: 9, 7: 8, 8: 2, 9: 8, 10: 12, 11: 14, 12: 24, 13: 10, 14: 5, 15: 11, 16: 13, 17: 9, 18: 3, 19:
7}
frequent itemsets of size 1: ['a', 'b', 'c', 'd', 'e', 'f']

memory for frequent itemsets of size 1 : 48
bitmap size: 20
memory for candidates counts of size 2 : 168
frequent itemsets of size 2 : [['a', 'b'], ['a', 'c'], ['a', 'e'], ['b', 'c'], ['b', 'f'], ['c', 'e'], ['d', 'e'], ['e', 'f']]

memory for frequent itemsets of size 2 : 96
memory for hash table counts for size 3 itemsets: 80
{0: 0, 1: 0, 2: 0, 3: 5, 4: 0, 5: 0, 6: 0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0, 15: 4, 16: 0, 17: 0, 18: 0, 19: 0}

bitmap size: 20
memory for candidates counts of size 3 : 16
frequent itemsets of size 3 : [['a', 'b', 'c']]
```

You can also use **input.txt**, **support = 3 and bucket size = 20** to run your code and compare the output with **output_pcy.txt**.

Problem 2: Multi-Stage Algorithm

Implement the Multi-Stage algorithm to generate frequent itemsets. You need to have **two** stages using two different hashing functions for finding frequent itemsets of each size. Use hashing functions which are independent of each other. Both the hashes will have the same number of buckets. Also keep in mind that while finding frequent pairs you should hash all the possible pairs but in finding frequent itemsets of **size >= 3**, as you know the frequent itemsets of **size-1**, you can use those and hash only those itemsets whose all possible subsets of **size-1** are frequent. The counts in the buckets can vary depending on the hashing function used. So do not try to match this with the output files provided.

Input parameters are same as above. For output please follow the format shown below:

Sample 1:
Executing code: **python <lastname>_<firstname>_multistage.py input2.txt 2 20**

```
memory for item counts: 48
memory for hash table 1 counts for size 2 itemsets: 80
{0: 0, 1: 0, 2: 0, 3: 0, 4: 1, 5: 3, 6: 2, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 1, 14: 2, 15: 3, 16: 1, 17: 0, 18: 0, 19: 0}
frequent itemsets of size 1: ['a', 'd', 'e']

memory for frequent itemsets of size 1 : 24
bitmap 1 size: 20
memory for hash table 2 counts for size 2 itemsets: 80
{0: 2, 1: 0, 2: 0, 3: 0, 4: 1, 5: 2, 6: 0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0, 15: 0, 16: 0, 17: 0, 18: 0, 19: 0}

memory for frequent itemsets of size 1 : 24
bitmap 1 size: 20
bitmap 2 size: 20
memory for candidates of size  2 : 24
frequent itemsets of size 2 : [['a', 'e'], ['d', 'e']]
```

Sample 2:
Executing code: **python <lastname>_<firstname>_ multistage.py input.txt 5 20**

```
memory for item counts: 96
memory for hash table 1 counts for size 2 itemsets: 80
{0: 0, 1: 0, 2: 0, 3: 7, 4: 4, 5: 10, 6: 9, 7: 8, 8: 2, 9: 8, 10: 12, 11: 14, 12: 24, 13: 10, 14: 5, 15: 11, 16: 13, 17: 9, 18: 3, 19:
7}
frequent itemsets of size 1: ['a', 'b', 'c', 'd', 'e', 'f']

memory for frequent itemsets of size 1 : 48
bitmap 1 size: 20
memory for hash table 2 counts for size 2 itemsets: 80
{0: 6, 1: 0, 2: 8, 3: 10, 4: 5, 5: 5, 6: 11, 7: 0, 8: 0, 9: 0, 10: 9, 11: 0, 12: 9, 13: 0, 14: 0, 15: 6, 16: 0, 17: 0, 18: 4, 19: 0}

memory for frequent itemsets of size 1 : 48
bitmap 1 size: 20
bitmap 2 size: 20
memory for candidates of size  2 : 156
frequent itemsets of size 2 : [['a', 'b'], ['a', 'c'], ['a', 'e'], ['b', 'c'], ['b', 'f'], ['c', 'e'], ['d', 'e'], ['e', 'f']]

memory for frequent itemsets of size  2 : 96
memory for hash table 1 counts for size 3 itemsets: 80
{0: 0, 1: 0, 2: 0, 3: 5, 4: 0, 5: 0, 6: 0, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0, 15: 4, 16: 0, 17: 0, 18: 0, 19: 0}

bitmap 1 size: 20
memory for hash table 2 counts for size 3 itemsets: 80
{0: 0, 1: 0, 2: 0, 3: 0, 4: 0, 5: 0, 6: 5, 7: 0, 8: 0, 9: 0, 10: 0, 11: 0, 12: 0, 13: 0, 14: 0, 15: 0, 16: 0, 17: 0, 18: 0, 19: 0}

bitmap 1 size: 20
bitmap 2 size: 20
memory for candidates counts of size  3 : 16
frequent itemsets of size 3 : [['a', 'b', 'c']]
```

You can also use **input.txt**, **support = 3 and bucket size = 20** to run your code and compare the output with **output_multistage.txt**.

Problem 3: Toivonen Algorithm

Implement the Toivonen algorithm to generate frequent itemsets. For this algorithm you need to use a sample size of less than 60% of your entire dataset. Use an appropriate sampling method to get the random sample set. Also perform a simple Apriori algorithm with the random sample set. Check for negative borders and run the algorithm again with a different sample set if required till there are no negative borders that have frequency > support.

Note: The number of iterations might differ in your output depending on the random sample which is generated.

Input Parameters:

1. Input.txt: This is the input file containing all transactions. Each line corresponds to a transaction. Each transaction has items that are comma separated. Use toivonen_test.txt to test this algorithm.
2. Support: Integer that defines the minimum count to qualify as a frequent itemset.

Output:

Line 1 <number of iterations performed>
Line 2 <fraction of transactions used>
Line 3 onwards <frequent itemsets lexicographically sorted>

Use **toivonen_test.txt** and **support = 20** to run your code and compare the output with **output_toivonen.txt**

Executing code: **python <firstname>_<lastname>_ toivonen.py toivonen_test.txt 20**

General Instructions:

1. Do not zip your files
2. Make sure your code compiles before submitting
3. Make sure to follow the output format and the naming format.
4. Make sure not to write the output to any files. Use standard output to print them.
5. We will be using Moss for plagiarism detection.