Data Mining Fall 2024 Lab Assignment #1

Student ID: 313551099
Student Name: 李以恩

I. Explain how to run your code in Step Ⅱ and Ⅲ

- For Step Ⅱ: apriori.py

```
# cmd Usage
python apriori.py -f [datasetName] -s [minSupport] -t [taskNum]

# Example: running for datasetA with minSupport = 0.3 % of task 1
python apriori.py -f datasetA.data -s 0.003 -t 1

# Example: Output file of task 1
step2_task1_datasetA_0.3_result1.txt
step2_task1_datasetA_0.3_result2.txt

# Example: Output file of task 2
step2_task2_datasetA_0.3_result1.txt
```

- For Step Ⅲ: fpgrowth.py

  Requirement: pip install mlxtend

```
# cmd Usage
python fpgrowth.py -f [datasetName] -s [minSupport]

# Example: running for datasetA with minSupport = 0.3 %
python fpgrowth.py -f datasetA.data -s 0.003

# Example: Output file
step3_task1_datasetA_0.3_result1.txt
```

II. Step Ⅱ

- **Report on the mining algorithms/codes:**
  - **The modifications you made for Task 1 and Task 2**
    - Modification of Task 1: Mining all Frequent Itemset
      1. Reading input
         a. Use blank space to strip the line: line.split(“ ”)[3:]
         b. Take the data from the 3rd column: line.split(“ ”)[3:]

```
def dataFromFile(fname):
    """Function which reads from the file and yields a generator"""
    with open(fname, "r") as file_iter:
        for line in file_iter:
            line = line.strip().rstrip(" ")
            record = frozenset(line.split(" ")[3:])
            yield record
```

2. Recording statistical data
    a. Use iteration_stats, num_before_pruning, num_after_pruning to keep track of the statistical data

```python
### Apriori mining process ###
def runApriori(data_iter, minSupport):
    """
    run the apriori algorithm. data_iter is a record iterator
    Return both:
     - items (tuple, support)
    """
    start_time = time.time()
    itemSet, transactionList = getItemSetTransactionList(data_iter)
    freqSet = defaultdict(int)
    largeSet = dict()
    num_before_pruning = 0
    num_after_pruning = 0
    iteration_stats = []

    k = 1
    num_before_pruning = len(itemSet)
    oneCSet= returnItemsWithMinSupport(itemSet, transactionList, minSupport, freqSet)
    num_after_pruning = len(oneCSet)
    currentLSet = oneCSet
    iteration_stats.append((k, num_before_pruning, num_after_pruning))

    k = 2
    while currentLSet != set([]):
        print(f"-------------------- Counting k = {k} --------------------")
        largeSet[k - 1] = currentLSet
        currentLSet = joinSet(currentLSet, k)
        num_before_pruning = len(currentLSet)
        currentCSet= returnItemsWithMinSupport(
            currentLSet, transactionList, minSupport, freqSet
        )
        num_after_pruning = len(currentCSet)
        iteration_stats.append((k, num_before_pruning, num_after_pruning))
        currentLSet = currentCSet
        k = k + 1

    end_time = time.time()
    exec_time = round(end_time - start_time, 4)
    toRetItems = []
    for key, value in largeSet.items():
        toRetItems.extend([(tuple(item), getSupport(item, freqSet, transactionList)) for item in value])
    return toRetItems, iteration_stats, exec_time
```

3. Writing result
    a. Sort the itemsets by descending order of support value: reverse=True
    b. Write of the specified data out

```python
def writeResults_task1_file1(items, filename="file1.txt"):
    # os.makedirs(os.path.dirname(filename), exist_ok=True)
    with open(filename, "w") as f:
        for item, support in sorted(items, key=lambda x: x[1], reverse=True):
            support_percentage = round(support * 100, 1)
            item_str = "{" + ", ".join(map(str, item)) + "}"
            f.write(f"{support_percentage}\t{item_str}\n")


def writeResults_task1_file2(items, iteration_stats, filename="file2.txt"):
    # os.makedirs(os.path.dirname(filename), exist_ok=True)
    with open(filename, "w") as f:
        f.write(f"{len(items)}\n")
        for item in iteration_stats:
            f.write(f"{item[0]}\t{item[1]}\t{item[2]}\n")
```

- Modification of Task 2: Mining all Frequent Closed Itemset

1. Create runApriori_closed()
   a. Use is_closed as the flag to check an itemset is closed or not
   b. Evaluation criteria: check whether there is a frequent itemset in the next combination layer with the same support value
      if item.issubset(superSet) and freqSet[item] == freqSet[superSet]

```python
### Apriori mining process for closed frequent sets ###
def runApriori_closed(data_iter, minSupport):
    """
    run the apriori algorithm. data_iter is a record iterator
    Return both:
     - items (tuple, support)
    """
    start_time = time.time()
    itemSet, transactionList = getItemSetTransactionList(data_iter)
    freqSet = defaultdict(int)
    largeSet = dict()  # Global dictionary which stores (key=n-itemSets,value=support) which satisfy minSupport
    closedSet = []

    k = 1
    oneCSet= returnItemsWithMinSupport(itemSet, transactionList, minSupport, freqSet)
    currentLSet = oneCSet

    k = 2
    while currentLSet != set([]):
        print(f"------------------- Counting k = {k} -------------------")
        largeSet[k - 1] = currentLSet
        currentLSet = joinSet(currentLSet, k)
        currentCSet= returnItemsWithMinSupport(
            currentLSet, transactionList, minSupport, freqSet
        )
        ### Find closed frequent sets ###
        num_old = len(closedSet)
        for item in largeSet[k - 1]:
            is_closed = True
            for superSet in currentCSet:
                if item.issubset(superSet) and freqSet[item] == freqSet[superSet]:
                    is_closed = False
                    break
            if is_closed:
                closedSet.append((tuple(item), getSupport(item, freqSet, transactionList)))
        num_new = len(closedSet)
        # print(f"num_old = {num_old}\tnum_new = {num_new}\tadding = {num_new - num_old}")
        currentLSet = currentCSet
        k = k + 1

    end_time = time.time()
    exec_time = round(end_time - start_time, 4)
    return closedSet, exec_time
```

2. Writing result
   a. Sort the itemsets by descending order of support value: reverse=True
   b. Write of the specified data out

```python
def writeResults_task2_file(items, filename="file.txt"):
    # os.makedirs(os.path.dirname(filename), exist_ok=True)
    with open(filename, "w") as f:
        f.write(f"{len(items)}\n")
        for item, support in sorted(items, key=lambda x: x[1], reverse=True):
            support_percentage = round(support * 100, 1)
            item_str = "{" + ", ".join(map(str, item)) + "}"
            f.write(f"{support_percentage}\t{item_str}\n")
```

- Modification of main function

```
minSupport = options.minS
minSupportName = str(round(minSupport * 100, 1))
datasetName = str(options.input).split('.')[0]
taskName = "task" + str(options.taskNum)
print(f"***** Doing task : {taskName}_{datasetName}_{minSupportName} *****\n")

### Doing task 1/2 ###
if options.taskNum == 1:
    items, iteration_stats, exec_time= runApriori(inFile, minSupport)
    filename1 = "step2_" + taskName + "_" + datasetName + "_" + minSupportName + "_" + "result1.txt"
    filename2 = taskName + "_" + datasetName + "_" + minSupportName + "_" + "result2.txt"
    writeResults_task1_file1(items, filename1)
    writeResults_task1_file2(items, iteration_stats, filename2)
    writeResults_time(exec_time, "step2_time.txt", filename1)

elif options.taskNum == 2:
    closedSet, exec_time = runApriori_closed(inFile, minSupport)
    filename_closed = "step2_" + taskName + "_" + datasetName + "_" + minSupportName + "_" + "result1.txt"
    writeResults_task2_file(closedSet, filename_closed)
    writeResults_time(exec_time, "step2_time.txt", filename_closed)
```

- **The restrictions (e.g., the scalability, etc.)**
  1. If the dataset is huge, like datasetC, it would take much time finding itemsets.
  2. If the minSupport is low, like 0.2% for datasetB, it also takes much time.

- **Problems encountered in mining**
  1. For task 1, it wasn't hard because we only needed to modify the reading and writing part, and also add some variables to keep track of pruning count.
  2. For task 2, we need to figure out a way to evaluate whether an itemset is closed or not. I added the criteria in the while loop, checking closed itemsets containing k-1 items from itemsets containing k items.

- **Any observations/discoveries**
  1. When minSupport is high -> the number of frequent itemsets candidates is lower -> take shorter time to explore, because can prune at earlier time
  2. When minSupport is low -> the number of frequent itemsets candidates is higher -> take longer time to explore, because can only prune at later time
  3. When the data amount is huge -> will generate much more set combination for frequent itemsets -> take much longer time to explore
  4. The number of closed frequent itemsets <= The number of frequent itemsets, but from my implementation, it usually takes longer to find closed frequent itemsets.

- **Paste the screenshot of the computation time**

```
step2_task1_datasetA_0.3_result1.txt        Time = 14.7195 sec
step2_task2_datasetA_0.3_result1.txt        Time = 16.3299 sec
step2_task1_datasetA_0.6_result1.txt        Time = 3.0032 sec
step2_task2_datasetA_0.6_result1.txt        Time = 2.9712 sec
step2_task1_datasetA_0.9_result1.txt        Time = 1.9111 sec
step2_task2_datasetA_0.9_result1.txt        Time = 1.9349 sec
step2_task1_datasetB_0.2_result1.txt        Time = 2834.3447 sec
step2_task2_datasetB_0.2_result1.txt        Time = 2814.8076 sec
step2_task1_datasetB_0.4_result1.txt        Time = 747.5079 sec
step2_task2_datasetB_0.4_result1.txt        Time = 773.7804 sec
step2_task1_datasetB_0.6_result1.txt        Time = 477.5865 sec
step2_task2_datasetB_0.6_result1.txt        Time = 488.8876 sec
step2_task1_datasetC_0.5_result1.txt        Time = 3024.3077 sec
step2_task2_datasetC_0.5_result1.txt        Time = 2959.1359 sec
step2_task1_datasetC_1.0_result1.txt        Time = 1622.9808 sec
step2_task2_datasetC_1.0_result1.txt        Time = 1381.5293 sec
step2_task1_datasetC_1.5_result1.txt        Time = 848.6731 sec
step2_task2_datasetC_1.5_result1.txt        Time = 922.8693 sec
```

- **Show the ratio of computation time of task 1 and task 2**

	A 0.3	A 0.6	A 0.9	B 0.2	B 0.4	B 0.6	C 0.5	C 1.0	C 1.5
step2 task1	14.7195	3.0032	1.9111	2834.3447	747.5079	477.5865	3024.3077	1622.9808	848.6731
step2 task2	16.3299	2.9712	1.9349	2814.8076	773.7804	488.8876	2959.1359	1381.5293	922.8693
	A 0.3	**A 0.6**	**A 0.9**	**B 0.2**	**B 0.4**	**B 0.6**	**C 0.5**	**C 1.0**	**C 1.5**
Ratio = task2 / task1 (%)	**110.94**	**98.93**	**101.25**	**99.31**	**103.51**	**102.37**	**97.85**	**85.12**	**108.74**

Since the number of closed frequent itemsets should less than or equal to the number of frequent itemsets, it may take shorter time for task 2 than task 1. However, in my implementation, since I check whether an itemset is closed in the same loop that apriori uses to check whether a candidate set satisfies the minimum support, in task 2, it actually does the same thing in task 1, and then checks the closed property. Thus, it makes sense that task 2 sometimes takes more time than task 1.

III. Step Ⅲ

- **Descriptions of your mining algorithm**
  - **FP-Growth algorithm**

    The FP-growth algorithm finds frequent itemsets without generating candidate sets, which makes it efficient for mining frequent patterns in large datasets. The method uses the FP-tree to store the transactions in a compressed way, which allows frequent patterns to be identified directly from the structure. Compared to the Apriori algorithm, it saves lots of time. The steps are:

    — Build the FP-tree —

    1. Count item frequency: filter out the items which don't meet minSupport
    2. Sort Items in Transactions: ensure common items are close to the root in the FP-tree, which help to maximize the compression
    3. Insert transactions to build the FP-tree by iterating each transaction

    — Mine the FP-tree —

    4. Identify Frequent Patterns by Suffix Paths
    5. Build Conditional FP-trees: which represents the set of transactions that include the item as the suffix
    6. Recursion for Pattern Generation: For each node, the support count is added up, and if it meets the minimum support threshold, it is added as a frequent pattern

  - **Open-sourced code**
    - **Link**: https://rasbt.github.io/mlxtend/user_guide/frequent_patterns/fpgrowth/
    - **Why choosing this code and How it works**

      At first, I found some fp-growth implementations on GitHub. However, most of them either got poor performance on the execution time or had some missing result. Therefore, instead of implementing the algorithm from scratch, I found the mlxtend library in python, which offers an fpgrowth function that requires minimal setup. The library provids the optimized fpgrowth function for speed and efficiency using pre-built algorithms that are faster than manual implementations, and also yields results with absolute accuracy.

It first uses the TransactionEncoder to do the one-hot encoding to the transaction dataset, like the example picture below. Then fpgrowth function is applied to the data and returns the frequent itemsets with support values in a dataframe form.

- Example of transaction data

```
dataset = [['Milk', 'Onion', 'Nutmeg', 'Kidney Beans', 'Eggs', 'Yogurt'],
           ['Dill', 'Onion', 'Nutmeg', 'Kidney Beans', 'Eggs', 'Yogurt'],
           ['Milk', 'Apple', 'Kidney Beans', 'Eggs'],
           ['Milk', 'Unicorn', 'Corn', 'Kidney Beans', 'Yogurt'],
           ['Corn', 'Onion', 'Onion', 'Kidney Beans', 'Ice cream', 'Eggs']]
```

- Example of the one-hot encoding of the data

	Apple	Corn	Dill	Eggs	Ice cream	Kidney Beans	Milk	Nutmeg	Onion	Unicorn	Yogurt
0	False	False	False	True	False	True	True	True	True	False	True
1	False	False	True	True	False	True	False	True	True	False	True
2	True	False	False	True	False	True	True	False	False	False	False
3	False	True	False	False	False	True	True	False	False	True	True
4	False	True	False	True	True	True	False	False	True	False	False

- **Differences/Improvements in your algorithm**

1. Data mining in main function
   a. Use TransactionEncoder() to do the one-hot encoding to the data
   b. Transform the data to the dataframe format
   c. Apply fpgrowth to the data, using use_colnames=True will return the results to convert the index values into the respective item names
   d. Use sort_values to sort the results in descending order of the support value

```
### Data Mining ###
start_time = time.time()
te = TransactionEncoder()
te_ary = te.fit(inFile).transform(inFile)                # encode the data
df = pd.DataFrame(te_ary, columns=te.columns_)           # transform to dataframe
df_res = fpgrowth(df, minSupport, use_colnames=True)     # use fpgrowth to mine
df_res = df_res.sort_values(by='support', ascending=False) # sort by descending order
end_time = time.time()
exec_time = round(end_time - start_time, 4)
```

2. **Reading input data and writing result**
   a. Adopt dataFromFile() in the apriori.py code, to append each record of each line to the transaction_list
   b. Write the results in the specified way, including timing the support value by 100 and excluding the single quote of the original data, for example let {'318', '212'} convert to {318, 212}.

```
def dataFromFile(fname):
    """Function which reads from the file and yields a generator"""
    transaction_list = []
    with open(fname, "r") as file_iter:
        for line in file_iter:
            line = line.strip().rstrip(" ")  # strip the line by " "
            record = line.split(" ")[3:]     # take the data from 3rd column
            transaction_list.append(record)
    return transaction_list


def writeResults(df_res, filename="file1.txt"):
    # os.makedirs(os.path.dirname(filename), exist_ok=True)
    with open(filename, "w") as f:
        for index, row in df_res.iterrows():
            row_support = str(round(row['support'] * 100, 1))
            row_itemsets = "{" + ", ".join(map(str, row['itemsets'])) + "}"
            f.write(f"{row_support}\t{row_itemsets}\n")
```

- **Computation time**
  - Paste the screenshot of the computation time

```
step3_task1_datasetA_0.3_result1.txt    Time = 0.1254 sec

step3_task1_datasetA_0.6_result1.txt    Time = 0.0469 sec

step3_task1_datasetA_0.9_result1.txt    Time = 0.0207 sec

step3_task1_datasetB_0.2_result1.txt    Time = 5.8405 sec

step3_task1_datasetB_0.4_result1.txt    Time = 4.1096 sec

step3_task1_datasetB_0.6_result1.txt    Time = 3.874 sec

step3_task1_datasetC_0.5_result1.txt    Time = 22.698 sec

step3_task1_datasetC_1.0_result1.txt    Time = 20.4573 sec

step3_task1_datasetC_1.5_result1.txt    Time = 18.1682 sec
```

  - Calcule the speedup

	A 0.3	A 0.6	A 0.9	B 0.2	B 0.4	B 0.6	C 0.5	C 1.0	C 1.5
step2 (sec)	14.7195	3.0032	1.9111	2834.3447	747.5079	477.5865	3024.3077	1622.9808	848.6731
step3 (sec)	0.1254	0.0469	0.0207	5.8405	4.1096	3.874	22.698	20.4573	18.1682
Speedup = (step 2 - step 3) / step 2 (%)	99.15	98.44	98.92	99.79	99.45	99.19	99.25	98.74	97.86

We can see that there is a significant improvement in speed of the fp-growth algorithm in step 3 compared to the apriori algorithm in step 2. Not only because that fp-growth does not require candidate generation, but also because the mlxtend python library has provided an optimized version of the funcion.

- **Discuss the scalability of your algorithm in terms of the dataset size** (i.e., the rate of change on computing time under different data sizes, the largest dataset size the algorithm can handle, etc)
  - For different data sizes, as the data amount increases, it requires much more time to run the algorithm. For example, for dataset B which has 100 times more data than dataset A, it takes nearly 50 times more time compared to dataset A. And as the dataset size increases to a certain level, the time needed increases faster, for example, for dataset C which has 5 times more data than dataset B, it takes nearly 5 times more time to dataset B.
  - For the toy dataset which has 1,000,000 data, it takes up to 40 seconds to mine the data, so the algorithm can deal with at least 1,000,000 data entries.

- **Test dataset verification**

```
PS C:\Users\ieeni\Desktop\##\DM\hw1\EN> python fpgrowth.py -f datasetToy.data -s 0.01
***** Doing task : datasetToy_1.0 *****
Time = 42.6068 sec

PS C:\Users\ieeni\Desktop\##\DM\hw1\EN> python ItemsetVerifier.py -r res_ans.txt -s step3_task1_datasetToy_1.0_result1.txt
Verification Successful: The frequent itemsets match in both files.
Number of frequent itemsets in res_ans.txt: 341
Number of frequent itemsets in step3_task1_datasetToy_1.0_result1.txt: 341
```