INFORMATION SYSTEMS
ASSIGNMENT 4

Submission by:
Ayça Avcı [S4505972]
Deepshi Garg [S4199456]

Task 1: Input Processing

- We expect an input file to be a *csv* of the following format:

```
item1, item2, item3, ... so on
 , t, ...
t, t, t,...
t, t, ...
... so on...
```

- We input the file using *dataframes* from *pandas* library:

```
df = pd.read_csv("myDataFile.csv", low_memory=False)
```

- Next, we extract the header from the file and assign an index to each item.
    - We first read all the column headers into *item_list*.
    - Then, we index these headers. This is done by making a dictionary such that *item_dict[item_name] = index_value*.
    - These index values are serial integers for every item in *item_list*.

```
item_list = list(df.columns)
item_dict = dict()

for i, item in enumerate(item_list):
    item_dict[item] = i + 1
```

- Now, we need to extract individual transactions from the data file.
    - First we create an empty list *transactions*.
    - Then, we iterate through the rows in the dataframe *df*.
    - We create a local variable *transaction* as an empty set.
    - Now we iterate through all the possible column items from *item_dict*, and check if the value of that column in this *row* is ***t***, i.e., true. If it is, then we add the assigned index value of this item to this *transaction* set.
    - Finally, we append this *transaction* to the list of all *transactions*.
    - Thus, *transactions* is a list, where each *transaction* is a set of *item index values.*

```python
transactions = list()

for i, row in df.iterrows():
    transaction = set()

    for item in item_dict:
        if row[item] == 't':
            transaction.add(item_dict[item])
    transactions.append(transaction)
```

Task 2: Implementation of the Apriori principle in determining the frequent itemsets

- We first define some Utility Functions first.

- *get_support(transactions, item_set):* This function calculates the support value for the given *item_set* from the provided list of *transactions.*
    - It initialises a local variable *match_count* to store the number of transactions where the given *item_set* is found.
    - It then iterates through the the list of *transactions*
    - For each *transaction,* it is checked whether the given *item_set* is a subset of the *transaction* or not. If it is, *match_count* is incremented.
    - Finally *support* value calculated by dividing the *match_count* by total number of *transactions* is returned

```python
def get_support(transactions, item_set):
    match_count = 0
    for transaction in transactions:
        if item_set.issubset(transaction):
            match_count += 1

    return float(match_count/len(transactions))
```

- *self_join(frequent_item_sets_per_level, level):* This function performs self join in the given list of frequent itemsets of previous level, and generates the candidate itemsets for the current *level.*
    - It takes 2 inputs: *frequent_item_sets_per_level* is a map of *level* to the list of *itemsets* found to be frequent for that *level.* Second argument is the current *level* number.
    - It first initialises the *current_level_candidaes* as an empty list, and *last_level_items* as the list of frequent itemsets from the previous level.
    - If there are no frequent itemsets from the previous level, it returns an empty list for *current_level_candidates.*
    - Otherwise, it iterates through each itemset in *last_level_items* starting from 0 for index *i*, and for each itemset in *last_level_items* starting from 1 for index *j*.

- It performs the union of itemsets at indices *is* and *j.*
- If this *union_set* is not already present in *current_level_candidates* and the number of elements in the *union_set* is equal to the *level* number, then this *union_set* is appended into *current_level_candidates.*
- We have the check for the number of elements in *union_set* to ensure that the *current_level_candidates* contain only the sets of fixed length. This is a requirement for Apriori Algorithm
- Finally, *current_level_candidates* is returned.

```
def self_join(frequent_item_sets_per_level, level):
    current_level_candidates = list()
    last_level_items = frequent_item_sets_per_level[level - 1]

    if len(last_level_items) == 0:
        return current_level_candidates

    for i in range(len(last_level_items)):
        for j in range(i+1, len(last_level_items)):
            itemset_i = last_level_items[i][0]
            itemset_j = last_level_items[j][0]
            union_set = itemset_i.union(itemset_j)

            if union_set not in current_level_candidates and len(union_set)
== level:
                current_level_candidates.append(union_set)

    return current_level_candidates
```

- *get_single_drop_subsets(item_set):* This function returns the subsets of the given *item_set* with one item less.
    - We first initialize the variable *single_drop_subsets* as an empty list.
    - Next, for each *item* in *item_set*, we create a temporary set *temp* as a copy of the *item_set* given.
    - We then remove this *item* from the *temp* set. It results in a subset of *item_set* without the *item*, i.e., a subset of length one less than the length of the *item_set*
    - We then append this *temp* set to the *single_drop_subsets*
    - Finally, we return the list *single_drop_subsets*.

```
def get_single_drop_subsets(item_set):
    single_drop_subsets = list()
    for item in item_set:
        temp = item_set.copy()
        temp.remove(item)
        single_drop_subsets.append(temp)
```

```
    return single_drop_subsets
```

- *is_valid_set(item_set, prev_level_sets):* This checks if the given *item_set* is valid, i.e., has all its subsets with support value greater than the minimum support value. It relies on the fact that *prev_level_sets* contains only those *item_sets* which are frequent, i.e., have support value greater than the minimum support value.
    - It first generates all the subsets of the given *item_set* with length one less than the length of the original *item_set.* This is done using the above described function *get_single_drop_subsets().* These subsets are stored in *single_drop_subsets* variable
    - It then iterates through the *single_drop_subsets* list.
    - For each *single_drop_subset,* it checks if it was present in the *prev_level_sets*. If it wasn't it means the given *item_set* is a superset of a non-frequent *item_set*. Thus, it returns *False*
    - If all the *single_drop_subsets* are frequent itemsets, and are present in the *prev_level_sets*, it returns *True*

```
def is_valid_set(item_set, prev_level_sets):
    single_drop_subsets = get_single_drop_subsets(item_set)

    for single_drop_set in single_drop_subsets:
        if single_drop_set not in prev_level_sets:
            return False
    return True
```

- *pruning(frequent_item_sets_per_level, level, candidate_set):* This function performs the pruning step of the Apriori Algorithm. It takes a list *candidate_set* of all the candidate itemsets for the current *level*, and for each candidate itemset checks if all its subsets are frequent itemsets. If not, it prunes it, If yes, it adds it to the list of *post_pruning_set*.
    - It first initialises an empty list variable *post_pruning_set*. This is to store the list of frequent itemsets for the current level after performing pruning operation on the given list of candidate sets.
    - If there are no *candidate_set,* it returns an empty list *post_pruning_set.*
    - Otherwise, it first creates a list of frequent itemsets from the previous level and stores it in *prev_level_sets*.
    - Then, it iterates over each *item_set* in *candidate_set* list.
    - For each *item_set*, it checks whether it is a valid itemset or not. This is done using the above described function *is_valid_set()*. This function uses the fact that all the subsets of the given *item_set* (formed by removing one item) need to be frequent itemsets for this *item_set* to be valid.
    - If this *item_set* is valid, it is appended to the list of *post_pruning_set*.
    - Finally *post_pruning_set* is returned.

```
def pruning(frequent_item_sets_per_level, level, candidate_set):
```

```
    post_pruning_set = list()
    if len(candidate_set) == 0:
        return post_pruning_set

    prev_level_sets = list()
    for item_set, _ in frequent_item_sets_per_level[level - 1]:
        prev_level_sets.append(item_set)

    for item_set in candidate_set:
        if is_valid_set(item_set, prev_level_sets):
            post_pruning_set.append(item_set)

    return post_pruning_set
```

- *apriori(min_support):* This is the main function which uses all the above described Utility functions to implement the Apriori Algorithm and generate the list of frequent itemsets for each level for the provided transactions and *min_support* value.
    - It first creates a default empty dictionary *frequent_item_sets_per_level*, which maps level numbers to the list of frequent itemsets for that level.
    - Next, it handles the first level itemsets. It means all the itemsets with only one item. To generate such itemsets, we iterate through the list of all items *item_list*. We calculate the support value of each *item* using the utility function *get_support()*. If this support value is greater than or equal to the provided *min_support* value, this *item_set* is added to the list of frequent itemsets for this level.
    - One thing to note here is that every itemset is stored as a pair of 2 values:
        - The itemset
        - The support value calculated for this itemset
    - Now, we handle the levels greater than 1
    - For each *level* greater than 1, we first generate the *current_level_candidates* itemsets by performing *self_join()* on the frequent itemsets of the previous level.
    - Next, we perform the pruning operation on these *current_level_candidates* using the *pruning()* utility function described above, and obtain the results in *post_pruning_candidates*
    - Now, if there is no itemset left after pruning, we break the loop. It means there is no point in processing for further levels.
    - Otherwise, for each *item_set* in *post_pruning_candidates,* we calculate the support value using the *get_support()* utility function.
    - If this support value is greater than or equal to the given *min_support*, we append this *item_set* into the list of frequent itemsets for this level.
    - Note that this append operation also happens in pair format as described above.
    - Finally, we return the dictionary *frequent_item_sets_per_level*.

```
from collections import defaultdict

def apriori(min_support):
```

```
    frequent_item_sets_per_level = defaultdict(list)
    print("level : 1", end = " ")

    for item in range(1, len(item_list) + 1):
        support = get_support(transactions, {item})
        if support >= min_support:
            frequent_item_sets_per_level[1].append(({item}, support))

    for level in range(2, len(item_list) + 1):
        print(level, end = " ")
        current_level_candidates = self_join(frequent_item_sets_per_level,
level)

        post_pruning_candidates = pruning(frequent_item_sets_per_level,
level, current_level_candidates)
        if len(post_pruning_candidates) == 0:
            break

        for item_set in post_pruning_candidates:
            support = get_support(transactions, item_set)
            if support >= min_support:
                frequent_item_sets_per_level[level].append((item_set,
support))

    return frequent_item_sets_per_level
```

- We specify the minimum support value for the given data here in variable *min_support* and invoke the *apriori()* function to generate the *frequent_item_sets_per_level.*

```
min_support = 0.005
frequent_item_sets_per_level = apriori(min_support)
```

Task 3: Implementation of the non-monotonicity property in the determination of the association rules

- Below code produces a dictionary called *item_support_dict* from *frequent_item_sets_per_level* that maps items to their support values.
    - First, a dictionary called *item_support_dict* is created to store key value pairs of items and their support values, and an empty list called *item_list* is created to store the name of items corresponding to *item_dict* values retrieved from *frequent_item_sets_per_level.*

- Keys and values are retrieved from the *item_dict* and put inside a list for the later use.
- For each *level* in *frequent_item_sets_per_level,* for each item-support pair, name of the item retrieved from the *key_list* that corresponds to the number in *set_support_pair*, and names are added to the *item_list*.
- Items names and their support values are mapped in the *item_support_dict* as a frozenset-float number pair.

```
item_support_dict = dict()
item_list = list()

key_list = list(item_dict.keys())
val_list = list(item_dict.values())

for level in frequent_item_sets_per_level:
    for set_support_pair in frequent_item_sets_per_level[level]:
        for i in set_support_pair[0]:
            item_list.append(key_list[val_list.index(i)])
        item_support_dict[frozenset(item_list)] = set_support_pair[1]
        item_list = list()
```

- *find_subset(item, item_length):* This function takes each item from the *item_support_dict* and its length *item_length* as parameter, and returns all possible combinations of elements inside the items.
    - It first creates an empty array called *combs* to store a list of combinations.
    - It appends a list of all possible combinations of items to the *combs* array.
    - To reach the combinations from an array directly, for *comb* array in *combs* array, and for each *elt* in *comb* array, each element appended to the *subsets* array.

```
def find_subset(item, item_length):
    combs = []
    for i in range(1, item_length + 1):
        combs.append(list(combinations(item, i)))

    subsets = []
    for comb in combs:
        for elt in comb:
            subsets.append(elt)

    return subsets
```

- *association_rules(min_confidence, support_dict):* This function generates the association rules in accordance with the given *minimum confidence* value and the provided dictionary of itemsets against their support values. It takes *min_confidence* and *support_dict* as a parameter, and returns *rules* as a list.

- For itemsets of more than one element, it first finds all their subsets calling the *find_subset(item, item_length)* function.
- For every subset A, it calculates the set B = itemset-A.
- If B is not empty, the confidence of B is calculated.
- If this value is more than *minimum confidence* value, the rule *A->B* is added to the list with the corresponding confidence value of B.

```
def association_rules(min_confidence, support_dict):
    rules = list()
    for item, support in support_dict.items():
        item_length = len(item)

        if item_length > 1:
            subsets = find_subset(item, item_length)

            for A in subsets:
                B = item.difference(A)

                if B:
                    A = frozenset(A)

                    AB = A | B

                    confidence = support_dict[AB] / support_dict[A]
                    if confidence >= min_confidence:
                        rules.append((A, B, confidence))

    return rules
```

Task 4: Output Processing

- Output of the *association_rules(min_confidence, support_dict)* function is calculated for given *min_confidence=0.6* below.

```
association_rules = association_rules(min_confidence = 0.6, support_dict =
item_support_dict)
```

- Number of rules and *association_rules* are printed. Rules are printed in the form of *A -> B <confidence: ... >*, where *A* and *B* can be a comma separated list, if they consist of more than one item.

```
print("Number of rules: ", len(association_rules), "\n")

for rule in association_rules:
    print('{0} -> {1} <confidence: {2}>'.format(set(rule[0]), set(rule[1]),
rule[2]))
```

- Here is the output of rules and their confidence values, including the number of rules:

```
Number of rules:  22

{'bottled_water', 'butter'} -> {'whole_milk'} <confidence: 0.6022727272727273>
{'domestic_eggs', 'butter'} -> {'whole_milk'} <confidence: 0.6210526315789474>
{'root_vegetables', 'butter'} -> {'whole_milk'} <confidence:
0.6377952755905512>
{'butter', 'tropical_fruit'} -> {'whole_milk'} <confidence: 0.6224489795918368>
{'butter', 'whipped_sour_cream'} -> {'whole_milk'} <confidence: 0.66>
{'yogurt', 'butter'} -> {'whole_milk'} <confidence: 0.6388888888888888>
{'curd', 'tropical_fruit'} -> {'whole_milk'} <confidence: 0.6336633663366337>
{'domestic_eggs', 'margarine'} -> {'whole_milk'} <confidence:
0.6219512195121952>
{'pip_fruit', 'domestic_eggs'} -> {'whole_milk'} <confidence:
0.6235294117647059>
{'domestic_eggs', 'tropical_fruit'} -> {'whole_milk'} <confidence:
0.6071428571428571>
{'onions', 'root_vegetables'} -> {'other_vegetables'} <confidence:
0.6021505376344086>
{'pip_fruit', 'whipped_sour_cream'} -> {'other_vegetables'} <confidence:
0.6043956043956045>
{'pip_fruit', 'whipped_sour_cream'} -> {'whole_milk'} <confidence:
0.6483516483516485>
{'citrus_fruit', 'root_vegetables', 'whole_milk'} -> {'other_vegetables'}
<confidence: 0.6333333333333333>
{'fruit_vegetable_juice', 'other_vegetables', 'yogurt'} -> {'whole_milk'}
<confidence: 0.6172839506172838>
{'pip_fruit', 'other_vegetables', 'root_vegetables'} -> {'whole_milk'}
<confidence: 0.675>
{'pip_fruit', 'root_vegetables', 'whole_milk'} -> {'other_vegetables'}
<confidence: 0.6136363636363636>
{'pip_fruit', 'other_vegetables', 'yogurt'} -> {'whole_milk'} <confidence:
0.625>
{'other_vegetables', 'root_vegetables', 'whipped_sour_cream'} -> {'whole_milk'}
<confidence: 0.6071428571428571>
{'other_vegetables', 'root_vegetables', 'yogurt'} -> {'whole_milk'}
<confidence: 0.6062992125984252>
```

```
{'other_vegetables', 'yogurt', 'tropical_fruit'} -> {'whole_milk'} <confidence: 0.6198347107438016>
{'root_vegetables', 'yogurt', 'tropical_fruit'} -> {'whole_milk'} <confidence: 0.7000000000000001>
```
