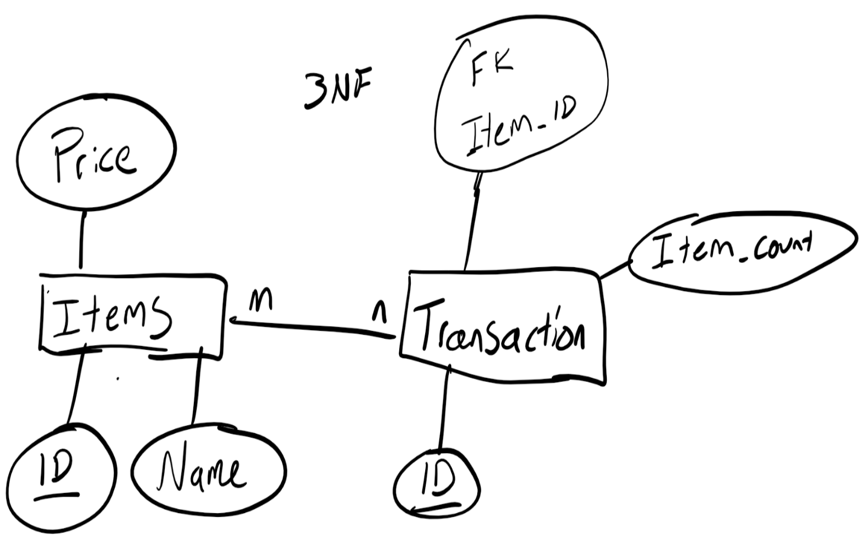
First: Nolan
Last: O'Connor

Database:

I designed my database with two entities. Price with the key attribute item_id, a name and a price (just for realism). It shares a many to many relationship with the attribute Transaction which contains a composite key of it's own transaction_id and the foreign key item_id. I also implemented a column of item_count or quantity though this has no effect on apriori. The database exists in third normal form.



The database itself has 7 tables, 5 transaction tables as years of transactions each containing 20 transactions. One test set which was only used for development purposes, and the items table of 30 convenience store items.

All Tables

```
+------------------------+
| Tables_in_cs637_midterm |
+------------------------+
| items                  |
| test                   |
| transactions_2019      |
| transactions_2020      |
| transactions_2021      |
| transactions_2022      |
| transactions_2023      |
+------------------------+
```

Items

```
mysql> select * from items;
+---------+-------------------------+-------+
| item_id | item_name               | price |
+---------+-------------------------+-------+
|       1 | Toilet Paper            |  9.99 |
|       2 | Paper Towels            | 12.99 |
|       3 | Dish Soap               |  5.99 |
|       4 | Eggs                    | 19.95 |
|       5 | Smoked Salmon Slices    | 10.95 |
|       6 | Cream Cheese            |  4.99 |
|       7 | Pack of Bagels          |  4.50 |
|       8 | Toothpaste              |  5.25 |
|       9 | Mouthwash               |  6.50 |
|      10 | Nolan's Famous Hotsauce |  1.99 |
|      11 | Tums Antacid            |  2.70 |
|      12 | AA Batteries            |  7.99 |
|      13 | Candles                 |  2.99 |
|      14 | Butter                  |  2.99 |
|      15 | Instant Coffee          |  8.15 |
|      16 | Milk                    |  5.99 |
|      17 | Sugar                   |  3.15 |
|      18 | Dried Pasta             |  1.99 |
|      19 | Chips                   |  4.50 |
|      20 | Chicken Thighs          |  8.99 |
|      21 | Drain Cleaner           |  6.75 |
|      22 | Diapers                 | 28.49 |
|      23 | Salsa                   |  3.50 |
|      24 | Baby Formula            | 19.99 |
|      25 | Bottled Water           |  7.00 |
|      26 | Shampoo                 |  5.50 |
|      27 | Ground Beef             |  8.00 |
|      28 | Vitamins                | 11.25 |
|      29 | Protein Powder          | 12.99 |
|      30 | Trash Bags              |  4.75 |
+---------+-------------------------+-------+
30 rows in set (0.01 sec)
```

Tables of transactions 2019-2023

mysql> SELECT * FROM transactions_2019;

transaction_id	item_id	quantity
1	16	1
1	28	1
1	29	1
2	5	1
2	6	1
2	16	1
2	28	1
2	29	1
3	1	1
3	10	1
3	11	1
4	1	1
4	30	1
5	1	1
5	3	1
5	30	1
6	19	1
6	23	1
6	30	1
7	3	1
7	10	1
7	11	1
8	2	1
8	20	1
8	26	1
8	27	1
9	15	1
9	18	1
9	20	1
9	21	1
9	25	1
10	5	1
10	16	1
10	28	1
10	29	1
11	1	1
11	30	1
12	1	1
12	2	1
12	30	1
13	29	1
14	5	1
14	16	1
14	28	1
15	22	1
15	24	1
16	30	1
17	22	1
17	24	1
18	14	1
19	7	1
19	10	1
19	11	1
20	5	1
20	6	1
20	16	1
20	28	1
20	29	1

mysql> SELECT * FROM transactions_2020;

transaction_id	item_id	quantity
1	16	1
1	28	1
1	29	1
2	4	1
2	13	1
2	14	1
2	16	1
2	17	1
3	1	1
3	2	1
3	14	1
4	7	1
4	14	1
5	5	1
5	6	1
5	7	1
6	4	1
6	6	1
6	7	1
7	8	1
7	10	1
7	11	1
8	4	1
8	13	1
8	14	1
8	16	1
9	1	1
9	10	1
9	11	1
9	20	1
9	30	1
10	8	1
10	9	1
10	13	1
10	19	1
11	8	1
11	9	1
12	5	1
12	6	1
12	10	1
13	26	1
14	6	1
14	7	1
14	27	1
15	10	1
15	11	1
16	25	1
17	22	1
17	24	1
18	21	1
19	6	1
19	7	1
19	14	1
20	15	1
20	18	1
20	19	1
20	20	1
20	23	1

mysql> SELECT * FROM transactions_2021;

transaction_id	item_id	quantity
1	5	1
1	6	1
1	7	1
2	1	1
2	2	1
2	3	1
2	10	1
2	11	1
3	6	1
3	7	1
3	14	1
4	22	1
4	24	1
5	4	1
5	13	1
5	14	1
6	4	1
6	13	1
6	17	1
7	8	1
7	10	1
7	11	1
8	16	1
8	19	1
8	23	1
8	29	1
9	4	1
9	13	1
9	14	1
9	16	1
9	17	1
10	5	1
10	6	1
10	7	1
10	30	1
11	6	1
11	7	1
12	5	1
12	6	1
12	10	1
13	26	1
14	6	1
14	7	1
14	27	1
15	10	1
15	11	1
16	25	1
17	22	1
17	24	1
18	21	1
19	6	1
19	7	1
19	14	1
20	15	1
20	18	1
20	19	1
20	20	1
20	23	1

mysql> SELECT * FROM transactions_2022;

transaction_id	item_id	quantity
1	1	1
1	2	1
2	5	1
2	6	1
2	7	1
2	14	1
3	6	1
3	7	1
3	8	1
3	9	1
4	4	1
4	13	1
4	14	1
4	16	1
4	17	1
5	8	1
5	9	1
6	7	1
7	1	1
7	5	1
7	6	1
7	14	1
7	30	1
8	15	1
8	19	1
8	23	1
9	10	1
10	1	1
10	2	1
10	3	1
10	16	1
10	27	1
11	4	1
12	5	1
13	6	1
14	7	1
14	14	1
15	5	1
15	6	1
16	16	1
16	28	1
17	8	1
17	9	1
17	10	1
17	21	1
17	30	1
18	19	1
18	24	1
19	4	1
19	13	1
19	14	1
19	16	1
20	5	1
20	6	1
20	8	1

mysql> select * from transactions_2023;

transaction_id	item_id	quantity
1	5	1
1	6	1
1	7	1
2	5	1
2	7	1
3	5	1
3	6	1
3	7	1
3	14	1
4	4	1
4	7	1
4	13	1
4	14	1
4	16	1
4	17	1
5	5	1
5	7	1
5	15	1
6	5	1
6	6	1
6	7	1
7	25	1
7	30	1
8	13	1
9	9	1
9	10	1
9	11	1
10	4	1
10	23	1
10	25	1
10	29	1
11	27	1
11	28	1
11	29	1
12	1	1
12	5	1
12	6	1
12	7	1
13	22	1
13	24	1
13	30	1
14	2	1
14	4	1
14	26	1
15	6	1
15	7	1
15	14	1
15	16	1
16	9	1
16	20	1
16	25	1
16	28	1
16	29	1
17	2	1
17	18	1
17	20	1
18	10	1
18	11	1
18	19	1
18	28	1
19	5	1
19	26	1
19	28	1
19	30	1
20	8	1
20	9	1
20	14	1

67 rows in set (0.00 sec)

Python Program

My program is split up into 3 python files.

1. apriori.py
   -Contains the main logic loop for the apriori algorithm
2. brute_force.py
   -Contains the main logic loop for the brute force algorithm
3. functions.py
   -Contains useful functions that both algorithms can use

Source code for apriori.py

```python
# remove pandas warning
import warnings
warnings.filterwarnings('ignore')
# ------------------------------
import mysql.connector
import pandas as pd
import functions as my_funcs
import time

conn = mysql.connector.connect(
 host="localhost",
 user="admin",
 password="root",
 database="cs637_midterm"
)

transaction_list = []
t_2023 = pd.read_sql('SELECT * FROM transactions_2023', con=conn)
t_2022 = pd.read_sql('SELECT * FROM transactions_2022', con=conn)
t_2021 = pd.read_sql('SELECT * FROM transactions_2021', con=conn)
t_2020 = pd.read_sql('SELECT * FROM transactions_2020', con=conn)
t_2019 = pd.read_sql('SELECT * FROM transactions_2019', con=conn)

# TESTING SET ONLY
test = pd.read_sql('SELECT * FROM test', con=conn)
test_list = test.groupby('transaction_id')['item_id'].apply(list).tolist()

tlist_2023 = t_2023.groupby('transaction_id')['item_id'].apply(list).tolist()
tlist_2022 = t_2022.groupby('transaction_id')['item_id'].apply(list).tolist()
tlist_2021 = t_2021.groupby('transaction_id')['item_id'].apply(list).tolist()
tlist_2020 = t_2020.groupby('transaction_id')['item_id'].apply(list).tolist()
tlist_2019 = t_2019.groupby('transaction_id')['item_id'].apply(list).tolist()

conn.close()

#-----------------
support_threshold = float(input("please enter the support threshold\n"))
confidence_threshold = float(input("please enter the confidence threshold\n"))
```

```
#------------------
data_point = int(input("which dataset would you like to run? Please input an
integer for choice\n 1- Transactions2023\n "
                  "2- Transactions2022\n 3- Transactions2021\n 4-
Transactions2020\n 5- Transactions2019\n"))
if data_point == 1:
 transaction_list = tlist_2023
elif data_point == 2:
 transaction_list = tlist_2022
elif data_point == 3:
 transaction_list = tlist_2021
elif data_point == 4:
 transaction_list = tlist_2020
elif data_point == 5:
 transaction_list = tlist_2019
else:
 print("Please restart and input a valid integer 1-5")

start_time = time.time()
# get unique single unique elements from dataset
unique_ints = sorted(set([i for sublist in transaction_list for i in
sublist]))
unique_groups = my_funcs.unique_groupings(1, unique_ints)
print(unique_ints)
print(unique_groups)

i = 0
rules = []
while len(unique_groups) != 0:
 print(f"~~~~~We begin a new loop: {i+1} at {time.time()-start_time}
seconds~~~~~")
 unique_ints = sorted(set([i for sublist in unique_groups for i in sublist]))
 if i != 0:
   unique_groups = my_funcs.unique_groupings(i + 1, unique_ints)
 print(f"The selected transaction list is {transaction_list}")
 print(f"The unique elements are {unique_groups}")

 # check the support of each unique grouping and then drop any below threshold
 supported_groups = my_funcs.calculate_support(unique_groups,
transaction_list)
 unique_groups = my_funcs.drop_check(supported_groups, support_threshold,
unique_groups)
 print(f"New working list: {unique_groups}")
 print(f"Transaction List: {transaction_list}")

 if(i != 0):
   # if confidence high enough add to list of rules
   supported_confidence = my_funcs.calculate_confidence(unique_groups,
transaction_list)
```

```
    unique_groups = my_funcs.drop_check(supported_confidence,
confidence_threshold, unique_groups)

    print(f"New working list: {unique_groups}")
    print(f"Transaction List: {transaction_list}")
    rules+=(my_funcs.new_rules(unique_groups, transaction_list))

 i+=1

print(f"\nThe program has completed running in {i} iterations.\nWe generated
{len(rules)} new rules"
      f"that satisfy Support > {support_threshold} and Confidence >
{confidence_threshold}")
print(f"The program took {time.time()-start_time} seconds to complete")
for rule in rules:
 print(rule)
```

Source code for brute_force.py

```
# remove pandas warning
import warnings
warnings.filterwarnings('ignore')
# ---------------------------------
import mysql.connector
import pandas as pd
import functions as my_funcs  # the functions I built for the assignment
from itertools import combinations  # used only to generate the combinations
of purchased items
import time # used to mark time at beginning and ending of program
import threading  #used to run a kill button method at the same time as the
other program

# connect to database
conn = mysql.connector.connect(
 host="localhost",
 user="admin",
 password="root",
 database="cs637_midterm"
)

# get the 5 tables from database
transaction_list = []
t_2023 = pd.read_sql('SELECT * FROM transactions_2023', con=conn)
t_2022 = pd.read_sql('SELECT * FROM transactions_2022', con=conn)
t_2021 = pd.read_sql('SELECT * FROM transactions_2021', con=conn)
t_2020 = pd.read_sql('SELECT * FROM transactions_2020', con=conn)
t_2019 = pd.read_sql('SELECT * FROM transactions_2019', con=conn)
```

```python
# TESTING SET ONLY
test = pd.read_sql('SELECT * FROM test', con=conn)
test_list = test.groupby('transaction_id')['item_id'].apply(list).tolist()

tlist_2023 = t_2023.groupby('transaction_id')['item_id'].apply(list).tolist()
tlist_2022 = t_2022.groupby('transaction_id')['item_id'].apply(list).tolist()
tlist_2021 = t_2021.groupby('transaction_id')['item_id'].apply(list).tolist()
tlist_2020 = t_2020.groupby('transaction_id')['item_id'].apply(list).tolist()
tlist_2019 = t_2019.groupby('transaction_id')['item_id'].apply(list).tolist()

conn.close()

#------------------
support_threshold = float(input("please enter the support threshold\n"))
confidence_threshold = float(input("please enter the confidence threshold\n"))

data_point = int(input("which dataset would you like to run? Please input an
integer for choice\n 1- Transactions2023\n "
                  "2- Transactions2022\n 3- Transactions2021\n 4-
Transactions2020\n 5- Transactions2019\n"))
if data_point == 1:
 transaction_list = tlist_2023
elif data_point == 2:
 transaction_list = tlist_2022
elif data_point == 3:
 transaction_list = tlist_2021
elif data_point == 4:
 transaction_list = tlist_2020
elif data_point == 5:
 transaction_list = tlist_2019
else:
 print("Please restart and input a valid integer 1-5")

# start kill button thread
input("Press x to end the program at any time,\npress enter to continue\n")
t = threading.Thread(target=my_funcs.kill_button)
t.start()

start_time = time.time()
# get unique single unique elements from dataset
unique_ints = sorted(set([i for sublist in transaction_list for i in
sublist]))
unique_groups = my_funcs.unique_groupings(1, unique_ints)
print(unique_ints)
print(unique_groups)
i = 0
rules = []

while True:
```

```
print(f"~~~~~We begin a new loop: {i+1} at {time.time()-start_time}
seconds~~~~~")
unique_ints = sorted(set([i for sublist in unique_groups for i in sublist]))
confidences = []
supports = []

if i != 0:
    unique_groups = my_funcs.unique_groupings(i + 1, unique_ints)
    confidences = my_funcs.calculate_confidence(unique_groups,
transaction_list, do_print=False)
 supports = my_funcs.calculate_support(unique_groups, transaction_list,
do_print=False)
 # I have disabled the output here as it gets a bit excessive but feel free to
uncomment
 #print(f"The selected transaction list is {transaction_list}")
 #print(f"The unique elements are {unique_groups}")

# here we use n for number of old rules. If it is different later we know the
program should end
 if i != 0:
    my_dict = {}
    n = len(rules)
    for e, group in enumerate(unique_groups):
       if supports[e] >= support_threshold and confidences[e] >=
confidence_threshold:
          my_dict[tuple(sorted(group[:-1]))] = (group[-1], supports[e],
confidences[e])

    for key in my_dict:
          value = my_dict[key]
          rules.append(f"{list(key)} -> {value[0]} | Support: {value[1]}
Confidence: {value[2]}")
          print(f"NEW RULE: {list(key)} -> {value[0]} | Support: {value[1]}
Confidence: {value[2]}")
    # if no new rules were found exit the program
    if n == len(rules):
       break

 i += 1
print(len(my_dict))
print(f"\nThe program has completed running in {i+1} iterations.\nWe generated
{len(rules)} new rules"
       f"that satisfy Support > {support_threshold} and Confidence >
{confidence_threshold}")
print(f"The program took {time.time()-start_time} seconds to complete")
for rule in rules:
 print(rule)
```

Source code for functions.py

```
from itertools import permutations
import os
import keyboard


def kill_button():
   while True:
       if keyboard.is_pressed('x'):
           print(" -kill button ended program")
           os._exit(0)
           break


def unique_groupings(length, unique_ints):
   if length == 1:
       groupings = [[i] for i in unique_ints]
   elif length > 1:
       groupings = list(permutations(unique_ints, length))
       groupings = [list(item) for item in groupings]
   else:
       return []
   return groupings


def calculate_support(unique_groups, transaction_list, do_print=True):
   # calculates the support for each item in unique_groups for appearance
within items of order_list
   result = []
   for group in unique_groups:
       count = 0
       for transaction in transaction_list:
           if all(x in transaction for x in group):
               count += 1
       if do_print:
           print(f"Item:{group}, Support: {round(count /
len(transaction_list), 2)}")
       result.append(count / len(transaction_list))
   return result


def drop_check(calculated_supports, support_threshold, unique_groups):
   supported_groups = []
   for i, support in enumerate(calculated_supports):
       if support < support_threshold:
           print(f"dropping{unique_groups[i]}")
       else:
           supported_groups.append(unique_groups[i])
   return supported_groups
```

```python
def calculate_confidence(unique_groups, transaction_list, do_print=True):
    # returns an array of confidence values for the unique groups
    result = []
    for group in unique_groups:
        X = group[:-1]  # everything but the last element
        Y = group[-1]  # the last element
        xy_count = 0  # appearance of X and Y together
        x_count = 0  # appearance of just X
        for transaction in transaction_list:
            if all(Xi in transaction for Xi in X):
                x_count += 1
                if Y in transaction:
                    xy_count += 1
        # calculate confidence
        if(x_count == 0):
            confidence = 0
        else:
            confidence = (round(xy_count / x_count, 2))
        result.append(confidence)
        if do_print:
            print(f"Item:{X}-> {Y}, Confidence: {round(confidence,2)}")

    return result


def new_rules(unique_groups, transaction_list):
    supports = calculate_support(unique_groups, transaction_list,
do_print=False)
    confidence = calculate_confidence(unique_groups, transaction_list,
do_print=False)
    new_rules = []
    my_dict = {}

    # first put all the rules together, the premise, the conclusion, and the
support and confidence for them.
    for i, group in enumerate(unique_groups):
        my_dict[tuple(sorted(group[:-1]))] = (group[-1], supports[i],
confidence[i])

    # we add the rule only if it is not a duplicated rule
    # this will make sure we do not have duplicated rules ie(1,2) implies 3 is
the same as (2,1) implies 3
    for key in my_dict:
        value = my_dict[key]
        new_rules.append(f"{list(key)} -> {value[0]} | Support: {value[1]}
Confidence: {value[2]}")
```

```
        print(f"NEW RULE: {list(key)} -> {value[0]} | Support: {value[1]}
Confidence: {value[2]}")
    return new_rules
```

Example Runthrough:

When you first run either **apriori.py** or **brute_force.py** the database will be loaded from mysql. You will then be presented with 3 choices. First you must enter a float for the **support_threshold**, Second you must enter a float for the **confidence_threshold**, Third you must choose which transactional table to run by inputting an integer value. (On the **brute_force.py** a fourth option will appear which will only have you confirm the kill button which is ‘x’)

```
C:\Users\nolan\PycharmProjects\Data_Mining_Midterm\venv\Scripts\python.exe
please enter the support threshold
.1
please enter the confidence threshold
.1
which dataset would you like to run? Please input an integer for choice
 1- Transactions2023
 2- Transactions2022
 3- Transactions2021
 4- Transactions2020
 5- Transactions2019
5
Press x to end the program at any time,
press enter to continue
```

At this point the timers for both scripts will begin. In both files I use the line below to grab all rows from the table which contain a unique item_id and put them as **unique_ints**.

```
# get unique single unique elements from dataset
unique_ints = sorted(set([i for sublist in transaction_list for i in sublist]))
unique_groups = my_funcs.unique_groupings(1, unique_ints)
print(unique_ints)
print(unique_groups)
```

This is a list of every item purchased. I then form groups of those ints using the **functions.unique_groupings(length, unique_ints)** which will return a list of groups of transactions to check for. So in the case of our initial run it will return single item elements because at first we are looking for all single items which have a support greater than the threshold.

Here are examples of 1, 2, and 3 length groupings for apriori. The first group is simply all the purchased items. Note that not every item is always included in the transaction table, which is why 4 is skipped for instance. (I thought it was more realistic this way as not every item is always purchased.

```
[1, 2, 3, 5, 6, 7, 10, 11, 14, 15, 16, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30]
[[1], [2], [3], [5], [6], [7], [10], [11], [14], [15], [16], [18], [19], [20], [21], [22], [23], [24], [25], [26], [27], [28], [29], [30]]
```

2 length set is built off of every single item which had high enough support

```
The selected transaction list is [[16, 28, 29], [5, 6, 16, 28, 29], [1, 10, 11], [1, 30], [1, 3, 30], [19, 23, 30], [3, 10, 11], [2, 20, 26, 27], [
The unique elements are [[1, 2], [1, 3], [1, 5], [1, 6], [1, 10], [1, 11], [1, 16], [1, 20], [1, 22], [1, 24], [1, 28], [1, 29], [1, 30], [2, 1], [
```

3 length groups are built off of every double group which had high support and confidence

```
The selected transaction list is [[16, 28, 29], [5, 6, 16, 28, 29], [1, 10, 11], [1, 30], [1, 3, 30], [19, 23, 30], [3, 10, 11], [2, 20, 26, 27], [15, 18,
The unique elements are [[1, 5, 6], [1, 5, 10], [1, 5, 11], [1, 5, 16], [1, 5, 22], [1, 5, 24], [1, 5, 28], [1, 5, 29], [1, 5, 30], [1, 6, 5], [1, 6, 10],
```

And so on…
In brute_force this function also works, we just don't ever drop previous groups for their lack of support/confidence.)

Now we enter the while loop and the main section of the program. The groups are initialized outside the main while loop because in **apriori** my while loop exits once the **unique_groups** length is zero, meaning all previous groups had been dropped and no new rules can be built. In **brute_force** we exit once no new rules have been created because the length of groups is always going to increase since old groups are never dropped. The program would otherwise continue to create permutations of old groups infinitely.

We start a loop by stating the loop number, and the time since the start of program. We also print out the selected **transaction_list** and the **unique_groups**. I found this helpful because you can see the process behind the algorithm. I had commented out these print lines in the **brute_force** version because the print statements were getting excessive and made it harder to follow.

```python
i = 0
rules = []
while len(unique_groups) != 0:
  print(f"~~~~~We begin a new loop: {i+1} at {time.time()-start_time} seconds~~~~~")
  unique_ints = sorted(set([i for sublist in unique_groups for i in sublist]))
  if i != 0:
    unique_groups = my_funcs.unique_groupings(i + 1, unique_ints)
  print(f"The selected transaction list is {transaction_list}")
  print(f"The unique elements are {unique_groups}")
```

At this point we gather a list of **supported_groups** which contains all groups from the **unique_groups** which have support greater than the threshold using the **functions.calculate_support** method. Which is essentially a list of fractions (times when that grouping appeared in all transactions divided by the total number of transactions)

```python
def calculate_support(unique_groups, transaction_list, do_print=True):
    # calculates the support for each item in unique_groups for appearance within items of order_list
    result = []
    for group in unique_groups:
        count = 0
        for transaction in transaction_list:
            if all(x in transaction for x in group):
                count += 1
        if do_print:
            print(f"Item:{group}, Support: {round(count / len(transaction_list), 2)}")
        result.append(count / len(transaction_list))
    return result
```

In **apriori** we then drop all groups which do not have support, using **functions.drop_check()** method and in **brute_force** we do not.

If we are not on the first iteration we also check for confidence using the **functions.calculate_confidence()** method which is essentially a fraction of (times X and Y appear together across the whole transaction list divided by the number of times X appeared by itself across the entire transaction list)
Once again, in the case of **apriori** we also drop any groups that do not meet or exceed the confidence threshold again using the **functions.drop_check()** method.

```
def calculate_confidence(unique_groups, transaction_list, do_print=True):
    # returns an array of confidence values for the unique groups
    result = []
    for group in unique_groups:
        X = group[:-1]  # everything but the last element
        Y = group[-1]  # the last element
        xy_count = 0  # appearance of X and Y together
        x_count = 0  # appearance of just X
        for transaction in transaction_list:
            if all(Xi in transaction for Xi in X):
                x_count += 1
                if Y in transaction:
                    xy_count += 1
        # calculate confidence
        if(x_count == 0):
            confidence = 0
        else:
            confidence = (round(xy_count / x_count, 2))
        result.append(confidence)
        if do_print:
            print(f"Item:{X}-> {Y}, Confidence: {round(confidence,2)}")

    return result
```

Now we generate our rules for the while loop.

In **apriori** we only need to use our working list of **unique_groups** and put them into a rule format. All groups that do not meet support or confidence thresholds have been removed already. So we pass **unique_groups** into **functions.new_rules()** method

```
def new_rules(unique_groups, transaction_list):
    supports = calculate_support(unique_groups, transaction_list, do_print=False)
    confidence = calculate_confidence(unique_groups, transaction_list, do_print=False)
    new_rules = []
    my_dict = {}

    # first put all the rules together, the premise, the conclusion, and the support and confidence for them.
    for i, group in enumerate(unique_groups):
        my_dict[tuple(sorted(group[:-1]))] = (group[-1], supports[i], confidence[i])

    # we add the rule only if it is not a duplicated rule
    # this will make sure we do not have duplicated rules ie(1,2) implies 3 is the same as (2,1) implies 3
    for key in my_dict:
        value = my_dict[key]
        new_rules.append(f"{list(key)} -> {value[0]} | Support: {value[1]} Confidence: {value[2]}")
        print(f"NEW RULE: {list(key)} -> {value[0]} | Support: {value[1]} Confidence: {value[2]}")
    return new_rules
```

new_rules() takes in the list of groups and the transaction list, it generates a list of supports and confidences based on the group. For each group it then adds the ruleset to a dictionary. I

decided to use a dictionary to remove any duplicate entries since in apriori [1,2] -> 3 is the same as [2,1] -> 3. So my method takes everything but the last element of the group (group[:-1]) and sorts them, then adds them to a dictionary as the key. So for the example group [1,2,3] what we mean is 1 and 2 implies 3. So we take everything but the last element [1,2] and sort it, which means if there was a duplicate of [2,1] it would also be sorted to [1,2] when you try to add a duplicate key to a dictionary nothing happens, which is why I chose to use a dictionary. The value for that key is then stored as a tuple of (the implication of the key, the support of the key and implication, and the confidence of the key and implication)
I then parse through the dictionary and generate strings of easy to read rules. When a new rule is created I add it to the **new_rules** set and print out the new rule so the program is transparent.

In **brute_force** we generate rules a little differently. Since no groups have ever been dropped we need to iterate through the list and check supports and confidences for each, then only add the new rule if these conditions are met. Other than that the logic works the same with using a dictionary to avoid duplications of rules and then printing them out in a more readable way to string.

Program Complete!

Again in Apriori once the length of new_groups is 0 we end (since this means all previous groups are dropped and therefore we cannot build any new rules) or in Brute Force once no new rules are generated, we end the main while loop.

Now all that's left is to print out the time it took to complete, the number of iterations we went through, and all the rules in our ruleset.

Conclusion

Here you can see the side by side outputs of apriori vs brute force on transactions_2023 with support > .1 and confidence > .1

```
The program has completed running in 4 iterations.
We generated 17 new rulesthat satisfy Support > 0.1 and Confidence > 0.1
The program took 0.19698882102966309 seconds to complete
[5] -> 7 | Support: 0.3 Confidence: 0.86
[6] -> 14 | Support: 0.1 Confidence: 0.4
[7] -> 16 | Support: 0.1 Confidence: 0.25
[10] -> 11 | Support: 0.1 Confidence: 1.0
[11] -> 10 | Support: 0.1 Confidence: 1.0
[14] -> 16 | Support: 0.1 Confidence: 0.5
[16] -> 14 | Support: 0.1 Confidence: 1.0
[25] -> 29 | Support: 0.1 Confidence: 0.67
[28] -> 29 | Support: 0.1 Confidence: 0.5
[29] -> 28 | Support: 0.1 Confidence: 0.67
[5, 6] -> 7 | Support: 0.2 Confidence: 1.0
[5, 7] -> 6 | Support: 0.2 Confidence: 0.67
[6, 7] -> 14 | Support: 0.1 Confidence: 0.4
[6, 14] -> 7 | Support: 0.1 Confidence: 1.0
[7, 14] -> 16 | Support: 0.1 Confidence: 0.67
[7, 16] -> 14 | Support: 0.1 Confidence: 1.0
[14, 16] -> 7 | Support: 0.1 Confidence: 1.0

Process finished with exit code 0
```

```
The program has completed running in 4 iterations.
We generated 17 new rulesthat satisfy Support > 0.1 and Confidence > 0.1
The program took 7.927527189254761 seconds to complete
[5] -> 7 | Support: 0.3 Confidence: 0.86
[6] -> 14 | Support: 0.1 Confidence: 0.4
[7] -> 16 | Support: 0.1 Confidence: 0.25
[10] -> 11 | Support: 0.1 Confidence: 1.0
[11] -> 10 | Support: 0.1 Confidence: 1.0
[14] -> 16 | Support: 0.1 Confidence: 0.5
[16] -> 14 | Support: 0.1 Confidence: 1.0
[25] -> 29 | Support: 0.1 Confidence: 0.67
[28] -> 29 | Support: 0.1 Confidence: 0.5
[29] -> 28 | Support: 0.1 Confidence: 0.67
[5, 6] -> 7 | Support: 0.2 Confidence: 1.0
[5, 7] -> 6 | Support: 0.2 Confidence: 0.67
[6, 7] -> 14 | Support: 0.1 Confidence: 0.4
[6, 14] -> 7 | Support: 0.1 Confidence: 1.0
[7, 14] -> 16 | Support: 0.1 Confidence: 0.67
[7, 16] -> 14 | Support: 0.1 Confidence: 1.0
[14, 16] -> 7 | Support: 0.1 Confidence: 1.0
x -kill button ended program

Process finished with exit code 0
```

As you can see both algorithms have the same output of 17 rules, however apriori only took ~.2 seconds, while brute force took nearly 8 full seconds.

Brute force gets exponentially worse because it is building permutations of infinitely growing data until it finishes. Below you can see that the brute force works well at first, starting both the first and second loops at 0.0 seconds. At the third iteration it jumps up to .014. By the start of the 4th it goes up to .34, and by the time it confirms no new rules are generated at the end of the 4th it is a whopping 7.9 seconds.

```
~~~~~We begin a new loop: 1 at 0.0 seconds~~~~~
~~~~~We begin a new loop: 2 at 0.0 seconds~~~~~
NEW RULE: [5] -> 7 | Support: 0.3 Confidence: 0.86
NEW RULE: [6] -> 14 | Support: 0.1 Confidence: 0.4
NEW RULE: [7] -> 16 | Support: 0.1 Confidence: 0.25
NEW RULE: [10] -> 11 | Support: 0.1 Confidence: 1.0
NEW RULE: [11] -> 10 | Support: 0.1 Confidence: 1.0
NEW RULE: [14] -> 16 | Support: 0.1 Confidence: 0.5
NEW RULE: [16] -> 14 | Support: 0.1 Confidence: 1.0
NEW RULE: [25] -> 29 | Support: 0.1 Confidence: 0.67
NEW RULE: [28] -> 29 | Support: 0.1 Confidence: 0.5
NEW RULE: [29] -> 28 | Support: 0.1 Confidence: 0.67
~~~~~We begin a new loop: 3 at 0.014606952667236328 seconds~~~~~
NEW RULE: [5, 6] -> 7 | Support: 0.2 Confidence: 1.0
NEW RULE: [5, 7] -> 6 | Support: 0.2 Confidence: 0.67
NEW RULE: [6, 7] -> 14 | Support: 0.1 Confidence: 0.4
NEW RULE: [6, 14] -> 7 | Support: 0.1 Confidence: 1.0
NEW RULE: [7, 14] -> 16 | Support: 0.1 Confidence: 0.67
NEW RULE: [7, 16] -> 14 | Support: 0.1 Confidence: 1.0
NEW RULE: [14, 16] -> 7 | Support: 0.1 Confidence: 1.0
~~~~~We begin a new loop: 4 at 0.341555118560791 seconds~~~~~
0

The program has completed running in 4 iterations.
We generated 17 new rulesthat satisfy Support > 0.1 and Confidence > 0.1
The program took 7.927527189254761 seconds to complete
```

And compared to apriori times

```
~~~~~We begin a new loop: 1 at 0.0 seconds~~~~~
```

```
~~~~~We begin a new loop: 2 at 0.0009419918060302734 seconds~~~~~
```

```
~~~~~We begin a new loop: 3 at 0.04716610908508301 seconds~~~~~
```

```
~~~~~We begin a new loop: 4 at 0.18604540824890137 seconds~~~~~
```

We can see that at iteration 1 we start also at 0.0. Interestingly loop 2 is behind brute force, this is because apriori has a little more work to do up front with checking support and confidences and dropping groups. By loop 3 we are once again a little behind brute force which is only at .014. Once we get to loop 4 though, apriori takes the lead at nearly half the time of brute force (.18 to .34. And then at the final output apriori is faster by a factor of 40. This would only increase as the iterations grow and on a side note I also ran brute_force with the transactions_2019 table and it crashed for using too much memory because the iterations were exceeding 6.

All Outputs Comparison

You can now see the output for each table at similar settings of **support >= .2 confidence >=.2**

Transaction 2019 outputs (apriori left, brute force right)

```
The program has completed running in 4 iterations.
We generated 11 new rulesthat satisfy Support > 0.2 and Confidence > 0.2
The program took 0.058602094650268555 seconds to complete
[1] -> 30 | Support: 0.2 Confidence: 0.8
[5] -> 28 | Support: 0.2 Confidence: 1.0
[16] -> 29 | Support: 0.2 Confidence: 0.8
[28] -> 29 | Support: 0.2 Confidence: 0.8
[29] -> 28 | Support: 0.2 Confidence: 0.8
[30] -> 1 | Support: 0.2 Confidence: 0.67
[5, 16] -> 28 | Support: 0.2 Confidence: 1.0
[5, 28] -> 16 | Support: 0.2 Confidence: 1.0
[16, 28] -> 29 | Support: 0.2 Confidence: 0.8
[16, 29] -> 28 | Support: 0.2 Confidence: 1.0
[28, 29] -> 16 | Support: 0.2 Confidence: 1.0

Process finished with exit code 0
```

```
The program has completed running in 4 iterations.
We generated 11 new rulesthat satisfy Support > 0.2 and Confidence > 0.2
The program took 5.031222581863403 seconds to complete
[1] -> 30 | Support: 0.2 Confidence: 0.8
[5] -> 28 | Support: 0.2 Confidence: 1.0
[16] -> 29 | Support: 0.2 Confidence: 0.8
[28] -> 29 | Support: 0.2 Confidence: 0.8
[29] -> 28 | Support: 0.2 Confidence: 0.8
[30] -> 1 | Support: 0.2 Confidence: 0.67
[5, 16] -> 28 | Support: 0.2 Confidence: 1.0
[5, 28] -> 16 | Support: 0.2 Confidence: 1.0
[16, 28] -> 29 | Support: 0.2 Confidence: 0.8
[16, 29] -> 28 | Support: 0.2 Confidence: 1.0
[28, 29] -> 16 | Support: 0.2 Confidence: 1.0
x -kill button ended program

Process finished with exit code 0
```

Transaction 2020 outputs (apriori left, brute force right)

```
The program has completed running in 3 iterations.
We generated 2 new rulesthat satisfy Support > 0.2 and Confidence > 0.2
The program took 0.012240886688232422 seconds to complete
[6] -> 7 | Support: 0.2 Confidence: 0.8
[7] -> 6 | Support: 0.2 Confidence: 0.8

Process finished with exit code 0
```

```
The program has completed running in 3 iterations.
We generated 2 new rulesthat satisfy Support > 0.2 and Confidence > 0.2
The program took 0.3757596015930176 seconds to complete
[6] -> 7 | Support: 0.2 Confidence: 0.8
[7] -> 6 | Support: 0.2 Confidence: 0.8
x -kill button ended program

Process finished with exit code 0
```

Transaction 2021 outputs (apriori left, brute force right)

```
The program has completed running in 3 iterations.
We generated 2 new rulesthat satisfy Support > 0.2 and Confidence > 0.2
The program took 0.011057376861572266 seconds to complete
[6] -> 7 | Support: 0.3 Confidence: 0.86
[7] -> 6 | Support: 0.3 Confidence: 1.0

Process finished with exit code 0
```

```
The program has completed running in 3 iterations.
We generated 2 new rulesthat satisfy Support > 0.2 and Confidence > 0.2
The program took 0.35111522674560547 seconds to complete
[6] -> 7 | Support: 0.3 Confidence: 0.86
[7] -> 6 | Support: 0.3 Confidence: 1.0
x -kill button ended program

Process finished with exit code 0
```

Transaction 2022 outputs (apriori left, brute force right)

```
The program has completed running in 3 iterations.
We generated 2 new rulesthat satisfy Support > 0.2 and Confidence > 0.2
The program took 0.011484384536743164 seconds to complete
[5] -> 6 | Support: 0.2 Confidence: 0.8
[6] -> 5 | Support: 0.2 Confidence: 0.67

Process finished with exit code 0
```

```
The program has completed running in 3 iterations.
We generated 2 new rulesthat satisfy Support > 0.2 and Confidence > 0.2
The program took 0.1775498390197754 seconds to complete
[5] -> 6 | Support: 0.2 Confidence: 0.8
[6] -> 5 | Support: 0.2 Confidence: 0.67
x -kill button ended program

Process finished with exit code 0
```

Transaction 2023 outputs (apriori left, brute force right)

```
The program has completed running in 4 iterations.
We generated 6 new rulesthat satisfy Support > 0.2 and Confidence > 0.2
The program took 0.012499094009399414 seconds to complete
[5] -> 7 | Support: 0.3 Confidence: 0.86
[6] -> 7 | Support: 0.25 Confidence: 1.0
[7] -> 6 | Support: 0.25 Confidence: 0.62
[5, 6] -> 7 | Support: 0.2 Confidence: 1.0
[5, 7] -> 6 | Support: 0.2 Confidence: 0.67
[6, 7] -> 5 | Support: 0.2 Confidence: 0.8

Process finished with exit code 0
```

```
The program has completed running in 4 iterations.
We generated 6 new rulesthat satisfy Support > 0.2 and Confidence > 0.2
The program took 8.282285451889038 seconds to complete
[5] -> 7 | Support: 0.3 Confidence: 0.86
[6] -> 7 | Support: 0.25 Confidence: 1.0
[7] -> 6 | Support: 0.25 Confidence: 0.62
[5, 6] -> 7 | Support: 0.2 Confidence: 1.0
[5, 7] -> 6 | Support: 0.2 Confidence: 0.67
[6, 7] -> 5 | Support: 0.2 Confidence: 0.8
x -kill button ended program

Process finished with exit code 0
```