Data Mining – Midterm Project

Wellington Cunha

NJIT ID: 31548454

NJIT UC ID: wc44

wc44@njit.edu

For this project, we selected Python to be the tool/language. So, the first step was to load the pip packages required:

```
import os
import pandas as pd
import random
```

As required, the first task was to create the list of of items available on our store (that we called inventory). For that we used the most common items we buy when shopping on grocery stores (Walmart, ShopRite, Stop & Shop), along with some other items commonly found in the same categories. We created a dictionary, converted it to a Pandas Dataframe and then saved it to a file:

```
inventory = [
    {"item_id": 1, "item_description": "Classic Coke"},
    {"item_id": 2, "item_description": "Sprite"},
    {"item_id": 3, "item_description": "Fanta"},
    {"item_id": 4, "item_description": "Apple Juice"},
    {"item_id": 5, "item_description": "Orange Juice"},
    {"item_id": 6, "item_description": "Pear"},
    {"item_id": 7, "item_description": "Apple"},
    {"item_id": 8, "item_description": "Grape"},
    {"item_id": 9, "item_description": "Lemon"},
    {"item_id": 10, "item_description": "Banana"},
    {"item_id": 11, "item_description": "Hot Pocket"},
    {"item_id": 12, "item_description": "Hungry Man"},
    {"item_id": 13, "item_description": "Meatlovers Pizza"},
    {"item_id": 14, "item_description": "Sliced Ham"},
    {"item_id": 15, "item_description": "Hard Salami"},
    {"item_id": 16, "item_description": "Provolone Cheese"},
    {"item_id": 17, "item_description": "Muenster Cheese"},
    {"item_id": 18, "item_description": "Bread"},
    {"item_id": 19, "item_description": "Milk"},
    {"item_id": 20, "item_description": "Coffee"},
    {"item_id": 21, "item_description": "Rice"},
    {"item_id": 22, "item_description": "Popcorn"},
    {"item_id": 23, "item_description": "Italian Sub"},
    {"item_id": 24, "item_description": "Butter"},
    {"item_id": 25, "item_description": "Eggs"},
    {"item_id": 26, "item_description": "Batteries"},
    {"item_id": 27, "item_description": "Shampoo"},
    {"item_id": 28, "item_description": "Toothpaste"},
    {"item_id": 29, "item_description": "Tylenol"},
    {"item_id": 30, "item_description": "Yogurt"},
]

inventory = pd.DataFrame(inventory)
inventory.to_csv("inventory.tsv", sep = "\t", index=False)
inventory.head()
```

Here is the content of our inventory database (inventory.tsv):

```
item_id item_description
1   Classic Coke
2   Sprite
3   Fanta
4   Apple Juice
5   Orange Juice
6   Pear
7   Apple
8   Grape
9   Lemon
10  Banana
11  Hot Pocket
12  Hungry Man
13  Meatlovers Pizza
14  Sliced Ham
15  Hard Salami
16  Provolone Cheese
17  Muenster Cheese
18  Bread
19  Milk
20  Coffee
21  Rice
22  Popcorn
23  Italian Sub
24  Butter
25  Eggs
26  Batteries
27  Shampoo
28  Toothpaste
29  Tylenol
30  Yogurt
```

Next, we created a function to generate the database by randomly combining items from our inventory. This function receives as parameters:

- **transactions**: the number of transactions to be generate into the database
- **min_items_per_transaction**: the minimum number of items per transaction
- **max_items_per_transaction**: the maximum number of items per transaction
- **file_name**: the name of the file to be saved

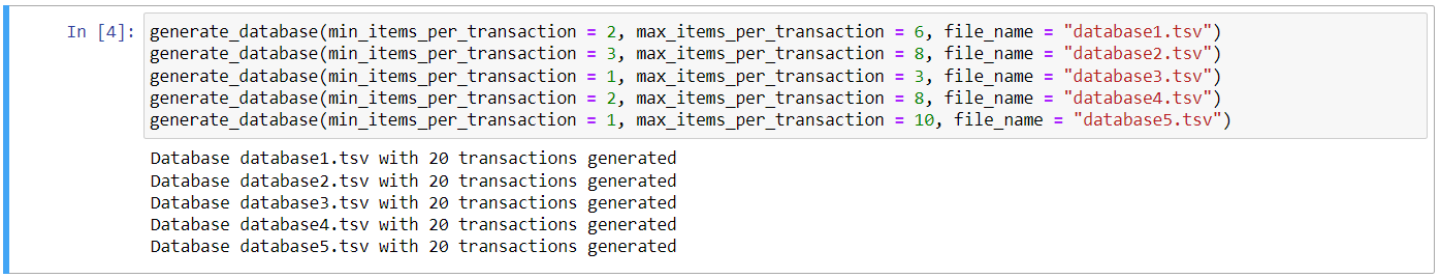
```
def generate_database(transactions = 20, min_items_per_transaction = 2,
        max_items_per_transaction = 6, file_name = "database"):

    database = []
    inventory = pd.read_csv("inventory.tsv", sep = "\t")
    for transaction in range(transactions):
        basket_items = random.sample(list(inventory["item_id"]),
                                     random.randint(min_items_per_transaction,
                                                    max_items_per_transaction))
        basket_items = inventory[
            inventory["item_id"].isin(basket_items)]["item_description"].tolist()
        database.append({
            "transaction_id": transaction + 1,
            "items": ','.join(basket_items)}
        )
    database = pd.DataFrame(database)
    database.to_csv(file_name, sep = "\t", index=False)
    print("Database ", file_name, " with ", transactions,
          " transactions generated", sep = "")
```

Then we ran the function above to generate the five databases, using different set of parameters. Starting with database 1:

```
generate_database(min_items_per_transaction = 2, max_items_per_transaction = 6,
file_name = "database1.tsv")
generate_database(min_items_per_transaction = 3, max_items_per_transaction = 8,
file_name = "database2.tsv")
generate_database(min_items_per_transaction = 1, max_items_per_transaction = 3,
file_name = "database3.tsv")
generate_database(min_items_per_transaction = 2, max_items_per_transaction = 8,
file_name = "database4.tsv")
generate_database(min_items_per_transaction = 1, max_items_per_transaction = 10,
file_name = "database5.tsv")
```

Here is the result of our execution in Jupyter Notebook:



```
In [4]: generate_database(min_items_per_transaction = 2, max_items_per_transaction = 6, file_name = "database1.tsv")
        generate_database(min_items_per_transaction = 3, max_items_per_transaction = 8, file_name = "database2.tsv")
        generate_database(min_items_per_transaction = 1, max_items_per_transaction = 3, file_name = "database3.tsv")
        generate_database(min_items_per_transaction = 2, max_items_per_transaction = 8, file_name = "database4.tsv")
        generate_database(min_items_per_transaction = 1, max_items_per_transaction = 10, file_name = "database5.tsv")

        Database database1.tsv with 20 transactions generated
        Database database2.tsv with 20 transactions generated
        Database database3.tsv with 20 transactions generated
        Database database4.tsv with 20 transactions generated
        Database database5.tsv with 20 transactions generated
```

Now, we check the values of our generated databases, starting with **Database #1**, generated with a minimum of 2 items and a maximum of 6 items per transaction:

```
transaction_id  items
1   Apple,Hard Salami,Muenster Cheese,Coffee,Rice,Toothpaste
2   Hard Salami,Provolone Cheese
3   Sprite,Hot Pocket,Hungry Man,Batteries,Yogurt
4   Hot Pocket,Popcorn,Tylenol
5   Orange Juice,Hard Salami,Batteries
6   Sliced Ham,Coffee,Tylenol
7   Apple,Bread
8   Orange Juice,Sliced Ham,Hard Salami
9   Orange Juice,Pear,Coffee
10  Apple,Banana,Hungry Man,Milk
11  Apple Juice,Orange Juice,Meatlovers Pizza,Batteries,Yogurt
12  Italian Sub,Eggs
13  Fanta,Meatlovers Pizza,Popcorn,Shampoo
14  Grape,Hard Salami,Batteries
15  Apple,Provolone Cheese,Batteries
16  Orange Juice,Milk,Toothpaste
17  Orange Juice,Rice,Italian Sub,Batteries,Toothpaste
18  Hot Pocket,Provolone Cheese
19  Fanta,Orange Juice,Sliced Ham
20  Grape,Lemon,Tylenol
```

Database #2, generated with a minimum of 3 items and a maximum of 8 items per transaction:

transaction_id	items
1	Banana,Butter,Toothpaste
2	Orange Juice,Pear,Grape,Hard Salami,Coffee,Italian Sub
3	Apple,Grape,Lemon,Meatlovers Pizza,Muenster Cheese,Coffee,Popcorn
4	Classic Coke,Apple Juice,Grape,Banana,Bread,Milk,Butter,Toothpaste
5	Sprite,Fanta,Orange Juice,Lemon,Hot Pocket,Batteries,Shampoo
6	Fanta,Grape,Lemon,Sliced Ham,Popcorn
7	Apple Juice,Pear,Banana,Bread,Milk,Italian Sub,Eggs,Yogurt
8	Lemon,Provolone Cheese,Milk,Rice,Italian Sub,Shampoo,Tylenol
9	Orange Juice,Milk,Eggs
10	Apple Juice,Grape,Hot Pocket,Popcorn,Batteries,Shampoo
11	Classic Coke,Sprite,Banana,Rice,Popcorn,Tylenol
12	Apple,Hungry Man,Muenster Cheese,Butter,Shampoo
13	Orange Juice,Pear,Hot Pocket,Coffee
14	Sprite,Pear,Grape,Lemon,Hungry Man,Hard Salami,Bread,Yogurt
15	Classic Coke,Pear,Banana,Italian Sub,Butter
16	Orange Juice,Grape,Meatlovers Pizza,Coffee,Batteries,Tylenol
17	Milk,Toothpaste,Yogurt
18	Sprite,Orange Juice,Apple,Bread,Butter,Tylenol
19	Grape,Hot Pocket,Hungry Man,Provolone Cheese,Bread,Tylenol
20	Apple Juice,Sliced Ham,Bread,Coffee,Rice,Popcorn,Yogurt

Database #3, generated with a minimum of 1 item and a maximum of 3 items per transaction:

transaction_id	items
1	Sprite,Milk
2	Sliced Ham
3	Hot Pocket,Toothpaste
4	Milk
5	Apple,Yogurt
6	Batteries
7	Grape,Eggs
8	Apple Juice,Apple,Provolone Cheese
9	Fanta,Muenster Cheese,Batteries
10	Classic Coke
11	Muenster Cheese,Rice
12	Hot Pocket
13	Eggs
14	Sprite
15	Tylenol
16	Banana,Eggs
17	Eggs,Tylenol
18	Yogurt
19	Orange Juice,Grape,Sliced Ham
20	Milk,Tylenol

Database #4, generated with a minimum of 2 items and a maximum of 8 items per transaction:

transaction_id	items
1	Fanta,Apple Juice,Meatlovers Pizza,Provolone Cheese,Muenster Cheese,Rice,Batteries,Toothpaste
2	Sprite,Apple Juice,Apple,Banana,Hot Pocket,Hard Salami,Muenster Cheese,Tylenol
3	Apple Juice,Orange Juice,Pear,Banana,Provolone Cheese,Batteries,Shampoo
4	Classic Coke,Apple,Coffee,Rice
5	Grape,Hungry Man,Milk,Tylenol
6	Fanta,Apple Juice,Milk,Tylenol
7	Fanta,Pear,Hungry Man,Provolone Cheese,Italian Sub,Butter,Tylenol
8	Sliced Ham,Bread,Toothpaste,Tylenol
9	Classic Coke,Sprite,Pear,Sliced Ham,Muenster Cheese,Batteries
10	Orange Juice,Grape,Muenster Cheese,Toothpaste,Tylenol
11	Classic Coke,Apple Juice,Pear,Hot Pocket,Bread,Butter,Shampoo
12	Hungry Man,Rice
13	Sprite,Apple,Hard Salami,Bread,Popcorn,Italian Sub,Tylenol
14	Italian Sub,Tylenol,Yogurt
15	Provolone Cheese,Butter,Toothpaste,Yogurt
16	Classic Coke,Apple Juice,Hot Pocket,Hungry Man,Sliced Ham,Muenster Cheese,Milk,Butter
17	Lemon,Sliced Ham
18	Sprite,Apple,Hot Pocket,Hard Salami,Butter,Batteries,Shampoo,Tylenol
19	Orange Juice,Lemon,Sliced Ham,Butter,Yogurt
20	Classic Coke,Provolone Cheese

And finally, **Database #5**, generated with a minimum of 1 item and a maximum of 10 items per transaction:

transaction_id	items
1	Orange Juice,Apple,Hard Salami,Provolone Cheese,Popcorn,Eggs,Batteries,Toothpaste,Yogurt
2	Toothpaste
3	Sprite,Pear,Apple,Hot Pocket,Sliced Ham,Provolone Cheese,Eggs,Tylenol,Yogurt
4	Banana,Hot Pocket,Hungry Man,Provolone Cheese,Bread,Coffee,Rice,Batteries,Tylenol,Yogurt
5	Grape,Lemon,Hot Pocket,Milk,Batteries
6	Pear,Meatlovers Pizza,Butter,Batteries
7	Fanta,Apple,Grape,Hot Pocket,Sliced Ham,Italian Sub,Toothpaste,Yogurt
8	Fanta,Grape
9	Apple Juice,Muenster Cheese,Milk,Coffee,Butter,Yogurt
10	Classic Coke,Fanta,Apple Juice,Grape,Hot Pocket,Rice,Eggs,Toothpaste
11	Classic Coke,Apple Juice,Hot Pocket,Toothpaste
12	Classic Coke,Yogurt
13	Classic Coke,Fanta,Hungry Man,Sliced Ham,Hard Salami,Provolone Cheese,Rice,Italian Sub,Batteries
14	Lemon,Banana,Milk,Rice,Batteries
15	Orange Juice,Hot Pocket,Meatlovers Pizza,Milk,Coffee,Popcorn,Eggs
16	Orange Juice,Apple,Meatlovers Pizza,Sliced Ham,Provolone Cheese,Coffee,Popcorn,Batteries,Shampoo,Tylenol
17	Hard Salami
18	Classic Coke,Grape,Hungry Man,Hard Salami,Bread,Milk,Coffee,Rice,Italian Sub,Shampoo
19	Sprite,Fanta,Hot Pocket,Provolone Cheese,Rice,Popcorn,Toothpaste
20	Lemon,Hot Pocket,Meatlovers Pizza,Sliced Ham

We created then a function to generate the combinations of items (item set). This function receives as parameters:

- **item_list**: the list of items
- **num_items**: the number of items to be combined in the item set

This function will be used on both algorithms (brute-force and Apriori)

```
test = ["a", "b", "c", "d"]

def generate_combinations(item_list, num_items):
    comb_list = []
    def comb(combinations, item_list, n):
        if n == 0:
            combined_items = combinations[:-1].split("|")
            combined_items.sort()
            comb_list.append(combined_items)
        else:
            for i in range(len(item_list)):
                comb(combinations + item_list[i] + "|", item_list[i+1:], n-1)
    comb("", item_list, num_items)
    return comb_list

print(generate_combinations(test, 1))
print(generate_combinations(test, 2))
print(generate_combinations(test, 3))
print(generate_combinations(test, 4))
```

Below is the evidence of execution of the tests above:

In [6]:
```
test = ["a", "b", "c", "d"]

def generate_combinations(item_list, num_items):
    comb_list = []
    def comb(combinations, item_list, n):
        if n == 0:
            combined_items = combinations[:-1].split("|")
            combined_items.sort()
            comb_list.append(combined_items)
        else:
            for i in range(len(item_list)):
                comb(combinations + item_list[i] + "|", item_list[i+1:], n-1)
    comb("", item_list, num_items)
    return comb_list

print(generate_combinations(test, 1))
print(generate_combinations(test, 2))
print(generate_combinations(test, 3))
print(generate_combinations(test, 4))
```

```
[['a'], ['b'], ['c'], ['d']]
[['a', 'b'], ['a', 'c'], ['a', 'd'], ['b', 'c'], ['b', 'd'], ['c', 'd']]
[['a', 'b', 'c'], ['a', 'b', 'd'], ['a', 'c', 'd'], ['b', 'c', 'd']]
[['a', 'b', 'c', 'd']]
```

We then created another function to check if an itemset belongs to a superset. This function returns 1 if the itemset (subset) belongs to the superset or 0 if the itemset does not belong to the superset or if the itemset has more items than the superset. The function receives the following parameters:

- **itemset**: the subset to be checked agains the superset
- **transaction_items**: the superset

```
transaction = ["a", "b", "c", "d"]

def check_belonging(itemset, transaction_items):
    belong = 0
    if len(itemset) > len(transaction_items):
        belong = 0
    elif(all(item in transaction_items for item in itemset)):
        belong = 1
    return belong

print(check_belonging(["a"], transaction))
print(check_belonging(["e"], transaction))
print(check_belonging(["b", "c"], transaction))
print(check_belonging(["a", "b", "c", "d"], transaction))
print(check_belonging(["a", "b", "c", "d", "e"], transaction))
```

And here are the test results of the function above:

```
In [7]: transaction = ["a", "b", "c", "d"]

        def check_belonging(itemset, transaction_items):
            belong = 0
            if len(itemset) > len(transaction_items):
                belong = 0
            elif(all(item in transaction_items for item in itemset)):
                belong = 1
            return belong

        print(check_belonging(["a"], transaction))
        print(check_belonging(["e"], transaction))
        print(check_belonging(["b", "c"], transaction))
        print(check_belonging(["a", "b", "c", "d"], transaction))
        print(check_belonging(["a", "b", "c", "d", "e"], transaction))

        1
        0
        1
        1
        0
```

Brute-force

We decided to first implement the brute-force algorithm, because it seemed less complex to develop and would help in developing the Apriori. And we also have created a function for it, that receives:

- **inventory**: a TSV file containing the list of items available on our store (as created above)
- **database**: a TSV file containing the transactions with items of our inventory (as created above)
- **min_support**: the minimum support in quantity (integer)
- **min_confidence**: the minimum confidence in the decimal fraction form

The function performs the brute-force and spits out the list of itemsets and their support and confidence values considering the parameters informed as a Pandas DataFrame.

```
def brute_force(inventory, database, min_support, min_confidence):
    inventory = pd.read_csv(inventory, sep="\t")
    inventory = list(inventory["item_description"])
    transactions = pd.read_csv(database, sep="\t")
    frequent_items = []
    num_transactions = len(transactions.index)

    ## Getting the support for each combination of items available on inventory
    for num_items in range(1, len(inventory)):
        itemset = generate_combinations(inventory, num_items)
        for each_combination in itemset:
            support = 0
            # Check for the presence of the item in the transaction and adds +1 to support if so
            for index, each_transaction in transactions.iterrows():
                support += check_belonging(each_combination,
                                           each_transaction["items"].split(","))
            # Add to our frequent items list if above the minimum support
            if support >= min_support:
                frequent_items.append({
                        "itemset": ','.join(each_combination),
                        "support": support,
                        "qty_items": len(each_combination)
                    }
                )
        ## Early-stop if there is no frequent items for the combinations of that size
        if not frequent_items or pd.DataFrame(frequent_items)["qty_items"].max() < num_items:
            break

    frequent_itemsets = pd.DataFrame(frequent_items)
    # Remove frequent itemsets with only one item
    frequent_itemsets = frequent_itemsets[frequent_itemsets["qty_items"] > 1]

    ## Creating association rules and getting the confidence
    association_rules = []
    for index, each_itemset in frequent_itemsets.iterrows():
        for each_item in each_itemset["itemset"].split(","):
            consequent = each_item
            antecedent = each_itemset["itemset"].split(",")
            antecedent.remove(consequent)
            confidence = 0
            # Check the combination on all transactions and add +1 to confidence if present
            for index, each_transaction in transactions.iterrows():
                confidence += check_belonging(antecedent,
                                              each_transaction["items"].split(","))
            # Add to association rules
            if each_itemset["support"] / confidence >= min_confidence:
                association_rules.append({
                        "antecedent": ",".join(antecedent),
                        "consequent": consequent,
                        "support": str(each_itemset["support"]) + "/" + str(num_transactions),
                        "support %": each_itemset["support"] / num_transactions,
                        "confidence": str(each_itemset["support"]) + "/" + str(confidence),
                        "confidence %": each_itemset["support"] / confidence
                    }
                )

    if not association_rules:
        print("No frequent itemset found for support =", min_support,
              "and confidence =", min_confidence, "in Brute Force algorithm")
        return

    return pd.DataFrame(association_rules).sort_values(by = ["antecedent", "consequent"])
```

Here is our test using the Database #1 with 2 as the minimum support (10%) and 0.5 as the minimum confidence (50%):

```
In [10]: df_brute_force = brute_force("inventory.tsv", "database1.tsv", min_support = 2, min_confidence = 0.5)
         df_brute_force
```

Out[10]:

	antecedent	consequent	support	support %	confidence	confidence %
1	Batteries	Orange Juice	3/20	0.15	3/6	0.500000
4	Rice	Toothpaste	2/20	0.10	2/2	1.000000
0	Sliced Ham	Orange Juice	2/20	0.10	2/3	0.666667
2	Toothpaste	Orange Juice	2/20	0.10	2/3	0.666667
3	Toothpaste	Rice	2/20	0.10	2/3	0.666667
5	Yogurt	Batteries	2/20	0.10	2/2	1.000000

Apriori

Apriori algorithm works almost in the same way as the brute-force, with the important difference that instead of using the inventory (available items) to generate the combinations of items, we use the apriori knowledge about most frequent items sold together, which means, the transactions themselves are used. Our algorithm will receive the following parameters:

- **database**: a TSV file containing the transactions with items of our inventory (as created above)
- **min_support**: the minimum support in quantity (integer)
- **min_confidence**: the minimum confidence in the decimal fraction form

and will output the same as our brute-force: list of itemsets and their support and confidence values as a Pandas DataFrame (for visualization purposes, the complete algorithm starts on the next page)

```python
def apriori(database, min_support, min_confidence):
    transactions = pd.read_csv(database, sep = "\t")
    frequent_items = []
    num_transactions = len(transactions.index)

    num_items = 1
    # Enter into an infinite loop to assess every possible combination
    while 1 == 1:
        for index, each_transaction in transactions.iterrows():
            itemset = generate_combinations(each_transaction["items"].split(","), num_items + 1)
            for each_combination in itemset:
                # Check if we have already calculated the support for frequent itemset
                if (not frequent_items or
                    pd.DataFrame(frequent_items)[
                        pd.DataFrame(frequent_items)["itemset"] == ','.join(each_combination)
                    ]["itemset"].count() == 0):
                    support = 0
            # Check for the presence of the item in the transaction and adds +1 to support if so
                    for index, each_transaction in transactions.iterrows():
                        support += check_belonging(
                            each_combination,
                            each_transaction["items"].split(","))
                    # Add to our frequent items list if above the minimum support
                    if support >= min_support:
                        frequent_items.append({
                                "itemset": ','.join(each_combination),
                                "support": support,
                                "qty_items": len(each_combination)
                            }
                        )
        num_items += 1
        ## Early-stop if there is no frequent items for the combinations of that size
        if not frequent_items or pd.DataFrame(frequent_items)["qty_items"].max() < num_items:
            break

    if not frequent_items:
        print("No frequent itemset found for support =", min_support, "in Apriori algorithm")
        return

    frequent_itemsets = pd.DataFrame(frequent_items)
    # Remove frequent itemsets with only one item
    frequent_itemsets = frequent_itemsets[frequent_itemsets["qty_items"] > 1]

    ## Creating association rules and getting the confidence
    association_rules = []
    for index, each_itemset in frequent_itemsets.iterrows():
        for each_item in each_itemset["itemset"].split(","):
            consequent = each_item
            antecedent = each_itemset["itemset"].split(",")
            antecedent.remove(consequent)
            confidence = 0
            # Check the combination on all transactions and add +1 to confidence if present
            for index, each_transaction in transactions.iterrows():
                confidence += check_belonging(antecedent, each_transaction["items"].split(","))
            # Add to association rules
            if each_itemset["support"] / confidence >= min_confidence:
                association_rules.append({
                        "antecedent": ",".join(antecedent),
                        "consequent": consequent,
                        "support": str(each_itemset["support"]) + "/" + str(num_transactions),
                        "support %": each_itemset["support"] / num_transactions,
                        "confidence": str(each_itemset["support"]) + "/" + str(confidence),
                        "confidence %": each_itemset["support"] / confidence
                    }
                )
    return pd.DataFrame(association_rules).sort_values(by = ["antecedent", "consequent"])
```

Here is our test using the Database #1 with 2 as the minimum support (10%) and 0.6 as the minimum confidence (60%)

```
In [12]: df_apriori = apriori("database1.tsv", 2, 0.6)
         df_apriori
```

Out[12]:

	antecedent	consequent	support	support %	confidence	confidence %
1	Rice	Toothpaste	2/20	0.1	2/2	1.000000
3	Sliced Ham	Orange Juice	2/20	0.1	2/3	0.666667
4	Toothpaste	Orange Juice	2/20	0.1	2/3	0.666667
0	Toothpaste	Rice	2/20	0.1	2/3	0.666667
2	Yogurt	Batteries	2/20	0.1	2/2	1.000000

We purposedly used a different confidence because we wanted to outer join the results of the two dataset to check if they are behaving as expected

```
In [13]: df_apriori.merge(df_brute_force, how = "outer", left_on=["antecedent", "consequent"], right_on=["antecedent", "consequent"],
                          suffixes=(' [Apriori]', ' [Brute Force]')).sort_values(by = ["antecedent", "consequent"])
```

Out[13]:

	antecedent	consequent	support [Apriori]	support % [Apriori]	confidence [Apriori]	confidence % [Apriori]	support [Brute Force]	support % [Brute Force]	confidence [Brute Force]	confidence % [Brute Force]
5	Batteries	Orange Juice	NaN	NaN	NaN	NaN	3/20	0.15	3/6	0.500000
0	Rice	Toothpaste	2/20	0.1	2/2	1.000000	2/20	0.10	2/2	1.000000
1	Sliced Ham	Orange Juice	2/20	0.1	2/3	0.666667	2/20	0.10	2/3	0.666667
2	Toothpaste	Orange Juice	2/20	0.1	2/3	0.666667	2/20	0.10	2/3	0.666667
3	Toothpaste	Rice	2/20	0.1	2/3	0.666667	2/20	0.10	2/3	0.666667
4	Yogurt	Batteries	2/20	0.1	2/2	1.000000	2/20	0.10	2/2	1.000000

We can see above that our two algorithms outputted the same frequent itemsets with the same support and confidence and that, as we raised the confidence level when executing Apriori, the itemset that didn't meet that threshold was removed from the final list.

As the final goal of the project is to compare the performance between both algorithms, we created a function to execute that comparison. This function receives:

- **inventory**: a TSV file containing the list of items available on our store (as created above)
- **database**: a TSV file containing the transactions with items of our inventory (as created above)
- **min_support**: the minimum support in quantity (integer)
- **min_confidence**: the minimum confidence in the decimal fraction form

The function prints the running time (in seconds) of each algorithm and, in the case we have association rules that meet the parameters, it returns the merged Pandas DataFrame (using outer join):

```
def compare_algorithms(inventory, database, min_support, min_confidence):
    import time
    start_time = time.time()
    df_apriori = apriori(database, min_support, min_confidence)
    apriori_time = time.time() - start_time
    start_time = time.time()
    df_brute_force = brute_force(inventory, database, min_support = min_support,
                                 min_confidence = min_confidence)
    brute_force_time = time.time() - start_time
    print(
        "Apriori time (s): ", round(apriori_time, 3),
        "\t\t\t\t",
        "Brute Force time (s): ", round(brute_force_time, 3), sep = ""
    )

    if df_apriori is not None:
        return df_apriori.merge(
            df_brute_force,
            how = "outer",
            left_on=["antecedent", "consequent"],
            right_on=["antecedent", "consequent"],
            suffixes=(' [Apriori]', ' [Brute Force]')
        ).sort_values(by = ["antecedent", "consequent"])
```

We executed the comparison for the **Database #1** as a way of testing the function and obtaining the difference in performance:

```
In [15]: compare_algorithms("inventory.tsv", "database1.tsv", min_support = 2, min_confidence = 0.5)
```

```
Apriori time (s): 0.619                 Brute Force time (s): 9.167
```

Out[15]:

	antecedent	consequent	support [Apriori]	support % [Apriori]	confidence [Apriori]	confidence % [Apriori]	support [Brute Force]	support % [Brute Force]	confidence [Brute Force]	confidence % [Brute Force]
0	Batteries	Orange Juice	3/20	0.15	3/6	0.500000	3/20	0.15	3/6	0.500000
1	Rice	Toothpaste	2/20	0.10	2/2	1.000000	2/20	0.10	2/2	1.000000
2	Sliced Ham	Orange Juice	2/20	0.10	2/3	0.666667	2/20	0.10	2/3	0.666667
3	Toothpaste	Orange Juice	2/20	0.10	2/3	0.666667	2/20	0.10	2/3	0.666667
4	Toothpaste	Rice	2/20	0.10	2/3	0.666667	2/20	0.10	2/3	0.666667
5	Yogurt	Batteries	2/20	0.10	2/2	1.000000	2/20	0.10	2/2	1.000000

As we already executed for database1 (above) and now we are going to execute for the rest of the databases, using different parameters for support and confidence, starting with **Database #2**:

```
In [16]: compare_algorithms("inventory.tsv", "database2.tsv", min_support = 3, min_confidence = 0.3)
```

```
Apriori time (s): 3.151                    Brute Force time (s): 9.026
```

Out[16]:

	antecedent	consequent	support [Apriori]	support % [Apriori]	confidence [Apriori]	confidence % [Apriori]	support [Brute Force]	support % [Brute Force]	confidence [Brute Force]	confidence % [Brute Force]
0	Apple Juice	Bread	3/20	0.15	3/4	0.750	3/20	0.15	3/4	0.750
1	Banana	Butter	3/20	0.15	3/5	0.600	3/20	0.15	3/5	0.600
2	Banana	Classic Coke	3/20	0.15	3/5	0.600	3/20	0.15	3/5	0.600
3	Bread	Apple Juice	3/20	0.15	3/6	0.500	3/20	0.15	3/6	0.500
4	Bread	Grape	3/20	0.15	3/6	0.500	3/20	0.15	3/6	0.500
5	Bread	Yogurt	3/20	0.15	3/6	0.500	3/20	0.15	3/6	0.500
6	Butter	Banana	3/20	0.15	3/5	0.600	3/20	0.15	3/5	0.600
7	Classic Coke	Banana	3/20	0.15	3/3	1.000	3/20	0.15	3/3	1.000
8	Coffee	Grape	3/20	0.15	3/5	0.600	3/20	0.15	3/5	0.600
9	Coffee	Orange Juice	3/20	0.15	3/5	0.600	3/20	0.15	3/5	0.600
10	Grape	Bread	3/20	0.15	3/8	0.375	3/20	0.15	3/8	0.375
11	Grape	Coffee	3/20	0.15	3/8	0.375	3/20	0.15	3/8	0.375
12	Grape	Lemon	3/20	0.15	3/8	0.375	3/20	0.15	3/8	0.375
13	Grape	Popcorn	3/20	0.15	3/8	0.375	3/20	0.15	3/8	0.375
14	Italian Sub	Pear	3/20	0.15	3/4	0.750	3/20	0.15	3/4	0.750
15	Lemon	Grape	3/20	0.15	3/5	0.600	3/20	0.15	3/5	0.600
16	Orange Juice	Coffee	3/20	0.15	3/6	0.500	3/20	0.15	3/6	0.500
17	Pear	Italian Sub	3/20	0.15	3/5	0.600	3/20	0.15	3/5	0.600
18	Popcorn	Grape	3/20	0.15	3/5	0.600	3/20	0.15	3/5	0.600
19	Yogurt	Bread	3/20	0.15	3/4	0.750	3/20	0.15	3/4	0.750

Database #3 (for this database, we have limited the items at 3 on purpose just to see how the algorithm would behave when not finding meaningful associations):

```
In [19]: compare_algorithms("inventory.tsv", "database3.tsv", min_support = 2, min_confidence = 0.2)
```

```
No frequent itemset found for support = 2 in Apriori algorithm
No frequent itemset found for support = 2 and confidence = 0.2 in Brute Force algorithm
Apriori time (s): 0.045                    Brute Force time (s): 0.92
```

Database #4:

```
In [20]: compare_algorithms("inventory.tsv", "database4.tsv", min_support = 3, min_confidence = 0.9)
```

```
Apriori time (s): 5.722                    Brute Force time (s): 342.543
```

Out[20]:

	antecedent	consequent	support [Apriori]	support % [Apriori]	confidence [Apriori]	confidence % [Apriori]	support [Brute Force]	support % [Brute Force]	confidence [Brute Force]	confidence % [Brute Force]
0	Apple,Hard Salami	Sprite	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
1	Apple,Hard Salami	Tylenol	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
2	Apple,Hard Salami,Sprite	Tylenol	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
3	Apple,Hard Salami,Tylenol	Sprite	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
4	Apple,Sprite	Hard Salami	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
5	Apple,Sprite	Tylenol	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
6	Apple,Sprite,Tylenol	Hard Salami	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
7	Apple,Tylenol	Hard Salami	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
8	Apple,Tylenol	Sprite	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
9	Hard Salami	Apple	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
10	Hard Salami	Sprite	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
11	Hard Salami	Tylenol	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
12	Hard Salami,Sprite	Apple	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
13	Hard Salami,Sprite	Tylenol	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
14	Hard Salami,Sprite,Tylenol	Apple	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
15	Hard Salami,Tylenol	Apple	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
16	Hard Salami,Tylenol	Sprite	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
17	Italian Sub	Tylenol	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
18	Sprite,Tylenol	Apple	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0
19	Sprite,Tylenol	Hard Salami	3/20	0.15	3/3	1.0	3/20	0.15	3/3	1.0

Database #5:

```
In [21]: compare_algorithms("inventory.tsv", "database5.tsv", min_support = 3, min_confidence = 0.65)
```

```
Apriori time (s): 9.499                    Brute Force time (s): 61.41
```

Out[21]:

	antecedent	consequent	support [Apriori]	support % [Apriori]	confidence [Apriori]	confidence % [Apriori]	support [Brute Force]	support % [Brute Force]	confidence [Brute Force]	confidence % [Brute Force]
0	Apple	Provolone Cheese	3/20	0.15	3/4	0.750000	3/20	0.15	3/4	0.750000
1	Apple	Sliced Ham	3/20	0.15	3/4	0.750000	3/20	0.15	3/4	0.750000
2	Apple	Yogurt	3/20	0.15	3/4	0.750000	3/20	0.15	3/4	0.750000
3	Eggs	Hot Pocket	3/20	0.15	3/4	0.750000	3/20	0.15	3/4	0.750000
4	Fanta,Hot Pocket	Toothpaste	3/20	0.15	3/3	1.000000	3/20	0.15	3/3	1.000000
5	Fanta,Toothpaste	Hot Pocket	3/20	0.15	3/3	1.000000	3/20	0.15	3/3	1.000000
6	Hot Pocket,Toothpaste	Fanta	3/20	0.15	3/4	0.750000	3/20	0.15	3/4	0.750000
7	Hungry Man	Rice	3/20	0.15	3/3	1.000000	3/20	0.15	3/3	1.000000
8	Orange Juice	Popcorn	3/20	0.15	3/3	1.000000	3/20	0.15	3/3	1.000000
9	Popcorn	Orange Juice	3/20	0.15	3/4	0.750000	3/20	0.15	3/4	0.750000
10	Popcorn	Provolone Cheese	3/20	0.15	3/4	0.750000	3/20	0.15	3/4	0.750000
11	Provolone Cheese	Batteries	4/20	0.20	4/6	0.666667	4/20	0.20	4/6	0.666667
12	Toothpaste	Hot Pocket	4/20	0.20	4/6	0.666667	4/20	0.20	4/6	0.666667
13	Tylenol	Provolone Cheese	3/20	0.15	3/3	1.000000	3/20	0.15	3/3	1.000000

Conclusion

As we can see in the statistics above, the brute-force method is more time-consuming (and we can use execution time as a proxy for other resources) than Apriori. The minimum difference of performance (obtained in Database #2) was in the order of three times more execution time for the brute-force method when compared with the Apriori, using the same database of transactions and parameters.

Even when there are no meaningful associations (as in our Database #3), the time taken by brute-force was higher than Apriori.

Another thing we could notice is that while Apriori execution time somehow grows linearly, according to the number of items that generated each frequent itemset, the brute-force grows exponentially, even using the same superset (inventory).

The Jupyter Notebook containing the codes, along with the databases, can be also found in GitHub: https://github.com/wellingtoncunha/data_mining/tree/master/mid_term_project.