Data Mining – Midterm Project

Wellington Cunha

NJIT ID: 31548454

NJIT UC ID: wc44

[wc44@njit.edu](mailto: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:

Calendar

Description automatically 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:

Graphical user interface, text, application

Description automatically generated

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:

Graphical user interface, text, application

Description automatically generated

**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**():**  item\_set **=** each\_itemset**[**"itemset"**].**split**(**","**)**  **for** num\_consequent **in** **range(**1**,** **len(**item\_set**)):**  **for** antecedent **in** generate\_combinations**(**item\_set**,** num\_consequent**):**  consequent **=** **[**e **for** e **in** item\_set **if** e **not** **in** antecedent**]**  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"**:** ","**.**join**(**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%):

Graphical user interface

Description automatically generated with medium confidence

**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)

|  |
| --- |
| **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**,**  "and confidence ="**,** min\_confidence**,** "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**():**  item\_set **=** each\_itemset**[**"itemset"**].**split**(**","**)**  **for** num\_consequent **in** **range(**1**,** **len(**item\_set**)):**  **for** antecedent **in** generate\_combinations**(**item\_set**,** num\_consequent**):**  consequent **=** **[**e **for** e **in** item\_set **if** e **not** **in** antecedent**]**  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"**:** ","**.**join**(**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 Apriori algorithmm"**)**  **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 0.6 as the minimum confidence (60%)

Graphical user interface, text, application

Description automatically generated

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

A screenshot of a computer

Description automatically generated with medium confidence

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.

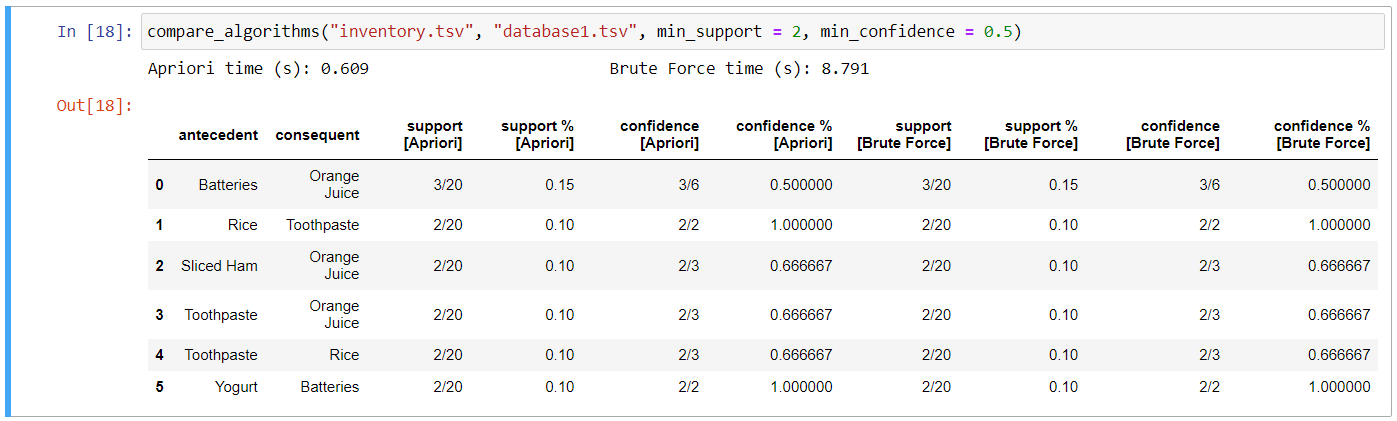
As the final goal of the project is to compare the performance between both algorithms, we created a function to execute that comparison. This function receives:

* **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:

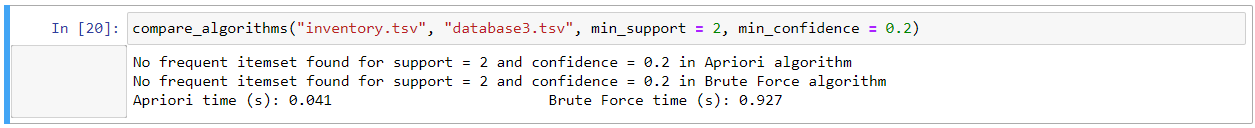
****

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**:

Graphical user interface, application

Description automatically generated with medium confidence

**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):



**Database #4**:

Graphical user interface, application, table

Description automatically generated

**Database #5**:

Graphical user interface, application

Description automatically generated

**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 almost 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>.