School of Computing and Digital Technology

5/9/2020

Data Mining

Case Study Report

Aleksandra Petkova - S6222041
Nour Aldin Al Mubarak - P4302185

Contents

Introduction

This project was developed for the Data Mining module at Teesside University with the aim to demonstrate and evaluate the use of popular computational techniques for data mining. The project contains implementations of Association Rules Mining, Collaborative Filtering and variety of datasets to test on.

Datasets

1. Online Retail: time series, transactional data – 8 attributes, 541K records
   Each row represents a whole sale purchase of an item along with details about the purchase such as: the invoice ID, stock code, item description, quantity purchased, price per unit, the ID and country of the customer who made the purchase.
2. Groceries: customer recite data – 9K records
   Each row represents a customer recite (list of the items bought) with no details about the purchase but the names of the items.
3. MovieLens: movie rating data – 100K records
   Each row represents one review by one customer.

Techniques

Association Rules Mining, Apriori (Core and Library) – Aleksandra Petkova, Nour Aldin Al Mubarak.

Important Note

Aleksandra and Nour built 2 separate Apriori algorithms – from scratch/core python (Aleksandra) and with a library (Nour). The version from scratch was running a lot slower than the version with the library so much so that 50k records from online retail took 20 minutes to process. Considering the dataset has 500k records that dataset was dropped for the version from scratch. Despite that the library version of Apriori runs on both algorithms with processing time to complete for 500k records taking a bit under 30 min on the same machine that the other version of the algorithm was tested. The Apriori report section was fully written by Aleksandra along with the RegX pre-processing code for the Online Retail dataset.

Brief

The **Apriori Algorithm** is one of many algorithms for **Association Rules Mining**, other popular examples are the **Eclat Algorithm**, **OPUS Search** and **FP-growth algorithm**. We chose to work with **Apriori** because it is a standard and simple method for association rules mining and is also the one, we are most familiar with.

Datasets Usage

1. *Online Retail* – Used for both **the algorithm written <u>from scratch</u>** and **the algorithm written with the <u>MLextend Library</u>**. It is the **largest dataset** from the available and has both **missing** and **erroneous data**, those are the key reasons it was first chosen to present the **Apriori**

Association Rules Mining algorithm. Despite that the amount of data was too much for the **Association Rules** algorithm written from scratch so we had to swap to the **Groceries dataset** instead. The library version still has the ability to run on this algorithm. All the **pre-processing** data was done on it because there is no **pre-processing** to do for the **Groceries dataset**.

2. *Groceries* – Much smaller dataset, no cleaning required, very simple to work with when building an **Association Rules Mining** algorithm. That is why this is the dataset used to build and test the first prototype of the algorithm written from scratch. Around beginning of April, we reverted back to using that dataset and stuck with it until the end.

3. *Simple Dataset* – Is a random **sample** from the *Groceries* dataset used to **validate** each step of the algorithm.

Pre-processing Done and Experiments - Online Retail Dataset:

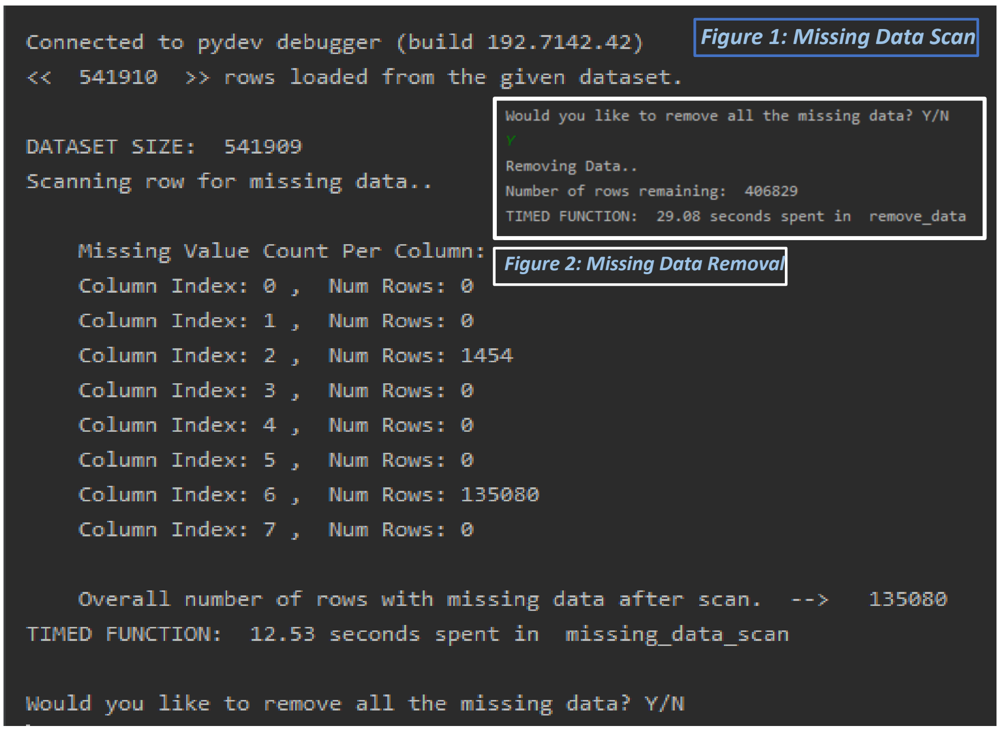


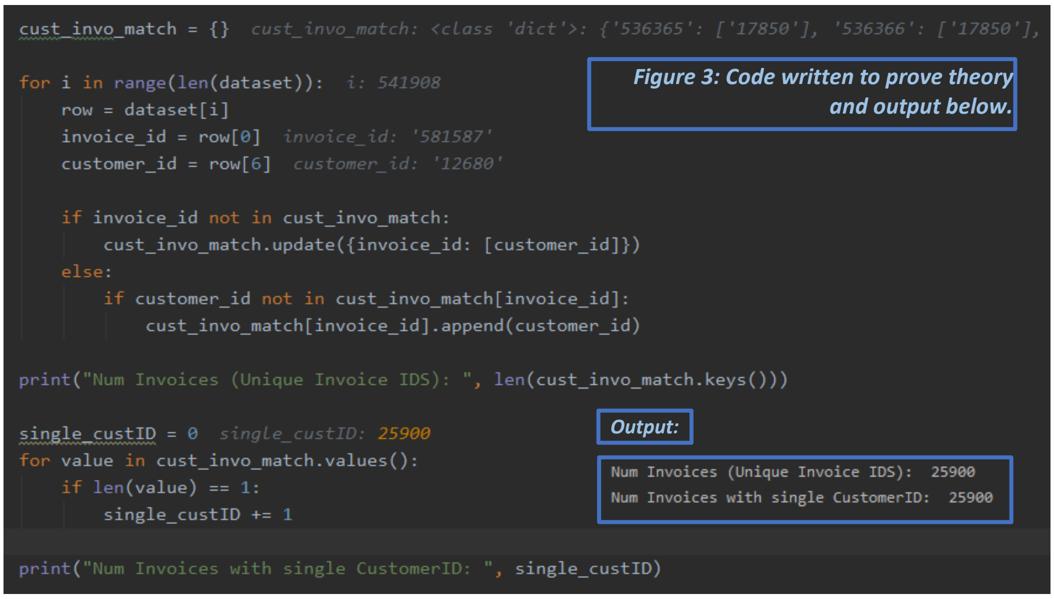
Figure 1: Missing Data Scan

```
Connected to pydev debugger (build 192.7142.42)
<<  541910  >> rows loaded from the given dataset.

DATASET SIZE:  541909
Scanning row for missing data..

    Missing Value Count Per Column:
    Column Index: 0 ,  Num Rows: 0
    Column Index: 1 ,  Num Rows: 0
    Column Index: 2 ,  Num Rows: 1454
    Column Index: 3 ,  Num Rows: 0
    Column Index: 4 ,  Num Rows: 0
    Column Index: 5 ,  Num Rows: 0
    Column Index: 6 ,  Num Rows: 135080
    Column Index: 7 ,  Num Rows: 0

    Overall number of rows with missing data after scan.  -->   135080
TIMED FUNCTION:  12.53 seconds spent in  missing_data_scan

Would you like to remove all the missing data? Y/N
```

```
Would you like to remove all the missing data? Y/N
Y
Removing Data..
Number of rows remaining:  406829
TIMED FUNCTION:  29.08 seconds spent in  remove_data
```

Figure 2: Missing Data Removal

1. **Missing Data Scan**: As you can see from Figure 1, there are only 2 columns that contain missing data – **Column 2 (Description)** and **Column 6 (CustomerID)**. The rows missing in **Column 6** represent about a **5th** of the whole dataset so ideally it would be good to fill them. According to the dataset structure (more information in the Datasets Section) the invoice is always sent to the same customer, i.e. the **InvoiceID** will only ever be linked to one **CustomerID**. This is confirmed by the investigation on this and the results from the mining shown on Figure 3.



```
cust_invo_match = {}  cust_invo_match: <class 'dict'>: {'536365': ['17850'], '536366': ['17850'],

for i in range(len(dataset)):  i: 541908
    row = dataset[i]
    invoice_id = row[0]  invoice_id: '581587'
    customer_id = row[6]  customer_id: '12680'

    if invoice_id not in cust_invo_match:
        cust_invo_match.update({invoice_id: [customer_id]})
    else:
        if customer_id not in cust_invo_match[invoice_id]:
            cust_invo_match[invoice_id].append(customer_id)

print("Num Invoices (Unique Invoice IDS): ", len(cust_invo_match.keys()))

single_custID = 0  single_custID: 25900
for value in cust_invo_match.values():
    if len(value) == 1:
        single_custID += 1

print("Num Invoices with single CustomerID: ", single_custID)
```

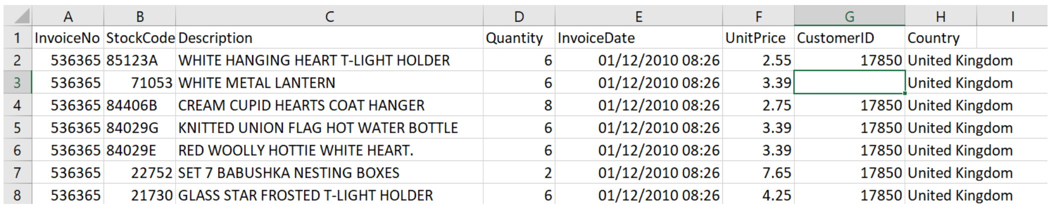
Figure 3: Code written to prove theory and output below.

Output:

```
Num Invoices (Unique Invoice IDS):  25900
Num Invoices with single CustomerID:  25900
```

If there was a case where one or more **CustomerIDs** existed in an Invoice with missing **CustomerIDs** that would mean the **cust_invo_match dictionary** would store the existing **CustomerID** and the empty string - **“”**. In the example shown below (Figure 4) I manually removed one of the customer IDs from an invoice to test how the scan will react to this.

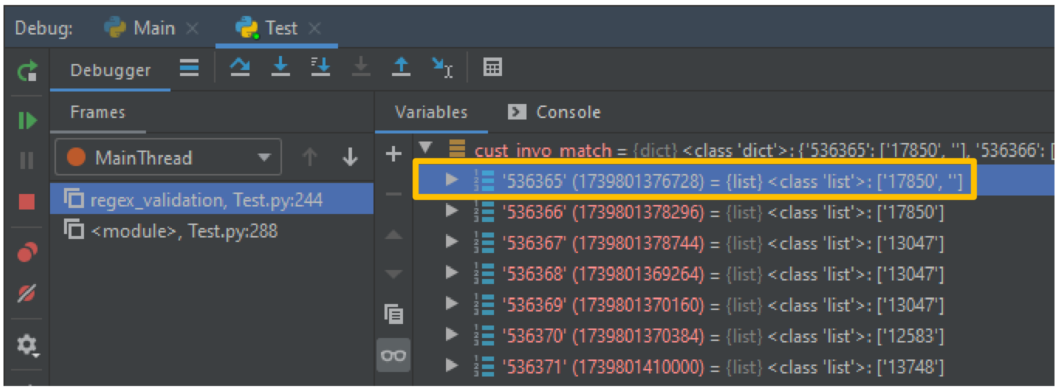
Figure 4, Experiment: Manually removing a CustomerID.



	A	B	C	D	E	F	G	H	I
1	InvoiceNo	StockCode	Description	Quantity	InvoiceDate	UnitPrice	CustomerID	Country	
2	536365	85123A	WHITE HANGING HEART T-LIGHT HOLDER	6	01/12/2010 08:26	2.55	17850	United Kingdom	
3	536365	71053	WHITE METAL LANTERN	6	01/12/2010 08:26	3.39		United Kingdom	
4	536365	84406B	CREAM CUPID HEARTS COAT HANGER	8	01/12/2010 08:26	2.75	17850	United Kingdom	
5	536365	84029G	KNITTED UNION FLAG HOT WATER BOTTLE	6	01/12/2010 08:26	3.39	17850	United Kingdom	
6	536365	84029E	RED WOOLLY HOTTIE WHITE HEART.	6	01/12/2010 08:26	3.39	17850	United Kingdom	
7	536365	22752	SET 7 BABUSHKA NESTING BOXES	2	01/12/2010 08:26	7.65	17850	United Kingdom	
8	536365	21730	GLASS STAR FROSTED T-LIGHT HOLDER	6	01/12/2010 08:26	4.25	17850	United Kingdom	

The empty string was indeed added as an option for **CustomerID** for the current **InvoiceID** (Figure 5). The fact that every invoice in the dataset only has one option for **CustomerID**, as shown on Figure 3, that means that there are no cases where at least one **CustomerID** is present on an invoice with one or more missing **Customer IDs**.

Figure 5: Experiment results after removing a customer ID.



```
regex_validation, Test.py:244
<module>, Test.py:288
```

```
cust_invo_match = {dict} <class 'dict'>: {'536365': ['17850', ''], '536366': [
  '536365' (1739801376728) = {list} <class 'list'>: ['17850', '']
  '536366' (1739801378296) = {list} <class 'list'>: ['17850']
  '536367' (1739801378744) = {list} <class 'list'>: ['13047']
  '536368' (1739801369264) = {list} <class 'list'>: ['13047']
  '536369' (1739801370160) = {list} <class 'list'>: ['13047']
  '536370' (1739801370384) = {list} <class 'list'>: ['12583']
  '536371' (1739801410000) = {list} <class 'list'>: ['13748']
```

This disproves the idea of filling the data when at least one **CustomerID** exists in an invoice that contains missing **CustomerIDs**. I could not find any other ways to fill the missing data so I removed all of it including the records with missing data in column 2.

2. **Regular Expression Validation System:** Catches any data that does not fit the set rules and displays insights to the user in the form of console output. **Example Regex**: **UnitPrice (Column 5)** has the regular expression **"^£{0,1}([0-9]+\.[0-9]*[1-9]+)|([1-9][0-9]*)"** which states, the data in the column may have 0 or 1 of £ symbol at the start of the column data – **"^£{0,1}"**, followed by either a floating point number that is > 0.0 – **"[0-9]+\.[0-9]*[1-9]+"**, or a whole number that is > 0 – **"[1-9][0-9]*".** If the data in the **column fails** this **validation**, it will be recorded into the **compromised_rows dictionary**, which the user has the option to inspect right after the full scan is complete. The output after the scan is shown in the screenshot on Figure 6. If the user types Y or y, they will be asked to **select a column** by entering the **column ID**. As you can see, I selected **Column 5** to follow up on the example given. The regex caught one issue, with 40 occurrences/rows, which is the unit price being 0. Depending on the owner of the dataset this may be intentional or it could be something to remove, but to make a choice I would have to look at the full row containing this data.

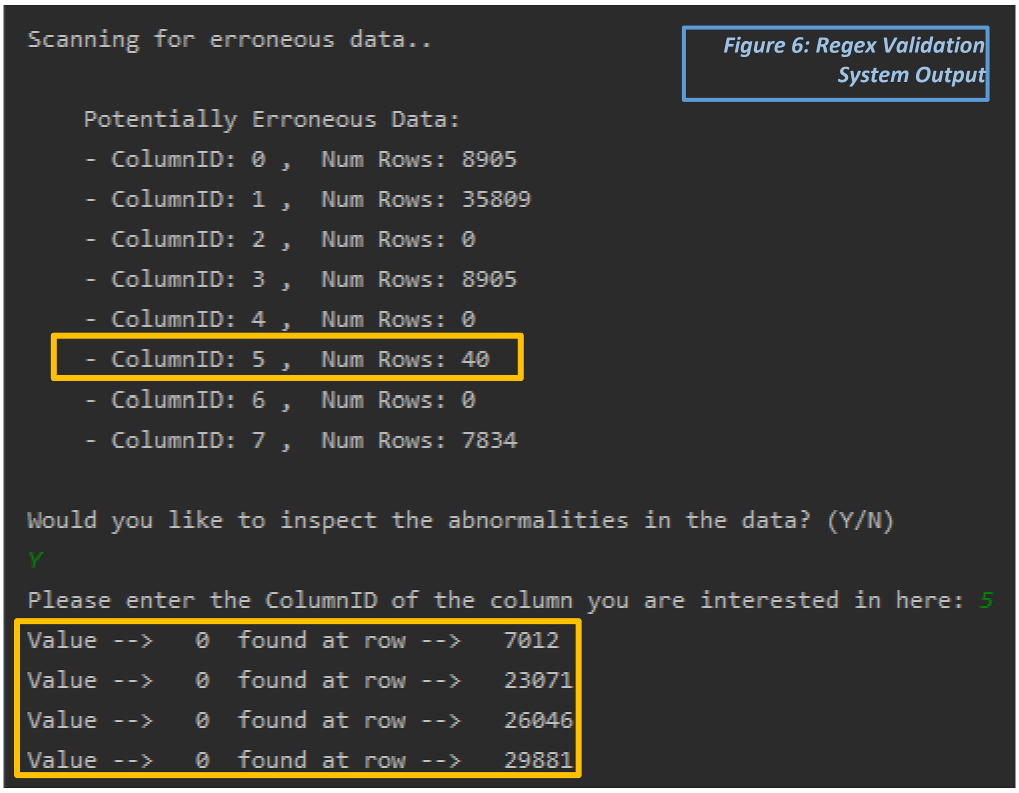
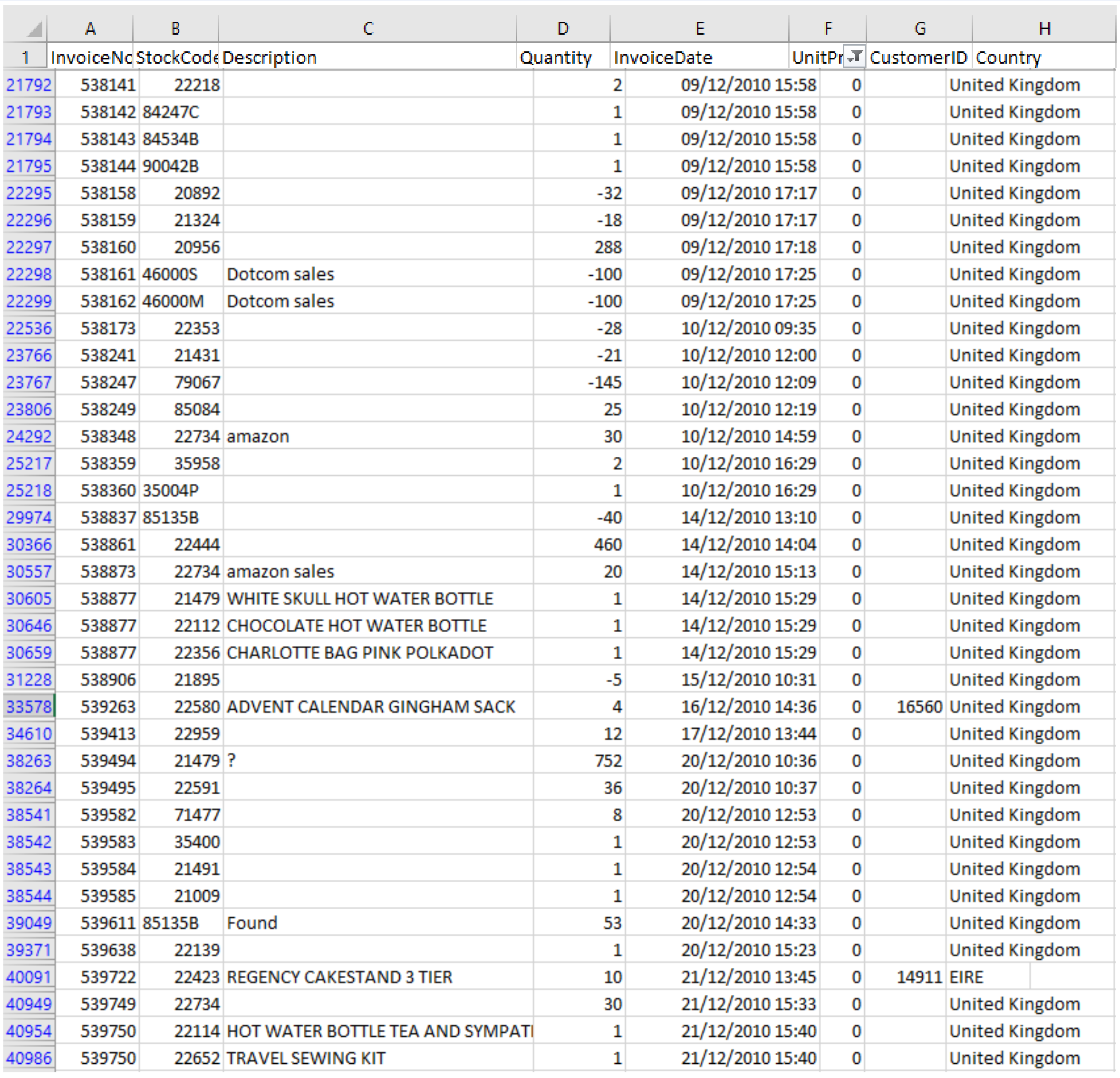


Figure 6: Regex Validation System Output

```
Scanning for erroneous data..

    Potentially Erroneous Data:
    - ColumnID: 0 ,  Num Rows: 8905
    - ColumnID: 1 ,  Num Rows: 35809
    - ColumnID: 2 ,  Num Rows: 0
    - ColumnID: 3 ,  Num Rows: 8905
    - ColumnID: 4 ,  Num Rows: 0
    - ColumnID: 5 ,  Num Rows: 40
    - ColumnID: 6 ,  Num Rows: 0
    - ColumnID: 7 ,  Num Rows: 7834

Would you like to inspect the abnormalities in the data? (Y/N)
Y
Please enter the ColumnID of the column you are interested in here: 5
Value -->   0  found at row -->   7012
Value -->   0  found at row -->   23071
Value -->   0  found at row -->   26046
Value -->   0  found at row -->   29881
```

Below (Figure 7) you can find a screenshot of what I found by manually inspecting the compromised rows. Here you can see that there are generally all kinds of issues with these rows, like empty cell for **CustomerID**, negative value for quantity, missing item description, etc. At this point we found out that the association Rules written from scratch is impossibly slow at dealing with frequency counting of itemsets larger than 1 element per set (on **50K rows** has **900k combinations** of **2-itemsets** and takes **52 minutes** to process). So, we decided to swap to the **Groceries dataset**, which has very feasible processing time. The library version the **Groceries dataset** does not require any cleaning though which is why we kept the cleaning for **Online Retail**.

Figure 7: Rows with UnitPrice 0



	A	B	C	D	E	F	G	H
1	InvoiceNc	StockCode	Description	Quantity	InvoiceDate	UnitPr	CustomerID	Country
21792	538141	22218		2	09/12/2010 15:58	0		United Kingdom
21793	538142	84247C		1	09/12/2010 15:58	0		United Kingdom
21794	538143	84534B		1	09/12/2010 15:58	0		United Kingdom
21795	538144	90042B		1	09/12/2010 15:58	0		United Kingdom
22295	538158	20892		-32	09/12/2010 17:17	0		United Kingdom
22296	538159	21324		-18	09/12/2010 17:17	0		United Kingdom
22297	538160	20956		288	09/12/2010 17:18	0		United Kingdom
22298	538161	46000S	Dotcom sales	-100	09/12/2010 17:25	0		United Kingdom
22299	538162	46000M	Dotcom sales	-100	09/12/2010 17:25	0		United Kingdom
22536	538173	22353		-28	10/12/2010 09:35	0		United Kingdom
23766	538241	21431		-21	10/12/2010 12:00	0		United Kingdom
23767	538247	79067		-145	10/12/2010 12:09	0		United Kingdom
23806	538249	85084		25	10/12/2010 12:19	0		United Kingdom
24292	538348	22734	amazon	30	10/12/2010 14:59	0		United Kingdom
25217	538359	35958		2	10/12/2010 16:29	0		United Kingdom
25218	538360	35004P		1	10/12/2010 16:29	0		United Kingdom
29974	538837	85135B		-40	14/12/2010 13:10	0		United Kingdom
30366	538861	22444		460	14/12/2010 14:04	0		United Kingdom
30557	538873	22734	amazon sales	20	14/12/2010 15:13	0		United Kingdom
30605	538877	21479	WHITE SKULL HOT WATER BOTTLE	1	14/12/2010 15:29	0		United Kingdom
30646	538877	22112	CHOCOLATE HOT WATER BOTTLE	1	14/12/2010 15:29	0		United Kingdom
30659	538877	22356	CHARLOTTE BAG PINK POLKADOT	1	14/12/2010 15:29	0		United Kingdom
31228	538906	21895		-5	15/12/2010 10:31	0		United Kingdom
33578	539263	22580	ADVENT CALENDAR GINGHAM SACK	4	16/12/2010 14:36	0	16560	United Kingdom
34610	539413	22959		12	17/12/2010 13:44	0		United Kingdom
38263	539494	21479	?	752	20/12/2010 10:36	0		United Kingdom
38264	539495	22591		36	20/12/2010 10:37	0		United Kingdom
38541	539582	71477		8	20/12/2010 12:53	0		United Kingdom
38542	539583	35400		1	20/12/2010 12:53	0		United Kingdom
38543	539584	21491		1	20/12/2010 12:54	0		United Kingdom
38544	539585	21009		1	20/12/2010 12:54	0		United Kingdom
39049	539611	85135B	Found	53	20/12/2010 14:33	0		United Kingdom
39371	539638	22139		1	20/12/2010 15:23	0		United Kingdom
40091	539722	22423	REGENCY CAKESTAND 3 TIER	10	21/12/2010 13:45	0	14911	EIRE
40949	539749	22734		30	21/12/2010 15:33	0		United Kingdom
40954	539750	22114	HOT WATER BOTTLE TEA AND SYMPATI	1	21/12/2010 15:40	0		United Kingdom
40986	539750	22652	TRAVEL SEWING KIT	1	21/12/2010 15:40	0		United Kingdom

Algorithm (*Source code can be found in AssociationRulesCore.py*)

1. Find all **unique items** by transforming the loaded dataset from 2D to 1D and applying a set cast on the whole dataset to remove any repetitions. The unique items are then sorted to allow for easier management of the data.
2. Create an item to number **map/dictionary**, with the itemset as the key and the item ID as the value to **improve performance** as string comparisons are expensive.

3. **Generate mapped data**

   3.1 Reformat the **unique items** using the **unique items map** so that each **1-itemsets** is represented by its **ID**.

   3.2 Reformat the **dataset** using the **unique items map** so that the algorithm is only working with item **IDs**.

4. Count the **frequency** of each unique item.

5. Calculate the **minimum support** by multiplying the **minimum relative support** by the highest **frequency**. (**Example**: if the highest frequency is 6 and the relative support is 0.5 (50%) the minimum support/frequency will be 3.)

6. Apply the **Apriori property** to filter out the **infrequent** 1-itemsets.

7. Append the survived 1-itemsets dictionary to the complete_survival_subsets_list.

8. Start main Association Rules Loop and keep running until there are no more itemsets left to work with.
   8.1. Generate a k+1 itemsets list from the survived k-itemsets dictionary.

   8.1.1 Sets contain only unique values so {0, 0}, {1,1}, {2,2}, etc should not exist! Also, multiple variations of the same subset/itemset should not be checked at this stage because they may not and most likely will not be frequent. Example: if {0,1} is checked {1,0} should not be checked unless for sure {0,1} is frequent. Then we want to see which item implies the other so we will try each variation. Below there are tables of the acceptable 2, 3, 4-itemsets that implement the rules mentioned. You can see that there is a clear pattern which was used to optimise the process for generating k+1 itemsets.

   based on a much smaller dataset (**simple_dataset.csv**) that is also available in the "**Data Repository**" folder. The example tables do not show the Apriori property, they simply visualise the outputs assuming all subsets survive.

*Table 1: **All possible set combinations 1-itemset → 2-itemset***

	{0}	{1}	{2}	{3}	{4}
{0}	X	X	X	X	X
{1}	{0, 1}	X	X	X	X
{2}	{0, 2}	{1, 2}	X	X	X
{3}	{0, 3}	{1, 3}	{2, 3}	X	X
{4}	{0, 4}	{1, 4}	{2, 4}	{3, 4}	X

*Table 2: **All possible set combinations 2-itemset → 3-itemset***

	{0}	{1}	{2}	{3}	{4}
{0, 1}	X	X	X	X	X
{0, 2}	X	X	X	X	X
{0, 3}	X	X	X	X	X
{0, 4}	X	X	X	X	X
{1, 2}	{0,1,2}	X	X	X	X
{1, 3}	{0,1,3}	X	X	X	X
{1, 4}	{0,1,4}	X	X	X	X
{2, 3}	{0,2,3}	{1,2,3}	X	X	X
{2, 4}	{0,2,4}	{1,2,4}	X	X	X
{3, 4}	{0,3,4}	{1,3,4}	{2,3,4}	X	X

*Table 3: **All possible set combinations 3-itemset → 4-itemset***

	{0}	{1}	{2}	{3}	{4}
{0,1,2}	X	X	X	X	X
{0,1,3}	X	X	X	X	X
{0,1,4}	X	X	X	X	X
{0,2,3}	X	X	X	X	X
{0,2,4}	X	X	X	X	X
{0,3,4}	X	X	X	X	X
{1,2,3}	{0,1,2,3}	X	X	X	X
{1,2,4}	{0,1,2,4}	X	X	X	X
{1,3,4}	{0,1,3,4}	X	X	X	X
{2,3,4}	{0,2,3,4}	{1,2,3,4}	X	X	X

8.2. Count the **frequency** of each itemset.
8.3. Apply the **Apriori property** to filter out the **infrequent** current k-itemsets.

9. Generate Rules out of the frequent itemsets.
   9.1. Ignore the 1-itemsets as no rules can be generated from 1 item.
   9.2. Split the itemset into two parts (X and Y).
   9.3. Calculate **confidence** of the rule **X implies Y or Y implies X**.
   9.4. If it is below the **minimum confidence** given, discard the rule.

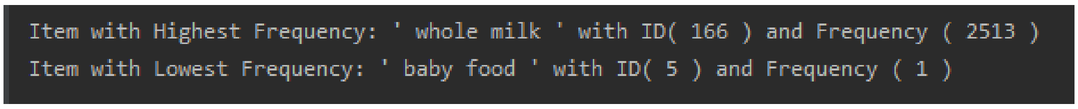
10. Display rules
    10.1. Open the file where the rules will be written - **association_rules.txt** (the "with" keyword was used to make sure Python's garbage collector cleans the file object)
    10.2. Go through every rule in the rules dictionary
        10.2.1. Extract the X and Y subsets, their frequencies/supports and the confidence of the rule.
        10.2.2. Construct a string out of that data to display the results.
        Example string: {pip fruit, root vegetables}, sup(153), rel sup(2%) ---> {whole milk}, sup(2513), rel sup(26%) - conf(58%)

11. Count the number of rules and itemsets and print that to the console.

Questions

1. How many **unique items** are there? - **169 unique items** found.
2. What is the **frequency** of each **unique item**?
   - Please refer to **frequent_itemsets.txt** file, located in the **outputs folder**.

3. Which items have the **highest frequency** and which one has the **lowest**?



```
Item with Highest Frequency: ' whole milk ' with ID( 166 ) and Frequency ( 2513 )
Item with Lowest Frequency: ' baby food ' with ID( 5 ) and Frequency ( 1 )
```

4. Which items are very **often bought together**? (Association rules with **relative support higher than 50%**)
   - Please refer to **association_rules.txt** file, located in the outputs folder.
5. Are there any **associations** in the dataset/s with **high confidence**? (**Rule confidence > 50%**)
   - Please refer to **Experiments and Evaluation** tables.
6. How many association rules have **high confidence**? (**Rule confidence > 50%**)
   - Please refer to **Experiments and Evaluation** tables.

Experiments and Evaluation *(What experiments can be done to prove it works)*

1. Display results with different support values, **highest and lowest support are calculated based of the itemsets that have survived the minimum support cap**. **All changes made from previous experiment are marked in red!**

Table 4, Exp1: Results Comparison of the algorithm - from scratch (Core Python) and with library (MLextend)

Parameter	Result
Minimum Support	63 occurrences
Lowest Support	77 occurrences
Highest Support	2513 occurrences
Minimum Relative Support	2.5%
Minimum Confidence	50%
Lift	N/A
Number of Survived Itemsets	688, with sup > 63
Number of Rules	52 rules

Table 5, Exp2: Changed Minimum relative support to 10%, with minimum rule confidence still at 50%.

Parameter	Result
Minimum Support	**251 occurrences**
Lowest Support	**736 occurrences**
Highest Support	2513 occurrences
Minimum Relative Support	**10%**
Minimum Confidence	50%
Lift	N/A
Number of Survived Itemsets	**86, with sup > 251**
Number of Rules	**0 rules**

Table 6, Exp3: The only change is to Minimum Rule Confidence, which was brought down from 50% to 10%

Parameter	Result
Minimum Support	251 occurrences
Lowest Support	736 occurrences
Highest Support	2513 occurrences
Minimum Relative Support	10%
Minimum Confidence	**10%**
Lift	N/A
Number of Survived Itemsets	86, with sup > 251
Number of Rules	**66 rules**

Table 7, Exp4: Changed both Minimum relative support and minimum confidence to 5% and 50%.

Parameter	Result
Minimum Support	**126 occurrences**
Lowest Support	**228 occurrences**
Highest Support	2513 occurrences
Minimum Relative Support	**5%**
Minimum Confidence	**50%**
Lift	N/A
Number of Survived Itemsets	**228, with sup > 126**
Number of Rules	**5 rules**

Collaborative Filtering (Core and Library) – Nour Aldin

Important Note: Nour built a CF algorithm with a library and is also the one who wrote this section of the report.

Brief

The algorithm has been developed both **from scratch** and with the **Surprise Library** to make recommendation based on Collaborative Filtering and predict user rating for a movie. Another popular algorithm for Collaborative Filtering is **Nearest Neighbour**, we chose to write it using the **Surprise library** because it has default implementation for a variety of CF algorithms and to write one from scratch to get a better understanding of the technique.

Datasets Usage

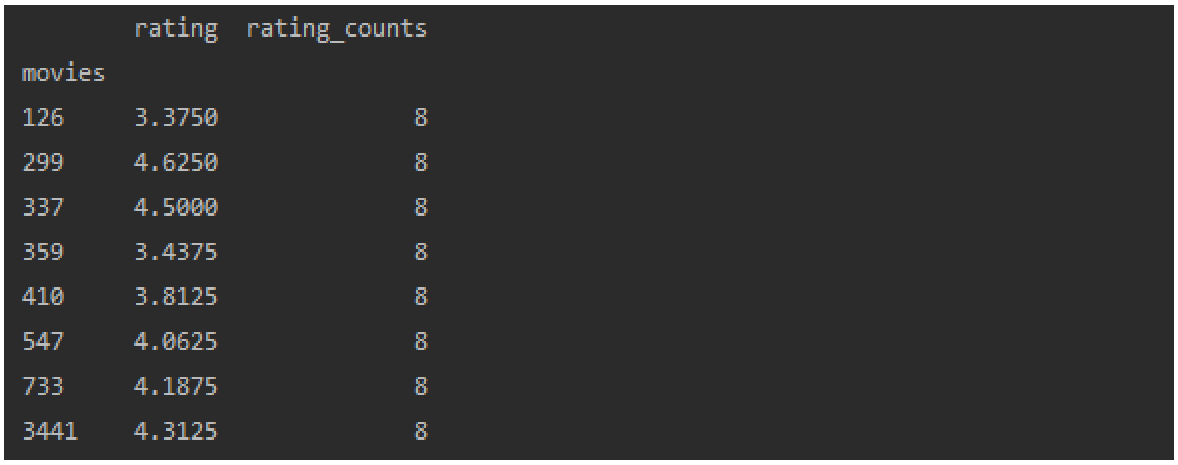
1. *Movielands*: This dataset (ml-latest-small) describes a movie recommendation service. It contains 100836 ratings and 3683 tag applications across 9742 movies. The data contains ratings from 610 unique users made between March 29, 1996 and September 24, 2018. The dataset was generated on September 26, 2018. The users were selected at random for inclusion. All selected users had rated at least 20 movies. No demographic information is included. Each user is represented by an id, and no other information is provided.
    - User Ids: *Movielands* users were selected at random for inclusion
    - Movie Ids: Only movies with at least one rating or tag are included in the dataset.
    - Ratings: Each line of this file after the header row represents one rating of one movie by one user.

    The lines within this file are ordered first by ushered, then, within user, by movieId. Ratings are made on a 5-star scale, with half-star increments (0.5 stars - 5.0 stars).

Experiments and Evaluation *(What experiments can be done to prove it works)*

1. Find the average rating and number of ratings for the movie?

    To find the average rating of each movie. To do so, we grouped the dataset by the title of the movie and then calculate the mean of the rating for each movie. Then we sorted the ratings in the descending order of their average ratings, then we have added the number of ratings for a movie to the ratings_mean_count data frame as showing in figure 8.



```
        rating  rating_counts
movies
126     3.3750              8
299     4.6250              8
337     4.5000              8
359     3.4375              8
410     3.8125              8
547     4.0625              8
733     4.1875              8
3441    4.3125              8
```

Figure 8 average rating and number of ratings for the movie

To have a clear visualization we have plotted a histogram chart for the number of ratings represented by the "rating_counts" column in the above data frame. This can be seen in Figure 9.

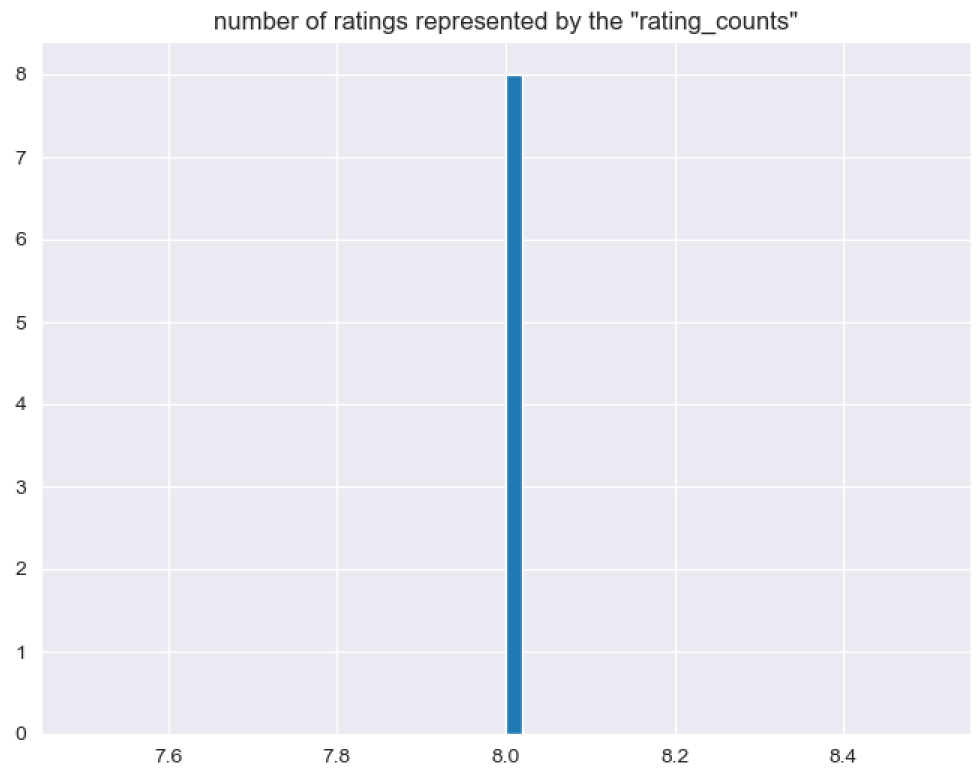


Figure 9. Number of ratings represented by the "rating_counts"

The output in figure 9, clearly shows that most of the movies have received 8 ratings. To view the average ratings for each movie, we have plotted a histogram chart as showing in Figure 10.

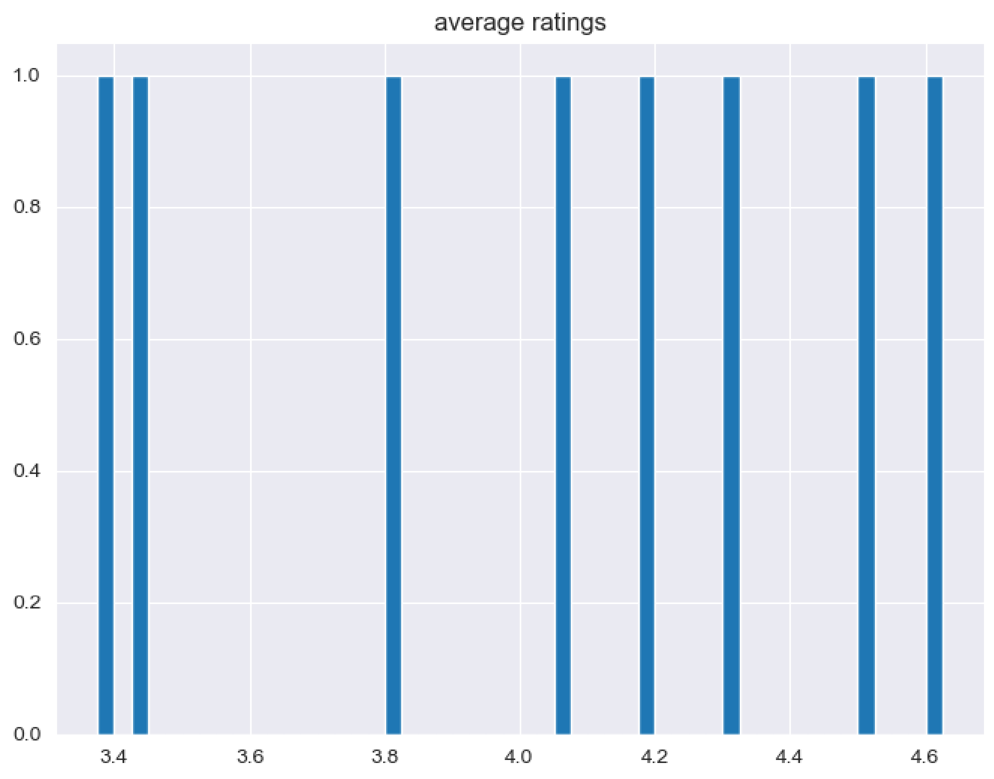


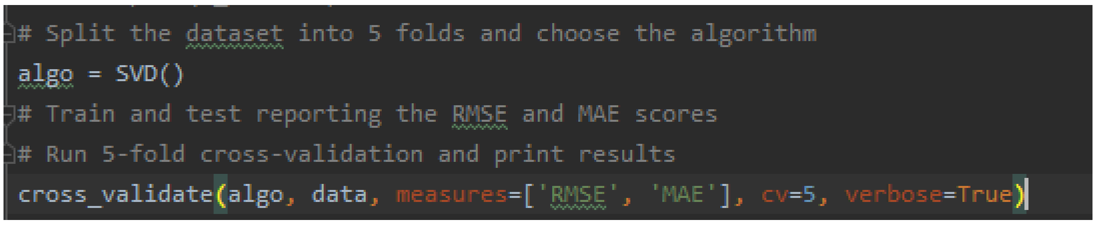
Figure 10. Average ratings.

2. How can we find the predicted rating?

As in many other Machine Learning algorithms training is used to optimize its predictions to match as closely as possible the actual results. So, in the context of collaborative filtering, our algorithm will try to predict the rating of a certain user-movie combination and it will compare that prediction to the actual prediction. The difference between the actual and the predicted rating is measured using classical error measurements such as Root mean squared error (RMSE) and Mean absolute error (MAE).

In the **Surprise library** we have a wide choice of algorithms to use and a wide choice of parameters to tune in each algorithm. One of the famous available algorithms mentioned is **SVD**. For the purpose of our example we will use the SVD algorithm. For the sake of simplicity, we will use the algorithm as is without adding many customisations through the available parameters.

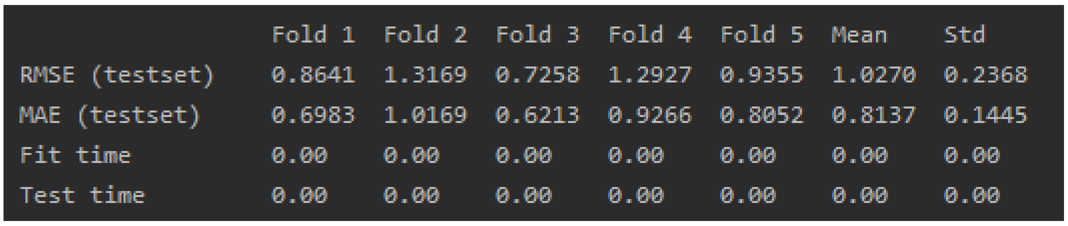
The **Surprise library** also supports the **RMSE** and **MAE** measurements so we will use those to measure the performance of our algorithm below in Figure 11.



```
# Split the dataset into 5 folds and choose the algorithm
algo = SVD()
# Train and test reporting the RMSE and MAE scores
# Run 5-fold cross-validation and print results
cross_validate(algo, data, measures=['RMSE', 'MAE'], cv=5, verbose=True)
```

Figure 11. optimization code

Figure 12. displays the performance of our algorithm through the use of 5-fold Cross Validation.



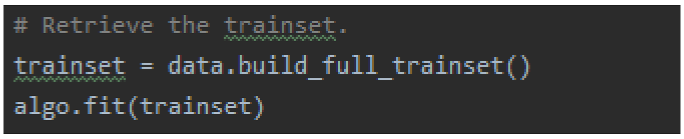
```
                  Fold 1  Fold 2  Fold 3  Fold 4  Fold 5  Mean    Std
RMSE (testset)    0.8641  1.3169  0.7258  1.2927  0.9355  1.0270  0.2368
MAE (testset)     0.6983  1.0169  0.6213  0.9266  0.8052  0.8137  0.1445
Fit time          0.00    0.00    0.00    0.00    0.00    0.00    0.00
Test time         0.00    0.00    0.00    0.00    0.00    0.00    0.00
```

Figure 12. SVD Performance

Predicting

Finally, we are interested in the predicted rating for a certain user-movie combination so we know whether the user will like this movie or not.

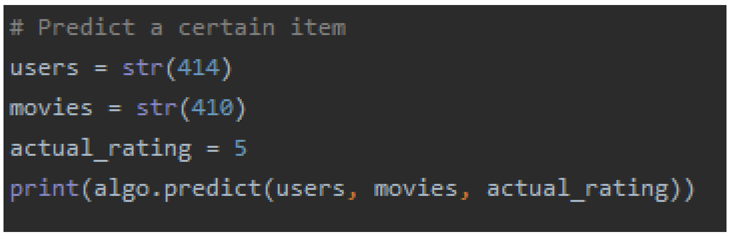
To do this, first we will train on the whole data set without splitting the data to get the best results possible as showing in Figure 13.



```
# Retrieve the trainset.
trainset = data.build_full_trainset()
algo.fit(trainset)
```

Figure 13. Train on the whole data.

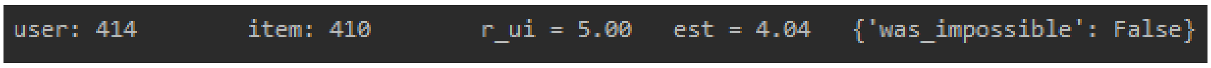
To predict a rating, we give the algorithm the users, movies and the actual rating data as follows in Figure 14.



```
# Predict a certain item
users = str(414)
movies = str(410)
actual_rating = 5
print(algo.predict(users, movies, actual_rating))
```

Figure 14. Predict the rating of a certain item.

And from the results we can see that the estimated rating is 4.04 compared to the actual rating of 5 as displaying in figure 15.



```
user: 414        item: 410        r_ui = 5.00   est = 4.04   {'was_impossible': False}
```

Figure 15. Predicted vs actual rating.

Contributions and Peer Mark

Table 2: Peer Mark and Justification

	Aleksandra	**Nour**
Aleksandra	7 – I believe this is a fair mark for me because I was always there to support everyone despite the fact that I also have my own work to do; Wrote one half of the report; Proof-read the full report and the presentation; Rewrote paragraphs and sentences multiple time to make sure it was written well; Setup and maintained the source control; Built the Apriori algorithm from scratch and spent a long time making sure it all works as intended; also spent a while improving the code to gain extra performance, as described in the Apriori section despite the hard work the core python version of the algorithm could not compete with the library version in terms of processing speed. Despite that I believe it does what was intended and has allowed me to learn a lot.	6 – I believe this a fair mark because despite being new to programming and data science, he showed initiative and improvement, always asked for help when stuck. This was also an issue as sometimes it seemed like there was not much effort put to fix the bug by himself before asking for help. This is not normally great however considering the load of work and pressure to complete everything in time I believe this is acceptable in some cases such as fixing a bug with loading a file, including a library, etc. He also wrote half of the report, the presentation and 2 algorithms with a library.
Nour	7- It is a fair mark because she was so helpful as well as she produced a good work as mentioned above and she was so motivated with great job.	6- This is a fair mark because I was new to programming, as I built Association Rules with library and Collaborative Filtering with help from Alex when I was stuck with any problem. Also I did put an effort to make sure every think is efficient and done, alongside I have written have of the report and presentation with proof reading for both.

Conclusion

Association rule mining algorithms such as Apriori are very useful for finding simple associations between our data items. They are easy to implement and have high explain-ability. However, for more advanced insights, such those used by Google or Amazon etc. In this project, we studied what a recommender system and association rules mining are and how we can create them in Python using many different libraries and also from scratch.

References

Garg, A., 2018. *Complete guide to Association Rules (1/2).* [Online]
Available at: https://towardsdatascience.com/association-rules-2-aa9a77241654
[Accessed Febuary 2020].

Malaeb, M., 2017. *The easy guide for building python collaborative filtering recommendation system.* [Online]
Available at: https://medium.com/@m_n_malaeb/the-easy-guide-for-building-python-collaborative-filtering-recommendation-system-in-2017-d2736d2e92a8
[Accessed April 2020].

Ng, A., 2016. *Association Rules and the Apriori Algorithm: A Tutorial.* [Online]
Available at: https://www.kdnuggets.com/2016/04/association-rules-apriori-algorithm-tutorial.html
[Accessed Febuary 2020].