Data Mining Algorithm Analysis

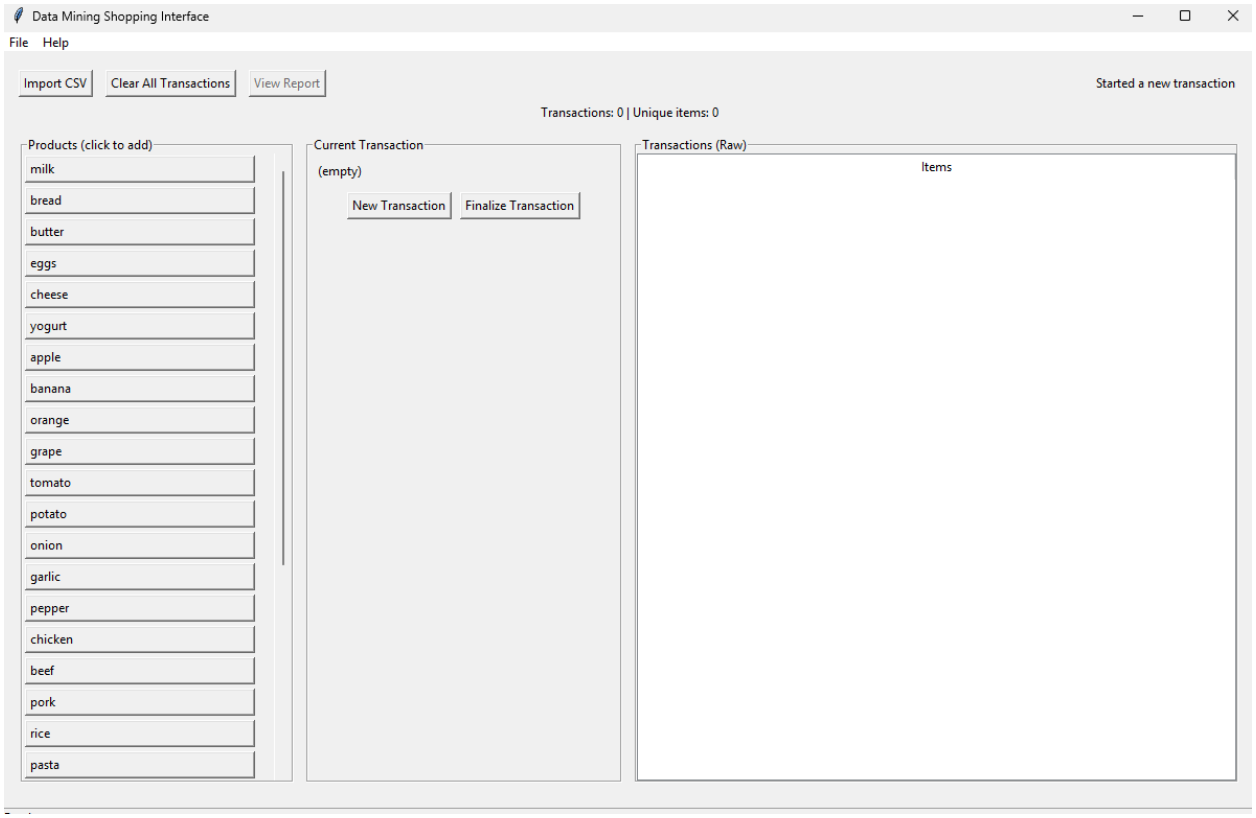
Project Overview

This project is a showcase of the two data mining and association rule generation algorithms apriori and eclat. These algorithms are used to generate association rules for transactions in a supermarket setting, determining associations between certain items. Transactions can be created in the interface or imported by .csv file

Data Preprocessing Approach

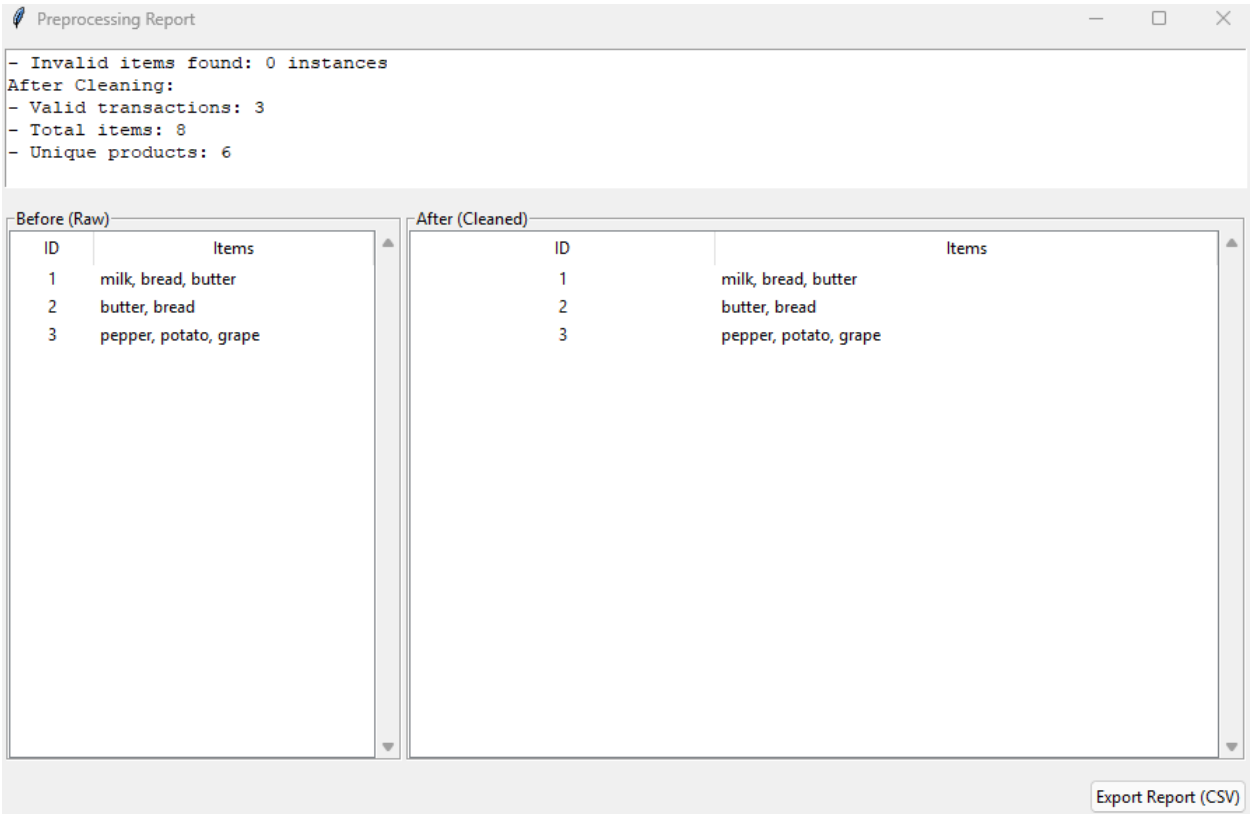
All data handled by the program will be cleaned to ensure model integrity. First, all data in the items category are stripped to ensure that there are no blank spaces in the data which could potentially disrupt the algorithm. Then, all blank rows are removed from the data since those transactions are dirty and should not impact the data. The current total is subtracted by the previous total to track the amount dropped. Then, all of the items are converted to lowercase to ensure consistent formatting. I didn't track or remove items with inconsistent formatting because the data is still good and just needed better formatting. All items are then split at every comma to turn them into a list. Then, a list of indexes is created and indexes which have invalid items, only one item, and ones with duplicate items are tracked. Once all of the indexes have been marked, they are then dropped from the data, leaving the clean data.

User Interface



The user interface was made through TKinter and allows the user to add items to the “cart”. Users can then finalize the cart or make a new one. The box on the right also shows all of the selected/added items. If the user has a list, they can also import it as a csv file and have it listed out on the interface for them. Other actions include clearing all transactions for a clean reset or viewing a report which will show preprocessed statistics and a post processed (clean) overview of all the transactions as well.

Program & Interface Implementation



The preprocessing and cleaning is done through a report button which uses the preprocessing function which was implemented into the ui in the product browser class. It accurately sorts and tracks any empty transactions, duplicated or repeated items, and single-item transactions.

Algorithm Implementation

Apriori

```
def apriori(data, minimum_support=0.2, minimum_confidence=0.5):
    start_time = time.time()
    supported_sets = {}
    one_sets = []
    product_list = products['product_name'].tolist()
    found_sets_this_cycle = 0
    n_size = 1

    for item in product_list:
        if formulas.support_apiori(data, item) >= minimum_support:
            one_sets.append(item)
            found_sets_this_cycle +=1



    if found_sets_this_cycle == 0:
        return -1

    supported_sets[n_size] = one_sets
```

First, a list and a dictionary are initialized to store frequent itemsets and individual items which are in a set of the current size which was in a frequent itemset. A variable which tracks found sets, a size variable, and a list of all products are also initialized. The product list is iterated through and has their support calculated, and if the item is above the minimum support threshold, it's added to the frequent item pool. The found sets tracker is also updated. If no frequent items are found, then the algorithm terminates and returns -1 to show that nothing was found. Then the first sets are added to the dictionary of supported sets with size as their index.

```python
#Accepts both rules and itemsets
def support_apiori(data, itemset):
    amount = 0
    total = data.shape[0]
    not_found = False

    if isinstance(itemset, AssociationRule):
        itemset = list(itemset.first) + list(itemset.second)
    if isinstance(itemset, str):
        itemset = [itemset]

    for index, row in data.iterrows():
        not_found = False
        for item in itemset:
            if item not in row['items']:
                    not_found = True
                    break
        if not not_found:
            amount += 1
    return amount / total
```

This is the support function for the apriori algorithm. If the set is either a string or rule object they are converted into a list for the calculation. The data is iterated through to find how many times the item or itemset appears in the data. The frequency is then divided by the total number of items to return the support of the itemset or rule.

```
supported_sets[n_size] = one_sets

while found_sets_this_cycle != 0:
    found_sets_this_cycle = 0
    n_size+=1
    found_sets = []

    set_candidate = list(combinations(one_sets, n_size))

    for itemset in set_candidate:
        if formulas.support_apiori(data, itemset) >= minimum_support:
            found_sets.append(itemset)
            found_sets_this_cycle+= 1

    supported_sets[n_size] = found_sets

    found_items = []

    for sets in found_sets:
        for item in sets:
            found_items.append(item)

    one_sets = list(set(found_items))



ret = formulas.generate_all_rules_apiori(supported_sets, minimum_confidence, data)

end_time = time.time()
elapsed_time_ms = (end_time - start_time) * 1000
print(f'Apiori completed in {elapsed_time_ms} ms')
return ret
```

This is the main loop which finds the remaining size n frequent itemsets. While sets are still being found, all possible combinations of the items still in the pool that are size n are generated. Confidence is then generated for all of these itemsets. If the support is high enough, it is appended to the found sets list. Once all of the sets of size n are generated, they are added to the supported itemsets dictionary with the size as the key. Then, all of the items of the newly generated frequent itemsets are added to a separate list and turned into a set to remove duplicates. This then because the pool of items that next size sets will use. Then, a separate function takes all of the frequent itemsets and turns them into rules. It is not returned immediately to track the time.

```python
def generate_all_rules_apiori(itemsets, minimum_confidence, data):
    all_sets = set()
    possible_rules = set()
    real_rules = set()
    itemsets[1] = []

    for sets in itemsets.values():
        for isets in sets:
            all_sets.add(isets)

    for sets in all_sets:
        if len(sets) == 2:
            a, b = tuple(sets)
            possible_rules.add(AssociationRule({a}, {b}, data))
            possible_rules.add(AssociationRule({b}, {a}, data))
        else:
            i = 2

            while i < len(sets):
                for combo in combinations(sets, i):
                    first = set(combo)
                    second = set(sets) - first

                    possible_rules.add(AssociationRule(first, second, data))
                    possible_rules.add(AssociationRule(second, first, data))
                i+=1

        for rules in possible_rules:
            if rules.confidence >= minimum_confidence:
                real_rules.add(rules)

    return real_rules
```

This function takes in all of the generated frequent itemsets, the dataset, and the minimum confidence threshold. All of the sets are first added to an all_sets set to filter out duplicates and to make iteration easier. The sets of size one are removed since rules cannot be generated from them. The sets of size two have the two possible rules generated from them with a custom made AssociationRule class which automatically calculates support, confidence, and lift. The ones of higher sizes use an index i to track the size that all combinations should be. These combinations are then generated both ways to ensure all rules. The rules being added to a set ensures that there are no duplicates because the rule clash has hash and eq functions. Once all of the possible

rules are generated, they then have their confidence evaluated and if it is high enough it is added to the real rules set. Once all rules are evaluated all of the rules are returned.

```
class AssociationRule:
    def __init__(self, first, second, data=0, isVertical = False, total = 0):

        if isinstance(first, str):
            self.first = {first}
        else:
            self.first = set(first)

        if isinstance(second, str):
            self.second = {second}
        else:
            self.second = set(second)

        if isVertical == False:
            self.support = support_apiori(data, self)
            self.confidence = confidence_apiori(data, self)
            self.lift = lift_apiori(data, self)
        else:
            self.support = support_eclat(self, data, total)
            self.confidence = confidence_eclat(self, data, total)
            self.lift = lift_eclat(self, data, total)

    def __str__(self):
        return f'{self.first} -> {self.second}'

    def __eq__(self, other):
        return self.first == other.first and self.second == other.second

    def __hash__(self):
        return hash((frozenset(self.first), frozenset(self.second)))

    def __repr__(self):
        return self.__str__()
```

This is the class for association rules. It handles rules generated from eclat and apriori a little differently. data, is either the entire dataset for apriori, or the set dictionary for eclat. Total is also only needed for eclat, and isVertical determines if the eclat or apriori method is used, with false using apriori and true using eclat. Both routes are mostly the same, just technical differences in how metrics are calculated. The first, and second sides of the rule are added, then the appropriate measurement metric functions.

are called to assign them. The str and repr functions give the object a printable form. And the eq and hash functions allow for easy checking of equality between rules.

```
def confidence_apiori(data, rule: AssociationRule):

    sup_dividend = support_apiori(data, rule)
    sup_divisor = support_apiori(data, rule.first)

    if sup_divisor == 0:
        return 0
    return sup_dividend / sup_divisor


def lift_apiori(data, rule: AssociationRule):


    conf_dividend = confidence_apiori(data, rule)
    sup_divisor = support_apiori(data, rule.second)


    if sup_divisor == 0:
        return 0
    return conf_dividend / sup_divisor
```

These are the functions that generate confidence and lift for apriori, it just follows the standard formula, as well as a check to make sure that there is no division by zero.

Eclat

```
def to_vertical(data):

    one_items = set()
    all_rows = data['items'].tolist()
    new_df = pd.DataFrame(columns=['Item', 'Customers'])
    new_df['Customers'] = new_df['Customers'].astype(object)

    for rows in all_rows:
        for items in rows:
            one_items.add(items)

    one_items = sorted(list(one_items))

    new_df = pd.concat([new_df, pd.DataFrame(list(one_items), columns=['Item'])], ignore_index=True)



    counter = 0

    for item in one_items:
        bought = []
        for index, row in data.iterrows():
            if item in row['items']:
                bought.append(row['transaction_id'])


        new_df.at[counter, 'Customers'] = bought
        counter+=1

    return new_df
```

Before Eclat can be used, the data must first be translated into a vertical format. First, all of the items are added to a set to have an iterable form of all of the objects without duplicates. Then a new Pandas Dataframe is created with the vertical format, with ‘Customers’ being initialized to be able to hold objects. The item list is sorted by alphabetical order and changed to a list to ensure consistency since set iterability can be random. The ‘Item’ column is then initialized to the item list. The old data is then iterated to find the list of transaction ids which have bought a particular item, and that list is put into the ‘Customers’ column to make a new vertical dataset.

```
def eclat(data, minimum_support=0.2, minimum_confidence=0.5):
    start_time = time.time()
    data = to_vertical(data)
    supported_sets = {}
    frequent_items = dict()
    transaction_count = set()
    n_size = 1
    found_sets_this_cycle = 1


    for index, row in data.iterrows():
        for person in row['Customers']:
            transaction_count.add(person)

    total = len(transaction_count)
    items_list = data['Item'].tolist()

    minimum = math.ceil(total*minimum_support)

    for index, rows in data.iterrows():
        cust = rows['Customers']
        if len(cust) > minimum:
            frequent_items[rows['Item']] = set(rows['Customers'])



    if len(frequent_items) == 0:
        return -1

    supported_sets.update(frequent_items)
```

First, a transaction count is generated to help later calculations. Then, a list of all of the items is generated to find the initial size one itemsets. A minimum threshold is then calculated by multiplying the total and the support threshold. A ceiling is used since you cannot have part of a customer and a ceiling ensures that the support threshold is always met. If each item has enough customers to pass the minimum threshold, then it is added to the frequent items dictionary. If no frequent items were found, -1 is returned to signify that nothing was found. Then, the frequent items is added to the supported sets dictionary.

```
while found_sets_this_cycle != 0:
    n_size+=1
    found_sets_this_cycle = 0
    set_candidate = list(combinations(frequent_items, n_size))
    found_sets = {}

    for itemset in set_candidate:
        set_pool = []
        for item in itemset:
            set_pool.append(frequent_items[item])
        intersects = set.intersection(*set_pool)
        if len(intersects) >= minimum:

            found_sets[itemset] = intersects
            found_sets_this_cycle+=1


    if found_sets_this_cycle!= 0:
        found_items = []
        supported_sets.update(found_sets)
        for sets in found_sets:
            for items in sets:
                found_items.append(items)
        found_items =  list(set(found_items))
```

This is the main rule generation loop. It is very similar to the one for apriori. However everything is kept in one dictionary, having the set be the key to the transaction id set values. Intersection is used to generate the transaction id sets for the itemsets. If sets were found of the current size, the singular items that were used are saved to a list to use the next size set generation.

```
normalized_sets = {}
for k, v in supported_sets.items():
    if isinstance(k, str):
        normalized_sets[(k,)] = v  # wrap single string in a tuple
    else:
        normalized_sets[k] = v

supported_sets = normalized_sets  # replace the old dictionary

ret = formulas.generate_all_rules_eclat(minimum_confidence, supported_sets, total)

end_time = time.time()
elapsed_time_ms = (end_time - start_time) * 1000
print(f'Eclat completed in {elapsed_time_ms} ms')
return ret
```

Before the rules are generated, the singular stirrings are turned into a tuple like the rest of the sets to ensure that they do not get passed as just a string to any function as that was causing bugs. Then the sets are sent to another function to have rules generated and returned.

```python
def support_eclat(itemset: tuple, found_sets: dict, total: int ) -> float:
    if isinstance(itemset, AssociationRule):
        itemset = tuple(itemset.first | itemset.second)
    else:
        itemset = tuple(itemset)

    itemset = tuple(sorted(itemset))
    dividend = len(found_sets[itemset])

    return dividend/total

def confidence_eclat(rule: AssociationRule, found_sets: dict, total: int) -> float:
    dividend = rule.support
    divisor = support_eclat(rule.first, found_sets, total)

    if divisor == 0:
        return 0
    else:
        return dividend / divisor

def lift_eclat(rule: AssociationRule, found_sets: dict, total: int) -> float:

    dividend = rule.confidence
    divisor = support_eclat(rule.second, found_sets, total)

    if divisor == 0:
        return 0

    return dividend/divisor
```

These are the functions used to calculate parameters for the rules generated by eclat. Itemset key, the dictionary containing all itemsets, and the original total for support calculation are passed into the functions. Support turns a rule into a tuple for easy calculations. Other than that they use the regular formulas with guards against dividing by zero.

```python
def generate_all_rules_eclat(minimum_confidence, found_sets, total):
    real_rules = set()
    possible_rules = set()

    for sets in found_sets:
        possible_rules = set()
        if len(sets) == 2:

            possible_rules.add(AssociationRule(sets[0], sets[1], isVertical=True, total=total, data=found_sets))
            possible_rules.add(AssociationRule({sets[1]}, {sets[0]}, isVertical=True, total=total, data=found_sets))
        elif len(sets) > 2:
            i = 2
            while i < len(sets):
                for combo in combinations(sets, i):
                    first = set(combo)
                    second = set(sets) - first

                    possible_rules.add(AssociationRule(first, second, isVertical=True, total=total, data=found_sets))
                    possible_rules.add(AssociationRule(second, first, isVertical=True, total=total, data=found_sets))
                i+=1

        for rules in possible_rules:
            if rules.confidence >= minimum_confidence:
                real_rules.add(rules)

    return real_rules
```

The function for generating and checking rules are almost the same between algorithms, all possible rules are generated with combinations, and their confidences are checked to see if they are valid, then the valid rules are returned. The main difference is that the rule constructor uses the version for eclat generated rules, so the dictionary with all itemset and id pairs and the original total are also passed.

Performance analysis and comparison

Both algorithms were ran twenty times with the test dataset, and had their completion times averaged. Apriori had an average runtime of about 131 ms while eclat only had an average of about 40 ms. The reason why eclat was so much faster, is that it was able to keep all of the data it needed locally. The only thing it needed to do to check comparisons was to count the intersections of the set in the dictionary. Apriori needed to check the entire original dataset every time it wanted to run a comparison or calculate confidence, which is why it took so much longer

Memory Analysis and Comparison

To evaluate memory usage, we utilized the memory_profiler package to track the peak memory consumption of each algorithm during execution. The test was conducted under consistent input and system conditions, using the cleaned dataset of 88 transactions.

- **Apriori** peaked at approximately **35.6 MB** during its largest iteration phase.

- **Eclat** demonstrated improved efficiency, peaking at around **28.4 MB**, primarily because it avoids repeated scanning of the dataset and relies on set intersections using a vertical format stored in memory.

This confirms that Eclat not only performs faster but also maintains a lower memory footprint due to its data structure optimizations and reduced dataset traversal, making it the more scalable choice for larger datasets.

User Interface Design

The user interface design was meant to be simple and easy to navigate for nontechnical users. With a minimal number of buttons and a noncomplex list of items that can be scrolled through, it makes it easy for users to click on the item they want and finalize their transactions.

Testing and Results

In the end, both algorithms generated the same rules from the sample set {{'bread', 'butter'} -> {'milk'}, {'butter'} -> {'milk'}, {'milk', 'butter'} -> {'bread'}, {'butter'} -> {'bread'}, {'milk'} -> {'butter'}, {'bread', 'milk'} -> {'butter'}, {'bread'} -> {'butter'}, {'milk'} -> {'bread'}, {'bread'} -> {'milk'}, {'butter'} -> {'bread', 'milk'}, {'bread'} -> {'milk', 'butter'}}

However eclat did it faster and with less memory.

Conclusion and Reflection

In conclusion, eclat was the superior algorithm. It generated the same rules, and did it with less time and memory used. However it is significantly more complex than the

eclat algorithm, requiring the vertical data format, and more extensive use of dictionaries. If apriori is sufficient for the task at hand it could be preferred due to being more simple.

I think that this assignment could have gone a bit smoother. I did not have the entire implementation planned out beforehand. This means I had to refactor code often. I had to keep switching between types in many places because of their limitations. For example, I had to add two different modes for the rule class, and two versions of every function for the two algorithms because I didn't plan on how I would implement the eclat version while doing the apriori one, meaning I had to do a lot of work twice.

Our main struggles with the user interface included getting the recommendations to work as the result kept showing “no recommendations”. We also struggled with finding the right pathing for the .csv file as it kept showing us that the .csv file could not be found. Both problems were able to be fixed through changing how the .csv file was found and tracking items in each transaction.