CSCE847 Assignment 2

Introduction to Data Mining

Arun Narenthiran Veeranampalayam Sivakumar, Jacob Shiohira

February 19, 2018

Table of Contents

Introduction **2**

Apriori Algorithm Implementation **2**

Requirements and Specifications 2

Algorithm Design 2

General Program Design 2

Extracting Input from the arff File 3

Handling the Class/Label of Each Instance 4

Main Data Structure Used 4

Determining Frequent Sets 4

Generating the Association Rules 5

Algorithm Runtime 5

Algorithm - Weka Comparison 6

Conclusion **8**

Introduction

Association analysis is a technique used to discover interesting relationships hidden in large data sets. Association rules can be used to represent these relationships amongst frequent itemsets. Rules generated through association analysis are utilized in a number of domains such as retail, bioinformatics, web mining, medical diagnosis, etc. For example, imagine in the analysis of patience in some niche of medical diagnosis, that association patterns may reveal interesting connections amongst patience living conditions, physical attributes, DNA, and ultimate diagnosis. Such associations may help doctors develop a better understanding of how a condition develops and how different areas of a person's life may contribute to the development of such a condition.

The main two issues with association analysis are concerned with the computing cost of generating association rules and discerning what rules have merit versus simply happen by chance. This assignment addresses the first issue by looking at the Apriori algorithm, which is used for generating frequent itemsets amongst a list of transactions in a dataset. The Apriori algorithm uses a candidate generation-and-test approach. Because of the size of most data sets today, it is naive to do an exhaustive search on the data set. So, the Apriori algorithm makes use of properties such as the downward closure property and the anti-monotone property to prune the number of itemsets considered. This approach helps cut down on computational complexity.

Apriori Algorithm Implementation

Requirements and Specifications

The implementation of the algorithm meets the following requirements. It runs on the CSE server with the following command:

```
$ python3 runApriori.py -i <input file> -s <minsup> -c <minconf> -o <output file>
```

More specific information is specified in the README file turned in. The output file is a text file that contains the association rules along with their support and confidence. The output file generated from the program is a text file that matches the style of output from Weka. The program is written to parse the necessary information from an *.arff* file and then proceed to run the apriori algorithm and generate the association rules.

Algorithm Design

General Program Design

The Apriori program is broken down into 3 main components based on the main purpose of each file.

`fileUtils.py`

This file deals with all functionality related to parsing the *.arff* input file and converting the data into a format that the apriori algorithm would use. Since there is an inherent structure to a valid *.arff* file, it is

rather easy to gather the components of the file necessary for the apriori algorithm. After parsing the *.arff* file, functions in this file convert the file data into a valid format for the apriori algorithm. For example, the *vote.arff* data set contains a list of headers, also known as attributes, and a list data that is all comma separated. All of the data is nominal, belonging to one of two classes - *n* or *y*. In this format, it would be difficult to discern the difference between a generic *n* and *y* without knowing the column to which it belonged. Thus, a function in this file first prepends the header of element of the data and then encodes all of the data to integers. By converting to integers, the algorithm will run more efficiently as opposed to dealing with the strings associated with each column. It is much faster to compare the equality of 0 and 1 rather than *export-administration-act-south-africa=y* and *education-spending=n*. Further, the dictionaries of data to integers and integers to data are stored, so that the algorithm can later derive some sort of semantic value from the list of association rules generated.

The benefit of separating this functionality from the main apriori algorithm is that if the data source changed, it would not be difficult to incorporate the change. All the apriori algorithm cares about is that it gets a set of transactions and a list of items.

`apriori.py`

This file deals with all functionality related to the actual implementation of the apriori algorithm. It is in its own file so as not to convolute the algorithm logic with other functions necessary to prepare the data and run the program. Outside of the code pertaining to the actual Apriori algorithm and the generation of association rules, this file contains 3 other types of functions: generation, get, and conversion functions. For example, `get_transactions_and_items_data` and `get_item_support` pertain to getting certain values that program knows about, `generate_itemsets_with_adequate_support` and `generate_subsets` pertain to generating combinations of itemsets, and `convert_itemset_ints_to_strs` pertains to converting the integer encoded data into meaningful strings. See the subsections titled **Main Data Structure Used, Determining Frequent Sets**, and **Generating the Association Rules** to read more about more specific details pertaining to the implementation of the algorithm.

`runApriori.py`

This file deals with all functionality related to actually running the apriori algorithm. It deals with the user input when the program is started via the CLI and sets the initial values necessary for the algorithm, such as minimum support and minimum confidence. Further, it contains logic regarding the two main runtime paths of the program - the running of the apriori algorithm, association rule generation, and subsequent association rule serialization; and the stress test of the algorithm, which measures how the level of support affects the runtime of the algorithm and the number of rules generated.

Extracting Input from the *arff* File

There are two main parts of the *arff* file that are important to the apriori algorithm. The first is the section containing *@attribute* at the beginning of each line. These lines tell the algorithm what the column header names are for each element in an instance of the data. The lines also provide insight to all of the classes that might appear in a column. However, this implementation does not take those into consideration when constructing the list of header attributes, which are also the column attributes. Instead, when parsing the

data, we only care about elements of a class when we see the element of the class for a specific attribute. Thus, at runtime when iterating through the data and prepending column header names, we then track the specific class. The second is the section containing the data, which is specified by a line containing *@data*. Any line after that until the end of the file is considered data that we will read in and use in the apriori algorithm and generation of association rules.

All lines that begin with % signify a comment at the beginning of the line and will be ignored. Any empty lines between instances of data are also ignored.

Handling the Class/Label of Each Instance

The issue with the *vote.arff* file was that a single column's *y* or *n* was not discernible from another column's *y* or *n*. Thus, association rules could only be derived between the *y's* or *n's.* In order to derive the maximum number of association rules, the program differentiates between different columns' *y* and *n.* To do this, each element is prepended with its column's header name. For example, a *y* in the *crime* column would become *crime=y*. Each *y* and *n* then becomes distinct such that there can be association rules between each *y* as well as each *n*.

This can be extended to any *arff* file containing nominal data. The issue then arises when there is numerical data. Whenever an attribute is numeric, each different number will lead to a different class. This could lead to an arbitrarily large number of classes depending on the number of transactions in a dataset. As far as I could tell, Weka does not allow the Apriori algorithm to be run on a dataset with numeric attributes. Any missing attributes of the dataset are excluded from transactions and thus not included in the derivation of association rules.

Main Data Structure Used

The main dataset used in both the apriori algorithm and the generation of association rules is the set. The set intuitively seems like an applicable data structure since the groups of items are called itemsets and candidate sets. Additionally, sets inherently maintain uniqueness of members. Sets, depending on the implementation, can also provide `O(1)` lookup times for membership of an item, which helps to decrease time complexity when checking for an item in an itemset. Additionally, it allows the algorithm to use the efficient built-in set operations such as union and intersection.

The sets of itemsets were paired with a number of dictionaries, which helped keep track of information such as the support count of individual itemsets. Additionally, a dictionary was used to map k to all of the k-itemsets with minimum support. Dictionaries were also used to map encoded integers to the strings of the header-column-data that each represented.

Determining Frequent Sets

Frequent sets are generated using the `generate_itemsets_with_adequate_support` function. The function is passed the `current_candidate_itemsets`, the list of `transactions`, the dictionary of `frequency_sets`, which keeps track of the number of occurrences of an item_set in the list of transactions. The function instantiates a set named `temporary_itemset`, which will be the frequent

itemsets to return. It also instantiates a local copy of the `frequency_sets` dictionary to keep track of the item sets found during a single call to the function.

The first time the function is called, it is passed the list of 1-itemsets. In every other call to the function, it is passed the current list of `current_candidate_itemsets`, which for the k-th call contains k-itemsets. The function then iterates through each item and checks if that particular item set is a subset of each line in the transactions. If the item set is a subset, then the number of instances is incremented in the `frequency_sets` and the `local_frequency_sets.` After testing all items of `current_candidate_itemsets` against the list of transactions, we iterate through the items of the `local_frequency_sets` and check to see if that item set meets the minimum support. If so, it gets added to the `temporary_itemset`. At the end of the for loop, the function returns the `temporary_itemset`.

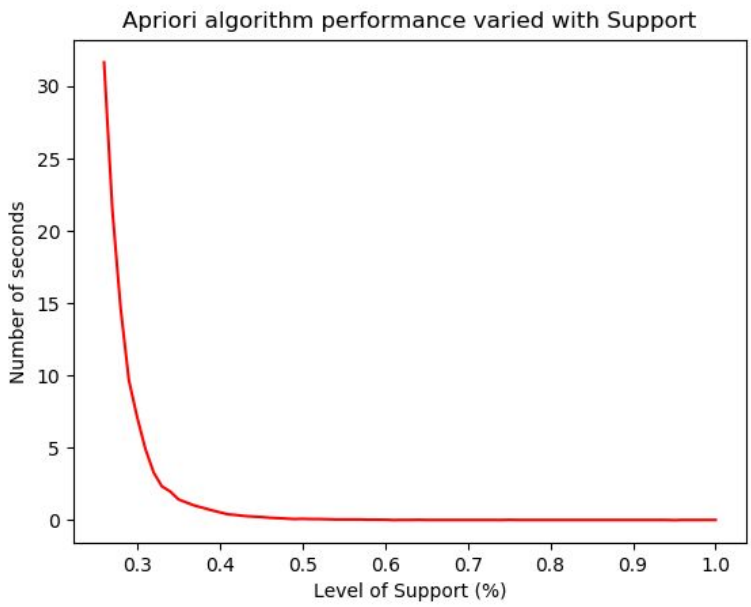
Generating the Association Rules

The association rules are generated as follows. The function `derive_association_rules` is passed the `itemsets_dict` that contains all itemsets of minimum support, the `frequency_set` contains the link between the itemsets and the number of times they occur, the `integer_to_data_dict` that will allow the sets of integers to be mapped to the strings of the form header=data. For example, 24 usually maps to Class=Democrat. The function creates an array called `association_rules`, which will contain the rules to be returned.

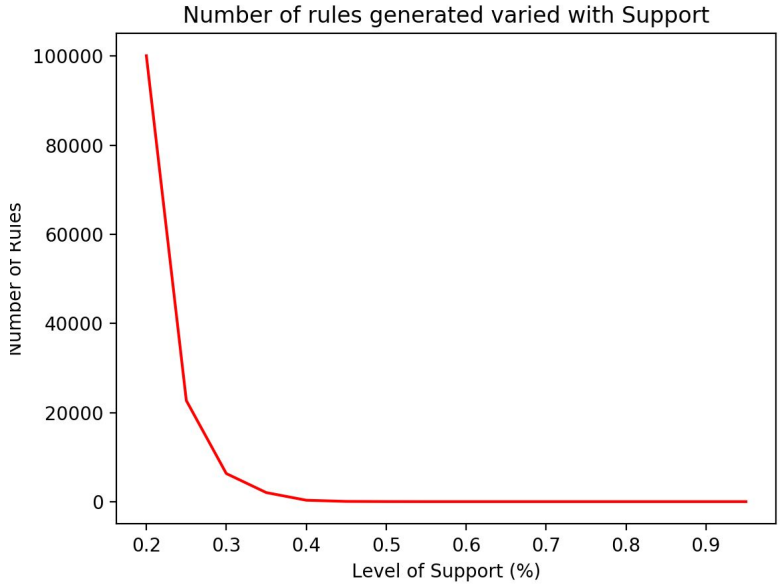
The function iterates through all the key-value pairs of the `itemsets_dict`. For each item in a value, subsets minus the empty set are created. For each subset, the set difference is calculated between the item and the subset. If a difference exists, the support is calculated for each the item and the subset in order to calculate the confidence. If the confidence is greater than or equal to the minimum confidence, the integers in the subset and the difference are converted to the strings of the form header=data. Then, a rule is created and appended to the `association_rules`. Once the function iterates through all of the items in the `itemsets_dict`, the function returns the `association_rules`.

Algorithm Runtime

The algorithm runtime varies with the level of minimum support and confidence, but as minimum support decreases, the apriori algorithm begins to grow exponentially. This is shown in the graph below. It can be seen that until a minimum support of approximately 0.3, the algorithm runs in about a second or less. However, at any level of support lower than that, the time to complete the algorithm increases exponentially. This result is because there as minimum support decreases, there are exponentially more frequent itemsets to consider.



Then, included is a plot of the number of rules generated as a function of minimum support. It can be seen that there are no rules generated by the program until a minimum support level of 0.58 where 1 rule is generated. As the minimum support decreases, we see the number of rules also increase exponentially.



Algorithm - Weka Comparison

Below are 2 tests from *vote.arff* that show the output from Weka and my program. The first test shows all the rules generated with a minimum support of 0.55 and a minimum confidence of 0.9. The second test shows all the rules generated with a minimum support of 0.58 and a minimum confidence of 0.9. All of the association rules are the same between the two outputs, but there are differences regarding the order in which they are listed. Additionally, when there are two or more items in an itemset on either side of the

implication, the outputs can vary based on the order. This is more common when the number of items in an itemset becomes large.

Test 1

Weka output for a support of .55 and confidence of .9:

```
 1. aid-to-nicaraguan-contras=y 257 ==> export-administration-act-south-africa=y 254 <conf:(0.99)>
 2. anti-satellite-test-ban=y 253 ==> export-administration-act-south-africa=y 250 <conf:(0.99)>
 3. adoption-of-the-budget-resolution=y 264 ==> export-administration-act-south-africa=y 259 <conf:(0.98)>
 4. physician-fee-freeze=n 258 ==> Class=democrat 253 <conf:(0.98)>
 5. physician-fee-freeze=n export-administration-act-south-africa=y 251 ==> Class=democrat 246 <conf:(0.98)>
 6. physician-fee-freeze=n 258 ==> export-administration-act-south-africa=y 251    <conf:(0.97)>
 7. physician-fee-freeze=n Class=democrat 253 ==> export-administration-act-south-africa=y 246 <conf:(0.97)>
 8. export-administration-act-south-africa=y Class=democrat 255 ==> physician-fee-freeze=n 246 <conf:(0.96)>
 9. Class=democrat 267 ==> export-administration-act-south-africa=y 255 <conf:(0.96)>
10. physician-fee-freeze=n 258 ==> export-administration-act-south-africa=y Class=democrat 246 <conf:(0.95)>
11. education-spending=n 264 ==> export-administration-act-south-africa=y 251 <conf:(0.95)>
12. Class=democrat 267 ==> physician-fee-freeze=n 253 <conf:(0.95)>
13. Class=democrat 267 ==> physician-fee-freeze=n export-administration-act-south-africa=y 246 <conf:(0.92)>
```

My program output for a support of .55 and confidence of .9:

```
Rule 1: aid-to-nicaraguan-contras=y 257 ==> export-administration-act-south-africa=y 254 <conf: (0.99)>
Rule 2: anti-satellite-test-ban=y 253 ==> export-administration-act-south-africa=y 250 <conf: (0.99)>
Rule 3: adoption-of-the-budget-resolution=y 264 ==> export-administration-act-south-africa=y 259 <conf: (0.98)>
Rule 4: physician-fee-freeze=n 258 ==> Class=democrat 253 <conf: (0.98)>
Rule 5: physician-fee-freeze=n export-administration-act-south-africa=y 251 ==> Class=democrat 246 <conf: (0.98)>
Rule 6: physician-fee-freeze=n 258 ==> export-administration-act-south-africa=y 251 <conf: (0.97)>
Rule 7: physician-fee-freeze=n Class=democrat 253 ==> export-administration-act-south-africa=y 246 <conf: (0.97)>
Rule 8: Class=democrat 267 ==> export-administration-act-south-africa=y 255 <conf: (0.96)>
Rule 9: export-administration-act-south-africa=y Class=democrat 255 ==> physician-fee-freeze=n 246 <conf: (0.96)>
Rule 10: education-spending=n 264 ==> export-administration-act-south-africa=y 251 <conf: (0.95)>
Rule 11: Class=democrat 267 ==> physician-fee-freeze=n 253 <conf: (0.95)>
Rule 12: physician-fee-freeze=n 258 ==> export-administration-act-south-africa=y Class=democrat 246 <conf: (0.95)>
Rule 13: Class=democrat 267 ==> physician-fee-freeze=n export-administration-act-south-africa=y 246 <conf: (0.92)>
```

Test 2

Weka output for a support of .58 and confidence of .9:

```
1. aid-to-nicaraguan-contras=y 257 ==> export-administration-act-south-africa=y 254 <conf:(0.99)>
2. adoption-of-the-budget-resolution=y 264 ==> export-administration-act-south-africa=y 259 <conf:(0.98)>
3. physician-fee-freeze=n 258 ==> Class=democrat 253 <conf:(0.98)>
4. Class=democrat 267 ==> export-administration-act-south-africa=y 255 <conf:(0.96)>
5. Class=democrat 267 ==> physician-fee-freeze=n 253 <conf:(0.95)>
```

My program output for a support of .58 and confidence of .9:

```
Rule 1: aid-to-nicaraguan-contras=y 257 ==> export-administration-act-south-africa=y 254 <conf: (0.99)>
Rule 2: physician-fee-freeze=n 258 ==> Class=democrat 253 <conf: (0.98)>
Rule 3: adoption-of-the-budget-resolution=y 264 ==> export-administration-act-south-africa=y 259 <conf: (0.98)>
Rule 4: Class=democrat 267 ==> export-administration-act-south-africa=y 255 <conf: (0.96)>
Rule 5: Class=democrat 267 ==> physician-fee-freeze=n 253 <conf: (0.95)>
```

Test 3

There was an additional test run on a *weather.nominal.arff*. There are 5 header attributes and 14 transactions in this *arff* file downloaded online. It is included in the *Data* folder of the submitted files. It appears as though Weka rounds when outputting the minimum support in some association rule generation. For example, it is impossible that there could be a minimum support of 30% with 4 instances out of 14 transactions. That would be 28.5%. Then, it can be seen that the association rules are the same.

Weka output for a support of .3 and confidence of .9:

```
1. outlook=overcast 4 ==> play=yes 4 <conf:(1)>
2. temperature=cool 4 ==> humidity=normal 4 <conf:(1)>
3. humidity=normal windy=FALSE 4 ==> play=yes 4 <conf:(1)>
```

My program output for a support of .285 and confidence of .9:

```
Rule 1: temperature=cool 4 ==> humidity=normal 4 <conf: (1.0)>
Rule 2: outlook=overcast 4 ==> play=yes 4 <conf: (1.0)>
Rule 3: humidity=normal windy=FALSE 4 ==> play=yes 4 <conf: (1.0)>
```

Conclusion

This implementation of the Apriori algorithm was written in Python and successfully generates the same set of association rules as Weka, as demonstrated above. For a minimum support of 0.5 and a minimum confidence of 0.9, the algorithm runs in 0.030 seconds, or 30 milliseconds, on average. For a vote set of approximately 400 instances with 17 columns in each instance, the run time seems sufficiently low. Of course, as minimum support is lowered, runtime for the algorithm increases because there are more itemsets to consider. The more itemsets to consider with increasingly decreasing support means that the association rules generated are less and less valuable. However, the exponentially increasing runtime for lower support is still much faster than considering all itemset combinations without taking advantage of properties such as the anti-monotone and downward properties.