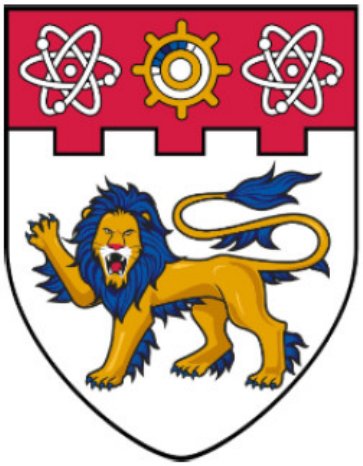

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4032 Data Analytics and Mining
Project 1 Report

Cao Shuwen (U1922953B)

Chang Heen Sunn (U1920383H)

Huang Runtao (U1923490F)

Yin Jia Rui

Part 1 - Introduction

In this project, we have implemented the Classification based Association Rule (CBA) M2 algorithm presented in the KDD98-012 paper [1], the explanation of the code will be elaborated in section 2 of the report. 5 datasets used in the paper and 2 self-sourced datasets were used to conduct classification task with the same setting stated in the paper. The classification results of the 7 datasets can be found in section 3. Classification accuracies of open softwares are in section 4. Last but not least, the designing of the improved algorithm using CMAR [2] is in section 5.

Part 2 - Implementation of CBA M2 Algorithm

2.1 Data Preprocessing

Even though most of the datasets used were downloaded from the UCI machine learning portal, there still exists one main problem in the datasets, which is the discretization of continuous value attributes. The discretization method we employed to solve the issue will be elaborated under this section.

2.1.1 Discretization

Discretization of continuous attributes was handled by the discretization.py python file. In the `binary_split` function, we will iterate through every value in the continuous data column, and use that value as a candidate split point to split to original data block into left and right data blocks. Then `calculate_entropy_gain` and `calculate_minimum_gain` functions will be called to calculate the entropy gain and threshold value for the current splitting. If the current entropy gain is greater than the threshold value, we append the set of `[value, entropy_gain, left_datablock, right_datablock]` into the split point list. When we finishing iterating through every value of the column, the split point list will contain sets of values that satisfy the minimum entropy gain threshold. And the last step we do is to sort the list based on `entropy_gain` and only return a set of values with the highest entropy gain.

Lastly, the main function in discretization.py is the `complete_split` function that includes a `recursive_split` that calls `binary_split` recursively to split the data block returned. As long as the `binary_split` returns a set of values, the first value in the set will be appended to the final split points, and the `recursive_split` will then call `binary_split` twice to split the `left_datablock` and `right_datablock` returned to recursively split the data block until there is no split point returned. Finally, we sort the final split points list to obtain the set of values that are needed to do discretization of continuous data.

However in some rare cases, when the amount of data in the dataset is inadequate or the splitting interval is too large, there will be no split points returned for the continuous data column. We added the condition in `preprocessing_main` to create split points ourselves when the length of the split points is 0. For simplicity, we just choose the values located at 33.3 66.7 percentile to split the continuous data into three equally.

2.2 CBA-RG

The main purpose of CBA-RG algorithm is to find all the ruleitems that have support that is greater than the minimum support (minsup). Our implementation for this part includes two python files, namely ruleitem.py and rulegenerator.py.

2.2.1 RuleItem

The ruleitem.py includes a RuleItem class that will manage all the ruleitems generated. It takes in the condset which consists a set of items, class label and the data list as the input, and output a ruleitem with its values of support count of condset (condsupCount), support count of the ruleitem (rulesupCount), support and confidence. The condsupCount and rulesupCount are calculated by the `calculate_supCount` function, while the `calculate_support` and `calculate_confidence` calculate the support and confidence of a ruleitem respectively.

There are also two print functions for us to better visualize the ruleitem being generated. The `print_ruleitem` will print out each ruleitem in <condset, y> format, while `print_rule` will print out the ruleitem in a condset -> y format as mentioned in the paper.

2.2.2 Rule Generator Algorithm

The rulegenerator.py contains the `FrequentRISet` class to hold the set of frequent ruleitems and the `CARs` class to hold the set of class association rules.

Our rule generator algorithm is different from the one in the CBA paper in the way that our pruning function is only called after the rule mining process, instead of pruning every time a new level of class association rules is generated.

2.3 CBA-CB M2 Classifier

For building up the classifier, we have implemented the CBA-CB algorithm to build an M2 classifier using the classification association rules (CARs) generated from rulegenerator.py. The main function `build_classifier_M2` in CBA_CB_M2.py consists of three stages as mentioned in the paper, our detailed implementation will be explained below.

In Stage 1, we iterate through all the data cases in the data list, and call `find_cRule_index` and `find_wRule_index` to find the index of the highest precedence rule that correctly (cRule) or wrongly (wRule) classifies the data case. If there exists a cRule_index, we add it to the U set, which is the set of all cRules, and update the number of class labels covered by the cRule to add 1. If both cRule and wRule were found, we then compare their precedence by calling the `compare_rules` function. When cRule > wRule, its a simple case where the data case should be covered by the cRule. We then add the cRule to the Q set, which is the set of cRules that have higher precedence than their corresponding wRules. The cRule will also be added to the mark set to indicate that it classifies a data case correctly. However, wRule > cRule is a complex case in which we cannot decide which rule will eventually cover the data case. Therefore, we add the set of values `<data_index, label, cRule_index, wRule_index>` into A set as defined in the paper.

In stage 2, we will iterate through all the `<data_index, label, cRule_index, wRule_index>` sets in A, as it contains all the data cases that we cannot decide which rule should cover in Stage 1. If the wRule_index is found in the mark set, which means it is the cRule of at least once, the wRule will therefore able to cover the data case, and we update the number of class labels covered by the cRule to minus 1, wRule to plus 1. However, if the wRule_index is not in the mark set, we call `find_wSet` to iterate through all rule indexes in U set to find all the cRules that wrongly classify the data case and have higher precedences than the cRule in the current set of A. Next, we iterate through the rule_w in the wSet as these are the rules that may replace cRule to cover the data case because they have higher precedences. This information will be put in the replace field of each rule_w, by adding the set of entry `<cRule_index, data_index, label>` and the number of class labels covered by the rule_w will be updated to plus 1 accordingly. Lastly, we union Q set and wSet as Q should contains all the rules to be used to build the classifier as stated in the paper.

In Stage 3, we choose the final set of rules to form our classifiers M2, the process consists of two steps. In the first step, we sort the Q set according to the relation ">" by calling `sort_CARs` function, to ensure Condition 1 is satisfied in the final selected rules. After sorting the list of rules in Q, we then iterate through all the rules in Q via their index. If the rule no longer correctly classifies any training data cases, meaning the number of label covered by the rule is 0,we will discard it. Vice versa, the rule will be used for our classifier to satisfy Condition 2. We will then try to replace all the rules in the current rule_index's replace field with the current rule as the rule precedes them. To do so, we iterate through every entry of `<cRule_index, data_index, label>` in the replace field generated in Stage 2. If the data case is found already being covered by a previous rule, the current rule will not replace the cRule to cover the data case, and vice versa. The number of label covered by the respective rules will then be updated, based on whether the current rule have replaced the cRule or not. Subsequently, for each selected rule, we update the label count and number of rule errors by calling `count_rule_errors` and `find_label_count`. A default label that represent the mode class label in the remaining training data were also computed using `default_label_selection`. We can therefore calculate the default label errors made by the selected default label by calling `count_default_label_errors`.The total number of errors that the selected rule and the default label will make can be calculated as well. Finally, we insert the set of `<rule-index, default_label, total_errors>` into the classifier using `rule_insertion`.

Last but not least, for step 2, we call `rule_cleaning` to find the position of the rule with a minimum number of total errors, and discard all the rules after the position that introduce more errors. Eventually, `classifiier_M2` is returned.

Part 3: Results

We did 10-fold cross-validation using CBA-CB-M2 classifier with rule pruning on 7 datasets to test our software build in Part 2. Each data file is split into training and testing datasets with a ratio of 9:1, the classifier is first built with the training dataset, and the accuracy rate on the testing dataset is then computed. Average accuracy and total run time are presented in the table below.

The 5 datasets used in the KDD'98 paper that we choose to conduct the classification tasks are glass, wine, iris, pima, and tic-tac-toe. Apart from these 5 datasets, we have also found 2 self-sourced datasets, namely caesarian and car. The accuracy is calculated by using one subtracted from the error rate. (accuracy = (1-error rate)*100%)

	Dataset	Accuracy	No. of class label	No. of attributes	Rule generator run time/s	Classifier run time/s	No. of CARs generated (with pruning)
Dataset in paper	glass	97.1%	6	9	3.60	0.01	19
	wine	100.0%	2	486	4.62	0.18	2965
	iris	100.0%	3	4	0.00	0.00	5
	pima	99.9.%	2	8	66.67	0.17	220
	tic-tac-toe	100.0%	2	9	7.27	0.51	857
Others	caesarian	100.0%	2	5	2.23	0.03	448
	car	99.0%	4	6	9.60	0.51	370

Part 4 - Open Softwares Classifiers

In this part, we adopted the Decision Tree (DT) Classifier, Random Forest (RF) Classifier, and Support Vector Machine (SVM) to perform the classification on the seven datasets we have. We applied the same 10-fold-cross-validation to all the classifiers. The table below shows the classification accuracy and its corresponding f-score of each classifier on all the datasets.

	Dataset	DT accuracy	DT f_score	RF accuracy	RF f_score	SVM accuracy	SVM f_score
0	glass	0.583117	0.563330	0.752597	0.708526	0.354978	0.186279
1	wine	0.921895	0.920598	0.972222	0.959691	0.551634	0.465381
2	iris	0.960000	0.952997	0.960000	0.959731	0.973333	0.973064
3	pima	0.710834	0.693754	0.773462	0.752278	0.757861	0.740124
4	tic-tac-toe	0.848739	0.837071	0.898783	0.901619	0.873739	0.871520
5	caesarian	0.537500	0.551479	0.575000	0.561486	0.575000	0.421795
6	car	0.903391	0.908216	0.866356	0.861814	0.710626	0.679711

Upon inspection, the best performing model is the Random Forest Classifier while the worst performing model is Support Vector Machine.

A side-to-side comparison between the CBA_CB M2 algorithm vs the open software classifiers:

Dataset	Accuracy			
	CBA M2	Decision Tree	Random Forest	SVM
glass	97.7%	58.3%	75.3%	35.5%
wine	98.7%	92.2%	97.2%	55.2%
iris	100.0%	96.0%	96.0%	97.3%
pima	99.9%	71.1%	77.3%	75.8%
tic-tac-toe	100.0%	84.9%	89.9%	87.4%
caesarian	100.0%	53.8%	57.5%	57.5%
car	100.0%	90.3%	86.6%	71.1%

As we can obviously tell, the CBA_CB M2 algorithm outperformed all the other open software classifiers by a huge extent in terms of accuracy score.

Part 5 - Improved Algorithm: CMAR

The Classification based on Multiple Association Rules method, CMAR [2], was adopted to improve the algorithm.

5.1 Construct FP-tree

Firstly, the training dataset will be transformed into a FP-tree.

```
f_list = ordered_F_list(training_data, min_sup)
FP_tree_root, FP_header_table = create_FP_tree(training_data, f_list)
```

In new_FP_Tree.py, `ordered_F_list()` takes the training dataset and the minimum support value to get attribute values whose support is larger than minimum support. These attribute values are put into the F-list in support descending order. Then, `create_FP_tree()` scans the dataset again to load every row of data into an FP-tree. During which, the infrequent attribute values, which are not in the F-list, will not be added into the FP-tree. The attribute values in the row of data will also be rearranged in F-list order. Each FP-tree node contains the attribute value, number of times appearing in the dataset in such patterns, and `accu_labels` dictionary, which contains all the class labels and counts from the child nodes after tree shrinking. The nodes which hold the last attribute of the reordered data will also record their class labels with counts.

5.2 Rule Mining

Secondly, rules will be generated from the FP-tree.

```
temp_CR_tree_root = rule_generator(f_list, FP_header_table, min_sup, min_conf,
training_data)
CRTroot, CR_header_table, num_rules = last_pruning(temp_CR_tree_root,
coverage_threshold, training_data)
```

3 rule pruning methods were introduced in the CMAR paper [2]. In new_FP_Tree.py, `rule_generator()` will perform the first 2 methods to generate rules from the FP-tree. The algorithm of `rule_generator()` is below:

```
1  rule_dic = {}
2  for attribute_value in reversed(f_list):
3       projected_paths = []
4       for node with attribute_value in FP_header_table:
5       get path from FP_tree_root to node (FP_tree_root is excluded)
6       projected_paths.append(path)
7       merge node.accu_labels into parent node's accu_labels dictionary
8filtered_path_list = []
9for path in projected_paths:
10      remove infrequent attribute values from path
11            filtered_path_list.append(path)
12      if len(filtered_path_list) == 1:
13            path = projected_paths[0]
14            get class label dictionary from the end of the path
15            Rule_dic[tuple of attributes in the path] = label_dic
16      if len(filtered_path_list) > 1:
17            for path in filtered_path_list:
18                  get class label dictionary from the end of the path
19                  base_tuple = tuple(first attribute in path)
20                  attri_list = list of rest of the attributes in path
21                  find_patterns(attri_list, base_tuple, label_dic, rule_dic)
```

```
22      if len(rule_dic) > 2000:
23            Break
24 CRTroot = CRTNode("CR root", None)
25 for rule_attri_tuple, label_dic in rule_dic.items():
26      label = the label with the most count in label_dic
27      if sup < min_sup: continue
28      if conf < min_conf: continue
29      if prune_x2(rule, training_data): continue
30      rule = [attributes, label_dic]
31      CRT_add_rule(CRTroot, rule,{}, prune = True)
32 return CRTroot
```

Line 5 uses a while loop to look for parent nodes and add them in the path. Line 9 scans the `projected_paths` list once and records the count for each attribute value. The attribute values whose counts are smaller than minimum support will be removed from paths. In line 12-15, according to the paper [2], to avoid triviality, only the full path should be adopted as the attributes in a rule. Line 21 calls `find_patterns()` to recursively find all combinations of attribute values in the path and add them into rule_dic with their class label dictionaries. Line 22-23 restricts the number of frequent patterns to be less than 2000, which is the same setting as CBA. line 26 chooses the class label with the largest count, because it guarantees that this label has the largest confidence and support. Line 27-28 prunes the rules that do not meet minimum support or confidence. Line 29 calls `prune_x2()` to apply the second rule pruning method, which prunes rules whose χ^2 values do not pass the 0.05 probability threshold [3]. Line 31 calls `CRT_add_rule()` to add the rule into a CR-tree, which is similar to FP-tree.

In `CRT_add_rule()` from new_CR_Tree.py, the first rule pruning method is applied, where rules that are more specific and with lower confidence are pruned. The pruning algorithm is below:

When the rule's end attribute has not been inserted into CR-tree:

```
1 if the new node to be inserted exists in CR-tree:
2      if the existing node is an end node of a rule:
3      if the new rule's confidence < existing rule's confidence:
4            return
```

In this case, the existing rule is shorter than the new rule, so the program should stop inserting the new rule if the existing rule's confidence is higher.

After the rule's end attribute has been inserted into CR-tree:

```
1 perform Depth-First Search from the end node of the new rule downwards
2 for every existing rule found:
3    if the new rule's confidence > existing rule's confidence:
4        if the end node of the existing rule is also a leaf node:
5            delete the end node and its parents upwards until another
                 end node or a branching node is reached
6        else:
7            set the class label, support, confidence and X^2 of the end
                 node to be None
```

In this case, the existing rule is longer than the new rule, so the program should set the existing rule's end node to be a normal node, if the new rule's confidence is higher.

After the CR-tree is generated, in new_CR_Tree.py, `last_pruning()` will perform the last rule pruning method and generate a new CR-tree to hold the unpruned rules for classification. The algorithm of `last_pruning()` is the same as the one explained in the CMAR paper [2].

5.3 Classification

After all the pruning processes are finished, in new_Classifier.py, `class_sup_preprocess()` is called to calculate the number of times each class label appears in the training dataset and the size of the training dataset. Then, `classify()` is called to classify a certain data object. The algorithm of `classify()` is below:

```
1  rule_dic = {}
2  use Depth-First Search to visit all the rules in the CR-tree
3  for rule in CR-tree:
4       for attribute in rule:
5       if attribute not in data_object:
6             break
7  get class label from rule
8  rule_dic[label].append(rule)
9  if rule_dic is empty:
10      return class label with the largest count in the training dataset
11 class_weighted_x2_dic = {}
12 for label, rule_list in rule_dic.items():
13      weighted_x2 = 0
14      for rule in rule_list:
15            calculate weighted x^2 for the rule
16            weighted_x2 += weighted x^2 for the rule
17      class_weighted_x2_dic[label] = weighted_x2
18 return the class with the highest weighted X^2
```

Line 1-8 uses Depth-First Search to find all the rules whose attributes match the data object. Line 9-10 sets the default class label if no rule in the CR-tree matches the data object. The default class label is generated in the same way as CBA, which uses the class label with the largest count in the training dataset. Line 11-17 calculates weighted χ^2 values for every rule and adds them up for every class label. Finally, line 18 returns the class label with the highest weighted χ^2 value.

5.4 Results

For CMAR, the minimum support is 1% and the minimum confidence is 50%. The database coverage threshold is 4. We tested the 7 datasets using our self developed CMAR program and the accuracy results are below:

`	CMAR (Self Developed)			CMAR (Paper)
	no.of rules	generator runtime	accuracy	accuracy
glass	16	0.17s	33.2%	70.1%
wine	23	0.17s	55.0%	95%
iris	30	0.01s	52.0%	94%
pima	73	0.89s	64.9%	75.1%
tic-tac-toe	60	0.75s	65.3%	99.2%
caesarian	73	0.03s	60.6%	-
car	186	1.42s	70.0%	-

Unfortunately, our self-developed CMAR program did not perform well. The output accuracies for each dataset are lower than the results found in the paper. This happens because we did not fully understand the improved version of the rule mining process as proposed in the paper. The way of combining frequent patterns and their class labels could be improved to generate more useful rules.

However, we noticed that the CMAR rule generator took much less time than the CBA algorithm to generate rules from the preprocessed datasets. This may be because compact data structures, such as FP-tree and CR-tree, are used to store and access training datasets and mined rules. These structures helped reduce time and space required by computation significantly.

Conclusions & Reflections

In conclusion, by summarizing all the results we have obtained in part 2, 3, 4 and 5, we have the side-by-side comparison table as follows:

Dataset	Accuracy				
	Part 2 & 3	Part 4			Part 5
	CBA M2	Decision Tree	Random Forest	SVM	CMAR (Self Developed)
glass	97.7%	58.3%	75.3%	35.5%	33.2%
wine	98.7%	92.2%	97.2%	55.2%	55.0%
iris	100.0%	96.0%	96.0%	97.3%	52.0%
pima	99.9%	71.1%	77.3%	75.8%	64.9%
tic-tac-toe	100.0%	84.9%	89.9%	87.4%	65.3%
caesarian	100.0%	53.8%	57.5%	57.5%	60.6%
car	100.0%	90.3%	86.6%	71.1%	70.0%

As we can see, the best performing classifier is the Classification based Association Rule (CBA) M2 algorithm which obtains almost perfect accuracy for each dataset on average. The worst performing classifier however, is the Classification based on Multiple Association Rules algorithm we built in part 5, which unfortunately has an accuracy even lower than the Support Vector Machine. In theory, CMAR should be performing better or at least more or less on par with CBA due to its utilizing the combined effect of multiple rules for classification. However, the compact FP-tree structure used in CMAR made the rule mining and pruning processes more efficient. Thus, CMAR took much less time in rule mining, compared to CBA.

All in all, this project is deemed challenging to all of us as this is our very first attempt to interpret and transform the pseudocode in the paper into reality. Despite the unsuccessful attempt in further improving on the CBA M2 algorithm we built in part 2, we certainly learnt a lot throughout the process by gaining more knowledge about the FP-growth and Apriori algorithm.

Contribution

Name	Contribution
Chang Heen Sunn	CBA classifier and open source software classifier
Cao Shuwen	Data preprocessing and CBA classifier
Huang Runtao	CBA Rule Generator, CMAR Program
Yin Jia Rui	Never do anything

References

1. B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In KDD'98, New York, NY, Aug. 1998.

2. Wenmin Li, Jiawei Han and Jian Pei, "CMAR: accurate and efficient classification based on multiple class-association rules," Proceedings 2001 IEEE International Conference on Data Mining, 2001, pp. 369-376, doi: 10.1109/ICDM.2001.989541.

3. T. D. V. Swinscow, "Statistics at square one: The BMJ," *The BMJ | The BMJ: leading general medical journal. Research. Education. Comment*, 28-Oct-2020. [Online]. Available: https://www.bmj.com/about-bmj/resources-readers/publications/statistics-square-one. [Accessed: 20-Oct-2021].

4. Some of the codes we used to build the CBA algorithm in part 2 are referenced based on this github repo:

   https://github.com/Williano/Data-Mining/tree/b24247ff3cb8eb0227885dd27287d4dace7aa629/wine_data_mining_research/association_classification