NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

CZ4032 Data Analytics and Mining

Project 2 Technical Review Report

Team 15

Cao Qingtian (U2020646L)
Chen Xingyu (U2021140F)
Jacintha Wee Yun Yi (U2021987K)
Valencia Lie (U2020738C)

Part 1

Introduction

Classification Based Association Rules (CBA) is a type of a rule-based classifier where rules are generated and mined from the dataset using an extension of Apriori's algorithm before classification is performed. There are two types of primary algorithms of CBA: M1 and M2. Although M2 is more efficient in terms of computational time, M1 is more straightforward in terms of implementation. Due to the above reasons, this report will focus on the implementation and performance of CBA-M1 only. The original implementation of CBA-M1 is referenced from Bing Liu, Wynne Hsu, Yiming Ma:Integrating Classification and Association Rule Mining. KDD 1998: 80-86. This report will also compare the differences in performance when the Classification Association Rules (CARs) are pruned and when it is not pruned.

Some of the codes below are adapted and taken from https://github.com/liulizhi1996/CBA.
Here are the list of terminologies and its definitions that were used to explain the algorithm of CBA-M1:

Terminology	**Description**
D	dataset
y	Target class
item	Column/feature of a dataset
cond_set	Set of items, represented as {'item1':v1, 'item2': v2}
ruleitem	A rule in form of cond_set->y
condsupCount	Number of cases in D that contain the condset.
rulesupCount	Number of cases in D that contain the cond_set and are labelled with class y
support	(rulesupCount/ |D|)* 100%
confidence	(rulesupCount/ condsupCount) * 100%

Table 1: List of terminologies and its definitions

Rule Generation

- Candidate Generation

  We enumerate all pairs in F_k to check whether two ruleitem in F_k can be merged into a candidate of k+1 ruleitem, the procedure is shown in the pseudo code below:

```
Fk1 = []
for ruleitem1 in Fk:
    # if the two ruleitemss do not share same y, we cannot merge them
    if ruleitem1.y != ruleitem2.y:
        return
    # assign y grom either ruleitem because it is the same
    y = ruleitem1.y
    rule1_items = ruleitem1.cond_set.keys
    rule2_items = ruleitem2.cond_set.keys
    sorted_rule1_items = sorted(rule1_items)
    sorted_rule2_items = sorted(rule2_items)
    # if the two ruleitems have the same items,
    # we cannot merge them to generate k+1-ruleitem
    if rule1_items == rule2_items:
        return
    # check whether the first k-1 keys in sorted item lists are the same
    for i in range(len(sorted_rule1_items)):
        if sorted_rule1_items[i] != sorted_rule2_items[i]:
            if i != len(rule1_items)-1:
                return

    intersect = rule1_items ∩ rule2_items
    for i in intersect:
        # for every item in both ruleitems, if the item's values are different
        # in two ruleitems, we cannot merge them
        if ruleitem1.cond_set[i] != ruleitem2.cond_set[i]:
            return
    # after checking the above, merge the pair
    union = rule1_items ∪ rule2_items
    for u in union:
        if u in rule1_items:
            candidate_cond_set[u] = ruleitem1.cond_set[u]
        else:
            candidate_cond_set[u] = ruleitem2.cond_set[u]
    candidate_ruleitem = (candidate_cond_set, y)
    # if the merged candidate_ruleitem is not inside Fk1, add it
    if candidate_ruleitem not in Fk1:
        Fk1.append(candidate_ruleitem)
```

Figure 1: Pseudocode of Candidate Generation

- CAR

CARs are generated by checking the confidence of the ruleitem in the frequent itemset and those that have confidence higher than the minimum threshold are shortlisted as a possible ruleitem in the CAR set.

```
rule_item.confidence >= minconf:
```

Figure 2: Screenshot of confidence checking, taken from source code

However, if there exists a ruleitem in the CAR set with the same condset, only the ruleitem with the highest confidence is saved in the set and the other is discarded.

For example:
If {(A a, B b)}-> (C, y) is present in the current CAR set but there exists another ruleitem with the same condset: {(A a, B b)} -> (C, n) from the frequent item set which has a higher confidence than the previously stated rule item, the latter is added to the CAR set and the former is discarded. If there is a tie in terms of confidence, one ruleitem is selected at random.

```
for item in self.rules:
    if item.cond_set == rule_item.cond_set and item.confidence < rule_item.confidence:
        self.rules.remove(item)
        self.rules.add(rule_item)
        return
    elif item.cond_set == rule_item.cond_set and item.confidence >= rule_item.confidence:
        return
self.rules.append(rule_item)
```

Figure 3: Screenshot of generating CAR from frequent itemset, taken from source code

- Pruning of rules

  Pruning is performed using the pessimistic error rate-based pruning technique from the C4.5 algorithm. The aim of pruning is to remove excessive rules, thereby substantially eliminating the number of rules generated. It can also avoid overfitting small datasets. The term 'pessimistic' in pessimistic error refers to downsizing the decision tree size one rule at a time in the original C4.5 algorithm to obtain the lowest error, essentially checking for the worst-case scenario (higher error rate) and avoiding it (by removal of rules). In M1, a rule r is pruned if its error rate is higher than that of rule r', obtained by removing one condition from rule r's conditions. In short, if a rule with lesser conditions yields a smaller error rate, that rule will be favoured and the rule with one more condition will be removed.

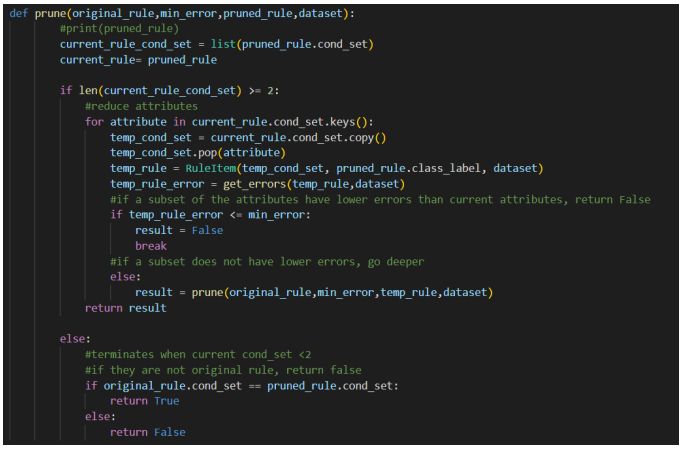


Figure 4: Screenshot of pruning CARs, taken from source code

The algorithm flow of the entire rule generation is as follows:

1. Frequent itemset 1 (F1) is obtained by eliminating those with support lower than the minimum support and CARs are generated from F1.

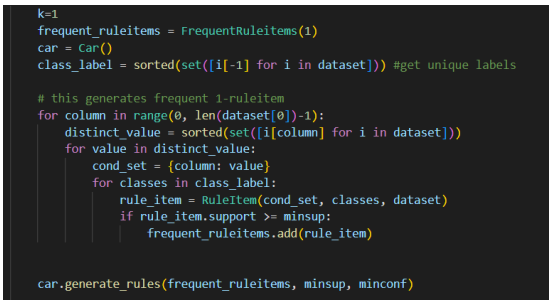


Figure 5: Screenshot of step 1 of algorithm flow, taken from source code

2. Next, candidates (C2) are generated from F1 in *generate_candidate*. We merge the two F_{k-1} items according to the Apriori algorithm, and then we generate the rules based on the candidates.
3. Then we generate rules based on the candidate, and add them to the Car object. The process stops once the number of CARs in the CARs set is more than 2000, or when there are no more frequent itemsets to be generated. If we are pruning, we set the upper limit of the k-frequent item to be generated at 6 to avoid long computational time, for the time complexity of pruning is 2^k for each k-frequent rule-item.

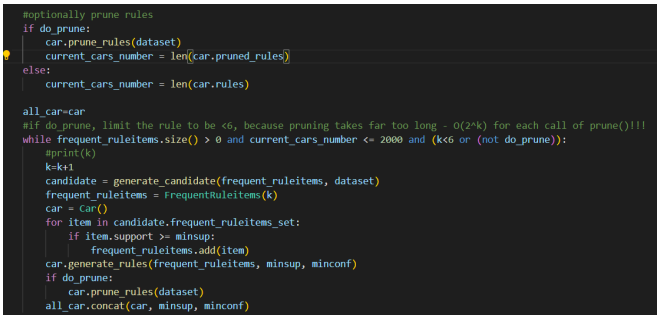


Figure 6: Screenshot of step 3 of algorithm flow, taken from source code

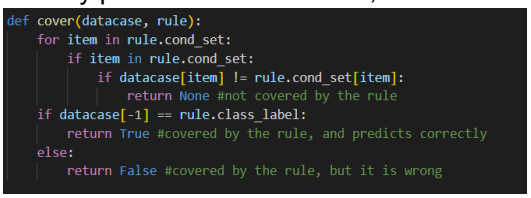
Classifier

First, we sort all rules generated through a *sort* function, by comparing confidence, followed by support (if confidence is the same), and then the order that it is generated(if both confidence and support are the same).

```
def sort(rule_list):
    def cmp_method(a, b):
        if a.confidence < b.confidence:
            return 1
        elif a.confidence == b.confidence:
            if a.support < b.support:
                return 1
            elif a.support == b.support:
                if len(a.cond_set) < len(b.cond_set):
                    return -1
                elif len(a.cond_set) == len(b.cond_set):
                    return 0
                else:
                    return 1
            else:
                return -1
        else:
            return -1
    rule_list.sort(key=cmp_to_key(cmp_method))
    return rule_list
```

Figure 7: Screenshot of sorting rules, taken from source code

Then, we go through the rules in sorted order, using a function *cover* to find cases covered by target rules. It has two uses: first, check if the data case's attribute set is contained as the rule. If it is not, it will return none, indicating that this data case is not covered by this rule. If it is covered, the function further detects whether the rule correctly predicts the class label, and returns True when it does.

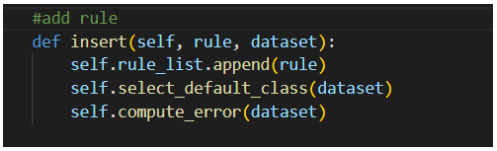


```
def cover(datacase, rule):
    for item in rule.cond_set:
        if item in rule.cond_set:
            if datacase[item] != rule.cond_set[item]:
                return None #not covered by the rule
    if datacase[-1] == rule.class_label:
        return True #covered by the rule, and predicts correctly
    else:
        return False #covered by the rule, but it is wrong
```

Figure 8: Screenshot of inspection of cases being covered by rules, taken from source code

If the *cover* function returns a value other than none, we add the data cases covered by it into a list named temp. If the rule is correct for at least one data case(namely, if *is_satisfy* returns True for at least 1 data case in the dataset D), we mark the rule. Then, we create a copy of the original dataset D, named temp_dataset. For all data cases that are in the list named temp, we take them out from the temp_dataset. In this way, we have a temp_dataset that the current rule fails to cover.

We then use the insert method within the Classifier class to add the rule in. This method takes in two arguments, namely rule and temp_dataset. It first adds the rule into the back of the rule_list attribute of the Classifier object, through a simple list append method. It also selects the default class for all the data cases in the temp_dataset, via *select_default_class*. The function just gives the most commonly used class labels in the temp_dataset. After that, this method uses *compute_error* to calculate errors for mis-classification for this classifier on the temp_dataset, using all the rules in the rule_list attribute of this classifier (in another word, using all rules that are in this classifier). Within the *compute_error* function, it also computes errors for the default class.The sum of errors made by all rules and the default class is all appended into an attribute of the classifier object (a list named error_list), where each value corresponds to the rule that is added in. (For instance, the fifth rule will have an error equal to the fifth element of the error_list).



```
#add rule
def insert(self, rule, dataset):
    self.rule_list.append(rule)
    self.select_default_class(dataset)
    self.compute_error(dataset)
```

Figure 9: Screenshot of rules insertion, taken from source code

After we consider all the rules, we use a discard method of the classifier class to remove the rules that make our classifier worse. We go through the error_list, find the minimum value, and discard all rules sharing the same index as errors after the minimum value. The classifier's default class will be re-assigned as the default class for the rule with the minimum value.

```
# remove useless rules
def discard(self):
    index = self.error_list.index(min(self.error_list))
    self.rule_list = self.rule_list[:(index+1)]
    self.error_list = None
    self.default_class = self.default_class_list[index]
    self.default_class_list = None
```

Figure 10: Screenshot of removal of some rules, taken from source code

Part 2

We used a train-test split with a ratio of 0.3, and the random state is set to 42. The 5 datasets chosen from the KDD'98 paper include the Australian, German, Iris, Tic-Tac-Toe, and Zoo datasets. The additional 3 datasets chosen from the UCI Machine Learning Repository are the Monks, Messidor, and Seeds datasets. The CBA-M1 algorithm was run on each of these datasets for both with and without pruning. The test accuracy is computed as 1 - error rate, and the classifier was timed on how long it takes to mine for the classification association rules and build the classifier.

Dataset	Description
Australian ('australian')	Financial data on credit card applications in Australia with 6 numerical and 8 categorical attributes. Used for binary classification.
German ('german')	Financial data on credit card applications in Germany with 7 numerical and 13 categorical attributes. Used for binary classification.
Iris ('iris')	Botanical data on the iris plant with 4 numerical attributes. Used for multiclass classification.
Monks ('monks')	Taken from the MONK's problem by Thrun et al with 6 categorical attributes. Used for binary classification.
Messidor ('messidor-features')	Medical data on diabetic retinopathy detection with 16 numerical and 3 categorical attributes. Used for binary classification.
Seeds ('seeds')	Botanical data on kernel geometrical properties of wheat with 7 numerical attributes. Used for multiclass classification.
Tic-Tac-Toe ('tic-tac-toe')	Game data on tic-tac-toe endgames with 9 categorical attributes. Used for binary classification.
Zoo ('zoo')	Life sciences data on attributes of various animals with 1 numerical and 14 categorical attributes. Used for multiclass classification.

Table 2: Names and descriptions of datasets chosen

Dataset	Without pruning: Accuracy	Without pruning: Run Time (s)	With pruning: Accuracy	With pruning: Run Time (s)
Australian	1.000	6.780	1.000	31.84
German	1.000	6.650	1.000	76.36
Iris	1.000	0.019	1.000	0.031

Monks	1.000	1.439	0.892	2.620
Messidor	1.000	30.89	1.000	4.316
Seeds	1.000	1.365	1.000	1.897
Tic-Tac-Toe	1.000	1.487	1.000	3.220
Zoo	1.000	26.65	0.903	3.787

Table 3: Accuracy and Run Time for CBA-M1 algorithm (with and without pruning) on the eight chosen datasets

The CBA-M1 algorithm produces very high test accuracies across the 8 datasets for both the prune and non-prune methods. Notably, majority of the test accuracies achieved is a perfect 1.0, with the Australian, German, Iris, Messidor, Seeds, and Tic-Tac-Toe datasets achieving perfect scores for both the prune and non-prune versions of the classifier. The worst test accuracy is using CBA-M1 with pruning on the Monks dataset with a score of 0.892, which is in itself still a high score. The reduction of pruning accuracy might be due to the limit we set on pruned rules, for 6-frequent rule items and above will not be considered in the pruning process, due to the large time complexity of the pruning algorithm (2^k for each k-frequent rule item). Nonetheless, the results show that the CBA-M1 algorithm is a very accurate classifier due to its ability to mine for associations between items and, by extension, capability of generating useful insights on data. Moreover, the algorithm works well on a variety of data with different compositions of data types such as numerical and categorical, as well as for binary and multiclass outputs. The CBA-M1 algorithm also runs very quickly for both pruning and non-pruning.

Part 3

We explored 4 other classification methods, Naive Bayes, Random Forest, Logistic Regression, and Support Vector Machine on the 8 datasets chosen in Part 2. Categorical string variables were encoded to integer values using scikit-learn preprocessing LabelEncoder function as part of a data preprocessing step before input to the respective models. A train-test split ratio of 0.3 and a random state of 42 was standardised across the 4 methods to generate data for training and validation.

Naive Bayes

The model we used, Naive Bayes, is from scikit-learn. Since there are a combination of categorical and numerical data cases in the dataset, we opt to change everything to categorical (i.e. classifying the numerical data into ranges, and use the label for ranges as a replacement value. This will be more in line with how CBA-RG handles the data). Since the values are now categorical, we use Multinomial object within the Naive Bayes model to do the classification, through the assumption of P(A,B,C)=P(A|B,C)*P(A|B)*P(C).

Random Forest

The Random Forest classifier from scikit-learn was used to perform classification. The performance of a base random forest classifier and a hypertuned one (using GridCV) are compared in order to ensure that the performance is maximised. Features were also examined and ranked according to their importance, with some less important features being eliminated for some datasets to increase its test accuracy (for example: Monks and Messidor). The parameter used for the base random forest classifier include n_estimator = 100 with a random seed of 42 and the range of parameters tuned using GridCV are as follows: 'n_estimators': [500, 600, 700], 'min_samples_split': [2, 5, 10], 'min_samples_leaf': [1,2,4], 'max_depth': [10, 20, 30].

Since there is randomness involved in the implementation of Random Forest, the experimentation is repeated 10 times, using seeds of [1, 12, 123, 1234, 12345, 123456, 1234567, 12345678, 123456789, 1234567890] and the results are averaged over the 10 runs. The best averaged results in terms of test accuracy and test F1 score are then documented in Table xx for each dataset.

Logistic Regression

The sci-kit learn LogisticRegression classifier was used to perform logistic regression on the datasets. The maximum number of iterations was set to 2500, and random state set to 42. Simple logistic regression is a linear regression function fitted with a sigmoid activation function at the end. As the sigmoid function outputs a value between a range of 0 and 1 inclusive, a threshold of 0.5 can be introduced to separate a prediction into one of two classes. Since logistic regression is originally meant for binary classification, for datasets with multiclass output (number of classes greater than 2) such as Iris, the multi_class parameter was set to 'multinomial' for multinomial linear regression. The logistic regression model was first fitted to the training data before prediction on the test data, after which accuracy and F1 score were calculated based on test performance.

SVM

The SVM (support vector machine) classifier from scikit-learn was used for this experiment. SVM is a binary classifier that aims to separate the dataset into two classes with a decision boundary that maximises the margins between the two classes.
SVM requires all variables to be numerical, hence, we encode categorical variables using LabelEncoder in scikit-learn, where each category of the variable will be given an index 0~n. In addition, since SVM only deals with binary classification, when a multi-class classification of k classes is encountered, it will be turned into k binary classifications of "belongs to class x" vs "does not belong to class x". To predict a data case, all k classifiers are run and the result would be a one-hot vector with the predicted class being the index of 1 in the vector. If there are multiple ones or no ones in the vector, we consider it as a mis-classification. Moreover, when the dataset is not linearly-separable, meaning a linear decision boundary cannot separate the dataset into two classes, a kernel trick is used to transform the dataset into higher dimensions so that it can be better separated by a linear line. In our experiments, we tried all possible kernels for each dataset and used the one that gives us the best result. The bracket after each accuracy score in the table below indicates the kernel used.

Results: Accuracy/ F1 (weighted average)

Dataset	Naive Bayes	Random Forest	Logistic	SVM
Australian	0.744/0.745	**0.871/0.870**	0.855/0.854	0.870/0.869 +
German	0.723/0.674	**0.766/**0.744	0.757/0.739	0.763/**0.751** +
Iris	0.556/0.733	**1.000/1.000**	**1.000/1.000**	0.978/0.989 ++
Monks	0.692/0.692	**1.000/1.000** *	0.715/0.716	0.938/0.938 ++
Messidor	0.624/0.624	0.710/0.710 **	**0.751/0.752**	0.749/0.748 +
Seeds	0.825/0.821	0.889/0.890	**0.921/**0.921	0.888/**0.930** +
Tic-Tac-Toe	0.670/0.538	0.938/0.936	0.691/0.639	**0.948**/**0.947** +++
Zoo	0.871/0.833	**0.935/**0.920	**0.935/0.921**	0.484/0.484 +++

* features eliminated: 'a3', 'a4' and 'a6'
** features eliminated: 'quality_assessment', 'pre-screening', 'result of the AM/FM-based classification'
+ linear
++ rbf
+++ poly

Table 4: Accuracy and F1 Score for the four classifiers on the eight chosen datasets

Comparing the performances of all four classifiers, it seems that Naive Bayes performed the worst, whereas the performance of SVM, Logistic Regression and Random Forest are overall quite comparable. However, the huge disparity in performance between all the classifiers mentioned above (Table 4) and CBA-M1 in part 1 (Table 3) proves how powerful CBA-M1 algorithm is in terms of classification when compared with other classifiers.

Part 4

Bagging on CBA-M1 was performed. We first perform a train-test-split with a ratio of 0.3, and the random state set to 42. The training data is used for sampling by the base estimators of the bagging ensemble. The BaggingClassifier from sci-kit learn was used to create the M1WithoutPrune and M1WithPrune BaseEstimator classes to perform bagging, with the number of base estimators set to 10 and random state set to 42. Each of the base CBA-M1 estimators were trained on a randomly sampled subset of the training dataset to generate rules based on this subset. A predictor function was implemented to obtain the predicted class by generating a cond_set for each instance of test data to be compared with the set of cond_set per rule generated by the CBA-M1 classifier during prior training. If all N cond_set of a rule from the classifier matches with N of the cond_set generated from the test data (where N <= the total number of cond_set of the test data), the corresponding class label is saved along with the rule. Should there be multiple compatible rules in the saved set, the majority class is taken as the output. If there is no compatible rule, the default class is returned. The final prediction for the bagging ensemble is decided by a majority vote across the 10 base estimators. The average test accuracy across the 10 base estimators per dataset is recorded in the table below.

Dataset	Without pruning: Accuracy	Without pruning: Run Time (s)	With pruning: Accuracy	With pruning: Run Time (s)
Australian	0.836	279.11	0.845	1728.98
German	0.703	351.16	0.697	1636.80
Iris	1.000	0.92	1.000	0.32
Monks	1.000	73.04	0.731	40.92
Messidor	0.645	1522.26	0.636	55.74
Seeds	0.810	71.17	0.841	30.27
Tic-Tac-Toe	0.799	73.40	0.691	38.48
Zoo	0.871	296.92	0.774	75.32

Table 5: Accuracy and run time of CBA-M1 with bagging on the eight chosen datasets

Overall, the run time for CBA-M1 with bagging is longer than CBA-M1 without bagging. This is because 10 base CBA-M1 estimators are instantiated and trained for bagging, hence training and prediction would take a longer time to complete. Moreover, there is a drop in performance using bagging on top of CBA-M1 as compared to using CBA-M1 alone for both with and without pruning. Intuitively, this drop in average performance can be explained by the loss of information as a result of subsampling, and in some instances, reusing copies of a certain data point in the sample set due to sampling with replacement. Such a way of sampling data does not allow the base estimator to learn from a full set worth of information, thus introducing bias into the learning of the individual base estimators. The bias adds up when the results from all 10 classifiers are aggregated. Hence, bagging can be explained as a bias-variance tradeoff; allowing some bias from the way data is subsampled for each classifier, while reducing variance due to aggregation. In our case, as the CBA-M1 algorithm is sensitive to the attributes of the data presented to it in order to mine for patterns representative of each class instance, it is understandable that bagging would hurt its performance mainly due to the data subsampling process.

Individual Contribution Claims:

1. Cao Qingtian: Code and Report
2. Chen Xingyu: Code and Report
3. Jacintha Wee Yun Yi: Code and Report
4. Valencia Lie: Code and Report