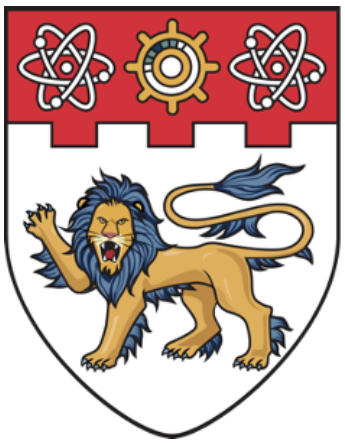

NANYANG TECHNOLOGICAL UNIVERSITY
SINGAPORE

# School of Computer Science & Engineering
# CZ4032 Data Analytics & Mining

Report Assignment 2

**Team Members**:
U2021469L Ng Yue Jie Alphaeus
U2022121D Gregory Wong Chun Jie
U2021490J Nguyen Viet Hoang
U2020797L Magdeline Ng Xuan Lynn
U2020331A Truong Vinh Khai

*Page count excludes Cover Page, Content Page and Reference*

# Content Page

# Introduction

For this project, our team implemented and evaluated a framework that integrates Association in the context of Classification, known as Classification Based on Associations (CBA). The framework consists of three steps:

1. Discretize continuous attributes.
2. Generate all Class Association Rules (CARs).
3. Build a classifier based on generated CARs.

After generating the rules and building the classifier, we compared the performance of the CBA algorithm with other Machine Learning classification methods. The performances of all models were compared, and we conclude that the integration of Association in Classification offers an effective method to solve real-world classification problems.

# Classification Based on Association rules (CBA) algorithm implementation

The CBA algorithm consist of 2 parts:

1. Rule Generation phase
2. Classifier building phase

## Rule Generation

In this phase, we started by defining a value of minSup and minConf. For these values, we followed the paper closely choosing a value of ***minSup = 0.01*** and ***minConf = 0.5***.

```
1   F1 = {large 1-ruleitems};
2   CAR1 = genRules(F1);
3   prCAR1 = pruneRules(CAR1);
4   for (k = 2; Fk-1 ≠ ∅; k++) do
5       Ck = candidateGen(Fk-1);
6       for each data case d ∈ D do
7           Cd = ruleSubset(Ck, d);
8           for each candidate c ∈ Cd do
9               c.condsupCount++;
10              if d.class = c.class then c.rulesupCount++
11          end
12      end
13      Fk = {c ∈ Ck | c.rulesupCount ≥ minsup};
14      CARk = genRules(Fk);
15      prCARk = pruneRules(CARk);
16  end
17  CARs = ∪k CARk;
18  prCARs = ∪k prCARk;
```

*Figure 1: Pseudocode for Stage 1 of CBA Algorithm*

A rule item is of the form <condset, y>, where condset is a set of items and y is a class label. We obtained the frequent itemset (Line 1) by counting the support for each rule item. If the support of the rule item is higher than *minSup*, we then generate a base list of CARs by adding it to the list F1.

In Line 2, we performed candidate generation, creating a power set from the list of frequent item sets and used this set for the subsequent runs of the algorithm. The frequent item set only gets added to the list of CARs if it satisfies the condition of a frequent item set (frequent), and if the confidence exceeds the *minConf* value (accurate). If two rules have the same condset, the rule with the higher confidence is added to the list of CARs. Hence, CAR consists of all rules that are frequent and accurate.

We repeated the candidate generation process and extracting of CARs as long as the number of CARs does not exceed 5000. So our maximum amount of possible CARs generated per dataset is 5000. The end result of this step will give you a set of CARs that can be used in the next phase to build a classifier.

Line 3 performs optional pruning, by looking through the infrequent subset of items from the set and removing the power set of those items from the set of rules.

For subsequent passes (lines 6-12), the seed set of rule items found to be frequent in the previous pass is used to generate new candidate rule items. In lines 8-10, the support counts for the condset and the rule item is incremented, to facilitate pruning and compute confidence of the rule item.

With the updated support counts (line 13), the new frequent rule items form Fk. At the end of the pass (line 14), the algorithm determines which of the candidate rule items are actually frequent to produce the CARs.

## Classifier Builder

In this phase, we built a classifier using the CARs generated in Stage 1 by selecting a subset of all the CARs.

```
1   R = sort(R);
2   for each rule r ∈ R in sequence do
3       temp = ∅;
4       for each case d ∈ D do
5           if d satisfies the conditions of r then
6               store d.id in temp and mark r if it correctly
                    classifies d;
7       if r is marked then
8           insert r at the end of C;
9           delete all the cases with the ids in temp from D;
10          selecting a default class for the current C;
11          compute the total number of errors of C;
12      end
13  end
14  Find the first rule p in C with the lowest total number
        of errors and drop all the rules after p in C;
15  Add the default class associated with p to end of C,
        and return C (our classifier).
```

*Figure 2: Pseudocode for Stage 2 of CBA Algorithm*

In line 1, we first performed rule sorting using the following criteria to choose the highest precedence rules for our classifier.

1. rule $A$ is ranked higher if confidence of rule $A$ is greater than that of rule $B$,
2. rule $A$ is ranked higher if confidence of rule $A$ is the same as confidence of rule $B$, but support of rule $A$ is greater than that of rule $B$,
3. rule $A$ is ranked higher if rule $A$ has a shorter antecedent (fewer conditions) than rule $B$.

*Figure 3: Criteria for rule sorting*

In lines 5-8, we mark a rule as long as it correctly covers a case in the dataset, as it can be a potential rule for our classifier. Those marked cases are then removed (line 9), and a default class is selected, which is the majority class in the remaining data.

We performed 10-fold cross validation when selecting the subset of rules, so each time the algorithm runs and calculates the total error of each classifier (Line 11). The mean error rate metric is then calculated from our 10-fold cross validation dataset.

# Improved implementation of CBA - Bagging

To try to improve the CBA algorithm, we continued building upon the unpruned CBA, and kept the 10-fold validation to find the average results. We chose unpruned over pruned as from the results we ran, we found both algorithms to have roughly the same metrics but unpruned CBA ran considerably quicker.

The dataset was first split into 10 equally sized partitions and 9 of the partitions was chosen to be the training dataset and the remaining partition to be the test dataset. Afterwards, bootstrapping (line 178) was performed on the training dataset, via random sampling with replacement on the 9 partitions. Then the majority of the predicted class of the 9 models was used to predict each row of data (line 194). The relevant metrics are stored and then the process is repeated 9-fold more times to find the average.

```
174         # Build 9 Models
175         for i in range(number_of_partitions-1):
176
177             # Split dataset randomly for bagging
178             subSample = getSubSamples(training_dataset, number_of_partitions-1)
179
180             # Train the model on this subsample
181             classifier = classifier_builder_m1(cars, subSample)
182             print("\nSub-Classifier", str(i) + ":")
183             classifier.print()
184             error_rate = getErrorRate(classifier, test_dataset)
185             error_total_rate += error_rate
186
187             # Check prediction for this model
188             pred_class.append(get_pred_only(classifier, test_dataset))
189
190             total_classifier_rule_num += len(classifier.ruleList)
191             # print("CBA-CB M1's run time with pruning: %f s" % cba_cb_runtime)
192
193         # Find the majority class guess from pred_class list,
194         pred = get_majority_predicted_class(pred_class)
195
```

*Figure 4: Code for bagging*

# Evaluation Method

To evaluate the performance of the CBA implementation and its improved version, we **compare them with 4 other classification methods**, namely Decision Tree, Random Forest, Support Vector Machine (SVM) and Neural Network.

For that purpose, 8 datasets are selected from the UCI Machine Learning repository, of which 5 were used in the KDD'98 paper. For each dataset, we repeatedly carry out the classification task, each time using one method from the list of 6 methods mentioned. The classification result of each run would be evaluated using 5 metrics: Mean Execution Time in ms, Mean Error Rate, Mean Macro-averaged Precision (Macro Precision), Mean Macro-averaged Recall (Macro Recall), and Mean Macro-averaged F1-score (Macro F1-score). All metrics are obtained from 10-fold cross-validations. The experiment results are recorded in the tables in Results below.

## Datasets

We have selected 5 datasets (in italic) from the paper KDD'98, and another 3 different datasets from the same repository.

1. *Iris*[1] - A dataset used to predict the class of an iris plant from 3 different classes.
2. *Tic Tac Toe*[2] - A dataset used to predict possible configurations of tic-tac-toe game
3. *Glass*[3] - A dataset used to predict the type of glass
4. *Lymphography*[4] - A dataset used to predict the class of lymph
5. *Car Evaluation*[5] - A dataset used to test constructive induction and structure discovery methods.
6. Haberman[6] - A dataset used to predict the survival of patients who had undergone surgery for breast cancer
7. Breast Cancer Wisconsin[7] - A dataset on details of breast cancer obtained by the University of Wisconsin Hospitals
8. Abalone[8] - A dataset used to predict the age of abalone based on other characteristics

[1] https://archive.ics.uci.edu/ml/datasets/iris
[2] https://archive.ics.uci.edu/ml/datasets/Tic-Tac-Toe+Endgame
[3] https://archive.ics.uci.edu/ml/datasets/glass+identification
[4] https://archive.ics.uci.edu/ml/datasets/Lymphography
[5] https://archive.ics.uci.edu/ml/datasets/Car+Evaluation
[6] http://archive.ics.uci.edu/ml/datasets/haberman%27s+survival
[7] https://archive.ics.uci.edu/ml/datasets/breast+cancer+wisconsin+(original)
[8] https://archive.ics.uci.edu/ml/datasets/Abalone

# Results

| Dataset | Mean Execution Time (ms) | | | | | | |
|---|---|---|---|---|---|---|---|
| | Unpruned CBA | Pruned CBA | Decision Tree | Random Forest | SVM | Neural Network | Improved CBA |
| *iris* | 30.664 | 87.360 | 5.840 | 162.166 | 5.869 | 2700000 | 83.845 |
| *tic-tac-toe* | 25815.581 | 46173.338 | 6.212 | 138.271 | 32.990 | 480000 | 41106.379 |
| *breast-w* | 86884 | 1179165 | 8.390 | 166.398 | 11.292 | 460000 | 31249 |
| *glass* | 39306 | 186408 | 6.465 | 156.933 | 9.945 | 398000 | 64885 |
| *lymph* | 23284 | 75925 | 6.839 | 128.178 | 6.145 | 285000 | 22440 |
| abalone | - | - | 31.197 | 433.057 | 1483.860 | 480000 | - |
| car | 13584 | 27781 | 8.892 | 155.940 | 144.381 | 720000 | 9618 |
| haberman | 12.881 | 25.349 | 6.042 | 149.583 | 17.558 | 210200 | 28.06 |

*Table 1: Mean Execution Time in ms of each algorithm*

| Dataset | Mean Error Rate (%) | | | | | | |
|---|---|---|---|---|---|---|---|
| | Unpruned CBA | Pruned CBA | Decision Tree | Random Forest | SVM | Neural Network | Improved CBA |
| *iris* | 5.3 | 5.3 | 4.0 | 16.2 | 2.7 | 2.5 | 10.0 |
| *tic-tac-toe* | 15.485 | 14.853 | 24.9 | 14.7 | 16.4 | 34.4 | 20.748 |
| *breast-w* | 3.32 | 3.32 | 37.5 | 32.0 | 8.4 | 7.7 | 5.47 |
| *glass* | 45.79 | 46.26 | 3.5 | 24.8 | 64.3 | 21.3 | 55.71 |
| *lymph* | 22.72 | 19.16 | 21.0 | 13.5 | 22.5 | 16.8 | 29.83 |
| abalone | - | - | 28.1 | 21.5 | 74.8 | 54.5 | - |
| car | 20.52 | 20.5 | 15.6 | 16.2 | 17.9 | 12.6 | 25.762 |
| haberman | 26.22 | 26.20 | 38.2 | 32.0 | 26.9 | 25.8 | 25.88 |

*Table 2: Mean Error Rate of each algorithm*

| Dataset | Mean Macro-averaged Precision | | | | | | |
|---|---|---|---|---|---|---|---|
| | Unpruned CBA | Pruned CBA | Decision Tree | Random Forest | SVM | Neural Network | Improved CBA |
| *iris* | 0.683 | 0.683 | 0.964 | 0.813 | 0.978 | 0.957 | 0.906 |
| *tic-tac-toe* | 0.850 | 0.850 | 0.514 | 0.866 | 0.856 | 0.666 | 0.780 |
| *breast-w* | 0.968 | 0.968 | 0.937 | 0.553 | 0.928 | 0.635 | 0.447 |
| *glass* | 0.408 | 0.401 | 0.638 | 0.747 | 0.061 | 0.788 | 0.455 |
| *lymph* | 0.651 | 0.675 | 0.704 | 0.766 | 0.627 | 0.844 | 0.32 |
| abalone | - | - | 0.720 | 0.787 | 0.096 | 0.488 | - |
| car | 0.502 | 0.502 | 0.810 | 0.813 | 0.758 | 0.789 | 0.342 |
| haberman | 0.419 | 0.417 | 0.530 | 0.553 | 0.367 | 0.725 | 0.419 |

*Table 3: Mean Macro-averaged Precision of each algorithm*

| Dataset | Mean Macro-averaged Recall | | | | | | |
|---|---|---|---|---|---|---|---|
| | Unpruned CBA | Pruned CBA | Decision Tree | Random Forest | SVM | Neural Network | Improved CBA |
| *iris* | 0.657 | 0.657 | 0.960 | 0.758 | 0.973 | 0.975 | 0.920 |
| *tic-tac-toe* | 0.773 | 0.776 | 0.748 | 0.837 | 0.795 | 0.578 | 0.698 |
| *breast-w* | 0.961 | 0.961 | 0.516 | 0.547 | 0.894 | 0.616 | 0.521 |
| *glass* | 0.283 | 0.279 | 0.670 | 0.733 | 0.170 | 0.791 | 0.44 |
| *lymph* | 0.654 | 0.686 | 0.705 | 0.768 | 0.625 | 0.832 | 0.425 |
| abalone | - | - | 0.719 | 0.786 | 0.101 | 0.496 | - |
| car | 0.519 | 0.525 | 0.762 | 0.758 | 0.693 | 0.789 | 0.45 |
| haberman | 0.506 | 0.500 | 0.537 | 0.547 | 0.498 | 0.734 | 0.507 |

*Table 4: Mean Macro-averaged Recall of each algorithm*

| | Mean Macro-averaged F1-score |
|---|---|

Dataset

| | Unpruned CBA | Pruned CBA | Decision Tree | Random Forest | SVM | Neural Network | Improved CBA |
|---|---|---|---|---|---|---|---|
| *iris* | 0.669 | 0.669 | 0.960 | 0.756 | 0.973 | 0.958 | 0.900 |
| *tic-tac-toe* | 0.796 | 0.798 | 0.734 | 0.839 | 0.808 | 0.578 | 0.702 |
| *breast-w* | 0.963 | 0.963 | 0.498 | 0.524 | 0.905 | 0.623 | 0.452 |
| *glass* | 0.320 | 0.312 | 0.628 | 0.711 | 0.089 | 0.771 | 0.447 |
| *lymph* | 0.646 | 0.671 | 0.700 | 0.761 | 0.614 | 0.812 | 0.339 |
| abalone | - | - | 0.719 | 0.785 | 0.090 | 0.452 | - |
| car | 0.500 | 0.503 | 0.737 | 0.756 | 0.675 | 0.765 | 0.385 |
| haberman | 0.433 | 0.455 | 0.514 | 0.524 | 0.422 | 0.785 | 0.436 |

*Table 5: Mean Macro-averaged F1-score of each algorithm*

| Dataset | **Number of CAR** | | | **Number of Rules in C** | | |
|---|---|---|---|---|---|---|
| | Unpruned CBA | Pruned CBA | Improved CBA | Unpruned CBA | Pruned CBA | Improved CBA |
| *iris* | 119 | 45 | 119 | 9 | 9 | 8 |
| *tic-tac-toe* | 4467 | 3471 | 4427 | 32 | 31 | 21 |
| *breast-w* | 8520 | 7419 | 5145 | 56 | 53 | 53 |
| *glass* | 6826 | 911 | 6809 | 44 | 44 | 31 |
| *lymph* | 11582 | 1766 | 11584 | 48 | 47 | 36 |
| abalone | - | - | - | - | - | - |
| car | 892 | 510 | 892 | 36 | 36 | 27 |
| haberman | 43 | 42 | 43 | 14 | 14 | 12 |

*Table 6: Number of CARs*

In terms of accuracy, CBA's performance is comparable to those of other classification methods, as can be seen on iris, tic-tac-toe, lymph, and haberman datasets. CBA with or without pruning offers almost identical results. On breast-w, CBA performs significantly better, while on glass and car, CBA performs worse. On abalone, CBA's rule generator did not generate any rules, making it unable to build the classifier, hence failing to carry out the

classification task. This is likely due to the nature of the classification target variable being numerical and continuous. Hence we observed that CBA only works on datasets with discrete classification target variables.

In most cases, CBA with Bagging does not improve the accuracy of the CBA algorithm. However, for iris and glass, CBA with Bagging does produce better Precision, Recall, and F1-score than non-bagging CBA implementations. This could be due to the clustering of instances of the same class in the dataset, which makes bagging effective in randomizing the train and test set in the 10-fold cross-validation process, hence, helping the rule generator to generate more helpful rules.

However, we note that the performance of our CBA implementation is correlated with the upper bound of the total number of possible candidate rules in 1 iteration of CBA-RG, m. As compared to CBA-CB, CBA-RG is the algorithm's bottleneck. The higher m is, the longer CBA-RG takes to finish execution. To ensure the algorithm finishes running in a reasonable time period while not compromising accuracy, m = 5000 is chosen. However, for many of the datasets, the size of the candidate rules in 1 iteration can be well above 5000, meaning the algorithm may have missed out on some important CARs to aid it in the classification task. This is observed as we tested with m = 8000, m = 10000 on some of the datasets and achieved significantly better results.

In terms of speed, unpruned CBA is comparable to other methods for datasets with few attributes (iris, haberman). The execution time for unpruned CBA then grows exponentially with increasing number of attributes in the dataset, making it inefficient. Since pruned CBA have to recursively prune the generated rules in addition to carrying out operations of unpruned CBA, its execution time usually more than doubles that of unpruned CBA. The execution time of CBA with Bagging (without pruning) varies, but is usually between the execution time for unpruned CBA and pruned CBA.

# Conclusion

Classification Based-on Association rules (CBA) is an effective classification algorithm, producing fast and accurate results on datasets with a small number of attributes. The algorithm execution time as well as the size of the candidate rule set in each iteration of CBA-RG scales exponentially with the number of attributes in datasets.

# References

Liu B., Hsu W., & Ma Y. (1998). Integrating Classification and Association Rule Mining - AAAI. https://www.aaai.org/Papers/KDD/1998/KDD98-012.pdf. Retrieved November 10 ,2022.

# CZ4032 Member Contribution summary

| **Task** | **Members** |
|---|---|
| Research on datasets | All members |
| Implementation of Neural Networks code | Khai |
| Implementation of CBA code | Alphaeus, Gregory, Hoang |
| Implementation of SVM code | Hoang |
| Implementation of Decision Tree and Random Forest code | Magdeline |
| Report | All members |
| Video | All members |