Project 1 – Association Rule Mining & Classification

CZ4032 Data Analytics & Mining

Name	Matric No	Contribution
Dandapath Soham	U1822646A	25%
Gupta Jay	U1822549K	25%
Kanodia Ritwik	U1822238H	25%
Mundhra Divyesh	U1822168E	25%

School of Computer Science & Engineering

Nanyang Technological University, Singapore

Part 2: Implementation

(A) Mining Class Association Rules:

Pseudocode of our implementation for Rule Generation:

```
[1] Generate all association rules from the dataset
[2] Convert the association rules to Class Association Rule (CAR) format
[3] Build classifiers using CARs
```

In [1], we use the *Apriori* algorithm to generate all association with support and confidence greater than the threshold (Frequent itemset mining). As stated in the paper, we use the support threshold as 1% and confidence threshold as 50%.

In [2], the association rules is generated in the format:

(class_label, (feature1, feature2, . . ., featureN), support, confidence)

We need to convert these association rules to the format of a class association rule (CAR) as follows:

{ feature1, feature2, . . ., featureN} => {Class} sup: support, conf: confidence, len: N+1, id:id

The features form the Antecedent (LHS) and the class label forms the consequent. Now the M1 and M2 classifier can be built using these rules in CAR format.

Few things to note here,

1. The Rule Generation algorithm in the KDD'98 paper is based on the *Apriori* algorithm along with iterative pruning of rules based on pessimistic error. The author however, clearly states it is **optional**. For ease of implementation, we are not doing any pruning.
2. We are using the *Apriori* algorithm directly instead of the Rule Generation algorithm. We use a package *pyFim* for generation of the association rules. Implementing *Apriori* from scratch did not seem feasible given the time constraints. Our rule mining algorithm is inspired by *PyArc* (Appendix A) repository. As a result, the number of rules generated is high but the final number of rules in the classifier is comparable to the results in the paper.

(B) Explain Pseudocode of classifier in our implementation based on KDD'98 Paper

M1 Algorithm

The M1 algorithm implemented in the KDD'98 paper has been broken down into three main steps. In our code too, we provide implementation for the three steps. Let's take a look at them one by one:

Step 1: Sort the rules based on their precedence

Two rules *r1* and *r2* are compared as follows to determine precedence:

1. If confidence of *r1* is greater than *r2*, *r1* takes precedence
2. If their confidence are the same, but the support of *r1* is greater than *r2*, *r1* takes precedence.
3. If their confidence and support are the same, but the ID of *r1* is less than *r2*, *r1* takes precedence.

In the KDD'98 paper, the third condition states that the rule which is generated earlier takes precedence. We implemented this by comparing the IDs of the rule. The rule which is generated earlier will have a lower ID.

For implementation in Python, we used the built-in *sort()* function but we override it to compare two elements using a custom function which compares *r1* and *r2* based on the above criteria and returns *True* if *r1* takes precedence else returns *False*. This implementation was inspired by *PyArc* (Appendix A).

Step 2: Find the classifier rules, default class and their errors

Pseudocode for our implementation:

```
[1] total_errors = []
[2] For each rule in rules:
[3]         temp = set()
[4]         for each data in dataset:
[5]                 if rule correctly classifies data:
[6]                         temp.add(data)
[7]                         rule.marked = True  // rule is able to classify at least one data
[8]         if rule is marked:
[9]                 Add the rule to the classifier
[10]                Remove the data cases in temp (covered by rule) from dataset
[11]                Create a frequency hash map for the class labels in the remaining dataset
[12]                Get class label with highest frequency and choose it as default class for classifier
[13]                Compute the total number of errors of the classifier up to this rule and default class
[14]                Add this to total_errors[]
```

In line [5], a rule is correctly able to classify a data if the consequent of the rule equals to the class label of the data.

In step [13], the rule error is calculated by the number of data points having a class label different than the rules consequent and the default class error is calculated by the number of data points having class label not equal to default class. Our implementation for this step is very similar to the KDD'98 paper. It was relatively easier to implement in Python.

Step 3: Discard rules in classifier that do not improve the accuracy

In the previous step, we checked if a rule was able to classify even one data case. If it did, we added it to the classifier and **then** calculated the classifier's error up to that rule. We stored this error in a list called *total_errors*. Afterwards, we simply find the rule at which the errors was minimum. We call this rule the ***threshold_rule*** and discard all rules that were added to the classifier after this as they do not contribute in reducing the error.

Pseudocode for our implementation:

```
[1] min_error = min(total_errors), error corresponding to threshold_rule
[2] threshold_index = total_errors.index(min_errors), index corresponding to threshold_rule
[3] Discard all rules that come after the threshold_index
[4] Set default_class as class of threshold_rule
[5] Create a classifier object with the final rules and default class
[6] Return classifier object
```

After step 3, we have our final classifier which can be used for classification.

M2 Algorithm

Stage 1: Pseudocode for our implementation:

```
[1] Declare three sets Q, A and U
[2] for each data in dataset:
[3]         Cc = rules with consequent same as data label
[4]         Cw = rules with consequent different from data label
[5]         cRule = maxCoverRule(Cc, data)
[6]         wRule = maxCoverRule(Cw, data)
[7]         U.add(cRule)
```

```
[8]     cRule.class_cases_covered[data.class]++
[9]     If the precedence of cRule is higher than wRule:
[10]            Q.add(cRule)
[11]            cRule.marked = True
[12]    else A.add(data, data.class, crule, wrule)
```

As described in the KDD'98 paper here is the description of the variables and data structures:

- **cRule**: Highest precedence rule correctly classifying data
- **wRule**: Highest precedence rule wrongly classifying data
- **Cc**: All rules with LHS (Consequesnt) same as the class label of the data
- **Cw**: All rules with LHS (Consequesnt) different than the class label of the data
- **Q**: set of *cRules* with higher precedence than their corresponding *wRule*
- **U**: set of all *cRules*
- **A**: store *(data, class_label, cRule, wRule)* whenever there is a conflict, i.e., *wRule* has higher precedence than *cRule*. We handle these cases in Stage 2.

In lines [3] and [4], we iterate through all rules and store rules with the same (different) class as data in *Cc (Cw).*

In lines [5] and [6], in the *maxCoverRule()*, since we had already sorted the rules based on precedence for M1 algorithm, we just assign the first rule in *Cc (Cw)* which would correspond to the highest precedence rule with consequent same (different) as data's class to *cRule (wRule)*. In each element of A, we simply store the data itself instead of storing only the ID as specified in the paper.

Stage 2: Pseudocode for our implementation:

```
[1] for each conflict in A:
[2]     datacase, class_label, cRule, wRule = conflict
[3]     if wRule.marked:
[4]             crule.class_cases_covered[class_label] --
[5]             wrule.class_cases_covered[class_label] ++
[6]     else wSet = allCoverRules(U, datacase, cRule)
[7]             for w in wSet:
[8]                     w.replace.add((cRule, datacase, class_label))
[9]                     wrule.class_cases_covered[class_label] ++
[10]            Q = Q.union(wSet)
```

The stage 2 is the 'slightly more than one pass over the dataset' that M2 algorithm needs. The main purpose is to resolve conflicts that took place in stage 1 where a higher precedence rule was present which classified the data incorrectly. The implementation is standard and the algorithm is inspired from the KDD'98 paper.

Stage 3: Pseudocode for our implementation:

```
[1] classfreq = frequency map of class labels in dataset
[2] Q = sort(Q)
[3] for each rule in Q:
[4]     if rule.class_cases_covered > 0:
[5]             for each (rule_replace, data, class_label) in rule.replace:
[6]                     if data.alreadycovered = True:
[7]                             rule. class_cases_covered[class_label]--
[8]                     else data.alreadycovered = True
[9]                             rule. class_cases_covered[class_label –
[10]            rule_errors += errors_of_rule(rule)
[11]            classfreq = update class freq by removing cases covered by rule
[12]            default_class = most frequent class in classfreq
```

```
[13]            default_errors = dataset_len — default_class_count
[14]            total_errors = rule_errors + default_errors
[15]            rule_list. add(rule)
[16]            default_class_list.add(default_class)
[17]            total_errors_list.add(total_errors)
[18] Find the min error from total_error_list corresponding to threshold rule
[19] Find corresponding threshold rule from rule_list and discard rules after it
[20] Set class label of threshold rule as default class of classifier
[21] Return classifier
```

Here [18], [19], [20], [21], similar to what we do in step 3 of M1 algorithm to discard unnecessary rules. The implementation for rule error calculation (*errors_of_rule()*) and finding the default class is also very similar to step 2 of M1 algorithm. The initial steps are standard and inspired by the KDD'98 paper.

The runtime for our implementation was much higher than that of the KDD'98 paper but the results were very similar. The M2 algorithm does offer a minor speed up over the M1 algorithm in the datasets chosen by us.

Part 3: Algorithm Evaluation on Selected Datasets

Datasets (obtained from UCI Machine Learning Repository)

- **breast-w** – Original Wisconsin Breast Cancer Database, from Dr Wolberg clinical cases.
- **iris** – Pattern recognition database containing 3 types of iris plants.
- **pima-diabetes** – Prediction of onset of diabetes, based on diagnostics
- **heart** – Cleveland database referring to the presence of heart disease in a patient.
- **tic-tac-toe** – Binary classification task on possible configurations of tic-tac-toe games.
- **acute-inflammations (additional)** – Presumptive diagnosis of two diseases in the urinary system.
- **balance-scale (additional)** – Psychological experiments based on a balance scale.

Results of CBA based on KDD'98 Paper

We have implemented a CBA Classifier based on the KDD'98 paper in Python. The QCBA algorithms needs continuous attributes to be discretized and hence we have used the discretization method mentioned in the KDD' 98 paper for all of our datasets. For discretization of the data points, we have used the MDLP library (MDLP Package) provided in Python which implements the entropy based discretization methods by Fayyad and Irani, 1993 (Entropy based discretization) . In our observation, the results of our implementation closely resemble those of the original paper, with deviation in results of less than 3%, on the five chosen datasets. The mean error rates are reported with *minsup = 1%* over 10 cross-validations, and rules generated are not subjected to pruning. In addition, the algorithm is ran on two other datasets selected from the UCI Repository.

Dataset Source	**Datasets**	**Our Implementation**			**KDD'98 Paper Results**		
		Mean Accuracy w/o pru.	**Mean Error Rate (CARs)**	**No. of Rules in C**	**Mean Accuracy w/o pru.**	**Mean Error Rate (CARs)**	**No. of Rules in C**
KDD'98 Paper	breast-w	97.9	2.1	48	95.8	4.2	49
	iris	92.7	7.3	10	92.9	7.1	5
	pima-diabetes	77.4	22.6	65	72.4	27.4	45
	heart	87.4	12.6	60	81.5	18.5	52
	tic-tac-toe	99.0	1.0	8	100.0	0.0	8
Additional	balance-scale	74.0	26.0	13	-	-	-
	acute inflammations	91.7	8.3	4	-	-	-

Table 1: Results of Implementing CBA

Part 4: Comparison with Other Classification Methods

Three other open-source classification methods with 10-fold cross validation are used to compare the classification accuracy of classifiers with the KDD'98 CBA-based classifier, namely Random Forest, Support Vector Machine (SVM), and Decision Tree. The algorithm is implemented using the *scikit-learn* library, with default parameters.

The KDD'98 CBA-based classifier outperforms all three other classifiers when the dataset is imbalanced, whereas the other classifiers give better accuracy in balanced datasets. For example, the classification accuracy of non-CBA based classifiers drop in the *breast-w*, *pima-diabetes*, and *tic-tac-toe* dataset, where the classes are distributed unequally with high variations.

Dataset Source	**Datasets**	**Scoring: Accuracy**		**Scoring: F1 Macro**	
		Mean Accuracy	**Mean Error Rate**	**Mean Accuracy**	**Mean Error Rate**
KDD'98 Paper	breast-w	97.2	2.8	97.1	2.9
	iris	94.7	5.3	93.0	7.0
	pima-diabetes	76.4	23.6	73.6	26.4
	heart	81.6	18.4	82.4	17.6
	tic-tac-toe	95.0	5.0	93.5	6.5
Additional	acute-inflammations	100.0	0.0	100.0	0.0
	balance-scale	82.9	17.1	60.2	39.8

Table 2: Results of **Random Forest** Classifier on Selected Datasets

Dataset Source	**Datasets**	**Scoring: Accuracy**		**Scoring: F1 Macro**	
		Mean Accuracy	**Mean Error Rate**	**Mean Accuracy**	**Mean Error Rate**
KDD'98 Paper	breast-w	97.2	2.8	96.9	3.1
	iris	96.0	4.0	95.3	4.7
	pima-diabetes	76.0	24.0	70.5	29.5
	heart	63.8	36.2	61.2	38.8
	tic-tac-toe	89.4	10.6	87.2	12.8
Additional	acute-inflammations	42.5	57.5	34.5	65.5
	balance-scale	90.3	9.7	62.6	37.4

Table 3: Results of **Support Vector Machine** Classifier on Selected Datasets

Dataset Source	**Datasets**	**Scoring: Accuracy**		**Scoring: F1 Macro**	
		Mean Accuracy	**Mean Error Rate**	**Mean Accuracy**	**Mean Error Rate**
KDD'98 Paper	breast-w	95.3	4.7	94.0	6.0
	iris	94.0	6.0	92.4	7.6
	pima-diabetes	69.3	30.7	67.0	33.0
	heart	77.9	22.1	75.1	24.9
	tic-tac-toe	86.6	13.4	85.4	14.6
Additional	acute-inflammations	100.0	0.0	100.0	0.0
	balance-scale	77.9	22.1	57.2	42.8

Table 4: Results of **Decision Tree** Classifier on Selected Datasets

Part 5: Improvement over the CBA Algorithm

Implementing the QCBA Algorithm

QCBA (Qualitative CBA) aims to reduce the loss that is incurred during the discretization of the numerical values in the pre-processing stage of CBA. It tunes the rules generated from the CBA algorithm and further prunes the rule set thus reducing the size of the classifier. This implementation was inspired by *PyArc* (Appendix A).

Step 1 : Refit the rules for a stricter boundary on the item sets

```
[1] For each rule in rules:
[2]     For each literal in rule.antecedent :
[3]         Compute the range of discrete values ∈ literal
[4]         Update the range of the literal in the antecedent
[5] Return the refitted rule set
```

Refitting the rules results in the literals present in the antecedent to cover smaller region/interval based on the classification class i.e., the consequent of the rule.

Step 2 : Prune the literals from the antecedent of each rule

```
[1] For each rule in rules:
[2]     While a literal is removed from the antecedent of the rule:
[3]         For each literal ∈ rule.antecedent:
[4]             cand.antecedent = rule.antecedent\literal
[5]             calculate the confidence of cand rule
[5]             if cand.conf > rule.conf:
[6]                 rule.antecedent = cand.antecedent
[7] Return the pruned rule set
```

This steps results in removal of literals from each rule which does not improve the performance of the classification model.

Step 3 : Trim the boundary to remove the regions with no correct instances covered

```
[1] For each rule in rules:
[2]     Compute the rows covered by rule
[3]     For each literal in rule.antecedent :
[4]         Compute the min and max for rule[literal]
[5]         Update the interval with range(min, max)
[6] Return the new rule set
```

This step is similar to the first step where for each literal the range of the literal covering each rule is updated.

Step 4 : Extend the coverage of the literals to improve or maintain the current conf

```
[1] For each rule in rules:
[2]     For each literal in rule.antecedant:
[3]         Extend the range of the literal's interval to form cand rule
[4]         Compute the cand.confidence
[5]         if cand.conf > min conf and cand.support > min supp:
[6]             update rule's literal
[7] Return the rule set
```

The literal range are extended to improve the accuracy. If the accuracy does not improve, then the rule is kept as is without any extension.

Step 5: Data Coverage pruning and default rule calculation

```
[1] Sort the rules according to the cba definitions
[2] For each rule in rules:
[3]     if rule does not classify any instance correctly:
[4]         remove the rule
[5]     calculate the default class from the cba algorithm
[6]     update the default class based on the cut off
[7] Return the rule set
```

The data coverage pruning reduces the number of CARs which do not correctly classify and instance. It also calculates the default class which is calculated based on the cut off rule defined in the CBA method.

Step 6 : Prune the last rule if it has the same class in the consequent as the default class

```
[1] Iterate backward rules till rule.consequent != default_class.consequent
[2]     if rule.consequent == default_class.consequent:
[3]         remove the rule
[4] Return the rules
```

Results & Discussion on the QCBA Algorithm

Dataset Source	**Datasets**	**CBA**		**QCBA**	
		Accuracy	**Error Rate**	**Accuracy**	**Error Rate**
KDD'98 Paper	breast-w	97.65	2.35	98.24	1.76
	iris	95.33	4.67	99.33	0.67
	pima-diabetes	73.04	26.96	74.34	25.66
	heart	89.76	10.24	86.13	13.87
	tic-tac-toe	74.84	25.16	74.94	25.06
Additional	acute-inflammations	99.17	0.83	100.0	0.0
	balance-scale	81.5	18.5	81.5	18.5

Table 5: Results of Implementing QCBA Algorithm

From the above table we observe that our QCBA implementation gives better results for most of the datasets compared to the CBA results. Among the seven datasets we have used, QCBA gave significantly better results for *breast-w* and *iris* datasets and a slight increase in accuracy for *pima-diabetes* and *acute-inflammations* dataset. For *tic-tac-toe* and *balance-scale*, both the CBA and QCBA implementations give the same results and for the remaining *heart* dataset, QCBA gave worse results as compared to CBA. This is expected as the QCBA paper mentioned that their implementation doesn't guarantee better results than CBA in every case. So for some datasets the QCBA results can be comparable or even worse than CBA.

One thing to notice here is that the CBA results are slightly different from the results obtained in Table 1 (Part 3). To implement the 10-fold cross validation for CBA in Part 3, we use the discretization algorithm (Fayyad and Irani 1993) mentioned in the KDD'98 paper. On the other hand while implementing the QCBA algorithm, we use the '*cut*' function provided by the '*pandas*' library to distribute the continuous values into multiple bins. Hence, the two different discretization algorithms, lead to slightly different results in these two sections.

END OF REPORT

References

[1] Rakesh Agrawal, Tomasz Imieliński, and Arun Swami. 1993. Mining association rules between sets of items in large databases. SIGMOD Rec. 22, 2 (June 1, 1993), 207–216. DOI: https://doi.org/10.1145/170036.170072

[2] Bing Liu, Wynne Hsu, and Yiming Ma. 1998. Integrating classification and association rule mining. In Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining (KDD'98). AAAI Press, 80–86.

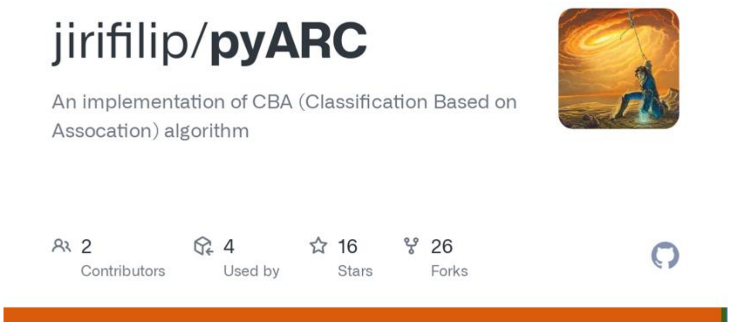
[3] Kliegr, Tomas. "QCBA: Postoptimization of quantitative attributes in classifiers based on association rules." arXiv preprint arXiv:1711.10166 (2017).

[4] Fayyad, Usama, and Keki Irani. "Multi-interval discretization of continuous-valued attributes for classification learning." (1993).

Appendix A

'pyArc' Repository by Jiří Filip and Tomáš Kliegr. Our implementation of CBA & QCBA algorithms in Python is referenced from this repository available on GitHub at this link: https://github.com/jirifilip/pyARC

A pre-print of the paper can be found here: https://easychair.org/publications/preprint/5d6G



jirifilip/**pyARC**

An implementation of CBA (Classification Based on Assocation) algorithm

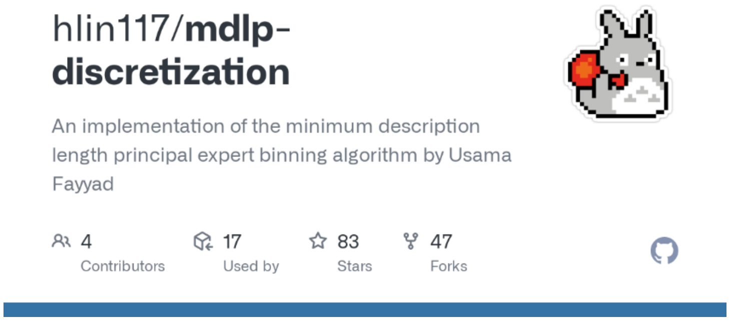
2 Contributors · 4 Used by · 16 Stars · 26 Forks

For experiments for other Machine Learning based methods such as Random Forrest, Decision Trees, and Support Vector Machine was ran on Kaggle Jupyter Notebook with supporting datasets on Kaggle and the UCI Machine Learning Repository.

Please upload the submitted Jupyter Notebook (.ipynb file) to Kaggle and initialize the datasets.

kaggle

'mdlp-discretization' Repository by Henry Lin. The data points in the datasets are discretized by an entropy based discretization method given by Fayyad and Irani, 1993 (Entropy based discretization).



hlin117/**mdlp-discretization**

An implementation of the minimum description length principal expert binning algorithm by Usama Fayyad

4 Contributors · 17 Used by · 83 Stars · 47 Forks