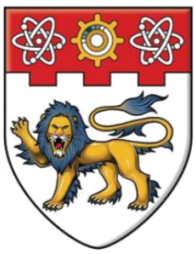

NANYANG TECHNOLOGICAL UNIVERSITY

School of Computer Science and Engineering

CX 4032: Data Analytics & Mining

Project 1 Report

Group Members:
Song Yu (U1922469E)
Lee Zhen Wei (U1922813B)
Zhao Peizhu(U1923719B)
Dong Yunxing(U1822594A)

Table of Content

1. Implementation of the CBA-CB M2 algorithm for classifier building

To reduce repetitive work for implementing the CBA-CB M2 algorithm, we decide to reuse the open-source Python project *Implementation of algorithm CBA (Classification Based on Associations)* (Liu et al., 2017), which provides a comprehensive implementation of essential data structures such as rule item & car, and algorithms in the paper, including CBA-RG, CBA-CB M1, and CBA-CB M2. In addition, it also implements a layer of data preprocessing, such as missing value imputation and discretization of continuous attributes to clean and unify the format of datasets available from the UCI Machine Learning Portal. Thus, it provides a solid foundation and framework that allow us to focus on the implementation of the CBA-CB M2 algorithm.

Particularly, we reuse the data structures including **RuleItem/Rule** (which represents ruleitems in the paper) and **Cars** (which represents the set of frequent ruleitems in the paper) designed by Liu. We also reuse the implemented CBA-RG algorithm to help generate CARs that we need to use for building the classifier and the driver functions in `cbaLib/validation.py` for testing result evaluation.

In our implementation of the CBA-CB M2 algorithm (the file is located at `cbaLib/cba_cb_m2_songyus_implementation.py`), we create a class for the classifier, which is used to instantiate classifier objects that have some common utility methods such as `discard_useless_rules()` and `get_error_rate()`. Below is a glance at this class:

```
class Classifier:
    def __init__(self):
        # Rule record is a tuple in the format (rule, default_class, total_errors)
        self.rule_records = []
        self.default_class = None
    def append_rule_record(self, new_rule_record):
        # add a new rule record that might contain the final candidate
    def set_default_class(self, new_default_class):
        # updates the default_class
    def discard_useless_rules(self): # corresponds to lines 18 - 20 in the paper
        # Find the first rule p in C with the lowest total_errors; Discard all the rules after p from C
    def get_error_rate(self, dataset):
        # get the error rate of this classifier given a dataset
```

We also create a function `build_m2()` which builds the classifier using the CBA-CB M2 algorithm. It consists of a series of utility functions such as max_cover_rule() used in the algorithm. We try to implement the algorithm as close to the pseudocode proposed in the paper as possible. The code structure and naming of variables/functions are made very similar, if not identical, to the original pseudocode to make it self-explanatory. Some modifications are introduced (e.g., we use the index to access rules in CARs and datacase in the dataset) to ensure the correct updates of variables of different data structures.

```
def build_m2(raw_cars, dataset):
    # util functions definitions #
    def max_cover_rule(cars_list, case):
        # finds the highest precedence rule that correctly/wrongly covers the case
        # returns the indexes of the cRule and wRule
    def all_cover_rules(U, cars_list, case, c_rule):
```

```
    # finds all the rules that wrongly classify the dID case and have higher
    # precedences than that of its cRule
def comp_class_distri(dataset):
    # counts the number of training cases in each class (line 1) in the initial training data.
def sortQ(q, cars_list):
    # sort the set of cRules by precedence. returns a list of sorted c_rule_idx
def update_class_distribution(rule, prev_class_distribution):
    # updates the distribution of class labels
def select_default(class_distribution):
    # get a default class i.e., the majority class in the remaining training data
def def_err(default_class, class_distribution):
    # get the number of errors that the default class will make
# Algorithm starts from here
classifier = Classifier()
# stage 1
Q, U, A = set(), set(), set()
for case in dataset:
    # lines 3-4 in stage 1
    c_rule_idx, w_rule_idx = max_cover_rule(cars_list, case)
    U.add(c_rule_idx)
    cars_list[c_rule_idx].class_cases_covered[case_class] += 1
    if cars_list[c_rule_idx].precedes(cars_list[w_rule_idx]):
        Q.add(c_rule_idx)
        cars_list[c_rule_idx].mark()
    else:
        A.add((case_id, case_class, c_rule_idx, w_rule_idx))
# stage 2
for record in A:
    case_id, case_class, c_rule_idx, w_rule_idx = record[0], record[1], record[2], record[3]
    if cars_list[w_rule_idx].get_mark_status():
        cars_list[c_rule_idx].class_cases_covered[case_class] -= 1
        cars_list[w_rule_idx].class_cases_covered[case_class] += 1
    else:
        w_set = all_cover_rules(U, cars_list, dataset[case_id], cars_list[c_rule_idx])
        for _rule_idx in w_set:
            cars_list[_rule_idx].add_replace_record((c_rule_idx, case_id, case_class))
            cars_list[_rule_idx].class_cases_covered[case_class] += 1
        Q = Q.union(w_set)
# stage 3
class_distribution = comp_class_distri(dataset)
rule_errors = 0
q_list = sortQ(Q, cars_list)
# A list to track id of covered cases
covered_cases_ids = []
for rule_idx in q_list:
    r = cars_list[rule_idx]
    if r.class_cases_covered[r.class_label] != 0:
        for entry in r.replace:
            entry_rul_idx, d_id, y = entry[0], entry[1], entry[2]
            if d_id in covered_cases_ids:
                cars_list[rule_idx].class_cases_covered[y] -= 1
            else:
                cars_list[entry_rul_idx].class_cases_covered[y] -= 1
        # update the covered_cases_ids list
        for case in dataset:
            if case not yet covered:
                case = None
                covered_cases_ids.append(case_idx)
        rule_errors += r.get_errors_in_dataset(dataset)
```

```
            class_distribution = update_class_distribution(r, class_distribution)
            default_class = select_default(class_distribution)
            default_errors = def_err(default_class, class_distribution)
            total_errors = rule_errors + default_errors
            classifier.append_rule_record((r, default_class, total_errors))
    # line 18 - 19
    classifier.discard_useless_rules()
    return classifier
```

2. CBA-CB M2(No Pruning) Classifier results

The results of our implementation of the CBA-CB M2 classifier without pruning are detailed as follows. The classifier is set up with the following values

Column 1: Lists the name of the 7 datasets
Column 2: Average error rate over 10 folds without pruning, minsup:0.01, minconf:0.5
Column 3: Number of rules generated (without pruning) for the given dataset
Column 4: Average time taken to generate rules with CBA-RG
Column 5: Average time taken to generate rules with CBA-CB M2 implementation execution
When compared with any instance of CBA-RG, M2 takes a shorter amount of time to complete running.
Column 6: Number of rules generated with the CBA-CB-M2 classifier.
Column 7: Classification Accuracy

Dataset	Avg Error Rate	CAR	**CBA-RG run time**	**CBA-CB M2 run time**	Number of rules	**Accuracy**
iris	3.3%	119	0.04s	0.01s	7	96.7
ionosphere	8.8%	2382	5.43s	0.75s	33	91.2
wine	9.6%	1598	41.77s	0.24s	29	90.4
zoo	6.0%	2464	2.0s	0.26s	10	94
heart	13.6%	586	2.94s	0.08s	8	86.4
car	13.2%	283	2.20s	0.53s	15	86.8
banknote_ authentication	6.1%	193	0.66s	0.36s	24	93.9
Average	8.66%	1089.2	5.43s	0.3s	18	91.3

CBA-CB M2 performs well on this dataset, with very high accuracy rates above 85%, and has a much more performant run time as compared to CBA-RG, improving run time from 5.43s to 0.3s when run times are averaged out.

Dataset Description	**Iris**	**Iono sphere**	**wine**	**zoo**	**heart**	**car**	**banknote**

#Attributes	**4**	**34**	**13**	**17**	**75**	**6**	**5**
#Instances	**150**	**351**	**178**	**101**	**303**	**1728**	**1372**
Area	**Life**	**Physical**	**Physical**	**Life**	**Life**	**N/A**	**Computer**
Characteristics	**Real**	**Real, Integer**	**Real, Integer**	**Categorical, Integer**	**Categorical Integer, Real**	**Categorical**	**Real**

Datasets chosen range from small data sets to bigger datasets, across a wide variety of characteristics and domains, to allow us to better test the robustness of our classifiers.

3. Empirical Evaluation with other Classifier results

The classification results produced by our implementation of CBA-CB M2 are compared against the classification results of the different classifiers on 7 datasets from the UCI ML repository and kaggle.com
RF: Random Forest, Gaus: GaussianNB, L: LinearDiscriminantAnalysis,

Dataset	Decision Tree	KNN	SVC	RF	XGB	Ada boost	Gradient	Gauss	L
iris	95	95.5	51.1	95.5	95.5	95.5	95.5	95.5	95.5
Iono sphere	84.5	88.7%	57.7	94.3	98.1	92.9	97.1	92.9	88.7
wine	90.7	74	35.1	98.1	96.2	53.7	96.2	98.1	100
zoo	83.8	93.5	48.3	100	93.5	74.1	93.5	96.7	96.7
heart	70.3	62.6	54.9	72.5	73.6	73.6	71.4	79.1	78
car	94	84.5	68.4	94	96.3	79.1	95.2	58.2	73.9
banknote	95	95.6	93	95.6	95.2	95.2	95.6	89.4	93.4
Average	87.6	72.3	58.3	92.8	92.6	80.5	92	87.1	89.4

Different classifiers excel on specific data-sets, it can be observed that the classification results for our CBA M2 classifier is comparable to the best classifiers here such as the random forest classifier, our classifier however is more consistent across different data sets and is performs well in the variety of datasets we have tested it on.

4. CMAR classifier implementation and evaluation

Overall introduction

CMAR is implemented from scratch in the project. The implementation includes FP-Tree, CR-Tree, rule pruning, and Classification. Customized data structures are introduced to store label information of tree nodes, which is not included in the CMAR paper (details in Appendix B). Rule mining is also modified from the recursive frequent pattern mining introduced in the paper as label information needs to be considered. Due to time constraints, data cleaning is not fully implemented hence few tests are performed. The CMAR classifier achieves a **3.5%** error rate on **car** dataset and a **0%** error rate on

Notice that, since existing modules implementing CMAR and its components are based on other algorithm frameworks, to eliminate other noise factors, we fully implemented the original CMAR pipeline proposed in the CMAR paper on our own.

Algorithm analysis

CMAR-RG uses a top-down divide-and-conquer strategy to decompose the rule generation problem for a whole dataset into sub-problems of rule generation over datasets where some elements are fixed or removed. CMAR-RG also uses a modified FP-tree structure to store datasets with class labels. This structure would be discussed in detail in Algorithm_cmar1. Some terms are defined below for reference later:

The projected dataset over element E of dataset S is a subset of records in S where E appears in each of them. Element E is removed in the projected subset for each record. We name it as proj(S, E).

The merged dataset over element E of dataset S is a set S' where E is removed for each record if the record contains E. We name it as merge(S, E). The following lemmas are critical to show the feasibility of CMAR-RG:

Lemma1: For any dataset S and an arbitrary element E appearing in S, a frequent and confident association rule Rs mined on the projected dataset over E of S is also frequent and confident with E appended to its precondition in dataset S (named as R). The proof is in the appendix.

Please refer to lemma2 about tree merging in Algorithm_cmar5. In brief, this lemma says that for any S and E, any rule mined on a dataset tree obtained by merging a tree of S over E (name as merge(S, E)) won't contain E.

By these two lemmas, for any dataset tree S and element E in S, all the frequent and confident rules in S can be divided into:

1. Rules containing E, which can be mined on proj(S, E).
2. Rules do not contain E, which can be mined on merge(S, E).

Since proj(S, E) and merge(S, E) both reduce the cardinality of distinct elements in S by 1, this divide and conquer strategy would have its end condition.

And hence, the complexity of vanilla CMAR is heavily affected by the depth of the tree. A dataset with rich attributes and small number of data records will have high complexity on CMAR, and many rules mined by CMAR are also an overkill for such a dataset.

Pseudocode and analysis

Algorithm_cmar1: the main flow of CMAR-RG

```
// Notice that our pseudocode ignores hyperparameters e.g. MINSUP details. All hyperparameters are in
full uppercases.
// This notice is valid across all algorithm pseudocode.
Input: dataset
Output: frequent and confident rule set
frequent_item_queue = find_frequent_items(dataset)
dataset_tree, head_table = make_CMARFPTree(dataset, frequent_item_queue)
full_dataset_tree = dataset_tree // deep copy
all_result_rules = []
for (queue_iterator in frequent_item_queue start from its end()):
    projected_dataset, projected_head_table = project_dataset_to_element(dataset_tree, head_table,
queue_iterator*)
    dataset_tree, head_table = merging_tree_for_cur_element(dataset_tree, head_table, queue_iterator*)
    current_elem_rules = mine_rules(projected_dataset, projected_head_table, queue_iterator*)
```

```
    all_result_rules += current_elem_rules if it is not None
result_rules = prune_rules(all_result_rules, full_dataset_tree)
return result_rules
```

Analysis

find_frequent_items: This function receives a dataset and produces all frequent elements. The returned data structure should preserve the descending order of count of each element in the dataset. Notice that it returns a queue just to indicate that it should preserve order information.

Algorithm_cmar2: make_CMARFPTree

```
// Notice: in this pseudocode, the definition of the tree structure is omitted.
// Each node has a map to store class labels and their counts, and a (possibly None)
// next pointer to maintain a queue for each element.
Input: dataset, frequent_item_queue
Output: a CMAR-FP tree and a head table containing all the frequent elements
tree = Tree()
head_table = Map()
queue_heads = Map()
for each record in dataset:
    node_to_insert = tree.root
    for each element in frequent_item_queue:
        if element in record:
            if element in node_to_insert.children():
                node_to_insert = tree[node_to_insert].find_child(element)
        else:
            node_to_insert = tree[node_to_insert].new_child(element)
            head_table[element].next = node_to_insert if reference is not None
                else queue_heads[element] = node_to_insert
            head_table[element] = node_to_insert
    node_to_insert.value[label of that record] += 1
```

Analysis (may need to add specific line numbers)

The algorithm constructs a CMAR-FP tree from the given vanilla dataset and a header table that stores frequent elements with corresponding nodes queue. For each record, the algorithm checks the appearance of each frequent element. If an element appears, a child corresponding to the element would be updated (if it exists) or created (if it does not exist) from the current node. The label count for this record's label is added to the last visited node when the frequent element in the queue is exhausted.

A header table is also maintained. It contains a queue head for each element, where the queue contains all the nodes of that element in the tree. When a new node of the element is created, it is appended to the queue's end.

The return value is the header table and an FP-tree storing all the records, where higher support elements are predecessors of lower support ones. Pruning, projection over the element, and merging (will be discussed in algorithm 5) could be performed conveniently on an FP-tree structure. This is a vanilla approach, and potential improvements could be intersecting element sets of each record to frequent queues to get an intersection at one time.

Algorithm_cmar3: project_dataset_to_element

```
Input: dataset_tree, head_table, element
Output: projected_dataset, projected_head_table
current_node = head_table[element]
projected_head_table = Map()
while current_node is not None:
    mark each predecessor and successor including tree root and current_node
```

```
    update next() for each same element in head_table to the marked predecessor
    update projected_head_table if the corresponding entry does not exist in map
delete all unmarked nodes in dataset_tree
return dataset_tree, projected_head_table
```

The algorithm constructs a projected dataset of S over E. It firstly finds the node queue in the head_table of the element. Then, nodes are traversed one by one, and each time the full path from the root to the leaf is marked. Lastly, all unmarked nodes are deleted. The remaining subtree is the projection of the element as each node of the element is traversed and all the predecessors and successors are marked.

The algorithm also updates head_table simultaneously when marking elements. Since head_table queues are only used for traversal, the order of elements in each queue does not matter. Hence, it appends each marked element to the corresponding queue.

Algorithm_cmar4: mine_rules

```
Input = dataset_tree, head_table, element_reduced
Output = rules for this dataset
for frequent_node retrieved in header_table.value:
    while frequent_node:
        rule_list += retrieve rule from the label dictionary in that FP node with support and
confidence threshold
        frequent_node = next node in that header queue
    return rule_list
```

Analysis

This algorithm retrieves all the rules in the leaves of a tree, given a prefix of this tree. The prefix is constructed by merging and projection operations which were done previously. As each tree is projected on the least supported element (add the element to prefix) and merged on that element (ignore the element from prefix) once, the prefix is guaranteed to cover all possible combinations of frequent labels.

Hence, in each recursive step, the function examines the least supported element nodes and sees in each node whether each label's count exceeds support and each label's count/sum of all label counts exceeds confidence. If so, include that rule.

Algorithm_cmar5: merging_tree_for_cur_element

```
Input: dataset_tree, head_table, element_to_merge
Output: new_tree, new_head_table
current_node = head_table[element_to_merge]
while current_node is not None:
    parent = current_node.parent()
    for each record in current_node.value:
        merge or add that record to the parent's value
    next_node = current_node.next
    delete current_node
    current_node = next_node
new_head_table = head_table.pop(element_to_merge)
return dataset_tree, new_head_table
```

Analysis

The algorithm performs a merging operation given dataset S and element E. It retrieves the head_table queue for the element and iterates all the nodes. For each node, the algorithm retrieves the class label information and merges it to the parent node. After merging all nodes of the element, the entry in head_table would be removed. Hence, semantically, mining on the merged tree gives the rules without merged elements. (This is also the proof for lemma 2 in overall analysis.)

An important assumption of this algorithm is that all the eliminated nodes must be the least supported nodes since they are the leaves. Otherwise, the algorithm should connect a node's subtree to the parent after merging, which adds unnecessary complexity.

Algorithm_cmar6: prune_rules

```
Input: rules, dataset
Output: pruned_rules
CRTree, CR_head_table = construct_CR_Tree(rules)
sort(rules, key = CBA_rule_order)
while rules and dataset are both not empty:
    for each rule in rules:
        mark all data records match the rule
        if it correctly covers at least one marked record:
            pruned_rules += this rule
        add cover count 1 to each marked record and remove if count >= threshold after addition
        unmark all records
return pruned_rules
```

Analysis

This algorithm prunes a given set of rules. It first constructs a CR-tree (similar structure to FP-tree where only support and confidence are stored). Hence, algorithm 2 (make_CMARFPTree) could be adopted. Additionally, in place pruning is performed when inserting a rule: for the node inserting this rule, (1) prune all the rules that are successors of this rule and have higher CBA rule ranking, and (2) do not insert this rule if a predecessor has higher CBA rule ranking. The pruning would only keep a set of Minimum Closed Rules. Then, it uses a similar database coverage method as CBA M1. The variation is just that a data record is only removed once covered by k-rules, k is a hyperparameter. This is prepared for multi-rule classification.

Algorithm_cmar7: classification

```
Input: rules, data_entry
Output: predicted label
for rule in rules:
    if rule.items matches data_entry.items:
        mark this rule with its predicted label
for each group of marked rules with the same label:
    compute the group's weighted chi-square score
pick the label with a maximum chi-square score
```

Analysis

This is the classification logic of CMAR. It receives a set of pruned rules. It finds all matching rules to the given data entry and groups them by the label. Then, It computes each group's weighted chi-square score and chooses the label with the highest weighted chi-square score. Semantically, the weighted chi-square score is positively correlated with both weighted support and confidence in the group, hence a larger weighted chi-square means a larger 'influence' of this rule compared to other rules.
A detailed definition of weighted chi-square can be seen in the CMAR paper.

5. References

a. Liu, B., Hsu, W., & Ma, Y. (1998, August). Integrating classification and association rule mining. In KDD (Vol. 98, pp. 80-86).
b. Liu, L., Chen, D., & Xiao, L. (2017). Implementation of algorithm CBA (Classification Based on Associations). liulizhi1996/CBA. Retrieved 2021, from https://github.com/liulizhi1996/CBA
c. Li, W., Han, J., & Pei, J. (2001, November). CMAR: Accurate and Efficient Classification Based on Multiple Class-Association Rules. *Proceedings 2001 IEEE International Conference on Data Mining.*

Appendix A

Lemma 1 For any dataset S and an arbitrary element E appearing in S, a frequent and confident association rule Rs mined on the projected dataset over E of S is also frequent and confident with E appended to its precondition in dataset S (named as R). The proof is in the appendix.

Proof: each record selected from S to be in proj(S, E) contains E. Hence, for any record in S, if it does not contain E, it won't be covered by any rule R since each R contains E. In another way, that record would be selected into proj(S, E) with E removed, hence any rule Rs covering it must have its R covering the original record since R = Rs + E in precondition and the record's E is removed when selected into proj(S, E). Hence, two sets of rules would cover the same set of records in proj(S, E) and S, having the same support and confidence. Proof done.

Appendix B

Data structure definitions

```
# DataEntry definition. Data read will be converted into this format for tree construction
class DataEntry:
    def __init__(self, items: list[str], label: dict[str:int], count: int = 1):
        self.items = items # the items (in order)
        self.label = label # e.g. {‘A’:1} means label A has one count, the count should be 1 when
                           # reading original data
        self.count = count # number of entries with such itemsets and labels, should be 1 when reading
                           # original data

# RuleEntry definition. The rule entry will be stored in the CR tree and feed to the classifier.
class RuleEntry:
    def __init__(self, items: set[str], label: str, support: int, confidence: float):
        self.items = items # items (in order)
        self.label = label # only a str as a rule can only have one label
        self.support = support
        self.confidence = confidence

# FP tree node definition.
class TreeNode:
   def __init__(self, name: str, occurrence: int, parent: TreeNode | None, label=None):
        self.name = name # item (element) stored in tree node
        self.count = occurrence # number of occurrence of the item
        self.next = None # next node, for header table queue traversal
        self.parent = parent # parent node
        self.children = {} # child nodes, key is element, value is node reference
        self.labels = label # e.g. {‘A’:1, ‘B’:1} label information, only leaf nodes will have

# CR tree node. Similar to FP tree node but label is str and support & confidence are introduced
class CRTreeNode:
   def __init__(self, name: str, support: int, confidence: float, parent: CRTreeNode, label=None):
        self.name = name
        self.confidence = confidence
        self.support = support
```

```
        self.next = None
        self.parent = parent
        self.children = {}
        self.label = label
```