B.Sc. Eng. Thesis

New Algorithms for Associative Classification

By

Gourab Kundu (0105001)

Md. Faizul Bari (0105009)

Sirajum Munir (0105066)

Department of Computer Science and Engineering
Bangladesh University of Engineering and Technology
June 2007

Submitted to

Department of Computer Science and Engineering

In partial fulfillment of the requirements for the degree of

Bachelor of Science in Computer Science and Engineering

Acknowledgements

We would like to express our honor and gratefulness to Dr. Md. Monirul Islam, Associate Professor, Department of Computer Science and Engineering, Bangladesh University of Engineering and Technology (BUET), for his continuous support, advice, care and teaching us how to carry on a research work. We have learned from him how to write, speak and present well. His endless patience, scholarly guidance, continual encouragement, constant and energetic supervision, constructive criticism, valuable advice, suggestions about new ways of thinking, reading many inferior drafts and correcting them at all stage have made it possible to complete this thesis.

We would also want to thank Professor Dr. Muhammad Masroor Ali, Head, Department of Computer Science and Engineering, BUET, for the provision of laboratory facilities.

Finally, we would like to thank all the faculty members and staff of Department of CSE, BUET, for their support and cooperation.

Abstract

Association classifiers have been the subject of intense research for the last few years. Experiments have shown that they generally result in higher accuracy than decision tree classifiers. Most of the associative classifiers generate rules in a level wise manner with minimum support pruning. Often this leads to generation of a large number of insignificant rules but at the same time good rules with relatively low support are not produced. In this Thesis, we introduce a novel approach for rule generation "weighted Apriori - Reverse Rule Generation" (weighted Apriori-RRG) which overcomes the above problem by generating a set of high confidence rules without any support pruning. Experiments on 8 datasets show that our approach achieves better accuracy than other state-of-the-art associative classification algorithms.

Contents

List of Tables

List of Algorithms

List of Symbols

CAR	Class Association Rules
CBA	Classification Based on Association
CMAR	Classification Based on Multiple Association Rules
CPAR	Classification Based on Predictive Association Rules
ARCPAN	Associative Classification Based on Positive and Negative Association Rules
L3	Live and Let Live
LAC	Lazy Associative Classification
ACN	Associative Classification with Negative Rules
RRG	Reverse Rule Generation

Chapter 1

Introduction

1.0 General Introduction

Databases are rich with **hidden information** that can be used for making intelligent business decision. Classification is a form of data analysis that can be used to extract models describing important data classes. Many classification methods have been proposed by researchers in machine learning, expert systems, statistics and neurobiology.

Data classification is a two step process. In the first step, a model is built describing a predetermined set of data classes. The model is constructed by analyzing database tuples described by attributes. Each tuple is assumed to belong to a predefined class, as determined by one of the attributes, called the **class label attribute**. In the context of classification, data tuples are also referred to as *samples* or *examples*. The data tuples analyzed to build the model collectively form the **training dataset**. The individual tuples making up the training set are referred to as **training examples** and are randomly selected form the training dataset. Typically, the learned model is represented in the form of *classification rules, decision trees, neural networks* or *mathematical formulas*. In the second step the model is used for classification. But first the predictive accuracy of the model is estimated on a separate test dataset. If the accuracy of the model were estimated based on the training dataset, the estimate could be optimistic since the learned model tends to **overfit** the data. Therefore, a separate test dataset is used. If the accuracy of the model is considered acceptable, the model can be used to classify future data tuples for which the class level is not known.

Association rule mining finds interesting association or correlation relationships among a large set of data items. The discovery of interesting association relationships among large amount of records can help in many decision making process, such as catalog design, cross marketing, loss-leader analysis etc.

Traditionally, greedy search algorithms are used to build such systems, for example, C4.5 [1] and rule induction [2]. This approach is very efficient and can achieve low error rates when used together with different pruning strategies to overcome the problem of overfitting. Association-rule mining is a data-mining technique for finding all large itemsets collectively, satisfying both syntactic and support constraints. Syntactic constraints restrict the items that can appear in a rule, either as the antecedent or the consequent. Support constraints are usually expressed by two parameters: *support* and *confidence*. The *support* for a rule is defined as the fraction of data instances that satisfy the union of items in the antecedent and the consequent of the rule. The *confidence* of a rule is the fraction of data instances containing antecedent items that also contain the consequent [3].

In particular, if the syntactic constraint is that the consequents are restricted to be class labels, and all other attributes to be the antecedents, this subset of association rules, called *class association rules* (CARs), is a good choice for classifying data instances. Following this approach, researchers in recent years have proposed algorithms to build classification systems based on CARs [4, 5, 6, 7]. Different from greedy algorithms, association-rule mining searches globally for all rules that associate class labels with combined attributes; thus, it is able to achieve global optimality. Study results show that this approach can achieve lower error rates than greedy algorithms [4, 5].

There are many fast algorithms for association-rule mining, such as Apriori [7]. In classification systems based on CARs, algorithms are adapted for the following reasons:

- Only the *class association rules* (CARs) are of interest in the system;

- Comparing to transactional data which consists of consumer product items, classification data usually contain attributes with continuous values;
- In general, the number of CARs is large. First, classification data tends to have attributes and class values highly related; and secondly, in building such systems, we want the CARs to cover most of the instances.

When we use CARs to classify a new instance, it is possible that more than one rule can be applied. For example, suppose a dataset with attributes *A1*, *A2*, and *A3*, and there are CARs *{a1, a3} => y* and *{a2} => n*, where *a1*, *a2*, *a3* are attribute values, *y* and *n* are class labels. Given a tuple {*a1*, *a2*, *a3*}, depending on which rule to choose, it can be classified either as *y* or *n*.

A classification algorithm based on CARs can be decomposed into three main phases. These are:

1. **Rule generation:** In this phase the CARs are generated from the dataset.
2. **Rule ordering:** Here the rules are ordered to form a classifier. The ordering can be done on many criteria. Such as *confidence, support, rule length* etc.
3. **Classification:** At this step the test data is classified by the ordered rules.

However, associative classification suffers from efficiency due to the facts that it often generates a very large number of rules in association rule mining, and also it takes efforts to select high quality rules from among them.

1.1 Literature Review

The first association rule mining algorithm was the Apriori algorithm [7] proposed by Agrawal, R., Imielinski, T., Swami in 1993. The Apriori algorithm generates the candidate itemsets in one pass through only the itemsets with large support in the previous pass, without considering the transactions in the database.

B. Liu, W. Hsu, and Y. Ma proposed a framework in 1998, named associative classification, to integrate association rule mining and classification [5]. The integration is done by focusing on mining a special subset of association rules whose consequent parts are restricted to the classification class labels, called “Class Association Rules” (CARs). This algorithm first generates all the association rules and then selects a small set of rules to form the classifiers. When predicting the class label for a coming sample, the best rule is chosen.

W. Li, J. Han, and J. Pei proposed an algorithm “Classification based on Multiple Association Rules” (CMAR) in 2001 [4], which utilizes multiple class-association rules for accurate and efficient classification. This method extends an efficient mining algorithm, FP-growth [8], constructs a class distribution- associated FP-trees, and predicts the unseen sample within multiple rules, using weighted χ^2.

Liu and Li’s approaches generate the complete set of association rules as the first step, and then select a small set of high quality rules for prediction. These two approaches achieve higher accuracy than traditional classification approaches such as C4.5. However, they often generate a very large number of rules in association rule mining, and take efforts to select high quality rules from among them.

In 2003 Yin et al proposed “Classification based on Predictive Association Rules” (CPAR) [9], which combines the advantages of both associative classification and traditional rule-based classification. CPAR adopts a greedy algorithm to generate rules directly from training data, and hence generates and tests more rules than traditional rule-based classifiers to avoid missing important rules, and uses expected accuracy to evaluate each rule and uses the best k rules in prediction to avoid overfitting.

The ARCPAN algorithm generates both positive and negative association rules and ranks them in terms of correlation coefficient. This set of rules is later used in the classification stage. This categorizer is used to predict to which classes’ new objects are attached. Given a new object, the classification process searches in this set of rules for those classes that are relevant to the object presented for classification. The set of positive and negative rules discovered are ordered by confidence and support.

Baralis et al [10] proposed “Live and Let Live” (L3), for associative classification. In this algorithm, classification is performed in two steps. Initially, rules which have already correctly classified at least one training case, sorted by confidence, are considered. If the case is still unclassified, the remaining rules (unused during the training phase) are considered, again sorted by confidence.

Unlike the eager associative classifier that extracts a set of ranked CARs from the training data, the lazy associative classifier induces CARs specific to each test instance. The lazy approach projects the training data, $\boldsymbol{D}$, only on those features in the test instance, $\boldsymbol{A}$. From this projected training data, $\boldsymbol{D_A}$, the CARs are induced and ranked, and the best CAR is used. From the set of all training instances, $\boldsymbol{D}$, only the instances sharing at least one feature with the test instance $\boldsymbol{A}$ are used to form $\boldsymbol{D_A}$. Then, a rule-set $\boldsymbol{R_A}$ is generated from $\boldsymbol{D_A}$. Since $\boldsymbol{D_A}$ contains only features in $\boldsymbol{A}$, all CARs generated from $\boldsymbol{D_A}$ must match $\boldsymbol{A}$**.** the lazy associative classifier produces better results than its eager counterpart.

1.2 Aim of the Thesis

Associative classification suffers from efficiency due to the facts that it often generates a very large number of rules in association rule mining, and also it takes efforts to select high quality rules from among them. The purpose of this work is to design and implement an association rule mining algorithm that can generate CARs in a time and memory efficient manner. The objective of this thesis was as follow:

1. Generating rules in some way that is efficient in time and memory.
2. Developing an efficient rule pruning technique.
3. Ordering the rules that will provide a robust classifier.

We designed and implemented eight different algorithms and computed their accuracy on the dataset found in UCI repository of machine learning database and found good result.

1.3 Thesis Organization

The rest of the thesis is organizes as:

Chapter 2 describes various previous classification algorithms based on association rules, such as CBA (Classification Based on Association), CMAR (Accurate and efficient classification based on multiple class-association rules), CPAR (Classification based on Predictive Association Rules), Lazy Associative Classification etc.

Chapter 3 contains our proposed algorithms with pseudo code and theoretical explanation. It also discusses the advantages and shortcomings of these algorithms.

Chapter 4 presents experimental results. Experimental results are given in tables comparing accuracy of our proposed algorithms with CBA, CMAR and CPAR. These results are also discussed quantitatively.

Chapter 5 concludes our work with a summary of the thesis and suggestions for future research directions.

Chapter 2

Background

2.0 Introduction

Knowledge discovery and data mining is generally known as the science of extracting useful information from large and complex datasets or databases. A data mining system is targeted at integrating knowledge discovery and data mining techniques into databases for adaptive and intelligent data analysis. One important data mining task is predicting the unknown value of a variable of interest given known values of other variables. For a classification problem, samples of different classes are accumulated, on which a classifier is modeled to predict future samples.

2.1 Association rule

Let **I** = $\{i_1, i_2, . . . , i_m\}$ be a set of items. Let **D** be set of database records where each record **T** is a set of items such that **T** is a subset of **I**. Let **A** be a set of items. A record **T** is said to contain **A** if and only if **A** is a subset of **T**. An Association rule is an implication of the form **A => B**, where **A** is a subset of **I**, **B** is a subset of **I** and $\mathbf{A} \cap \mathbf{B} = \{\}$. The rule **A => B** holds in the transaction set D with

support = the prior probability of **A** and **B**

$$= P(\mathbf{A} \cup \mathbf{B})$$

$$= (|\, \mathbf{A} \cup \mathbf{B} \,|) \,/\, |\, \mathbf{D} \,|$$

and

confidence = the conditional probability of **B** given **A**

= P (**B | A**)

= (| **A** U **B** |) / | **A** |

2.2 Association rule mining

Association rule mining is one of the best studied models for data mining. In recent years, the discovery of association rules from databases is an important and highly active research topic in the data mining field. Association rule mining searches for interesting association or correlation relationships among items in a given dataset.

2.3 The Apriori Algorithm

Agrawal, R., Imielinski, T., Swami [3] proposed the first association rule mining algorithm in 1993 to discover patterns in transactional databases from the retail industry and business. The idea to discover association rules is also named “market basket analysis” because it looks for associations among items that a customer purchases in a retail shop. They proposed the Apriori algorithm. The Apriori algorithm generates the candidate itemsets in one pass through only the itemsets with large support in the previous pass, without considering the transactions in the database. An itemset with support larger than or equal to the minimum support is called a frequent itemset. The idea of the Apriori algorithm lies in the “downward-closed” property of support, which means if an itemset is a frequent itemset, then each of its subsets is also a frequent itemset. The candidate itemsets having k items can be generated by joining frequent itemsets having k-1 items, and removing all subsets that are not frequent. The Apriori algorithm starts by finding all frequent 1-itemsets (itemsets with 1 item); then consider 2-itemsets, and so forth. During each iteration only candidates found to be frequent in the previous iteration are used to generate a new candidate set during the next iteration. The algorithm terminates when there are no frequent k-itemsets. To improve the efficiency of the Apriori algorithm, many variations of the Apriori algorithm have been designed including hashing, transaction reduction, partitioning the data (mining on each partition and then combining the results), and sampling the data (mining on a subset of the data).

2.4 Class Association rule mining

Let **C** = {c_1, c_2, . . . , c_n} is the set of classes. Then a class association rule (CAR) is an implication of the form **A** => **B** where **A** is a subset of **I** and **B** ∈ **C**. Mining algorithms like Apriori can be used to mine CARs. These types of algorithms are used to formulate interesting rules from a given dataset that are used for classification. Because of their easy interpretability, the mined association rules may also be utilized for understanding the relationships among different items and the impact of different factors upon the final classification.

There are some good numbers of associative classification algorithms available now. All claim to offer some benefits, either in accuracy or in reduction of computation time. Here is a brief description of the major classification algorithms:

2.4.1 Classification Based on Association (CBA)

B. Liu, W. Hsu, and Y. Ma proposed a framework, named associative classification, to integrate association rule mining and classification [5]. The integration is done by focusing on mining a special subset of association rules whose consequent parts are restricted to the classification class labels, called “Class Association Rules” (CARs). This algorithm first generates all the association rules and then selects a small set of rules to form the classifiers. When predicting the class label for a coming sample, the best rule is chosen. It consists of two parts, a *rule generator* (called CBA-RG), which is based on algorithm Apriori for finding association rules and a *classifier builder* (called CBA-CB).

The key operation of CBA-RG is to find all *ruleitems* that have support above *minsup*. A *ruleitem* is of the form : <*condset,* y> where *condset* is a set of items, *y is* a class label. These rule items are called CARs (Class Association Rules). The support count of the *condset* (called *condsupCount*) is the number of cases in *D* (Database) that contain the *condset*. The support count of the *ruleitem* (called *rulesupCount*) is the number of cases in *D* that contain the *condset* and are labeled with class *y*. Each *ruleitem* basically represents a rule [*condset->y]* whose *support* is (*rulesupCount* / |*D*|) *100%, where |*D*| is the size of the dataset, and whose *confidence* is r*ulesupCount/condsupCount*)*100%. *Ruleitems* that satisfy *minsup* are called *frequentRuleitems*, while the rest are called *infrequentRuleitems*. For all the *ruleitems* that have the same *condset*, the *ruleitem* with the highest confidence is chosen as the *possible rule* (PR) representing this set of *ruleitems*. If there are more than one *ruleitem* with the same highest confidence, we randomly select one *ruleitem*. If the confidence is greater than *minconf*, we say the rule is *accurate*. The set of *class association rules* (CARs) thus consists of all the PRs that are both frequent and accurate.

Let *R* be the set of generated rules and *D* the training data. The basic idea of the CBA-CB algorithm is to choose a set of high precedence rules in *R* to cover *D*. In classifying an unseen case, the first rule that satisfies the case will classify it. If there is no rule that applies to the case, it takes on the default class.

2.4.2 Classification based on Multiple Association Rules (CMAR)

W. Li, J. Han, and J. Pei proposed an algorithm "Classification based on Multiple Association Rules" (CMAR) [4], which utilizes multiple class-association rules for accurate and efficient classification. This method extends an efficient mining algorithm, FP-growth [8], constructs a class distribution- associated FP-trees, and predicts the unseen sample within multiple rules, using weighted χ^2. Liu and Li's approaches generate the complete set of association rules as the first step, and then select a small set of high quality rules for prediction. These two approaches achieve higher accuracy than traditional classification approaches such as C4.5. However, they often generate a very large number of rules in association rule mining, and take efforts to select high quality rules from among them.

2.4.3 Classification based on Predictive Association Rules (CPAR)

Yin et al proposed "Classification based on Predictive Association Rules" (CPAR) [9], which combines the advantages of both associative classification and traditional rule-based classification. CPAR adopts a greedy algorithm to generate rules directly from training data, and hence generates and tests more rules than traditional rule-based classifiers to avoid missing important rules, and uses expected accuracy to evaluate each rule and uses the best k rules in prediction to avoid overfitting.

2.4.4 Associative Classifier Based On Positive and Negative Association Rules (ARCPAN)

This classifier generates both positive and negative association rules and ranks them in terms of correlation coefficient. This set of rules is later used in the classification stage. This categorizer is used to predict to which classes' new objects are attached. Given a new object, the classification process searches in this set of rules for those classes that are relevant to the object presented for classification. The set of positive and negative rules discovered are ordered by confidence and support.

2.4.5 Live and Let Live (L3)

Baralis et al [10] proposed "Live and Let Live" (L3), for associative classification. In this algorithm, classification is performed in two steps. Initially, rules which have already correctly classified at least one training case, sorted by confidence, are considered. If the case is still unclassified, the remaining rules (unused during the training phase) are considered, again sorted by confidence.

2.4.6 Lazy Associative Classification (LAC)

Unlike the eager associative classifier that extracts a set of ranked CARs from the training data, the lazy associative classifier induces CARs specific to each test instance. The lazy approach projects the training data, $\boldsymbol{D}$, only on those features in the test instance, $\boldsymbol{A}$. From this projected training data, $\boldsymbol{D_A}$, the CARs are induced and ranked, and the best CAR is used. From the set of all training instances, $\boldsymbol{D}$, only the instances sharing at least one feature with the test instance $\boldsymbol{A}$ are used to form $\boldsymbol{D_A}$. Then, a rule-set $\boldsymbol{R_A}$ is generated from $\boldsymbol{D_A}$. Since $\boldsymbol{D_A}$ contains only features in $\boldsymbol{A}$, all CARs generated from $\boldsymbol{D_A}$ must match $\boldsymbol{A}$**.** the lazy associative classifier produces better results than its eager counterpart.

Using association rules for classification helps to solve the understandability problem in classification rule mining. Many rules produced by standard classification systems are difficult to understand because these systems use domain independent biases and heuristics to generate a small set of rules to form a classifier. However, these biases may not be in agreement with the knowledge of the human user, result in that many generated rules are meaningless to user, while many understandable and meaningful rules are left undiscovered.

Chapter 3

Proposed Algorithms

3.1 Support Adjusted Recursive Classifier

Basic Idea

Reduce running time. Instead of exhaustively generating all frequent item-sets, generate highly supported ones first. If the rules from them are enough to classify, no need to generate item-sets of higher size. If not, then try to find frequent item-sets from remaining attributes using a low support count.

General Procedure

We start with a high support count (Say 30%) & mine frequent item-sets of size upto 3 (can be made a user parameter). We generate all rules from the item-sets generated. We selectively choose some rules based on confidence measure. For each rule taken in order find all examples that fall under the rule (satisfy the premise of the implication). If confidence of rule is above acceptance threshold (say 96%) then return that rule and remove all examples covered by that rule. Otherwise, try to construct a classifier for the examples covered by current rule taking into account attributes other than those in current rule recursively but this time with a lower support count. For each rule found in this way, perform an <AND> operation of it with current rule and return it.

Advantages

- All associative classification algorithms use a very low support threshold (as low as 1%) to avoid missing any interesting classification rule.
- Consequence is very large amounts of item-sets generated.
- Our approach tries with very frequent item-sets first. If they are sufficient, huge pruning is possible. Say we get the rule A=a1->R=Yes considering 50% support .If this rule classifies well, we can skip generation of all rules of the form A=a1 ^ X=x ->R=Yes. This can lead to considerable performance boost.

Open issues

- What will be initial support?
- How can we adjust support value for subsequent recursive calls?
- How can we avoid fragmentation (result of continuous partitioning), replication (Same examples covered by multiple rules) etc.?
- How can we reduce no of I/O operations? We can address a record by keeping records in main memory. But we cannot keep all examples covered by a rule in main memory.
- Possible item-sets with high confidence & relatively low support will be penalized. Because our approach is to include highly supported rule & then increase their confidence by adding new constraints on other attributes.

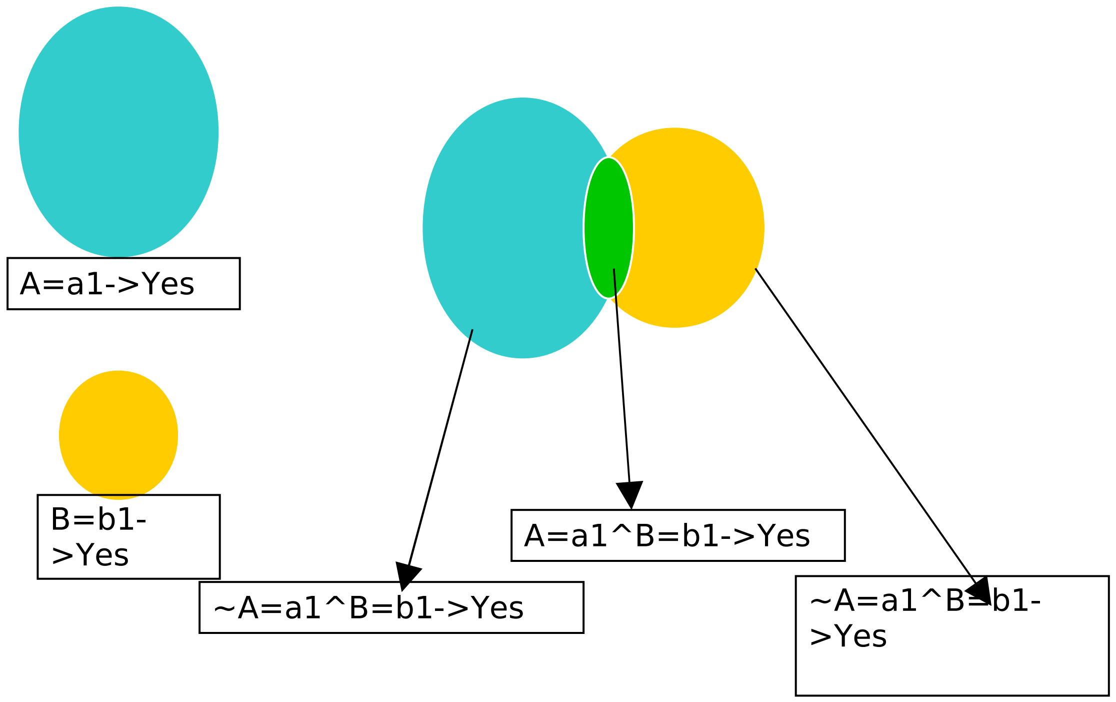
Problems

The above approach was good. It was implemented and it led to good results. But we later found that an algorithm recently published has close similarity with it. So we do not discuss this algorithm in details here.

3.2 ACN: Associative Classifier with Negative Rules

Some existing classifiers use negative rules for classification. They discover rules of the form a1^b1^c1->Yes and ~ (a1^b1^c1) -> Yes and a1^b1^c1->~Yes. Generally negative association rule mining is a difficult task and it's an ongoing research activity. In this classifier we consider a subset of rules that have at most one negated literal. So consider a1^b1^c1->Y and a1^b1^~c1->Y but not a1^~b1^~c1->Y. Rules of this form are very important since it can express semantics like "If I have a playing partner and that partner is not Robin, then I am going to enjoy sport" (because I have some problems with Robin), Essence of our algorithm is that we only consider negated rules that arise naturally during APriori rule mining process so that no extra overhead is needed. During APriori mining, when we generate a Candidate A=a1^B=b1->Yes from two frequent ruleItems A=a1->Yes and B=b1->Yes, we can generate two more ruleItems of the form A=a1^B=~b1->Yes and ~A=a1^B=b1->Yes which can have higher conf. & sup than A=a1^B=b1->Yes. Support and confidence of the new 2 rules can easily be Calculated based on already available information.

supp (A=a1^~B=b1) = supp(A=a1) - supp(A=a1^B=b1)
rulesup (A=a1^~B=b1) = rulesup(A=a1)-rulesup(A=a1^B=b1)
conf (A=a1^~B=b1) = (rulesup(A=a1^~B=b1)/supp(A=a1^~B=b1))



Increased User Perceivness

- Say an attribute “Bilirubin” has 5 values “Normal”, ”BelowNormal”, ”AboveNormal”, ”Very Low”, ”Very High”
- There is a hidden rule of the form A=a1^B=~Normal->Disease
- Such rule can be perceived by user only if he sees 4 rules of the form
- A=a1^B=“BelowNormal”->Disease
- A=a1^B=“AboveNormal”->Disease
- A=a1^B=“Very Low”->Disease
- A=a1^B=“Very High”->Disease
- But because each rule separately has low support,because of heuristic rule selection and support pruning in CBA, it is not guaranteed that all 4 will be selected.

- But if u combine them in 1 rule with B=~Normal , it will have high support and therefore higher probability to be included in the final classifier.
- We argue that we do not curtail the benefits accrued from existing positive rules,but only incorporate some cheap but good quality negated rules to enhance the perceiveness of user and also accuracy.
- We cannot claim that we generate all rules of this form,because rules of this form are very very large in number.We generate a subset of such rules that come as byproducts of APriori Association Mining.

Algorithm

```
P1=find_frequent_1p_itemsets(D)
N1=find_frequent_1n_itemsets(D)
For(k=2;Lk-1!=empty;k++)
                PCk= candidates generated for level k
    for each candidate generated
         for each literal on the candidate
                  create a new negative rule by negating that literal
               add this rule to NCk
   calculate supports for each candidate of PCk
    for each c in Ck   update siblings of c in NCk
        Lk=candidates in PCk  that pass support threshold
    Nk=candidates in  NCk  that pass support threshold
```

Some Important Facts

Negative rules are generated and stored but they do not take part in generating new rules. They only get mixed up with positive rules in the sorting phase and compete for a Place in the final classifier. However, this suffers because of

- For each positive candidate, no of negative rules generated is equal to number of conditions in its premise.
- Say, if 20000 rules each of 4 conditions are generated, 80000 negative rules from them will emerge. This can be large!!!!

So we have to perform pruning to cut down this large number of rules

Rule Ranking Criteria

- A rule ri is ranked higher than rj if
- Confidence(ri)>confidence(rj)
- Correlation(ri)>correlation(rj)
- Support(ri)>support(rj)
- Rulesize(ri)<rulesize(rj)
- If ri is positive & rj is negative

Database Coverage

Sort rules based on rule ranking criteria. For each rule taken in order if rule classifies at least one remaining training example correctly include that rule in classifier and delete those examples. If database in uncovered select majority class from remaining examples Else select majority class from entire training set.

Experimental Fact

Say, a rule A=a1^B=!b1 has confidence 80%. But a rule A=a1^B=b2 has confidence 100%. So this rule is selected and examples covered by this rule are removed. Now it can happen that the confidence of A=a1^B=!b1 has dropped so much that over remaining examples it is inaccurate because its previous high accuracy was largely due to the rule A=a1^B=b2. To remove this problem for negative rules, constraint has been adopted.

If a negative rule does not classify at least 55% of the remaining examples, it cannot be included.

More Pruning

- All rules should be postively correlated.
- So rules with correlation <0 are bad rules and they are pruned.
- Rules with correlation greater than a threshold are good rules.We first try to cover database using these rules.But if database remains uncovered, then we take help of the rules that are postively correlated but correlation < threshold.
- Experimentally set threshold = 0.2

Accuracy

- ACN achieves comparable accuracy with CMAR & CPAR.
- ACN achieves better accuracy than CBA.
- In some datasets specially where CBA generates large number of rules in final classifer, ACN reduces the number of rules.
- CMAR & CPAR covers each training example multiple times and so number of rules in final classifier is huge.
- So ACN achieves good accuracy while generating small number of rules in final classifier.

Features of ACN

- This is a framework, generating some negative rules of this form and using them with positive rules for classification.
- It can enhance user understandability .
- It achieves good accuracy.
- It generates small number of rules in final classifier.

3.3 Level Adaptive Classifier

Basic Idea:

Almost all classifiers use Apriori Rule mining Algorithm to mine the rule set. Apriori is an algorithm that generates frequent rule items on a level wise manner. That is,
At first, all rules with one antecedent are mined, and then all rules with two antecedents and so on. This can generate a large number of rules and the number of rules in each new level can grow in an exponential manner. There are datasets with more than 20 attributes, so a rule of 20 antecedents can only be generated if all the subsets of that rule, 2^20 are generated previously. So this can be intractable. Moreover, rules of larger number of antecedents are generally over fitting rules and do not yield good performance on test data. But there is no way to ascertain when to stop generating rules in any state-of-the-art classifier. So this classifier was an attempt to make the "**max length of a rule"** parameter adaptive. The target was to achieve efficiency without sacrificing accuracy.

Description:

The idea was to generate all rules of level 1 first just as in APriori and construct a classifier using these rules and calculate the number of errors made by this classifier on the validation set. Then level 2 rules are generated and a new classifier is constructed using both level1 and level2 rules. Again, this classifier's performance on validation set is noted by recording the number of errors committed on validation set. The number of errors in this case is compared with the number of errors with level1 rules. If the number of errors does not decrease, we can make the assumption that level 2 rules don't help too much. In this way, classifiers are constructed using rules of level 1, then level 1 and 2, then level 1 ,2 ,3 etc. When two or more classifiers of higher length rules perform worse than previous classifiers, we can convince ourselves that the new long rules are over fitting and we can safely discard them and stop rule generation at that phase(do not generate any more higher length rules).

Algorithm

```
    L1=find_frequent_1_itemset(D);
```

```
Construct a classifier
Find number of errors on validation set

For (k=2; Lk-1!=empty ;k++)
{
    Generate $L_k$ = frequent_k_itemset($L_k$-1);
    Construct a classifier using rules generated so far
    Find number of errors on validation set and compare with previous
    number of errors
    If number of errors increases
        Break;
}
```

This classifier is used to classify future test instances

Problems:

Although this classifier achieved good gain in speed, it performed well in only a few datasets. Further investigation revealed the possible following reasons:

- We either take the whole batch of rules of particular length or reject them entirely. But it is quite intuitive that this can lead to fall in accuracy. Because typically not all rules in a level are bad. Some are good and some are ugly.

- We think the major problem is validation set. Unless the dataset is too large, the validation set will be reasonably small. If the dataset is M records long, for 10 fold cross validation, we first take M/10 records as test set and then divide the remaining 9M/10 records into validation set and training set. According to the well accepted rule of 2:1 ratio for training versus validation, we have 3M/10 records for validation set and 6M/10 records for training set. Unless dataset size M is very large, ranking rules according to small number of validation examples can be problematic.

3.4 RRG algorithm

Reverse Rule Generation (RRG) algorithm generates association rules in a completely reverse way from the existing algorithms. Before describing the algorithm in formal definition, lets take a look what we are going to do by an example. Say, we have the following training examples:

A	B	C	Target classification
a1	b1	c1	Yes
a1	b1	c2	Yes
a2	b2	c1	No
a2	b2	c2	No

At first we will fix a *satisfactoryConfidence*. Say it is 50%. Then we will generate one rule from each training example. So, at first step we have 4 rules. They are like these:

R1: A=a1,B=b1,C=c1=>yes

R2: A=a1,B=b1,C=c2=>yes

R3: A=a2,B=b2,C=c1=>no

R4: A=a2,B=b2,C=c2=>no

Note that all 4 rules have confidence 100%. These rules are enqueued in a queue (say it is *q*). Now dequeue a rule from *q* and remove one attribute constraint at a time. If R1 is dequeued then the 3 rules will be constructed by removing one attribute constraint at a time:

R11: A=a1,B=b1=>yes

R12: B=b1,C=c1=>yes

R13: A=a1,C=c1=>yes

Now enqueue the newly constructed rules in *q* that have confidence greater than or equal to *satisfactoryConfidence* and go on in this way.

So, the RRG algorithm looks like this:

1. *satisfactoryConfidence* = 0.5;

```
2. ruleList = $\Phi$;
3. q = $\Phi$;
4. for each record rec $\in$ training example
5.     r = constructRule(rec);
6.     ruleList = ruleList $\cup$ r;
7.     enqueue(q,r);
8.while (q is not empty)
9.     r = dequeue(q);
10.    for each attribute A $\in$ r
11.        r2 = constructRule2(A, r);
12.        if (confidence of r2 $\geq$ satisfactoryConfidence and r2 $\notin$ ruleList)
13.            ruleList = ruleList $\cup$ r2;
14.            enqueue(q,r2);
```

satisfactoryConfidence and *q* are described earlier. *ruleList* is a list that will contain the generated CARs. Line 1-3 represents initialization. Line 4-7 describes how training examples having confidence greater than or equal to *satisfactoryConfidence* are directly converted to CARs. *ConstructRule* function (line 5) serves this purpose in a way described earlier. *enqueue* function enqueues rule *r* into queue *q*. Line 8-14 generates rules by removing one attribute at a time from the rules found by dequeuing *q*. *constructRule2* function (line 11) is doing a major task by constructing rule *r2* from *r* by removing attribute *A. constructRule2* function also calculates the confidence of rule *r2*. Finally, we get all of our generated rules in *ruleList.*

Classifier Construction

ruleList still contains a lot of rules. They all will not be used in the classifier. The classifier construction algorithm looks like this:

```
1. finalRuleSet = $\Phi$;
2.  dataSet = D;
3. sort(ruleList);
3. for each rule $r \in$ ruleList
4.         if r correctly classifies at least one training example $d \in$ dataset then
5.                 remove d from dataset;
6.                 insert r at the end of finalRuleSet;
```

Lines 1-2 are for initialization purpose. *finalRuleSet* is a list that will contain rules that will be used in the classifier. *sort* function (line 3) sorts *ruleList* in descending order of confidence, support and rule length. Lines 4-6 take only those rules in the *finalRuleSet* which can correctly classify at least one traing example. Note that the insertion in *finalRuleSet* ensures that all the rules of *finalRuleSet* will be sorted in descending order of confidence, support and rule length.

When a new test example is to be classified, classify according to the first rule in the *finalRuleSet* that covers the test example.

Advantages

- There is no support pruning. All associative classification algorithms use a very low support threshold (as low as 1%) to generate association rules. In that way some high quality rules that have higher confidence, but lower threshold will be missed. Here we are getting those high quality rules as there is no support pruning.

Disadvantages

- The reverse rule generation process generates more specific rules. It fails to generate more general rules in some datasets. For this reason, when test cases are tested for classification, it is high likely that there exists no rules in the classifier that can cover the test case. Eventually we are forced to classify the test case in a default class which leads to a poor accuracy in some datasets.

- The algorithm has computational complexity O(k2^n) in the worst case, where k = number of records in the dataset and n = number of attributes in each record.

Open Issues

- What will be the value of *satisfactoryConfidence*?
- Can we impose a limit on the generation of rules? What will be that limit? Can we make that limit adaptive?

5.CBA-RRG algorithm

There are two phases to mine the association rules under this algorithm. During the first phase, it generates most of the association rules in a manner as it is done in APRIORI algorithm. Then during the next phase it generates remaining association rules in the reverse manner of the earlier phase. The second phase is the phase that improves the performance of the original CBA algorithm.

Let *D* be the dataset. Let *I* be the set of all items in *D*, and *Y* be the set of class labels. The key operation of CBA-RRG is to find all *ruleitems* that have support above *minsup*. A *ruleitem* is of the form: <*condset*, *y*> where *condset* is a set of items, *y* Є *Y* is a class label. The support count of the *condset* (called *condsupCount*) is the number of cases in *D* that contain the *condset*. The support count of the *ruleitem* (called *rulesupCount*) is the number of cases in *D* that contain the *condset* and are labeled with class *y*. Each *ruleitem* basically represents a rule: *condset ->y*, whose *support* is (*rulesupCount* / |*D*|) *100%, where |*D*| is the size of the dataset, and whose *confidence* is (*rulesupCount / condsupCount*) * 100%. *Ruleitems* that satisfy *minsup* are called *frequent ruleitems*, while the rest are called *infrequent ruleitems*.

For example, the following is a *ruleitem*: <{(A, 1), (B, 1)}, (class, 1)>, where A and B are attributes. If the support count of the *condset* {(A, 1), (B, 1)} is 3, the support count of the *ruleitem* is 2, and the total number of cases in *D* is 10, then the support of the *ruleitem* is 20%, and the confidence is 66.7%. If *minsup* is 10%, then the *ruleitem* satisfies the *minsup* criterion. We say it is *frequent*.

For all the *ruleitems* that have the same *condset*, the *ruleitem* with the highest confidence is chosen as the *possible rule* (PR) representing this set of *ruleitems*. If the confidence is greater than *minconf*, we say the rule is *accurate*. The set of *class association rules* (CARs) thus consists of all the PRs that are both frequent and accurate.

The first phase generates most of the frequent *ruleitems* by making multiple passes over the data. It begins with the generation of *1-ruleitem* where *k-ruleitem* denote a *ruleitem* whose *condset* has *k* items. Then it counts the support of generated individual *ruleitem* and determines whether it is frequent. In each subsequent pass, it starts with the set of *ruleitems* found to be frequent in the previous pass. It uses this set to generate new possibly frequent *ruleitems,* called *candidate ruleitems*. The actual supports for these candidate *ruleitems* are calculated during the pass over the data. At the end of the pass, it determines which of the candidate *ruleitems* are actually *frequent*. From this set of frequent *ruleitems*, it produces the rules (CARs).

Let *Fk* denote the set of *frequent k-ruleitems*. Each element of this set is of the following form: <(*condset*, *condsupCount*), (*y, rulesupCount*)>. Let *Ck* be the set of candidate *k-ruleitems*.

The first phase of CBA-RRG algorithm is given in Figure 1.

```
1 $F1$ = {large 1-ruleitems};
2 $CAR1$ = genRules($F1$);
3 $prCAR1$ = pruneRules($CAR1$);
4 for ($k$ = 2; $Fk$-1 ≠ Φ ; $k$++) 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: The CBA-RRG algorithm

Line 1-3 represents the first pass of the algorithm. It counts the item and class occurrences to determine the frequent 1-*ruleitems* (line 1). From this set of 1-*ruleitems*, a set of CARs (called *CAR*1) is generated by *genRules* (line 2). *CAR*1 is subjected to a pruning operation (line 3) (which can be optional). Pruning is also done in each subsequent pass to *CARk* (line 15). The function *pruneRules* uses minimum confindence (say *minConf*) defined by the user. It prunes a rule as follows: If rule *r*'s confidence is less than *minConf*, then rule *r* is pruned. This pruning can cut down the number of rules generated substantially.

For each subsequent pass, say pass *k*, the algorithm performs 4 major operations. First, the frequent *ruleitems Fk*-1 found in the (*k*-1)*th* pass are used to generate the candidate *ruleitems Ck* using the *condidateGen* function(line 5). It then scans the database and updates various support counts of the candidates in *Ck* (line 6-12). After those new frequent *ruleitems* have been identified to form *Fk* (line 13), the algorithm then produces the rules *CARk* using the *genRules* function (line 14). Finally, rule pruning is performed (line 15) on these rules.

The *candidateGen* function is similar to the function *Apriori-gen* in algorithm Apriori. The *ruleSubset* function takes a set of candidate *ruleitems Ck* and a data case *d* to find all the *ruleitems* in *Ck* whose *condsets* are supported by *d*. This and the operations in line 8-10 are also similar to those in algorithm Apriori. The difference is that we need to increment the support counts of the *condset* and the *ruleitem* separately whereas in algorithm Apriori only one count is updated. This allows us to compute the confidence of the *ruleitem*. They are also useful in rule pruning. The final set of class association rules is in *CARs* (line 17). Those remaining rules after pruning are in *prCARs* (line 18).

The second phase generates the remaining association rules. Before describing the algorithm in formal definition, lets take a look what we are going to do by an example. Say, we have the following training examples:

A	B	C	Target classification
a1	b1	c1	Yes
a1	b1	c2	Yes
a2	b2	c1	No
a2	b2	c2	No

At first we will fix a *satisfactoryConfidence*. Say it is 95%. Then we will generate one rule from each training example. So, at first step we have 4 rules. They are like these:

R1: A=a1,B=b1,C=c1=>yes

R2: A=a1,B=b1,C=c2=>yes

R3: A=a2,B=b2,C=c1=>no

R4: A=a2,B=b2,C=c2=>no

Note that all 4 rules have confidence 100%. These rules are enqueued in a queue (say it is *q*). Now dequeue a rule from *q* and remove one attribute constraint at a time. If R1 is dequeued then the 3 rules will be constructed by removing one attribute constraint at a time:

R11: A=a1,B=b1=>yes

R12: B=b1,C=c1=>yes

R13: A=a1,C=c1=>yes

Now enqueue the newly constructed rules in *q* that have confidence greater than or equal to *satisfactoryConfidence* and go on in this way.

So, the second phase of CBA-RRG looks like this:

```
1. satisfactoryConfidence = 0.5;
2. ruleList = $\Phi$;
3. q = $\Phi$;
4. for each record rec $\in$ training example
5.     r = constructRule(rec);
6.     ruleList = ruleList $\cup$ r;
7.     enqueue(q,r);
8.while (q is not empty)
9.     r = dequeue(q);
10.    for each attribute A $\in$ r
11.        r2 = constructRule2(A, r);
12.        if (confidence of r2 $\geq$ satisfactoryConfidence and r2 $\notin$ ruleList)
13.            ruleList = ruleList $\cup$ r2;
14.            enqueue(q,r2);
```

satisfactoryConfidence and *q* are described earlier. *ruleList* is a list that will contain the generated CARs. Line 1-3 represents initialization. Line 4-7 describes how training examples having confidence greater than or equal to *satisfactoryConfidence* are directly converted to CARs. *ConstructRule* function (line 5) serves this purpose in a way described earlier. *enqueue* function enqueues rule *r* into queue *q*. Line 8-14 generates rules by removing one attribute at a time from the rules found by dequeuing *q*. *constructRule2* function (line 11) is doing a major task by constructing rule *r2* from *r* by removing attribute *A*. *constructRule2* function also calculates the confidence of rule *r2*.

finally the merging of rules from phase 1 and 2 is done by:
prCARs = *prCARs* ∪ *ruleList*;

Classifier Construction

prCARs still contains a lot of rules. They all will not be used in the classifier. The classifier construction algorithm looks like this:

```
1. finalRuleSet = Φ;
2.  dataSet = D;
3. sort(prCARs);
3. for each rule r ∈  prCARs
4.         if r correctly classifies at least one training example d ∈ dataset then
5.                 remove d from dataset;
6.                 insert r at the end of finalRuleSet;
```

Lines 1-2 are for initialization purpose. *finalRuleSet* is a list that will contain rules that will be used in the classifier. *sort* function (line 3) sorts *prCars* in descending order of confidence, support and rule length. Lines 4-6 takes only those rules in the *finalRuleSet* which can correctly classify at least one traing example. Note that the insertion in *finalRuleSet* ensures that all the rules of *finalRuleSet* will be sorted in descending order of confidence, support and rule length.

When a new test example is to be classified, classify according to the first rule in the *finalRuleSet* that covers the test example.

Advantages

- Here in this approach we have overcome the limitation of RRG algorithm which could not generate more general rules. Here we have both high-support-moderate-confidence rules as well low-support-high-confidence rules. So we have a larger amount of high quality rules which should improve the accuracy of the classifier.

Disadvantages

- The algorithm has computational complexity $O(k2^n)$ in the worst case, where k = number of records in the dataset and n = number of attributes in each record.
- With such a huge computational expense, the achievement in accuracy is not satisfactory enough.

Open Issues

- What will be the value of *minsup* and *minconf* ?
- What will be the value of *satisfactoryConfidence*?
- Can we impose a limit on the generation of rules at both phases? What will be that limit? Can we make that limit adaptive?

6. Weighted CBA-RRG algorithm

There are three phases to mine the association rules under this algorithm. During the first phase, it generates most of the association rules in a manner as it is done in APRIORI algorithm. Then during the next phase it generates remaining association rules in the reverse manner of the earlier phase. The third phase calculates the weight of each generated rule.

Let *D* be the dataset. Let *I* be the set of all items in *D*, and *Y* be the set of class labels. The key operation of CBA-RRG is to find all *ruleitems* that have support above *minsup*. A *ruleitem* is of the form: <*condset*, *y*> where *condset* is a set of items, *y* ▪ *Y* is a class label. The support count of the *condset* (called *condsupCount*) is the number of cases in *D* that contain the *condset*. The support count of the *ruleitem* (called *rulesupCount*) is the number of cases in *D* that contain the *condset* and are labeled with class *y*. Each *ruleitem* basically represents a rule: *condset ->y*, whose *support* is (*rulesupCount* / |*D*|) *100%, where |*D*| is the size of the dataset, and whose *confidence* is (*rulesupCount / condsupCount*) * 100%. *Ruleitems* that satisfy *minsup* are called *frequent ruleitems*, while the rest are called *infrequent ruleitems*.

For example, the following is a *ruleitem*: <{(A, 1), (B, 1)}, (class, 1)>, where A and B are attributes. If the support count of the *condset* {(A, 1), (B, 1)} is 3, the support count of the *ruleitem* is 2, and the total number of cases in *D* is 10, then the support of the *ruleitem* is 20%, and the confidence is 66.7%. If *minsup* is 10%, then the *ruleitem* satisfies the *minsup* criterion. We say it is *frequent*.
For all the *ruleitems* that have the same *condset*, the *ruleitem* with the highest confidence is chosen as the *possible rule* (PR) representing this set of *ruleitems*. If the confidence is greater than *minconf*, we say the rule is *accurate*. The set of *class association rules* (CARs) thus consists of all the PRs that are both frequent and accurate.

The first phase generates most of the frequent *ruleitems* by making multiple passes over the data. It begins with the generation of *1-ruleitem* where *k-ruleitem* denote a *ruleitem* whose *condset* has *k* items. Then it counts the support of generated individual *ruleitem* and determines whether it is frequent. In each subsequent pass, it starts with the set of *ruleitems* found to be frequent in the previous pass. It uses this set to generate new possibly frequent *ruleitems*, called *candidate ruleitems*. The actual supports for these candidate *ruleitems* are calculated during the pass over the data. At the end of the pass, it determines which of the candidate *ruleitems* are actually *frequent*. From this set of frequent *ruleitems*, it produces the rules (CARs).

Let *Fk* denote the set of *frequent k-ruleitems*. Each element of this set is of the following form: <(*condset*, *condsupCount*), (*y*, *rulesupCount*)>. Let *Ck* be the set of candidate *k-ruleitems*.

The first phase of CBA-RRG algorithm is given in Figure 1 which is run on the two-third portion of the training data.

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

Figure 1: The CBA-RRG algorithm

Line 1-3 represents the first pass of the algorithm. It counts the item and class occurrences to determine the frequent 1-*ruleitems* (line 1). From this set of 1-*ruleitems*, a set of CARs (called *CAR*1) is generated by *genRules* (line 2). *CAR*1 is subjected to a pruning operation (line 3) (which can be optional). Pruning is also done in each subsequent pass to *CARk* (line 15). The function *pruneRules* uses minimum confindence (say *minConf*) defined by the user. It prunes a rule as follows: If rule *r*'s confidence is less than *minConf*, then rule *r* is pruned. This pruning can cut down the number of rules generated substantially.

For each subsequent pass, say pass *k*, the algorithm performs 4 major operations. First, the frequent *ruleitems Fk*-1 found in the (*k*-1)*th* pass are used to generate the candidate *ruleitems Ck* using the *condidateGen* function(line 5). It then scans the database and updates various support counts of the candidates in *Ck* (line 6-12). After those new frequent *ruleitems* have been identified to form *Fk* (line 13), the algorithm then produces the rules *CARk* using the *genRules* function (line 14). Finally, rule pruning is performed (line 15) on these rules.

The *candidateGen* function is similar to the function *Apriori-gen* in algorithm Apriori. The *ruleSubset* function takes a set of candidate *ruleitems Ck* and a data case *d* to find all the *ruleitems* in *Ck* whose *condsets* are supported by *d*. This and the operations in line 8-10 are also similar to those in algorithm Apriori. The difference is that we need to increment the support counts of the *condset* and the *ruleitem* separately whereas in algorithm Apriori only one count is updated. This allows us to compute the confidence of the *ruleitem*. They are also useful in rule pruning. The final set of class association rules is in *CARs* (line 17). Those remaining rules after pruning are in *prCARs* (line 18).

The second phase generates the remaining association rules. Before describing the algorithm in formal definition, lets take a look what we are going to do by an example. Say, we have the following training examples:

A	B	C	Target classification
a1	b1	c1	Yes
a1	b1	c2	Yes

a2	b2	c1	No
a2	b2	c2	No

At first we will fix a *satisfactoryConfidence*. Say it is 50%. Then we will generate one rule from each training example. So, at first step we have 4 rules. They are like these:

R1: A=a1,B=b1,C=c1=>yes

R2: A=a1,B=b1,C=c2=>yes

R3: A=a2,B=b2,C=c1=>no

R4: A=a2,B=b2,C=c2=>no

Note that all 4 rules have confidence 100%. These rules are enqueued in a queue (say it is *q*). Now dequeue a rule from *q* and remove one attribute constraint at a time. If R1 is dequeued then the 3 rules will be constructed by removing one attribute constraint at a time:

R11: A=a1,B=b1=>yes

R12: B=b1,C=c1=>yes

R13: A=a1,C=c1=>yes

Now enqueue the newly constructed rules in *q* that have confidence greater than or equal to *satisfactoryConfidence* and go on in this way.

So, the second phase of CBA-RRG looks like this which is run on the two third portion of the training data:

```
1. satisfactoryConfidence = 0.5;
2. ruleList = $\Phi$;
3. q = $\Phi$;
4. for each record rec $\in$ training example
5.         r = constructRule(rec);
6.         ruleList = ruleList $\cup$ r;
7.         enqueue(q,r);
8.while (q is not empty)
9.         r = dequeue(q);
```

```
10.     for each attribute $A \in r$
11.         $r2$ = constructRule2($A$, $r$);
12.         if (confidence of $r2 \geq$ satisfactoryConfidence and $r2 \notin$ ruleList)
13.             $ruleList = ruleList \cup r2$;
14.             enqueue(q,$r2$);
```

satisfactoryConfidence and *q* are described earlier. *ruleList* is a list that will contain the generated CARs. Line 1-3 represents initialization. Line 4-7 describes how training examples having confidence greater than or equal to *satisfactoryConfidence* are directly converted to CARs. *ConstructRule* function (line 5) serves this purpose in a way described earlier. *enqueue* function enqueues rule *r* into queue *q*. Line 8-14 generates rules by removing one attribute at a time from the rules found by dequeuing *q*. *constructRule2* function (line 11) is doing a major task by constructing rule *r2* from *r* by removing attribute *A*. *constructRule2* function also calculates the confidence of rule *r2*.

finally the merging of rules from phase 1 and 2 is done by:
$prCARs = prCARs \cup ruleList$;

The third phase calculates the weight of each generated rule. To do this we will include two attributes *numCorrect*, *numMiscorrect* to each rule. The value of these attributes is calculated in this phase by employing a validation test. The corresponding algorithm looks like this which is run on the rest-one-third portion of the training data :

```
1. for each rule $r \in prCARs$ do
2.     $r.numCorrect = 0$;
3.     $r.numMiscorrect = 0$;
4. end
5. for each rule $r \in prCARs$ do
6.     for each record $rec \in$ training example do
```

```
7.          if( r covers rec)then
8.               if( r correctly classifies rec)then
9.                    r.numCorrect++;
10.              else
11.                   r.numMisCorrect++;
12.    end
13. end
```

Line 1-4 represents initialization. Line 5-13 describes how each attribute *numCorrect* and *numMisCorrect* of each rule is calculated by scanning the training examples (the rest one third portion) which is quite self-descriptive.

Classifier Construction

prCARs still contains a lot of rules. They all will not be used in the classifier. The classifier construction algorithm looks like this:

```
1. finalRuleSet = $\Phi$;
2. dataSet = D;
3. sort(prCARs);
3. for each rule $r \in$ prCARs do
4.     if r correctly classifies at least one training example $d \in$ dataset then
5.          remove d from dataset;
6.          insert r at the end of finalRuleSet;
7. end
```

Lines 1-2 are for initialization purpose. *finalRuleSet* is a list that will contain rules that will be used in the classifier. *sort* function (line 3) sorts *prCars* in descending order of confidence, weight(that is, *numCorrect* – *numMiscorrect*), support and rule length. Lines 4-6 takes only those rules in the *finalRuleSet* which can correctly classify at least one traing example. Note that the insertion in *finalRuleSet* ensures that all the rules of *finalRuleSet* will be sorted in descending order of confidence, weight (that is *numCorrect* – *numMiscorrect*), support and rule length.

When a new test example is to be classified, classify according to the first rule in the *finalRuleSet* that covers the test example.

Advantages

- This approach is better than CBA-RRG in this respect that if two rules have same confidence and covers a test case and gives controversial classification, then we can choose the better rule on the basis of higher weight which should improve the accuracy of the classifier.

Disadvantages

- The algorithm has computational complexity $O(k2^n)$ in the worst case, where k = number of records in the dataset and n = number of attributes in each record.

Open Issues

- What will be the value of *minsup* and *minconf* ?
- What will be the value of *satisfactoryConfidence*?
- Can we impose a limit on the generation of rules at first and second phases? What will be that limit? Can we make that limit adaptive?
- Can we take alternate measures of weight? Can the make the weight adaptive?

7. Modified CBA

Basic Idea:

For constructing the final classifier from the set of generated rules, CBA adopts the following algorithm:

```
Sort the set of generated rules using some priority criteria
For each rule taken in order
    If the rule classifies at least one remaining example correctly
        Insert that rule in the final classifier
        Remove all examples covered by that rule
```

Now the following case can be considered as an example:
A=a1 ^ B=b1->class1 [classifies 200 examples correctly with 100% confidence]
A=a1->class1 [classifies 230 out of 250 examples correctly with 92% confidence]

It is evident that the high accuracy of the second rule is only due to first rule. If the first rule is taken in the classifier and the examples covered by that rule are removed, then the second rule will classify only 30 remaining examples out of 50 with 60%confidence.So this rule should not be included in the final classifier. But according to the database covering scheme used by CBA, this second rule will be included because it can classify at least 1(actually 30) examples. Ideally, this rule should be rejected and other rules should be given chance to cover the 50 examples covered by this rule.

Algorithm

Rule generation and rule sorting same as CBA

Classifier Construction:

```
For each rule taken in order
    Find the number of remaining examples the rule covers c
    Find the number of remaining examples the rule correctly classifies d
    If d is at least 1
```

```
        Find the percentage p of remaining correct classification c*100/d
        If  p is at least some threshold
            Take the rule in the classifier and remove examples
            covered by the rule
        Else ignore the rule
```

This classifier ensures that a rule will be included in the final classifier only if it classifies at least 50% of remaining instances that it covers.

Problems:

Practically this did not work out as it seems theoretically. Accuracy went down. This can be explained from the theoretical background as follows:

- In this process, there is always a chance that we over fit the data. A rule that was good in the overall dataset should not be rejected just because some of it's examples have been covered and it has become useless.
- If smaller number of rules are generated, then selecting rules in this way can leave the dataset uncovered.

8. A Scalable Classifier

Basic Idea:

Database coverage is an issue that associative classifiers must deal with. It is a highly expensive operation in terms of processing time since in worst case the complexity is O(rd) where r is the total number of rules used in coverage and d is the number of examples in the dataset. Since the number of examples in the dataset is a constant, if the database can be covered with a smaller number of rules then the running time of this phase will be substantially reduced. Then these smaller numbers of covering rules can be used to generate more rules which can be used in the classification stage.

The second benefit is if a smaller number of rules are selected in a lower level, the number of rules generated in the next levels can be reduced by huge amount since only the selected rules will be used for further generating the bigger rules.

So the possible benefit here is twofold. It can reduce both coverage time and rule generation time.

So the idea was to cover the dataset when the number of rules in hand is relatively small. For example, an association miner may generate n1 number of rules of size 1, n2 number of rules of size 2, n3 number of rules of size 3 etc. In general, we have n3>>n2>>n1 because of combinatorial explosion. If we can cover the dataset using some rules from level 1 (say n1' rules where n1'<n1), the complexity of database coverage will be O(n1d) which can be much smaller than O(rd) because r>>n1. After this, the n1' rules are used as level 1 rules to generate all subsequent rules. And the number of these rules will also be much smaller because (n1-n1') rules were discarded during database coverage. Finally, all generated rules are used in classification.

Algorithm:

```
L1=find_frequent_1_itemset(D);
Sort L1 according to some criteria
Set cover count of each example in the training set to 1

Rule Set=empty;
Until the database is empty
    For each rule in L1
        If the rule classifies at least one remaining example
            Insert that rule in the Rule Set
        For each example covered by that rule
            Increase the cover count of that example by 1
        Remove all examples from database whose cover count=Some
constant p
```

Using Rule Set as level 1 rules, generate rules using APriori for the next levels and add them to Rule Set

Sort Rule Set according to some criteria and Use this sorted Rule Set as classifier for classification of test data set

Problem:

1. Determining the cover count constant p is a major problem. If we choose p to be small, then only a small number of rules may be selected after database coverage. When these rules are used to generate subsequent rules, total number of rules may be too small to produce good accuracy. Again, if we choose p to be large, then many rules may be selected in dataset coverage and the advantage of running time reduction diminishes.

2. Some rules of level 1 that appear to be bad can produce great rules later. So if these rules are discarded after database coverage, some very important rules are bound to be missed.

3.9 Lazy RRG

Basic Idea:

The basic problem of RRG when implemented in the above ways is that the rules generated by RRG do not generalize well over the data. When a classifier is constructed from these rules, it performs well over training data but in almost of the cases, it fails to cover a test instance let alone classify it correctly. This problem is illustrated below with an example:

Say, we have a training instance

A	B	C	D	E	Class
a1	b1	c1	d1	e1	class1

and a test instance

A	B	C	D	E	Class
a1	b1	c2	d1	e2	?

Assume that the following rules are 2 good rules:

A=a1 ^ B=b1→Class=class1

C=c1 ^ E=e1→Class=class1

RRG proceeds as follows:

A=a1 ^ B=b1 ^ C=c1 ^ D=d1^E=e1→Class=class1

Now it will create 5 rules each of length 4 by leaving one antecedent a time

B=b1^C=c1^D=d1^E=e1→Class=class1

A=a1^C=c1^D=d1^E=e1→Class=class1

A=a1^B=b1^D=d1^E=e1→Class=class1

A=a1^B=b1^C=c1^E=e1→Class=class1

A=a1^B=b1^C=c1^D=d1→Class=class1

Support and confidence of each rule will be calculated. A rule will remain only if it has higher confidence than a pre-specified constant (we name it "satisfactory confidence").Now each rule of length 4 will similarly produce 4 rules of length 3 and so on. A rule of smaller length will be produced only if at least one of it's parent rule has more than satisfactory confidence. It may happen that a larger parent rule has small confidence and so it is pruned. This prevents the generation of it's children rules which may be very good. Again, it is easily understandable that smaller rules(rules with small number of antecedents) are very general and particularly effective for classifying test instances. So if we have lot of good specific rules of high length but fail to generate smaller rules, we may suffer from the problem of lack of generalization of training data.

Moreover, since in this approach, there is no support pruning, an excessively large number of rules can be produced because of combinatorial explosion. A rule of length l can give rise to up to 2^l rules in the worst case. So RRG also takes a long time to run.

The above two problems can be solved by noting the structure of test instance. In the above example, see that any rule that has C=c1 or E=e1 as an antecedent is meaningless because then it cannot cover the test instance given. So instead of taking the full training example to create the first rule A=a1^B=b1^C=c1^D=d1^E=e1→Class=class1, we can delete these 2 conditions and take the first rule as A=a1^B=b1^D=d1→Class=class1. Now in this way, we can make a rule for each test instance and for each training example. The set of such rules will then be used to generate more rules by the "remove one antecedent" method.

Algorithm:

Rule Generation:

```
Rule Set= empty;

for each test instance ti
          for each training instance (d1,d2,….,dn,y) where d1,d2,…..,dn are values for
          the n attributes and y is the class,
                 create a rule of the form D1=d1^D2=d2^…..^Dn=dn→class=y
                 delete all Di=di from the above rule if the value of i'th attribute of test
                 instance ti is not equal to di for i=1,2,……n
                 if this rule was not previously added and
                 confidence>satisfactoryConfidence
                     insert this rule in the Rule Set
insert all rules from Rule Set in a queue

while queue is not empty
    R=front rule of queue
    for each attribute in R
        remove the attribute and create a new rule using the remaining attributes
        if that rule was not obtained previously and it has confidence greater than
        satisfactory confidence, add this rule to Rule Set
```

Classification:

```
    Prune the Rule Set based on some parameters
    Sort the Rule Set according to some criteria
```

Classify a test instance by the first rule from the sorted rule set that covers the test instance

Property 1:

The above process prunes only unnecessary rules.

Property 2:

We can miss some good quality rules. But no bad quality rule can **ever** be produced at any level of rule generation.

Experimental Facts:

There are mainly 2 parameters here. One is satisfactory confidence. Only rules with confidence higher than satisfactory confidence are produced. So if satisfactory confidence is made higher, smaller number of rules will be produced and if satisfactory confidence is made lower, higher number of rules will be produced.

The other parameter is “correlation coefficient” introduced in literature previously by some authors. We have found that it is useful in improving accuracy. So we are taking a threshold limit. Rules with correlation less than the threshold are pruned.

We are varying both parameters from dataset to dataset for increasing classification accuracy.

Advantages:

Compared to other approaches, this classifier has the following advantages:

- It is actually a greedy method to directly generate a set of high confidence rules. Other classifiers use association rule mining to generate the set of rules. As a result, a lot of mined rules may have low confidence and so they get pruned later. But in this novel rule generation algorithm, we generate directly a subset of the best rules.
- Support threshold plays a critical role in associative classification. All other approaches use minimum support threshold pruning. A rule will not be generated if it does not have minimum support although it may have high confidence or correlation or other desirable properties. So the other approaches may miss some good rules. But this rule generation process is completely support less. No rule is pruned based on minimum support. So we can hope to get some good quality low support but high confidence rules here.
- Exceptional rules are in general low supported. So this method helps to generate exceptional rules.
- Setting a value for support threshold is a difficult task for user. Since there is no support threshold here, there is less trouble for user.
- No bad rule (low confidence) is ever generated. And moreover, all produced rules will classify at least one test instance. So no useless rule is also generated.

Disadvantages:

The classifier has the following disadvantages:

- It can not generate all high confidence rules. Say, if the rule A=a1^B=b1^C=c1→class1 is highly confident, A=a1→class1 is also highly confident but none of the rules A=a1^B=b1→class1, A=a1^C=c1→class1 is highly confident, then the last two rules will be pruned and so the rule A=a1→class1 will not be generated.
- Support threshold plays an important role in making the rule mining process tractable. A lot of rules get pruned because they don't have minimum support threshold which helps to reduce the running time of rule generation phase. But in this case, since there is no support pruning, the number of rules generated can be very large especially in some datasets.
- There is a chance of over fitting.

Data Structure:

The performance of the algorithm depends partly on the method of checking for duplicate rules. We felt that if we check the existing rule List linearly to find whether a new rule has already been produced before, it can be time consuming. So we decided to implement a search tree to store the rules. In linear checking, the complexity is O(rn) where n is the current number of rules in rule List and r is the maximum rule length. But with this search tree, the complexity of checking duplicates is O(r) only.

This data structure can also help to reduce running time from another direction. Say we have two rules

A=a1^B=b1^C=c1→class1

A=a1^B=b1^D=d1→class1

The first rule will produce the following rules:

A=a1^B=b1→class1

A=a1^C=c1→class1

B=b1^C=c1→class1

The second rule will produce the following rules:

A=a1^B=b1→class1

A=a1^D=d1→class1

B=b1^D=d1→class1

Now, note that the rule A=a1^B=b1→class1 is produced from both parent rules. Using the tree, we can stop generation of duplicate rules because in the tree, the parent rules will share the same common prefix in the tree.

Chapter 4

Experimental Results

4.0 Introduction

To evaluate the accuracy of our proposed algorithms, we have performed an extensive performance study. In this section, we report our experimental results on comparing RRG, CBA-RRG, weighted CBA-RRG, and ACN against three popular classification methods: CBA, C4.5 and CMAR.

4.1 Experimental Setup

4.1.1 CBA-RRG

In the experiments, the parameters of the six methods are set as follows.

All C4.5 parameters are default values. We test both C4.5 decision tree method and rule method. Since the rule method has better accuracy, we only report the accuracy for rule method.

For CBA, we set support threshold to 1% and confidence threshold to 50% and disable the limit on number of rules. Other parameters remain default.

For CMAR, the support and confidence thresholds are set as same as CBA. The database coverage threshold is set to 4 and the confidence difference threshold to 20%.

For RRG, CBA-RRG, weighted CBA-RRG *minConf* is set to 50%. *satisfactoryConfidence* was set to 50%. Maximum no. of rules in a level was set to 30,000 in CBA. Discretization of continuous attributes is done using the DMII tool found from the site of National University of Singapore (NUS).

The accuracy of each dataset is obtained from 10-fold cross-validations. We used 8 datasets from UCI ML Repository (Merzand Murphy 1996) for the purpose. We use C4.5's shuffle utility to shuffle the data sets.

4.1.2 Result

The experimental result is shown in the following table:

Dataset	CBA	CMAR	C4.5	RRG	CBA-RRG	Weighted CBA-RRG
Pima	72.9	75.1	75.5	**76.82**	73.958	74.3489
Iris	94.7	94	95.3	92	**95.333**	94
Heart	81.9	82.2	80.8	75.185	81.9	**82.22**
Glass	**73.9**	70.1	68.7	73.3644	**73.9**	73.364
tic-tac	**99.6**	99.2	99.4	65.344	**99.6**	**99.6**
wine	**95**	**95**	92.7	76.404	**95**	94.9438
Austral	84.9	86.1	84.7	57.68	84.8	**86.5217**
Diabetes	74.5	75.8	74.2	**77.99**	73.307	73.046
Average	84.675	84.6875	83.9125	74.34843	84.72475	**84.75555**

Table 4.1

Column 1: It lists the name of 8 datasets.

Column 2: It shows CBA's accuracy by 10-fold cross-validations using the original datasets.

Column 3: It shows CMAR's accuracy by 10-fold cross-validations using the original datasets.

Column 4: It shows RRG's accuracy by 10-fold cross-validations using the original datasets.

Column 5: It shows CBA-RRG's accuracy by 10-fold cross-validations using the original datasets.

Column 6: It shows weighted CBA-RRG's accuracy by 10-fold cross-validations using the original datasets.

Table2 shows a comparison of C4.5, CBA and CBA-RRG on average number of rules generated by them. From the table, it is apparent that C4.5 generates a very small number of rules. CBA generates more rules than C4.5. CBA-RRG generates more rules than CBA because CBA-RRG generates all the rules generated by CBA and augments those rules with some high-confident-low-support rules.

dataset	C4.5	CBA	CBA-RRG
pima	27.5	45	66.6
iris	5.3	5	7.3
heart	18.9	52	58.9
glass	27.5	27	33.6
tic-tac	0.6	8	11.4
wine	7.9	10	18.3
austra	13.5	148	162.6
german	29.5	172	204.2
zoo	7.8	7	16.5
diabetes	27.6	57	71.8
average	16.61	53.1	65.12

Table 4.2

4.1.3 Comparison

The RRG approach generates more specific rules. It fails to generate more general rules in some datasets like tic-tac, wine, austra etc. For this reason, when test cases are tested for classification, it is high likely that there exists no rules in the classifier that can cover the test case. Eventually we are forced to classify the test case in a default class which leads to a poor accuracy in these datasets.

In CBA-RRG approach we have overcome the limitation of RRG algorithm which could not generate more general rules. Here we have both high-support-moderate-confidence rules as well low-support-high-confidence rules. So we have a larger amount of high quality rules which should improve the accuracy of the classifier. This is why the average accuracy of CBA-RRG is better than that of RRG.

The weighted CBA-RRG approach is better than CBA-RRG in this respect that if two rules have same confidence and covers a test case and gives controversial classification, then we can choose the better rule on the basis of higher weight. So, the average accuracy of weighted CBA-RRG is better than that of CBA-RRG.

4.2.1 ACN

In the experiments, the parameters of the six methods are set as follows.

All C4.5 parameters are default values. We test both C4.5 decision tree method and rule method. Since the rule method has better accuracy, we only report the accuracy for rule method.

For CBA, we set support threshold to 1% and confidence threshold to 50% and disable the limit on number of rules. Other parameters remain default.

For CMAR, the support and confidence thresholds are set as same as CBA. The database coverage threshold is set to 4 and the confidence difference threshold to 20%.

For ACN, the correlation threshold is set to .2,minconf is set to 50%, remaining accuracy is set to 55%.Discretization of continuous attributes is done using the DMII tool found from the site of National University of Singapore (NUS). The accuracy of each dataset is obtained from 10-fold cross-validations. We used 13 datasets from UCI ML Repository (Merzand Murphy 1996) for the purpose. We use C4.5's shuffle utility to shuffle the data sets.

4.2.2 Result

The experimental result is shown in the following table:

	CAN	CBA	CMAR	CPAR
Diabetes	75.5	74.5	75.8	75.1
Led7	72.1	72.1	72.5	73.6

Pima	73.4	72.9	75.1	73.8
Tic-tac	99.3	99.6	99.2	98.6
Wine	95.5	95	95	95.5
Glass	72.9	73.9	70.1	74.4
Iris	94.7	94.7	94	94.7
Heart	83.7	81.9	82.2	82.6
Vehicle	69.7	68.9	68.8	69.5
Zoo	96	96.8	97.1	95.1
Sonar	79.3	77.5	79.4	79.3
Lymph	83.1	81.9	83.1	82.3
Austra	85.9	85.4	86.1	86.2
Average	83.2	82.7	83.0	83.1

Table 4.3

Column 1: It lists the name of 13 datasets.

Column 2: It shows ACN's accuracy by 10-fold cross-validations using the original datasets.

Column 3: It shows CBA's accuracy by 10-fold cross-validations using the original datasets.

Column 4: It shows CMAR's accuracy by 10-fold cross-validations using the original datasets.

Column 5: It shows CPAR's accuracy by 10-fold cross-validations using the original datasets.

4.2.3 Comparison

ACN achieves good accuracy compared to other state-of-the-art classification algorithms. Introduction of negative association rules result in a much larger rule set which contain better and more accurate rules.

Chapter 5

Conclusion

5.1 Summery

In our thesis eight different algorithms have been proposed. The first algorithm is "Associative Classifier with Negative Rules (ACN)" which utilizes negative rules along with positive rules, which gives the classification system more representational capability. More semantics can be expressed by this classifier than traditional classifiers. Experimental results show that this algorithm outperforms traditional associative classifiers on some datasets.

The second algorithm is "Level Adaptive Classifier (LAC)". This is an attempt to make the maximum rule length of the generated rules adaptive depending on the semantic structure of the dataset. This reduces running time considerable and achieves comparable accuracy with other algorithms.

The "Reverse Rule Generation (RRG)" algorithm is an extraordinary algorithm which generates rule in the reverse manner. Initially the training set is taken as the rule set. Then each rule is decomposed by leaving out each attribute iteratively and inserting the rule in the rule set if the has confidence greater than a pre-specified threshold *satisfactoryConfidence.* Most of the association rule mining algorithm uses support pruning, which results in the pruning of some good quality rule with low support but high confidence. The RRG algorithm doesn't use support pruning, so it generates all high confidence rules. In fact it can be proved that RRG generates the complete set of high confidence rules.

"CBA-RRG" algorithm is an improvement of the previous algorithm. Though RRG generate all high confidence rules it takes a lot of computing time to generate all rules. So in this approach we have designed a hybrid algorithm that used both the advantages of the CBA and RRG algorithms, at the same time eliminates the limitation of both the algorithms. This algorithm achieves better average accuracy than both CBA and RRG algorithms.

The next approach is the "Weighted-CBA-RRG" algorithm. The only difference with the CBA-RRG algorithm is that the generated rules are validated on a validation set to find the value of (correctly classified – misclassified validation records). This weight is used to sort the rules in the classifier. From experimental result we found that this algorithm gives better accuracy than the CBA-RRG algorithm.

The sixth algorithm is the "Modified CBA". The CBA algorithm coves dataset during classifier building. But the covering technique ignores the fact that the confidence of a rule ***r*** on the uncovered dataset can be adversely effected by the inclusion of an another rule ***q*** in the final rule set where there is a moderate number of dataset records covered by both ***q*** and ***r***.

The next approach is named as "A Scalable Classifier". In this approach relatively small number of rules is used to cover the database. Then these smaller numbers of covering rules is used to generate more rules which can be used in the classification stage. This reduces the required database covering time. The second benefit is if a smaller number of rules are selected in a lower level, the number of rules generated in the next levels can be reduced by huge amount since only the selected rules will be used for further generating the bigger rules. So the benefit here is twofold. It can reduce both coverage time and rule generation time.

The last approach is the "Lazy RRG (LRRG)" algorithm. The basic problem of RRG when implemented in the above ways is that the rules generated by RRG do not generalize well over the data. To solve this problem we have proposed the LRRG algorithm. The only difference is in the initial rule set construction approach. Here one rule is generated for each pair of <training set ***i***, test set ***j***> for all ***i***, ***j***, by removing from ***i*** all the attribute values that are not in ***j***. This algorithm gives better accuracy that the RRG algorithm.

As a long term research plan, our goal is to develop a CAR mining algorithm that is both time and memory efficient, mine rules that are of high quality, develop a more effective rule pruning approach and build a more robust classifier. The algorithms presented in this report can be viewed as a preliminary step towards that goal.

5.2 Future work Direction

Following are some suggestions to extend our work:

1. In the ACN algorithm we have used only rules with only one negated literal. Rules of different forms e.g. with more than one negated literal or negated class level etc. can be used.
2. The database covering technique in the algorithms can be changed.
3. In the LAC algorithm the classifier building phase at each step adds additional cost. So we can generate rules up to say 2 or 3 length with out building the classifier then apply LAC. This will reduce running time of the algorithm.
4. The stopping criteria of the LAC algorithm can be changed or made adaptive depending on the dataset.
5. In LAC a rule set of size (length of rules) n is accepted or rejected as a whole. But it is quite intuitive that this can lead to fall in accuracy. Because typically not all rules in a level are bad. Some are good and some are bad. A validation test can be applied to find out bad rues and prune only them.
6. In weighted CBA-RRG, we used *numCorrect-numMiscorrect* as weight of each rule. A better estimate for weight can be established.

References

[1] Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning*. Morgan Kaufmann.
[2] Cohen, W. W. 1995. Fast Effective Rule Induction. *Machine Learning: Proceedings of the Twelfth International Conference (ICML-95).* p.115-123. Morgan Kaufman, 1995.
[3] Agrawal, R., Imielinski, T., Swami, A. 1993. Mining Association Rules between Sets f Items in Large Databases. *Proc. Of the ACM SIGMOD Conference on Management of Data.* Washington, D.C., p. 207-216
[4] W. Li, J. Han, and J. Pei. CMAR: Accurate and efficient classification based on multiple class-association-rules. In *ICDM'01*, pp. 369-376, San Jose, CA, Nov. 2001.
[5] B. Liu, W. Hsu, and Y. Ma. Integrating classification and association rule mining. In *KDD'98*, pp. 80-86, New York, NY, Aug. 1998
[6] Liu, B., Ma, Y., and Wang, C.-K. 2001. Classification Using Association Rules: Weaknesses and Enhancements. In Grossman, R. L., et al, (eds), *Data Mining for Scientific and Engineering Applications*. Kluwer Academic Publishers, p.591-601.
[7] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In VLDB'94, pp. 487-499, Santiago, Chile, Sept. 1994.
[8] J. Han, J. Pei and Y. Yin. “Mining frequent patterns without candidate generation,” In Proc. 2000 ACM-SIGMOD Int. Conf. on Management of Data (SIGMOD’ 00), pages 1-12, Dallas, TX, May 2000.
[9] X. Yin and J. Han, “CPAR: Classification based on Predictive Association Rules,” Proc. Of SIAM Int. Conf. on Data Mining (SDM'03), pp. 331-335, San Francisco, CA, 2003.
[10] E.Baralis, P. Gazza, “A lazy approach to pruning classification rules,” Proc. IEEE Int. Conf. on Data Mining (ICDM’04), pages 35-42, 2002.
[11] Adriano Veloso, Wagner Meira Jr., Mohammed J. Zakib, “Lazy Associative Classification,” Proc. IEEE Int. Conf. on Data Mining (ICDM’06)