SEARCHING FOR THE OPTIMAL ORDERING OF CLASSES IN RULE INDUCTION

SEZIN ATA
B.S., Computer Engineering, IŞIK UNIVERSITY, 2009
B.S., Mathematics Engineering, IŞIK UNIVERSITY, 2009

Submitted to the Graduate School of Science and Engineering
in partial fulfillment of the requirements for the degree of
Master of Science
in
Computer Engineering

IŞIK UNIVERSITY
2012

IŞIK UNIVERSITY
GRADUATE SCHOOL OF SCIENCE AND ENGINEERING

SEARCHING FOR THE OPTIMAL ORDERING OF CLASSES IN RULE INDUCTION

SEZIN ATA

APPROVED BY:

Assoc. Prof. Olcay Taner Yıldız (Thesis Supervisor)	Işık University	_____
Assist. Prof. Ali İnan	Işık University	_____
Assist. Prof. Mehmet Önal	Işık University	_____

APPROVAL DATE: ..../..../....

SEARCHING FOR THE OPTIMAL ORDERING OF CLASSES IN RULE INDUCTION

Abstract

In this thesis, we work on rule induction algorithms, basically Ripper. These algorithms solve a $K > 2$ class problem by transforming it into a sequence of $K-1$ two class problems. As a heuristic, these algorithms learn classes in the order of increasing prior probabilities. Although the heuristic works well in practice, there is much room for improvement. We propose two algorithms for that purpose.

The first algorithm, namely Forward Ordering Search (FOS) starts with the ordering heuristic provided and searches for better orderings by swapping consecutive classes. For a dataset with K classes, the ordering space will be as large as $K!$. Since FOS is an example of Steepest Ascent Hill Climbing (Gradient Search), starting with the heuristic ordering will only give local maximum in the search space. In order to improve the performance, we use 10 random initial orderings as in Random-Restart (Steepest Ascent) Hill-Climbing. The best performance between 10 random initial orderings is the result of Random-Restart FOS.

The second algorithm, namely Pairwise Error Approximation (PEA), transforms the ordering search problem into an optimization problem and uses the solution of the optimization algorithm to extract the optimal ordering. In this algorithm, the number of random orderings to construct the optimization problem is a parameter and we try several values of this parameter to see the effect on the performance.

We compare our algorithms with the original Ripper on 13 datasets from UCI repository [1]. Experimental results show that, our algorithms produce rule sets that are significantly better than those produced by Ripper proper in general and the number of rules and conditions of the produced rule sets are comparable with Ripper proper. Even though the accuracy of Random-Restart FOS is better than FOS, the time complexity of the algorithm is far worse than FOS. The average error estimation results of PEA promote the consistency of our pairwise assumption and show the relationship between the accuracy and the number of random orderings to extract the optimal ordering.

KURAL ÇIKARIMDA OPTİMAL SINIF SIRALAMASINI ARAMA

Özet

Bu tezde, CN2 ve Ripper kural çıkarım algoritmaları üzerinde çalıştık. Bu algoritmaların ortak özelliği $K > 2$ sınıflı veri kümelerini sınıflandırırken, $K-1$ adet 2 sınıflı probleme çevirerek sınıflandırmalarıdır. Bulgusal yaklaşıma göre, bu algoritmalar sınıfları, artan önsel olasılıklarına göre öğrenirler. Biz de çalışmamızda, kural çıkarım algoritmalarının sınıf sıralamalarına bağlı olarak performanslarının nasıl değişeceğini araştırırız. Bu amaçla, iki algoritma sunarız.

Sunulan ilk algoritma, FOS (ileriye doğru-sıralama arama algoritması), ilk olarak bulgusal yaklaşımın sıralamasıyla başlar. Yan yana sınıfların yer değişimleri ile oluşturulmuş sıralamaları, daha iyi performans elde edildiği sürece, iteratif şekilde karşılaştırır. Bu arama En Dik Tırmanış Algoritması' na bir örnek olduğu için tüm arama uzayında ancak yerel bir başarı noktası bulacak şekilde gerçekleşir. Tüm arama uzayı, $K > 8$ sınıflı veri kümeler için 8!' den büyük bir uzaydır. Bu nedenle, performansı arttırmak için Rasgele-Başlangıç Dik Tırmanış Algoritması' nda olduğu gibi, rasgele 10 farklı başlangıç sıralamasıyla FOS algoritmasını çalıştırırız. Bu sonuçların en iyisi, Rasgele-Başlangıç FOS' un sonucunu belirler.

Sunduğumuz ikinci algoritma olan İkili Hata Yaklaşıklaması Algoritması, sıralama arama problemimizi, sıralamaların sınıf ikililerini kullanarak, optimizasyon problemine çevirir. Problemin çözümünü optimal sıralamayı bulmak için kullanırız. Optimizasyon probleminin parametreleri olarak rasgele sıralamalar üretiriz ve çeşitli sayıda rasgele sıralamalarla, sıralama sayısının performansa etkisini gözlemleriz.

Algoritmalarımızın sonuçlarını Ripper kural çıkarım algoritmasıyla 13 veri kümesi üzerinde karşılaştırırız. Elde ettiğimiz sonuçlar genel olarak, bulduğumuz sıralamaların performans ve karmaşıklıkları açısıdan daha iyi kural kümeleri oluşturduğunu gösterir. Ayrıca Rasgele-Başlangıç FOS algoritmasının performansının FOS' tan iyi olmasına rağmen, algoritmanın karmaşıklığının FOS' tan kat kat fazla olduğunu gözlemleriz. Son olarak, PEA algoritması için hesapladığımız ortalama kestirim hatası sonuçları, algoritmayı oluşturmamıza neden olan varsayımımızın tutarlılığını destekler ve dogru sonuçlarla rasgele sıralama sayısı arasındaki ilişkiyi gösterir.

Acknowledgements

First of all, I would like to thank to my supervisor, Assoc. Prof. Olcay Taner Yıldız for his kindly advices and guidance. He also made available to use the tools implemented in library ISELL and this helped me all along my thesis work and I believe it will. I am very grateful to my family and my friends for their motivation and I also thank to Doğan Kırcalı for his critical MATLAB support.

To my family. . .

Table of Contents

List of Figures

List of Symbols

C_i	Class i
d	Number of attributes
D	Training Sample
$e_{\pi(j)\pi(k)}$	Expected error of the pair $\pi(j)\pi(k)$
E_{π_i}	Error Rate of Ordering π_i
$\hat{E}_{\pi_i}$	Estimated Error of Ordering π_i
$\overline{E}_{total}$	Average Total Estimation Error
G	Grow Sample
K	Number of classes
n	Sample Size
n_{c_i}	Number of instances of Class i in the training set
N	Number of Random Orderings
P	Prune Sample
r	rule
RS	Ruleset
V	Validation Sample
π_H	Heuristic ordering
π_i	Ordering with index i
$\pi(j)\pi(k)$	Pair, class at index j coming before the class at index k in ordering π

List of Abbreviations

ACO	Ant Colony Optimization
FOS	Forward Ordering Search
ILP	Inductive Logic Programming
IREP	Incremental Reduced Error Pruning
MDL	Minimum Description Length
PEA	Pairwise Error Approximation
RIPPER	Repeated Incremental Pruning to Produce Error Reduction
RR-FOS	Random-Restart Forward Ordering Search

Chapter 1

Introduction

In everyday life, people unconsciously have various classification problems in their minds. People, brands, technologies, automobiles, etc. are classified using the data which is collected intentionally or unintentionally. The collected data are processed in human brain among neurons and a classification decision is made. These decisions might be vital in some cases. In a world with numerous data and complicated circumstances, human brain is inadequate to process all that data. Machine learning, a field of computer science, aims to accomplish these classification tasks that the human brain can not cope with ease.

In this thesis, we work on the area of rule induction, specifically the classical form of the rule induction that is called propositional rule induction (attribute-value rule learning) [2]. Propositional rule induction algorithms produce rule sets as the main model. Rule sets contain rules and each rule consists of conditions merged via AND's and classifies class C_i. Rule induction algorithms have various advantages: (i) They are easy to understand and modify. (ii) They learn fast and can be used on very large data sets with a large number of instances. (iii) They do their own feature extraction/dimensionality reduction and can be used on data sets with a large number of features.

The most common application areas of rule induction algorithms are data mining, pattern recognition, bioinformatics, natural language processing, etc. PRISM [3],

JoJo [4], AQ [5], RIPPER [6], PREPEND [7], CN2 [8] are some examples of the propositional rule induction algorithms.

There are two main groups of rule induction algorithms: Separate-and-conquer algorithms and divide-and-conquer algorithms. This thesis is mainly related with the algorithms following separate and conquer strategy [9]. According to this strategy, when a rule is learned for class C_i, the covered examples are removed from the training set. This procedure proceeds until no examples remain from class C_i in the training set.

If we have two classes, we separate positive class from negative class. But if we have $K > 2$ classes, as a heuristic, every class is classified in the order of their increasing prior probabilities, i.e., in the order of their sample size. The aim of this thesis is (i) to determine the effect of this ordering on the performance of the algorithms and (ii) to propose better algorithm for selecting the ordering. We propose two algorithms to find the optimal class ordering.

First algorithm, forward ordering search (FOS), does Steepest Ascent Hill Climbing (Gradient Search) -which is similar to the Best First Search- in the ordering space. It starts from the heuristic's class ordering and at each step, the neighbor ordering that has the smallest error is selected. The algorithm terminates when none of the neighbor orderings has less error than the current best ordering.

Second algorithm, pairwise error approximation (PEA), assumes that the error of an ordering is the sum of $K(K-1)/2$ pairwise errors of classes. We train N random orderings and use the error of them as training data to estimate the pairwise errors. Given the estimated pairwise errors, the algorithm searches for the optimal ordering exhaustively.

In the beginning of the thesis, we work with CN2 and Ripper. Later on, we see that Ripper is usually significantly better than CN2, and we choose Ripper to continue with. In FOS, since the algorithm has a local optima problem, in

order to improve the performance of the algorithm, we run the algorithm with 10 different initial orderings, and called this algorithm Random-Restart FOS.

For PEA algorithm, we investigate if the size of the random set of orderings affect the performance of the algorithm. We train Ripper with $N = 10, 20, 30, \ldots, 100$ random orderings and for each N, the algorithm exhaustively searches for the optimal ordering. After all, we compare the performances of the obtained orderings.

This thesis is organized as follows: We briefly survey rule induction algorithms in Chapter 2. In Chapter 3, we discuss the base algorithm Ripper. In Chapter 4, we discuss our motivation and explain our proposed algorithms FOS and PEA. The experimental results are demonstrated in Chapter 5, and we conclude in Conclusion.

Chapter 2

Rule Induction Algorithms

Rule induction algorithms produce a classification model consisting of rules. The role of the rules is to classify the examples of the dataset, matching up with their class labels. There are two types of rules: Propositional rules and first order logic rules. This thesis deals with propositional rules. Propositional rules contain a conjunction of terms and a class label assigned to an instance that is covered by the rule [9]. The terms are of the form $x_i = v$, $x_i < \theta$ or $x_i \geq \theta$, depending on respectively whether the input feature x_i is discrete or continuous. An example rule containing two propositions for class C_1 is

IF $(x_0 < 1.9)$ **AND** ($x_1 =$ "green") **THEN** Class $= C_1$

where feature x_0 is continuous and feature x_1 is discrete.

A rule set is a list of rules induced by the rule induction algorithms. An example rule set for famous iris problem is:

If $F_3 < 1.9$ **and** $F_4 \geq 5.1$ **Then** Class = "iris-setosa"
Else
If $F_3 < 4.7$ **Then** Class = "iris-versicolor"
Else Class = "iris-virginica"

In this example, rule set contains two rules and each rule classifies one class. Examples which are not covered by neither of the rules, are covered by the default rule as *iris-virginica*.

There are two basic groups of rule induction algorithms: Separate-and-conquer algorithms (also called as sequential covering algorithms) and divide-and-conquer algorithms. Separate-and-conquer algorithms [9] find the best rule that explains the part of the training set recursively. Each iteration, it *separates* the examples (even false positives) those are covered by this rule from the training set. This procedure proceeds until no examples remain.

Divide-and-conquer algorithms (tree induction algorithms) greedily find the best split that separates data according to some predefined impurity measure such as information gain, entropy, Gini index, etc. After *dividing* the examples according to the best split, the algorithms *conquer* the resulted partitions recursively. C4.5 is famous example of tree induction algorithms, which searches for the best split and the best feature at each node with respect to the information gain [10].

Separate and conquer algorithms have some common processes. Figure 2.1 shows a generalized pseudocode, which is valid for almost all separate and conquer algorithms. For each class C_i, the initial rule is determined and the conditions are added to the rule with respect to the search direction. Then, the rule is simplified and this simplification is called pre-pruning. After pre-pruning, the rule is added to the rule set with respect to the search strategy. Finally, constructed rule set is simplified and this simplification is called post-pruning.

```
1 Rule Set SeparateConquer(D)
2   for each class C_i
3     while D contains positive examples
4       Rule = rule with respect to search direction
5       while Rule covers negative examples
6         Grow-Rule(Rule, search direction, D)
7       Pre-Prune(Rule, D)
8       Add Rule to the Rule Set with respect to search strategy
9       Remove examples covered by Rule from D
10    Post-Prune(Rule Set, D)
11  return Rule Set
```

Figure 2.1: Pseudocode for separate and conquer algorithms with dataset D

2.1 Search Direction

2.1.1 Top-down

Top-down search starts with choosing the most-general rule, 'true', as the initial rule which covers all training data. The rule is then specialized by adding conditions. After each specialization, the new rule covers a subset of the previously covered data.

2.1.2 Bottom-up

The bottom-up strategy starts with the most-specific rule which usually covers single example. Each iteration generalizes the rule by removing conditions. After each generalization, the new rule covers new examples plus previously covered data.

2.1.3 Bi-directional

Bi-directional approach is a combined version of top-down and bottom-up strategies. The initial rule might be (i) the most-general rule, (ii) the most-specific rule

or (iii) a random rule, which is neither specific nor general. It allows to use both generalization and specialization operators.

2.2 Search Strategy

2.2.1 Hill-Climbing

Hill climbing is a search technique used in artificial intelligence. It is also used in rule induction algorithms to find the best rule for a given training set. First, an initial rule is defined depending on the *search direction* (top-down or bottom up). After that, the initial rule is improved iteratively. When there is no further improvement for the rule, the search stops. This usually returns a local optima.

2.2.2 Beam search

One who avoids to stick in the local optimum of hill-climbing, might choose beam search for better performance. In addition to considering the best rule, beam search also keeps track of a fixed number of alternatives [9]. While hill-climbing has to decide upon a single improvement at each step, beam search finds the best refinement over all alternative rules.

2.2.3 Best-First

This technique searches for the best rule in the rule space over a training set. It stores all candidate rules after each refinement process of a rule and selects the best candidate rule. It is similar to Beam-Search approach with an infinite beam size.

2.2.4 Stochastic

Stochastic search aims to avoid local optima. For that purpose, it makes more than one refinements to a rule at a time (for higher jumps) and makes random refinement over the rules. The rules which are more promising have higher chance to be improved. Genetic algorithm is an example of stochastic search, it allows refinements over random conditions of the rules by cross-over with another rule.

2.3 Pruning

Pruning methods prevent over-fitting and cope with noisy data.

2.3.1 Pre-pruning

In pre-pruning after construction of a rule, the conditions of the rules are pruned one by one and the condition that causes the most increase in information gain measure is chosen to be removed. Pruning stops when there is no improvement.

2.3.2 Post-pruning

After the completion of the classification model, the rules are pruned if their removal decreases the error on training set and this method is called post-pruning.

2.4 Survey

Table 2.1 demonstrates some separate and conquer algorithms with their search techniques and strategies.

The first application of Ant Colony Optimization (ACO) task is the Ant-Miner algorithm [11]. Ant-Miner uses separate and conquer strategy. Each path constructed by an artificial ant is mapped into a classification rule. Artificial ant

Table 2.1: Examples of Separate and Conquer Algorithms

	Search Strategy				Search Direction			Pruning	
Alg.	Hill Climb.	Beam	Best First	Stoch.	Top	Bottom	Bi-direct	Pre	Post
Ant-Miner				✓		✓		✓	
Ant-Miner+				✓		✓		✓	
AQ	✓	✓			✓				
AQ15	✓	✓			✓				✓
ATRIS				✓			✓		✓
BEXA	✓	✓			✓			✓	✓
CN2	✓	✓			✓			✓	
DLG	✓	✓				✓			
GBAP				✓	✓				
GROW	✓				✓				✓
IREP	✓				✓			✓	✓
JoJo	✓						✓		
LERILS	✓	✓				✓			✓
POSEIDON	✓	✓			✓				✓
PREPEND	✓				✓				
PRISM	✓				✓				
REP	✓					✓			✓
SIA				✓		✓			✓
SWAP-1	✓						✓		✓

starts with an empty rule and iteratively adds term to the rule [12]. After construction of a rule it is immediately pruned. When an ant completes its rule, the amount of pheromone is updated and another ants start to construct their rule, using the new amounts of pheromone to guide its search [13]. There are several versions of this sequential covering algorithm Ant-Miner+ [14], GBAP [15].

Ant-Miner+ is an improved version of Ant-Miner and it defines the environment as a directed acyclic graph. All ants begin in start vertex and walk through their environment to the end vertex, they construct a rule step by step.

AQ [5] algorithm performs top down search strategy and selects a random example. It specializes the most general rule until it still covers the selected example, but none of the negative examples. For that reason, constructed rules of AQ are dependent on specific examples.

AQ15 [5] uses rigid constraints about total number of specializations. It constructs specializations at most the total number of attributes per iteration.

ATRIS [16] combines stochastic local optimization with deterministic local optimization. For stochastic search it uses simulated annealing based algorithms and for deterministic search it performs variants of k-opt algorithms. K-opt is a local search strategy which starts with a tour and improves the tour iteratively by performing a sequence of k-opt moves. Each k-opt move replaces k edges in a tour with k new edges.

BEXA [17] aims, not to exclude potential specializations as avoiding the useless ones. For that purpose it allows dynamic restrictions on the number of constructed specializations: (i) The prevent-empty-conjunctions restriction, prevents conjunctions that cover no positive examples. (ii) The irredundancy restriction, constructs conditions as conjunctions with only a few terms. (iii) The uncover-new-negatives restriction, ensures each newly excluded value or interval to uncover at least one new negative instance.

CN2 [8] specializes conditions via either adding a new conjunctive term or removing a disjunctive element. It chooses a larger search space for the specialization process to examine all specializations of a condition. Thus, CN2 includes the rules which do not fit the training data perfectly as well. CN2 also uses rigid constraints about total number of specializations as AQ15. It constructs only pure conjunctions and pure means an expression containing only elementary atoms [18]. The main goal of these restrictions is to reduce the learning time but the main problem is that they may cause to lose some important specializations and allow potentially useless specializations.

DLG [19] is a variant of AQ family, using generalization.

GBAP [15] uses Ant Colony Optimization as its search technique and is guided by a context-free grammar.

GROW [20] is an alternative method for over-fit and simplify technique of REP. It adds the most promising generalization of a rule to an initially empty theory, instead of removing the most useless rule or condition from the over-fitting theory [21].

I-REP [22] applies the integration of pre-pruning and post-pruning. This algorithm immediately prunes the rule, after learns it and uses prune set for that purpose. During the post-pruning, the algorithm deletes any final sequence of conditions from the rule.

The algorithm JoJo [4] is a standard separate and conquer algorithm which enables to add or relax one term at a time using bi-directional strategy.

LERILS [23] has two parts. (i) It creates a large rule pool from randomly chosen positive examples and, (ii) combines the rules in the rule pool to obtain a final rule set. Both parts use randomized local search with a version of a minimum description length heuristic. For each class, these parts try to find an optimal subset of constructed pool of rules iteratively.

POSEIDON [24] is a standard separate and conquer algorithm using beam-search with top-down strategy.

Examples belonging to the most common class should be handled by the default rule for the efficiency. For that purpose, PREPEND [7] puts the new learned rule before the previously learned rules during the construction of the rule set. Thus, the most general rule will be learned first and will placed at the end of the list as default rule.

PRISM [3] is based on ID3. It induces modular rules to avoid some disadvantages of decision trees. For each class, during the construction process of the rules the training set is restored to its initial state. Hence the classes are considered separately, their order of training is immaterial.

REP [25] divides training data into a growing set and a pruning set. It starts with an initial rule set that over-fits the growing set. Then, it simplifies the rule set

and during pruning it deletes any single condition or any single rule that increases error on the pruning set.

SIA [26] performs genetic algorithm to obtain the best rule set. SIA constructs candidate rules and these rules constitute the generation of the genetic algorithm. During the construction of the rules, random generalizations are applied to the examples in a bottom-up manner. The generation produces new rules, by applying cross-over between the conditions of the two rules randomly.

SWAP-1 [27] uses bi-directional search strategy, it allows conditions to swap in and out. Rules are constructed by these swapping operations of the conditions. Weakest Link pruning strategy is used during the post-pruning of the rules. This strategy starts with an initial rule set and each subsequent rule sets are generated by pruning the rule set from its weakest link, i.e, it is pruned by deleting single rules or single components causing maximum increase in information gain measure. Strategy produces ordered rule sets in decreasing complexities.

Chapter 3

Ripper

In this study, we use Ripper as the base algorithm for searching the optimal class ordering. Ripper, Repeated Incremental Pruning to Produce Error Reduction, is an Irep [22] based rule induction algorithm. It basically learns rules, generates a rule set, optimizes the rule set and finally returns the rule set. The algorithm is formally called RIPPERk since the learned rule set is optimized k times, typically twice [6].

The pseudocode of Ripper is given in Figure 3.1. When there are $K > 2$ classes, the classes of the dataset are increasingly sorted according to their prior probabilities resulting in the permutation, π (Line 1). For each class $\pi(p)$, its examples are considered as positive and the examples of the remaining classes $C_{p+1}, ..., C_K$ are considered as negative (Line 4). Rules are grown (Line 9), pruned (Line 10) and added (Line 16) one by one to the rule set. If the recent rule set' s description length is 64 bits more than the previous rule set's description length rule adding stops and the rule set is pruned (Lines 12-14). The description length of a rule set is the number of bits to represent all the rules in the rule set, plus the description length of examples not covered by the rule set [28]. Ripper uses

$$DescLen = \parallel k \parallel + klog_2\frac{n}{k} + (n-k)log_2\frac{n}{n-k} \tag{3.1}$$

```
1  Ruleset Ripper(D,π)
2    RS = {}
3    for p = 1 to K
4      Pos = π(p), Neg = π(p + 1), ..., π(K)
5      RS_p = {}
6      DL = DescLen(RS,Pos,Neg)
7      while D contains positive samples
8        Divide D into Grow set G and Prune set P
9        r = GrowRule(G)
10       PruneRule(r, P)
11       DL' = DescLen(RS_p + r, Pos, Neg)
12       if DL' > DL + 64
13         RS = PruneRuleSet(RS_p + r, Pos, Neg)
14         return RS
15       else
16         RS_p = RS_p + r
17         Remove examples covered by r from D
18     for i = 1 to 2
19       OptimizeRuleset(RS_p, D)
20     RS = RS + RS_p
21   return RS
```

Figure 3.1: Pseudocode for learning a ruleset using Ripper on dataset D according to ordering π

```
1  Rule GrowRule(D)
2    r = {}
3    while r covers negative examples
4      Use exhaustive search to find best condition c
5      r = r ∪ c
6  return r
```

Figure 3.2: Pseudocode for growing a rule using dataset D

bits to send rule r with k conditions, where n is the number of possible conditions that could appear in a rule and ‖k‖ is the number of bits needed to send the integer k [6]. If there are no remaining positive examples (Line 7) rule adding stops. After learning a rule set, it is optimized twice (Line 18).

Figure 3.2 shows pseudocode of the growing a rule. Learning starts with an empty rule (Line 2), and conditions are added one by one. At each iteration the

```
1  Rule PruneRule(r)
2    improved = True
3    E_best = rvm(r)
4    while improved
5      improved = False
6      for each condition c in r
7        r = r − c
8        E = rvm(r)
9        if ( E ≥ E_best)
10         improved = True
11         remove = c
12         E_best = E
13       r = r ∪ c
14     if improved
15       r = r−remove
16   return r
```

Figure 3.3: Pseudocode for pruning rule r

algorithm finds the condition with maximum information gain on the dataset D (Lines 4) by using the following formula:

$$Gain(R', R) = s(log_2 \frac{N'_+}{N'} - log_2 \frac{N_+}{N}) \tag{3.2}$$

where N is the number of examples, N_+ is the number of true positives covered by rule R and N', N'_+ represent the same descriptions for the candidate rule R'. s is the number of true positives after adding the condition in R [29] [28]. When the best condition is found, we add that condition to the rule (Lines 5). We stop adding conditions to a rule when there are no negative examples left in the grow set (Line 3).

The pseudocode for pruning a rule is given in Figure 3.3. We search for a condition whose removal causes the most increase in rule value metric (Lines 9-12) and if such a condition is found, we remove it (Lines 14-15). Rule value metric is calculated by

```
1 Ruleset PruneRuleSet(RS, Pos, Neg)
2   for each rule r in RS in reverse order
3     DL = DescLen(RS, Pos, Neg)
4     DL' = DescLen(RS − r, Pos, Neg)
5     if DL' < DL
6       RS = RS − r
7   return RS
```

Figure 3.4: Pseudocode for pruning ruleset RS on dataset D

```
1 Ruleset OptimizeRuleset(RS, D)
2   for each rule r in RS
3     Divide D into Growset G and Pruneset P
4     r_replace = GrowRule(G)
5     PruneRule(r_replace, P)
6     r_revise = GrowRule(G,r)
7     PruneRule(r_revise, P)
8     RS_replace = RS - r + r_replace
9     RS_revise = RS - r + r_revise
10    E = Error(RS)
11    E_replace = Error(RS_replace)
12    E_revise = Error(RS_revise)
13    r_min = The rule with min(E, E_replace , E_revise)
14    RS = RS - r + r_min
15  return RS
```

Figure 3.5: Pseudocode for optimization of rule set RS on dataset D

$$Rvm(R) = \frac{p - n}{p + n} \tag{3.3}$$

where p and n are the number of true and false positives in the pruning set. We stop pruning when there is no more improvement in rule value metric (Line 4).

The pseudocode for pruning a ruleset is given in Figure 3.4. We search for a rule whose removal decreases the description length of the rule set (Lines 2-4). If such a rule is found it is removed from the rule set (Lines 5-6).

The pseudocode for optimizing a rule set is given in Figure 3.5. In the optimization phase, two alternatives are grown for each rule (Line 2). The replacement

rule, is grown (Line 4) and pruned (Line 5) starting with an empty rule. The revision rule, is grown (Line 6) and pruned (Line 7) starting with the current rule. These two rules and the original rule are compared and the one with the smallest error on D (Lines 10-13) is selected and replaced with the original rule (Line 14).

Chapter 4

Proposed Algorithms

4.1 Motivation

Propositional rule induction algorithms learn a rule set from a training set. An example rule set containing two rules for famous iris problem is:

If $F_3 < 1.9$ **and** $F_4 \geq 5.1$ **Then** iris-setosa
Else
    If $F_3 < 4.7$ **Then** iris-versicolor
    Else iris-virginica

Sequential covering algorithms, learn rules to separate a positive class from a negative class. In the example above, Ripper first learn rules to separate class *iris-setosa* from both classes *iris-versicolor* and *iris-virginica*, then learn rules to separate class *iris-versicolor* from class *iris-virginica*.

At each iteration of the covering algorithms, the examples (even false positives) covered by the rule is removed from the training set. Removing examples during the training, causes order dependencies between rules [2]. The last learned rule is dependent on the previous rules and their covered examples.

While testing an example, the sequential covering algorithm considers each rule in an order. The first rule that covers the example, defines the predicted class of

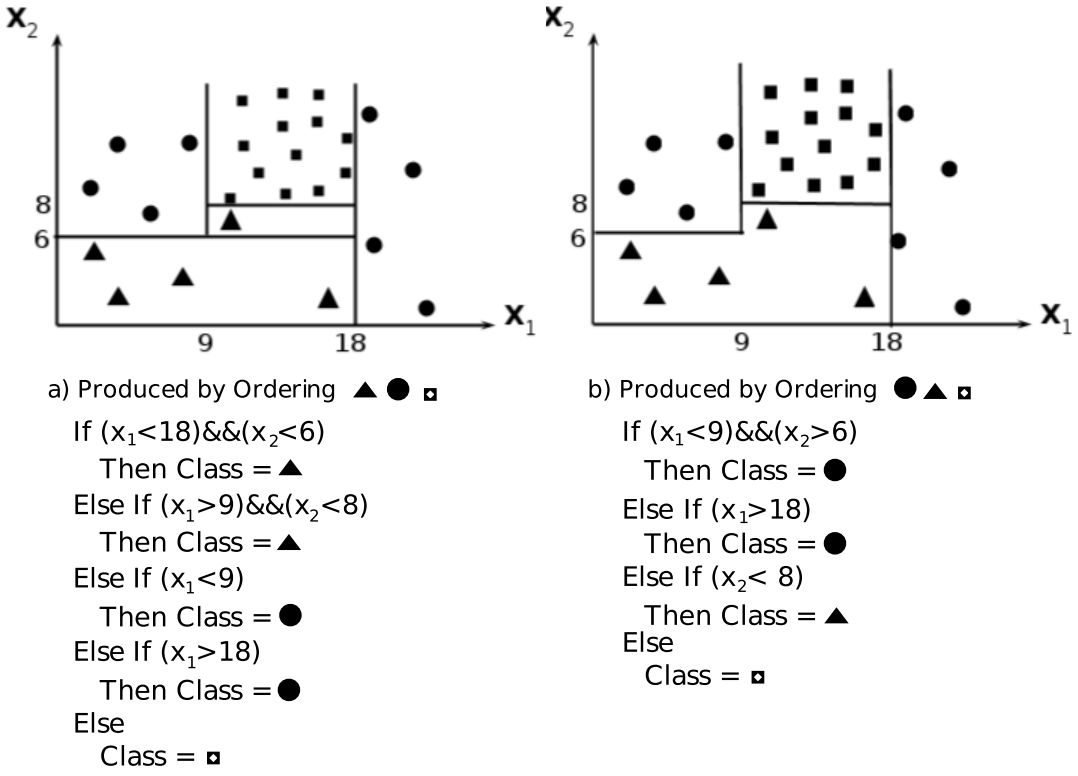

Figure 4.1: For two different class orderings, separation of data and learned rule-sets.

the example. There is an alternative unordered testing technique which checks all the rules and take a vote over the rules that cover the example. This technique is not considered in our study.

In the training part, the ordering of classes is selected heuristically and may not be optimal in terms of error and/or complexity. Common approach that is used in *covering algorithms* is, training the classes in the order of their increasing prior probabilities. In Figure 4.1 we see an example case, where two different orderings produce two different rule sets with the same error but different complexity, one composed of four rules with six terms, other composed of three rules with four terms. Although we prefer the second ordering, the heuristic may lead us to the first ordering.

After we realize that the heuristic ordering is not the optimal one, we search for an optimal ordering of classes in the ordering space. Since during training every

class affects the training set of the next class to be trained, first we focus on consecutive changes in the ordering space.

4.2 Forward Ordering Search (FOS)

Our first proposed algorithm, namely forward ordering search, views optimizing the ordering of classes in Ripper as a search in the state space of all possible orderings. In our case, the search space contains all possible permutations of classes. Although the search space is finite, it is not possible to train/validate all orderings and select the best one from $K!$ distinct orderings. Hence, there is a need to find the best ordering by visiting only a small part of the search space. Forward search algorithm starts from an initial state and we define an exchange operator that modify the ordering and allow moving from one state to another. We select the ordering that the heuristic provides as the initial state. Although not the optimal ordering, the heuristic ordering generally works well.

At each iteration of the exchange operator, the state evaluation function compares the goodness value of the next state(s) with the current state and accepts/rejects the operator depending on whether the goodness value is improved or not. The state evaluation function trains and validates the Ripper algorithm via 10×10-fold cross-validation with the ordering(s) corresponding to the next state(s) and favors the most accurate ordering. We stop the search when no candidate improves the current best. Another possibility is to stop when the error rate falls below a certain level, or when a fixed number of iterations are made. The pseudocode of FOS is given in Figure 4.2.

When there are $K > 2$ classes, the classes of the dataset are increasingly sorted according to their prior probabilities and this is initial heuristic permutation, π (Line 1). neighborPermutations() method (Figure 4.3) is the exchange operator that generates the next states. Let say we have the ordering $C_1C_2C_3 \ldots C_{K-1}C_K$. The exchange operator creates the following $K-1$ candidate orderings: $C_2C_1C_3 \ldots C_{K-1}C_K$, $C_1C_3C_2 \ldots C_{K-1}C_K$, $C_1C_2C_4 \ldots C_{K-1}C_K$, ..., $C_1C_2C_3 \ldots C_KC_{K-1}$

```
1  FOS(D, π)
2     π_best = π
3     visited[ ] = {False}
4     improved = True
5     while improved
6        improved = False
7        maxDifference = 0
8        π = neighborPermutations(π_best)
9        for i = 1 to K − 1
10          if !visited[π_i]
11             E_best = Error(Ripper(D, π_best))
12             E = Error(Ripper(D, π_i))
13             difference = E_best - E
14             visited[π_i] = True
15             if difference > maxDifference
16                maxDifference = difference
17                π_best = π_i
18                improved = True
19    return (π_best, E_best)
```

Figure 4.2: Pseudocode of FOS on dataset D with K classes and heuristic ordering π

```
1  π neighborPermutations(π)
2     for i = 1 to K − 1
3        π_i = π
4        Swap(π_i, i, i+1)
5     return π
```

Figure 4.3: Pseudocode of neighborPermutations(), for an ordering π on a dataset with K classes

(Line 8). Unvisited states are trained with Ripper and error values are kept (Lines 11-12). If there is an improvement in error, we update the best ordering value (Line 17). If the algorithm can not find a better ordering, it returns current best ordering with its error (Line 19).

Figure 4.4 shows an example run of FOS algorithm for a dataset with 3 classes. The algorithm starts with the heuristic ordering ▲ ● ■. The average error of the heuristic ordering is 0.35. The best of the two candidate orderings, ● ▲ ■, has an average error of 0.13 and is better than the current best ordering. Therefore

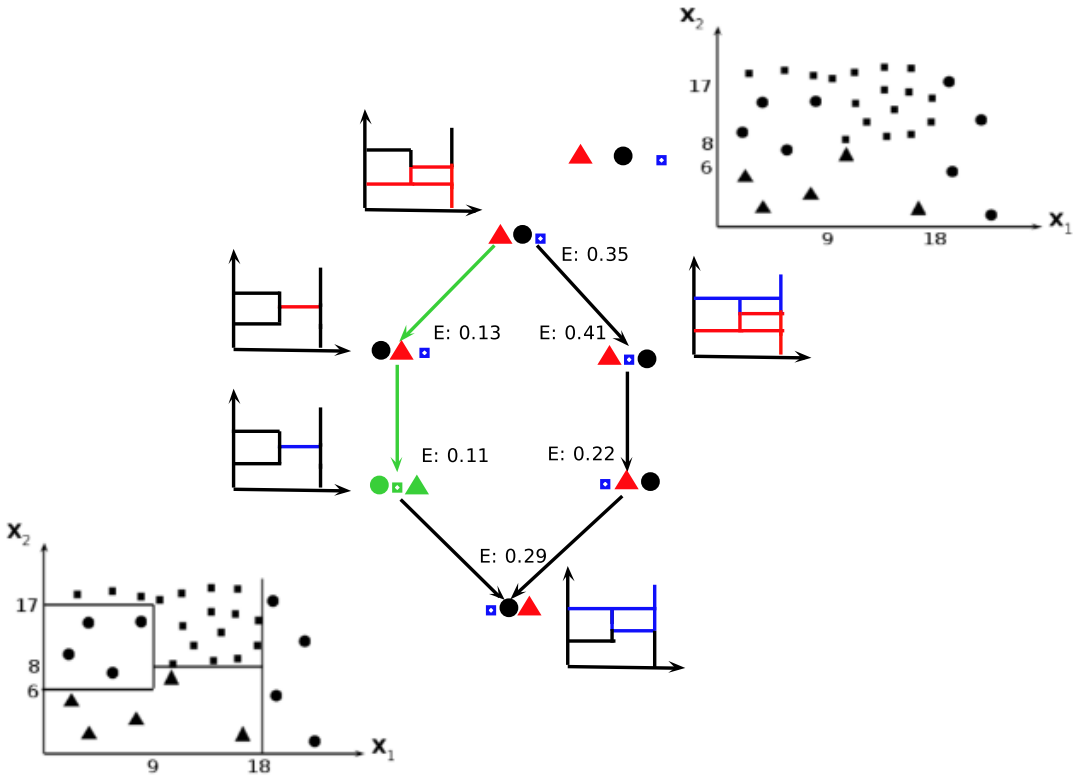

Figure 4.4: An example run of FOS algorithm for a dataset with $K = 3$.

```
1 RR - FOS (D, π_H)
2    (π_best, E_best) = FOS (D, π_H)
3    for j = 1 to 10
4       π_j = RandomOrdering()
5       (π, E ) = FOS (D, π_j)
6       if E < E_best
7          E_best = E
8          π_best = π
9    return π_best
```

Figure 4.5: Pseudocode of RR - FOS on dataset D with K classes and the heuristic ordering π_H

is accepted as the new best ordering. In the second iteration, the new candidate ● ■ ▲ with an average error of 0.11 is better than the current best ordering, and is accepted as the new best ordering. In the third iteration, the new candidate with an average error of 0.29 is not better than the current best ordering and the algorithm stops.

FOS is an example of Steepest Ascent Hill Climbing [30] and the problem with this algorithm is, it may get stuck in a local optima. In order to improve the performance of the algorithm, we use Random - Restart (Steepest Ascent) Hill Climbing. We run FOS algorithm with 10 random initial orderings in addition to the heuristic ordering. The output of Random-Restart FOS is the ordering with the best error rate among 11 distinct orderings.

The pseudocode of Random-Restart Fos is given in Figure 4.5. Since, we generate ten random orderings, the "for loop" is up to ten (Line 4). For each random ordering we run FOS algorithm (Line 5) and compare the error of the ordering with current best error (Line 6). We update best error if we find a better ordering (Line 8). After completing all random orderings, the algorithm returns the ordering with minimum error (Line 9). Not surprisingly, the results of RR-FOS are more accurate than FOS.

4.3 Pairwise Error Approximation (PEA)

4.3.1 Theory

Our second proposed algorithm, pairwise error approximation, assumes that the expected error of an ordering, that is, the expected error of the Ripper algorithm trained with that ordering, is the sum of $K(K-1)/2$ pairwise expected errors of classes.

Formally, the expected error of the Ripper algorithm with ordering π is defined as

$$\hat{E}_{\pi} = \sum_{j=1}^{K-1} \sum_{k>j}^{K} e_{\pi(j)\pi(k)} \tag{4.1}$$

where $\pi(j)$ represents j' th class in permutation π and $e_{\pi(j)\pi(k)}$ represents the error contribution of separation of class $\pi(j)$ from class $\pi(k)$. For example, the

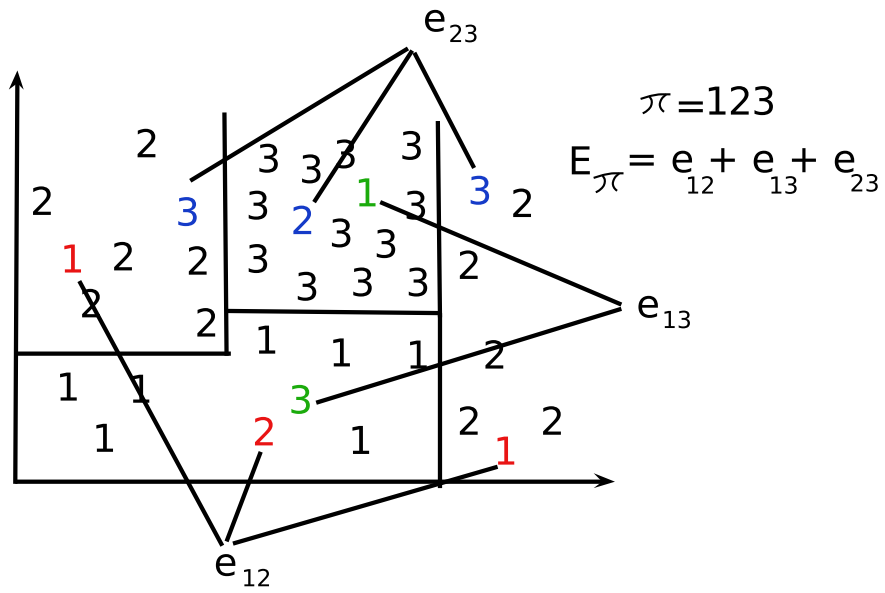

Figure 4.6: The expected error of the ordering 123 and its components (e_{ij}'s) for a dataset with $K = 3$.

expected error of the ordering $\pi = 123$ (three class problem) is defined as

$$\hat{E}_{123} = e_{12} + e_{13} + e_{23} \tag{4.2}$$

$e_{\pi(j)\pi(k)}$ contains two types of instances (See Figure 4.6):

- False positives, instances of class $\pi(k)$ covered by the rules of class $\pi(j)$.
- False negatives, instances of class $\pi(j)$ covered by the rules of class $\pi(k)$.

Since we can not estimate $e_{\pi(j)\pi(k)}$' s from a single ordering, we run Ripper algorithm N times with N random orderings π_i and get the test errors E_{π_i}. The average estimation error over N runs is defined as

$$\overline{E}_{total} = \frac{1}{N} \sum_{i=1}^{N} (E_{\pi_i} - \hat{E}_{\pi_i})^2 \tag{4.3}$$

For example, for a dataset with three classes, if we train Ripper with orderings 123, 132 and 213, the average estimation error is

$$\overline{E}_{total} = [(e_{12}+e_{13}+e_{23}-E_{123})^2+(e_{13}+e_{12}+e_{32}-E_{132})^2+(e_{21}+e_{23}+e_{13}-E_{213})^2]/3 \tag{4.4}$$

In order to minimize the average estimation error, we take the partial derivatives of $\overline{E}_{total}$ with respect to all possible pairs $e_{\pi(j)\pi(k)}$'s and solve the following system of linear equations

$$\forall_{j,k} \frac{\partial \overline{E}_{total}}{\partial e_{jk}} = 0 \tag{4.5}$$

with e_{jk}'s as unknown variables.

System of linear equations is in $\mathbf{Ax} = \mathbf{B}$ form. If we have the error rates of all $K!$ orderings, the elements of the coefficient matrix $\mathbf{A}$ will be

$\forall i, j, k, l$ s.t $0 < i \neq j \neq k \neq l \leq K$, $K > 3$,

- A$[\pi(i), \pi(j)][\pi(i), \pi(j)] = K!/2$.
- A$[\pi(i), \pi(j)][\pi(k), \pi(l)] = K!/4$
- A$[\pi(i), \pi(j)][\pi(k), \pi(j)] = K!/3$
- A$[\pi(i), \pi(j)][\pi(k), \pi(i)] = K!/6$
- A$[\pi(i), \pi(j)][\pi(j), \pi(k)] = K!/6$
- A$[\pi(i), \pi(j)][\pi(i), \pi(k)] = K!/3$

For example, for a dataset with $K = 4$ classes, there are 4! orderings and the coefficient matrix $\mathbf{A}$ is:

$$
\mathbf{A} = \begin{array}{c} \\ 12 \\ 13 \\ 14 \\ 21 \\ 23 \\ 24 \\ 31 \\ 32 \\ 34 \\ 41 \\ 42 \\ 43 \end{array}
\begin{array}{c}
\begin{array}{cccccccccccc} 12 & 13 & 14 & 21 & 23 & 24 & 31 & 32 & 34 & 41 & 42 & 43 \end{array} \\
\left(\begin{array}{cccccccccccc}
\frac{K!}{2} & \frac{K!}{3} & \frac{K!}{3} & 0 & \frac{K!}{6} & \frac{K!}{6} & \frac{K!}{6} & \frac{K!}{3} & \frac{K!}{4} & \frac{K!}{6} & \frac{K!}{3} & \frac{K!}{4} \\
\frac{K!}{3} & \frac{K!}{2} & \frac{K!}{3} & \frac{K!}{6} & \frac{K!}{3} & \frac{K!}{4} & 0 & \frac{K!}{6} & \frac{K!}{6} & \frac{K!}{6} & \frac{K!}{4} & \frac{K!}{3} \\
\frac{K!}{3} & \frac{K!}{3} & \frac{K!}{2} & \frac{K!}{6} & \frac{K!}{4} & \frac{K!}{3} & \frac{K!}{6} & \frac{K!}{4} & \frac{K!}{3} & 0 & \frac{K!}{6} & \frac{K!}{6} \\
0 & \frac{K!}{6} & \frac{K!}{6} & \frac{K!}{2} & \frac{K!}{3} & \frac{K!}{3} & \frac{K!}{3} & \frac{K!}{6} & \frac{K!}{4} & \frac{K!}{3} & \frac{K!}{6} & \frac{K!}{4} \\
\frac{K!}{6} & \frac{K!}{3} & \frac{K!}{4} & \frac{K!}{3} & \frac{K!}{2} & \frac{K!}{3} & \frac{K!}{6} & 0 & \frac{K!}{6} & \frac{K!}{4} & \frac{K!}{6} & \frac{K!}{3} \\
\frac{K!}{6} & \frac{K!}{4} & \frac{K!}{3} & \frac{K!}{3} & \frac{K!}{3} & \frac{K!}{2} & \frac{K!}{4} & \frac{K!}{6} & \frac{K!}{3} & \frac{K!}{6} & 0 & \frac{K!}{6} \\
\frac{K!}{6} & 0 & \frac{K!}{6} & \frac{K!}{3} & \frac{K!}{6} & \frac{K!}{4} & \frac{K!}{2} & \frac{K!}{3} & \frac{K!}{3} & \frac{K!}{3} & \frac{K!}{4} & \frac{K!}{6} \\
\frac{K!}{6} & \frac{K!}{6} & \frac{K!}{4} & \frac{K!}{6} & 0 & \frac{K!}{6} & \frac{K!}{3} & \frac{K!}{2} & \frac{K!}{3} & \frac{K!}{4} & \frac{K!}{6} & \frac{K!}{6} \\
\frac{K!}{4} & \frac{K!}{6} & \frac{K!}{3} & \frac{K!}{4} & \frac{K!}{6} & \frac{K!}{3} & \frac{K!}{3} & \frac{K!}{3} & \frac{K!}{2} & \frac{K!}{6} & \frac{K!}{6} & 0 \\
\frac{K!}{6} & \frac{K!}{6} & 0 & \frac{K!}{3} & \frac{K!}{4} & \frac{K!}{6} & \frac{K!}{3} & \frac{K!}{4} & \frac{K!}{6} & \frac{K!}{2} & \frac{K!}{3} & \frac{K!}{3} \\
\frac{K!}{3} & \frac{K!}{4} & \frac{K!}{6} & \frac{K!}{6} & \frac{K!}{6} & 0 & \frac{K!}{4} & \frac{K!}{6} & \frac{K!}{6} & \frac{K!}{3} & \frac{K!}{2} & \frac{K!}{3} \\
\frac{K!}{4} & \frac{K!}{3} & \frac{K!}{6} & \frac{K!}{4} & \frac{K!}{3} & \frac{K!}{6} & \frac{K!}{6} & \frac{K!}{6} & 0 & \frac{K!}{3} & \frac{K!}{3} & \frac{K!}{2}
\end{array}\right)
\end{array}
$$

For datasets with more than 6 classes, it is not feasible to run all permutations to get the coefficient matrix, so we work with a limited number of random orderings N and construct matrix $\boldsymbol{A}$ with these orderings. The solution set of the linear equations $\boldsymbol{B}$ keeps, for each pair, the sum of the test errors of the orderings that the pairs occur.

After solving the linear equation, we have the error contributions of all class pairs $e_{\pi_i(j)\pi_i(k)}$ and we can estimate the error of any ordering π_i using Equation 4.1 without actually running Ripper with that ordering π_i. We search all possible class orderings to get the optimal ordering exhaustively:

$$\pi_{optimal} = \arg\min_{\pi_i} \hat{E}_{\pi_i} \tag{4.6}$$

4.3.2 Algorithm

The pseudocode of PEA is given in Figure 4.7. The algorithm tries to estimate the expected error contribution of each pair of classes. There are $K * (K-1)$ pairs for a dataset with K classes. It initially generates N random orderings

```
1  PEA(D)
2    E_best = ∞
3    for i = 1 to N
4      π_i = RandomOrdering()
5      E_{π_i} = Error(Ripper(D, π_i))
6    A = constructCoefficientMatrix(π)
7    B = constructSolutionSet(π, E)
8    e = A^{-1} B
9    for i = 1 to K!
10     E = 0
11     for j = 1 to N
12       for k = j + 1 to N
13         E += e_{π_i(j)π_i(k)}
14     if E < E_best
15       E_best = E
16       π_best = π_i
17   return π_best
```

Figure 4.7: Pseudocode of PEA for dataset D with K classes.

```
1  constructCoefficientMatrix(π)
2    for i = 1 to N
3      for j = 1 to K − 1
4        for k = j + 1 to K
5          for l = 1 to K − 1
6            for m = l + 1 to K
7              A[π_i(j)π_i(k), π_i(l)π_i(m)] + +
8    return A
```

Figure 4.8: Pseudocode of constructCoefficientMatrix for a dataset with K classes where $\boldsymbol{\pi}$ holds N orderings.

(Line 4). Then Ripper is trained with these N orderings via 10×10-fold cross-validation (Line 5). Then, we count the number of occurrences of the pairs within N orderings. This gives us the coefficient matrix (Line 6). For every pair we sum the test errors of the orderings they occur and this result constitutes the solution set of the linear equations (Line 7). We solve the linear equations and find the estimated errors of all pairs (Line 8). The optimal ordering with minimum total estimation error is returned by the algorithm (Lines 9-16).

The method constructCoefficientMatrix (See Figure 4.8) has an outer loop with

```
1 constructSolutionSet(π, E)
2   for n = 1 to N
3     for i = 1 to K − 1
4       for j = i + 1 to K
5         B_{π_n(i)π_n(j)} += E_n
6   return B
```

Figure 4.9: Pseudocode of constructSolutionSet for a dataset with K classes. $\boldsymbol{\pi}$ holds N orderings and $\boldsymbol{E}$ holds validation errors of those orderings.

N iterations to consider N orderings in $\boldsymbol{\pi}$ (Line 2). It counts how many times the class at index j comes before the class at index k, at the same time the class at index l comes before the class at index m in N orderings (Lines 3-7).

The pseudocode in Figure 4.9 shows the construction of the solution set, $\mathbf{B}$. Method searches N orderings for every pair $\pi(i), \pi(j)$. The error of a pair is the sum of the validation errors of the orderings in which this pair occurs (Line 5).

Chapter 5

Experiments

5.1 Setup

We work with 22 different datasets from UCI repository [31]: 10 of them have three, 3 of them have four, 1 of them has five, 1 of them has six, 3 of them have seven and 10 of them have ten classes. The datasets with $K \leq 6$ classes are used only for testing the accuracy of the heuristic ordering (See Table 5.2). The datasets we use in our study are the datasets with $K > 6$ (See Table 5.1).

The error rates of CN2 and Ripper are very close to each other. Since heuristic ordering works better in Ripper (See Table 5.2), we decide to continue on Ripper. We compare the performance of our proposed algorithms FOS, RR-FOS and PEA with original Ripper in terms of generalization error, number of rules and number of conditions in the rule sets they generate. We use 10×10-fold cross-validation to generate training and test sets and paired t test [32] to compare algorithms with a confidence level of $\alpha = 0.05$. For PEA algorithm, we applied Friedman and post-hoc Nemenyi test to analyze the effect of N.

5.2 Motivation

We want to analyze the accuracy of the heuristic ordering - the ordering according to increasing prior probabilities of classes - that the rule induction algorithms use.

Table 5.1: Description of the datasets. d: Number of attributes, K: Number of classes, n: Sample size

Dataset	d	K	n
balance	4	3	625
car	6	4	1728
cmc	9	3	1473
dermatology	34	6	366
hayesroth	4	3	160
iris	4	3	150
led7	7	10	3200
leukemia1	5327	3	72
leukemia2	11225	3	72
mfeatfac	216	10	2000
mfeatfou	76	10	2000
mfeatmor	6	10	2000
mfeatpix	240	10	2000
mfeatzer	47	10	2000
nursery	8	5	12960
ocr	256	10	600
optdigits	64	10	3823
pendigits	16	10	7494
segment	19	7	2310
shuttle	9	7	58000
splice	60	3	3175
srbct	2308	4	83
tae	5	3	151
vehicle	18	4	846
wave	20	3	5000
wine	13	3	178
winequality	11	7	6497
yeast	8	10	1484

For that purpose, we run Ripper and CN2 with all possible orderings of classes for datasets with $K \leq 6$. We use 100×10-fold cross-validation to generate training and test sets and compare heuristic ordering with other orderings. Table 5.2 shows the ranking of the heuristic ordering for those datasets. As seen in Table 5.2, the heuristic ordering is not optimal in general. Although the heuristic ordering performs the best among 3 datasets, when the number of classes increases the performance of the heuristic ordering significantly diminishes. For example,

Table 5.2: The rank of the heuristic ordering in $K!$ orderings on CN2 and Ripper for datasets $K \leq 6$.

Dataset	Ripper	CN2
balance	4/6	3/6
cmc	3/6	3/6
hayesroth	1/6	1/6
iris	3/6	1/6
leukemia1	1/6	1/6
leukemia2	1/6	5/6
splice	2/6	4/6
tae	5/6	6/6
wave	5/6	6/6
wine	2/6	2/6
car	17/24	19/24
srbct	4/24	1/24
vehicle	24/24	24/24
nursery	20/120	109/120
dermatology	330/720	505/720

on *Vehicle* dataset the heuristic ordering gets the worst performance among all possible orderings. Similarly, on *Dermatology* dataset nearly the half of the orderings are better than the heuristic ordering. These results support our claim that the heuristic ordering is not the best and there is much room for improvement.

5.3 FOS and RR - FOS Results

Table 5.3 shows average and standard deviation of the error rates of rule sets produced by Ripper, FOS, and RR - FOS. Entries in bold face show the superiority of an algorithm on both of the algorithms and the entries in italic face show the superiority of an algorithm on one of algorithms. We see from the results that, although heuristic ordering works well in general, it is not the optimal and FOS helps us to find significantly better orderings than the heuristic ordering in terms of error rate. FOS is significantly better than Ripper on twelve datasets out of thirteen datasets. RR - FOS is significantly better than Ripper on all datasets and also significantly better then FOS on seven datasets (*led7*, *mfeatfou*, *mfeatmor*,

Table 5.3: Average and standard deviation of error rates of the rule sets produced by Ripper, FOS, and RR - FOS

Dataset	Ripper	FOS	RR - FOS
led7	31.01 ± 2.63	*29.58 ± 2.75*	**28.74 ± 2.68**
mfeatfac	11.78 ± 2.41	*11.20 ± 2.07*	*11.20 ± 2.07*
mfeatfou	30.88 ± 3.04	*29.49 ± 3.29*	**27.09 ± 3.17**
mfeatmor	31.82 ± 2.95	*30.48 ± 2.53*	**27.93 ± 2.55**
mfeatpix	12.88 ± 2.60	*11.59 ± 2.46*	**11.14 ± 2.00**
mfeatzer	32.13 ± 2.78	*31.49 ± 2.78*	*31.26 ± 2.86*
ocr	26.56 ± 5.73	*25.25 ± 5.78*	**22.91 ± 4.83**
optdigits	10.95 ± 1.74	*9.84 ± 1.61*	**8.75 ± 1.67**
pendigits	5.31 ± 0.89	5.15 ± 0.78	**4.53 ± 0.86**
segment	6.71 ± 1.85	*4.39 ± 1.48*	*4.28 ± 1.41*
shuttle	0.03 ± 0.02	*0.01 ± 0.01*	*0.01 ± 0.01*
winequality	46.51 ± 1.71	*46.22 ± 1.74*	*46.18 ± 1.53*
yeast	43.5 ± 4.12	*42.54 ± 3.75*	*42.54 ± 3.75*

mfeatpix, *ocr*, *optdigits*, *pendigits*). These indicate that the orderings produced by RR - FOS construct more accurate rule sets than the orderings produced by FOS.

Table 5.4 shows average and standard deviation of rule counts of the rule sets produced by Ripper, FOS, and RR - FOS. The second table shows pairwise comparison results on the rule counts by paired-t test. These results indicate that the produced ordering by RR - FOS generates rule sets those have better complexities than FOS and Ripper' s heuristic ordering. We also see that the heuristic ordering generates rule sets having worse complexities than the ordering of FOS.

Table 5.5 shows average and standard deviation of condition counts of the rule sets produced by Ripper, FOS, and RR - FOS. The second table shows pairwise comparison results on the condition counts by paired-t test. These results indicate that although we are trying to optimize the error rates of the rule sets, optimized rule sets have also less or equal complexity than the unoptimized rule sets. We also see that the orderings generated via RR-FOS produce rule sets with better complexities than FOS, besides their accuracy.

Table 5.4: The first table gives the average and standard deviation of rule counts of the rule sets produced by Ripper, FOS, and RR - FOS. The second table shows pairwise test results on the rule counts.

Dataset	Ripper	FOS	RR - FOS
led7	15.3 ± 1.5	15.8 ± 1.2	14.4 ± 1.6
mfeatfac	21.7 ± 1.8	21.8 ± 1.5	21.8 ± 1.5
mfeatfou	18.9 ± 1.9	19.1 ± 2.3	18.7 ± 2.0
mfeatmor	17 ± 1.7	17.3 ± 1.8	15.3 ± 1.9
mfeatpix	21.3 ± 1.9	20.6 ± 2.2	22.4 ± 2.5
mfeatzer	21.7 ± 1.6	21.1 ± 1.9	21.7 ± 1.9
ocr	13.5 ± 1.9	14.1 ± 1.3	14.4 ± 1.43
optdigits	39.9 ± 2.1	38.3 ± 2.1	36.2 ± 3.1
pendigits	53 ± 2.3	54 ± 1.8	46.1 ± 2.1
segment	16.9 ± 1.2	14.9 ± 1.3	13.2 ± 0.9
shuttle	11.3 ± 0.6	8.9 ± 0.5	8.1 ± 0.3
winequality	8.3 ± 1.4	8.5 ± 1.7	8.9 ± 1.5
yeast	11.4 ± 1.5	10.7 ± 0.9	10.7 ± 0.9

	Ripper	FOS	RR - FOS
Ripper	-	0	0
FOS	2	-	0
RR - FOS	5	6	-

Table 5.6 shows the number of visited orderings of FOS and RR-FOS. We see that the complexity of RR-FOS is 10 times more than FOS on 4 datasets (*led7*, *mfeatfac*, *mfeatfou*, *winequality*). This indicates that although RR - FOS is more accurate than FOS, the number of orderings (states) visited by RR - FOS is much more than the orderings visited by FOS. The reason of this result is that RR - FOS runs FOS for 10 times with different initial orderings. On the other hand, in the remaining datasets since FOS has a deficiency of sticking into a local optima at the very first steps of the algorithm, the number of visited states might be very small at some iterations of RR-FOS.

Table 5.5: The first table gives the average and standard deviation of condition counts of the rule sets produced by Ripper, FOS and RR - FOS. The second table shows pairwise test results on the condition counts.

Dataset	Ripper	FOS	RR - FOS
led7	22.7 ± 6.1	14.6 ± 6.1	12.4 ± 8.5
mfeatfac	41.8 ± 3.7	32.4 ± 3.8	32.4 ± 3.8
mfeatfou	29.8 ± 5.7	31.4 ± 7.5	31.5 ± 4.4
mfeatmor	35.1 ± 4.6	36.9 ± 3.9	35.4 ± 5.6
mfeatpix	137.2 ± 6.0	127 ± 6.4	120.8 ± 7.7
mfeatzer	179.5 ± 4.9	182.3 ± 4.9	162.1 ± 7.7
ocr	25.2 ± 4.9	22.7 ± 4.4	25.3 ± 4.6
optdigits	58.4 ± 5.4	57.4 ± 6.2	58.7 ± 9.6
pendigits	36.8 ± 8.6	38 ± 4.1	29.4 ± 8.5
segment	68.1 ± 5.2	67 ± 3.34	65.9 ± 1.7
shuttle	48 ± 1.7	46.4 ± 2.3	50.8 ± 0.9
winequality	48 ± 6.2	50.7 ± 6.9	47.2 ± 5.5
yeast	57.7 ± 4.1	60.9 ± 3.7	60.9 ± 3.7

	Ripper	FOS	RR - FOS
Ripper	-	1	3
FOS	3	-	0
RR - FOS	3	4	-

5.4 PEA Results

Table 5.7 shows average and standard deviation of error rates of the rule sets produced by Ripper and PEA with $N = 10, 20, 30, \ldots, 100$. We see that Ripper using orderings produced by PEA constructs more accurate rule sets on 7 datasets *shuttle*, *segment*, *ocr*, *optdigits*, *pendigits*, *mfeatmor*, *led7* at least for 9 different N values. We also see that PEA can not produce significantly better orderings than the heuristic ordering on 3 datasets (*yeast*, *mfeatzer*, *mfeatfac*). We can say that after exceeding 30 random orderings, the performance of PEA generally increases on 6 datasets (*segment*, *winequality*, *mfeatpix*, *mfeatfac*, *mfeatfou*,*led7*). Additionally, PEA orderings generate more accurate rule sets than heuristic ordering on 8 datasets with $N = 100$. These results show that our pairwise error estimation works well in general.

Table 5.6: Number of distinct orderings visited by FOS and RR - FOS.

Dataset	FOS	RR - FOS
led7	18	368
mfeatfac	33	332
mfeatfou	25	484
mfeatmor	340	698
mfeatpix	260	624
mfeatzer	232	536
ocr	80	457
optdigits	216	670
pendigits	58	345
segment	30	242
shuttle	59	467
winequality	12	228
yeast	166	474

Figure 5.1 shows Nemenyi' s post hoc test results for error rates and it shows us the effect of using different number of random orderings on accuracy. The test finds that the results are significantly equal but $N = 80$ is the most accurate one than the other N values on all datasets according to the rankings and it also indicates heuristic ordering of Ripper is the significantly least accurate one.

The overall performance of PEA is worse on datasets *yeast, mfeatzer, mfeatfac* and *winequality*. The unbalanced distribution of the samples through the classes might be the reason on *winequality* (The distribution of sample sizes: $n_{c_1} = 30$, $n_{c_2} = 216$, $n_{c_3} = 2138$, $n_{c_4} = 2836$, $n_{c_5} = 1079$, n_{c_6} =193, $n_{c_7} = 5$) and *yeast* (The distribution of sample sizes: $n_{c_1} = 244$, $n_{c_2} = 429$, $n_{c_3} = 463$, $n_{c_4} = 44$, $n_{c_5} = 35$, $n_{c_6} = 51$, $n_{c_7} = 163$, $n_{c_8} = 30$, $n_{c_9} = 20$, $n_{c_{10}} = 5$). These four datasets have higher estimation error $\overline{E}_{total}$, as expected.

Table 5.8 shows average and standard deviation of rule counts of the rule sets produced by Ripper and PEA with $N = 10, 20, 30, \ldots, 100$. We see that, the number of rules in the constructed rule sets via PEA orderings are significantly less than the constructed rule sets via heuristic ordering on 4 datasets (*shuttle*, *segment*, *optdigits*, *pendigits*). On these datasets, PEA produces not only significantly accurate but also significantly less complex rule sets.

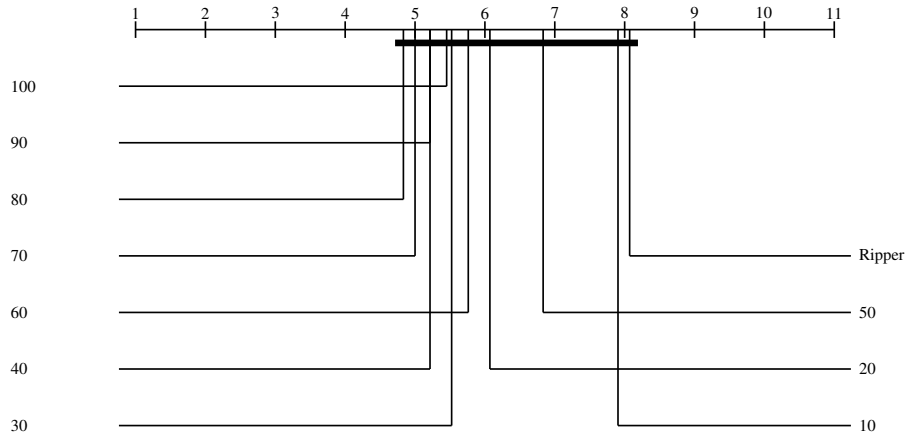

Figure 5.1: Post-hoc Nemenyi test results of PEA (error rates).

Figure 5.2 shows Nemenyi' s post hoc test results for rule counts. Although, the test shows us the results are significantly equal, rankings show that exceeding $N = 30$ produce rule sets those are less complex in general. This result also indicates that using more than 30 random orderings, not only increases the accuracy but also reduces the complexity of the rule set constructed by the produced ordering of PEA.

Table 5.9 shows average and standard deviation of condition counts of the rule sets produced by Ripper and PEA with $N = 10, 20, 30, \ldots, 100$. We see that the rule sets, constructed via PEA orderings, have less number of conditions on the same 4 datasets (*shuttle*, *segment*, *optdigits*, *pendigits*) than the rule sets generated via heuristic ordering. Again on those datasets PEA generates better models both in terms of generalization error and complexity.

Figure 5.3 shows Nemenyi' s post hoc test results for condition counts. There two groups that are significantly equal. It is difficult to make a generalization about the effect of the number of the random orderings in condition count results but ranking shows that Ripper and N values 90, 80 are generating rule sets with significantly less conditions.

Table 5.10 shows the average estimation error ($\overline{E}_{total}$) of PEA with $N = 10, 20, 30,$

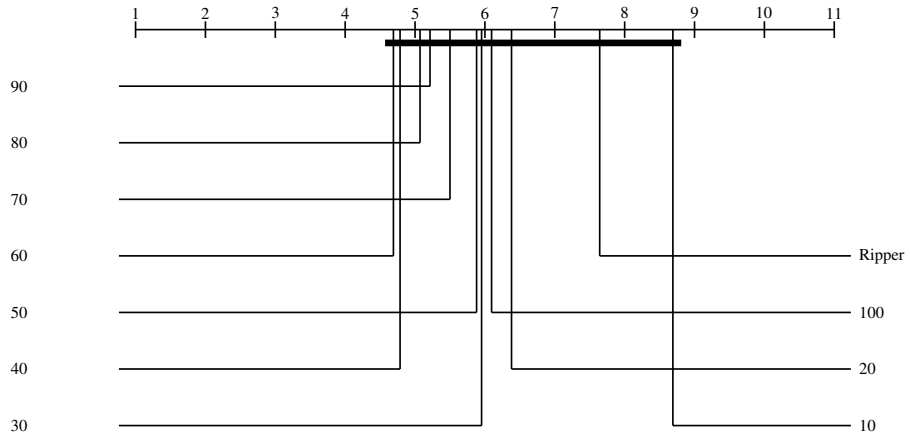

Figure 5.2: Post-hoc Nemenyi test results of PEA (rule counts).

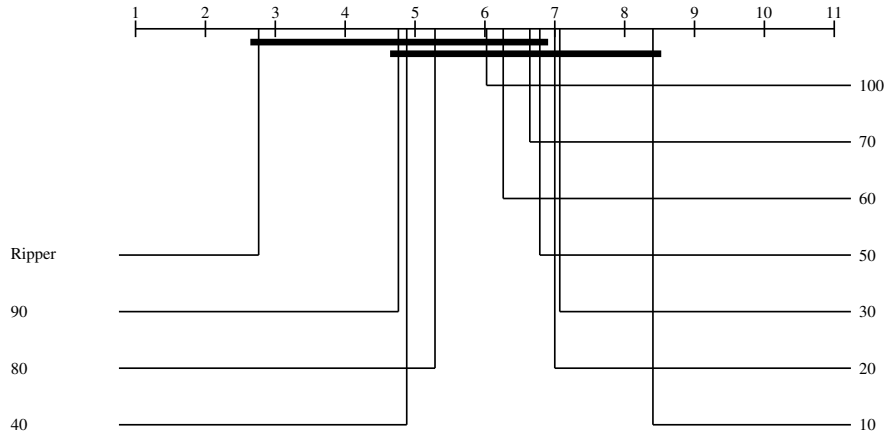

Figure 5.3: Post-hoc Nemenyi test results of PEA (condition counts).

..., 100. We see that the orderings produced by PEA has an average estimation error of less than 2 on all datasets except *winequality* and *yeast*. Additionally, almost all datasets with $N < 50$, we achieve an average estimation error of less than 0.0005.

These results are on the same random training orderings in order to get meaningful results, we decide to calculate the average estimation errors ($\overline{E}_{total}$) using 10 different random test orderings.

Table 5.11 shows the average estimation errors $\overline{E}_{total}$ obtained on these 10 test

orderings. As expected, the average estimation errors usually decrease as the number of the random orderings (N) increases. Since the average estimation errors are quite small, these results promote our assumption that the error of an ordering is approximately the sum of pairwise errors of classes. On the other hand, the datasets *mfeatzer*, *winequality* and *yeast* have higher average estimation errors and the accuracy of their constructed rule sets are lower than the other datasets can be seen in Table 5.7.

One might wonder if every pair $e_{\pi(j)\pi(k)}$ is guaranteed to appear in a set of used random ordering in PEA. Table 5.12 shows, the average and standard deviation of occurrences of each pair in N random orderings. We see that, each pair appears in N random orderings approximately N / 2 times and also the standard deviations show that the pairs have very low probability, not to be included in N random orderings.

In this thesis, we do not limit the error estimations of the pairs $e_{\pi(j)\pi(k)}$ can take. Then by, we observe that some datasets have negative $e_{\pi(j)\pi(k)}$ values and wonder about if the negativity affects the error estimation of the orderings. Table 5.13 shows the percentages of the negative $e_{\pi(j)\pi(k)}$'s for each dataset. Although the dataset *shuttle* has a very small average estimation error result in PEA, nearly half of the $e_{\pi(j)\pi(k)}$'s are negative. On the other hand the datasets *yeast, mfeatzer, mfeatfac* and *winequality* which we can not observe any improvement via PEA, have high percentages of negative $e_{\pi(j)\pi(k)}$'s. This consequence shows that there is no direct connection between performance and negativity of the $e_{\pi(j)\pi(k)}$'s but we can say that preventing the negativity might increase the performance.

Table 5.7: Average and standard deviation of error rates of the rule sets produced by Ripper, PEA with $N = 10, 20, 30, \ldots, 100$

D.set	Shuttle	Segment	Winequality	Ocr	Optdigits
Rip.	0.04 ± 0.02	6.71 ± 1.85	46.52 ± 1.71	26.57 ± 5.73	10.95 ± 1.74
10	**0.02 ±0.01**	**5.38 ± 1.53**	63.40 ± 2.74	**24.70 ± 6.33**	**10.45 ± 1.51**
20	0.04 ±0.02	6.50 ± 1.60	65.46 ± 2.56	26.15 ± 6.09	**8.49 ± 1.37**
30	**0.02 ±0.01**	**4.65 ± 1.51**	46.74 ± 1.54	**22.95 ± 5.68**	**8.73 ± 1.52**
40	**0.02 ± 0.01**	**4.48 ± 1.38**	46.53 ± 1.52	**21.82 ± 6.07**	**9.21 ± 1.61**
50	**0.02 ±0.01**	**5.07 ± 1.42**	53.41 ± 1.93	**24.90 ± 5.02**	**8.86 ± 1.33**
60	**0.02 ±0.01**	**5.25 ± 1.61**	46.47 ± 1.60	**25.32 ± 5.15**	**8.34 ± 1.55**
70	**0.02 ±0.01**	**4.68 ± 1.60**	46.27 ± 1.90	**23.58 ± 5.46**	**8.76 ± 1.68**
80	**0.02 ±0.01**	**4.48 ± 1.27**	**46.23 ± 1.74**	**23.93 ± 4.62**	**9.25 ± 1.57**
90	**0.02 ± 0.01**	**4.49 ± 1.42**	46.51 ± 1.57	**24.62 ± 5.73**	**9.03 ± 1.46**
100	**0.03 ± 0.01**	**4.48 ± 1.27**	**46.18 ± 1.71**	**22.57 ± 5.00**	**8.82 ± 1.42**

D.set	Pendigits	Yeast	Mfeatzer	Mfeatmor	
Rip.	5.32 ± 0.89	43.50 ± 4.12	32.14 ± 2.78	31.82 ± 2.95	
10	**4.69 ± 0.72**	58.66 ± 4.90	36.94 ± 2.98	**29.50 ± 2.71**	
20	**4.62 ± 0.69**	56.07 ± 4.94	31.96 ± 2.58	**30.32 ± 3.11**	
30	**4.64 ± 0.79**	51.47 ± 5.04	36.29 ± 3.07	**29.45 ± 3.06**	
40	**4.92 ± 0.79**	57.30 ± 4.91	41.20 ± 3.00	**29.44 ± 3.33**	
50	**4.79 ± 0.74**	55.86 ± 4.61	32.26 ± 3.07	**30.54 ± 3.12**	
60	**4.72 ± 0.78**	50.52 ± 4.60	36.92 ± 2.98	**31.14 ± 3.08**	
70	**4.77 ± 0.80**	42.95 ± 3.81	37.54 ± 2.96	**29.84 ± 3.34**	
80	**4.76 ± 0.88**	43.70 ± 3.99	36.58 ± 2.92	**28.47 ± 2.87**	
90	**4.68 ± 0.73**	57.54 ± 4.37	38.69 ± 3.32	**30.07 ± 3.11**	
100	**4.96 ± 0.84**	55.17 ± 4.66	38.38 ± 3.09	32.39 ± 2.71	

D.set	Mfeatpix	Mfeatfac	Mfeatfou	Led7	
Rip.	12.89 ± 2.60	11.79 ± 2.41	30.89 ± 3.04	31.01 ± 2.63	
10	14.53 ± 2.33	12.83 ± 2.15	**29.81 ± 3.39**	**30.40 ± 2.98**	
20	**11.94 ± 2.32**	11.68 ± 2.15	30.97 ± 3.13	**29.78 ± 2.68**	
30	12.83 ± 2.19	12.32 ± 2.27	30.90 ± 3.50	**30.14 ± 2.80**	
40	**11.97 ± 2.15**	11.89 ± 2.52	**28.89 ± 3.22**	**29.92 ± 2.98**	
50	12.76 ± 2.45	13.63 ± 2.62	**29.55 ± 3.86**	**29.86 ± 2.67**	
60	**12.05 ± 2.30**	12.30 ± 2.47	31.92 ± 3.57	**29.48 ± 2.87**	
70	**12.10 ± 2.54**	12.43 ± 2.25	**29.29 ± 2.99**	**30.16 ± 2.51**	
80	12.69 ± 2.48	12.35 ± 2.47	**29.36 ± 2.95**	**29.66 ± 2.92**	
90	**11.66 ± 2.21**	11.82 ± 2.30	**29.75 ± 3.27**	**29.56 ± 2.78**	
100	**11.97 ± 2.25**	11.53 ± 2.54	**28.48 ± 3.51**	31.12 ± 2.71	

Table 5.8: Average and standard deviation of rule counts of the rule sets produced by Ripper, PEA with $N = 10, 20, 30, \ldots, 100$

D.set	shuttle	segment	winequality	ocr	optdigits
Rip.	11.3 ± 0.7	16.9 ± 1.3	8.3 ± 1.4	13.5 ± 1.9	39.9 ± 2.1
10	**8.6 ± 0.7**	**15.7 ± 1.2**	9.9 ± 1.6	15.1 ± 1.4	**36.7 ± 1.6**
20	**9.5 ± 1.1**	**15.0 ± 1.4**	9.9 ± 1.7	14.8 ± 2.2	**35.4 ± 2.2**
30	**8.5 ± 0.5**	**13.6 ± 1.3**	7.9 ± 1.7	15.2 ± 1.5	38.8 ± 2.9
40	**8.6 ± 0.5**	**13.2 ± 1.5**	9.3 ± 2.1	15.4 ± 1.3	**36.6 ± 3.0**
50	**8.8 ± 0.6**	**13.4 ± 1.5**	8.6 ± 1.9	13.2 ± 2.0	**35.5 ± 3.3**
60	**9.0 ± 0.8**	**12.0 ± 1.1**	7.9 ± 2.1	13.3 ± 1.7	**37.1 ± 3.1**
70	**9.0 ± 0.7**	**13.2 ± 1.9**	9.7 ± 1.7	13.9 ± 1.3	**35.8 ± 3.5**
80	**8.8 ± 0.8**	**14.6 ± 1.0**	8.5 ± 1.7	14.1 ± 1.5	**36.4 ± 2.7**
90	**8.6 ± 0.5**	**13.3 ± 1.3**	7.5 ± 2.0	14.7 ± 1.3	**36.4 ± 3.3**
100	**9.7 ± 0.9**	**14.6 ± 1.0**	8.6 ± 2.0	14.6 ± 1.7	**36.2 ± 2.3**

D.set	pendigits	yeast	mfeatzer	mfeatmor	
Rip.	53.0 ± 2.4	11.4 ± 1.5	21.7 ± 1.6	17.0 ± 1.7	
10	**48.8 ± 2.5**	11.6 ± 1.6	**19.1 ± 1.3**	16.9 ± 2.4	
20	**48.1 ± 3.0**	10.5 ± 1.4	21.4 ± 2.1	17.5 ± 1.6	
30	**48.7 ± 2.8**	10.5 ± 1.1	20.2 ± 2.4	16.6 ± 2.3	
40	**48.8 ± 2.3**	10.5 ± 1.1	**18.8 ± 1.5**	**15.3 ± 2.4**	
50	**48.0 ± 1.7**	11.5 ± 2.3	22.2 ± 1.5	16.2 ± 2.0	
60	**46.0 ± 2.2**	10.6 ± 1.7	**19.6 ± 2.4**	16.7 ± 1.4	
70	**50.5 ± 2.2**	10.6 ± 0.8	**18.7 ± 3.0**	16.7 ± 1.9	
80	**47.8 ± 2.2**	10.5 ± 1.1	20.5 ± 1.4	15.8 ± 1.8	
90	**49.5 ± 2.2**	10.8 ± 1.9	**18.0 ± 2.8**	16.5 ± 0.9	
100	**50.7 ± 2.3**	11.5 ± 1.7	**18.3 ± 2.3**	16.5 ± 1.7	

D.set	mfeatpix	mfeatfac	mfeatfou	led7	
Rip.	21.3 ± 1.9	21.7 ± 1.9	18.9 ± 2.0	15.3 ± 1.6	
10	23.3 ± 2.3	22.2 ± 2.1	20.0 ± 2.7	15.9 ± 1.2	
20	22.6 ± 1.6	20.1 ± 1.9	18.8 ± 2.5	15.1 ± 1.3	
30	22.8 ± 1.4	21.7 ± 1.7	18.1 ± 1.7	16.0 ± 1.3	
40	22.3 ± 1.6	20.7 ± 1.5	19.2 ± 1.9	**13.8 ± 1.3**	
50	21.4 ± 1.6	23.7 ± 1.3	18.5 ± 1.4	16.6 ± 0.7	
60	22.7 ± 2.3	21.6 ± 1.6	18.7 ± 1.8	**14.1 ± 1.5**	
70	21.5 ± 1.4	22.0 ± 2.4	17.7 ± 1.7	15.1 ± 1.4	
80	22.6 ± 1.9	21.8 ± 1.3	**17.4 ± 1.8**	15.6 ± 1.5	
90	20.2 ± 0.6	21.9 ± 2.1	19.5 ± 1.3	15.4 ± 1.4	
100	**19.4 ± 2.3**	21.8 ± 1.7	19.3 ± 2.2	14.6 ± 1.5	

Table 5.9: Average and standard deviation of condition counts of the rule sets produced by Ripper, PEA with $N = 10, 20, 30, \ldots, 100$

D.set	shuttle	segment	winequality	ocr	optdigits
Rip.	22.7 ± 1.7	41.8 ± 5.2	29.8 ± 6.2	35.1 ± 4.9	137.2 ± 5.4
10	**14.6 ± 1.9**	**34.5 ± 3.3**	33.0 ± 6.4	38.3 ± 3.8	**131.6 ± 8.6**
20	**19.3 ± 3.4**	41.4 ± 4.0	34.5 ± 7.7	38.4 ± 5.0	**113.9 ± 9.0**
30	**14.9 ± 1.8**	**29.9 ± 3.3**	26.0 ± 7.4	35.4 ± 3.8	**123.3 ± 9.8**
40	**15.0 ± 2.0**	**26.7 ± 4.2**	31.9 ± 8.6	34.5 ± 2.5	**118.4 ± 9.4**
50	**12.2 ± 0.9**	**28.7 ± 4.0**	28.5 ± 7.3	33.2 ± 5.2	**114.1 ± 10.5**
60	**16.1 ± 3.1**	**27.7 ± 2.7**	28.5 ± 9.5	33.0 ± 3.8	**114.1 ± 9.1**
70	**16.1 ± 2.3**	**26.9 ± 4.3**	34.6 ± 7.7	35.8 ± 3.6	**114.2 ± 12.7**
80	**15.1 ± 2.9**	**28.2 ± 3.2**	31.4 ± 7.0	33.2 ± 4.8	**119.0 ± 8.0**
90	**13.6 ± 1.6**	**25.5 ± 3.3**	26.5 ± 8.5	35.9 ± 3.2	**114.2 ± 9.1**
100	**18.2 ± 3.1**	**28.2 ± 3.2**	31.0 ± 8.9	36.8 ± 2.5	**115.1 ± 7.3**

D.set	pendigits	yeast	mfeatzer	mfeatmor	
Rip.	179.5 ± 8.7	25.2 ± 4.1	58.4 ± 4.9	36.8 ± 4.7	
10	**169.4 ± 9.8**	25.8 ± 5.9	**51.4 ± 5.1**	35.0 ± 6.0	
20	**166.2 ± 11.4**	25.7 ± 5.6	58.3 ± 5.1	36.1 ± 3.1	
30	**172.9 ± 12.6**	27.3 ± 3.4	**50.7 ± 6.3**	33.4 ± 6.6	
40	**163.7 ± 8.7**	25.0 ± 3.5	**51.9 ± 6.5**	**28.5 ± 4.8**	
50	**170.0 ± 9.7**	29.2 ± 7.9	59.1 ± 4.7	**31.2 ± 5.2**	
60	**166.9 ± 5.4**	27.5 ± 5.2	**54.1 ± 6.6**	36.4 ± 4.6	
70	**176.8 ± 7.9**	25.0 ± 3.0	53.3 ± 9.6	34.3 ± 5.1	
80	**163.5 ± 8.1**	24.4 ± 3.8	55.0 ± 4.9	**29.2 ± 3.2**	
90	**163.1 ± 10.3**	23.7 ± 4.1	**50.4 ± 8.0**	**32.9 ± 2.8**	
100	**171.3 ± 4.7**	28.0 ± 6.5	**47.5 ± 6.7**	**32.8 ± 4.1**	

D.set	mfeatpix	mfeatfac	mfeatfou	led7	
Rip.	68.1 ± 6.1	48.0 ± 3.8	48.0 ± 5.8	57.7 ± 6.2	
10	79.0 ± 10.0	52.3 ± 4.8	53.1 ± 7.6	60.0 ± 6.7	
20	70.8 ± 5.2	**42.4 ± 4.4**	47.4 ± 8.3	55.3 ± 6.9	
30	73.0 ± 4.7	49.3 ± 4.4	49.1 ± 4.9	64.1 ± 5.9	
40	70.2 ± 4.6	47.5 ± 3.5	50.8 ± 7.0	**49.5 ± 6.9**	
50	73.4 ± 3.5	58.0 ± 3.8	46.7 ± 2.5	69.1 ± 3.1	
60	73.3 ± 8.8	47.9 ± 4.4	50.3 ± 4.7	**51.1 ± 6.5**	
70	69.3 ± 3.7	49.4 ± 4.6	**44.3 ± 5.5**	58.7 ± 6.5	
80	71.6 ± 4.6	47.6 ± 3.4	**43.7 ± 5.4**	59.8 ± 4.7	
90	**62.5 ± 3.8**	48.2 ± 2.9	53.2 ± 4.7	60.0 ± 6.8	
100	**59.6 ± 6.7**	46.0 ± 4.7	47.6 ± 6.5	57.0 ± 7.7	

Table 5.10: The average estimation error $\overline{E}_{total}$ of PEA with $N = 10, 20, 30, \ldots, 100$.

N	led7	mfeatfac	mfeatfou	mfeatmor	mfeatpix	mfeatzer	ocr
10	0	0	0	0	0	0	0
20	0	0	0	0	0	0	0
30	0	0	0	0	0	0	0
40	0	0	0	0	0	0	0
50	0.021	0.003	0.146	0.194	0.003	0.316	0.046
60	0.049	0.053	0.181	0.954	0.046	0.531	0.332
70	0.115	0.044	0.484	0.433	0.060	1.264	0.253
80	0.206	0.041	0.563	0.801	0.093	1.734	0.633
90	0.137	0.061	0.683	1.718	0.092	1.528	0.768
100	0.190	0.061	0.602	1.471	0.100	1.895	0.492

N	optdigits	pendigits	segment	shuttle	winequality	yeast	
10	0	0	0	0	0	0	
20	0	0	0	0	0	0	
30	0	0	0.017	0.107	10.146	0	
40	0	0	0.060	0.102	11.969	0	
50	0.007	0.001	0.025	0	20.410	0.149	
60	0.012	0.003	0.037	0.007	17.686	2.684	
70	0.023	0.007	0.059	0	10.661	3.865	
80	0.046	0.008	0.047	0.055	22.761	5.292	
90	0.032	0.010	0.066	0.048	21.793	6.197	
100	0.037	0.008	0.069	0.240	21.410	7.151	

Table 5.11: The average estimation error $\overline{E}_{total}$ calculated over 10 test orderings.

N	led7	mfeatfac	mfeatfou	mfeatmor	mfeatpix
10	2.65 ± 1.79	1.87 ± 1.50	3.72 ± 3.96	10.56 ± 13.75	3.13 ± 2.77
20	0.35 ± 0.60	0.37 ± 0.72	2.91 ± 2.21	6.54 ± 8.92	0.21 ± 0.22
30	1.34 ± 2.08	0.11 ± 0.12	7.79 ± 10.60	1.96 ± 3.24	0.14 ± 0.13
40	1.04 ± 1.50	0.18 ± 0.25	3.88 ± 6.14	3.40 ± 3.39	0.30 ± 0.46
50	0.93 ± 1.27	0.76 ± 0.78	1.97 ± 2.44	19.13 ± 24.80	0.05 ± 0.05
60	0.19 ± 0.25	0.06 ± 0.07	0.87 ± 0.76	4.70 ± 6.00	0.14 ± 0.13
70	0.11 ± 0.10	0.09 ± 0.13	0.45 ± 0.56	0.78 ± 1.24	0.09 ± 0.13
80	0.13 ± 0.22	0.04 ± 0.05	0.38 ± 0.35	2.17 ± 2.63	0.09 ± 0.09
90	0.06 ± 0.07	0.04 ± 0.07	0.13 ± 0.20	0.88 ± 0.98	0.02 ± 0.01
100	0.04 ± 0.04	0.03 ± 0.03	0.19 ± 0.15	0.23 ± 0.14	0.02 ± 0.02

N	mfeatzer	ocr	optdigits	pendigits	
10	18.33 ± 26.31	11.58 ± 10.47	1.08 ± 1.80	0.133 ± 0.14	
20	7.22 ± 13.33	6.21 ± 4.81	0.07 ± 0.13	0.087 ± 0.10	
30	2.50 ± 2.09	2.90 ± 3.43	0.08 ± 0.08	0.040 ± 0.06	
40	6.54 ± 9.66	0.75 ± 1.30	0.12 ± 0.25	0.032 ± 0.02	
50	13.35 ± 20.38	1.27 ± 1.45	0.11 ± 0.14	0.050 ± 0.05	
60	3.14 ± 2.98	0.32 ± 0.25	0.07 ± 0.07	0.015 ± 0.02	
70	1.97 ± 2.91	0.33 ± 0.29	0.03 ± 0.02	0.006 ± 0.01	
80	2.54 ± 3.12	0.17 ± 0.15	0.02 ± 0.02	0.007 ± 0.01	
90	1.30 ± 1.34	0.42 ± 0.81	0.02 ± 0.02	0.003 ± 0.00	
100	0.94 ± 1.48	0.12 ± 0.16	0.01 ± 0.02	0.005 ± 0.01	

N	segment	shuttle	winequality	yeast	
10	0.497 ± 0.72	0 ± 0	124.41 ± 132.62	60.02 ± 61.05	
20	0.863 ± 1.36	0.005 ± 0.004	375.04 ± 585.84	8.06 ± 16.17	
30	0.094 ± 0.09	0.366 ± 0.667	69.18 ± 87.25	12.05 ± 14.97	
40	0.026 ± 0.04	0.045 ± 0.072	16.06 ± 22.36	42.99 ± 39.42	
50	0.024 ± 0.03	0 ± 0	15.01 ± 17.12	72.34 ± 104.12	
60	0.035 ± 0.04	0.001 ± 0.002	6.37 ± 3.66	10.93 ± 13.25	
70	0.015 ± 0.02	0 ± 0	6.08 ± 8.14	9.66 ± 21.24	
80	0.007 ± 0.01	0.002 ± 0.002	4.32 ± 3.57	4.50 ± 6.64	
90	0.009 ± 0.01	0.002 ± 0.002	3.65 ± 4.08	3.83 ± 3.23	
100	0.004 ± 0.01	0.012 ± 0.008	2.67 ± 2.53	3.95 ± 5.47	

Table 5.12: Average and standard deviation of occurrences of each pair in N random orderings.

N	led7	mfeatfac	mfeatfou	mfeatmor	mfeatpix	mfeatzer	ocr
10	5 ± 1.4	5 ± 1.3	5 ± 1.3	5 ± 1.5	5 ± 1.6	5 ± 1.7	5 ± 1.5
20	10 ± 1.9	10 ± 2.1	10 ± 2.8	10 ± 1.8	10 ± 2	10 ± 2.6	10 ± 2.8
30	15 ± 3.8	15 ± 3.1	15 ± 2.4	15 ± 2.8	15 ± 2.3	15 ± 2.7	15 ± 2.1
40	20 ± 2.9	20 ± 3.8	20 ± 2.9	20 ± 3.5	20 ± 2.9	20 ± 2.8	20 ± 3.2
50	25 ± 3.8	25 ± 4.2	25 ± 4.1	25 ± 2.4	25 ± 4.9	25 ± 3.4	25 ± 3.5
60	30 ± 4.8	30 ± 3.2	30 ± 5.2	30 ± 3.1	30 ± 3.4	30 ± 4.2	30 ± 4.3
70	35 ± 3.8	35 ± 3.5	35 ± 4.5	35 ± 3.9	35 ± 3.5	35 ± 3.4	35 ± 4.3
80	40 ± 4.3	40 ± 4.6	40 ± 3.7	40 ± 5.7	40 ± 4.5	40 ± 4	40 ± 3.7
90	45 ± 6.4	45 ± 5	45 ± 4.8	45 ± 6.5	45 ± 5.3	45 ± 4.9	45 ± 5
100	50 ± 5	50 ± 4.3	50 ± 3.8	50 ± 6.2	50 ± 5.3	50 ± 4.6	50 ± 4.4

N	optdigits	pendigits	segment	shuttle	winequality	yeast	
10	5 ± 1.7	5 ± 1.1	5 ± 1.8	5 ± 1.3	5 ± 1.5	5 ± 1.5	
20	10 ± 2.1	10 ± 2.3	10 ± 2.2	10 ± 2.4	10 ± 2	10 ± 2.8	
30	15 ± 3.1	15 ± 2.1	15 ± 2.2	15 ± 2.5	15 ± 3.2	15 ± 2.6	
40	20 ± 3	20 ± 3.8	20 ± 2.4	20 ± 3.9	20 ± 3.3	20 ± 3.3	
50	25 ± 3.8	25 ± 3.3	25 ± 2.6	25 ± 3.2	25 ± 2.5	25 ± 4	
60	30 ± 3.7	30 ± 3.7	30 ± 5	30 ± 6.1	30 ± 4.7	30 ± 3.5	
70	35 ± 3.9	35 ± 3.4	35 ± 4.2	35 ± 4.9	35 ± 6	35 ± 4.7	
80	40 ± 5	40 ± 3.4	40 ± 4.5	40 ± 4.8	40 ± 4.8	40 ± 3.7	
90	45 ± 6.2	45 ± 4.2	45 ± 5.8	45 ± 4.9	45 ± 3.8	45 ± 3.9	
100	50 ± 4.1	50 ± 3.5	50 ± 3.8	50 ± 6.3	50 ± 8	50 ± 3.6	

Table 5.13: Percentage of the negative $e_{\pi(j)\pi(k)}$ values for PEA algorithm

N	led7	mfeatfac	mfeatfou	mfeatmor	mfeatpix	mfeatzer	ocr
10	0	0	0	1.1	0	1.1	0
20	0	0	0	2.2	0	0	0
30	0	0	14.4	0	7.8	8.9	3.3
40	1.1	6.7	15.6	21.1	11.1	14.4	7.8
50	3.3	23.3	21.1	33.3	10	36.7	22.2
60	2.2	3.3	5.6	30	12.2	16.7	10
70	0	1.1	2.2	2.2	10	20	4.4
80	0	0	4.4	8.9	1.1	8.9	6.7
90	0	1.1	0	8.9	0	5.6	3.3
100	0	0	0	2.2	0	6.7	4.4

N	optdigits	pendigits	segment	shuttle	winequality	yeast	
10	0	0	4.8	19	0	4.4	
20	0	0	35.7	42.9	40.5	1.1	
30	1.1	0	14.3	45.2	21.4	26.7	
40	5.6	5.6	4.8	50	4.8	42.2	
50	11.1	16.7	2.4	26.2	4.8	37.8	
60	11.1	2.2	4.8	47.6	2.4	34.4	
70	3.3	0	2.4	14.3	2.4	26.7	
80	2.2	0	2.4	50	0	20	
90	1.1	0	2.4	42.9	0	6.7	
100	1.1	0	2.4	50	2.4	15.6	

Conclusion

Current heuristic approach used in Ripper that orders the classes in a dataset according to their sample sizes, usually does not give the most accurate classification. In this thesis, we propose two algorithms to improve this heuristic.

The first algorithm, FOS, is based on Steepest Ascent Hill Climbing in the ordering space and guaranteed to find a better ordering if there exist one. The second algorithm PEA, does not guarantee to find a better ordering but its complexity is usually better than FOS and the performance of the algorithm is satisfying. We analyze the effect of the number of random orderings used for PEA and realize that there is no major difference after exceeding a certain number of random orderings. Since FOS algorithm searches for a local optimum, the proposed algorithm Random-Restart FOS improved the accuracy of FOS by extending the search space.

This study is the first step to realize the importance of the training ordering of classes, later on we can pretend each ordering as a classifier and get better classifiers by the ensemble of the orderings.

References

[1] A. Asuncion and David J. Newman. UCI machine learning repository, 2007. URL `http://www.ics.uci.edu/$\sim$mlearn/{MLR}epository.html`.

[2] D. J. Hand Michael Berthold. *Intelligent Data Analysis: An Introduction*. Springer-Verlag, 2003.

[3] J. Cendrowska. PRISM: An algorithm for inducing modular rules. *International Journal of Man-Machine Studies*, 27:349–370, 1987.

[4] D. Fensel and M. Wiese. Refinement of rule sets with JOJO. In *Proceedings of the 6th European Conference on Machine Learning*, pages 378–383, 1993.

[5] R. S. Michalski, I. Mozetič, J. Hong, and N. Lavrač. The multi-purpose incremental learning system AQ15 and its testing application to three medical domains. In *Proceedings of the 5th National Conference on Artificial Intelligence*, pages 1041–1045, Philadelphia, PA, 1986.

[6] W. W. Cohen. Fast effective rule induction. In *The Twelfth International Conference on Machine Learning*, pages 115–123, 1995.

[7] G. I. Webb and N. Brkič. Learning decision lists by prepending inferred rules. In *Proceedings of the AI'93 Workshop on Machine Learning and Hybrid Systems*, Melbourne, Australia, 1993.

[8] P. Clark and T. Niblett. The CN2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[9] J. Fürnkranz. Separate-and-conquer learning. *Artificial Intelligence Review*, 13:3–54, 1999.

[10] J. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[11] A. A. Freitas R. Parpinelli and H. S. Lopes. Data mining with an ant colony optimization algorithm. *IEEE Transions on Evolutionary Computation*, 6: 321–332, 2002.

[12] K. Sumangala and G. Nithya. Comparative study on bio-inspired approach for soil classification. *International Journal of Computer Applications*, 38: 32–37, 2012.

[13] H. S. Lopes R. Parpinelli and A. A. Freitas. An ant colony based system for data mining: Applications to medical data. In *Proc. Genetic and Evol. Comput. Conf.*, page 791, 2001.

[14] J. Vanthienen M. Snoeck D. Martens, M. D. Backer and B. Baesens. Classification with ant colony optimization. *IEEE Transions on Evolutionary Computation*, 11:651–665, 2007.

[15] J. R. Romero J. L. Olmo and S. Ventura. Using ant programming guided by grammar for building rule-based classifiers. *IEEE Transactions On Systems, Man, And CyberneticsPart B: Cybernetics*, 41:1585–1599, 2011.

[16] I. Kohonenko and M. Kovačič. Learning as optimization: Stochastic generation of multiple knowledge. In *Proceedings of the 9th International Workshop on Machine Learning*, pages 257–262, 1992.

[17] H. Theron and I. Cloete. BEXA: A covering algorithm for learning propositional concept descriptions. *Machine Learning*, 24:5–40, 1996.

[18] D. Haussler. Quantifying inductive bias: Ai learning algorithms and valiant's learning framework. *Artificial Intelligence*, 36:177–221, 1988.

[19] G.I Webb S.P. Yip. Function finding in classification learning. In *Proceedings of the 2nd Pacific Rim International Conference on Artificial Intelligence*, pages 555–559, 1992.

[20] W. W. Cohen. Efficient pruning methods for separate–and–conquer rule learning systems. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 988–994, 1993.

[21] J. Fürnkranz. Pruning algorithms for rule learning. *Machine Learning*, 27: 139–172, 1997.

[22] J. Fürnkranz and G. Widmer. Incremental reduced error pruning. In *11th International Conference on Machine Learning*, pages 378–383, New Brunswick, New Jersey, 1994. Morgan Kaufmann.

[23] M. Chisholm and P. Tadepalli. Learning decision rules by randomized iterative local search. In *Proceedings of the 19th International Conference on Machine Learning*, pages 75–82, 2002.

[24] F. Bergadano, S. Matwin, R. S. Michalski, and J. Zhang. Learning two-tiered descriptions of flexible concepts: The POSEIDON system. *Machine Learning*, 8:5–43, 1992.

[25] C. A. Brunk and M. J. Pazzani. An investigation of noise-tolerant relational concept learning algorithms. In *Proceedings of the 8th International Workshop on Machine Learning*, pages 389–393, 1991.

[26] G. Venturini. SIA: A supervised inductive algorithm with genetic search for learning attributes based concepts. In *Proceedings of the 6th European Conference on Machine Learning*, pages 280–296, Vienna, Austria, 1993.

[27] S. M. Weiss and N. Indurkhaya. Reduced complexity rule induction. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 678–684, 1991.

[28] Ethem Alpaydın. *Introduction to Machine Learning*. The MIT Press, 2010.

[29] J. R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.

[30] Ben Coppin. *Artificial Intelligence Illuminated*. Jones and Bartlett Publishers, 2004.

[31] C.L. Blake and C.J. Merz. UCI repository of machine learning databases, 2000. URL `http://www.ics.uci.edu/$\sim$mlearn/MLRepository.html`.

[32] T. G. Dietterich. Approximate statistical tests for comparing supervised classification learning classifiers. *Neural Computation*, 10:1895–1923, 1998.

Curriculum Vitae

Sezin Ata was born in 11 January 1986, in Kayseri. She received her B.S. degree in Mathematics Engineering and Computer Engineering in 2008 from Işık University. She has been working as a teaching assistant at the Department of Computer Engineering of Işık University from 2009 to 2012. Her research interests are analysis of algorithms, pattern recognition, machine learning.