On Using Logic Synthesis for Knowledge Discovery

Jeffrey A. Goldman

Wright-Laboratory
WL/AARA-3
2690 C Street STE 1
Wright-Patterson AFB
Ohio 45433-7408[†]

Mark L. Axtell

Veda, Inc.
5200 Springfield Pike
Suite #200
Dayton, Ohio 45431[*]

Abstract

Learning from data is the central theme of Knowledge Discovery in Databases (KDD) and the Machine Learning (ML) community. In order to handle large databases, certain assumptions are necessary to make the problem tractable. Without introducing explicit domain knowledge, a natural assumption is Occam's Razor. However, the requirement to find solutions of low complexity is not limited to KDD and ML. For example, in the logic synthesis community, low complexity solutions are sought for realizing circuits. Although the logic synthesis paradigms discussed are certainly not new, it is still a relatively unknown phenomenon when referring to these tools' ability as machine learning programs. The purpose of this paper is to demonstrate the applicability of circuit design tools to the KDD and ML communities. Specifically, we will exhibit results from C4.5 (a typical machine learning algorithm), Espresso (a 2-level minimization circuit design tool), and Function Extrapolation by Recomposing Decompositions (FERD).

1 Introduction

Only recently have ideas from logic synthesis and machine learning begun to converge [4] [8]. The goals are similar. In KDD and ML, at the core of every problem is how to effectively generalize concepts from data [11]. In the circuit design community, the end product is the realization of a minimum complexity circuit with respect to the number of gates, depth, inputs, literals, or product terms. The problems are analogous. In KDD, we have a database with fields and records. A set of records defines the concept. In logic synthesis, the fields are the binary variables (inputs) of the circuit and each record is a specification. The entire set of these records defines the circuit.

In this paper, we will examine the performance of two circuit design methods on typical machine learning problems and compare the results with a standard decision tree learner, C4.5. We will look at their performance on several small, noise free, binary functions without any missing values. We will also explore their performance on the heart disease database from the University of California-Irvine[1] (UCI). The comparison will serve to highlight the shortcomings of FERD but at the same time not eliminate synthesis as a viable technique to KDD problems. Some limited initial background for each of the three systems is given so that the reader can understand the mechanics of the algorithms presented. The goal is not to present algorithms but rather to demonstrate the applicability of logic synthesis to KDD and ML. All three algorithms share the common goal of a consistent, minimal complexity solution. All three differ in their method to find it.

2 The C4.5 System

A detailed study of C4.5 is given in [12]. The intention of this section is to familiarize the reader with general information about how C4.5 learns a concept. It is important to note that C4.5 is equipped to handle noisy data, conflicting data, continuous variables, and other features that is not our primary concern in this discussion. Here, we are testing performances given binary variables and 100% truthful data.

[*]Email: maxtell.dytn@veda.com
[†]Email: goldmanj@aa.wpafb.af.mil

[1]At present, the point of contact for this database is David W. Aha (aha@ics.uci.edu) and it is important to the authors that he is recognized here for his efforts to maintain it.

The C4.5 system has many different options that can be altered by the user for a given learning environment. The default options are as follows. First, given a training set, C4.5 builds a decision tree using the gain ratio criterion. In short, C4.5 chooses a test for the tree if it splits the training data into two unbalanced groups (i.e. Only Positive, Only Negative, Largest Positive). The measure is also normalized. In essence, the gain metric is a measure of Entropy. Second, C4.5 has a threshold default of 2 for a given test in a tree. The test must have at least 2 outcomes with a minimum number of cases. To be more precise, the sum of the weights of the cases for at least two subsets must attain a minimum of 2. We would increase this value if we had noisy data.

It is useful at this point to discuss the option that allows C4.5 to build n trees retaining the best. The reason C4.5 does not produce an optimal tree (optimal in the sense that it is the smallest decision tree possible, consistent with the training set) every time is because the problem is NP-complete [7]. Thus, the gain metric is a heuristic in order to build a near optimal tree in polynomial time.

Some shortcomings of C4.5 are mentioned in [12]. One problem is that C4.5 cannot correctly classify cases in which there are non-rectangular regions. For example, in dealing with continuous variables on a two-dimensional plane, the regions bounded by the x-axis, the y-axis, and the line $y = x (x, y > 0)$ do not lend themselves to being covered by rectangular regions. Instead, the triangular regions are approximated. Problems arising related to this are poorly delineated regions and fragmented regions. The author attributes fragmented regions to not having enough data to correctly classify. Similarly, C4.5 is not equipped to handle "XOR" type relations. The inherent problem is its inability to deal with replication in such disjunctive concepts as: (A and B) or (C and D) [10].

Several parameter options were tested on the entire set of functions [5]. Different options such as grouping, windowing, threshold, and number of trees were changed in order to yield C4.5's best performance. After many full experimental runs, it was determined C4.5 performed best on the training set when the options it was given was -m 1 and -t 10. The "m" parameter is a variable that has to do with the amount of noise in the training data. Since we were not testing noisy data, it was set to it's lowest value[2]. Although the more trees C4.5 builds, the better the performance, there was no significant change in going from ten trees to one-hundred trees. However, there was a difference between one tree and ten trees.

3 Function Decomposition

It is noted to the reader that this is simply an overview of Ashenhurst-Curtis decomposition. For a more rigorous explanation of the inner workings of function decomposition or function extrapolation, the reader is referred to [1], [2], and [14]. The purpose of this section is to give the reader a feel for function decomposition.

Decomposing a function involves breaking it up into smaller subfunctions. These smaller functions are further broken down until all subfunctions will no longer decompose. For a given function, the number of ways to choose two sets of variables (the partition space) is exponential. The decomposition space is even larger, since there are several ways the subfunctions can be combined and there are several levels of subfunctions possible.

FERD works as follows. A set of variables is partitioned into two sets. The decomposition matrix (similar to a Karnaugh map) formed from these two sets consist of 0's, 1's and "don't cares" (or unknowns) corresponding to the outputs of the samples. Columns are then matched and combined until it is no longer possible. The matching criterion used is: if corresponding entries in columns are different (i.e., one entry is a "0" and the other is a "1"), then they do not match. In all other cases, they do match. All column entries for the two columns must match for them to be combined. Once we combine all such columns, they are re-encoded with new states[3]. The algorithms we will discuss simply encode columns in order, using the minimum number of variables necessary. For example, if we have 8 unique columns, i.e. no two columns match, we use 3 variables to encode them. This entire process is then repeated with this new encoded set of variables and a new 2-set partition until no further decomposition is possible.

The FERD algorithms that we discuss all have the above in common. They differ in the number of partitions explored. The simplest method chooses one partition at random. It then tries to merge columns, encode new states, and then pick another partition at random from this new state until the one it picks

[2] Technically speaking, "0" is this parameter's lowest value but -m 1 yielded a slightly better performance and thus, was used.

[3] This is equivalent to the general vertex graph coloring problem.

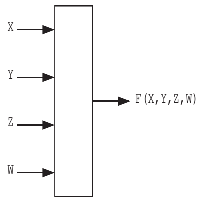

Figure 1: Lookup Table

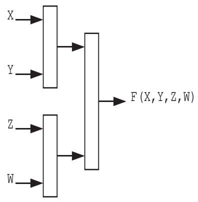

Figure 2: Decomposition

does not decompose to a smaller number of unique columns. We will call this version of FERD "lotto." The reader will note this is not even a hill-climbing algorithm because only one partition at a given level is explored.

A more complex algorithm looks at all partitions at a given level and 40 random partitions one level deeper. Essentially, this is a 2-ply look ahead search where we only explore 40 grand children. It chooses a particular partition over another based on the function's decomposed function cardinality (DFC). DFC is simply a measure of complexity for the circuit. DFC is calculated by adding the cardinalities of each of the subfunctions in the decomposition. The cardinality of an n-variable binary function is 2^n. We illustrate the measure in the figures. In Figure 1, we have a function on four variables with cardinality $2^4 = 16$. In Figure 2, we show the same function after it has been decomposed. The DFC of this representation for the original function is $2^2 + 2^2 + 2^2 = 12$. The DFC measures the relative complexity of a function. When we search through the possible decompositions for a function, we choose one with the smallest DFC. This decomposition is our learned concept. We will refer to this 2-ply look ahead FERD algorithm as "brute-force."

DFC has several properties not explored here. It suffices to say DFC is a special measure of complexity for circuits and functions in general [14]. In addition, the FERD approaches, unlike C4.5 do not have restrictions of finding concepts like "XOR." Decomposition in general uses universal gates to avoid any bias to a specific representation.

4 Espresso

Espresso is a logic minimization tool distributed by the University of California at Berkeley as part of the *OCTTOOLS VLSI* tool set [9]. It accepts binary functions in various formats and produces a minimized 2-level (disjunctive normal form) representation of the input function.

Espresso was developed in the early 1980's as a tool for programmable logic array (PLA) design. The main objectives were to produce near globally optimum minimization with limited computing resources. At the time, many VLSI logic minimization tools were based on the classical Quine-McCluskey algorithm and were unsuitable for large problems. The two operations which fundamentally limited the Quine-McCluskey based routines were the facts that: generating all prime implicants is $\bigcirc(\frac{3^n}{n})$, and the extraction of a minimum prime cover is known to be NP-complete [3].

Espresso's basic goal, as a logic minimization tool, is to take a verbose representation of a logic function and produce a condensed representation by exploiting the inherent structure of the function's "on-set[4]" $\bigcup$ "don't care-set[5]." This exploitation is accomplished by extrapolating across the "don't care-set" in a way which imposes an optimal 2-level structure on the function. This is analogous to a machine learning system which accepts as input a number of samples of a function, "learns" the underlying structure, and extrapolates across the unknown part of the function using the learned structure. Espresso operates in fundamentally the same manner except it is only able to learn (recognize, extract, impose) functions which lend themselves well to 2-level structure.

The first generation of Espresso, Espresso-I, relied on improving the internal efficiencies of the basic algorithms. These improvements resulted in incremental gains in Espresso's performance. In the middle 1980's it was demonstrated experimentally that the bottleneck in the basic logic minimization algorithm was in generating all prime implicants [15]. Indeed, a vast majority of these primes are immediately discarded as part of the minimum prime cover extraction. This work led to new versions of Espresso which altered the basic algorithm by generating only a subset of the total prime implicant set. The resulting improvement in runtime allowed for development of Espresso-EXACT which is able to produce results which are provably minimal for many functions. Minimal or optimal in this context refers to finding a 2-level sum of products representation (disjunctive normal form) for the circuit such that no other representation contains more products. For example, the expression ($X_1\overline{X_2}$ and

[4]A function's "on-set" is the set of all product terms such that the output is a "1." For our studies, the on-set is the set of all data records with an ouput of "1."

[5]A function's "don't care-set" is the set of all product terms such that the output is an unknown (not a "0" or a "1"). For our studies, the don't care-set is the set of all possible records where the output is unknown.

$\overline{X_1}X_3)$ contains 2 products.

The general 8-step Espresso algorithm is described in detail in [3] and is briefly outlined here.

- •1. Complement: compute the complement of the input and the "don't care-set" (this is the off-set).
- •2. Expand: expand each implicant into a prime implicant.
- •3. Essential Primes: extract the essential primes.
- •4. Irredundant Cover: find a minimal irredundant cover.
- •5. Reduce: reduce each implicant to a minimum essential implicant.
- •6. Iterate 2,4,5 until no improvement.
- •7. Lastgasp: try one last time to reduce, expand and cover.
- •8. Makesparse: make the structure as sparse as possible

The fundamental Espresso algorithm has since been modified to include some improvements and heuristic tuning but is still regarded as the standard 2-level logic minimization tool in the (VLSI) design automation community.

5 Small Problems

5.1 Overview

This section describes the first set of test functions. Each of these functions consist of 8 binary variables with a single binary output. All of the functions are void of noise and missing values. Each function was chosen as to in some way represent a "database-like" concept. It is noted to the reader that the logic synthesis approaches generalize to discrete and continuous data fields [13] as well as multi-valued outputs.

5.2 Experimental Design

Here, we show an abridged version of our test results with brute-force decomposition (FERD), Espresso, and C4.5. Table 1 shows the actual functions tested.

[6] C4.5 with pruning learned this function with 46 samples.

[7] The average with the better score for F_8 is 124.6.

	Original Function
F_1	$= (x_1x_3) + \overline{x}_2$
F_2	$= (x_1\overline{x}_2x_3)(x_4 + \overline{x}_6)$
F_3	$= (\overline{x_1 + x_2}) + (\overline{x}_1x_4x_6)$
F_4	$= \overline{x}_4$
F_5	$= (x_1x_2\overline{x}_4) + (x_3\overline{x}_5x_7x_8) +$ $(x_1x_2x_5x_6x_8) + (\overline{x}_3\overline{x}_5)$
F_6	$= x_2 + x_4 + x_6 + x_8$
F_7	$= (x_1x_2) + (x_3x_4) + (x_5x_6) + (x_7x_8)$
F_8	$= (x_1\overline{x}_2)$ XOR (x_1x_5)
F_9	$= (x_2$ XOR $x_4)(\overline{x}_1$ XOR $(x_5x_7x_8))$
F_{10}	$= (x_1 \Rightarrow x_4)$ XOR $(\overline{x_7x_8}(x_2 + x_3))$

Table 1: The Functions Tested

Original Function	Actual DFC	Average Error		
		C4.5	Decomposition	Espresso
F_4	2	0	0	0
F_1	8	0.32	0	0.96
F_8	8	6.35	0	5.64
F_6	12	2.48	3.72	2.64
F_3	12	1.28	2.72	3.52
F_2	16	2.76	2.4	12.86
F_{10}	20	17.52	8.18	17.16
F_9	20	13.79	6.55	16.54
F_7	28	20.69	10.53	5.69
F_5	36	10.52	11.11	8.44
Average		7.57	4.52	7.345

Table 2: DFC and Average Error

Original Function	Actual DFC	# of Samples		
		C4.5	Decomposition	Espresso
F_4	2	8	7	9
F_1	8	31	25	40
F_8	8	83[6]	25	51
F_6	12	74	67	76
F_3	12	61	76	54
F_2	16	97	126	113
F_{10}	20	200	60	181
F_9	20	224	104	205
F_7	28	256	126	51
F_5	36	249	251	229
Average		128.3[7]	86.7	100.9

Table 3: DFC and # of Samples Needed to Learn

The tests on the individual functions were as follows. First, each method was given a random set of data to train on ranging from 25 to 250 out of a total of 256 possible cases. Once the method was trained, the entire 256 cases were tested and the number of differences were recorded as errors. This procedure was repeated 10 times for a given sample training size in intervals of 25. Thus, the total number of runs for each function was 100 of varying sample size. None of the learning was incremental. All of the runs were independent.

For brevity, the comparison graphs were omitted in lieu of a condensed summary of each of the learning curves. For each function, the average number of errors for the entire run was recorded in Table 2. It was then possible to compare brute-force decomposition, Espresso, and C4.5 with these averages for a given function. Table 3 shows the number of samples necessary before the learning method obtains a concept such that in all ten separate runs, the number of errors was 0. The value at the bottom of each table is the average over all of the functions. The smaller the number here, the better the performance.

6 Analysis of Small Problems

Examining the data from these 10 functions, we can see the superior performance of the brute-force decomposition technique. The FERD (decomposition) approach has a lower average number of errors than C4.5 on more than half of the functions. Espresso also keeps pace with C4.5 and in fact demonstrates a more robust performance. Specifically, Espresso is able to learn functions F_7 through F_{10} more quickly than C4.5. This is because those functions have an "XOR" relation or include a replicated disjunctive concept. Espresso has no such limitation in constructing its 2-level minimization which makes it a more robust pattern finding tool.

The FERD approach is the clear overall winner in these tests with the small variable functions. This is because a "top-down" approach is possible with a small enough number of variables. It is also because DFC is a very good indicator of complexity. In fact, there is a correlation between the number of samples needed to learn a function and its DFC ($\rho = 0.9$ for FERD). FERD was able to outperform C4.5 for essentially the same reasons of Espresso. It outperformed Espresso because FERD has no tie to a specific structural representation like 2-level minimization. In other words, functions that do not lend themselves to a good 2-level minimization will give Espresso problems. FERD looks for and possibly finds decompositions which may be 4 levels deep or more. Furthermore, FERD uses universal gates which does not tie it to some specific representation.

The FERD approach and Espresso were also able to find discovered knowledge more easily than C4.5. For example, function 8 was listed in the table as (X_1 and $\overline{X}_2$) xor (X_1 and X_5). The FERD approach found the function: X_1 and $\overline{(X_2 \text{ xor } X_5)}$. This new function, equivalent to the original representation, is more simplified and in a sense, a discovery. On the other hand, C4.5 and Espresso yielded the more complex function expression $X_1\overline{X}_2\overline{X}_5$ or $X_1X_2X_5$. However, both FERD and Espresso found functions 4, 6, and 7 exactly as they were represented in Table 1. C4.5 found function 4 exactly as well. However, it found function 6 as: X_2 or $\overline{X}_2X_4$ or $\overline{X}_2\overline{X}_4X_8$ or $\overline{X}_2\overline{X}_4X_6\overline{X}_8$. C4.5's concept for function 7 is horrendously complicated! Some of the other functions FERD and Espresso found were less informative. However, those unusual representations exactly matched the outputs of the original function.

It is further noted that these trends relating to average error for FERD and C4.5 occur across a much larger set of 8 variable binary functions [6]. The same is true for Espresso and C4.5. However, we cannot over generalize conclusions from this small narrow domain. Instead, we need to see what these mechanisms can do with problems more analogous to real-world databases. More apropos to KDD and ML are the databases found at UCI which we explore in the next section.

7 Heart Disease Database

The Cleveland heart disease database[8] used in our experiments was the "processed" data where only 14 of the original 76 attributes were used in classification. For our purposes, we will further modify the data (convert to binary) as shown in Table 4 for testing with all learning algorithms. The intervals (for continuous type data) were basically chosen to be evenly distributed along the appropriate line segment. There were no special rules to delineate the data in order to be advantageous to a particular learning algorithm. Some of the variables were discrete and had a natural encoding. E.g., variable 9 for chest pain had 4 values, thus it was encoded with 2 binary variables.

[8]The authors wish to express their gratitude to Dr. Robert Detrano, M.D., Ph.D. and the V.A. Medical Center, Long Beach and Cleveland Clinic Foundation for the use of their database.

One additional alteration was made to the database. Six of the 303 patient records contained some missing values. They were all removed because at present, the logic synthesis tools have no reasonable way of handling this data. It was decided rather than encoding the missing value as a separate value or using some other method, we would simply omit those records. Also, because of the binary encoding, it happened that 2 different records in the original database were encoded to the same values including their output. This duplicate entry was discarded. The end result is a transformation of the 14 variable discrete/continuous data to a 26 binary variable problem. There are a total of 296 records.

We now proceed to the test results from these methods. The graph presented is a learning curve (Figure 3), calculated in exactly the same manner as the smaller data set problems. The number on the x-axis is the number of samples each learning method had for training. Each was then tested on the entire 296 samples and the number of errors (the number of differences) were recorded. The "Chance Line" was created by randomly guessing a "0" or a "1" output with an equal chance for each guess[9]. Thus, this algorithm would be expected to achieve 50% correctness on the remaining cases.

8 Analysis of Larger Problems

In our tests with the heart disease database, the learning curve shows the relatively equal performance of Espresso and C4.5. Both of which are better than the lotto decomposition method. The brute-force decomposition strategy is not tractable for this problem. It would have to explore $40 * 2^{26}$ partitions just at the top level.

Various other tractable search strategies (not shown here) for FERD perform roughly the same as the lotto strategy yet none of them performed at the level of C4.5 or Espresso. The problem with FERD for a large number of variables and only a handful of samples is in the column matching. For example, suppose the 26 variable heart problem was split into a partition of two 13 variable sets. This creates roughly an 8,000,000 by 8,000,000 matrix with 200 or so entries with a "1" or a "0" and the rest are filled in with "don't cares."

	Heart Variable	Value	Binary Encoding
x_1 x_2	variable 3 Age		00 $\leq$ 30 01 31-50 10 51-70 10 $>$70
x_3	variable 4	Sex	0=female 1=male
x_4 x_5	variable 9 chest pain	typical angina atypical angina non-anginal pain asymptomatic	00 =1 01 =2 10 =3 10 =4
x_6 x_7 x_8	variable 10 resting blood pressure (mm Hg)		000 $\leq$ 100 001 101-115 010 116-130 011 131-145 100 146-160 101 161-175 110 176-190 111 $>$190
x_9 x_{10} x_{11}	variable 12 serum cholesterol (mg/dl)		000 $\leq$ 160 001 161-200 010 201-240 011 241-280 100 281-320 101 321-360 110 361-380 111 $>$380
x_{12}	variable 16 fasting blood sugar		0= $\leq$ 120mg/dl 1= $>$120mg/dl
x_{13} x_{14}	variable 19 resting electro- cardiographic results	normal having ST-T wave abnormality[10] showing probable or definite left ventricular hypertrophy[11]	00 =0 01 =1 10=2
x_{15} x_{16} x_{17}	variable 32 Maximum Heart Rate Achieved		000 $\leq$ 100 001 101-115 010 116-130 011 131-145 100 146-160 101 161-175 110 176-190 111 $>$190
x_{18}	variable 38 exercise induced angina		0=no 1=yes
x_{19} x_{20}	variable 40 ST depression induced by exercise relative to rest		00 $\leq$ 1.5 01 $>$1.5 - 3.0 10 $>$3.0 - 4.6 10 $>$4.6
x_{21} x_{22}	variable 41 slope of the peak exercise ST segment	unsloping flat downsloping	00 =1 01 =2 10 =3
x_{23} x_{24}	variable 44 number of major vessels colored by flourosopy	0-3	00 =0 01 =1 10 =2 11 =3
x_{25} x_{26}	variable 51 Thal	normal fixed defect reversible defect	00 =3 01 =6 10 =7
	Output	Healthy Disease(1-4)	0 1

Table 4: Heart Disease Database Binary Conversion

[9] In actuality, this database has a 55/45 distribution but the "Chance Line" created assuming that distribution is nearly identical to the one presented.

[10] T wave inversions and/or ST elevation or depression of $>$ 0.05 mV

[11] by Estes' criteria

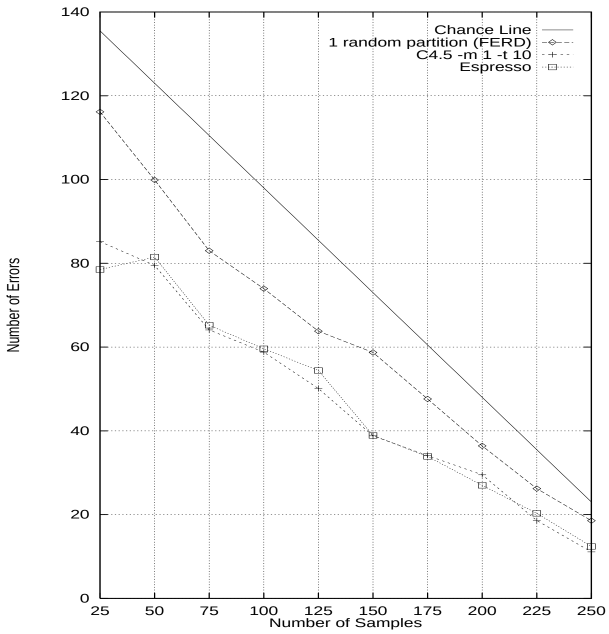

Figure 3: Learning Curve for the Heart Disease Data

It is extremely unlikely that we compare 2 numerical entries in this matrix. The majority of the time, we will be comparing 2 "don't cares" or some value and a "don't care." This means we will be initially combining columns without a real justification. What's more, since the encoding is essentially arbitrary, we introduce dependencies almost at random.

This problem of decomposition of functions with an extremely large number of "don't cares" limits the FERD approach to a small number of variables. In those instances, the decomposition matrix has many more "cares" and meaningful column merges can be accomplished. It may be possible for the FERD approach to be more successful in large variable cases where we have many samples. In our 26 variable heart problem, we would probably need hundreds of thousands of samples. This would create more meaningful column comparisons. However, we are still left with the encoding problem. Once we have merged columns, they need to be labeled in such a way as to carry meaning to the next level of decomposition.

Espresso on the other hand, is able to find a better representation through 2-level minimization. Large variable functions are tractable for Espresso because of its constructive approach. Its performance rivals that of C4.5. We feel this performance is owed to Espresso's ability to construct a solution of low complexity. The synthesis of low complexity circuit subfunctions is analogous to the construction of maximum information gain decision trees. A decision tree and the 2-level minimization of a circuit, contain more structural information than a look-up table. Logic minimization, which has been around for decades, provides further proof that a low complexity solution is the right structure to search for, whether it is in KDD, ML, or Logic Synthesis.

9 Conclusions

It is difficult to generalize on the few examples we have explored. However, it is clear that the logic synthesis community has much to offer. The FERD approaches to circuit design (i.e. the learning from data problem) are superior in the limited binary domain with 12 variables or less. But when we move to larger variable problems, the methods break down. This is the result of trying to search a vast space of partitions and trying to match columns with an extremely large number or "don't cares." How to encode these columns is yet another impasse for this approach. On the other hand, a constructive approach like 2-level minimization seems more promising for larger variable problems, as well it should. In the examples we have explored, Espresso can perform as well as C4.5. Moreover, Espresso thrives at learning 2-level concepts (disjunctive normal form). The commonality of desired low complexity exists in knowledge discovery as well as in circuit design. Tools and methods in logic synthesis, like Espresso, need to be studied for their applicability to the KDD and ML communities.

10 Future Work

It was previously thought that by refining the FERD approaches, we could achieve the same performance for larger variable functions. From these studies, that does not appear to be possible. Instead, a more promising direction may be to try to explore methods of removing variables from the original data set until a FERD approach becomes tractable. This dimensionality reduction is not a new idea but in this case, the FERD solution is the superior approach.

Missing values and noise are still not adequately part of the circuit design world but are a reality in KDD and ML. It will be necessary to find solutions to these issues if we are to use logic synthesis tools in KDD.

References

[1] Robert L. Ashenhurst. The decomposition of switching functions. In *Proceedings of the International Symposium on the Theory of Switching*, April 1957.

[2] Mark L. Axtell, Timothy D. Ross, and Michael J. Noviskey. Pattern theory in algorithm design. In *NAECON Proceedings*. IEEE and AIAA, May 1993.

[3] Robert K. Brayton, Gary D. Hachtel, Curtis T. McMullen, and Alberto L. Sagiovanni-Vincentelli. *Logic Minimization Algorithms for VLSI Synthesis*. Kluwer Academic Publishers, Boston, 1984.

[4] Arlindo Manuel Limede de Oliveira. *Inductive Learning by Selection of Minimal Complexity Representations*. PhD thesis, University of California at Berkeley, December 1994.

[5] Jeffrey A. Goldman. Pattern theoretic knowledge discovery. In *6th IEEE International Conference on Tools with Artificial Intelligence*. IEEE, November 1994.

[6] Jeffrey A. Goldman, Timothy D. Ross, and David A. Gadd. Pattern theoretic learning. In *AAAI Spring Symposium Series on Systematic Methods of Scientific Discovery*. AAAI, March 1995.

[7] L. Hyafil and R.L. Rivest. Constructing optimal binary decision trees is NP-Complete. *Information Processing Letters*, 5(1):15–17, 1976.

[8] Ron Kohavi. Bottom-up induction of oblivious read-once decision graphs. In *European Conference on Machine Learning*, 1994.

[9] Electronics Research Laboratory. OCTTOOLS-5.2. User's guide, University of California at Berkeley, May 1993.

[10] Giulia Pagallo and David Haussler. Boolean feature discovery in emperical learning. *Machine Learning*, 5:71–99, 1990.

[11] Gregory Piatetsky-Shapiro, Christopher Matheus, Padhraic Smyth, and Ramasamy Uthurusamy. Kdd-93: Progress and challenges in knowledge discovery in databases. *Artificial Intelligence Magazine*, 15, No. 3:77–82, Fall 1994.

[12] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, Palo Alto, California, 1993.

[13] Timothy D. Ross, Jeffrey A. Goldman, David A. Gadd, Michael J. Noviskey, and Mark L. Axtell. On the decomposition of real-valued functions. In *Third International Workshop on Post-Binary ULSI Systems in affiliation with the Twenty-Fourth International Symposium on Multiple-Valued Logic*, 1994.

[14] Timothy D. Ross, Michael J. Noviskey, Timothy N. Taylor, and David A. Gadd. Pattern theory: An engineering paradigm for algorithm design. Final Technical Report WL-TR-91-1060, Wright Laboratory, USAF, WL/AART, WPAFB, OH 45433-6543, August 1991.

[15] R. Rudell and A. Sangiovanni-Vincentelli. Exact minimization of multiple-valued functions for PLA optimization. *Proceedings of the International Conference on CAD*, pages 352–355, 1986.