New Functional Representation for the Decomposition of Machine Learning Problems

Craig M. Files
Department of Electrical Engineering
Portland State University
Portland, OR 97207-0751, USA
email: cfiles@ee.pdx.edu

Abstract

The central idea in machine learning is to gather information from a given data set. This can be a very difficult task because practical databases are usually very large. To address this difficulty, the assumption of Occam's Razor (simplest is best) and explicit domain knowledge *are used to reduce the search space [3, 4]. Machine learning is not an exact methodology or formal theory of learning, but is an attempt at learning based on heuristics and probabilities. Machine learning is becoming more and more important as computers provide relatively inexpensive means to collect and store data. Plus, it has been shown by [10] that logic synthesis methods (specifically functional decomposition) can be used for finding solutions in the machine learning paradigm.*

One problem in machine learning is the concept of missing attributes (input don't know). The concept of a missing attribute is much like an input don't care in logic synthesis. But, the concepts differ in that an input don't care covers all possible values of the input, while an input don't know only represents one possible value of the input. This paper presents an new functional representaton for functions with don't knows and extends the representation to functional decomposition.

1 Introduction

The term **machine learning** is defined as a process by which a machine gains the capability to solve a problem by examining *examples* or data. Machine learning is a process called *inductive learning* [6, 8] which uses empirical evidence, in the form of examples, to derive *rules* for the given data. These systems use *automated inductive inference* (in the machine learning paradigm) to give rules or meaning to the data (known as *classification*). **Rules** are operations on variables, such as the average function, addition, subtraction, etc. Finding rules on variables is a process of discovering patterns and relationships that exist between variable combinations [9]. In other words, learning is done by using all the known outputs of a given function to help in the determination of values for all the don't knows in the output of the given function. By treating examples as cares and considering functions with many don't knows, a machine learning problem can be directly converted to a logic synthesis problem.

Definition 1. *The components of a function $y = f(x_0, x_1, \ldots, x_{n-1})$, where the set $\{x_0, x_1, \ldots, x_{n-1}\}$ is defined as the* **input attributes***, minterms are defined as* **examples***, and the output y is the* **target concept** *to be learned.*

Definition 2. *During decomposition of machine learning functions, $y = F(x_0, x_1, \ldots, x_{n-1})$ any x_i is known as either an input attribute or an* **intermediate concept** *(e.g., a Φ composite function in Curtis decomposition).*

The setting of unknown values to known values is done by creating a network of multi-valued input and multi-valued output blocks by decomposing the original function into a hierarchical network of multi-leveled blocks (intermediate concepts). A machine learning algorithm is evaluated on its learning effectiveness by how it reduces the *error* of the resulting network. **Error** is how well the algorithm in question sets don't know terms to care terms. A common method for evaluating a machine learning algorithm is to select a *training set*, which is a random sampling of the original known values from the *test* function. The result of learning the *training set* is then compared to the original test function. If the expression has a high error rate then it does not approximate the *test* function well and is not a useful way to describe the function.

An example of a machine learning problem is teaching a computer to learn the concept of a tree. The user can specify data, stating that a tree has roots in the ground, has branches, and is made of wood. In most situations the more information given should lead to a better concept. Occasionally though, more information (e.g., samples, variables) may actually remove some items from the group. For instance, adding the concept that a tree has leaves would exclude all conifer trees, or deciduous trees in the winter time. By adding information it is possible that some elements may not be defined in the set. Without concise information about a subject, items that do not belong in the concept may be included. For example, given the initial description of a tree, a telephone pole is a tree.

Given that induction is a method of extrapolating samples of a function, the extrapolating process is very complex unless some reasonable simplification assumptions are used. For a Boolean function over n-attributes, the function has a truth table with 2^n rows. Any truth table with 2^n rows can represent 2^{2^n} different functions. Because of the large number of possible functions, it is difficult to find a hypothesis function g that approximates f given a small set of examples of f.

To allow reasonable results in the extrapolation process for finding g, assumptions must be used. The assumption that is generally made is the one of low complexity, as in **Occam's Razor** (also known as Ockham's Razor). In Occam's Razor the most likely hypothesis is the simplest one that is consistent with all observations. This assumption has been applied and theoretically analyzed in several papers, including [3, 4]. Plus, Occam-based learning theory has been extended to actual implementations.

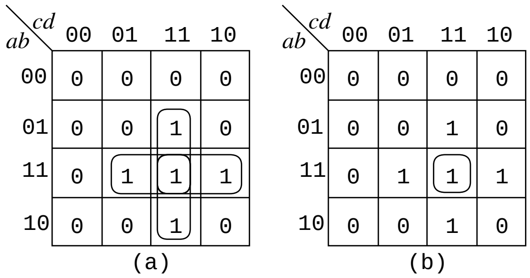

Figure 1. Don't knows in the input variables.

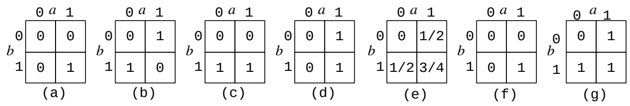

Figure 2. Using probability to find a function from cubes $a-$ and $-b$.

2 Missing Attributes

The possibility exists that data collected could have *missing attributes* in the example. A missing attribute is represented by an input don't know to denote the missing data. To associate the concept of a missing attribute to that of logic synthesis, an input don't know is akin to an input don't care. In logic synthesis, an input A is a don't care when, given a set of all other inputs, all possible values of A do not result in a change in the output of the function.

Definition 3. *An* **input don't care** *is defined as covering all possible values for a given variable.*

In machine learning, the possibility exists that data collected could have a missing attribute in the data.

Definition 4. *An* **input don't know** *is defined as representing one (unknown) data value, not all possible values for a given variable.*

For example, a questionnaire asks for a person's age. It is possible that the person does not want to state his/her age; in general, instead of guessing at an approximate age, a don't know is placed in the category. In terms of logic synthesis, a don't care implies that the person is all the ages between 0 and 150. While a don't know implies that a person has only one age between 0 and 150.

One possible method of determining missing attributes is to find two input cubes C_1 and C_2 that intersect and have the same output value. Thus, $C_1 \cap C_2 \neq \emptyset$ and the minimal solution for the missing attributes for these two cubes is $C_1 \cap C_2$. This is illustrated in the next example.

Example 1. *Given the set of input cubes (a dash indicates a don't know, or a don't care): $-bcd$, $a-cd$, $ab-d$, and $abc-$. Figure 1a shows the logic synthesis reduction method, where all the minterms are placed in essential prime implicants. The result is a minimum circuit with four product terms and one sum term. Figure 1b shows the machine learning reduction. The cube: $-bcd$, is thought to have a don't know such that $-bcd$ specifies that either the minterm $abcd$ exists or the minterm $\overline{a}bcd$ exists. Because the intersection of all the input cubes is $abcd$, the minimal solution is $abcd$. (Note: the minimal solution does not imply that it is the best solution).*

Another method of determining missing attributes is to use probabilities. Probabilities are used to determine the value of a missing attribute by determining how probable a value is. A baseline probability is defined as any value occurring more than X-times. If this occurs, then the value is accepted and a don't know can represent all values in the range when the baseline probability is zero or it is possible that no values are in the range. This methodology can be performed by using a preprocessing algorithm that determines (probabilistically) what the don't know values should be changed too. The following example illustrates the concept of using probabilities for determining the value of a don't know.

Example 2. *Figures 2a through 2d show the possible combinations when given the cubes: $a-$ and $-b$. Because there are four combinations, then the probability of finding any of the Karnaugh maps in Figures 2a-d, is $\frac{1}{4}$. Figure 2e shows the probabilities of finding an output value with a value of one. By assuming the probability baseline in the interval [0,1], every minterm is calculated to be true or false. By setting the probability baseline to be greater than 0.5 then only the minterm ab is true, resulting in the Figure 2f. If the probability baseline is greater than 0.4, the function is shown in Figure 2g.*

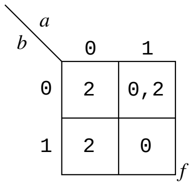

Figure 3. Missing attributes using relations; two values for the minterm $a\overline{b}$.

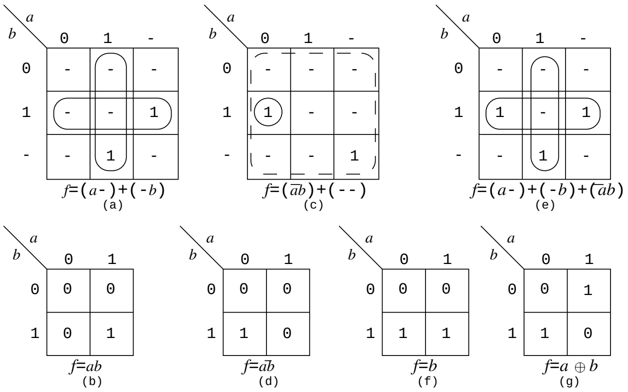

Figure 4. New representation to represent missing attributes.

One can appreciate that this topic is of great importance to the machine learning community and the author is unaware of any published papers that fully explore this topic. In [7], the authors use relations as an attempt at this problem, but the relations are only used to identify overlapping cubes. The relations treat don't knows as don't cares, but overlapping cubes of different values are stored. Example 3 illustrates the concept of a relation.

Example 3. *Given the cubes: $a-$ and $-\overline{b}$, the relation recognizes that the two cubes overlap at $a\overline{b}$. For instance, if $a-$ has the output value 0, and $-\overline{b}$ has the output value 2 then the cube $a\overline{b}$ has the output value of 0 or 2. The use of relations only preserves the information that the output of the minterm $a\overline{b}$ has two different values. This is shown in Figure 3a.*

For logic synthesis methods to be used on machine learning problems, missing attributes/input don't knows need to be represented in the data structure. A possible representation is to create a new cofactor for each variable that contains don't know information.

Theorem 1. *Given a Q_f-valued function $f(x_0, x_1, \ldots, x_{n-1})$, each variable x_i has value Q_i, so that f may be denoted as $f : Q_0 \times Q_1 \times \ldots \times Q_{n-1} \rightarrow Q_f$. To represent an input don't know, each variable x_i has value $Q_i + 1$, where the $(Q_i + 1)^{th}$ value represents a don't know. The function f may now be denoted as $f : (Q_0 + 1) \times (Q_1 + 1) \times \ldots \times (Q_{n-1} + 1) \rightarrow Q_f$. This is called the* **Don't Know** *(DK) representation.*

Proof. $f : Q_0 \times Q_1 \times \ldots \times Q_{n-1} \rightarrow Q_f \subset f : (Q_0 + 1) \times (Q_1 + 1) \times \ldots \times (Q_{n-1} + 1) \rightarrow Q_f$, thus, the original function is still represented by the DK representation of the function.

□

For the partition $A|B$, a **DK partition matrix** is defined as the $(Q_0 + 1) \times (Q_1 + 1) \times \ldots \times (Q_{n-1} + 1)$ functional values of f, arranged in $|Q_A|$ rows, and $|Q_B|$ columns. An example of a DK partition matrix is shown in Figure 4a, where the partition $\{a\}|\{b\}$ is shown. In the original function, a and b are two valued variables, but to represent input don't knows the added cofactor '-' is used.

After creating the DK representation of f, and forcing the input don't knows in the function to values, it is necessary to transform the DK representation of f back to f's original functional representation.

Theorem 2. *Given that f is denoted as $f : (Q_0 + 1) \times (Q_1 + 1) \times \ldots \times (Q_{n-1} + 1) \rightarrow Q_f$ then it is possible to transform f back to $f : Q_0 \times Q_1 \times \ldots \times Q_{n-1} \rightarrow Q_f$, iff each $Q_i + 1$ value for variable x_i is not defined.*

Proof. If each $Q_i + 1$ value is not defined, then all input don't knows in the function are set to a given value. The addition of the $Q_i + 1$ value of each variable is not needed and can be removed. This has no effect on f because the $Q_i + 1$ value was not originally defined in the original function.

□

Example 4. *Given the cubes: $a-$ and $-b$, the Karnaugh map for $f(a, b)$ is shown in Figure 4a. The '-' for an input represents a don't know. The cube $a-$ represents either minterm $a\overline{b}$ or ab, this is shown in the Karnaugh map by circling all the values in column with $a = 1$. The cube $b-$ is shown in the Karnaugh map by circling all values in the row when $b = 1$. Because the cubes $a-$ and $-b$ intersect on the Karnaugh map at ab, one possible solution is to set the don't knows of the cubes $a-$ and $-b$ to result in the minterm ab, as shown in Figure 4b (the input don't knows have been removed from the Karnaugh map shown in Figure 4a). In Figure 4b, f is denoted as $f : 2 \times 2 \rightarrow \{0, 1\}$.*

Given the cubes $\overline{a}b$ and $--$, the Karnaugh map for f is shown in Figure 4c. The dotted line is used in the Karnaugh map to illustrate that the cube $--$ represents all values in f. Because, $\overline{a}b \cap -- = \overline{a}b$, a possible solution is $\overline{a}b$ as shown in Figure 4d.

Given the cubes $a-$, $-b$ and $\overline{a}b$, the Karnaugh map for f is shown in Figure 4e. Two possible solutions are shown in Figures 4f,g.

It is also possible to extend the DK representation to MDDs, where each non-terminal node in the decision diagram has an extra cofactor. The extra cofactor defines the input don't know of each variable in the function. This is called the **DK MDD** representation. Example 5 illustrates the DK MDD representation.

Example 5. *Given the cubes shown in Figure 5a, the BDD representation of these cubes using don't cares is shown in Figure 5b. Note that anytime an output value is not given by one of the input cubes, the output is thought to be a don't care. Using the DK MDD representation of holding don't knows is shown in Figure 5c. The diagram in this case is much different than that of Figure 5b. The best choice is to map the don't know values onto care values (not the don't care output values). Two mappings are possible by doing this. The first is by taking the don't know* a_0b_- *and mapping it onto the known value* a_0b_1*. The second is by mapping* a_1b_- *to* a_1b_0*. These two mappings produce the DK MDD structure shown in Figure 5d.*

Figure 5d illustrates the case of a don't know whose value cannot be mapped to a care term. In fact, a_-b_0 *cannot be mapped onto* a_1b_0 *because the output values are different. But, by a process of elimination, the only output value that the don't know (*a_-b_0*) can be mapped to is* a_0b_0*. This produces the BDD (not reduced) structure shown in Figure 5e.*

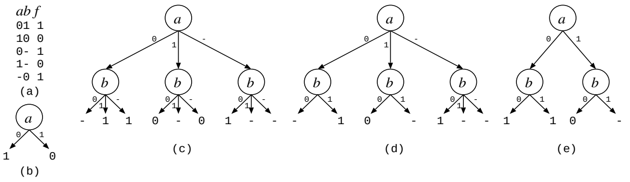

Figure 5. MDDs with input don't knows.

Functional decomposition may also be done using this representation of don't knows. This decomposition is called the **Don't Know Decomposition** (DKD).

Given a n-input, single output, multi-valued function f, don't knows are represented in f by adding an extra cofactor to each variable in f. This creates a DKD partition with two types of columns and rows: a non-don't know, which does not represent a don't know for an input variable; a don't know, which does represent a don't know for a given input variable.

Definition 5. *Two columns in a DKD partition are* **compatible** *if:*

1. *for every non-don't know row, the output value of both columns are equal, or at least one of the output values is a don't know,*

2. *any associated don't knows represented by either column must intersect with any of the non-don't know represented row output value-sets of the other column.*

The column multiplicity of a partition matrix is identified when compatibilities between non-don't know represented columns are present. Columns that represent don't knows are not used for finding column multiplicities, they are only used for representation purposes.

b \ a	0	1	2	-
0	0	-	1	1
1	0	0	-	-
-	-	1	0	-
	0	1	1	

Figure 6. Don't know compatible columns.

Example 6. *Definition 5 is illustrated graphically in Figure 6. Column "0" and column "1" are compatible for every non-don't know row, but the don't know represented in column "1" does not intersect with any of the values in column "0". This implies that column "0" and column "1" are not compatible. column "0" and column "2" are also not compatible because the columns are not compatible in the top row. column "1" and column "2" are compatible because for every non-don't know row the columns are compatible and the don't know represented in both columns, intersect with values in the other column. There is no reason to evaluate column "-" because it represents the don't know column, and does not affect the column multiplicity.*

Example 7. *Figure 7 illustrates Curtis decomposition with don't cares (Figure 7b) and DKD (Figure 7c) with bound variables* a *and* b *and given the input cubes in Figure 7a. The decomposition process forces the don't know values shown in Figure 7c to create the partition matrix in Figure 7d. Of course, many different decompositions are possible, but it is important to show that the decomposition process actually leads to the setting of the don't knows.*

From the example it is important to notice that using Curtis partitions with don't cares has a larger column multiplicity ν than that of the DKD partitions. Using DKD partition matrices will result in the same or smaller decompositions than using Curtis decomposition partitions with don't cares.

References

[1] M. Anthony and N. Biggs. *Computational Learning Theory*. Cambridge University Press, New York, NY, 1992.

[2] C. Apte, S. J. Hong, S. Prasad, and B. Rosen. Ramp: Rules abstraction for modelling prediction. *Technical report*, 1995.

[3] A. R. Barron and R. L. Barron. Statistical learning networks: A unifying view. *Symposium on the Interface: Statistics and Computing Science*, page 12, 1988.

[4] A. Blumer, A. Ehrenfeucht, D. Haussler, and M. K. Warmuth. Occam's razor. *Information Processing Letters*, pages 377–80, 1987.

[5] S. J. Hong. RMINI: A heuristic algorithm for generating minimal rules from examples. *Pacific Rim International*

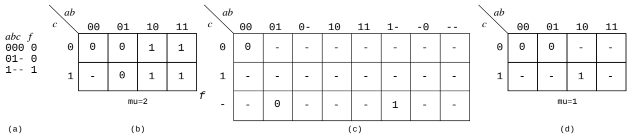

Figure 7. Decomposition with input don't knows (a) input cubes (b) Curtis partition $\{c\}|\{a,b\}$ (c) DKD partition $\{c\}|\{a,b\}$ (d) after don't knows forced to values

Conference on Systematic Methods of Scientific Discovery, 1994.

[6] R. Michalski. A theory and methodology for inductive learning. In R. Michalski, J. Carbonell, and T. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach*, pages 83–134, 1983.

[7] M. Perkowski, M. Marek-Sadowska, L. Jozwiak, T. Luba, S. Grygiel, M. Nowicka, R. Malvi, Z. Wang, and J. S. Zhang. Decomposition of multiple-valued relations. *International Symposium on Multi-Valued Logic*, pages 13–18, 1997.

[8] J. Quinlan. Learning efficient classification procedures and their application to chess end games. In R. Michalski, J. Carbonell, and T. Mitchell, editors, *Machine Learning 1: an Artificial Intelligence Approach*, Redwood City, CA, 1983. Morgan Kaufmann.

[9] T. Ross, M. Axtell, M. Noviskey, and D. Gadd. Pattern theory paradigm for system design. *Midwest Symposium on Circuits and Systems*, pages 721–724, 1993.

[10] T. Ross, M. Noviskey, D. Gadd, and J. Goldman. Pattern theoretic feature extraction and constructive induction. *ML-COLT Workshop on Constructive Induction and Change of Representation*, 1994.