

An Algorithm for Extracting Logic Propositions from Numerical Data and its Implementation with DERIVE™.

Terence Etchells

School of Computing and Mathematical Sciences
Liverpool John Moores University, UK.

Abstract

This article is based on the work of Hiroshi Tsukimoto of the Systems and Software Engineering Laboratory, Toshiba Corporation [1]. The aim of this work is to review the work of Tsukimoto and implement his algorithms in **Derive for Windows**.

If a regression analysis has been performed on some data set, then the resulting expression is an algebraic one involving variables x , y , z , ... etc. depending on the number of variables. For example, imagine that we are given a set of data of three variables x , y and z . A linear regression plane of the form $z = ax + by + c$ can be readily obtained. What does this regression line tell us about the underlying structure of the data? What we require is a Logic proposition that describes the hidden structure that generates the data. For example such a proposition could be $z = (x \wedge \neg y)$, which would be interpreted as the property z is **true** if the property x is **true** and the property y is **not true** (i.e. false). The essence of this paper is to describe and implement an algorithm for extracting such propositions from given regression hyper planes or data.

Background

Logic Vectors

Propositions in Boolean logic can be represented as vectors, these vectors are called **Logic Vectors**. We will spend some time looking at the construction of Logic vectors and their building blocks **atoms**. As it was thought for some time that atoms were the building blocks of matter, atoms in the Boolean sense are the building blocks of Boolean functions.

Let us take the simplest Boolean function x . The function x can take only two values 0 or 1 (i.e. elements from the set $\{0,1\}$) as it is Boolean, and also consider its negation $\neg x$ which can also only take values $\{0,1\}$. The truth table of x and $\neg x$ is

x	$\neg x$
1	0
0	1

Figure 1

Truth table for $\neg x$

which tells us some thing very important, that all Boolean functions of a single variable will either be a contradiction (e.g. $x \wedge \neg x$), a tautology (e.g. $x \vee \neg x$), x or $\neg x$. This is because a truth table of a Boolean function of one variable can only contain the values $[1,1]$, $[1,0]$, $[0,1]$ or $[0,0]$. Here we have written the vertical column of the truth table as a horizontal vector, the idea of the truth values of a function being represented as a vector is paramount to the ensuing work. To illustrate this further we will study the truth tables of some Boolean functions of a single variable, namely $\neg((x \wedge \neg x) \vee \neg(x \wedge x))$; $\neg x \vee \neg(x \wedge x)$; $x \wedge x \vee x$.

As DERIVE is used extensively in this work we will use it from the outset to produce truth tables.

`TRUTH_TABLE(x, $\neg(x \cdot \neg x \cdot \neg(x \cdot x))$, $\neg x \cdot \neg(x \cdot x)$, $x \cdot x \cdot x$)`

„	<code>x</code>	<code>$\neg(x \cdot \neg x \cdot \neg(x \cdot x))$</code>	<code>$\neg x \cdot \neg(x \cdot x)$</code>	<code>$x \cdot x \cdot x$</code>	†
	<code>true</code>	<code>true</code>	<code>false</code>	<code>true</code>	
	<code>...</code>	<code>false</code>	<code>true</code>	<code>false</code>	
	<code>false</code>	<code>false</code>	<code>true</code>	<code>false</code>	‡

As it is desirable to have values of 0 or 1 instead of the variables `true` and `false`, the functions `TEST(arg)` `TRUTH(tt)` are introduced.

`TEST(arg) := IF(arg = true, 1, 0, arg)`

`TRUTH(tt) := VECTOR(VECTOR(TEST(ttSUBrSUBm), m, 1, DIMENSION(tt`)), r, 1, DIMENSION(tt))`

„	<code>x</code>	<code>$\neg(x \cdot \neg x \cdot \neg(x \cdot x))$</code>	<code>$\neg x \cdot \neg(x \cdot x)$</code>	<code>$x \cdot x \cdot x$</code>	†
	<code>true</code>	<code>true</code>	<code>false</code>	<code>true</code>	
	<code>...</code>	<code>false</code>	<code>true</code>	<code>false</code>	
	<code>false</code>	<code>false</code>	<code>true</code>	<code>false</code>	‡

„	<code>x</code>	<code>x</code>	<code>$\neg x$</code>	<code>x</code>	†
	<code>1</code>	<code>1</code>	<code>0</code>	<code>1</code>	
	<code>...</code>	<code>0</code>	<code>0</code>	<code>1</code>	
	<code>0</code>	<code>0</code>	<code>1</code>	<code>0</code>	‡

As we can see the application of the `TRUTH()` function simplifies the Boolean expressions on the top row of the truth table to their minimised (reduced) form. DERIVE's ability to reduce Boolean expressions to a minimum form automatically is utilised in later work. So any Boolean function of a single variable can be written in terms of the 'atoms' x and $\neg x$. Indeed, they may reduce to a tautology [1,1] or a contradiction [0,0], but these are not atoms because an atom must be orthogonal to all other atoms and be of unit size. We should think of these 'atoms' as orthogonal unit vectors in a 2 dimensional Euclidean space, and as we do not think of matter comprising of nothing, or indeed everything, the Logic vectors [0,0] and [1,1] are not atoms.

We now extend these ideas to Boolean functions of more than one variable. The atoms of single variable functions arose very naturally from the truth tables, so the truth tables of Boolean functions of x and y should help the development of the 2 variable atoms.

`TRUTH(TRUTH_TABLE(x, y))``

„	<code>x</code>	<code>1</code>	<code>1</code>	<code>0</code>	<code>0</code>	†
	<code>1</code>	<code>1</code>	<code>0</code>	<code>1</code>	<code>0</code>	
	<code>...</code>	<code>1</code>	<code>0</code>	<code>1</code>	<code>0</code>	
	<code>y</code>	<code>1</code>	<code>0</code>	<code>1</code>	<code>0</code>	‡

As it can be seen from the truth table above, for Boolean functions of two variables, the vector that represent their truth values have dimension $2^2 = 4$. We can see that the truth vector of $x = [1,1,0,0]$ and that of $y = [1,0,1,0]$, so neither x nor y are atoms in a 2 variable Boolean expression as they are not of unit length. Hidden in the truth table above is the structure of the atoms! Reading a 1 as a proposition and a 0 as its negation the truth table above can be read as;

$$[x \wedge y, x \wedge \neg y, \neg x \wedge y, \neg x \wedge \neg y]$$

We will now verify that $x \wedge y, x \wedge \neg y, \neg x \wedge y$ and $\neg x \wedge \neg y$ are indeed the atoms of Boolean functions of two variables x and y .

TRUTH(TRUTH_TABLE(x, y, x • y, x • ¬ y, ¬ x • y, ¬ x • ¬ y))

x	y	x • y	x • ¬ y	¬ x • y	¬ x • ¬ y	†
1	1	1	0	0	0	
1	0	0	1	0	0	
0	1	0	0	1	0	
... 0	0	0	0	0	1	‡

As we can see the Boolean expressions $x \wedge y$ and $x \wedge \neg y$ have the ‘truth’ vectors $[1,0,0,0]$ and $[0,1,0,0]$, respectively. We will now call theses ‘truth’ vectors, **Logic Vectors** to agree with the relevant literature.

We will now adopt the convention of omitting \wedge operators in Boolean expressions (as product is implied, so is consistent with scalar algebra), so that large expressions do not appear as bulky. Also, we will adopt the \bar{x} notation to represent the negation of x (i.e. $\neg x$), purely to save space.

The atoms for 3 variable Boolean expressions are;

$$[xyz, xy\bar{z}, x\bar{y}z, x\bar{y}\bar{z}, \bar{x}yz, \bar{x}y\bar{z}, \bar{x}\bar{y}z, \bar{x}\bar{y}\bar{z}]$$

and the number of atoms is $2^3 = 8$. So, in general the number of atoms for an n variable Boolean expression is 2^n .

T logic

A scalar algebraic model for classical logic

We seek an elementary algebraic model for classical logic. In other words a set of definitions and functions that allows ‘normal’ algebra to act in a Boolean fashion. The first problem that can be identified occurs with the simple \wedge operator. We consider the \wedge operator a product operator and $x \wedge x = x$, with elementary algebra then product of x and x is x^2 . However, if we linearise (i.e. chop the exponent off) we can mimic the \wedge operation.

This linearisation is defined by the function t_x , which maps the polynomial $f(x)$ to the linear polynomial $r(x)$.

Any polynomial $f(x)$ can be written in the form

$$f(x) = x(1-x)q(x) + r(x) \quad (1.1)$$

where $q(x)$ and $r(x)$ are the quotient and remainder of $\frac{f(x)}{x(1-x)}$, respectively.

Clearly, the substitution of 0 or 1 into $x(1-x)$ will result in 0, hence at 0 and 1 $f(x) = r(x)$, which is linear in x . Hence, we have an efficient (i.e. fast) method of linearising any polynomial, with respect to a particular variable; divide the function by $x(1-x)$ and find the remainder.

Fortunately, DERIVE has an internal function **REMAINDER(u, v, x)**, which will evaluate the remainder of the quotient u/v with respect to the variable x .

The next few lines of DERIVE commands show the linearisation

$\tau(x^3y^3 + x^2y^2 + xy + x + y + 1) = 3xy + x + y + 1$ in action.

```
REMAINDER(x3 · y3 + x2 · y2 + x · y + x + y + 1, x · (1 - x), x)
```

```
x · (y3 + y2 + y + 1) + y + 1
```

```
REMAINDER(x · (y3 + y2 + y + 1) + y + 1, y · (1 - y), y)
```

```
x · (3 · y + 1) + y + 1
```

The same expression in a slightly different form. We will now automate this process with user defined DERIVE functions. An extremely useful DERIVE function is **VARIABLES(u)**, which returns a vector of all the free variables in the expressions u , e.g. **VARIABLES(3xy+2x+3z)** returns $[x, y, z]$ when evaluated. The next two functions automate the linearisation procedure.

```
¼_AUX(w1w, v1v) :=  
(ITERATE([r1r + 1, REMAINDER(u1u, v1vSUBr1r(1-v1vSUBr1r), v1vSUBr1r)], [r1r, u1u], [1, w1w], DIMENSION(v1v)))SUB2
```

```
¼(w1w) := ¼_AUX(w1w, VARIABLES(w1w))
```

Programming notes:

r1r, w1w, v1v are obscure variables to avoid clashes with functions that may be defined later and call this particular function. It is not wise practice to use the same variables in a functions that call one another. The **V SUB r** function extracts the r^{th} element of the vector **V**. The use of auxiliary functions avoids the repetition of calculations, by passing evaluated results into other functions.

The τ function in action:

```
¼(x3 · y3 + x2 · y2 + x · y + x + y + 1)
```

```
x · (3 · y + 1) + y + 1
```

We now have a function that will automatically linearise a polynomial of any order and any number of variables. Functions of this type, with two or more variables are called **multi linear functions**.

The τ function can be used to simplify a sequence of \wedge operations on many variables, for example the Boolean expression $((x \wedge x \wedge y) \wedge (x \wedge y \wedge z)) \wedge (x \wedge z)$ converts to the elementary algebraic expression $((xxy)(xyz))(xz) = x^4 y^2 z^2$ and $\mathbf{t}(x^4 y^2 z^2) = xyz$, which converts back to $x \wedge y \wedge z$.

More formally we have

$$x \wedge y \Leftrightarrow \mathbf{t}(xy).$$

The \vee operation is the of the form $\mathbf{t}(x + y - xy)$. As a test we know that $y \vee y = y$, so using the definition for \vee we get

$$\mathbf{t}(y + y - y^2) = \mathbf{t}(2y - y^2) = 2y - y = y$$

so

$$x \vee y \Leftrightarrow \mathbf{t}(x + y - xy).$$

The last definition is that of the negation operator and we have

$$\neg x \Leftrightarrow \mathbf{t}(1 - x).$$

As an example the Boolean expression $x \vee (\neg x \wedge y)$ simplifies to $x \vee y$. Using the above definitions this translates to

$$\begin{aligned} \mathbf{t}(x + (1 - x)y - x(1 - x)y) &= \mathbf{t}(x + y - xy - xy + x^2 y) \\ &= x + y - xy \end{aligned}$$

which is of course $x \vee y$.

Orthonormal Expansions

In the ensuing work, it will be important to be able to express continuous algebraic expressions in their orthonormal form. For example, the expression $x + y - xy$ can be written as $xy + x(1 - y) + (1 - x)y + 0(1 - x)(1 - y)$. As we can see each of the terms is the continuous equivalents of the Boolean atoms, e.g. $(1 - x)y \equiv \neg x \wedge y$. Logic vectors naturally arise from expressions which are written in this orthonormal form. In our continuous Boolean logic, the expression $x \wedge y$ is equivalent to $x + y - xy$ and its Logic vector is $[1, 1, 1, 0]$. We can see from the identity

$$x + y - xy \equiv 1xy + 1x(1 - y) + 1(1 - x)y + 0(1 - x)(1 - y)$$

the logic vector appears naturally from the coefficients of each of the terms.

More formally, the orthonormal system is defined as

$$\mathbf{f}_i = \prod_{j=1}^n e(x_j) \quad (i = 1, \dots, 2^n, j = 1, \dots, n)$$

where $e(x_j) = 1 - x_j$ or x_j . In other words the \mathbf{f}_i 's are Boolean atoms.

For example, if $n=3$ then $\mathbf{f}_2 = e(x_1)e(x_2)e(x_3) = x_1 x_2 (1 - x_3)$. The definition above is not precise enough to help us determine when $e(x_j) = 1 - x_j$ or x_j , as we could see in the example we chose $e(x_3) = 1 - x_3$ but $e(x_1) = x_1$ and $e(x_2) = x_2$. If we are to successfully automate this process a more precise definition is called for.

We define $e(x_j, p) = \begin{cases} x_j & \text{if } p = 0 \\ 1 - x_j & \text{if } p = 1 \end{cases}$

and $BD(n, r)$ as the r^{th} digit (from the right) of n in binary form. For example, $BD(5, 3) = 1$ as $5 = 101_2$ and the third digit from the right is 1. Then we can now define f_i as

$$f_i = \prod_{j=1}^n e(x_j, BD(i-1, n+1-j)) \quad (i = 1, \dots, 2^n, j = 1, \dots, n)$$

we will elaborate on how this definition works.

In the example above, we used $n = 3$ variables and this generates the orthonormal expressions

orthonormal expressions	p values
$x_1 x_2 x_3$	000
$x_1 x_2 (1 - x_3)$	001
$x_1 (1 - x_2) x_3$	→ 010
$x_1 (1 - x_2) (1 - x_3)$	011
\vdots	\vdots

we can see that the switch to decide whether $p=0$ or 1 forms a binary sequence starting at 0 and finishing at $2^n - 1$.

So that this definition may be coded we need a function that will convert a decimal integer in to its binary equivalent. To make use of the binary digits we will need to produce a vector of the binary digits, e.g. convert 5 to the vector [1,0,1]. The function **BINARY_AUX(m2m, n2n)** produces such a vector of the first **m2m** binary digits of the decimal integer **n2n**. This function relies on the division by 2 algorithm that is taught (or used to be) in most high school mathematics curricula.

```

BINARY_AUX(m2m, n2n) := REVERSE_VECTOR (ITERATES(
  ...  $\frac{\text{MOD}(p2p, 2)}{2}$ , ...  $\frac{\text{MOD}(n2n, 2)}{2}$ , ...  $\frac{\text{MOD}(m2m - 1, 2)}{2}$  ...
)

```

We now define the functions

EE1(x1x, p1p) := IF(p1p = 0, x1x, 1 - x1x, 1 - x1x)

%4(vars, i1i, n1n) := %f EE1(vars SUBj, (BINARY_AUX(n1n, i1i)) SUBj)

to automate the above definitions.

To see these functions in action, we will find the orthonormal expressions for the variables $[x, y, z]$.

VECTOR(%4([x, y, z], i, 3), i, 0, 7)

$[x \cdot y \cdot z, x \cdot y \cdot (1 - z), x \cdot z \cdot (1 - y), x \cdot (y - 1) \cdot (z - 1), y \cdot z \cdot (1 - x),$
 $y \cdot (x - 1) \cdot (z - 1), z \cdot (x - 1) \cdot (y - 1), (1 - z) \cdot (x - 1) \cdot (y - 1)]$

To automate the process of producing a vector of the orthonormal expressions we define the functions:

```

num ORTHONORM_AUX(exps, vars, num) := VECTOR(%4(vars, i, num), i
, 0, 2 - 1)

```

```

ORTHONORM(exps) :=
ORTHONORM_AUX(exps, VARIABLES(exps), DIMENSION(VARIABLES(exps)))

```

Our task is to convert an expression, say $xy - x + 1$, into its orthonormal form. We can do this by writing, for example,

$$xy - x + 1 = axy + bx(1 - y) + c(1 - x)y + d(1 - x)(1 - y)$$

Clearly, we can find the coefficient a by substituting $x=1$ and $y=1$, (1,1) into the above expression. We get

$$1 \cdot 1 - 1 + 1 = a \cdot 1 \cdot 1 + b \cdot 1 \cdot 0 + c \cdot 0 \cdot 1 + d \cdot 0 \cdot 0 \Rightarrow a = 1$$

repeating with (1,0), (0,1) and (0,0) will give b , c and d respectively. Notice that we are substituting the co-ordinates of a square. In general, for functions of n variables we substitute the co-ordinates of an n dimensional hyper-cube into the function to find the elements of the logic vector.

The function **HYPER_CUBE(n3n)** produces a matrix of co-ordinates of the $n3n$ unit hyper-cube.

```

HYPER_CUBE(n3n) := VECTOR(BINARY_AUX(n3n, w3w), w3w, 0, 2^n3n - 1)

```

The following functions calculates the Logic vector of a Boolean function in T logic form.

```

LOGIC_VECTOR_AUX(u2u, vars, vect) :=
VECTOR( lim u2u, r2r, DIMENSION(vect), 1, -1)
vars ~ vect^m r2r

```

```

LOGIC_VECTOR_AUX2(u2u, vars) :=
LOGIC_VECTOR_AUX(u2u, vars, HYPER_CUBE(DIMENSION(vars)))

```

```

LOGIC_VECTOR(u2u) := LOGIC_VECTOR_AUX2(u2u, VARIABLES(u2u))

```

Using this function on $xy - x + 1$ we have;

```

LOGIC_VECTOR(x·y - x + 1)=[1, 0, 1, 1]

```

Logic Propositions from Regression Functions

Until the work of Tsukimoto and Morita there have not been any methods for extracting logical propositions from regression planes. For example, the function

$$z = 0.71x - 0.99y + 0.25$$

is obtained from a least squares fit of a set of data points. What is the underlying Boolean structure of the data. Looking at the equation, if we consider in the region $[0,1]$ near 0 is not true and near 1 is true. If we apply our method for finding Logic

vectors to this regression line we will get a vector whose elements are in the interval [0,1] instead of the set {0,1}.

LOGIC_VECTOR(0.71·x - 0.99·y + 0.25)=[-0.03, 0.96, -0.74, 0.25]

We now need to find the discrete Boolean Logic vector that is closest to this continuous 'Logic vector'. If \underline{C} is the continuous Logic vector and \underline{D} is the discrete Boolean Logic vector, \underline{C}_i is the i^{th} element of the continuous Logic vector and \underline{D}_i is the i^{th} element of the discrete Boolean Logic vector. Note that $\underline{D}_i = 0$ or 1 . The nearest Boolean vector minimises $\sum (\underline{C}_i - \underline{D}_i)^2$. Minimising each term independently, we have that $\underline{D}_i = 1$ if $\underline{C}_i \geq 0.5$ and $\underline{D}_i = 0$ otherwise. This approximation method is regarded as pseudo maximum likelihood method using the principle of indifference.

Applying this algorithm to the example above gives us the logic vector [0, 1, 0, 0] which happily is an atom and easily recognisable as $x \wedge \neg y$. So according to this method of approximation the nearest Logic proposition that describes the underlying Logic structure of the data is $z = x \wedge \neg y$.

The rest of this section describes the code for automating the extraction of the Logic propositions from multilinear regression hyper planes.

The maximum likelihood approximation is coded as

```
MAX_LIKELY_HOOD_AUX(vect1) :=
VECTOR(IF(vect1 < 0.5, 0, 1), rrr, 1, DIMENSION(vect1))
rrr
```

```
MAX_LIKELY_HOOD(ff) := MAX_LIKELY_HOOD_AUX(LOGIC_VECTOR(ff))
```

The maximum likelihood expression is automated with:

```
BOOLEAN_LIKELY_HOOD(ff) := MAX_LIKELY_HOOD(ff) • ORTHONORM(ff)
```

e.g. **BOOLEAN_LIKELY_HOOD(0.71·x - 0.99·y + 0.25) = x·(1 - y)**

Now the task is to convert this algebraic expression into its Boolean logic equivalent. This requires some inventive programming within the Derive environment. As space for this article is short listings and their purpose are given, the detail is left to the reader.

```
NEGATE(xx, rr) := IF(rr = 1, xx, ¬ xx)
```

Makes a variable xx $\neg xx$ if $rr=1$, leaves as xx else.

```
PROP_BINARY_AUX(nn) := VECTOR(BINARY_AUX(nn, xx), xx, 2nn - 1, 0, -1)
```

```
PROP_BINARY(ff) := PROP_BINARY_AUX(DIMENSION(VARIABLES(ff)))
```

Produces a hyper cube for a given expression of dimension

```
NEGATE_MAT_AUX(www, ddd) :=
```

```
VECTOR(VECTOR(NEGATE(www, (PROP_BINARY_AUX(ddd))), rr, 1, ddd), ss, 1, 2ddd, f)
```

```
NEGATE_MAT_AUX_1(www) := NEGATE_MAT_AUX(www, DIMENSION(www))
```

NEGATE_MAT(ff) := NEGATE_MAT_AUX_1(VARIABLES(ff))

Produces a $n \times 2$ matrix with the n variables of ff in the first column and their Boolean negation in the second column.

VECT_AND_AUX(vars, nnn) :=
 $(\text{ITERATES}(\text{„rrr} + 1, \text{vvv} \cdot \text{vars} \quad \uparrow, [\text{rrr}, \text{vvv}], \text{„1}, \text{vars} \quad \uparrow, \text{nnn} - 1))$
 $\dots \quad \text{rrr} + 1 \uparrow \quad \dots \quad 1 \uparrow$
nnn, 2

VECT_AND(vect) := VECT_AND_AUX(vect, DIMENSION(vect))

Converts a vector $[x, y, z, \dots]$ into the logical expression $x \wedge y \wedge z \wedge \dots$

VECT_OR_AUX(vars, nnn) :=
 $(\text{ITERATES}(\text{„rrr} + 1, \text{vvv} \cdot \text{vars} \quad \uparrow, [\text{rrr}, \text{vvv}], \text{„1}, \text{vars} \quad \uparrow, \text{nnn} - 1))$
 $\dots \quad \text{rrr} + 1 \uparrow \quad \dots \quad 1 \uparrow$
nnn, 2

VECT_OR(vect) := VECT_OR_AUX(vect, DIMENSION(vect))

Converts a vector $[x, y, z, \dots]$ into the logical expression $x \vee y \vee z \vee \dots$

ZERO_FALSE(vect) :=
 $\text{VECTOR}(\text{IF}(\text{vect} = 0, \text{false}, \text{vect} \quad , \quad \text{vect} \quad), \text{r}, 1, \text{DIMENSION}(\text{vect}))$
 $\quad \quad \quad \text{r} \quad \quad \quad \text{r} \quad \quad \quad \text{r}$

Converts any 0 in a vector to the statement false.

PROPOSITION_AND_AUX(ff, nmat) :=
 ϵ
 $\text{VECTOR}(\text{VECT_AND}(\text{nmat} \quad), \text{r}_-, 1, 2)$
 $\cdot \quad \quad \quad \text{r}_- \quad \quad \quad \text{f}$

PROPOSITION_AND(ff) := PROPOSITION_AND_AUX(ff, NEGATE_MAT(ff))

Creates a vector of all atoms for ff , e.g if $ff=xy+x$ then this function produces the vector $[x \wedge y, x \wedge \neg y, \neg x \wedge y, \neg x \wedge \neg y]$

ELEMENT_PROD(vect1, vect2) :=
 $\text{VECTOR}(\text{vect1} \quad \cdot \text{vect2} \quad , \text{r}_-, 1, \text{DIMENSION}(\text{vect1}))$
 $\quad \quad \quad \text{r}_- \quad \quad \quad \text{r}_-$

Produces an element by element vector product (Logic And).

EXTRACT_PROP(ff) := VECT_OR(ZERO_FALSE(ELEMENT_PROD(MAX_LIKELY_HOOD(f f), PROPOSITION_AND(ff))))

This final function takes any multilinear function and produces a pseudo-maximum likelihood Boolean function that best describes the underlying Boolean structure.

Experiment

These functions will generate ‘noisy’ data that has an underlying Boolean structure of $x \wedge \neg y \wedge z$.

[F3(x) := IF(x < 0.5, 0, 1), G3(y) := IF(y < 0.6, 0, 1), H3(z) := IF(z < 0.4, 0, 1)]

PP(t1t) := IF(t1t = 0, $\frac{\epsilon \text{RANDOM}(1)}{2}$, $\frac{\text{RANDOM}(1)}{2} + \frac{1}{2}$, $\frac{\text{RANDOM}(1)}{2} + \frac{1}{2}$, $\frac{1}{2}$, $\frac{1}{2}$ f

APPEND(APPEND(VECTOR(VECTOR(VECTOR([x, y, z, PP(F3(x) • - G3(y) • H3(z)]), x, 0, 1, 0.25), y, 0, 1, 0.25), z, 0, 1, 0.25)))

Using Derive's FIT() function we can find a multilinear regression curve of the form $w = axyz + bxy + cxz + dyz + ex + fy + gz + h$. In this experiment it turns out to be

$$\mathbf{0.794624 \cdot x \cdot y \cdot z - 0.111104 \cdot x \cdot y - 0.845843 \cdot x \cdot z + 0.689468 \cdot x - 0.773399 \cdot y \cdot z + 0.0832168 \cdot y + 0.735344 \cdot z + 0.203654}$$

We now use our function to extract the underlying Boolean proposition

$$\mathbf{EXTRACT_PROP(0.794624 \cdot x \cdot y \cdot z - 0.111104 \cdot x \cdot y - 0.845843 \cdot x \cdot z + 0.689468 \cdot x - 0.773399 \cdot y \cdot z + 0.0832168 \cdot y + 0.735344 \cdot z + 0.203654)}$$

$x \cdot \neg y \cdot z$ Success!!

References

- [1] The Discovery of Logical Propositions from Noisy Data, Hiroshi Tsukimoto, AAAI Workshop on Knowledge Discovery in Databases, pp 205-216.