Neural Networks, Derive and Finding Rules in Data

T.A. Etchells
School of Computing and Mathematical Sciences, Byrom Street,
Liverpool L3 3AF, England.

1. Introduction

The current interest in the use of neural networks in medical applications has raised the important issue of explaining individual inferences by the network [1]. This is important from a practical point of view in order to properly verify and validate the neural network model, but also from a legal standpoint as the doctrine of 'learned intermediaries' places on the clinician a responsibility to understand any inferences derived from the model.

This paper proposes a principled method to overcome common limitations in current rule-extraction models, meeting the requirements of accuracy in representing the decision inferences made by the network, together with computational efficiency to scale-up to high numbers input dimensions. The latter is particularly important in medical applications as the variables are often categorical, which in 1-from-N coding leads to a proliferation of binary attributes.

An important reason for not using the model structure directly in rule extraction is that the simplest and most informative decision surfaces may require complex networks, while over-simple networks e.g. of minimal size after pruning, may block those surfaces by pushing the model configuration into what amounts to local minima in the space of decision surfaces [2]. This apparently surprising result means that simple models can get in the way of simple rules and is consistent with good practice in network design where the network complexity is controlled by appropriate regularization, rather than by cutting out nodes [1].

The final section of this paper demonstrates some of the strategies used to implement this principled method in the computer algebra system Derive 6.

2. Theoretical framework

Tsukimoto [3] showed that the logic from a multi-layer perceptron with output $y = f(x_1, x_2, \cdots, x_n)$ can be optimally resolved by approximating the response surface in [0,1] as a 2^n multilinear function of the form

$$y \approx \sum_{i=1}^{2^n} \left(a_i \prod_{j=1}^{n} e(x_j) \right)$$ where $e(x_j) = x_j$ or $1 - x_j$ and a_i are constants to be determined.

For example a 2 input network can be approximated in [0,1]

$$a_1x_1x_2 + a_2x_1(1-x_2) + a_3(1-x_1)x_2 + a_4(1-x_1)(1-x_2)$$

The constants a_i are generated from the network output $f(x_1, x_2, \cdots, x_n)$ by substituting $x_j = 1$ or 0 into the network.

For example a 2 input network

$$f(x_1, x_2) \approx f(1,1)x_1x_2 + f(1,0)x_1(1-x_2) + f(0,1)(1-x_1)x_2 + f(0,0)(1-x_1)(1-x_2)$$

By definition, for the values $x_j \in \{0,1\}$

$$\sum_{i=1}^{2^n}\left(a_i\prod_{j=1}^{n}e(x_j)\right) = f(x_1, x_2, \cdots, x_n),$$

i.e. the multilinear function is equal to the network response at all the vertices of an *n* dimensional unit hypercube. The vertices of a unit hypercube corresponds to Boolean atoms and hence if the response of a network is greater than or equal to 0.5 at a vertex, the Boolean atom at that vertex is present within the disjunctive normal form of the Boolean function that best approximates the network's response surface.

For example, if a 2 input network is approximated to the multilinear function

$$0.1x_1x_2 + 0.9x_1(1-x_2) - 0.1(1-x_1)x_2 + 0.85(1-x_1)(1-x_2) \rightarrow x_1\bar{x}_2 \vee \bar{x}_1\bar{x}_2 = \bar{x}_2$$

This approximation is exact for the values $x_j \in \{0,1\}$ and it has been shown [3] that the best Boolean approximation to the logic fitted by the response surface is obtained by rounding the coefficients a_i to binary values. The main limitation of this framework is that it is exponential in complexity. Tsukimoto [3] proposes a polynomial algorithm for the MLP. However, this is a decompositional approach, that is to say it builds the logic of the response surface by propagating rules generated individually for each hidden and output node. It can be shown with a counter-example that this can lead to incorrect rules. Consequently the preferred approach to implement this theoretical framework is the so-called 'pedagogical rules ', which use only the input-output response of the network ignoring the detailed structure of the analytical model used to fit a smooth surface to the data. This scalar model of Boolean logic applied to multi-linear approximations of neural networks (which are exact in $\{0,1\}$) and the evaluation of the nearest Boolean atoms are exploited in the algorithm developed below.

3. OSRE Algorithm

In general, response surfaces are smooth approximations generated by interpolating the best fit to a finite data sample and are unreliable when extrapolating beyond the data space containing the sample. In a high dimensional input space the training data occupies only a small fraction of the total space and outside the regions where data are present the response surface is in fact being extrapolated which, with non-linear models, is generally unreliable. The algorithm presented here uses the training data to identify the region of the space for which the response surface has been accurately constructed. Consequently this algorithm scales as the number of data, rendering the method scaleable to high input dimensions on which the complexity depends linearly.

Tsukimoto's scalar Boolean model shows that we can find the optimal Boolean function that describes the outputs of a network with respect to binary inputs [3]. The Boolean function is constructed from the disjunction of the scalar atoms that have an activation greater than 0.5. This is an exponential problem as the number of atoms for n inputs is 2^n, however the number of atoms we need to find the disjunction of can be restricted to the number of atoms with activation greater than 0.5 in which the training data lies. One further consideration is that even with a manageable number of atoms to evaluate, each of the atoms is a conjunction of length n. Whilst powerful computer algebra systems can find disjunctions of a large number of Boolean functions, there is an issue in terms of the length of time to perform the simplification of this disjunction and the comprehensibility of any such simplification.

We can circumvent the need to find the simplification of all the active data atoms by use of the Boolean identity

$(x \wedge y) \vee (\neg x \wedge y) = y$,

where x is a variable and y is **any** Boolean function. These variables form a conjunctive factorisation of the simplification of all these atoms. For example, if it is found the atom $x_1\overline{x}_2x_3x_4$ is an activated data item and the neighbours $x_1x_2x_3x_4$ and $x_1\overline{x}_2x_3\overline{x}_4$ are the only neighbours that are not activated then the simplification of the conjunction of the activated data item and it's activated neighbours simplifies to $\overline{x}_2x_4B(x_1, x_3)$ where B is some Boolean function of x_1 and x_3.

Therefore, if an atom is active and its nearest neighbour in the x direction is also active, then variable x cancels out in the simplification of the disjunction of these two atoms. Another way of viewing this is that the activation of the surface does not change between the two atoms, so the surface does not change in the region of these two atoms and hence the variable that is negated has no influence on changing the response of the network in the region of these two atoms. However, if the activation of the atoms does change from greater than 0.5 to less than 0.5, then the variable is influential in changing the response of the network within this region. We term this a *discriminating* variable. By visiting all the 'neighbours' of a data item, the influential variables within this region of the space are identified. By visiting all the data items and their neighbours we can identify all the influential variables in the space occupied by the data and space immediately surrounding the data.

As Tsukimoto [3] shows, the disjunction of all the active atoms provide the optimal Boolean function for the network, this subset of rules then provide optimal rules for

the region occupied by the data. Hence this method is providing optimal rules for the training data set. In fact this algorithm transcribes to the RULENEG algorithm developed by Pop *et al* [4] and another related algorithm in [5] also traced back to the Probably Approximately Correct (PAC) framework of Valiant [6]. So far we have shown here that RULENEG will generate optimal rules from Binary data sets.

If there are categorical or ordinal variables that are not binary, then RULENEG cannot be applied to the data **even** if the variable attributes are represented by separate binary variables using 1-from-N coding. This is because RULENEG visits atomic neighbours, but the very nature of 1-from-N coding rules out the possibility of atomic neighbours. The difference between RULENEG and the algorithm presented here is demonstrated in the figures below, where black spheres indicate permitted atoms and the dotted arrows show the search pattern.

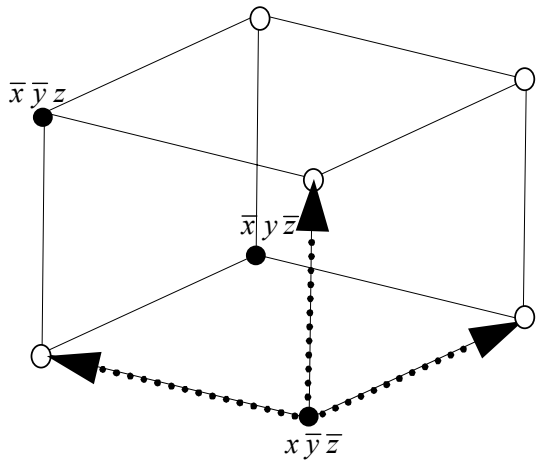

Fig. 1. The RULENEG algorithm inspects nearest neighbours in the atomic hypercube, which violates 1-from-N coding.

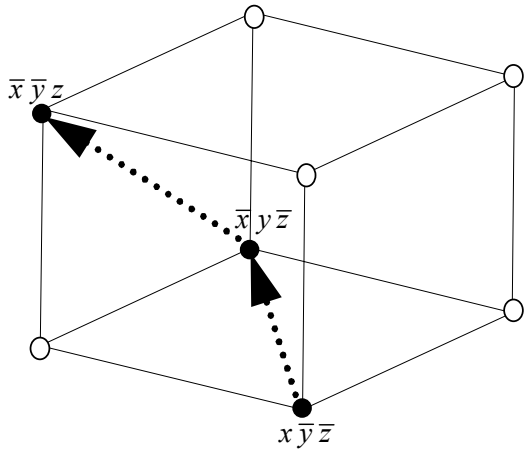

Fig. 2. The proposed algorithm inspects only the *singe variable atomic sub space* of valid codes.

We can determine the conjunctive factors of influential variables by searching the multi-variable space for changes in the network's response, sweeping each variable over its possible values whilst keeping the all other variables constant.

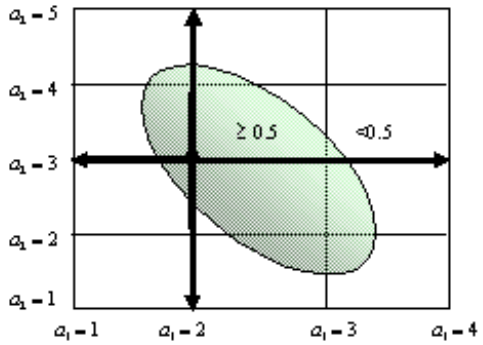

Fig. 3. Illustration of the proposed search algorithm for conjunctive factors of influential variables.

If there are m variables, each with up to n values there are at most n^m points to evaluate, which is not tractable to very high values of m. Moreover, restricting the search to the data predicted to be within class and searching in orthogonal directions from the data point reduces the search space to a polynomial problem.

Consider a variable with three attributes coded as 1-from 3 binary vectors. If each of the elements of these data items are x, y and z then corresponding atoms in Boolean form are $x\,\overline{y}\,\overline{z}$, $\overline{x}\,y\,\overline{z}$ and $\overline{x}\,\overline{y}\,z$ respectively. Atoms in these forms are the atomic representation of a single variable therefore we will refer to them as *single variable atoms* and the space they occupy as the *single variable atomic subspace*. If we find that presenting the binary variables to the network that inputs [1,0,0] and [0,1,0] have activations greater than 0.5 and [0,0,1] is less than 0.5, then the rule will be the disjunction of the atoms $x\,\overline{y}\,\overline{z}$ and $\overline{x}\,y\,\overline{z}$. However,

$$x\,\overline{y}\,\overline{z} \vee \overline{x}\,y\,\overline{z} = (x\,\overline{y} \vee \overline{x}\,y) \wedge \overline{z} = (x \vee_e y) \wedge \overline{z}$$

which is the equivalent of

$$(a_1 = 1 \vee a_1 = 2) \wedge a_1 \neq 3\,.$$

In an analogous fashion to RULENEG, we now have a methodology for finding rules by observing the changes in activations from each atom to its nearest valid neighbour. These atoms are comprised of sub-atoms that represent the different variables as indicated below

$$[a_{1,1}, a_{1,2}, a_{1,3}, \cdots, a_{1,n} \,|\, a_{2,1}, a_{2,2}, a_{2,3}, \cdots, a_{2,n} \,| \cdots | \, a_{m,1}, a_{m,2}, a_{m,3}, \cdots, a_{m,n}]\,.$$

Each of these sub atoms are unit orthogonal vectors which have only one element with the value 1 and the others being 0.

The goal of rule extraction is to find understandable rules that discriminate between the classes, that is they will be representative of a great deal of the data in class but are not representative of a great deal of data out of class. In essence in this rule extraction

algorithm constructs the disjunction of all of the positively activated *single variable atoms* present in the data. This is likely to lead to rules that are very large, in terms of number and of length, and hence not comprehensible.

We can now develop a rule extraction algorithm for multi-valued variables

Algorithm

1. Train the neural network with binary attributes using 1-from-N coding.
2. Loop over the training data.
3. For each data point if the activation is <0.5, move to the next data point
   Else, sweep the attributes of the first variable **whilst** keeping all the other variables values fixed.
   Evaluate the network response and if it is <0.5 then append 0 to a list, else append 1
   If the list contains only 1's then move on to the next variable, else
   Reverse the list, set **rule(1)**=vector of positions where the element=1
   Repeat for each variable and place the results in **rule(i)**
4. Add **rule(1)** AND **rule(2)** AND ….. **rule(n)** to a list **ConjunctRule**
5. Repeat for all training data, resulting in a set of rules the size of the training data.
6. Remove repeated rules.
7. Take each rule generated and test with the training data; find how many of training data that are in class are correctly classified by the rule (c) and how many of the out of class data are incorrectly classified as in class by this rule (i). For m data points in class and n data out of class, evaluate the ratio $\frac{c+(n-i)}{m+n}$. This ratio determines how accurate the rule was in discriminating between in and out of class for the training data. Order the rules in terms this ratio.
8. Take the rule with the highest ratio and determine if the disjunction of it with any the rules below equals the highest rule (i.e. the lower ratio rule subsumes into the higher ratio rule). If so remove all these lower rules. Repeat for rule with the next highest ratio and keep repeating down the hierarchy of rules until no further rules can be deleted from the list. This results in a list of rules in order of accuracy for the training data.

4. Results

The method was applied to standard datasets, namely the three standard Monks datasets and the Wisconsin breast cancer database, from the repository of machine-learning data at the University of California Irvine [7].

As the Monks problems are artificially generated, the rules are known. The data set consists of 6 variables that have the values

$a_1 = 1,2,3 \quad a_2 = 1,2,3 \quad a_3 = 1,2$

$a_4 = 1,2,3 \quad a_5 = 1,2,3,4 \quad a_6 = 1,2$

and two classes.

Rules extracted for Monks 1

The algorithm extracted 4 distinct rules only from the network
$(a_5 = 1) \vee (a_1 = 1)(a_2 = 1) \vee$
$(a_1 = 2)(a_2 = 2) \vee (a_1 = 3)(a_2 = 3)$

The known rule is
$(a_5 = 1) \vee (a_1 = a_2)$.

Rules Extracted for Monks 2

The algorithm extracted 15 distinct rules, too many to present here. One such rule was
$(a_6 = 1)(a_5 = 1)(a_4 \neq 1)(a_3 \neq 1)(a_2 \neq 1)(a_1 \neq 1)$
with every other rule being of this form, permutating through all 15 possibilities of the different indexes.

The known rule is
EXACTLY TWO of
$\{a_1 = 1,\ a_2 = 1,\ a_3 = 1,\ a_4 = 1,\ a_5 = 1,\ a_6 = 1\}$

Rules Extracted for Monks 3

The algorithm extracted 4 rules
$(a_5 \leq 3) \vee (a_5 \leq 3)(a_2 \leq 2) \vee (a_5 = 3)(a_4 = 1)$
The known rule is
$(a_5 = 3)\ (a_4 = 1)\ \vee (a_5 \neq 4)(a_2 \neq 3)$

In each of the Monks cases the algorithm has extracted a rule that is **exactly** equivalent to the known rule.

Wisconsin breast cancer data

This data has 9 variables each with discrete values from 1 to 10. There are two classes, benign and malignant. Initially the algorithm produced 8 rules, not mutually exclusive, from which we pruned out the rules which were least specific for malignancy of which there were 4 rules. Each of these rules was further reduced by removing attributes that had little effect on the sensitivity and specificity of the rule, resulting in the rule

$(a_6 \leq 9)(a_1 \leq 9)(a_8 \leq 2)$

Of the 114 benign tumours in the training data, this rule identifies 106 correctly and 7 of the 86 malignant tumours incorrectly, i.e. 185/200 correct classifications (92.5%). Of the 330 benign tumours in the test data, this rule identifies 313 correctly (specificity is 94.8%) and 4 of the 153 malignant tumours incorrectly (sensitivity is 97.4%). i.e. 462/483 correct classifications (94.8%).

5. Tools and Hurdles in Programming OSRE in Derive 6

In order to efficiently implement the OSRE algorithm in Derive, a number of functions needed to be developed that performed particular tasks.

Removing repetitions in a dataset

It is inefficient to have repetitions of the same data when using the OSRE algorithm as it causes the algorithm to search areas of the space that have already been searched.

In Derive, a *set* does not allow repetitions of the elements, so converting a vector of data items into a set will remove any repetitions and converting this set back to a vector produces a copy of the original vector with repetitions removed.

Step 1

To convert a vector to a set

```
VECTOR_2_SET(v) := MAP_LIST(v  , k_, {1, ..., DIM(v)})
                              k_
VECTOR_2_SET([1, 2, 3, 0, 4, 4, 0, 4, 0, 4, 0, 5, 5, 5, 7])

={0, 1, 2, 3, 4, 5, 7}
```

Step 2

Convert the set back to a vector

```
VECTOR(v, v, {0, 1, 2, 3, 4, 5, 7})

=[0, 1, 2, 3, 4, 5, 7]
```

or as a single function

```
COMPRESS(v):=VECTOR(u,u,VECTOR_2_SET(v))
```

The COMPRESS function is a particularly useful function in the OSRE algorithm as otherwise it would create many identical rules as it scans the data space.

For one particular part of the implementation of the OSRE algorithm I needed to be able to identify the position or positions of a particular element in a vector. This function did the trick:

```
Positions(a,v,count,v_,size):=
PROG(
 v_:=[],
 size:=DIM(v),
 count:=1,
 LOOP(
  IF(count>size, RETURN REVERSE(v_)),
  IF(IDENTICAL?(a,v sub count),
   v_:=ADJOIN(count,v_)
  ),
 count:+1
 )
)
```

Checking rules found by OSRE with the data

An important part of the OSRE algorithm is that once a rule has been identified it needs to be validated with the data. The algorithm produces rules in a form, for example

$\left[\{a_1,[1,2,3]\},\{a_3,[3,4]\},\{a_5,[1,2]\}\right]$

which is a representation of the conjunctive rule

$(1 \le a_1 \le 3) \wedge (1 \le a_3 \le 3) \wedge (1 \le a_5 \le 2)$ (*)

or more compactly

$(1 \le a_1 \le 3)(1 \le a_3 \le 3)(1 \le a_5 \le 2)$

The difficult task was to get Derive to interrogate the data with the rules generated to test for the sensitivity, how many times the rules is correct; and specificity, how many times the rule predicted incorrectly.

Step 1

Convert each rule in the form $\{a_1,[1,2,4,5]\}$

into the form

$\left((1 \le a_1 \le 2) \vee (4 \le a_1 \le 5)\right)$ (**)

In the first implementation in Derive 5.06, the rule in the form found in (*) was used within a SELECT function to determine the data that obeyed this rule. e.g.

SELECT($(1 \le a_1 \le 3) \wedge (1 \le a_3 \le 3) \wedge (1 \le a_5 \le 2)$, *a,* data*)*

However, the SELECT function did unexpected things within PROG and LOOPS and I never did resolve what was going wrong but a work around was to write a new SELECT function:

```
MYSELECT(u,k,n):=APPEND(VECTOR(IF(u,[k],[]),k,n))
```

In Derive 5.06 this method of identifying data that obeyed the rules worked very well. However, when the code was used in Derive 6, the both `SELECT()` and `MYSELECT()` functions for certain rules hung! The reason for this, as far as I can gather, is that the simplification routine within DERIVE that resolves expressions of the form found in (*) was changed. In effect expressions involving terms of the form (**) within terms of the form (*),

e.g. $\left((1 \le a_1 \le 2) \vee (4 \le a_1 \le 5)\right) \wedge (1 \le a_3 \le 3) \wedge (1 \le a_5 \le 2)$

were taking inordinate time to simplify or hung in a loop. This particular example does simplify in DERIVE 6 very easily but the OSRE algorithm genereates much bigger and complicated expressions and it was these expression that trouble DERIVE 6 (but not DERIVE 5.06).

This was actually a good lesson, in that the simplification of theses expressions is unnecessary for our purposes and a method of selecting the data that obeyed the rules without the need to construct (and hence simplify) the conjunctive expressions of the form found in (*).

Convert a rule vector into and interval

e.g $\left[\{a_1,[1,2,3]\},\{a_3,[3,4]\},\{a_5,[1,2]\}\right] \Rightarrow \left[1 \le a_1 \le 3, 3 \le a_3 \le 4, 1 \le a_5 \le 2\right]$

Step 1

Convert he rule vector into a matrix of the consecutive integers:

```
Intervals(v, size, intervals, interval, intervals, counter) :=
  Prog
    size := DIM(v)
    intervals := []
    counter := 1
    interval := [FIRST(v)]
    Loop
      If counter = size
         Prog
           interval := REVERSE(ADJOIN(v↓counter, interval))
           RETURN REVERSE(ADJOIN(interval, intervals))
      If v↓(counter + 1) > v↓counter + 1
         Prog
           interval := REVERSE(ADJOIN(v↓counter, interval))
           intervals := ADJOIN(interval, intervals)
           interval := [v↓(counter + 1)]
      counter :+ 1
```

e.g.

```
Intervals([1, 2, 3, 5, 6, 7, 9, 10, 11]) =
```

$$\begin{bmatrix} 1 & 3 \\ 5 & 7 \\ 9 & 11 \end{bmatrix}$$

Step 2

Convert each row of the matrix formed in step 1 into expressions of the form $(1 \le a_1 \le 3) \vee (5 \le a_1 \le 7) \vee (9 \le a_1 \le 11)$.

```
RulesToIntervals(rules,counter,size,displayrules,rule):=
PROG(
 counter:=1,
 size:=DIM(rules),
 logicrules:=true,
 displayrules:=[],
 LOOP(
  IF(counter>size,
   RETURN [REVERSE(displayrules),rules]
  ),
  rule:=RuleToInterval(rules SUB counter),
  displayrules:=ADJOIN(rule,displayrules),
  counter:+1
 )
)
```

CONTS

e.g.

```
RuleToInterval({a sub 1,[1, 2, 3, 5, 6, 7, 9, 10, 11]})=

9 ≤ a  ≤ 11 ∨ 5 ≤ a  ≤ 7 ∨ 1 ≤ a  ≤ 3
    1            1           1
```

Step 3

A final function that converts a conjunction of rules into a vector of the conjunctions of the form developed in step 2.

```
RulesToIntervals(rules,counter,size,displayrules,rule):=
PROG(
 counter:=1,
 size:=DIM(rules),
 logicrules:=true,
 displayrules:=[],
 LOOP(
  IF(counter>size,
   RETURN [REVERSE(displayrules),rules]
  ),
  rule:=RuleToInterval(rules SUB counter),
  displayrules:=ADJOIN(rule,displayrules),
  counter:+1
 )
)
```

e.g.

```
RulesToIntervals([{a , [2, 3]}, {a , [1, 2, 3, 5]}, {a , [1, 2]}, {a , [1]}])=
                  9             8                    2           1

⎡  2 ≤ a  ≤ 3   a  = 5 ∨ 1 ≤ a  ≤ 3   1 ≤ a  ≤ 2    a  = 1  ⎤
⎢       9        8           8           2         1        ⎥
⎢                                                           ⎥
⎣ {a , [2, 3]}  {a , [1, 2, 3, 5]}   {a , [1, 2]}  {a , [1]} ⎦
   9             8                    2            1
```

(original rule vector added as a second row in case it is required at a later stage).

Checking the accuracy of the rules with the data

In Derive 5.06 the conjunction of the rules found in step 3 of above,
e.g.

$$(1 \le a_9 \le 3) \wedge ((1 \le a_8 \le 3) \vee a_8 = 5) \wedge (1 \le a_2 \le 2) \wedge a_1 = 1, \qquad (***)$$

were constructed and the SELECT() function was used to check for the number of occurrences of the data that obey this rule. As mentioned earlier in DERIVE 6, some of the large expressions like (***) generated by this code took inordinate amounts of time to simplify and some expressions never simplified within hours. Another strategy was needed!

The ConjTest Function

The strategy was to not form a conjunctive expression form from the elements of the vector formed from the first row of the `RulesToIntervals()` function, but to test each rule separately on each data item and count every data item for which every rule worked. The function `False?(v)` returns a value of false if any element of v has the

values false, else it returns true, i.e. if all the rules hold for a data item `False?(v)` returns `true`.

e.g.

```
SUBST([2 ≤ a  ≤ 3, a  = 5 ∨ 1 ≤ a  ≤ 3, 1 ≤ a  ≤ 2, a  = 1], a, [1, 1, 0, 1, 1, 1, 9, 2, 1, 2, 0])=
      [     9       8           8          2       1   ]
```

```
[false, true, true, true].
```

```
False?([false, true, true, true]) = false
```

```
ConjTest(v,var,data,class,classindex,test,total_in,total_out):=
PROG(
  total_in:=0,
  total_out:=0,
  LOOP(
   IF(data=[], RETURN [total_in,total_out]),
   test:=False?(SUBST(v,var,FIRST(data))),
   IF(test AND FIRST(data) sub classindex=class,
    total_in:+1
   ),
   IF(test AND FIRST(data) sub classindex/=class,
    total_out:+1
   ),
   data:=REST(data)
 )
)
```

Class is the value of the class the data item is in and *classindex* refers to the column of the data matrix that contains the classification values. In the example in this paper, the class values of the data are in the 11^{th} column and the class values are either 1 or 0.

```
ConjTest([2 ≤ a  ≤ 3, a  = 5 ∨ 1 ≤ a  ≤ 3, 1 ≤ a  ≤ 2, a  = 1], a, data, 1, 11)=
         [     9       8           8          2       1   ]
```

```
[67, 26]
```

In this particular dataset the number of data in class 1 is 78 and out of class is 481, hence the sensitivity of this rule is

$$\frac{67}{78}$$

```
0.8589743589
```

and the specificity is

$$1 - \frac{26}{481}$$

```
0.9459459459
```

Which for this data is a good rule!

6. References

1. LISBOA, P.J.G. 'A review of evidence of health benefit from artificial neural networks in medical intervention', Neural Networks, Invited Paper, 15, 1, 9-37,2002.

2. LISBOA, P.J.G., ETCHELLS, T.A AND POUNTNEY, D.C. 'Minimal MLPs do not model the XOR logic' Neurocomputing, Rapid Communication, 48, 1-4,1033-1037, 2000

3. TSUKIMOTO, H., "Extracting rules from trained neural networks", IEEE Transactions on Neural Networks, vol. 11, no. 2, pp. 377-389, March 2000.

4. POP, E., HAYWARD, R. AND DIEDERICH, J. 'RULENEG: Extracting rules from a trained ANN by stepwise negation' Technical Report, QUT NRC (December, 1994).

5. CRAVEN, M.W. AND SHAVLIK, J.W. 'Using sampling and queries to extract rules from trained neural networks' Machine Learning: Proc. 11th International Conference, pg. 37-45, 1994.

6. VALIANT, L.G. 'A theory of the learnable' Communications of the ACM, 27: 1134-1142, 1984.

7. ftp://ftp.ics.uci.edu/pub/machine-learning-databases/