Data Analytics and Boolean Algebras

Hans van Thiel

November 28, 2012

KvK Amsterdam 34350608

Passeerdersstraat 76
1016 XZ Amsterdam
The Netherlands

T: + 31 20 6247137
E: hthiel@muitovar.com
W: http://muitovar.com

Abstract

Categorical data tables can be modeled as finite, atomic, Boolean algebras. These models can be used to discover predictive rules in the data. The rules can be expressed by a Boolean algebraic formula in a normalized disjunctive form.

The attributes and their values define a maximal algebra, and there exists a Boolean homomorphism from this to the algebra of actual data. The homomorphism kernel can be determined and transformed into a normalized disjunctive form. The predictive factors of all binary attributes can be derived immediately from this kernel.

For attributes with more than two values a Boolean homomorphism can be constructed from the maximal algebra to the part of the table which contains the complement of the value to be determined. The predictive factors are in the subset, of the kernel, which matches the atoms which do contain the attribute-value.

The Boolean meets which predict an attribute-value are partially ordered, and this order can be determined by induction.

Keywords: Data Analytics, Categorical Data, Boolean Algebras, Induction

1 Introduction

This paper is about finding predictive factors or rules in categorical data.

As such it fits into the general context of data mining, knowledge discovery, data analytics and machine learning, but also multi-variate data analysis, empirical induction and maybe database theory. The field is very broad and diverse, and no directly related work has been found. The methods and theory discussed in this paper have been developed independently.

This method of finding rules is limited to categorical data, which are structured or formatted as a table of attribute-values, like the table in section 5. Each attribute may have two or more different values.

The goal is to be able to determine, for any attribute-value, the other attribute-values which predict the selected value. The format of predictive factors or rules is a disjunction (or) of conjunctions (and). For example, ((A) or (B and C) or (B and D)) implies E. See 5.

The rules are not statistical or probabilistic by nature, but totally deterministic. They are found by modeling the data table as a Boolean algebra, defining a *maximal* or *theoretical* Boolean algebra which is determined solely by the attributes and their values, and a Boolean homomorphism from this to the actual table.

The homomorphism kernel, calculated by applying De Morgan's law to the data table (algebra), is in conjunctive form, a meet of joins of attribute-values.

This conjunctive form can be rewritten as a disjunctive form, a join of meets. In the process of rewriting, the expression is normalized by using the fact that any values of the same attribute are disjunct.

This simplification through normalization of a Boolean expression is the basis of the data analytics.

Rules for binary attributes can be found from the normalized kernel through elementary Boolean algebra, almost by direct querying.

Rules for attributes with more than two values cannot be found in this way. It is, however, possible to define a homomorphism on that part of the table which does not contain the value. The kernel of this homomorphism can then be compared to the part of the table which does contain the value, and those meets (and) which match are those which make up the join (or).

The structure of the paper is as follows. Section 2 is about the application of the theory of Boolean Algebras to categorical data tables.

Subsection 2.1 discusses how a *set of things* may be fit into a set of *attributes and discrete values* and how this may be modeled as a finite, atomic, Boolean algebra. Subsection 2.2 details how a data table defines such an algebra, with the table rows as atoms. In subsection 2.3 a *maximal* Boolean algebra is defined from the attributes and their values only. Subsection 2.4 defines a function from the maximal algebra (table) to the actual algebra (table) and shows that it is a Boolean homomorphism.

Section 5 discusses the application to data analytics. The kernel of the homomorphism is just the Boolean complement (in the maximal algebra) of the *data algebra*. From this, after normalizing, the predictive factors for binary attributes follow almost directly (subsection 2.1) and those for values of other attributes by calculating (and normalizing) the kernel of another homomorphism. This kernel is the complement in the maximal algebra, of the complement of the attribute-value in the data algebra (subsection 3.2). Finally, in subsection 3.3

the (possible) partial order of the meets in the join is discussed, together with a simple, brute force, algorithm to find it. Transforming a meet of joins into a join of meets and normalizing it involves comparing all attribute-values. A brute force computation would be of exponential order and subsection 4.1 discusses a heuristic approach and its implementation in Haskell.

Section 5 shows the results of the application on a very well known, small, data table. Section 6 sums up a few conclusions.

2 Boolean Algebras

2.1 Finite Boolean Algebra

Let X be a finite or infinite set and $P(X)$ the set of subsets of X. Then $P(X)$ is a Boolean Algebra, defined by join ($\vee$), meet ($\wedge$) and complement ($'$). Let V be a finite set of values of an attribute a where

1. each element of X belongs to some $v \in V_a$
2. no element of X belongs to more than one $v \in V_a$

In other words, attribute a defines an equivalence relation on the set X, and the values define a partition. The subsets of X, denoted by the values, are elements of a subalgebra of $P(X)$. A Boolean join in this algebra is the union of the sets denoted by the values, a complement is the union of the disjunct subsets and the meet of the intersection, which is the empty set, is $\mathbf{0}$ in the Boolean algebra.. Let b be another attribute, which also defines an equivalence relation on X. Now we can define Boolean meet, join and complement through the sets denoted by the values of both attributes. The meet of a_i and b_j, for example, would be the intersection of the corresponding sets in X, which is the Boolean meet in $P(X)$.

A set of attribute values, defined by the sets $V_a, U_b, W_c \ldots$ of each attribute $a, b, c \ldots$ in a finite set A of attributes define a finite subset of elements of the Boolean Algebra $P(X)$.

Such a subset E corresponds to a subalgebra of $P(X)$. This (finite) boolean algebra is atomic, and each of its elements can be written as a join of atoms.

Let $i \in P(X)$ and $j \in \{0, 1\}$. Let $p(i, j) = i$ if $j = 1$ and $p(i, j) = i'$ if $j = 0$. An atom is

$$p_f = \bigwedge_{i \in E} p(i, f(i))$$

where f is a function from E to $\{0, 1\}$ and $p_f \neq 0$. Each atom is is a meet of either an element i or its complement. If the number of atoms of the subalgebra generated by E is n, the number of elements is 2^n.

See [1] for an extensive treatment.

2.2 Data Algebra

Let T be a table with n rows and m columns, where each column represents an attribute, and each cell is a value of the attribute in its column.

If each attribute-value denotes a subset of a set X, as discussed in 2.1, each attribute defines a partition of this set. The Boolean join of all values of an attribute is $\mathbf{1}$ in $P(X)$. A meet of values of the same attribute is $\mathbf{0}$.

A row in T denotes the intersection of the subsets that are denoted by the attribute-values in the row. This defines their Boolean meet. Each row has exactly one value for each attribute. Therefore, an intersection with any set denoted by any value not in the row is the empty set (**0** in the corresponding Boolean algebra). So the rows are the smallest elements in the subalgebra of $P(X)$ defined by the table. Since n is finite, the Boolean algebra defined by T is finite, and the rows denote the atoms.

2.3 Maximal Algebra

Usually the number of possible table rows, using only the attributes and values of the table, will be larger than the actual number of rows. The number is bounded by

$$n = \prod_{a \in A} k_a$$

where k_a is the number of values of attribute a. For example, if attribute a has 2 values, attribute b has 10 and c has 3, the maximal number of rows is 60. This *maximal* table also defines a Boolean algebra.

In disjunctive normal form, as a join of meets, the maximal element **1** is written as the join of all possible rows, which are meets of attribute-values. These are the atoms.

In conjunctive normal form, as a meet of joins, the maximal element **1** is written as the meet of the joins of all values of each attribute. For the above example:

$$\mathbf{1} = (a_1 \vee a_2) \wedge (b_1 \vee b_2 \vee \ldots b_{10}) \wedge (c_1 \vee c_2 \vee c_3)$$

Each of the joins, in this notational form, is also **1**.

2.4 Boolean Homomorphism

The set of rows of a table T is a subset of the rows of its maximal table M. Each of these sets is the set of atoms of the corresponding Boolean algebras. Let f be the function from M to T such that, if x is an atom,

$$f(x) = \begin{cases} x \text{ if } x \in T \\ \mathbf{0} \text{ if } x \notin T \end{cases}$$

If x is not an atom, then it is a join of atoms u_i and:

$$f(x) = f(\bigvee_i u_i) = \bigvee_i f(u_i)$$

Because any $f(u_i)$ evaluates to either itself or **0**, $f(x) = x$ for any x. On the left x stands for an element in M, on the right it is in T.

To show that f is a Boolean homomorphism from M to T it is sufficient (see [1]) to show that, for each x and y,

$$f(x \vee y) = f(x) \vee f(y) \quad (1)$$

$$f(x') = (f(x))' \quad (2)$$

The first equation is satisfied by definition. To show the second part, divide an element x into atoms that are elements of T and those that are not.

$$x = (\bigvee_i u_i) \vee (\bigvee_j v_j)$$

The complement x' may be split the same way

$$x' = (\bigvee_k w_k) \vee (\bigvee_l z_l)$$

Both Boolean algebras are finite, so $i + j + k + l$ is the number of atoms of M and $i + k$ is the number of atoms of T. The joins of u and v are disjunct, because one is part of x, and the other of the complement of x (in M).

$$f(x) = \bigvee_i f(u_i) \vee \mathbf{0} = \bigvee_i f(u_i) = \bigvee_i u_i$$

$$f(x') = \bigvee_k f(w_k) \vee \mathbf{0} = \bigvee_k f(w_k) = \bigvee_k w_k$$

Because in the Boolean algebra T,

$$(\bigvee_i u_i) \vee (\bigvee_k w_k) = \mathbf{1}$$

equation 2 is satisfied, and the function f is a homomorphism.

3 Data Analytics

The kernel K of f is the join of those atoms (rows) of M which do not occur in T. This is the complement, in M, of those atoms (rows) which do occur in T. It can be calculated by applying De Morgan's laws.

The resulting formula is in conjunctive form, a meet of joins. This can be transformed into disjunctive normal form by comparing all attribute-values and simplifying through

$$a_i \wedge a_j = \begin{cases} a_i \text{ if } j = i \\ \mathbf{0} \text{ if } j \neq i \end{cases}$$

A join can only be $\mathbf{0}$ if each element in the join is $\mathbf{0}$. De value of $f(K)$ is $\mathbf{0}$, so $f(x \wedge y \wedge \ldots) = \mathbf{0}$ for each meet in the normalized kernel K.

For any x and y in a Boolean algebra ([1])

$$x \leq y \text{ iff } x \wedge y' = \mathbf{0}$$

3.1 Binary Attributes

If an attribute has two values, the complement of each is the other attribute-value. So, for any a_1, it is a simple matter to lookup the meets with a_2 in the normalized kernel. For example, if

$$\begin{aligned} &(x \wedge a_2) \vee \\ &(y \wedge a_2) \vee \\ &(z \wedge a_2) = \mathbf{0} \end{aligned}$$

then

$$x \leq a_1 \text{ and}$$
$$y \leq a_1 \text{ and}$$
$$z \leq a_1$$

If two Boolean elements are both smaller than a third, their join is also smaller ([1]), so the result is:

$$(x \vee y \vee z) \leq a_1$$

3.2 More than Two Values

If an attribute has more than two values, the elements smaller than (or equal to) its complement can also be found by querying the kernel K. For example, if a has three values, then, if $x \wedge a_2 = \mathbf{0}$ then $x \leq (a_1 \vee a_3)$.

Because $a_1 = (a_1 \vee a_3) \wedge (a_1 \vee a_2)$ the meets covered by a_1 could be found by calculating the meet of the joins covered by the complements. A somewhat easier way, however, is to use a different homomorphism.

If a has n values, then, to calculate the join of meets below a_i, take those atoms (rows) of T that do not contain a_i. These atoms (rows) define a Boolean algebra C, and a homomorphism may be defined from M to C, as discussed in 2.4. The kernel L is the complement in M of the atoms (rows) of C.

But this consists not only of the atoms not in T, but also of the atoms which are in T, but not in C. These, however, are the atoms which contain a_i.

The kernel L is easily calculated, as a meet of joins, through De Morgan's law and may then be rewritten as a normalized join of meets. For each of these meets it can then be checked whether they cover an atom with a_i or not. A meet covers an atom if the set of its values is a subset of the set of values which make up the atom.

The number of values of any attribute is, of course, limited by the number of atoms. A special case is a table where each row has a unique identifier. The rows could list distinguishing properties of some object or concept, the name of which is in a special column. Or the identifier could be the name of an individual, with the row itself a list of associated data.

The algorithm also works for binary attributes, but a different homomorphism kernel needs to be recalculated into disjunctive normal form for each separate attribute-value. This takes computing resources. Querying the same normalized kernel, as described in 3.1, is obviously preferable.

3.3 Partial Order

In the join of meets in the formula

$$(x \vee y \vee z \vee \ldots) \leq a_i$$

each meet $x, y \ldots$ has at most one value of any attribute. It is an element of the Boolean algebra T, so it is a join of atoms. Any atom is a meet of attribute-values, with exactly one value for each attribute.

For any u, v in a Boolean algebra, $u \wedge v \leq v$. So, if $u \wedge v$ is a meet of attribute-values, it is smaller than (or equal to) a meet v of attribute-values from a subset of the attribute-values which make up $u \wedge v$.

Conversely, if v consists of some set of attribute-values, and r is a meet of a superset, then as $r = v \wedge u$ for some u.

Let B be the set of attribute-values in meet x and R_1 the set of attribute-values in atom r_1. Then

$$x \geq r_1 \text{ iff } B \subseteq R_1$$

So, the atoms below a meet of attribute-values x are exactly those whose attribute-values are supersets of the attribute-values of x.

All the meets in

$$(x \vee y \vee z \vee \ldots) \leq a_i$$

are joins of atoms (of T) below a_i. Suppose,

$$x = r_1 \vee r_2 \vee r_3 \quad \text{and}$$
$$y = r_1 \vee r_2$$

In this case $y < x$. If x and y cover the same atoms, then $x = y$.

In general, if the set of atoms which make up the join equal to x is X, and the set which makes up the join equal to y is Y, then

$$x \leq y \text{ iff } X \subseteq Y$$

The meets of attribute-values below some attribute-value are partially ordered. Particularly interesting are the meets at the top, which have no larger elements besides a_i and those at the bottom, which have no smaller elements besides $\mathbf{0}$.

The algorithm is a kind of *empirical induction.* In the classic example, if the set of things which are ravens is a subset of the set of things which are black, then ravens are black.

4 Implementation

4.1 Transforming a Meet of Joins into a Join of Meets

It makes sense to code an attribute-value as a tuple of an index into an array of attributes, and an index into an array or list of its values. Comparisons of attribute-values can then be fast. In particular, a meet of two values of the same attribute can be seen to be $\mathbf{0}$ if the firsts of their index tuples are equal and the seconds unequal.

To transform a meet of joins into a join of meets, each attribute-value in each join must be compared with each attribute-value in each other join.

So, if the number of joins is m and the number of attribute-values in each join is n, the number of comparisons is n^m. However, if v is an attribute-value in a meet of joins then that meet may be partitioned into joins with v and without v. Let rs be the meet of joins which do not contain v. Then,

$$(v \vee x) \wedge (v \vee y) \wedge \cdots \wedge rs = (v \vee (x \wedge y \wedge \ldots)) \wedge rs = (v \wedge rs) \vee (x \wedge y \wedge rs)$$

But $x \wedge y \wedge rs$ is just the original meet of joins with attribute-value v removed.

The process of removing an attribute-value can be repeated until one of the joins $x, y \ldots$ becomes empty. Then that join is $\mathbf{0}$ and any meet with that row is $\mathbf{0}$. The meet of joins reduces to a join,

$$(v_1 \wedge rs_1) \vee (v_2 \wedge rs_2) \vee \cdots \vee \mathbf{0}$$

Now the process of partitioning in joins with some attribute-value and joins without that value can be applied to each rest rs_i as well.

It seems reasonable that the number of comparisons will be smaller if the number of joins in each rs is smaller. This will be the case when the frequency of occurrence of v will be greater, since an attribute-value occurs at most once in any row.

The partitioning of a meet of joins rs terminates when the value v occurs in all the joins and, therefore, the set of joins without v is empty.

However, it is also possible that a meet of joins rs is not empty, but contains a join (row) consisting only of values of the same attribute as v.

$$v_i \wedge (v_j \vee v_k \ldots) = \mathbf{0}$$

for different $i, j, k \ldots$. In that case rest rs, which is a meet, will also be $\mathbf{0}$, and so will $v_i \wedge rs$.

The above suggests an intermediate tree data structure, with an attribute-value as node, and a list of trees as children. The leaves of the tree are the empty lists or some mark to denote failure of the meet.

Extracting all the branches of the tree, leaving out the ones marked as $\mathbf{0}$, results in a join of meets.

Next, each meet will have to be compared with all other meets to be able to apply the cancellation law

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

A list (forest) of trees as sketched above can be built quite naturally, using mutual recursion, in the functional programming language Haskell [2]. The success or failure of building a meet (branch) can be modeled with the standard *Maybe* data type. Haskell also has powerful constructs for list processing and filtering, and the transformation of a meet of joins to a join of meets, as sketched above, could be implemented in approximately 100 lines of Haskell code.

4.2 Induction of Partial Order

The meets of attribute-values below some attribute-value all cover those rows of the data table which contain the predicted attribute-value. So, for each meet, determine the rows (atoms) whose set of attribute-values is a superset of the set of attribute-values of the meet.

Two different meets may cover the same atoms, in which case they are equal (both $x < y$ and $y < x$). The first step is to determine which meets are equal. These equivalence classes are the nodes in the poset graph.

One node is below another, iff the set of atoms it covers is a subset of the set of atoms covered by the second. For this to be the case, the second set must be larger than the first (equals are in the same node). So, if the set of nodes are ordered according to size (list length) of the atoms they cover, a node only has to be checked against a later (larger) node.

Weather	Temperature	Humidity	Windy	Fishing
sunny	hot	high	no	bad
sunny	hot	high	yes	bad
cloudy	hot	high	no	good
rain	mild	high	no	good
rain	cool	normal	no	good
rain	cool	normal	yes	bad
cloudy	cool	normal	yes	good
sunny	mild	high	no	bad
sunny	cool	normal	no	good
rain	mild	normal	no	good
sunny	mild	normal	yes	good
cloudy	mild	high	yes	good
cloudy	hot	normal	no	good
rain	mild	high	yes	bad

Table 1: The well known data table from Quinlan [3]

From this list of larger nodes, only the adjacent ones are second nodes in the edge. These can be found by removing all nodes whose atoms are supersets of any other node. This process of determining edges for a node is then repeated for each node in the list of nodes. For this to work, the predicted attribute-value, which covers all meets, must be in the list. Because the list is sorted by the number of covered atoms, it will be at the end.

Using the Haskell functional graph library, it is straightforward to construct a directed graph from nodes and edges, and display it in a basic GraphViz format. Because a node contains all equals, the resulting graph of the partial order is acyclic. Furthermore, because it contains the predicted attibute-value, it is a tree.

5 Data Analysis: an Example

The above table is taken from Quinlan [3]. The number of rows, and therefore the number of atoms in its Boolean algebra is 14. There are 5 attributes, 3 binary and 2 with 3 values, so the number of atoms in the maximal algebra is $2^3 \times 3^2 = 72$.

The table was read in .csv format and automatically coded as listed below.

```
--- tableToArray ---
array (0,4) [
(0,("Weather",["sunny","cloudy","rain"])),
(1,("Temperature",["hot","mild","cool"])),
(2,("Humidity",["high","normal"])),
(3,("Windy",["no","yes"])),
(4,("Fishing",["bad","good"]))]
```

For brevity the coded symbols will be used instead of the names. The following is the Boolean complement of the table in conjunctive form. So, each row is a join of the attrinute-values, and the table is a meet of these rows.

```
--- compTable ---
[[(3,1),(0,1),(4,1),(0,2),(1,1),(1,2),(2,1)],
[(3,0),(0,1),(4,1),(0,2),(1,1),(1,2),(2,1)],
[(0,0),(4,0),(3,1),(0,2),(1,1),(1,2),(2,1)],
[(0,0),(1,0),(4,0),(3,1),(0,1),(1,2),(2,1)],
[(0,0),(1,0),(2,0),(4,0),(3,1),(0,1),(1,1)],
[(0,0),(1,0),(2,0),(3,0),(0,1),(4,1),(1,1)],
[(0,0),(1,0),(2,0),(3,0),(4,0),(0,2),(1,1)],
[(1,0),(3,1),(0,1),(4,1),(0,2),(1,2),(2,1)],
[(1,0),(2,0),(4,0),(3,1),(0,1),(0,2),(1,1)],
[(0,0),(1,0),(2,0),(4,0),(3,1),(0,1),(1,2)],
[(1,0),(2,0),(3,0),(4,0),(0,1),(0,2),(1,2)],
[(0,0),(1,0),(3,0),(4,0),(0,2),(1,2),(2,1)],
[(0,0),(2,0),(4,0),(3,1),(0,2),(1,1),(1,2)],
[(0,0),(1,0),(3,0),(0,1),(4,1),(1,2),(2,1)]]
```

This is transformed into a normalized join of meets, using an implementation of the algorithm discussed in 4.1. Each line is a meet of attribute-values. The 9 lines with *Fishing:bad* $(4, 0)$ and the 5 with *Fishing:good* $(4, 1)$ have been marked manually for the reader's convenience.

```
--- meetsToJoins ---
[[(1,2),(2,0)],
[(0,1),(4,0)],                    x
[(1,2),(0,1),(3,0)],
[(1,2),(4,0),(3,0)],              x
[(1,2),(4,0),(0,0)],              x
[(1,2),(0,0),(3,1)],
[(0,2),(3,1),(4,1)],                  x
[(0,1),(1,1),(2,1)],
[(0,1),(1,1),(3,0)],
[(0,1),(3,1),(1,0)],
[(1,0),(3,1),(2,1)],
[(1,0),(0,2)],
[(1,0),(3,1),(4,1)],                  x
[(1,0),(2,1),(4,0)],              x
[(1,0),(2,1),(0,0)],
[(1,0),(0,0),(4,1)],                  x
[(0,0),(4,1),(2,0)],                  x
[(4,0),(2,1),(1,1)],              x
[(0,0),(1,1),(4,0),(3,1)],        x
[(0,0),(1,1),(3,0),(2,1)],
[(0,0),(1,1),(3,0),(4,1)],            x
[(0,0),(1,1),(2,0),(3,1)],
[(0,0),(2,1),(4,0)],              x
[(4,0),(2,1),(3,0)],              x
[(0,2),(3,1),(2,1),(1,1)],
[(0,2),(4,0),(3,0)]]              x
```

Since *Fishing* is a binary attribute, the two values $4, 0)$ and $(4, 1)$ are each other's complements, and the dependencies can just be read out from the listing.

For example,

$$(0,1) \wedge (4,0) = \mathbf{0}$$
$$(0,1) \wedge (4,1)' = \mathbf{0}$$
$$(0,1) \leq (4,1)$$

which translates into *Weather:cloudy* implies *Fishing:good*. The other rules for *Fishing* can be found similarly.

Of course, the following reasoning is also possible,

$$(0,1) \wedge (4,0) = \mathbf{0}$$
$$(0,1)' \wedge (4,0) = \mathbf{0}$$
$$(4,0) \leq ((0,2) \vee (0,3))$$

which means *Fishing:bad implies Weather:sunny or Weather:rain*. In other words, bad fishing implies the weather is not cloudy, the logical contraposition of the statement *cloudy = good*. Dependencies of the other binary attributes can be found similarly. For example,

$$(1,0) \wedge (3,1) \wedge (2,1) = \mathbf{0}$$

So, *Temperature:hot* and *Windy:yes* implies *Humidity:normal*.

The Boolean meets which determine the values of attributes with more than two values cannot be found this way, only complements. But these can be found with a Boolean homomorphism of M to the algebra C which does not contain that value, as discussed in 3.2.

For $(0,1)$ (*Weather:cloudy*) C is,

```
--- table wihout (0,1) ---
[[(0,0),(1,0),(2,0),(3,0),(4,0)],
[(0,0),(1,0),(2,0),(3,1),(4,0)],
[(0,2),(1,1),(2,0),(3,0),(4,1)],
[(0,2),(1,2),(2,1),(3,0),(4,1)],
[(0,2),(1,2),(2,1),(3,1),(4,0)],
[(0,0),(1,1),(2,0),(3,0),(4,0)],
[(0,0),(1,2),(2,1),(3,0),(4,1)],
[(0,2),(1,1),(2,1),(3,0),(4,1)],
[(0,0),(1,1),(2,1),(3,1),(4,1)],
[(0,2),(1,1),(2,0),(3,1),(4,0)]]
```

As before, each row is a join of attribute-values, and the table is the meet of the rows. The kernel of the homomorphism of M to C, rewritten and normalized as join of meets, is:

```
--- meets to joins ---
[[(0,1)],                    x
[(1,0),(4,1)],               x
[(1,0),(2,1)],               x
[(1,0),(0,2)],
[(1,2),(2,0)],
[(1,2),(4,0),(3,0)],
```

```
[(1,2),(4,0),(0,0)],
[(1,2),(3,1),(4,1)],          x
[(1,2),(3,1),(0,0)],
[(3,1),(4,1),(2,0)],          x
[(3,1),(4,1),(0,2)],
[(2,1),(4,0),(1,1)],
[(2,1),(4,0),(0,0)],
[(3,1),(2,1),(0,2),(1,1)],
[(3,1),(0,0),(4,0),(1,1)],
[(3,1),(0,0),(2,0),(1,1)],
[(4,1),(2,0),(0,0)],
[(4,1),(0,0),(3,0),(1,1)],
[(2,1),(4,0),(3,0)],
[(2,1),(0,0),(3,0),(1,1)],
[(4,0),(3,0),(0,2)]]
```

The meets in this join which cover one of the four atoms of T which do contain $(0, 1)$ are the ones marked above. The set of attribute-values of each meet is a subset of the set of attribute-values of one or more of those atoms..

```
--- reduce (0,1) ---
[[(0,1)],
[(1,0),(4,1)],
[(1,0),(2,1)],
[(1,2),(3,1),(4,1)],
[(3,1),(4,1),(2,0)]]
```

It can easily be checked in the original data table that, for example, *Temperature:hot* together with *Humidity:normal* indeed imply *Weather:cloudy*.

Of course the general algorithm for attribute-values can also be applied to binary attributes. For *Fishing:bad*, for example, the result is,

```
--- reduce (4,0) ---
[[(4,0)],
[(0,0),(2,0)],
[(0,0),(1,0)],
[(0,0),(3,0),(1,1)],
[(1,0),(3,1)],
[(3,1),(0,2)]]
```

These are the same 5 meets which are found by querying the kernel of the homomorphism to the algebra T, as shown above. Their partial order can be found by induction, as discussed in 3.3 and 4.2. It is shown in the figure.

There are no equals, each node consists of just one meet. There are two least general meets, *Temperature:hot and Windy:yes* and *Weather:sunny and Windy:no and Temperature:mild*. There are also two most general meets, *Windy:yes and Weather:rain* and *Weather:sunny and Humidity:high*. All imply *Fishing:bad* which is at the root.

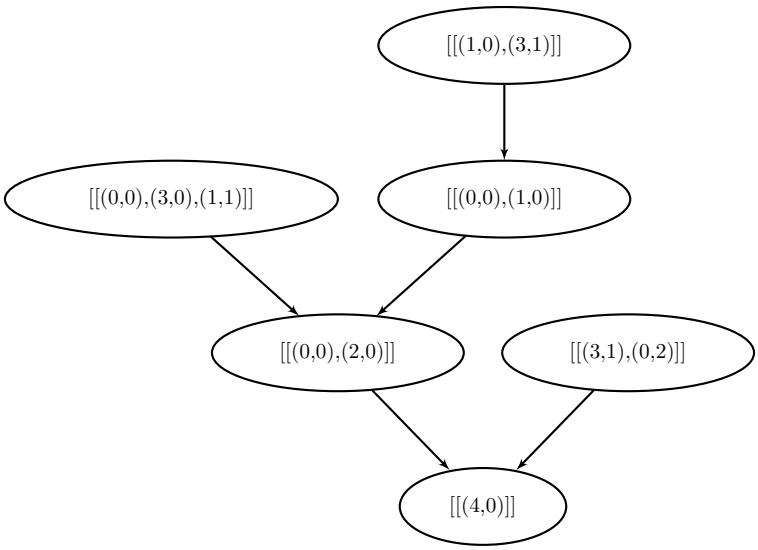

Figure 1: The partial order of the meets predicting *Fishing:bad*

6 Conclusions

Categorical data tables can be modeled as Boolean algebras, and these models can be used to derive predictive rules. Having a clear and distinct mathematical model for the data which are to be analyzed is, in itself, very useful.

The results are not probabilistic or statistical, but deterministic. The format of the predictive rules, a join (or) of meets (and) of attribute-values is natural.

The theory of Boolean algebras provides a mathematical toolbox for further research and development. Subjects of interest are, among others, analysis of data tables where each row has a unique identifier (concept analysis), expansion of an algebra by adding new attributes, the effects of adding or deleting rows to an existing table, and the analysis of the complement (the unobserved data).

References

[1] Steven Givant and Paul Halmos, *Introduction to Boolean Algebras*. Springer, 2009.

[2] Simon Thompson, *The Craft of Functional Programming*. Second Edition, Pearson, 1999.

[3] J.R. Quinlan, *Induction of Decision Trees*. Machine Learning 2, 81 – 106, 1986.