Expert Data Mining: k-value

Introduction

For this next iteration of the Expert Data Mining software, the program now works with k-value instead of just Boolean. *K*-value refers to how many values that a particular attribute can have. The below k-values follow a convention that they are always 0 through $k - 1$.. For example, a k-value of 2 is equivalent to Boolean values 0 and 1, but a k-value of 3 is equivalent to values of 0 through 2. Let's say that we have a dataset of just 2 dimensions, and the *k*-value for each attribute is 3. The following datapoints will be in the dataset:

(0, 0)

(1, 0)

(0, 1)

(1, 1)

(2, 0)

(0, 2)

(2, 1)

(1, 2)

(2, 2)

In this example, $\mathbf{k} = (3, 3, k_{n+1})$. The dimension of the *k*-value vector is one more than the dimension of the n-D points. That is because the value of k_{n+1} refers to the *k*-value of the function itself. A k_{n+1} of three would correspond to three possible values that the n-D points could be equal to. Those values would be between 0 and 2 for the current implementation. The *k*-value vector is generally defined as:

$\mathbf{k} = (k_1, k_2, \ldots, k_n, k_{n+1})$

Furthermore, the sections below will describe how *k*-value will be different for the corresponding options. First, *k*-value attributes will be described. Then, k_{n+1} will be explored in greater depth.

K-value Attributes

Majority Flag

One of the goals of the majority flag is to minimize the number of questions that needs to be answered by the domain expert to restore a *k*-valued function when n-D points that have multiple values (*k*-value). The *k*-valued approach is also the approach which we use for binary functions. We ask a user if they believe that for the function to have a value equal to one or greater, then at least half of the values of the n-D points need to be equal to 1 or greater. We require that half of the Hamming norm of the n-D points will be needed to a function value of 1 or greater.

For example, let's take a non-binary n-D point, **a**, where

k = (3, 5, 2, 2)

a = (2, 4, 1)

a is the largest n-D point that is possible for the given k-values. Obviously, the smallest n-D point will be (0, 0). Its **generalized Hamming norm,** G(x), is the sum of the values; e.g., G(**a**) = 7. However, there is an issue with simply using the generalized Hamming norm for majority flag: there can be "lopsided" n-D points. Since G(**a**) = 5, we look for n-D points where there generalized Hamming norm is 3. For Boolean, this is not an issue because each point in the datapoint can only be one of two values, which means for a majority of those points to be a value of 1, then half or at least half of the points will be a 1. For the given k-values above, then the generalized Hamming norm, G(x), needs to be 4 for the n-D point to be considered a majority datapoint. However, the n-D point (0, 3, 0) would fall into this category. We consider this datapoint to be "lopsided" because although the generalized Hamming norm is the correct value, less than half of the points are equal to 1 or greater. The solution to this is using "middle points." Middle points are simply *k*-valued n-D points where at least half of the points are a value of 1 or greater, similar to majority flag for Boolean. However, the generalized Hamming norm still needs to be equal to half that of the maximum generalized Hamming norm possible for the given k-values

For our above *k*-values, then the middle points are:

(1, 2, 1)

(2, 1, 1)

(1, 3, 0)

(2, 2, 0)

If we only use the generalized Hamming norm, then a possible majority flagged datapoint would be (0, 4, 0). The implementation of middle points is another pilot question we ask the user in the case that they chose to use the majority flag option.

Furthermore, just like the Boolean version, if the user decides to use the majority flag, then all majority vectors are asked first, and once those are done, a predefined default sequence of questions is asked. If the user does not know roughly how many majority vectors will have $f(\mathbf{x}) > 0$ (successful at some level), then all majority vectors are asked first.

If the user specifies a number of these majority vectors, then we ask at random, and if for half of those majority vectors $f(\mathbf{x}) > 0$, a predefined default sequence of questions is asked. The idea is that it is likely that for a given dimension, it will take half or slightly over half of the attributes of the n-D point **x** to get $f(\mathbf{x}) > 0$, but that is not guaranteed. The majority flag can be paired with different methods of changing the order of question, including moving from chain to chain without finishing full exploration of a current chain called chain jumping. In this case, after the majority questions are asked, the Hansel Chains which have majority vectors with $f(\mathbf{x}) > 0$ are asked before returning to the predefined default sequence of Hansel Chains.

Furthermore, when we restore the function for $k_{n+1} > 2$, we can restore the function for all the different values of *k* from 0 to k_{n+1}. This is shown in the example results that are attached.

True Attributes

Originally, a *true attribute* for Boolean function f was an attribute that needed to be true for the datapoint **x**, $f(\mathbf{x}) = 1$. That is still the case with k-value ($k > 2$) arguments of the functions, but now the values of those attributes need to be specified. For example, if the k-value of an attribute x_i is 4, then the user could specify that the value of x_1 needs to be at least 2, $x_1 \geq 2$ to get $f(\mathbf{x}) = 1$ if $f(\mathbf{x})$ has only two values.

f-changes

f-changes serves the same role as it was for Binary functions allowing a user to update the value of function f due to new knowledge available, like data errors discovered. However, if the *f*-change causes a violation of monotonicity and the user decides to add a k-valued ($k > 2$) attribute to fix the violation, then we need to ask a user to defined the k-value for the new attribute.

k_{n+1} (k-value function)

As stated before, k_{n+1} refers to the number of values of the function f. Similar to how the Hansel Chains are ordered, the values are generally ordered in a monotonically increasing fashion. This is because we need to be able to expand from one n-D point to another. If the function values are nominal, then we can't expand across function values. For example, if we have some generic datapoints, **a** and **b,** and **a** expands **b** in the positive direction, **b** > **a,** then the $f(\mathbf{b}) \geq f(\mathbf{a})$. If $f(\mathbf{a}) = 1$ and assuming that $k_{n+1} = 3$, then $f(\mathbf{b}) \geq 1$. If $f(\mathbf{a}) = 2$ and assuming that $k_{n+1} = 3$, then $f(\mathbf{b}) = 2$. Without being able to assume that a function value of 2 is monotonically greater than a value of 1, these types of expansions would not be possible.

Furthermore, expansions are slightly different for values of $k_{n+1} > 2$. For Boolean functions ($k_{n+1} = 2$), If a datapoint has a function value of 1, then can expand it to another datapoint to have a value of 1. The same is true for a function value of 0. Previously, we called these one-to-one expansions and zero-to-zero expansions. However, for of $k_{n+1} > 2$, we can expand in both directions (to greater and smaller points). For example, consider Boolean datapoints:

a = (1, 0, 1)

b = (1, 1, 1)

c = (1, 0, 0)

If $k_{n+1} = 3$ and $f(\mathbf{a}) = 1$, then we will expand to both **b** and **c.** Therefore, $f(\mathbf{c}) \leq f(\mathbf{a}) \leq f(\mathbf{b})$. The term "one-to-one" and "zero-to-zero" do not apply anymore. Instead, now we generalize the terminology: an "up expansion" is an expansion that occurs to a datapoint that is monotonically greater than the source of the expansion. A "down expansion" is an expansion that occurs to a datapoint that is monotonically lesser than the source of expansion. In this example, f(**b**) is the up expansion and f(**c**) is the down expansion. One caveat with up and down expansions is that we don't necessarily know the exact function value for f(**C**) and f(**b**), just that the function value for them are f(**c**) ≤ 1 and f(**b**) ≥ 1. We still need to confirm with the user if f(**c**) = 0 or 1 and if f(**b**) = 1 or 2. Therefore, we refer to the middle values for any value of k_{n+1} to be weak values. If a user assigns a function value to a datapoint, and that function value is the smallest or greatest possible value with respect to k_{n+1}, then one of expansions from that datapoint will be a "strong" value. Using the previous example, if f(**a**) = 2, then f(**b**) = 2 and f(**c**) ≤ 2. Strong answers using the lowest and highest possible function value will reduce the number of questions that need to be asked to the user.

Nested Functions

Let's say that we have a two-level hierarchy of functions, a parent and children. A child function is representative of some attributes of the parent function. For Boolean case, it's assumed that is some attribute, say x_1 is a Boolean attribute, then the child must be a Boolean function because it determines of x_1 is true or false. Therefore, if x_1 is k-value, let's say that $k = 3$, then the child function must have $k_{n+1} = 3$. The k-value of the parent must be equal to the k_{n+1} value of the child. This is automatically determined by the program. If a parent attribute is given a k-value of 3, then when the child function is created, the value of k_{n+1} is automatically assumed to be 3 as well.