Paper ID: p1133108

Circle: Design and Implementation of a Classifier Based on Circuit Minimization

ABSTRACT

We present Circle, a classification algorithm based on the priciples of boolean function minimization. This classification process uses a recursive method to generate a set of implicants (or rules). The novelty of this algorithm is in the fact that the rules generated contain information about not only presence of features, but also their absence in determining class values. Although function minimization is inherently exponential on the number of attributes, we introduce several optimization techniques to reduce the complexity to the extent that we are able to scale the algorithm to 1000 columns, the limit in most commercial database systems. Circle is levelwise algorithm that iteratively produces implicants. For portions of the training set that are misclassified, Circle recurs producing additional implicants that will be logically conjoined to the previous implicants. Several optimization techniques were applied to the base Circle algorithm, as well as numerous ways it can be configured for specifc data mining tasks. Circle is completely implemented in Java with a JDBC-compliant database backend. One of the primary applications of Circle is mining bioinformatics data, and particularly genomic data. We present results of our experiments running circle on several well-known data sets for machine learning, as well as special large genomic data sets.

1. INTRODUCTION

Both inexpensive storage and the ability to generate and collect information has outpaced any reasonable expectation to interpret this information without automatic or at least semi-automatic techniques. In the area of bioinformatics, *e.g.* genomics, the accumulation and kinds of information discovered from high-throughput sequencing of proteins[1] far outpaces even Moore's law. There are two kinds of signficant challenges facing bioinformatics: operations and computation. Operations defined loosely is the design and implementation of information systems that allow general search; provide grid services via the web for deployment of software, data sets; provide web portals for scientists focused on various aspects of bioinformatics to submit and post new findings to public repositories of bioinformatic information that are shared throughout bioinformatics communities; provide a suitable structured environment to do *in silico* science–computation.

The other challenge is computation, including the development of models, algorithms, and data mining[2]. One of the primary tasks of bioinformaticians is to make sense of the sequenced proteins, *i.e.* function. A protein's function, as has become to be widely believed, is determined by its three-dimensional structure that is, in turn, determined directly by the linear sequence of amino acids making up the protein. Crystallization is currently the only means of directly determining structure, but is labor intenstive and very difficult. High-throughput techniques have matured to the point that it is far easier to sequence many thousands of proteins rather than crystallizing a few. In treating proteins as collections of strings, bioinformaticians realized that by "aligning" strings, similar proteins would likely share similar 3D structure, and therfore, function. As an example we present an alignment of about the last 30 residues of human alpha globulin to leghemoglobin from yellow lupin [2]:

```
HBA_HUMAN  NAVAHV---D--DMPNALSALSDLHAHKL
           +A  ++          +L+ L+++H+ K
LGB2_LUPLU EAAIQLQVTGVVVTDATLKNLGSVHVSKG
```

The '-' is called a "gap", reflecting a biological event that either removed or inserted a residue (in this example, a character). The + reflects biological similarity–enough that the pair is considered a match. The matches represent motifs, substrings that are likely important and probably essential to the in function of the proteins. Proteins are seldom short–containing sometimes several thousand characters and virtually never of the same length. One fundamental task is discovering motifs [3, 13, 14, 16]. An excellent portal for testing algorithms for motif discovery is the PROSITE database [6]. These families (proteins that possess like functions) are human-curated and accepted as valid. The motifs in PROSITE are given as regular expressions without Kleene closure. A typical for family PS00434 is L-x(3)-[FY]- K-H-x-N-x-[STAN]- S-F-[LIVM]- R-Q-L-[NH]-x-Y-x- [FYW]-[RKH]-K-[LIVM]. The xs stand for "don't care" and [] means disjuntion. Challenges include predicting which family a protein belongs to and more interestingly, discovering the motif of family. Traditional data mining techniques are, at best, difficult to apply to problems like these in bioinformatics. Indeed, discerning what exactly are pertinent features is an area of research [4, 8]. What we observed is that

[1]A protein for purposes of this paper is a string over an alphabet $|\Sigma| = 20$, where the symbols denote amino acids, organic molecules that, when connected, form a chain that is biologically active, *e.g.*, as an enzyme.

[2]Data mining and general search from operations are significantly different. In general search, scientists might be interested in seeking authors, citations, protein sequences.

the amount of and interaction of nominal data has made statistical approaches challenging, though statistical classifiers have had the most success as demonstrated by their ubiquity in both the literature and industry (see [5]). Formally, the task is to establish a function f that when presented with a set of values correctly identifies the class it belongs to. In classification, the task is made somewhat easier (problematically, though not computationally), because in constructing f one has a collection of correctly classified values (features) *a priori*. "Supervised learning" is used to describe this situation of having this information. Formally, the problem is, given a set $\{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \ldots, (\mathbf{x}_n, y_n)\}$, where $\mathbf{x}_i$ represent values to be classified and y_i represent a class value discover a "good" method (usually a function, rule-set, or decision tree) denoted by f such that $f(\mathbf{x}_i) = y_i$. Deciding upon what exactly is good is decided by the context of the problem, though discovering f that is right most of the time is what is generally sought [12, 15, 9].

What was discovered independently were two different deterministic methods to that relied upon the principle that a boolean variable could be ignored if it did not contribute to the output value. In its simplest form, $xy + x\overline{y} = x(y + \overline{y}) = x1 = x$, where x and y are Boolean values. The graphical method is called Kaurnaugh or Veitch-Kaurnaugh maps (K-maps, KV-maps) [7]. This method creates a grid of cells that represent the sum-of-products (SOP), a canonical form where the Boolean input values are represented as full conjunctions. The method is to circle all the 1s, while drawing circles as large as possible avoiding cells that do not explicitly contain 1. As few circles as possible are used. In addition, circles can only contain cells that differ on one Boolean attribute as described above. Once the circling is complete, a disjunction of the Boolean variables whose two values are not adjacent can be read straight from the grid.

Because graphical methods become quite clumsy beyond a few variables (virtually impossible), and they are problematic to to create procedurally, the Quine-McCluskey method [11] is used to for Boolean functions of many variables. Essentially the method involves grouping the conjuncts as small material implications called "implicants." For example, the first row of Fig. 1 forms an implicant $1110 \rightarrow 1$ (an order is assumed on the variables). The implicants are grouped based upon the number of 1s, since this helps mitigate the amount work explained next. Then iterating over this group, a new implicant is added if there are two other implicants that differ by a single Boolean value. The new implicant is identical to the other two, except a "don't care" symbol * is placed at the point where they differ. When this iterative is complete, the next phase is to judiciously pick so-called "prime implicants"–implicants that cannot be combined with other implicants–so that the original conjuncts are covered.

From this, we were inspired to create a logic-theoretic classifer called *Circle* (named for the task). Circle works on any kind of relation, though we are focusing now on relations with predominantly nominal data. Circle works by creating a set of implicants (or rules) via minimization. To investigate Circle fully, we have created a full implementation, freely available, and are currently in the process of adding optimizations. We have begun testing Circle on both synthetic datasets and bioinformatic data that we are currently working with. Our contributions in this paper focus on two aspects

- implementation of such a classifier–including some existing optimizations and directions for improvement
- preliminary results on both synthetic data and bioinformatic data we are currently working with

B_1	B_2	B_3	B_4	f
1	1	1	0	1
1	1	0	0	1
1	0	1	1	1
1	0	1	0	1
1	0	0	0	1
0	0	0	1	1
0	0	0	0	1
0	1	0	0	1
0	1	0	1	1
0	1	1	0	1
0	1	1	1	1

Figure 1: A Boolean function (classifier) f.

The remaining paper is as follows. First, in Sec. 2 we present our classifier graphically, what we call classifier maps (C-maps) much in the spirit of K-maps. This provides an good, intuitive foundation for the procedural algorithm. Next, we provide some preliminary formal language definitions in Sec. 3. We next discuss briefly the rule sets that Circle produces with respect to monotonicity in Sec. 5, followed by a description of the implementation in Sec. 6 and implementation in Sec 7. We then discuss experiments we performed on several synthetic datasets as well as bioinformatics data in Sec. 7.1. We conclude with a summary and future work in Sec. summary.

2. CIRCLE VIA C-MAPS

We will first introduce the Circle classifier graphically. As mentioned in the introduction, having been inspired by work in Boolean circuit minimization, we thought it natural and appealing to present Circle similarly. We will be using the relation **History** in Fig. 3. These values represent patient histories at a fictitious hospital. We are interested in the `Prognosis` of the patient, given his or her `Treatment`, `Symptoms`, `Age`, and `Sex`. The `PID` is nothing more than an identifier to make discussion easier. We create a grid called a classifier-map or C-map. A C-map is shown in Fig. 2(a). The C-map is rectangular collection of square cells. Each cell is indexed by an expression that spans both the top and left side of the figure. The expression is based upon **one** of the instances–in this case, we are using tuple t_{13}. Notice the leftmost cell is indexed by `X` and `Cough` at the top, and `young` and `male` on the left. The expressions are both positive and negative.

In Fig. 2(b) we have shown the progressive correct circling, beginning with tuple t_{23} as a finely dotted line. A suitable next larger circling includes the $\emptyset$ to the right. Another suitable cell of size two could have been directly above. In either case, the largest circle contains the four cells, shown as a solid line. Observe the circle is constrained in all directions. We continue circling until no tuple is left uncovered. The result of our circling is shown in Fig. 2(c).

Once we have completed circling, the next step is to describe the circles via the essential indices. These are attribute values that index the circle whose positive and negative values are adjacent. In the case of the first circle described, observe that $\overline{\texttt{X}}$ and $\overline{\texttt{male}}$ are essential. Conjoining these two negative values gives the rule $\texttt{Treatment} \neq \texttt{X} \wedge \texttt{Sex} \neq \texttt{male} \Rightarrow \texttt{sick}$. The final set of rules, hence, become:

$$\begin{aligned}
\texttt{Treatment} = \texttt{X} &\Rightarrow \texttt{cured} \\
\texttt{Treatment} \neq \texttt{X} \wedge \texttt{Symptom} \neq \texttt{cough} &\Rightarrow \texttt{sick} \\
\texttt{Treatment} \neq \texttt{X} \wedge \texttt{Sex} \neq \texttt{male} &\Rightarrow \texttt{sick} \\
\texttt{Age} \neq \texttt{young} \wedge \texttt{Sex} = \texttt{male} &\Rightarrow \texttt{cured}
\end{aligned}$$

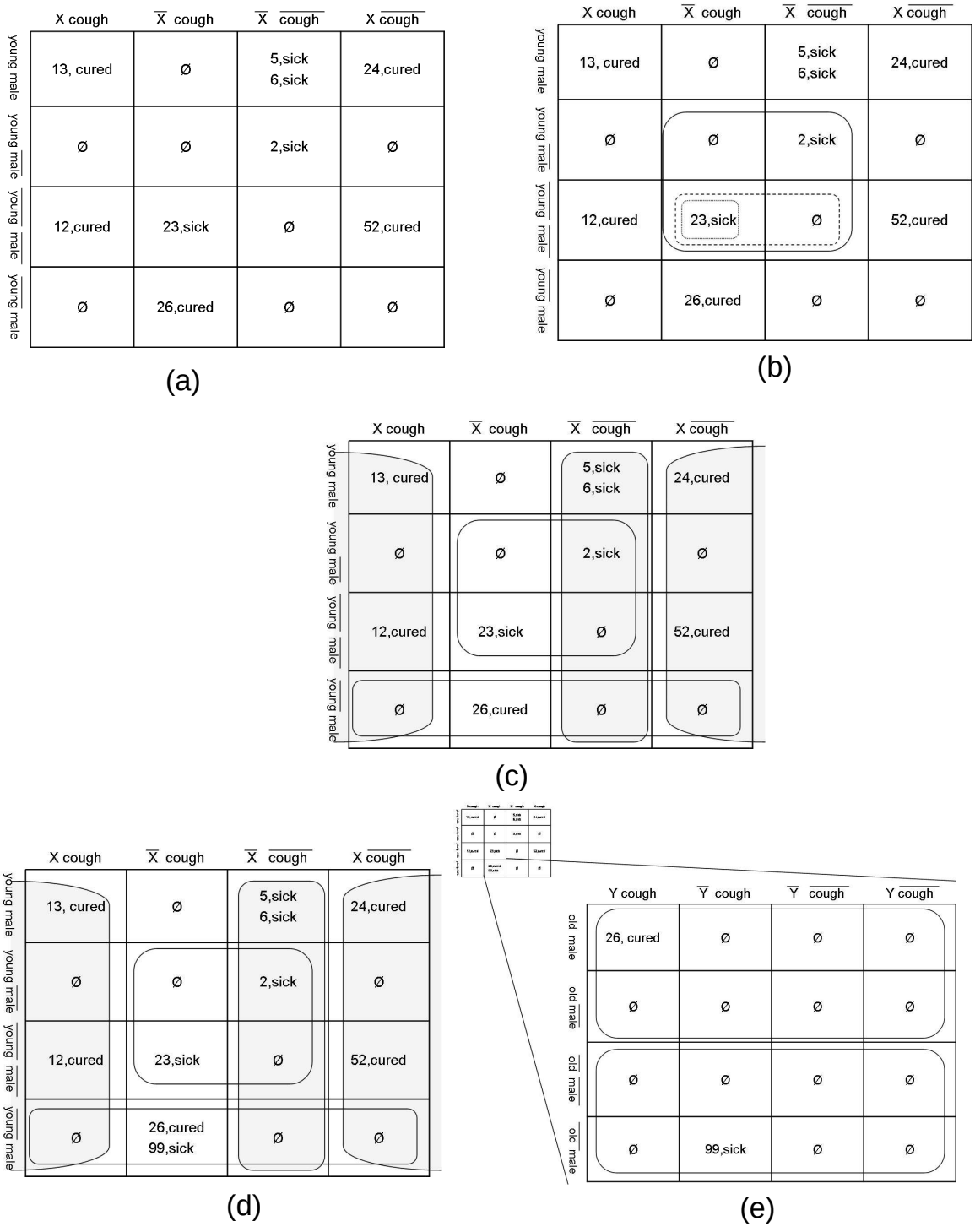

Figure 2: Illustration of the Circle algorithm

PID	Treatment	Symptoms	Age	Sex	Prognosis
13	X	cough	young	male	cured
24	X	fever	young	male	cured
12	X	cough	adult	female	cured
26	Y	cough	old	male	cured
23	Y	cough	adult	female	sick
52	X	sneeze	adult	male	cured
2	W	sneeze	young	female	sick
5	Z	sneeze	young	male	sick
6	Z	fever	young	male	sick

Figure 3: A relation History of patient data from a hospital.

2.1 Ambiguity and Recursion

Suppose there is an additional tuple $t_{99} = (\mathtt{Z}, \mathtt{c}ough, \mathtt{a}dult, \mathtt{m}ale, \mathtt{s}ick)$. Using the original C-map, we can placed it and the other nine tuples along with $\emptyset$ as shown in Fig. 2(d). We have additionally circled the cells. Observe that one cell shares two tuples whose classifier values differ, *i.e.*, t_{99} and t_{26}. The circle containing these two values is *ambiguous*, since two different class values are present. What this means is that the conjunction of attribute values cannot distinguish between the two classes. Notice the circle is as large as possible, being bounded by cells that contained either class value separately, but not both. A simple solution is to recursively create another C-map with only the ambiguous tuples from the circle. This is shown in Fig. 2(e). Notice this new C-map is indexed in this case by tuple t_{26}'s feature values. Two maximal circles are shown in the figure. From these two maps we can observe that the tuples are separated by the feature value `old`. Conjoining this information to the original circle gives the rules

$$\begin{aligned} \mathtt{Age} \neq \mathtt{young} \wedge \mathtt{Sex} = \mathtt{male} \wedge \mathtt{Age} \neq \mathtt{old} &\Rightarrow \mathtt{sick} \\ \mathtt{Age} = \mathtt{old} \wedge \mathtt{Sex} = \mathtt{male} &\Rightarrow \mathtt{cured} \end{aligned}$$

3. PRELIMINARIES

We assume a relation **r** over a set of attributes R. From R we distinguish a single attribute C as the classifier and the remaining attributes $\{A_1, A_2, \ldots, A_n\}$ as so-called features or feature attributes. Thus, $R = \{A_1, \ldots, A_n\} \cup \{C\}$. C contains the class values (classes), and the remaining attributes are the means of achieving the classification. For the initial portion of the paper, we will use the relation `History` in Fig.3. The feature attributes are $\{\mathtt{Treatment}, \mathtt{Symptoms}, \mathtt{Age}, \mathtt{Sex}\}$. The classifier is `Prognosis`. The task is to arrive at some function $f : A_1 \times A_2 \times \cdots A_n \rightarrow C$ that correctly classifies any tuple $t \in \mathbf{r}$; that is, we want to find f that makes the following implicantion true:

$$\forall t \; t \in \mathbf{r} \; \Rightarrow f(t.A_1, t.A_2, \ldots, t.A_n) = t.C \qquad (1)$$

4. DEFINITIONS

DEFINITION 1. *A* **minterm** *is a non-empty word of fixed length over the alphabet $\{0, 1, *\}$. We will assume the length to be n, the number of feature attributes. Using the relation* **History** *in Fig.3, for example, $110*$ is a minterm. We provide a subscript $i \geq 1$ of the word to denote the i^{th} symbol in a minterm. For example, if $\alpha = 110*$, then α_2 means 1.* □

In a minterm 0 stands for **false**, 1 for **true**, and * for **don't care**. What exactly is meant by each of these will be developed presently. Continuing with minterms, a notion of logical distance is developed next.

DEFINITION 2. *Two minterms α and β are* **logically adjacent** *when they differ by exactly one symbol at the i^{th} position and either $\alpha_i = 1$ and $\beta_i = 0$ or $\alpha_i = 0$ and $\beta_i = 1$. For example, $10 * 1$ is logically adjacent to $00 * 1$, but not adjanct to $00 * 0$.* □

Minterms can be linearly ordered based on the number of 1s. This observation proves useful when looking for adjacent minterms, since they must be consecutive in the ordering.

DEFINITION 3. *An* **implicant** *is a valuation of feature attributes together with the set of tuples that are determined. The valuation depends upon an arbitrary, but fixed tuple $t \in \mathbf{r}$ and a minterm α. Each of the n symbols in the minterm uniquely corresponds to a feature attribute, e.g. the first symbol α_1 corresponds to A_1, the second α_2 to A_2, etc. A valuation of feature attributes is achieved by associating a tuple $t \in \mathbf{r}$ and minterm α as a pair (t, α) in the following way: If $\alpha_i = 1$, then the corresponding attribute value is $t.A_i$; If $\alpha_i = 0$, then the corresponding value is any value from the active domain except $t.A_i$; If $\alpha_i = *$, then the corresponding value is any value from the active domain.*

A tuple–minterm pair (t, α) can be used to determine a tuple $s \in \mathbf{r}$, denoted $\alpha \vdash_t s$ when $s.A_i = t.A_i$ if $\alpha_i = 1$, or $s.A_i \neq t.A_i$ if $\alpha_i = 0$. For example, using the relation **History**, $\mathtt{110*} \vdash_{t_{13}} t_{12}$. *An implicant is a triple $(t, \alpha, \{s \in \mathbf{r} | \alpha \vdash_t s\})$. For example, $(t_{13}, \mathtt{110*}, \{t_{12}\})$* □

PROPOSITION 4. *Given a relation* **r** *and an arbitrary, but fixed tuple $t \in \mathbf{r}$, form the set*

$$\mathcal{M} = \{(t, \alpha, \{s \in \mathbf{r} | \alpha \vdash_t s\})\}$$

where the minterm α is word in $\{0, 1\}^n$. Any tuple $s \in \mathbf{r}$ belongs to one and only one implicant.

PROOF *Assume the contrary, that s belongs to two different implicants. The implicants' minterms must disagree on at least one symbol at position i. Then $s.A_i$ will either agree with $t.A_i$ and s will belong to the minterm with $\alpha_i = 1$, or the converse; but not both.* □

There are a couple of important observations to be drawn from Proposition 4. First, the set **r**, given any tuple, is partitioned amongst one or more of the implicants. Second, many implicants will have feature values that do not determine any tuples.

DEFINITION 5. *Two implicants (t, α, s) and (t', α', s') are* **consistent** *if they agree on the tuple and either they determine the same set of classifier values or at least one of the set s, s' is empty. Formally, $t = t'$ and either $(\{s.C | (t, \alpha, s)\} = \{s.C' | (t', \alpha', s')\}$ or $s = \emptyset$ or $s' = \emptyset$. For example, $(t_{12}, 1000, \emptyset)$ and $(t_{12}, 1001, \{t_{24}\})$ are consistent, whereas, $(t_{12}, 0100, \{t_{23}\})$ and $(t_{12}, 0101, \{t_{26}\})$ are not.* □

By inconsistent it is either the case that the tuples do not agree or the non-empty set of classifier values determined are not equal[3].

DEFINITION 6. *A* **cover** *is a collection of implicants that are consistent. An unambiguous cover is a collection of implicants that have exactly one classifier value.* □

DEFINITION 7. *A* **prime implicant** *is an implicant that is not logically adjacent to another implicant that has a weaker minterm. An implicant that uniquely covers one or more tuples is essential, otherwise it is redundant.* □

[3]It is clear the consistency might be weakened to be subset; we have not yet investigated this, but imprudent use would require more recursion which, as an expensive procedure, should on the face be avoid.

The implicant $(t_{12}, 0**0, \{t_2, t_{12}, t_{13}\})$ is prime.

We will now formally describe the implicant production. Given two logically adjacent and consistent implicants (t, α, s) and (t', α', s'), that differ in their respective minterms at position i, we can produce a weaker, consistent implicant $(t, \alpha_1\alpha_2 \cdots \alpha_{i-1} * \alpha_{i+1} \cdots \alpha_n, s \cup s')$.

The bodies of the rules can, and often will, contain information about what feature values are *not* present. This is signficant not only in the generation of implicants, but also when Circle recurs. A rule body can have a feature involving negation, for example in Fig. 3 $\texttt{Age} \neq \texttt{old}$, or both an equality and inequalty for a particular attribute and using different values. To simplify discussion, we will call $\texttt{Age} \neq \texttt{old}$ a negative feature and $\texttt{Age} = \texttt{young}$ a positive feature. Notice that we can be safely ignore the negative feature (though maybe keeping it around), and instead focus on $\texttt{Age} = \texttt{young}$. One can imagine the $\wedge$ in this case functioning as a kind of intersection and $\texttt{Age} \neq \texttt{old}$ as the relative complement to value `old`.

One of several important details is how Circle works bears scrutiny. First, we examine how a tuple is selected. For our initial research here, we have implemented a random pick. Clearly, not all tuples will provide equally good rules. We have found in practice though that this simple strategy performs well. as well as ordering tuples based on the number of implicants that have conflicts. The former technique seems to work well in practice, though we are interested in discoverying good heuristics. We next consider how new implicants are produced. Earlier, from Sec.4, we saw that covers can be successively formed by *circling* the largest adjacent, consistent regions of the C-map; futhermore, the size of these regions must be a power of two. These regions correspond to "weaker" implicants–in the sense that the i^{th} minterm has fewer specific valuations with respect to the tuple-minterm pairing. Regions of the C-map that are empty, are treated like class "don't cares". These empty regions help determine which input values are unnecessary.

Another important detail is how the redundant prime implicants are selected. One can imagine a number of different approaches, but for this paper, we have implemented a simple, but effective greedy algorithm detailed in section 6 where we discuss Circle's implementation and application to various synthetic datasets, as well as datasets drawn from our current work in bioinformatics. We will conclude here by discussing one more computational detail, then examine some of the more appealing logic-theoretic aspects of Circle.

5. MODELS

From our previous discuss, we have seen that Circle initially produces a collection of rules that potentially contain positive as well as negative features. This rule set is actually non-monotonic with respect to adding tuples to the relation. Non-monotonicity provides a powerful means of reasoning, in Circle's case as a classifier. While an in depth discussion is not appropriate here, we would like to point a significant problem that monotonic classifiers face. The feature values must be known *apriori*; encountering a heretofore unknown value might result in the classifier either having nothing important to say or, worse yet, simply breaking. Circle, by virtue of Proposition 4 will find a rule class that holds true.

PROPOSITION 8. *Suppose Circle produces a rule set $\mathcal{R} = \{X_1 \Rightarrow Y_1, \ldots, X_k \Rightarrow Y_k\}$ over some relation* **r** *with features $\{A_1, A_2, \ldots, A_n\}$ and classifer C. A new instance s is presented that has at least one new feature value not present in the active domain of at least one attribute. Then only one class of the rules will hold true.*

PROOF *The rules were created using a sequence of tuples $t_1, \ldots, t_\ell$. The sequence of minterms that uniquely determine s is found by comparing the individual attribute values of t_i to s. For example, the first minterm for tuple t_1 is found by iterating over the n different attributes, concatenating 1 if $t.A_i = s.A_i$ and 0 otherwise. By Proposition 4, s can belong in only one implicant. Because of logically adjacency and consistency, s will remain in a single class as weaker implicants are produced.* □

Another interesting problem is adding a new training instance. One can imagine several approaches to incorporating the instance into the classifier as discussed in [9], though we will briefly sketch one of our own–a lazy, top-down approach that allows the new tuple to be distinguished from the implicants to which it belongs. For sake of illustration, assume a new tuple $s = (\texttt{W}, \texttt{sleepy}, \texttt{young}, \texttt{f}, \texttt{cured})$.

Notice that s triggers rules R_2, R_3 for the the class `sick`. Juxtaposing s and these two rules we see that by simply using the first two attribute values we can distinguish between the two classes, easily modifying R_3 and R_4 without much work and adding an additional rule to capture the exception, *i.e.*

$$
\begin{array}{ll}
R_{2a} & \texttt{Treatment} \neq \texttt{X} \wedge \texttt{Symptoms} \neq \texttt{sleepy} \\
 & \wedge\, \texttt{Symptom} \neq \texttt{cough} \Rightarrow \texttt{sick} \\
R_{2b} & \texttt{Treatment} \neq \texttt{X} \wedge \texttt{Symptoms} \neq \texttt{sleepy} \\
 & \wedge\, \texttt{Sex} \neq \texttt{male} \Rightarrow \texttt{sick} \\
R_{3a} & \texttt{Treatment} = \texttt{W} \wedge \texttt{Symptom} = \texttt{sleepy} \Rightarrow \texttt{cured}
\end{array}
$$

We can move toward a monotonic rule set by treating the negative features as relative complement. We call this rule set "mid-monotonic" for lack of a better term. For the original rule set above we have

$$
\begin{array}{ll}
R'_1 & \texttt{Treatment} = \texttt{X} \Rightarrow \texttt{cured} \\
R'_2 & \texttt{Treatment} \in \{\texttt{Y}, \texttt{W}, \texttt{Z}\} \\
 & \wedge\, \texttt{Symptom} \in \{\texttt{fever}, \texttt{sneeze}\} \Rightarrow \texttt{sick} \\
R'_3 & \texttt{Treatment} \in \{\texttt{Y}, \texttt{W}, \texttt{Z}\} \wedge \texttt{Sex} = \texttt{f} \Rightarrow \texttt{sick} \\
R'_4 & \texttt{Age} \in \{\texttt{old}, \texttt{adult}\} \wedge \texttt{Sex} = \texttt{male} \Rightarrow \texttt{cured}
\end{array}
$$

What is interesting to observe is that though these rules capture all the instances, they are over-specified and actually try to speak to instances that are not present. To complete our monotonic rule set, we simply verify which of the rules are supported by the instances.

Continuing with our example,

$$
\begin{array}{ll}
R''_1 & \texttt{Treatment} = \texttt{X} \Rightarrow \texttt{cured} \\
R''_2 & \texttt{Treatment} = \texttt{Z} \wedge \texttt{Symptom} \in \{\texttt{fever}, \texttt{sneeze}\} \\
 & \Rightarrow \texttt{sick} \\
R''_3 & \texttt{Treatment} = \{\texttt{W}, \texttt{Y}\} \wedge \texttt{Sex} = \texttt{f} \Rightarrow \texttt{sick} \\
R''_4 & \texttt{Age} \in \{\texttt{old}, \texttt{adult}\} \wedge \texttt{Sex} = \texttt{male} \Rightarrow \texttt{cured}
\end{array}
$$

Although space limitations does not permit us to discuss this here, we believe each of these different kinds of rule sets have something potentially interesting and important to say about the instances. An interesting experiment is to witness what rules would be generated if a functionally dependency were present. Formally, given a set of attributes R and two non-empty subsets $X, Y \subseteq R$, a functional dependency, denoted $X \rightarrow Y$, means the X values functionally determine the Y values. Slightly abusing our notation we have

$$\forall t \, \forall s \, (t.X = s.X) \Rightarrow (t.Y = s.Y) \tag{2}$$

This looks remarkably like Eq. 1 Among the nicer properties is an axiomization that allows reasoning about FDs. In a machine learning sense, FDs represent a kind of classifier where the X represents

```
CoreCircle(r)
 //r=relation to run Circle
  if size(classes(r)) < 2
    return;
  else
    t = pick(r);
    I[0] = MakeInitialImplicants(t, r);
    i = 0;
    while(I[i] != emptyset and i < |r|)
      I[i+1] = MakeNextImplicants(I[i]);
      i++;
    end while
    PI = MakePrimeImplicants(I);
    P = MakeEssentialImplicants(PI, r);
    for each (t, b1b2...bk, r') in P
      if size(r') < 2 or size(classes(r')) < 2
         MakeFullRule(t, b1b2...bk, r')
      else
         MakePartialRule((t, b1b2...bk, r'),
                              CoreCircle(r'))
      end if
    end for
  end if
end Circle
```

Figure 4: The Core Circle algorithm

the features, and Y the classifier. Suppose in a relation the FD $X \rightarrow Y$ holds, and there are k different X values. If Circle were to encounter an FD, and fortuitously X was chosen as the feature set and Y as the class labels, Circle would produce this rule set: $\mathtt{X} = \mathtt{x}_1 \Rightarrow \mathtt{y}_1, \mathtt{X} = \mathtt{x}_2 \Rightarrow \mathtt{y}_2, \ldots \mathtt{X} = \mathtt{x}_{k-1} \Rightarrow \mathtt{y}_{k-1}, \mathtt{X} \neq \mathtt{x}_{k-1} \Rightarrow \mathtt{y}_k$. Our observation is not a blind exercise, since this rule set represents the original dataset virtually unchanged. So, one aspect of making sure Circle does not waste time by reproducing the dataset is to include, for example, information dependency (IDs) measures that can give an indication of how close the data selected is to an FD. See [1]. IDs that are close to 0, represent FDs. Armed with this knowledge, Circle can have some chance to avoid discovering trivial classifiers.

6. THE CIRCLE ALGORITHM

The Core of Circle is based on a similar setting of the Quine-Mcclusky method for minimizing boolean functions. The algorithm takes a relation r as the parameter, and generates a set of rules set of rules that cover r. As before, r has the structure $R < id, A_1, A_2, \ldots, A_k, A_c >$, where each row has an identifying attribute, k attributes on which clustering is to be performed, and the classifier attribute A_c. Figure 4 shows the Core Circle algorithm. For the algorithm, and implicant is of the form $(t, b_1 b_2 \ldots b_k, \mathbf{r}')$, where $t \in \mathbf{r}$, $b_1 b_2 \ldots b_k$ is a minterm where $b_i \in \{0, 1, *\}$, and $\mathbf{r}' \subseteq \mathbf{r}$.

A crucial part of Circle is the tuple on which the minterms are created. The result of Circle can vary based on this starting tuple. Although the rulesets can be different, the set of rules will always produce a full cover of the relation. Different methods of generating the first tuple have been tested with Circle:

1. *Random tuple*: A random tuple is picked from the relation as the minterm-generating tuple. This can be the first tuple in the relation.

2. *Logic-theoretic heuristic*: This heuristic is based on the fact that the most inefficient step of Circle is the recursion, and the recursion in Circle is caused by conflicts in the classifier values for the same equivalence class of the minterms. This heuristic generates implicants for every pair of tuples, and picks the tuple which has the least number of equivalence classes with conflicts. In the case of the same number of conflicting equivalence classes, we pick the tuple with the least total number of tuples in the conflicting classes.

```
MakeInitialImplicants(t, r)
 // t=picked initial row
 // r=input relation to Circle
  M = generate all minterms of size |r|
  I[0] = InitializeImplicants(M)
  for every tuple t1 in r do
    add class(t1) to Implicant(minterm(t1,t)) in I[0]
  end for
end MakeInitialImplicants

MakeNextImplicants(I)
  //I = set of implicants
  Set Iout to empty
  Sort I in ascending order of 1s
  i1 = 0; i2=0;
  while i1 <=size(I) and i2 <= size(I) do

    set i2 s.t. num1(I(i2))= num1(I(i1))+1
    while (num1(I(i2))<num1(I(i1))+2)
      if (hammingdistance(I(i1),I(i2))==1
           and (classes(I(i1))==classes(I(i2))))
        Add combine(I(i1),I(i2)) to Iout
      end if
      i2++
    end while
    i1++
  end while
  return Iout
end MakeNextImplicants
```

Figure 5: MakeInitialImplicants and MakeNextImplicants

Note that since Circle was designed as a classifier and hence needs at least two classes to create its rules, one interesting application of Circle is to use it as a logic minimizer, in which case it can generate a minimal set of implicants even for a single class. In this case, the initial check for at least two distinct classes is omitted, and the result is a minimal set of rules to generate the single class, based on all of the tuples in r.

The *MakeEssentialImplicants* algorithm (shown in Figure 6) selects a minimal set of essential implicants from the Prime implicants generated. A greedy algorithm is used to generate the essential prime implicants. For each of the essential prime implicants created, a full rule can be created for every implicant for which the set of classifiers is 1. For all the implicants with a set of classifiers more than 1, CoreCircle is recursively called with the tuples covered by such implicants.

The last step is translating implicants to rules, the algorithm for which is shown in Figure 7.

6.1 Analysis of Circle

The Circle algorithm is centered around the recursive Core Circle algorithm. The most computationally expensive part of CoreCircle is the *MakeInitialImplicants* procedure, which is exponential in k, the number of attributes in r. This causes the execution time for Circle to explosively increase with the increase in the number of attributes. In our experiments, we found that Circle completes within 10-15 minutes in a workstation-quality machine (2x 1.6Ghz Zeon processors with 1GB RAM) for up to 15 attributes, after which

```
MakeEssentialImplicants(P,r)
  //P=set of prime implicants
  //r= input relation to Circle
  EP = emptyset;
  Sort the implicants in P in descending order
    of the number of rows covered
  while not Covers(EP, r) and not empty(P)
    i = RemoveFirst(P);
    EP = EP U {i};
    for each rule in P do
      remove rows already covered by EP
    end for
    Re-sort EP as above
  end while
  return EP;
end MakeEssentialImplicants
```

Figure 6: The Heuristic Pick, and The Greedy Essential Prime Implicant Generation Algorithm

```
MakeFullRule(t, m, r)
 //t = picked tuple
 //m = a minterm
 //r = subrelation containing 1 class
 // See text for details
end MakeFullRule

MakePartialRule(t, m, r, C)
  //t = picked tuple
  //m = a minterm
  //r = subrelation with >1 classes
  //C = recursively generated rules
  C1 = Generate conjunctive clause for t
  for each rule c in C
   for each conjunct ci in c
    if ci is of the form att_i=val_i
                    of  att_i != val_i
       and there is a conjuct in C1
             of the form att_i=val
       drop ci from c
    end if
    concatenate C1, c
   end for
end MakePartialRule
```

Figure 7: MakeFullRule and MakePartialRule

the execution time slows down rapidly. In order to cope with this problem, we incorporated a number of improvements in Circle, including (i) the use of threading to take advantage of multiple CPUs in an SMP architecture; (ii) the reduction of attributes in the recursion step by dropping attributes from the recursive call for which a positive value has already been generated, and (iii) generation of Random Attribute sets when Circle is run on high dimensional data.

7. CIRCLE IMPLEMENTATION

Circle is implemented in Java, configurable to use any JDBC-compliant database in the backend (or ODBC compliant databases using Sun's JDBC-ODBC bridge)[4]. The system was tested with Microsoft Access, Microsoft SQL server, and Oracle. The current implementation is configurable through a configuration file which could be specified from the command line, or set up using a simple web interface. Development of a fully configurable web-based application is part of the future enhancements.

[4]A binary version of the implementation is freely available http://www.kelley.iu.edu/sengupt/circle.

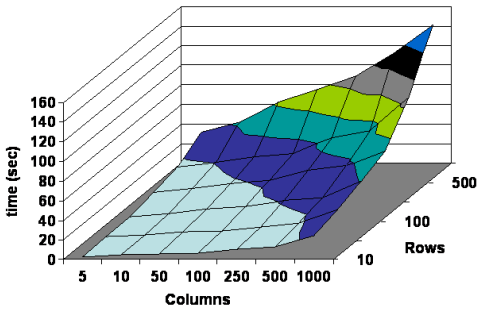

Figure 8: Performance of Circle showing dependence of time on both rows and columns of input

7.1 Experiments with Circle

Circle was tested in several experiments on real as well as synthetic data sets. All experiments were performed on a Workstation-class system with 2 x 1.6 Ghz CPU and 1.0G system RAM, running Windows Server 2003 Enterprise Edition. The experimentally evaluated performance, using n-fold cross validation, in our case four, matched the expected theoretical complexity of Circle. Deterministic Circle with no randomization demonstrated exponential behavior, but the randomized Circle with 8, 9 as well as 10 attributes improved the performance of Circle significantly. Here we summarize some of the performance factors of Circle.

1. *Quality of Rules*: Circle generates rules that are succinct, mimimal, and contains information which not only reflect positive and monotonic characteristics, but also negative and non-monotonic characteristics.

2. *Running time*: Core Circle is exponential on the number of attributes k, because of the need for generating all possible minterms of size k. We have shown that iteratively building a small random subset of attributes can produce rules that are similar in quality and yet can be determined in a fraction of the time. Figure 8 shows the running time of Circle on the "Internet" data set from UCI, containing information on pop-up ads in different websites.

3. *Cross-testing errors*: Because of the presence of the negative conjuncts, the rules produced by Circle are highly robust. In our experiments, we have found that the accuracy of the rules produced by Circle does not depend very heavily on the sample size. We ran Circle on the Wisconsin breast cancer dataset from the UCI repository [10], and ran Circle by only varying the sample percentage. Figure 9 shows the variation of running time and accuracy with the variation of training percentage.

7.2 Preliminary results on bioinformatics data

For several protein family sets in PROSITE[6] we included all subsequences of length three that occurred at least twice. We used this protocol because of the enormity of the features generated. Our preliminary results are very positive, having a success rate of more than 75%. As for the more interesting problem of automatically identifying the regular expressions, Circle terminated in a little under one hour, finding 201 rules which covered at least 50% of the human-generated curation sequences with some patterns being as high as 65%.

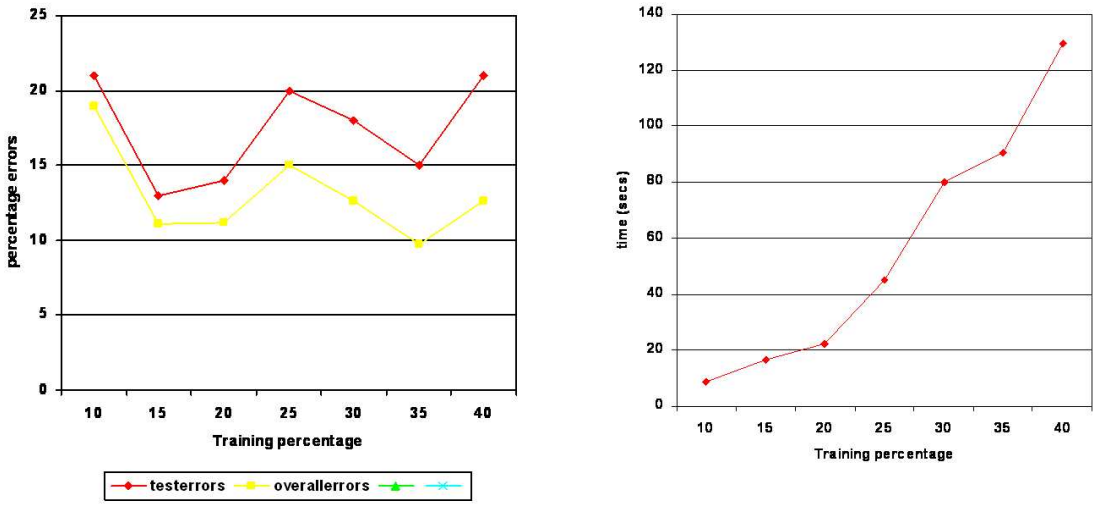

Figure 9: Graphs showing relative independence of accuracy with folding

8. CONCLUSION AND FUTURE WORK

We have begun work on creating a classifier whose inspiration is drawn from Boolean function minimization. The classifer called Circle possesses a number of attractive properties, *e.g.*, non-monotonicity. On the initial synthetic datasets we examined, Circle performed quite well. We are also encouraged by preliminary success on the bioinformatics data that drove all of this in the first place. A number of serious limitations in terms of complexity still need to be addressed. We have opted for an easy first-solution, probabilistically employing Circle. There are, no doubt, other techniques that can be adopted, by smartly guiding Circle's classification through the use of information theory. There is a lot work to do on the rule sets themselves, perhaps mining them or adding support and confidence as is done with association rules. We are also interested in applying Circle to streams, where work has begun in developing classifiers and clustering.

One interesting future direction is providing simple logspace arithmetic functions that are Church-Rosser, for instance addition. We can easily imagine implicants extended with counts–a kind of support. As implicants are weakened, their respective counts can be combined through adding. Consistency can then be modified to take into account large differences in the support. Another pursuit is clustering. We can *a priori* decide the number of clusters and build rules agglomeratively. Of course, we will begin looking at how to handle numerical attributes in the future. We are also interested in dealing with unknown values–a nagging problem present in all real data.

9. REFERENCES

[1] M. Dalkilic and E. Robertson. Information dependencies. In *Proc. of the 19th ACM SIGACT-SIGMOD-SIGART Symposium on the Principles of Database Systems (PODS-00)*, pages 245–253, 2000. Extended Abstract.

[2] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis*. Cambridge, 1998.

[3] W. J. Ewens and G. R. Grant. *Statistical Methods in Bioinformatics: An Introduction*. Springer, 2001.

[4] V. Guralnik and G. Karypis. A scalable algorithm for clustering protein sequences. In *Procceedings of the Workshop on Data Mining in BioInformatics (BIOKDD 2001)*, San Francisco, CA, USA, August 2001. available at http://www.cs.rpi.edu/ zaki/BIOKDD01/.

[5] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning - Data Mining, Inference and Prediction*. Springer, 2001.

[6] K. Hoffmann, P. Bucher, L. Falquet, and A. Bairoch. The PROSITE database: Its status in 1999. *Nucleic Acids Research*, 27:215–219, 1999.

[7] M. Karnaugh. The map method for synthesis of combinatorial logic circuits. *Trans. AIEE. pt I*, 72(9):593–599, November 1953.

[8] L. Liao and W. S. Noble. Combining pairwise sequence similarity and support vector machines for remote protein homology detection. In *Proceedings of the Sixth International Conference on Computational Molecular Biology*, pages 225–232, April 18-21 2002.

[9] G. F. Luger. *Artificial Intelligence - Structures and Strategies for Complex Problem Solving*, chapter 8, pages 305–320. Addison Wesley, 4th edition edition, 2002.

[10] O. L. Mangasarian and W. H. Wolberg. Cancer diagnosis via linear programming. *SIAM News*, 23(5):1 & 18, September 1990.

[11] E. L. McCluskey Jr. Minimization of boolean functions. *Bell System Technical Journal*, 35:1417–1444, April 1959.

[12] T. M. Mitchell. *Machine Learning*. McGraw Hill, 1997.

[13] D. W. Mount. *Bioinformatics - Sequence and Genome Analysis*. Cold Spring Harbor Laboratory Press, 2001.

[14] A. Murzin, S. Brenner, T. Hubbard, and C. Chothia. SCOP: A structural classification of proteins database for the investigation of sequences and structures. *Journal of Molecular Biology*, 247:536–540, 1995.

[15] S.L.Salzberg. On comparing classifiers: Pitfalls to avoid and a recommended approach. *Data Mining and Knowledge Discovery*, 1:317–328, 1997.

[16] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.