Genetic Algorithms as a Data Discretization Method

Mike Waldron
Department of Math and Computer Science
South Dakota School of Mines and Technology
Rapid City, South Dakota, 57701
michael.waldron@gold.sdsmt.edu

Dr. Manuel Penaloza
Department of Math and Computer Science
South Dakota School of Mines and Technology
Rapid City, South Dakota, 57701
manuel.penaloza@sdsmt.edu

March 10, 2005

Abstract

Data discretization is the process of converting continuous attributes in an input file into a nominal form by grouping them into discrete partitions and replacing continuous values with the corresponding partition label. This is an important step in the process of machine learning, since it's easier for classifiers to deal with nominal attributes than it is to deal with continuous attributes. The problem of how best to accomplish this conversion is well-understood. Over the years, many different methods of performing discretization have been proposed, explored, and implemented. Here, we present what we believe to be a fresh approach: the use of a genetic algorithm.

1 Introduction

Data discretization is the process of grouping numeric attributes into discrete partitions. This paper will provide some background for readers unfamiliar with the process of discretization, and present our approach, which we believe is unique: a Genetic Algorithm. Throughout this paper we'll refer to the use of a genetic algorithm as a discretization technique as simply "GA".

1.1 The Motivation for Discretization

Discretization is necessary and helpful for a number of reasons. The main reason is that human beings naturally move about in the world dealing with discrete rather than continuous quantities. Consider for a moment the famous iris data set [2]. An experienced gardener is unlikely to say that the petal length of the *setosa* variety of iris fall into the range $[1.0, 1.9]$ centimeters with a certain probability. This is an awkward way to go about identifying iris types, even though it's very precise. Gardeners will typically leave such statements up to researching botanists, and prefer instead to frame their knowledge in terms of *setosa*s having "shorter" petals than either of the other two varieties. "Shorter," in this case, is a discrete partition label that associates all the petal length measurements for the *setosa*s together in an intuitive fashion. When classifiers operate on discrete data with intuitive partition labels, the results are much easier to understand.

Another reason to discretize numeric data is that discrete data is generally better received by classifiers. Some classifiers are simply unable to handle numeric data. Those that can sometimes make inappropriate assumptions about the data they're working on–one common assumption is that the data is normally distributed. Using real-world data, this is not often the case. Even classifiers that can avoid making inappropriate assumptions about the underlying probability distributions tend to operate much more quickly when dealing with discrete data, since they're freed from having to repeatedly sort the data at each step [9].

1.2 Classification of Discretization Methods

Many different methods of discretization have been explored over the years. In [5], Liu et. al. provide a good survey of the various methods. Of particular interest is their "taxonomy" which categorizes methods based on three key features:

1. Merging vs splitting. Merging is a "bottom-up" technique where each instance starts out in its own partition. As the algorithm runs, partitions are merged. Splitting is a "top-down" approach where the data is initially un-partitioned, and partitions are created as the algorithm runs.

2. Supervised vs. unsupervised. A supervised method takes class labels into consideration, whereas an unsupervised method does not.

3. Discretization measure. Four separate measures are mentioned: dependency, binning, entropy and accuracy. In this paper, we concern ourselves only with the entropy measure. Entropy calculation is discussed in section 3.5.1.

Dougherty et. al., present a different way of classifying discretization methods in [3], that uses three separate and orthogonal *axes*, as follows:

1. Global vs. local. Local methods produce partitions based on localized regions of the instance space, where global methods produce partitions based on the instance space considered as a whole.

2. Supervised vs. unsupervised, which has the same definition as in the Liu taxonomy [5].

3. Static vs. dynamic. Static methods consider each attribute independently of the others, and make multiple passes through the data to discretize an entire dataset. Dynamic methods consider all continuous attributes in parallel, and can capture interdependencies between attributes.

1.3 Overview

Section 2 will demonstrate a strong similarity between GA and the Minimum Description Length Principle (MDLP) method developed by Fayyad and Irani [4]. In doing so, it will demonstrate a justification for comparing these two methods to evaluate GA's suitability as a means of discretizing continuous attributes.

Section 3 will introduce the GADiscrete application, and delve into technical issues such as the mapping of vocabulary between the genetic algorithm and machine learning domains, how genetic operators are applied to this problem, how fitness is evaluated, and when GADiscrete decides it has found the best discretization scheme.

Section 4 will document an experiment we ran to gauge the effectiveness of GA. It will present results we were able to achieve, and compare them to results from MDLP and also to results from applying a classifier directly to the continuous data.

Finally, section 5 will conclude with a look at outstanding problems and possibilities for future work in this area.

2 Comparing GA to MDLP

GA can't be strictly classified using the Liu taxonomy, because it is neither a merging nor splitting technique. It's closer to a splitting technique, but does not recursively generate partitions. Instead, arbitrary sets of partition boundaries of varying cardinalities are created, considered, and destroyed "all-at-once." Strictly interpreted, Liu's definition of splitting appears to require recursion, and GA is clearly not recursive.

Ignoring our discomfort with the merging vs. splitting issue for a moment, GA bears a striking similarity to the MDLP method developed by Fayyad & Irani [4]. Both methods are classified the same way by both classifications–as supervised, entropy-based methods using the Liu taxonomy, and as global, supervised, static methods using the Dougherty classification. Because of the high degree of similarity between the two methods, and because we have already implemented MDLP as part of an ongoing data mining research effort, we are interested in particular in comparing GA to MDLP as a means of measuring GA's suitability for this purpose.

MDLP works by sorting the instances by a single attribute value, and initially treating the entire data set as a single partition. Then, at each step, the set of *boundary points* of each partition are considered for new partition boundaries. (A boundary point is a division between instances in the sorted data where both the attribute value and class change.) The boundary point that produces the greatest information gain is selected as a cut point. This creates two partitions out of one, and each new partition is put back on the stack for further consideration.

The trick is knowing when to stop partitioning. Naturally, an unadorned recursive algorithm will stop on its own when each instance is in its own partition, in which case entropy has been reduced to zero. However, this is of little use. Simply replacing numeric attributes with discrete labels in a one-to-one fashion really isn't discretization at all. Clearly, a more sophisticated stopping criterion is needed. Fayyad and Irani address the stopping problem with the minimum description length principle, from which the method takes its name. Roughly speaking, MDLP reframes the stopping criteria as a communication problem, and stops when the cost to transmit two partitions resulting from a split exceeds the cost of transmitting the same set of instances as a single partition.

There are limitations on where a data set may be partitioned. A change in attribute value is required of a potential cutpoint, since any discretization scheme must be a surjection [8] from continuous attribute values to discrete labels. This is necessary to prevent ambiguity as to the meaning of the partition labels. Both GA and MDLP adhere to this requirement. However, MDLP introduces an additional restriction that the class label must *also* change. This is to avoid unnecessarily splitting instances of the same class, which Fayyad and Irani claim results in larger and lower-quality trees [4]. We've found experimentally that requiring a change in the class label actually lowers the accuracy of a classifier operating on the discretized data, and have therefore elected to consider partition boundaries wherever the attribute value changes. In spite of this change, GA remains a supervised method since the class labels are required to calculate entropy.

3 GADiscrete

GADiscrete is a Java application that uses a genetic algorithm to find a good set of cutpoints. Generally speaking, four characteristics set GAs apart from other algorithms: populations of chromosomes, and the application of genetic operators crossover, mutation, and selection. Although some definitions of a GA include an inversion operator, Mitchell [6] points out that inversion is not widely accepted, and its advantages are debatable. GADiscrete was implemented with no small amount of influence from Mitchell's work, and therefore does not apply an inversion operator.

3.1 Mapping Of Terms

There are two sets of terms that come into play when discussing discretization by means of a GA: those from the GA domain, and those from the machine-learning domain. Therefore, we'll take a moment to map terms between the two domains:

gene refers to a possible cutpoint in a raw data file. In other words, a point in a data file where a partition boundary may be drawn between instances. Genes are held in a gene pool, and represented as binary numbers within chromosomes.

chromosome refers to a collection of genes, as in biology. Represents one possible discretization of an input file, a "discretization scheme." The genes in a chromosome are represented by a set of bits C (a "bitset") with a cardinality equal to the number of genes in the gene pool. If bit C_i is "set" ($C_i = 1$) then the chromosome contains the ith gene in the gene pool. If C_i is "cleared" ($C_i = 0$), then the chromosome *does not* contain the ith gene in the gene pool.

organism is a synonym for chromosome when used with respect to GADiscrete. This is inconsistent with the biological definition of the word, but makes it easier to use the term "population."

population refers to a collection of organisms, which is to say a collection of chromosomes within GADiscrete.

Throughout the rest of this paper, we'll constrain ourselves to using genetic terms in lieu of machine learning terms.

3.2 User-Defined Parameters

GADiscrete makes use of seven parameters that the user can tune at runtime to adjust the behavior of the GA. Constraints are enforced to prevent out-of-range or illogical values from being used. The parameters, along with their constraints, are as follows:

G_{min} is the minimum number of genes a chromosome is required to have. A chromosome with fewer than G_{min} genes is not survivable, and will die. $G_{min} > 0$ is enforced.

G_{max} is the maximum number of genes a chromosome is allowed. A chromosome with more than G_{max} genes is not survivable, and will die. $G_{max} > 0$ and $G_{max} \geq G_{min}$ are enforced.

P_c is the probability of crossover. $0 < P_c < 1$ is enforced. $P_c = 0$ is not permitted because the mutation and selection operators do nothing if no offspring are produced. Setting $P_c = 0$ would reduce the GA to a static collection of randomly-generated chromosomes.

P_m is the probability of mutation. $0 \leq P_m < 1$ is enforced.

T_f is a fitness threshold. Its meaning is complex, therefore the discussion of T_f is deferred until section 3.5.1. T_f is a percentage, and is therefore constrained to $0 \leq T_f < 1$.

N_g is the maximum number of generations that the GA should run. Depending on the input data and other parameters, the GA may stop before producing N_g generations. This is discussed in detail in section 3.6. $N_g > 0$ is enforced.

N_p is the population size. Specifically, it is the number of organisms in the population between generations. N_p is constant during a GA discretization run. N_p is constrained as $N_p > 2$, since two organisms are required at a minimum for crossover to work.

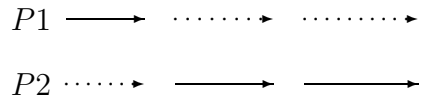
It will also be convenient to define the variable G_c, which will denote the gene count of an arbitrary chromosome.

3.3 Crossover

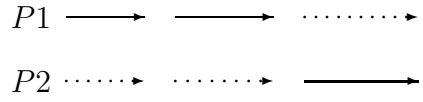
Loosely put, crossover mixes the genes of two parent chromosomes to create two offspring. At all times, the GA maintains a population of chromosomes divided into two categories: parents and offspring. To perform crossover, each parent chromosome is selected in turn ("P1") and given a chance to "breed" with another parent chromosome. Whether or not the crossover actually occurs depends upon a randomly-generated value v such that $0 \leq v < 1$. If $v \leq P_c$, then a mate ("P2") is selected at random from among the parent chromosomes, and crossover proceeds. Otherwise, no offspring are produced. This is a slight departure from Mitchell's definition of crossover [6], in which the offspring are simply clones of the parents if $v > P_c$. As a notational convenience, the offspring will be referred to as "O1" and "O2," although they have no intrinsic ordering. Crossover is commutative: breeding P1 with P2 is equivalent to breeding P2 with P1.

GADiscrete uses two-point crossover, which means that two locii are chosen randomly that divide P1 and P2's genes into three sections. That, coupled with crossover's commutivity means that there are three types of crossover that can occur. Which type is used for each crossover operation is determined randomly. We identify the three types as "early crossing," "late crossing," and "both crossing." Their definitions, with accompanying diagrams, follow. In the diagrams, solid lines indicate gene partitions received by O1, and dashed lines represent gene partitions received by O2.

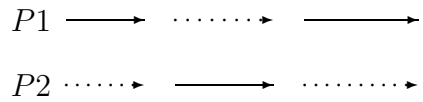
early means "switch to the other parent's genes early." In other words, O1 receives P1's first section, and P2's second and third section. O2 receives all other sections.



late means "switch to the other parent's genes late." In other words, O1 receives P1's first and second section, and P2's third section. O2 receives all other sections.



both means "switch to the other parent's genes early, then switch back again late." In other words, O1 receives P1's first and third section, and P2's second section. O2 receives all other sections.



The locii for P1 and P2 are not necessarily the same. Neither are P1 and P2 required to have the same number of genes. Furthermore, up to two empty gene sections are allowed. In other words, it's possible that all the genes of a chromosome might land in a single section.

Sometimes, as in nature, crossover fails to produce offspring. There are three ways this can happen:

1. The chromosomes are clones of the parents. This happens when, for example, the first two gene sections of both parents contain zero genes, and late crossing is utilized. Then O2 is a clone of P1, and O1 is a clone of P2. Allowing clones to survive would cause the population to homogenize too rapidly, potentially trapping the GA in one particular area of the search space.

2. P1 and P2 are actually the same chromosome. This is possible because P2 is selected at random from the population, which includes P1. When this happens, crossover is aborted immediately. The likelihood of producing clones in this case is high, and as has just been discussed, clones are undesirable.

3. G_c of an offspring fails to meet the constraint $G_{min} \leq G_c \leq G_{max}$. Such chromosomes are not survivable, and are simply deleted.

Detection and deletion of unfit offspring is used instead of prevention simply because it's easier to implement.

3.4 Mutation

Mutation changes a chromosome by either adding or subtracting a gene at random. It has the effect of moving the GA to a different area of the search space. This helps keep the GA from getting trapped at local fitness maxima, increasing the odds that it will converge at the global fitness maximum instead. Whether mutation occurs depends on a randomly-generated value v such that $0 \leq v < 1$. When $v \leq P_m$, the chromosome is mutated.

Mutation is the simplest of the three operators. Chromosomes are represented as a bitmap with as many bits as there are genes in the gene pool. A bit that is "set" (has the value 1) indicates that the chromosome contains the corresponding gene, whereas a bit that is "cleared" (has the value 0) indicates that the chromosome *does not* contain the corresponding gene. Then, mutation is simply a matter of flipping a randomly-selected bit in the bitset. This is guaranteed to either add or subtract a gene from the chromosome. Of course, this means that a potential exists for a mutated chromosome to have $G_c > G_{max}$ or $G_c < G_{min}$. When mutation invalidates a chromosome in this fashion, the chromosome is deleted.

Naturally, mutation also changes a chromosome's fitness, so that fitness must be calculated again after mutation occurs. Fitness is discussed in detail in 3.5.1.

3.5 Selection

Selection is by far the most complex of the three genetic operators used by GADiscrete, because it requires a definition of fitness, and defining fitness is non-trivial.

3.5.1 Fitness

For GADiscrete, there are two components that determine a chromosome's fitness. The first is entropy–low entropy is more desirable. The second is simplicity–simpler chromosomes are better. Unfortunately, these are conflicting goals. Entropy can be lowered all the way to zero if we're willing to accept arbitrarily complex chromosomes. This occurs when each instance is assigned to its own partition. However, this is of no help. The point of discretization is to group continuous attributes into ranges and assign them discrete labels. If there's no grouping, then discretization hasn't happened at all. Likewise, if the simplicity is carried to an extreme, the resulting chromosome would also feature the highest entropy.

We desire, then, the lowest entropy possible using as few genes as possible. Clearly, a balance must be struck, since the addition of genes lowers entropy. GADiscrete employs a fitness function that calculates two fitnesses: one referred to as the "raw" fitness and the fitness used in selection which is simply called "fitness."

The raw fitness is the difference between the maximum possible entropy of a chromosome

and its actual entropy. Maximum entropy is determined by the number of unique class labels c in the input file, and is calculated as $\log_2 c$. A chromosome's entropy is calculated as

$$\sum_{i=1}^{n} \left(\left(|P_i|(\log_2 |P_i|) - \sum_{j=1}^{m} |P_{im}|(\log_2 |P_{im}|) \right) \left(\frac{|P_i|}{q} \right) \right)$$

with n the number of partitions, P_i the set of instances in the ith partition, m the number of unique class labels in P_i, P_{im} the set of instances of partition P_i that have the mth class label, and q the number of instances in the entire input file.

Before we continue, it behooves us to introduce some additional variables. Let G_X represent the number of genes in chromosome X, G_{ce} the "excess" genes of an arbitrary chromosome, or the number of genes by which G_c exceeds G_{min}. Also let F_c represent the fitness of an arbitrary chromosome, F_{cr} the raw fitness of an arbitrary chromosome, F_X the fitness of chromosome X, and F_{Xr} the raw fitness of chromosome X.

The fitness threshold T_f is a percentage used to specify how much better F_{Br} must be than F_{Ar} for chromosome B to be accepted over chromosome A, where B is maximally complex ($G_B = G_{max}$) and A is minimally complex ($G_A = G_{min}$). In other words, the GA will select B over A when

$$F_{Br} > F_{Ar}(1 + T_f)$$

F_c is calculated from F_{cr} by introducing a penalty proportional to G_{ce}. A chromosome with $G_c = G_{min}$ suffers no penalty, and will have $F_c = F_{cr}$. A chromosome with $G_c = G_{max}$ suffers a maximum penalty, and will have $F_c < F_{cr}$. The penalty p is calculated as

$$p(G_e) = b^{Ge}$$

where

$$b = \sqrt[G_{eMax}]{1 + T_f}$$

and

$$G_{eMax} = G_{max} - G_{min}$$

p is used as a divisor for the raw fitness, and F_c is calculated as

$$F_c = \frac{F_{cr}}{p}$$

3.5.2 The Roulette Wheel

Selection in GADiscrete is fitness-proportional, which means that less-fit chromosomes are not necessarily barred from reproducing, it's just less likely. As Mitchell points out in [6], choosing the fittest chromosomes for crossover at each generation and discarding the less-fit ones results in homogenizing the population too quickly. As with the production of clones resulting from crossover, this can maroon the GA in a particular area of the search space. Allowing less-fit chromosomes to breed occasionally helps preserve diversity. As in nature, diversity results in a fitter overall population.

Fitness-proportional selection is implemented as a virtual roulette wheel, where the width of each section of the wheel is variable according to a chromosome's fitness. Fitter chromosomes get wider sections, less-fit ones get narrower sections. Once all chromosomes in the population have been allotted their space on the wheel, a random number in $[0, 1)$ is generated that indicates a particular section on the wheel. The corresponding chromosome is selected to survive to the next generation, and is removed from the wheel. To avoid leaving a blank spot on the wheel, it must be re-normalized after every spin: the sections are re-allocated based on the remaining chromosomes as before, and the whole process is repeated again. This continues until N_p organisms have been selected from the overall population (parents and offspring). These selected organisms become the parents of the next generation, and those organisms that have not been selected are simply discarded.

3.6 The Stopping Criterion

Deciding when to stop the evolution in a GA is difficult. GAs are most useful when the search space is large enough that exhaustive search methods are impractical. Short of an exhaustive method, one can't tell whether an optimal solution has been achieved. Clearly, some sort of heuristic must be applied.

Aytug and Koehler have developed a method for determining an upper bound on the number of generations required to see all possible populations [1]. This requires fewer generations than a brute-force exhaustive search, but unfortunately in many cases still requires too many generations for our purposes. They acknowledge that their stopping criterion is a theoretical bound. However, we have practical concerns about the runtime, and need to tread a fine line between stopping the GA before convergence and letting it run unnecessarily long.

We briefly experimented with a method that attempted to detect convergence, by assuming that at convergence, the population would be completely homogeneous. As it turned out, this is not the case. Generally, when the GA gets to a point where it's returning very good results, the final population *exhibits a high degree of homogeneity*, but is not completely homogeneous. What we've seen, for example, is a final population of 25 chromosomes that are various combinations of a small subset of the available genes in the gene pool. Instead of attempting to quantify homogeneity, we've instead implemented a simple method that checks for monotonic decline in the sum of fitnesses over the population. When the overall fitness declines monotonically for 10% of N_g, the GA assumes it has already seen the fittest

chromosome it's going to find, and stops. $N_g(0.1)$ was chosen experimentally by leaving $N_g = 10000$ and observing that in general, 1000 generations are sufficient to return a very fit chromosome.

4 Results

	Continuous	MDLP	GA
iris	96.0000%	94.6667%	95.4403% $\pm$ 0.8204%
adult	82.8426%	83.2836%	84.0894% $\pm$ 0.1368%
breast	96.3397%	97.6574%	96.3353% $\pm$ 0.4347%

Table 1: Naïve-Bayes Accuracies

	MDLP	GA
iris	93.3333 %	95.6567% $\pm$ 0.8766%
adult	78.5591 %	84.1203% $\pm$ 0.1505%
breast	95.0220 %	96.3533% $\pm$ 0.4180%

Table 2: ID3 Accuracies

To gauge the effectiveness of GA, we ran an experiment that discretized the datasets *adult*, *iris*, and *breast-cancer-wisconsin* from the UCI Machine Learning Repository [2]. Instances containing missing attribute values were removed from *adult* before processing. We ran 27 separate trials, each of which used slightly different parameters, as follows:

$$N_g = 10000$$

$$N_p = 25$$

$$G_{min} = 1$$

$$G_{max} = 10$$

$$T_f \in \{25\%, 50\%, 75\%\}$$

$$P_c \in \{25\%, 35\%, 45\%\}$$

$$P_m \in \{1.5\%, 2.5\%, 3.5\%\}$$

The variable parameters were treated as orthogonal axes to allow us to compare results of varying any one of them against the other two. In effect, then, each trial is a point in the GA's parameter space, and the experiment as a whole can be thought of as a 3x3x3 cube where each point represents a single trial. At each trial, the GA was run five separate times,

producing 135 runs over the entire experiment. This was done because although the GA converges well, it doesn't produce the same fittest chromosome each time.

We also discretized the same three datasets using our own implementation of MDLP. Then, two classifiers were run on the discretized data: the ID3 and Naïve-Bayes implementations in the Weka data mining tool [9]. Naïve-Bayes was also run against the raw datasets, but this test was omitted for ID3 because it's unable to deal with continuous attributes. Weka exposes its classifiers in the form of an API, which is extremely useful. To automate the experiment, we wrote a Java driver that calls the Weka API with appropriate parameters to perform the classifications. This program was then invoked via a bash script, to reduce errors and decrease the amount of time required to run the experiment. Throughout the experiment, classifiers were trained using five-fold cross-validation. Accuracies achieved using the Naïve-Bayes classifier are summarized in table 1. Accuracies achieved using the ID3 classifier are summarized in table 2. Results for GA are presented as an average over all 135 runs, plus or minus one standard deviation. This notation is unecessary for continuous data and MDLP-discretized data, since those results are invariant.

It's interesting to note the disparity in accuracies among the datasets. Accuracies for *iris* vary roughly from 93% to 96%, while the accuracies for *adult* are much lower–approximately 78% to 84%. There are at least two reasons for this. First, classifiers operate on an implicit assumption that the class depends exclusively on a conjunction of attribute values [7]. A classifier will always exhibit a non-zero error rate, to the degree that this assumption is untrue. For example, it's probably possible for a *setosa* to have measurements identical to those of a *virginica*. Using nothing but the four measurements given, a classifier would be unable to distinguish two such irises with 100% accuracy. A second reason is the complexity of the conjunction of attribute values. The more complex this conjunction is, the longer a training data set must be to provide the same coverage of the problem space. For a fixed size training data set, the only option is for classifier accuracy to decrease as the complexity of the conjunction of attribute values increases. It would be nice to be able to quantify the effects of these two complications to obtain an upper bound on accuracy, but to do so would require quantifying the "arbitrariness" in a dataset (which presents an obvious paradox), and also require quantifying the inductive behaviors of all current and future classifiers.

It appears that varying parameters within the "cube" as we described had little effect on the final discretization, as evidenced by the low standard deviations that resulted from the GA trials. There are certainly parameter settings that will result in worse discretizations than what we've achieved here: setting P_c to something near-zero will practically guarantee that the "fittest" chromosome is actually just a randomly-generated member of the initial population. However, we suspect that there aren't many combinations of parameters that would lead to significantly better results. As we've discussed, there is an upper limit on the attainable accuracy inherent in most data sets, and at worst, GA produces results which are comparable to well-established and accepted methods of discretization.

5 Outstanding Problems and Future Work

GADiscrete's stopping criterion is very informal, and could be entirely inappropriate for data sets that feature a high degree of interdependence between attributes. It might be beneficial to develop a more sophisticated stopping heuristic. One possibility is to detect when the same subset of genes is being used repeatedly from generation to generation. Another possibility is to check for cycles where the same populations repeatedly appear with some small period.

Dougherty et. al., make a brief mention of dynamic discretization methods that are capable of capturing interdependencies between attributes [3], but decline to discuss them further. It would be interesting to investigate dynamic methods more thoroughly to see whether they could be applied toward producing an upper bound on classifier accuracy. A hard upper bound on accuracy probably still would prove elusive, due to complications mentioned in the previous section, but dynamic methods could possibly provide a rough upper bound.

6 Acknowledgements

A genetic algorithm is an artificial metaphor for biological genetics. It's difficult to create a metaphor for something one doesn't understand. Since I know very little of biology, I'd like to thank my wife, Jeannette Waldron, M.D., for helping me gain an understanding of genetics sufficient to implement a genetic algorithm. –mjw

We would also like to express our appreciation of the Weka data mining tool [9], developed by Dr. Ian Witten and Dr. Eibe Frank of the University of Waikato, New Zealand. It is an excellent tool which they've made freely available to the academic community. Without it, this project would have been significantly more difficult.

References

[1] Haldun Aytug and Gary J. Koehler, *New stopping criterion for genetic algorithms*, European Journal of Operational Research (1996), no. 126, 662–674.

[2] C.L. Blake and C.J. Merz, *UCI repository of machine learning databases*, 1998, http://www.ics.uci.edu/∼mlearn/MLRepository.html.

[3] James Dougherty, Ron Kohavi, and Mehran Sahami, *Supervised and unsupervised discretization of continuous features*, Machine Learning: Proceedings of the Twelfth International Conference, 1995.

[4] Usama M. Fayyad and Keki B. Irani, *Multi-interval discretization of continuous-valued attributes for classification learning*, IJCAI Proceedings, 1993, pp. 1022–1027.

[5] Huan Liu, Farhad Hussain, Chew Tim Lan, and Manoranjan Dash, *Discretization:an enabling technique*, Data Mining and Knowledge Discovery (2002), 6,393–423.

[6] Melanie Mitchell, *An introduction to genetic algorithms*, MIT Press, Cambridge, 1996.

[7] Tom M. Mitchell, *Machine learning*, McGraw-Hill, Boston, 1997.

[8] Eric W. Weisstein, *Surjection*, From *MathWorld*, A Wolfram Web Resource, http://mathworld.wolfram.com/Surjection.html.

[9] Ian H. Witten and Eibe Frank, *Data mining: Practical machine learning tools with java implementations*, Morgan Kaufmann, San Francisco, 2000.