To the Graduate Council:

I am submitting herewith a thesis written by Isaac Sherman entitled "On the Role of Genetic Algorithms in the Pattern Recognition Task of Classification." I have examined the final paper copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

_____

Bruce MacLennan, Major Professor

We have read this thesis
and recommend its acceptance:

_____

Dr. Bruce MacLennan

_____

Dr. Hairong Qi

_____

Dr. Catherine Schuman

Accepted for the Council:

_____

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

To the Graduate Council:
I am submitting herewith a thesis written by Isaac Sherman entitled "On the Role of Genetic Algorithms in the Pattern Recognition Task of Classification." I have examined the final electronic copy of this thesis for form and content and recommend that it be accepted in partial fulfillment of the requirements for the degree of Master of Science, with a major in Computer Science.

Bruce MacLennan, Major Professor

We have read this thesis
and recommend its acceptance:

Dr. Bruce MacLennan

Dr. Hairong Qi

Dr. Catherine Schuman

Accepted for the Council:

Dixie L. Thompson

Vice Provost and Dean of the Graduate School

(Original signatures are on file with official student records.)

On the Role of Genetic Algorithms in the Pattern Recognition Task of Classification

A Thesis Presented for

The Master of Science

Degree

The University of Tennessee, Knoxville

Isaac Sherman

May 2017

dedication...

Acknowledgements

I would like to thank...

Beep boop son, beep boop

Abstract

Abstract text goes here...

Contents

List of Tables

List of Figures

Chapter 1

Introduction

April 12 deadline- two passes (a week each) gives us roughtly last week of march for rough draft. Look up correlation coefficient //

A Genetic Algorithm (GA) is a biologically inspired form of computing. GAs can be used for many different purposes, from optimization to classification to design and testing. They can be applied anywhere that a solution can be encoded and then autonomously evaluated. However, they are most well understood as a general solution to optimization as a form of stochastic gradient descent, similar conceptually to simulated annealing. In this paper, I attempt to narrow the role of GAs as they pertain to pattern recognition and classification. In the remainder of this chapter I will describe GAs in general terms and discuss the motivations of this paper. In chapter 2, I will discuss the specifics of the GAs which I use in this paper in detail. In chapter 3 I will describe the experiments conducted and discuss the results. In Chapter 4 I will describe future avenues of study which are available.

Genetic Algorithms Introduction Imagine a colony of rabbits. The rabbits are quite content to munch on clovers and thistles and the like. Some rabbits are much more content than others, and have significant girth. One day foxes find the rabbits. There are many more rabbits than foxes, and the foxes can't eat all of them. The heaviest rabbits are both the slowest and the most appetizing to the foxes, and they

are the first to go, but many die. They have failed the evolutionary filtering process. However, the rabbits that survive are thinner, and the most successful are likely faster. These are the rabbits which survive to populate the next generation.

GAs encapsulate this process, though usually with much less mayhem. The algorithm is as follows:

```
Population := RandomInitialization()
        while True:
                for each Solution in Population:
                        Evaluate Solution
                        Assign Fitness to Solution
                newPopulation = SurviveAndBreed(Population)
                Population = newPopulation
        end while
```

Stopping Conditions This algorithm can run for a predetermined number of generations, indefinitely, or until another specific criteria is met. Consider the problem of finding a way of combining 4 operands with 3 operators to achieve a particular value. There are often many ways to solve this problem. If one was to use a GA to solve the equation, the algorithm could stop as soon as it had a valid solution values for a, b, c, d, and the 3 operators which satisfied the equation. For example:

$$a\ op_1\ b\ op_2\ c\ op_3\ d = x$$
$$3 \times 7 \times 3 + 5 = 68$$
$$9 \times 7 + 8 - 3 = 68$$

Solution Encodings Now lets look specifically at what is meant by a solution. First, GAs usually have some encoding based ultimately on a string of 0s and 1s, called

a bitstring[1]. Continuing with our example, we know that there are 4 operands which have a value between 0 and 9, or 10 values total. To encode that in a bitstring we use the values 0000-1010. We could also include some error correcting code to randomly reassign the bits if they go outside of the values, ensuring that any solution randomly generated contains valid data (we could also use more bits and/or arbitrarily assign values via modulo arithmetic, but this is the most instructive method for our current purposes). Next, we have the operators, which can be $+$, $-$, $*$, $/$. These fit nicely within the 00-11 bit range, with no need of error correction. So we have a bitstring with the form xxxx-xx-xxxx-xx-xxxx-xx-xxxx[2], 22 bits which satisfy the constraints of our problem. The second line of the above example would be 0011-10-0111-10-0011-00-0101.

Fitness Functions Following along with the algorithm, we need to assign fitness. In this case the closer one is to the solution, the better, usually [3]. There is research showing that a fitness function should be differentiable, at least as far as the solution space. Point discontinuities aren't an issue if they occur outside of the solution space, which is important for us because we're going to make use of one. Specifically, the function

$$F(s) = \frac{1}{|E(s) - x|}$$

Where E(s) is the result returned by evaluating s, the numeric string. So if we had the string 3-6*6+2, then E(s)= -16. Assuming x is still 68, then the fitness for that particular s would be $\frac{68}{84} = 1.19E - 2$, extremely low. When E(s) = x or F(s) = ∞ we break out of the infinite loop. This would be a discontinuity, but the function is still differentiable across where we're evaluating it.

[1] Other genetic alphabets are possible as well, though less common.

[2] We use the hyphens here only for readability- the bitstrings contain only 1s and 0s.

[3] There are examples of x where this won't work as well- in a shortened version of our example problem, for instance, if the target is 25, $2 \times 3 \times 4$ will quickly dominate the fitness landscape but isn't actually any closer to a valid solution than $5 \times 5 + 9$.

Breeding and Survival Now that we have a fitness function, we can evaluate the entire population and determine who has the highest fitness. With a random seed, the first generation is usually not very fit. Regardless, the next step is to see who survives and who breeds into the next generation. Survival is usually an arbitrary matter of copying the best solution(s) *in toto* to the next generation. [4] This is typically referred to as elitism, and it is usually a percentage around 10% of the population that is uncritically copied into the next generation. Properly done, this guarantees monotonically non-decreasing fitness of the best solution from one generation to the next.

The next step is to pick some fraction of the population as breeding stock. The most fit are generally given preferential treatment, but not exclusive preference. This is in large part because in GAs as in life, genetic diversity is a critical trait to the overall fitness of a population. For GAs, it means that diversity speeds convergence to ensure that even the less fit have a chance to propagate into the next generations. The method we employ, a fairly standard one, is Roulette Uniform Selection. To get an intuition for the algorithm, see Figure 1.1. In it, you can see that RUS chooses markers (represented by the arrows) equally spaced between the beginning of the population and the end. Everywhere a marker falls means a copy of that solution gets passed into the breeding population for the next generation. This is usually guaranteed to get at least 1 copy of the most fit individual, but everything else is based on chance.

There is some discussion that suggests that proportional selection provides the weakest selective pressure of several types of selection processes and thus that other methods should be employed or that supplementary approaches be takenBack (1994). While we have found this to be true to some extent selective pressure that is too strong can cause premature convergenceAffenzeller and Wagner (2003), we implemented tournament selection, linear selection, and a Biased Random Key selection scheme

[4] Some variations have ages, where all solutions will live a certain amount of time, and more fit solutions have longer lifespans than less fit solutions. These will not be covered further in this document.

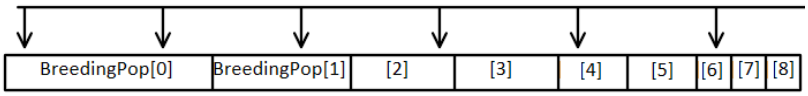

Figure Figure 1.1: An example of RUS. The bar represents the cumulative fitness of the population. The arrows represent which members of the population go on to become a breeding member. In this example, 3 gets skipped even though it has higher fitness than 4 and 6, while 0 gets copied into the breeding population twice.

Ruiz et al. (2015) before settling on RUS because in testing the GA would typically converge to minima prematurely. Thus we are actively choosing to preserve diversity over rapid premature convergence.

With a breeding population in place, we can begin the breeding process. This is typically accomplished via an operation called crossover, which I will explain below.

A	*1001*	*10*	*0011*	*11*	*1001*	*00*	*0111*
B	**0011**	**01**	**1000**	**01**	**0001**	**10**	**1001**

We begin with two solutions. Crossover takes and returns 2 bitstrings. It also has several subtypes, which I will illustrate in sequence. First is one-point crossover. This means that before crossing, a crossover point is chosen and offspring are produced as a copy, and when this point is reached, the offspring cross over and begin taking material from the other parent.

A'	*1001*	*1***1**	**1000**	**01**	**0001**	**10**	**1001**
B'	**0011**	**0***0*	*0011*	*11*	*1001*	*00*	*0111*

Two point crossover is similar to one point, except that crossing happens twice.

A'	*1001*	*10*	*00***00**	**01**	**0001**	**10**	**1***111*
B'	**0011**	**01**	**10***11*	*11*	*1001*	*00*	*0***001**

Notice that here the final digit of A' and the second digit of B' becomes invalid (1111 is not between 0000 and 1001)- this is not controlled for in Crossover, but rather handled later by evaluating the offspring.

The final standard type is uniform crossover, which makes a check at each bit to crossover. It results in much more mixing, depending on the probability of a crossover event. One advantage is that it treats the beginning and end as the same, which can't be said for either of the first two. Another is that it allows the designer to specify, directly and exactly, how much gene mixing should occur on average.

One thing that might be noticed is that regardless of where the crossover point is, with these methods if both A and B contain a particular bit in the same location, it will appear in both offspring. This is one reason that diversity is important- if the population becomes too homogeneous, it will be unable to change except through random mutation, which we discuss shortly.

One other form of crossover we will deal with is one that counters this potential vulnerability. It is a shifted crossover, so that one bitstring shifts forward a random number of bits, and then crossover proceeds normally.

Other forms of crossover are possible, though they are not as widely represented in the literature. One is a modified uniform crossover that checks to cross only at each "word", that is at each column representing a digit or operator in our example. This gives the designer some control over how much contiguous information is exchanged per crossover. Other variations with three or more parents, or even random asexual reproduction are possible. Logical operations are viable, though again care must be taken to not increase homogeneity overmuch.

Mutation Mutation is fairly straightforward. Usually after crossover and before insertion in the next generation, that is, only affecting the offspring of breeding and not elitism, each bit in the bitstring has a chance to change. The algorithm is perhaps the most instructive:

```
1 for (Solution s in Crossover(A,B)):
2       for each bit in s.Bits:
3               if(MutationChance > Random(0,1)) bit = !bit
```

```
        end for
end for
```

There are some implementations that vary in that they will change random values to 1 or 0 rather than flipping bits (in other words, values will change about half as often). A standard value for mutation is small, about $\frac{1.0}{Solution.Length}$ which means on average 1 bit will change per solution per generation.

What is less straightforward are the effects of mutation over a population. While values of 0 often stagnate in local minima, an upper correlate doesn't seem to exist- one could set mutation high, say 25%, and turn off crossover entirely, and proceed with elitism and mutation alone and arrive at solutions. In practice, this is much slower than using crossover. A general rule of thumb is to keep mutation low, and increase it even more slowly to combat stagnation.

Drawbacks While GAs have great versatility, there are some drawbacks which significantly limit their utility.

Swiss-Army Chainsaw First, GAs are not the perfect solution for anything. At their core, they are a biased-walk[5]. This means that while they will come to a local optima, there is no guarantee they will achieve a global optimum. If there exists a tailored solution for a problem, using that will probably work faster and better.

And Quick to Anger Second, GAs are subtle things. The fitness function, not given its due in this writeup, can be the difference between a quickly converging solution and processors spinning their wheels for days or months coming to one sub-satisfactory solution after another. A GA will optimize the fitness function- and that's all it will do. So if you want to maximize a metric, say accuracy for a classifier, be aware that it might do exactly that by simply guessing the most prevalent answer in

[5]With a random-walk on one side of the spectrum and a guided approach on the other.

the dataset. This will get it to a local optima, and it might be surprisingly difficult to get it out of it.

Furthermore, there are hyperparameters which effect the GA directly but can be difficult to tease out. What should the elitism percent be? It probably depends on your other parameters. There are ways of optimizing these, and they themselves might be amenable to a further GA, except that evaluating them is time consuming- should your fitness function be speed of convergence? Or the best solution arrived at within 100 generations? That might seem too long, but use fewer generations and you run the risk of invoking too much sensitivity to starting conditions to draw meaningful conclusions. There are some rules of thumb to assist with these situations, but they only mitigate the problem, they don't eliminate it entirely.

Toolset Finally, a GA is only as good as its toolbox. On Earth, that toolkit was physics. All of physics, and massively, embarrassingly parallel at that. That's difficult to take advantage of digitally, where you not only have the responsibility of developing an encoding but also defining the universe your population lives in. For instance, the heart of this paper is whether to use a GA to optimize a classifier or to use a GA to classify things. The classifier has theory underpinning its toolkit, the GA has only whatever fitness function and encoding it is supplied. Or, to get back to our example, it might speed convergence to increase mutation rates and restrict crossover to occur only at the breakpoints of binary words. The downside of this is that GAs are supposed to be able to solve any problem, and while they can, they also require a certain amount of customization to not waste everyone's time. The harder the problem, the more customization that is usually required. And at that point, if a tailored solution of some kind exists it's probably easier to code up and implement than an equivalent GA. At their core, the GA is a biased walk, but building in paths will hopefully make that walk much quicker.

Theoretic Underpinnings Intuition is often insufficient for or even anathema to scientific inquiry, and so far that is all we have relied upon to understand why GAs work. The Fundamental Theorem of Genetic Algorithms is used to explain much of it. First, where we explicitly represent populations as collections of bitstrings, we may impose an additional ordering on them, the schema. Returning to our example, consider the equations:

$$3 \times 7 \times 3 + 5 = 68$$
$$9 \times 7 + 8 - 3 = 68$$
$$9 \times 7 + 6 \times 2 = 75$$

These translate to the bitstrings

$$0\mathbf{011\text{-}10\text{-}0111\text{-}10} - 0011 - 00 - \mathbf{0}101$$
$$1\mathbf{00\text{1-}10\text{-}0111\text{-}00} - 1000 - 01 - \mathbf{0}011$$
$$1\mathbf{00\text{1-}10\text{-}0111\text{-}00} - 0110 - 10 - \mathbf{0}010$$

Let's just assume that our x is close to, but is not 68. Let's say it's 71. This means that each of these bitstrings will have a relatively high fitness. Specifically, the first two have a fitness of $\frac{1}{|71-68|} = \overline{.3}$ and the third has a fitness of .25. However, a close inspection of all bitstrings side-by-side shows that they have a considerable amount of overlap. All begin with odd numbers multiplied by seven, etc.. You can see the overlapping sections in bold- but pay careful attention to the long contiguous sequence. Schemata are a way of considering a population theoretically. A schema introduces a third character into the alphabet of bitstrings- an * indicating either 1 or 0. Schemata are of any length, and may be defined by their offset and bitstring (their length may be obtained from their bitstring). Thus, offset 0 and "***1-10-0111-**" is a valid schema which describes both solutions. In fact, there is one schema that describes both strings, which is already represented by retaining the symbols where

it is bold and replacing it with * where they don't match.

It should be obvious that doing any calculations with schemata are infeasible, simply because for even short bitstrings, the possible schemata to describe the population increase exponentially. Let m be the length of a bitstring, and S be the set of schemata which can describe the population. Then let

$$|S| = \sum_{i=1}^{m} 3^i(m - i + 1) = \frac{3}{4}(-2m + 3^{m+1} - 3)$$

While we could try to constrain this by only using the schemata that describe the currently existing population, that problem is also exponential, and potentially worse computationally, because any member of the population has

$$\sum_{i=1}^{m} 3(m - i + 1) = \frac{3}{2}m(m + 1)$$

schemata that describe it, but then these would need to be compared with the sets generated by all 2^n combinations of members of the population.

However, feasibility of computation aside, we can use this to gain a further and more formal understanding of how GAs function.

Informally, if we assume that schemata describe our population, whatever they may be specifically, we may also assume that short, more fit schemata will have a greater than average fitness than is represented in the population, and as such these short, fit schemata are likely to increase in representation throughout the population. Shorter schemata will survive because longer schemata are more likely to be broken up by crossover. Three more concepts need to be understood before the equation itself: First, is the order of a schema H, which is the number of non-wildcard bits it contains. Higher is more specific. Second is the defining length, which is the distance between the first and last non-wildcard bits. If we return to our example, and let H_a= offset 0, "***1-10-0111" and H_b= offset 0, "*0*1-10-**11", then $o(H_a) = \delta(H_a) = 7$, while $o(H_b) = 6$ and $\delta(H_b) = 8$. Third is $f(H)$, which describes the average fitness of a

schema. This is defined as

$$f(H) = \sum_{s \epsilon H(P)} \frac{F(s)}{|H(P)|}$$

where H(P) is the schema H applied to the population P, which returns a subpopulation of solutions, and where s is one such solution returned.

With these concepts understood, the Fundamental Theorem of Genetic Algorithms is as follows:

$$r(H, t+1) \geq r(H, t) \frac{f(H)}{\overline{F_t}} \left[1 - \left(p_c + o(H) p_m \right) \right] \tag{1.1}$$

Where $\overline{F_t}$ is the average fitness of the population at time t. p_c and p_m are probabilities of crossover and mutation, respectively, and $r(H, t)$ is the number of representations of a schemata H at a time step t in a population. The type of crossover plays a role, here. Depending on implementation, in single point crossover the probability of crossover occurring is usually 1. Instead, a random number from 1 to L is usually chosen, with each index being equally likely, and crossover occurs at that index. Thus, for single point crossover,

$$p_c^{SinglePoint} = \frac{\delta(H)}{L-1}$$

For two point crossover, the odds of breaking a given schema is much more likely, because it is the outcome of 2 events not happening, that is

$$p_c^{TwoPoint} = 1 - \frac{L - \delta(H)}{L-1} \frac{L - \delta(H)}{L-2}$$

from which we get the general

$$p_c^{NPoint} = 1 - \frac{(L - \delta(H))^n}{\frac{(L-1)!}{(L-1-n)!}}$$

for n point crossover.

Uniform crossover is implemented differently, and is generally done with a rate, which

we'll call γ. Since this is the case, it makes calculating p_c more straightforward.

$$p_c^{Uniform} = 1 - \gamma^{o(H)}$$

So, while representations for schema which are more fit than others will increase over time, the particular type of crossover can have a significant impact on how much representation they gain. It is worth noting that a uniform crossover with even a moderate rate of crossover (say, .3) can rapidly lead to the eradication of all higher-order schemata. Also worth noting is that when a schema applies to both solutions, both offspring will also belong to that schema using standard crossover methods. Finally, if an implementation includes a probability other than 1 for any of the forms of n point crossover, that probability is simply multiplied to the appropriate p_c.
A few observations about this function. As overall length of the bitstring increases, the inhibition of single-point crossover decreases, but inhibition of n point crossover generally increase, and if the mutation rate is linked inversely to length then it does as well. Uniform crossover is less directly linked to length, though not independent: as L increases, the numbers of higher order schemata increase exponentially. Survival of a particular schema becomes very unlikely without a strong selective pressure toward retention. Further, high order and fitness schemata with greater defining lengths rapidly become unlikely to obtain except through elitism.

Motivations Over the course of researching how GAs were being used in recent times a broad pattern emerged. With regards to classification, there were two basic schools of thought: one in which GAs were used to optimize classifiers and one which used GAs as classifiers directly. Both camps have *prima facie* impressive examples of these approaches. Thus, this project was born- to attempt to answer which approach is more generally applicable.
GA are widely used for numerous purposes, but we seek to ascertain their suitability

for pattern recognition, specifically classification. In this paper, we attempt to determine how suitable GAs are for the task of classification. To determine credibility, we propose 3 tests. First, we will run a GA on the datasets acting directly as a classifier. Second, we will run two classifiers, a lone Decision Tree, and a Naive Bayes classifier with default settings as a control. Finally, we will run the same classifiers optimized through a GA which will run for a modest number of generations. We will collect confusion matrices from the resulting classifiers and compare them.

To ensure that our approach is generally applicable, we propose datasets from divergent fields of study and with data in different configurations. With minimal adjustments, the program we have written is capable of automating this process over almost any dataset, but we have chosen 3 from the UCI repositoryLichman (2013) : Yeast Dataset (Yeast)Paul Horton (1996), Bach's Chorales Dataset (Bach)Daniele P. Radicioni and Roberto Esposito (2014), and Cardiotocography Dataset (Cardio)J. P. Marques de S et al. (2010). This gives us 4 different configurations to use, as Cardio has two modes that it runs in, with the pattern class code (1-10) or the fetal class code (Normal, Suspect, Pathologic). This gives us data from biology, music, and medicine, which serves as a broad start. However, we are also making the code available in in its entirety for anyone who wants to extend this to other domains of study.

Priors Going into this project, a case for either side could be made. On the case for the Pure GA approach you have a few points. First and foremost versatility- the GA is only limited by the encoding, and a clever encoding could exploit nuances that people wouldn't be likely to notice in the data. Secondly, there's something about the simplicity of only having 1 component to debug- this could speed development which would mean more effort could be spent developing the encoding and the fitness function. One con is the inverse of the first pro- that encoding needs to exploit the nature of the data, and so much time can be spent customizing the approach to the data, but this is also antithetical to the idea of a generic solution to classification, so

whatever encoding we use must be good enough to apply to many problems.

For the Hybrid camp, the pros are primarily that the GA gets to leverage the theoretical robustness of the external classifier. That is to say that there has been a great deal of work to make classifiers extremely good at what they do- and writing a GA to compete with or exceed beyond that work is difficult. There are several cons, though- the external classifier needs to be written and debugged separately and then in conjunction. Also, there are now 2 or more entities to be maintained which means much less time can be spent on any one project, unless you simply tie into some other classification package, though that may have its own challenges.

Thus, before going into this project, we slightly favored the hybrid approach. But we wanted to make sure that there was plenty of versatility for the GA to make use of, so for the implementation of our pure GA we settled on CellNet Kharma et al. (2004), which is a variable length GA which has a new breeding operator. We discuss this project in depth in Chapter 2.

Literature Review

Hybrid Approach In Schuman et al. (2014) they use classifiers to breed numerous neural networks in parallel and test them against MNIST. In Ocak (2013) the authors use a support vector machine (SVM) optimized by a GA to predict fetal well-being, with considerable success. In Marchetti et al. (2013) GAs were used for feature selection and then the features generated were passed to a logistic regression classifier. In Wu et al. (2015) GA were used in conjunction with particle swarms to optimize a neural network for predicting rainfall. While GAs performed better in conjunction with the particle swarms than alone, the idea of using a GA to build a better classifier is present. Chou et al. (2014) and Duan et al. (2014) discuss using a GA to optimize another SVM, this time emphasizing the mileage obtained from leveraging the SVM for curve fitting while the GA handles the optimization of the SVM itself. Devos et al. (2014) takes a similar approach, but instead focuses

on using the GA to determine which combination of preprocessing methods to use. The GA is also used to determine two meta parameters for the SVM itself. Once these are determined, the SVM is used to handle the classification. In Uysal and Gunal (2014) they use GAs to focus their Latent Semantic Indexing approach on promising semantic features. They use two different approaches and find that both are much more effective on a wide range of tests than the approach without the GA optimization. Salari et al. (2014) used an ensemble approach which is quite novel. First, they use doctors to decide which features of the datasets in question to use. Then, they give these features to a Feed Forward Neural Net(FFNN) on one hand and a GA on the other. The GA generates several different arrays of features, and then these arrays and the results from the FFNN are fed to a k nearest neighbor to find fuzzy classes, then those classes are iteratively pared down until they are no longer fuzzy. They apply their model to multiple datasets, and compare across a wide variety of methods with a variety of metrics, over which they show statistically significant gains on nearly every method, metric, and dataset. Alexandre et al. (2015) hybridizes a GA with an Extreme Learning Machine, with considerable gains in both binary and multiple class classification over multiple learning methods.

Purist Here, we look at papers where GAs act directly as the classifier. We have included papers where rules for classification are generated. Dehuri et al. (2008) uses two different GA approaches to classification rule generation, one a simple one similar to the "canonical" approach and one a multi-objective optimizer. The multi-objective GA performed well, though areas for improvement were discussed in the conclusion. One area particularly limiting the GA was the number of attributes the GA was working with. The interestingness of the discovered rules was quite high, though sometimes the comprehensibility was lacking. Still, the rules were highly predictive on the datasets employed. In Kozeny (2015) they used 3 GAs to calculate credit scores, and compared their accuracies. While they achieved interesting results with their methods, the most promising one scales a $O(2^L)$, which makes it unfeasible

except for short feature sets. In Srikanth et al. (1995) variable length GAs are used to draw fuzzy ellipses in a decision space, and thresholding is used to make the borders crisp. This approach is shown to be comparable over their selected dataset to a back-propagation neural net (BPNN).

Fidelis et al. (2000) uses a GA to develop rules for diagnosing breast cancer and dermatology. They achieve 95% accuracy on the dermatology but only 67% on the breast cancer- which is, by the author's own admission, a much more difficult dataset with a great deal more noise. Beside the results, the rules themselves were of interest, and evolved 3 separate times from different random seeds. The rules themselves seemed to hit the knowledge discovery trifecta of predictiveness, comprehensibility, and interestingness.

In Hoque et al. (2012) and Li et al. (2012), GAs are used to solve an intrusion detection problem. While both papers boasts a discovery rate of nearly 100%, if one ignores the denial of service attacks their discovery plummets. Unfortunately, it isn't clear that the authors are entirely at fault for this- the data used for the DARPA challenge contained 5,283 non-DoS attacks, but 391,000 DoS attacks (it also contained 97,000 non-attacks). Its small wonder that the GAs would optimize according to this aspect of the data. Tseng et al. (2008) is another example of rule discovery, but this time applied to land-cover classification. They specifically use a GA as an alternative to other methods such as a Neural Net(NN) or other Bayesian classifiers because the GA has shown that the rules it discovers are more comprehensible, even if less predictive. In other words, while NNs may be good at solving a problem, exactly how they solve it is not always clear, and that can be problematic for many domains. Unfortunately, while they get good accuracy they don't do a head-to-head comparison. One possible reason is because the database is very small, at only about 400 samples.

Finally, and we will discuss the inspired CellNet family of papers in much more detail in Chapter 2. However, in the original Kharma et al. (2004) CellNET paper, they used a variable length GA to classify the ced (2002) database. While they obtained modest results, the stated goal of CellNet was to develop a GA which could

approach any dataset with minimal intervention. They continued in Kowaliw et al. (2004), where they employed two populations of competitive solutions called hunters and prey, where hunters are trying to classify a handwritten digit from the CEDAR database and prey are trying to obscure said image. The results in this paper were much more impressive, and their overfitting problem diminished significantly as well.

Chapter 2

Methods and Implementations

2.1 Overview

In this chapter we'll discuss the design decisions we've made in detail. We have 3 distinct phases. The first phase, preprocessing, is converting the data files into a unified dataset in memory and configuring for the following phases. The second phase is the hybrid approach where we optimize external classifiers, followed by the purist approach where we use our genetic algorithm (GA) to develop classification rules. Each of these approaches generate confusion matrices which we analyze.

The general flow of the algorithm can be seen in Figure 2.1. Loading and normalization gets all algorithms on the same page, ensuring an apples-to-apples comparison. First we have the hybrid approaches. Within a hybrid approach, we first see what optimizing without including features yields- this is referred to in the code and in the program as the hybrid approach. It has all the same constraints as the parent approach. After the baseline, we optimize a second time, this time including feature selection, by which we mean that we characterize a dataset with n variables as either included (1) or excluded (0) in a bitstring, and encode the same variables to the classifier as in the baseline method. We have selected two distinct classifiers: a multiclass naïve Bayes algorithm and a simple single decision tree. Each of these are

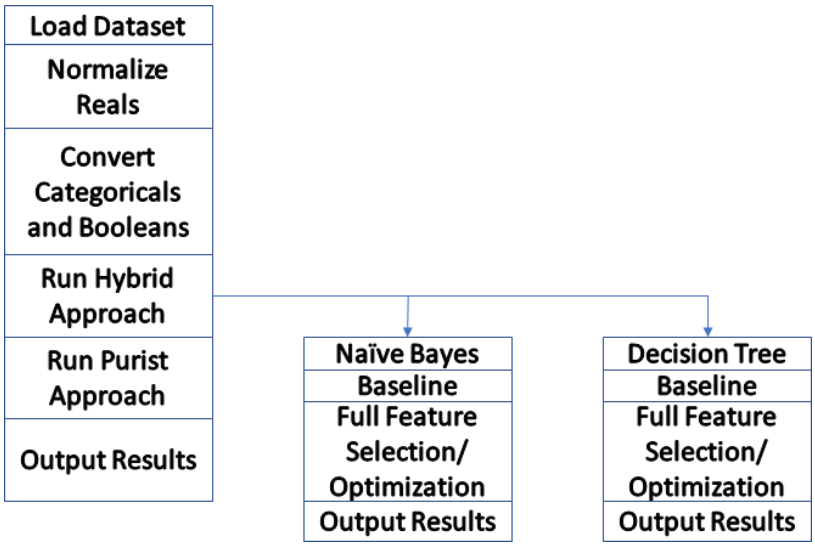

Figure Figure 2.1: Birds-eye view of the flow of the program. Hybrid approaches are not run in parallel, because at time of coding MATLAB doesn't support multi-threading via a COM server.

run twice per dataset- once with feature selection disabled (the baseline) and once with it enabled. We will discuss those methods in more detail in their respective sections.

The purist approach is much more straightforward. There is only one mode which it runs in, there's no explicit feature selection. That is, there is feature selection, but all features are available to the algorithm at any time- just some may or may not be included. This portion of the algorithm is threaded.

2.2 Preprocessing

We do some preprocessing of the data. First and foremost, there is a text file that is read which describes the dataset and points to where it is on the filesystem. Below is an example- this tells the program where it can find X and Y and how to parse them.

```
#Dataset Name
CardioData
#Class Names File
../../../Data/Cardio/classNames.txt
#TrainingSet X Path
../../../Data/Cardio/trainingXY.csv
#TrainingSet Y Path
../../../Data/Cardio/trainingXY.csv
#TestingSet X Path
../../../Data/Cardio/testingXY.csv
#TestingSet Y Path
../../../Data/Cardio/testingXY.csv
#X ignore list, comma separated and starting with w if it's a whitelist
(otherwise, blacklist)
b, 29, 30
#Y ignore list, as above
w, 29
#Categorical Variables, white/blacklist, comma separated
w,
#Boolean Variables, as above
w,
```

For instance, in this case X(the data) and Y(the labels) are in the same file, but represented as different columns. They could easily be stored as two files. After those files are explained, the next uncommented line describes the columns to ignore or include, specified with b for blacklist or w for whitelist respectively. In this example, columns 29 and 30 are ignored for X, but only 29 is included for Y. This is because for this dataset, there are two sets of labels and thus two separate description files to use the file differently. Using the same notation, we can declare categorical and boolean variables for X. Y is assumed to be categorical.

After we read how to parse the file, we read the files themselves. At this point we also convert booleans and categorical variables to doubles so that they can fit in the same matrix. First, however, we want to normalize the Reals. First we gather max, min and mean from each column in the dataset. Then we use a function to squeeze the values down between .1 (b) and .9 (t) for reasons that will be seen later in the section on the purist approach.

$$x_i' = b + (b - t)\frac{(x^i - x_i^{min})}{x_i^{max} - x_i^{min}}$$

Next, categorical variables are given the treatment motivated and described fully in Zhang et al. (2015), the conclusion being that every category label is replaced with a real number which maximizes Pearson's Correlation Coefficient. That is,

$$\begin{aligned} X &= X_{\mathbb{R}} \cup X_{cat} \cup X_{bool} \\ X_{\mathbb{R}} &= \{x_1, x_2, ...x_n\} \\ X_{cat} &= \{x_1, x_2, ...x_c\} \\ X_{bool} &= \{x_1, x_2, ...x_b\} \\ 1 &\leq i \leq c \\ L &= \{x_i | x_i \epsilon X_{cat}\} \\ C_l &= \{x, i | x \epsilon X \wedge x_{n+i} = l \epsilon L\} \end{aligned}$$

Where X is the dataset, $X_{\mathbb{R}}$ is the real portion of the dataset, andX_{cat} and X_{bool} are the categorical and boolean portions. The index i iterates over the columns of X_{cat}, which lets us derive L, which is the set of all categories in the dataset. L in turn gives us a means of devising C_l, that is, the subset of X which consists of all members of x which belong to the category l. Now it is possible to look at C_l and determine which

values will maximize r^2 for each label l, which we here call R(l).

$$R(l) = \frac{\sum_{x \epsilon C_l} \sum_{j=1}^{n} x_j}{|C_l| n} \tag{2.1}$$

It can be seen as the mean of all the other values over C_l. Calculating this in practice is much more straightforward- simply step through X one sample at a time, and maintain running sums and counts for each unique label, calculate the means at the end and then extend $X_{\mathbb{R}}$ with the newly calculated values. In the case of multiple categorical columns, unless there is a perfect correlation between two category labels each label will have its own value (though this can't be proven to be unique, only generated from a unique set of numbers).

Booleans are only given the treatment of being converted to values .25 for false and .75 for true. Again, the reasons we don't use 0 and 1.0 will be made clear in the section on our purist approach.

Now that the data has been modified, some additional bookkeeping is accomplished. Conversions to numerics from string class labels and vice versa are computed and stored, since mathematical approaches prefer integers while human-readable outputs are in the terms given us by the dataset. Also, global values such as Elitism Percentage and Population Size are modified at this point and are effectively constant for the rest of the program. Variables that may be configured by the user are: Max Generations, Record Interval, Population Size, and Complexity bounds. The Max Generations sets the stopping condition of the algorithms, and defaults to 100. The Record Interval determines how often to save data to the disk- data is collected every generation, but is only saved to disk in where $Gmod(RI) = 0$, and the default is 25. Complexity bounds are discussed in more detail in the purist approach, and have no effect on the hybrid approach. Other modifications to semi-constants are made at this point determining the length of chromosomes for both approaches based on the characteristics of the datasets- particularly the number of features.

2.3 Hybrid Approach

As mentioned previously, the hybrid approach consists of 2 methods which each consist of two different ways of running them. First we will discuss the multi-class naïve Bayes (McNB) approach, followed by the decision tree (DTree) approach.

McNB Naïve Bayes is one of the most basic of classifiers, but it is powerful and versatile. Without going into extreme detail, it generates probability distributions for each class across every dimension in the feature set from the training data, and picks the most likely class for a given sample point. Typically, the probability distributions are Gaussian, however any density function can be used. We use this because it should provide a fairly low bar to compete with- support vector machines (SVMs) are much more complex, tend to be extremely reliable and robust, and are close to something resembling the industry standard, but don't perform well with multiple classes. McNB is closer to statistical modeling and is not what we consider machine learning, though no bright line distinction exists. It is a theoretically grounded but simple statistical method well suited to being a first pass at the data or being used in conjunction with other methods. Being simple, it is also relatively fast- only 2 passes through the data are necessary to build the parameters for the PDFs, so building the model can be done in linear time. Once built, checking a given variable can be done in constant time.

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)} \tag{2.2}$$

The standard formulation of Bayes' theorem, in our case it is more useful to form it thus:

$$P(\omega_j|x) = \frac{p(x|\omega_j)P(\omega_j)}{p(x)} \tag{2.3}$$

Here, ω_j is the likelihood of belonging to a particular class. Upper case Ps are simple probabilities, lower case ps are more complex functions. So $p(x|\omega_j)$ is the PDF, and $P(\omega_j)$ is the prior, which can be thought of as *a priori* how likely a particular class is

to present. To build the PDF, if we're using a Gaussian, we need mean and standard deviation for each class. The Gaussian PDF is built using the training set, tested on the testing set, and the results are scored in a confusion matrix. This can be determined from the dataset, in which case it uses the frequency in the Training set to generate priors, or set manually. In either case, this can be seen to scale a particular PDF. In fact, this is similar to what $p(x)$ does- except that where $P(\omega_j)$ scales a class, $p(x)$ scales all classes, and it serves as a normalizing factor to constrain values between 0 and 1. It could be established using the law of total probability, or it could be some arbitrarily high constant[1].

Our implementation, which we'll refer to as the McNB optimizer[2] optimizes the fitcnb [3]

The distribution uses 2 bits and can be any of the following values:

- Kernel uses a smoothing function, described below.
- Multinomial represents every class as a single multinomial distribution.
- Multivariate multinomial characterizes each feature as an independent multinomial distribution based on the unique values found in the feature.
- Normal distributions behave as described above.

Kernel type uses two bits, though these go unused unless the the distribution is kernel. Then the variables are smoothed via various functions outlined below.

[1]It doesn't really make a difference, since the selected class will just be the one with the highest score given x. Furthermore, if it affects all classes equally nothing is served by making complex calculations every time the classifier is called. Simply calculate the highest value it could take initially and use that in subsequent calls.

[2]To avoid confusion, our optimizers are optimizing classifiers, they are not classifiers but optimizers. Later, we present our classifier.

[3]For much further detail on Mathworks' implementation see http://www.mathworks.com/help/stats/fitcnb.html. function in MATLAB over the following parameters: distribution, kernel, score transform, and priors. Further, it optimizes over the dataset by choosing which features are included. It is entirely defined by its bitstring, the first several bits correspond one-to-one to the features in the dataset. An optimizer with all 1s (or all 0s to avoid having no data) will include the entire dataset. Otherwise, a 1 indicates that the column is included, a 0 removed.

- **Box** uses a uniform, box-like smoothing window.
- **Epanechnikov** is a very efficient, rounded kernel. Minimizes Asymptotic Mean Integrated Square Error (AMISE)Stefanie Scheid (2004) therefore optimal in that sense.
- **Gaussian** is a standard normal function but used in this case for smoothing.
- **Triangular** is another form of smoothing, with a peak of 1 at 0 and zero at -1 and 1.

Regardless of which form of smoothing, the goal is the same- to create a distribution of a random variable which can then be modeled. This is done using something like a histogram, which is then smoothed into a continuous function using the kernel chosen above. This becomes $p(x|\omega_j)$ in the equation 2.3.

For priors, each class extends our optimizer's bit length by 3 bits. Each class then has a prior of $(1+0-8) = 1-9$ which is later summed and turned into a probability distribution summing to unity. For instance, if there are 2 classes and one has a prior of 3 while the other has a prior of 7, these are converted into percentages of 30% and 70%, respectively. These become the $P(\omega_j)$ in equation 2.3.

Finally, the score transform takes up 3 bits and can be any of eight values. This is used internally in the MATLAB function. It can be any of the following values:

- DoubleLogit transforms the score to $\frac{1}{1+e^{-2x}}$
- Invlogit $log(\frac{x}{1-x})$
- Logit $\frac{1}{1+e^{-x}}$
- None x
- Sign $\frac{x}{|x|}$, or 0 when x = 0.
- Symmetric $2x-1$

- Symmetricismax 1 if max of class, 0 otherwise
- Symmetriclogit $\frac{2}{1+e^{-x}} - 1$

Once these are determined, the optimizer is evaluated. This means that the training set is passed to the optimizer, it is trained, and then the testing set is passed to it, from which we get the fitness of the classifier. Fitness is average accuracy. That is, suppose we have a confusion matrix C

C=

Predict:	ω_1	ω_2	ω_3	**Total**
True ω_1	4	2	8	**14**
True ω_2	2	2	6	**10**
True ω_3	5	5	110	**120**
Total	**11**	**9**	**124**	**144**

The dataset in this case has 3 classes, ω_1, ω_2, and ω_3, and has 144 total samples. Of those, 14 are class ω_1, 10 are class ω_2, and 120 are class ω_3. Meanwhile, the classifier predicted that 11 of the samples were ω_1, 9 were ω_2, and 124 were ω_3. To determine accuracy, the equation is $\frac{\sum_{i=1}^{3} C_{ii}}{144}$. This is a good metric for relatively evenly distributed classes. However, in highly skewed cases as this one, this treats more rare classes as less important. For instance, in this case, the accuracy is $\frac{116}{144} = .805$, which might seem like it is doing a decent job, and maybe it is, if the concern is finding Cs. Average accuracy is slightly more complicated to calculate, but still straightforward enough. First, lets define the Total column more formally:

$$T(\omega_i) = \sum_{j=1}^{N} C_{ij}$$

Where N is the number of classes. Thus, average accuracy, $\overline{A}$ can be defined as

$$\overline{A} = \frac{\sum_{i=1}^{N} \frac{C_{ii}}{T(\omega_i)}}{N}$$

In our example, $\overline{A}$ is .467, which seems more like the classifier is doing barely better than chance. In fact, if it had just guessed everything was ω_3, accuracy would be higher (.833), but $\overline{A}$ would have been .333. Further, $\overline{A}$ is equivalent to accuracy if all classes are equally distributed, thus there is no disadvantage to using it. Average accuracy is a major factor to the fitness functions used in this project.
We will next discuss another type of optimizer, and then we will discuss what they have in common.

CTree Decision Trees are an algorithm which take a dataset and make usually boolean decisions from its features, which generates a hierarchy resembling a tree. They are very easy to compute, but usually aren't the most robust of classifier unless used in ensembles. However, finding an optimum decision tree has been shown to be NP complete Hyafil and Rivest (1976), so greedy approaches are often used to approximate the perfect tree. Decision trees are referred to as Classifier Trees or Regression Trees, depending on their task.
Decision trees are typically generated using a greedy algorithm to maximize the split criterion at each of several splits. The Gini impurity is $1 - \sum_i p^2(i)$, where $p(i)$ is the fraction of samples belonging to ω_i which reach the node. It is distinct from the Bayesian prior because it doesn't apply equally to all samples. For instance, there might be 10 classes in a dataset, but if only one class would reach a node, then the sum of $p(i)$ would equal 1, and the Gini impurity would be 0. Thus, it can also be seen as the probability of *misclassifying* a sample based on the distribution at that node. Gini impurity is maximized as every decision, insuring that nodes are diverse. There is often a limit to how deep the tree can get, that is the maximum number of decisions can be made for any given sample. Gini impurity is closely related to entropy, and some decision trees are implemented using information gain instead- however, the difference in output is minimal, while Gini is marginally easier to compute. For completeness, entropy of a node may be defined as $E = -\sum_i p(i)log(p(i))$ and it is possible to use entropy gain as the split criterion. A third criterion is twoing, which

is quite different. It tries to find a division of samples whose class makeups are as homogeneous as possible and also make close to 50% of the samples at the node the node, and then it tries to find a split to make that grouping possible. For instance, if 4 classes were at a node, and two of the classes made up 50% of the samples at the node, the algorithm would try to find a split that would maximally separate those two classes from the others.

The next optimizer we'll discuss is the Classifier Tree (CTree) optimizer. This is optimizing MATLAB's fitctree[4] function over the following fields: Merge Leaves, Maximum Splits, Min Leaf Size and Split Criterion. Each CTree optimizer is completely defined by their bitstring, which are typically much shorter than for McNB. It too begins with a representation of the features in the dataset, 1s for included, 0s for excluded, and either all 1s or zeros mean the entire dataset is included.

- **Merge Leaves** takes 1 bit and is either on or off. Merge leaves looks at leaves from a parent node and if the amount of their risk (a term which we believe corresponds to Gini impurity, but we can't find sources backing that up) and that of their offspring is at or greater than that of a parent. In our CTree optimizers, this takes up one bit of the bitstring.

- **Maximum Splits** defines how many splits a tree can have. The tree is built iteratively, layer by layer, splitting as needed until it hits this number. In our optimizers, 6 bits are reserved for it, yielding values of 3-66.

- **Min Leaf Size** This is the minimum number of samples that need to reach this node to be considered a standalone leaf. Beyond this number (specifically, at twice this number) a leaf become a parent node split into two children. Five bits are reserved for it, yielding values between 1 and 32.

- **Split Criterion** can take on 3 values. Gini's diversity index as discussed above, twoing, and deviance. When deviance is selected, the rule is maximally reducing

[4]See https://www.mathworks.com/help/stats/fitctree.html for examples and further details.

deviance with every split (effectively using entropy rather than impurity). With Twoing, it will try to make an optimally balanced tree, erring toward balance rather than composition (in practice, these tend to be similar to entropy based trees). These take up 2 bits of the CTree optimizer's bitstring.

Optimizers Our optimizers use inheritance to share common code. So both CTree and McNB use many of the same mechanisms when it comes to evaluation and evolution. First, both of them use MATLAB as their engine. Unfortunately, while support is planned for a future release, evaluations done through the COM server (as opposed to done through the MATLAB SDK and compiler) do not support multi-threading, so even if multiple threads were used in the native code, there would be no performance gains to speak of. In fact, while we don't have metrics any longer, execution was considerably slower when we were using the multi-threaded model. We did at one point make use of the MATLAB SDK to compile the MATLAB code into native, and this was indeed faster- but there's considerable overhead involved and unless you have a MATLAB educational license, considerable cost. Thus we have opted for the sub-optimal but considerably more cost effective implementation.

There are many commonalities. For instance, initialization of a new Optimizer is really only dependent on the length of that class. The core mechanism is the same in both of these classes (and several others that we have implemented outside the scope of this thesis). Also, once an Optimizer is initialized, there may be errors. While the concrete classes can handle the details, in the abstract the general principle holds that once an Optimizer is initialized, it needs to be prepared for evaluation. For instance, if CTree's split criterion is 3, when only 0-2 are allowed, then those bits need to be rerolled. The rerolling itself is common Optimizer code as are several utility functions, such as logical operators between two Optimizers.

Scoring binary or multi-class Optimizers, saving them, much of the file IO, most of the life-cycle of an Optimizer, and memory management is handled in the common Optimizer code, meaning that implementing a new type of Optimizer can take a

tiny fraction of the time implementing a new optimizer can- and even the MATLAB dependence is optional. The relevant code is actually in the Globals file and the Eval functions in the concrete subclasses. Optimizer does provide a virtual convenience method of ComEvalFunc which handles common cases, but there's no requirement for that to be called in a subclass. The upshot is that this could be used with an entirely different sort of function- it wouldn't necessarily need to be a classifier at all[5]. It could easily be sub-classed and modified to optimize different criteria altogether. The other heavy lifter in terms of inheritance is done by Evolver.

Evolver Evolver is the class which makes up the bulk of the evolutionary algorithm. Where Optimizer provides a framework for the finer details, Evolver handles the broad strokes of evolution. It maintains a population of Optimizers and handles their life cycles in a way that is both extensible and straightforward.

This is in large part due to the fact that the evolution is entirely class-independent, implemented using templates. While it requires that the class being optimized is a subclass of Optimizer, that's really the only requirement; simply sub-classing Evolver and adding a new interface would allow dramatic changes to the function being optimized.

First, let's look at our modified algorithm, as this will provide more details.

```
1 AdvanceGeneration():
2         #P is the population and a class variable
3         EvaluateAllOptimizers(P)
4         GetMetrics()
5         P.ReverseSort()
6         P = GenerateNextGeneration(P)
7         RemoveDuplicates(P)
8
9 GenerateNextGeneration(P):
```

[5]We plan on using this framework to experiment with integer Linear Programming in the future, for instance.

```
10          BreedingPop := StochasticRUS(P)
11          NextGen := Elitism(P)
12          FillListFromBreedingPop(NextGen, BreedingPop, P.Count, UniformXOver)
13          MutateNonElites(NextGen)
14          return NextGen
15
16  FillListFromBreedingPop(N, B, size, Func):
17          E := B.Count * ElitePercent
18          while(N.Count < size):
19                  k := j := RNG.Next(0, E)
20                  while(j==k)
21                          k = RNG.Next(0, B.Count)
22                  for each offspring in Func(B[j], B[k])
23                          N.Add(offspring)
24
25          while (N.Count > size)
26                  N.pop()
```

So lets examine the high-points of these algorithms. While EvaluateAllOptimizers might seem straightforward, in the details of that function we can either use multi-threading or not (we do not if we are using the COM interface to MATLAB), and we maintain a hash table of all tested Optimizers and their fitness, along with secondary characteristics if those are important. This is from an earlier instantiation of our system where evaluation was extremely costly and so the extra overhead was worth it to save cycles. This is only possible because performance is entirely defined by the bitstring, this lets us test each string once and never test it again. Finally, in FillListFromBreedingPop, Func is UniformCrossover, but that's just one of several modules supplied. In fact, as the signature is written[6], it could easily take any sort of

[6]The signature is Optimizer[] BreedingFunction(Optimizer A, Optimizer B)- however, there is also a related function template provided which takes a variable number of parents.

two-parent breeding function, with any number of offspring. Three or more parents could also be implemented if that was desired, though this would require sub-classing. RemoveDuplicates is also important- any Optimizers that are removed are replaced with randomly generated new ones whose bitstrings are checked to be unique before replacement is finished, so the population size is maintained. The reason this is important is because the breeding selection we're using in FillListFromBreedingPop is a variation of the Biased Random Key model Ruiz et al. (2015) which provides great selective pressure but makes duplicates more likely as the elites become more homogeneous. It is a modification because the model in the paper guarantees that every offspring will have at least 1 elite parent. Our implementation doesn't, because we're drawing both parents from the breeding population and because of RUS there's no guarantee that any Optimizer other than the one at B[0] will be elite. Instead, they are very likely to be elite.

The metrics we capture in GetMetrics are simply best and average fitness, but this provides an entry point to capture population metrics in a subclass or modification of the code. In GenerateNextGeneration, we use RUS and Elitism as discussed in Chapter 1. We also mutate the non-elites after our next generation has been determined.

OptimizerProgram There's only one more major component to the hybrid approach, and that is the OptimizerProgram class, which handles hyperparameters to Evolver. How often to write data to disk, whether or not to multi-thread, population size, how many generations to run and file IO, as well as insuring all the directories for file IO exist are among the duties handled in this class. Incidentally, we employ what we refer to as a baseline mode, which means running Evolver two different ways. The baseline method runs Evolver with all columns turned on- in other words, it restricts the evolutionary process from functioning as a feature selector and only optimizes parameters to the classifier itself. Then it runs again, this time without the restriction. This provides a baseline comparison to see how much performance

changed when feature selection is included in the optimization. There are of course many, many more details in the code itself, which is available in the appendix, and comments provide motivation for much of the detail in situ. This writeup should cover the high points, however, and give a reader an idea of what they are looking at in the code.

2.4 Purist Approach

In this section, we will discuss the implementation of a pure approach to classification. Classification is executed by the evolutionary algorithm itself. To understand all the parts working together, a bottom-up approach is instructive. However, to guide that discussion, let us begin by motivating the algorithm. For an evolutionary algorithm, we need a population of solutions to evolve. In this case, we borrow the terminology from Kharma et al. (2004) and say the population comprises Hunters, which are the form our solution will take. These hunters each have one or more chromosomes, which each have one or more cells. Now we shall look in each component in detail.

Cells The cells are the fundamental building block of the hunter. Each cell codes for a function, an upper and a lower limit, and a not flag. Each cell has the ability to vote on a sample, which is an array of some number of doubles, each of which must be $\epsilon[0, 1)$. When a cell votes on a sample, it is equivalent to saying, "feature f is [not] between lower limit and upper limit". Where not is the not flag, and feature f is a particular entry in the sample. Lower and upper limits are each binary encoded reals, which use 8 bits each. The encoding is straightforward- the bits have all been shifted so that rather than the least significant bit being the 2^0 power, it is the 2^{-9}, allowing the most significant bit to be the 2^{-1}, giving each limit the ability to code for 0 to $\frac{511}{512} = 1 - 2^{-9} \approxeq .998$. This is why in preprocessing we squeeze values down so that they are attainable by Cells. The limits take 8 bits each, then the not flag takes another, and the feature bits take $\lceil log_2(Features) \rceil$ bits

There is some error checking here, the lower limit bits cannot be greater than the upper limit bits, and so they'll be swapped if that occurs. Also, the feature bits, unless there are a power of 2 features in the dataset will have illegal values. If these occur, all of the feature bits are rerolled randomly until compliant.

Finally, there is also a join bit whose purpose is simply whether or not to include the next cell in the vote- if the join bit is false, even if there are other cells to vote, then voting ends. This illustrates a breach in the chain, and is analogous to a form of gene regulation. More importantly, it allows genetic information to accumulate without affecting a particular Hunter's vote. This sort of functionality, that is the ability to turn off a gene while it mutates and changes, has been shown in Zhang (2003) to be a critical component in gene duplication's role in increasing informational complexity- we hope to take advantage of a very powerful evolutionary mechanism by providing a means of doing so.

One other thing worth noting- as written, cells functions are one-to-one mappings as an index to a sample, but this doesn't need to be the case. If a function can take an array of doubles and return a sensible value between 0 and 1, then it would fit into this piece of the puzzle. Most obviously, neural networks can fit this criteria- it would be possible to, say, use an auto-encoder for feature extraction to get down to a certain number of features, and then use the feature index to extract one of those. This, however, would require a somewhat less generic approach than our thesis requires, as neural nets require extensive training and most of the datasets we're using are far too small. Coupling a simple neural net or two might to this algorithm might be one means of significantly increasing performance.

Chromosome Next, we have the chromosome, which is simply a sequence of one or more cells, with a few bits added. First are the class bits, which make up the first $\lceil log_2(Classes) \rceil$ bits, and these mean that votes from cells in chromosome count toward that class. Next are 2 affinity bits, which we will discuss in detail in Merger.

In brief, they describe how the chromosome will behave with other chromosomes. Finally, a not flag, which inverts the votes of their cells.
Cells vote in sequence. Each vote is logically ANDed with the next. If at any point a vote is false, voting stops and false is returned. The Chromosome then takes this vote and passes it up to the Hunter which called it after inverting it if the not flag so dictates.

Hunter At the highest level of the critter hierarchy is the Hunter. Here, there are no additional bits, they simply aggregate the votes of their on or more chromosomes. We can now discuss explicitly the voting process.

```
1  Hunter.Vote(Sample x):
2          Counts[] := new Int[Classes]
3          for each Chromosome c in Chromosomes:
4                  if Chromosome.Vote(x) == TRUE:
5                          Counts[c.ClassBits.ToInt()]+=1
6          MaxIndex = HighestIndexOf(Counts)
7          return Counts[MaxIndex]
8
9  Chromosome.Vote(Sample x):
10         Result = TRUE
11         for each Cell c in Cells:
12                 Result &= c.Vote(x)
13                 if (Result == FALSE or c.JoinBit == FALSE)
14                         break
15         return Result^NotFlag //Where ^ is Exclusive Or
16
17 Cell.Vote(Sample x):
18         Value = Functions[FunctionBits](x)
19         Result = Value > lowerLimit & Value < upperLimit
20         return Result ^ NotFlag
```

In case of a tie, Hunter returns the lowest index, and in case of no votes returns a -1, which represents uncertainty. Thus, as written voting is deterministic, and as such a Hunter's performance is determined entirely by its bitstring. This enables the cheap storage and recall of even very complicated Hunters.

Complexity is certainly an issue, but before we can discuss it we need to discuss how breeding operators can handle this new complexity. Either because with variable length genomes crossover can't work unmodified, or because complexity wouldn't increase past whatever was assigned at generation 0. Both of these cases could be true without modification of the typical GA.

First, we will discuss our modifications to the classical crossover operators. Here we differ from our inspiration for this portion of our paper Kharma et al. (2004). While their crossover operator is modified to accommodate different length genes, our crossover treats each collection of genetic material distinctly.

Crossover Our crossover is invoked at the hunter level. Any two hunters may be crossed. Crossover may occur at the level of swapping Chromosomes, or may go deeper, so that two Chromosomes can swap cells, or it can go deeper still, such that two cells can crossover as normal, since all cells are the same length. In the highest level case, the operation can be thought of as crossover with chromosomes laid out contiguously. In the event that hunters have a different number of chromosomes, the remainder are allocated randomly according to the crossover rate.

In the middle case, Chromosomes perform a similar function with cells. Cells are left untouched but are swapped back and forth, functioning similarly to bits in a standard crossover operation.

In the lowest case (which we call Uniform), the case we have implemented, there's crossing over at all levels. It is probably most easily seen in algorithm form.

```
Hunter.Crossover(Hunter a, Hunter b):
        target = new Hunter(), notTarget = new Hunter()//Empty hunters for
            receiving genetic code
        least = min(a.Chromosomes, b.Chromosomes)
        most = max(a.Chromosomes, b.Chromosomes)
        if(max = a.Chromosomes) MaxHunter = a
        else MaxHunter = b
        for i = 0 to least:
                newChromosomes = Chromosome.CrossOver(a[i], b[i])
                if (RNG.Next < CrossoverChance)
                        switchTargets(target, notTarget)
                target.AddChromosome(newChromosomes[0])
                notTarget.AddChromosome(newChromosomes[1])
        for i = least to most:
                if (RNG.Next < CrossoverChance)
                        switchTargets(target, notTarget)
                target.AddChromosome(MaxHunter[i])
        return target,notTarget;
```

One minor addition here is that target and notTarget are actually pointers to hunters (and below to Chromosomes), though it obfuscates the algorithm unnecessarily to spell that out in pseudocode, particularly because switch targets is intuitive even if not explicit. As you can see, this allows us to cross hunters of any length. The algorithm for Chromosomes is similar, except that unlike hunters they have genetic material of their own to cross.

```
Chromosome.Crossover(Hunter a, Hunter b):
        target = new Chromosome(), notTarget = new Chromosome()least =
            min(a.Cells, b.Cells)
        most = max(a.Cells, b.Cells)
        if(max = a.Chromosomes) MaxChromosome = a
        else MaxChromosome = b

        for i = 0 to ChromosomeBitLength:
                target.bits[i] = a.bits[i];
                notTarget.bits[i] = b.bits[i];
                if (RNG.Next < CrossoverChance)
                        switchTargets(target, notTarget)
        //Now that the chromosome specific bits are crossed, we may proceed

        for i = 0 to least:
                newChromosomes = Chromosome.CrossOver(a[i], b[i])
                if (RNG.Next < CrossoverChance)
                        switchTargets(target, notTarget)
                target.AddChromosome(newChromosomes[0])
                notTarget.AddChromosome(newChromosomes[1])
        for i = least to most:
                if (RNG.Next < CrossoverChance)
                        switchTargets(target, notTarget)
                target.AddCell(MaxChromosome[i])
        return target,notTarget;
```

Cell.Crossover is the usual implementation of uniform crossover. Complexity is an issue here- if many chromosomes increased fitness, this could set off a run-away race toward

Chapter 3

Computational Fluid Dynamics

Chapter 4

Conclusions

Bibliography

Bibliography

(2002). CEDAR Databases. 16

Affenzeller, M. and Wagner, S. (2003). A self-adaptive model for selective pressure handling within the theory of genetic algorithms. In *International Conference on Computer Aided Systems Theory*, pages 384–393. Springer. 4

Alexandre, E., Cuadra, L., Salcedo-Sanz, S., Pastor-Snchez, A., and Casanova-Mateo, C. (2015). Hybridizing Extreme Learning Machines and Genetic Algorithms to select acoustic features in vehicle classification applications. *Neurocomputing*, 152:58–68. 15

Back, T. (1994). Selective pressure in evolutionary algorithms: a characterization of selection mechanisms. In *Proceedings of the First IEEE Conference on Evolutionary Computation. IEEE World Congress on Computational Intelligence*, pages 57–62 vol.1. 4

Chou, J.-S., Cheng, M.-Y., Wu, Y.-W., and Pham, A.-D. (2014). Optimizing parameters of support vector machine using fast messy genetic algorithm for dispute classification. *Expert Systems with Applications*, 41(8):3955–3964. 14

Daniele P. Radicioni and Roberto Esposito (2014). UCI Machine Learning Repository: Bach Choral Harmony Data Set. 13

Dehuri, S., Patnaik, S., Ghosh, A., and Mall, R. (2008). Application of elitist multi-objective genetic algorithm for classification rule generation. *Applied Soft Computing*, 8(1):477–487. 15

Devos, O., Downey, G., and Duponchel, L. (2014). Simultaneous data pre-processing and SVM classification model selection based on a parallel genetic algorithm applied to spectroscopic data of olive oils. *Food Chemistry*, 148:124–130. 14

Duan, L., Guo, L., Liu, K., Liu, E. H., and Li, P. (2014). Characterization and classification of seven Citrus herbs by liquid chromatographyquadrupole time-of-flight mass spectrometry and genetic algorithm optimized support vector machines. *Journal of Chromatography A*, 1339:118–127. 14

Fidelis, M. V., Lopes, H. S., and Freitas, A. A. (2000). Discovering comprehensible classification rules with a genetic algorithm. In *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No.00TH8512)*, volume 1, pages 805–810 vol.1. 16

Hoque, M. S., Mukit, M. A., and Bikas, M. A. N. (2012). An Implementation of Intrusion Detection System Using Genetic Algorithm. *International Journal of Network Security & Its Applications*, 4(2):109–120. arXiv: 1204.1336. 16

Hyafil, L. and Rivest, R. L. (1976). Constructing optimal binary decision trees is NP-complete. *Information processing letters*, 5(1):15–17. 27

J. P. Marques de S, J. Bernardes, and D. Ayres de Campos (2010). UCI Machine Learning Repository: Cardiotocography Data Set. 13

Kharma, N., Kowaliw, T., Clement, E., Jensen, C., Youssef, A., and Yao, J. (2004). PROJECT CellNet: EVOLVING AN AUTONOMOUS PATTERN RECOGNIZER. *International Journal of Pattern Recognition and Artificial Intelligence*, 18(06):1039–1056. 14, 16, 33, 36

Kowaliw, T., Kharma, N., Jensen, C., Moghnieh, H., and Yao, J. (2004). CellNet Co-Ev: Evolving Better Pattern Recognizers Using Competitive Co-evolution. In *Genetic and Evolutionary Computation GECCO 2004*, pages 1090–1101. Springer, Berlin, Heidelberg. DOI: 10.1007/978-3-540-24855-2_119. 17

Kozeny, V. (2015). Genetic algorithms for credit scoring: Alternative fitness function performance comparison. *Expert Systems with Applications*, 42(6):2998–3004. 15

Li, L., Zhang, G., Nie, J., Niu, Y., and Yao, A. (2012). The Application of Genetic Algorithm to Intrusion Detection in MP2p Network. In *Advances in Swarm Intelligence*, pages 390–397. Springer, Berlin, Heidelberg. DOI: 10.1007/978-3-642-30976-2_47. 16

Lichman, M. (2013). *UCI Machine Learning Repository*. University of California, Irvine, School of Information and Computer Sciences. 13

Marchetti, M., Onorati, F., Matteucci, M., Mainardi, L., Piccione, F., Silvoni, S., and Priftis, K. (2013). Improving the Efficacy of ERP-Based BCIs Using Different Modalities of Covert Visuospatial Attention and a Genetic Algorithm-Based Classifier. *PLOS ONE*, 8(1):e53946. 14

Ocak, H. (2013). A Medical Decision Support System Based on Support Vector Machines and the Genetic Algorithm for the Evaluation of Fetal Well-Being. *Journal of Medical Systems*, 37(2):9913. 14

Paul Horton (1996). UCI Machine Learning Repository: Yeast Data Set. 13

Ruiz, E., Albareda-Sambola, M., Fernndez, E., and Resende, M. G. C. (2015). A biased random-key genetic algorithm for the capacitated minimum spanning tree problem. *Computers & Operations Research*, 57:95–108. 5, 32

Salari, N., Shohaimi, S., Najafi, F., Nallappan, M., and Karishnarajah, I. (2014). A Novel Hybrid Classification Model of Genetic Algorithms, Modified k-Nearest Neighbor and Developed Backpropagation Neural Network. *PLOS ONE*, 9(11):e112987. 15

Schuman, C. D., Birdwell, J. D., and Dean, M. E. (2014). Spatiotemporal classification using neuroscience-inspired dynamic architectures. *Procedia Computer Science*, 41:89–97. 14

Srikanth, R., George, R., Warsi, N., Prabhu, D., Petry, F. E., and Buckles, B. P. (1995). A variable-length genetic algorithm for clustering and classification. *Pattern Recognition Letters*, 16(8):789–800. 16

Stefanie Scheid (2004). Introduction to Kernel Smoothing | Kernel (Operating System) | Probability Density Function. 25

Tseng, M.-H., Chen, S.-J., Hwang, G.-H., and Shen, M.-Y. (2008). A genetic algorithm rule-based approach for land-cover classification. *ISPRS Journal of Photogrammetry and Remote Sensing*, 63(2):202–212. 16

Uysal, A. K. and Gunal, S. (2014). Text classification using genetic algorithm oriented latent semantic features. *Expert Systems with Applications*, 41(13):5938–5947. 15

Wu, J., Long, J., and Liu, M. (2015). Evolving RBF neural networks for rainfall prediction using hybrid particle swarm optimization and genetic algorithm. *Neurocomputing*, 148:136–142. 14

Zhang, J. (2003). Evolution by gene duplication: an update. *Trends in Ecology & Evolution*, 18(6):292–298. 34

Zhang, Z., McDonnell, K. T., Zadok, E., and Mueller, K. (2015). Visual Correlation Analysis of Numerical and Categorical Data on the Correlation Map. *IEEE Transactions on Visualization and Computer Graphics*, 21(2):289–303. 21

Appendix

Appendix A

Summary of Equations

A.1 Cartesian

some equations here

A.2 Cylindrical

some equations also here

Vita

Vita goes here...