An Investigation of Genetics-Based Machine Learning as Applied to Global Crop Yields

Will Gantt

April 3, 2017

1 Introduction

2 Background

In this section, I provide some context for this project. First, I introduce the Genetics-Based Machine Learning paradigm and show where learning classifier systems fit into it. Next, I give a general overview of learning classifier systems and how they work. I then highlight relevant previous research in the field, and I conclude by talking specifically about the earlier work of Nelson and Congdon in [citation] out of which my project developed.

2.1 Genetics-Based Machine Learning

Genetics-Based Machine Learning (GBML) is a machine learning approach based on the use of evolutionary computation (EC). Briefly, EC comprises a set of optimization algorithms that evolve populations of candidate solutions to a particular problem over a number of generations via a process inspired by biological evolution. Many of the most interesting and important problems in machine learning have large and noisy solution spaces, and it is in such a contexts that EC techniques have proved particularly effective.

The precise relationship between various GBML-related terms and concepts is difficult to articulate, and usage in the community is often loose. GBML, once taken to refer exclusively to LCSs, is now considered a broader term that encompasses a number of other algorithms, including genetic programming, evolutionary ensembles, evolutionary neural networks, and genetic fuzzy systems [Kovacs, 2012]. What is important to note here is simply that LCSs exist within a larger machine learning problem-solving framework that covers a number of different EC-based algorithms.

2.2 What is a Learning Classifier System?

Introduced by John Holland in 1975, Learning Classifier Systems (LCSs) are GBML algorithms that evolve a population of rules (footnote about how these are often called classifiers). Each rule consists of a condition that indicates when the rule applies, as well as an action to take if the condition is met. As LCSs have applications to a variety of learning problems, the exact function of the rules depends largely on the task. With supervised learning problems, rules typically try to categorize a set of training examples by mapping the features of those examples to particular classes. In these cases, the rules condition describes the features of the examples to which it applies, and the action specifies the class to be assigned to examples matching the condition. Problems in reinforcement learning, by contrast, often involve determining the best action for an agent to take in response to various environmental cues. Here, a rules condition describes a set of cues and its action determines how the agent should react given those cues.

Regardless of the types of problems to which they are applied, all LCSs share certain basic structures and mechanisms. Roughly, these are:

1. *Population.*
2. *Discovery Component.*
3. *Learning Component.*

2.3 Developments in LCS Research

2.4 Global Crop Yields

3 System Design

Below, I present my design for a Michigan-style LCS for the classification of examples with real-valued attribute vectors. I have divided the explanation into four parts. The first part explains the representation of rules, including a short description of all of their attributes and a longer discussion of the choice of representation of the condition in particular. The second describes the learning component and offers a general overview of the GA as well as more detailed analysis of its constituent modules. The third covers the testing component—the algorithm for the classification of new examples once a rule set has been created. Finally, the fourth enumerates the parameters of the LCS.

3.1 Rule Representation

Rules are represented as objects with the following attributes:

- ID: A variable that uniquely identifies the rule in the population.
- Classification: The class, given by an integer value, according to which an example matching the rule will be classified.
- # True Positives: The number of examples that the rule both matches and correctly classifies.
- # True Negatives: The number of examples that are neither matched by the rule nor of the same class.
- # False Positives: The number of examples that the rule matches, but incorrectly classifies.
- # False Negatives: The number of examples that have the same class as the rule but that the rule does not match.
- # Don't Care: The number of Attribute objects in the rule's condition (a vector of Attribute objects) whose "don't care" variable is currently set to true.
- Fitness 1: An initial fitness value used to determine a preliminary ranking of the rules.
- Fitness 2: A fitness value based on Fitness 1, but that takes into account the fact that there may be multiple rules that cover the same examples.
- Condition: A vector of Attribute objects that describes the attribute values an example must have to be matched by the rule.

As mentioned earlier, much of LCS research has concerned itself with the classification of examples with binary-valued attributes that simply indicate the presence or absence of some feature in the example. The conditions of rules used in LCSs of this sort may therefore be represented as strings of characters from the ternary alphabet 0,1,#, where the '#' indicates an attribute that is irrelevant. When dealing with real values, however, determining how best to represent the condition is more difficult. With real values, not only does the search space expand to a theoretically infinite size, but the genetic operators (crossover and mutation) have to be adapted

as well. Thus, the condition must be constructed in a manner that (a) reduces the search space to a reasonable size, and (b) does not hinder or over-complicate the functioning of the genetic operators.

One approach that has proved successful, and the one I have implemented in this system, is to discretize the attributes of the rules condition into some number of quantiles. Each attribute in the rules condition not marked as irrelevant then covers a range of values specified by one of the quantiles. Note that a condition constructed in this way meets both of the criteria described in the preceding paragraph: It radically restricts the search space (since each attribute will have a small number of quantiles) and it does not make any fundamental alterations to the behavior of the genetic operators (as will be shown in more detail further on in this section). The number of quantiles used is the same across all non-class attributes and is specified by the user. The condition is represented as a vector of Attribute objects, each of which has a name, a quantile number, an upper and lower bound describing the range of that quantile, and a boolean "dont care" value indicating whether the attribute is relevant for the rule to which it belongs.

3.2 Rule Discovery and Learning

Overview

As with virtually all other LCSs, a genetic algorithm serves as the discovery component of my system. One complete run of the LCS attempts to characterize a single "target" class from the data. The population is initialized with a user-specified number of rules whose class attribute values are set to that of the target class, and whose condition attributes are randomly either set as "dont cares" or else assigned a random quantile. For each rule in the population, the system tallies the number of true positives, true negatives, false positives, and false negatives among the examples in the training set. The rule is then assigned a preliminary fitness based on those values. Once this has been done for all of the rules, they are ranked by fitness.

Next, a second fitness is calculated using the same procedure, except that an example may count as a true positive for at most one rule. In computing the original fitness, it frequently happens that a single example will be counted as a true positive for multiple rules. In order to avoid such redundancy and to encourage diversity in the population, the second fitness counts an example as a true positive only for the fittest rule (based on the first fitness) that covers it.

The N best rules as evaluated by this second fitness are then immediately

added to the next generation. To fill the remainder of the population for the next generation, $pop\_size - N$ rules are selected for reproduction from the current one using stochastic universal sampling. The selected rules are paired and crossed over to produce two offspring, which are mutated and added to the next generation. The process then repeats for a user-specified number of iterations.

Fitness Update

Both the first and second fitnesses are computed using the odds ratio. A given example is either a positive instance ($T+$) of the target class or a negative instance ($T-$). Similarly, a given rule will classify the example either as a positive instance of the target class ($R+$) or a negative instance ($R-$). The four possible combinations of the examples actual class ($T+/T-$) and the class that a given rule selects for that example ($R+/R-$) yield the following matrix:

Clockwise from the top left, the elements of the matrix correspond to the categories true positive, false positive, true negative, and false negative (for a given rule and example pair). As its name suggests, the odds ratio is a ratio of two odds:

1. The odds that a rule classifies an actual positive example of the target class as a true positive (a/c).

2. The odds that a rule classifies a negative example of the target class as a true positive (b/d).

Dividing (1) by (2) gives:

$$\frac{\left(\frac{a}{c}\right)}{\left(\frac{b}{d}\right)} = \frac{ad}{bc} = \frac{|TP| \cdot |TN|}{|FP| \cdot |FN|}$$

[What is the advantage of odds ratio?] There is no difference in the calculation of the two fitnessesonly in the tallying of true and false positives and negatives. In the first fitness, an example counts as a true positive for all the rules that match it. In the second fitness, an example counts as a true positive only for the fittest rule that matches it, as determined by the first fitness. For a rule that matches the examples, but that is not the fittest that does so, the example is simply not included in any of tallies. Because the second fitness is the one that drives the genetic algorithm, those rules that

are redundantthat cover examples already covered by fitter ruleswill over time be weeded from the population. Observe that this schema also rewards those rules that cover just a few examples, so long as those examples are not covered by fitter rules.

Selection

Selection is done on the basis of the second fitness alone using stochastic universal sampling (SUS), a superior alternative to traditional roulette wheel selection developed by James Baker in [citation]. In roulette wheel selection (also known as fitness-proportionate selection), members of a population are selected randomly, where the probability of a members being selected is given by:

$$\frac{fitness_k}{\sum_{i=1}^{N} fitness_i}$$

The basic algorithm is shown in [figure #]. First, the sum of the fitnesses of all members of the population is computed. Next, a number r in the range $[0, fitness_sum]$ is determined by randomly selecting a value from the interval $[0, 1]$ and multiplying that value by the sum of the fitnesses. The members of the population are then iterated over. At each iteration, the fitness of the current member is subtracted from r. The member whose fitness, when subtracted from r, causes r to fall below 0 is the one selected. It is clear, therefore, that the larger the the fitness of a member, the more likely it is that that member will be chosen.

The primary shortcoming of roulette wheel selection is the extent to which it is biased against rules of lesser fitness. Generally speaking, we want the rules selected to be among the fitter ones, but in order to avoid premature convergence, some low-fitness rules must participate regularly in the crossover.

SUS reduces this bias. Instead of generating a new random variable for each selection, SUS uses a single random variable to make all of its selections. This turns out to be of great utility in increasing the frequency with which low-fitness rules are selected.

[Figure #] shows the SUS algorithm in pseudocode. It will help, in understanding the behavior of SUS, to imagine the interval $[0, fitness_sum]$ on the number line. Further, imagine that each member of the population is given its own subinterval within the larger interval that is equivalent in length to its fitness (see [figure #]). Finally, suppose that the subintervals are arranged left-to-right in descending order of fitness.

To begin, a value known as the "pointer interval" is computed by dividing the sum of the fitnesses ($fitness\_sum$) by the number of rules to be selected (N). Then, a random number r is chosen from the range $[0, pointerinterval]$. Beginning at r, the algorithm selects N positions along $[0, fitness\_sum]$ by counting out intervals of length $pointer_interval$. The rules corresponding to the subintervals in which the positions fall are the ones selected.

[A graphic comparing roulette wheel selection and SUS]

Genetic Operators and Replacement

The LCS employs only the two canonical genetic operators, crossover and mutation, which are described in detail below.

Crossover. The genetic algorithm uses single-point crossover. For every set of parents, a random integer in the range $[1, num\_attributes - 1]$ is generated. This value represents the crossover point. The condition of each of the offspring rules is determined by mapping the values of the attributes to the left of the crossover point in one of the parents to the corresponding attributes in the offspring. The same is then done with the other parent, but taking all of the attribute values to the right of the crossover point instead.

Mutation. Following crossover, the mutation operator iterates over the condition attributes of each offspring. Each attribute has the same probability (p_mutate) of being modified. If it has been determined that an attribute is to be mutated, a second random number in the interval $[0, 1]$ is generated. If that number is less than or equal to a probability p_dont_care, the attributes "dont care" variable is set to true. If the number is greater than p_dont_care, the attribute is randomly assigned a new quantile.

[A graphic of how crossover works.]

After both operators have been applied, all of the offspring replace all of the non-elite ($pop\_size - N$) members of the current generation. The resulting population will be the one used in the next generation.

3.3 Classification of Examples

The classification of examples in the test set is very straightforward. At the end of *num_iters* iterations, the genetic algorithm terminates. The elite rules of the final population are the only ones used for the classification task. The reason for this is two-fold:

1. *Survival of the Fittest.* The elite rules in the final population are (mostly) those that have weathered repeated applications of the selection procedure and have come out on top. Among the non-elites,

which will have just been generated through crossover and mutation on the final iteration, there are likely to be many bad rules.

2. *Economy.* We are interested in minimizing the number of rules needed to accurately characterize the target class.

Once these rules have been identified, the LCS iterates over the examples in the test set. If a rule that covers the current example is found among the set of elites, the example is classified as a member of the target class. If no matching rule is found, the example is classified as a member of another, default class. Since the system attempts to characterize just one class per run (the target class), it matters little what is chosen for the default class, so long as it is different from the target class. After a class has been selected for the example, the system compares the selected class with the actual class of the example and categorizes it accordingly as a true positive, true negative, false positive, or false negative. The tallies of all four categories are outputted once all examples have been classified.

3.4 Parameters

The LCS has the following parameters:

- *pop_size:* the number of rules in the population.
- *num_iters:* the number of iterations the LCS will run before terminating.
- *target_class:* the class in the training data that the LCS will attempt to characterize.
- *elitism_rate:* the approximate fraction of the population that is to be retained from one generation to the next. [Fix description.]
- *p_mutate:* the probability that a condition attribute will be mutated when the mutation operator is invoked.
- *p_dont_care:* if it has been determined that an attribute is to be mutated, the probability that its "don't care" variable will be set to true (instead of assigning it a new quantile).
- *training_set:* an object of the Dataset class containing all of the examples in the training set and other relevant information.
- *test_set:* another object of the Dataset class containing all of the examples in the test set and other relevant information.