Decision Trees:
A comparison of genetic and information-theoretic classification trees

Corbin Rosset

April 10, 2015

Abstract

In this paper we compare the performance of decision trees constructed by two different machine learning algorithms on binary classification problems. The first is an information-theoretic algorithm that selects for attributes which most aggressively reduce entropy in the training set. The second leverages the concept of natural selection to build populations of decision trees probabilistically. We show that while the performance of the algorithms is similar, the underlying flexibility of the genetic algorithm allows for greater generality and propensity to learn the data, but at the cost of tuning its numerous parameters.

1 Introduction

Classification is a prototypical machine learning problem in which an artificial intelligence agent learns a concept via induction to map objects to a discrete-valued class or label based on available attributes. A real world object like a car or a house is encoded by an electronic representation. The encoding of an instance of such an object i usually takes the form of a vector $\Theta_i = \langle \alpha_1, \alpha_2, ..., \alpha_m \rangle$, where an attribute α_j in the set α can take on a value from a set of discrete values $v_j = \langle v_{j,1}, v_{j,2}, ..., v_{j,k} \rangle$. Thus a set of n encodings, hereafter referred to as "examples", oriented column-wise is a $m \times n$ matrix populated by values. The agent attempts to "decode" the real-world identity of a novel example Θ_i, based solely on the values Θ_i's attributes assume. In supervised learning, an agent develops a strategy to predict identity classes by "training" with examples that have already been correctly labeled with an identity [2]. The hope is that the strategy will generalize to previously unseen examples.

The input to an agent undergoing training is an n element subset $\mathcal{R}$ of some universe $\mathcal{U}$ of possible input examples correctly labeled with a class y_i from a set $\mathcal{Y}$ of possible classes

$$\mathcal{R} = \{(x_1, y_1), (x_2, y_2), ..., (x_n, y_n)\}$$

Each y_i was generated by some concealed, omniscient function $f : \mathcal{U} \to \mathcal{Y}$, which the agent attempts to approximate with a hypothesis function $h : \mathcal{U} \to \mathcal{Y}'$, $\mathcal{Y}' \subseteq \mathcal{Y}$ that minimizes a loss function $L : \mathcal{Y} \times \mathcal{Y}' \to \mathbb{R}^+$. Sometimes f is merely a distribution, other times it can be a complicated, even nonlinear, combination of attributes in $\mathcal{U}$ [2]. The loss function can also be quite complicated, as not all errors should be treated equally. Sometimes, a false positive is far less consequential than a false negative (incorrectly labeling a dangerous person as clear when passing through airport security is an example). Thus depending on the application of a classifier, loss should not always be solely related to accuracy.

Learning h can be then be interpreted as a local search for a global minima to L through the space of all solutions $\mathcal{H}$; and constructing a classifier can be reduced to finding the best h given a set of input data:

$$h^* = \underset{h \in \mathcal{H}}{\operatorname{argmin}} \{ L(f(x), h(x)) \; \forall \; x \in \mathcal{R} \}$$

Once h has been selected, the finalized agent is then tested on a labeled test set $\mathcal{T}$ satisfying $\mathcal{T} \cap \mathcal{R} = \varnothing$. The most successful hypotheses will generalize, meaning they correctly predict y_i for novel examples [1].

One of the many tradeoffs involved in the search for h is one between overfitting and generalization: the former prefers a complicated hypothesis with which to fit the training data, but could fail to generalize because it memorizes or "overfits" it. The latter prefers a simpler hypothesis function that may generalize more, but the risk of neglecting too much information is incurred. Too much bias towards either approach leads to suboptimal error on $\mathcal{T}$. The agent is also vulnerable to error if there are inconsistencies in the training labels, or if the choice of attributes and values is not expressive enough to capture all there is to know about objects in $\mathcal{U}$. Conversely, if the description of an object contains too much "noise", then the complexity of the search space increases exponentially in the number of attributes [3].

Decision trees are just one model used to construct a classifier that wields h, others include Artifical Neural Networks, Support Vector Machines, and Reinforcement Learning to name a few. The next section will discuss how decision trees are represented, and what algorithms are used to construct them.

2 Learning Decision Trees

A decision tree is a deterministic approximation function which models the true mapping from an object Θ to a discrete-valued class or label $y \in \mathcal{Y}$. A decision tree may be represented as a cascade of `if` statments applied to the attributes α of an object; the cascade can be built into a tree down which objects are sorted by the conditions imposed by the `if` statements until they reach a leaf node, which finally provides a classification. Each internal node in the tree tests a single attribute, α_i, and each outgoing edge of the node corresponds to a possible value v_j that α_i can assume. Any path from the root to a leaf is isomorphic to a boolean expression in conjunctive normal form: each branch eminating from a node is a disjunction of attribute values, and a path to a node is a conjunction of attribute tests [1]. An example of a decision tree is shown in Figure 1.

Generally, decision trees are robust to misclassified training examples and missing

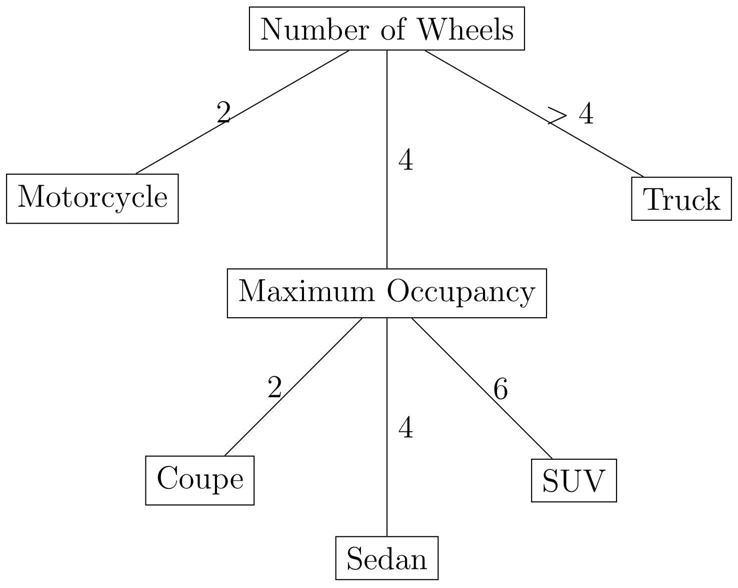

Figure 1: An example of a decision tree that classifies motor vehicles

attribute values that affect small quantities of the training data. Even without such errors, decision trees may experience difficulty approximating a function h on the set of attributes simply because $\mathcal{H}$ is so expansive: with m attributes, a boolean function in conjunctive normal form over the set of attributes that classifies each example, there are 2^{2^m} different functions for h. Searching for the globally optimal function is subject to the same difficulties experienced by any local search algorithm [3]. Below we discuss how two different algorithms approach this problem.

2.1 Information-Theoretic Methods

Learning a decision tree traditionally involves top-down expansion of a single decision tree by selecting attributes to generate new nodes in a greedy way. The canonical algorithms are ID3 and C4.5, both of which use a utility heuristic to recursively choose which among the unselected attributes to serve as a new internal node in the tree. The utility function is based on a statistical property, information gain, that "measures how well a given attribute separates the training examples according to their target classification" [1], which is based on the entropy, defined as:

$$\texttt{Entropy}(S) = -\sum_{k} P(s)\log_2 P(s)$$

Where S is any multiset, $s \in S$, and $P(s)$ is the probability of selecting s from S. It is difficult to extract patterns from high entropy data because it is noisy; each s occurs with roughly equal probability. Entropy is zero if there is perfect homogeneity in a dataset [1]. The information gain of an attribute α is defined as how much the knowledge of α reduces entropy in the training data:

Algorithm 1 Traditional Algorithm from [3]

```
Input: set of examples, set of attributes, set of parentExamples
Output: a decision tree
 1: procedure ID3($examples, attributes, parentExamples$)
 2:     if $examples$ is empty then
 3:         return Plurality($parentExamples$)
 4:     else if all $examples$ have same classification then
 5:         return the classification
 6:     else if $attributes$ is empty then
 7:         return Plurality($examples$)
 8:     else
 9:         $\alpha \leftarrow \text{argmax}_{\alpha_i \in \text{attributes}} \text{Utility}(\alpha_i, examples)$
10:                     ▷ Utility is information gain in ID3 or gain ratio in C4.5
11:         $tree \leftarrow$ tree with root attribute$\alpha$
12:         for all values $v_i$ of $\alpha$ do
13:             $exs \leftarrow \{e : e \in examples$ and $e.\alpha = v_i\}$
14:             $subtree \leftarrow ID3(exs, attributes \setminus \alpha, examples)$
15:             add a branch to $tree$ with label $\alpha = v_i$ and subtree $subtree$
16:         end for
17:         return $tree$
18:     end if
19: end procedure
```

$$\texttt{InformationGain}(\mathcal{R}, \alpha_i) = \texttt{Entropy}(\mathcal{R}) - \sum_{v \in v_i} \frac{|\mathcal{R}_v|}{|\mathcal{R}|} \texttt{Entropy}(\mathcal{R}_v)$$

Where $v \in v_i = \langle v_{i,1}, v_{i,2}, ..., v_{i,k} \rangle$ is a set of all possible values associated with a particular attribute, α_i; and $\mathcal{R}_v = \{s \in \mathcal{R} | \alpha_i = v\}$ is the subset of all training examples $\mathcal{R}$ for which α_i takes on value v. The rightmost term, considered first without the summation, is the entropy of that subset $\mathcal{R}_v$, weighted by the fraction of all $\mathcal{R}$ that take on value v. The summation considers all values v that attribute α_i can assume, and piecewise calculates the remaining entropy in $\mathcal{R}$ by considering the entropy in $\mathcal{R}_v$ for all $v \in v_i$. Since it is assumed all examples in $\mathcal{R}$ take on exactly one value for attribute α_i, partitioning $\mathcal{R}$ by v is a mutually exclusive and collectively exhaustive procedure. Thus $InformationGain(\mathcal{R}, \alpha_i)$ calculates the remaining entropy of $\mathcal{R}$ after the value for α_i is known for each example. Algorithm 1 shows ID3, the traditional information-theoretic decision tree algorithm (TA) implemented for this paper.

Limitations of ID3

ID3 maintains and refines only one hypothesis function h in its search, and furthermore, once a node is added to the tree, it is never reconsidered. This is analogous to a hill-climbing search without the capability to backtrack or switch hypotheses. Such greedy behavior encourages convergence to a local optimimum, rather than a global one. In this case, selecting the most "useful" attributes from a set of yet unselected attributes may result in a tree that fails to minimize L on $\mathcal{T}$ in its entirety. Such subtleties are not a result of flawed mathematics, but rather overly optimistic assumptions about how attributes and their values encode information.

The problem of overfitting the training data is a symptom of high dimensionality in the attribute and value spaces, as well as noisy data, or too few examples. Overfitting has occured whenever a hypothesis h chosen by the algorithm has smaller error than another hypothesis h' over $\mathcal{R}$, yet h' has smaller error over $\mathcal{T}$. With too many attributes, those selected by the algorithm will have similar (and low) information gains, which leads to deep trees and complicated hypotheses. With noisy examples in $\mathcal{R}$, the algorithm will add more internal nodes in response to what seems like a more complex pattern in the data, when really such an aberration should be ignored. Unnecessary complexity, at least deeper in the tree, can be detected using statistical significance tests. If an attribute close to a leaf is found to be irrelevant, the node is replaced with a leaf node whose classification is the plurality of the leaves below it. Such a process is known as χ^2 pruning, and is run with a set of labeled examples separate from $\mathcal{T}$ and $\mathcal{R}$ once the decision tree has been built [1].

$InformationGain$ also favors attributes with the most values because even minor contributions to entropy reduction by each individual value will accumulate. The edge case to consider is a unique identifier attribute, where there is a unique value for every example. The tree would memorize these strings and perfectly predict the label for every example in $\mathcal{R}$. Such a tree would not recognize the unique identifier of novel examples and would fail to classify them at all. More commonly, if there are several attributes, each with many values, then ID3 will likely make a node for each attribute, increasing the branching factor and complexity. The adversary here is not the data itself, but the degree of uniformity found in the training data by splitting on these unfortunate attributes [3]. After all, an attribute whose values split the training data into equally sized subsets has provided no more information than randomly partitioning the training data.

C4.5 and Other Corrected Algorithms

Algorithms such as C4.5 introduce new utility functions to limit the complexity of h. C4.5 uses gain ratio to penalize attributes with values that uniformly distribute the data. Gain ratio is defined as

$$GainRatio = \frac{InformationGain(\mathcal{R}, \alpha)}{SplitInformation(\mathcal{R}, \alpha)}$$

$$SplitInformation(\mathcal{R}, \alpha) = -\sum_{v \in v_i} \frac{|\mathcal{R}_v|}{|\mathcal{R}|} log_2(\frac{|\mathcal{R}_v|}{|\mathcal{R}|})$$

where v is value in v_i, the set of values that any attribute α_i can assume. The fewer examples that v partitions out of $\mathcal{R}$, the less significant α_i becomes. Unfortunately, if an attribute partitions too great a fraction of $\mathcal{R}$, the denominator of $SplitInformation$

becomes very small and *GainRatio* approaches infininity. There are ways to account for this edge case.

.

2.2 Genetic Algorithms

Genetic algorithms attempt to simulate the mechanics of the theory of natural selection, which has proven to be a good real world optimization technique. Rather than refining a single hypothesis as a traditional information-theoretic algorithm does, a genetic algorithm (GA) will consider populations of hypotheses. Each hypothesis is applied to the training data to generate a measure of fitness, $fit : h \times \mathcal{R} \to \mathbb{R}$[1] The initial population of decision tree hypotheses is built randomly, and successive generations of offspring hypotheses are reproduced by mechanisms such as crossover and mutation applied to selected parents in the parent generation. The qualities of the individuals with the greatest fitness are perpetuated, either because some percentage of the fittest parents are replaced into the new generation, or because the fittest individuals are most frequently selected for reproduction [2]. A child hypothesis should amount to a recombinant version of the best current hypotheses, with moderate to intense probabilities involved.

One of the advantages of GAs is that they overcome suboptimality as it arises from locally greedy decisions. GAs evaluate fitness on fully formed hypotheses, and while many such hypotheses are rather poor, a small subset of the population will contain useful traits, and such traits will be combined and perpetuated into highly fit individuals.

There are many parameters to tune for a genetic algorithm to work properly. Initializing the zeroth generation involves building every hypothesis randomly. To build a random tree with reasonable size bounds, one can specify a probability that the next randomly generated node is a leaf. The author found it best to use the following formula to increase the probability of creating a leaf node as the depth of the tree increases:

$$P(leaf) = 1 - \frac{1}{(\frac{|\alpha|+b}{|\alpha|})^d}$$

Where $|\alpha|$ is the number of attributes, b is the expected branching factor for any attribute selected uniformly at random, and d is the depth of the tree. This function approaches one, but is mindful of the expected size of a decision tree given the dimensionality of the attributes involved.

After establishing the initial population, but before selection and reproduction even begin, the *replace* percent of the population with the greatest fitness are automatically preserved for the next generation. If the algorithm mandates that every generation have the same population size, then the remaining uncreated hypotheses

[1]Just as objects are encoded as vectors of attributes, decision tree hypotheses must also be encoded. For the purposes of this paper, the tree itself was its own encoding, but this atypical.

[2]a GA can be interpreted as a variant of stochastic beam search

Algorithm 2 Genetic Algorithm from [3]

```
Input: fitness function, set of attributes, set of training examples, population size,
crossover probability, mutation rate, replacement rate
Output: a decision tree
▷ fit() : $tree \times examples \to \mathbb{R}$
 1: procedure ID3(fit(), $examples, popSize, crossProb, mutProb, replace$)
 2:     $population \leftarrow popSize$ randomly generated trees                ▷ limit size
 3:     repeat
 4:         $newPopulation \leftarrow$ empty set
 5:         add the top $replace \times popSize$ most fit individuals to $newPopulation$
 6:         for $i = 1$ to $(popSize - newPopulation.size)$ do
 7:             if rand() $\leq crossProb$ then
 8:                 $parent1 \leftarrow$ selectIndividual($population$, fit())
 9:                 $parent2 \leftarrow$ selectIndividual($population$, fit())
10:                                 ▷ select from a fitness-proportionate distribution
11:                 $child \leftarrow$ reproduce($parent1, parent2$)
12:                       ▷ crossover happens with some probability in reproduce
13:             else
14:                 $child \leftarrow$ random tree
15:             end if
16:             if rand() $\leq mutProb$ then
17:                 $child \leftarrow$ mutate($child$)
18:             end if
19:         end for
20:     until a max number of generations have been made
21:     return the best tree found over all iterations
22: end procedure
```

are built either by crossover (which occurs about 60 to 80 percent of the time) or by random construction just like the initial population. If reproduction occurs, parent hypotheses are selected from a probability mass distribution applied to *population* specified by:

$$P(h) = \frac{\texttt{fit}(h, \mathcal{R})}{\sum_{h_i \in population} \texttt{fit}(h_i, \mathcal{R})}$$

Once parents have been selected, the child is initialized to a copy of one parent, with a random subtree replaced by a random subtree of the other parent. Mutation, if it occurs, could change the attribute of a node [3], change a node near the bottom of the tree into a leaf, or change the classification of a leaf.

Like the traditional algorithm, the challenge for a GA is to avoid premature convergence. If the population is too small, contains too little diversity, or enough

[3]while preserving validity of the tree, that is, not changing the number of values an attribute can assume

generations are not produced, then suboptimality is likely. Also, without introducing enough ”fresh” random trees, or not crossing subtrees aggressively enough, the algorithm could spin on a local optimum.

3 Experimental Methods and Data Sets

There were three primary data sets: yea or nea votes cast by congress for bills, MONK data, and mushroom examples to classify as poisonous or edible. The MONK data contained three different sets, and were created for a competition and were naturally challenging. Each algorithm, TA and GA, was run on all of the data sets; for each run, the accuracy, precision, and recall of the algorithm was recorded for classifying both the training and testing data. Some data sets had unique identifiers, they were removed in preprocessing. Experiments mostly applied to the GA, since it had many parameters to tune. Modifying the number of generations, crossover rates, and population sizes are reported below.

4 Results

The author performed several experiments that measured the performance of the Genetic Algorithm as a function of a single parameter. Classification accuracy was used as a representation of overrall performance, but similar results would have been found using precision and recall. The standard parameters used by the author were

Table 1: Standard Parameters used for the Genetic Algorithm

GA parameter	**Value**
Population Size	100
Probability of Mutation	0.06
Probability of Crossover	0.75
Percentile of Top Individuals to Be Elites	0.06
Number of Generations	50
Minimum Fitness Allowed for Selection	0.5
Impose Fitness Penalty if Tree Too Large	yes
Boost Fitness if High Precision	yes
Select an Elite for Reproduction at Most	once

The reader should assume that unless otherwise specified by a graph or table, that the standard values were used in an experiment.

There are several features of the implementation of our algorithm that merit attention. The first is the control of tree size by placing a fitness penalty on trees whose size exceeds a certain threshold. See the above section for details. Below is a table comparing tree sizes with and without the fitness penalty:

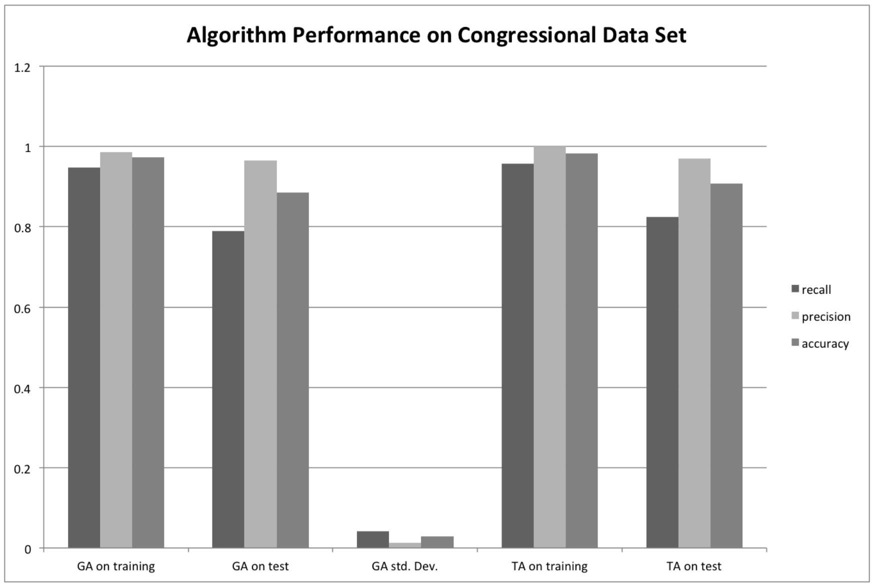

Figure 2: Comparison of the performance of the genetic and traditional algorithms on their respective training and test sets for the Congressinal Voting data

Table 2: Effect of Fitness Penalty on Number of Nodes in Tree

Size Penalty	Average	Std. Dev.
Output Tree	195	96
Average of Final Population	169	68
No Size Penalty		
Output Tree	507	290
Average of Final Population	451	127

Table 2 shows the results of ten runs of GA on the Mushroom data set, chose because it had the highest expected branching factor of all data sets - 5.72. For each run, the number of nodes in the output tree and the average number of nodes in the final population were recorded. This data was averaged over all the runs and the standard deviation thereof recorded. While there was no real difference in the performance of the trees with and without size restrictions, it is clear that without them, redundant nodes accumulate.

Figures one through five compare the accuracy, recall, and precision of the genetic and traditional algorithms. Since the genetic algorithm is nondeterministic, it was run ten times over each data set, with the average accuracy, precision, and recall recorded for both test and training data. The standard deviation over the test data is also recorded for the genetic algorithm since, at times, the output contains significant variance. Because the traditional algorithm is deterministic, it was only run once. The y-axis is understood to be performance of all three metrics: accuracy, precision, and recall, since they are all values on the range zero to one.

Figures 6 through 9 hold all other parameters constant while varying one other parameter deemed worthy of inspection.

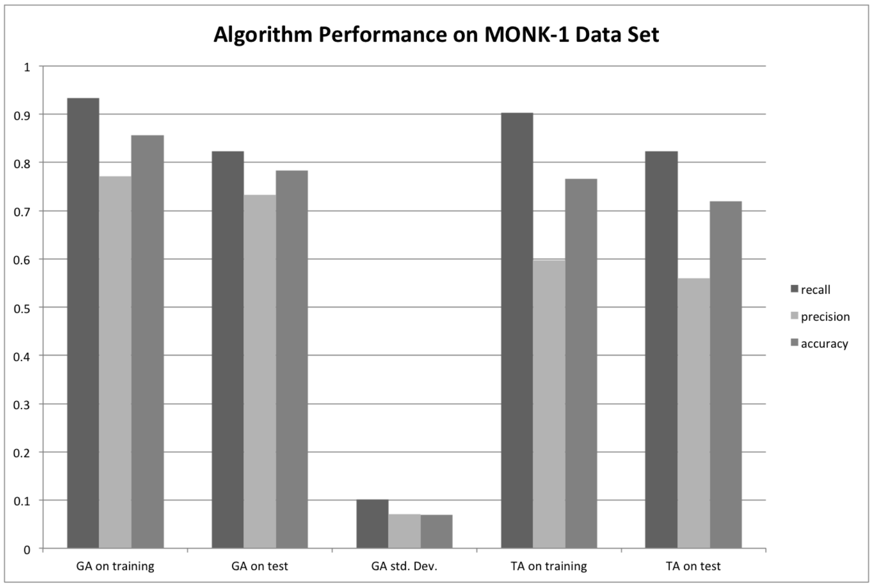

Figure 3: Comparison of the performance of the genetic and traditional algorithms on their respective training and test sets for the MONK-1 data

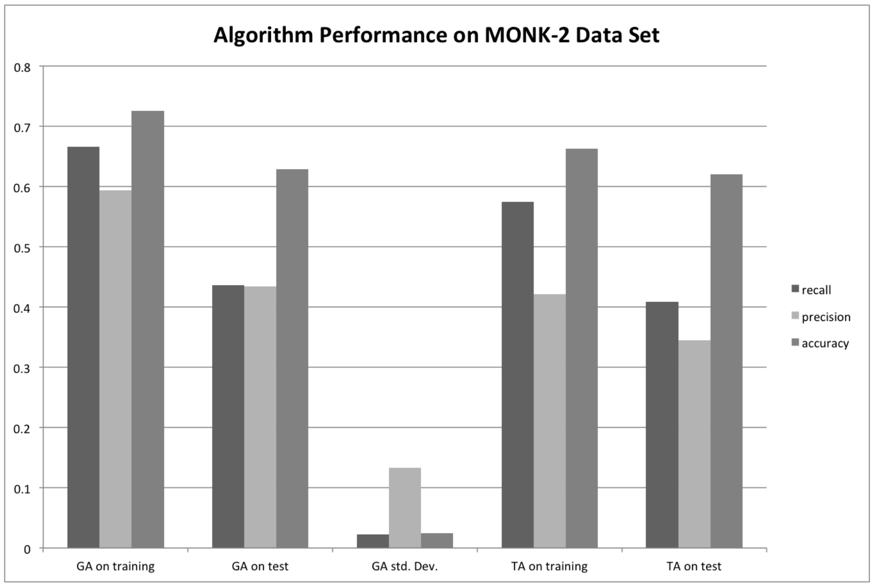

Figure 4: Comparison of the performance of the genetic and traditional algorithms on their respective training and test sets for the MONK-2 data. Note the maximum performance on the range is only 0.8

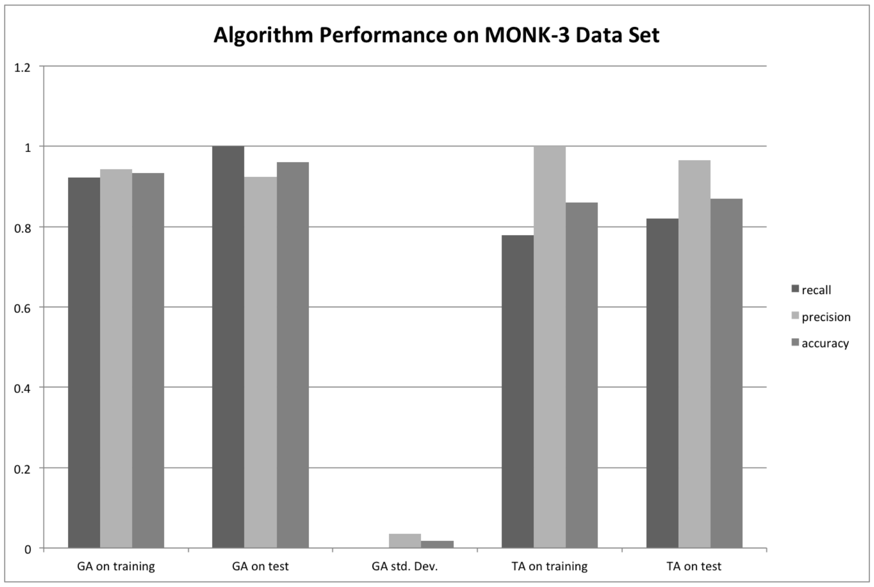

Figure 5: Comparison of the performance of the genetic and traditional algorithms on their respective training and test sets for the MONK-3 data

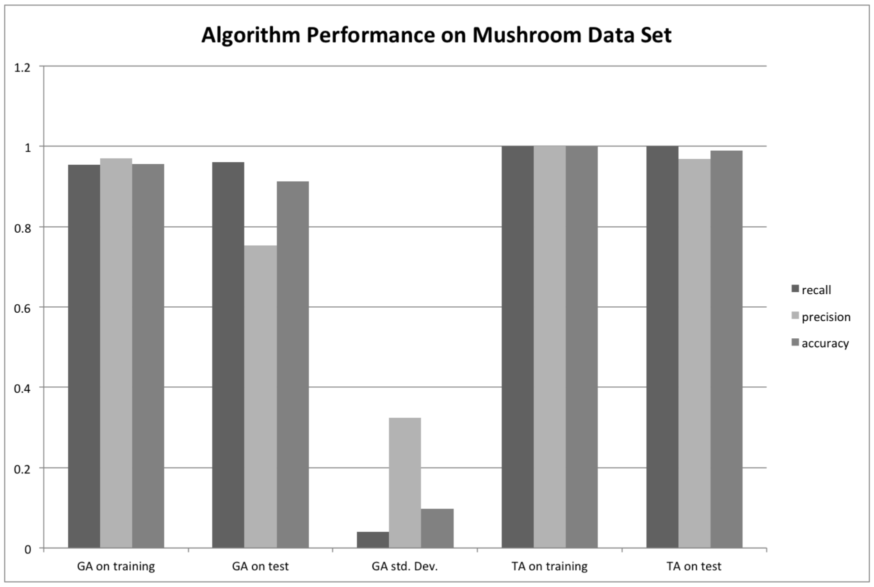

Figure 6: Comparison of the performance of the genetic and traditional algorithms on their respective training and test sets for the Mushroom data

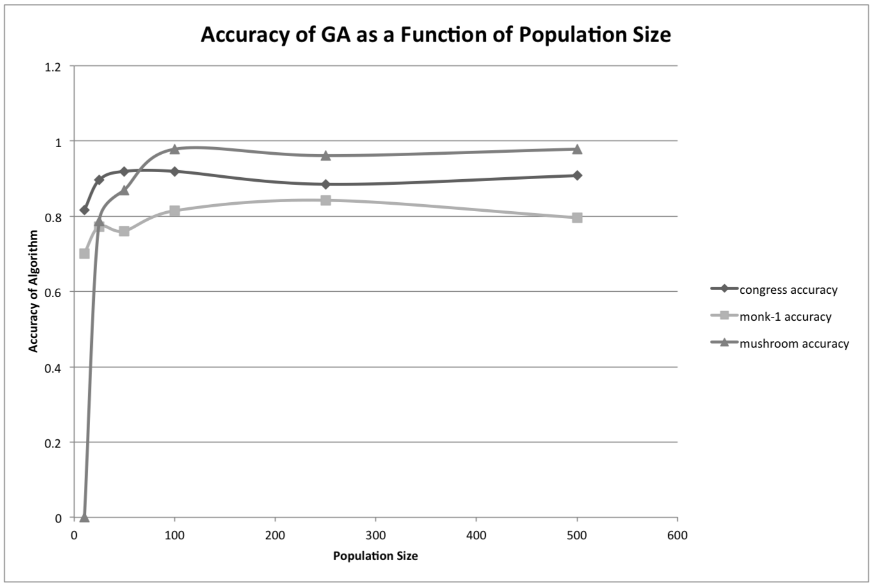

Figure 7: The accuracy of the genetic algorithm was recorded for varying population sizes, averaged over five runs. Only three data sets were used for clarity

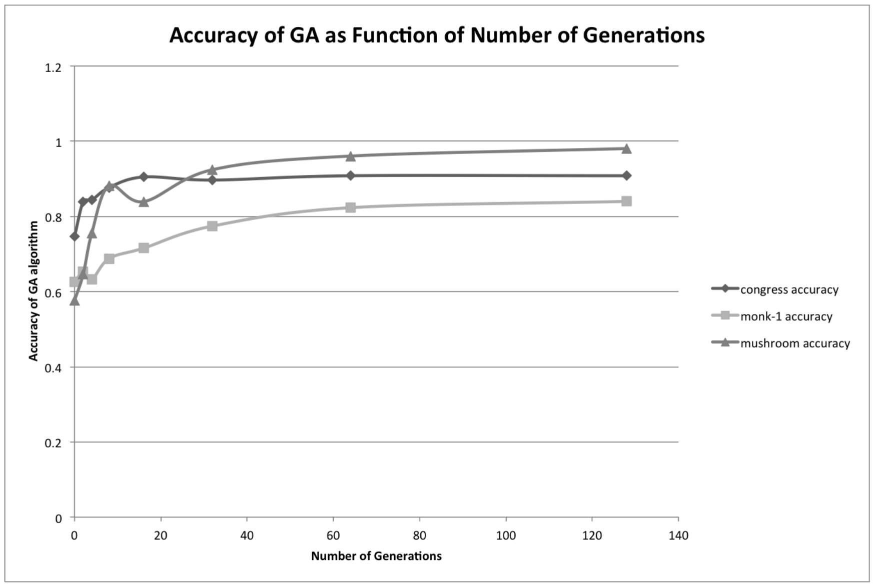

Figure 8: The accuracy of the genetic algorithm was recorded for varying numbers of generations. Each point represents the average of five runs.

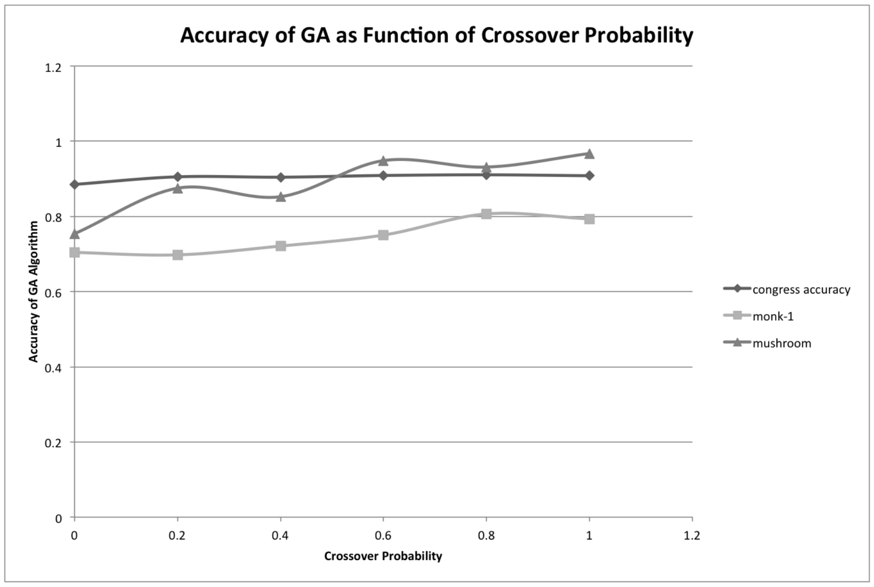

Figure 9: The accuracy of the genetic algorithm was recorded for varying probabilities of crossover. Each point represents the average of five runs. Only three data sets were used for clarity.

5 Discussion

Based on the results represented in figures one through five, there is relatively little difference in performance between the traditional and genetic algorithms, on average. Because the genetic algorithm is probabilistic, performance metrics such as accuracy are subject to what seems to be a discrete normal distribution. In some cases, even when the expected accuracy of the genetic algorithm is lower than that of its traditional counterpart, a particular run of the GA may outperform the deterministic algorithm. One such example was perfect learning (on all three performance metrics) of the Mushroom data by an instance of a genetic algorithm, albeit the traditional algorithm was not far behind.

An advantage of the traditional algorithm, evaluated with either information gain or gain ratio, is that only one tree is considered, and it is always the same size for the same input, making the algorithm essentially linear in the number of attributes and examples. It is also completely deterministic and performs reasonably well on "nice" data sets. The genetic algorithm differs from the traditional one in that it exhibits greater flexibility and robustness, but at the cost of longer runtime and highly application-specific parameter tuning.

It is not recommended to use a genetic algorithm when a simpler, traditional algorithm with similar performance can be applied. Firstly, the genetic algorithm is likely to add (possibly many) redundant nodes, which leads to loss of clarity if a human user wishes to actually view the tree. Secondly, the genetic algorithm requires many tree traversals regardless of how optimized the implementation is. If the number of values most attributes can assume is large, then the branching factor of the tree is high, leading to expensive operations. Of course increasing the population size linearly increases runtime, and increasing the number of generations leads to an expected $O(n^2)$ increase in runtime (depending on how deep traversals go, whether trees are deep copied, how random trees are generated, etc). Lastly, there are several subtle pitfalls that can result from poor parameter choices. For example in Figure 8, choosing too few generations could cause performance to fall 10 to 15 percentage points short of that obtained by true convergence. Likewise, allowing too many top-performing "elites" to be replaced into the next generation would result in lower diversity and likely premature convergence to a suboptimal result. The risk of overfitting the training is quite large. Compare the difference in performance of the GA on the training and test sets in figures one through five; although the author's standard parameters effectively prevent overfitting, the difference could be quite marked. It should be noted that the GA's confidence on training data is most deceptive on the toughest-to-learn data sets (MONK-2 in Figure 4).

However, on data sets that are particularly tough for the traditional algorithms to learn, the genetic algorithm at least possesses the potential to do well, if by random construction. Again in Figure 4, the GA outperformed the TA on the test examples for all three metrics. Another advantage of the genetic algorithm is it considers the global performance of an entire classification tree, which is ultimately more relevant than the local attribute-by-attribute consideration of the greedy traditional algorithm.

Although no recipe for an immediately portable genetic algorithm exists, a couple of generalizations can be made. All else being equal, the most important parameter is the number of generations: Figure 8 shows there is really no upper bound for this, but performance can quickly and unpredictably deteriorate if it is below a certain threshold, say 40 generations. Population size, however, is more flexible based on the results in Figure 7. The author theorizes that for a proper tree encoding, a population as low as 20 can generate optimal results if the number of generations is high enough, all other parameters being equal. An optimal value for elite replacement, if the user decides to employ such a strategy, should be rather low, say around 10 percent. This is especially true if the algorithm mandates a steady population size. The author believes that replacing more than the top 40 percent into every new generation would result in premature convergence, and indeed it would challenge the definition of elite. One must be vigilant that the elites don't all converge to essentially the same tree too quickly; diversity within elites can be leveraged to produce even better trees. Lastly, when selecting individuals for reproductin based on a fitness-proportionate strategy, it is recommended to cap the number of times an elite can be selected, if at all. Without such restraints, it is likely that "incest" would destroy the critical structures that contribute to an elite's good performance.

It is worth investigating why, even with massive population sizes and excessive crossover or mutation rates, a near-optimal decision tree was not produced by the genetic algorithm. Here the author speculates that the structure of a decision tree itself is to blame. One major assumption of a decision tree is that attributes are independent of one another; that the information obtained by partitioning the training set on one attribute does not influence how to partition the data set with other attributes. This is a rather a naive assumption that rarely manifests itself in real world scenarios. If, for instances, examples could be split based on pairs of attributes, or functions applied to pairs of attributes, or even larger subsets of attributes — then perhaps the genetic algorithm could more successful. However, the added exponential growth of power sets would probably make algorithm intractable.

6 Conclusions

Machine learning has progressed tremendously in the design of algorithms for specific purposes under moderately stringent assumptions. Increasing the generality of an algorithm, if it can be done at all, is costly either in terms of the number of parameters or time complexity. Decision trees are no different. Information-theoretic algorithms for decision tree learning perform very well on learning simple concepts from the data, and are flexible enough to handle high dimensionality in the attribute space. However, there comes a point when the complexity of the concept to be learned cannot be achieved by a locally greedy search. A genetic algorithm has the potential to learn a broader class of concepts and is in this way more general. Genetic algorithms are difficult to optimize since there are so many degrees of freedom, and often the optimal parameters are highly application specific. For increasingly complicated search spaces, the time complexity of genetic algorithms increases exponentially because it assumed that the depth of the decision trees grows, as well as the population size and generations. For both of these algorithms, premature convergence to a local optimum is always an issue. In general, one should apply a traditional algorithm first, and only thereafter a genetic algorithm.

References

[1] Mitchell, Tom M. Machine Learning. New York: McGraw-Hill, 1997.

[2] Mohri, Mehryar, Afshin Rostamizadeh, and Ameet Talwalkar. Foundations of Machine Learning. Cambridge, MA: MIT Press, 2012.

[3] Russell, Stuart J, and Peter Norvig. Artificial Intelligence : a Modern Approach. 3rd ed. Upper Saddle River, N.J.: Prentice Hall, 2010.