Symbolic Regression via Pareto Genetic Programming

By
Jacob Tyler Kinsman

Submitted in partial fulfillment of the requirements of the Commonwealth Honors College
University of Massachusetts Lowell
2015

Honors Mentor: Professor Fred G. Martin, Department of Computer Science

_____ Date: _____
Author's Signature

_____ Date: _____
Honors Mentor's Signature

Signatures of Committee Members (at least one):

_____ Date: _____

_____ Date: _____

Contents

1 Acknowledgments

I would like to thank Jeremy Poulin, Michael Stowell, Kaitlyn Carcia, Anthony Salani, Robert Donald, and the Engaging Computing Group at the University of Massachusetts Lowell for their personal and academic support of my research efforts. I sincerely thank Dr. Fred G. Martin for serving as my advisor and challenging me to reach my full academic potential. I am grateful to Dr. Tingjian Ge for serving as a committee member, and offering me his expertise and mentorship within the machine learning and genetic algorithm domains. Finally, I offer my sincerest gratitude to Deborah Kinsman and Chelsea Graham for their encouragement and emotional support.

This material is based upon work supported by the National Science Foundation under Grant Numbers IIS–1123972 and IIS–1123998. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

2 Introduction

2.1 Problem Formalization

The reward of scientific inquiry is the data obtained. Technology has made data collection easier than ever before, facilitating innumerable scientific discoveries. Before any discovery can be made, researchers must fully understand the data they have collected. Data contains a set of empirical observations, where each observation measures a dependent variable with respect to at least one independent variable. The primary knowledge to be gained as the result of any scientific experiment is the set of relationships between these variables, because its characterization lends insight to the process being studied. The increasing availability of data does not address the concern that determining the underlying relationships within it remains a challenging computational problem.

The problem we hope to investigate can be formalized as follows: Given a collection of empirical values pertaining to an independent variable, x, and a dependent variable, y, can we discover a trustworthy regression function that approximates each value of y as a function of its corresponding x value? Specifically, we are interested in doing so without making any a priori assumptions regarding the relationship between the variables x and y. We add this stipulation to remove circular dependency. The choice of a priori assumption restricts the type of relationships that may be discovered. Presumably, this choice cannot be reasonably made, as regression analysis would not be useful in investigative work if the type of relationship between two variables were already known. This circular dependency is common in many industry-standard regression analysis techniques.

2.2 Typical Approach

Describing the relationship in data mathematically is a well-studied statistical problem within the machine learning domain known as regression analysis. It is used to approximate the relationships of variables within data. However, regression techniques require a

hypothesis, a parameterized function of the independent variables, in order to optimize the parameters to best fit the data. A hypothesis forces an a priori assumption regarding the relationship between variables within the data that may or may not be reasonably justified. This requirement typically results in finding suboptimal regression functions that are structured by human intuition instead of objectivity. Many of these techniques also require either explicit or numerical gradients, which may be difficult to compute, or possibly undefined along the interval of the data.

The second shortcoming of statistical regression techniques is the infinite search space of continuous mathematical functions that could describe a relationship between variables in data. As a result, many machine learning models in practice restrict the search space to continuous polynomials (less than a specified degree), exponential, and logarithmic functions. In most cases, this is sufficient, as high-degree polynomials can accurately model a tremendous number of functions, including periodic functions. An accurate approximation of an underlying regression function in data is sufficient for predictive models. It is not necessarily ideal for pedagogical or academic work, where the identification of the true structure in the data is more important than an approximation adequate for predictive work.

2.3 Our Approach: Introduction

This research abandons the traditional regression analysis methods in favor of one that can calculate robust, trustworthy regression functions without a priori knowledge. We achieve this by disseminating the regression analysis problem into two smaller problems: *searching* for the structure of a function that describes the relationship between variables in data, and *optimizing* the parameters associated with that function.

The structure of a suitable regression function is discovered through Symbolic Regression, a genetic algorithm. A genetic algorithm is a metaheuristic for an optimization problem with an infinite search space (Koza, 1992). It works by iteratively refining a randomized set of candidate solutions in a process that mimics Charles Darwin's Theory of Evolution. The

randomized candidate solutions will slowly converge towards accurate regression functions through an evolutionarily-motivated structural refinement. By way of physical analogy, our goal is to evolve accurate regression functions, the same way giraffe and cheetah populations advantageously evolved long necks and strong hind legs, respectively.

The second component of our disseminated regression analysis problem—*optimizing* the parameters associated with the function—will be accomplished via Particle Swarm Optimization (PSO). PSO is an optimization metaheuristic derived from swarm intelligence (Kennedy & Eberhart, 1995). Much like Symbolic Regression, PSO is motivated by a natural phenomenon, specifically the migration and communication patterns of animal species. This is in stark contrast to traditional regression analysis techniques that are derived mathematically. PSO has a number of advantages over traditional numerical optimization techniques, such as scalability to high-dimensional search (Shi & Eberhart, 1999), intuitive simplicity, and greater computational efficiency (Kennedy & Eberhart, 1995). PSO also does not require an explicit or numerical gradient, which prevents restricting the type of regression functions discovered.

2.4 Genetic Algorithms

Before symbolic regression can be analyzed in detail, it is important to fully discuss the mechanics of genetic algorithms, and how symbolic regression fits within this paradigm. The input to the algorithm characterizes the optimization problem, while the implementation details of the primitive operations determine the structure of solutions that are likely to be found. The work flow of a typical genetic algorithm is detailed in Figure 1, located on the next page.

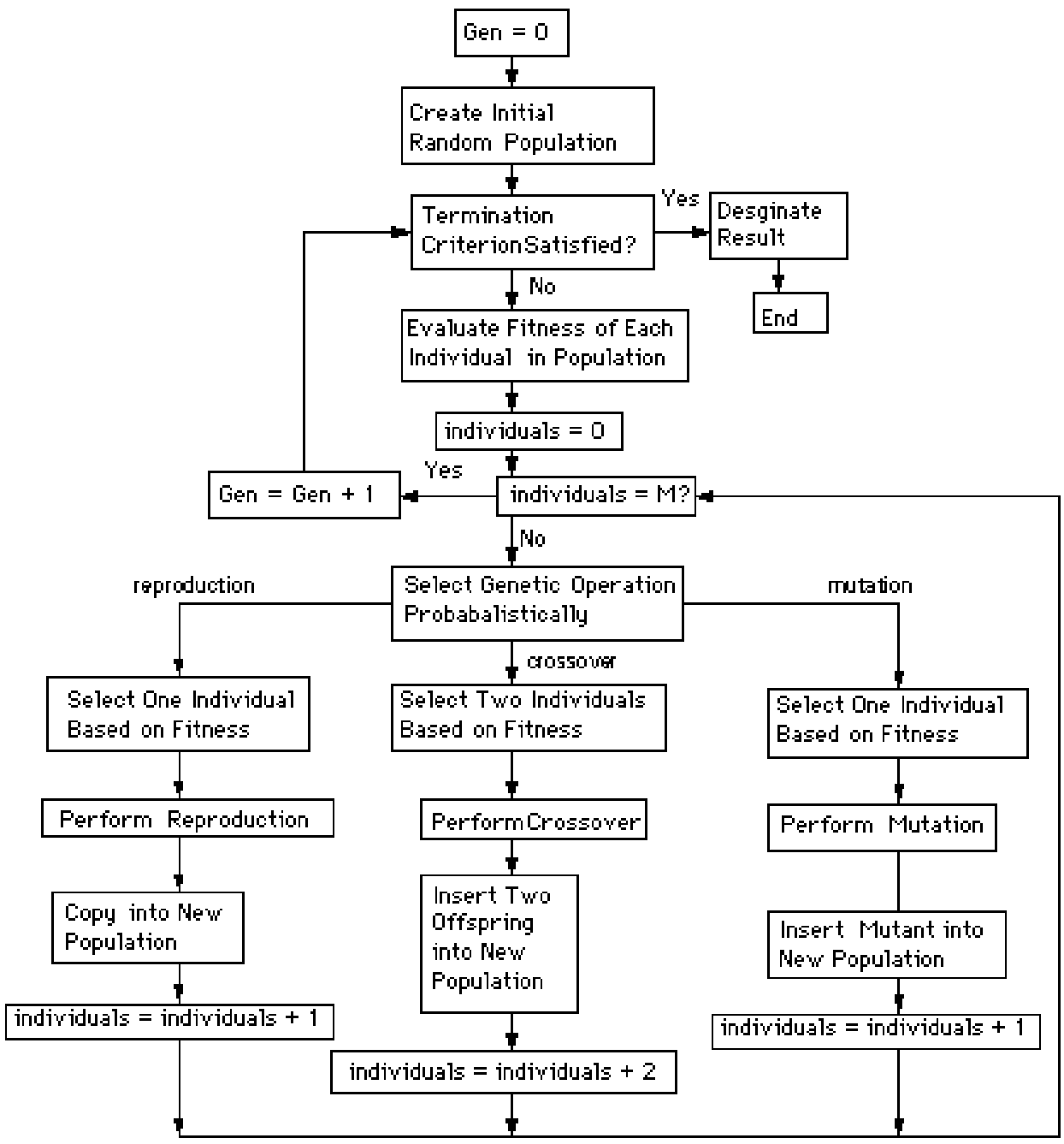

Figure 1: Workflow of a genetic algorithm

2.4.1 Terminals, Operators, and Fitness

A genetic algorithm takes as input a set of terminals, a set of operators, and a fitness function. These entities collectively formalize the search problem (Koza, 1992). The members of the terminal set are literal values that may appear within candidate solutions, such as numbers, strings, and truth values. The members of the operator set are functions that may be applied to one or more terminals. Operators are closed over terminals, meaning that the result of applying an operator to one or more terminals is itself a terminal value. A candidate solution to the proposed optimization problem is constructed from the components specified in the operator and terminal sets. The fitness function assigns a numerical value to a candidate solution that denotes how well it solves the given optimization problem.

2.4.2 Individual

An individual is a candidate solution to an optimization problem. It is the atomic unit within genetic algorithms (Koza, 1992). The solution an individual represents is characterized by the unique, nested combination of operators and terminals it contains. A genetic algorithm initially creates a number of randomly-generated individuals, which we will call a population. The process of evolution is then simulated to produce the next generation of the population. A genetic algorithm iteratively refines newer and newer generations of the population, where individuals within newer generations are typically more adept solutions to the provided optimization problem (Koza, 1992).

2.4.3 Genetic Operators

Genetic operators are primitive operations that a genetic algorithm applies to an individual in the current generation to create an individual in the next generation. These operators are applied probabilistically until a new generation of equal size to the prior one has been created. There are four genetic operators: selection, reproduction (asexual reproduction), crossover (sexual reproduction), and mutation.

2.4.4 Selection

The selection genetic operator chooses an individual within the population to undergo either reproduction, crossover, or mutation. The selection process is crucial because it ensures the algorithm performs better than random search (Koza, 1992). It is responsible for preserving the Darwinian notion of 'survival of the fittest,' in that individuals whose fitness metric is higher have a greater probability of being selected. It is equally important that the selection mechanism employed is not overly biased towards fit individuals because the population will otherwise become too homogeneous, leading to the discovery of local optima as opposed to global optima (Koza, 1992).

2.4.5 Reproduction

During reproduction, the selected individual is placed into the next generation of the population unmodified. The reproduction operator is designed to simulate the asexual reproduction that occurs in nature. This mechanism is desirable because it reinforces the Darwinian notion of 'survival of the fittest,' while also ensuring that the most fit individual found, the nearest-optimal solution, exists within the last generation to be returned. The reproduction operation is quintessential in creating populations of individuals converged on smaller, more optimal regions of the search space (Koza, 1992).

2.4.6 Crossover

During crossover, two individuals are first selected as parents. The two parents then sexually reproduce two children in a process that mimics genetic recombination. Specifically, the first child contains a randomly-selected portion of each parent. The second child then contains the sections of each parent that were not used to compose the first child. A combination of two fit solutions to an optimization problem is likely a fit solution itself (Koza, 1992).

2.4.7 Mutation

Mutation is a minor genetic operator, and is applied with far lower probability than either reproduction or crossover. During mutation, a randomly-chosen location within the selected individual is chosen as a mutation site. That portion of the individual is then deleted, and replaced with a new randomly-generated substructure. The mutant is then added into the next generation of the population. While mutation on its own is unlikely to produce fit individuals, it is useful in preserving genetic variance, which is important (alongside a proper selection mechanism) in preventing future populations from becoming too genetically homogeneous (Koza, 1992).

2.5 Symbolic Regression

Symbolic Regression is a genetic algorithm for function discovery. It is used to identify a trustworthy regression function that fits the provided data without any a priori knowledge, as shown below in Figure 2.

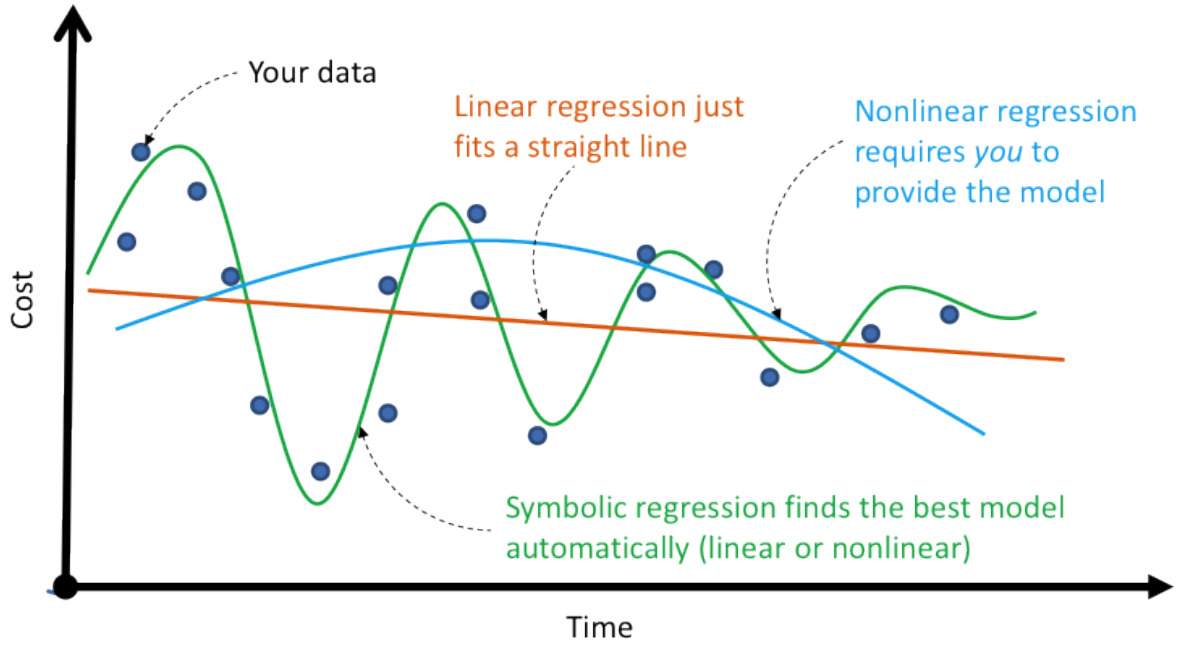

Figure 2: A data set illustrating the advantages of Symbolic Regression over traditional regression analysis techniques

We will use Symbolic Regression to solve the first half of our disseminated regression analysis problem–*searching* for the structure of a function that describes the relationship between variables in data. The terminals in the symbolic regression algorithm are x and the ephemeral constant. The variable x denotes the independent variable of the regression, and the ephemeral constant is a placeholder that will be replaced by an appropriate constant value during numerical optimization (Kommenda, Kronberger, Winkler, Affenzeller, & Wagner, 2013).

The operator set consists of domain-protected mathematical functions, including addition, subtraction, multiplication, division, exponentiation, logarithm, square-root, and trigonometric functions. Domain-protection ensures that functions which are not closed over $\mathbb{R}$ have their inputs preprocessed in such a way that closes them —for example, taking the absolute value of the inner expression of the logarithm. Domain protection is a necessary compromise in Symbolic Regression to prevent floating point errors during evolution (Koza, 1992).

The fitness function for Symbolic Regression is typically the sum of squared-errors function (SSE) prevalent in the machine learning literature, given by:

$$SSE = \sum_{i=1}^{n} (t_i - y_i)^2 \tag{1}$$

where the vector t contains truth measurements of the dependent variable, and the vector y contains the result of evaluating the individual at the i^{th} measurement of the independent variable. While this function seems intuitively ideal—calculating fitness by the squared differences between the regression function and the dependent variable for each measurement of the independent variable—it is flawed in that the R^2 value of a regression function is inversely-proportional to its fitness value. While it is possible to phrase the genetic algorithm as a minimization problem instead, we opt to normalize fitness values such that they increase

monotonically with R^2 value for mathematical convenience, as given by:

$$Fitness = \frac{1}{1 + SSE} \tag{2}$$

which reformulates Symbolic Regression as a maximization problem by mapping an individual's fitness to the interval [0, 1], where the extrema of the interval correspond to high and low residual error, respectively.

The last crucial component of formalizing Symbolic Regression is the structure of the individual, or candidate solution to the optimization problem. An individual is formatted as a Lisp symbolic expression (s-expression), whose in-order traversal yields a valid mathematical function. The terminal nodes are members of the terminal set, whereas the internal nodes are members of the operator set. Each sub tree can be thought of as being recursively parenthesized, meaning that nesting within the s-expression overrides traditional order of operations. Figure 3 exemplifies a number of sample individuals.

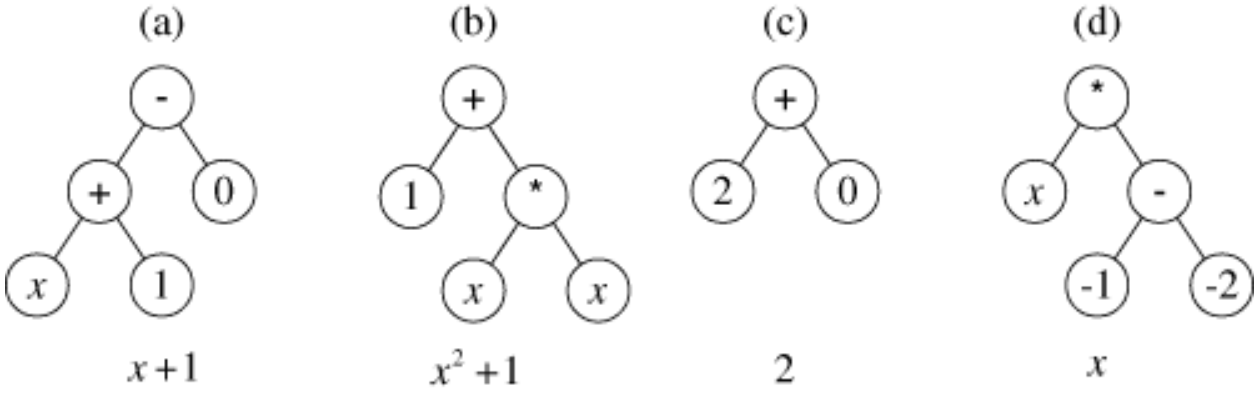

Figure 3: Example Lisp s-expressions that represent various mathematical functions

2.6 Particle Swarm Optimization

Once the structure of the regression function is identified, its associated parameters are optimized via Particle Swarm Optimization, a optimization metaheuristic (Kennedy & Eberhart, 1995). Unlike traditional numerical optimization procedures, PSO does not

require either explicit or numerical gradients. If we chose a technique that required a form of gradient, we would need to either abandon our preference to not enforce a priori assumptions, or implement a computer algebra system (CAS) capable of differentiation. The process of dynamic differentiation, coupled with traditional numerical optimization would result in a regression system much more expensive in terms of both running and development time. This trade off seems unreasonable given the effectiveness and scalability of PSO shown in Shi and Eberhart's paper, "Empirical Study of Particle Swarm Optimization." This paper also shows that PSO generalizes well to high-dimensional optimization, which turns out to be particularly desirable in the context of Symbolic Regression.

Much like Symbolic Regression, the intuition behind Particle Swarm Optimization is best explained via physical analogy. Formally, Symbolic Regression finds a function with n associated parameters, and Particle Swarm Optimization searches for a vector $v \in \mathbb{R}^n$ such that substituting the elements of v in the corresponding coefficient locations within the function maximize the function's goodness of fit.

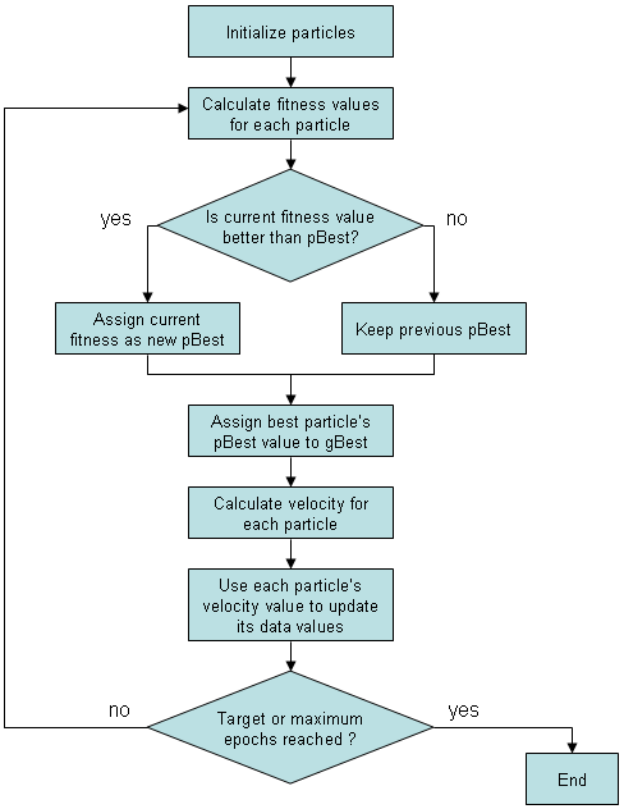

Figure 4: Flowchart describing the PSO algorithm in pseudocode

By way of physical analogy, imagine a group of mice in small colonies distributed over a section of land. PSO initializes a set of particles, which can be thought of as a family of hawks that fly above the section of land and look down upon it. The hawk's location, which we represent abstractly as a vector of coefficients, has a certain goodness associated with it. Since the hawks are hungry, the goodness of a hawk's location is given by the number of mice immediately below it.

The hawks can communicate with one another, and must use their communication to help other hawks approach a region with the most mice. PSO guides the hawks by having each hawk iteratively fly some distance in a new direction, where the magnitude and direction of flight is specified by a combination of moving towards the location where the hawk currently moving saw the most mice (personal best), and the location where any hawk in its family has seen the most mice (global best). The comparative weighting in the linear combination of personal and global bests then adjusts the rate of exploration and exploitation, respectively (Kennedy & Eberhart, 1995).

To formulate PSO mathematically, the land on which the mice reside is $\mathbb{R}^n$, the number of mice on a point within $\mathbb{R}^n$ is the R^2 value of the function found by symbolic regression when its parameters are set to the values specified at that point, and the hawks are the randomly-initialized particles that perform stochastic search over the problem's search space (Kennedy & Eberhart, 1995). The algorithm is described in pseudocode in Figure 4, located on the previous page.

2.7 Research Goals

This research aims to extend the analysis capabilities of the iSENSE project, a collaborative, online data visualization platform. iSENSE features a suite of visualizations that encourage data exploration and analysis through interactivity. iSENSE offers both a time line and scatter plot within its visualization suite, which currently includes a traditional regression model. We seek to extend the regression analysis tools within iSENSE to contain Symbolic Regression functionality, and use iSENSE as a platform to test the algorithm's pedagogical effectiveness in a user study.

Additionally, this project seeks to explore the comparative effectiveness of different enhancements and variations for Symbolic Regression. While the algorithm has long-since been established as an effective strategy for the regression analysis problem, the best practices regarding the implementation of its components, including numerical optimization, are still

open in research.

2.8 Research Questions

We hope to answer three major questions as a result of conducting this research. First, is symbolic regression a practical regression strategy for the analysis of visualized data? Second, which variations of the symbolic regression algorithm are most generally effective in balancing model accuracy and simplicity? Third, is a symbolic regression model useful in the pedagogy of data science? The success of our project is based upon the ability to answer these questions with the data obtained from our software implementation and user study.

3 Software Implementation

3.1 Baseline Implementation

Our baseline Symbolic Regression implementation for this research project aims to extend the iSENSE regression model to include Symbolic Regression functionality. Since iSENSE visualizations are interactive by design, our implementation must find a suitable regression function in mere seconds, where most systems may take minutes or hours. As a result, we will be limited to a smaller portion of the search space. This will likely result in discovering less accurate regression functions. We hope to improve the accuracy of discovered models by using PSO. This research also hopes to survey a number of different implementation strategies for reproduction, crossover, and fitness in order to provide empirical data regarding their comparative effectiveness. The entirety of the software implementation was written in CoffeeScript, a functional language that transcompiles directly to JavaScript.

3.1.1 Reproduction Techniques

The first reproduction technique surveyed is fitness-proportional reproduction. As its name implies, the algorithm reproduces individuals from the population with probability proportional to its fitness. Fitness-proportional reproduction was first proposed by John R. Koza Jr., the founder of genetic algorithms. The algorithm first computes the fitness of each individual within the population, and uses these values to create a cumulative probability distribution function (CDF). A random number in [0, 1) is chosen, and the individual reproduced is the first one whose cumulative fitness is less than the random number chosen. The algorithm reproduces fit individuals a considerable percentage of the time—a desirable characteristic for reproduction algorithms known as selection pressure (Koza, 1992). However, fitness-proportional reproduction does not exhibit minimum spread. Minimum spread is a desirable quality in reproduction algorithms because it ensures unfit individuals are also reproduced with appropriate probability, ensuring future generations do not become

genetically homogeneous (Man, Tang, & Kwong, 1999).

The second reproduction technique surveyed is tournament reproduction. Tournament reproduction is governed by two parameters, a tournament size and probability. The algorithm randomly samples individuals from the population until a sub group of size equal to the tournament-size parameter is created. The algorithm then reproduces the most fit individual within the tournament with a probability equal to the tournament-probability parameter. If the most fit individual is not reproduced, the second most-fit individual is reproduced with probability equal to the tournament-probability parameter squared. This process is repeated until an individual is chosen. Tournament Reproduction is a strong hybrid technique because it offers configurable selection pressure (based upon the tournament size parameter) with guaranteed minimum spread (Miller & Goldberg, 1995).

The final reproduction technique surveyed is stochastic universal sampling (SUS). SUS is an ideal candidate for Symbolic Regression since it can perform an arbitrary number of reproductions simultaneously. The number of reproductions to be performed, n, is a parameter of the algorithm. SUS builds a fitness-proportional cumulative distribution function, similar to the fitness-proportional reproduction algorithm. It then discretizes the CDF into n equally-sized regions. SUS then generates a random number between zero and $\frac{1}{n}$. The first individual to be reproduced is the first individual whose cumulative fitness is less than the random number chosen. The second individual to be reproduced is the first individual whose cumulative fitness is less than twice the random number chosen, but greater than the cumulative fitness of the first individual chosen. This process is repeated to perform n simultaneous reproductions. SUS is computationally efficient because it can perform multiple reproductions from a single cumulative distribution function. SUS is also ideal because it provides both selection pressure and minimum spread (Grefenstette, 2013). The SUS reproduction process is outlined in Figure 5.

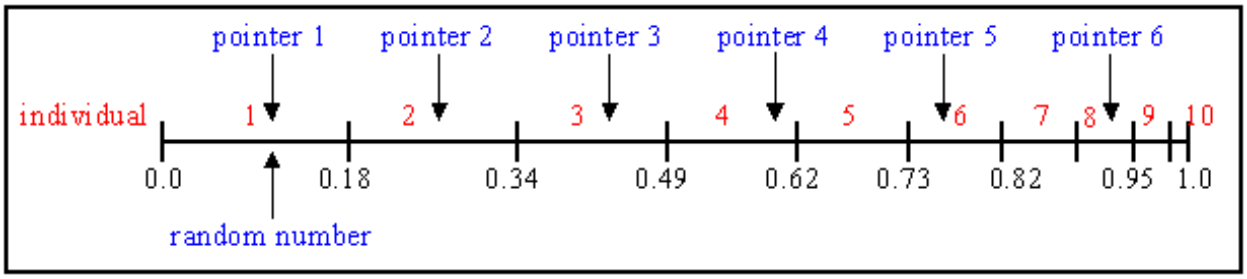

Figure 5: This figure illustrates a sample SUS run over a CDF with ten individuals. In this case, individuals one, two, three, four, six, and eight will be reproduced.

3.1.2 Crossover Techniques

The first crossover technique surveyed is cut and splice crossover, initially proposed by John R. Koza Jr. (1992). This approach generates two random numbers uniformly between zero and the size of the first and second parents, respectively. The point chosen on each parent individual is referred to as its crossover point. Two new children are then formed, such that the first child is the portion of the first parent before its crossover point, and the portion of the second parent after its crossover point. The second child formed is the portion of the first parent after its crossover point, and the portion of the second parent before its crossover point.

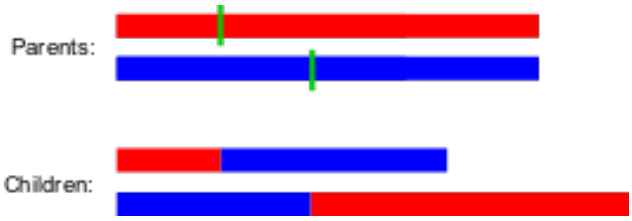

Figure 6: A visual representation of the cut and splice crossover algorithm

The second crossover technique surveyed is single point crossover, also initially proposed by Koza. This approach generates a single crossover point between zero, and $min(s_1, s_2)$ where s_1 and s_2 are the size of the first and second parents, respectively. Two new children are then formed by swapping the portion of the two parents after their crossover point. Cut and splice crossover is believed to be superior to many alternative techniques, because the children will likely have different sizes than its parents, helping to preserve genetic variance within the population (Koza, 1992).

The final crossover technique surveyed is two point crossover, also proposed by Koza. This approach generates a crossover point, θ_1, uniformly between zero, and $min(s_1, s_2)$ similar to the single point crossover algorithm. However, a second crossover point, θ_2 is then generated uniformly between θ_1, and $min(s_1, s_2)$. Two new children are then formed by swapping the portion of the two parents between their θ_1 and θ_2 crossover points, as shown in Figure 8, located below.

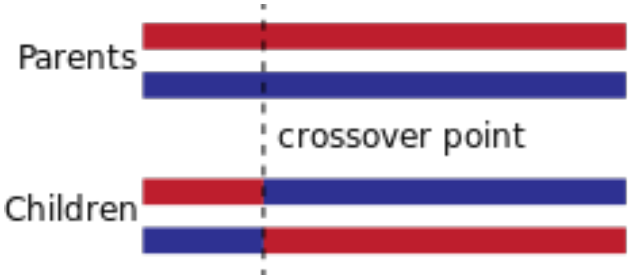

Figure 7: A visual representation of the single point crossover algorithm

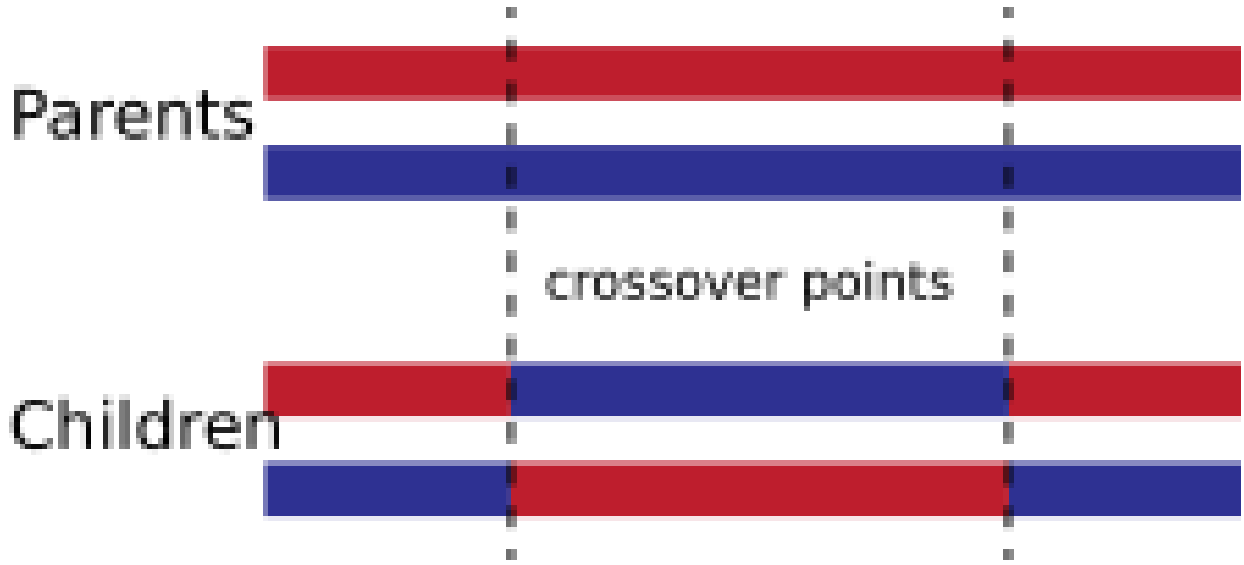

Figure 8: A visual representation of the two point crossover algorithm

3.1.3 Mutation Techniques

The mutation operator is performed very sparingly, and is used to preserve genetic variance as future generations converge towards near-optimal solutions (Koza, 1992). Since good individuals are unlikely to be produced via mutation, minimal research has gone towards the development of new techniques. The only technique surveyed is Koza's Point

Mutation algorithm, which simply replaces the subtree within the individual beginning at a randomly-chosen position with a new randomly-generated tree. This technique has been shown to successfully preserve genetic variance (Koza, 1992). The technique is illustrated on a sample individual in Figure 9.

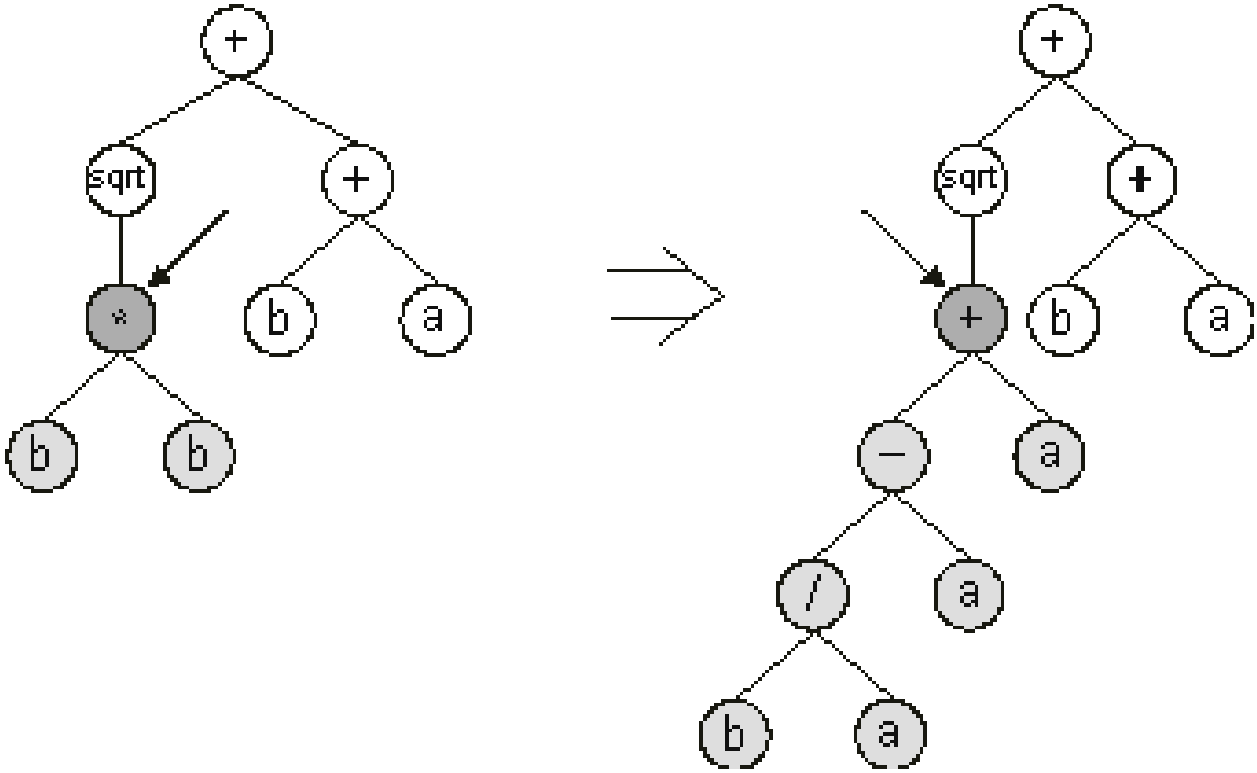

Figure 9: A visual representation of the point mutation algorithm

3.1.4 Fitness Techniques

The first fitness technique surveyed is the canonical sum of squared-errors (SSE) function, given by (2). This function is a strong baseline implementation because it is intuitively straight forward, and commonly used within the Machine Learning domain. While SSE effectively identifies fit individuals, it is inadequate in that it cannot identify individuals that are near optimal—for example, a function with the correct order of magnitude, but improper coefficients. In cases with a large number of data points, SSE may deem near-optimal regression functions remarkably unfit.

The second fitness technique surveyed is the mean squared-errors (MSE) function, given by

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(t_i - y_i)^2 \tag{3}$$

which is essentially the average squared-error between a data point, t_i, and its corresponding truth value, y_i. MSE is another strong baseline implementation in that it is commonly used in practical Symbolic Regression implementations (Keijzer, 2004). MSE is typically preferable to SSE mathematically because it does not penalize near-optimal regression functions as heavily, making them more likely to become the structural basis for individuals in future generations (Koza, 1992).

The third fitness technique surveyed is the normalized mean squared-error (NMSE) function, given by

$$NMSE = \frac{1}{n}\sum_{i=1}^{n}\frac{(t_i - y_i)^2}{\bar{t} * \bar{y}} \tag{4}$$

which seeks to assign higher fitness to near-optimal regression functions by normalizing by the difference in means between the vector t and the vector y. NMSE also prevents unjustly penalizing near-optimal regression functions by taking the average of squared-errors, as opposed to the total. While NMSE is not used as frequently, it has shown to be an effective fitness function for Symbolic Regression (Vladislavleva & Smits, 2009).

This research also surveys scaled fitness and pareto fitness, two more advanced fitness-calculation techniques recently introduced within the genetic algorithms literature. These techniques are discussed in sections 2.2.1 and 2.2.2, respectively.

3.1.5 Selection Techniques

Selection and reproduction are the same process within the context of Symbolic Regression. This is not the case in all genetic algorithms (Koza, 1992). As a result, this research surveys the fitness-proportional, tournament, and stochastic universal sampling selection algorithms. These algorithms each behave as previously described. This simplification seemed

reasonable because many Symbolic Regression implementations opt to use the same strategy for both selection and reproduction in practice.

3.2 Enhancements

Symbolic Regression is effective in discovering accurate, robust, and trustworthy regression functions. However, the iSENSE visualization system is interactive, and therefore does not accommodate for the running time Symbolic Regression requires to discover an adequate regression function. A result is produced in mere seconds, where many similar models take minutes or hours. We hope to compensate for the (comparatively) small amount of the search-space we can explore with multiple numerical optimization techniques. For this approach to be effective, we require a mathematical distinction between poor regression functions, and accurate regression functions with suboptimal parameters. Assuming this distinction can be made, we then seek a regularization mechanism to prevent our model from over fitting to the data. These two concerns are addressed by the first two Symbolic Regression enhancements our work implements: scaled and pareto fitness.

3.2.1 Scaled Fitness

Scaled fitness addresses the challenge of distinguishing between poor regression functions, and accurate regression functions with suboptimal parameters. Scaled fitness is an individual evaluation technique developed by Maarten Keijzer in his 2004 paper, "Scaled Symbolic Regression." The technique is developed from the observation that the covariance between the values of the dependent variable, t, and the values of our regression function, y, can be used to find a coefficient and offset that minimize the residuals of any regression function (Keijzer, 2004). Keijzer contributes an additional theoretical result in this paper of great importance—scaled fitness performs as well as mean-squared error in the worst case (Keijzer, 2004). The offset and coefficient can be calculated closed-form, and are guaranteed to improve the fitness of any individual in all but the worst case—when the individual's fitness

is already perfect. Scaled regression returns 0 and 1 for its offset and coefficient in order to ensure the regression function remains unchanged if it is already perfectly fit. The scaled fitness equations are detailed below.

$$ScaledFitness = \frac{1}{n}\sum_{i=1}^{n}(t_i - (a + by_i))^2 \tag{5}$$

$$a = \bar{t} - b\bar{y} \tag{6}$$

$$b = \frac{cov(t, y)}{var(y)} \tag{7}$$

3.2.2 Pareto Fitness

One shortcoming of Symbolic Regression is that it does not possess a regularization mechanism. Regularization is highly desirable in regression models to prevent over fitting to the data. Over fitting results in regression functions that poorly generalize the structure within the data.

To address over fitting, we seek to consider some other property regarding the individual when calculating its fitness. Specifically, we would like our fitness metric to consider both the accuracy of the regression—denoted by its R^2 value—and its expression complexity, a metric defined to characterize the number of non-linear operations a function contains. We achieve this by assigning a complexity cost to each operator in the operator set, and penalizing subtrees recursively based upon the operator at the root of the tree. Despite being a simple heuristic, calculating an expression complexity allows us to immediately create a numerical measurement of the complexity of a function, and consider this value when assigning fitness. Our system uses a normalized linear combination of an individual's scaled fitness and expression complexity to calculate its fitness, which will guide the search space towards accurate regression functions of simpler structure, effectively providing regularization.

In considering both the accuracy and complexity of an individual when evaluating its fitness, we have reformulated Symbolic Regression as a pareto (multi-objective) optimization

problem. We seek to concurrently maximize an individual's goodness-of-fit (equivalent to minimizing its inaccuracy) while also minimizing its expression complexity. The set of individuals we are interested in are located within the pareto front, ones that are fit according to all evaluation metrics. The pareto front for Symbolic Regression is explained in Figure 10.

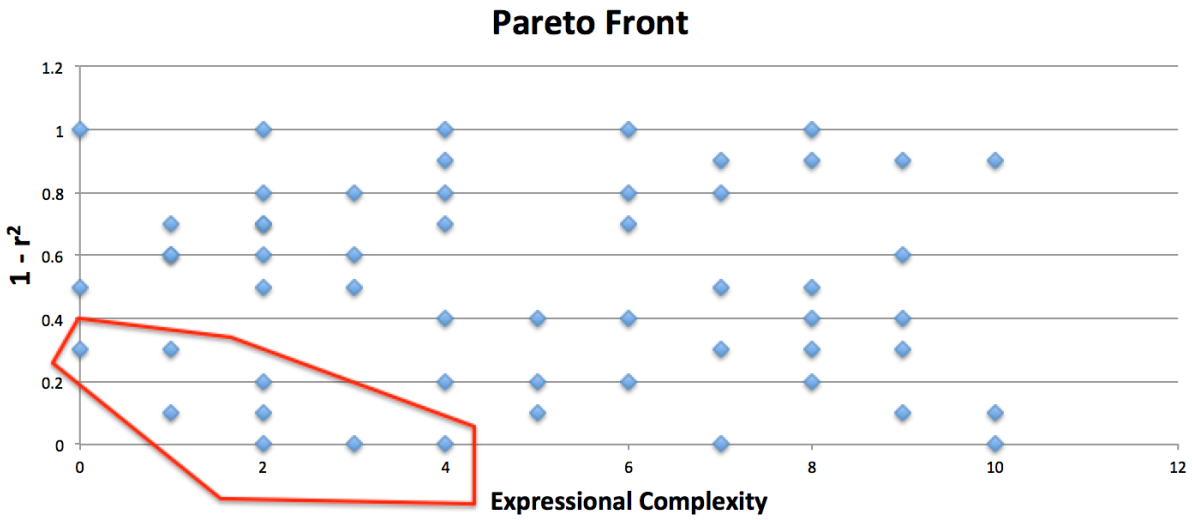

Figure 10: Plot of R^2 value as a function of expression complexity. The data points in the upper-left are too simple. The data points in the lower-right are over fit. The data points in the upper-right are poor regression functions. The data points circled in red are simple and accurate regression functions. This is the pareto front for Symbolic Regression.

3.2.3 Particle Swarm Optimization

The final Symbolic Regression enhancement implemented is the Particle Swarm Optimization algorithm for the optimization of constants. During PSO, the ephemeral constants within an individual are replaced with a set of constants that are near-optimal in maximizing an individual's goodness of fit. Since scaled fitness will further optimize an individual's goodness-of-fit, near-optimal parameters are sufficient, saving us a considerable amount of time during the optimization process. While typical PSO techniques use a large swarm, our implementation favors a neighborhood-based approach, involving multiple smaller swarms. Smaller swarms typically result in less accurate parameters, but are more computationally

efficient (Kennedy & Eberhart, 1995). Running multiple small swarms increases exploration over the search space, increasing the probability one of the particles will converge to near-optimal parameters. Our implementation flies five neighborhoods of ten particles each over one hundred iterations.

Now that neighborhood-based PSO is formalized, our full numerical optimization approach can be explained. An individual's fitness is calculated in two steps. First, neighborhood-based PSO is used to find approximate parameters. Once those parameters are found, we use scaled fitness to further enhance the individual's goodness of fit. The approach is outlined in Figure 11.

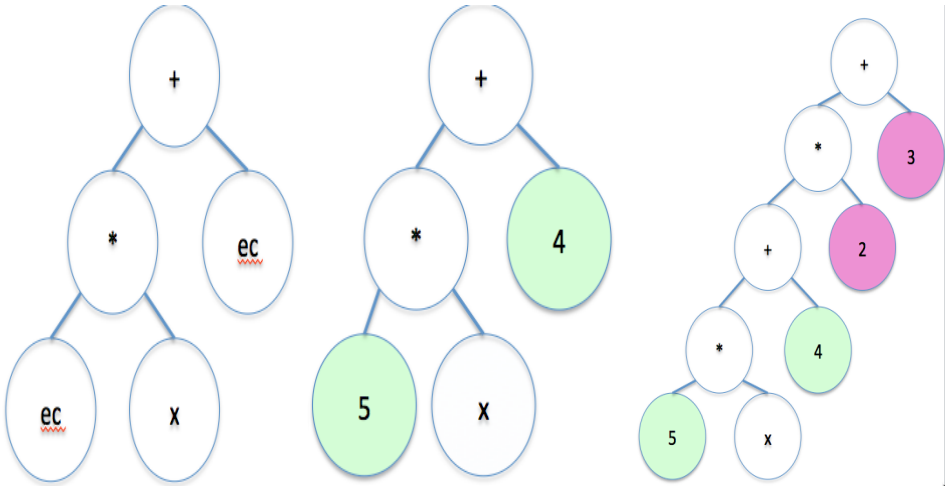

Figure 11: Sample numerical optimization procedure for Symbolic Regression discovering the function *f(x) = 10x + 11*. The first image shows an individual that was discovered using Symbolic Regression. The second image shows the individual after PSO has been performed. The third image shows the individual after both PSO and scaling have occurred. The target function is discovered after PSO and scaling are performed.

4 User Study

4.1 Overview

This research conducted a user study to investigate our third research question—is a symbolic regression model useful in the pedagogy of data science? The experiment enrolled ten students in science, technology, engineering, and mathematics (STEM) undergraduate degree programs. Each participant was asked to watch a series of video tutorials for the iSENSE visualization suite, and then use the system to complete two data visualization and analysis exercises. Each exercise asked ten questions about one of two fictitious data sets created for the purpose of the study. The first five questions of each exercise tested the participant's ability to answer general questions pertaining to the data set. The second set of five questions tested the participant's ability to identify the type of mathematical relationship between specific variables within the data. The ten users were equally split into an experimental and control group. The experimental group used a version of iSENSE extended with Symbolic Regression, whereas the control group only had access to the currently-existing regression model.

4.2 Analysis

It is important to describe the goal of our study—and its inherent strengths and weaknesses—before we discuss any empirical data and observations. The goal of this study is to determine the pedagogical effectiveness of Symbolic Regression models in data analysis. Since our study was limited to ten participants split into two groups, we will not be able to contribute results that are statistically-significant to any meaningful probability. This research aims to determine if Symbolic Regression has the potential for effective pedagogical use as an interactive analysis tool. We hope our method and results will encourage and shape future academic work on the subject.

While each group had their own advantages and challenges during the study, a number

of mistakes were shared equally by members of both groups. The most prevalent mistake in this category is over fitting. Despite thorough explanation during the video tutorials on the problems of over fitting—and how it can be detected—every student enrolled in the study made at least one over fitting mistake, with many participants over fitting relationships on multiple occasions. Over fitting is an easy mistake to make, but could be avoided with further regression analysis instruction that lies outside the scope of this study. Participants in each group also frequently made dependency mistakes, which involve placing the dependent and independent variables on the opposite axes. While in most cases this mistake is accidental, video footage of two participants, one from each group, suggests the mistake can be cognitive. This was not a mistake we expected to encounter because topics such as dependent and independent variables, correlation, and causation are covered in many introductory-level high school and college statistics courses. These two observations alone suggest that Symbolic Regression's pedagogical effectiveness may increase as students gain more knowledge in statistics.

Participants in the experimental group had a marginal advantage over participants in the control group. Watching participants interact with the Symbolic Regression model provided insight regarding the properties that were pedagogically valuable. First, all participants in the experimental group successfully found and used the Symbolic Regression controls without being explicitly instructed on how to do so. This suggests the front-end implementation within the regression controls seemed natural. It also indicates that the Symbolic Regression functionality is deemed reasonable and reliable to participants without the requisite knowledge to realize that objective function discovery is rare within such systems, and difficult to implement in practice. This assumption is a double-edged sword, however, as many participants were unaware of Symbolic Regression's shortcomings as a result of this lack of background knowledge.

The second major insight gained from watching participants interact with the Symbolic Regression model was that it was used in two very different ways. Two participants used

Symbolic Regression to drive their analysis, whereas the other three used it to verify their own hypothesis.

The former two participants exhibited a clear pattern in using the model. They would set up the axes to display the desired relationship, and immediately create a Symbolic Regression. If the regression generated seemed satisfactory to them, they would record the answer. If the regression was unsatisfactory, they would then go about creating typical regressions similar to the one discovered. While this analysis technique uses Symbolic Regression exactly as it is intended—for objective function discovery without a priori knowledge—it is not necessarily ideal. These participants often ran into problems when Symbolic Regression was unable to provide an accurate fit to the data. Although this occurred infrequently, it would occur more often given more complex relationships. It is problematic because the student would then explore regression functions near a local optima as opposed to the global optima, which may result in diverging their search as opposed to converging it.

The latter group of three participants also exhibited a clear usage pattern. They would first set up the axes, then use the traditional regression controls. Once they found a regression structure they believed to be correct, they would then create a Symbolic Regression to verify their hypothesis. This technique is ideal because it is skeptical towards the Symbolic Regression model. Symbolic Regression is a heavily randomized algorithm, and is not guaranteed to find an optimal relationship between variables. These participants correctly classified more regression functions on average. Additionally, this inherent skepticism towards the functionality led two of the three participants to realize that Symbolic Regression is randomized, and therefore has the potential to produce different regression functions. These two participants both performed extraordinarily well in the regression analysis portion of the assessment.

The only challenge specific to participants in the experimental group was that the majority of the participants were too trusting of the regression functions discovered by Symbolic Regression. This trust occasionally resulted in erroneous classifications. While regression models must always be used with skepticism, this is especially the case with Symbolic Re-

gression. While this error can be corrected in part with a greater knowledge of statistics, the tool would be more effective if the participant knew *exactly* why such skepticism was necessary, which would require disclosing that Symbolic Regression was used, in addition to explaining the complex algorithm itself. This observation lends support to the idea that Symbolic Regression may be more pedagogically useful to students with a strong background in statistics and regression analysis.

The most indicative data pertaining to Symbolic Regression's pedagogical value is summarized in Figure 12, which outlines the performance of participants in the experimental and control groups on the data analysis questions. The entirety of the performance data can be seen in Figures 13 and 14 on the next page. Participants in the experimental group successfully classified optimal regression functions more often than participants in the control group, indicating that future research on the pedagogical effectiveness of Symbolic Regression may be fruitful. Performance on the general questions component of the assessment does not seem to predict performance on the regression questions component, although a larger sample of participa1nts is required to substantiate this claim.

Group	General Questions (per cent correct)	Regression Questions (per cent correct)
Experimental	82	62
Control	86	48

Figure 12: Summary of user study assessment performance by participant group

Participant Id	Group	General Questions (per cent correct)	Regression Questions (per cent correct)
1	experimental	100	40
2	experimental	80	20
3	experimental	80	80
4	experimental	80	80
5	experimental	100	40
6	control	80	80
7	control	80	60
8	control	80	40
9	control	100	60
10	control	100	20

Figure 13: User study assessment performance on the first of two data sets

Participant Id	Group	General Questions (per cent correct)	Regression Questions (per cent correct)
1	experimental	100	60
2	experimental	40	60
3	experimental	100	100
4	experimental	40	60
5	experimental	100	80
6	control	80	60
7	control	80	20
8	control	100	40
9	control	80	60
10	control	80	40

Figure 14: User study assessment performance on the second of two data sets

5 Conclusion

5.1 Research Evaluation

This research project was largely successful. A symbolic regression model was successfully designed and implemented into iSENSE. An accurate regression function is successfully discovered in a mere 4—7 seconds, as opposed to the minutes or hours required by more powerful models. The short running time of this implementation lends itself to use in an interactive context, and has shown promising pedagogical value. Section 6.2 contains trial data using our Symbolic Regression implementation to discover a set of sample target regression functions.

We had to make a number of compromises to the Symbolic Regression algorithm to achieve this impressive running time. For example, we use a population size of one hundred individuals over one hundred generations. These painfully-small parameters mean we explore a mere ten thousand candidate solutions in the best case, and many fewer in practice. Even with a heuristic-based search, this is simply not enough exploration to consistently discover accurate and trustworthy regressions. As a result, our implementation works abysmally without numerical optimization by PSO and scaling. We owe a large part of the model's accuracy to these numerical optimization techniques.

The second major sacrifice we made to achieve this running time was to only perform PSO on the single most fit individual discovered. Neighborhood-based PSO is a pseudo linear-time algorithm, which is computationally-efficient by numerical optimization standards. However, the run-time coefficient hidden in PSO's time complexity makes it impractical to perform liberally within a limited execution window. This simplification leads to candidate individuals being assessed based upon their scaled fitness with the randomly-chosen ephemeral constant value. This may lead to fit unjustly penalizing individuals with high potential fitness, and failing to properly explore their locality within the search space. Intuitively, our approach places all of its faith within the single candidate that looks most promising at first glance.

The last shortcoming of our work is the language of implementation. Given the entirety of this research occurred during one semester, we desired a high-level language with rapid prototyping that would be simple to integrate with the iSENSE visualization system, which is written primarily in JavaScript. Our implementation uses CoffeeScript, a high-level scripting language with direct JavaScript trans-compilation. In addition to running slightly less efficiently than pure JavaScript, CoffeeScript is inherently limited in its efficiency because it runs in a web browser. Browsers typically do not permit JavaScript to run across multiple threads. This results in a singly-threaded implementation of an algorithm that has been shown to gain substantial performance benefits from concurrency in practice (Zalzala & Green, 1999).

5.2 Future Work

The first enhancement we would like to implement is to rewrite the algorithm in a more efficient language, such as C++. In addition to the obvious performance increase provided by working in a more efficient language, porting the implementation to C++ would allow us to run the algorithm on multiple threads. This could be implemented in one of three ways. First, separate threads could run the algorithm independently. Synchronization would then simply require selecting the most fit individual discovered by any of the threads. The second method also involves running the algorithm on separate threads, but then synchronizes the population on each thread after they have performed one iteration of Symbolic Regression. These techniques are both proven to provide efficient search space exploration (Zalzala & Green, 1999). The population is then filled greedily with the most fit individuals among the populations on each thread. The final technique runs the algorithm on one thread, and spawns three child threads to perform crossover, reproduction, and mutation respectively. We are most interested in exploring this technique, as it may gain insight regarding the exact amount of search space exploration before Symbolic Regression typically becomes effective.

The second enhancement we would like to implement pertains to our pareto fitness im-

plementation. Our current implementation has proven effective in practical usage, although it may encounter issues when working with larger data sets. As the number of data points increase, so does the expected residual error. As a result, our choice to implement a constant penalty mechanism based upon the structure of the tree while calculating fitness proved challenging. When average residual error is low, the individual's expression complexity did not receive enough consideration. When average residual error is high, as is the case with large data sets, the individual's expression complexity will be comparatively much larger than its normalized fitness. This could result in favoring simpler models over more complex ones when the increase in complexity is justified given the improvement in accuracy. The technique is inherently flawed, although it was standard just a few years ago. We did not anticipate this shortcoming as being an issue, since iSENSE is primarily tailored towards pedagogical analysis in a secondary education setting, where this shortcoming would not be an issue.

Given this challenge, we would like to rework pareto fitness in the manner described by E.J. Vladislavleva and G.F. Smits in their 2009 paper, "Order of Non-linearity as a Complexity Measure for Models generated by Symbolic Regression via Pareto Genetic Programming." The paper proposed a novel mathematical technique for deriving a quantity to denote a function's expressional complexity. The algorithm performs normalization to map the data over the interval [-1, 1]. The algorithm then walks down the individual, and uses a set of inductively-defined rules to recursively determine the non-linearity based upon the operator on the node currently being evaluated, and the non-linearity of its right and left sub trees. Non-linear operations, such as exponentiation, trigonometric functions, and logarithms, are then penalized based upon the degree of the Chebyshev polynomial required to approximate the function over the normalized range to a desired precision. Adjusting the precision effectively controls the amount of regularization the model performs (Smits & Vladislavleva, 2009). The Chebyshev polynomials are orthogonal, and possess the property that they are infinite, as defined by a simple recurrence relation. These polynomials pos-

sess the important property that given enough terms, they can eventually approximate any function over the interval [-1, 1] with arbitrary accuracy. The terms begin simple, and grow increasingly complex over time. Therefore, operations that are more difficult to model over their normalized range are likely to be more non-linear. The algorithm then penalizes these sub trees appropriately (Smits & Vladislavleva, 2009).

The final component of the work we would like to extend is the user study. While our work indicates that future work exploring the pedagogical value of Symbolic Regression may be fruitful, we did not possess the time to perform a thoroughly-exhaustive study. Our initial findings suggest that Symbolic Regression's pedagogical value may increase with the user's knowledge of statistics. We would like to enroll a greater number of students for a similar user study. The students will be grouped based upon the extent of their education in mathematics. Each group will then be further divided into an experimental and control group. We would like to perform a second iteration of the study with larger samples, where each group can provide statistically-significant results. We would also like to see if the performance disparity between the experimental and control groups grows in groups with greater mathematical knowledge, as we have previously hypothesized.

6 References

Cut Splice Crossover [photograph]. Retrieved May 8, 2015, from: http://en.wikipedia.org/wiki/Crossover_%28genetic_algorithm%29#/media/File:CutSpliceCrossover.png

Dolan, K. (2009). *Genetic programming source*. Retrieved from http://geneticprogramming.us/Selection.html

Flowchart for Genetic Programming [photograph]. Retrieved April 8, 2015, from: http://www.geneticprogramming.com/Tutorial/flowchart.gif

Genetic Programming [photograph]. Individual representation. Retrieved April 9, 2015, from: http://www.genetic-programming.com/BBB3663gen0.gif

Grefenstette, J. (2013). *Genetic algorithms and their applications*. (2 ed., pp. 12-21). Psychology Press. Retrieved from https://books.google.com/books?hl=en&lr=&id=MYJ_AAAAQBAJ&oi=fnd&pg=PA14&dq=stochasticuniversalsampling&ots=XvpFtq5xCz&sig=XN0Kf9KKe09X_rjwObmydVJ53to

Keijzer, M. (2004). Scaled symbolic regression. *Genetic Programming and Evolvable Machines*, 5(3), 259-269. doi:10.1023/B:GENP.0000030195.77571.f9

Kennedy, J., & Eberhart, R. (1995). *Particle swarm optimization*. Manuscript submitted for publication, School of Engineering and Technology, Purdue University, Indianapolis, IN, . Retrieved from http://www.cs.tufts.edu/comp/150GA/homeworks/hw3/_reading6%1995%particleswarming.pdf

Kommenda, M., Kronberger, G., Winkler, S., Affenzeller, M., & Wagner, S. (2013). *Effects of constant optimization by nonlinear least squares minimization in symbolic regression*. Informally published manuscript, Applied Sciences, University of Upper Austria, Hagenburg, Austria. Retrieved from http://research.fh-ooe.at/files/publications/3431_GECCO_2013_Kommenda.pdf

Koza, J. (2003, August 27). *Example of a run of genetic programming (symbolic regression of a quadratic polynomial)*. Retrieved from http://www.genetic-programming.com/

gpquadraticexample.html

Koza, J. R. (1992). Introduction to genetic programming. In *Genetic programming: On the programming of computers by means of natural selection* (6th ed., pp. 17-61). Retrieved from http://www.ru.lv/peter/zinatne/ebooks/MIT%20-%20Genetic%20Programming.pdf.

Kwiesielewicz, M. Selsus [photograph]. Retrieved May 9, 2015, from: http://www.pg.gda.pl/~mkwies/dyd/geadocu/selsus.gif

Man, K., Tang, K. S., & Kwong, S. (2012). Genetic algorithms: Concepts and designs. (pp. 22-45). Medford, MA: Springer Science & Business Media. Retrieved from https://books.google.com/books?id=RYPuBwAAQBAJ&pg=PA27&lpg=PA27&dq=whatisminimumspreadgene source=bl&ots=84RV2VL9oN&sig=pzPeDvVImZUjAduC9y6NFv0PwjQ&hl=en&sa=X&ei=Ql9RVZ6LNsnlsAWik4CYDQ&ved=0CCgQ6AEwAg

Miller, B. L., & Goldberg, D. E. (1995). *Genetic algorithms, tournament selection, and the effects of noise*. Manuscript submitted for publication, Department of Computer Science, University of Illinois at Urbana-Champaign, Illinois, IL, . Retrieved from http://www.complex-systems.com/pdf/09-3-2.pdf

One Point Crossover [photogrpah]. Retrieved May 8, 2015, from: http://en.wikipedia.org/wiki/Crossover_%28genetic_algorithm%29#/media/File:OnePointCrossover.svg

Particle Swarm Optimization [photograph]. Retrieved April 9, 2015, from: http://mnemstudio.org/ai/pso/images/pso1.gif

Shi, Y., Eberhart, R. C. (1999). Empirical study of particle swarm optimization. CEC 99. *Proceedings of the 1999 Congress on Evolutionary Computation*, 3(1), 1945-1950. doi: 10.1109/CEC.1999.785511

Smits, G., & Kotanchek, M. (n.d.). Pareto-front exploitation in symbolic regression. *Genetic Programming Theory and Practice, 2*, 283-299. Retrieved January 30, 2015, from http://www.evolved-analytics.com/sites/EA_Documents/Legacy/GPTP04/GPTP04_

ParetoGP_Preprint.pdf

Symbolic Regression [photograph]. Symbolic Regression Explanation. Retrieved April 10, 2015, from: http://cdn2.hubspot.net/hub/288190/file-229965993-png/images/symbolic_regression.png?t=1427728670065

Two Point Crossover [photograph]. Retrieved May 8, 2015, from: http://upload.wikimedia.org/wikipedia/commons/c/cd/TwoPointCrossover.svg

Vladislavleva, E. J., Smits, G. F. (2009). Order of non-linearity as a complexity measure for models generated by symbolic regression via pareto genetic programming. *IEEE Transactions on Evolutionary Computation*, 13(2), 333-349. doi:10.1109/TEVC.2009.2014211

Zalzala, A. M. S., & Green, D. (1999). *Mtgp: A multithreaded java tool for genetic programming applications*. Manuscript submitted for publication, Department of Computing & Electrical Engineering, Heriot-Watt University, Edinburgh, United Kingdom. Retrieved from http://www.cs.bham.ac.uk/~wbl/biblio/cache/http_deron.csie.ncue.edu.tw_AI_paper_MTGP_20a_20multithreaded_20Java_20tool_20for_20genetic_20programming_20applications.pdf

Zhang, C. (2014, March 4). Electrical engineering and computer science, University of Tennessee, Knoxville. Retrieved from: http://web.eecs.utk.edu/~czhang24/projects/cs528Project2Zhang.pdf

7 Appendix

7.1 Selected Source Code

```
1  ###
2    * Copyright (c) 2011, iSENSE Project. All rights reserved.
3    *
4    * Redistribution and use in source and binary forms, with or
       without
5    * modification, are permitted provided that the following
       conditions are met:
6    *
7    * Redistributions of source code must retain the above copyright
        notice, this
8    * list of conditions and the following disclaimer.
       Redistributions in binary
9    * form must reproduce the above copyright notice, this list of
       conditions and
10   * the following disclaimer in the documentation and/or other
       materials
11   * provided with the distribution. Neither the name of the
       University of
12   * Massachusetts Lowell nor the names of its contributors may be
       used to
13   * endorse or promote products derived from this software without
        specific
14   * prior written permission.
15   *
16   * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
       CONTRIBUTORS "AS IS"
17   * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
       LIMITED TO, THE
18   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
       PARTICULAR PURPOSE
19   * ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS
       BE LIABLE FOR
20   * ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
       CONSEQUENTIAL
21   * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
       SUBSTITUTE GOODS OR
22   * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
       INTERRUPTION) HOWEVER
23   * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
       STRICT
```

```
24    * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
          IN ANY WAY
25    * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
          POSSIBILITY OF SUCH
26    * DAMAGE.
27    *
28  ###
29
30  ##############################################################################
31  # Programmer:    Jacob Kinsman                                               #
32  #                                                                            #
33  # Assignment:    Honors Project                                              #
34  #                                                                            #
35  # File:          Binary Tree                                                 #
36  #                                                                            #
37  # Description:   This file defines a modified binary tree that               #
38  #                will be used to represent prefix expressions in             #
39  #                my symbolic regression implementation.                      #
40  ##############################################################################
41
42
43  $ ->
44
45    if namespace.controller is "visualizations" and
46    namespace.action in ["displayVis", "embedVis", "show"]
47
48      window.symregr ?= {}
49      symregr.add = (a, b) -> a + b
50      symregr.subtract = (a, b) -> a - b
51      symregr.multiply = (a, b) -> a * b
52      symregr.safeDiv = (a, b) -> if b is 0 then 1 else a / b
53      symregr.pow = (a, b) -> Math.pow(a, b)
54      symregr.exp = (a) -> Math.exp(a)
55      symregr.cos = (a) -> Math.cos(a)
56      symregr.sin = (a) -> Math.sin(a)
57      symregr.safeLog = (a) -> Math.log(Math.abs(a))
58      symregr.safeSqrt = (a) -> Math.sqrt(Math.abs(a))
59      class window.BinaryTree extends Object
60
61        # Initial ephemeral constant value on the interval [-1, 1)
62        @ephemeralConstant:  Math.random() * 2 - 1
63
64        @terminals = [
65          'x', 'ec'
66        ]
```

```
67
68        @operators = [
69          symregr.add, symregr.subtract, symregr.multiply, symregr.
              safeDiv, symregr.pow,
70          symregr.exp, symregr.cos, symregr.sin, symregr.safeLog,
              symregr.safeSqrt
71        ]
72
73        constructor: (parent = null) ->
74          @data = null
75          @right = null
76          @left = null
77          @parent = parent
78
79        # Returns deep copy of binary tree object
80        @clone: (tree, parent = null) ->
81          return tree if tree is null or typeof tree isnt 'object'
82          temp = new BinaryTree(parent)
83          for key of tree when (typeof(tree[key]) isnt 'function'
              and key isnt 'parent')
84            temp[key] = @clone(tree[key], temp)
85          temp['data'] = tree.data
86          temp
87
88        # Returns true if a and b are equivalent objects
89        # (not necessarily references to the same object in memory).
90        @isEqual: (a, b) ->
91          [leftEq, rightEq] = [true, true]
92          if a is null and b isnt null or a isnt null and b is null
93            return false
94          if (a.isTerminal() and b.isTerminal()) is true and a.data
              is b.data
95            return true
96          if '' + a.data isnt '' + b.data or a.treeSize() isnt b.
              treeSize() or a.maxDepth() isnt b.maxDepth()
97            return false
98          if a.left isnt null and b.left isnt null
99            leftEq = BinaryTree.isEqual(a.left, b.left)
100         if a.right isnt null and b.right isnt null
101           rightEq = BinaryTree.isEqual(a.right, b.right)
102         leftEq and rightEq
103
104       # Checks if the tree is terminal (i.e., no children)
105       isTerminal: ->
106         @left is null and @right is null
```

```
107
108         # Returns number of nodes in the tree
109         treeSize: ->
110           @__query((a, b) -> a + b)
111
112         # Returns the maximum depth of the tree
113         maxDepth: ->
114           @__query(Math.max)
115
116         # Internal method used to abstract the treeSize() and
              maxDepth()
117         # member functions
118         ###
119         # WARNING:  INTERNAL METHOD. DO NOT CALL.
120         ###
121         __query: (combiner) ->
122           rest = 0
123           if @left? and @left isnt null
124             rest = combiner(rest, @left.__query(combiner))
125           if @right? and @right isnt null
126             rest = combiner(rest, @right.__query(combiner))
127           return 1 + rest
128
129         # Inserts a single datum in the tree at @data,
130         # or the @data member of the tree located at
131         # pos = 'left' or pos = 'right'
132         ###
133         # WARNING:  MUTATES THE BINARY TREE
134         ###
135         insertData: (data, pos = null) ->
136           if pos is 'left'
137             if @data is null
138               console.log "Error inserting #{data} into left child
                    of tree."
139               return
140             if @left is null
141               @left = new BinaryTree(@)
142             @left.data = data
143           else if pos is 'right'
144             if @data is null
145               console.log "Error inserting #{data} into right child
                    of tree."
146               return
147             if @right is null
148               @right = new BinaryTree(@)
```

```
149           @right.data = data
150         else
151           @data = data
152
153       # Delete a single datum in the tree at @data,
154       # or the @data member of the tree located at
155       # pos = 'left' or pos = 'right'
156       ###
157       # WARNING:  MUTATES THE BINARY TREE
158       ###
159       deleteData: (pos = null) ->
160         if pos is 'right'
161           if @right.isTerminal()
162             @right = null
163             return
164           console.log "Error deleting #{@right.data}, results in
                invalid binary tree."
165           null
166         else if pos is 'left'
167           if @left.isTerminal()
168             @left = null
169             return
170           console.log "Error deleting #{@left.data}, results in
                invalid binary tree."
171           null
172         else
173           if @isTerminal()
174             @data = null
175             return
176           console.log "Error deleting #{@data}, results in invalid
                binary tree."
177           null
178
179       # Allows the user to index into the tree, following Preorder
            traversal:
180       # ROOT, left, right
181       index: (index) ->
182         @__access(index)
183
184       # Determine how far away the node at position 'index' is
            from the root
185       # of the binary tree. This is used to determine how long the
            mutation
186       # tree at a given point can be to maintain the maximum depth
            of the
```

```
187       # tree during the point mutation genetic operation.
188       depthAtPoint: (index, curDepth = 1) ->
189         @__access(index, false, curDepth)
190
191       # Internal method used to abstract the index and depth at
              point member
192       # functions.
193       ###
194       # WARNING:  INTERNAL METHOD. DO NOT CALL
195       ###
196       __access: (index, value = true, curDepth = 1) ->
197         if index is 0
198           return if value is true then @ else curDepth
199         leftSize = if @left is null then 0 else @left.treeSize()
200         rightSize = if @right is null then 0 else @right.treeSize
              ()
201         if index > leftSize + rightSize
202           -1
203         else if index > leftSize
204           if @right isnt null then @right.__access(index -
                leftSize - 1, value, curDepth + 1) else -1
205         else
206           if @left isnt null then @left.__access(index - 1, value,
                curDepth + 1) else -1
207
208       # Given a tree, replace it with a randomly-generated tree
              whose maximum
209       # depth is given by maxDepth.
210       ###
211       # WARNING:  MUTATES THE BINARY TREE
212       ###
213       generate: (maxDepth = 10, curDepth = 1) ->
214         if curDepth is maxDepth
215           @insertData(BinaryTree.terminals[Math.floor(Math.random
                () * BinaryTree.terminals.length)])
216         else
217           randomGene = Math.floor(Math.random() * (BinaryTree.
                terminals.length + BinaryTree.operators.length))
218           if randomGene < BinaryTree.terminals.length
219             @insertData(BinaryTree.terminals[randomGene])
220           else
221             gene = BinaryTree.operators[randomGene - BinaryTree.
                  terminals.length]
222             @insertData(gene)
223             @left = new BinaryTree(@)
```

```
224             @left.generate(maxDepth, curDepth + 1)
225             if gene.length isnt 1
226               @right = new BinaryTree(@)
227               @right.generate(maxDepth, curDepth + 1)
228
229         # Evaluate the Binary tree numerically for a given input
               value
230         evaluate: (x, val = null) ->
231           if @data is 'x'
232             if val isnt null then val else x
233           else if @data is 'ec'
234             BinaryTree.ephemeralConstant
235           else if typeof(@data) is 'number'
236             @data
237           else
238             if @data.length is 1
239               @data(@left.evaluate(x))
240             else
241               @data(@left.evaluate(x), @right.evaluate(x))
242
243         # Insert the BinaryTree object 'tree' at the location of the
               BinaryTree
244         # specified by index
245         ###
246         # WARNING:  MUTATES THE BINARY TREE 'THIS', DOES NOT MUTATE
               ARGUMENT TREE
247         ###
248         insertTree: (tree, index = 0) ->
249           replacementPoint = @index(index)
250           if index is null or replacementPoint is -1
251             console.log "Error inserting #{tree} at location
                  specified. Index does not exist in the tree."
252             return null
253           start = replacementPoint.parent
254           if start isnt null
255             if start.left isnt null and BinaryTree.isEqual(
                  replacementPoint, start.left)
256               start.left = BinaryTree.clone(tree)
257             else
258               if start.right isnt null and BinaryTree.isEqual(
                     replacementPoint, start.right)
259                 start.right = BinaryTree.clone(tree)
260             start.__updateParents()
261           else
262             @data = tree.data
```

```
263           @right = BinaryTree.clone(tree.right)
264           @left = BinaryTree.clone(tree.left)
265           @parent = null
266           @__updateParents()
267
268         # Updates binary tree element's parents to reflect the
              result of
269         # an insertTree merger to parent.right or parent.left
270         ###
271         # WARNING:  MUTATES THE BINARY TREE
272         ###
273         ###
274         # WARNING:  INTERNAL METHOD. DO NOT CALL.
275         ###
276         __updateParents: ->
277           if @right isnt null
278             @right = BinaryTree.clone(@right, @)
279             @right.__updateParents()
280           if @left isnt null
281             @left = BinaryTree.clone(@left, @)
282             @left.__updateParents()
283
284         # Given two parent trees, create two new child trees by
              crossover.
285         # Both parent trees are given a randomly-selected crossover
              point.
286         # The first child is the part of the first parent before its
               crossover
287         # point, and the section of the second parent after its
              crossover point.
288         # The second child is the part of the first parent after its
               crossover
289         # point, and the part of the second parent before its
              crossover point.
290
291         @crossover: (tree1, tree2) ->
292           [tree1a, tree1b] = [BinaryTree.clone(tree1), BinaryTree.
                clone(tree1)]
293           [tree2a, tree2b] = [BinaryTree.clone(tree2), BinaryTree.
                clone(tree2)]
294
295           [crossoverPointOne, crossoverPointTwo] =
296             [Math.floor(Math.random() * tree1.treeSize()), Math.
                  floor(Math.random() * tree2.treeSize())]
297           [childOne, childTwo] = [BinaryTree.clone(tree1a),
```

```
          BinaryTree.clone(tree2a)]
298       childOne.insertTree(childTwo.index(crossoverPointTwo),
            crossoverPointOne)
299       childTwo.insertTree(tree1b.index(crossoverPointOne),
            crossoverPointTwo)
300       [childOne, childTwo]
301
302     # Given a tree, construct a string representation of the
          mathematical
303     # function the tree describes
304     @stringify: (tree) ->
305
306       # Helper method to properly parenthesize nested terms
307       parenthesize = (string) ->
308         if not isNaN(Number(string)) or string is 'x' then
              string else "(#{string})"
309
310       switch tree.data
311         when symregr.add
312           "#{parenthesize(@stringify(tree.left))} + #{
                parenthesize(@stringify(tree.right))}"
313         when symregr.subtract
314           "#{parenthesize(@stringify(tree.left))} - #{
                parenthesize(@stringify(tree.right))}"
315         when symregr.multiply
316           "#{parenthesize(@stringify(tree.left))} * #{
                parenthesize(@stringify(tree.right))}"
317         when symregr.safeDiv
318           "#{parenthesize(@stringify(tree.left))} / #{
                parenthesize(@stringify(tree.right))}"
319         when symregr.pow
320           "#{parenthesize(@stringify(tree.left))} <sup>#{
                parenthesize(@stringify(tree.right))}</sup>"
321         when symregr.exp
322           "e <sup>#{parenthesize(@stringify(tree.left))}</sup>"
323         when symregr.cos
324           "cos(#{parenthesize(@stringify(tree.left))})"
325         when symregr.sin
326           "sin(#{parenthesize(@stringify(tree.left))})"
327         when symregr.safeLog
328           "log(|#{parenthesize(@stringify(tree.left))}|)"
329         when symregr.safeSqrt
330           "sqrt(|#{parenthesize(@stringify(tree.left))}|)"
331         when 'x'
332           'x'
```

```
333             when 'ec'
334               "#{roundToFourSigFigs(BinaryTree.ephemeralConstant)}"
335             else
336               "#{roundToFourSigFigs(tree.data)}"
337
338         # Given a tree, construct a string of valid coffeescript
                code that can be 'evaled' to
339         # mimic the symbolic regression.
340         @codify: (tree) ->
341           # Helper method to properly parenthesize nested terms
342           parenthesize = (string) ->
343             if not isNaN(Number(string)) or string is 'x' then
                  string else "(#{string})"
344
345           getFunc = (tree) ->
346             switch tree.data
347               when symregr.add
348                 "symregr.add(#{parenthesize(getFunc(tree.left))}, #{
                      parenthesize(getFunc(tree.right))})"
349               when symregr.subtract
350                 "symregr.subtract(#{parenthesize(getFunc(tree.left))
                      }, #{parenthesize(getFunc(tree.right))})"
351               when symregr.multiply
352                 "symregr.multiply(#{parenthesize(getFunc(tree.left))
                      }, #{parenthesize(getFunc(tree.right))})"
353               when symregr.safeDiv
354                 "symregr.safeDiv(#{parenthesize(getFunc(tree.left))
                      }, #{parenthesize(getFunc(tree.right))})"
355               when symregr.pow
356                 "symregr.pow(#{parenthesize(getFunc(tree.left))}, #{
                      parenthesize(getFunc(tree.right))})"
357               when symregr.exp
358                 "symregr.exp(#{parenthesize(getFunc(tree.left))})"
359               when symregr.cos
360                 "symregr.cos(#{parenthesize(getFunc(tree.left))})"
361               when symregr.sin
362                 "symregr.sin(#{parenthesize(getFunc(tree.left))})"
363               when symregr.safeLog
364                 "symregr.safeLog(#{parenthesize(getFunc(tree.left))
                      })"
365               when symregr.safeSqrt
366                 "symregr.safeSqrt(#{parenthesize(getFunc(tree.left))
                      })"
367               when 'x'
368                 'x'
```

```
369             when 'ec'
370               "#{BinaryTree.ephemeralConstant}"
371             else
372               "#{tree.data}"
373
374         'return ' + getFunc(tree)
```

```
1  ###
2    * Copyright (c) 2011, iSENSE Project. All rights reserved.
3    *
4    * Redistribution and use in source and binary forms, with or
       without
5    * modification, are permitted provided that the following
       conditions are met:
6    *
7    * Redistributions of source code must retain the above copyright
       notice, this
8    * list of conditions and the following disclaimer.
       Redistributions in binary
9    * form must reproduce the above copyright notice, this list of
       conditions and
10   * the following disclaimer in the documentation and/or other
       materials
11   * provided with the distribution. Neither the name of the
       University of
12   * Massachusetts Lowell nor the names of its contributors may be
       used to
13   * endorse or promote products derived from this software without
       specific
14   * prior written permission.
15   *
16   * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
       CONTRIBUTORS "AS IS"
17   * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
       LIMITED TO, THE
18   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
       PARTICULAR PURPOSE
19   * ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS
       BE LIABLE FOR
20   * ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
       CONSEQUENTIAL
21   * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
       SUBSTITUTE GOODS OR
22   * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
       INTERRUPTION) HOWEVER
23   * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
       STRICT
24   * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
       IN ANY WAY
25   * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
       POSSIBILITY OF SUCH
26   * DAMAGE.
```

```
27    *
28 ###
29
30 ##########################################################################
31 # Programmer:    Jacob Kinsman                                            #
32 #                                                                         #
33 # Assignment:    Honors Project                                           #
34 #                                                                         #
35 # File:          Individual                                               #
36 #                                                                         #
37 # Description:   This file defines an "individual" to be used in          #
38 #                my symbolic regression algorithm. In genetic             #
39 #                algorithms, each individual in a population              #
40 #                describes a candidate solution, to which a               #
41 #                genetic operator may be applied.                         #
42 #                                                                         #
43 ##########################################################################
44
45
46 $ ->
47
48   if namespace.controller is "visualizations" and
49   namespace.action in ["displayVis", "embedVis", "show"]
50
51     class window.Individual extends Object
52
53       # Create a a binary tree to represent a random mathematical
             function
54       constructor: (tree = null, maxDepth = 10) ->
55         if tree is null
56           @tree = new BinaryTree
57           @tree.generate(Math.floor(Math.random() * (maxDepth - 1)
                 ) + 2)
58         else
59           @tree = BinaryTree.clone(tree)
60         @depth = @tree.maxDepth()
61         @maxDepth = maxDepth
62
63       # Numerically evaluate the individual function at the value
             n
64       evaluate: (n) ->
65         @tree.evaluate(n)
66
67       # Point mutation genetic operator
68       @mutate: (individual) ->
```

```
69          length = individual.tree.treeSize()
70          mutationSite = Math.floor(Math.random() * length)
71          mutant = BinaryTree.clone(individual.tree)
72          mutation = new BinaryTree
73          mutation.generate(Math.floor(Math.random() * (individual.
              maxDepth - 1)) + 2)
74          mutant.insertTree(mutation, mutationSite)
75          ret = new Individual(mutant, mutant.maxDepth())
76
77        # Crossover genetic operator (cut and splice approach)
78        @crossover: (individual1, individual2) ->
79          [childOne, childTwo] = BinaryTree.crossover(individual1.
              tree, individual2.tree)
80          [new Individual(childOne, childOne.maxDepth()), new
              Individual(childTwo, childTwo.maxDepth())]
81
82        # Crossover genetic operator (single-point approach)
83        @onePointCrossover: (individual1, individual2) ->
84          [treeOne, treeTwo] = [BinaryTree.clone(individual1.tree),
              BinaryTree.clone(individual2.tree)]
85          mutationSite = Math.floor(Math.random() * Math.min(treeOne
              .treeSize(), treeTwo.treeSize()))
86          [childOne, childTwo] = [BinaryTree.clone(treeOne),
              BinaryTree.clone(treeTwo)]
87          childOne.insertTree(treeTwo.index(mutationSite),
              mutationSite)
88          childTwo.insertTree(treeOne.index(mutationSite),
              mutationSite)
89          [new Individual(childOne, childOne.maxDepth()), new
              Individual(childTwo, childTwo.maxDepth())]
90
91        # Crossover genetic operator (two-point approach)
92        @twoPointCrossover: (individual1, individual2) ->
93          [tree1a, tree1b] = [BinaryTree.clone(individual1.tree),
              BinaryTree.clone(individual1.tree)]
94          [tree2a, tree2b] = [BinaryTree.clone(individual2.tree),
              BinaryTree.clone(individual2.tree)]
95          [tree1c, tree2c] = [BinaryTree.clone(individual1.tree),
              BinaryTree.clone(individual2.tree)]
96          mutationSite1 = Math.floor(Math.random() * Math.min(tree1a
              .treeSize(), tree2a.treeSize()))
97          mutationSite2 = Math.floor(Math.random() *
98            Math.min(tree1a.treeSize() - mutationSite1, tree2a.
                treeSize() - mutationSite1)) + mutationSite1
99          [childOne, childTwo] = [BinaryTree.clone(tree1a),
```

```
          BinaryTree.clone(tree2a)]
100       [childOneTail, childTwoTail] = [BinaryTree.clone(tree1a.
            index(mutationSite2)),
101         BinaryTree.clone(tree2a.index(mutationSite2))]
102       childOne.insertTree(tree2a.index(mutationSite1),
            mutationSite1)
103       childTwo.insertTree(tree1a.index(mutationSite1),
            mutationSite1)
104       childOne.insertTree(childOneTail, Math.min(mutationSite2,
            childOne.treeSize() - 1))
105       childTwo.insertTree(childTwoTail, Math.min(mutationSite2,
            childTwo.treeSize() - 1))
106       [new Individual(childOne, childOne.maxDepth()), new
            Individual(childTwo, childTwo.maxDepth())]
107
108     # Fitness-proportional reproduction genetic operator
109     @fpReproduce: (individuals, points, func, distribution =
          null) ->
110       if distribution is null
111         sumFitnesses = 0
112         individualFitnesses = []
113         for populant in individuals
114           individualFitness = eval "populant.#{func}(points)"
115           sumFitnesses = sumFitnesses + individualFitness
116           individualFitnesses.push individualFitness
117         individualFitnesses = individualFitnesses.map((y) -> if
              isNaN y then 0 else y)
118         distribution = for number, index in individualFitnesses
119           cumulativeFitness = individualFitnesses[0..index].
                reduce (pv, cv, index, array) -> pv + cv
120           cumulativeFitness / sumFitnesses
121       rand = Math.random()
122       for probability, i in distribution
123         if rand < probability
124           indiv = individuals[i]
125           return new Individual(indiv.tree, indiv.maxDepth)
126
127     # Fitness-proportional selection genetic operator
128     @fpSelection: (individuals, points, func, distribution =
          null) ->
129       @fpReproduce(individuals, points, func, distribution)
130
131     # Tournament reproduction genetic operator
132     @tournamentReproduce: (individuals, points, func,
          tournamentSize = 10, probability = 0.8) ->
```

```
133         tournament = for i in [0...tournamentSize]
134           participant = Math.floor(Math.random() * individuals.
                length)
135           [tree, maxDepth] = [participant.tree, participant.
                maxDepth]
136           {individual: new Individual(tree, maxDepth), index: i}
137         for participant in tournament
138           participant.fitness = eval "participant.individual.#{
                func}(points)"
139         tournament.map((participant) -> if isNaN participant.
              fitness then 0 else participant.fitness)
140         tournament.sort (a, b) -> Number(b.fitness) - Number(a.
              fitness)
141         for participant, ind in tournament
142           rand = Math.random()
143           if rand < probability * Math.pow(1 - probability, ind)
144             return new Individual(participant.individual.tree,
                  participant.individual.maxDepth)
145         return new Individual(tournament[0].individual.tree,
              tournament[0].individual.maxDepth)
146
147       # Tournament selection genetic operator
148       @tournamentSelection: (individuals, points, func,
            tournamentSize = 10, probability = 0.8) ->
149         @tournamentReproduce(individuals, points, func,
              tournamentSize, probability)
150
151       # Stochastic Universal Sampling reproduction genetic
            operator
152       ###
153       # WARNING: PERFORMS REPRODUCTION/SELECTION ALL AT ONCE
154       ###
155       @susReproduce: (individuals, points, func, offspring,
            distribution = null) ->
156         fitnesses = []
157         children = []
158         if distribution is null
159           fitnesses = for individual in individuals
160             eval "individual.#{func}(points)"
161           sumFitnesses = fitnesses.map((y) -> if isNaN(y) then 0
                else y).reduce((pv, cv, index, array) -> pv + cv)
162           distribution = for number, index in fitnesses
163             cumulativeFitness = fitnesses[0..index].reduce (pv, cv
                  , index, array) -> pv + cv
164             cumulativeFitness / sumFitnesses
```

```
165         distance = 1 / offspring
166         pointer = Math.random() * distance
167         [lastChild, numChildren] = [0, 0]
168         for i in [0...offspring]
169           for j in [lastChild...distribution.length]
170             if distribution[j] > (pointer * (numChildren + 1))
171               lastChild = j
172               numChildren = numChildren + 1
173               child = new Individual(individuals[j].tree,
                    individuals[j].maxDepth)
174               children.push(child)
175               break
176         children
177
178       # Stochastic Universal Sampling selection genetic operator
179       ###
180       # WARNING:  PERFORMS REPRODUCTION/SELECTION ALL AT ONCE
181       ###
182       @susSelection: (individuals, points, func, offspring,
            distribution = null) ->
183         @susReproduce(individuals, points, func, offspring,
              distribution)
184
185       # Calculates an individual's fitness by its sum of squared-
            error over points
186       sseFitness: (points, raw = false) ->
187         fitnessAtPoints = for point in points
188           Math.pow(point.y - @evaluate(point.x), 2)
189         fitness = fitnessAtPoints.reduce (pv, cv, index, array) ->
              pv + cv
190         if isNaN(fitness)
191           fitness = Infinity
192         return if raw then fitness else 1 / (1 + fitness)
193
194       # Calculates an individual's fitness by its mean squared-
            error over points
195       mseFitness: (points) ->
196         fitness = 1 / (1 + (1 / points.length) * @sseFitness(
              points, true))
197         fitness
198
199       # Calculates an individual's fitness by scaled fitness
200       # (technique employed in the scaled symbolic regression
            paper by Keijzer)
201       scaledFitness: (points, params = false) ->
```

```
202         # define inputs (xs), targets (ts), and outputs (ys)
203         xs = (point.x for point in points)
204         ys = (@evaluate(point.x) for point in points)
205         ts = (point.y for point in points)
206
207         # calculate sum of inputs, targets, and outputs
208         xSum = xs.reduce (pv, cv, index, array) -> pv + cv
209         ySum = ys.reduce (pv, cv, index, array) -> pv + cv
210         tSum = ts.reduce (pv, cv, index, array) -> pv + cv
211
212         # calculate average of inputs, targets, and outputs
213         xAvg = xSum / xs.length
214         yAvg = ySum / ys.length
215         tAvg = tSum / ts.length
216
217         # calculate each input's distance from the mean, xAvg
218         xMeanDiffs = (x - xAvg for x in xs)
219         # calculate each output's distance from the mean, yAvg
220         yMeanDiffs = (y - yAvg for y in ys)
221         # calculate each target's distance from the mean, tAvg
222         tMeanDiffs = (t - tAvg for t in ts)
223
224         # calculate the pair-wise terms of cov(y,t)
225         ytCovTerms = for i in [0...ys.length]
226           (yMeanDiffs[i] * tMeanDiffs[i])
227         # calculate the covariance of t and y
228         ytCov = ytCovTerms.reduce (pv, cv, index, array) -> pv +
              cv
229
230         # calculate the pair-wise terms of var(y)
231         yVarTerms = for ydiff in yMeanDiffs
232           Math.pow(ydiff, 2)
233         # calculate the variance of y
234         yVar = yVarTerms.reduce (pv, cv, index, array) -> pv + cv
235
236         # b = cov(y,t) / var(y)
237         b = if yVar is 0 then 1 else ytCov / yVar
238         a = tAvg - b * yAvg
239
240         # If we need a and b (params = true), return them
241         if params then return [a, b]
242
243         # calculate scaled residuals (target[i] - (a + b * output[
              i])) ^ 2
244         scaledResiduals = for i in [0...points.length]
```

```
245           Math.pow(ts[i] - (a + b * ys[i]), 2)
246         # calculate sum of scaled residuals
247         sumScaledResiduals = scaledResiduals.reduce (pv, cv, index
              , array) -> pv + cv
248
249         # Calculate scaled fitness
250         fitness = (1 / points.length) * sumScaledResiduals
251         return if isNaN(fitness) or fitness is Infinity then 0
              else 1 / (1 + fitness)
252
253       # Calculate an individual's fitness by its normalized mean-
            squared
254       # error over points
255       nmseFitness: (points, raw = false) ->
256         values = (@evaluate(point.x) for point in points)
257         averageValue = values.reduce((pv, cv, index, array) -> pv
              + cv) / values.length
258         targets = (point.y for point in points)
259         averageTarget = targets.reduce((pv, cv, index, array) ->
              pv + cv) / targets.length
260         indFitness = for i in [0...values.length]
261           Math.pow(targets[i] - values[i], 2) / (averageTarget *
                averageValue)
262         fitness = indFitness.reduce((pv, cv, index, array) -> pv +
              cv) / points.length
263         if isNaN(fitness)
264           fitness = 0
265         if raw then fitness else 1 / (1 + fitness)
266
267       # Calculate's an individual's fitness via pareto genetic
268       # programming. Nonlinearity is concurrently minimized
            alongside
269       # the maximization of the fitness function specified.
270       #
271       # Goodness of fit is (by default) calculated via Normalized
            Mean-Squared Error,
272       # and non-linearity is calculated through a visitation-
            length heuristic
273       # as specified in the M. Keijzer and J. Foster paper.
274       paretoFitness: (points, func = 'scaledFitness') ->
275         nonLinearity = (tree) ->
276           return 0 if tree is null
277           switch tree.data
278             when symregr.add, symregr.subtract, symregr.multiply,
                  symregr.safeDiv
```

```
279             tree.treeSize() * (nonLinearity(tree.left) +
                    nonLinearity(tree.right))
280           when symregr.sin, symregr.cos
281             3 * tree.treeSize() * (nonLinearity(tree.left) +
                    nonLinearity(tree.right))
282           when symregr.safeSqrt, symregr.pow, symregr.exp,
                  symregr.safeLog
283             2 * tree.treeSize() * (nonLinearity(tree.left) +
                    nonLinearity(tree.right))
284           else 1
285       fitness = Math.pow(eval("this.#{func}(points)"), -1) - 1
286       nonlinearity = nonLinearity(@tree)
287       return if isNaN(fitness + nonlinearity) or (fitness +
                nonlinearity) is Infinity
288         0
289       else
290         (1 / fitness + nonlinearity)
291
292     ###
293     # Particle Swarm Optimization is a nonlinear optimization
            strategy used to enhance
294     # the performance of symbolic regression. The terminal set
            consists of the
295     # dependent variable, x, and a single constant referred to
            in the literature as the
296     # ephemeral constant. PSO finds a near-optimal value of
            these ephemeral constants
297     # to maximize the individual's fitness, separating the task
            of identifying the correct
298     # function, and the constant values it contains.
299     ###
300     ###
301     # WARNING:  MUTATES THE INDIVIDUAL
302     ###
303     @particleSwarmOptimization: (populant, points) ->
304       # Initialize algorithm parameters
305       fitness = 'mseFitness'
306       maxFitness = 1
307       numParticles = 50
308       maxPosition = 1000
309       minPosition = -1000
310       maxVelocity = 50
311       minVelocity = -50
312       maxIterations = 200
313       numNeighborhoods = 5
```

```
314         c1 = 2
315         c2 = 2
316         tree = populant.tree
317         treeValues = for i in [0...tree.treeSize()]
318           value = tree.index(i)
319           if value.data is 'ec' or typeof(value.data) is 'number'
                then value.data else null
320         constants = []
321         for value, index in treeValues
322           if value isnt null then constants.push {value: value,
                index, index}
323         return populant if constants.length is 0
324         particles = []
325         for i in [0...numParticles]
326           dimensions = for j in [0...constants.length]
327             index = constants[j].index
328             velocity = Math.random() * (maxVelocity - minVelocity)
                    + minVelocity
329             position = Math.random() * (maxPosition - minPosition)
                    + minPosition
330             {velocity: velocity, position: position, personalBest:
                    position, index: index}
331           particles.push {neighborhood: i % numNeighborhoods,
                dimensions: dimensions}
332         neighborhoodBests = (-Infinity for i in [0...
              numNeighborhoods])
333         dimensionBests = (-Infinity for i in [0...numNeighborhoods
              ])
334         for particle in particles
335           evaluationPoint = new Individual(tree)
336           for dimension in particle.dimensions
337             evaluationPoint.tree.index(dimension.index).insertData
                  (dimension.position)
338           currentFitness = eval "evaluationPoint.#{fitness}(points
                )"
339           particle.bestFitness = currentFitness
340           if currentFitness >= maxFitness
341             return evaluationPoint
342           if currentFitness >= neighborhoodBests[particle.
                neighborhood]
343             neighborhoodBests[particle.neighborhood] =
                  currentFitness
344             dimensionBests[particle.neighborhood] = (dim.position
                  for dim in particle.dimensions)
345         for i in [0...maxIterations]
```

```
346           for particle in particles
347             evaluationPoint = new Individual(tree, populant.
                  maxDepth)
348             for dimension, index in particle.dimensions
349               dimension.velocity = Math.min(Math.max(dimension.
                    velocity + (Math.random() * c1 *
350                   (dimension.personalBest - dimension.position)) + (
                      Math.random() * c2 *
351                     (dimensionBests[particle.neighborhood][index] -
                        dimension.position)), minVelocity),
                        maxVelocity)
352               dimension.position = Math.max(Math.min(dimension.
                    position + dimension.velocity, maxVelocity),
353                   minVelocity)
354               evaluationPoint.tree.index(dimension.index).
                    insertData(dimension.position)
355             currentFitness = eval "evaluationPoint.#{fitness}(
                  points)"
356             if currentFitness > particle.bestFitness
357               particle.bestFitness = currentFitness
358               for dimension, index in particle.dimensions
359                 dimension.personalBest = dimension.position
360             if currentFitness >= maxFitness
361               return new Individual(evaluationPoint.tree, populant
                    .maxDepth)
362             if currentFitness > neighborhoodBests[particle.
                  neighborhood]
363               neighborhoodBests[particle.neighborhood] =
                    currentFitness
364               dimensionBests[particle.neighborhood] = (dim.
                    position for dim in particle.dimensions)
365         [bestIndex, bestFitness] = [0, 0]
366         for fitness, index in neighborhoodBests
367           if neighborhoodBests[i] >= bestFitness
368             bestIndex = index
369             bestFitness = fitness
370         result = new Individual(tree, populant.maxDepth)
371         for constant, index in constants
372           result.tree.index(constant.index).insertData(
                dimensionBests[bestIndex][index])
373         result
```

```
1 ###
2    * Copyright (c) 2011, iSENSE Project. All rights reserved.
3    *
4    * Redistribution and use in source and binary forms, with or
       without
5    * modification, are permitted provided that the following
       conditions are met:
6    *
7    * Redistributions of source code must retain the above copyright
       notice, this
8    * list of conditions and the following disclaimer.
       Redistributions in binary
9    * form must reproduce the above copyright notice, this list of
       conditions and
10   * the following disclaimer in the documentation and/or other
       materials
11   * provided with the distribution. Neither the name of the
       University of
12   * Massachusetts Lowell nor the names of its contributors may be
       used to
13   * endorse or promote products derived from this software without
       specific
14   * prior written permission.
15   *
16   * THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND
       CONTRIBUTORS "AS IS"
17   * AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT
       LIMITED TO, THE
18   * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A
       PARTICULAR PURPOSE
19   * ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS
       BE LIABLE FOR
20   * ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
       CONSEQUENTIAL
21   * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
       SUBSTITUTE GOODS OR
22   * SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
       INTERRUPTION) HOWEVER
23   * CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT,
       STRICT
24   * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING
        IN ANY WAY
25   * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
       POSSIBILITY OF SUCH
26   * DAMAGE.
```

```
27    *
28 ###
29
30 ##############################################################################
31 # Programmer:    Jacob Kinsman                                                #
32 #                                                                             #
33 # Assignment:    Honors Project                                               #
34 #                                                                             #
35 # File:          Symbolic Regression                                          #
36 #                                                                             #
37 # Description:   This file contains the symbolic regression                  #
38 #                algorithm used to calculate regression                       #
39 #                functions.                                                   #
40 ##############################################################################
41
42 $ ->
43
44   if namespace.controller is "visualizations" and
45   namespace.action in ["displayVis", "embedVis", "show"]
46
47     ###
48     # Begin constant declarations. The following constants are
          used
49     # when the user does not specify the algorithm parameters
          desired.
50     ###
51
52     window.symregr ?= {}
53
54     # Fitness function to be used (scaled mean-squared error)
55     symregr.FITNESS = 'scaledFitness'
56     # Selection operator to be used (fitness-proportional)
57     symregr.SELECTION = 'fpSelection'
58     # Reproduction operator to be used (fitness-proportional)
59     symregr.REPRODUCTION = 'fpReproduce'
60     # Crossover operator to be used (cut-and-splice method)
61     symregr.CROSSOVER = 'crossover'
62     # Mutation operator to be used (point mutation)
63     symregr.MUTATION = 'mutate'
64     # Maximum depth of an individual's expression tree
65     symregr.IDEPTH = 4
66     # Maximum length of an expression tree
67     symregr.MDEPTH = 6
68     # Number of individuals in a population at any given time
69     # NOTE:  MUST BE A MULTIPLE OF FOUR TO USE THE
```

```
70      #         OPTIMIZED VERSION OF THE ALGORITHM CORRECTLY.
71      symregr.POPSIZE = 100
72      # Probability that a child individual is produced through
73      # reproduction, and inserted (unmodified) into the next
74      # generation of the population
75      symregr.REPRODUCTIONPR = .3
76      # Probability that two offspring are produced through a
           genetic
77      # crossover operation
78      symregr.CROSSOVERPR = .6
79      # Probability a mutation occurs, and the mutant is inserted
           into
80      # the next generation of the population
81      symregr.MUTATIONPR = .1
82      # Maximum number of iterations (simulated generations) to
83      # be performed while searching for an adequately-fit
84      # candidate solution (termination criteria)
85      symregr.MAXITERS = 100
86      # Any individuals with fitness greater than MAXFITNESS
87      # will be immediately returned as a candidate solution
88      # (termination criteria)
89      symregr.MAXFITNESS = 1
90      # tournament size parameter for tournament selection and
           reproduction
91      symregr.TSIZE = 10
92      # Probability that the most fit individual of the tournament
           is
93      # selected during tournament selection and reproduction
94      symregr.TPR = 0.8
95      ###
96      # Batch sizes for optimized symbolic regression implementation
           ;
97      # Each new generation contains MBS individuals created
98      # by genetic mutation, RBS individuals created by
99      # genetic reproduction, etc.
100     ###
101     symregr.MBS = symregr.MUTATIONPR * symregr.POPSIZE
102     symregr.RBS = symregr.REPRODUCTIONPR * symregr.POPSIZE
103     symregr.CBS = symregr.CROSSOVERPR * symregr.POPSIZE
104     ###
105     # End batch size declarations for optimized symbolic
           regression implementation
106     ###
107     ###
108     # End constant declarations.
```

```
109     ###
110
111     ###
112     # Return the scaled expression tree calculated by scaled
            fitness
113     ###
114     scaledIndividual = (populant, points, individualDepth) ->
115       [a, b] = populant.scaledFitness(points, true)
116       scaledTree = new BinaryTree
117       scaledTree.insertData(symregr.add)
118       scaledTree.insertData(symregr.multiply, 'right')
119       scaledTree.insertData(a, 'left')
120       scaledTree.right.insertData(b, 'left')
121       scaledTree.right.insertData(1, 'right')
122       scaledTree.insertTree(populant.tree, 4)
123       return new Individual(scaledTree, Math.max(individualDepth,
              scaledTree.maxDepth()))
124
125     window.symbolicRegression = (points) ->
126
127       # Stochastic Universal Sampling size is non-configurable for
              traditional symbolic regression
128       susSS = 2
129       susRS = 1
130
131       # Shorter eval statements
132       I = "Individual"
133
134       # Keep track of location of individuals with max/min fitness
              within population
135       maxIndex = 0
136
137       max = (pv, cv, index, array) ->
138         if cv is Math.max(pv, cv) then maxIndex = index
139         Math.max(pv, cv)
140
141       # Create initial population
142       population = (new Individual(null, symregr.IDEPTH) for i in
              [0...symregr.POPSIZE])
143
144       # Calculate fitness of initial population
145       fitnesses = for populant in population
146         eval "populant.#{FITNESS}(points)"
147       # Prevent the selection and reproduction of any trees that
              are too complex
```

```
148         for pop, i in population
149           if pop.tree.maxDepth() >= symregr.MDEPTH
150             fitnesses[i] = 0
151         # Determine most fit individual from initial population
152         mostFit = fitnesses.reduce max, 0
153         bestIndividual = population[maxIndex]
154
155         # Check if the most fit individual from the
156         # initial population is sufficiently fit
157         if mostFit >= symregr.MAXFITNESS
158           return unless symregr.FITNESS in ['scaledFitness', '
                paretoFitness']
159           Individual.particleSwarmOptimization(bestIndividual,
                points)
160           else
161           scaledIndividual(Individual.particleSwarmOptimization(
                population[maxIndex], points), points,
                individualDepth)
162
163         newPopulation = []
164
165         # Else, begin symbolic regression algorithm
166         for i in [0...symregr.MAXITERS]
167           # Initialize empty new population
168           newPopulation = []
169
170           ###
171           # Begin creation of a new generation of individuals
172           ###
173           while newPopulation.length isnt symregr.POPSIZE
174
175             # Step 1: Select one individual  for reproduction with
                  probability reproductionProbability
176             if Math.random() <= symregr.REPRODUCTIONPR
177               # Perform user-specified reproduction with appropriate
                    arguments
178               switch symregr.REPRODUCTION
179                 when 'fpReproduce'
180                   newPopulation.push(eval("#{I}.#{symregr.
                        REPRODUCTION}(population, points, symregr.
                        FITNESS)"))
181                 when 'susReproduce'
182                   newPopulation.push(
183                     eval("#{I}.#{symregr.REPRODUCTION}(population,
                          points, symregr.FITNESS, susRS)")[0])
```

```
184             when 'tournamentReproduce'
185               newPopulation.push(
186                 eval("#{I}.#{symregr.REPRODUCTION}(population,
                        points, symregr.FITNESS, symregr.TSIZE,
                        symregr.TPR)")
187               )
188
189           # Step 2: First, select two individuals for crossover
                with probability
190           #         crossoverProbability. If two individuals are
                selected, use the
191           #         crossover methodology specified to produce two
                 new individuals
192           if Math.random() <= symregr.CROSSOVERPR and symregr.
                POPSIZE - newPopulation.length >= 2
193             parents = switch symregr.SELECTION
194               when 'fpSelection'
195                 parentOne = eval "#{I}.#{symregr.SELECTION}(
                      population, points, symregr.FITNESS)"
196                 parentTwo = eval "#{I}.#{symregr.SELECTION}(
                      population, points, symregr.FITNESS)"
197                 [parentOne, parentTwo]
198               when 'susSelection'
199                 eval "#{I}.#{symregr.SELECTION}(population, points
                      , symregr.FITNESS, susSS)"
200               when 'tournamentSelection'
201                 parentOne =
202                   eval "#{I}.#{symregr.SELECTION}(population,
                        points, symregr.FITNESS, symregr.TSIZE,
                        symregr.TPR)"
203                 parentTwo =
204                   eval "#{I}.#{symregr.SELECTION}(population,
                        points, symregr.FITNESS, symregr.TSIZE,
                        symregr.TPR)"
205                 [parentOne, parentTwo]
206             children = eval("#{I}.#{symregr.CROSSOVER}(parents[0],
                      parents[1])")
207             newPopulation.push(children[0])
208             newPopulation.push(children[1])
209
210           # Step 3:  Select one individual for mutation with
                probability mutationProbability
211           if Math.random() <= symregr.MUTATIONPR and
212             symregr.POPSIZE - newPopulation.length >= 1 and
213               newPopulation.length isnt 0
```

```
214             switch SELECTION
215               when 'fpSelection'
216                 for _ in [0...symregr.MBS]
217                   mutant = eval "#{I}.#{symregr.SELECTION}(
                        population, points, symregr.FITNESS,
                        distribution)"
218                   newPopulation.push(eval("#{I}.#{symregr.MUTATION
                        }(mutant)"))
219               when 'susSelection'
220                 mutants = eval "#{I}.#{symregr.SELECTION}(
                       population, points, symregr.FITNESS, susSS,
                       distribution)"
221                 for mutant in mutants
222                   newPopulation.push(eval("#{I}.#{symregr.MUTATION
                        }(mutant)"))
223               when 'tournamentSelection'
224                 for _ in [0...symregr.MBS]
225                   mutant =
226                     eval "#{I}.#{symregr.SELECTION}(population,
                          points, symregr.FITNESS, symregr.TSIZE,
                          symregr.TPR)"
227                   newPopulation.push(
228                     eval("#{I}.#{symregr.MUTATION}(mutant)"))
229         ###
230         # End creation of a new generation of individuals
231         ###
232
233         # Set current population to the next generation
234         population = newPopulation
235         # Calculate fitness of the next generation
236         fitnesses = for populant in population
237           eval "populant.#{symregr.FITNESS}(points)"
238
239         # Prevent the selection and reproduction of any trees that
              are too complex
240         for pop, i in population
241           if pop.tree.maxDepth() >= symregr.MDEPTH
242             fitnesses[i] = 0
243
244         # Find the most fit individual in current population
245         bestFitnessInPopulation = fitnesses.reduce(max, 0)
246
247         # Test primary termination condition
248         if bestFitnessInPopulation >= symregr.MAXFITNESS
249           return unless symregr.FITNESS in ['scaledFitness', '
```

```
            paretoFitness']
250           Individual.particleSwarmOptimization(population[
                maxIndex], points)
251         else
252           scaledIndividual(
253             Individual.particleSwarmOptimization(population[
                  maxIndex], points), points, symregr.IDEPTH)
254
255       # Update the fittest individual found if the fittest
            individual
256       # from this generation is more fit than the fittest
            individual found
257       # thus far.
258       if bestFitnessInPopulation > eval "bestIndividual.#{
            symregr.FITNESS}(points)"
259         bestIndividual = population[maxIndex]
260
261     # The maximum number of iterations have been performed, so
          we
262     # return the fittest individual that has been found.
263     unless symregr.FITNESS in ['scaledFitness', 'paretoFitness']
264       Individual.particleSwarmOptimization(bestIndividual,
            points)
265     else
266       scaledIndividual(Individual.particleSwarmOptimization(
            bestIndividual, points), points, symregr.IDEPTH)
267
268   # Optimized symbolic regression implementation with
        deterministic population
269   # ratios and batch selection and reproduction. Comments
        omitted for brevity.
270   window.optimizedSymbolicRegression = (points) ->
271
272     maxIndex = 0
273     I = "Individual"
274     max = (pv, cv, index, array) ->
275       if cv is Math.max(pv, cv) then maxIndex = index
276       Math.max(pv, cv)
277
278     population = (new Individual(null, symregr.IDEPTH) for i in
          [0...symregr.POPSIZE])
279     fitnesses = for populant in population
280       eval "populant.#{symregr.FITNESS}(points)"
281
282     for pop, i in population
```

```
283           if pop.tree.maxDepth() >= symregr.MDEPTH
284             fitnesses[i] = 0
285
286         mostFit = fitnesses.reduce max, 0
287         bestIndividual = population[maxIndex]
288
289         if mostFit >= symregr.MAXFITNESS
290           return unless symregr.FITNESS in ['scaledFitness', '
                paretoFitness']
291           Individual.particleSwarmOptimization(bestIndividual,
                points)
292           else
293           scaledIndividual(Individual.particleSwarmOptimization(
                population[maxIndex], points), points, symregr.IDEPTH
                )
294
295         newPopulation = []
296         for i in [0...symregr.MAXITERS]
297           newPopulation = []
298           sumFitnesses = fitnesses.map((y) -> if isNaN(y) then 0
                else y).reduce((pv, cv, index, array) -> pv + cv)
299           distribution = for _, index in fitnesses
300             cumulativeFitness = fitnesses[0..index].reduce (pv, cv,
                index, array) -> pv + cv
301             cumulativeFitness / sumFitnesses
302           switch symregr.REPRODUCTION
303             when 'fpReproduce'
304               for _ in [0...symregr.RBS]
305                 newPopulation.push(
306                   eval("#{I}.#{symregr.REPRODUCTION}(population,
                        points, symregr.FITNESS, distribution)"))
307             when 'susReproduce'
308               children =
309                 eval "#{I}.#{symregr.REPRODUCTION}(population,
                      points, symregr.FITNESS, symregr.RBS,
                      distribution)"
310               for child in children
311                 newPopulation.push child
312             when 'tournamentReproduce'
313               for _ in [0...symregr.RBS]
314                 newPopulation.push(
315                   eval("#{I}.#{symregr.REPRODUCTION}(population,
                        points, symregr.FITNESS, symregr.TSIZE, symregr
                        .TPR)"))
316
```

```
317         parents = []
318         switch symregr.SELECTION
319           when 'fpSelection'
320             for _ in [0...(2 * Math.round(symregr.CBS / 4))]
321               parents.push(eval("#{I}.#{symregr.SELECTION}(
                      population, points, symregr.FITNESS, distribution
                      )"))
322               parents.push(eval("#{I}.#{symregr.SELECTION}(
                      population, points, symregr.FITNESS, distribution
                      )"))
323           when 'susSelection'
324             size = (2 * Math.round(symregr.CBS / 4))
325             parents = eval "#{I}.#{symregr.SELECTION}(population,
                     points, symregr.FITNESS, size, distribution)"
326           when 'tournamentSelection'
327             for _ in [0...(2 * Math.round(symregr.CBS / 4))]
328               parents.push(
329                 eval("#{I}.#{symregr.SELECTION}(population, points
                        , symregr.FITNESS, symregr.TSIZE, symregr.TPR)
                        "))
330               parents.push(
331                 eval("#{I}.#{symregr.SELECTION}(population, points
                        , symregr.FITNESS, symregr.TSIZE, symregr.TPR)
                        "))
332         children = []
333         for j in [0...parents.length] by 2
334           res = eval("#{I}.#{symregr.CROSSOVER}(parents[j],
                 parents[j+1])")
335           children.push res[0]
336           children.push res[1]
337         newPopulation = newPopulation.concat(children)
338         switch symregr.SELECTION
339           when 'fpSelection'
340             for _ in [0...symregr.MBS]
341               mutant = eval "#{I}.#{symregr.SELECTION}(population,
                      points, symregr.FITNESS, distribution)"
342               newPopulation.push(eval("#{I}.#{symregr.MUTATION}(
                      mutant)"))
343           when 'susSelection'
344             mutants = eval "#{I}.#{symregr.SELECTION}(population,
                     points, symregr.FITNESS, symregr.MBS, distribution)
                     "
345             for mutant in mutants
346               newPopulation.push(eval("#{I}.#{symregr.MUTATION}(
                      mutant)"))
```

```
347             when 'tournamentSelection'
348               for _ in [0...symregr.MBS]
349                 mutt = eval "#{I}.#{symregr.SELECTION}(population,
                        points, symregr.FITNESS, symregr.TSIZE, symregr.
                        TPR)"
350                 newPopulation.push(eval("#{I}.#{symregr.MUTATION}(
                        mutt)"))
351
352           population = newPopulation
353           fitnesses = for populant in population
354             eval "populant.#{symregr.FITNESS}(points)"
355
356           for pop, i in population
357             if pop.tree.maxDepth() >= symregr.MDEPTH
358               fitnesses[i] = 0
359
360           bestFitnessInPopulation = fitnesses.reduce(max, 0)
361           if bestFitnessInPopulation >= symregr.MAXFITNESS
362             return unless symregr.FITNESS in ['scaledFitness', '
                    paretoFitness']
363             Individual.particleSwarmOptimization(population[
                      maxIndex], points)
364           else
365             scaledIndividual(
366               Individual.particleSwarmOptimization(population[
                        maxIndex], points), points, symregr.IDEPTH)
367           if bestFitnessInPopulation > eval "bestIndividual.#{
                  symregr.FITNESS}(points)"
368             bestIndividual = population[maxIndex]
369
370         unless symregr.FITNESS in ['scaledFitness', 'paretoFitness']
371           Individual.particleSwarmOptimization(bestIndividual,
                  points)
372         else
373           scaledIndividual(Individual.particleSwarmOptimization(
                  bestIndividual, points), points, symregr.IDEPTH)
```

7.2 Selected Symbolic Regression Performance Data

All regressions summarized below were calculated with scaled fitness and particle swarm optimization. As previously noted, our system was unable to find accurate regression functions in such a small execution window without additional numerical optimization. Each regression performed 100 generations of search over a population of 100 individuals. An individual's maximum depth was limited to 9. The reproduction, crossover, and mutation probabilities were .6, .3, and .1, respectively. Each row of the table denotes the average results achieved running the algorithm ten times with the specified parameters.

Our results describe the accuracy a Symbolic Regression can achieve within an execution window reasonable for interactivity. Empirical results suggests the R^2 value of the regression function is inversely-proportional with the size of the individual required to represent it. This seems reasonable given that the cardinality of the subset of all trees with a fixed depth increases exponentially with the trees' depth.

While our results do not show a single technique that outperformed others, They suggest that SUS is outperformed by both the fitness-proportional and tournament selection algorithms. Additionally, we note that tournament selection seems to be more effective when the objective function can be represented by a small individual. Similarly, fitness-proportional selection tends to increasingly out perform the other techniques as the optimal individual becomes larger. This increase in accuracy tends to be accompanied by an increase in the average size and depth of individuals discovered. This could be indicative of over fitting.

These statements cannot be proven from the empirical data we collected. However, the evident trends unique to specific crossover and selection algorithms suggests that such inquiry warrants future work. The results are summarized in Figure 15, beginning on the next page.

Function	Reproduction and Selection	Crossover	R^2	Tree Size	Tree Depth	Optimal Tree Size	Optimal Tree Depth
$\frac{1}{x}$	fitness-proportional selection	cut and splice	1	9.9	4.5	3	1
$\frac{1}{x}$	fitness-proportional selection	one-point	.999	10.4	5	3	1
$\frac{1}{x}$	fitness-proportional selection	two-point	.928	11.1	5.2	3	1
$\frac{1}{x}$	tournament selection	cut and splice	1	11.3	4.8	3	1
$\frac{1}{x}$	tournament selection	one-point	1	12.3	4.8	3	1
$\frac{1}{x}$	tournament selection	two-point	1	13.1	5.8	3	1
$\frac{1}{x}$	stochastic universal sampling	cut and splice	.798	9.8	4.2	3	1
$\frac{1}{x}$	stochastic universal sampling	one-point	.805	11.6	5.2	3	1
$\frac{1}{x}$	stochastic universal sampling	two-point	.716	11.7	5	3	1

Function	Reproduction and Selection	Crossover	R^2	Tree Size	Tree Depth	Optimal Tree Size	Optimal Tree Depth
$\tanh x$	fitness-proportional selection	cut and splice	.768	11.6	5.6	15	4
$\tanh x$	fitness-proportional selection	one-point	.725	13.7	6.3	15	4
$\tanh x$	fitness-proportional selection	two-point	.768	15.4	5.8	15	4
$\tanh x$	tournament selection	cut and splice	.327	15.3	7.2	15	4
$\tanh x$	tournament selection	one-point	.584	19.5	7.4	15	4
$\tanh x$	tournament selection	two-point	.509	16.7	7.1	15	4
$\tanh x$	stochastic universal sampling	cut and splice	.818	18	6.9	15	4
$\tanh x$	stochastic universal sampling	one-point	.723	15.3	6.2	15	4
$\tanh x$	stochastic universal sampling	two-point	.863	11.3	5.5	15	4

Function	Reproduction and Selection	Crossover	R^2	Tree Size	Tree Depth	Optimal Tree Size	Optimal Tree Depth
$-3(x+1)^{10} - 38.9$	fitness-proportional selection	cut and splice	.641	29.8	8.4	8	4
$-3(x+1)^{10} - 38.9$	fitness-proportional selection	one-point	.422	18.5	7.5	8	4
$-3(x+1)^{10} - 38.9$	fitness-proportional selection	two-point	.525	17.1	7.3	8	4
$-3(x+1)^{10} - 38.9$	tournament selection	cut and splice	.290	21.4	7.6	8	4
$-3(x+1)^{10} - 38.9$	tournament selection	one-point	.587	30.2	7.5	8	4
$-3(x+1)^{10} - 38.9$	tournament selection	two-point	.380	26.1	7.7	8	4
$-3(x+1)^{10} - 38.9$	stochastic universal sampling	cut and splice	.155	12.7	5.8	8	4
$-3(x+1)^{10} - 38.9$	stochastic universal sampling	one-point	.412	15.7	6.1	8	4
$-3(x+1)^{10} - 38.9$	stochastic universal sampling	two-point	.213	18	7.2	8	4

Function	Reproduction and Selection	Crossover	R^2	Tree Size	Tree Depth	Optimal Tree Size	Optimal Tree Depth
$10x \sin x + \frac{4 \cos x}{x}$	fitness-proportional selection	cut and splice	.959	15.4	6.1	13	4
$10x \sin x + \frac{4 \cos x}{x}$	fitness-proportional selection	one-point	.743	17.5	6.9	13	4
$10x \sin x + \frac{4 \cos x}{x}$	fitness-proportional selection	two-point	.897	12.4	5.7	13	4
$10x \sin x + \frac{4 \cos x}{x}$	tournament selection	cut and splice	.999	12.3	4.7	13	4
$10x \sin x + \frac{4 \cos x}{x}$	tournament selection	one-point	.999	12.2	4.8	13	4
$10x \sin x + \frac{4 \cos x}{x}$	tournament selection	two-point	.999	14.1	5.4	13	4
$10x \sin x + \frac{4 \cos x}{x}$	stochastic universal sampling	cut and splice	.685	14.3	6	13	4
$10x \sin x + \frac{4 \cos x}{x}$	stochastic universal sampling	one-point	.565	16.3	7.2	13	4
$10x \sin x + \frac{4 \cos x}{x}$	stochastic universal sampling	two-point	.408	14.9	6.2	13	4

Function	Reproduction and Selection	Crossover	R^2	Tree Size	Tree Depth	Optimal Tree Size	Optimal Tree Depth
$\frac{1}{1+e^{-x}}$	fitness-proportional selection	cut and splice	.559	20.3	7	8	4
$\frac{1}{1+e^{-x}}$	fitness-proportional selection	one-point	.734	12.6	5.9	8	4
$\frac{1}{1+e^{-x}}$	fitness-proportional selection	two-point	.570	13.3	5.8	8	4
$\frac{1}{1+e^{-x}}$	tournament selection	cut and splice	.771	16.5	6.5	8	4
$\frac{1}{1+e^{-x}}$	tournament selection	one-point	.574	19.4	6.7	8	4
$\frac{1}{1+e^{-x}}$	tournament selection	two-point	.594	14	5.9	8	4
$\frac{1}{1+e^{-x}}$	stochastic universal sampling	cut and splice	< 0	17	6.3	8	4
$\frac{1}{1+e^{-x}}$	stochastic universal sampling	one-point	.407	14.3	6.1	8	4
$\frac{1}{1+e^{-x}}$	stochastic universal sampling	two-point	< 0	15.3	6.6	8	4

Function	Reproduction and Selection	Crossover	R^2	Tree Size	Tree Depth	Optimal Tree Size	Optimal Tree Depth
$-11.7\ln(x^2+1)+25.8\cos x+17.2$	fitness-proportional selection	cut and splice	.712	27	8.8	15	6
$-11.7\ln(x^2+1)+25.8\cos x+17.2$	fitness-proportional selection	one-point	.454	17.9	7.2	15	6
$-11.7\ln(x^2+1)+25.8\cos x+17.2$	fitness-proportional selection	two-point	.468	21.2	8.3	15	6
$-11.7\ln(x^2+1)+25.8\cos x+17.2$	tournament selection	cut and splice	.599	17	6.3	15	6
$-11.7\ln(x^2+1)+25.8\cos x+17.2$	tournament selection	one-point	.556	14.3	5.6	15	6
$-11.7\ln(x^2+1)+25.8\cos x+17.2$	tournament selection	two-point	.450	14.3	5.6	15	6
$-11.7\ln(x^2+1)+25.8\cos x+17.2$	stochastic universal sampling	cut and splice	.300	15.4	7.2	15	6
$-11.7\ln(x^2+1)+25.8\cos x+17.2$	stochastic universal sampling	one-point	.373	13.4	6.3	15	6
$-11.7\ln(x^2+1)+25.8\cos x+17.2$	stochastic universal sampling	two-point	.236	15.2	6.3	15	6

Function	Reproduction and Selection	Crossover	R^2	Tree Size	Tree Depth	Optimal Tree Size	Optimal Tree Depth
$2.4x^3 - 11.2x^2 + 8.6x - 52.1$	fitness-proportional selection	cut and splice	.870	40.2	8.9	17	5
$2.4x^3 - 11.2x^2 + 8.6x - 52.1$	fitness-proportional selection	one-point	.800	26.6	8.2	17	5
$2.4x^3 - 11.2x^2 + 8.6x - 52.1$	fitness-proportional selection	two-point	.928	20.4	7.7	17	5
$2.4x^3 - 11.2x^2 + 8.6x - 52.1$	tournament selection	cut and splice	.595	20.2	5.9	17	5
$2.4x^3 - 11.2x^2 + 8.6x - 52.1$	tournament selection	one-point	.841	21.6	6.4	17	5
$2.4x^3 - 11.2x^2 + 8.6x - 52.1$	tournament selection	two-point	.600	19.4	6.6	17	5
$2.4x^3 - 11.2x^2 + 8.6x - 52.1$	stochastic universal sampling	cut and splice	.662	17.1	6.3	17	5
$2.4x^3 - 11.2x^2 + 8.6x - 52.1$	stochastic universal sampling	one-point	.574	17.5	6.4	17	5
$2.4x^3 - 11.2x^2 + 8.6x - 52.1$	stochastic universal sampling	two-point	.632	17.5	6.1	17	5

Function	Reproduction and Selection	Crossover	R^2	Tree Size	Tree Depth	Optimal Tree Size	Optimal Tree Depth
$e^{0.1x} + \frac{\tan 2x}{\sqrt{\lvert x \rvert}} + 256.6$	fitness-proportional selection	cut and splice	.481	21.3	7.9	19	6
$e^{0.1x} + \frac{\tan 2x}{\sqrt{\lvert x \rvert}} + 256.6$	fitness-proportional selection	one-point	.513	15.5	7	19	6
$e^{0.1x} + \frac{\tan 2x}{\sqrt{\lvert x \rvert}} + 256.6$	fitness-proportional selection	two-point	.517	13	6.3	19	6
$e^{0.1x} + \frac{\tan 2x}{\sqrt{\lvert x \rvert}} + 256.6$	tournament selection	cut and splice	.410	18.2	6.7	19	6
$e^{0.1x} + \frac{\tan 2x}{\sqrt{\lvert x \rvert}} + 256.6$	tournament selection	one-point	.244	20	7	19	6
$e^{0.1x} + \frac{\tan 2x}{\sqrt{\lvert x \rvert}} + 256.6$	tournament selection	two-point	.380	19.7	7.3	19	6
$e^{0.1x} + \frac{\tan 2x}{\sqrt{\lvert x \rvert}} + 256.6$	stochastic universal sampling	cut and splice	.413	17.5	6.4	19	6
$e^{0.1x} + \frac{\tan 2x}{\sqrt{\lvert x \rvert}} + 256.6$	stochastic universal sampling	one-point	.498	16.2	6.5	19	6
$e^{0.1x} + \frac{\tan 2x}{\sqrt{\lvert x \rvert}} + 256.6$	stochastic universal sampling	two-point	.308	12.9	5.4	19	6

Figure 15: Symbolic Regression performance data with different implementation techniques.