Universiteit Leiden
The Netherlands

Opleiding Informatica

Symbolic Regression of Implicit Functions

F. E. C. Ruijter

Supervisors:
Dr. M. T. M. Emmerich & H. Wang, MSc

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
`www.liacs.leidenuniv.nl`

24/08/2017

Contents

Chapter 1

Introduction

Suppose we do some kind of experiment, we measure some variables and end up with a dataset. Now we want relationships between the variables, which help to describe our system.

We could first think hard about the problem and try to come up with a model, which we then test on the gathered data. However it might be hard to produce a theoretical model that accurately explains the data. In this thesis we will investigate a specific algorithm for finding a relationship between the variables in the dataset.

More precisely, if we measured variables $x, y, z, \ldots$, then we are looking for some expression using some of these variables that is zero on the dataset, but not just identically zero everywhere. For example, if y is given by a function f of x then $y - f(x) = 0$ is a good solution.

In particular we are trying to find an implicit function, i.e. a function g such that $g(x, y, z, \ldots) = 0$ (or close to zero at least) for all measurements and such that g is given by an expression with prescribed operators. The operators could include $+$ and $\times$ for example, but sin and cos as well.

The idea of finding the shape of a formula for a set of data is known as symbolic regression. Symbolic regression contrasts with normal regression, which is about finding the values for the constants in a given formula that best fit with the observed data.

Chapter 2

A Theoretical Description of The Problem

Let us first clearly define which problem we are trying to solve.

We have done some measurements producing a data series $(Y_i)_i \in \mathbb{R}^n$, where each Y_i represents a measurement vector, for example a time and a position. In the simplest instance of this problem we want to find a function f, such that $f(Y_i) = 0$, for all i, because as long as $f \neq 0$, this tells us something about the data. For example if $f(x, y) = xy - 1 = 0$ then $y = 1/x$.

We note that if we restrict f to be k times continuously differentiable, that this is a description of a C^k submanifold of $\mathbb{R}^n$ containing the data. If we restrict f to be a linear, non-zero functional, then this describes a hyperplane in $\mathbb{R}^m$.

Considering that even the space of candidate solutions is extremely large, due to the required choice of constants, it becomes interesting how closely we can approximate f based on the data. In this thesis we present an approach using an evolutionary algorithm.

Chapter 3

Related Work

The idea of applying evolution to programs goes back all the way to Alan Turing who proposed it in his article [1], as a solution strategy for solving what would later be known as the turing challenge.

In this challenge, one is supposed to write a computer program which comunicates with the user via text. A number of judges get the task of figuring out whether they are talking with a computer program or via a computer to another human. The goal is to make your program fool the judges.

Turing proposed evolving computer programs for this task, inspired by how humans evolved, but he never actually implemented this. The ideas were not forgotten tough and an influential theoretical treatment [2] was published in 1975 by John Holland, which spurred further devolopment that lead to the first succesful application as showcased by Richard Forsyth [3] in 1981.

Another milestone was the book "Genetic Programming" by John Koza in 1992 [4]. *Genetic Programming* aims to evolve computer programs giving certain outputs given certain inputs. This book gives a comprehensive introduction to genetic programming, including the problem of Symbolic regression. Suppose one has a variable x determining a variable y and we have a series of data points (x_i, y_i). A candidate solution $\hat{f}$ is a function of x, it is good when $\hat{f}(x_i)$ is close to y_i. Koza suggests to take

$$\mathrm{Fit}(\hat{f}) = \sum_i |y_i - \hat{f}(x_i)|$$

as the fitness function. This can easily be generalised to multiple dependent and independent variables. However symbolic regression of implicit functions is not treated in this book.

The most succesful work in symbolic regression of implicit functions is by Schmidt and Lipson [5]. Their research showed that working with the values of a candidate solution f does not work well. In spirit of the previous example, one might try setting $y_i = 0$ for each i and take $\mathbf{x}$ to be the vector of all measured variables, whith measurements $(\mathbf{x}_i)_i$ and then take for a candidate solution $\hat{f}$

$$\mathrm{Fit}(\hat{f}) = \sum_i |\hat{f}(\mathbf{x}_i)|$$

as the fitness function as implied by Koza. However, for every f we see that $\hat{f}(\mathbf{x}) = 0$ minimises the fitness, which is exactly the trivial solution we need to avoid. Various methods to deal with this while still using the values $\hat{f}(\mathbf{x}_i)$ directly, run into problems as described by Schmidt and Lipson.

For example, if one simplifies every candidate first and heavily penalises 0, one will most likely find that a certain simplification was not found by the simplification algorithm. And if one were to look at the value on a certain point away from the data, one would just evolve solutions which spike near this point and are (almost) zero everywhere else. See Table 1-1 in their book chapter [5] for more direct methods and their difficulties.

They found that it is much better to work with the derivative $\nabla\hat{f}$, we describe it in detail in section 5.4.1.

Chapter 4

A Mathematical Overview

Let (Y_i) be a sequence of measurements and f an underlying relation, that is, if the measurements are without noise $f(Y_i) = 0$ for each i. We assume that f is analytic. We set $Z = f^{-1}[\{0\}]$, this is the set of measurements that could occur according to this relation. Assume for now that Z is connected.

4.1 Bounding the Error of a Candidate Solution

The fact that it works to just look at ∇f is in essence a consequence of the main theorem of calculus, $\int_a^b h'(t)dt = h(b) - h(a)$. In the context of multi-variable calculus it still applies if we use ∇h and a path integral. We will show that if a candidate solution g has a close derivative to f, its values $g(\mathbf{x})$ are close to the values $f(\mathbf{x})$ up to a constant.

Let us choose a measurement $\mathbf{z}_0 = \mathbf{Y}_i$ and call for every i the length of the shortest path in Z between $\mathbf{z}_0$ and $\mathbf{Y}_i$ by $l(\mathbf{Y}_i)$. We set $D := \max_i l(\mathbf{Y}_i)$ to be the maximum of the lengths these shortest paths.
Theorem 1. *Assume there exists an $\epsilon > 0$ such that for all $\mathbf{z} \in Z$ we get $||\nabla g(\mathbf{z}) - \nabla f(\mathbf{z})|| < \epsilon$, then for all $\mathbf{z} \in Z$ we also get $-D\epsilon < g(\mathbf{z}) - g(\mathbf{z}_0) < D\epsilon$.*

Proof. Indeed write $h(\mathbf{z}) = g(\mathbf{z}) - g(\mathbf{z}_0)$, so that $h(\mathbf{z}_0) = 0$, then $|h(\mathbf{z})| = |h(\mathbf{z}_0) - (h(\mathbf{z}_0) - h(\mathbf{z}))| = |h(\mathbf{z}_0) + \int_{\mathbf{z}_0}^{\mathbf{z}} \nabla h(\mathbf{x}) \cdot d\mathbf{x}|$. Here we applied the gradient theorem.

Now we apply the triangle inequality, fill in h and introduce f. We find that

$$\begin{aligned}
|h(\mathbf{z})| &= |h(\mathbf{z}_0) + \int_{\mathbf{z}_0}^{\mathbf{z}} \nabla h(\mathbf{x}) \cdot d\mathbf{x}| \\
&\leq |g(\mathbf{z}_0) - g(\mathbf{z}_0)| + |\int_{\mathbf{z}_0}^{z} (\nabla g(\mathbf{x}) - \nabla(g(\mathbf{z}_0))) \cdot d\mathbf{x}| \\
&= |\int_{\mathbf{z}_0}^{\mathbf{z}} \nabla g(\mathbf{x}) \cdot d\mathbf{x}| \\
&= |\int_{\mathbf{z}_0}^{\mathbf{z}} (\nabla g(\mathbf{x}) - \nabla f(\mathbf{x})) \cdot d\mathbf{x} + \int_{\mathbf{z}_0}^{\mathbf{z}} \nabla f(\mathbf{x}) \cdot d\mathbf{x}| \\
&= |\int_{z_0}^{z} (\nabla g(\mathbf{x}) - \nabla f(\mathbf{x})) \cdot d\mathbf{x}|.
\end{aligned}$$

Finally we estimate the absolute value of this integral. Let P be the shortest path from $\mathbf{z}_0$ to $\mathbf{z}$ in Z, then we already know that its length is $l(\mathbf{z})$ so that $|\int_{z_0}^{z} (\nabla g(\mathbf{x}) - \nabla f(\mathbf{x})) \cdot d\mathbf{x}| = |\int_P (\nabla g(\mathbf{x}) - \nabla f(\mathbf{x})) \cdot d\mathbf{x}| \leq l(\mathbf{z}) \cdot \sup_{\mathbf{x} \in P} ||\nabla g(\mathbf{x}) - \nabla f(\mathbf{x})||$.

We also know that $l(\mathbf{z}) \leq D$ and that $||\nabla g(\mathbf{x}) - \nabla f(\mathbf{x})|| < \epsilon$ for all $\mathbf{x} \in Z \supset P$. We conclude that $|h(\mathbf{z})| \leq l(\mathbf{z}) \cdot \sup_{\mathbf{x} \in P} ||\nabla g(\mathbf{x}) - \nabla f(\mathbf{x})|| < D\epsilon$ as we needed to prove. □

Of course we don't actually know f, it is the goal. By doing linear regressions around a point p in Z we estimate the tangent hyperplane around p of f, that is $\ker(\nabla f(p))$. Since this hyperplane has codimension 1 it determines $\nabla f(p)$ up to multiplication by a constant $c(p) \in \mathbb{R}$.

Let us now prove that we can actually take $||\nabla f(z)|| = 1$ for all $z \in Z$ where z is not a critical point. We set $\hat{f}(z) = f(z)/||\nabla f||$. Note that this is only well defined whenever $\nabla f(z) \neq 0$, that is whenever z is not a critical point. If we further assume that f is analytic, that is, f has a locally convergent power series for every point of its domain, then the set where $\nabla f(z) = 0$ has measure zero.

Let us now calculate the gradient of $\hat{f}$, by using the product rule:

$$\nabla \hat{f}(z) = \frac{\nabla f}{||\nabla f||} + f(z)\nabla\left(\frac{1}{||\nabla f||}\right)$$

Note that $f(z) = 0$ for all $z \in Z$ such that the second term drops out. Then we find $||\nabla \hat{f}(z)|| = \frac{||\nabla f(z)||}{||\nabla f(z)||} = 1$, while $\hat{f}(z) = f(z)/||\nabla f(z)|| = 0$.

So from now on let us assume that f has a gradient of norm one on Z. We will normalise the linear approximations as well. If the approximation had no measurement errors, then this would make it equal to the gradient of f up to multiplication by -1, as they have to give rise to the same tangent plane.

Suppose that we have a candidate solution g which has a gradient of norm 1 on all of the data points, then we can rewrite $||\nabla f(z) - \nabla g(z)||$ using the innerproduct $\langle \nabla f(z), \nabla g(z)\rangle$ by writing it out as $||\nabla f(z) - \nabla g(z)||^2 = \langle \nabla f(z) - \nabla g(z), \nabla f(z) - g(x)\rangle = \langle \nabla f(z), \nabla f(z)\rangle + \langle \nabla g(z), \nabla g(z)\rangle - 2\langle \nabla f(z), \nabla g(z)\rangle = 2(1 - \langle \nabla f(z), \nabla g(z)\rangle)$.

We conclude that as g estimates f better, $1 - \langle \nabla f(z), \nabla g(z)\rangle$ estimates the quality of g better.

4.2 Overfitting

One should be careful not to admit solutions that are too complicated. In general this problem is called overfitting. For example, if we let L_i be the best linear approximations of f around Y_i with $||\nabla L_i|| = 1$ then we can find a function $h(z) = \prod_i L_i(z)$ which is zero on all Y_i:

$$h(Y_j) = \prod_i L_i(Y_j) = L_j(Y_j)\prod_{i\neq j} L_i(Y_j) = 0$$

Indeed $\hat{h}(z) = h(z)/||\nabla h||$ has both the same values, as well as the same gradients as f at all Y_i:

$$\begin{aligned}\nabla h(Y_j) &= \sum_k \nabla L_k(Y_j)\prod_{i\neq k} L_i(Y_j)\\ &= \nabla L_j(Y_j)\prod_{i\neq j} L_i(Y_j)\end{aligned}$$

Here we use that $\prod_{i\neq k} L_i(Y_j) = 0$ whenever $k \neq j$, since one term is $L_j(Y_j)$ and this is zero by definition. So we see that h has gradients in the right directions, but with different lenght. However if we normalize it we get:

$$\begin{aligned}\nabla\left(\frac{h}{||\nabla h||}\right)(Y_j) &= \left(\frac{\nabla h}{||\nabla h||}\right)(Y_j) + h(Y_j)\nabla\left(\frac{1}{||\nabla h||}\right)(Y_j)\\ &= \left(\frac{\nabla h}{||\nabla h||}\right)(Y_j)\\ &= \frac{\nabla L_j(Y_j)\prod_{i\neq j} L_i(Y_j)}{||\nabla L_j(Y_j)\prod_{i\neq j} L_i(Y_j)}\\ &= \frac{\nabla L_j(Y_j)\prod_{i\neq j} L_i(Y_j)}{||\nabla L_j(Y_j)\prod_{i\neq j} L_i(Y_j)||}\end{aligned}$$

Note that $\nabla L_j(Y_j) = \nabla f(Y_j)$ and $||\nabla f(Y_j)|| = 1$

$$\begin{aligned}\nabla\left(\frac{h}{||\nabla h||}\right)(Y_j) &= \frac{\nabla f(Y_j)}{||\nabla f(Y_j)||}\frac{\prod_{i\neq j} L_i(Y_j)}{|\prod_{i\neq j} L_i(Y_j)|} \\ &= \nabla f(Y_j)\cdot \pm 1\end{aligned}$$

In practice, we can only know ∇f up to a constant from the data, this means that $\hat{h} = h/||\nabla h||$ is as good at estimating f from the data as possible.

This means that one should not allow expressions that are too large, because otherwise one will get this solution that essentially memorises all the datapoints Y_i and the gradient of f there.

Chapter 5

The Algorithm

We use a simple evolutionary algorithm, shown in Algorithm 1. Essentially we take a member of the population, copy, crossover and mutate it, and then remove the very worst member of the population, as evaluated by the fitness function. This is known as an elitist steady state algorithm in the literature.

5.1 Example

Let us first describe an example to elucidate this type of algorithm. We look at a solution space consisting of binary sequences of some length n. Our goal is to find a sequence $(b_1, \ldots, b_n)$ with a minimal weight $w = \sum_{i=1}^{n} b_i$. Clearly the sequence with minimal weight is just $(0, 0, \ldots, 0)$, but we want to solve this problem using an evolutionary algorithm.

For our evolutionary algorithm, we need to specify the individuals, the mutation operator and the fitness function. Elements of the solution space are a natural choice for the individuals. We will mutate by flipping a randomly chosen bit and we will evaluate the fitness if an individual by calculating its weight. We note that the simplest mutation operator independently flips each bit with a certain probability. The same estimate as below can be obtained in this case. For convenience of exposition we will use the former mutation operator.

We will use a population of size p and randomly choose a single individual to mutate, discarding afterwards the worst individual in the population. We will now give an estimate for the required number of generations to solve this problem.

Let T_i denote the best individual in the population after i steps. We can now calculate that $P(w(T_{i+1}) = k | w(T_i) = k+1) > \frac{1}{p}\frac{k+1}{n}$, where the right side just indicates the probability of improving the best solution. Let us write S_k for number of generations spent with the lowest weight in the population being k. By realising that S_k is just the first time that an improvement occurs, we see that S_k can be overestimated by a geometric distribution with parameter $\frac{k+1}{np}$ such that $E[S_k] < \frac{np}{k+1}$.

Now we estimate the required number of generations as $\sum_{i=1}^{n} S_i = np \sum_{i=1}^{n} \frac{1}{k+1} < np \int_1^{n+1} 1/t \; dt = np \log n$.

Of course this is just a toy model, but we do see that this converges much quicker than a just randomly guessing, which would take 2^n tries on average. Because we neglected all the suboptimal solutions in our analysis, it suggests choosing p as small as possible (i.e. 1), this does not generalise however.

5.2 Individuals

The individuals in our evolutionary algorithm to find relations are formulas, which are modelled as trees. Trees admit many operations in an easy fashion. The internal nodes of the tree represent operations like

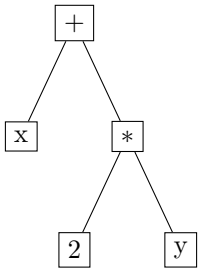


Figure 5.1: A tree representing the expression $x + 2y$.

adding, multiplying or applying some other function, while the leaves contain references to variables and constants. For example $x + 2 * y$ will yield the tree in Figure 5.1

For our experiments, we have just used + and $*$ as internal nodes and real constans (possibly negative) and variables as leaves. We have also restricted the number of nodes to 20, because the hash function that was used to distinguish two individuals was only exact up to 20 nodes.

5.3 Estimation of The Search Space

Let us try to estimate how large the search space in the symbolic regression problem is. For each leaf we either have a constant, or a variable. Since we optimise the constants afterwards, we essentially have only one option. For now we will assume that we have two variables.

Internal nodes must either be + or $*$ so there are two options for those. Leaves could be a constant or either of the two variables, which leaves us with three options for each leaf. Indeed our expressions form binary trees. A binary tree always has an odd number of nodes $2k + 1$ of wich k are internal nodes and $k + 1$ are leaves.

So given a tree structure with $2k + 1$ nodes, we have $2^k \cdot 3^{k+1}$ individuals. Notice that we do not apply any simplification or normalisation: $2 + 3$, 5 and $3 + 2$ are all different individuals.

The number of full binary trees with k internal nodes is given by the k-th Catalan number C_k.

Since we limit our trees to 20 (i.e. 19) nodes, we get at least

$$I_9 = C_9 \cdot 2^9 \cdot 3^{10} = 146993273856 \approx 2^{37}$$

different individuals, where I_k is the number of individuals with k internal nodes. The total number of individuals with less than 19 nodes is $\sum_{k=0}^{9} I_k < \sum_{k=0^9} \frac{1}{6^{9-k}} I_9 < I_9 \sum_{k=0}^{\infty} 6^{-k} = \frac{I_9}{1-1/6} = \frac{6}{5} I_9 < 2^{38}$.

We see that we can store our trees as sequences of 38 bits. If our mutation operator were as simple as the one in the example and our fitness operator could just count the distance from the optimal solution, we would expect convergence in about $38 \log 38 \approx 138$ generations, assuming $p = 1$. Of course the fact the our operators are quite different and that the role of p is not correctly captured by this example means that this estimate cannot give more than a guideline.

Let us now return to describing the evolutionary algorithm.

5.4 Evaluating the Fitness

Schmidt and Lipson have reported that for implicit functions of two variables x and y, it works well to compare differently estimated derivatives. We will first give their method and then describe various other fitness functions inspired by it.

5.4.1 The Fitness Function of Schmidt and Lipson

This is essentially the fitness function that was reported in [5].

A relationship between x and y is a function $f(x,y)$ such that $f(X_i,Y_i)=0$, for all (noiseless) datapoints $(X_i,Y_i)_i$. From the gradient ∇f, we can gather the local linear relationships by using Taylor's Theorem. As long as $\nabla f(x,y)\neq 0$ the function f is locally approximated by a non-zero linear one.

Let $L_i(x,y)=\alpha_i x+\beta_i y+\gamma_i=0$ be linear approximations to the data around $p_i:=(X_i,Y_i)$. Then we can estimate $\left[\frac{\partial y}{\partial x}\right]_{p_i}$ as $-\left(\left[\frac{\partial L_i}{\partial x}\right]_{p_i}\Big/\left[\frac{\partial L_i}{\partial y}\right]_{p_i}\right)=-\alpha_i/\beta_i$. At the same time we can calculate this derivative using f as $\left[\frac{\partial y}{\partial x}\right]_{p_i}=\left(\left[\frac{\partial f}{\partial x}\right]_{p_i}\Big/\left[\frac{\partial f}{\partial y}\right]_{p_i}\right)$. So if g is a candidate solution then $-\left(\left[\frac{\partial g}{\partial x}\right]_{p_i}\Big/\left[\frac{\partial g}{\partial y}\right]_{p_i}\right)$ should estimate $\left[\frac{\partial y}{\partial x}\right]_{p_i}$.

We can calculate the partial derivatives $\left[\frac{\partial g}{\partial x}\right]_{p_i}$ using symbolic methods, although we opted to estimate them using a finite difference method instead:

$$\left[\frac{\partial g}{\partial x}\right]_{p_i}=\frac{g(x+h)-g(x-h)}{2h}+O(h^2)$$

A comparison of these two estimates of $\left[\frac{\partial y}{\partial x}\right]_{p_i}$ now tells us how well g estimates a local linear relation. Let us write $\delta_i x=\left[\frac{\partial f}{\partial x}\right]_{p_i}$ and $\delta_i y=\left[\frac{\partial f}{\partial y}\right]_{p_i}$. Suppose that there are N data points, then Schmidt and Lipson suggest the fitness function

$$\mathrm{Fit}(g)=\frac{1}{N}\sum_i^N \log(1+\left|\left(-\frac{\alpha_i}{\beta_i}\right)-\left(-\frac{\delta_i x}{\delta_i y}\right)\right|).$$

The extra $\log(1+\ldots)$ is there to reduce the impact of points where there is not a good fit.

In practice we calculate the coefficients of L_i by a linear regression of the closest neighbours of p_i.

5.4.2 Example Calculation

Let us calculate this fitness function in a simple case now. We take $f(x,y)=x^2+y^2-1$ as the target function, $g(x,y)=x^2-1-y$ as the candidate solution, and $p_1=(X_1,Y_1)=(\frac{1}{2}\sqrt{2},-\frac{1}{2}\sqrt{2})$ and $p_2=(X_2,Y_2)=(-\frac{1}{2}\sqrt{2},-\frac{1}{2}\sqrt{2})$ as our only two data points. We assume that we have the exact gradients of f around p_1 and p_2 available.

We calculate $L_1(x,y)=x-\sqrt{2}-y=0$ and $L_2(x,y)=-x-\sqrt{2}-y=0$, so that $-\alpha_1/\beta_1=1$ and $-\alpha_2/\beta_2=-1$.

When we calculate the gradient of g we find $\nabla g=(2x,-1)^T$, so that $-\frac{\delta_1 x}{\delta_1 y}=-\frac{2\frac{1}{2}\sqrt{2}}{-1}=\sqrt{2}$, while $-\frac{\delta_2 x}{\delta_2 y}=-\frac{2\cdot-\frac{1}{2}\sqrt{2}}{-1}=-\sqrt{2}$.

We can now calculate the fitness as

$$\mathrm{Fit}(g)=\frac{1}{2}\log(1+|1-\sqrt{2}|)+\frac{1}{2}\log(1+|(-1--\sqrt{2}|)=2\frac{1}{2}\log(\sqrt{2})=\frac{1}{2}\log 2\approx 0.35$$

5.5 Extending to Many Variables

In the multi-variate case it is not directly obvious which measure of distance between the local linear approximations to the data and the local derivative of the candidate solution is best.

One option is to take the sum of absolute differences. Let us call the coordinates by (x_i) and

$$\left[\frac{\delta x_i}{\delta x_j}\right]_{p_i} := -\frac{\left[\frac{\partial g}{\partial x_j}\right]_{p_i}}{\left[\frac{\partial g}{\partial x_i}\right]_{p_i}}$$

be the estimate of $\left[\frac{\partial x_i}{\partial x_j}\right]_{p_i}$ using a candidate solution g and

$$\left[\frac{\Delta x_i}{\Delta x_j}\right]_{p_i} := -\frac{\left[\frac{\partial L_i}{\partial x_j}\right]_{p_i}}{\left[\frac{\partial L_i}{\partial x_i}\right]_{p_i}}$$

be the estimate using a local linear regression.

One method is to simply take the sums of absolute differences:

$$\mathrm{Fit}_{sad}(g) = \sum_i \log(1 + \sum_{j\neq k} \left| \left[\frac{\delta x_k}{\delta x_j}\right]_{Y_i} - \left[\frac{\Delta x_k}{\Delta x_j}\right]_{Y_i} \right|).$$

Another is to look at the sums of squared differences

$$\mathrm{Fit}_{ssd}(g) = \sum_i \log(1 + \sum_{j\neq k} \left(\left[\frac{\delta x_k}{\delta x_j}\right]_{Y_i} - \left[\frac{\Delta x_k}{\Delta x_j}\right]_{Y_i} \right)^2).$$

However the value of $\left(\left[\frac{\delta x_k}{\delta x_j}\right]_{Y_i} - \left[\frac{\Delta x_k}{\Delta x_j}\right]_{Y_i} \right)$ obtained for j and k is not independent of the value obtained for k and j, even if the exact dependency is hard to formulate. So maybe taking the sum of independent absolute differences is better:

$$\mathrm{Fit}_{sisd}(f) = \sum_i \log \left(1 + \sum_{j<k} \min \left(\left(\left[\frac{\delta x_j}{\delta x_k}\right]_{Y_i} - \left[\frac{\Delta x_j}{\Delta x_k}\right]_{Y_i} \right)^2, \left(\left[\frac{\delta x_k}{\delta x_j}\right]_{Y_i} - \left[\frac{\Delta x_k}{\Delta x_j}\right]_{Y_i} \right)^2 \right) \right)$$

We decided to take the minimum, because it gets rid of an instability due to the data being too close to an axis. For example suppose that at a certain data point p we have $\nabla f(p) = (0.999, 0.045)^T$, while a hypothetical solution g has $\nabla g(p) = (0.99, 0.14)^T$. This seems quite similar, but

$$\frac{\left[\frac{\partial f}{\partial x_1}\right]_{p_i}}{\left[\frac{\partial f}{\partial x_2}\right]_{p_i}} - \frac{\left[\frac{\partial g}{\partial x_1}\right]_{p_i}}{\left[\frac{\partial g}{\partial x_2}\right]_{p_i}} \approx 22.34 - 7.02 \approx 15.3,$$

while on the other hand

$$\frac{\left[\frac{\partial f}{\partial x_2}\right]_{p_i}}{\left[\frac{\partial f}{\partial x_1}\right]_{p_i}} - \frac{\left[\frac{\partial g}{\partial x_2}\right]_{p_i}}{\left[\frac{\partial g}{\partial x_1}\right]_{p_i}} \approx 0.0448 - 0.143 \approx -0.102.$$

We see that when we take the minimum, such instabilities are avoided.

In a similar way we define $\mathrm{Fit}_{siad}(g)$, the fitness by taking the sum of the independent absolute differences.

Finally we can take the Euclidean inner product to assess how well the gradient of f estimates the gradient of the data:

$$\mathrm{Fit}_{sip}(g) = \sum_i \log \left(1 + \left(1 - \frac{|\langle [\nabla g]_{Y_i}, [\nabla L_i]_{Y_i} \rangle|}{\|[\nabla g]_{Y_i}\| \cdot \|[\nabla L_i]_{Y_i}\|} \right) \right).$$

We already saw in Section 4.1 that it makes sense to look at the the inner product of the gradients to estimate the norm of the difference between f and g. In particular we saw that

$$1 - \frac{\langle [\nabla f]_{Y_i}, [\nabla L_i]_{Y_i} \rangle}{\|[\nabla f]_{Y_i}\| \cdot \|[\nabla L_i]_{Y_i}\|} = \frac{1}{2} \|[\nabla f]_{Y_i} - [\nabla L_i]_{Y_i}\| .$$

Since we only know $[\nabla L_i]_{Y_i}$ up to a sign, a negative inner product should not be penalised, so we take the absolute value. The logartihm is there because it was easiest on the implementation. It will have little effect since $\log(1 + x) \approx x$ for x close to zero by Taylor's Theorem.

5.6 Generating Mutations

The other essential part of the algorithm is mutating the individuals. We are supplied with an individual g to mutate.

Depending on the configuration, we may first decide to apply a crossover operation. In this case we select a second individual g and combine f and g to continue the proces. For example we may produce $g * h$.

Next we apply a mutation operation to the tree. For example we may change a variable to a constant. Finally we optimise the constants.

If it turns out that the resulting number of nodes is too large (the maximum is 20 by default) or the an expression of this structure was already tried, then we first try to apply the mutation operation to the original g again. If after 5 restarts still no valid mutation was generated, we choose a new g.

Without restarting, the mutation process may get in an infinite loop, when all instances one mutation away from g have already been examined.

5.6.1 Crossover

If crossover is enabled there is a chance of 0.05 that we apply it. It consists of randomly choosing a second individual g and combining f and g as $f + g$ or $f * g$. Which is used depends on what is enabled, if both are enabled either happens with equal chance.

We choose $f + g$ and $f * g$, because theoretically it seems it might work: If f and g both are small on the dataset, then probably $f + g$ is also small on the dataset. One might expect that the deviations from zero of f and g are independent, such that they cancel on average and $f + g$ has smaller deviations from 0.

On the other hand $f * g$ is zero whenever either f or g is zero.

5.6.2 Mutations

We decided to adopt local mutations. That means that we first choose a mutation point in the tree and then apply one of the following transformations on this node:

1. *constant-or-variable* $\longrightarrow$ *constant-or-random-variable* : If the selected node is a leaf, we may change it to a constant, or any variable.
2. *constant-or-variable* $\longrightarrow$ *constant-or-variable plus-or-times constant-or-random-variable* : If the selected node n is a leaf, we may replace it by $l + l_2$ or $l * l_2$, where l_2 is a random constant or variable.
3. *plus-or-times* $\longrightarrow$ *times-or-plus* : If the node is an internal node, we can change the type of internal node it is.
4. x *plus-or-times* y $\longrightarrow$ x | y : If a node is an internal node, we can replace it by one of its children.

An expression is mutated by visiting every node in the tree in post-order. That is, first the children are potentially recursively mutated. Once the children are done, we need to decide if this node should be mutated and which mutation should be chosen.

The default is that every node is mutated with a probability of 0.2. This means that compared to giving all nodes the same probability, the first nodes have a higher probability to be selected and due to the order they are always low in the tree. All possible mutations of this node occur with equal weight. Since we go over each node and decide whether this should be the mutation point, it is possible, although unlikely, that no node is picked. In this case we simply restart.

5.6.3 Optimising Constants

It turns out that randomly picking the constants does not lead to a quick convergence. Doing a local search to optimise the constants is necessary, several methods were tried, by default a Newton iteration is used to optimise the fitness value as a function of the constants, but we also implemented two other methods, they are all compared in the experiments section.

If we have a candidate solution $g_{\mathbf{c}}(\mathbf{x})$, and a fitness function $\mathrm{Fit}(g_{\mathbf{c}})$, we can see the fitness as a function of the values of the constants as $h(\mathbf{c}) = \mathrm{Fit}(g_{\mathbf{c}})$.

In a Newton iteration, we start with an initial guess $\mathbf{c}_0$ and then we recursively define new values for each $i = 1, \ldots$ via

$$\mathbf{c}_{i+1} = \mathbf{c}_i + (H_h^{-1})_{\mathbf{c}_i}(\nabla h)_{\mathbf{c}_i},$$

where $(H_h)_{\mathbf{c}_i}$ is the Hessian of h around $\mathbf{c}_i$, that is, the Jacobian of ∇h. Typically this method will converge fairly quickly to a critical point, but calculating the Hessian is quite expensive.

Another option is to use a gradient descent method. For this we first calculate the gradient of $h(\mathbf{c})$ around $\mathbf{c}_i$, which we call γ. Next we try to find the $\alpha \in \mathbb{R}$ such that $T(\alpha) = h(\mathbf{c} + \alpha\gamma)$ is minimal. If we neglect higher order terms, this is at $\alpha = T'(0)/T''(0)$. This yields the formula:

$$\mathbf{c}_{i+1} = \mathbf{c}_i + \frac{T'(0)}{T''(0)}\gamma.$$

Finally we can do a pattern search. This method is quite different because it does not involve derivatives at all. Let $\mathbf{e}_1, \ldots, \mathbf{e}_n$ the unit coordinate vectors corresponding to vectors of constants $\mathbf{c}$. Define $D_i = \{\mathbf{c}_i + 2^{-i}\mathbf{e}_j \mid j = 1, \ldots n\}$, that is all the options to tak a step of size 2^{-i} on the j-th coordinate.

Then

$$\mathbf{c}_{i+1} = \arg\min_{\mathbf{c} \in D_i} h(\mathbf{c})$$

Algorithm 1 The main evolutionary algorithm.

```
1:  procedure EvolutionaryAlgorithm(P, N, S)
2:      for k := 1, ..., N do
3:          Pick an individual f ∈ P, or f = 0 if P = ∅.
4:          P := P ∪ {MutationOf(f)}
5:          if |P| > S then
6:              ĝ := arg max_{g∈P} {Fitness(g)}
7:              P := P \ {ĝ}
8:          end if
9:      end for
10:     return P
11: end procedure
```

5.7 Validation

Each time, we let a formula evolve using a small sample from the N data points (Y_i), then for the final evolved formula g we calculate a validation score using all available data points as

$$\mathrm{Val}(g) = \sqrt{\frac{1}{N}\sum_i \left(\frac{g(Y_i) - \mu(g)}{||\nabla g(Y_i)||}\right)^2}$$

where $\mu(g) = \frac{1}{N}\sum_i g(Y_i)$ is the average value of g on the data. When $\nabla g = 0$ we ignore the point (decreasing N) and when more than 5% of the points is rejected this way, we return ∞ as the validation score.

We chose this formula, because $(g-\mu(g))/||\nabla g||$ is unaffected by adding and multiplying g with constants. Indeed we want to find the contour line of g that most closely matches the data, which is the one for $g = \mu(g)$. On this contour line we normalise the magnitude of the gradient to 1 by dividing by $||\nabla g||$.

In Figure5.2 we can see how this normalization affects a candidate relation $g(x, y) = x^2 - y$ of a dataset satisfying $f(x, y) = x^2 + y^2 - 1 = 0$, where all the data points (x, y) have $y < 0$.

In the limit $N \to \infty$ we can estimate this as

$$\mathrm{Val}(g) \approx \sqrt{\int \left(\frac{g(\mathbf{y}) - \mu(g)}{||\nabla g(\mathbf{y})||}\right)^2 d\mathbf{y} \Big/ \int 1 \, d\mathbf{y}}.$$

In the same way we calculate $\mu(g)$ in the limit as

$$\mu(g) = \int g(\mathbf{y}) d\mathbf{y} \Big/ \int 1 \, d\mathbf{y} \; .$$

Standard integration techniques show that this is equal to $\frac{1}{2} + \frac{2}{\pi}$ on the lower half circle $\{(x, y) \mid x^2 + y^2 = 1 \wedge y \leq 0\}$.

We also calculate $\nabla g(x, y) = \begin{pmatrix} 2x \\ -1 \end{pmatrix}$ and so $||\nabla g|| = \sqrt{4x^2 + 1}$.

Unfortunately the integral

$$\mathrm{Val}(g) \approx \sqrt{\int \frac{(x^2 - y + \frac{1}{2} + \frac{2}{\pi})^2}{4x^2 + 1} d\mathbf{y} \Big/ \pi}$$

seems very hard to integrate symbolically, so we estimated it using numerical software as $\mathrm{Val}(g) \approx 0.0587$. If one does the same calculation for $g'(x, y) = y$ then one finds that $\mathrm{Val}(g') \approx 0.31$, which is larger, because it is a worse fit, but it is not extremely large, because it is still approximates the data fairly well.

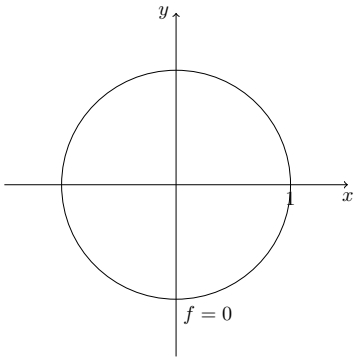
If one chooses an estimating function g'' whose gradient has a pole in the dataset, then the validation score becomes infinite in the limit. $g''(x, y) = yx^2$ shows this behaviour, for example.

5.8 Results

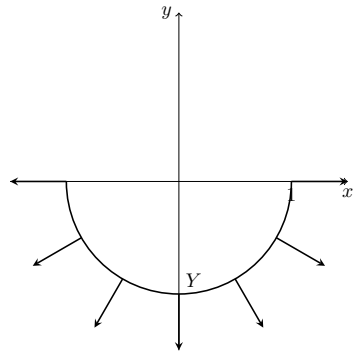
Let us remind the reader that for an evolved relation g we compute for each datapoint Y the value $\frac{g(Y)-\mu(g)}{||\nabla g(Y)||}$ and take the square root of the average of the squares of these values to compute the validation score:

$$\mathrm{Val}(g) = \sqrt{\frac{1}{N}\sum_i \left(\frac{g(Y_i) - \mu(g)}{||\nabla g(Y_i)||}\right)^2}$$

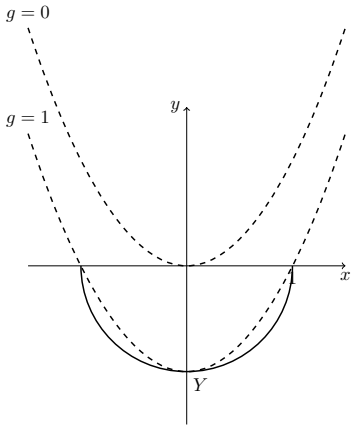
In fact this subexpression $\frac{g(Y)-\mu(g)}{||\nabla g(Y)||}$ estimates the distance from Y to the zero set of g, so we feel justified to claim that if the the validation score is significantly smaller than the range of the data, that g is a good solution and hence that the algorithm succeeded.



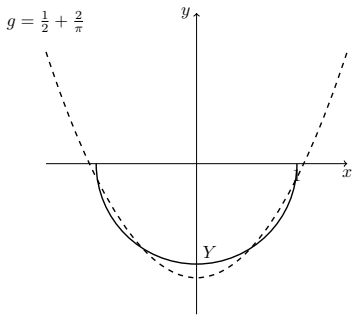
(a) The generating function.



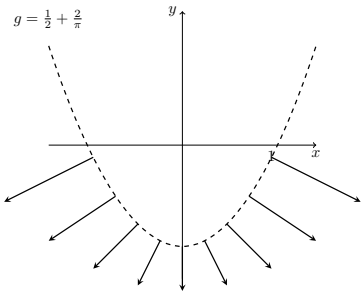
(b) The generated data Y. The data with normalized gradients superimposed. Note that the arrows could be in the opposite direction.



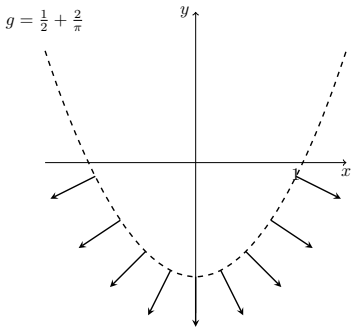
(c) Two contour lines of g superimposed on the data Y.



(d) The contour line $g(x, y) = \mu(g) = \frac{1}{2} + \frac{2}{\pi}$ minimizing the squared difference with the data Y.



(e) The chosen contour with the the gradient superimposed.



(f) The contour $(g-\mu(g))/\nabla g = 0$, with the associated gradient.

Figure 5.2: An example illustrating how normalizing an approximation $g(x, y) = x^2 - y$ before validation affects its gradient and contour lines.

From Figure 5.3 we conclude that the algorithm works well except for: *random-quartic, random-implicit-curve* and *line-circle-interspersed*. The reason for this is that with the default settings, the target relation of these data sets is unreachable, because it requires too many nodes for its expression.

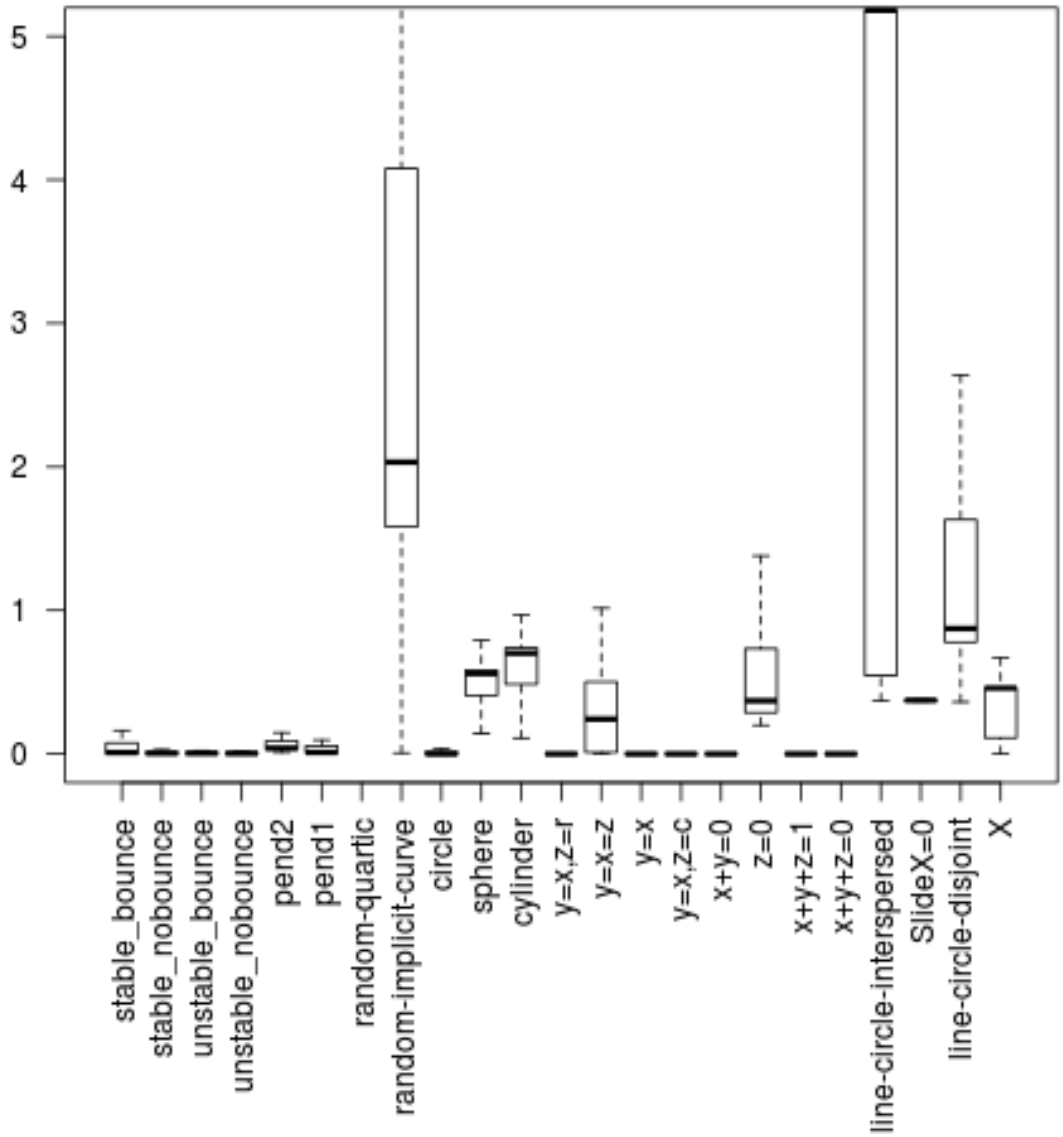


Figure 5.3: boxplots of validation at default settings.

Chapter 6

Experiments

Our experiments our focused on finding out how to effectively execute the basic ideas given by Schmidt and Lipson, and to give evidence for the choices they describe.

1. pattern search vs. gradient descent vs. (quasi-)Newton to search the constants.
2. Using a constant probability to pick nodes for mutations, or smaller based on number of children.
3. Forcing a mutation, or not.
4. Pruning multiple occurence of the same instance.
5. Various fitness formula's.
6. The size of the selected sample.
7. Crossover vs. pure genetic programming.

To compare these, we created a suite of data sets to test the options mentioned above. Let us describe them now:

1. *stable-bounce*: Contains (y_i, v_i) modelling a falling mass. The step formula is $y_{n+1} = y_n + (v_n + v_{n+1})/(h/2)$ for the position and $v_{n+1} = v_n - 10h$ for the speed. If y_n is negative a bounce is executed: $v_{n+1} = -v_n$ and $y_{n+1} = y_n$. The initial conditions are $v_0 = 3$ and $y_0 = 0$.
2. *stable-nobounce*: Contains (y_i, v_i) modelling a falling mass. The step formula is $y_{n+1} = y_n + (v_n + v_{n+1})/(h/2)$ for the position and $v_{n+1} = v_n - 10h$ for the speed. The data is cropped so that y_n is never negative. The initial conditions are $v_0 = 3$ and $y_0 = 0$.
3. *unstable-bounce*: Contains (y_i, v_i) modelling a falling mass. The step formula is$y_{n+1} = y_n + v_n/h$ for the position and $v_{n+1} = v_n - 10h$ for the speed. If y_n is negative a bounce is executed: $v_{n+1} = -v_n$ and $y_{n+1} = y_n$. The initial conditions are $v_0 = 3$ and $y_0 = 0$.
4. *unstable-nobounce*: Contains (y_i, v_i) modelling a falling mass. The step formula is$y_{n+1} = y_n + v_n/h$ for the position and $v_{n+1} = v_n - 10h$ for the speed. The data is cropped so that y_n is never negative. The initial conditions are $v_0 = 3$ and $y_0 = 0$.
5. *pend2*: Contains pairs (x_i, y_i) modelling the position of a swinging pendulum. The fixed point of the pendulum is at $(0, 1)$, the initial position is $(\frac{1}{2}\sqrt{2}, 1 - \frac{1}{2}\sqrt{2})$ and its initial speed is zero at first. We consider the angle θ_i and the angular speed α_1. We use the update formulas $\alpha_{i+1} = \alpha_i - hg \sin\theta_i$ and $\theta_{i+1} = \theta_1 + (\alpha_{i+1} + \alpha_{i+1})/(h/2)$. We store $v_{x,i} = \cos\alpha_i, v_{y,i} = \sin\alpha_i$ and $y_i = \sin\theta_i$. One relation among these variables is the fact that the total energy is constant: $v_x^2 + v_y^2 + 2g(1 - y) = c$.
6. *pend1*: Contains pairs (x_i, y_i) modelling the position of a swinging pendulum. The fixed point of the pendulum is at $(0, 1)$, the initial position is $(\frac{1}{2}\sqrt{2}, 1 - \frac{1}{2}\sqrt{2})$ and its initial speed is zero at first. We consider the angle θ_i and the angular speed α_1. We use the update formulas $\alpha_{i+1} = \alpha_i - hg \sin\theta_i$ and $\theta_{i+1} = \theta_1 + (\alpha_{i+1} + \alpha_{i+1})/(h/2)$. We store $x_i = \cos\theta_i$ and $y_i = \sin\theta_i$.

7. *random-quartic*: Contains points (x_i, y_i). A random polynomial f of degree at most 4 was generated by choosing integer coefficients uniformly at random from $-10, \ldots, 10$. x_i is uniformly distributed on $-5, 5$ and $y_i = f(x_i)$. The generated polynomial was $-10 \cdot x^0 + -7 \cdot x^1 + -3 \cdot x^2 + -9 \cdot x^3 + -3 \cdot x^4$.

8. *random-implicit-curve*: Contains points (x_i, y_i). A random polynomial $f(X, Y) = \sum_{k=0}^{2} \sum_{l=0}^{2} a_{k,l} X^k Y^l$ was chosen by taking each $a_{k,l}$ uniformly at random from the integers $-10, \ldots, 10$. Then we have taken points (x'_i, y'_i) uniformly from $[-5, 5] \times [-5, 5]$. Of course such points are not on $f = 0$, so we applied a root finding algorithm to obtain the point (x_i, y_i), by starting with (x'_i, y'_i) and applying newton iterations to the function $(x, y) \mapsto (f(x, y), 0)$. The generated polynomial was $0 = -1 \cdot x^0 y^2 + 1 \cdot x^3 y^0 + -1 \cdot x^1 y^0$

9. *circle*: Contains points (x_i, y_i) on the unit circle. The angle is uniformly distributed on $[0, 2\pi]$.

10. *sphere*: Contains points (x_i, y_i, z_i). x_i and y_i are first chosen independent and uniformly distributed on $[-1, 1]$, but pairs with $x_i^2 + y_i^2 > 1$ are rejected. Finally $z_i = \pm\sqrt{1 - x_i^2 - y_i^2}$, with equal probability for positive and negative values of z_i.

11. *cylinder*: Contains points (x_i, y_i, z_i). x_i and y_i are independent and uniformly distributed on $[-1, 1]$ and $z_i = \pm\sqrt{1 - x_i^2}$, with equal probability for positive and negative values of z_i.

12. *y=x,z=r*: Contains pairs (x_i, y_i, z_i), where $x_i = y_i$ and x_i and z_i are independent and uniformly distributed on $[0, 1]$.

13. *y=x=z*: Contains pairs (x_i, y_i, z_i), where $x_i = y_i = z_i$ and x_i is uniformly distributed on $[0, 1]$.

14. *y=x*: Contains pairs (x_i, y_i), where $x_i = y_i$ and x_i is uniformly distributed on $[0, 1]$.

15. *y=x,z=c*: Contains pairs (x_i, y_i, z_i), where $x_i = y_i$ and x_i is uniformly distributed on $[0, 1]$, while $z_i = 1$.

16. *x+y=0*: Contains pairs (x_i, y_i), where $x_i = -y_i$ and x_i is uniformly distributed on $[0, 1]$.

17. *z=0*: Contains pairs (x_i, y_i, z_i), where $z_i = 0$ and x_i and y_i are independent and uniformly distributed on $[0, 1]$.

18. *x+y+z=1*: Contains pairs (x_i, y_i, z_i), where $x_i + y_i = 1 - z_i$ and x_i and y_i are independent and uniformly distributed on $[0, 1]$.

19. *x+y+z=0*: Contains pairs (x_i, y_i, z_i), where $x_i + y_i = -z_i$ and x_i and y_i are independent and uniformly distributed on $[0, 1]$.

20. *line-circle-interspersed*: Contains points (x_i, y_i). Half of the points are uniformly distributed on the line segment between $(0, 0)$ and $(1, 1)$, the other half are uniformly distributed on a circle of radius 1 around $(0, 0)$.

21. *SlideX=0*: Contains points (x_i, y_i, z_i). The points are generated by taking two random values r, s independently from $[0, 1]$. Then we add the points $(r, s, -r - s)$ and $(r, -s, -r + s)$.

22. *line-circle-disjoint*: Contains points (x_i, y_i). Half of the points are uniformly distributed on the line segment between $(0, 0)$ and $(1, 1)$, the other half are uniformly distributed on a circle of radius 1 around $(2, 0)$.

23. *X*: Contains points (x_i, x_i) and $(x_i, -x_i)$, where x_i is uniformly distributed on $[0, 1]$.

6.0.1 Comparison of Options

Let us now describe how to decide which of several options is better. First, we will restrict ourselves to a single test case, since it can be meaningfully said that option A works better on a certain testcase than option B, but whether it works better in an application depends on whether ones application is like the particular testcase. Suppose there are two options, A and B, then we generate two dataseries by running the program and recording the validation scores in $(X_i)_i$ and $(Y_i)_i$ respectively. Recall that this score reflects how far away this solution is from the data, so a lower score is better. The results X_i and Y_i

are typically not all equal, because the evolution is based on randomness, and sometimes the program is more 'lucky' than other times.

We use the Mann-Whitney U-test, which we will now describe. See section 11.2.3 of [6] for a more comprehensive treatment.

(X_i) and (Y_i) are sequences of indpendent identically distributed variables. Our null hypothesis is that for each i and j the validation scores X_i and Y_j are identically distributed, which means that A and B are equally good. In this case we have that $\mathbb{P}(X_i < Y_j) = \mathbb{P}(X_i > Y_j)$ for all i and j. Let us set

$$Z_{ij} = \begin{cases} 1 & \text{If } X_i < Y_j \\ 0 & \text{Else} \end{cases}$$

Now we calculate the test statistic

$$U = \sum_{i,j} Z_{ij}$$

Now let u_0 be the observed value of U. Given a significance level α we accept the null-hypothesis that both methods are equally good if

$$P(U < u_0) \geq \alpha$$

and otherwise reject it, implying that option A is better than option B.

Upfront, we have counted how many hypothesis tests we will do in total, say c and we set the signifcance level to $\alpha := 0.05/c$, this is called the Bonferroni correction, see note 12.2.2.2 in [6].

6.1 Algorithm Parameters

We need to decide how large of a population to keep and how many points of data to use in the evolution. One might be tempted to use all available data points, but this turns out not only to be unnecessary, but to make the algorithm unusably slow.

So we select a sample and calculate a local linear regression around each point in the sample using all points of the dataset.

If we get n-dimensional data, we use $3n^2$ data points by default and the local regressions around them. We have tested whether the size of this sample has any effect on the solution quality.

We conclude from the results in Table 6.3 that it is not beneficial to use 32 or more data points, except perhaps in the simplest of cases. It seems likely that in this case overfitting occurs. It is somewhat surprising such small numbers of data points provide enough information already.

Next we wanted to know how large we should choose our population. By default it is set to 100. We tested population sizes of 25, 50, 100 and 200.

The results in Table 6.1 show that for the simple linear tests a smaller population is better, but for the more complicated tests, no significant difference was observed.

6.2 Local Search

It is imperative that a solution uses the right constants, otherwise it will never be any good. However, which method should one use for the optimisation of the constants?

Since calculating the fitness for a given set of constants is relatively expensive, it is also important to use not too many fitness calls. Indeed a poorer optimisation of the local constants, may allow for more generations of the evolutionary algorithm if We compared a Newton iteration, a gradient descent and a pattern search as described in section 5.6.3

All methods are run until $h(\mathbf{c}_i) - h(\mathbf{c}_{i+1}) < 0.1$, i.e. as long as sufficient progress is being made.

Comparison	N_s	Tests
50-formulas < 100-formulas	0	(y=x,z=r: 0.00213), (line-circle-interspersed: 0.00281)
50-formulas < 200-formulas	0	(y=x,z=r: 7.03e-05)
200-formulas < 50-formulas	0	(unstable-nobounce: 0.0182)
200-formulas < 100-formulas	0	(pend2: 0.0237)
200-formulas < 25-formulas	0	(circle: 0.0165)
25-formulas < 50-formulas	0	(y=x: 5.7e-05), (y=x,z=c: 5e-05)
25-formulas < 100-formulas	2	y=x, y=x,z=c, (unstable-bounce: 0.0151), (y=x,z=r: 5.59e-05)
25-formulas < 200-formulas	3	y=x,z=r, y=x, y=x,z=c, (pend1: 0.0225)

Table 6.1: Comparison of local searches using our test set procedure.

Comparison	N_s	Tests
Newton < Gradient-descent	4	random-implicit-curve, y=x,z=r, y=x,z=c, X, (pend2: 0.00317), (pend1: 0.00876), (y=x=z: 0.00959)
Newton < Pattern-Search	14	stable-bounce, stable-nobounce, unstable-bounce, unstable-nobounce, pend2, pend1, random-implicit-curve, y=x,z=r, y=x=z, y=x, y=x,z=c, line-circle-interspersed, SlideX=0, X
Gradient-descent < Newton	0	(random-quartic: 0.0133)
Gradient-descent < Pattern-Search	14	stable-bounce, stable-nobounce, unstable-bounce, unstable-nobounce, pend2, pend1, random-implicit-curve, y=x,z=r, y=x=z, y=x, y=x,z=c, line-circle-interspersed, SlideX=0, X, (random-quartic: 0.0115)
Pattern-Search < Newton	2	circle, line-circle-disjoint, (cylinder: 0.000345), (z=0: 0.000242)
Pattern-Search < Gradient-descent	3	circle, z=0, line-circle-disjoint, (cylinder: 4.28e-05)

Table 6.2: Comparison of local searches using our test set procedure.

Comparison	N_s	Tests
4-points < 32-points	8	stable-nobounce, unstable-bounce, unstable-nobounce, random-quartic, random-implicit-curve, sphere, cylinder, line-circle-interspersed, (circle: 0.000397), (y=x=z: 4.32e-05), (X: 0.000118)
4-points < 16-points	6	stable-nobounce, unstable-bounce, unstable-nobounce, random-quartic, random-implicit-curve, line-circle-interspersed, (circle: 9.62e-05), (sphere: 0.00103), (y=x=z: 0.000269)
4-points < 8-points	4	stable-nobounce, unstable-nobounce, random-quartic, random-implicit-curve, (unstable-bounce: 0.000158), (y=x=z: 0.0029), (line-circle-interspersed: 0.00211)
32-points < 4-points	5	pend2, pend1, y=x,z=r, y=x, y=x,z=c, (stable-bounce: 0.0116), (z=0: 0.0134), (SlideX=0: 0.00169)
32-points < 16-points	3	y=x,z=r, y=x, y=x,z=c, (pend2: 0.0152), (pend1: 0.00403)
32-points < 8-points	3	y=x,z=r, y=x, y=x,z=c, (stable-bounce: 0.0108), (pend2: 4.03e-05), (pend1: 0.00716), (SlideX=0: 0.00605)
16-points < 4-points	3	y=x,z=r, y=x, y=x,z=c, (pend2: 0.00276), (pend1: 0.00496), (z=0: 0.000416), (SlideX=0: 0.00328)

16-points < 32-points	3	stable-nobounce, unstable-bounce, unstable-nobounce, (random-implicit-curve: 0.000263), (cylinder: 0.00574)
16-points < 8-points	3	y=x,z=r, y=x, y=x,z=c, (SlideX=0: 0.0116)
8-points < 4-points	3	y=x,z=r, y=x, y=x,z=c, (pend1: 0.00133)
8-points < 32-points	8	stable-nobounce, unstable-bounce, unstable-nobounce, random-quartic, random-implicit-curve, cylinder, line-circle-interspersed, X, (circle: 0.024), (sphere: 6.12e-05)
8-points < 16-points	4	stable-nobounce, unstable-nobounce, random-quartic, line-circle-interspersed, (unstable-bounce: 0.00885), (random-implicit-curve: 0.00185), (circle: 0.00678), (sphere: 0.0068), (cylinder: 0.0123), (X: 0.00837)

Table 6.3: Comparison of various sample sizes in our algorithm using the test procedure.

Algorithm	Mean Fitness Calls
Newton	19561
Gradient-descent	11755
Pattern-Search	1736

Table 6.4: The amount of fitness calls used over the entire run of the program, averaged over all tests and 800 runs.

As we can see from the results in Table 6.2, pattern search works quite poorly except on the two tests containing a circle. Newton iteration and Gradient Descent work about equally well and much better than Pattern Search on most of the other tests.

It is also of interest which of these methods use more calls to the fitness function and which use less. The fitness function is memorized, so calling it unnecessarily again, will typically not lead to extra calls being counted.

It is clear that the methods using derivatives, like Newton Iteration, will require more fitness calls per step. On the other hand, because they typically converge more quickly, they will typically take fewer steps. In Table 6.4

We see here that the derivative based methods typically require the most fitness calls, which is not all that surprising.

6.3 Fitness

While Schmidt and Lipson give a good approach for a dataset with two variables, it is not clear how this should be generalised to larger numbers of variables. We gave several suggestions in paragraph 5.5.

From Table 6.6 we conclude that using the sum of squared errors, Fit_{ssd}, is a bad idea, it performs much worse than the other two. It is much better still to take the inner product over the sum of independent squared errors.

6.4 Mutation

The first thing we want to investigate is how to choose the node where the local mutation should take place. We have also researched whether one should force a mutation to happen or not, and whether to prune multiple occurences of the same formula.

6.4.1 How to Choose the Node to Mutate

One reasonable option is to visit all nodes in pre order and pick each node with a predetermined probability. This gives nodes that are higher, and more left a higher probability than those that are more right and lower. This will be denoted by *inverse*.

Another option is to give each node the same probability of mutating, this will be denoted by *c-0.2* and *c-0.5*, when using the implied probabilities.

It seems counterproductive to delete subtrees, so maybe we should lower the probability of this by checking again wether the root of the subtree the subtree that would be deleted is should be modified. We use the same procedure as above for this.

No significant results were obtained, but as Table 6.5 show a preference for a constant probability of 0.2

6.4.2 Should Mutations be Forced?

One could wonder, is it worth it to rerun mutation if the previous run did not actually mutate. Allowing non-mutation will allow solutions to just clone themselves sometimes. For clarity, it is allowed to mutate an instance in a way leading to it being a copy of some other instance, just not to clone directly.

We have tested wether stopping cloning makes a difference, but no significant results were obtained. We conclude that it does not matter either way, perhaps because this is a rare event, or because non-direct ways of obtaining the same instances have the same effect.

Comparison	N_s	Tests
c-0.5-single < c-0.2-single	0	(y=x=z: 0.0108)
c-0.5-single < c-0.2-double	0	(y=x=z: 0.0175)
c-0.5-double < inverse-double	0	(stable-nobounce: 0.0139), (random-quartic: 0.0153)
inverse-double < c-0.5-single	0	(cylinder: 0.013)
inverse-double < c-0.2-double	0	(circle: 0.0194)
inverse-single < c-0.5-single	0	(cylinder: 0.0103)
inverse-single < inverse-double	0	(sphere: 0.0131)
c-0.2-double < inverse-double	0	(stable-nobounce: 0.0102), (random-quartic: 0.00264)
c-0.2-double < inverse-single	0	(random-quartic: 0.0187)

Table 6.5: Comparing options to choose the node to mutate. No significant results were found

Comparison	N_s	Tests
Sum-Squared-errors < Sum-Indep-sq-er	3	stable-bounce, pend2, SlideX=0, (pend1: 0.000257)
Sum-Squared-errors < Inner-Product	3	stable-bounce, pend2, SlideX=0, (x+y+z=0: 0.0127)
Sum-Indep-sq-er < Sum-Squared-errors	10	stable-nobounce, unstable-bounce, unstable-nobounce, random-quartic, random-implicit-curve, circle, y=x=z, y=x, line-circle-interspersed, X, (line-circle-disjoint: 0.000328)
Sum-Indep-sq-er < Inner-Product	0	(x+y+z=0: 0.00479)
Inner-Product < Sum-Squared-errors	14	stable-nobounce, unstable-bounce, unstable-nobounce, pend1, random-quartic, random-implicit-curve, circle, sphere, cylinder, y=x=z, y=x, line-circle-interspersed, line-circle-disjoint, X
Inner-Product < Sum-Indep-sq-er	9	pend2, pend1, random-implicit-curve, sphere, cylinder, y=x,z=r, y=x=z, y=x, y=x,z=c, (X: 0.000146)

Table 6.6: Comparison of various local fitness functions in our algorithm using the test procedure.

6.4.3 Excluding Repetitions

It appears that repetions of the same tree are unnecessary. Is it worthwile avoid duplicates? We implemented a simple hash function to compare trees. The hash function is injective if all trees have less then $64/\lfloor \log(V+3) \rfloor \approx 20$ nodes, where V is the number of variables allowed (not used).

On the one hand a solution which already appeared cannot be strictly better than all solutions seen yet, because it is just as good as itself. On the other hand, maybe boosting generation of children of this new solution is helpful.

Comparison	N_s	Tests
reject-duplicates < allow-duplicates	13	stable-bounce, stable-nobounce, unstable-bounce, unstable-nobounce, pend2, pend1, circle, sphere, y=x,z=r, y=x=z, y=x, y=x,z=c, X, (x+y+z=1: 9.25e-05), (x+y+z=0: 0.00135)
allow-duplicates < reject-duplicates	4	random-quartic, z=0, line-circle-interspersed, line-circle-disjoint, (random-implicit-curve: 0.00434)

Table 6.7: Comparing options to allow or reject duplicates. On almost every test a significant result was obtained, in most cases it is better to reject. Only in the most complicated tests this is not the case.

From Table 6.7 we can conclude that it is not helpful to have duplicates.

Comparison	N_s	Tests
no-crossover < geometric	0	(stable-bounce: 0.00153), (line-circle-disjoint: 0.0168)
no-crossover < mixed-crossover	0	(stable-bounce: 0.0243), (x+y+z=1: 0.00705)
geometric < mixed-crossover	0	(pend1: 0.0151), (X: 0.0188)
geometric < arithmetic	0	(pend1: 0.00677)
mixed-crossover < arithmetic	0	(pend2: 0.0222)
arithmetic < geometric	0	(stable-bounce: 0.0168), (x+y+z=0: 0.0142)

Table 6.8: Comparing options to enable crossover. No significant results, although *no-crossover* comes the closest.

6.5 Crossover

We implemented two types of Crossover, but is it actually worthwile, and if so, which one is best? We remind the reader the the types of Crossover are taking the sum of two previous instances, which corresponds to taking the arithmetic mean, and taking the product of two instances, which loosely corresponds to taking the geometric mean. We already described this in subsection 5.6.1

In Table 6.8 we have gathered the results. *mixed-crossover* means that crossover happens at the same rate as when *geometric* or *arithmetic* is used and both types have a chance of $\frac{1}{2}$ to occur.

While there are some promising results nothing significant was found, so if there is any effect at all, it will be small. We conclude that it is not necessary to implement these types of crossover.

It seems slightly surprising that *geometric* works well for *slideX*, but once one realises that the exact form the relation is a product of two planes, it is clear that trying to form products of candidate solutions would help. It is on the other hand quite surprising that the same does not apply to the test dataset X, maybe because it is too easy.

Chapter 7

Conclusions

We have found that using an evolutionary algorithm to find relationships in measured data is feasible. It is however critical to use a fitness function that can effectively recognize the trivial solution. Working with the gradient works very well for this and matching the gradient with the inner product has a theoretical basis.

It is also necessary to use an optimisation for the constants of your relation, because the evolutionary algorithm otherwise doesn't find the required constants in a reasonable amount of time. Although it may be an artifact of the stopping criterion, we advise to use Newton's method to perform this optimisation.

Although crossover is considered very important in genetic programming, we cannot confirm that it is helpful in this context. This may be due to the choice of the crossover operations.

It will be interesting to see if this system performs well in higher dimensional settings and with the addition of more function nodes.

It should also be interesting to see if the evaluation of instances can be sped up by compiling the trees to a linear form or even to machine code, although the current speed seems to be sufficient for daily use, even if a bit slow for experiments measuring its performance.

Bibliography

[1] A. M. Turing, "Computing machinery and intelligence," *Mind*, vol. 59, no. 236, pp. 433–460, 1950.

[2] J. H. Holland, *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control and Artificial Intelligence.* Cambridge, MA, USA: MIT Press, 1992.

[3] R. Forsyth, "Beagle—a darwinian approach to pattern recognition," *Kybernetes*, vol. 10, no. 3, pp. 159–166, 1981.

[4] J. R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection.* Cambridge, MA, USA: MIT Press, 1992.

[5] M. Schmidt and H. Lipson, "Symbolic regression of implicit equations," in *Genetic Programming Theory and Practice VII* (R. Riolo, U.-M. O'Reilly, and T. McConaghy, eds.), (Boston, MA), pp. 73–85, Springer US, 2010.

[6] J. A. Rice, *Mathematical Statistics and Data Analysis.* Belmont, CA: Duxbury Press., third ed., 2006.