Utrecht University

A Department of Information and Computing Sciences Master's Thesis

Sensitivity Analysis Based Feature-Guided Evolution for Symbolic Regression

Author:
S. de Vries

Supervisors:
Dr. Ir. D. Thierens
Dr. A.J. Feelders

June 7, 2018

Abstract

The problem of Symbolic Regression (SR) is to find a mathematical expression which best models a given dataset. Research into SR primarily takes place in Genetic Programming (GP), with the evolutionary algorithm called Standard GP (SGP) at its basis. In this work, we set out to improve upon SGP, using the Sensitivity-based Genetic Programming (SensGP) algorithm.

A thorough examination of SR literature in the field of GP led to the conclusion that algorithms which are improvements of SGP frequently enhance the search process. This is accomplished by guiding the evolution by using additional information about parts of solutions, called features. By conducting comparison experiments between algorithms from the literature, we confirmed this conclusion. As a result, the feature-guided evolutionary process was chosen to be the basis for SensGP.

At the core of the SensGP algorithm lies the use of sensitivity analysis to measure feature importance. The Mean Squared Error (MSE) of a feature is measured on the original data and on the data for which selected variables have had their values shuffled. The difference in these MSE values is used to calculate a feature importance score, which is later used to reintroduce these features into the population.

In addition to the basic SensGP algorithm, we experimented with two variants of SensGP: a Model Dependent approach, in which only the models with the lowest MSE are used for feature importance calculation, and a Variable Importance approach, in which feature importance is measured by the sensitivity of the model they appear in to the variables contained in the feature.

These algorithms are compared to SGP in a number of configurations. Although these experiments did not result in a statistically significant difference in model quality between SGP and SensGP, we present a number of ways in which SensGP might be further refined. Further research will have to establish if these adjustments can make SensGP a useful addition to the variety of SR algorithms in the field of GP.

Abbreviations

BP - Behavioural Programming

CART - Classification and Regression Trees

CCD - Cyclical Coordinate Descent

EA - Evolutionary Algorithm

EFS - Evolutionary Feature Synthesis

ERC - Ephemeral Random Constant

FFNN - Feed Forward Neural Network

FFX - Fast Function Extraction

GOMEA - Gene-pool Optimal Mixing Evolutionary Algorithm

GP - Genetic Programming

LASSO - Least Absolute Shrinkage and Selection Operator

IMS - Interleaved Multistart Scheme

ML - Machine Learning

ModelDep - Model Dependent Sensitivity-based Genetic Programming

MRGP - Multiple Regression Genetic Programming

MSE - Mean Squared Error

OLS - Ordinary Least Squares

Recalc - RecalculateScore

SensGP - Sensitivity-based Genetic Programming

SL - Statistical Learning

SR - Symbolic Regression

UDV - Uniform Depth Variation

VarImp - Variable Importance Sensitivity-based Genetic Programming

VM - Virtual Machine

Acknowledgements

I would like to express my gratitude to Dirk Thierens, by whom I was introduced to Evolutionary Algorithms in the first place. His guidance this past year has played an important role in the development of this work.

I wish to additionally offer my special thanks to Marco Virgolin, who has been willing to offer assistance throughout the research. Being able to make use of his code as a basis to expand upon, as well as being able to count on his constructive criticism and advice have been of great value.

Contents

1 Introduction

1.1 Relevance of the Research

Due to the decreasing costs of computer technology, its use has grown exponentially over the past decades. Almost everyone has a mobile phone nowadays and even ordinary household appliances are being connected to the internet, to allow for remote control by their users. This enormous increase of computational power also gives rise to a growth in the amount of data being collected. Our location is available through our phones, whenever we use public transport we have to check-in and even when shopping for groceries, the items which we buy are often registered, so that a store may use this data to improve its marketing strategy. Similarly, to be competitive in industry, processes must be tuned to optimality using all available sensory information. The above-mentioned give rise to a growing interest in predictive data analysis: more data is being collected and to remain competitive this data needs to be used to improve productivity.

1.2 Symbolic Regression

The field of Statistical Learning (SL) provides a number of tools for analysis of large datasets, with which predictive functions can be constructed. Of these, especially regression analysis, the estimation of the relationship among variables, is of interest to us. The focus of this thesis in particular is Symbolic Regression (SR), a type of regression analysis in which a mathematical model is sought which best describes a dataset. Whereas in the standard regression problem, good values of the coefficients within a given model structure have to be determined to find a good fit, in SR this model structure is not given or unknown beforehand.

Symbolic Regression as an object of study resides mainly within the field of Genetic Programming (GP). In this subfield of Evolutionary Algorithms (EA), a population of programs (solutions) is evolved within the solution space of a particular problem, in search of a solution as close to optimal as possible. An optimal program here is defined as a program which achieves the best score possible on a predefined fitness function. In GP, after a population of programs has been evaluated against this fitness criterion, a selection process takes place. From the selected programs of the population of the previous iteration of the algorithm, new solutions are instantiated through mutation, the random alteration of a solution, and crossover, the exchange of subprograms between solutions.

In recent years, the most promising techniques in SR have strayed from the traditional approach in which selection, crossover and mutation by themselves are deemed sufficient to guide the evolutionary algorithm to good solutions. In these new methods, more emphasis is put on the closer examination of the individual solutions, looking at which parts of a program, called features, cause it to perform well. The information about these features is then used in a variety of ways to enhance the traditional evolutionary process.

1.3 Objectives

The main objective of this work is to design and test a new method for SR. The ambition is for this method to improve upon SGP.

In order to accomplish this goal, we conduct research into different methods from the literature, comparing them experimentally and studying which principles behind these techniques might be used effectively to develop novel SR methods.

Using the knowledge obtained from this study, we design three methods, Senstivity-based GP (SensGP), Model Dependent SensGP and Variable Importance SensGP, and compare these on a number of datasets to SGP. This comparison will enable us to test if our objective has been accomplished.

1.4 Outline

In the following sections, we first present an overview of the relevant literature in Section 2, starting with SL and regression, followed by going specifically into SR. Genetic Programming and its connection to SR are discussed, as well as the importance of variable and feature selection. Relevant techniques from the wider domain of Machine Learning will be discussed thereafter. Next, we shed some light on the more recent developments in the field of SGP in Section 3, placing emphasis on techniques in which the features play a central role. Additional attention is directed towards how these features might be ranked and used to guide the evolution.

Thereafter, we will go into more detail about our own research, posing the research questions we set out to answer in Section 4. Details about the implementation are discussed in Section 5. This includes descriptions of the soft- and hardware used as well as descriptions of the datasets used in our experiments. The statistical test we use and how constants are handled are discussed here as well.

The next part of the thesis is focused on experimentation. First, a comparison is made between techniques from the literature in Section 6. Then, experimentation with a new algorithm, named Sensitivity-based GP (SensGP), is presented in Section 7. An overview of the complete results can be found in Section 8 and finally, suggestions for future research are presented in Section 9.

2 Background Theory

In this section, we review the theory relevant to the research described in this thesis. Statistical Learning, the domain in which this research takes place, is described in Section 2.1. Regression is explained briefly in Section 2.2, after which we move on to Symbolic Regression in Section 2.3. Genetic Programming, the method which is primarily used for Symbolic Regression, is discussed in Section 2.4. In Section 2.5 variable selection is expanded upon, as it is of great importance to feature selection. In Section 2.6, three Machine Learning techniques are discussed, which are used by some of the algorithms we examine in Section 3.

2.1 Statistical Learning

Statistical Learning (SL) stems from the field of statistics and consists of a set of tools which can be used to model data. These models serve to further our understanding of the dataset or to enable us to make predictions. With recent advances in computer science, the field has become of great interest. The term SL is often used interchangeably with the term Machine Learning (ML). Although distinctions between the fields can be made based on their underlying assumptions and fields of origin [35], these are not clearly agreed upon by the science community.

In SL, a two different types of learning can be distinguished: Supervised- and Unsupervised Learning. In essence, supervised SL can be seen as the estimation of a true underlying model or function $F(X)$ of the data such that:

$$Y = F(X) + \epsilon, \tag{1}$$

with Y the response or dependent variable, $X = \{x_1, x_2, ..x_n\}$ the input or independent variable(s) and ϵ the error or noise term, which consists of factors not included in $F(X)$ [21].

In Unsupervised Learning, there is no predefined response variable Y, and the goal is not to learn an input-output relationship, but rather to learn an underlying data structure or relationships among variables. In this work, we are only interested in Supervised Learning.

2.2 Regression

As previously stated, a Supervised Learning problem has a response variable Y for which it searches a model based on the problem inputs. When Y is a quantitative variable, i.e. its domain is (ordered) numerical, the problem is referred to as regression problem. If its domain consists of a number of different classes, it is called a classification problem.

Regression first made its appearance in the early 1800s, where the method of (Ordinary) Least Squares (OLS) was discussed in a paper by Legendre and Gauss. This was the first occurrence of what is presently known as linear regression. Over the years, the approach has been extended to techniques such as logistic regression and later on generalized linear models. From there, non-linear approaches started to appear, like regression trees, and with advances in computational power came techniques the field of ML.

In ordinary regression, a predefined model structure is imposed on the function $F(X)$ by assumption. This model consists of terms depending on the input variables, multiplied by parameters β_i. The coefficients in vector β will have to be estimated, which is called training or fitting the model. There exist a number of methods for fitting the model, with the most common one being the aforementioned OLS approach.

In OLS, the Mean Squared Error (MSE) of the model estimate $\hat{f}(X_j)$ is minimized with respect to the coefficients β_i. The MSE is calculated as:

$$MSE = \frac{1}{k}\Sigma_{j=1}^{k}(Y_j - \hat{f}(X_j))^2, \tag{2}$$

where k is the number of data records in the data set, Y_j is the value of Y for the jth data record and X_j are the values for all variables x_i in X for the jth data record. The MSE is used throughout this thesis as a measure of model quality.

2.3 Symbolic Regression

A major drawback of ordinary regression is having to choose the functional form of the model. The model structure is often not known exactly beforehand and thus has to be estimated, which can result in poor model performance if the chosen structure differs significantly from the true problem structure.

This is where SR attempts to make a difference: not only the coefficients are tuned to make a function fit the data, the functional form is also discovered in SR. According to Koza [26] the first application of the technique was around 1960 by multiple researchers, among whom Westervelt [44]. Symbolic Regression has been studied ever since and is presently being used in industry, with applications like Eureqa [11] becoming increasingly popular.

One of the challenges in SR comes from the fact that with unlimited freedom in the construction of a model, any dataset can be perfectly fit. Such a fit can be made by using a polynomial of degree equal to the number of data points minus one and applying the OLS method. While this might seem like a good characteristic, such models will likely suffer from overfitting, meaning a model is tuned to represent the training data to such an extent that it starts losing its predictive value of the true model $F(X)$ in order to fit the training data better. In doing so, the ability of a model to generalize to unseen (test) data diminishes.

In regression, while accuracy is the leading measure in selecting a good model, when two models are comparable in quality, the simpler model is preferred, due to the Occam's Razor principle. This principle states that when there are multiple solutions to a problem, the least complex should be favoured, as it makes the fewest assumptions and can thus be more easily tested.

As mentioned before, by increasing the complexity of a model, its accuracy can be improved upon up to an arbitrarily small distance from a perfect fit on any given dataset. To measure the magnitude of this overfitting, a dataset is usually split into two distinct parts, the training set and the test set. The model is learned on the training set and then tested on the test set. The difference in size of the model error is an indication of the amount of overfitting that occurred.

One way to deal with this complexity versus accuracy dilemma is to employ a technique called Pareto-GP. This technique keeps an archive in addition to the population, in which for each different complexity level defined, the individual

with the best fitness score is stored. The archive is used in combination with the current population to generate a new population, and the archive is updated if any of the newly generated individuals outperforms an archive member of the same complexity level. The output of the program is the entire archive instead of a single solution, allowing the opportunity for an expert to determine which solution provides the optimal complexity-accuracy balance. This method is called Pareto-GP, as the archive can be seen as a Pareto optimal front in the complexity-accuracy space. A more detailed overview of this method can be found in (Vladislavleva) [41].

Symbolic regression is the main focus of this work. We will study the technique from the field of GP (see section 2.4), as most of the SR related research takes place within the field. Emphasis will be on feature extraction (see Section 2.5) and how these features might be used to guide the evolutionary process.

2.4 Genetic Programming

As mentioned, GP can be seen as the home field of SR, with papers on SR outside of the field being scarce (but not absent, see [31]). The reason for this is that the problem was first explored in this context and the tree structure used in GP is very suitable for SR [26]. In this thesis as well, evolutionary techniques are used to study SR, and in the following we first explain the basics of an Evolutionary Algorithm (EA) and then look at GP in relation to SR.

At the start of an EA, a population of solutions to a predefined problem is initialized. This problem is characterized by its fitness function, a function which converts a solution to a fitness score, used to quantify how well a solution performs on the problem. This population of solutions is then evolved together by evolutionary operators, in hopes of improving the population each generation (iteration) of the algorithm.

This generational loop consists of roughly the same steps for every EA [43]:

1. Fitness evaluation of individuals in the population. The solutions are tested according to a predefined fitness function (in SR this is commonly MSE).
2. Selection of individuals for genetic manipulation. This can be done in multiple ways, e.g. tournament selection, ranked-based selection or fitness proportional selection.
3. Genetic operations to produce a new generation: the reproduction operator takes an individual from the parent generation and places it into the new generation unaltered, allowing good individuals to survive in the population. The mutation operator takes a single individual as input and modifies it according to the mutation strategy used. the crossover operator takes two individuals and combines or mixes them in a predefined manner, resulting in one or more new solutions.

These steps are repeated until a termination criterion such as a time limit or a predetermined fitness value threshold is met.

In GP, the solutions in a population are more specifically called programs, and the first successful evolution of programs was presented by Forsyth [13] in 1981, whose work was extended to its current representation by Cramer et al. [8] through the introduction of a tree-like structure to GP. Interestingly, the application discussed here was the evolution of multiplication functions with two inputs and a single output variable, showing a remarkable relation to SR.

In the years to follow, the technique has been further expanded upon. As mentioned by Koza [26], many different kinds of seemingly unrelated problems can be cast into a problem of program discovery, as a program can implement any computable function. In its most basic form, however, all GP algorithms share a couple of properties [4], which we will describe in the following.

A GP program is assembled from two types of primitives:

- Terminals, which are the problem inputs as well as any constants and zero-argument functions with side-effects.
- Functions, including statements and operators with one or more arguments.

The choice of exactly which functions and terminals are allowed in the evolution is crucial and should be made according to the problem at hand, as a solution can only be found within the space spanned by these primitives.

To form an executable program, these primitives have to be assembled into a structure. As stated before, the traditional structure used in GP is a tree structure, due to easy swapping of subtrees, but this is not necessarily true in all applications of GP. The tree nodes can be evaluated as soon as all of their inputs are available, possibly storing the result to prevent unnecessary computation in the future. How the tree is traversed differs per implementation, with common methods being prefix (left-to-right, starting from the root) or postfix (left-to-right, starting at the leaves) order.

Two methods are frequently used to initialize trees in GP:

- Grow: the tree is grown iteratively by addition of a randomly selected nodes, from both the terminal and function primitives. Selection of a terminal node means the end of a tree branch. Function nodes get the number of child nodes equal to the function arity attached to them. Trees are given a fixed maximum depth d and if a branch reaches this depth, a node is randomly chosen from only the terminal set. This method produces trees with branches of different depths, up to a maximum of d.
- Full: the tree is constructed by randomly selecting a node from the function set if the current depth is less than the maximum depth d or from the terminal set if the current depth is equal to d. This method produces branches of predetermined depth d.

Commonly, a combination of the two is used, called Ramped-half-and-half: to promote diversity in the structure of different trees, if the maximum depth is set at d, the population is divided into $d-1$ different parts with depth of 2 up to d, and half of the group is generated using the Grow method, while the other half is generated using the Full method.

The application of GP, as it is described in this section, to the SR problem is what we refer to in the remainder as Standard Genetic Programming (SGP). Standard Genetic Programming is used as a basis of comparison to other algorithms in Sections 6 and 7.

2.5 Variable Selection

An important part of model building in general is the selection of important variables from the space of inputs. When datasets are large, there might be a lot of irrelevant and redundant variables. An irrelevant variable is defined as a variable that has no significant influence on the output variable of interest and a redundant variable is a variable which provides no meaningful additional information in the presence of another specific variable (e.g. a measure of the temperature in both degrees Celsius and Fahrenheit).

Potential benefits of variable selection include obtaining a greater comprehension of the process from which the data originated, increasing predictive properties of the found model, as it uses only the most relevant variables in model construction and reduction in training time of the search algorithm, as the size of the dataset to train on is effectively be reduced. Especially beneficial can be the construction of subsets of variables, which are useful more useful together than by themselves in building good predictors [16].

There are a number of reasons why individual variable performance on the problem can be deceiving, and it can thus be beneficial to select variables together.

Four of these reasons, confirmed by experiments by Guyon et all. [16], are:

- Noise reduction and better separation of classes can be obtained by considering presumably redundant variables.
- Perfectly correlated variables are truly redundant, while highly correlated variables can add information when considered together.
- A variable which provides no additional information about the dependent variable by itself can provide a significant improvement when taken with others.
- Two variables that are do not provide any useful information about the output variable by themselves can be useful together.

These reasons indicate the selection of features can be beneficial over the selection of individual variables. The definition of a feature that is used in this work is: any combination of variables, constants and function operators which results in an evaluable function, including the input variables by themselves (e.g. $x_2^2 - 3.4 \cdot \sqrt{x_1 \cdot x_3}$ is a feature, but so is x_3). The term feature is used interchangeably with subprogram, subexpression and building block.

Variable and feature selection methods can be grouped into three categories [16]: filters, wrappers and embedded methods.

2.5.1 Filters

A filter is a preprocessing step, which has no dependence on the particular model building method used, in contrast to the wrapper and embedded methods. It can be applied to individual variables, where each one is scored according to a chosen scoring function $S(i)$, or to select groups of variables (features) together.

When selection of variables occurs based on their individual merit, the selection method is called a variable ranking method. For each input variable, the score is computed and a high score is assumed to indicate a valuable variable. A cut-off rank or percentage is chosen to determine which variables are considered by the modelling algorithm. Ranking variables in this manner does suffer from the aforementioned issues, however. Alternatively, variables are combined into subsets with their frequency of occurrence in these subsets based on their score, after which further analysis takes place.

Compared to wrappers and embedded methods, filters are relatively cheap computation wise, as computation of just n scores (for n independent variables) is required.

An often used ranking method is the Pearson Correlation Coefficient:

$$\rho_{x_i,Y} = \frac{cov(x_i, Y)}{\sigma_{x_i}\sigma_Y}, \tag{3}$$

where $cov(x_i, Y)$ is the covariance between x_i and Y, σ_{x_i} the standard deviation of x_i and σ_Y the standard deviation of Y. The square of $\rho_{x_i,Y}$ is called the coefficient of determination, which is a measure of the linear relationship between x_i and Y.

Another ranking criteria commonly found in the literature is the mutual information between two variables. The mutual information is defined as:

$$I(i) = \int_{x_i}\int_{y} p(x_i, y) \cdot Log\frac{p(x_i, y)}{p(x_i)p(y)}dxdy. \tag{4}$$

This function depends on the probability densities of x_i and y, which are often not available and difficult to approximate. In case of discrete variables we use instead:

$$I(i) = \Sigma_{x_i}\Sigma_y P(X = x_i, Y = y) \cdot Log\frac{P(X = x_i, Y = y)}{P(X = x_i)p(Y = y)}dxdy. \tag{5}$$

This measures the dependency between the probability densities of both variables. The advantage of the mutual information is that it includes non-linear effects, although it is more computationally expensive to calculate than the correlation coefficient.

When selecting groups of variables together, these methods become more computationally expensive, as each group will have to be scored individually. Therefore, if there are n input variables and groups up to size k are to be evaluated, this requires n^{k-1} times as much computational power compared to scoring the individual variables. The computational cost can be reduced by combining only variables which show individual merit into sets, or alternatively a wrapper or embedded method might be used for testing features containing multiple variables.

2.5.2 Wrappers

The wrapper method views the learning algorithm used to solve the problem at hand as a black-box optimization machine. It feeds the learner data corresponding to subsets of variables, for which the learner returns a solution to the problem. The quality of this solution is then used to score the subset. The approach can be seen as a brute-force method, requiring large amounts of computation, as the learning process is generally very costly.

Smart strategies to determine which subsets to explore can be deployed, such as branch-and-bound, simulated annealing or even genetic algorithms [24]. Greedy strategies can be used, where subsets are built by either forward selection or backward elimination, to prevent the evaluation of too many subsets.

2.5.3 Embedded Methods

Embedded methods incorporate the feature selection into the learning algorithm. This eliminates the need to separate the training set into a training and validation set, as is needed for a wrapper method, and the learner does not need to be retrained for every feature under investigation. Examples of these are decision tree algorithms [29] [5] and regularization methods (see Section 2.6).

2.6 Machine Learning Techniques

As stated in Section 2.1, ML is closely related to SL and the distinction is not always clear. In this section, techniques from ML are discussed which have found a number of applications in SR algorithms as of late. The regularization methods Ridge Regression, LASSO and the Elastic Net are discussed in the following.

2.6.1 Ridge Regression

The first regression method discussed based on regularization is Ridge Regression [18]. As discussed before, often the OLS approach is used to determine model coefficients, whereby the MSE of the model is minimized. This corresponds to determining:

$$\hat{\beta} = \underset{\beta}{Argmin}\{|Y - X\beta|^2\}, \tag{6}$$

where Y are the responses, X is the model matrix consisting of vectors corresponding to observations containing values for each distinct variable and β are the coefficients.

While this gives the most accuracy on the training set, the model will likely not generalize well. To solve this problem, a so called L_2-norm is added to the function:

$$\hat{\beta} = \underset{\beta}{Argmin}\{|Y - X\beta|^2 + \lambda_2|\beta|^2\}, \tag{7}$$

where $|\beta|^2$ is defined as $\Sigma_{j=1}^{n}\beta_j^2$, n is the number of variables and λ_2 is a tuning parameter. The effect of adding such a penalty to the optimization function, is that fitted parameters are encouraged to be relatively small, which reduces overfitting. For $\lambda_2 = 0$, we get the OLS estimate where bias is usually low but variance large, and if we set $\lambda_2 = \infty$ we obtain $\hat{\beta} = 0$. For values of

λ_2 in between, a shrinking of the model coefficients takes place and a trade-off between bias and variance is in effect, where larger λ_2 corresponds to more bias but smaller variance.

A disadvantage of Ridge Regression when use in the context of feature selection is that it does not set any values to zero, thus no selection is performed natively by the algorithm and the result is not very easy to interpret. In the case of identical predictors, each would get a the exact same coefficient, of size equal to the coefficient any of these predictor would have gotten individually, divided by the number of identical predictors.

2.6.2 LASSO

An alternative to Ridge Regression is LASSO: the Least Absolute Shrinkage and Selection Operator [36]. The LASSO method has a lot in common with Ridge Regression, but instead of an L_2-norm, an L_1-norm is added as penalty to the objective function. This results in the following formula:

$$\hat{\beta} = \underset{\beta}{Argmin}\{|Y - X\beta|^2 + \lambda_1|\beta|_1\}, \tag{8}$$

where $|\beta|_1$ is defined as $\Sigma_{j=1}^{n}|\beta_j|$, n is the number of variables and λ_1 is a tuning parameter. The result of using this different norm is that, in contrast to Ridge Regression, coefficients are set to 0 and thus selection is performed, while other coefficients are shrunk in a fashion similar to Ridge Regression.

The reason as to why coefficients are set to exactly 0 is shown in Figure 1: When in a two dimensional space, $|Y - X\beta|^2$ is shown as the elliptical contours, centred on the OLS estimates. In Figure 1(a), the constraint area equal to $\Sigma_{j=1}^{n}|\beta_j|$ is shown, taking the form of a rotated square. In Figure 1(b), the constraint area equal to $\Sigma_{j=1}^{n}\beta_j^2$ can be seen to be circular. The solution provided by LASSO and Ridge Regression is where the contours and the constrained area first touch. In case of the LASSO contour, this may occur at the corner of the area, which is positioned at one of the axes, resulting in a coefficient which is set to 0. In case of the circular shape, this rarely occurs.

Disadvantages of the LASSO, however, are that if $n > k$, with k the number of data records, at most k variables can be selected by the algorithm. Furthermore, if variables have very high pairwise correlations, LASSO likely selects only one, whereas in some applications it is desirable to obtain the entire group. Finally, if $k > n$ and the input variables are highly correlated, Ridge Regression generally has better prediction capabilities [46].

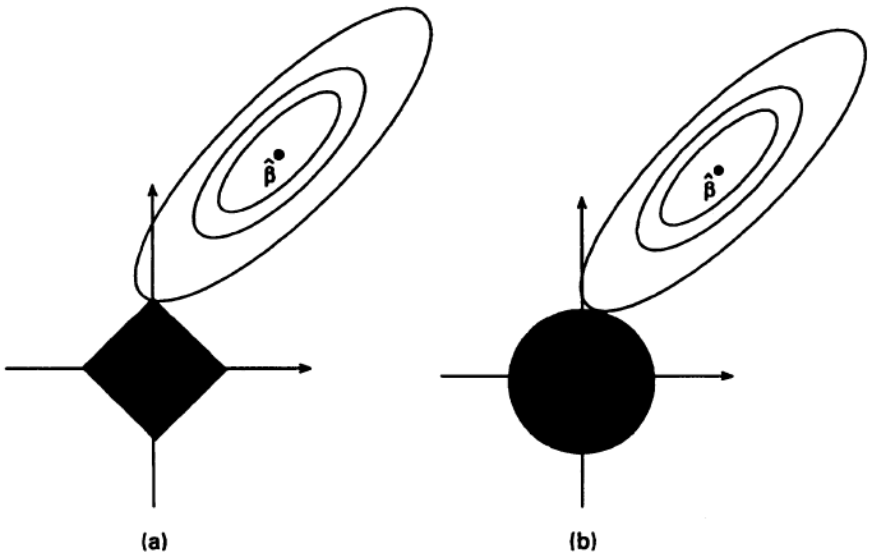


Figure 1: Illustration of the intersection between constraint and OLS centred contour planes, demonstrating the difference in the setting of coefficients between LASSO and Ridge Regression. By Tibshirani [36].

2.6.3 Elastic Net

In 2004, Zou et al. [46] proposed a combination of the former two methods, suspecting it might profit from the advantages of either technique. A naive implementation consists of joining the penalties used before, resulting in the following formula:

$$\hat{\beta} = \underset{\beta}{Argmin}\{|Y - X\beta|^2 + \lambda_2|\beta|^2 + \lambda_1|\beta|_1\}. \tag{9}$$

Alternatively, if we define $\alpha = \lambda_2/(\lambda_1 + \lambda_2)$, this equation can be written as:

$$\hat{\beta} = \underset{\beta}{Argmin}\{|Y - X\beta|^2 + \alpha|\beta|^2 + (1-\alpha)|\beta|_1\}. \tag{10}$$

Empirical evidence has shown the method to work well only when close to either Ridge Regression or LASSO. This is due to an effect called double shrinkage occurring, which does not reduce variance and introduces additional bias. To solve this issue, the estimators are multiplied by a factor of $(1 + \lambda_2)$. This simple scaling improves performance significantly.

2.6.4 Cyclical Coordinate Descent

The regularization techniques discussed in the previous pages are very promising by themselves, but their usefulness really took off after the introduction of the Cyclical Coordinate Descent (CCD) method [14]. The technique is an approach for making these regularization methods more efficient. First, the partial derivative with respect to the coefficient of one of the input variables β_j is calculated, keeping the other coefficients fixed. Then the estimate of β_j is updated by computing the least-squares coefficient on the partial residual due to j, and applying a threshold and scaling to manage the double shrinkage problem mentioned before.

Computing the gradient naively takes $O(k)$ operations, with k the number of data records, by calculating the residuals for every observation. Cycling through all n variables thus takes $O(k \cdot n)$ operations. Improvement on this can be made by storing a matrix of inner products between the variables, and saving computation costs by exploiting this matrix, calculating $O(n)$ new inner products when a coefficient changes. With m non-zero coefficient, a cycle can then be performed in $O(n \cdot m)$ operations. The algorithm then computes solutions for decreasing values of α, starting at the smallest version for which all coefficients are 0. The algorithm converges if a complete cycle does not change the features with non-zero coefficients.

2.6.5 Fast Function Extraction

While most SR research takes place in the GP field, in 2011 the Fast Function Extraction (FFX) method was developed by McConaghy [31]. In contrast to GP, FFX is a deterministic technique, making use of the Elastic Net for its model building. It was found to perform well on a number of different real world datasets.

The algorithmic steps performed by the FFX algorithm are:

1. Enumeration of basis functions.
2. Use of the elastic net to find coefficient values.
3. Application of non-dominated-filter.

These steps are explained in the remainder of this section.

Enumeration of basis functions

To generate a large amount of basis functions to be used as features in the regularization step, a number of nested loops is employed. First, each input variable is raised to each of the preselected exponents (e.g. 0.5, 1.0, 2.0) and afterwards, a number of preselected operators of arity one (e.g. $\text{Log}_{10}, \sqrt{\cdot}$) is applied to form new expressions. These expressions are evaluated intermediately to make sure they return a valid result.

In the following step, these new univariate basis functions are combined into more complex interacting-variable bases, through multiplication. In this step, any operator() $\cdot$ operator() multiplications are disregarded to reduce overall complexity. Again, results are tested intermediately for validity.

Use of the elastic net to find coefficient values

Next, these bases are fed to the regularization algorithm. A log-spaced set of values for α is calculated beforehand, for which to execute the pathwise learning via the Elastic Net. The efficient implementation of Elastic Net with the CCD from Section 2.6.4 is used. In contrast to the usual execution of the Elastic Net, the algorithm halts when a pre-specified number of coefficients have been assigned a non-zero value, as the addition of more variables at this point is deemed to make the model no longer interpretable. This causes a significant reduction in computing time.

Application of non-dominated-filter

In the final step of the algorithm, for each candidate model returned by the algorithm the number of bases in the model and its accuracy are calculated. Only if the accuracy of a model is higher than every other model of the same number of bases, it is added to the output set, in similar fashion to the Pareto-GP method discussed in Section 2.3.

To enhance FFX for improved scalability, first univariate coefficients are learned and only the $k \leq O(\sqrt{n})$ best variables are used to generate higher order features. Due to the manner in which multi-variable bases are generated, this results in a reduction of the computational complexity of the algorithm from $O(k \cdot n^4)$ to $O(k \cdot n^2)$, making it suitable for application to datasets of high dimensionality. Using this technique, problems with up to 1468 input variables are solved in experiments conducted by the author.

3 Extensions of Traditional GP

In recent years, SGP has been improved upon in numerous ways. In this section, some of the most promising improvements, relevant to our research, are discussed. This section is split up into two parts discussing different types of methods: Those which are based on the more traditional approach of GP, where a solution is looked at as a whole (i.e. non feature-based) and the methods in which the feature plays a more central role.

3.1 Non Feature-Based

In this subsection, two improvements upon SGP are discussed: GP with Fast Function Extraction and GP with Gene-Pool Optimal Mixing Evolutionary Algorithm. Both of these only use the overall solution quality in finding new models, disregarding the parts of which they consist:

3.1.1 GP with Fast Function Extraction

Fast Function Extraction has been shown to be an effective technique for SR. As it is a deterministic method, however, it offers little model flexibility. In the construction of bases, only interactions between variables up to the second order are allowed, to prevent a dramatic increase in the number of bases which have to be considered.

Icke et al. [19] combined FFX with GP into a single method in the hope of combating these issues, with their stated goal being to "ease the burden of GP-SR in feature extraction and help it excel in model building". Their experiments showed the technique to be an improvement upon both of the methods separately.

The implementation of GP-FFX consists of the following steps:

1. Feature construction by using a variant of FFX, in which function bases are generated.
2. Model construction with the fast Elastic Net implementation using CCD.
3. Extraction of the unique bases from the non-dominated models generated in the previous step.
4. Model building by application of SGP, easing the initial burden of SGP by providing a good initial direction in which the search may be expanded.

The technique showed significantly better results than ordinary SGP on datasets of high dimensionality and was able to handle discovery of the functional form of more functions than FFX, since the latter algorithm is restricted when it comes to higher order interactions.

3.1.2 GP with Gene-Pool Optimal Mixing Evolutionary Algorithm

A second approach very recently explored by Virgolin et al. [40] is GP with the Gene-Pool Optimal Mixing Evolutionary Algorithm (GP-GOMEA). What makes GP-GOMEA interesting is that it produces smaller, and thus more interpretable solutions than most SR algorithms do. Additionally, it is a model-based EA,

meaning it builds a model which attempts to capture the structure of good individuals.

In GP-GOMEA, this model is used to guide the creation of new individuals. The model used is a linkage model, more specifically the family-of-subsets model (FOS). The model consists of a set of sets of locations (loci) within the program genotype. The variables within such a set are to be varied together, since the model has learned they are related. A number of different implementations of this linkage tree can be imagined, and in this paper the authors chose to use the Univariate method, where only a single position of the genotype can be varied, the Linkage Tree method, where single loci are merged into sets in a bottom-up fashion based on mutual information, and the Random Tree method, where these subsets are constructed using randomly generated mutual information.

The linkage model is used in combination with the GOM operator, which performs crossover between a parent individual and a randomly selected donor individual of the population. The genes (nodes) corresponding to the loci contained in the linkage model are copied from the donor to the parent and the fitness of the newly created individual is measured, accepting a new individual only if its fitness improves upon the fitness of the parent individual. If no new best fitness has been found in a predetermined number of rounds through application of this greedy method, the best individual so far is used to mix with.

A recent work to appear at GECCO'18 [39], suggests GP-GOMEA can be effectively used to learn small yet accurate expressions for SR, however, the linear scaling method proposed by Keijzer [22] is needed to make small expressions achieve good performance.

An additional technique presented in the original GP-GOMEA paper is the Interleaved Multistart Scheme (IMS). In IMS, multiple GP runs with different population- and genotype sizes are interwoven, in order to automate the process of searching for good values for these parameters. A generational step size g is specified, and after having run g generations with one set of parameters, another run with double the population size performs a generation. Every two runs, the maximum allowed tree depth is increased by one.

An IMS run R can be can be terminated before the specified termination criteria (e.g. elapsed time or number of evaluations) is met. This occurs when:

- A new overall best solution is found which has a larger tee depth than the individuals of run R have.
- The individuals in the population of run R have converged to a single genotype.
- Another run with a larger population size than run R has obtained a better average fitness than run R, or than a second run R^*, given that R^* has a larger population size than R.

When all runs have terminated, the algorithm halts and returns the best solution found.

3.2 Feature-Based

Most of the promising techniques appearing in the SR field as of late seem to have a more feature-centred approach to evolution. Instead of the individual, its parts are the central units of interest. Whereas standard GP aims to identify good building blocks by assuming that their presence is reflected in the fitness of the individual they appear in, causing them to become more prevalent in the population as more generations pass, these new techniques guide the evolutionary process by looking at the components that make up these individuals more explicitly. Four such feature-guided methods are discussed in this subsection.

3.2.1 Multiple Regression

We start by discussing one of the most commonly known algorithms in SR, which is considered to be state-of-the-art. Multiple Regression Genetic Programming (MRGP) [1] takes a population of models, breaks them down into building blocks and performs a form of regression called Least Angular Regression [12] to re-instantiate a model using the value these building blocks have when applied to the data. This causes the selection pressure to be put explicitly on the subexpressions (features), in contrast with regular GP, where features are preserved indirectly through the fitness of the model they are part of. In this approach, however, no explicit selection is performed on the building blocks. A model has its subexpressions decoupled and through regression linearly combined, with optimal coefficients attached to the expressions.

Two variants of the algorithm are proposed in the paper. The first is a post-run approach, in which MRGP is applied to the best solution found at the end of a run. The authors experiment with different definitions of what is seen as a subexpression or feature in this algorithm. Five definitions for what a subexpression is, are adopted and tested:

- Root node
- Root node and leaves
- All tree nodes
- Root node and input variables
- All tree nodes and input variables

The authors found that when using any of the above strategies with post-run MRGP, this generally resulted in models of higher accuracy than a competent GP or ordinary multiple regression algorithm would. The latter two strategies performed especially well.

The second variant of MRGP carries the name inline MRGP. This approach uses multiple regression within the evolutionary cycle, explicitly guiding the evolutionary process. By measuring the usefulness of a program not by its overall MSE, but by the MSE of a the combination of its subexpressions after regression, there is more room for optimization of these subexpressions. Selecting solely on the basis of post-regression fitness is unlikely to be a good idea, however, as larger models have more subexpressions which can be tuned and will therefore

likely result in a better regressed model than a model which is smaller in size and is therefore less tunable, while the smaller model might contain more useful building blocks. In order to prevent this from happening, a multi-objective usefulness function is applied, with as a second objective the model complexity.

To this end, four different complexity measures are employed:

- Tree complexity, which is equal to the number of nodes in the program.
- Sum of t-statistics, which sums the t-statistic corresponding to each coefficient of the expressions within in a model, which is a reflection of how significant the expression is within the model. The sum of these is defined as the complexity measure, penalizing models which use a larger number of significant predictors.
- Minimum Description Length (MDL), measured on the model its parameters. This measure implicitly includes the t-statistic and therefore measures the same effect, which is even enlarged by this measure.
- Saturated Minimum Description Length (SMDL), in which the t-statistic values are bounded, ensuring very useful predictors with a high t-value are penalized less severely.

Experiments show the Sum of t-statistics and Tree complexity measures to be the most valuable, while the MDL and especially SMDL approaches can be lacking. Overall, the inline variant of the algorithm outperforms the post-run variant, showing the promise of feature-guided evolution.

3.2.2 Behavioural Programming

A second approach which adopts the philosophy that the fitness measure of an individual alone provides insufficient information for efficient traversal of the search space is Behavioural Programming [27]. The authors propose to measure intermediate program behaviour, i.e. performance of partial solutions, and use the result to increase the search efficiency.

In GP in particular, the authors argue, solutions can be readily inspected and subprograms evaluated due to the tree structure used. Additionally, there is no reason to rely solely on a task-oriented fitness function other than that Evolutionary Algorithms are inspired by Darwinism and thus this practise has slowly become commonplace in the field. Using additional information as a search driver could potentially be more informative and consequently result in a more efficient evolution.

In comparison to SGP, three changes are implemented:

- **Behavioural evaluation:** When a given program is applied to an input, all intermediate evaluations, corresponding to nodes in the GP tree being applied to the data, are stored in a list, forming a trace. This trace will be of equal length on every data record for a given program. Such a trace is calculated for all data records in the training set, forming a trace table. Subsequently, a machine learning algorithm is applied to the table, treating every evaluation value as a feature. This results in a model p of the data,

i.e. a behavioural description of the intermediate program evaluation. in the construction of this model, an error measure $e(p)$ and a complexity measure $c(p)$ are calculated.

- **Archiving of useful subprograms:** In order to make optimal use of the information extracted in the form of a behavioural model, the subprograms corresponding to the good features chosen by the machine learning algorithm have to be easily retrievable from the representation. These features are then stored in a global archive, implemented as a priority queue of fixed length. The priority of a feature p' from model p is set to be inversely proportionate with the model error and complexity of p, which are combined into a measure called utility:

$$u(p') = \frac{1}{1 + e(p) \cdot c(p)}. \tag{11}$$

  When the maximum capacity of the queue is reached, the queue is reset and repopulated based on the utility of the subprograms.

- **Use of an archive-based mutation operator:** To be able to exploit these presumably useful subprograms, the authors designed a mutation operator. First, a parent program is selected to be modified. One of its nodes is randomly selected to be replaced together with its associated subtree, by a feature from the archive. Subprogram selection takes place in proportion to the utility of a program. This operator is applied in conjunction with conventional mutation as otherwise no new subprograms could be introduced into the population.

To ensure the probability of a mutation or crossover action happening at a certain depth is not larger at greater depths due to the number of nodes growing with depth, a technique the authors dub uniform node selector is used. This technique calculates tree depth d and then selects a depth $\hat{d}$ from which to select nodes uniformly from the range [0,d]. Subsequently, a random node is selected from the chosen depth $\hat{d}$.

The approach was tested experimentally, on a test set in which no constants were present. In this environment, results show the approach generally outperforms SGP on the studied benchmarks, as well as producing somewhat better models in terms of model complexity and generalization properties. From their results the authors conclude again that the objective function we want to optimize does not necessarily correlate with the amount of computation power required to solve the problem, and using other objectives to assist the search could increase efficiency.

3.2.3 Evolutionary Feature Synthesis

The next method we highlight is Evolutionary Feature Synthesis (EFS) [2]. The authors observed the success of the MRGP, BP and a technique named Kaizen Programming [9] and were inspired to create a method using some of the principles from these papers. As the name suggests, the feature is once more the centrepiece of the evolutionary process.

In contrast to previous methods, the feature is the unit of selection. As in BP, overall model quality is not the main guiding principle of the evolutionary search. A populationin EFS consists of features instead of models and by use of the LASSO, a model can be generated from the population. The efficient CCD implementation of the LASSO from Section 2.6.3 is used.

The approach involves three steps which are performed in a cycle:

1. **Model Generation:** Using the LASSO, a model is build from the features currently in the population. If overall model fitness, calculated by the R^2 measure, is better than the best fitness found so far, the model is archived.

2. **Feature Composition:** New features are generated by using the pool of existing features and applying an operator from a predetermined set of operators to them. Depending on the arity of the operator, one or two features are selected to form a new feature. The selection of these featured is done by tournament selection based on their previously computed feature score. The pool of features will always be set to include the p original parameters of the problem, as well as q additional features formed in previous iterations. From these, μ new features will be constructed and added to the population, resulting in a population of size $p + q + \mu = M$. The process is illustrated in Figure 2.

   To limit the overall complexity of the features and the model which is generated from them, feature size is constrained. The size of a feature is defined to be the number of original problem parameters occurring in the feature, added to the number of operators present in the feature. In this work, the maximum feature size is set to 5.

   A significant improvement in computation time is achieved by this method compared to the SGP, as a feature can be individually evaluated against the dataset, whereas in traditional GP, whenever a subtree changes, the values of all nodes in the entire tree need to be recalculated (or part of them, if values of all nodes are stored). Additionally, these feature computations are easy to execute in parallel.

3. **Feature Subset Selection:** The LASSO is applied once more, generating useful information which can be used for feature selection.

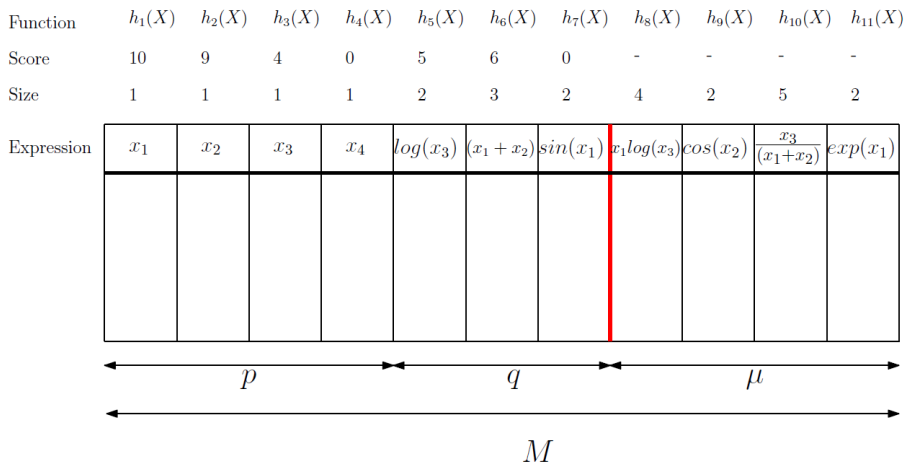


Figure 2: The feature composition process for EFS, with p the original variables of the problem, $p+q$ the current features in the population and μ the newly constructed features. By Arnaldo [2].

Three different approaches for model selection are tested by the authors, each aiming to remedy flaws found in the previous approach during experimentation.

The first approach used by the authors allows for a flexible model size. Using the LASSO, a set of weight vectors β^α is obtained, for a number of different α. It is important to node here that α equals λ_1, the tuning parameter in the LASSO algorithm (see Section 2.6.2). As an additional benefit of the LASSO, the coefficient of multiple correlation R^2, which is used to assess model quality for ranking features in this work, can be calculated without incurring additional computation cost. The weights corresponding to the value of α that maximizes the R^2 are chosen. Then selection is performed as follows: The p original parameters are kept, as well as any feature which has obtained a non-zero weight. A disadvantage of this method is that there is a possibility that all coefficients will be non-zero and thus no selection is made at all, causing the evolution to halt prematurely.

To combat this flaw, a second method is devised in which the model size is fixed, such that in every cycle exactly $p+q$ features are selected instead of a varying amount. This corresponds to the situation shown in Figure 2. To enable this exact selection, a complete ranking of all features is made, using the following feature importance measure:

$$Importance(j) = \sum_{\alpha_i \in \Gamma} Score(j, \beta^{\alpha_i}). \tag{12}$$

$$Score(j, \beta^{\alpha_i}) = \begin{cases} R^2_{\alpha_i} \ if \ \beta^{\alpha_i}_j \neq 0 \\ 0 \ otherwise \end{cases}. \tag{13}$$

Here Γ is the set of preselected α values traversed by the LASSO. During selection, the p original variables as well as the q variables of highest rank are moved

to the next generation.

Experiments using this new strategy highlighted yet another issue. Model error is not systematically reduced in each step. The authors explain this behaviour as being the result of highly correlated features sharing importance. This may cause two very similar, but individually important features to both be discarded in the selection process because their shared importance is smaller than the q features of highest rank, while if only one of the features had occurred in the model it would have obtained a sufficiently high score. This can result in a good feature disappearing from the population and the MSE of the fitted model increasing.

To remedy this flaw, the authors introduce correlation filtering, using the Pearson Correlation Coefficient from Equation 3. Performing a complete analysis of all M features in the population would require M^2 passes over the data, which the authors deem too expensive. Therefore, they propose to only calculate the correlation coefficient of a feature with its parent (as these are the most likely to be highly correlated), disregarding a newly created feature if the coefficient is higher than a predefined threshold (taken to be 0.95 here). This method decreases the overall variance of the method and therefore increases robustness.

EFS is compared experimentally to MRGP, LASSO, a Feed Forward Neural Network (FFNN), Multiple Regression and a method called Vowpal Wabbit. It is found that EFS and the FFNN produce models with the lowest error on average. In terms of speed, EFS and MRGP were the worst methods, however, with the FFNN much faster. In terms of complexity, EFS performed better than the FFNN, the latter producing models which are hardly interpretable.

3.2.4 Embedded Feature Construction

The final feature-based approach discussed is GP with Embedded Feature Construction (GPEFC) [6]. An overview of the method can be seen in Figure 3.

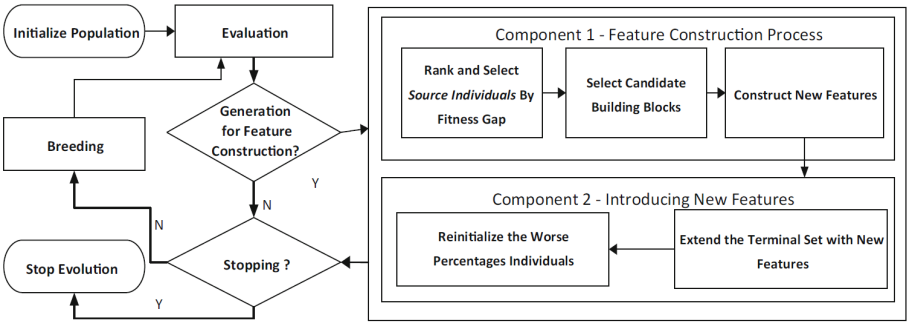


Figure 3: An overview of the GPEFC method. By Chen [6].

Standard GP and GPEFC differ mainly in two ways. GPEFC has a unique way of identifying potentially valuable subprograms and of introducing these into the population:

- **Identification and construction:** The authors base their method on the idea that an individual which has a large fitness increase with respect to its parent(s) is likely to contain new, useful subprograms. Therefore, an individual j with parents p_1 and p_2 is scored by its FitnessGain as:

$$FitnessGain(j) = minimum(Fitness(p_1), Fitness(p_2)) - Fitness(j), \tag{14}$$

  where a lower fitness is defined here to be a better fit (in contrast to the usual definition). A predefined percentage of individuals with the highest fitness gain is selected and these are dubbed source individuals. From these source individuals, good features are to be selected, taking into account two measures: The depth or equivalently number of levels the feature consists of, and the activeness, which is defined as the number of times a feature appears in the pool of source individuals. To keep the complexity of the features within acceptable bounds, while retaining their usefulness, the maximum depth is set to two or three. Whenever a building block has an activeness of more than half the number of source individuals, it is deemed important.

  From each of the important building blocks, exactly one new feature is constructed in an unspecified manner. These newly constructed featured are added to the terminal set.

- **Introduction into the population:** As mentioned, the new features are used to extend the terminal set. The extended set is then used in two ways. Firstly, new individuals are constructed from scratch using only the new terminal set, increasing the probability of newly discovered individuals making it into the population. Secondly, through mutation entire subtrees of existing individuals in the population are replaced by new subprograms from the extended terminal set.

To calculate the fitness, the Normalised Root Mean Squared Error is used. It is defined as:

$$NRMSE = \sqrt{k/(k-1) \cdot MSE/\delta_t}, \tag{15}$$

where k is the number of exemplars in the dataset and δ_t the standard deviation of the target outputs.

It is found the method produces programs of smaller size than SGP on most of the training sets used. It produces more distinguished features, although the increase is almost negligible. In term of goodness-of-fit, GPEFC has smaller errors on all problems studied than SGP. This extends to the test sets as well, indicating models generalize better. In terms efficiency, differences can be observed based on the dataset used, where if a set can be explained by more compact features, GPEFC might be more efficient, while losing to standard GP if more complicated features are required. A statistical significance test shows GPEFC performs statistically equal or better to standard GP in all cases.

3.3 Feature Importance

From the experiments described in the previous two subsections, we observe that the novel methods, which focus on a more feature-guided approach to GP have great potential. The overall fitness function is found to often provide too little information to accurately guide the search in the direction of good solutions if the fitness landscape is not straightforward.

One more technique which has not seen much application within GP, but is commonly used in other fields and has potential to be used in a feature-centric approach is sensitivity analysis. In the paper Sensitivity-Like Analysis for Feature Selection in GP [10], this approach is applied in the field of SR, but limited to the input features.

3.3.1 Sensitivity-Like Analysis for Feature Selection in GP

The author suggests frequency based approaches for feature selection might be lacking and sets out to put this to the test. He proposes to use a sensitivity-like importance measure for the selection of good features. Through a comparison, he tests his hypothesis and integrates the sensitivity-like importance measure into the evolutionary process.

To research if frequency does indeed provide a meaningful criterion for feature selection in GP, two methods are used to measure the presence of the input variables in the final models of a GP run. The first is based purely on the frequency, counting how many of the models in the final population of a GP run contain a certain variable. A second measure is the proportional feature use, which counts the number of total terminals used in a model and uses it to scale the the feature presence with. These two frequency based measures are compared to the variable importance measure of two well known algorithms: Random Forests [5] and Classification and Regression Trees (CART) [29].

Additionally, a new variable importance measure which can be applied to GP is introduced. This sensitivity-like measure is calculated as follows:

1. At then end of a run, the model producing the best fit on the test data is returned.
2. The model is applied to a hold-out partition of the data, and its MSE is calculated.
3. For each variable occurring in the model, the test data is shuffled in turn and the MSE of the model on the shuffled data is computed, resulting in a measure of how sensitive the model is to the given variable.
4. The changes in MSE are normalized against the largest occurring value, resulting in an importance value in the range [-1,1] for all variables.

After the execution of 100 runs of the GP, Random Forest and CART algorithms, the different variable importance measures were applied to the resulting populations of the GP runs and compared to each other as well as to a variable importance analysis of the true underlying distributions of these problems. From the results it is clear that the frequency measure does not provide very valuable information about which variables are important. In almost all cases, all of the features had a significant presence in the final model. The proportional measure

turned out to be not much better than a scaled version of the presence measure and did not provide more insight into the worth of the variables. The novel variable importance measure introduced, as well as the CART and Random Tree methods performed much better, showing similar results for variable importance to each other and the true distributions. From these results the conclusion is drawn that a presence measure is not a good indication of variable importance and the presence of a variable can possibly be attributed factors like bloat or the use of variables to generate constants by self-division.

After reaching this conclusion the author sought to include the variable importance measure into GP, to which end he proposes two extensions to regular GP:

- Implementing bloat control by using the solution size as a secondary selection factor.
- Adapting the probabilities of terminal selection throughout the run. At first all terminal probabilities are equal. After every run the variable importance is recalculated on the best performing individual in the population, using the variable importance analysis explained above. Variables are ranked on their importance, and given a probability of being used in the next generation proportional to their rank.

It was found that especially the latter method of integrating the variable importance measure into the algorithms worked well, as did a combination of both methods. Measuring the frequency of building blocks after the run with the sensitivity-like approach resulted in variable presence which did not differ significantly from before, however, indicating once more that frequency might not be suitable for identification of good features.

3.4 Recap of Techniques Discussed

As quite a few techniques are discussed, we give a succinct summary of the core principles behind each technique:

- **GP & Fast Function Extraction [19]:** Initial population construction by enumeration of basis functions, after which an elastic net fit is made as in FFX. Non-dominated models are selected and fed to SGP.
- **GP Gene-Pool Optimal Mixing Algorithm [40]:** Population of models, model structure is learned by a number of different methods to construct a family-of-subsets model. Corresponding to the subsets in this model, locations of the genotype are varied together, accepting improvements greedily.
- **Multiple Regression GP [1]:** Population of models, taken apart into subprograms according to a number of different strategies. Regression is applied to these subprograms to evolve new models, which are scored for selection by a combination of model error and one of four complexity measures.
- **Behavioural Programming [27]:** Population of models, evaluated on the dataset and the partial results for each subexpression are stored in a trace table. This trace table is fed to an ML algorithm. The features picked by the ML algorithm are archived and later introduced into the population by an archive-based mutation operator.
- **Evolutionary Feature Synthesis [2]:** Population of features, used to form a model together via LASSO. The contribution of a feature is measured as the sum of all R^2 values of the models generated by LASSO in which the feature has been given a non-zero coefficient. The features in the population are ranked and a select number advance into the new generation, after which they undergo mutation to form new features.
- **GP Embedded Feature Construction [6]:** Population of models, good features are identified by measuring the fitness gain of an individual compared to its parent(s). A percentage of the fittest models is used to obtain features from, by analysing in how many of these fittest models these features occur. The more frequent of these features are mutated and added to the terminal set, which is used both for mutation and to construct entirely new solutions from.
- **Sensitivity-Like Analysis [10]:** Population of models, after each generation a sensitivity-like analysis is performed on each input variable of the best model in the population by shuffling the training data corresponding to that variable and calculating how sensitive the model MSE is to this change. The variables with the highest sensitivity obtain a larger probability of being selected for mutation in the next generation.

These techniques form the bases of the theoretical comparison described in Section 6 and experimentation with a new algorithm in Section 7.

4 Problem Description

In this work, we set out to find an improvement of SGP for solving SR problems. To achieve this goal, we first compare a number of techniques from the literature, with the purpose of learning what aspects of these algorithms might be effectively used in the design of a novel SR method.

To this extent, we compare the feature-based algorithms EFS, BP and MRGP with SGP and GP-GOMEA. Multiple configurations were used for SGP, as well as for GP-GOMEA. Since these algorithms are implemented in different programming languages, additional effort is made to estimate the impact of this difference, based on the number of subexpression evaluations on the dataset. This study and its results are presented in Section 6.

After having conducted this comparison, we designed an algorithm ourselves, with hopes of improving upon SGP. Based on the review of the relevant literature and the results of the comparative study, it appeared a feature-centered guided evolution was likely to be a promising direction for improvement. More specifically, we chose to experiment with the application of sensitivity analysis to calculate a feature imortance score, which is then used to guide the evolutionary process. This is a novel approach as far as we are aware, based on the input variable ranking method presented by Grant Dick [10]. For the particulars of this experiment, see Section 7.

4.1 Research Questions

The project has two clear goals in mind: the exploration of the feature-based GP-SR paradigm and its possible improvement through sensitivity-based feature guided evolution. To this end, we first seek to answer the following research questions:

- How do EFS, BP, MRGP and GP-GOMEA perform in comparison to SGP and each other?

And:

- What insights can be obtained from the comparison of these methods which might prove useful for application in further research into improving SGP?

The former question will be answered through a comparison of the named algorithms. To be able to answer the question, we must define what kind of performance we are interested in. This leads to the sub-questions of which technique will find the best fitting models on the training data and whether these models generalize well to the test data (i.e. do not overfit on the training data).

The latter question is a somewhat subjective one, but very important nevertheless. Many answers are possible, and we will only attempt to discover general principles which to use in accomplishing our main goal of improving upon SGP with a novel method. An answer will be sought by examining the results obtained in answering the first question as well as using the information from the literature study.

After having answered these first two questions, we arrive at the arguably more interesting part of the research, where we look for improvement upon SGP. Any insights gained through the experimental comparison of the existing techniques will be used to guide us in the design of a Sensitivity-based GP algorithm.

The remainder of the thesis is then spent finding an answer to the final and central question of this work:

- Can SGP be improved upon by introducing a sensitivity analysis based feature-guided evolution technique, called Sensitivity-based Genetic Programming (SensGP)?

To answer this question, we experiment with different algorithms incorporating a sensitivity-based importance measure, introduced in Section 7.1. In addition to the basic SensGP algorithm, which is explored in Section 7.2, two variants to SensGP are researched as well:

1. Model Dependent SensGP (ModelDep): only a pre-specified percentage of the fittest individuals in the population is used to determine feature importance. This approach is examined in Section 7.3.
2. Variable Importance SensGP (VarImp): features are scored based on how sensitive the model they appear in is to the variables contained in that feature. Results can be found in Section 7.4.

In Section 7.5 these methods are compared to each other and to SGP, which allows us to formulate an answer to the final research question.

5 Implementation

In this section, we describe the details of the experimental setup used in answering the research questions. First, we describe the computer programs used for both the comparison of the algorithms and the experimentation with SensGP, in section 5.1. Next, we provide a description of the hardware that was used to perform computational experiments in Section 5.2. Thereafter, the datasets used in this work are examined in Section 5.3. In Section 5.4 we describe the statistical test we used to examine the results. Finally, in Section 5.5 we discuss how constants are included into the algorithms.

5.1 Software

For our experiments, we used 4 different programs. The first three are the implementations of EFS, BP and MRGP. These algorithms are all part of the FlexGP project by the Anyscale Learning For All (ALFA) group at the Computer Science and Artificial Intelligence Laboratory (CSAIL) at MIT. The project has "scalable machine learning using genetic programming" as its goal.

The code for these algorithms has been made publically available on the FlexGP website [3]. All three of these programs are written in the `Java` programming language. To interact with it, NetBeans IDE version 8.2 was used.

The fourth program, which we used for SGP and GP-GOMEA and which was used to implement SensGP, was written by Marco Virgolin for his GP-GOMEA research [40]. The choice for this implementation of SGP was made as it is written in `C++`, a language we are comfortable with, as well as that we were able to count on the support of the author. To interact with the code, Microsoft Visual Studio 2015 Community Edition was used.

From the original code, we created a minimal version in which only basic GP-GOMEA and SGP were present, and from there started all further experimentation with SensGP. Using this minimal framework saved a considerable amount of time compared to having to implement a SGP experimentation setup from the ground up. Furthermore, the fact that the code of SensGP is written in the SGP and GP-GOMEA framework allows for the fairest comparison possible, as any part of the execution of the algorithms which is shared, is implemented by the exact same code.

5.2 Hardware

Two computer setups were used to perform the experiments needed for evaluation of the algorithms of interest in this work. An Asus laptop was used to perform all of the comparisons amongst different techniques from the literature. For experiments with the Sensitivity-based technique, a combination of this laptop and 4 Virtual Machines (VMs) operating on the Microsoft Azure Platform [33] was used. The experiments executed on different setups are never compared to each other, to ensure no algorithms is given an unfair advantage due to the different processing speeds of these machines. The details of these machines are presented in the remainder of this subsection.

Asus Laptop

For most of the experiments, an Asus laptop was used. The specifications of this laptop are shown in Table 1:

Table 1: Specifications of Asus Laptop.

Specification	Value
Type	Asus N56V
CPU	Intel Core i7 3630QM 2.4GHz
Cores	4
Memory	6 GB

Microsoft Azure

Microsoft Azure is a cloud computing platform created by Microsoft. With this service, multiple virtual machines can be used for computation (up to a maximum of 4 virtual cores with a student subscription). The specifications of the virtual machines are shown in Table 2:

Table 2: Specifications of Microsoft Azure Virtual Machines.

Specification	Value
Type	Virtual CPU Standard DS1 version 2
CPU	Max IOP's 3200
Cores	1
Memory	3.5 GB

No further specifications on these virtual machines are supplied by Microsoft, but from experimental results we conclude that a virtual CPU from the Microsoft Azure platform was slightly outperformed by the Asus laptop.

5.3 Description of Datasets

In the field of Genetic Programming, benchmarks have historically been quite poor [32]. They have remained in the field through tradition, not because of their usefulness in providing challenging, realistic testing grounds for GP algorithms. After the aforementioned paper on benchmark performance was published, steps were taken by consulting the GP community, resulting in a work which proposed a number of standard problems as alternatives to some of the worst datasets [45]. These are deemed good alternatives and "first appeared in papers with large numbers of citations, suggesting that they are well-known" [45]. All problems looked at in this work are taken from the proposed SR benchmarks, with the exception of the of the Keijzer-1 dataset, as we had already begun testing on it before we discovered the existence of this list.

In our experiments, we use two smaller datasets, Keijzer-1 [22] [23] and Nguyen-7 [38], as well as six larger datasets of which Vladislavleva-4 [42], Pagie-1 [34] and Korns-12 [25] are artificial datasets constructed from a formula, Energy Heating [37] [28] is a dataset obtained from computer simulation results and Dow Chemical [45] and Red Wine [7] [28] are real world datasets. All of these datasets are described in some detail in the remainder of this section:

Keijzer-1

The Keijzer-1 dataset is a small dataset, consisting of only 20 data records, with 1 independent variable x. The values of the independent variable are from the range [-1,1], with corresponding values for the dependent variable in the range [-0.25,0.1]. The dataset was popularized in Keijzer [22], where a list of datasets is presented, commonly known as the Keijzer set. Originally, the function appeared in a paper he co-wrote with Babovic [23].

The data is generated from the following formula:

$$f(x) = 0.3 \cdot x \cdot sin(2\pi x). \tag{16}$$

Nguyen-7

Nguyen-7 is the second small dataset we use for our experiments. It features only 20 data records in both train and test sets and a single independent variable. The value of the independent variable is drawn from the uniform distribution in the range [0,2], resulting in values of the dependent variable in the range [0,3].

The data is generated from the following formula:

$$f(x) = Ln(x + 1) + Ln(x^2 + 1). \tag{17}$$

Vladislavleva-4

The first of the larger datasets we use is Vladislavleva-4. The dataset consists of 1024 data records in the training set, and 5000 records in the test set. There are 5 independent variables present in this dataset, but the formula from which the data is generated can be straightforwardly expanded to include any number of variables. The values of the independent variables are drawn from the uniform distribution with the range [0.05,6.05] in the training set and [-0.25,6.35] in the test set, with corresponding values for the dependent variables rarely outside the range [0, 1]. As described in [45],"the problems require extrapolation, not just interpolation" and the algorithm used by Vladislavleva "appears to have most difficulties in discovering the simple and harmonious input–output relationship" on this problem.

The data is generated from the following formula:

$$f(x_1, ..., x_5) = \frac{10}{5 + \Sigma_{i=1}^{5}(x_i - 3)^2}. \tag{18}$$

Pagie-1

The Pagie-1 dataset is used in comparing SensGP and its variants to SGP. The set consists of 625 data records in both training and test sets. Two independent variables are present in the dataset. In the training set, the values of these independent variables are evenly spaced points on a grid with range [-5,5], positioned at intervals of 0.4. In the test set the values are selected from the uniform distribution from [-5,5] for both variables. The corresponding values for the dependent variable are generally found in the range [1,2], with a couple of exceptions falling in the range [0,1]. While Pagie-1 is a smooth problem, it is known to present considerable difficulty [17] [34].

The data is generated from the following formula:

$$f(x_1, x_2) = \frac{1}{1 + x_1^{-4}} + \frac{1}{1 + x_2^{-4}}. \tag{19}$$

Korns-12:

The final artificially constructed dataset we look at is Korns-12. It is an order of magnitude larger than the other large datasets we use, with 10.000 data records in both the train and test sets. The dataset consists of 5 independent variables, of which only 2 have an effect on the dependent variable. The underlying reason for including the three additional independent variables is to "test the ability of the system to discard unimportant variables and avoid using them to over-fit" [45]. The values for all of these variables are drawn from the uniform distribution with the range [-50,50], resulting in typical values for the dependent variable in the range [0,4.5].

The data is generated from the following formula:

$$f(x_1, ..., x_5) = 2 - 2.1 \cdot cos(9.8 \cdot x_1) \cdot sin(1.3 \cdot x_4). \tag{20}$$

Energy Heating:

The Energy Heating dataset [37] describes the heating efficiency of buildings with distinct shapes. These different shapes and coressponding heating efficiency scores have been simulated in Ecotect, an environmental analysis tool. The dataset is comprised of 8 independent variables and 768 data records. The dependent variable is the heating load, and its values are contained in the range [0,50]. It is available on the UCI Machine Learning Repository website [28].

Dow Chemical:

Dow Chemical is a real world dataset, consisting of 747 training data records and 319 test records. The dataset features 57 distinct variables, many more than each of the other sets we tested. Data ranges vary widely for each variable, as can be expected for real world data. Typical values for the dependent variable are found within the range [2,4.5]. The data, originating from a chemical process plant, was the subject of the 2010 EvoStar SR competition [45].

Red Wine:

The Red Wine dataset contains information about red wines from Portugal. In this dataset, the dependent variable is a grade for the quality of the wine, based on expert ratings. This grade is given in in the range [0,10]. The data was used in Cortez et al. [7] and consists of 11 independent variables conveying physiochemical information about the wines. The set consists of 1599 data records. It is available on the UCI Machine Learning Repository website [28].

5.4 Statistical Test

To be able to quantitatively compare results when a qualitative analysis is insufficient, the Mann-Whitney U test is used in this work [30].

The reason for choosing this test, and not the students t-test, is that it is nonparametric, i.e. it makes fewer assumptions on the underlying distribution of the compared datasets. In GP, the quality of the best model found in different runs varies substantially depending on the initial population. This is also the reason as to why it is common in GP to report the model quality found in the median run of an experiment, instead of reporting the mean model quality. Assuming the results to be normally distributed, which is a requirement for many tests, would be unjustified.

In the Mann-Whitney U test, a probability value (p-value) is calculated, which tests the null hypothesis that the difference between true medians of the two datasets equals 0, versus the alternative hypothesis that this difference is unequal to 0. A small p-value indicates the null hypothesis is unlikely to be true, while a large value indicates the two true medians are equal. In calculating this p-value, the values of the datasets are grouped together and ranked based on their solution quality. Then, they are returned to their original dataset and the sum of all ranks is calculated for each dataset. Based on the difference in the sum of rank between the datasets, a probability value is calculated. We reject the null hypothesis if we find $p < \alpha$, where we take the significance level α to be 0.05.

5.5 Inclusion of Constants

For the inclusion of constants in GP, a technique called Ephemeral Random Constant sampling is often used. This technique, explained by Koza in chapter 10.2 of his book [26], is used to generate nodes called ERC nodes, which are added to the terminal set. Whenever such a node is included into a solution, a number is sampled randomly from a predifined range and assigned to the node as its value for the remainder of the run. Often, this range is set to be [-1,1], an arbitrary choice which does not take the data into account. In this work, we sample from the range [data$_{min}$,data$_{max}$] instead, as in [39], as we suspect this to be a more effective approach. Here, data$_{min}$ is the smallest value found among the independent variables in the dataset, while data$_{max}$ is the largest.

6 Comparative Study

In this section, we discuss the experiments conducted to answer the first two of our research questions: to see how the different algorithms compare and to discover useful principles which can be used in the design of SensGP. We discuss the experimental setup in Section 6.1. Next, the results of the experiments are presented in Section 6.2. We discuss the differences due to implementation language in Section 6.3. In Section 6.4 we formulate an answer to the first two research questions.

6.1 Experimental Details

Since the algorithms we want to compare are written in different languages, differences in performance might occur that can not be attributed to the algorithms themselves. These interlanguage differences will skew the results, although the performance between `C++` and `Java` is ordinarily quite similar. In a study by Gherardi et al. [15] `Java` was shown to be slower on a number of 3D modelling benchmarks by a factor of up to 1.51. The exact difference varies on an application to application basis, however, which is why we will try to quantify the impact of the difference in implementation language on our experiments in Section 6.3.

To compare these algorithms written in different languages on the basis of a different quantity than time, we count the number of node evaluations performed during each evaluation of a model on a data record. It is impossible, however, to compensate for any differences in inherent performance between `C++` and `Java` by using this measure as the criterion for when the algorithms should terminate, since the number of evaluations performed per generation and per unit of time varies immensely between these algorithms. While e.g. EFS spends a large amount of time performing LASSO fits, SGP uses the majority of its computational resources evaluating solutions on the data.

Therefore, the termination criterion that was decided upon for the algorithms written in `Java`, was the time needed by SGP to complete 50 million evaluations. This was approximately 6 minutes on the Vladislavleva-4 dataset and 1 minute on the Keijzer-1 dataset. Nevertheless, we analysed differences in the number of node evaluations to obtain more insight into how these algorithms operate, the results of which can be seen in Section 6.3.

Multiple configurations for the `C++` algorithms were tested. The names SGP 10 and SGP 17 indicate that trees were allowed a maximum tree depth of 10 and 17 respectively. We tested these seperately to see if reducing the size of the models would reduce the amount of overfitting. We tested GP-GOMEA with both the Linkage Tree (LT) and Random Tree (RT) configurations, to see if learning a structure via the LT would yield better results than using a randomized structure. Differently from [39], no linear scaling is used, despite it being recommended. Since this option was not available for the algorithms written in `Java`, using this options would make for an unfair comparison. All of the SGP and GP-GOMEA settings were tested using the Interleaved Multistart Scheme (IMS) introduced in Section 3.1.2 as well, on the Keijzer-1 dataset.

All of the computations necessary for the comparative experiments were executed on the Asus laptop, of which the details can be found in Section 5.2.

A specification of all parameters used can be found in Table 3. The time

limit termination criterion which was used for EFS, BP and MRGP, was set to 1 minute for the Keijzer-1 dataset and 6 minutes for the Vladislavleva-4 dataset. The other algorithms used the node evaluation limit as a stop criterion. For the inclusion of constants, we used ERC, which is described in Section 5.5.

Table 3: Parameter Settings.

Parameter	**Value**
Time limit	1, 6 minutes
Evaluation limit	50 million
Population size	512
Terminal set	independent variables + ERC
Function set	$(+, -, \times, \div, .^2, \sqrt{\cdot})$
Tree initialization method	Ramped-half-n-half
Minimum initial tree height	2
Maximum initial tree height	4
Maximum tree height	17
Maximum tree height GP-GOMEA	5
Tournament size	7
SGP crossover/mutation rate	0.9/0.1
Elitism	1
IMS base population size	256
IMS generational step size	8
MRGP method	inline
MRGP complexity	tree complexity
BP archive size	10
BP archive mutation rate	0.9
EFS q	$3 \cdot p$
EFS μ	$1 \cdot p$

6.2 Results

The results from the experiments performed to see how the various algorithms discussed earlier compare to each other can be viewed in Tables 4, 5, 6, and 7. In these tables, the time in seconds is shown, the number of node evaluations, as well as the MSE of the best model in the population on both the training and the test data. All of these algorithms were run ten times and as there is no single median run from an even number of runs, the better one of the two median runs (so the fifth best run) is depicted in these tables.

In Tables 4 and 6, the median run is selected based on the test MSE, while in Tables 5 and 7 it has been selected based on training MSE. While test MSE is the more important measure, as it shows how well a given algorithm generalizes and gives a better indication of whether the algorithm would perform well as a predictive device in a real world scenario, training MSE also provides valuable information about the inherent capability of an algorithm to search for improvement. Therefore, we chose to examine both.

In Table 4, the results of running the algorithms on the Keijzer-1 dataset are shown, with the median run selected based on test MSE.

Table 4: Comparison on the Keijzer-1 dataset, test MSE centred.

Algorithm	**Time (s)**	**Evals** ($\cdot 10^3$)	**Train MSE** ($\cdot 10^{-3}$)	**Test MSE** ($\cdot 10^{-3}$)
EFS	50	119	1.687	1.344
BP	60	1273	1.695	0.702
MRGP	65	122	4.813	18.301
SGP 10	36	50030	0.095	5.0736
SGP 10 IMS	42	50369	**0.007**	**0.157**
SGP 17	36	50109	0.021	3.228
SGP 17 IMS	43	50119	0.077	0.421
GOM LT	44	50025	6.696	7.000
GOM LT IMS	76	81093	0.217	0.292
GOM RT	35	50006	6.692	6.959
GOM RT IMS	27	54185	1.425	1.130

We observe that overall, SGP performs very well. The best and third best test MSE are obtained by SGP 10 IMS and SGP 17 IMS respectively. The differences between the algorithms are quite large and it is difficult to come to solid conclusions about the performance of the techniques based on the test data, as for many algorithms there is a large difference between train and test MSE. This is to be expected, as the dataset is very small, increasing the chance of overfitting occurring.

In Table 5, results of the same experiment are shown, but with the median run being selected based on training MSE.

Table 5: Comparison on the Keijzer-1 dataset, train MSE centred.

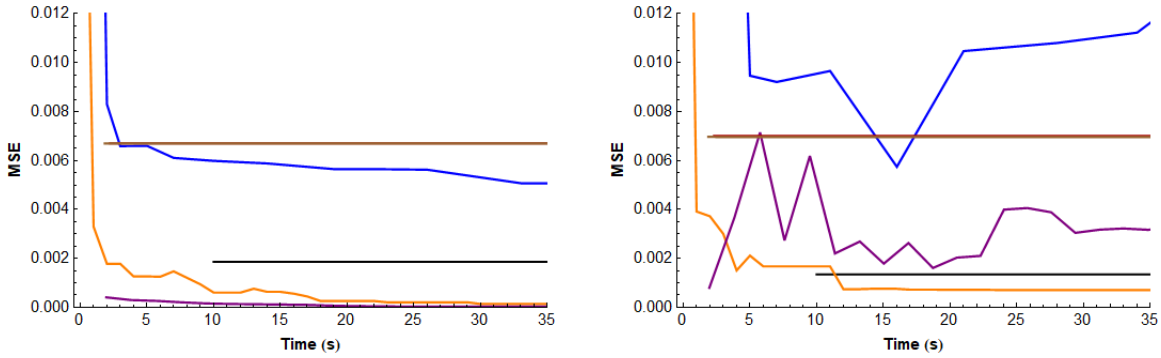
Algorithm	**Time (s)**	**Evals** ($\cdot 10^3$)	**Train MSE** ($\cdot 10^{-3}$)	**Test MSE** ($\cdot 10^{-3}$)
EFS	50	73	1.857	1.158
BP	60	1491	0.119	0.789
MRGP	68	122	4.785	24.112
SGP 10	38	50100	0.032	52.626
SGP 10 IMS	41	50038	0.075	0.176
SGP 17	37	50109	**0.022**	3.228
SGP 17 IMS	43	50026	0.107	**0.106**
GOM LT	41	50004	6.703	6.866
GOM LT IMS	78	80424	0.423	0.322
GOM RT	36	50003	6.693	6.959
GOM RT IMS	28	54758	1.206	1.511

From these results as well, it is apparent that SGP is good at finding solutions for this low dimensional problem. We hypothesize this is due to the size of the problem being so small that SGP, which does not have an explicitly guided evolution, explores more of the search space. Its solutions are likely to be more diverse and thus more of the relatively small problem space is explored, while the other algorithms might be hindered by their guided approach.

The guided algorithms might converge the search direction too fast and end up getting stuck in a local optimum, spending much of their time on ML techniques rather than purely exploring the space.

To investigate this hypothesis, we visualised the course of these median runs over time. All configurations with IMS and SGP 10 are not considered, to reduce the comparison to one configuration per algorithm. Both the training and test medians are shown in Figure 5, with a legend defining which algorithm corresponds to which color in Figure 4.

Figure 4: Legend.



(a) Training set.

(b) Test set.

Figure 5: Mean Squared Error (MSE) versus Time (seconds) on the Keijzer-1 dataset for different algorithms. The associated legend is shown in Figure 4.

The graphs of GP-GOMEA with LT and RT are indistinguishable from each other as they find models with almost identical MSE, on both the train and test sets. We observe that EFS and both GP-GOMEA methods have already converged to their final solution at the first time of measurement. This suggests these algorithms are indeed converging their search process too fast. BP, MRGP and SGP, however, are continuously improving their best solution over time on the training data, although the best model found in the median run by MRGP is far behind the other two algorithms. From the depiction of the model performance on the test set, we observe that both the MRGP and SGP runs suffer from overfitting on the training set, with their model quality lower at the end of the run than it had been earlier.

Next, we inspect the results of the experiments on the larger Vladislavleva-4 dataset. While it was interesting to see how the algorithms performed on the small Keijzer-1 dataset, datasets are normally much larger and more complicated and thus these results are not really reflective of the real capabilities of these algorithms. A larger challenge will be presented to the algorithms by experimentation on the Vladislavleva-4 dataset, and we will primarily use results from this experiment to search for answers to our first two research questions.

Table 6 shows the results of applying the algorithms to the larger dataset, with the median run being selected based on test MSE. The IMS configurations

were excluded from this experiment, since all algorithms for which IMS has not been implemented are being tested for only 1 parameter configuration, which would give the IMS configurations an unfair advantage. Additionally, because of the implementation we use, the runs with IMS were found to exceed the set evaluation limit. The algorithm would have multiple runs active at a time, which would all be allowed to complete their calculations, resulting in up to 40% additional evaluations performed before the algorithm halted. For SGP, the run with tree depth 10 was excluded as its performance was not as good as that of tree depth 17.

Table 6: Comparison on the Vladislavleva-4 dataset, test MSE centred.

Algorithm	**Time (s)**	**Evals** ($\cdot 10^3$)	**Train MSE**	**Test MSE**
EFS	360	594	0.0059	**0.0066**
BP	361	154	0.0378	0.0317
MRGP	380	353	**0.0026**	0.0186
SGP	332	50163	0.0121	0.0762
GOM LT	460	50007	0.0375	0.0370
GOM RT	356	50106	0.0365	0.0351

The results differ enormously from the results on the Keijzer-1 dataset. Whereas EFS, BP and MRGP were all behind SGP in the previous experiment, they now prove to be the three best algorithms, with EFS taking the lead by a large margin. Overall scores for the Test MSE are more closely tied to the training MSE, implying that overfitting has a smaller impact here than it did earlier, as is expected due to the size of the dataset.

In Table 7, the same experiment is shown, with the median run being selected based on training MSE.

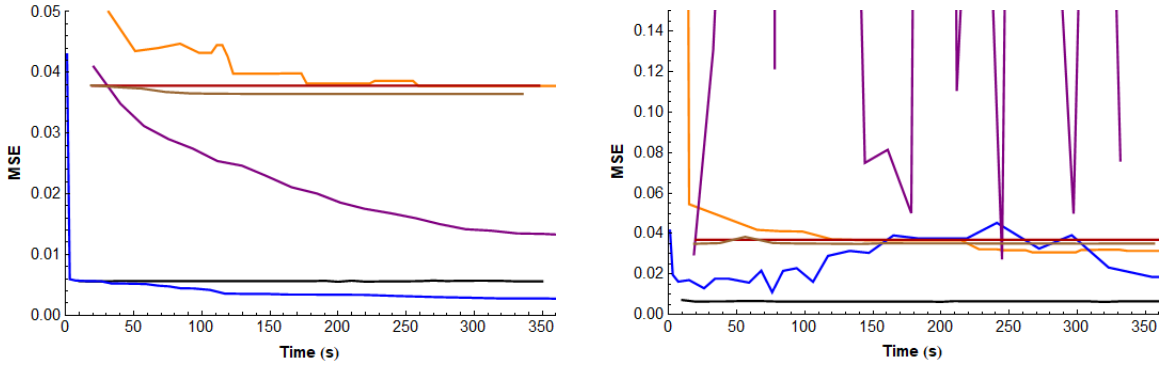
Table 7: Comparison on the Vladislavleva-4 dataset, train MSE centred.

Algorithm	**Time (s)**	**Evals** ($\cdot 10^3$)	**Train MSE**	**Test MSE**
EFS	350	577	0.0056	101.499
BP	363	99	0.0380	0.0315
MRGP	380	354	**0.0026**	**0.0186**
SGP	369	50017	0.0132	16.3259
GOM LT	348	50152	0.0378	0.0346
GOM RT	335	50106	0.0365	0.0350

Here we observe MRGP and EFS taking a clear lead, with SGP the third best and GP-GOMEA and BP performing the worst. From these results, it seems that in case of the more realistic dataset, the guided evolution can provide a meaningful improvement over SGP. GP-GOMEA does not seem to be an improvement. In case of the LT, this behaviour might be explained by of the nature of the SR problem. GP-GOMEA intends to learn the problem structure from the solutions in the population, but in SR the solutions are formulas which have no fixed form. Any term can be located at any point in a formula and two completely different

looking solutions might be semantically identical. In the RT model, however, the structure is constructed randomly, yet the results are very similar to the LT variant. Therefore, a more feasible explanation for the mediocre performance is that because of the bounded solution size of GP-GOMEA there is less room to fine-tune the models. As mentioned in Section 3.1.2, it is recommended to use linear scaling to increase the performance of these small models, but this would have made for an unfair comparison to the algorithms written in `Java`, as they did not have this method available to them.

To further investigate how these algorithms operate, we visualised the course of the median runs through time, both train and test median centred. The results are shown in Figure 6, with the same legend applying as before, shown in Figure 4.



(a) Training set.

(b) Test set.

Figure 6: Mean Squared Error (MSE) versus Time (seconds) on the Vladislavleva-4 dataset for different algorithms. The associated legend is shown in Figure 4.

Here we observe from the training data, as in the experiments on the Keijer-1 dataset, that EFS and both GP-GOMEA versions converge extremely quickly. The best solution is found before the first time of measurement and (almost) no improvement follows. BP, MRGP and SGP on the other hand keep improving over time. While EFS clearly performs the best on the test set, the continuous improvement of these aforementioned three algorithms suggests that if they are allowed more time to search for improvement, especially MRGP has the potential to find even better models than EFS.

What is also quite clear from this visualisation, is the unstable nature of the SGP solution on the testing set. Although the the fit on the training data improves steadily over time, how well the corresponding model describes unseen data is completely unpredictable. Definite conclusions as to the generalisability can not be drawn, however, as the run depicted here could be an exception.

6.3 Analysis of Evaluations

To study the differences in performance between algorithms due to the different languages they were written in, we measured the number of node evaluations on the dataset. To ensure that this comparison was not biased towards any algorithm, we did not just multiply the number of individuals in a population with the number of data records in the dataset. Instead, we defined an evaluation to be the calculation of the value of a single node, on a single data record. Therefore, if a solution consists of k nodes, in a single pass of a dataset consisting of n records, $k \cdot n$ evaluations are counted.

This definition of an evaluation proved not to be an honest measure for comparison, however, as in EFS, BP and MRGP a lot of computation time is spent performing regression rather than evaluation of solutions. From Table 7 we observe that the difference in number of evaluations performed in a similar time frame can be upwards of two orders of magnitude. The motivation for looking at the node evaluations in addition to the time spent was to see if any performance differences caused by the use of `C++` or `Java`, could be negated. Clearly this measure is not suitable, as the difference is of a completely different order of magnitude than could be explained by the difference in languages (see Section 6.1).

In addition to our previous measure, we explored another measure to research the speed difference between the implementations. We scaled the number of node evaluations by a scaling factor, which was calculated to be the average time spent by an algorithm on model evaluation divided by the total time spent during a generation.

The Scaled Evals are then computed by dividing the number of node evaluations by this scaling factor. This measure might be a better indication of what the speed reduction due to difference in implementation language might be, as it gives an indication of how many evaluations could have been made by the algorithm if all time had been spent on model evaluation. The results are shown in Table 8.

Table 8: Scaling Factors.

Algorithm	**Scaling factor (Std. dev.)**	**Evals** ($\cdot 10^3$)	**Scaled Evals** ($\cdot 10^3$)
EFS	0.048 (0.012)	594	12375
BP	0.217 (0.043)	154	710
MRGP	0.0492 (0.017)	353	7175
SGP	0.941(0.003)	50163	53308
GOM LT	0.927 (0.009)	50007	53945
GOM RT	0.949 (0.003)	50106	52799

The first thing to note is that BP presents a significant outlier in this set. The time spent by BP on evaluating solutions is located between on the one hand EFS and MRGP, which spent about 5% of their computation time evaluating solutions, and the SGP and GP-GOMEA algorithms on the other hand, which spent over 90% of their time evaluating. This is quite strange, as the number of total evaluations made in the BP algorithm is the lowest of all the algorithms. This suggest the evaluation procedure of BP is quite a bit slower than that of

the other algorithms.

We notice that EFS has an approximate difference in scaled evaluation of factor 4 compared to SGP. MRGP is worse of, by a factor of around 7. These numbers, while they are rough estimates, are much closer to those found by Gherardi et al [15], and could be partially due to differences in implementation language. This suggest that a noticeable performance increase might be achieved by implementing EFS and MRGP in `C++` instead.

6.4 Discussion

The results of the comparisons presented in this section allow us to answer the first two of our research questions.

The first question we posed is:

- How do EFS, BP, MRGP and GP-GOMEA perform in comparison to SGP and each other?

Throughout Section 6.2 we commented in detail on the results of the experiments and to answer this question, we discuss the overall trend shown by the different algorithms.

GP-GOMEA does not perform particularly well on either the Keijzer-1 or the Vladislavleva-4 dataset. On the smaller set, the results vary a lot depending on the settings used, but even its best configuration is outperformed by SGP on both train and test MSE. On the larger dataset, a better test MSE is found than by SGP on average, while SGP still proves better at finding model improvement as the training MSE shows. The overall performance of GP-GOMEA can be explained by the very restricted maximum model size compared to the other algorithms. To find competitive models, the use of linear scaling might be necessary.

BP performs decently on the Keijzer-1 dataset, scoring better on train and test MSE than MRGP and EFS, but lacking behind SGP. On the larger dataset, it is clearly outperformed by EFS and MRGP, while obtaining slightly better results than SGP on the test MSE. When looking at training MSE, it is outperformed by SGP.

EFS and MRGP are both outperformed by the other algorithms on the smaller dataset. As stated before, this is likely due to the guided searching mechanism of these algorithms focusing the search process too quickly into a small area of the problem space. On the larger Vladislavleva-4 dataset, however, the guided evolution is much more effective. Both in test MSE and train MSE, the algorithms significantly outperform all of the other algorithms.

SGP performs well on the smaller dataset, but can not compete with MRGP and EFS on the Vladislavleva-4 dataset. From looking at the differences between the test and train MSE, we suspect the models to be overfit on the training data more often than the models generated by the other algorithms.

The second question we posed is especially important, as it provides the basis for the remainder of this work:

- What insights can be obtained from the comparison of these methods which might prove useful for application in further research into improving SGP?

For solving small instances of SR, applying SGP to the problem seems to be a good choice, as its search mechanism is not explicitly guided into a specific direction, allowing a more diverse population to exist. Therefore, given a small dataset with few variables, a good solution can likely be found using SGP.

When a larger dataset with a greater number of variables needs to be modelled, it can be very beneficial to guide the evolutionary process in a promising search direction. MRGP and EFS demonstrate this approach works well on the Vladislavleva-4 dataset, and from the literature study we also conclude a guided evolution is a promising direction for research into improving SGP. Both MRGP and EFS guide the evolution focusing on a feature-based approach, applying ML techniques to measure which features are important and using these important features to construct new models. Due to the good performance of these algorithms, we elected to focus on this principal of a feature-based guided evolution as well, in the design of a novel SR algorithm.

7 Sensitivity-Based Approach

Using what we learned from the comparison of several algorithms in the previous section, as well as exploration of the literature, we designed an algorithm to improve upon SGP: Sensitivity-based Genetic Programming (SensGP).

This approach relies on the calculation of a feature importance score through sensitivity-like analysis, based on the paper Sensitivity-like Analysis for Feature Selection in Genetic Programming by Grant Dick [10]. These feature importances are saved in an archive, which we refer to as the dictionary, and are used to determine the probability of a feature being inserted into the population at a later time, through a SensitivityMutate operator.

Our reasons for suspecting this approach has merit are the following:

- The feature-centred/-guided evolution approach is at the heart of many of the methods currently being explored in research, such as EFS and MRGP. On problems with many variables, these approaches regularly prove to outperform SGP, which lacks any guidance of the evolutionary process other than through selection pressure.
- The sensitivity-like approach is shown by Grant Dick in his paper to be a good importance measure for single variable importance. When embedded into SGP to guide the evolution, it is shown to improve upon SGP. The technique remains unexplored for multi-variable features, however.

In this section we systematically explore the capabilities of the SensGP algorithm, via the following structure:

First, in Section 7.1 we present the general idea behind the approach. In Section 7.2 the base SensGP algorithm is described in detail, the parameters are explained and experiments are conducted to find good values for these parameters. Two options, Uniform Depth Variation (UDV) and Complexity Importance, are explored. Finally, we show how well the algorithm performs with the selected parameter settings in comparison to SGP, on the Vladislavleva-4 dataset. Then, in Section 7.3 we explore a variant of SensGP in which only a given percentage of the best individuals of the population are used in calculation of feature scores. We present the algorithmic adjustments made to SenGP, examine the parameters of this elitism-inspired approach and show experimental results. In Section 7.4 another variant is analysed, in which features are ranked based on a measure of how much the variables contained in this feature impact the overall model that contains this feature. This approach is most like the original sensitivity-like approach of Grant Dick, but extended to features containing multiple variables. Its algorithmic details are given, its parameters are examined and finally, experimental results are presented. Next, in Section 7.5 we compare SGP to all three variants of SensGP in a number of configurations. For this comparison we use all of the datasets described in Section 5.3. Finally, in Section 7.6 we formulate an answer to the final research question.

7.1 Basic Concept

In his paper Sensitivity-like Analysis for Feature Selection in Genetic Programming, Grant Dick suggests to use a novel method to calculate the importance of the input variables. This is done as follows:

1. At the end of a run, the fittest individual is considered.
2. Its MSE is computed on a hold-out partition of the data.
3. For each variable, the values in this partition are temporarily shuffled and model MSE is calculated again. The change in error compared to the MSE before shuffling is computed.
4. The changes in MSE are normalised to the largest change in MSE.

At first, this method is applied only as a measure of variable importance and the information obtained is not used in the evolutionary process itself. Later on in his work, he suggests to integrate this method into SGP, to enhance its search properties. The resulting method consists of the following steps:

1. All terminal selection probabilities are set to be equal initially.
2. After each generation, variable importance is calculated as described above.
3. The variables are ranked based on this importance measure.
4. The ranks are normalized by the sum of the ranks of all variables.
5. In the next generation, the probability of choosing a variable during mutation is set to be proportional to its rank.

This approach, named Adaptive Terminal Selection, shows at least some improvement on all test cases explored.

In the basic version of SensGP, we use the idea of calculating such an importance value for the input variables and extent it to include multi-variable features. We do not limit the calculation of feature importance to the best model, nor are all feature importance scores reset every generation. In this variant, evaluation of features is done separately on the dataset, independently from the model in which they appeared. All features are stored in a dictionary, alongside their importance.

The feature importance is a measure of the predictive value of a feature on the original dataset versus the predictive value on the dataset for which all data entries corresponding to the variables of the feature have been shuffled. The importance score of a feature t, which is represented as a tree in our program, is calculated as follows:

$$Importance(t) = \frac{MSE(t)}{MSE(t)_{shuffled}}, \tag{21}$$

where $MSE(t)$ is the MSE of t on the original dataset and $MSE(t)_{shuffled}$ is the MSE of t on the dataset for which all variable values of the variables present in t have been shuffled. A low importance score indicates that the feature has strong predictive properties, as shuffling the data for its variables increases the MSE. A more detailed description of the SensGP algorithm is presented in the next section.

7.2 Sensitivity-Based Genetic Programming

In this section, the SensGP algorithm is explained in-depth. In Section 7.2.1 the algorithm is discussed. All of the parameters to be specified in the algorithm are examined in Section 7.2.2. Some additional options which can be enabled are discussed in Section 7.2.3. Finally, we compare the SensGP to SGP in Section 7.2.4.

7.2.1 Algorithmic Details

Sensitivity-based GP is an extension of SGP and therefore makes use of the same traditional algorithmic order of program execution, depicted in Algorithm 1:

Algorithm 1 Algorithmic structure of SGP

```
1: procedure SGP(Data)
2:   InitializeRun()
3:   while (!Terminated) do
4:     SelectionForVariation()
5:     PerformVariation()
6:     EvaluatePopulation()
7:     Terminated = TerminationCriteriaMet()
8:   return BestModel
```

How each of these steps is altered in case of SensGP, is described below:

InitializeRun

In the initialize step, we need to ensure that the dictionary is filled with features, as otherwise we will not be able to perform the sensitivity mutation in the PerformVariation step in the first generation. In an FFX-like manner (see Section 2.6.3) all one- and two-variable feature combinations are generated, using the different operators available. Each of these combinations has its feature importance calculated by calling the CalculateImportance() method shown in Algorithm 2:

Algorithm 2 Calculating the importance of all features in a population

```
1: procedure CalculateImportance(treesToInspect,variablesInTree)
2:   CalculateFitness(treesToInspect)
3:   for i = 0, i < treesToInspect.size, i++ do
4:     if variablesInTree[i].size≠ 0 then
5:       t = treesToInspect[i]
6:       if t ∉ featureDictionary then
7:         unshuffledError = t.GetFitness()
8:         shuffledError = ComputeShuffeledDataFitness(t,variablesInTree[i])
9:         featureImportance = unshuffledError / shuffledError
10:        if featureImportance ≠ 1 then
11:          featureImportance = CheckUniqueKey(featureImportance)
12:          featureDictionary.Insert(featureImportance,t)
```

The algorithm takes a population of features as well as a vector in which the variables contained per feature are stored. Its purpose is to calculate the feature importance of each of the features and insert them into the dictionary.

On line 2 of the algorithm, the fitnesses of all features in the population are calculated and saved, to be retrieved later on line 7. On line 4 of the algorithm, a check is performed to see if the feature is not a constant, since in that case there are no variables to which we can measure sensitivity. On line 6, a check is performed to see if the feature is not already contained in the dictionary. Since feature scores in the basic variant of SensGP are calculated independently of the model they appear in, calculating the score for any feature which is already in the dictionary is obsolete.

On line 8, the shuffledError is calculated via the ComputeShuffledDataFitness() procedure. This procedure takes a feature as well as the variables occurring in it as its arguments and proceeds to randomly shuffle all data entries in the dataset, corresponding to these variables. If a feature contains multiple variables, these are shuffled together and will end up in the same new data record. All values for the variables not contained in the feature remain in the same data records as in the original dataset. After this operation is performed, the MSE of the feature is measured and returned.

FeatureImportance is then calculated straightforwardly by dividing the unshuffled MSE by the shuffled MSE. A check is performed on line 10 of the algorithm to see if the featureImportance is not equal to 1. The reason for this is that any feature with MSE equal to one is effectively a constant feature. When features have a featureImportance of 1, even though they contain variables, this is caused by a self-division or self-subtraction of the variable being contained in the feature (e.g. x_2/x_2). For such features, our importance measure is of no use and these are therefore not stored in the dictionary.

Next, it is ensured the featureImportance value is unique (to allow for two-way lookup in the dictionary) by the CheckUniqueKey() function. If the key (featureImportance) is already contained in the dictionary, the key is incremented by the smallest amount possible, restricted by the implementation of the double type in `C++`, until a unique value is found. Finally, the importance-feature pair is inserted into the dictionary.

After having inserted these pre-generated features into the dictionary through the CalculateImportance() method, the initialization phase takes the same step as in SGP: the population of models to be evolved in the generational loop of the algorithm is initialized by the Ramped-half-and-half method described in Section 2.4. Then, the function AfterPopulationGenerationImportance() is called, with the purpose of extracting all distinct features from the models in the freshly generated population and inserting these features together with their importance intro the dictionary.

AfterPopulationGenerationImportance() takes as a parameter a percentage corresponding to how many individuals of the population should be evaluated, which is set to 100% at this point as we want to evaluate the entire population. The procedure starts by evaluating and inserting all features corresponding to the input variables of the dataset into the dictionary through the AddSingleVarsToDict() method. This is not strictly necessary at this point, but is needed in the Model Dependent variant of SensGP, discussed in Section 7.3. All trees in the population are inspected, traversing every node for each tree

and checking its arity. If a node has an arity > 0, its subtree is checked for the presence of variable nodes. If the subtree contains variables, the subtree and its variables are saved in separate vectors and after all trees in the population have been inspected, these vectors are passed to the CalculateImportance() method described in Algorithm 2.

After the entire initial population has been inspected and all unique features are saved in the dictionary, the initialization is complete and the algorithm moves to the generational loop.

SelectionForVariation

In this specific implementation of SGP, and therefore SensGP, selection of individuals for mutation and crossover happens in the PerformVariation step and will be described there.

PerformVariation

The variation procedure works as follows: first, an operator is selected from the pool of operators in proportion to their given probabilities. These probabilities can be set via the crossover and mutation parameters. Once a given operator is selected, either one (in case of mutation) or two (for crossover) individuals are selected from the population by tournament selection. So far, the procedure mimics SGP precisely, but in SensGP, we adjusted the mutation operator. If the mutation operator is picked, a third parameter, sensitivity, comes into play. A random number is generated and if it is higher than the sensitivity parameter, ordinary mutation occurs. If it is lower, SensitivityMutate() is called.

SensitivityMutate() functions mostly like ordinary mutation, but instead of randomly generating a new tree via the Full or Grow methods, a feature is selected from the dictionary. This is done in one of two ways, either by tournament selection or a rank-proportionate selection based on feature importance. These methods for selection will both be described in Section 7.2.2. After a feature has been selected, it is inserted at the location of a random node of the tree undergoing the mutation operation, as would happen with ordinary mutation.

In the basic variant of SensGP, the individual that has undergone mutation is checked for new promising features at this point, by traversing up the tree towards its root node, starting at the location where the new feature was just inserted. For all nodes encountered, the subtree with this node as its root is checked for size, and if the subtree is smaller than the MaxDictSize parameter, it has its feature importance calculated by the CalculateImportance() method. In this way, new features enter the dictionary after the initialization phase.

EvaluatePopulation

As with SGP, after the variation step the new population is evaluated on the dataset and sorted by MSE.

TerminationCriteriaMet

At the end of the generational loop, in correspondence with SGP, the termination criteria are checked. Specifically, this can be a time limit, a limit to the number of node evaluations or a limit to the number of generations.

7.2.2 Parameters

With the algorithmic details having been laid out in the previous section, in this section we will discuss the influence of the parameters mentioned, as well as motivating through experiments which parameter values are suitable for usage in further experimentation.

Population Size

First, we assert the effect of the population size on SenGP. A commonly encountered value for the population size in the literature is 512, so we experiment with values on either side.

In Figure 7 the medians of 10 runs of SensGP, lasting five minutes each, are shown, for values of the population size parameter equal to 128, 256, 512, 1024 and 2048. This experiment was performed on the Microsoft Azure VMs.

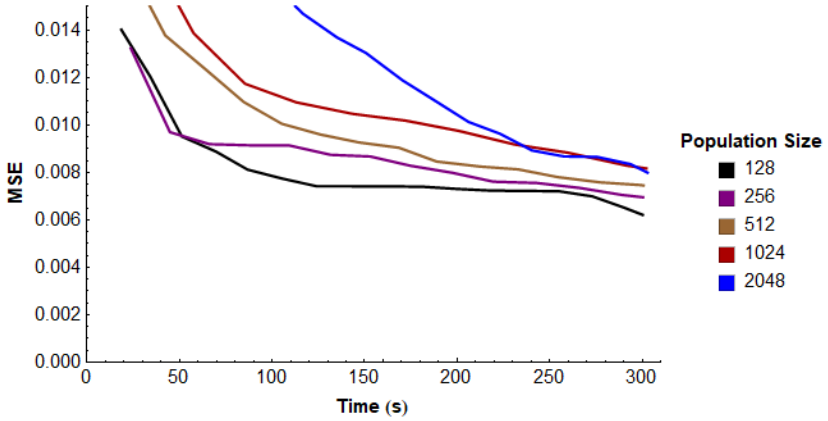


Figure 7: Mean Squared Error (MSE) versus Time (seconds) on training data for different population sizes.

From this figure, the model MSE values for the runs with different population sizes do not seem to differ significantly. Therefore, the initial value of 512 will be used throughout the remainder of this work.

Mutation, Crossover and Sensitivity Mutation

Picking the right values for the mutation, crossover and sensitivity mutation parameters must be carried out together, as their balance determines their impact on the algorithm. Tests with only the sensitivity mutation operator and no ordinary mutation were performed, but due to the similarity of the sensitivity mutation to the crossover operator the population would converge too quickly. Consequently, this setting was not an improvement over SGP, and thus this scenario will not be considered in the search for optimal parameter values. Similarly, having no sensitivity mutation is not of interest, as then we would obtain the results for SGP, which we already compare our end results to. The third possibility of not using the crossover operator is interesting, however, as sensitivity mutation shares some characteristics with crossover,

by introducing only features into the population which originate from other individuals. Therefore we did test settings for Pr(crossover) equal to 0.

Due to the way we implemented the SensitivityMutate() method, the sensitivity parameter value determines how often the method is picked in proportion to standard mutation. To show the impact of all three parameters simultaneously, we plot the value of mutation versus sensitivity, as the setting for crossover is fixed by $1 - Pr(mutation)$. Results are obtained by taking the median run of 5, with each run lasting 5 minutes. These computations were performed on the Microsoft Azure VMs.

The results of an initial grid search, with values for mutation in the range $[0.2, 1]$ and values for sensitivity in the range $[0.2, 0.8]$ with steps of 0.2, are shown in Figure 8.

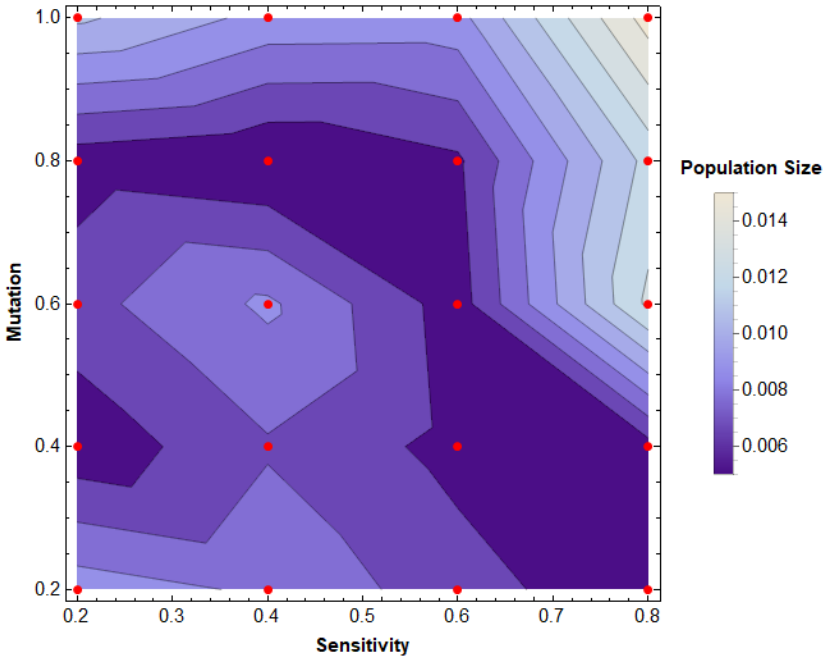


Figure 8: The Mean Squared Error of the median run of an initial exploration experiment of different parameter values of the mutation and sensitivity parameters on training data.

From this figure, we observed an area where settings seemed promising, the dark blue curve-shaped valley, starting at $Pr(mutation) = 0.8, Pr(sensitivity) = 0.2$ and ending at $Pr(mutation) = 0.2, Pr(sensitivity) = 0.8$.

We tested more parameter configurations in this promising area, adding them to the contour plot. The final result can be observed in Figure 9.

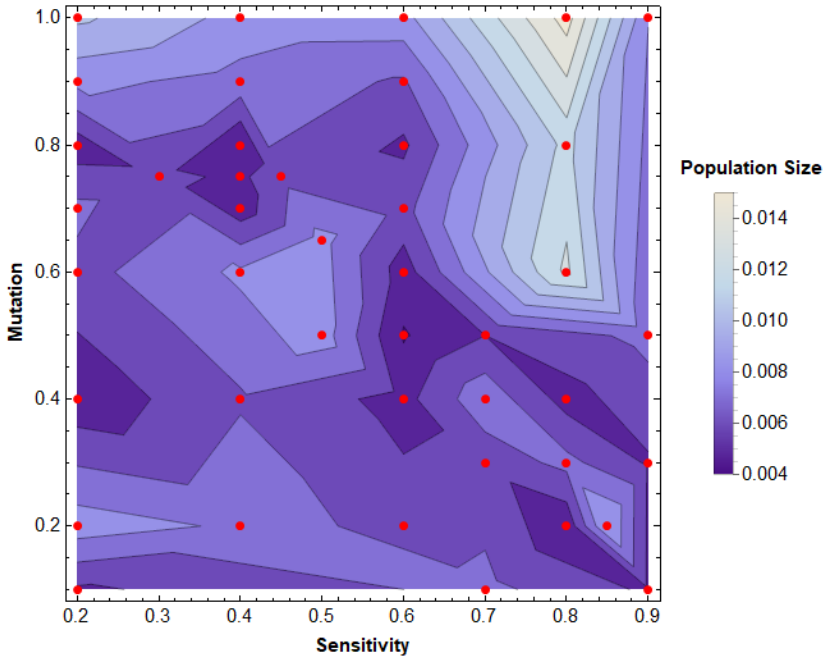


Figure 9: The Mean Squared Error of the median run after further investigation of different parameter values of the mutation and sensitivity parameters on training data.

Two parameter settings clearly stand out, and seem stable when taking neighbouring points into account. These were the settings:

Configuration 1: $Pr(mutation) = 0.75$, $Pr(sensitivity) = 0.4$.
Configuration 2: $Pr(mutation) = 0.5$, $Pr(sensitivity) = 0.6$.

We performed a thorough test between these remaining two value-pairs. For this comparison, we ran each setting 15 times, with a time limit of 10 minutes on the Asus laptop. The median runs are shown in Figure 10.

The average and standard deviation of these experiments have been calculated as well and are presented in Table 9.

Table 9: Comparison of two different pairs of values for the mutation and sensitivity parameters of SensGP.

Setting	**Configuration 1**	**Configuration 2**
Pr(mutation)	0.75	0.5
Pr(sensitivity)	0.4	0.6
Mean MSE	0.0067 (0.0024)	0.0042 (0.0013)

From this final experiment, it is clear that Configuration 2 is the superior configuration. The result is found to be significant by the Mann-Whitney U test, resulting in a p-value of 0.0021. Therefore, in the remainder of this work

the settings $Pr(mutation) = 0.5$, $Pr(sensitivity) = 0.6$ are used as the default settings.

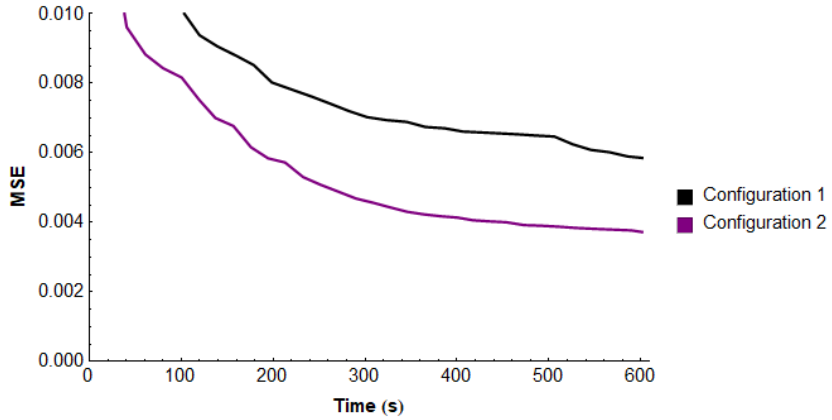


Figure 10: Mean Squared Error (MSE) versus Time (seconds) on training data for Configuration 1 and Configuration 2.

TimesAsLikely and TournamentRanking

In the SensitivityMutate() method, the first step is to select a feature from the dictionary to later insert into an individual. This can be done in two ways in the current version of SensGP:

- Rank-based selection. In rank-based selection, a TimesAsLikely parameter must be specified. This parameter determines how many times more likely it is that the feature with the smallest feature importance score in the dictionary is selected than the feature with the highest score. This sounds counter intuitive, but for feature importance a lower score indicates a better feature. The probabilities for all features with scores in-between are obtained based on their rank x, where the highest rank is assigned to the feature with the lowest feature importance. The probability of a feature with rank x to be selected is then calculated as:

$$Pr(x) = x * (TimesAsLikely - 1)/(NumberOfFeatures - 1) \quad (22)$$

- Tournament selection. Features are randomly selected from the dictionary and a tournament is held to see which feature has the lowest feature importance. This feature is selected for use in mutation. The TournamentRanking parameter determines between how many randomly picked features the tournament is held.

We tested different values for both of these parameters on the Microsoft Azure VMs, running 10 iterations of each setting. The results for TimesAsLikely are shown in Figure 11.

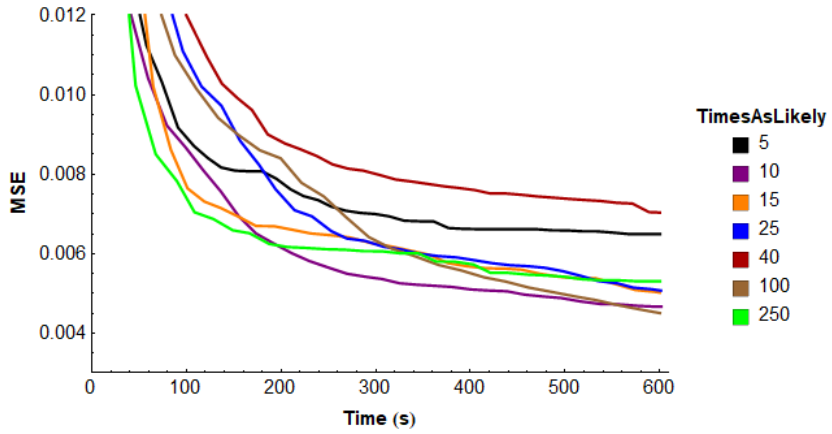


Figure 11: Mean Squared Error (MSE) versus Time (seconds) on training data for different values of TimesAsLikely.

There is no clear pattern to be observed, the two worst runs are those with TimesAsLikely values 5 and 40, while the values in-between all perform better. The region between these values appears to be a solid choice and any value picked in this region is presumed not to have a large negative impact on the performance of SensGP.

The results of experiments with different values of the TournamentRanking parameter are shown in Figure 12.

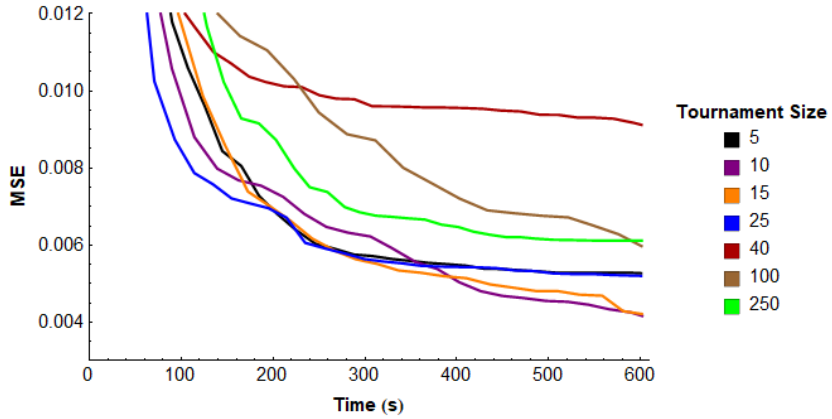


Figure 12: Mean Squared Error (MSE) versus Time (seconds) on training data for different Tournament Sizes.

The results for the TournamentRanking parameter are very similar to those for the TimesAsLikely parameter. There appears to be an outlier for the value 40, while all surrounding values perform comparably.

We expected tournament ranking might be a significant speed-up over the rank-based selection, as in the latter method, the dictionary has to be traversed many times by an algorithm known as roulette wheel selection. Therefore, we

measured how much of the total time spent during a run was spent on the feature selection methods. The Visual Studio Analysis tool was used for these measurements.

By switching from tournament ranking using a very small tournament size of 3, to rank-based selection with a TimesAsLikely value of 5, we found that the time spent on the SensitivityMutate() method as a whole increased from 7.96% to 9.67% of the total computation time. In the Model Dependent and Variable Importance variants of SensGP, we found these increases to be from 1.63% to 1.67% and from 0.89% to 1.63% respectively. While tournament selection does prove to be faster, when looking at overall computation time the difference is negligible. If parameter values for TournamentRanking and TimesAsLikely increase, this difference will be further reduced, as a larger tournament size means more feature importance scores have to be compared. As the probabilities of feature selection are easy to calculate explicitly using Equation 22, resulting in an increased understanding of with what probability features are selected, we opted to stick with the rank-based method, with a TimesAsLikely value of 25.

Recalculation

As a run progresses in time, the size of the dictionary grows, as new features are inserted into it, while no features are ever removed. Especially with constants enabled, there are many different features which might be inserted into the dictionary. This can have a negative effect on the speed of the algorithm, as having more features to choose between means more time is spent selecting a feature through rank-based selection. Additionally, the overall quality of the features being selected could be reduced, as having more features in the dictionary decreases the chances selecting good features.

Therefore, we decided to add an optional mechanism, which empties the dictionary every certain number of generations. These generations are specified as those that result in a value of 0 when the modulus with respect to the RecalculateScore parameter is taken. Then, in case of SensGP, the dictionary is re-instantiated using the features from the models of the best percentage of the population specified by the RecalculationFactor parameter, by calling the AfterPopulationGenerationImportance() method discussed in Section 7.2. The Recalculation option is of particular importance in the Model Dependent and Variable Importance variants of SensGP and will be discussed in relation to these in Sections 7.3 and 7.4.

We performed experiments for 5 values of RecalculateScore, with RecalculationFactor set to 0.05. For each experiment, the median of 10 runs was taken. The experiments were performed on the Asus Laptop. The results of these experiments are depicted in Figure 13.

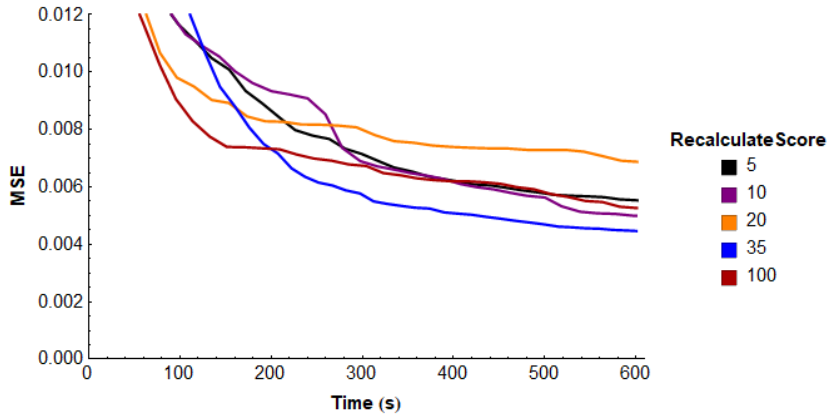


Figure 13: Mean Squared Error (MSE) versus Time (seconds) on training data for different values of RecalculateScore with RecalculationFactor set to 0.05.

The results are quite similar, but the values 20 and 35 seem to stand out as the best and worst value. As it would be very strange for the runs with the lowest two and the highest parameter values to produce models of nearly equal quality, while the two values in-between represent the best and worst settings, we suspect these results are a consequence of the variance. Overall, using the Recalculation option does not seem to improve upon ordinary SensGP.

Maximum Feature Size

In the SensitivityMutate() method of SensGP, only trees which are smaller than a certain number of nodes are used for importance calculation. This is done as we expect features with more nodes might be too complex to be suitable for insertion into another individual. To test this hypothesis, we made size into a parameter named MaxDictSize and performed a comparison experiment. We tested the values 7, 10, 12 and 15, performing 10 runs with each setting using the Asus Laptop, observing the median runs. The results are shown in Figure 14.

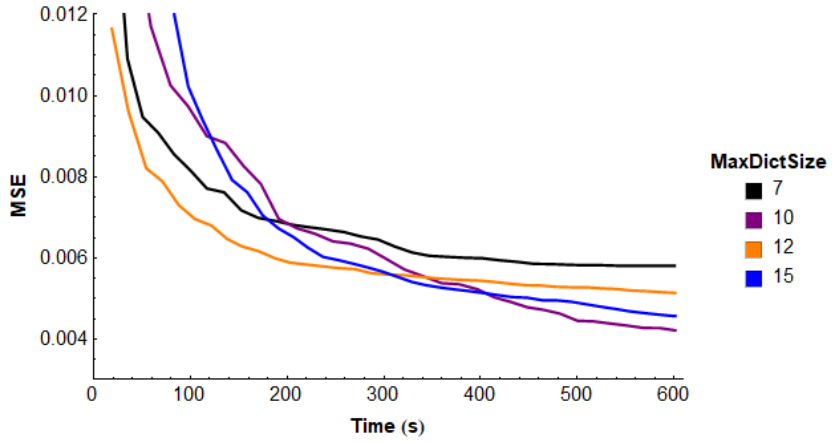


Figure 14: Mean Squared Error (MSE) versus Time (seconds) on training data for different values of MaxDictSize.

From this figure, is appears that setting the maximum feature size to 10 achieves the best results.

Final Parameter Values Selected

A complete list specifying the parameter values used in the experiments in the remainder of this section is shown in Table 10. For the ModelDep, VarImp and Recalculation parameters different values are used every experiment, which are then further specified. For the experiments on the Nguyen-7 dataset we added the Log() function to the function set, and for the experiments on the Korns-12 dataset we added the Sine() and Cosine() functions.

Table 10: Parameter Settings.

Parameter	**Value**
Time limit	10 minutes
Population size	512
Terminal set	independent variables + ERC
Function set	$(+, -, \times, \div, .^2, \sqrt{\cdot})$
Tree initialization method	Ramped-half-n-half
Minimum initial Tree height	2
Maximum initial Tree height	4
Maximum tree height	17
Tournament size	7
SGP crossover/mutation rate	0.9/0.1
SensGP crossover/mutation rate	0.5/0.5
Elitism	1
SensitivityMutate	0.6
TimesAsLikely	25
Maximum feature size	10

7.2.3 Additional Options

In addition to the tunable parameters discussed in the previous section, we implemented three more switchable options into SensGP: Uniform Depth Variation, Complexity Importance and the inclusion or exclusion of constants.

Uniform Depth Variation

Uniform Depth Variation (UDV) [20] is an adjustment to the standard crossover, mutation and sensitivity mutation operators, which is also used in BP (See Section 3.2.2). Normally, the variation operation is applied to a randomly selected node. Since the solutions in a GP population are stored in a tree structure, each subsequent level of depth is likely to contain a larger number of nodes than the previous level. In case of a full tree of binary function nodes, the relation between the depth level k in the tree and the number of nodes at this level equals 2^k.

In practice, a tree will not be completely filled, as terminals and operators of arity one can be contained at all levels of the tree. Still, on average the number of nodes per depth level of the tree increases with depth. This causes

the probability of a variation operation occurring somewhere in the larger depth levels of a tree to be larger than at the levels near the root. This can be unwanted behaviour, since nodes located near the root are more likely to have a larger impact upon total tree fitness.

To combat this possible issue, we added to option of using UDV into SensGP. In UDV, instead of selecting a node randomly from the set of all nodes in the tree, first a depth level is picked from the set of all depths occurring in the tree, with equal probability. Then from the set of all nodes contained in this depth level of the tree, a random node to apply the variation operator to is selected. Selecting nodes in such a manner increases the probability of nodes near the root node of the tree being used for variation.

The usefulness of the UDV option is researched, with the results shown in Figures 15 and 16, as well as in Table 11, together with those from Complexity Importance and SensGP for comparison. For these experiments, the Virtual Machines from the Microsoft Azure service were used and the median run out of 10 is shown.

The figures suggest there is no added benefit of enabling the UDV option in comparison to using SensGP, as on both the train and test data the median run has the worst performance of the three configurations which are compared. The p-value to SensGP is not below the threshold of 0.05 in either case, however, so the difference is not statistically significant.

Complexity Importance

Keeping the size of solutions in the population relatively small is beneficial for two main reasons. First of all, it reduces the chance of overfitting on the training data, as there is less room for fine tuning in smaller solutions. Secondly, the models produced are more legible, which makes people more inclined to use them, as a model which is understandable gives some assurance as to what it does.

To this end, we tested complementing our feature importance measure with a Complexity Importance measure. Before the feature importance x of a feature is inserted into the dictionary, it is first scaled as:

$$ComplexityImportance(x) = x \cdot (\frac{numberOfNodes}{10} + 1). \tag{23}$$

This formula was chosen, as it scales the ordinary feature score, but the impact of the solution size is reduced to prevent it from becoming the dominating factor. If a straightforward multiplication of the importance by the number of nodes was made, the impact of the size might have been too large. The number 10 is picked as it proved to be a good maximum feature size, as shown in 7.2.2.

The results of this experiment, together with those from SensGP and UDV are shown in Figures 15 and 16. For these experiments, 10 runs were performed on the Microsoft Azure VMs.

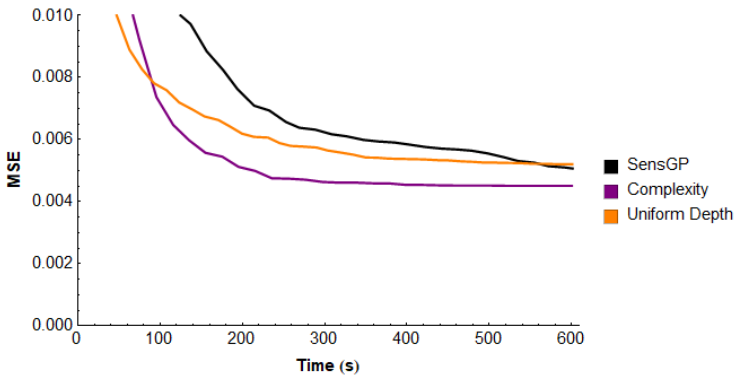


Figure 15: Mean Squared Error (MSE) versus Time (seconds) on training data for different options of SensGP.

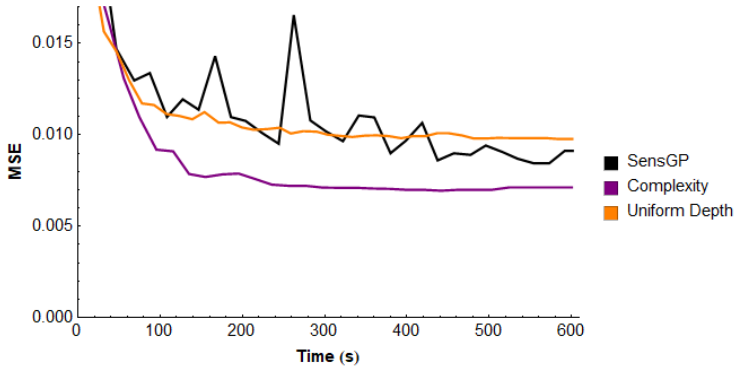


Figure 16: Mean Squared Error (MSE) versus Time (seconds) on test data for different options of SensGP.

Here we observe that while uniform depth mutation grants results similar to SensGP, the Complexity Importance measure seems to improve upon SensGP. From a Mann-Whitney U test, a p-value of 0.52 was obtained, however, on both the training and test sets, indicating this result is not significant.

Table 11: Comparison of SensGP to SensGP with Uniform Depth Variation and Complexity Score.

Training	Median MSE	Mean MSE	Std. dev.	p-value to SensGP
SensGP	0.0051	0.0055	0.002	0.97
Complexity	0.0045	0.0051	0.0018	0.52
Uniform Depth	0.0052	0.0077	0.0052	0.47
Testing	**Median MSE**	**Mean MSE**	**Std. dev.**	**p-value to SensGP**
SensGP	0.0091	0.0099	0.0041	0.97
Complexity	0.0071	0.0087	0.0038	0.52
Uniform Depth	0.0098	0.021	0.027	0.16

Constants

A significant issue in SensGP is presented by having to store constants. Currently, constants are included into the dictionary by value. While this works, it is not very elegant and can result in a lot of additional, unnecessary features being included. The reason for this is that if a feature, say $0.35 \cdot x_1$, is included into the dictionary, any feature $(0.35 + \epsilon) \cdot x_1$ might be generated through mutation or crossover anywhere else in the population and then included into the dictionary as well. As this can happen for any value of $\epsilon \simeq 0$, as allowed by the `C++` double type, there are many semantically almost identical features which could be inserted into the dictionary.

Saving the constant as a constant without value, on the other hand, would give a wrong indication of the usefulness of the feature. If, for example, the previously mentioned feature $0.35 \cdot x_1$ would be saved as $C \cdot x_1$, this would mean only one entry for this combination would be allowed in the dictionary. It would represent both $0.35 \cdot x_1$ and e.g. $1234.45 \cdot x_1$, which would likely be of widely varying usefulness to the problem, yet would obtain identical importance scores.

Although constants are essential in real datasets and we thus chose to perform all experiments with constants being included into SensGP, to see if there is indeed a greater impact of constants on SensGP than on SGP, we decided to conduct an experiment where both of these algorithms were not allowed to generate constants.

The results of this experiment, performed on the Asus laptop, can be seen in Figures 17 and 18, as well as Table 12.

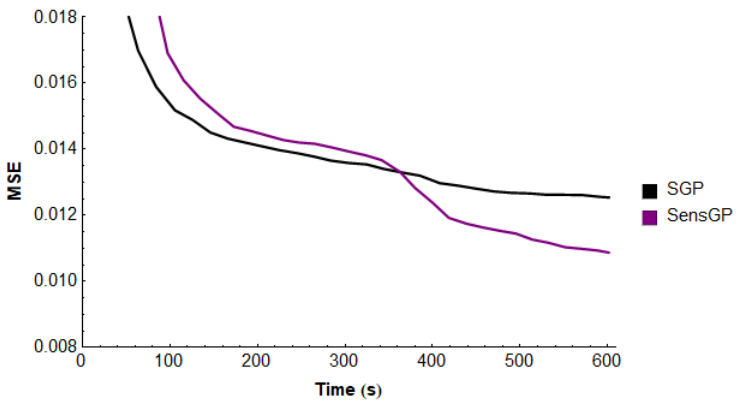


Figure 17: Mean Squared Error (MSE) versus Time (seconds) on training data for SGP and SensGP with constants disabled.

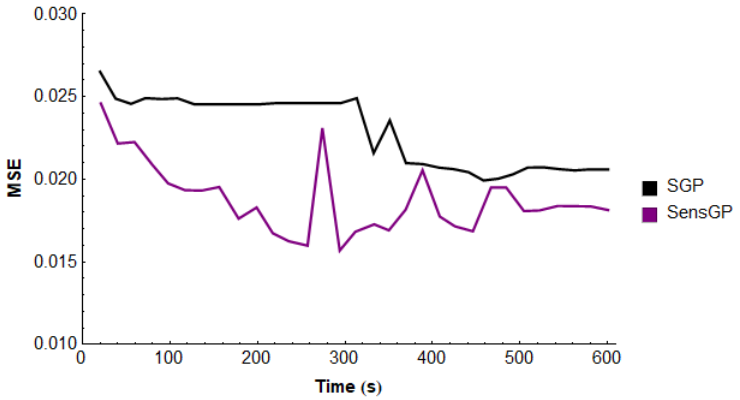


Figure 18: Mean Squared Error (MSE) versus Time (seconds) on test data for SGP and SensGP with constants disabled.

When compared to results of both algorithms with constants enabled in Section 7.2.4, we observe that without the ability to evolve constants directly, the performance of both algorithms decreases dramatically. The median run is almost three times worse than when constants are allowed, for both algorithms.

Table 12: Comparison of SensGP to SGP with constants disabled.

Training	Median MSE	Mean MSE	Std. dev.	p-value to SGP
SGP	0.0125	0.0138	0.0050	0.9830
SensGP	0.0109	0.0119	0.0043	0.3840
Testing	Median MSE	Mean MSE	Std. dev.	p-value to SGP
SGP	0.0206	0.0199	0.0062	0.9830
SensGP	0.0181	0.0180	0.0040	0.4550

It is not necessarily clear from this comparison that SensGP had more difficulties in incorporating constants into its models due to use of the dictionary, as in both the constant and no constant case it outperforms SGP by around the same factor as on the Vladislavleva-4 dataset.

7.2.4 Experiments

In this section, the performance of SensGP is compared to that of SGP on the Vladislavleva-4 dataset. In Figure 19, as well as in Table 13 of Appendix A, the results of executing 15 runs of both of these algorithms on the training set can be seen. All of these experiments were performed on the Asus laptop described in Section 5.2.

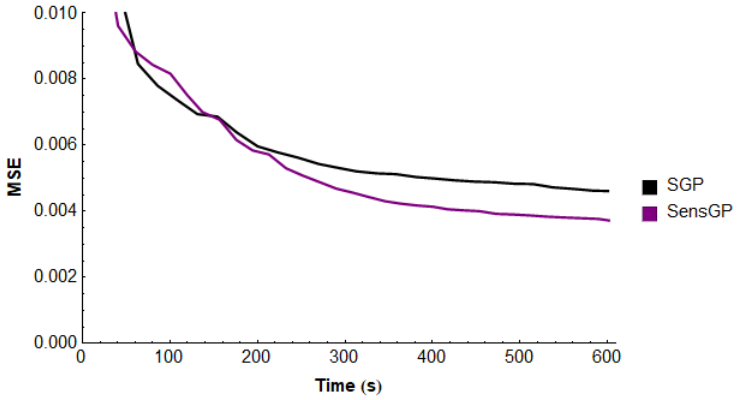


Figure 19: Mean Squared Error (MSE) versus Time (seconds) on training data for SGP and SensGP.

In Figure 20 and Table 13, as before, the same experiment is shown but on the test set.

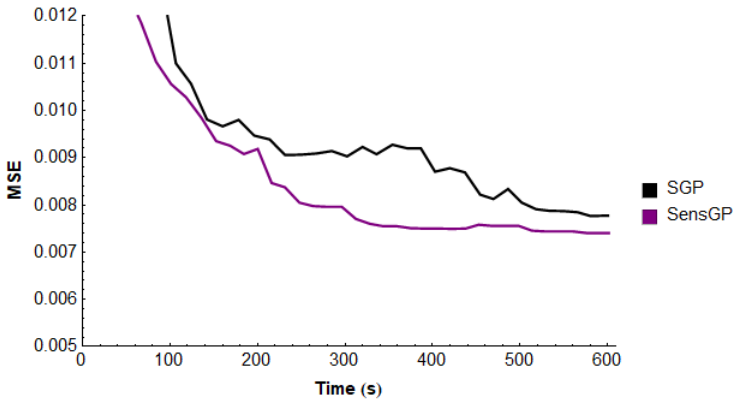


Figure 20: Mean Squared Error (MSE) versus Time (seconds) on test data for SGP and SensGP.

These figures show that the median run of the SensGP algorithm outperforms the median run of SGP. On the training data, the MSE scores achieved in the median runs by SensGP and SGP are 0.00373 and 0.00463 respectively. On the test data the MSE scores were 0.00741 and 0.00777. From the results of the The Mann-Whitney U test performed we observe that these differences are not statistically significant, however, with p-values of 0.106 on the training data and 0.229 on the test data.

7.3 Model Dependent

In addition to the SensGP algorithm, we experimented with two major adaptations of it. In this section the Model Dependent (ModelDep) approach is discussed. The underlying idea behind ModelDep is that SensGP might be improved upon by evaluating the importance only of the features from the best models in the population. We hypothesize that, since these models perform better as a whole on the given problem set, its parts should on average also be better than those of the average model.

In examining this approach, we have put additional effort into testing how well it works in combination with the Recalculation parameter. We expect that as the population evolves, new models will become the best models, and getting rid of the features that appeared in previously good models might keep the search going in a fresh direction.

7.3.1 Algorithmic Details

A couple of changes have been introduced into SensGP to make this approach function. First of all, in the initialization step, no importance calculations are made. At the end of the initialization step, the newly generated population is evaluated and its models are sorted by MSE. This is essential, as the first step of each generation of ModelDep is to call the AfterPopulationGenerationImportance() function, discussed in Section 7.2, with the ModelDep parameter. This parameter is multiplied by the population size to determine how many individuals should be inspected. Since we want the best models from the population to be used for feature importance calculation, instead of using random models, the population must be sorted on fitness at this point. If the value for ModelDep is zero, only the single best model is used for feature importance calculation.

To ensure the dictionary is never empty, the features corresponding to the input variables are always evaluated and added to the dictionary by the AddSingleVarsToDict() method. In absence of this measure, the dictionary being empty at the moment of variation has been observed to happen, especially in the case for a single model being analysed each generation, specifically when the current best model or models are effectively a constant (the MSE = 1 case).

In Sensitivity Mutation, the evaluation of subtrees of models which have just been modified is omitted. The only importance calculation is performed at the start of the generation. If the current generation is divisible by the parameter setting for RecalculateScore, the dictionary is cleared at the end of the generation, but unlike in SensGP, it is not immediately reinstated, as this will happen at the beginning of the next generation.

7.3.2 Experiments

In order to compare the ModelDep configurations to SGP and SensGP, we had to determine good values for the ModelDep and RecalculateScore (Recalc) parameters. We chose to test the values of 0, 2 and 5 for ModelDep, as evaluating more than 5% of the population for feature importance would allow mediocre models into the selection (in case of 512 population size, already 25 models are inspected). For the RecalculateScore parameter, we tested the values of 1,2,5,10 and 20, although the values 1 and 2 only for the ModelDep 0 setting, for which we did not evaluate the RecalculateScore equal to 20 setting. We also

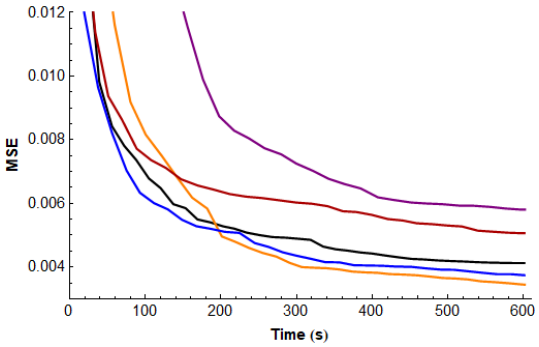
evaluated the algorithms without recalculation. The median results of running 10 iterations of each of these tests, on the Microsoft Azure VMs, are shown in Figures 21(a-f), for both train and test MSE, on the next page.

We observe that the ModelDep 5 setting produces models with worse average performance than the other settings did. Therefore, we disregard this configuration in choosing good parameters for further testing.

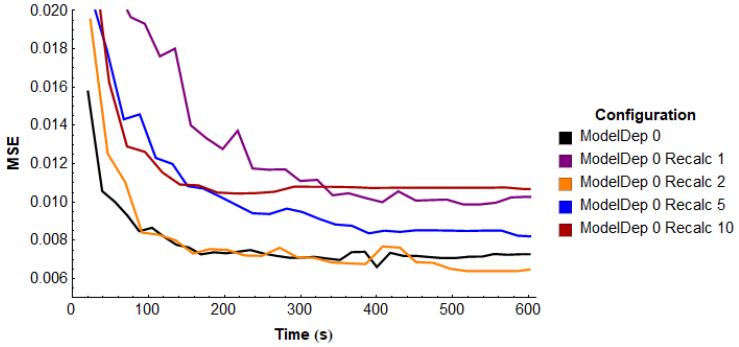
For Modeldep 0, Recalc 1 and 10 clearly perform worse than the other settings. The other three configurations are all quite good, but since ModelDep 0 Recalc 2 performs the best on both the training and the test set, we chose to use it in future experiments.

In the case of ModelDep 2, we observe that the only value which is noticeably performing worse in this test is that of Recalc 5. As the other settings are close together and as we are more interested in measuring the effect of a larger Recalc value than in measuring the effect for another low value (which we picked for ModelDep 0), we chose to continue with Recalc 20 in further experiments. Additionally, we wanted to test a variant of ModelDep without recalculation, for which we used ModelDep 2.

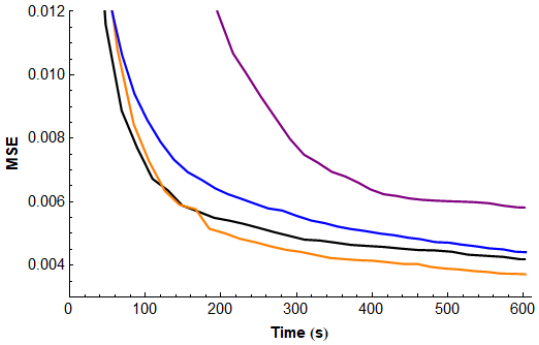
A comparison of the Model Dependent variant to the other techniques will be shown in Section 7.5.



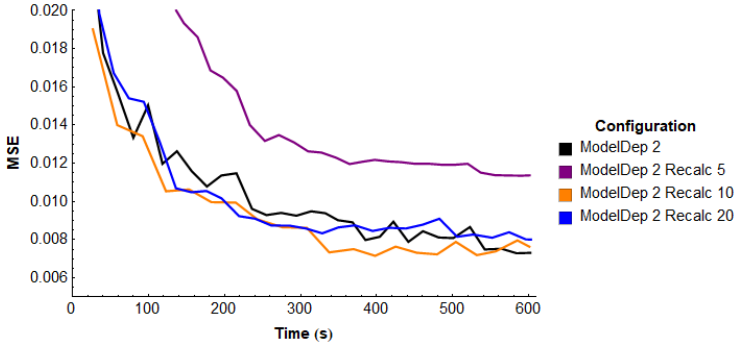
(a) ModelDep 0; Training Set.



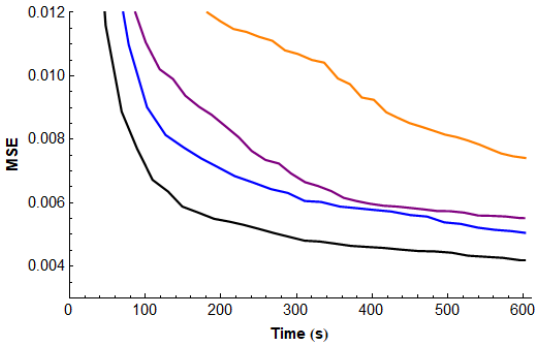
(b) ModelDep 0; Test Set.



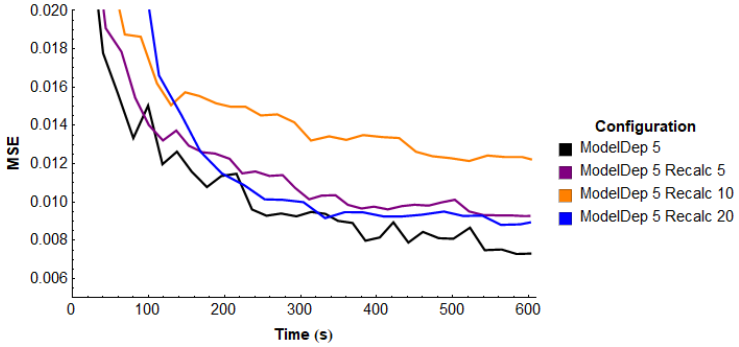
(c) ModelDep 2; Training Set.



(d) ModelDep 2; Test Set.



(e) ModelDep 5; Training Set.



(f) ModelDep 5; Test Set.

Figure 21: Mean Squared Error (MSE) versus Time (seconds) for different configurations of Model Dependent SensGP.

7.4 Variable Importance

The second variant of SensGP, named Variable Importance (VarImp), is most like the original idea presented by Grant Dick in his paper Sensitivity-like Analysis for Feature Selection in Genetic Programming [10]. The method presents one way of bringing the analysis from its original version of looking at individual variables to incorporating features of 2nd and higher order as well. In this variant, all nodes in a tree will be traversed and each subtree is scored depending on the variables which occur in it. The importance score of the feature will be equal to that of the entire model, but with the variables to shuffle equal to the variables present in the feature.

For example, we take the model to be:

$$\frac{(x_1 + 0.3) \cdot x_1}{x_2 - x_3}. \tag{24}$$

If the importance of the feature $(x_1 + 0.3)$ is to be calculated, first the MSE of the entire solution is calculated. Then the value of the entire solution is evaluated on the dataset with all values for x_1 shuffled. These two MSE values are then divided by each other as usual to obtain the importance score. Consequently, the feature $((x_1 + 0.3) * x_1)$ and x_1, which also occur in the model, will obtain the same importance value as $(x_1 + 0.3)$. If these features are later found in other models, the feature score of these features will be set to the best score encountered so far. This implementation favours smaller features, as they appear in more models.

As in the Model Dependent approach, additional effort is put into examining the RecalculateScore (Recalc) parameter, since we expect new models to overtake the old models as the population evolves, and getting rid of the features that appeared in previously good models might keep the search going in a fresh direction.

7.4.1 Algorithmic Details

A couple of adjustments to SensGP have to be made to make this approach work. In large part, the Model Dependent variant is followed. The initialization is identical, and in the generational step, the algorithm is identical except for one method.

Instead of calling AfterPopulationGenerationImportance(), an alternative function is used, named EqualVariableImportance(). Again, the percentage of the population to be reviewed is required as a parameter, calculated by multiplication of the VarImp parameter by the population size. The EqualVariableImportance() operation is identical to AfterPopulationGenerationImportance(), in that it analyses for each tree to be inspected which subtrees contain variable nodes, which are then inspected for feature importance. The VarImp approach differs from ModelDep in that after evaluation of every individual in the original population, the CalculateVariableImportance() method is called, instead of at the end when all trees in the population have been processed. The reason for this is that the model from which a feature originated needs to be known to be able to apply this variant of importance calculation.

The CalculateVariableImportance() method functions globally like Algorithm 2, with a couple of minor differences. On line 7 of the algorithm, the unshuffled MSE is set equal to the MSE of the original model, from which all of the features to be inspected originate. On line 8, the shuffled MSE is calculated from the original model as well, but with the variables to shuffle equal to those of the feature. Also, while in the original version of the algorithm every feature is evaluated, in this variant a feature is checked for whether its variable combination has already been evaluated in the current model, saving computation of features containing the same variables.

When all features have been inspected, the variables are inserted into the dictionary. Features of the same variables will get an importance score which is incremented by the smallest possible amount, as in SensGP. If the feature is already present in the dictionary due to evaluation with respect to another model, it is checked if the current importance is an improvement and if so, the feature score is updated in the dictionary.

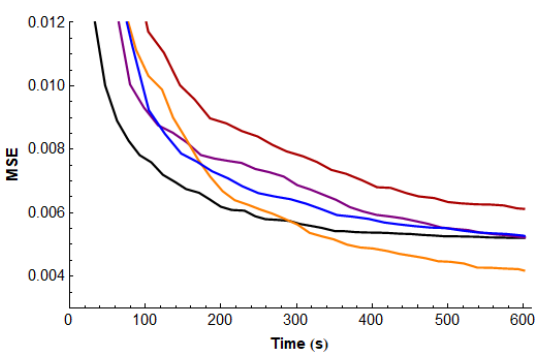
7.4.2 Experiments

Similar to the Model Dependent approach, we had to determine good values for the VarImp and RecalculateScore parameters. We chose to test the same values of 0, 2 and 5 for VarImp, as evaluating importance for more than 5% of the population would likely lead to the inclusion of models which are not much better than average. For the RecalculateScore parameter, we tested the values of 1,2,5,10 and 20, although the values 1 and 2 only for the VarImp 0 setting, for which we did not evaluate the RecalculateScore equal to 20 setting. We also evaluated the algorithms without recalculation. The median results of running 10 iterations of each of these tests on the Microsoft Azure VMs are shown in Figures 22(a-f), for both train and test MSE, on the next page.

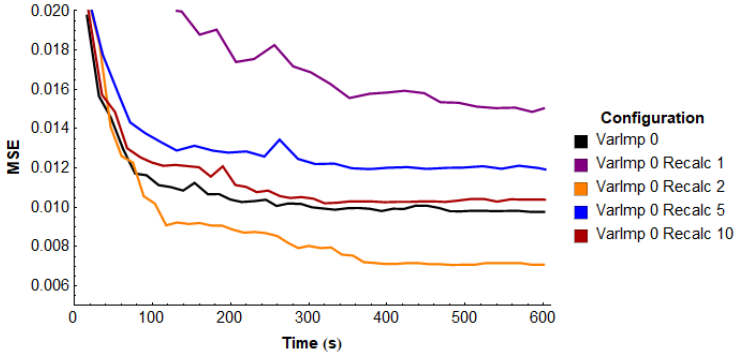
We observe that the VarImp 5 setting performs overall worse than the other settings, although for Recalc 5 and 20 the performance is still quite competitive. Since the other two VarImp settings prove more useful, however, we disregard the VarImp 5 setting in our further experiments.

When looking at VarImp 0 and 2, we observe that for both the training and testing focussed median runs, Varimp 0 Recalc 2 and VarImp 2 Recalc 20 outperform the rest. These settings also performed well in the ModelDep case and as this symmetry makes for an interesting comparison of the techniques, we decided to select these configurations as those which will be used for future experimentation. Using VarImp without recalculation did not seem to result in good models, so we do not use it in further testing.

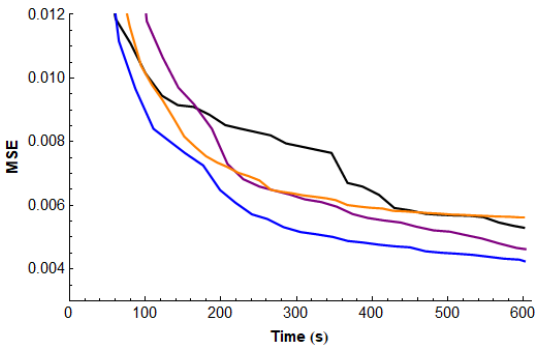
The performance of the Variable Importance variant in comparison to the other techniques will be discussed in Section 7.5.



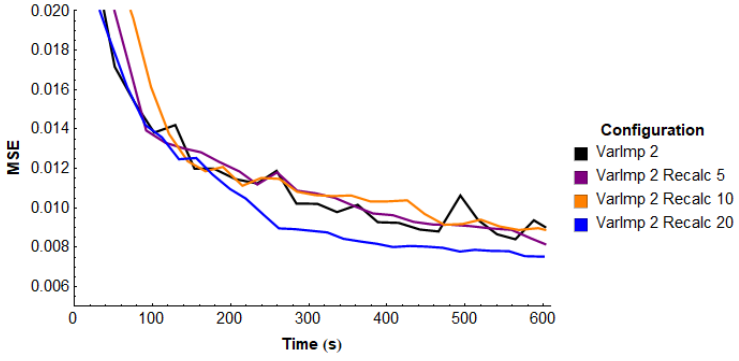
(a) VarImp 0; Training set.



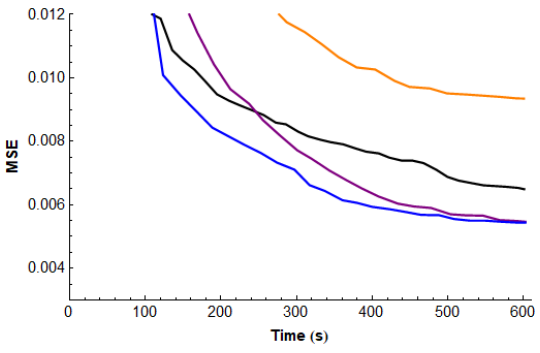
(b) VarImp 0; Test set.



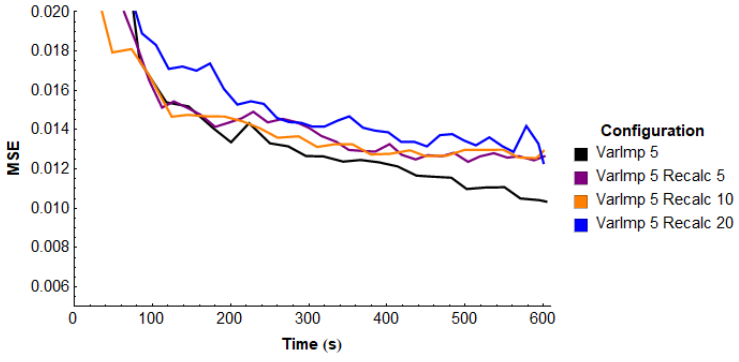
(c) VarImp 2; Training set.



(d) VarImp 2; Test set.



(e) VarImp 5; Training set.



(f) VarImp 5; Test set.

Figure 22: Mean Squared Error (MSE) versus Time (seconds) for different configurations of Variable Importance SensGP.

7.5 Overall Comparison

In this section, we present the experimental results of the comparison between all of the algorithms discussed in this section to SGP. The following configurations are compared, as discussed in the preceding parts of this section: SGP, SenGP, ModelDep 2, ModelDep 0 Recalc 2, ModelDep 2 Recalc 20, VarImp 0 Recalc 2 and VarImp 2 Recalc 20. In the remainder of this Section, each method is identified by their unique color in the figures, which is depicted in Figure 23

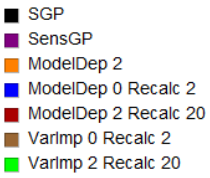


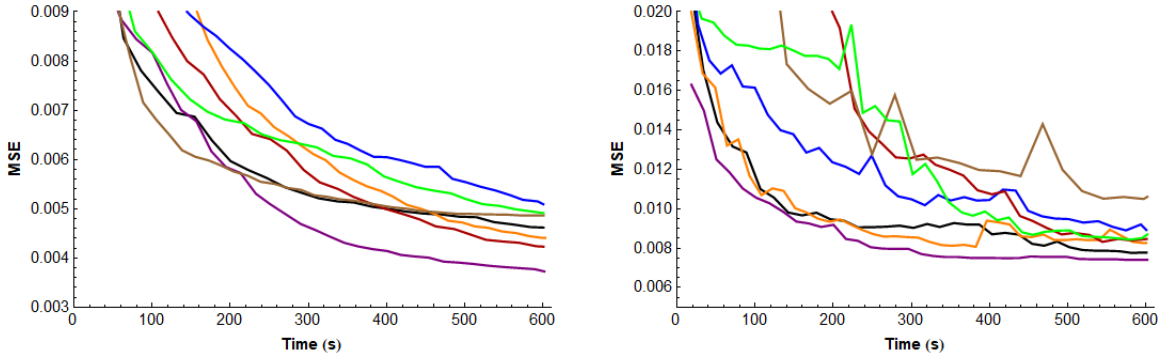
Figure 23: Legend.

We tested these algorithms on the datasets discussed in Section 5.3: Vladislavleva-4, Pagie-1, Korns-12, Dow Chemical, Red Wine, Energy Heating, Keijzer-1 and Nguyen-7. Each configuration was run 15 times on the Asus laptop, with the exception of Pagie-1 and Korns-12, for which the Microsoft Azure VMs were used. The median results of all of these tests are shown in the figures in the remainder of this section. We start by examining the larger datasets, as we believe them to be more challenging and thus more suitable for testing if an algorithm has merit.

For each of these algorithms, the end of run median MSE, mean MSE with standard deviation and p-value with respect to SGP are shown in Tables 13 to 20 in Appendix A. These values are shown for both training and test data and we will refer to them throughout this section.

The first three datasets discussed are those generated artificially based on a mathematical formula.

In Figure 24 as well as Table 13 the results of running all of the algorithms on the Vladislavleva-4 dataset are shown. SensGP has obtained the best median training and test MSE on this dataset. ModelDep 2 and ModelDep 220 also perform better than SGP on the training data, but not on the test data. None of these runs are found to be significantly different to SGP. SensGP has the lowest p-value to SGP, at 0.106 on the training data and 0.229 on the test data.

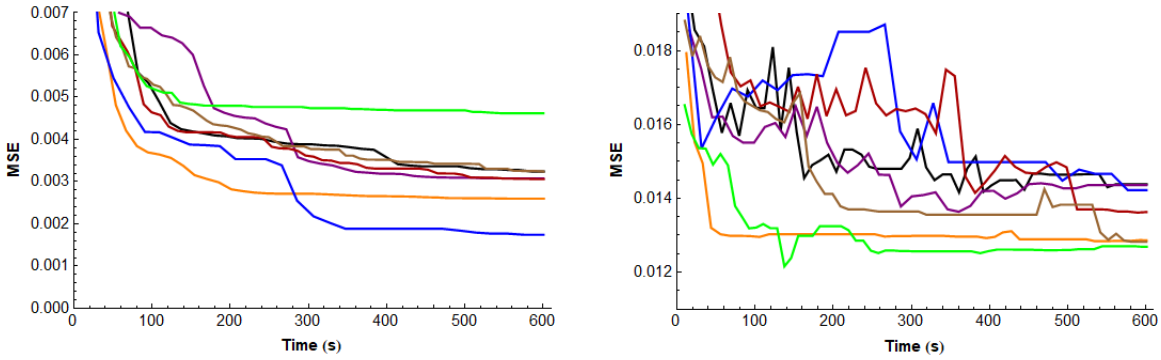


(a) Training set.

(b) Test set.

Figure 24: Mean Squared Error (MSE) versus Time (seconds) on the Vladislavleva-4 dataset for different algorithms. The associated legend is shown in Figure 23.

In Figure 25 and Table 14 the results of running all the algorithms on the Pagie-1 dataset are shown. The results of four of the algorithms are very close to each other on the training data, with ModelDep 2 and ModelDep 002 performing better than these and VarImp 220 performing much worse. On the test set, however, VarImp 220 reaches the best MSE, suggesting the other algorithms are overfitting on the training data. None of the associated p-values compared to SGP are smaller than 0.05, however, and thus they can not be said to be significantly better.



(a) Training set.

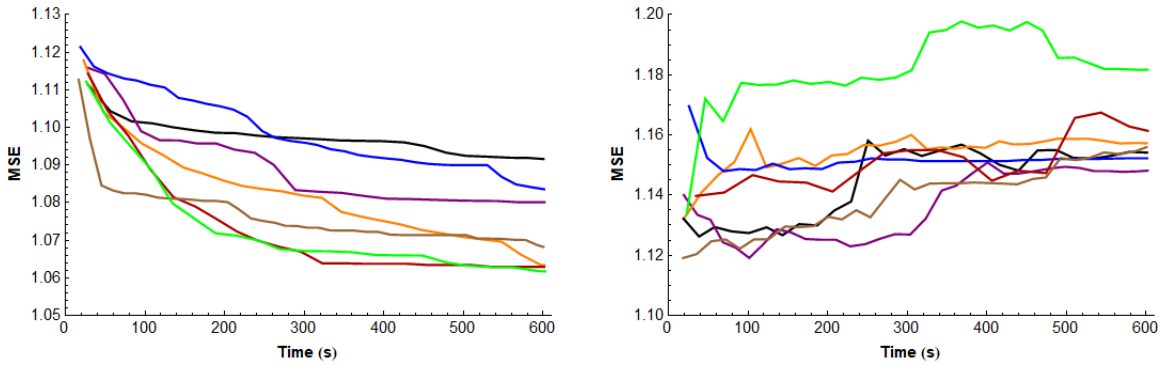
(b) Test set.

Figure 25: Mean Squared Error (MSE) versus Time (seconds) on the Pagie-1 dataset for different algorithms. The associated legend is shown in Figure 23.

In Figure 26 and Table 15 the results of running all of the algorithms on the Korns-12 dataset are shown. Korns-12 turns out to be a vary hard problem, as none of the algorithms seems to do a decent job at finding a good description of the data. Apparently, the underlying formula of the data with its sine and cosine term is too difficult of a problem for these algorithms. We conclude this as a MSE of just over 1 is achieved by all of the algorithms, while typical values for

the dependent value range from 0 to 4. This means the model does not perform much better than taking a constant as the predictive formula would. We describe the results from this test below, but are careful in drawing conclusion about the performance of the algorithms based on this dataset.

We observe that on the training data ModelDep 2, ModelDep 220 and VarImp 220 perform the best, while SGP has the worst performance of all the algorithms. Inspection of the Mann-Whitney U test scores shows us that both ModelDep 2 and VarImp 220 are found to be statistically better than SGP, with a p-value of 0.023 and 0.028 respectively. On the test data however, these these algorithms as well as ModelDep 220 have the worst performance, which indicates the algorithms were overfitting on the training set.



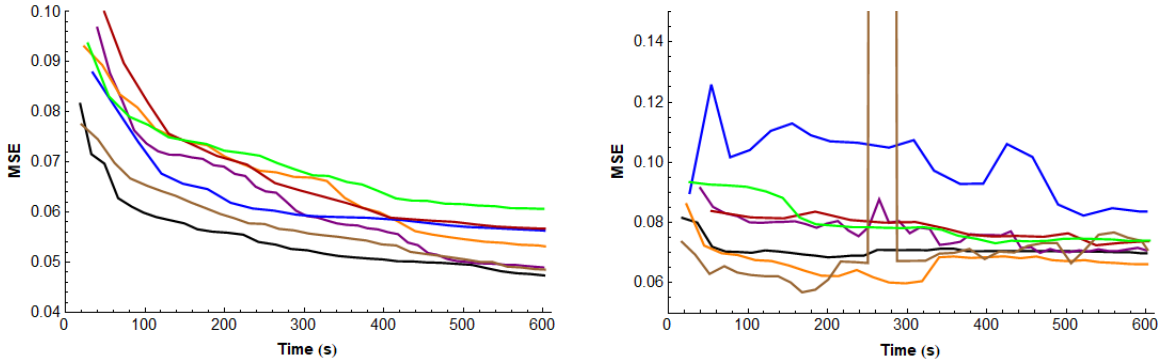
(a) Training set.

(b) Test set.

Figure 26: Mean Squared Error (MSE) versus Time (seconds) on the Korns-12 dataset for different algorithms. The associated legend is shown in Figure 23.

Next, we move to the datasets containing real world data (from a simulation, in the case of Energy Heating).

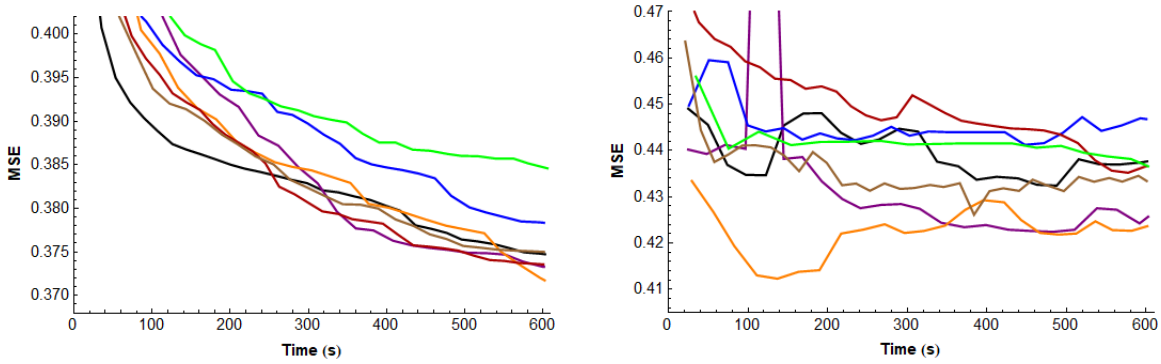
In Figure 27 and Table 16 the results of running all of the algorithms on the Dow Chemical dataset are shown. We observe SGP has the best median performance on the train set here, with SensGP and VarImp 002 very close. On the test data, all algorithms perform comparatively, except for ModelDep 002 which performs worse than the others. On the train set, ModelDep 220 and VarImp 220 are found to perform statistically worse than SGP, with p-values of 0.034 and 0.023 respectively. On the test data, the difference between the algorithms is not statistically significant.



(a) Training set. (b) Test set.

Figure 27: Mean Squared Error (MSE) versus Time (seconds) on the Dow Chemical dataset for different algorithms. The associated legend is shown in Figure 23.

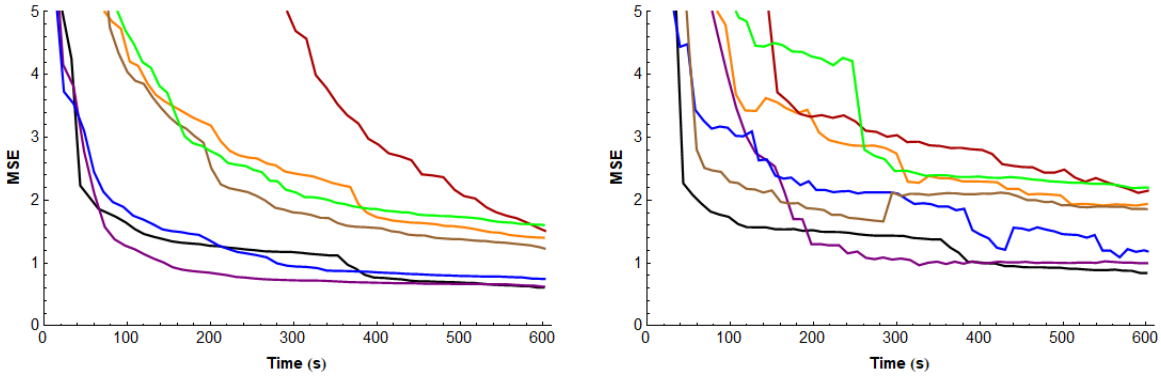
In Figure 28 and Table 17 the results of running all of the algorithms on the Red Wine dataset are shown. The ModelDep 2 Algorithm had the best median run, on both the train and test data. The median MSE was 0.372 and 0.424 respectively. Corresponding p-values are 0.361 and 0.171, however, so these results are not statistically significant. None of the other algorithms differ statistically from SGP either.



(a) Training set. (b) Test set.

Figure 28: Mean Squared Error (MSE) versus Time (seconds) on the Red Wine dataset for different algorithms. The associated legend is shown in Figure 23.

In Figure 29 and Table 18 the results of running all of the algorithms on the Energy Heating dataset are shown. Here we observe a clear separation of algorithms. The algorithms that perform relatively well are SGP, SensGP and ModelDep 002, while ModelDep 2, Modeldep 220, VarImp 002 and VarImp 220 perform much worse. The Mann-Whitney U test confirms these findings, with the p-values of the latter 4 configurations compared to SGP below 0.01. These findings hold for both the train and test data.

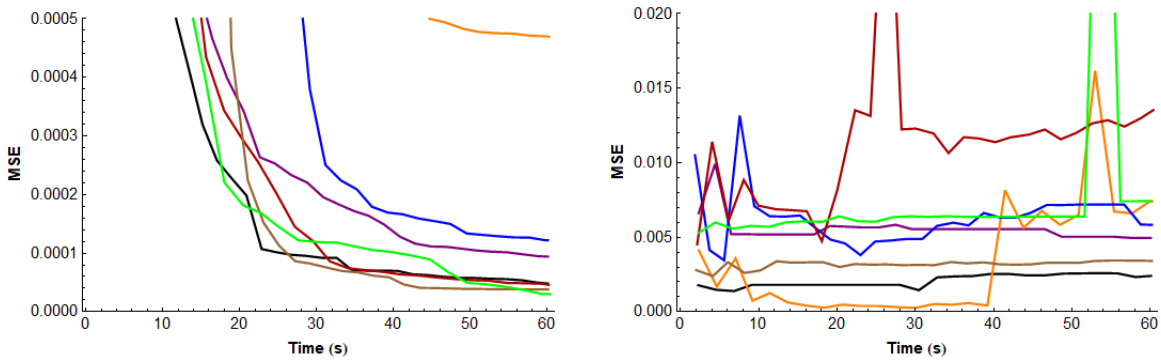


(a) Training set. (b) Test set.

Figure 29: Mean Squared Error (MSE) versus Time (seconds) on the Energy Heating dataset for different algorithms. The associated legend is shown in Figure 23.

Having discussed all of the experiments on the larger datasets, we discuss the smaller datasets in the remainder of this subsection.

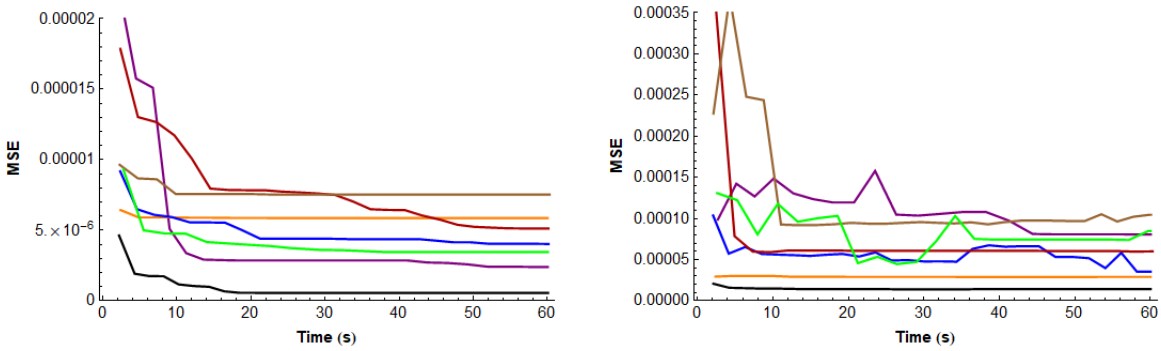
In Figure 30 and Table 19 the results of running all of the algorithms on the Keijzer-1 dataset are shown. As we had seen before in the Section 6, SGP is good at solving this problem, and it is no surprise the algorithm achieves the best test score. Both of the VarImp configurations, as well as ModelDep 220 show better results on the train set, however. Although ModelDep 2 seems to be clearly outperformed on the train data, the p-value when compared to SGP is only 0.089 and the difference is thus not significant. When we look at the mean MSE of ModelDep 2, we observe it to be comparable to the other algorithms. The difference between SGP and ModelDep 2 can thus be explained by the algorithm having a couple of quite good runs, as well as a larger number of mediocre runs, of which the median is one.



(a) Training set. (b) Test set.

Figure 30: Mean Squared Error (MSE) versus Time (seconds) on the Keijzer-1 dataset for different algorithms. The associated legend is shown in Figure 23.

In Figure 31 and Table 20 the results of running all of the algorithms on the Nguyen-7 dataset are shown. SGP clearly outperforms the other algorithms here, on both the train and test data. From the significance test we observe the difference to SGP to be statistically significant for all other algorithms. On the test data, however, none of the results are statistically significant any more, suggesting SGP is overfitting on the train data.



(a) Training set.

(b) Test set.

Figure 31: Mean Squared Error (MSE) versus Time (seconds) on the Nguyen-7 dataset for different algorithms. The associated legend is shown in Figure 23.

7.6 Discussion

After having reviewed the strength of the algorithms on a number of different datasets, we can formulate an answer to the final research question at the heart of this work:

- Can SGP be improved upon by introducing a sensitivity analysis based feature-guided evolution technique, called Sensitivity-based Genetic Programming (SensGP)?

To answer this question, we first briefly describe the results of the experiments with the different configurations. We start with the larger, arguably more important datasets.

On the Vladislavleva-4, Pagie-1 and Red Wine datasets, no algorithm is found to be statistically better than another one, with a different algorithm obtaining the best median MSE on each set: SensGP for Vladislavleva 4 on both train and test data, ModelDep 2 for Red Wine on both train and test data and ModelDep 002 and VarImp 220 on the train and test data of Pagie-1 respectively.

On the Korns-12 dataset, ModelDep 2 and VarImp 220 perform statistically better than SGP on the training set. On the test set SensGP finds the best median MSE, but none of the algorithms are found to be statistically different from SGP. As mentioned in Section 7.5, all of the tested algorithms perform poorly on this dataset, indicating it might be too difficult of a problem to test these algorithms and therefore we do not base our conclusion on this experiment.

On the Dow Chemical dataset, ModelDep 220 and VarImp 220 are found to be statistically worse than SGP on the training data. This is not true when it

comes to the test data, however. The algorithm with the lowest median MSE on the training data is SGP and on the test data it is ModelDep 2.

On the Energy Heating dataset, a large difference in performance is observed. ModelDep 2, ModelDep 220, VarImp 002 and VarImp 220 are found to perform significantly worse than SGP on both train and test data. SGP has the best median MSE on both the train and test data, but SensGP and ModelDep 002 perform almost as well as SGP does and no statistical difference was observed between these three configurations.

In none of these tests, SGP and SensGP were found to be statistically better than one another, while some of the configurations of ModelDep and VarImp were found to be worse than SGP on two of the datasets.

On the smaller Keijzer-1 dataset, no significant differences are found. On the training data VarImp 220 achieved the best median MSE, on the test data this was done by SGP.

The Nguyen-7 dataset is the dataset where we observe the largest difference. All algorithms perform significantly worse than SGP on the training data. On the test data this difference is no longer significant for any of the algorithms, however. This leads us to conclude SGP overfits the data on this particularly small dataset.

A definite statement of whether one of the techniques is better overall can not be made from these experiments. SensGP and ModelDep 002 are only found to perform significantly worse than SGP on the training data of Nguyen-7, a difference which disappears on the test data. Four variants of SensGP are found to be worse than SGP on the Energy Heating Set. ModelDep 220 and VarImp 220 are found to perform worse than SGP on the training part of Dow Chemical as well while ModelDep 2 and VarImp 220 perform better on the training set of Korns-12. These findings are not decisive enough to definitively assert under which circumstances these different algorithms perform the best and suggest further research into approaches based on sensitivity analysis is needed in order to so.

8 Conclusion

The main objective of this work was to attempt to design a novel algorithm which would improve upon Standard Genetic Programming (SGP). From the beginning, we felt the most promising direction to focus our research on had to be a feature-guided evolution approach.

With this focus in mind, we conducted a literature study, where we started examining the problem of Symbolic Regression (SR) and the field of Genetic Programming (GP) from the bottom up. We researched variable selection methods, machine learning techniques and studied a number of specific algorithms in detail: SGP, to be used as our baseline algorithm and frame of reference, Evolutionary Feature Synthesis (EFS), Multiple Regression GP (MRGP), Behavioural Programming, and the GP-Gene-Pool Optimal Mixing Evolutionary Algorithm.

To increase our understanding of these algorithms and eventually use what knowledge we obtained about them to achieve the goal of this thesis, we decided to make a comparison between these promising techniques from the literature. From this comparison, we learned that the guided evolution algorithms EFS and MRGP clearly outperform SGP on a sizeable, widely accepted dataset, Vladislavleva-4, proving the feature-guided principle to be promising direction for improving upon SGP.

During our study of the relevant literature, we encountered an interesting paper, which focused on applying sensitivity analysis to individual variables in SR solutions. Subsequently, an algorithm was presented in which the information gained by this analysis was successfully used to explicitly guide the search in an evolutionary algorithm. Due to the absence of the exploration of sensitivity-based techniques in SR literature and their proven potential as an input variable importance measure, we decided upon designing an algorithm which implements a sensitivity-based feature importance score and using this to guide the evolutionary process.

We created the Sensitivity-based Genetic Programming algorithm (SensGP), as well as two variants of SensGP, named Model Dependent SensGP (ModelDep) and Variable Importance SensGP (VarImp). These algorithms were compared to each other as well as SGP, on a number of different datasets to measure how well the approaches were able to describe the data.

The algorithms had comparable performance on the datasets used in our experiments. The six configurations which were tested resulted in models which were often comparable to SGP in model quality. For none of the large datasets, SensGP and one of the configurations of ModelDep were found to result in models of significantly different quality than SGP did. The four other configurations of ModelDep and VarImp were all found to perform worse on one of the large datasets, Energy Heating, with two of them performing worse on another large dataset as well, but only on the training data. On four of the five relevant large datasets, improvements upon SGP were found by one or more of these configurations on the test data. None of these improvements were found to be statistically significant, however.

We conclude that using a sensitivity-based feature importance measure can be used to obtain models comparable in strength to those generated by SGP. The potential of SensGP has not been fully explored, however, and in Section 9 we suggest a number of possible improvements, which might further increase the performance of SensGP.

9 Future Work

During the design and testing of the Sensitivity-based approach, a number of topics suitable for further research have come to mind. The most interesting are listed below, along with a description of varying detail, including possible approaches and/or difficulties.

9.1 Commutative Filtering

The first research topic is a possible optimization to the dictionary. It has occurred to us that some features, although syntactically different, are semantically identical. A simple example of this, for instance, would be: $x_1 \cdot x_2$ and $x_2 \cdot x_1$. Currently, both of these features will be stored in the dictionary, with an importance which will be approximately equal (given a sufficiently large dataset). This effectively doubles the chance of this semantically identical feature being introduced into the population.

This (possible) issue is present for every feature which contains one or more commutative operations such as addition and multiplication. Since these operators occur in almost any tree of larger size, a lot of semantically identical features could be stored in the dictionary at any time. Any feature can be present up to 2^k times in different syntactic forms, where k is the number of commutative operator nodes in the tree. This could significantly raise the probability of a semantically identical feature being picked.

Therefore, it is worth researching what the frequency of semantically identical features occurring in the dictionary is. If it is found that too many semantically identical trees are present, it could be beneficial to implement a technique to combat this.

If the dataset that is used is large enough, one heuristic method is to search in the dictionary for features having similar importance and checking them explicitly. This allows for a tunable accuracy of finding semantically identical features versus the computational cost of doing so, depending on how many neighbours of a feature in the dictionary are checked.

Another method would be to explicitly check for a given tree if any of the trees which are obtained by changing the order of the child nodes of any commutative operator node, are already contained in the dictionary. The cost of this method depends on the number of commutative operator nodes k contained in the tree and grows rapidly, as 2^k, as discussed before. A mixture of these techniques is possible as well, where trees with few (commutative) nodes are checked explicitly and the others are checked heuristically.

9.2 Constants

The proper inclusion of constants into SensGP is an area where there is a lot of potential for improvement, as is explained in Section 7.2.3 under Constants. Finding a suitable way to include these constants into the dictionary and using them later on might prove to be a significant improvement of the method as a whole.

A possible route to achieving this would be to save the features as $C \cdot x_1$, setting their importance equal to the best importance score encountered so far by the different values of C which have been tested. Then, when inserted into a tree,

their value should be optimized for that specific tree by OLS. If an improvement is found, its importance should be updated in the dictionary.

9.3 Importance Analysis

One of the interesting subjects for further study would be to focus on the exact value of the importance score. We identify two situations which are particularly interesting.

Importance Equals One

Experiments showed that the importance scores of features are frequently exactly equal to 1. Further investigation led to the conclusion that this happened when either the feature contains no variables at all and thus a shuffling of the dataset has no impact, or when the feature contains only variables of which the effect is nullified. This happens when a self-subtraction (resulting in 0) or self-division (resulting in 1) is present in a tree. An individual which does contain variables is then still not affected by the shuffling of the data corresponding to these variables and is effectively a constant as well.

Currently, features with importance equal to 1 are simply not inserted into the dictionary. An alternative option might be to replace these features in the trees they occur in as well. Either a newly constructed feature could be put in their place, or the subtree could be be replaced by a new randomly instantiated constant node.

Exceptionally Large Importance

While any particularly large importance value most likely means somewhere in the shuffled data a combination of variables has occurred which leads to e.g. a division by some number close to 0, in some cases these values might provide valuable information.

SensGP relies on measuring feature importance by calculating how much better a feature predicts on the original data than if it would have been given random data for its variables. A valuable feature is expected to perform much worse on the shuffled data. The reverse might be true, however, for the inverse of this valuable feature. The inverse of a feature might actually perform a lot better on the shuffled data then on the unshuffled set. Therefore, it could be interesting to see if by inverting a feature with a large importance value, a good feature can sometimes be obtained.

9.4 Weighting Importance by MSE

Like in the complexity based approach described in Section 7.2.3, feature importance might be weighted in another fashion before the features are inserted into the dictionary. In basic SensGP, all feature importance is calculated without taking into account the model quality. Experimentation with other importance measures could be performed, where feature importance is weighted by the MSE of the model it occurs in.

9.5 Deterministic Mutation Location Selection

A radically different approach is to use the results from the sensitivity analysis of the individuals in the population to determine which parts of an individual should be replaced during mutation. We experimented briefly with this approach.

For each individual that was picked for mutation, the feature score of all relevant (i.e. containing a variable) subsets were calculated as in Variable Importance SensGP and the feature with the highest importance score was replaced by a randomly generated new feature. We tried accepting this mutation greedily or after a set number of iterations.

While we believe this approach has a lot of potential, there are a number of issues which have to be resolved. Firstly, naively analysing every relevant feature in a model takes too much time in practise, as these calculations need to be performed in a model-dependent manner, and are therefore not saved in the dictionary. This results in the method spending much more time evaluating importance than in SensGP, where feature importance of already encountered features can be looked up in the dictionary. Furthermore, randomly generating new trees to insert at the selected location, until an improvement in model MSE had been found, proved to take more time than we had hoped. The algorithm could get stuck for a long time at the random generation of new features, while no increase in model performance was found.

References

[1] Ignacio Arnaldo, Krzysztof Krawiec, and Una-May O'Reilly. Multiple regression genetic programming. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 879–886. ACM, 2014.

[2] Ignacio Arnaldo, Una-May O'Reilly, and Kalyan Veeramachaneni. Building predictive models via feature synthesis. In *Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation*, pages 983–990. ACM, 2015.

[3] ALFA at CSAIL MIT. The flexgp project. [Accessed May 12th 2018]. URL: `http://flexgp.csail.mit.edu/index.php`.

[4] Wolfgang Banzhaf, Peter Nordin, Robert E Keller, and Frank D Francone. *Genetic programming: an introduction*, volume 1.

[5] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[6] Qi Chen, Mengjie Zhang, and Bing Xue. Genetic programming with embedded feature construction for high-dimensional symbolic regression. In *Proceedings of Intelligent and Evolutionary Systems: The 20th Asia Pacific Symposium*, pages 87–102. Springer, 2017.

[7] Paulo Cortez, António Cerdeira, Fernando Almeida, Telmo Matos, and José Reis. Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems*, 47(4):547–553, 2009.

[8] Nichael Lynn Cramer. A representation for the adaptive generation of simple sequential programs. In *Proceedings of the first international conference on genetic algorithms and their applications*, pages 183–187. Carnegie-Mellon University, 1985.

[9] Vinícius Veloso De Melo. Kaizen programming. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 895–902. ACM, 2014.

[10] Grant Dick. Sensitivity-like analysis for feature selection in genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 401–408. ACM, 2017.

[11] Renáta Dubčáková. Eureqa: software review. *Genetic programming and evolvable machines*, 12(2):173–178, 2011.

[12] Bradley Efron, Trevor Hastie, Iain Johnstone, Robert Tibshirani, et al. Least angle regression. *The Annals of statistics*, 32(2):407–499, 2004.

[13] Richard Forsyth. BEAGLE - a Darwinian approach to pattern recognition. *Kybernetes*, 10(3):159–166, 1981.

[14] Jerome Friedman, Trevor Hastie, and Rob Tibshirani. Regularization paths for generalized linear models via coordinate descent. *Journal of statistical software*, 33(1):1, 2010.

[15] Luca Gherardi, Davide Brugali, and Daniele Comotti. A java vs. c++ performance evaluation: a 3d modeling benchmark. In *Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots*, pages 161–172. Springer, 2012.

[16] Isabelle Guyon and André Elisseeff. An introduction to variable and feature selection. *Journal of machine learning research*, 3(Mar):1157–1182, 2003.

[17] Robin Harper. Spatial co-evolution: quicker, fitter and less bloated. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 759–766. ACM, 2012.

[18] Arthur E Hoerl and Robert W Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[19] Ilknur Icke and Joshua C Bongard. Improving genetic programming based symbolic regression using deterministic machine learning. In *Proceedings of the IEEE Congress on Evolutionary Computation*, pages 1763–1770. IEEE, 2013.

[20] Christian Igel and Kumar Chellapilla. Investigating the influence of depth and degree of genotypic change on fitness in genetic programming. In *Proceedings of the 1st Annual Conference on Genetic and Evolutionary Computation-Volume 2*, pages 1061–1068. Morgan Kaufmann Publishers Inc., 1999.

[21] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[22] Maarten Keijzer. Improving symbolic regression with interval arithmetic and linear scaling. In *Proceedings of the European Conference on Genetic Programming*, pages 70–82. Springer, 2003.

[23] Maarten Keijzer and Vladan Babovic. Genetic programming, ensemble methods and the bias/variance tradeoff–introductory investigations. In *Proceedings of the European Conference on Genetic Programming*, pages 76–90. Springer, 2000.

[24] Ron Kohavi and George H John. Wrappers for feature subset selection. *Artificial intelligence*, 97(1-2):273–324, 1997.

[25] Michael F Korns. Accuracy in symbolic regression. In *Genetic Programming Theory and Practice IX*, pages 129–151. Springer, 2011.

[26] John R Koza. *Genetic programming: on the programming of computers by means of natural selection*, volume 1. MIT press, 1992.

[27] Krzysztof Krawiec and Una-May O'Reilly. Behavioral programming: a broader and more detailed take on semantic GP. In *Proceedings of the 2014 Annual Conference on Genetic and Evolutionary Computation*, pages 935–942. ACM, 2014.

[28] M. Lichman. Uci mach. learning repository. [Accessed May 25th 2018], 2015. URL: `http://archive.ics.uci.edu/ml`.

[29] Wei-Yin Loh. Classification and regression trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(1):14–23, 2011.

[30] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[31] Trent McConaghy. FFX: Fast, scalable, deterministic symbolic regression technology. In *Genetic Programming Theory and Practice IX*, pages 235–260. Springer, 2011.

[32] James McDermott, David R White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, et al. Genetic programming needs better benchmarks. In *Proceedings of the 14th annual conference on Genetic and Evolutionary Computation*, pages 791–798. ACM, 2012.

[33] Microsoft. Microsoft azure. [Accessed May 12th 2018]. URL: `http://azure.microsoft.com`.

[34] Ludo Pagie and Paulien Hogeweg. Evolutionary consequences of coevolving targets. *Evolutionary computation*, 5(4):401–418, 1997.

[35] Tavish Srivastava. Differences between machine learning and statistical modeling. [Accessed May 11th 2018]. URL: `https://www.analyticsvidhya.com/blog/2015/07/difference-machine-learning-statistical-modeling`.

[36] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 267–288, 1996.

[37] Athanasios Tsanas and Angeliki Xifara. Accurate quantitative estimation of energy performance of residential buildings using statistical machine learning tools. *Energy and Buildings*, 49:560–567, 2012.

[38] Nguyen Quang Uy, Nguyen Xuan Hoai, Michael O'Neill, Robert I McKay, and Edgar Galván-López. Semantically-based crossover in genetic programming: application to real-valued symbolic regression. *Genetic Programming and Evolvable Machines*, 12(2):91–119, 2011.

[39] Marco Virgolin, Tanja Alderliesten, Arjan Bel, Cees Witteveen, and Peter AN Bosman. Symbolic regression and feature construction with GP-GOMEA applied to radiotherapy dose reconstruction of childhood cancer survivors. In *Proceedings of GECCO '18: Genetic and Evolutionary Computation Conference*. ACM, 2018.

[40] Marco Virgolin, Tanja Alderliesten, Cees Witteveen, and Peter AN Bosman. Scalable genetic programming by gene-pool optimal mixing and input-space entropy-based building-block learning. In *Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1041–1048. ACM, 2017.

[41] Ekaterina Vladislavleva et al. *Model-based problem solving through symbolic regression via Pareto genetic programming*. CentER, Tilburg University, 2008.

[42] Ekaterina Vladislavleva, Guido Smits, and Dick Den Hertog. Order of nonlinearity as a complexity measure for models generated by symbolic regression via Pareto genetic programming. *IEEE Transactions on Evolutionary Computation*, 13(2):333–349, 2009.

[43] Matthew Walker. Introduction to genetic programming. *Technical paper: University of Montana*, 2001.

[44] Franklin Herbert Westervelt. *A study of automatic system simulation programming and the analysis of the behavior of physical systems using an internally stored program computer*. PhD thesis, University of Michigan, 1960.

[45] David R White, James Mcdermott, Mauro Castelli, Luca Manzoni, Brian W Goldman, Gabriel Kronberger, Wojciech Jaśkowski, Una-May O'Reilly, and Sean Luke. Better GP benchmarks: community survey results and proposals. *Genetic Programming and Evolvable Machines*, 14(1):3–29, 2013.

[46] Hui Zou and Trevor Hastie. Regularization and variable selection via the elastic net. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 67(2):301–320, 2005.

A Tables Containing the Overall Comparison Results

Table 13: Comparison of SR algorithms on the Vladislavleva-4 dataset.

Training	Median MSE	Mean MSE	Std. dev.	p-value to SGP
SGP	0.00462	0.00846	0.0086	0.983
SensGP	0.00373	0.00423	0.0013	0.106
ModelDep 2	0.00441	0.00549	0.00311	0.34
ModelDep 002	0.00509	0.00578	0.00345	0.868
ModelDep 220	0.00423	0.00497	0.00201	0.481
VarImp 002	0.00487	0.00546	0.00276	0.678
VarImp 220	0.00491	0.00579	0.00317	0.74
Testing	Median MSE	Mean MSE	Std. dev.	p-value to SGP
SGP	0.00777	0.0118	0.00922	0.983
SensGP	0.00741	0.00767	0.00325	0.229
ModelDep 2	0.00826	0.00932	0.00435	0.868
ModelDep 002	0.00893	0.00955	0.00405	0.709
ModelDep 220	0.00847	0.00901	0.00485	0.619
VarImp 002	0.0106	0.0419	0.114	0.507
VarImp 220	0.0087	0.0109	0.00704	0.534

Table 14: Comparison of SR algorithms on the Pagie-1 dataset.

Training	Median MSE	Mean MSE	Std. dev.	p-value to SGP
SGP	0.00324	0.00364	0.00188	0.983
SensGP	0.00308	0.00401	0.00273	0.934
ModelDep 2	0.0026	0.00343	0.00261	0.431
ModelDep 002	0.00174	0.00289	0.00249	0.184
ModelDep 220	0.00306	0.00483	0.00474	0.967
VarImp 002	0.00324	0.00429	0.00295	0.934
VarImp 220	0.00462	0.00435	0.0021	0.561
Testing	Median MSE	Mean MSE	Std. dev.	p-value to SGP
SGP	0.0144	0.0155	0.00633	0.983
SensGP	0.0144	0.0143	0.00383	0.74
ModelDep 2	0.0129	0.0564	0.141	0.772
ModelDep 002	0.0142	0.0202	0.023	1.00
ModelDep 220	0.0136	0.0172	0.0147	0.59
VarImp 002	0.0128	0.0496	0.128	0.803
VarImp 220	0.0127	0.159	0.477	0.868

Table 15: Comparison of SR algorithms on the Korns-12 dataset.

Training	Median MSE	Mean MSE	Std. dev.	p-value to SGP
SGP	1.092	1.082	0.027	0.983
SensGP	1.080	1.079	0.026	0.740
ModelDep 2	1.063	1.062	0.022	0.023
ModelDep 002	1.084	1.074	0.034	0.561
ModelDep 220	1.063	1.071	0.032	0.407
VarImp 002	.068	1.062	0.040	0.281
VarImp 220	1.062	1.059	0.028	0.028
Testing	Median MSE	Mean MSE	Std. dev.	p-value to SGP
SGP	1.154	1.167	0.047	0.983
SensGP	1.148	1.151	0.017	0.590
ModelDep 2	1.157	1.168	0.034	0.507
ModelDep 002	1.152	16.280	56.470	0.648
ModelDep 220	1.161	1.165	0.024	0.740
VarImp 002	1.156	1.172	0.036	0.507
VarImp 220	1.182	1.197	0.070	0.074

Table 16: Comparison of SR algorithms on the Dow Chemical dataset.

Training	Median MSE	Mean MSE	Std. dev.	p-value to SGP
SGP	0.047	0.052	0.0079	0.98
SensGP	0.049	0.051	0.011	0.71
ModelDep 2	0.053	0.057	0.013	0.26
ModelDep 002	0.056	0.055	0.013	0.26
ModelDep 220	0.057	0.058	0.0097	0.034
VarImp 002	0.049	0.049	0.0084	0.48
VarImp 220	0.061	0.06	0.0085	0.023
Testing	Median MSE	Mean MSE	Std. dev.	p-value to SGP
SGP	0.07	3.80	14.0	0.98
SensGP	0.071	0.082	0.037	0.74
ModelDep 2	0.066	0.072	0.015	0.71
ModelDep 002	0.084	0.37	1.00	0.16
ModelDep 220	0.074	0.078	0.019	0.53
VarImp 002	0.071	0.089	0.075	0.74
VarImp 220	0.074	0.092	0.046	0.48

Table 17: Comparison of SR algorithms on the Red Wine dataset.

Training	Median MSE	Mean MSE	Std. dev.	p-value to SGP
SGP	0.375	0.379	0.014	0.983
SensGP	0.373	0.375	0.011	0.320
ModelDep 2	0.372	0.374	0.011	0.361
ModelDep 002	0.378	0.378	0.009	0.934
ModelDep 220	0.374	0.375	0.011	0.431
VarImp 002	0.375	0.375	0.010	0.590
VarImp 220	0.385	0.387	0.008	0.106
Testing	Median MSE	Mean MSE	Std. dev.	p-value to SGP
SGP	0.438	0.442	0.035	0.983
SensGP	0.426	0.457	0.108	0.361
ModelDep 2	0.424	0.476	0.188	0.171
ModelDep 002	0.447	0.492	0.140	0.648
ModelDep 220	0.437	1.090	2.430	0.868
VarImp 002	0.433	0.436	0.016	0.836
VarImp 220	0.437	0.441	0.021	0.934

Table 18: Comparison of SR algorithms on the Energy Heating dataset.

Training	Median MSE	Mean MSE	Std. dev.	p-value to SGP
SGP	0.610	0.713	0.413	0.983
SensGP	0.624	0.792	0.404	0.431
ModelDep 2	1.405	1.401	0.741	0.005
ModelDep 002	0.749	1.002	0.617	0.300
ModelDep 220	1.514	2.096	1.825	0.004
VarImp 002	1.232	1.512	0.933	0.002
VarImp 220	1.605	1.846	1.056	0.000
Testing	Median MSE	Mean MSE	Std. dev.	p-value to SGP
SGP	0.841	1.069	0.567	0.983
SensGP	1.000	1.032	0.460	0.772
ModelDep 2	1.941	2.219	1.379	0.008
ModelDep 002	1.188	1.547	1.057	0.340
ModelDep 220	2.152	2.894	2.156	0.003
VarImp 002	1.860	2.144	1.252	0.009
VarImp 220	2.203	2.782	1.804	0.000

Table 19: Comparison of SR algorithms on the Keijzer-1 dataset.

Training	Median MSE	Mean MSE	Std. dev.	p-value to SGP
SGP	$4.83 \cdot 10^{-5}$	$4.03 \cdot 10^{-4}$	$6.42 \cdot 10^{-4}$	$9.83 \cdot 10^{-1}$
SensGP	$9.41 \cdot 10^{-5}$	$6.43 \cdot 10^{-4}$	$7.52 \cdot 10^{-4}$	$2.13 \cdot 10^{-1}$
ModelDep 2	$4.69 \cdot 10^{-4}$	$5.43 \cdot 10^{-4}$	$5.6 \cdot 10^{-4}$	$8.9 \cdot 10^{-2}$
ModelDep 002	$1.22 \cdot 10^{-4}$	$3.29 \cdot 10^{-4}$	$5.05 \cdot 10^{-4}$	$6.78 \cdot 10^{-1}$
ModelDep 220	$4.59 \cdot 10^{-5}$	$2.59 \cdot 10^{-4}$	$4.14 \cdot 10^{-4}$	$5.61 \cdot 10^{-1}$
VarImp 002	$3.81 \cdot 10^{-5}$	$4.69 \cdot 10^{-4}$	$6.62 \cdot 10^{-4}$	$4.31 \cdot 10^{-1}$
VarImp 220	$3.00 \cdot 10^{-5}$	$3.71 \cdot 10^{-4}$	$5.92 \cdot 10^{-4}$	$5.34 \cdot 10^{-1}$
Testing	Median MSE	Mean MSE	Std. dev.	p-value to SGP
SGP	$2.41 \cdot 10^{-3}$	$4.19 \cdot 10^{-2}$	$1.02 \cdot 10^{-1}$	$9.83 \cdot 10^{-1}$
SensGP	$4.96 \cdot 10^{-3}$	$1.93 \cdot 10^{24}$	$7.22 \cdot 10^{24}$	$4.81 \cdot 10^{-1}$
ModelDep 2	$7.42 \cdot 10^{-3}$	2.13	7.16	$2.45 \cdot 10^{-1}$
ModelDep 002	$5.84 \cdot 10^{-3}$	$5.69 \cdot 10^{1}$	$2.13 \cdot 10^{2}$	$7.72 \cdot 10^{-1}$
ModelDep 220	$1.35 \cdot 10^{-2}$	$4.42 \cdot 10^{1}$	$1.29 \cdot 10^{2}$	$3.4 \cdot 10^{-1}$
VarImp 002	$3.42 \cdot 10^{-3}$	$3.34 \cdot 10^{1}$	$1.24 \cdot 10^{2}$	$9.01 \cdot 10^{-1}$
VarImp 220	$7.43 \cdot 10^{-3}$	$7.25 \cdot 10^{-2}$	$1.45 \cdot 10^{-1}$	$3.4 \cdot 10^{-1}$

Table 20: Comparison of SR algorithms on the Nguyen-7 dataset.

Training	Median MSE	Mean MSE	Std. dev.	p-value to SGP
SGP	$5.55 \cdot 10^{-7}$	$8.93 \cdot 10^{-7}$	$1.26 \cdot 10^{-6}$	$9.83 \cdot 10^{-1}$
SensGP	$2.39 \cdot 10^{-6}$	$7.42 \cdot 10^{-6}$	$1.45 \cdot 10^{-5}$	$6.19 \cdot 10^{-3}$
ModelDep 2	$5.86 \cdot 10^{-6}$	$5.58 \cdot 10^{-6}$	$4.48 \cdot 10^{-6}$	$2.23 \cdot 10^{-4}$
ModelDep 002	$4.03 \cdot 10^{-6}$	$1.75 \cdot 10^{-5}$	$3.98 \cdot 10^{-5}$	$6.71 \cdot 10^{-4}$
ModelDep 220	$5.13 \cdot 10^{-6}$	$5.43 \cdot 10^{-6}$	$5.03 \cdot 10^{-6}$	$1.4 \cdot 10^{-3}$
VarImp 002	$7.53 \cdot 10^{-6}$	$7.56 \cdot 10^{-5}$	$1.65 \cdot 10^{-4}$	$9.66 \cdot 10^{-5}$
VarImp 220	$3.46 \cdot 10^{-6}$	$7.42 \cdot 10^{-5}$	$1.77 \cdot 10^{-4}$	$1.4 \cdot 10^{-3}$
Testing	Median MSE	Mean MSE	Std. dev.	p-value to SGP
SGP	$1.42 \cdot 10^{-5}$	$2.61 \cdot 10^{-4}$	$7.67 \cdot 10^{-4}$	$9.83 \cdot 10^{-1}$
SensGP	$8.06 \cdot 10^{-5}$	$2.28 \cdot 10^{-4}$	$2.98 \cdot 10^{-4}$	$5.64 \cdot 10^{-2}$
ModelDep 2	$2.89 \cdot 10^{-5}$	$9.66 \cdot 10^{-5}$	$1.22 \cdot 10^{-4}$	$1.99 \cdot 10^{-1}$
ModelDep 002	$3.51 \cdot 10^{-5}$	$2.84 \cdot 10^{-4}$	$6.57 \cdot 10^{-4}$	$3.84 \cdot 10^{-1}$
ModelDep 220	$5.99 \cdot 10^{-5}$	$1.49 \cdot 10^{-4}$	$1.71 \cdot 10^{-4}$	$8.15 \cdot 10^{-2}$
VarImp 002	$1.04 \cdot 10^{-4}$	$2.4 \cdot 10^{-4}$	$2.46 \cdot 10^{-4}$	$5.64 \cdot 10^{-2}$
VarImp 220	$8.47 \cdot 10^{-5}$	$3.68 \cdot 10^{-3}$	$1.33 \cdot 10^{-2}$	$1.06 \cdot 10^{-1}$