CODA with Symbolic Regression

Remco Spelthan
Department of Data Science and Knowledge Engineering
Maastricht University
Maastricht, The Netherlands

***Abstract*—APG uses CODA to validate whether every mutation of a client's pension is calculated correctly. Currently, this is being done by linear regression, but this requires domain knowledge and pre-processing, and doesn't explain all mutations. In this report a Genetic Algorithm with Symbolic Regression is introduced, with the aim to explain certain groups as well as, or better than the linear regression method CODA currently uses. To explain the mutations, additional methods such as LASSO and Variable Switch were introduced. The Genetic Algorithm is shown to be an effective method to combine input variables and explain mutations. Future research should be done towards to further increase the explanation rate and decrease run time.**

CONTENTS

This thesis was prepared in partial fulfilment of the requirements for the Degree of Bachelor of Science in Data Science and Artificial Intelligence, Maastricht University. Supervisor(s): Anna Wilbik, Rachel Cavill. External Supervisor(s): Arjen de Vries

I. INTRODUCTION

APG is the biggest pension service provider in the Netherlands and one of the biggest in the world, it manages pension assets of around 4.5 million Dutch citizens for its clients [1]. Keeping track of pension assets of these clients is a priority of APG, but validating whether every mutation of a client's pension asset is calculated correctly by hand is infeasible. CODA is a machine learning application that aims to solve this problem through different kinds of supervised and unsupervised learning that automatically finds groups of mutations that use the exact same calculation function. This reduces the number of mutations that have to be validated by domain experts, and therefore reduces time spent on validating mutations. An example of a calculation function is the following:

$$y = \alpha x_1 x_2 x_3 \tag{1}$$

Up until now, CODA has only used forms of linear regression and applying domain knowledge directly to find calculation rules of mutation groups. After using these methods, there are still groups of mutations that remain unexplained. These groups are currently not explainable using forms of linear regression. Using domain knowledge to determine calculation rules of mutations for these unexplained groups without an automated process is not viable time-wise. These calculation rules for mutations also tend to change on a year-to-year basis. Therefore, CODA will have to be more adaptable and flexible.

CODA also has to be time efficient, since the total scope of the CODA project has around 2 million mutations. Since we focus this project on a single type of product, meaning a single dependent variable, we only need to explain around 400000 mutations, divided over around 200 groups. This means that there are a total of 200 functions to be found.

Since research on Genetic Algorithms being used for pension fund administration data is very limited (or even non-existent), we need to limit our search terminology to finding equations with low error in data. In this report, a Genetic Algorithm (GA) with Symbolic Regression (SR) is introduced for CODA, which aims to explain the calculation rules of mutation groups without applying domain knowledge beforehand. A GA with SR will take more time to run than using predefined calculation rules without an automated process [2]. This leads to the following research questions this report attempts to answer:

- How can we reduce the time it takes for a Genetic Algorithm, combined with Symbolic Regression to find a feasible solution for a given group of mutations?

To limit the domain knowledge our GA is allowed to use, we need to use most, if not all, input variables. This is to ensure that a correct calculation rule can be generated using the given input variables. This leads to our second research question:

- How can we increase the amount of input variables we use, such that we can create functions with less domain knowledge, while still giving accurate solutions?

To make our GA more efficient and generate more accurate solutions, we need to introduce methods in our GA that either limit the search space of mathematical expressions, or increase the speed with which a certain method is carried out. Therefore we will implement a coefficient optimization method, a mutation method and present the general pipeline used in our GA. These methods will all be weighed against each other to evaluate which are needed to answer our research questions. Our GA differs from pre-existing GA's in the sense that it combines previously existing methods and optimizes them for this use case.

One requirement is that the model's prediction is allowed to deviate exactly €1 from the original value. We are also given 17 input variables, which our method can select to use in its function. For the scope of this report, the data for the year 2017, 2018, 2019 and 2020 are considered. For the implementation of our Genetic Algorithm with Symbolic Regression, we use the DEAP package [3].

The rest of this report is structured as follows: First, we introduce previous literature, next we introduce and explain the methods we used for this report, then we explain how we performed the experiments and then we show the results of the experiments. After that we discuss the results and give our conclusion together with a recommendation for APG.

II. Related Work

This section begins with a summary of work on Symbolic Regression with a focus on uses combined with Genetic Algorithms. The second section discusses Symbolic Regression in combination with other methods, such as Deep Learning. We conclude with how our approach differs from the introduced methods.

A. Symbolic Regression in combination with Genetic Algorithms

Symbolic Regression has an advantage over other methods, which is that Symbolic Regression doesn't make an assumption about the structure of the function we want our datapoints to be fitting. Since Genetic Programming also doesn't make an assumption about its final result, only that we can achieve it by means of natural selection, these two methods are well suited for eachother [4].

The most popular methods in Genetic Programming for these combinations are Multiple Regression Genetic Programming (MRGP) and e-Lexicase Selection (EPLEX). These two methods were the best performing in a large benchmark study of recent symbolic regression methods [5]. Where the MRGP method was the best performing Genetic Programming method across all the datasets, and EPLEX was the best performing Genetic Programming method in the test dataset. MRGP is a hybrid Genetic Programming method that combines a tree-based Genetic Program with LASSO. It decouples and linearly combines a program's subexpressions and performs multiple regression on the target variables [6]. This new output is used as an assessment for fitness, and not the original output the Genetic Algorithm produced. After this the Genetic Program continues with its own original solution.

EPLEX is based on a selection method in Genetic Programming that considers test cases seperately, rather than in aggregate, when performing its parent selection. EPLEX differs from this original method, because it redefines the the way parents are selected. In EPLEX, they are selected as follows: First, the entire population is added to a selection pool. Second, the fitness cases are shuffled, then the individuals in the pool with a fitness worse than the best fitness on this case among the pool are removed. Then, if more than one individual remains in the pool, the first case is removed and 3 is repeated with the next case. If only one individual remains, it is the chosen parent. If no more fitness cases are left, a parent is chosen randomly from the remaining individuals [7].

B. Deep Symbolic Regression

Symbolic Regression combined with Deep Learning is relatively under-explored in comparison to Genetic Programming. In Deep Symbolic Regression for recurrent sequences, transformers (Seq2Seq models without RNN) that infer the function or recurrence relation of underlying sequences of integers or floats are described [8]. This method works by training Transformers on OEIS sequences, and this paper shows that it outperforms built-in Mathematica functions for recurrence prediction [8]. This method also is able to yield fairly accurate approximations of constants.

In Symbolic Regression via Deep Reinforcement Learning Enhanced Genetic Programming Seeding, a hybrid between neural-guided and genetic programming is introduced, which

uses the neural-guided component to seed the starting population of a random Genetic Programming component, eventually learning better starting populations [9]. This method yields, according to the author, better results then most state-of-the-art methods. A problem with this method is that it seems to only outperform other methods in specific circumstances.

C. Differences of reviewed approaches with our work

This work belongs to the group of Symbolic Regression in combination with Genetic Algorithms approaches. Similarly to the MRGP approach, we also use regression methods to further optimize a solution given by our Genetic Program. However, MRGP performs multiple regression on a decoupled symbolic function, while our methods only needs to optimize a single coefficient already present in the program. We only need to optimize a single coefficient since the groups in the dataset provided by APG usually only needs to have a single coefficient. Where MRGP [6] continues with the original solution the Genetic Program generated, we continue with the optimized solution our regression method gives us. We do this since in our Genetic Program there is, when applicable, already a coefficient present. Since optimizing this coefficient doesn't change the structure of the symbolic formula found by the Genetic Program, we continue with this optimized individual. Our method is also different from the EPLEX method, since we use tournament selection. We use tournament selection in our case since it is easily implemented and efficient [10]. Our approach is also different from Deep Symbolic Regression methods, since our method uses a Genetic Programming approach. We decided against using Deep Symbolic Regression as a method, since this field is relatively new, there aren't robust libraries for its application and project time constraints [11].

III. Methods

Our Genetic Algorithm combined with Symbolic Regression needs a lot of changes made to it to run efficiently and generate accurate results. First, we will give a brief explanation of the problem with Genetic Algorithms and Symbolic Regression, and then we will provide methods that will be able to help solve these problems.

A. Genetic Algorithm and Symbolic Regression

Symbolic Regression is a method where mathematical expressions are searched such that we can find a model that fits a given dataset the best. With this, we usually want our model to be the most accurate and simplistic. Genetic programming is the most common method for symbolic regression [12]. The drawback of this combination is that Genetic Programming has a low convergence speed, which results in longer run times for problems with a big dataset and a large number of input variables [13]. Another problem with Genetic Algorithms is that it can have the tendency to converge prematurely, such that the solution we get is suboptimal [14]. These problems need to be (partially) solved with methods described in the following sections.

B. Coefficient Optimization

Optimizing coefficients is a fast way to get an optimal solution for the current function structure [15]. This can help us in regard to finding a good solution faster, increasing the convergence speed of our Genetic Algorithm [6].

1) Gradient Descent: The first method we use to optimize coefficients is a simple form of gradient descent. Gradient descent is an optimization algorithm, where it minimizes a function by moving in the direction of the opposite of the gradient, as such:

$$a_{n+1} = a_n - \gamma \Delta F(a_n) \tag{2}$$

We use our gradient descent method in combination with our genetic algorithm, such that we can optimize the coefficients in the formulas every 5 generations.

2) LASSO: The second we use to optimize coefficients is LASSO, which performs L1 regularization. L1 regularization can result in sparser models than other types of regularization [16] [17], which is useful in our case since we prioritize smaller solutions. The goal here is to minimize the following function:

$$\sum_{i=1}^{n}(y_i - \sum_{j} x_{ij}\beta_j)^2 + \lambda \sum_{j=1}^{p} |\beta_j| \tag{3}$$

We use our LASSO method in combination with our genetic algorithm, such that we can optimize the coefficients in the formulas every 5 generations.

C. Input Variable Switch

In our dataset, we have a lot of very similar input variables, where for most cases the only difference between these input variables is that some of them have a slightly more correct value than the other one. This can result in premature convergence for our Genetic Algorithm, since it can get good enough input variable instead of choosing the more correct one. To solve this, a mutation step is introduced which after every 5 generations randomly changes on input variable in the individuals function. This can decrease the chances of premature convergence taking place, since there is now a higher probability that a more correct version of the function is found.

D. Genetic Algorithm

A Genetic Algorithm is an algorithm inspired by natural selection. Here we briefly describe which methods the Genetic Algorithm we used have and how they work. The individual, which is a symbolic function, that is inside the Genetic Algorithm is represented in a tree structure as follows:

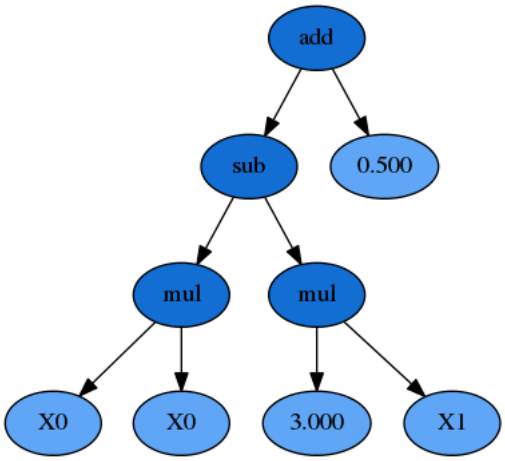

Fig. 1. Individual Tree Structure

The Genetic Algorithm also has a fitness function, which we describe later.

a) Initialization: The Genetic Algorithm uses the Ramped Half-n-Half initialisation to generate random individuals. This method works by using the full method, which generates an individual where each leaf has the same size, and the grow method, which generates an individual where each leaf might have a different size. This method is shown to be a good enough for our use case [18], but it still has some biases [19].

b) Selection: The Genetic Algorithm uses Tournament Selection [20], which is a form of selection that runs several tournaments in which each tournament a certain number of individual compete. The winner of each tournament is selected for crossover and mutation if applicable.

c) Crossover: The Genetic Algorithm uses a one point crossover method. This method randomly selects a single point in the two trees ready for crossover, and switches the branches below that point with the other tree [21].

d) Mutation: The Genetic Algorithm uses uniform mutation, which randomly selects a random point in the individual tree, and then replaces the subtree at that point as a root by the individual generated by the initialization method mentioned in the initialization paragraph.

E. Genetic Algorithm Pipeline

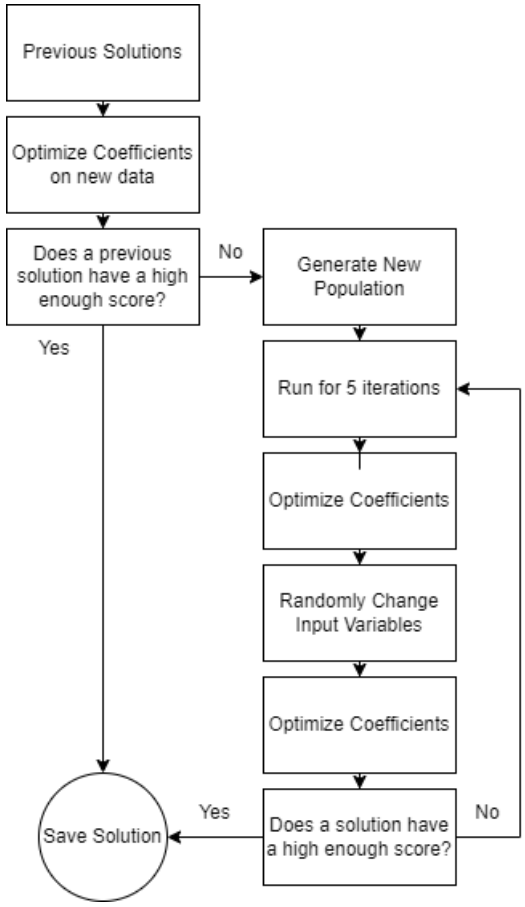

Fig. 2. The Genetic Algorithm Pipeline

In Figure 2 we see the pipeline we use for the Genetic Algorithm. In this pipeline, we start with information from our previous run on another group of data (Note: we have different groups of mutations in our dataset, and we run the Genetic Algorithm individually on those groups. When we actually run the code, the Genetic Algorithm starts with no information). We then optimize coefficients found in the previous solutions, with either gradient descent or LASSO on the new data group. We do this so that we get a more optimal result for our data group right away. Next, we check if one of our solutions has a high enough score, which is another way to say, "do more than 96% of the datapoints within this group fall within the allowed error margin of the solution's function?". If the answer is yes, then we can just save our solution and move on to the next data group. If the answer is no, we need to generate a new population so that we can try to generate a function that scores higher. After we generate a new population, we run the Genetic Algorithm with only tournament selection, one-point crossover and uniform mutation for 5 generations. After these 5 generations, we optimize our coefficients with either gradient descent or LASSO. After this step, we randomly change an input variable for every individual that is currently in the population, and after this we optimize the coefficients

for these new solutions as well. If the current best solution has a high enough score, we save this solution. If the current best solution doesn't have a high enough score, we continue back at the step where we run our Genetic Algorithm for 5 iterations. Note that for every iteration of these 5 steps, we save the best solution. We do this so that when the algorithm finishes, we have the best solution found up until the last iteration. We stop the loop after 40 iterations.

F. Fitness Function

Here we explain the fitness function used in our Genetic Algorithm. Since these methods are described in my previous KE@Work report, I won't go into full detail here. I will briefly describe them and their effects, but no experiments with these methods will be conducted.

1) RMSE: The RMSE is a quadratic scoring metric that also takes into account the average size of the error. The RMSE metric gives relatively higher weight to larger errors, resulting in the Genetic Algorithm preferring solutions with larger big errors. This is what we need in our Genetic Algorithm since we want our function to be very accurate, and using RMSE will steer our algorithm in the right direction.

2) Parsimony Pressure: A problem that affects Genetic Algorithms with Symbolic Regression, is that the Genetic Algorithm, when not restricted, tends to increase the size of the function, with slight improvements every iteration. This greatly reduces the use for our final function, since this function is usually not interpretable by humans, because of its length. Therefore we introduce a parsimony penalty into the fitness function, which punishes longer solutions. This results in the final solution that our Genetic Algorithm gives being shorter, and therefore more interpretable.

3) Penalty for errors above allowed margin: For this dataset, we want the final function to be able to succesfully hit most datapoints within an allowed error margin. To prioritize this, we introduce a penalty that punishes functions where the error is larger then the allowed error margin, but doesn't give any punishment to datapoints that are below the allowed error margin. This results in our Genetic Algorithm prioritizing functions that are able to hit more datapoints, while also still being able to find functions.

IV. Experiments

For our experiments, we want to compare our implementations when they are activated, versus when they are not activated. We do this so that we can easily see the benefit or drawback of a certain method. We test our Genetic Algorithm against a simple linear regression method previously implemented by APG. This linear regression method uses domain knowledge to combine certain input variables beforehand, and then applies regression. We present the percentage the Genetic Algorithm was able to explain for each year, and for all years combined. We also report on how many groups in that year the Genetic Algorithm was better at than the older linear regression method implemented by APG. To get a more in depth view of what the Genetic Algorithm is doing and how it performs, we record the maximum, minimum and average accuracy of every population in the first group. We also measure the maximum, minimum and average diversity of the populations by recording the average distance between solutions in our population. For the accuracy and diversity calculations, we only look at the first group we encounter, since this group will not benefit from previous information. We also measure the time it takes for each experiment.

We will compare all results against a baseline for the Genetic Algorithm, to keep this report from being uncluttered. The baseline with these methods include: the coefficient optimization method being set to LASSO and the input variable switch being turned on.

A. Coefficient Optimization

For our coefficient optimization experiments, we want to experiment with both gradient descent and LASSO regression. We compare gradient descent and LASSO by running gradient descent with a rate of $1e{-}10$ and 700 iterations, and LASSO with a rate of $5e{-}10$ and 2000 iterations.

B. Input Variable Switch

For this method, we want to experiment with whether we activate this method or not and see what the differences are.

C. Everything On/Off

We also want to see what the difference would be if we just ran the Genetic Algorithm without any of the improvements we made. To do this, we compare the results when we have all previously mentioned methods turned off, and when they are turned on.

V. Results

In this results section, the greater than sign ($>$) means that the Genetic Algorithm is better than the linear regression method. The double equals sign (==) means that they perform the exact same, and the smaller than sign ($<$) means that the Genetic Algorithm performed worse than the linear regression method.
Regarding the groups that are tested on, these are categorized by APG beforehand and are assumed to be correct.

A. Older Linear Regression Method

Old LR	**2017**	**2018**	**2019**	**2020**	**All**
Run Time	1 min	2 min	2 min	2 min	11 min
Score	96.81%	99.64%	99.16%	87.69%	96.15%
# points	19319	122646	70489	77226	289680

This older linear regression method is the method against we compare all other methods when we mention whether the Genetic Algorithm is better or worse than the linear regression method.

B. Coefficient Optimization

Gradient Descent	2017	2018	2019	2020	All
GA >Linear	1	4	6	6	18
GA == Linear	31	29	37	40	134
GA <Linear	21	22	12	6	63
Run Time	8 h	5 h	4.5 h	3.75 h	24 h
Score	83.47%	91.96%	94.04%	87.73%	94.86%

In the table above we see that for every year the score is above 80%. We also notice that the run times for gradient descent are fairly high. These run times are all above 3.5 hours, meaning that this method takes the longest out of any of the other methods tested. When we run all years at once, we see that it takes 24 hours to run. We also see that information for previous years is used when running all at once, since the groups better than the old linear regression method aren't an addition.

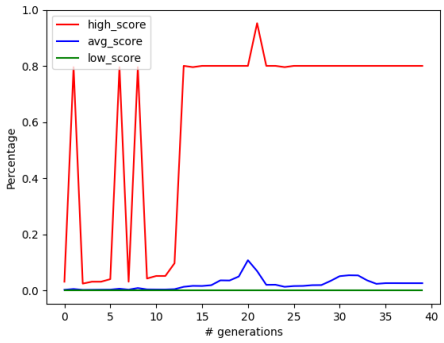

Fig. 3. Accuracy Gradient Descent

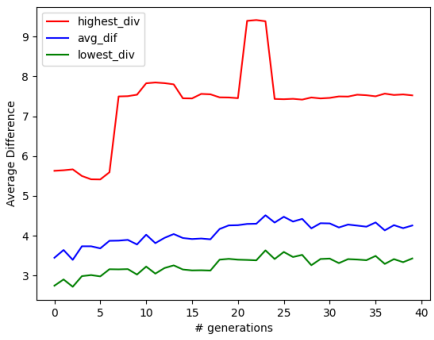

Fig. 4. Diversity Gradient Descent

We see that the with the gradient descent method the Genetic Algorithm needs to run for about 40 generations and it's accuracy doesn't come above the 96% threshold. We notice that for the diversity it spikes around 22 generations.

C. Input Variable Switch

Var Switch Off	2017	2018	2019	2020	All
GA >Linear	1	5	3	4	24
GA == Linear	33	32	41	33	152
GA <Linear	19	18	11	15	39
Run Time	39 min	42 min	24 min	58 min	33 min
Score	95.39%	76.34%	98.395%	83.84%	94.23%

In the table above we see that for every year the score is above 70%. We also notice that run times we have per year are all under an hour. When we use all years in a run at once, we notice that the Genetic Algorithm gets closer to the old linear regression method. Instead of explaining 96.15%, it is able to explain 94.23%.

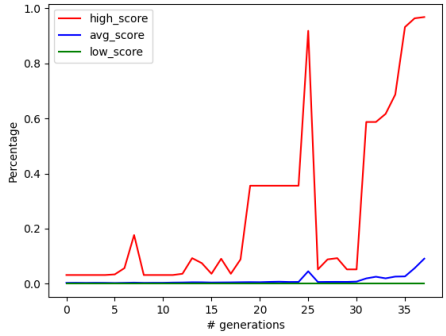

Fig. 5. Accuracy Variable Switch Turned Off

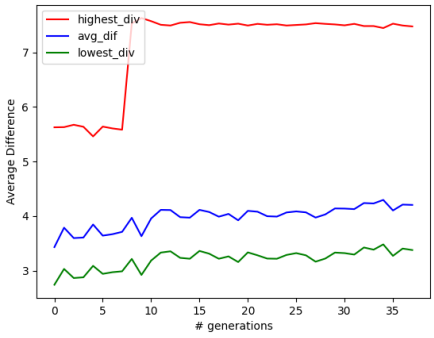

Fig. 6. Diversity Variable Switch Turned Off

We see that the with the variable switch turned off, the Genetic Algorithm needs to run for about 35 generations and it's accuracy at the end goes above the 96% threshold. We notice that for the diversity it spikes around 10 generations.

D. Everything On/Off

Everything On	2017	2018	2019	2020	All
GA >Linear	1	2	2	5	28
GA == Linear	35	32	43	38	164
GA <Linear	17	21	10	9	23
Run Time	6 min	15 min	11 min	10 min	36 min
Score	96.47%	90.48%	95.66%	87.67%	95.699%

In the table above we see that for every year the score is above 85%. We also notice that run times we have per year are all under 15 minutes, making this the fastest method. When we use all years in a run at once, this method is able to get the closest to the old linear regression method. Instead of explaining 96.15%, it is able to explain 95.699%.

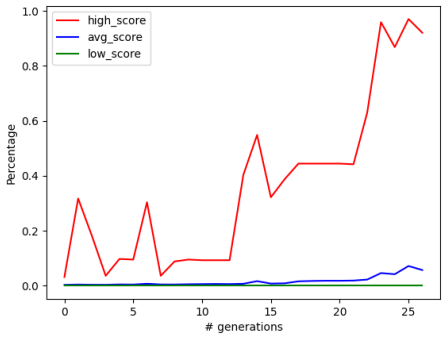

Fig. 7. Accuracy Everything On

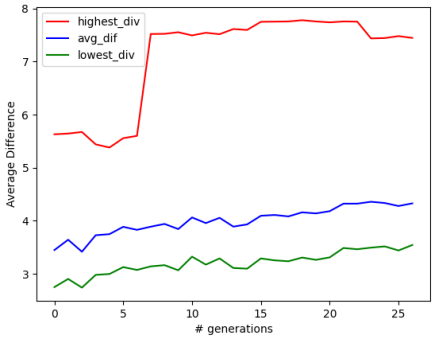

Fig. 8. Diversity Everything On

We see that the with the every mentioned method turned on, the Genetic Algorithm needs to run for about 25 generations and it's accuracy at the end goes above the 96% threshold. We notice that for the diversity it spikes around 20 generations.

Everything Off	**2017**	**2018**	**2019**	**2020**	**All**
GA >Linear	0	0	3	1	19
GA == Linear	7	0	39	0	124
GA <Linear	46	55	13	51	72
Run Time	42 min	62 min	12 min	55 min	75 min
Score	18.43%	7.34%	90.69%	16.27%	73.23%

In the table above we see that for every year except 2019, the score is around 15%. The year 2019 is, 90.69%. We also notice that run times we have per year are all around an hour, except for 2019. When we use all years in a run at once, this method is able to explain around 73.23%.

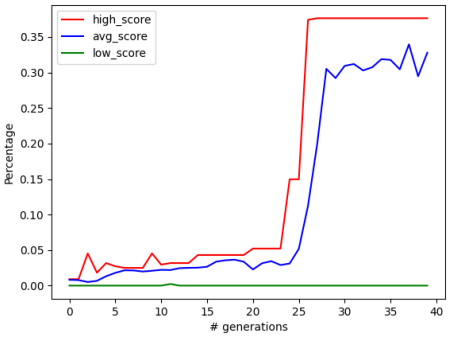

Fig. 9. Accuracy Everything Off

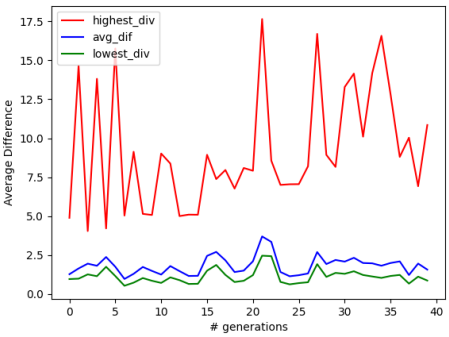

Fig. 10. Diversity Everything Off

We see that the with every mentioned method turned off, the Genetic Algorithm needs to run for the full 40 generations and it's accuracy doesn't get above the 96% threshold. We notice that the diversity here is more fluctuating, and it spikes around 20 generations.

VI. Discussion

We first notice that when we compare the Gradient Descent method to the LASSO method (so the Gradient Descent table to the Everything On table), we see that it takes the gradient descent method way more time to come to a comparable conclusion that the LASSO method. This makes the Gradient Descent method we implemented infeasible to run on a large scale, when a method that is this much faster is available.

We notice that in our results, the overall best performer is the Genetic Algorithm with every method we mentioned turned on. A close second is where the variable switch is turned off. We see that the table where the variable switch method is turned off has a year that has an unusually high performance, the year 2019. This is probably due to this being a year that doesn't have a lot of outliers anyway, we see this when we look at the height of the percentage in the old linear regression method. Another possible explanation is that the groups in this year are more linear, making it easier for this method to find a good solution. We also see that when the

variable switch method is turned off, it takes longer for our Genetic Algorithm to find a solution.

When all of our implemented methods are turned off, we see that our Genetic Algorithm, with every year combined, can only explain 72.23% of cases. In comparison to other settings of the Genetic Algorithm, we see that this performs the worst by far.

VII. Conclusion

In conclusion, we showed that we can greatly reduce the time it takes for our Genetic Algorithm to run to find a solution for the given group of mutations. We see that when everything method we implemented is turned off, it can take up too 75 minutes to get a result, and even then the result tends to be quite bad. Our Genetic Algorithm, with the methods we mentioned added, tends to yield far faster run times and higher explanation rates, and we see this back in the results. We see that in the "Everything On" table, the Genetic Algorithm tends to get a score in the 90-95% range. Mind that this high score comes from an algorithm which still has these 17 input variables. When our methods are turned off, we see that the scores are lower and that the formulas the Genetic Algorithm provides are usually incorrect and using the wrong input variables. This means, that with our methods, we have successfully increased the amount of input variables we can use.

The recommendation for APG is that they should use the Genetic Algorithm, combined with the methods we implemented. Future research should be done to further increase the explanation rate of the Genetic Algorithm and reduce the run time.

References

[1] APG. Asset management.

[2] Bart Ian Rylander. *Computational complexity and the genetic algorithm*. University of Idaho, 2001.

[3] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.

[4] Riccardo Poli, William B. Langdon, and Nicholas Freitag McPhee. *A Field Guide to Genetic Programming*. Lulu Enterprises, UK Ltd, 2008.

[5] Patryk Orzechowski, William La Cava, and Jason H. Moore. Where are we now? a large benchmark study of recent symbolic regression methods. In *Proceedings of the Genetic and Evolutionary Computation Conference*, GECCO '18, page 1183–1190, New York, NY, USA, 2018. Association for Computing Machinery.

[6] 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*, GECCO '14, page 879–886, New York, NY, USA, 2014. Association for Computing Machinery.

[7] William La Cava, Lee Spector, and Kourosh Danai. Epsilon-lexicase selection for regression. In *Proceedings of the Genetic and Evolutionary Computation Conference 2016*, GECCO '16, page 741–748, New York, NY, USA, 2016. Association for Computing Machinery.

[8] Stéphane d'Ascoli, Pierre-Alexandre Kamienny, Guillaume Lample, and François Charton. Deep symbolic regression for recurrent sequences. *arXiv preprint arXiv:2201.04600*, 2022.

[9] Terrell N. Mundhenk, Mikel Landajuela, Ruben Glatt, Claudio P. Santiago, Daniel faissol, and Brenden K. Petersen. Symbolic regression via deep reinforcement learning enhanced genetic programming seeding. In A. Beygelzimer, Y. Dauphin, P. Liang, and J. Wortman Vaughan, editors, *Advances in Neural Information Processing Systems*, 2021.

[10] Brad L. Miller and David E. Goldberg. Genetic algorithms, tournament selection, and the effects of noise. *Complex Syst.*, 9, 1995.

[11] William La Cava, Patryk Orzechowski, Bogdan Burlacu, Fabricio De França, Marco Virgolin, Ying Jin, Michael Kommenda, and Jason Moore. Contemporary symbolic regression methods and their relative performance, 07 2021.

[12] Chen Chen, Changtong Luo, and Zonglin Jiang. Elite bases regression: A real-time algorithm for symbolic regression. 07 2017.

[13] Hmida Hmida, Sana Ben Hamida, Amel Borgi, and Marta Rukoz. Scale genetic programming for large data sets: Case of higgs bosons classification. *Procedia Computer Science*, 126:302–311, 01 2018.

[14] Yee Leung, Yong Gao, and Zong-Ben Xu. Degree of population diversity - a perspective on premature convergence in genetic algorithms and its markov chain analysis. *IEEE Transactions on Neural Networks*, 8(5):1165–1176, 1997.

[15] Maarten Keijzer. Scaled symbolic regression. *Genetic Programming and Evolvable Machines*, 5:259–269, 09 2004.

[16] Andrew Y. Ng. Feature selection, l1 vs. l2 regularization, and rotational invariance. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, page 78, New York, NY, USA, 2004. Association for Computing Machinery.

[17] Robert Tibshirani. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 58(1):267–288, 1996.

[18] John Koza and Riccardo Poli. *Genetic Programming*, pages 127–164. 01 2005.

[19] Edmund Burke, Steven Gustafson, and Graham Kendall. Ramped half-n-half initialisation bias in gp. pages 1800–1801, 07 2003.

[20] Tobias Blickle. Tournament selection. *Evolutionary computation*, 1:181–186, 2000.

[21] Riccardo Poli and W. Langdon. Genetic programming with one-point crossover and point mutation. 12 2001.

APPENDIX

A. Genetic Algorithm

In this section, we'll briefly explain all the unusual parts about our Genetic Algorithm.

1) Fitness Formula:

0: **if** $|\hat{y} - y| > 1$ **then**
0: $y_{diff} = \hat{y} - y$
0: **else**
0: $y_{diff} = 0$
0: **end if**=0

Resulting in the fitness function:

$$fitness = \sqrt{\frac{\sum_N^i (y_{diff\,i})^2}{N}} + PP + NZP \tag{4}$$

Where N is the sample size, PP is the Parsimony Penalty, and NZP is the Non-Zeroes Penalty. Here we see that we first take the error margin into account with the if statement. In the fitness function, we use RMSE aswell, since it gives higher weight to larger errors, and we want to minimise those. However, since RMSE doesn't take into account having individual mutations explained (e.g. the error is zero), we have to add a penalty that discourages non-zeroes. We also in our case want our formula to be shorter, since a longer formula may be slightly better performance wise, but this could lead to overfitting.

2) Error Margin: We take the allowed error margin into account before we perform the fitness calculation. We do this since our GA will then be able to prioritize solutions which fall into our error margin.

0: **if** $|\hat{y} - y| > 1$ **then**
0: $y_{diff} = \hat{y} - y$
0: **else**
0: $y_{diff} = 0$
0: **end if**=0

3) Parsimony Penalty: We want to prioritize shorter solutions over longer ones, since longer solutions tend to be nonsensical and will result in the Genetic Algorithm not giving us a solution that conforms to the domain knowledge we have.

$$PP = \alpha * function\_length \tag{5}$$

Where α is the parsimony coefficient, and $function\_length$ is the length of the solution's function. This length of the function takes into account the number of input variables, plus the number of operators.

4) Non-Zeroes Penalty: In our data we want to explain as many individual mutations in a group as possible. To correctly explain a individual mutation, the error has to be within the given error boundary. If the error is within that certain boundary (€1 in our case), then the difference is set to 0. This means we can penalize errors that are bigger than the €1 deviation, such that we can prioritize solutions with more mutations within the error margin.

$$NZP = \beta * nonzeroes\_count \tag{6}$$

Where β is the non-zeroes coefficient, and $nonzeroes\_count$ is the count of non-zeroes (every prediction with an error above 1) in y_{diff}.

B. Warm-start

A warm-start in a Genetic Algorithm means using information of previous runs to inform the current run. In this Genetic Algorithm, we do this by having a hall of fame, which saves the top 20 from the previous run. Since this hall of fame consists of 20 formulas, the chance is lowered that the Genetic Algorithm encounters a group of which 90% is explained by accident. Before starting our Genetic Algorithm each run, we first optimize the coefficients of the previous solutions for our current data. Then, if our best solution explains 90% of the mutations in that group, we use this formula, if it explains less than 90% of the mutations, we start our Genetic Algorithm. We use 90% here since when the solution explains more than 90%, it is very likely that it is correct. If it is below 90%, there is a chance that it is incorrect. This method is arbitrary, and there should be experiments with this in the future.