Universidade Nova de Lisboa

Information Management School

Year 2017/2018

Master's degree in Advanced Analytics

Predictive Models

Project 2: Regression problem

Group formed by:

Carolina Bellani M20170098

Gonçalo Pereira M20170450

Sofia Jerónimo M20170070

Inês Rato M20170592

List of Contents

List of figures

Introduction

For this project we have to solve a regression problem having 287 data instances and 277 input features.
We will focus more on obtaining a good generalizing model with preferable a small size following the Occam's razor logic. Even if we could obtain a good model with a gigantic size, in this project and in the techniques implemented we are trying to avoid bloat due to computational limitations while obtaining a good and simple model.
Another thing to take into consideration is that most of the methods implemented in GP will extend and work smoothly in GPSP. With that in mind, we will focus mostly on the "bigger" techniques that affect the main mechanism of the algorithm instead of some minor modifications such as a different crossovers or mutations.

Relevant techniques implemented in the project

Feature selection: at the start, we verified some statistics on the input variables. After obtaining these results, we tried a few methods to reduce the search space using less variables. First, we selected 79 variables by variable worth using SAS Enterprise Miner, which could be used using the parameter 'Use_Important_Vars'. In the second method, we noticed that 53 variables only had "0" for all observations, giving us absolutely no relevant information. Therefore, these variables were removed from the sets and will not be used for the rest of the evolutive process.
Variable absolute frequencies control: at the start we implemented two methods, 'importVars' and 'exportVars'. The idea is to understand the most important variables by the frequency they appear in the best individuals. First, we have a .txt file filled with the same amount as the number of variables. Every N generations and if certain conditions are satisfied, such as low training and unseen error, the count of the variables used is added to the .txt file. For example, if in a good individual we have 10 times the variable X15 we add 10 to the count. After a considerable amount of runs the 'importVars' method will import these frequencies into an array which will be used in different methods.
Parallel Populations: the parallel populations' method is fascinating because it allows the individuals to evolve in different environments. If we take a metaphor it would be like having the population of each continent evolving in a separate way adapting to the conditions (selection pressure) on that specific environment, all with one goal in mind, the survival (natural selection). After a N number of generations, we swap the best individuals in order to develop the race on the other continent. The only down-set we noticed on this method was that, after a few swaps, all populations lose their specific features and just become a much more homogenous population (population convergence), just like the different traits of human beings are getting mixed due to globalization.
We also implemented a method to define the parameters for each population. So, if it's the population X, it will use a different set of parameters than the rest. This was meant so we could have the highest possible diversity throughout all the populations and guarantee a different evolutive process for each one of them.
Input variable splitting: at the start, we had 277 (+1 target variable) input variables, which were divided into 4 sets. These sets were created by two shuffling criteria, the first one we just split the variables in 4 sets (**Figure 1**) and in the second one, we loop through all the variables and assign each one to a different set (**Figure 2**). Each of these sets will be assigned to a population, in case the populations are more than 4, it will loop back to the first one. This means that when using this method, we should define a number size of populations multiple of 4.

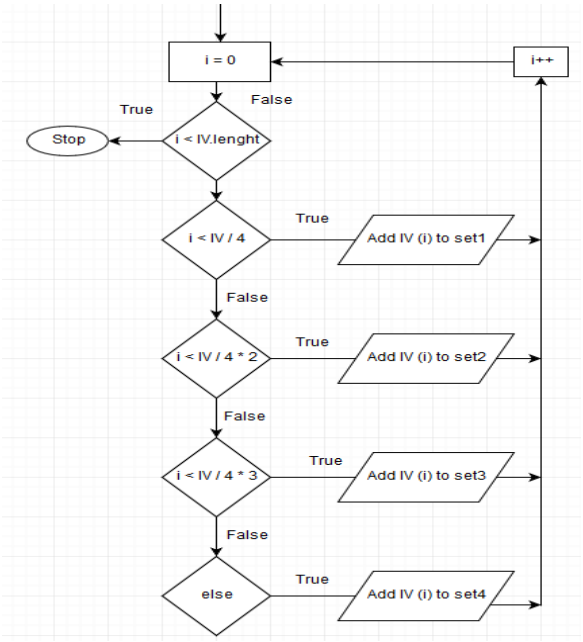

Figure 14. Shuffling Criteria 1

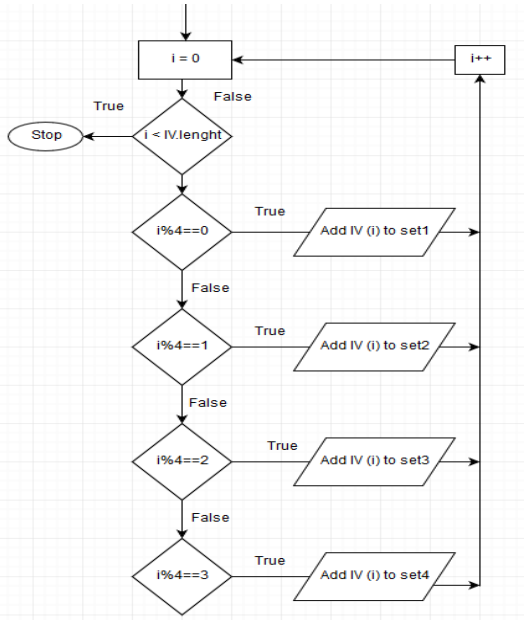

Figure 15. Shuffling Criteria 2

Observations Splitting: following the same logic as in the previous method, we split the 287 observations into 2 or 4 sets. To split the observations, we merely created an evaluate method on the training set that uses the corresponding set. That way the specific method that uses this technique will be optimized only for that input split.

Steady State GP (SSGP) centralized: this method was implemented as a continuation of the parallel populations. Seems like the problem of the previous method is that we lose the diversity due to the migrations and gene flow. In order to solve it, we created a centralized population. All the different populations are isolated from each other, allowing the 'speciation' of individuals to each environment, while every N generations, they 'feed' their best individual into a centralized population. In the previous metaphor, it would be as selecting the best fitted human individuals on earth to send to Mars and create a super colony there.

In this centralized population, we are using SSGP. In theory, this mechanism does not allow an 'normal' evolutionary process over time since it does not generate different offspring , it simply select an individual, perform the variation operations (mutation and crossover) on it and put it back in the same population. We decided to use SSGP on the central population mostly because the main goal of this population is to have the best individuals from the remaining populations. This method makes it easier because it avoids the extra step of creating offspring populations and replace it in the original population. For counting and equilibrium reasons, although there are not 'generations' in SSGP, we will use the same ones as for the previous populations [1].

Ratio Bucketing: is a multi-objective optimization technique. In this case, our primary objective is to optimize the sizing parallelly to the error. For this method, we sort the population by training error. After, using the defined number of buckets, we put the sorted individuals filling the buckets in which the first bucket will contain the N best individuals. [2]

Elitism: we select the N best individuals from the previous population to insert into the offspring, thus removing the worst individuals. To take advantage of elitism to its maximum, we will use it to

reduce bloat besides maintaining the elites obviously, keeping it between the 30% - 50% threshold mark. [3]
Parsimony measure: is a multi-objective optimization as well, in which we select a different feature as secondary objective. In this case, we are redefining the training error taking into account the size of the individual. This measure is problem subjective, so it took a while to experiment and reach the sweet spot between prioritizing the error and account the individual size. After the correct calibration of this measure, the individual will have some difficulty getting past a certain size, but when it does, it means the error is compensating the size. [2]

Synergies of the techniques

Along this project we decided to focus on the generalization ability of a medium to small individual. Knowing our objective, we understand that our biggest challenge, after we have the bloat controlled, will be the diversity of our individuals. We need to keep the diversity while maintaining a low error and small individual. For this reason, we focus mostly on different methods and topologies to get the best diversified individuals together, hoping to creating the next level solution.
Putting together most of the methods implement, we initialize with a different set of parameters, input variable sets and observations set. That way, we have several parallel populations, each one using a subset of the variables and being evaluated on a different subset of the observations. This will allow to optimize each section of the dataset to the fullest, while controlling the size which will give us a smaller change of overfitting too fast. During this run, we can or not, swap the best individuals between the populations. In case we do swap, the results of the populations will generally be better. Although we lose some of the diversity that we were searching, if not all, after some time. In case we do not swap, we will have a centralized population that will optimize by itself while getting the best individuals from all the populations, which obviously have a lot of diversity reaching our goal of having a heterogenous population of good individuals.
Considering that we have almost the same proportion of observations per number of variables, when using the input variables and observation splitting techniques, we take ¼ of the input variables and use ½ of the observations. This will cause some overlap of observations in the training, although it will not bias our result and might be useful in the future since, in a certain way, we are increasing the observations per variable used.

Methods implemented in the project

Individual's methods: The individual class holds the information relevant to the individuals in our population. Besides the default methods already implemented, we added a few more to use in sync with the other techniques implemented. The relevant functions implemented were the evaluateSplitted(), in which we evaluate the individual in just a split of the dataset with the corresponding method evaluateOnTrainingDataSplitted() in order to optimize the model with that split. The evaluateOnUnseenData() will be the default one, since it makes no sense to evaluate on a split of the unseen data. For some of the methods implemented, we will use highLow(), which uses several inner methods to compare the average of the real target with the average of the predicted target. We also added some methods to better understand the individuals we are working with, such as getNumberOfOperators, getNumberOfConstants, getNumberOfVariables and some boolean such as isVariable, isLF, isOperator and so on.

Population's methods: similar methods as before were implemented such as evaluateSplitted, and several population manipulation functions like sortIndividuals and getBests for Elitism; sortIndividualsParsemony() and get methods for the parsemony measure; assignBuckets() and get methods for buckets rationing and steadyStateSwap for SSGP.
Program Elements' methods: the project had the 5 basic operations: Addition, Subtraction, Multiplication, ProtectedDivision (by default) and the Logistic Function. Following this train of thought, we implemented several operators such as absolute value, average, cos, sin, tan, ln, $\log_{10}$, min, max, negative, powers, power of 2, reminder, Sqrt, and conditionals. For the conditionals, we implemented 2 different operators.

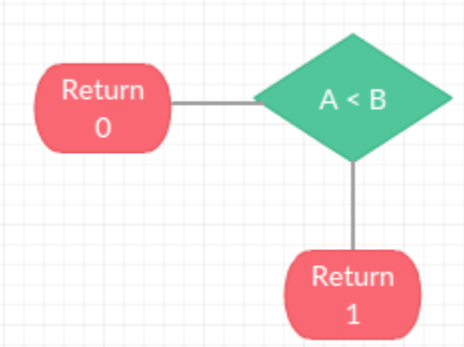

Figure 16. If Operator

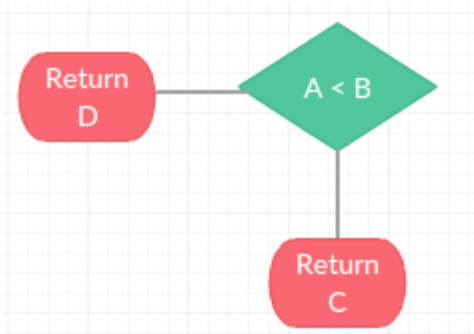

Figure 17. Conditional Operator

In the first one, if A < B it returns 1, which will ensure that the corresponding subtree will carry on, while if its false we cancel its subtree (**Figure 3**). In the second one, we just return the argument C or D depending on the outcome (**Figure 4**).
After some testing, we decided to use the default operators and the 4 arities "if" operators, while leaving the rest with a possibility to be used with the parameter 'Use_All_Operators'.

Initialization methods

IlyaInitialization: we create a population using 'rampedHalfAndHalf', select the best individual and insert it into the population being created. This process will be repeated until our main population to be initialized is completely full. In this method, we can create several individuals through a different set of parameters to guarantee a decent degree of diversity and avoid convergence.
IlyaInitializationSplitted: similar to the previous initialization only that here we initialize each population using a different set of variables. On the previous initialization, we initialize a population taking just ¼ of the input variables.
GrowSplitted: same as default Grow method except that we only use a set of the input variables.
FullSplitted: same as default Full method except that we only use a set of the input variables.
GrowRoulette: similar logic to the default Grow method with the difference that, when it has to insert an input variable, it does a roulette wheel selection on the past variable absolute frequencies to select a variable.
FullRoulette: similar logic to the default Full method, with the difference that, when it has to insert an input variable, it does a roulette wheel selection on the past variable absolute frequencies to select the variable.

Genetic Programming (GP) methods

ParameterChangePops: This method works in sync with the multiple populations, as it will assign a different crossover and/or mutation for each population.

Selection methods

SelectionMenu: to facilize the choice of the selection's methods.
TournamentSelection: adapted for parallel populations, SSGP and tournamentSelectionPOP.
RouletteWheelSelection: as Genetic Algorithm, but with adapted calculation of the fitness.
RankingBucketSelection: For this selection, we had a few preparations. First, we split the individuals into N buckets (parameter) by fitness. So, if we have 100 individuals and 25 buckets we will have 4 individuals for each bucket, and in the first bucket we have the best 4 individuals of the population. After, we just go to the first bucket and get the individual with the smallest size. This method works very well in reducing bloat and as a multi-objective optimizer, the only concern we noticed while we were testing is that, sometimes, it may focus too much on the size of the individual. Of course, the parameter needs to be tuned in and it will be used when trying to get the best and smallest individual.

Crossovers methods

CrossoverMenu: to facilize the choice of the crossover's methods.
Hoist: the offspring is generated by selecting a point inside a copy of an existing individual chosen based on fitness and "hoisting" it up to be the entire new individual.[4]
CrossoverHoisted: the offspring is generated by applying the Hoist crossover with two individuals.
CrossoverAvgImproved: default crossover but choosing one parent above the average training error and one below.
CrossoverSizeFair: the crossover point is chosen by comparing the size and if it is bigger, it selects another point.

Mutations methods

MutationMenu: to facilize the choice of the mutation's methods.
MutationShrink: replaces a randomly chosen subtree with a randomly created terminal. [4]
MutationSubTree: replaces a randomly selected subtree with another randomly created subtree.
MutationChangeVar: we select N random variables and swap them by a randomly selection of other variables.
MutationChangeOperator: we select N random operators and swap them by randomly selecting other operators.
MutationSwap: two positions are randomly chosen and swapped between them.
MutationSwapAll: same as above with the difference that there is repetition across all positions until all of them are in different places.
MutationCreate: we replace the individual by a newly created one. [4]

Genetic Semantic Genetic Programming (GSGP) Methods

Since our goal from the start was to implement and try all relevant techniques, all of those (except for the crossovers and mutations) were expanded to the GSGP and should affect its results in a similar way to how they affect GP. Concerning the crossovers and mutations, we opted to use the default ones since these geometric operators work in perfect sync with the low computational expense of GSGP.

Results

All the mentioned results are the average of at least 10 runs. For selecting the best individual, we are saving the results by 3 criteria: the best training error, the best unseen error (which is merely a saved statistic and is not in anyway on the evolutive process) and the absolute best error (which is the best unseen error of the individuals that have the best training error). The presented result is always from the absolute best error, which is the selection method for the final best object delivered. It is worth to take into consideration that, while running the algorithm, we might have several best training errors due to the multiple populations being used and the absolute best error will be the best of the bests errors.
While testing a specific technique and comparing the results, the 'default' method will always be the exact same process and parameters with the exception of the method being tested, holding all factors fixed except one. The crossovers and mutations are not presented here since they do not show much difference individually and they are used alternately in each population. Also, only the meaningful methods are presented graphically in the follow section.

Initialization

Comparing the two main initializations methods, a slight modification of EDDA and ramped Half and Half, we can see an improvement in both the training and unseen errors **(Figure 5)**. This improvement is focused on the first portion of the generations since the algorithm already starts with an elite population.

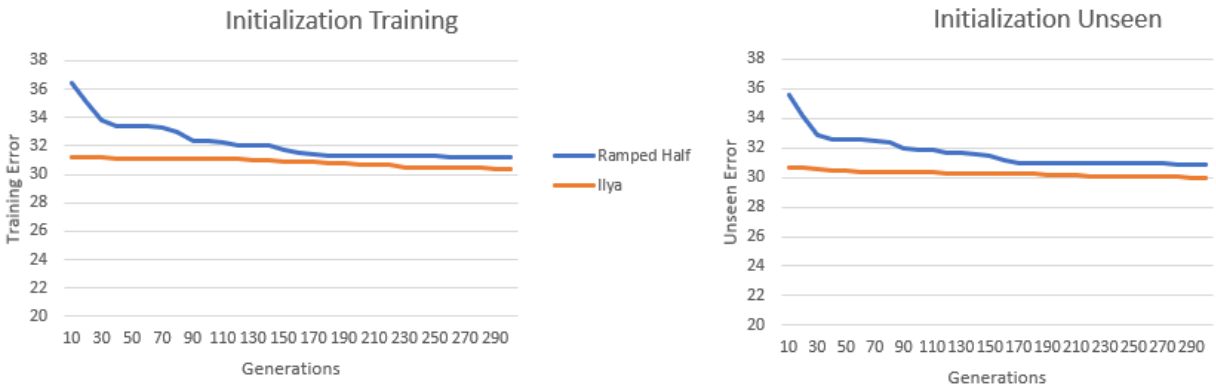

Figure 18. Initialization methods for training and unseen errors across different generations

Selection

Regarding the selection methods, the ranking bucket selection seems to be the best method in training, although with very high oscillations (**Figure 6**). For this reason, we will discard it and select the roulette as the best method for this category.

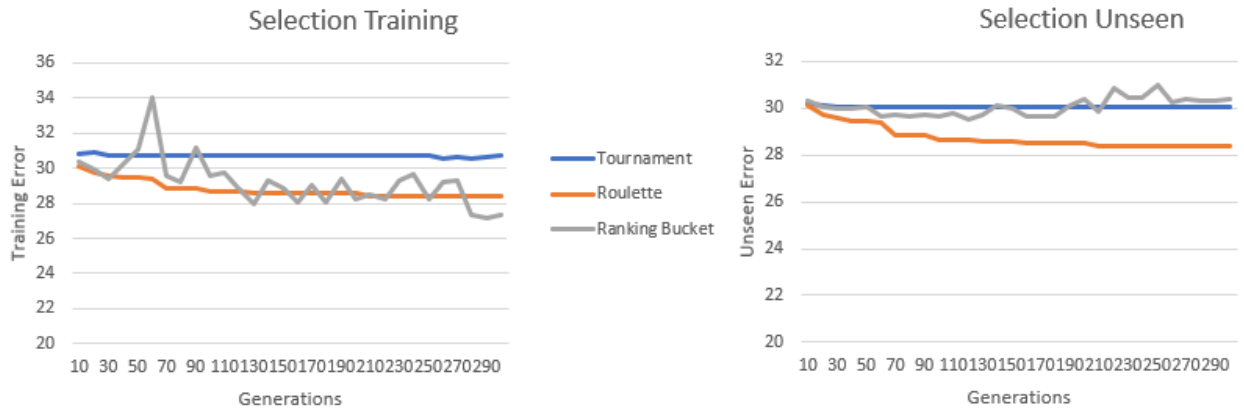

Figure 19. Selection Methods for training and unseen errors across different generations

Parallel Populations

One of the main methods implemented and that will be the stepping stone for a lot more, is the parallel populations. Even if its main purpose is to be used synchronized with the other techniques, we need to test its results. That way, we are comparing a population of size 800 with 8 populations of size 100 each, for an equivalent total. As expected, with the same individuals, the parallel populations outperform a single population and it guarantees its status as a base for the methods that follow (**Figure 7**).

Figure 20. Population comparison for both training and unseen error across different generations

Parsimony

The parsimony measure was implemented as a multi-objective technique, which it accomplished since the average size of the individual was reduced by 10 to 40 at the end of only 300 generations, while keeping the average error constant (**Figure 8**). Although it is a small difference between methods, the worse training error but better unseen error might be explained by the inability to grow too much and overfit the data, giving a slightly better generalization ability.

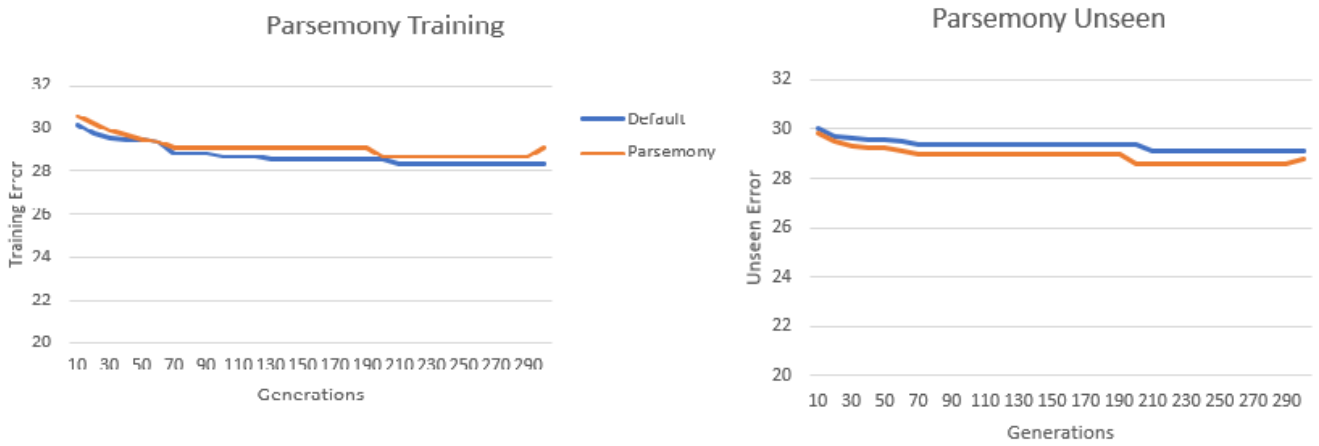

Figure 21. Parsimony measure vs. default for both training and unseen error across different generations

Important Variables

Following the previous analysis on the variable worth, we used only these variables which gave unexpected results. The training error improved significantly, while the unseen error seems to stay the same until it massively overfits our data (**Figure 9**). The most reasonable explanation for this is that most of our targets' behavior can be explained by a handful of variables, but for the optimum result only these variables are not enough, like always, the devil is in the details.

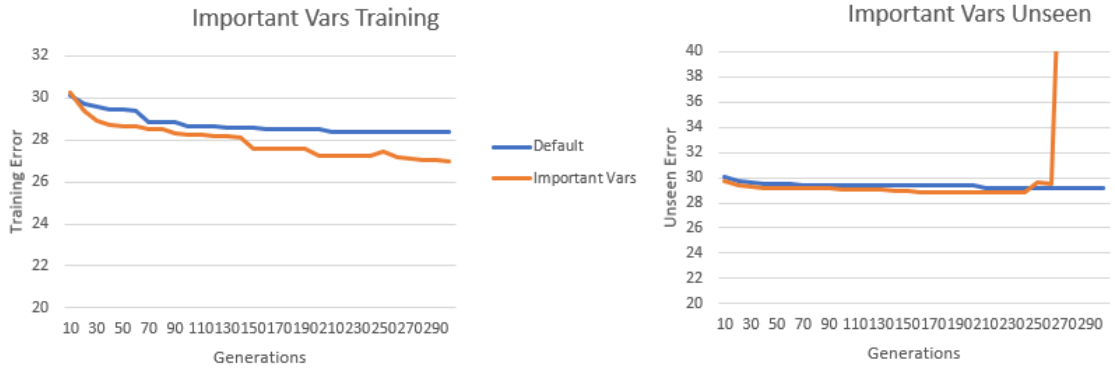

Figure 22. Important Variables vs. default for both training and unseen error across different generations

Variable Splits

Following the preprocessing mentioned before, for this method, we split the 244 remaining variables in 4 sets and assigned each of them to a population. At the start of the runs, we do not notice much of a difference, but exactly at generation 50, when we swap the bests between populations and expand the knowledge of these variables to the remaining evolutive environments, this method seems to provide a slight improvement (**Figure 10**).

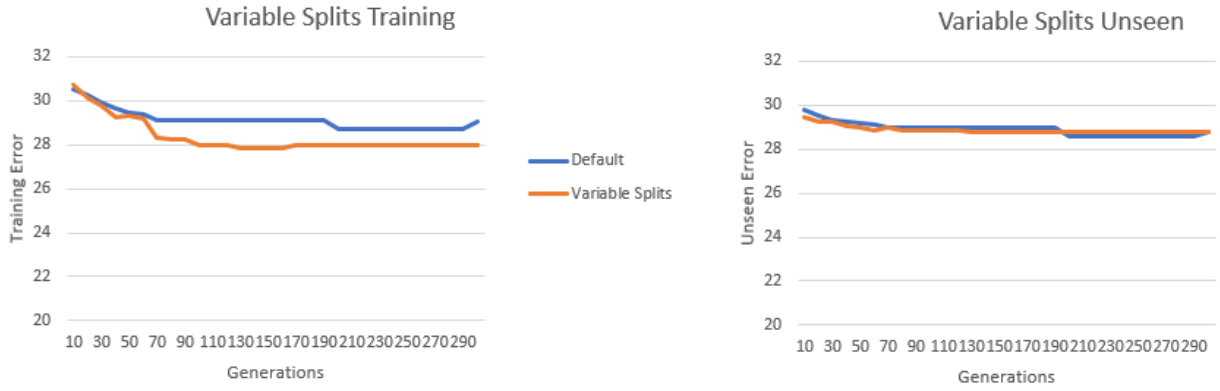

Figure 23. Variable Splits vs. default for both training and unseen error across different generations

Input Splits

On this method, we tried splitting the observations in 25% and 50%, none of them giving good results contrarily to what it was expected (**Figure 11**). For the final algorithm, we will test this technique synced with other methods, although after these results, we do not expect a much better performance.

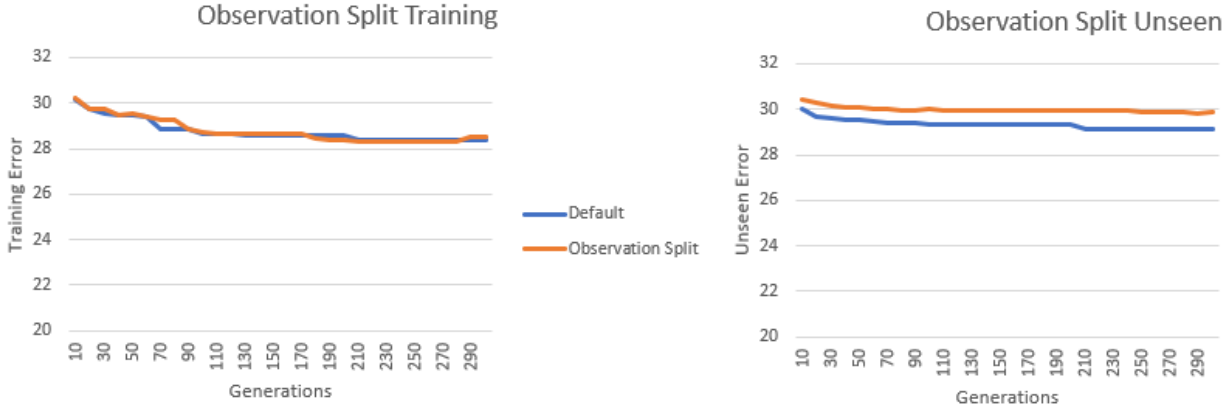

Figure 24. Input Splits vs. default for both training and unseen error across different generations

Central SSGP

The SSGP centralized was thought out to be used as a main connection between other methods. For that reason, we do not except good results by itself. Besides that, it tends to overfit the training data, which is logical considering that it is receiving the best training error individuals from the remaining populations (**Figure 12**). The main obstacle so far seems to be the lack of generalization ability from our 'main' population, even though, we provide it more than enough diversity of good individuals that could have potentially good generalization ability.

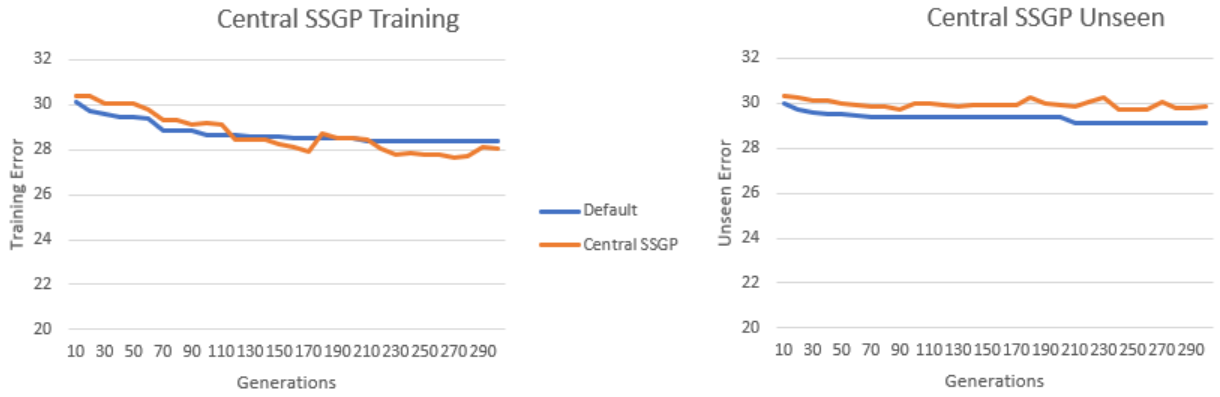

Figure 25. Centralized SSGP vs. default for both training and unseen error across different generations

For the remaining methods, we are not presented the results because they did not revealed in any way a major difference versus the default. Using all the operators was the same as using only the selected few while the variable memory showed, in no way, any improvement, not even on the initialization which is surprising result.

Final algorithm

The final algorithm chosen was using the roulette selection, the parsimony measure, the variable splitting of type 1 and the SSGP centralized. The specific parameters of number of populations, swap best every X generations, among others were not mentioned since they miss the main scope of this project, although they were tuned due to trial and error. In **Figure 13**, we have the final algorithm result compared to the rest of the methods tested. We can see that final algorithm was better with a training and unseen error under 26.

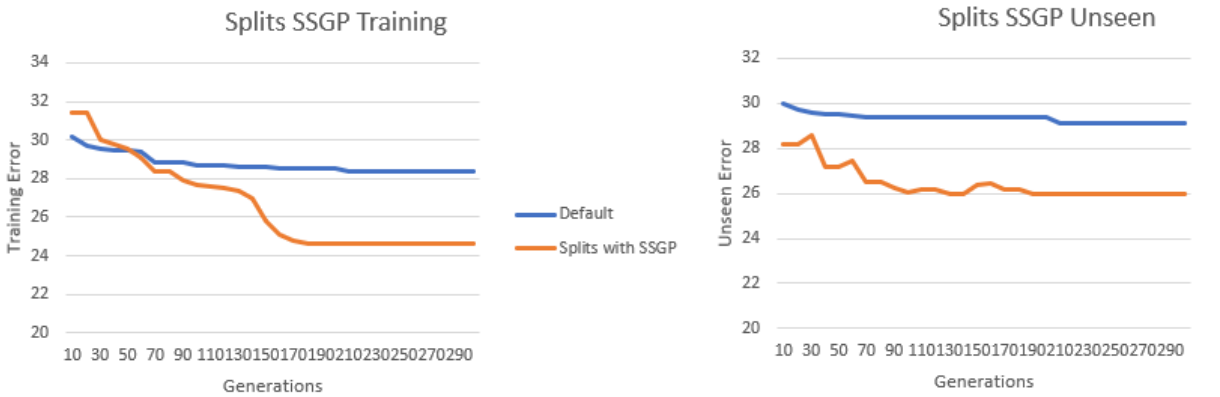

Figure 26. Variable Splits with SSGP vs. default for both training and unseen error across different generations

Final individual

The final individual has a training error of 23.73 and an unseen error of 26.67. It was obtained using the methods mentioned above with the slight exception that we used the operators 'average', 'reminder' and 'conditional' as well. Regarding the structure, the individual tree has a size of 490 and depth 35, which is really low when compared to the other individuals that attempt to achieve this kind of result.

Conclusions

Along this project the main problem was the lack of data when compared to the number of input variables, which makes it extremely difficult to obtain a good result, and even hard to get a result with an acceptable generalization ability. Another obstacle is the computational power required in the, very likely case, that the global optimum is a massive individual not possibly obtainable. In conclusion, we tried to focus on the best, smallest, generalizable individual which might have been a bit too ambitious in such a complex problem.

Bibliography

1. Kinnear Jr. K. Advances in Genetic Programming. Volume I. MIT Press Cambridge; 1994.
2. Luke S, Panait L. Lexicographic Parsimony Pressure. Proceedings of the Genetic and Evolutionary Computation Conference. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc.; 2002. pp. 829–836. Available: http://dl.acm.org/citation.cfm?id=646205.682619
3. Poli R, McPhee NF, Vanneschi L. Elitism reduces bloat in genetic programming. Proceedings of the 10th annual conference on Genetic and evolutionary computation - GECCO '08. Atlanta, GA, USA: ACM Press; 2008. pp. 1343–1344. doi:10.1145/1389095.1389355
4. Poli R, Langdon B, McPhee N. A Field Guide to Genetic Programming. Lulu Press; 2008.