Paper ML03

Symbolic Regression: Survival of the Best Fit

Hector Leitch, Veramed, Twickenham, United Kingdom

ABSTRACT

Machine learning is a field that has seen rapid growth and development over recent years in conjunction with increasing computer power. While a widely discussed topic, it is often unclear exactly how machine learning can be applied to practical problems we face. This paper introduces Symbolic Regression: an example of genetic programming applied to model selection. Genetic programming uses a population of individual models that are improved over several generations, resulting in very powerful prediction models. Symbolic Regression is built from the ideas of simple linear models, combining them to the framework of genetic programming. The key decisions the programmer must make, and the possible difficulties are also highlighted. The program is typically developed in a way that explores a much larger model space than traditional step-wise selection techniques and so there is also a chance to explore unknown and unexpected patterns within the data.

INTRODUCTION

There is no doubt that machine learning has earned its place as one of the hottest buzzwords in recent years. Driverless cars, Netflix recommendations, facial recognition, fraud detection; the list of applications is endless. With all the excitement, it is easy to get lost dreaming about hugely complex methods that are not realistically accessible for most.

This paper introduces Symbolic Regression, a powerful tool for model selection. By focusing on a well-known and common problem, we will demonstrate a more accessible route to using machine learning. This paper first introduces Genetic Programming, and then adds familiar concepts from simple linear regression to give a step-by-step walkthrough of Symbolic Regression. Furthermore, the advantages and disadvantages are discussed to highlight opportunities we might have for its use in the Pharmaceutical industry.

GENETIC PROGRAMMING

Genetic programming is a method in machine learning for optimizing programs, inspired by the natural selection of genes conferring advantageous attributes through evolution over generations. The rough idea is to have a population of programs that are gradually improved by 'natural' selection. Genetic Programming can be described in a very general sense by piecing together its components, so we will save mathematical representation and statistical models for later. For now, we will define the objective and components in simple terms and build an idea of how the program would run in general.

OBJECTIVE

The objective of Genetic Programming is simply to find a program that performs the given task well.

POPULATION AND GENERATIONS

The population is a collection of programs of fixed size. We say we have moved from the current generation to a new generation when we replace the programs in our population with new ones.

GENOTYPES AND PHENOTYPES

A genotype is the information or 'DNA' required to define a program in the population. This means that to create a new generation of the population, we only need to choose the genotypes rather than complete programs. A phenotype is a complete program; one that has been 'grown' from its genotype.

For example, if we have genotypes that consist of a few strings of explicit code then turning these into phenotypes would involve inserting these strings into a pre-determined program skeleton.

FITNESS FUNCTION AND SELECTION

The fitness function is a function used to evaluate how well the programs in the population perform. Choice of fitness function is very important, as a poor choice may hinder the speed of improvement between generations. The fitness function may consist of additional components to penalize slow or overly complicated programs, so that in general the programs being carried forward are useful in practice.

Given the fitness of each program in the population, we then select which programs will be discarded and which will contribute to creating the next generation. The method of selection is also an important choice and a good choice depends on the problem.

CROSSOVER AND MUTATION

Once selection of the fittest programs from the current generation has occurred, we are ready to create a new collection of genotypes and create the next generation. This stage is where we take inspiration from real life evolution and inheritance of genetic properties.

Crossover is where we take two halves from the genotypes of selected programs and combine them to form a new one. This simulates 'parent' programs passing half of their DNA to form a new 'child' program. By selecting the fittest solutions for crossover, we hope that features that lead to strong performing programs are carried through to the next generation through crossover.

Mutation is where we change part of a genotype at random to create a new one. This drives the exploration of different possible programs by introducing new features to the population that are not present in existing genotypes. The balance between crossover and mutation is something that should be tuned by the programmer to balance the exploration of programs through mutation and guidance towards high-performing programs through crossover.

TERMINATION CONDITION

The termination condition is a condition that is checked at each generation. This may use the fitness function to decide if an adequate solution has been found, in which case it would be checked just before selection occurs. It may be practical to include a limit for the number of generations in the termination criteria so that termination is guaranteed.

ALGORITHM

Now we have defined the features of Genetic Programming, the following algorithm outlines the overall process:

Input: Population size n, selection size $n_s < n$, fitness function, termination criteria
Output: A collection (size n) of programs

1. Initialize n genotypes at random.
2. Create programs (phenotypes) from the genotypes.
3. Evaluate the fitness of each phenotype with the fitness function.
4. If (termination criteria not satisfied) then:
    a. Select n_s fittest individuals.
    b. Perform crossover and mutation to create a new generation of n genotypes.
    c. Go back to step 2.

    Else:
    a. Output current generation of phenotypes.
    b. Terminate.

SYMBOLIC REGRESSION

This section of the paper builds Symbolic Regression by pairing the features defined for Genetic Program to features found in linear regression models and model selection. We will start with an introduction (or reminder) to linear regression.

LINEAR REGRESSION

Linear regression is a way of modelling a response variable (or dependent variable) y in terms of explanatory variables (or independent variables) $\boldsymbol{x}$ according to a linear predictor function. Without loss of generality, the predictors may include covariates, factors and interactions. Each observation of the response is modelled as a random variable:

$$Y_i = \beta_0 + \beta_1 x_{i1} + \cdots + \beta_p x_{ip} + \epsilon_i$$

Where $\epsilon_i \sim N(0, \sigma^2)$ is a random variable representing error or noise and $\beta_0, \ldots, \beta_p$ are unknown parameters; the formula without the error term $y_i = \beta_0 + \cdots + \beta_p x_{ip}$ is called the model formula which is used to predict the expectation

of Y_i. Given multiple observations in the vector $\boldsymbol{y}$ and the values of the predictor variables for each observation in a matrix $\boldsymbol{X}$, we can write this relationship using matrix notation as:

$$\boldsymbol{y} = \boldsymbol{X\beta} + \boldsymbol{\epsilon}$$

Given a model formula, we can estimate the unknown parameters from given data $\boldsymbol{y}$ and $\boldsymbol{X}$ according to assumptions about the error variables $\boldsymbol{\epsilon}$. These are:

1. That the errors are independent of each other.
2. That the variance σ^2 is constant.

After the unknown parameters have been estimated, we can check whether the assumptions appear to be valid. If not, we should select a different model.

STEPWISE SELECTION

In the case that we want to improve our choice of model, stepwise selection is a simple method to do so. Given a model formula $y = \beta_0 + \cdots + \beta_p x_p$ we consider new model formulae of the form $y = \beta_0 + \cdots + \beta_{p+1} x_{p+1}$ or $y = \beta_0 + \cdots + \beta_{p-1} x_{p-1}$ i.e. we remove or add a single term (covariate, factor or interaction term). Proposing new models in this way means that the model space (set of possible models) explored by stepwise selection only includes models with x_i terms coming directly from the data.

In order to say whether the new model is better than the current model, we typically use an estimator to compare models such as RMSE (root mean squared error) or AIC (Akaike Information Criterion). The choice of estimator may vary depending on the problem; some have higher penalties for more complex models. RMSE requires the original predictor values to be substituted into the model formula with parameter estimates for each record, creating what are known as the fitted values $\hat{\boldsymbol{y}}$. The RMSE is defined as:

$$RMSE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2}$$

GENETIC PROGRAMMING WITH LINEAR MODELS

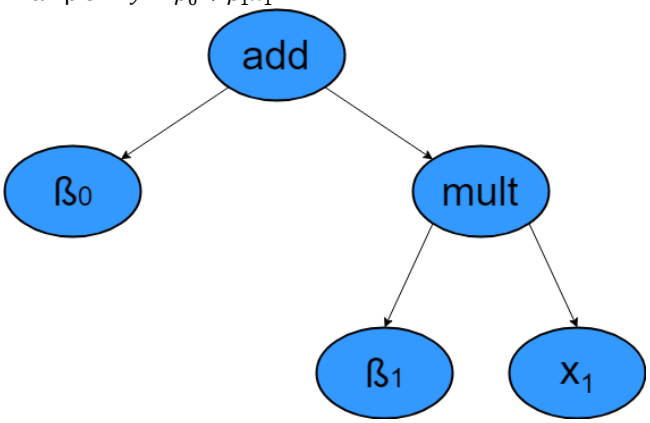
In the above sections, we have introduced concepts of linear models and model selection that can be paired with the building blocks of GP. The table below pairs key terms together with a short explanation:

GP Term	Linear Models Term	Explanation
Genotype	Model Formula	If we assume all models in the population hold the same assumptions with regards to the errors, all we need to define our model is the model formula. This means that crossover and mutation will involve moving and changing certain terms in the formula.
Fitness function	RMSE (or any alternative)	RMSE can be used to compare models and so is suitable as a fitness function. In this case, it is important to note that a smaller score is better.
Phenotype	Model with parameter estimates and fitted values	The fitness function is always a function of the phenotype, so our phenotype consists of all the arguments we need for the fitness function. We also include the parameter estimates with each model in the population so that they are available when the algorithm terminates.

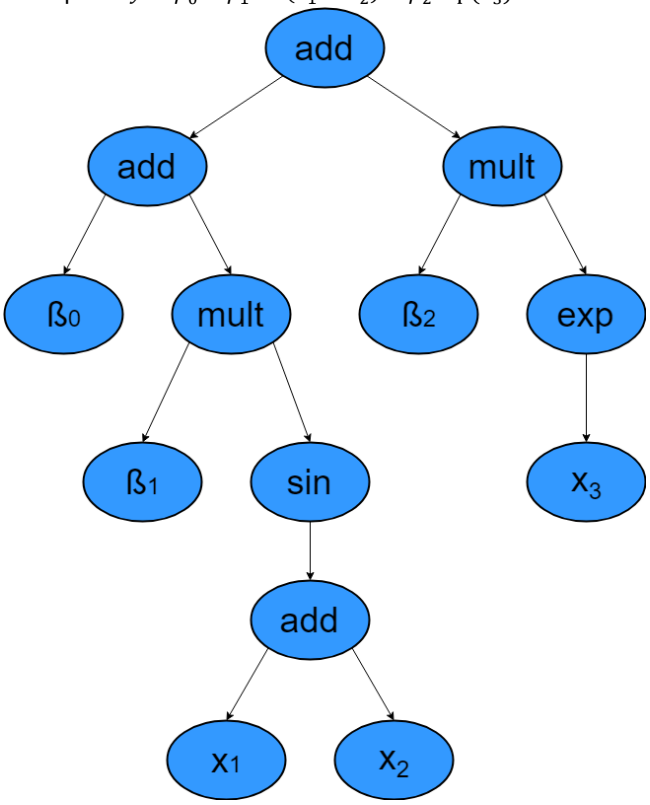
REPRESENTING A MODEL FORMULA

When we create a new population of models, it is the genotypes or model formulae that must be specified. In Symbolic Regression, an effective way to represent model formulae is to use inverted trees. Expressions in software are often represented in trees or lists as they can be evaluated recursively by the program. In tree representation, each node is a binary operator $(+, -, \times, \div)$, an analytic function $(sin, cos, exp, log, \ldots)$, or a constant or variable. By including analytic functions and division, the model space we can explore by adding more terms becomes much greater than in stepwise selection. The examples below show how we can represent both simple and complex model formulae with trees.

Example 1: $y = \beta_0 + \beta_1 x_1$



Example 2: $y = \beta_0 + \beta_1 \sin(x_1 + x_2) + \beta_2 \exp(x_3)$



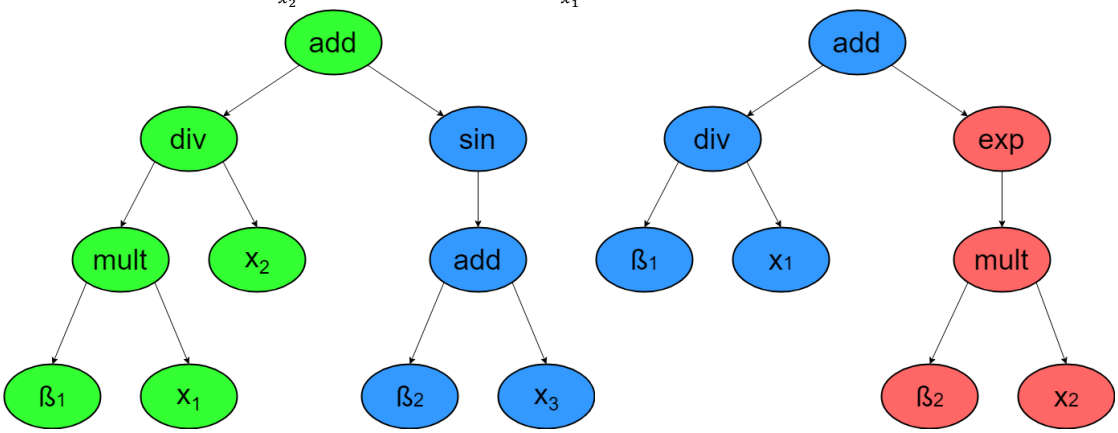
Notice that for each formula, there is a single 'root' node where its edges are only directed away from it. Edges directed away from a node represent the inputs, so a binary operator (+,-,*,/) will have two edges connecting to its inputs, an analytic function (sin, cos, exp, log) will have one edge connecting to its input, and constants, parameters and variables will have no edges directed away from them (i.e. they will be leaves of the tree). This provides us with an easy way to check if a given formula is valid.

The genetic operations, crossover and mutation, that drive the learning process in Genetic Programming are also easy to define and visualize using tree representation.

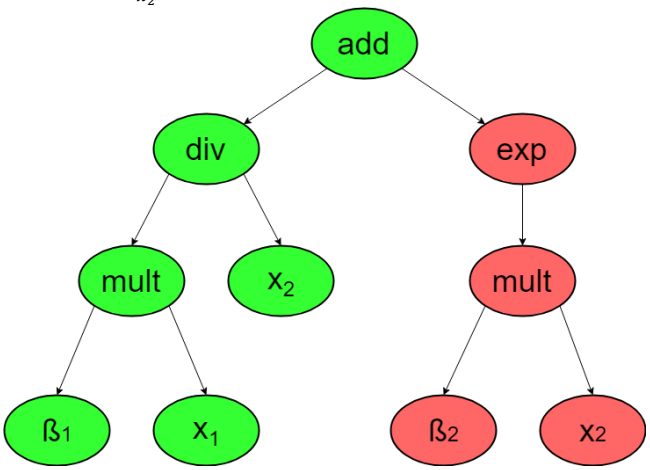
Crossover can be defined as follows: Select two 'parent' trees and find the root node of each. From the root node of each tree, randomly select one of the two subtrees connected to the root node. Finally, remove the selected subtree

from one of the parent trees and replace it with the subtree selected from the other parent tree. To better visualize this process, see the example below.

Crossover example: $y = \frac{\beta_1 x_1}{x_2} + \sin(\beta_2 + x_3)$ and $y = \frac{\beta_1}{x_1} + \exp(\beta_2 x_2)$

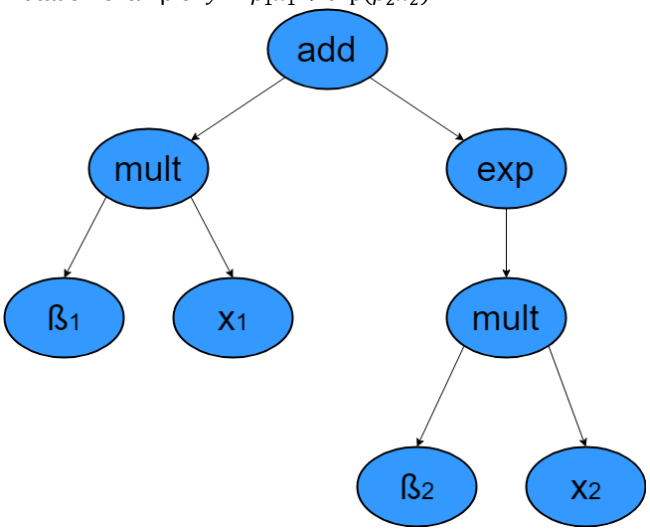


Create $y = \frac{\beta_1 x_1}{x_2} + \exp(\beta_2 x_2)$

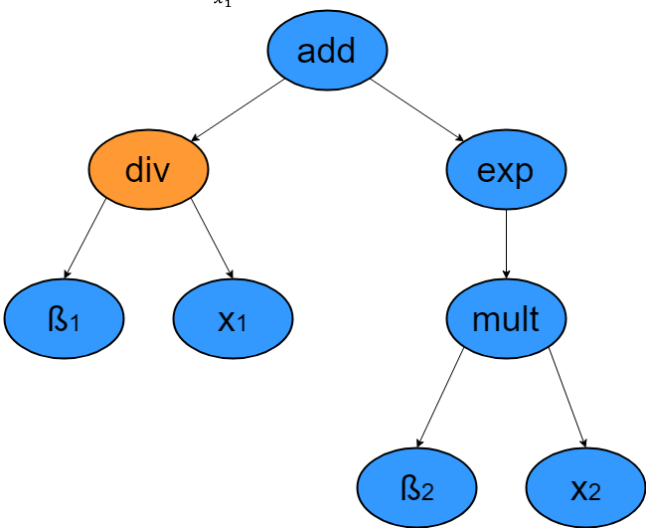


Mutation on trees is very simple: select any node from a given tree and randomly replace it such that the resulting tree still represents a valid formula (i.e. the replacement node will have the same number or edges as the original node). Again, we illustrate with a simple example.

Mutation example: $y = \beta_1 x_1 + \exp(\beta_2 x_2)$



May mutate to: $y = \frac{\beta_1}{x_1} + \exp(\beta_2 x_2)$



TERMINATING THE PROGRAM

Symbolic Regression is a hill-climbing algorithm, where the fitness of the best individual is improved over iterations by searching the model space and 'climbing' when an improvement is found. Because Symbolic Regression explores such a large model space, it is impossible to guarantee that we will find the global maximiser to the fitness function and instead we are likely to converge to a local maximum. We aim to choose criteria for termination that avoids stopping too early, for example at a local maximum that is not of satisfactory quality, but also avoids spending too long searching when no improvement will be made in reasonable time.

We have several options for termination, including:

1. A satisfactory model is found
2. There have been a predetermined number of new generations with no further improvement to the best model in the population
3. A specified maximum number of generations have been created

A good idea is to use a combination of multiple criteria, as the process can be computationally expensive with a large population.

SOFTWARE PACKAGES AND EXAMPLE

Symbolic Regression is mostly found as part of packages designed for Genetic Programming. One package that yielded satisfying results fairly quickly was the `gplearn`[1] module for Python. The documentation came with a simple example of rediscovering a formula used to generate some data. Below is a similar example, with noise added to the data to obscure the exact true formula and test whether it can still be recovered.

Task:
We generate data according to the formula:

$$y = \sin(x_0) - \log(x_0 + x_1) + noise$$

The input data will contain only x_0, x_1 and y so the program will have to find the correct transformations to yield a program that is close to the original.

Code (with annotations):

```
# Import modules
from gplearn.genetic import SymbolicRegressor
from sklearn.utils.random import check_random_state
import numpy as np
import graphviz

# Create 100 observations of (x0, x1, y) according to
# y = sin(x0) - log(x0+x1) + error
rng = check_random_state(0)

X_train = rng.uniform(1, 10, 200).reshape(100,2)
error = np.random.normal(0,0.1,100)
y_train = np.sin(X_train[:,0]) - np.log(X_train[:,0] + X_train[:,1]) + error[:]

# Symbolic Regression
est_gp = SymbolicRegressor(
            population_size=5000,
            generations=20,
            metric='rmse',
            stopping_criteria=0.01,                                  } [1]
            function_set=('add','sub','mul','div','sin','cos','log'),} [2]
            p_crossover=0.7,
            p_subtree_mutation=0.1,
            p_hoist_mutation=0.05,
            p_point_mutation=0.1,                                    } [3]
            max_samples=0.9,                                         } [4]
            verbose=1,
            parsimony_coefficient=0.01,                              } [5]
            random_state=0,
            n_jobs=2,
            low_memory=True)

est_gp.fit(X_train, y_train)
```

[1] The `metric` option is the fitness function, set to RMSE. The `stopping_criteria` is the value that the RMSE must drop to for the program to terminate early. We have set this unreasonably low so that we can see how the program evolves over 20 generations.

[2] We have specified a small `function_set` that includes the ones the program will need to complete the task. In the `gplearn` package we also have the option to define our own custom functions.

[3] The `p_:` options give us control over the mix of crossover and mutation. Here there are 3 types of mutation; as well as point mutation, there are some more complex subtree mutations available.

[4] `max_samples` is a proportion of the data that the fitness of each program is evaluated on. Setting this to <1 will affect the output as shown below.

[5] The `parsimony_coefficient` is an added penalty component to the fitness function that penalises the length of the program.

The following output is produced as the program runs:

```
C:\Users\my laptop>python gp_demo.py
    |   Population Average    |             Best Individual               |          |
---- ------------------------- ------------------------------------------- ----------
 Gen    Length          Fitness      Length          Fitness      OOB Fitness  Time Left
   0     16.41          1256.72          16         0.668868         0.523743      2.35m
   1     10.35          4.24544           7         0.454296         0.424384      2.69m
   2      8.02          3.06016          11         0.167011         0.143822      2.76m
   3      5.23          2.35589          11         0.163258         0.178441      2.52m
   4      6.68          2.42705          13         0.100454          0.13603      2.39m
   5      7.16          2.19775          13         0.100633         0.134836      2.23m
   6      7.74          2.97218          12        0.0999695         0.142125      2.08m
   7     10.20          3.68371          11        0.0992991         0.148291      2.01m
   8     11.15          3.69579          13        0.0938386         0.159005      1.89m
   9     11.28          3.37919          11        0.0954348         0.169596      1.67m
  10     10.79          3.26476          11        0.0979742         0.145913      1.51m
  11      9.98          4.34642           9        0.0965168         0.149869      1.38m
  12      9.29           3.0286           9        0.0945621         0.160693      1.18m
  13      8.98          3.37017           9        0.0929515         0.168939     59.71s
  14      8.93          4.81803           9        0.0932181         0.167612     48.43s
  15      8.99          3.32473           9        0.0949927         0.158391     40.06s
  16      8.95           3.3789           9        0.0946718         0.160111     30.42s
  17      8.97          3.66818           9        0.0947525         0.159681     20.37s
  18      9.07          4.33807           9        0.0938311         0.164504     10.25s
  19      9.01           3.1354           9        0.0956595         0.154737      0.00s
```

The output shows the list length (number of nodes in the tree representation) and fitness score for the best individual of each generation, and the population averages. Because we evaluated the model on a subset of the data, we also get an 'Out of Bag' fitness for the best individual, where the model is evaluated on the rest of the data too. This can warn us if the model is overfitting to the original subset.

Result:
Unfortunately, `gplearn` does not allow us to look at the entire population and only shows us the best individual program. This is contained in the `_program` object:

```
# Print the resulting formula in list form
print(est_gp._program)
```

```
sub(0.022, sub(log(add(X1, X0)), sin(X0)))
```

To check that this program is close to our original program, we need to expand out the list:

$$y = 0.022 - (\log(x_1 + x_0) - \sin(x_0))$$
$$= 0.022 + \sin(x_0) - \log(x_0 + x_1)$$

The main terms are all present, plus a negligible intercept term.

ADVANTAGES, DISADVANTAGES AND USE IN THE PHARMACEUTICAL INDUSTRY

ADVANTAGES

Symbolic Regression has obvious advantages by design: it is designed to explore a wide space of possible models so the likelihood that it will find one that fits the data well is very high, and it can be tuned through the fitness function to appropriately penalize overfitting to training data. Including functions such as log() and sin() give the algorithm the ability to try out various transformations of each variable, which without the context of the data can be very difficult to find.

If selecting the best model is our goal, we can take the best individual from the final generation as in the example using `gplearn`. However, there is also the option to set up a program that retains the whole population and look at a collection of the best models from the final generation.

DISADVANTAGES

Symbolic Regression is very computationally expensive, which is why it is a good idea to include number of generations or program run time in the termination criteria.

There are also many parameters that need to be decided and which can influence the learning speed, such as the balance between mutation and crossover, which require experience in order to tune properly. In the example program

using `gplearn`, the best individual model is very sensitive to changes in the tuning parameters as there are not many generations.

HOW CAN WE USE IT?

The advantages of Symbolic Regression make it very good at context-free discovery of relationships within data. If we do not know much about what relationships may be in our data, Symbolic Regression offers a selection of candidates for us to consider exploring further. If we already have data where variables are properly transformed and scaled according to context, the advantage gained by searching a wider model space may not be worth the extra programming and computational cost. Therefore, a potential scenario where Symbolic Regression would perform well would be as a tool in exploratory analysis for proposing questions to be investigated further in the early stages of drug development.

Examples of Symbolic Regression in the pharmaceutical industry have started to emerge already. Symbolic Regression has been tested as a tool for developing QSAR (Quantitative Structure Activity Relationship) models for toxicity[2] and predictive pharmacokinetic models[3].

REFERENCES

[1] Trevor Stevens (2019) gplearn Documention v0.4.1, Retrieved from: https://buildmedia.readthedocs.org/media/pdf/gplearn/stable/gplearn.pdf

[2] Charles Hii, Dominic P. Searson and Mark J. Willis (2011) Evolving Toxicity Models using Multigene Symbolic Regression and Multiple Objectives. *International Journal of Machine Learning and Computing, Vol. 1, No. 1,* Retrieved from http://ijmlc.org/papers/05-L0037.pdf

[3] Archetti, F., Lanzeni, S., Messina, E. et al. Genet Program Evolvable Mach (2007) 8: 413. https://doi.org/10.1007/s10710-007-9040-z

ACKNOWLEDGMENTS

Andrew Holmes and Diana Stuart, Veramed, for supporting me in writing and reviewing.

Draw.io – a handy website for making flow diagrams.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Hector Leitch
Veramed Ltd.
5th Floor, Regal House, 70 London Rd.
Twickenham / TW1 3QS
Email: hector.leitch@veramed.co.uk