GPLearn Testing Report

December 6, 2019

Animesh KC
Emily/Lee Ingram
Glenn Rhodes

Abstract

Software testing is an integral part of the software development life cycle helping to ensure a functional product which meets user requirements and produces a reliable output. To demonstrate the importance of rigorous testing we tested the genetic programming python library GPLearn. To test GPLearn various testing methods covering both black box and white box testing were employed. For black box testing the primary methods utilized were equivalence class testing and error guessing. Equivalence classes were used to cover the possible function set GPLearn accepts while error guessing was used to extend test coverage further for functions embedded within functions. Equivalence class testing did not result in any faults being found; however, error guessing was able to identify faults. These faults were the program being unable to meet the desired fitness and occured when using complex chained embedded functions. White box testing utilized branch coverage testing to test the core methods of the SymbolicRegressor class. All white box testing was completed without detecting any faults; although, only partial branch coverage of the core methods of the SymbolicRegressor class were possible, due to time constraints.

Introduction

The software under test is the GPLearn python toolkit. GPLearn implements a python genetic program toolkit specifically designed for symbolic regression problems. Symbolic regression is a form of analysis that attempts to find an expression of best fit for a given data set by combining mathematical operators from a pool of mathematical expressions. Genetic programming extends this by iterating for several generations in which the initial population of randomly generated possible solutions (individuals) are either mated with another individual, mutated, or both. The fitness of all individuals is then ranked and if none of the individuals achieve the desired level of fitness another generation is executed, with a biased selection so the best fit individuals make up the majority of the population for the next generation. The mating process consists of selecting two individuals and recombining them to produce two new offspring analogous to biological mating. The mutation process requires only a single individual and alters one or more gene of the individual to create a new individual.

Black Box

Black Box testing for GPLearn aims to look at the different types of functions that could be modelled by the library. GPLearn's function set includes addition, subtraction, multiplication, division, sine, cosine, tangent, logarithm, square root, inverse, maximum, and minimum. To test these functions, equivalence classes were chosen as the primary method, as functions can be partitioned into these 12 categories. Each equivalence class is defined by its most exterior function. For example, for the function sin(max(x+y)), this will belong to the sine equivalence class, whereas max(sin(x) + sin(y)) will belong to the maximum equivalence class. This enables the equivalence classes to be mutually exclusive and collectively exhaustive for cases where one function is embedded within another. Table 1 of the appendix showcases the equivalence classes for GPLearn.

GPLearn's invalid classes include areas where logarithm, division, inverse, and square root are undefined. GPLearn implements protected outputs for these cases so that a particular result in the test data or function generation does not break the program. For example, for the function x/y, GPLearn outputs 0 when -0.001 <=y <= 0.001 so that the program does not break from computing division by zero. However, this would lead to skewed results hence making testing these inputs separately a necessity.

The test cases chosen for equivalence testing were simple ones that lie within these equivalence classes, in order to determine whether GPLearn is in general effective for mapping these functions. However, further error guessing tests were done with functions that GPLearn was more likely to have problems with modelling. To test GPLearn with the created test cases, the tester must first create a set of training and testing data that GPLearn will use to generate its function and evaluate its accuracy respectively. The training and testing data used were randomly generated for values that lie within the domain of the test. Each test used 50 randomly generated x and y values for both the training and test data, and for the sake of consistency, each equivalence class was tested with 2 input variables, even though some functions only require one variable. As a result, 3D plots could also be generated for every function. To evaluate the accuracy of the generated function compared to the testing data, GPLearn outputs an R^2 score. It is expected that valid equivalence classes will achieve an R^2 score above 0.9, as that would entail the generated function is sufficiently close to the tested function. A value of 1.0 represents that the generated function is identical to the tested function. Negative R^2 values are possible for results where the test data does not follow the trend of the generated function trend. For invalid equivalence classes, an R^2 value below 0.5 is expected, including negative values. A maximum of 20 generations were used for each function, but many functions will not require that many generations.

Table 2 of the appendix shows the equivalence class test cases, while Table 3 shows the error guessing test cases. Any test cases that are not highlighted passed. All valid equivalence classes passed with an R^2 value of at least 0.998, and all invalid classes achieved a score below 0.5 as expected. For error guessing however, there were significant errors. Three of the 6 error guessing test cases led to R^2 values less than 0.9, and one of them featured a worse R^2 score than even any invalid equivalence class.

The errors found by error guessing can be explained as limitations within GPLearn's function generation. Some functions are too complex for the software to accurately determine the function within 20 generations. It could be possible that with further generations, the results would be more accurate, but clearly there are time constraints in testing results that prevent iterating through an indefinite number of generations, especially for sufficiently large data sets. However, it has been seen that for the majority of functions that a user may test, GPLearn will be sufficiently accurate. Most users will not require a chain of embedded functions, and even in this case, the simpler embedded functions work properly, as seen by the error guessing test cases that passed. Ultimately, although GPLearn's function generation is not perfect for a limited number of test generations, it performs as expected for the majority of functions that a user may need. Further testing may involve much larger generation sizes, to see whether results are more favorable in that case.

White Box

The white box testing, in this instance, will be carried out through the use of Control Flow Graphs (CFDs). The CFD coverage criterion used will be Branch Coverage. Branch Coverage makes sense in this case as GPLearn is a complex module with many interrelated parts, and due to the control flow structure of GPLearn, Statement Coverage and Branch coverage are equivalent. In fact, GPLearn is so complexly written that only specific section of the code will be considered for testing. The criteria for choosing which sections of code are tested is if they are both related to the main functionality of GPLearn and if the results of those code sections can be consistently interpreted by a test script; with more time, testing larger portions of the library is possible. In addition to only testing the SymbolicRegressor class, only its fit, predict, and __str__ methods will be tested, and as the fit method contains non-mission-critical code, sections of the fit method will also be ignored to save time.

The actual testing was done by a Python module named Pytest. Pytest is similar to JUnit and its counterparts; however, Pytest better utilizes follows Python's design philosophy (simpler is better). Except for the pytest.raises method, all testing in this instance is done with Python's built-in 'assert' statement.

As one may expect from a module that is commonly used in its applicable field, GPLearn has passed all Branch-complete tests. It's possible that omission of non-mission-critical code has in-turn obfuscated important faults, but given that these tests focus on the testable infrastructure needed to run a basic genetic programming algorithm, any hidden faults aren't likely to severely impact the GPLearn library.

Conclusion

Testing of GPLearn exposed a small number of minor faults, though on the whole GPLearn proved to be a highly robust and well implemented python library for genetic programming. Black box testing through equivalence classes found no faults with the output results of valid equivalence classes always falling within the acceptable error and invalid equivalence classes behaving as expected. The faults which were found were all found using the black box error guessing method and occured when more complicated nested functions were provided as input with GPLearn unable to reach the acceptable error within the fixed number of generations. White Box branch coverage testing revealed no faults within the GPLearn methods that were examined.

Given the complexity of the SymbolicRegressor class, more thorough white box testing is possible, but extremely time consuming. As such it is only possible to draw conclusions about the library's reliability from a few core methods. This indicates that a significant chunk of GPLearn is functionally correct, but not giving any indications as to the state of the rest of the code. For black box testing, it isn't entirely certain how functions embedded within other functions will behave. It was assumed that exterior functions of the same type would behave the same way, but it is evident that this isn't the case, which makes partitioning them into thorough equivalence classes difficult. The stochastic nature of GPLearn also makes balck box testing difficult as the exact output for any given input cannot be certain whether a defined margin of error is achieved can be tested..

As GPLearn has very complex code, further white box testing may prove cost-prohibitive. As such, applying mutation testing to GPLearn's existing test files may prove more fruitful. If any mutants are left alive from the initial mutation tests, then one need only create more JUnit-like tests to deal with the uncovered sections of code, and kill the remaining mutants. Although more thorough black box testing could be done, this would be time-consuming and wouldn't be likely to lead to more efficient testing.

GPLearn has a few other less essential features that may be worth testing if given more time. For example, it could be fruitful to test GPLearn's function factory and fitness factory functions that enable the user to set their own functions or fitness metrics.

Appendix

Table 1: Black Box Equivalence Classes

Note: gpn refers to built in GPLearn functions; the functions are numbered based on how many parameters they take

Input Condition	Regular Classes	Invalid Classes
Addition: x1 + x2	x0, x1 E R (1)	
Subtraction: x1 - x2	x0, x1 E R (2)	
multiplication: x1*x2	x0, x1 E R (3)	
division: gpn.div2(x1,x2)	x0 E R, x1 < -0.001 (4) x0 E R, x1 > 0.001 (5)	x0 E R, -0.001 <= x1 <= 0.001 (6)
square root: gpn.sqrt1(x1)	x0 >=0 (7)	x0 < 0 (8)
logarithm: gpn.log1(x1)	x0 >= 0.001 (9)	x0 < 0.001 (10)
absolute: gpn.abs1(x1)	x0 E R (11)	
maximum: gpn.max2(x1, x2)	x0, x1 E R (12)	
minimum: gpn.min2(x1, x2)	x0, x1 E R (13)	
sine: gpn.sin1(x1), x1 in radians	x0 E R (14)	
cosine: gpn.cos1(x1), x1 in radians	x0 E R (15)	
tangent: gpn.tan1(x1), x1 in radians	x0 E R (16)	
inverse: gpn.inv1(x1)	x0 < -0.001 (17) x0 > 0.001 (18)	-0.001 <= x0 <= 0.001 (19)

Table 2: Black Box Equivalence Test Cases:

ID & Description	Input and test domain	Expected R^2 compared to Test Data	Actual R^2 compared to test data
1 Testing the sinusoidal functions together along with the basic subtraction to join the terms (Classes 2, 14, 15)	z = gpn.cos1(x0) - gpn.sin1(x1) -10 <= x0 <= 10 -10 <=x1 <= 10	>0.9	1.0
2 Testing the tangent function (Class 16)	z = gpn.tan1(x0 + x1) -10 <= x0 <= 10 -10 <= x1 <= 10	>0.9	1.0
3 Testing division and inverse where the denominator is below 0. Testing addition to join terms. (Classes 1, 4, 17)	z = gpn.div2(x0, x1) + gpn.inv1(x1) -10 <= x0 <= 10 -10 <= x1 < -0.001	>0.9	1.0
4 Testing division and inverse where the denominator is above 0. Testing multiplication to join terms (Classes 3, 5, 18)	z = gpn.div2(x0, x1) * gpn.inv1(x1) -10 <= x0 <= 10 0.001 < x1 <= 10	>0.9	1.0
5 Testing Square Root of positive numbers (Class 7)	z = gpn.sqrt1(x0 + x2) 0 <= x0 <= 5 0 <= x1 <= 5	>0.9	1.0
6 Testing logarithms of positive numbers (Class 9)	z = gpn.log1(x0 / x1) 0.001 <= x0 <= 10 0.001 <= x1 <= 1	>0.9	1.0
7 Testing absolute value function (Class 11)	z = gpn.abs1(5*x0 - 2* x2) -10 <= x0 <= 10 -10 <= x1 <= 10	>0.9	0.998

8 Testing max and min functions (Classes 12 and 13)	z = gpn.max2(x0, 5*x1) * gpn.min2(4*x0, x1) -10 <= x0 <= 10 -10 <= x1 <= 10	>0.9	0.998
9 Testing Invalid class 6	z = gpn.div2(8*x0, x1) -10 <= x0 <= 10 -0.001 <=x1 <=0.001	<0.5	-0.07
10 Testing Invalid class 8	z = gpn.sqrt1(x0 - 2x1) -10 <= x0 < 0 0 < x1 <= 10	<0.5	-0.19
11 Testing Invalid class 10	z = gpn.log1(x0 * x1) -10 <= x0 < 0 0 < x1 <= 10	<0.5	-0.19
12 Testing Protected class 19	z = gpn.inv1(x0 - x1) -10 <= x0 <= 10 -0.001 <= x1 <= 0.001	<0.5	0.05

Table 3: Black Box Error Guessing Test Cases

Note Highlighted cases indicate failed test cases

ID & Description	Input and test domain	Expected R^2 compared to Test Data	Actual R^2 compared to test data
1 Testing trigonometric functions within themselves	gpn.sin1(x0 + gpn.cos1(x1)) -10 <= x0 <= 10 -10 <=x1 <= 10	>=0.9	0.39
2 Testing division with embedded functions Choose a domain that will not cause division by zero or logarithm error	gpn.div2(gpn.log1(5*x0 +8), gpn.log1(6*x1 - 2)) -1 <= x0 <= 10 2 < x1 < 10	>=0.9	0.9
3 Testing square root with embedded functions	gpn.sqrt1(gpn.cos1(x0) * gpn.inv1(gpn.sin1(x1)) 0 <= x0 <= 10 0 <= x1 <= 10	>=0.9	0.81

4 Testing maximum and minimum within themselves	gpn.max2(gpn.min2(gpn. .cos1(x0), gpn.sin1(x1)), gpn.tan1(x0/x1)) 0 <= x0 <= 10 0 <= x1 <= 10	>=0.9	1.0
5 testing absolute with embedded functions	gpn.abs1(inv1(x0) * log1(8*x1)) 0.001 < x0 <= 10 0 < x1 <= 10	>=0.9	0.62
6 Testing full assortment of functions together	z = gpn.div2(x0, x1) +gpn.cos1(2x0) - gpn.sin1(3x1) * gpn.max1(6x0 +10, 2x1 - 8) - gpn.min1(x1 + 5, 3x0) + gpn.inv1(x1) - gpn.tan1(x0) * gpn.log1(x0) + gpn.sqrt1(x0) * gpn.abs1(-2x0) 0.001 <= x0 <= 10 -10 <=x1 < -0.001	>=0.9	-27.14

Table 4: White Box Test Cases

Test ID	Description	Expected	Actual
TSR.str.before_fit	str(sr) == repr(sr)	True	True

TSR.str.after_fit	sr.fit(data)
	is_gplearn_equation(str(sr))
		True	True

TSR.predict.before_fit	sr.predict(data)	NotFittedError	NotFittedError

TSR.predict.after_fit.1	sr.fit(one_feature_data)
	sr.predict(two_feature_data)
		ValueError	ValueError

TSR.predict.after_fit.2	sr.fit(data) type(sr.predict(data)) == float	True	True
TSR.predict.after_fit.3	sr.fit(one_label_data) sr.predict(data).shape == 1 by none	True	True
TSR.fit.function_set_init.non_existant_built_in_function	Sr.function_set = [<not defined function string>] sr.fit(data)	Sr.fit raises value error	Sr.fit raises value error

TSR.fit.function_set_init.existant_built_in_function.1	Sr.function_set = [<defined function string>]
	sr.fit(data)
	<defined function string> in gplearn._function_map
		True	True

TSR.fit.function_set_init.existant_built_in_function.2	Sr.function_set = [<defined function string>]
	sr.fit(data)
	<defined function string> in sr._function_set
		True	True

TSR.fit.function_set_init.add_new_function	User_f = make_user_func()
	Sr.function_set = [user_f]
	sr.fit(data)
	User_func in sr._function_set
		True	True

TSR.fit.function_set_int.add_invalid_function	Sr.function_set = [<not correctly defined user func>] sr.fit(data)	ValueError	ValueError

TSR.fit.function_set_init.empy_function_set	Sr.function_set = []
	sr.fit(data)
		ValueError	ValueError

TSR.fit.fit_metric_sel.user_def_metric	User_m = make_user_metic()
	Sr.metric = user_m
	sr.fit(data)
	Sr._metric == user_m
		True	True

TSR.fit.fit_metric.incorrect_fit_metric_name	Sr.metric = <inccorrect user specified metric> sr.fit(data)	ValueError	ValueError
TSR.fit.genetic_mod_op_probs_sum_gt_one	Sr.p_crossover = <value gt than one> sr.fit(data)	ValueError	ValueError

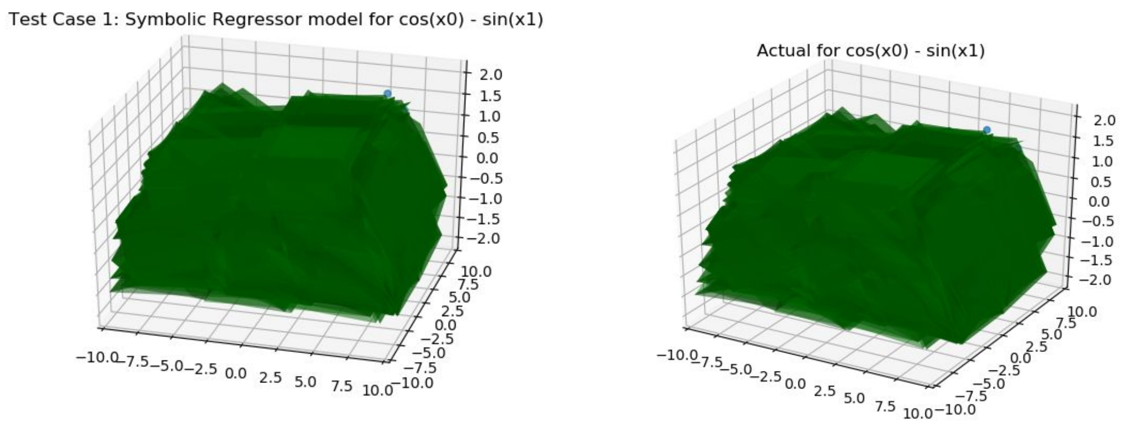
TSR.fit.invalid_pop_init_method	Sr.init_method = <incorrect init method>
	sr.fit(data)
		ValueError	ValueError

TSR.fit.invalid_constant_range	Sr.const_range = <invalid_range_tuple>
	sr.fit(data)
		ValueError	ValueError

TSR.fit.program_init_depth.invalid_range_tuple	Sr.init_depth = <incomplete range tuple>
	sr.fit(data)
		ValueError	ValueError

TSR.fit.program_init_depth.invalid_range	Sr.init_depth = <incorrect range tuple>
	sr.fit(data)
		ValueError	ValueError

TSR.fit.feature_name.more_names_then_features	sr.feature _names = <list of two feature names> sr.fit(<single feature data>)	ValueError	ValueError
TSR.fit.feature_name.wrong_type	Sr.feature_names = <invalid type for feature name> sr.fit(data)	ValueError	ValueError



Figures 1-2: Black Box testing plots for a single case

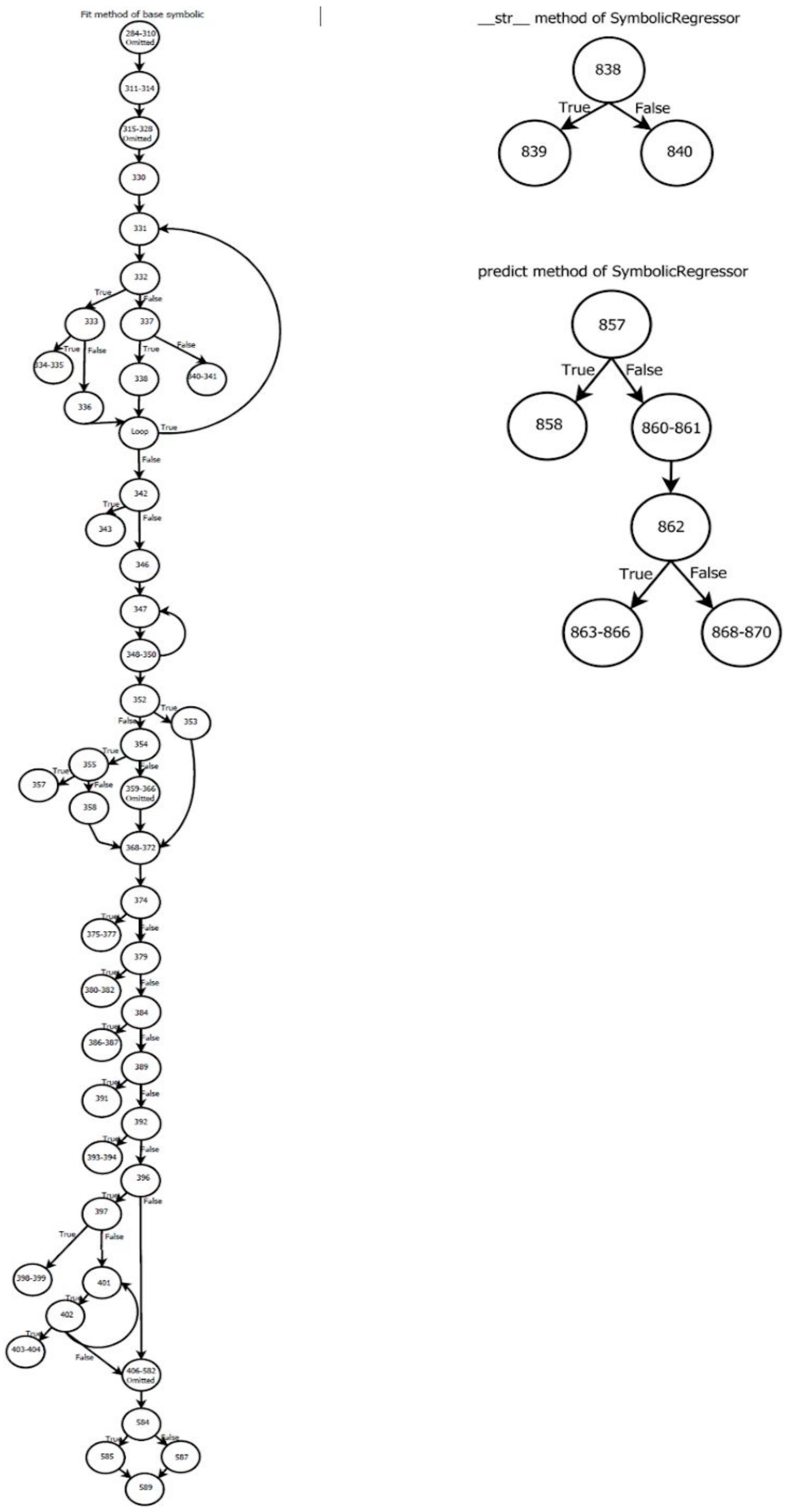


Figure 3-5: Control Flow Diagrams
Reduced CFD for Fit method on left. CFDs for __str__ and Predict on right