gplearn Documentation

Release 0.3.0

Trevor Stephens

Jun 07, 2018

Contents

Genetic Programming in Python, with a scikit-learn inspired API: gplearn

> *One general law, leading to the advancement of all organic beings, namely, multiply, vary, let the strongest live and the weakest die.*
>
> *–Charles Darwin, On the Origin of Species* (1859)

`gplearn` implements Genetic Programming in Python, with a scikit-learn inspired and compatible API.

While Genetic Programming (GP) can be used to perform a very wide variety of tasks, `gplearn` is purposefully constrained to solving symbolic regression problems. This is motivated by the scikit-learn ethos, of having powerful estimators that are straight-forward to implement.

Symbolic regression is a machine learning technique that aims to identify an underlying mathematical expression that best describes a relationship. It begins by building a population of naive random formulas to represent a relationship between known independent variables and their dependent variable targets in order to predict new data. Each successive generation of programs is then evolved from the one that came before it by selecting the fittest individuals from the population to undergo genetic operations.

`gplearn` retains the familiar scikit-learn `fit`/`predict` API and works with the existing scikit-learn pipeline and grid search modules. The package attempts to squeeze a lot of functionality into a scikit-learn-style API. While there are a lot of parameters to tweak, reading the documentation here should make the more relevant ones clear for your problem.

`gplearn` currently supports regression through the *`SymbolicRegressor`* as well as transformation for automated feature engineering with the *`SymbolicTransformer`*, which is designed to support regression problems, but should also work for binary classification. Future versions of the package will expand this class to support more complicated multi-target classification problems, and much more is planned too.

`gplearn` is built on scikit-learn and a fairly recent copy (0.18.1+) is required for installation. If you come across any issues in running or installing the package, please submit a bug report.

I hope you get some excellent results from using `gplearn`! If you do, please drop me a line on my blog about how you used it.

Next up, read some more details about *what Genetic Programming is*, and how it works. . .

Contents:

CHAPTER 1

Introduction to GP

Genetic Programming in Python,
with a scikit-learn inspired API:
gplearn

Owing to this struggle for life,

any variation, however slight and from whatever cause proceeding,

if it be in any degree profitable to an individual of any species,

in its infinitely complex relations to other organic beings and to external nature,

will tend to the preservation of that individual,

and will generally be inherited by its offspring.

– Charles Darwin, On the Origin of Species (1859)

gplearn extends the scikit-learn machine learning library to perform Genetic Programming (GP) with symbolic regression.

Symbolic regression is a machine learning technique that aims to identify an underlying mathematical expression that best describes a relationship. It begins by building a population of naive random formulas to represent a relationship between known independent variables and their dependent variable targets in order to predict new data. Each successive generation of programs is then evolved from the one that came before it by selecting the fittest individuals from the population to undergo genetic operations.

Genetic programming is capable of taking a series of totally random programs, untrained and unaware of any given target function you might have had in mind, and making them breed, mutate and evolve their way towards the truth.

Think of genetic programming as a stochastic optimization process. Every time an initial population is conceived, and with every selection and evolution step in the process, random individuals from the current generation are selected to undergo random changes in order to enter the next. You can control this randomness by using the random_state parameter of the estimator.

So you're skeptical. I hope so. Read on and discover the ways that the fittest programs in the population interact with one another to yield an even better generation.

1.1 Representation

As mentioned already, GP seeks to find a mathematical formula to represent a relationship. Let's use an arbitrary relationship as an example for the different ways that this could be written. Say we have two variables X0 and X1 that interact as follows to define a dependent variable y:

$$y = X_0^2 - 3 \times X_1 + 0.5$$

This could be re-written as:

$$y = X_0 \times X_0 - 3 \times X_1 + 0.5$$

Or as a LISP symbolic expression (S-expression) representation which uses prefix-notation, and happens to be very common in GP, as:

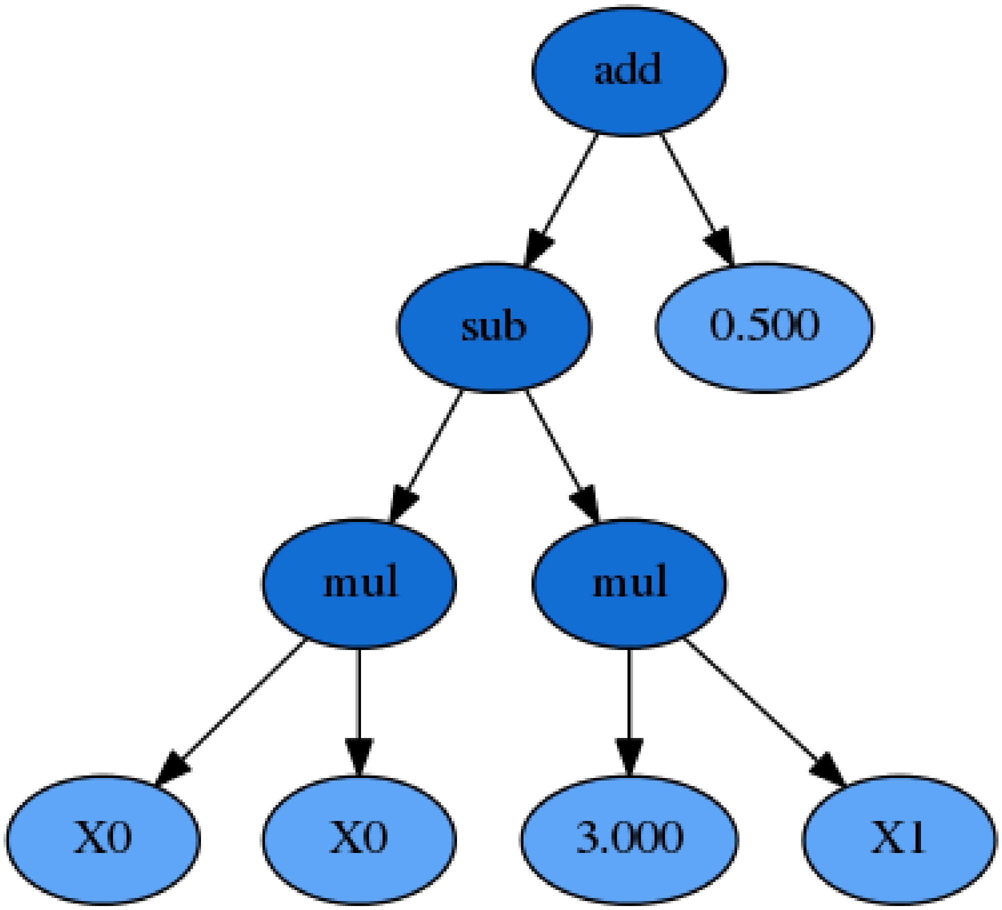
$$y = (+(-(\times X_0 X_0)(\times 3 X_1))0.5)$$

Or, since we're working in python here, let's express this as a numpy formula:

```
y = np.add(np.subtract(np.multiply(X0, X0), np.multiply(3., X1)), 0.5)
```

In each of these representations, we have a mix of variables, constants and functions. In this case we have the functions addition, subtraction, and multiplication. We also have the variables X_0 and X_1 and constants 3.0 and 0.5. Collectively, the variables and constants are known as terminals. Combined with the functions, the collection of available variables, constants and functions are known as the primitive set.

We could also represent this formula as a syntax tree, where the functions are interior nodes, shown in dark blue, and the variables and constants make up the leaves (or terminal nodes), shown in light blue:



Now you can see that the formula can be interpreted in a recursive manner. If we start with the left-hand leaves, we multiply X_0 and X_0 and that portion of the formula is evaluated by the subtraction operation (once the $X_1 \times 3.0$ portion is also evaluated). The result of the subtraction is then evaluated by the addition operation as we work up the syntax tree.

Importantly for GP the $X_0 \times X_0$ sub-expression, or sub-tree, can be replaced by any other valid expression that evaluates to a numerical answer, even if that is a constant. That sub-expression, and any larger one such as everything below the subtraction function, all reside adjacent to one another in the list-style representation, making replacement of these segments simple to do programatically.

A function has a property known as its arity. Arity, in a python functional sense, refers to the number of arguments that the function takes. In the cases above, all of the functions require two arguments, and thus have an arity of two. But other functions such as `np.abs()`, only require a single argument, and have an arity of 1.

Since we know the arity of all the available functions in our function set, we can actually simplify the S-expression and remove all of the parentheses:

$$y = + - \times X_0 X_0 \times 3 X_1 0.5$$

This could then be evaluated recursively, starting from the left and holding onto a stack which keeps track of how much cumulative arity needs to be satisfied by the terminal nodes.

Under the hood, `gplearn`'s representation is similar to this, and uses Python lists to store the functions and terminals. Constants are represented by floating point numbers, variables by integers and functions by a custom `Function` object.

In `gplearn`, the available function set is controlled by an arguments that is set when initializing an estimator. The default set is the arithmetic operators: addition, subtraction, division and multiplication. But you can also add in some transformers, comparison functions or trigonometric identities that are all built-in. These strings are put into the `function_set` argument to include them in your programs.

- 'add' : addition, arity=2.
- 'sub' : subtraction, arity=2.
- 'mul' : multiplication, arity=2.
- 'div' : division, arity=2.
- 'sqrt' : square root, arity=1.
- 'log' : log, arity=1.
- 'abs' : absolute value, arity=1.
- 'neg' : negative, arity=1.
- 'inv' : inverse, arity=1.
- 'max' : maximum, arity=2.
- 'min' : minimum, arity=2.
- 'sin' : sine (radians), arity=1.
- 'cos' : cosine (radians), arity=1.
- 'tan' : tangent (radians), arity=1.

You should chose whether these functions are valid for your program.

You can also set up your own functions by using the `functions.make_function()` factory function which will create a gp-compatible function node that can be incorporated into your programs. See *advanced use here*.

1.2 Fitness

Now that we can represent a formula as an executable program, we need to determine how well it performs. In a throwback to Darwin, in GP this measure is called a program's fitness. If you have used machine learning before, you may be more familiar with terms such as "score", "error" or "loss". It's basically the same thing, and as with those other machine learning terms, in GP we have to know whether the metric needs to be maximized or minimized in order to be able to select the best program in a group.

In `gplearn`, several metrics are available by setting the `metric` parameter.

For the `SymbolicRegressor` several common error metrics are available and the evolution process seeks to minimize them. The default is the magnitude of the error, 'mean absolute error'. Other metrics available are:

- 'mse' for mean squared error, and
- 'rmse' for root mean squared error.

For the `SymbolicTransformer`, where indirect optimization is sought, the metrics are based on correlation between the program's output and the target, these are maximized by the evolution process:

- 'pearson', for Pearson's product-moment correlation coefficient (the default), and
- 'spearman' for Spearman's rank-order correlation coefficient.

You can also set up your own fitness measures by using the `fitness.make_fitness()` factory function which will create a gp-compatible fitness function that can be used to evaluate your programs. See *advanced use here*.

Evaluating the fitness of all the programs in a population is probably the most expensive part of GP. In `gplearn`, you can parallelize this computation by using the `n_jobs` parameter to choose how many cores should work on it at once.

If your dataset is small, the overhead of splitting the work over several cores is probably more than the benefit of the reduced work per core. This is because the work is parallelized per generation, so use this only if your dataset is large and the fitness calculation takes a long time.

1.3 Closure

We have already discussed that the measure of a program's fitness is through some function that evaluates the program's predictions compared to some ground truth. But with functions like division in the function set, what happens if your denominator happens to be close to zero? In the case of zero division, or near-zero division in a computer program, the result happens to be an infinite quantity. So there goes your error for the entire test set, even if all other fitness samples were evaluated almost perfectly!

Thus, a critical component of rugged GP becomes apparent, we need to protect against such cases for functions that might break for certain arguments. Functions like division must be modified to be able to accept any input argument that could turn up to return a valid number at evaluation so that nodes higher up the tree can successfully evaluate their output.

In `gplearn`, several protected functions are used:

- division, if the denominator lies between -0.001 and 0.001, returns 1.0.
- square root returns the square root of the absolute value of the argument.
- log returns the square root of the absolute value of the argument, or for very small values less than 0.001, it returns 0.0.
- inverse, if the argument lies between -0.001 and 0.001, returns 0.0.

In this way, no matter the layout of the input data or structure of the evolved program, a valid numerical output can be guaranteed, even if we must sacrifice some interpretability to get there.

If you define your own functions, you will need to guard for this as well. The *`functions.make_function()`* factory function will perform some basic checks on your function to ensure it will guard against the most common invalid operations with negative or near-zero operations.

1.4 Sufficiency

Another requirement of a successful GP run is called sufficiency. Basically, can this problem be solved to an adequate level with the functions and variables available.

For toy symbolic regression tasks like that solved in example 1, this is easy to ascertain. But in real life, things are less easy to quantify. It may be that there is a good solution lurking in that multi-dimensional space, but there were insufficient generations evolved, or bad luck turned the evolution process in the wrong direction. It may also be possible that no good relationship can be found through symbolic combinations of your variables.

In application, try to set the constant range to a value that will be helpful to get close to the target. For example, if you are trying to regress on a target with values from 500 – 1000 using variables in a range of 0 – 1, a constant of 0.5 is unlikely to help, and the "best" solution is probably just going to be large amounts of irrelevant additions to try and get close to the lower bound. Similarly, standardizing or scaling your variables and targets can make the problem much easier to learn in some cases.

If you are using the trigonometric functions, make sure to convert any degree angles into radians as well. If you don't have any angles to convert, consider if these functions are useful for your problem, though a seasonal relationship might be discoverable if there is a temporal element in the data.

If you think that the problem requires a very large formula to approach a solution, start with a larger program depth. And if your dataset has a lot of variables, perhaps the "full" initialization method makes more sense to kick start the initial population with bigger programs that encompass more of the data than "grow" might yield.

1.5 Initialization

When starting a GP run, the first generation is blissfully unaware that there is any fitness function that needs to be maximized. These initial naive programs are a totally random mix of the available functions and variables. But the user might know a little bit more about the problem before hand and give the evolution process a kick in the right direction in terms of the complexity of the problem at hand. Probably the biggest aspect that goes into this decision is the number of features in your dataset.

The first parameter to look at is the `init_depth` of the programs in the initial population. This controls the range of program sizes to initialize in the first generation (after that it's up to evolution). `init_depth` accepts a tuple of two integers. When generating the initial population, a random maximum depth is chosen within this range for each individual, and the program is grown to satisfy this requirement. The default range of 2 – 6 is generally a good starting point, but if your dataset has a lot of variables, you may wish to make larger programs at first, if only to have more of them included in the initial population.

Next, you should consider `population_size`. This controls the number of programs generated in both the initial population and every generation following it. If you have very few variables, and have a limited function set, a smaller population size may suffice. If you have a lot of variables, or expect a very large program is required you may want to start with larger programs. More likely, the number of programs you wish to maintain will be constrained by the amount of time you want to spend evaluating them.

Finally, you need to decide on the `init_method` appropriate for your data. For all options, the root node is a function to avoid having degenerative programs representing only a single variable or constant in the initial population.

For the 'grow' method, nodes are chosen at random from both functions and terminals, allowing for smaller trees than `init_depth` allows. This tends to grow asymmetrical trees as terminals can be chosen before the max depth is reached. If your dataset has a lot of variables, this will likely result in much smaller programs that the `init_depth` range requests. Similarly, if you have very few variables and have chosen a lot of function sets, you will likely see programs approaching the maximum depth range in the population.

The 'full' method chooses nodes from the function set until the max depth is reached, and then terminals are chosen. This tends to grow 'bushy' symmetrical trees.

The default is the 'half and half' method. Program trees are grown through a 50/50 mix of 'full' and 'grow', making for a mix of tree shapes in the initial population. When combined with `init_method='half and half'` this yields the well-known 'ramped half and half' initialization method which seeds the population with lots of programs of different sizes and shapes, leading to a diverse mix of representations.

1.6 Selection

Now that we have a population of programs, we need to decide which ones will get to evolve into the next generation. In `gplearn` this is done through tournaments. From the population, a smaller subset is selected at random to compete, the size of which is controlled by the `tournament_size` parameter. The fittest individual in this subset is then selected to move on to the next generation.

Having a large tournament size will generally find fitter programs more quickly and the evolution process will tend to converge to a solution in less time. A smaller tournament size will likely maintain more diversity in the population as more programs are given a chance to evolve and the population may find a better solution at the expense of taking longer. This is known as selection pressure, and your choice here may be governed by the computation time.

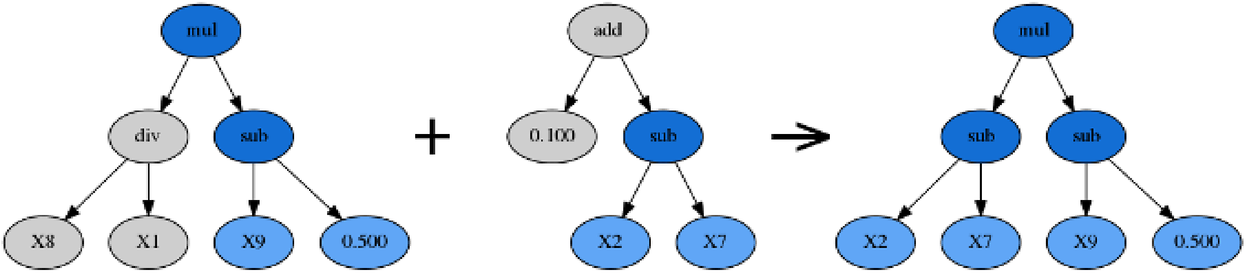
1.7 Evolution

As discussed in the selection section, we use the fitness measure to find the fittest individual in the tournament to survive. But this individual does not just graduate unaltered to the next generation, genetic operations are performed on them. Several common genetic operations are supported by `gplearn`.

Crossover

Crossover is the principle method of mixing genetic material between individuals and is controlled by the p_crossover parameter. Unlike other genetic operations, it requires two tournaments to be run in order to find a parent and a donor.

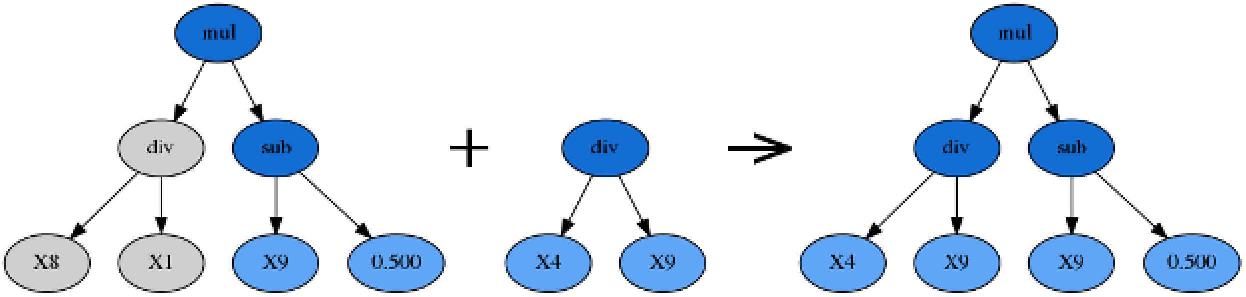
Crossover takes the winner of a tournament and selects a random subtree from it to be replaced. A second tournament is performed to find a donor. The donor also has a subtree selected at random and this is inserted into the original parent to form an offspring in the next generation.



Subtree Mutation

Subtree mutation is one of the more aggressive mutation operators and is controlled by the p_subtree_mutation parameter. The reason it is more aggressive is that more genetic material can be replaced by totally naive random components. This can reintroduce lost functions and operators into the population to maintain diversity.

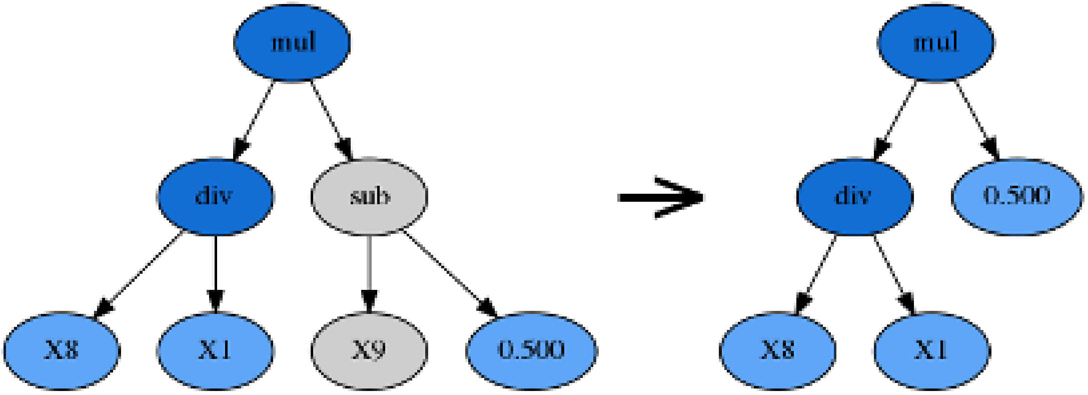
Subtree mutation takes the winner of a tournament and selects a random subtree from it to be replaced. A donor subtree is generated at random and this is inserted into the original parent to form an offspring in the next generation.



Hoist Mutation

Hoist mutation is a bloat-fighting mutation operation. It is controlled by the p_hoist_mutation parameter and solely removes genetic material from tournament winners.

Hoist mutation takes the winner of a tournament and selects a random subtree from it. A random subtree of that subtree is then selected and this is 'hoisted' into the original subtrees location to form an offspring in the next generation.

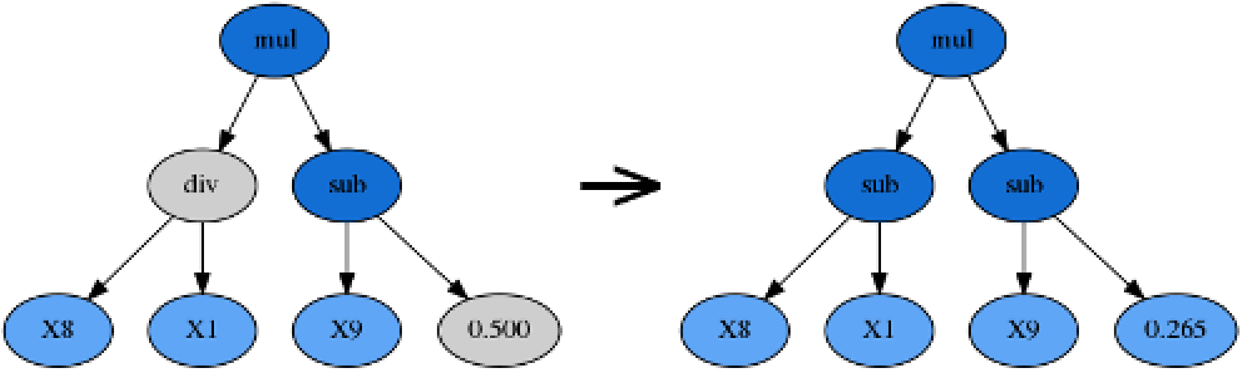


Point Mutation

Point mutation is probably the most common form of mutation operations in genetic programming. It can reintroduce lost functions and operators into the population to maintain diversity.

Point mutation takes the winner of a tournament and selects random nodes from it to be replaced. Terminals are replaced by other terminals and functions are replaced by other functions that require the same number of arguments as the original node. The resulting tree forms an offspring in the next generation.

Functions and terminals are randomly chosen for replacement as controlled by the p_point_replace parameter which guides the average amount of replacement to perform.



Reproduction

Should the sum of the above genetic operations' probabilities, as set by their independent probabilities, be less than one, the balance of genetic operations shall fall back on reproduction. That is, a tournament winner is cloned and enters the next generation unmodified.

1.8 Termination

There are two ways that the evolution process will stop. The first is that the maximum number of generations, controlled by the parameter `generations`, is reached. The second way is that at least one program in the population has a fitness that exceeds the parameter `stopping_criteria`, which defaults to being a perfect score. You may need to do a couple of test runs to determine what metric is possible if you are working with real-life data in order to set this value appropriately.

1.9 Bloat

A program's size can be measured in two ways: its depth and length. The depth of a program is the maximum distance from its root node to the furthest leaf node. A degenerative program with only a single value, ie y = X0, has a depth of zero. The length of a program is simply the number of elements in the formula, or the count of the number of nodes.

An interesting phenomena is often encountered in GP where the population sizes grow larger and larger with no significant improvement in fitness. This is known as bloat and leads to longer and longer computation times with little benefit to the solution.

Bloat can be fought in `gplearn` in several ways. The principle weapon is using a penalized fitness measure during selection where the fitness of an individual is made worse the larger it is. In this way, should there be two programs with identical fitness competing in a tournament, the smaller program will be selected and the larger one discarded. The `parsimony_coefficient` parameter controls this penalty and may need to be experimented with to get good performance. Too large a penalty and your smallest programs will tend to be selected regardless of their actual performance on the data, too small and bloat will continue unabated. The final winner of the evolution process is still chosen based on the unpenalized fitness, otherwise known as its raw fitness.

A recent paper introduced the covariant parsimony method which can be used by setting `parsimony_coefficient='auto'`. This method adapts the penalty depending on the relationship between program fitness and size in the population and will change from generation to generation.

Another method to fight bloat is by using genetic operations that make programs smaller. `gplearn` has hoist mutation which removes parts of programs during evolution. It can be controlled by the `p_hoist_mutation` parameter.

Finally, you could increase the amount of subsampling performed on your data to get more diverse looks at individual programs from smaller portions of the data. `max_samples` controls this rate and defaults to no subsampling. As a bonus, if you choose to subsample, you also get to see the "out of bag" fitness of the best program in the verbose reporter (activated by setting `verbose=1`). Hopefully this is pretty close to the in-sample fitness that is also reported.

1.10 Transformer

The *`SymbolicTransformer`* works slightly differently to the *`SymbolicRegressor`*. While the regressor seeks to minimize the error between the programs' outputs and the target variable based on an error metric, the transformer seeks an indirect relationship that can then be exploited by a second estimator. Essentially, this is automated feature engineering and can create powerful non-linear interactions that may be difficult to discover in conventional methods.

Where the regressor looks to minimize the direct error, the transformer looks to maximize the correlation between the predicted value and the target. This is done through either the Pearson product-moment correlation coefficient (the default) or the Spearman rank-order correlation coefficient. In both cases the absolute value of the correlation is maximized in order to accept strongly negatively correlated programs.

The Spearman correlation is appropriate if your next estimator is going to be tree-based estimator such as a Random Forest or Gradient Boosting Machine. If you plan to send the new transformed variables into a linear model, it is probably better to stick with the default Pearson correlation.

The *`SymbolicTransformer`* looks at the final generation of the evolution and picks the best programs to evaluate. The number of programs it will look at is controlled by the `hall_of_fame` parameter.

From the hall of fame, it will then whittle down the best programs to the least correlated amongst them as controlled by the `n_components` parameter. You may have the top two programs being almost identical, so this step removes that issue. The correlation between individuals within the hall of fame uses the same correlation method, Pearson or Spearman, as used by the evolution process.

Convinced?

See some examples, *explore the full API reference* and *install the package*!

CHAPTER 2

Examples

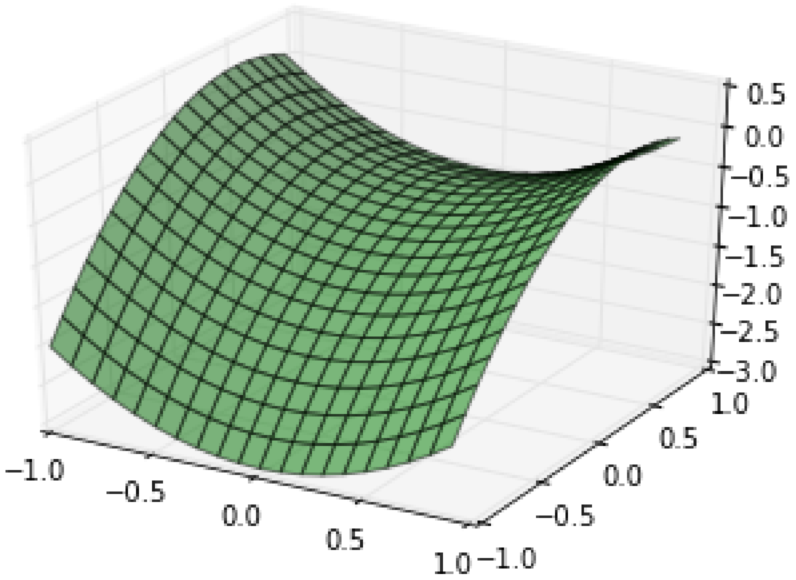
The code used to generate these examples can be found here as an iPython Notebook.

2.1 Example 1: Symbolic Regressor

This example demonstrates using the *SymbolicRegressor* to fit a symbolic relationship.

Let's create some synthetic data based on the relationship $y = X_0^2 - X_1^2 + X_1 - 1$:

```
x0 = np.arange(-1, 1, 1/10.)
x1 = np.arange(-1, 1, 1/10.)
x0, x1 = np.meshgrid(x0, x1)
y_truth = x0**2 - x1**2 + x1 - 1

ax = plt.figure().gca(projection='3d')
ax.set_xlim(-1, 1)
ax.set_ylim(-1, 1)
surf = ax.plot_surface(x0, x1, y_truth, rstride=1, cstride=1,
                       color='green', alpha=0.5)
plt.show()
```



We can create some random training and test data that lies on this surface too:

```
rng = check_random_state(0)

# Training samples
X_train = rng.uniform(-1, 1, 100).reshape(50, 2)
y_train = X_train[:, 0]**2 - X_train[:, 1]**2 + X_train[:, 1] - 1

# Testing samples
X_test = rng.uniform(-1, 1, 100).reshape(50, 2)
y_test = X_test[:, 0]**2 - X_test[:, 1]**2 + X_test[:, 1] - 1
```

Now let's consider how to fit our *SymbolicRegressor* to this data. Since it's a fairly small dataset, we can probably use a large population since training time will still be pretty fast. We'll evolve 20 generations unless the error falls below 0.01. Examining the equation, it looks like the default function set of addition, subtraction, multiplication and division will cover us. Let's bump up the amount of mutation and subsample so that we can watch the OOB error evolve. We'll also increase the parsimony coefficient to keep our solutions small, since we know the truth is a pretty simple equation:

```
est_gp = SymbolicRegressor(population_size=5000,
                           generations=20, stopping_criteria=0.01,
                           p_crossover=0.7, p_subtree_mutation=0.1,
                           p_hoist_mutation=0.05, p_point_mutation=0.1,
                           max_samples=0.9, verbose=1,
                           parsimony_coefficient=0.01, random_state=0)
est_gp.fit(X_train, y_train)

    |    Population Average   |             Best Individual              |
---- ------------------------- ------------------------------------------ ----------
 Gen   Length          Fitness   Length          Fitness      OOB Fitness  Time Left
   0    38.13     458.57768152        5   0.320665972828   0.556763539274      1.28m
   1     9.97    1.70232723129        5   0.320201761523   0.624787148042     57.78s
   2     7.72    1.94456344674       11   0.239536660154   0.533148180489     46.35s
   3     5.41   0.990156815469        7   0.235676349446   0.719906258051     37.93s
   4     4.66   0.894443363616       11   0.103946413589   0.103946413589     32.20s
```

(continues on next page)

(continued from previous page)

```
   5      5.41    0.940242380405        11    0.060802040427    0.060802040427      28.15s
   6      6.78      1.0953592564        11    0.000781474035    0.000781474035      24.85s
```

The evolution process stopped early as the error of the best program in the 9th generation was better than 0.01. It also appears that the parsimony coefficient was just about right as the average length of the programs fluctuated around a bit before settling on a pretty reasonable size. Let's look at what our solution was:

```
print est_gp._program

sub(add(-0.999, X1), mul(sub(X1, X0), add(X0, X1)))
```

Interestingly, this does not have the same structure as our target function. But let's expand the mathematics out:

$$y = (-0.999 + X_1) - ((X_1 - X_0) \times (X_0 + X_1))$$

$$y = X_1 - 0.999 - (X_1 X_0 + X_1^2 - X_0^2 - X_0 X_1)$$

$$y = X_0^2 - X_1^2 + X_1 - 0.999$$

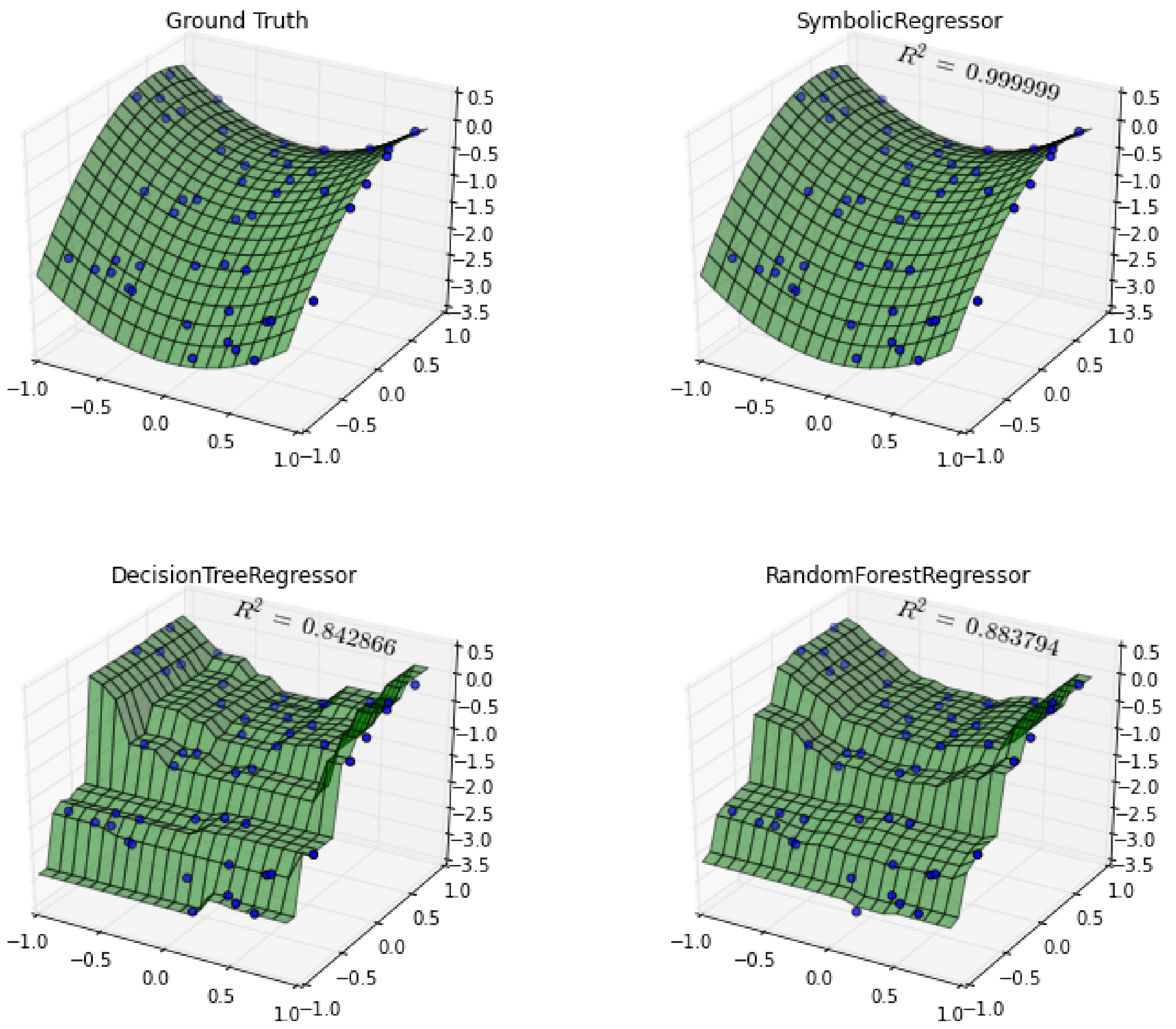
Despite representing an interaction of X_0 and X_1, these terms cancel and we're left with the (almost) exact relationship we were seeking!

Great, but let's compare with some other non-linear models to see how they do:

```
est_tree = DecisionTreeRegressor()
est_tree.fit(X_train, y_train)
est_rf = RandomForestRegressor()
est_rf.fit(X_train, y_train)
```

We can plot the decision surfaces of all three to visualize each one:

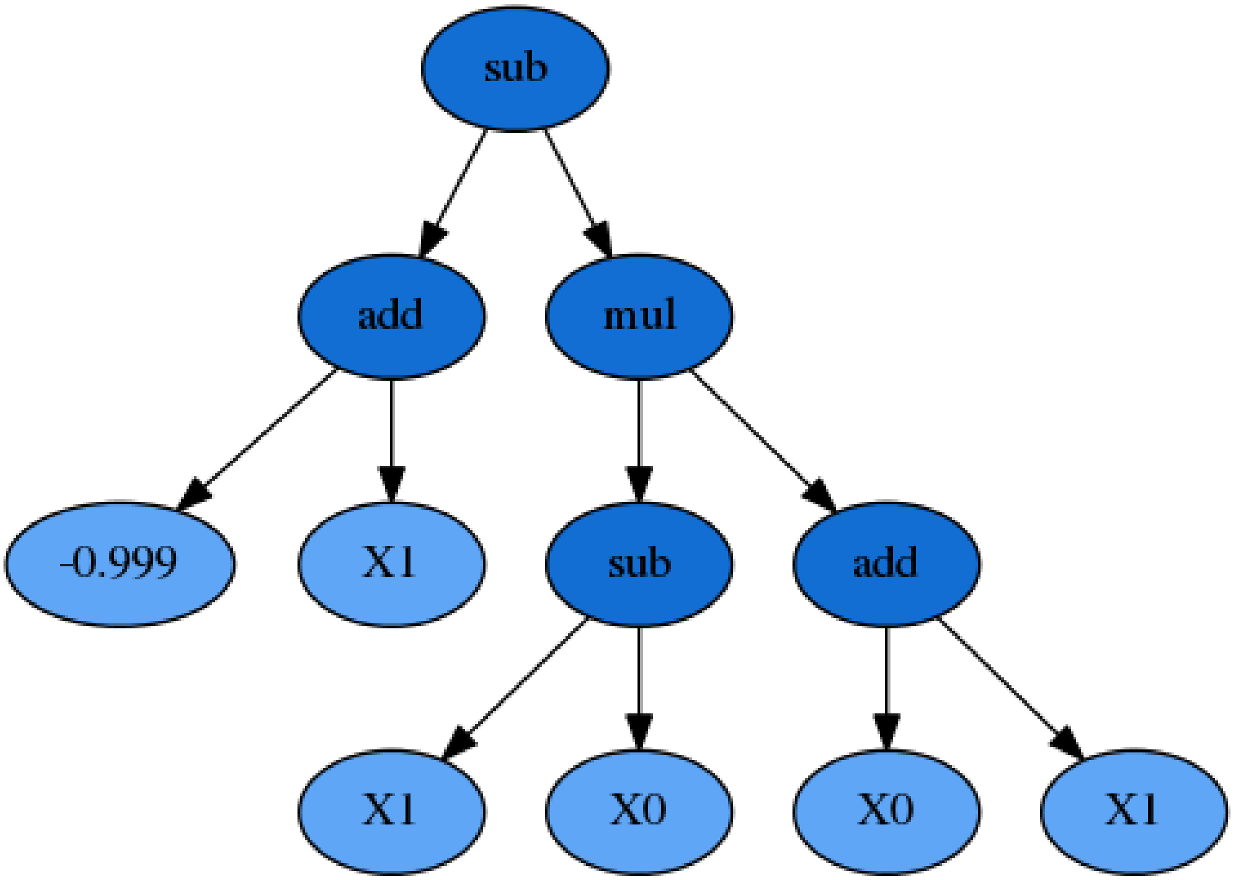
```
y_gp = est_gp.predict(np.c_[x0.ravel(), x1.ravel()]).reshape(x0.shape)
score_gp = est_gp.score(X_test, y_test)
y_tree = est_tree.predict(np.c_[x0.ravel(), x1.ravel()]).reshape(x0.shape)
score_tree = est_tree.score(X_test, y_test)
y_rf = est_rf.predict(np.c_[x0.ravel(), x1.ravel()]).reshape(x0.shape)
score_rf = est_rf.score(X_test, y_test)

fig = plt.figure(figsize=(12, 10))

for i, (y, score, title) in enumerate([(y_truth, None, "Ground Truth"),
                                       (y_gp, score_gp, "SymbolicRegressor"),
                                       (y_tree, score_tree, "DecisionTreeRegressor"),
                                       (y_rf, score_rf, "RandomForestRegressor")]):

    ax = fig.add_subplot(2, 2, i+1, projection='3d')
    ax.set_xlim(-1, 1)
    ax.set_ylim(-1, 1)
    surf = ax.plot_surface(x0, x1, y, rstride=1, cstride=1, color='green', alpha=0.5)
    points = ax.scatter(X_train[:, 0], X_train[:, 1], y_train)
    if score is not None:
        score = ax.text(-.7, 1, .2, "$R^2 =\/ %.6f$" % score, 'x', fontsize=14)
    plt.title(title)
plt.show()
```



Not bad *SymbolicRegressor*! We were able to fit a very smooth function to the data, while the tree-based estimators created very "blocky" decision surfaces. The Random Forest appears to have smoothed out some of the wrinkles but in both cases the tree models have fit very well to the training data, but done worse on out-of-sample data.

We can also inspect the program that the *SymbolicRegressor* found:

```
graph = pydotplus.graphviz.graph_from_dot_data(est_gp._program.export_graphviz())
Image(graph.create_png())
```



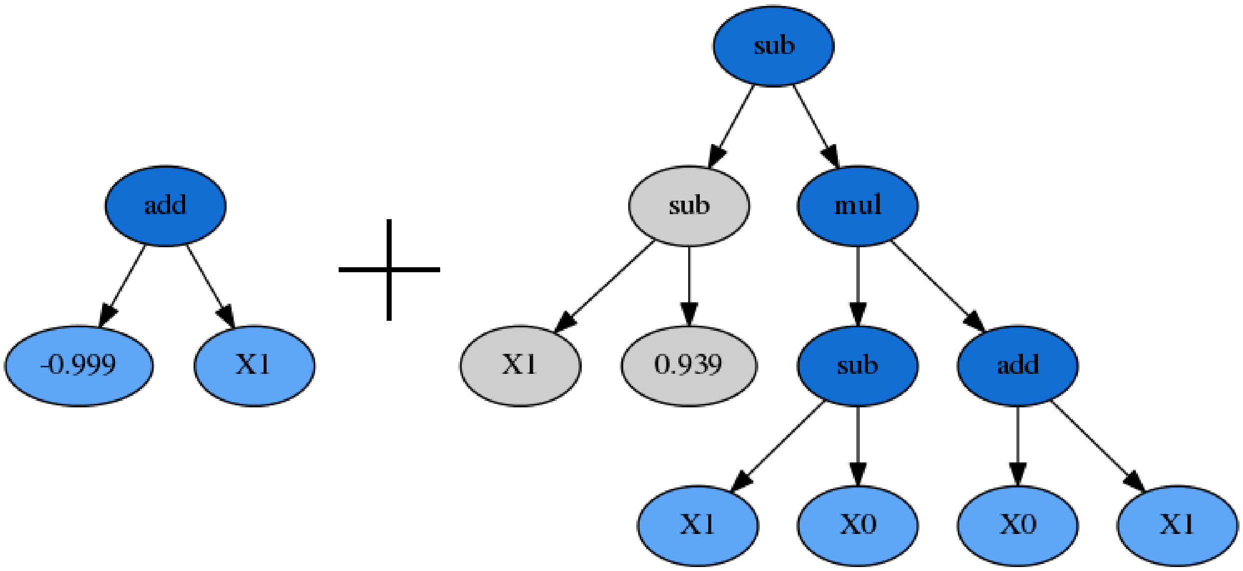
And check out who its parents were:

```
print est_gp._program.parents

{'method': 'Crossover',
 'parent_idx': 1555,
 'parent_nodes': [1, 2, 3],
 'donor_idx': 78,
 'donor_nodes': []}
```

This dictionary tells us what evolution operation was performed to get our new individual, as well as the parents from the prior generation, and any nodes that were removed from them during, in this case, Crossover.

Plotting the parents shows how the genetic material from them combined to form our winning program:

```
idx = est_gp._program.parents['donor_idx']
fade_nodes = est_gp._program.parents['donor_nodes']
graph = est_gp._programs[-2][idx].export_graphviz(fade_nodes=fade_nodes)
graph = pydotplus.graphviz.graph_from_dot_data(graph)
Image(graph.create_png())
```



2.2 Example 2: Symbolic Transformer

This example demonstrates using the *SymbolicTransformer* to generate new non-linear features automatically.

Let's load up the Boston housing dataset and randomly shuffle it:

```
rng = check_random_state(0)
boston = load_boston()
perm = rng.permutation(boston.target.size)
boston.data = boston.data[perm]
boston.target = boston.target[perm]
```

We'll use Ridge Regression for this example and train our regressor on the first 300 samples, and see how it performs on the unseen final 200 samples. The benchmark to beat is simply Ridge running on the dataset as-is:

```
est = Ridge()
est.fit(boston.data[:300, :], boston.target[:300])
print est.score(boston.data[300:, :], boston.target[300:])

0.759145222183
```

So now we'll train our transformer on the same first 300 samples to generate some new features. Let's use a large population of 2000 individuals over 20 generations. We'll select the best 100 of these for the `hall_of_fame`, and then use the least-correlated 10 as our new features. A little parsimony should control bloat, but we'll leave the rest of the evolution options at their defaults. The default `metric='pearson'` is appropriate here since we are using a linear model as the estimator. If we were going to use a tree-based estimator, the Spearman correlation might be interesting to try out too:

```
function_set = ['add', 'sub', 'mul', 'div',
                'sqrt', 'log', 'abs', 'neg', 'inv',
                'max', 'min']
gp = SymbolicTransformer(generations=20, population_size=2000,
                         hall_of_fame=100, n_components=10,
                         function_set=function_set,
                         parsimony_coefficient=0.0005,
```

(continues on next page)

(continued from previous page)

```
                         max_samples=0.9, verbose=1,
                         random_state=0, n_jobs=3)
gp.fit(boston.data[:300, :], boston.target[:300])
```

We will then apply our trained transformer to the entire Boston dataset (remember, it still hasn't seen the final 200 samples) and concatenate this to the original data:

```
gp_features = gp.transform(boston.data)
new_boston = np.hstack((boston.data, gp_features))
```

Now we train the Ridge regressor on the first 300 samples of the transformed dataset and see how it performs on the final 200 again:

```
est = Ridge()
est.fit(new_boston[:300, :], boston.target[:300])
print est.score(new_boston[300:, :], boston.target[300:])

0.841750404385
```

Great! We have improved the R^2 score by a significant margin. It looks like the linear model was able to take advantage of some new non-linear features to fit the data even better.

Next up, *explore the full API reference* or just skip ahead *install the package*!

CHAPTER 3

API reference

3.1 Symbolic Regressor

class gplearn.genetic.**SymbolicRegressor**(*population_size=1000*, *generations=20*, *tournament_size=20*, *stopping_criteria=0.0*, *const_range=(-1.0, 1.0)*, *init_depth=(2, 6)*, *init_method='half and half'*, *function_set=('add', 'sub', 'mul', 'div')*, *metric='mean absolute error'*, *parsimony_coefficient=0.001*, *p_crossover=0.9*, *p_subtree_mutation=0.01*, *p_hoist_mutation=0.01*, *p_point_mutation=0.01*, *p_point_replace=0.05*, *max_samples=1.0*, *warm_start=False*, *n_jobs=1*, *verbose=0*, *random_state=None*)

A Genetic Programming symbolic regressor.

A symbolic regressor is an estimator that begins by building a population of naive random formulas to represent a relationship. The formulas are represented as tree-like structures with mathematical functions being recursively applied to variables and constants. Each successive generation of programs is then evolved from the one that came before it by selecting the fittest individuals from the population to undergo genetic operations such as crossover, mutation or reproduction.

Parameters

population_size [integer, optional (default=500)] The number of programs in each generation.

generations [integer, optional (default=10)] The number of generations to evolve.

tournament_size [integer, optional (default=20)] The number of programs that will compete to become part of the next generation.

stopping_criteria [float, optional (default=0.0)] The required metric value required in order to stop evolution early.

const_range [tuple of two floats, optional (default=(-1., 1.))] The range of constants to include in the formulas.

init_depth [tuple of two ints, optional (default=(2, 6))] The range of tree depths for the initial population of naive formulas. Individual trees will randomly choose a maximum depth from this range. When combined with *init_method='half and half'* this yields the well- known 'ramped half and half' initialization method.

init_method [str, optional (default='half and half')]

- 'grow' : Nodes are chosen at random from both functions and terminals, allowing for smaller trees than *init_depth* allows. Tends to grow asymmetrical trees.
- 'full' : Functions are chosen until the *init_depth* is reached, and then terminals are selected. Tends to grow 'bushy' trees.
- 'half and half' : Trees are grown through a 50/50 mix of 'full' and 'grow', making for a mix of tree shapes in the initial population.

function_set [iterable, optional (default=('add', 'sub', 'mul', 'div'))] The functions to use when building and evolving programs. This iterable can include strings to indicate either individual functions as outlined below, or you can also include your own functions as built using the `make_function` factory from the `functions` module.

Available individual functions are:

- 'add' : addition, arity=2.
- 'sub' : subtraction, arity=2.
- 'mul' : multiplication, arity=2.
- 'div' : protected division where a denominator near-zero returns 1., arity=2.
- 'sqrt' : protected square root where the absolute value of the argument is used, arity=1.
- 'log' : protected log where the absolute value of the argument is used and a near-zero argument returns 0., arity=1.
- 'abs' : absolute value, arity=1.
- 'neg' : negative, arity=1.
- 'inv' : protected inverse where a near-zero argument returns 0., arity=1.
- 'max' : maximum, arity=2.
- 'min' : minimum, arity=2.
- 'sin' : sine (radians), arity=1.
- 'cos' : cosine (radians), arity=1.
- 'tan' : tangent (radians), arity=1.

metric [str, optional (default='mean absolute error')] The name of the raw fitness metric. Available options include:

- 'mean absolute error',
- 'mse' for mean squared error,
- 'rmse' for root mean squared error.

parsimony_coefficient [float or "auto", optional (default=0.001)] This constant penalizes large programs by adjusting their fitness to be less favorable for selection. Larger values penalize the program more which can control the phenomenon known as 'bloat'. Bloat is when evolution is increasing the size of programs without a significant increase in fitness, which is costly for computation time and makes for a less understandable final result. This parameter may need to be tuned over successive runs.

If "auto" the parsimony coefficient is recalculated for each generation using c = Cov(l,f)/Var(l), where Cov(l,f) is the covariance between program size l and program fitness f in the population, and Var(l) is the variance of program sizes.

p_crossover [float, optional (default=0.9)] The probability of performing crossover on a tournament winner. Crossover takes the winner of a tournament and selects a random subtree from it to be replaced. A second tournament is performed to find a donor. The donor also has a subtree selected at random and this is inserted into the original parent to form an offspring in the next generation.

p_subtree_mutation [float, optional (default=0.01)] The probability of performing subtree mutation on a tournament winner. Subtree mutation takes the winner of a tournament and selects a random subtree from it to be replaced. A donor subtree is generated at random and this is inserted into the original parent to form an offspring in the next generation.

p_hoist_mutation [float, optional (default=0.01)] The probability of performing hoist mutation on a tournament winner. Hoist mutation takes the winner of a tournament and selects a random subtree from it. A random subtree of that subtree is then selected and this is 'hoisted' into the original subtrees location to form an offspring in the next generation. This method helps to control bloat.

p_point_mutation [float, optional (default=0.01)] The probability of performing point mutation on a tournament winner. Point mutation takes the winner of a tournament and selects random nodes from it to be replaced. Terminals are replaced by other terminals and functions are replaced by other functions that require the same number of arguments as the original node. The resulting tree forms an offspring in the next generation.

Note : The above genetic operation probabilities must sum to less than one. The balance of probability is assigned to 'reproduction', where a tournament winner is cloned and enters the next generation unmodified.

p_point_replace [float, optional (default=0.05)] For point mutation only, the probability that any given node will be mutated.

max_samples [float, optional (default=1.0)] The fraction of samples to draw from X to evaluate each program on.

warm_start [bool, optional (default=False)] When set to `True`, reuse the solution of the previous call to fit and add more generations to the evolution, otherwise, just fit a new evolution.

n_jobs [integer, optional (default=1)] The number of jobs to run in parallel for *fit*. If -1, then the number of jobs is set to the number of cores.

verbose [int, optional (default=0)] Controls the verbosity of the evolution building process.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

See also:

SymbolicTransformer

References

[1], *[2]*

fit (*X*, *y*, *sample_weight=None*)
Fit the Genetic Program according to X, y.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples]] Target values.

sample_weight [array-like, shape = [n_samples], optional] Weights applied to individual samples.

Returns

self [object] Returns self.

get_params (*deep=True*)
Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

predict (*X*)
Perform regression on test vectors X.

Parameters

X [array-like, shape = [n_samples, n_features]] Input vectors, where n_samples is the number of samples and n_features is the number of features.

Returns

y [array, shape = [n_samples]] Predicted values for X.

score (*X*, *y*, *sample_weight=None*)
Returns the coefficient of determination R^2 of the prediction.

The coefficient R^2 is defined as (1 - u/v), where u is the residual sum of squares ((y_true - y_pred) ** 2).sum() and v is the total sum of squares ((y_true - y_true.mean()) ** 2).sum(). The best possible score is 1.0 and it can be negative (because the model can be arbitrarily worse). A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

Parameters

X [array-like, shape = (n_samples, n_features)] Test samples.

y [array-like, shape = (n_samples) or (n_samples, n_outputs)] True values for X.

sample_weight [array-like, shape = [n_samples], optional] Sample weights.

Returns

score [float] R^2 of self.predict(X) wrt. y.

set_params (***params*)
Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

3.2 Symbolic Transformer

class gplearn.genetic.**SymbolicTransformer**(*population_size=1000*, *hall_of_fame=100*, *n_components=10*, *generations=20*, *tournament_size=20*, *stopping_criteria=1.0*, *const_range=(-1.0, 1.0)*, *init_depth=(2, 6)*, *init_method='half and half'*, *function_set=('add', 'sub', 'mul', 'div')*, *metric='pearson'*, *parsimony_coefficient=0.001*, *p_crossover=0.9*, *p_subtree_mutation=0.01*, *p_hoist_mutation=0.01*, *p_point_mutation=0.01*, *p_point_replace=0.05*, *max_samples=1.0*, *warm_start=False*, *n_jobs=1*, *verbose=0*, *random_state=None*)

A Genetic Programming symbolic transformer.

A symbolic transformer is a supervised transformer that begins by building a population of naive random formulas to represent a relationship. The formulas are represented as tree-like structures with mathematical functions being recursively applied to variables and constants. Each successive generation of programs is then evolved from the one that came before it by selecting the fittest individuals from the population to undergo genetic operations such as crossover, mutation or reproduction. The final population is searched for the fittest individuals with the least correlation to one another.

Parameters

population_size [integer, optional (default=500)] The number of programs in each generation.

hall_of_fame [integer, or None, optional (default=100)] The number of fittest programs to compare from when finding the least-correlated individuals for the n_components. If *None*, the entire final generation will be used.

n_components [integer, or None, optional (default=10)] The number of best programs to return after searching the hall_of_fame for the least-correlated individuals. If *None*, the entire hall_of_fame will be used.

generations [integer, optional (default=10)] The number of generations to evolve.

tournament_size [integer, optional (default=20)] The number of programs that will compete to become part of the next generation.

stopping_criteria [float, optional (default=1.0)] The required metric value required in order to stop evolution early.

const_range [tuple of two floats, optional (default=(-1., 1.))] The range of constants to include in the formulas.

init_depth [tuple of two ints, optional (default=(2, 6))] The range of tree depths for the initial population of naive formulas. Individual trees will randomly choose a maximum depth from this range. When combined with *init_method='half and half'* this yields the well- known 'ramped half and half' initialization method.

init_method [str, optional (default='half and half')]

- 'grow' : Nodes are chosen at random from both functions and terminals, allowing for smaller trees than *init_depth* allows. Tends to grow asymmetrical trees.

- 'full' : Functions are chosen until the *init_depth* is reached, and then terminals are selected. Tends to grow 'bushy' trees.
- 'half and half' : Trees are grown through a 50/50 mix of 'full' and 'grow', making for a mix of tree shapes in the initial population.

function_set [iterable, optional (default=('add', 'sub', 'mul', 'div'))] The functions to use when building and evolving programs. This iterable can include strings to indicate either individual functions as outlined below, or you can also include your own functions as built using the `make_function` factory from the `functions` module.

Available individual functions are:

- 'add' : addition, arity=2.
- 'sub' : subtraction, arity=2.
- 'mul' : multiplication, arity=2.
- 'div' : protected division where a denominator near-zero returns 1., arity=2.
- 'sqrt' : protected square root where the absolute value of the argument is used, arity=1.
- 'log' : protected log where the absolute value of the argument is used and a near-zero argument returns 0., arity=1.
- 'abs' : absolute value, arity=1.
- 'neg' : negative, arity=1.
- 'inv' : protected inverse where a near-zero argument returns 0., arity=1.
- 'max' : maximum, arity=2.
- 'min' : minimum, arity=2.
- 'sin' : sine (radians), arity=1.
- 'cos' : cosine (radians), arity=1.
- 'tan' : tangent (radians), arity=1.

metric [str, optional (default='pearson')] The name of the raw fitness metric. Available options include:

- 'pearson', for Pearson's product-moment correlation coefficient, and
- 'spearman' for Spearman's rank-order correlation coefficient.

parsimony_coefficient [float or "auto", optional (default=0.001)] This constant penalizes large programs by adjusting their fitness to be less favorable for selection. Larger values penalize the program more which can control the phenomenon known as 'bloat'. Bloat is when evolution is increasing the size of programs without a significant increase in fitness, which is costly for computation time and makes for a less understandable final result. This parameter may need to be tuned over successive runs.

If "auto" the parsimony coefficient is recalculated for each generation using c = Cov(l,f)/Var(l), where Cov(l,f) is the covariance between program size l and program fitness f in the population, and Var(l) is the variance of program sizes.

p_crossover [float, optional (default=0.9)] The probability of performing crossover on a tournament winner. Crossover takes the winner of a tournament and selects a random subtree from it to be replaced. A second tournament is performed to find a donor. The donor also has a subtree selected at random and this is inserted into the original parent to form an offspring in the next generation.

p_subtree_mutation [float, optional (default=0.01)] The probability of performing subtree mutation on a tournament winner. Subtree mutation takes the winner of a tournament and selects a random subtree from it to be replaced. A donor subtree is generated at random and this is inserted into the original parent to form an offspring in the next generation.

p_hoist_mutation [float, optional (default=0.01)] The probability of performing hoist mutation on a tournament winner. Hoist mutation takes the winner of a tournament and selects a random subtree from it. A random subtree of that subtree is then selected and this is 'hoisted' into the original subtrees location to form an offspring in the next generation. This method helps to control bloat.

p_point_mutation [float, optional (default=0.01)] The probability of performing point mutation on a tournament winner. Point mutation takes the winner of a tournament and selects random nodes from it to be replaced. Terminals are replaced by other terminals and functions are replaced by other functions that require the same number of arguments as the original node. The resulting tree forms an offspring in the next generation.

Note : The above genetic operation probabilities must sum to less than one. The balance of probability is assigned to 'reproduction', where a tournament winner is cloned and enters the next generation unmodified.

p_point_replace [float, optional (default=0.05)] For point mutation only, the probability that any given node will be mutated.

max_samples [float, optional (default=1.0)] The fraction of samples to draw from X to evaluate each program on.

warm_start [bool, optional (default=False)] When set to `True`, reuse the solution of the previous call to fit and add more generations to the evolution, otherwise, just fit a new evolution.

n_jobs [integer, optional (default=1)] The number of jobs to run in parallel for *fit*. If -1, then the number of jobs is set to the number of cores.

verbose [int, optional (default=0)] Controls the verbosity of the evolution building process.

random_state [int, RandomState instance or None, optional (default=None)] If int, random_state is the seed used by the random number generator; If RandomState instance, random_state is the random number generator; If None, the random number generator is the RandomState instance used by *np.random*.

See also:

`SymbolicRegressor`

References

[1], *[2]*

`fit` (*X*, *y*, *sample_weight=None*)

Fit the Genetic Program according to X, y.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples]] Target values.

sample_weight [array-like, shape = [n_samples], optional] Weights applied to individual samples.

Returns

self [object] Returns self.

`fit_transform`(*X*, *y*, *sample_weight=None*)

Fit to data, then transform it.

Parameters

X [array-like, shape = [n_samples, n_features]] Training vectors, where n_samples is the number of samples and n_features is the number of features.

y [array-like, shape = [n_samples]] Target values.

sample_weight [array-like, shape = [n_samples], optional] Weights applied to individual samples.

Returns

X_new [array-like, shape = [n_samples, n_components]] Transformed array.

`get_params`(*deep=True*)

Get parameters for this estimator.

Parameters

deep [boolean, optional] If True, will return the parameters for this estimator and contained subobjects that are estimators.

Returns

params [mapping of string to any] Parameter names mapped to their values.

`set_params`(***params*)

Set the parameters of this estimator.

The method works on simple estimators as well as on nested objects (such as pipelines). The latter have parameters of the form `<component>__<parameter>` so that it's possible to update each component of a nested object.

Returns

self

`transform`(*X*)

Transform X according to the fitted transformer.

Parameters

X [array-like, shape = [n_samples, n_features]] Input vectors, where n_samples is the number of samples and n_features is the number of features.

Returns

X_new [array-like, shape = [n_samples, n_components]] Transformed array.

3.3 User-Defined Functions

`gplearn.functions.make_function`(*function*, *name*, *arity*)

Make a function node, a representation of a mathematical relationship.

This factory function creates a function node, one of the core nodes in any program. The resulting object is able to be called with NumPy vectorized arguments and return a resulting vector based on a mathematical relationship.

Parameters

function [callable] A function with signature *function(x1, *args)* that returns a Numpy array of the same shape as its arguments.

name [str] The name for the function as it should be represented in the program and its visualizations.

arity [int] The number of arguments that the *function* takes.

3.4 User-Defined Fitness Metrics

`gplearn.fitness.`**`make_fitness`**(*function*, *greater_is_better*)

Make a fitness measure, a metric scoring the quality of a program's fit.

This factory function creates a fitness measure object which measures the quality of a program's fit and thus its likelihood to undergo genetic operations into the next generation. The resulting object is able to be called with NumPy vectorized arguments and return a resulting floating point score quantifying the quality of the program's representation of the true relationship.

Parameters

function [callable] A function with signature function(y, y_pred, sample_weight) that returns a floating point number. Where *y* is the input target y vector, *y_pred* is the predicted values from the genetic program, and sample_weight is the sample_weight vector.

greater_is_better [bool] Whether a higher value from *function* indicates a better fit. In general this would be False for metrics indicating the magnitude of the error, and True for metrics indicating the quality of fit.

CHAPTER 4

Advanced Use

4.1 Introspecting Programs

If you wish to learn more about how the evolution process came to the final solution, gplearn provides several means to examine the best programs and their parents. Most of these methods are illustrated *in the examples section*.

Both *SymbolicRegressor* and *SymbolicTransformer* overload the print function to output a LISP-style flattened tree representation of the program. Simply print(est) the fitted estimator and the program will be output to your session.

If you would like to see more details about the final programs, you can access the underlying _Program objects which contains several attributes and methods that can yield more information about them.

SymbolicRegressor has a private attribute _program which is a single _Program object that was the fittest found during evolution.

SymbolicTransformer on the other hand has a private attribute _best_programs which is a list of _Program objects of length n_components being the least-correlated and fittest programs found during evolution. *SymbolicTransformer* is also iterable so you can loop through the estimator itself to access each underlying _Program object.

Each _Program object can also be printed as with the estimator themselves to get a readable representation of the programs. They also have several attributes that you can use to further understand the programs:

- raw_fitness_ : The raw fitness of the individual program.
- fitness_ : The penalized fitness of the individual program.
- oob_fitness_ : The out-of-bag raw fitness of the individual program for the held-out samples. Only present when sub-sampling was used in the estimator by specifying max_samples < 1.0.
- depth_ : The maximum depth of the program tree.
- length_ : The number of functions and terminals in the program.

For example with a *SymbolicTransformer*:

```
for program in est_gp:
    print(program)
    print(program.raw_fitness_)

    div(div(X11, X12), X10)
    0.840099070652
    sub(div(mul(X4, X12), div(X9, X9)), sub(div(X11, X12), add(X12, X0)))
    0.814627147552
```

Or if you want to access the individual programs:

```
print(est_gp._best_programs[0])

div(div(X11, X12), X10)
```

And for a *`SymbolicRegressor`*:

```
print(est_gp)
print(est_gp._program)
print(est_gp._program.raw_fitness_)

add(sub(add(X5, div(X5, 0.388)), X0), div(add(X5, X10), X12))
add(sub(add(X5, div(X5, 0.388)), X0), div(add(X5, X10), X12))
4.88966783112
```

You can also plot the programs as a program tree using Graphviz via the `export_graphviz` method of the `_Program` objects. In a Jupyter notebook this is easy using the `pydotplus` package:

```
from IPython.display import Image
import pydotplus
graph = est_gp._program.export_graphviz()
graph = pydotplus.graphviz.graph_from_dot_data(graph)
Image(graph.create_png())
```

This assumes you are satisfied with only seeing the final results, but the relevant programs that led to the final solutions are still retained in the estimator's `_programs` attribute. This object is a list of lists of all of the `_Program` objects that were involved in the evolution of the solution. The first entry in the outer list is the original naive generation of programs while the last entry is the final generation in which the solutions were found.

Note that any programs in earlier generations that were discarded through the selection process are replaced with `None` objects to conserve memory.

Each of the programs in the final solution and the generations that preceded them have a attribute called `parents`. Except for the naive programs from the initial population who have a `parents` value of `None`, this dictionary contains information about how that program was evolved. Its contents differ depending on the genetic operation that was performed on its parents to yield that program:

- **Crossover:**
    - 'method': 'Crossover'
    - 'parent_idx': The index of the parent program in the previous generation.
    - 'parent_nodes': The indices of the nodes in the subtree in the parent program that was replaced.
    - 'donor_idx': The index of the donor program in the previous generation.
    - 'donor_nodes': The indices of the nodes in the subtree in the donor program that was donated to the parent.

- **Subtree Mutation:**
  - ‘method’: ‘Subtree Mutation’
  - ‘parent_idx’: The index of the parent program in the previous generation.
  - ‘parent_nodes’: The indices of the nodes in the subtree in the parent program that was replaced.
- **Hoist Mutation:**
  - ‘method’: ‘Hoist Mutation’
  - ‘parent_idx’: The index of the parent program in the previous generation.
  - ‘parent_nodes’: The indices of the nodes in the parent program that were removed.
- **Point Mutation:**
  - ‘method’: ‘Point Mutation’
  - ‘parent_idx’: The index of the parent program in the previous generation.
  - ‘parent_nodes’: The indices of the nodes in the parent program that were replaced.
- **Reproduction:**
  - ‘method’: ‘Reproduction’
  - ‘parent_idx’: The index of the parent program in the previous generation.
  - ‘parent_nodes’: An empty list as nothing was changed.

The `export_graphviz` also has an optional parameter `fade_nodes` which can take a list of nodes that should be shown as being altered in the visualization. For example if the best program had this parent:

```
print(est_gp._program.parents)

{'parent_idx': 75, 'parent_nodes': [1, 10], 'method': 'Point Mutation'}
```

You could plot its parent with the affected nodes indicated using:

```
idx = est_gp._program.parents['parent_idx']
fade_nodes = est_gp._program.parents['parent_nodes']
print(est_gp._programs[-2][idx])
graph = est_gp._programs[-2][idx].export_graphviz(fade_nodes=fade_nodes)
graph = pydotplus.graphviz.graph_from_dot_data(graph)
Image(graph.create_png())
```

4.2 Exporting

If you want to save your program for later use, you can use the `pickle` or `cPickle` libraries to achieve this:

```
import cPickle as pickle

est = SymbolicRegressor()
est.fit(X_train, y_train)
```

Simply dump your model to a file:

```
with open('gp_model.pkl', 'wb') as f:
    pickle.dump(est, f)
```

You can then load it at another date easily:

```
with open('gp_model.pkl', 'rb') as f:
    est_gp = pickle.load(f)
```

And use it as if it was the Python session where you originally trained the model.

4.3 Custom Functions

This example demonstrates modifying the function set with your own user-defined functions using the *functions.make_function()* factory function.

First you need to define some function which will return a numpy array of the correct shape. Most numpy operations will automatically do this. The factory will perform some basic checks on your function to ensure it complies with this. The function must also protect against zero division and invalid floating point operations (such as the log of a negative number).

For this example we will implement a logical operation where two arguments are compared, and if the first one is larger, return a third value, otherwise return a fourth value:

```
def _logical(x1, x2, x3, x4):
    return np.where(x1 > x2, x3, x4)
```

To make this into a `gplearn` compatible function, we use the factory where we must give it a name for display purposes and declare the arity of the function which must match the number of arguments that your function expects:

```
logical = make_function(function=_logical,
                        name='logical',
                        arity=4)
```

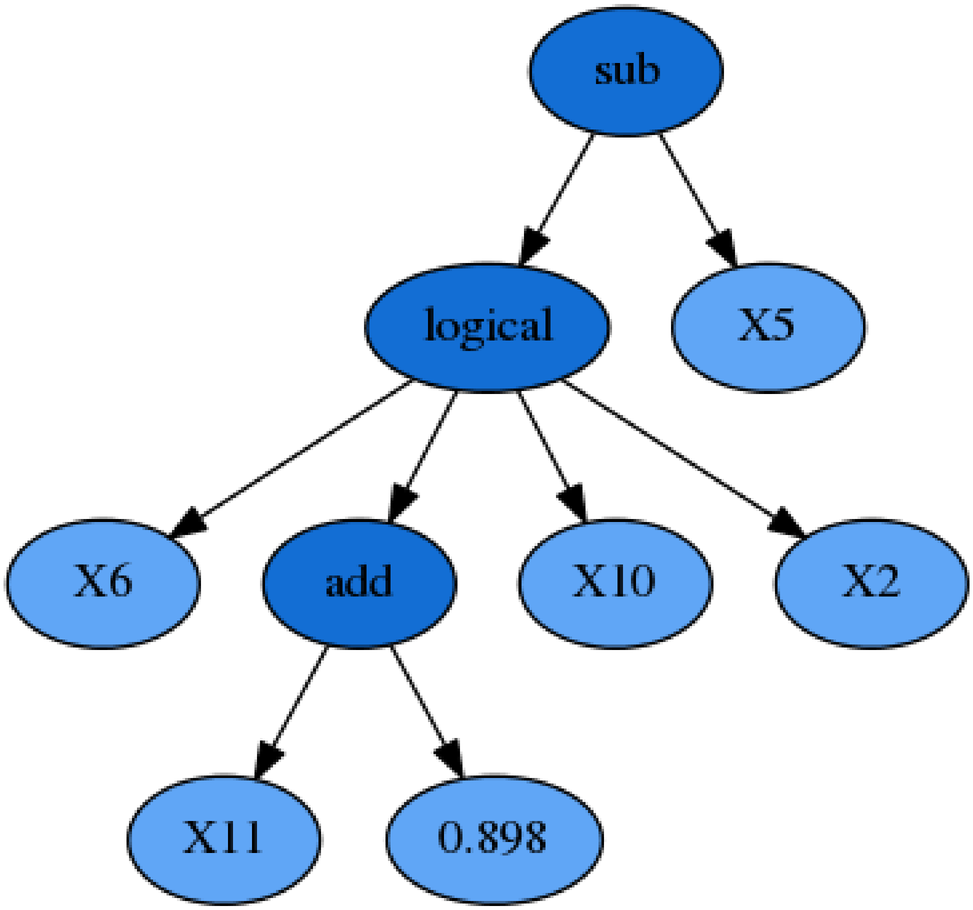
This can then be added to a `gplearn` estimator like so:

```
gp = SymbolicTransformer(function_set=['add', 'sub', 'mul', 'div', logical])
```

Note that custom functions should be specified as the function object name (ie. with no quotes), while built-in functions use the name of the function as a string.

After fitting, you will see some of your programs will have used your own customized functions, for example:

```
sub(logical(X6, add(X11, 0.898), X10, X2), X5)
```



In other mathematical relationships, it may be necessary to ensure the function has *closure*. This means that the function will always return a valid floating point result. Using `np.where`, the user can protect against invalid operations and substitute problematic values with a default such as 0 or 1. One example is the built-in protected division function where infinite values resulting by divide by zero are replaced by 1:

```
def _protected_division(x1, x2):
    with np.errstate(divide='ignore', invalid='ignore'):
        return np.where(np.abs(x2) > 0.001, np.divide(x1, x2), 1.)
```

Or a custom function where floating-point overflow is protected in an exponential function:

```
def _protected_exponent(x1):
    with np.errstate(over='ignore'):
        return np.where(np.abs(x1) < 100, np.exp(x), 0.)
```

For further information on the types of errors that numpy can encounter and what you will need to protect against in your own custom functions, see here.

4.4 Custom Fitness

You can easily create your own fitness measure to have your programs evolve to optimize whatever metric you need. This is done using the `fitness.make_fitness()` factory function. Let's say we wish to measure our programs using MAPE (mean absolute percentage error). First we would need to implement a function that returns this value.

The function must take the arguments y (the actual target values), y_pred (the predicted values from the program) and w (the weights to apply to each sample) to work. For MAPE, a possible solution is:

```
def _mape(y, y_pred, w):
    """Calculate the mean absolute percentage error."""
    diffs = np.abs(np.divide((np.maximum(0.001, y) - np.maximum(0.001, y_pred)),
                             np.maximum(0.001, y)))
    return 100. * np.average(diffs, weights=w)
```

Division by zero must be protected for a metric like MAPE as it is generally used for cases where the target is positive and non-zero (like forecasting demand). We need to keep in mind that the programs begin by being totally naive, so a negative return value is possible. The np.maximum function will protect against these cases, though you may wish to treat this differently depending on your specific use case.

We then create a fitness measure for use in our evolution by using the *fitness.make_fitness()* factory function as follows:

```
mape = make_fitness(_mape, greater_is_better=False)
```

This fitness measure can now be used to evolve a program that optimizes for your specific needs by passing the new fitness object to the metric parameter when creating an estimator:

```
est = SymbolicRegressor(metric=mape, verbose=1)
```

4.5 Continuing Evolution

If you are evolving a lot of generations in your training session, but find that you need to keep evolving more, you can use the warm_start parameter in both *SymbolicRegressor* and *SymbolicTransformer* to continue evolution beyond your original estimates. To do so, start evolution as usual:

```
est = SymbolicRegressor(generations=10)
est.fit(X, y)
```

If you then need to add further generations, simply change the generations and warm_start attributes and fit again:

```
est.set_params(generations=20, warm_start=True)
est.fit(X, y)
```

Evolution will then continue for a further 10 generations without losing the programs that had been previously trained.

CHAPTER 5

Installation

gplearn requires a recent version of scikit-learn (which requires numpy and scipy). So first you will need to follow their installation instructions to get the dependencies.

Now that you have scikit-learn installed, you can install gplearn using pip:

```
pip install gplearn
```

Or if you wish to install to the home directory:

```
pip install --user gplearn
```

For the latest development version, first get the source from github:

```
git clone https://github.com/trevorstephens/gplearn.git
```

Then navigate into the local gplearn directory and simply run:

```
python setup.py install
```

or:

```
python setup.py install --user
```

and you're done!

CHAPTER 6

Contributing

gplearn welcomes your contributions! Whether it is a bug report, bug fix, new feature or documentation enhancements, please help to improve the project!

In general, please follow the scikit-learn contribution guidelines for how to contribute to an open-source project.

If you would like to open a bug report, please open one here. Please try to provide a Short, Self Contained, Example so that the root cause can be pinned down and corrected more easily.

If you would like to contribute a new feature or fix an existing bug, the basic workflow to follow (as detailed more at the scikit-learn link above) is:

- Open an issue with what you would like to contribute to the project and its merits. Some features may be out of scope for gplearn, so be sure to get the go-ahead before working on something that is outside of the project's goals.
- Fork the gplearn repository, clone it locally, and create your new feature branch.
- Make your code changes on the branch, commit them, and push to your fork.
- Open a pull request.

Please ensure that:

- Only data-dependent arguments should be passed to the fit/transform methods (X, y, sample_weight), and conversely, no data should be passed to the estimator initialization.
- No input validation occurs before fitting the estimator.
- Any new feature has great test coverage.
- Any new feature is well documented with numpy-style docstrings & an example, if appropriate and illustrative.
- Any bug fix has regression tests.
- Comply with PEP8.

Currently gplearn uses Travis CI for testing, Coveralls for code coverage reports, and Landscape for code quality checks. These applications should automatically run on your new pull request to give you guidance on any problems in the new code.

CHAPTER 7

Release History

7.1 Version 0.3.0

- Fixed two bugs in *genetic.SymbolicTransformer* where the final solution selection logic was incorrect and suboptimal. This fix will change the solutions from all previous versions of *gplearn*. Thanks to iblasi for diagnosing the problem and helping craft the solution.
- Fixed bug in *genetic.SymbolicRegressor* where a custom fitness measure was defined in *fitness.make_fitness()* with the parameter *greater_is_better=True*. This was ignored during final solution selection. This change will alter the results from previous releases where *greater_is_better=True* was set in a custom fitness measure. By sun ao.
- Increase minimum required version of `scikit-learn` to 0.18.1. This allows streamlining the test suite and removal of many utilities to reduce future technical debt. **Please note that due to this change, previous versions may have different results** due to a change in random sampling noted here.
- Drop support for Python 2.6 and add support for Python 3.5 and 3.6 in order to support the latest release of `scikit-learn` 0.19 and avoid future test failures. By hugovk.

7.2 Version 0.2.0

- Allow more generations to be evolved on top of those already trained using a previous call to fit. The *genetic.SymbolicRegressor* and *genetic.SymbolicTransformer* classes now support the `warm_start` parameter which, when set to `True`, reuse the solution of the previous call to fit and add more generations to the evolution.
- Allow users to define their own fitness measures. Supported by the *fitness.make_fitness()* factory function. Using this a user may define any metric by which to measure the fitness of a program to optimize any problem. This also required modifying the API slightly with the deprecation of the `'rmsle'` error measure for the *genetic.SymbolicRegressor*.
- Allow users to define their own functions for use in genetic programs. Supported by the *functions.make_function()* factory function. Using this a user may define any mathematical relationship with

any number of arguments and grow totally customized programs. This also required modifying the API with the deprecation of the `'comparison'`, `'transformer'` and `'trigonometric'` arguments to the *genetic.SymbolicRegressor* and *genetic.SymbolicTransformer* classes in favor of the new `function_set` where any combination of preset and user-defined functions can be supplied. To restore previous behavior initialize the estimator with `function_set=['add2', 'sub2', 'mul2', 'div2', 'sqrt1', 'log1', 'abs1', 'neg1', 'inv1', 'max2', 'min2']`.

- Reduce memory consumption for large datasets, large populations or many generations. Indices for in-sample/out-of-sample fitness calculations are now generated on demand rather than being stored in the program objects which reduces the size significantly for large datasets. Additionally "irrelevant" programs from earlier generations are removed if they did not contribute to the current population through genetic operations. This reduces the number of programs stored in the estimator which helps for large populations, high number of generations, as well as for runs with significant bloat.

7.3 Version 0.1.0

Initial public release supporting symbolic regression tasks through the *genetic.SymbolicRegressor* class for regression problems and the *genetic.SymbolicTransformer* class for automated feature engineering.

Bibliography

[1] 10. Koza, “Genetic Programming”, 1992.

[2] 18. Poli, et al. “A Field Guide to Genetic Programming”, 2008.

[1] 10. Koza, “Genetic Programming”, 1992.

[2] 18. Poli, et al. “A Field Guide to Genetic Programming”, 2008.

Index