fastgplearn

Release 0.0.14

boliqq07

Nov 13, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION

FastGPLearn implements Genetic Programming in Python, with a `scikit-learn` inspired and compatible API. And the fastgplearn applied the `torch` and `numpy` backend for fast calculated, make it accessible for `CUDA` .

While Genetic Programming (GP) can be used to perform a very wide variety of tasks, FastGPLearn is purposefully constrained to solving symbolic regression problems.

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.

The optional operator including (`'add'`, `'sub'`, `'mul'`, `'div'`, `'ln'`, `'exp'`, `'pow2'`, `'pow3'`, `'rec'`, `'max'`, `'min'`, `'sin'`, `'cos'`).

FastGPLearn retains the familiar `scikit-learn` **fit/predict** API. You can get started with FastGPLearn as simply as:

```
>>> from fastgplearn.skflow import SymbolicRegressor
>>> est = SymbolicRegressor()
>>> est.fit(X_train, y_train)
>>> y_test_pred = est.predict(X_test)
>>> test_score = est.score(X_test,y_test)
```

FastGPLearn supports regression through the *fastgplearn.skflow.SymbolicRegressor* , binary classification with the *fastgplearn.skflow.SymbolicClassifier* .

NOW, TRY IT !

CHAPTER

TWO

INSTALL

To suit the needs of different people, the installation is divided into 3 optional parts.

1. Install with pip:

```
pip install fastgplearn
```

Note: The package has been installed, but advise to use c++ compiler and torch for more power speed up, as 2 and 3. See more speed test: *Benchmark* .

2. optional:

```
fastgplearn cc_numpy
```

Note: For windows platform, C++14 or more needed (Note help, VS Buildtools, Proposed English version.)

3. optional:

```
fastgplearn cc_torch
```

Note: Torch is needed (pytorch.org), For linux, windows platform, C++14 or more needed (Note help, VS Buildtools, Proposed English version.)

2.1 Requirements

Packages:

Dependence	Name	Version
necessary	python	>=3.6
necessary	numpy	>=1.17
recommend	torch	>=1.7

CHAPTER

THREE

GUIDE

3.1 Parameters details

For both *fastgplearn.skflow.SymbolicRegressor* and *fastgplearn.skflow.SymbolicClassifier* .

3.1.1 Init Parameters:

Parameters name	Type	Default	Suggest Range	Definition
population_size	(int)	10000	[50, 1000000]	number of population
generations	(int)	20	[10,...]	number of generations
tournament_size	(int)	20	[5,20]	number of for each turn of tournament
stopping_criteria	(float)	0.95	[0,1]	early stopping criteria
constant_range	tuple of float	(0,1.0)	/	constants would choice from range randomly
constants	tuple of float	None	/	if given, the parameter constant_range would be ignored, and just use the constants offered
depth	tuple of float	(2,5)	1st [1,...], 2ed [2,8)	(min_depth,max_depth), keep the max of depth is not more than 8 !
function_set	tuple of string	(+-*/)	/	optional: ('add', 'sub', 'mul', 'div', "ln", "exp", "pow2", "pow3", "rec", "max", "min", "sin", "cos")
n_jobs	(int)	1	[1,...]	n jobs to parallel
verbose	(bool)	True	True,False	print message
p_mutate	(float)	0.5	(0,1)	mutate probability
p_crossover	(float)	0.5	(0,1)	crossover probability
random_state	(int)	None	/	random state
hall_of_fame	(int)	3	[0,10]	hall of frame number to add to next generation
store_of_fame	(int)	3	[0,10]	hall of frame number to return result
method_backend	(string)	"p_numpy"		optional: ("p_numpy","c_numpy","p_torch","c_torch")
device	(string)	"cpu"	/	optional: ("cpu","cuda:0", ...) depend on your computer device
func_p	(np.ndarray)	None	/	specific the probability values of each function
sci_template	list, str	"default"	/	user self-defined list template or "default" or None

3.1.2 Fit Parameters:

Fit parameters in `SymbolicRegressor().fit()` or `SymbolicClassifier().fit()` method.

Parameters name	Type	Default	Suggest Range	Definition
X	(np.ndarray)	/	/	with shape (n_sample, n_fea)
y	(np.ndarray)	/	/	with shape (n_sample,)
xs_p	(np.ndarray)	None	/	specific the probability values of each feature
x_label	list of string	None	/	specific the name values of each feature

3.1.3 Other Parameters:

Other parameters present in `predict()` or `score()`, or `top_n()` method.

Parameters name	Type	Default	Suggest Range	Definition
X	(np.ndarray)	/	/	with shape (n_sample, n_fea)
y	(np.ndarray)	/	/	with shape (n_sample,)
n	(int)	0	0	specify the number of expression to calculate or score
scoring	(str)	/	/	for regression, default is "r2", for classification is "accuracy"

3.2 Benchmark

Test device: E5 2680 v4, 14 core.

3.2.1 Package Comparison

This comparison for different packages (gplearn, BindingGP) speed.

Test datasets: boston datasets from `scikit-learn` .

pop*gen times (s)	fastgplearn	gplearn	BindingGP
1000*10	0.15	4.91	29.09
10000*10	1.49	50.66	295.34
10000*10 (8 core)	1.45	57.01	126.20
100000*10	15.8	490.00	3053.01
100000*10 (8 core)	11.5	422.67	1452.58

3.2.2 Backend Comparison

This comparison for different backend and their parallelization performance.

Conclusion

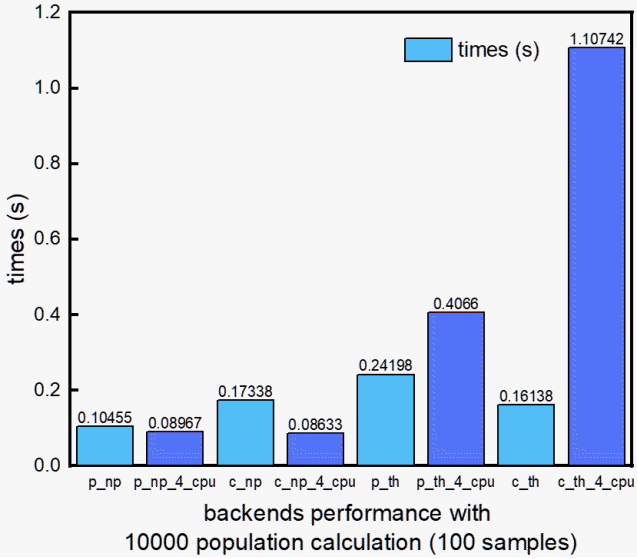
1. For large `samples` datasets (more than 1000), `torch` > `numpy`.
2. For any `population` variation, `numpy` > `torch`.
3. For "c_torch", keep n_jobs==1. c_torch is already well parallelized in c++ level, and does not need to be parallelized in python code.

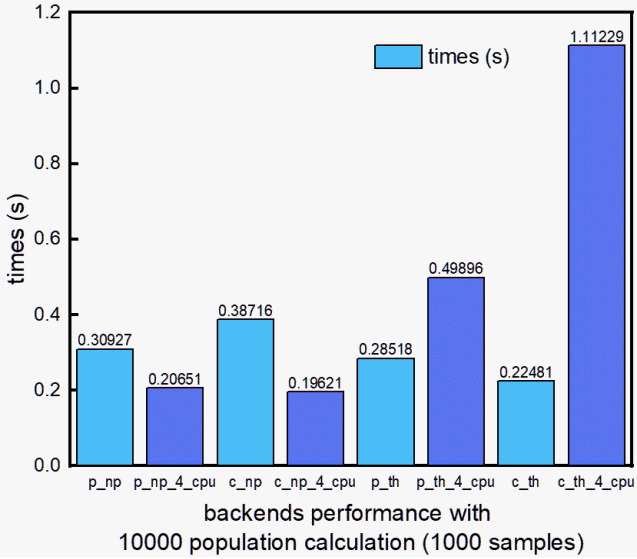
Code

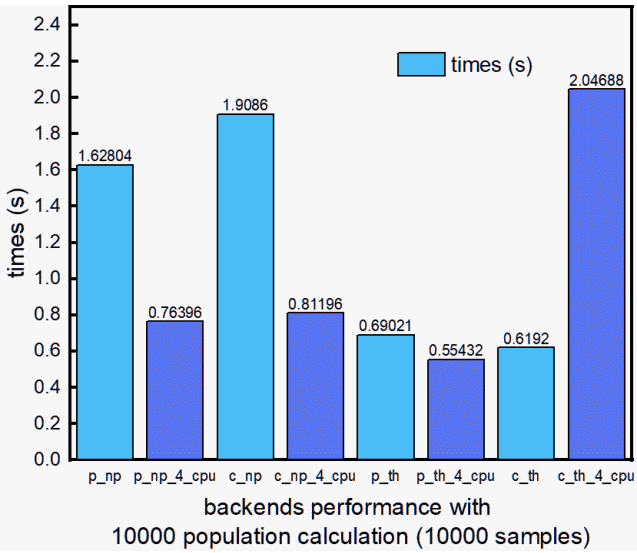
```
>>> np.random.seed(0)
>>> deps = generate_random(func_num=13, xs_num=10, pop_size=10000, depth_min=1, depth_
↪max=3)
>>> deps = deps.tolist()
```

```
>>> "p_numpy"
>>> rs1 = p_np_score(deps, x, y, func_index=func_index)
>>> "p_numpy"
>>> rs2 = p_np_score_mp(deps, x, y, func_index=func_index, n_jobs=4)
>>> "c_numpy"
>>> rs3 = c_np_score(deps, x, y, func_index=func_index)
>>> "c_numpy"
>>> rs4 = c_np_score_mp(deps, x, y, func_index=func_index, n_jobs=4)
```

```
>>> "p_torch"
>>> rs1 = p_torch_score(deps, x2, y2, func_index=func_index)
>>> "p_torch"
>>> rs2 = p_torch_score_mp(deps, x2, y2, func_index=func_index, n_jobs=4)
>>> "c_torch"
>>> rs3 = c_torch_score(deps, x2, y2, func_index=func_index)
>>> "c_torch"
>>> rs4 = c_torch_score_mp(deps, x2, y2, func_index=func_index, n_jobs=4)
```







3.3 Formula Template

Add user-preferential formula template restriction could accelerate the computation.

Parameter `sci_template` could be `"default"`, `None`, or user-defined template (type: list).

1. `"default"`: We built-in some formula templates. Such as $y = exp(?/?)$.
2. `None`: Not use template.
3. User-defined template (type: list): consult the author for details.

For target problem:

$y = exp((X_0 + X_2)/X_1)$

The `"default"` would be far faster than `None` to find the target expression, It is natural.

Test Code:

```
>>> import numpy as np
>>> from fastgplearn.skflow import SymbolicRegressor
>>> np.random.seed(0)
>>> x = np.random.random(size=(100, 10))
>>> x = x + 1
>>> x[:, 0] = 5*x[:, 0]
>>> x[:, 2] = 5*x[:, 2]
>>> y = np.random.random(size=100) * 0.01 + np.exp((x[:, 0]+x[:, 2]) / x[:, 1],)
>>> x = x.astype(np.float32)
>>> y = y.astype(np.float32)
```

With template:

```
>>> sr1 = SymbolicRegressor(population_size=10000, generations=30, stopping_criteria=1.0,
>>>                         constant_range=(0, 1.0), depth=(2, 4),
>>>                         function_set=('add', 'sub', 'mul', 'div',"exp"),random_
↪state=0,
>>>                         sci_template="default")
>>>                         # sci_template=None)
>>> sr1.fit(x, y)
>>> sr1.top_n(30)
```

```
    Person Corr Results
---------------------------
gen 1: 0.9776749610900879
gen 2: 0.9776749610900879
gen 3: 0.9776749610900879
gen 4: 0.9787307381629944
gen 5: 0.9787307381629944
gen 6: 0.9787307381629944
gen 7: 0.9787307381629944
gen 8: 0.9787307381629944
gen 9: 1.0
Reach the stopping criteria and terminate early at generation 9
fit time 1.4015464782714844

The top 30 result:
Scoring by r2: ( score, expression, coef, intercept )
(0.9999999999999033, 'exp_(div_(add_(x2,x0),x1))', 1.0000004, -0.25)
(0.9579139337012704, 'div_(sub_(div_(x2,x6),x6),add_(sub_(x1,x8),div_
```

Without template:

```
>>> sr2 = SymbolicRegressor(population_size=10000, generations=30, stopping_criteria=1.0,
```

(continues on next page)

(continued from previous page)

```
>>>                          constant_range=(0, 1.0), depth=(2, 4),
>>>                          function_set=('add', 'sub', 'mul', 'div',"exp"),random_
↪state=0,
>>>                          #sci_template="default")
>>>                          sci_template=None)
>>> sr2.fit(x, y)
>>> sr2.top_n(30)
```

```
    Person Corr Results
----------------------------
gen 1: 0.9776749610900879
gen 2: 0.9776749610900879
gen 3: 0.9776749610900879
gen 4: 0.9776749610900879
gen 5: 0.9776749610900879
gen 6: 0.9788792133331299
gen 7: 0.9788792133331299
gen 8: 0.9788792133331299
gen 9: 0.9788792133331299
gen 10: 0.9788792133331299
gen 11: 0.9788792133331299
gen 12: 0.9788792133331299
gen 13: 0.9788792133331299
gen 14: 0.9788792133331299
gen 15: 0.981613278388977
gen 16: 0.981613278388977
gen 17: 0.981613278388977
gen 18: 0.981613278388977
gen 19: 0.981613278388977
gen 20: 0.981613278388977
```

CHAPTER

FOUR

FASTGPLEARN

4.1 fastgplearn package

4.1.1 Subpackages

fastgplearn.backend package

Submodules

fastgplearn.backend.p_numpy module

fastgplearn.backend.p_numpy.**cos_**(*a*, *b*)

fastgplearn.backend.p_numpy.**exp_**(*a*, *b*)

fastgplearn.backend.p_numpy.**find_add_mask**(*popi*, *single_start=6*)

fastgplearn.backend.p_numpy.**find_add_mask_all**(*pop*, *single_start=6*)

fastgplearn.backend.p_numpy.**find_used_index**(*popi*, *single_start=6*)

fastgplearn.backend.p_numpy.**find_used_index_total**(*pop*, *single_start=6*)

fastgplearn.backend.p_numpy.**get_corr_together**(*fake_ys*, *y*)

Parameters

- **fake_ys** (*np.ndarray*) – with shape (n_results, n_sample,).
- **y** (*np.ndarray*) – with sample (n_sample,).

Returns with shape (n_result,)

Return type corr (np.ndarray)

fastgplearn.backend.p_numpy.**get_sort_accuracy_together**(*fake_ys*, *y*)

Parameters

- **fake_ys** (*np.ndarray*) – with shape (n_results, n_sample,).
- **y** (*np.ndarray*) – with sample (n_sample,).

Returns with shape (n_result,)

Return type corr (np.ndarray)

fastgplearn.backend.p_numpy.**ln_**(*a*, *b*)

fastgplearn.backend.p_numpy.**max_**(*a*, *b*)

fastgplearn.backend.p_numpy.**min_**(*a*, *b*)

fastgplearn.backend.p_numpy.**p_np_cal**(*ve*, *xs*, *y*, *func_index=None*, *single_start=6*)
Batch calculate.

fastgplearn.backend.p_numpy.**p_np_score**(*ve*, *xs*, *y*, *func_index*, *clf=False*, *single_start=6*)
Batch score.

fastgplearn.backend.p_numpy.**p_np_score_mp**(*ve*, *xs*, *y*, *func_index=None*, *n_jobs=1*, *clf=False*, *single_start=6*)
Batch score with n_jobs.

fastgplearn.backend.p_numpy.**p_np_str_name**(*ve*, *xns*, *cns=None*, *y=None*, *func_index=None*, *real_names=None*)
Batch get name of expression,(without coef and intercept).

fastgplearn.backend.p_numpy.**pow2_**(*a*, *b*)

fastgplearn.backend.p_numpy.**pow3_**(*a*, *b*)

fastgplearn.backend.p_numpy.**rec_**(*a*, *b*)

fastgplearn.backend.p_numpy.**sin_**(*a*, *b*)

fastgplearn.backend.p_torch module

fastgplearn.backend.p_torch.**cos_**(*a*, *b*)

fastgplearn.backend.p_torch.**exp_**(*a*, *b*)

fastgplearn.backend.p_torch.**get_corr_together**(*fake_ys*, *y*)

Parameters

- **fake_ys** (*torch.Tensor*) – with shape (n_results, n_sample,).
- **y** (*torch.Tensor*) – with sample (n_sample,).

Returns with shape (n_result,)

Return type corr (torch.Tensor)

fastgplearn.backend.p_torch.**get_sort_accuracy_together**(*fake_ys*, *y*)

Parameters

- **fake_ys** (*torch.ndarray*) – with shape (n_results, n_sample,).
- **y** (*torch.ndarray*) – with sample (n_sample,).

Returns with shape (n_result,)

Return type corr (torch.ndarray)

fastgplearn.backend.p_torch.**ln_**(*a*, *b*)

fastgplearn.backend.p_torch.**max_**(*a*, *b*)

fastgplearn.backend.p_torch.**min_**(*a*, *b*)

fastgplearn.backend.p_torch.**p_torch_cal**(*ve*, *xs*, *y*, *func_index=None*, *single_start=6*)
Batch calculate.

fastgplearn.backend.p_torch.**p_torch_score**(*ve*, *xs*, *y*, *func_index*, *return_numpy=False*, *clf=False*, *single_start=6*)
Batch score.

fastgplearn.backend.p_torch.**p_torch_score_mp**(*ve*, *xs*, *y*, *func_index=None*, *n_jobs=1*, *return_numpy=False*, *clf=False*, *single_start=6*)
Batch score with n_jobs. It's slow!!!

fastgplearn.backend.p_torch.**pow2_**(*a*, *b*)

fastgplearn.backend.p_torch.**pow3_**(*a*, *b*)

fastgplearn.backend.p_torch.**rec_**(*a*, *b*)

fastgplearn.backend.p_torch.**sin_**(*a*, *b*)

fastgplearn.cli package

Submodules

fastgplearn.cli.cc_numpy module

class fastgplearn.cli.cc_numpy.**CLICommand**
Bases: object

Compile pyx or c++ code.

Example

$ fastgplearn cc_numpy

static add_arguments(*parser*)

static run(*args*, *parser*)

fastgplearn.cli.cc_torch module

class fastgplearn.cli.cc_torch.**CLICommand**
Bases: object

Compile pyx or c++ code.

Example

$ fastgplearn cc_torch

static add_arguments(*parser*)

static run(*args*, *parser*)

fastgplearn.cli.main module

exception fastgplearn.cli.main.**CLIError**
Bases: Exception

Error for CLI commands.

A subcommand may raise this. The message will be forwarded to the error() method of the argument args.

class fastgplearn.cli.main.**Formatter**(*prog*, *indent_increment=2*, *max_help_position=24*, *width=None*)
Bases: argparse.HelpFormatter

Improved help formatter.

fastgplearn.cli.main.**main**(*prog='fastgplearn'*, *description='fastgplearn command line tool.'*, *args=None*)

4.1.2 Submodules

4.1.3 fastgplearn.gp module

fastgplearn.gp.**choice**(*a*, *size=None*, *replace=True*, *p=None*)
Generates a random sample from a given 1-D array

New in version 1.7.0.

Note: New code should use the choice method of a default_rng() instance instead; please see the random-quick-start.

Parameters

- **a** (*1-D array-like or int*) – If an ndarray, a random sample is generated from its elements. If an int, the random sample is generated as if it were np.arange(a)
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. Default is None, in which case a single value is returned.
- **replace** (*boolean, optional*) – Whether the sample is with or without replacement. Default is True, meaning that a value of a can be selected multiple times.
- **p** (*1-D array-like, optional*) – The probabilities associated with each entry in a. If not given, the sample assumes a uniform distribution over all entries in a.

Returns **samples** – The generated random samples

Return type single item or ndarray

Raises **ValueError** – If a is an int and less than zero, if a or p are not 1-dimensional, if a is an array-like of size 0, if p is not a vector of probabilities, if a and p have different lengths, or if replace=False and the sample size is greater than the population size

See also:

randint, shuffle, permutation

Generator.choice which should be used in new code

Notes

Setting user-specified probabilities through `p` uses a more general but less efficient sampler than the default. The general sampler produces a different sample than the optimized sampler even if each element of `p` is 1 / len(a).

Sampling random rows from a 2-D array is not possible with this function, but is possible with *Generator.choice* through its `axis` keyword.

Examples

Generate a uniform random sample from np.arange(5) of size 3:

```
>>> np.random.choice(5, 3)
array([0, 3, 4]) # random
>>> #This is equivalent to np.random.randint(0,5,3)
```

Generate a non-uniform random sample from np.arange(5) of size 3:

```
>>> np.random.choice(5, 3, p=[0.1, 0, 0.3, 0.6, 0])
array([3, 3, 0]) # random
```

Generate a uniform random sample from np.arange(5) of size 3 without replacement:

```
>>> np.random.choice(5, 3, replace=False)
array([3,1,0]) # random
>>> #This is equivalent to np.random.permutation(np.arange(5))[:3]
```

Generate a non-uniform random sample from np.arange(5) of size 3 without replacement:

```
>>> np.random.choice(5, 3, replace=False, p=[0.1, 0, 0.3, 0.6, 0])
array([2, 3, 0]) # random
```

Any of the above can be repeated with an arbitrary array-like instead of just integers. For instance:

```
>>> aa_milne_arr = ['pooh', 'rabbit', 'piglet', 'Christopher']
>>> np.random.choice(aa_milne_arr, 5, p=[0.5, 0.1, 0.1, 0.3])
array(['pooh', 'pooh', 'pooh', 'Christopher', 'piglet'], # random
      dtype='<U11')
```

`fastgplearn.gp.`**`crossover`**(*pop_np*, *p_crossover=0.5*)

Corssover.

Parameters

- **pop_np** (*np.ndarray*) – population
- **p_crossover** (*float*) – probability for crossover.

Returns population with shape (n_pop,2**depth_max).

Return type pop (np.ndarray)

`fastgplearn.gp.`**`csub_science`**(*pop*, *sci_template*)

This would change the init pop!!!

pyx version for sci substitute.

fastgplearn.gp.**generate_random**(*func_num*, *xs_num*, *pop_size=10*, *depth_min=1*, *depth_max=5*, *p=None*, *func_p=None*, *xs_p=None*)

Generate the first population. Each individual with ordered: [mark,1,2,3,4,5,6,7, 103,102,100,102,102,103,102,100] 1.First part: mark index of root. 2.second part: index of x gene and f gen. 3.Third part: protect index of x gene.

Parameters

- **func_num** (*int*) – func number.
- **xs_num** (*int*) – x number (n_fea).
- **pop_size** (*int*) – population size.
- **depth_min** (*int*) – min depth of expression.
- **depth_max** (*max*) – max depth of expression.
- **p** (*None*) – (just for test).
- **func_p** (*np.ndarray*) – with shape of (n_func), probability,.
- **xs_p** (*np.ndarray*) – with shape of (n_fea), probability.

Returns with shape (n_pop,2**depth_max), population .

Return type pop (np.ndarray)

fastgplearn.gp.**mutate**(*mutate_pop*, *func_num*, *xs_num*, *depth_min=1*, *depth_max=5*, *p_mutate=0.8*, *p=None*, *func_p=None*, *xs_p=None*)

Mutate. Each individual with ordered: [mark,1,2,3,4,5,6,7, 103,102,100,102,102,103,102,100] 1.First part: mark index of root. 2.second part: index of x gene and f gen. 3.Third part: protect index of x gene.

Parameters

- **func_num** (*int*) – func number.
- **mutate_pop** (*np.ndarray*) – with shape (n_pop,2**depth_max),population.
- **xs_num** (*int*) – x number (n_fea).
- **depth_min** (*int*) – min depth of expression.
- **depth_max** (*max*) – max depth of expression.
- **p** (*None*) – (just for test).
- **func_p** (*np.ndarray*) – with shape of (n_func), probability,.
- **xs_p** (*np.ndarray*) – with shape of (n_fea), probability.
- **p_mutate** (*flaot*) – probability for mutate.

Returns population with shape (n_pop,2**depth_max).

Return type pop (np.ndarray)

fastgplearn.gp.**mutate_random**(*pop_np*, *func_num*, *xs_num*, *pop_size=10*, *depth_min=1*, *depth_max=5*, *p_mutate=0.8*, *p=None*, *func_p=None*, *xs_p=None*)

Mutate. Each individual with ordered: [mark,1,2,3,4,5,6,7, 103,102,100,102,102,103,102,100] 1.First part: mark index of root. 2.second part: index of x gene and f gen. 3.Third part: protect index of x gene.

Parameters

- **func_num** (*int*) – func number.
- **pop_size** (*int*) – population size.

- **pop_np** (*np.ndarray*) – with shape (n_pop,2**depth_max),population.
- **xs_num** (*int*) – x number (n_fea).
- **depth_min** (*int*) – min depth of expression.
- **depth_max** (*max*) – max depth of expression.
- **p** (*None*) – (just for test).
- **func_p** (*np.ndarray*) – with shape of (n_func), probability,.
- **xs_p** (*np.ndarray*) – with shape of (n_fea), probability.
- **p_mutate** (*flaot*) – probability for mutate.

Returns population with shape (n_pop,2**depth_max).

Return type pop (np.ndarray)

fastgplearn.gp.**mutate_sci**(*func_num*, *xs_num*, *pop_size=10*, *depth_min=1*, *depth_max=5*, *p=None*, *func_p=None*, *xs_p=None*, *sci_template=None*)

Mutate. Each individual with ordered: [mark,1,2,3,4,5,6,7, 103,102,100,102,102,103,102,100] 1.First part: mark index of root. 2.second part: index of x gene and f gen. 3.Third part: protect index of x gene.

Parameters

- **func_num** (*int*) – func number.
- **pop_size** (*int*) – population size.
- **xs_num** (*int*) – x number (n_fea).
- **depth_min** (*int*) – min depth of expression.
- **depth_max** (*max*) – max depth of expression.
- **p** (*None*) – (just for test).
- **func_p** (*np.ndarray*) – with shape of (n_func), probability,.
- **xs_p** (*np.ndarray*) – with shape of (n_fea), probability.
- **sci_template** (*list of list*) – the science expression templates.

Returns population with shape (n_pop,2**depth_max).

Return type pop (np.ndarray)

fastgplearn.gp.**select_index**(*score*, *num_percent=0.3*, *method='tournament'*, *tour_num=3*)

Selection.

Parameters

- **score** (*np.ndarray*) – score with shape (n_res,)/
- **num_percent** (*int*,*float*) – number or percent of population.
- **method** (*str*) – "tournament" or "k_best".
- **tour_num** (*int*) – tournament size .

Returns index of selection to population.

Return type index (np.ndarray)

fastgplearn.gp.**set_seed**(*seed*)

Set random seed.

fastgplearn.gp.**sub_re_hall99**(*inds*, *func_num*, *xs_num*)
sub the 99 in halls.

fastgplearn.gp.**sub_science**(*pop*, *sci_template*)
This would change the init pop!!! sci substitute.

4.1.4 fastgplearn.sci_formula module

4.1.5 fastgplearn.skflow module

class fastgplearn.skflow.**SymbolicClassifier**(*population_size=10000*, *generations=20*, *stopping_criteria=0.95*, *store_of_fame=50*, *hall_of_fame=3*, *store=False*, *p_mutate=0.2*, *p_crossover=0.5*, *select_method='tournament'*, *tournament_size=5*, *device='cpu'*, *sci_template=None*, *constant_range=None*, *constants=None*, *depth=(2, 4)*, *function_set=('add', 'sub', 'mul', 'div', 'pow2', 'pow3')*, *n_jobs=1*, *verbose=0*, *random_state=None*, *method_backend='p_numpy'*, *func_p=None*)

Bases: *fastgplearn.skflow.SymbolicEstimator*

A Genetic Programming symbolic classifier.

A symbolic classifier 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.

The default score for find expression is accuracy.

Examples:

```
>>> from fastgplearn.skflow import SymbolicRegressor
>>> est_gp = SymbolicRegressor(population_size=5000,
...                     generations=20, stopping_criteria=0.01,
...                     p_crossover=0.7, p_mutate_=0.1,
...                     max_samples=0.9, verbose=1,
...                     random_state=0)
>>> est_gp.fit(X_train, y_train)
>>> est_gp.top_n()
>>> test_score = est_gp.score(X_test,y_test)
```

Parameters

- **population_size** (*int*) – number of population, default 10000.
- **generations** (*int*) – number of generations, default 20.
- **tournament_size** (*int*) – tournament size for selection.
- **stopping_criteria** (*float*) – criteria of correlation score, max 1.0.
- **constant_range** (*tuple*) – floats. constant_range=(0,1.0)
- **constants** (*tuple*) – floats. constants=(-1,1,2,10), if given, The parameter constant_range would be ignored.
- **depth** (*tuple*) – default (2, 5), The max of depth is not more than 8.

- **function_set** (*tuple*) – tuple of str. optional: ('add', 'sub', 'mul', 'div',"max", "min", "ln", "exp", "pow2", "pow3", "rec", "sin", "cos").
- **n_jobs** (*int*) – n jobs to parallel.
- **verbose** (*bool*) – print message.
- **p_mutate** – mutate probability.
- **p_crossover** (*float*) – crossover probability.
- **random_state** (*int*) – random state
- **hall_of_fame** (*int*) – hall of frame number to add to next generation.
- **store_of_fame** (*int*) – hall of frame number to return result.
- **method_backend** (*str*) – optional: ("p_numpy","c_numpy","p_torch","c_torch")
- **device** (*str*) – default "cpu", "cuda:0", only accessible of torch.
- **func_p** (*np.ndarray,tuple*) – with shape (n_function,), probability values of each function.
- **sci_template** (*str,list*) – None, "default" or user self-defined list template, default None.

best_expression(*scoring='accuracy'*)
Print the best expression.

static cla(*pre_y*)
classification tool.

fit(*X: numpy.ndarray*, *y: numpy.ndarray*, *xs_p: Optional[numpy.ndarray] = None*, *x_label=None*)
Fitting.

Parameters

- **X** (*np.ndarray*) – with shape (n_sample,n_fea).
- **y** (*np.ndarray*) – with shape (n_sample,).
- **xs_p** (*np.ndarray*) – with shape (n_fea,), probability values of each xi.
- **x_label** (*np.ndarray*) – with shape (n_fea), names of xi.

predict(*X*, *y=None*, *n=0*)
Return the real predicted y.

Parameters

- **X** (*np.ndarray*) – array-like of shape (n_samples, n_features).
- **vectors** (*Input*) –
- **features** (*where n_samples is the number of samples and n_features is the number of*) –
- **y** (*np.ndarray*) – array-like of shape (n_samples,).
- **n** –

Returns array-like of shape (n_samples,).

Return type y (np.ndarray)

score(*X*, *y*, *scoring='accuracy'*, *n=0*)
Return the mean accuracy on the given test data and labels.

Parameters

- **X** (*np.ndarray*) – array-like of shape (n_samples, n_features).
- **y** (*np.ndarray*) – array-like of shape (n_samples,).
- **scoring** (*str*) – see also sklearn.metrics.
- **n** (*int*) – calculate by the n_ed expression.

Returns Mean accuracy of `self.predict(X)` wrt. *y*.

Return type score (float)

single_coef_logistic(*X*, *y*)
Fitting by sklearn.linear_model.LogisticRegression.

top_n(*n=0*, *scoring='accuracy'*)
Print the top n result. The best one is index 0.

Parameters

- **scoring** (*str*) – see also sklearn.metrics.
- **n** (*int*) – calculate by the n_ed expression.

class fastgplearn.skflow.**SymbolicEstimator**(*population_size=10000*, *generations=20*, *stopping_criteria=0.95*, *store_of_fame=50*, *hall_of_fame=3*, *store=False*, *p_mutate=0.2*, *p_crossover=0.5*, *select_method='tournament'*, *tournament_size=5*, *device='cpu'*, *sci_template=None*, *constant_range=None*, *constants=None*, *depth=(2, 5)*, *function_set=('add', 'sub', 'mul', 'div', 'pow2', 'pow3')*, *n_jobs=1*, *verbose=0*, *random_state=None*, *method_backend='p_numpy'*, *func_p=None*)

Bases: `sklearn.base.BaseEstimator`, `abc.ABC`

Parameters

- **population_size** (*int*) – number of population, default 10000.
- **generations** (*int*) – number of generations, default 20.
- **tournament_size** (*int*) – tournament size for selection.
- **stopping_criteria** (*float*) – criteria of correlation score, max 1.0.
- **constant_range** (*tuple*) – floats. constant_range=(0,1.0)
- **constants** (*tuple*) – floats. constants=(-1,1,2,10), if given, The parameter constant_range would be ignored.
- **depth** (*tuple*) – default (2, 5), The max of depth is not more than 8.
- **function_set** (*tuple*) – tuple of str. optional: (‘add’, ‘sub’, ‘mul’, ‘div’,”max”, “min”, “ln”, “exp”, “pow2”, “pow3”, “rec”, “sin”, “cos”).
- **n_jobs** (*int*) – n jobs to parallel.
- **verbose** (*bool*) – print message.
- **p_mutate** – mutate probability.
- **p_crossover** (*float*) – crossover probability.
- **random_state** (*int*) – random state

- **hall_of_fame** (*int*) – hall of frame number to add to next generation.
- **store_of_fame** (*int*) – hall of frame number to return result.
- **method_backend** (*str*) – optional: ("p_numpy","c_numpy","p_torch","c_torch")
- **device** (*str*) – default "cpu", "cuda:0", only accessible of torch.
- **func_p** (*np.ndarray,tuple*) – with shape (n_function,), probability values of each function.
- **sci_template** (*str,list*) – None, "default" or user self-defined list template, default None.

filter_sci_perset(*sci_template*)
Get the available sci available

fit(*X: numpy.ndarray*, *y: numpy.ndarray*, *xs_p: numpy.ndarray = None*, *x_label=None*)
Fitting.

Parameters

- **X** (*np.ndarray*) – with shape (n_sample,n_fea).
- **y** (*np.ndarray*) – with shape (n_sample,).
- **xs_p** (*np.ndarray*) – with shape (n_fea,), probability values of each xi.
- **x_label** (*np.ndarray*) – with shape (n_fea), names of xi.

abstract predict(*X*, *y=None*, *n=0*)
Return the real predicted y.

refresh_xcs()
Refresh X and constant for each generation.

refresh_xcs_more()
Refresh X and constant for each generation for torch.

run_gp()
Run the GP processing.

score(*X*, *y*, *scoring*, *n=0*)
Score.

single_cal(*n*, *new_x=None*, *with_coef=True*)
Get the temp predict y of n_ed expression name (without coef and intercept),This is not the final result!

single_name(*n*)
Get the name of n_ed expression name.

class fastgplearn.skflow.**SymbolicRegressor**(*population_size=10000*, *generations=20*, *stopping_criteria=0.95*, *store_of_fame=50*, *hall_of_fame=3*, *store=False*, *p_mutate=0.2*, *p_crossover=0.5*, *select_method='tournament'*, *tournament_size=5*, *constant_range=None*, *constants=None*, *depth=(2, 4)*, *function_set=('add', 'sub', 'mul', 'div', 'pow2', 'pow3')*, *sci_template=None*, *device='cpu'*, *n_jobs=1*, *verbose=0*, *random_state=None*, *method_backend='p_numpy'*, *func_p=None*)

Bases: *fastgplearn.skflow.SymbolicEstimator*

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.

The default score for find expression is R (correlation coefficient), Thus this score needs to be further calculated.

Examples:

```
>>> from fastgplearn.skflow import SymbolicRegressor
>>> est_gp = SymbolicRegressor(population_size=5000,
...                     generations=20, stopping_criteria=0.01,
...                     p_crossover=0.7, p_mutate_=0.1,
...                     max_samples=0.9, verbose=1,
...                     random_state=0)
>>> est_gp.fit(X_train, y_train)
>>> est_gp.top_n()
>>> test_score = est_gp.score(X_test,y_test)
```

Parameters

- **population_size** (*int*) – number of population, default 10000.
- **generations** (*int*) – number of generations, default 20.
- **tournament_size** (*int*) – tournament size for selection.
- **stopping_criteria** (*float*) – criteria of correlation score, max 1.0.
- **constant_range** (*tuple*) – floats. constant_range=(0,1.0)
- **constants** (*tuple*) – floats. constants=(-1,1,2,10), if given, The parameter constant_range would be ignored.
- **depth** (*tuple*) – default (2, 4), The max of depth is not more than 8.
- **function_set** (*tuple*) – tuple of str. optional: ('add', 'sub', 'mul', 'div', "max", "min", "ln", "exp", "pow2", "pow3", "rec", "sin", "cos").
- **n_jobs** (*int*) – n jobs to parallel.
- **verbose** (*bool*) – print message.
- **p_mutate** – mutate probability.
- **p_crossover** (*float*) – crossover probability.
- **random_state** (*int*) – random state
- **hall_of_fame** (*int*) – hall of frame number to add to next generation.
- **store_of_fame** (*int*) – hall of frame number to return result.
- **method_backend** (*str*) – optional: ("p_numpy","c_numpy","p_torch","c_torch")
- **device** (*str*) – default "cpu", "cuda:0", only accessible of torch.
- **func_p** (*np.ndarray*) – with shape (n_function,), probability values of each function.
- **sci_template** (*str,list*) – None, "default" or user self-defined list template, default None.

best_expression(*scoring='r2'*)
Print the best expression.

predict(*X*, *y=None*, *n=0*)
Return the real predicted y.

Parameters

- **X** (*np.ndarray*) – array-like of shape (n_samples, n_features).
- **vectors** (*Input*) –
- **features** (*where n_samples is the number of samples and n_features is the number of*) –
- **y** (*np.ndarray*) – array-like of shape (n_samples,).
- **n** (*int*) – calculate by the n_ed expression.

Returns array-like of shape (n_samples,).

Return type y (np.ndarray)

score(*X*, *y*, *scoring='r2'*, *n=0*)
Return the r2 score (default) on the given test data and labels.

Parameters

- **X** (*np.ndarray*) – array-like of shape (n_samples, n_features).
- **y** (*np.ndarray*) – array-like of shape (n_samples,).
- **scoring** (*str*) – see also sklearn.metrics.
- **n** (*int*) – calculate by the n_ed expression.

Returns Mean r2 of `self.predict(X)` wrt. *y*.

Return type score (float)

static single_coef_linear(*X*, *y*)
Fitting by sklearn.linear_model.LinearRegression.

top_n(*n=0*, *scoring='r2'*)
Print the top n result. The best one is index 0.

Parameters

- **scoring** (*str*) – see also sklearn.metrics.
- **n** (*int*) – calculate by the n_ed expression.

fastgplearn.skflow.**randint**(*low*, *high=None*, *size=None*, *dtype=int*)
Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the "discrete uniform" distribution of the specified dtype in the "half-open" interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

Note: New code should use the `integers` method of a `default_rng()` instance instead; please see the random-quick-start.

Parameters

- **low** (*int or array-like of ints*) – Lowest (signed) integers to be drawn from the distribution (unless `high=None`, in which case this parameter is one above the *highest* such integer).
- **high** (*int or array-like of ints, optional*) – If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if `high=None`). If array-like, must contain integer values
- **size** (*int or tuple of ints, optional*) – Output shape. If the given shape is, e.g., `(m, n, k)`, then `m * n * k` samples are drawn. Default is None, in which case a single value is returned.
- **dtype** (*dtype, optional*) – Desired dtype of the result. Byteorder must be native. The default value is int.

  New in version 1.11.0.

Returns **out** – *size*-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

Return type int or ndarray of ints

See also:

random_integers similar to *randint*, only for the closed interval [*low*, *high*], and 1 is the lowest value if *high* is omitted.

Generator.integers which should be used for new code.

Examples

```
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0]) # random
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1], # random
       [3, 2, 2, 0]])
```

Generate a 1 x 3 array with 3 different upper bounds

```
>>> np.random.randint(1, [3, 5, 10])
array([2, 2, 9]) # random
```

Generate a 1 by 3 array with 3 different lower bounds

```
>>> np.random.randint([1, 5, 7], 10)
array([9, 8, 7]) # random
```

Generate a 2 by 4 array using broadcasting with dtype of uint8

```
>>> np.random.randint([1, 3, 5, 7], [[10], [20]], dtype=np.uint8)
array([[ 8,  6,  9,  7], # random
       [ 1, 16,  9, 12]], dtype=uint8)
```

4.1.6 fastgplearn.tools module

class fastgplearn.tools.**Hall**(*size=10*)

Bases: object

Hall of Frame.

Examples:

```
>>> hall = Hall(size=50)
>>> hall.update(inds, gen_i, score, consts)
>>> hall[i]
```

best_constant()

Return the best individual's constant for next generation.

change0()

Change the unused constants to 0.

get_share_parameter(*x_num*, *single_start*)

sort_and_hash()

Remove the repeat result, (Imperfect guarantee,due to the different individuals could be with same expression).

top_n(*n*)

Return the top n result.

update(*inds*, *gen_i*, *score*, *consts*)

Add individual.

class fastgplearn.tools.**Logs**(*head_msg=''*)

Bases: object

Log the message.

Examples:

```
>>> log = Logs()
>>> log.record("score:0.9")
>>> log.print(log)
>>> "score:0.9"
```

print(*head=False*, *row=True*)

prints(*row=True*)

record(*msg*)

record_and_print(*msg*, *row=False*)

records(*msg*)

fastgplearn.tools.**find_add_mask_all_merge**(*pop*, *single_start=6*)

CHAPTER

FIVE

EXAMPLES

5.1 Regression

Prepare the data

```
>>> from sklearn.datasets import load_boston
>>> x, y = load_boston(return_X_y=True)
```

Fitting

```
>>> from fastgplearn.skflow import SymbolicRegressor
>>> sr = SymbolicRegressor(population_size=1000, generations=10, stopping_criteria=0.95,
>>>                         store=False, p_mutate=0.2, p_crossover=0.5, select_method=
↪"tournament",
>>>                         tournament_size=5, hall_of_fame=3, store_of_fame=50,
>>>                         constant_range=(0, 1.0), constants=None, depth=(2, 5),
>>>                         function_set=('add', 'sub', 'mul', 'div'),
>>>                         n_jobs=1, verbose=True, random_state=0, method_backend='p_
↪numpy', func_p=None,
>>>                         sci_template="default")
>>> sr.fit(x, y)
```

Result

For result, you can specify the number of expression to calculate or score.

```
>>> sr.top_n(n = 10)
>>> res0 = sr.score(x, y, n=0)
>>> pre_y = sr.predict(x, y=None, n=0)
```

5.2 Classification

For binary classification problem.

Prepare the data

```
>>> from sklearn.datasets import load_iris
>>> x, y = load_iris(return_X_y=True)
>>> x=x[y<2] # for binary problem
```

(continues on next page)

(continued from previous page)

```
>>> x[49,:]=4 # just add noise
>>> y=y[y<2]
```

Fitting

```
>>> from fastgplearn.skflow import SymbolicClassifier
>>> sr = SymbolicClassifier(population_size=1000, generations=10, stopping_criteria=0.95,
>>>                         store=False, p_mutate=0.2, p_crossover=0.5, select_method=
↪"tournament",
>>>                         tournament_size=5, hall_of_fame=3, store_of_fame=50,
>>>                         constant_range=(0, 1.0), constants=None, depth=(2, 5),
>>>                         function_set=('add', 'sub', 'mul', 'div'),
>>>                         n_jobs=1, verbose=True, random_state=0, method_backend='p_
↪numpy', func_p=None,
>>>                         sci_template="default")
>>> sr.fit(x, y)
```

Result

For result, you can specify the number of expression to calculate or score.

```
>>> sr.top_n(n = 10)
>>> res0 = sr.score(x, y, n=0)
>>> pre_y = sr.predict(x, y=None, n=0)
```

CHAPTER

SIX

CONTACT

Thanks for your reading.

This project is one alpha version, if you have question, bugs or writing errors to feedback.

Please contact with 986798607@qq.com.

Fast-GP-Learn

FastGPLearn implements Genetic Programming in Python, with a `scikit-learn` inspired and compatible API. And the FastGPLearn applied the `torch` and `numpy` backend for fast calculated, make it accessible for `CUDA`.

CHAPTER

SEVEN

FEATURES:

1. Easy to use CUDA.
2. Bulk calculation and fast.
3. Match with `scikit-learn` and `torch`.

CHAPTER

EIGHT

LINKS

Github | English Version |

CHAPTER

NINE

INDEX

- genindex
- modindex

CHAPTER

TEN

SUPPORT

PYTHON MODULE INDEX

f

INDEX

H

L

M

P

R

S

T

U