BGP

boliqq07

Dec 02, 2021

CONTENTS:

CHAPTER

ONE

INTRODUCTION

symbolic regression

BGP (Binding Genetic Programming)

Binding Genetic Programming is developed on Genetic Programming. This tool is a symbol regression tool with dimension calculation, which is aimed to establish expressions with physical limitation. This tool built formulas, or called expression, using genetic algorithm, by combined the `blocks`, which is including `operators` (+ - * / .etc) , `features` (x_1, x_2 .etc) and `numerical term` (1,0.5,0.16).

Dimension calculation and **Artificial binding** are embedded in this tool. These added modules are aimed to reduce the invalid search space, especially in specific physical domain knowledge.

The numerical terms (coefficient,and intercept) in expressions are added by **coefficient fitting**. which is different to most of Genetic Programming packages. We offered different methods to site the numerical terms, to control the expressions to generate the best suitable one near the expected outcome.

Some helpful code can be copied from others package and adapt to new environment. Infringement contents would be removed.

CHAPTER

TWO

INSTALL

2.1 Requirements

Packages:

Dependence	Name	Version
necessary	sympy	>=1.6
necessary	deap	>=1.3.1
necessary	scikit-learn	>=0.22.1
recommend	torch	>=1.5.0
recommend	pymatgen	
recommend	scikit-image	
recommend	minepy	

2.2 Method 1

Install with pip

```
pip install BindingGP
```

note If VC++ needed for windows wheel, such as spblib , Please download the dependent packages named (spglib-1.*-cp3*-*.whl) from Python Extension Packages and install offline. please reference to Method 2.

2.3 Method 2

Install by step:

1. sympy

```
pip install sympy>=1.6
```

Reference: https://www.sympy.org/en/index.html

2. deap

```
pip install deap
```

Reference: https://github.com/DEAP/deap

3. pymatgen

```
conda install /your/local/path/spglib-1.*-cp3*-*.whl
pip install pymatgen
```

Reference: https://github.com/materialsproject/pymatgen, https://github.com/spglib/spglib/tree/develop/python

3. pymatgen(options)

```
conda install --channel conda-forge pymatgen
```

Reference: https://github.com/materialsproject/pymatgen

4. scikit-learn

```
conda install sklearn
```

Reference: https://github.com/materialsproject/pymatgen

5. mgetool:

```
pip install mgetool
```

Reference: https://github.com/Mgedata/mgetool

6. BGP:

```
pip install BindingGP
```

CHAPTER

THREE

QUICK START

The symbol are conformed with the ‘’sklearn-style” type, which can be easily to modeling with `fit`, `predict`, `score`.

```
if __name__ == "__main__":
    # data
    from sklearn.datasets import load_boston
    from bgp.skflow import SymbolLearning

    data = load_boston()
    x = data["data"]
    y = data["target"]
    c = [6, 3, 4]

    # start->
    sl = SymbolLearning(loop="MultiMutateLoop", pop=500, gen=2, random_state=1)
    sl.fit(x, y, c=c)
    score = sl.score(x, y, "r2")
    print(sl.expr)
```

And return the results:

```
>>>62.33 - 2.156*x10
```

Note **When the result of one problem is not stable, the final expression is changed with random_state (random seed). The random seeds between window and linux are different.**

More Examples:

skflow

Examples

CHAPTER

FOUR

GUIDE

4.1 skflow

Contains:

- Class: *bgp.skflow.SymbolLearning*

  One "sklearn-type" implement to run symbol learning. We recommend this approach when rapid modeling. The SymbolLearning could implement most of the functions and without other assistance functions.

For example, the data can be import from sklearn.

```
if __name__ == "__main__":
    from sklearn.datasets import load_boston
    from bgp.skflow import SymbolLearning

    data = load_boston()
    x = data["data"]
    y = data["target"]
    c = [1, 2, 3]
```

Import `SymbolLearning` and add the parameter (such as, with 500 population each generation, with 3 generations, calculate the dimensions(units) of expressions, with 2 elites feedback, add coefficient in expression, with random state = 1).

```
from bgp.skflow import SymbolLearning
sl = SymbolLearning(loop="MultiMutateLoop", pop=500, gen=3, cal_dim=True,
                        re_hall=2, add_coef=True, random_state=1
                        )
```

Fitting data and add the binding with `x_group`.

```
sl.fit(x, y, c=c,x_group=[[1, 3], [0, 2], [4, 7]]))
score = sl.score(x, y, "r2")
print(sl.expr)
```

The detail of `x_group` can be found in *Remarks*.

The `SymbolLearning` could implement most of the functions and without other assistance functions.

Except

- user-defined new operations
- user-defined probability of operation occurrence

- user-defined probability of features mutual influence

For these realizations, we could customer the pset (base.SymbolSet) in advance and pass to "pset" parameters. For in-depth customization, please refer to `base` part and `flow` part.

More Examples:

Examples

Parameters and **Methods** can be found in *bgp.skflow.SymbolLearning*.

Attributes

loop: str the running loop in flow part.

best_one: SymbolTree the best one of expressions.

expr: sympy.Expr the best one of expressions.

y_dim: Dim dim of calculate y.

fitness: float score

The call relationship(correspondence) is as follows:

`flow.loop` –> `skflow.SymbolLearning`

`base.pset.add_features_and_constants` –> `skflow.SymbolLearning.fit`

`base.pset.add_operations` –> `skflow.SymbolLearning.fit`

4.2 Remarks

This part is not a module but some notes about key parameters and core problems. The examples in *Examples*, *Complexity Control*

Contains:

- Binding: `x_group` in *bgp.skflow.SymbolLearning.fit()* or *bgp.base.SymbolSet.add_features()*.
- Dim: *bgp.functions.dimfunc.Dim*
- Dimension: `cal_dim`, `dim_type` in *bgp.skflow.SymbolLearning* or or `BaseLoop`, `x_dim`, `y_dim`, `c_dim` in *bgp.skflow.SymbolLearning.fit()* or `BaseLoop`.
- Coefficients: `add_coef`, `inner_add`, `inter_add`, `out_add`, `flat_add`, `vector_add` in *bgp.skflow.SymbolLearning* or `BaseLoop`.

4.2.1 Binding

Assume there is (x_1,x_2,x_3,...,x_n) features in data. if you want to make x_1,x_2 banded:

```
x_group = [[1,2],]
```

if you want make each banded with size, such as (x_1,x_2),(x_3,x_4),...,(x_{n-1},x_n):

```
x_group = 2
```

The group size should be more than 2.

4.2.2 Dim

The `Dim` is *bgp.functions.dimfunc.Dim*.

The default dimension SI system with 7 number.

The basic unit are:

{'meter': "m", 'kilogram': "kg", 'second': "s", 'ampere': "A", 'mole': "mol", 'candela': "cd", 'kelvin': "K"}

The basic unit can be represented by: ['length', 'mass', 'time', 'current', 'amount_of_substance', 'luminous_intensity', 'temperature']

1.can be constructed by list of number.

2.can be translated from a sympy.physics.unit.

Examples:

```
from sympy.physics.units import Kg
scale,dim = Dim.convert_to_Dim(Kg)
```

Examples:

```
dim=[1,0,1,0,1,0,0]
dim = Dim(dim)
```

Supplementary Note 1: Dimensional calculation, when the data should be units, but you do not use dimensional calculation subjectively, in this case, the model can be performed, that is, the data is assumed to be completely dimensional-uniformed data.

Supplementary Note 2: Dimensional calculation, from different units to standard units for calculation, which must have shrinkage coefficient.

1. We suggest that the data be processed into the data value corresponding to the standard unit (dimensional reference), that is, the scaling coefficient is multiplied into the data.
2. We have also integrated unit to dimensional transformation tools from sympy units.

(Dim.convert_x, dim.convert_XI, dim.convert_x are used to convert x, y, and C respectively and the converted data and dimensions are obtained)

In either case, the formula for the final result is correct, but the coefficient value is not your initial data (but the data corresponding to the standard units).You may need to manually re-fit the coefficient values.

Supplementary Note 3: Dimension calculation, some units are too small or too large, the data will be very small or very large when multiplied by the scaling factor, such as 10^16.

If data is pre-processed using the `MagnitudeTransformer` provided by us, in this case you also need to manually re-fit the coefficient value.

Supplementary Note 4: The unit of coefficient, we do not provide the unit of coefficient value, the default unit of coefficient is calculated by means of completion.

Supplementary Note 5: The calculation roles for dimension can be seen in Developer Manual.pdf,

4.2.3 Dimension:

For *bgp.skflow.SymbolLearning*.

The `cal_dim` is only valid when `x_dim, y_dim` are given. When it is True, the dimension of result expression would be checked with `dim_type`.

In default, the `dim_type` is "coef",

Without coefficient: that is `dim_type``=``y_dim` ,

With coefficient: assume expression is y=af(x)+b, a,b have dimension, f(x)'s dimension is not nan .

Of course, we can tighten the restriction, such as make the `dim_type` = `y_dim` make the expression must have dimension.

The more strict from top to bottom:

Parameters:

"coef" af(x)+b. a,b have dimension, f(x)'s dimension is not dnan.

"integer" af(x)+b. f(x) is with integer dimension.

[Dim1,Dim2] f(x)'s dimension in list.

Dim f(x) ~= Dim. (see fuzzy)

Dim f(x) == Dim.

None f(x) == pset.y_dim

4.2.4 Coefficients:

`add_coef, inner_add, out_add, flat_add, vector_add, inter_add` in *bgp.skflow.SymbolLearning* or `BaseLoop`.

`add_coef` is 'main switch' or others.

Assume the initial expression is y=f(x)

add_coef: The main switch of coefficients. default: Add the coefficients of expression. such as y=cf(x).

inter_add: Add the intercept of expression. such as y=f(x)+b.

out_add: Add the coefficients of expression. such as y=a(x), but for polynomial join by + and -,the coefficient would add before each term. such as y=af1(x)+bf2(x).

flat_add: flatten the expression and add the coefficients out of expression. such as y=af`1(x)+bf`2(x)+ef`3(x), (the old expression: y = x*(f1(x)+f2(x)+f3(x))).

inner_add: Add the coefficients inner of expression. such as y=cf(ax).

vector_add: only valid when x_group is True, add different coefficients on group x pair.

For the `inner_add, inter_add, out_add, flat_add`, just only one can be selected.

4.3 base

Base objects for symbolic regression.

Contains:

- Class: *bgp.base.SymbolSet*
- Class: *bgp.base.CalculatePrecisionSet*
- Class: *bgp.base.SymbolTree*
- others

4.3.1 SymbolSet

For example, the data can be imported from sklearn.

```
if __name__ == "__main__":
    from sklearn.datasets import load_boston

    data = load_boston()
    x = data["data"]
    y = data["target"]
    c = [1, 2, 3]
```

The `SymbolSet` is a presentation set contains some 'blocks', which are including features (x_1, x_2 .etc) operators (+ - * / .etc) , and numerical term (2, 3, 0.5). which can be added by `add_features`, `add_operations`, `add_constants` respectively.

The detail of `add_features`,``add_operations`` can be found in *Remarks*.

```
from bgp.base import SymbolSet
pset0 = SymbolSet()
pset0.add_features(x, y)
pset0.add_constants(c, )
pset0.add_operations(power_categories=(2, 3, 0.5),
         categories=("Add", "Mul", "exp"),
         special_prob =  {"Mul": 0.5,"Add": 0.4,"exp": 0.1}
         power_categories_prob = "balance")
```

Then the mode can be built with `skflow.SymbolLearning`, just replace the fit parameters: 'pset'.

```
from bgp.skflow import SymbolLearning
sl = SymbolLearning(loop="MultiMutateLoop", pop=500, gen=3,
                    cal_dim=True, re_hall=2, add_coef=True, cv=1,
                    random_state=1
                    )
sl.fit(pset=pset0)
score = sl.score(x, y, "r2")
print(sl.expr)
```

4.3.2 SymbolTree

Individual Tree, each tree is one expression.

Generate expressions from pset.

```
pset = SymbolSet()

individual = SymbolTree.genGrow(pset, height , height+1,)

population = [SymbolTree.genFull(pset, height , height+1,) for _ in range(5000)]
```

4.3.3 CalculatePrecisionSet

Definite the operations, features, and fixed constants. One calculation ability extension for SymbolSet. For example:

```
cp = CalculatePrecisionSet(pset, scoring=[r2_score, ],score_pen=[1, ], filter_
↪warning=True)
```

The cp could calculate the individual by:

```
result = cp.calculate_detail(individual)
```

or calculate population:

```
result = cp.parallelize_score(population)
```

4.4 flow

Some definitions loop for genetic algorithm.

Contains:

- Class: *bgp.flow.BaseLoop*

  one node mate and one tree mutate.

- Class: *bgp.flow.MultiMutateLoop*

  one node mate and (one tree mutate, one node Replacement mutate, shrink mutate, difference mutate).

- Class: *bgp.flow.OnePointMutateLoop*

  one node Replacement mutate: (keep height of tree)

- Class: *bgp.flow.DimForceLoop*

  Select with dimension : (keep dimension of tree)

```
if __name__ == "__main__":
    pset = SymbolSet()
    stop = lambda ind: ind.fitness.values[0] >= 0.880963
    bl = OnePointMutateLoop(pset=pset, gen=10, pop=1000, hall=1, batch_size=40, re_
↪hall=3, \n
                      n_jobs=12, mate_prob=0.9, max_value=5, initial_min=1, initial_max=2,␣
↪\n
```

(continues on next page)

(continued from previous page)

```
                    mutate_prob=0.8, tq=True, dim_type="coef", stop_condition=stop,\n
                    re_Tree=0, store=False, random_state=1, verbose=True,\n
                    stats={"fitness_dim_max": ["max"], "dim_is_target": ["sum"]},\n
                    add_coef=True, inter_add=True, inner_add=False, cal_dim=True, vector_
↪add=False,\n
                    personal_map=False)
    bl.run()
```

The **Parameters**, **Methods**, and **Attributes** for all loops are same.

- Parameters

  The Parameters is the same with `skflow.SymbolLearning`, except the 'loop' parameter in `skflow.SymbolLearning`.

- Methods

run: run the loop.

The `flow.BaseLoop.run` is the base of `skflow.SymbolicLearning.fit`

Suggested Using

`skflow`: One sklearn-type implement to run symbol learning

Advanced Using

`base`: The storage form of expression (tree-style), integration calculate methods, and define the present set for features and operations

`flow`: Genetic algorithm loop

For developers

`preprocess`: One "sklearn-type" implement to transform X

`gp`: Genetic algorithm method

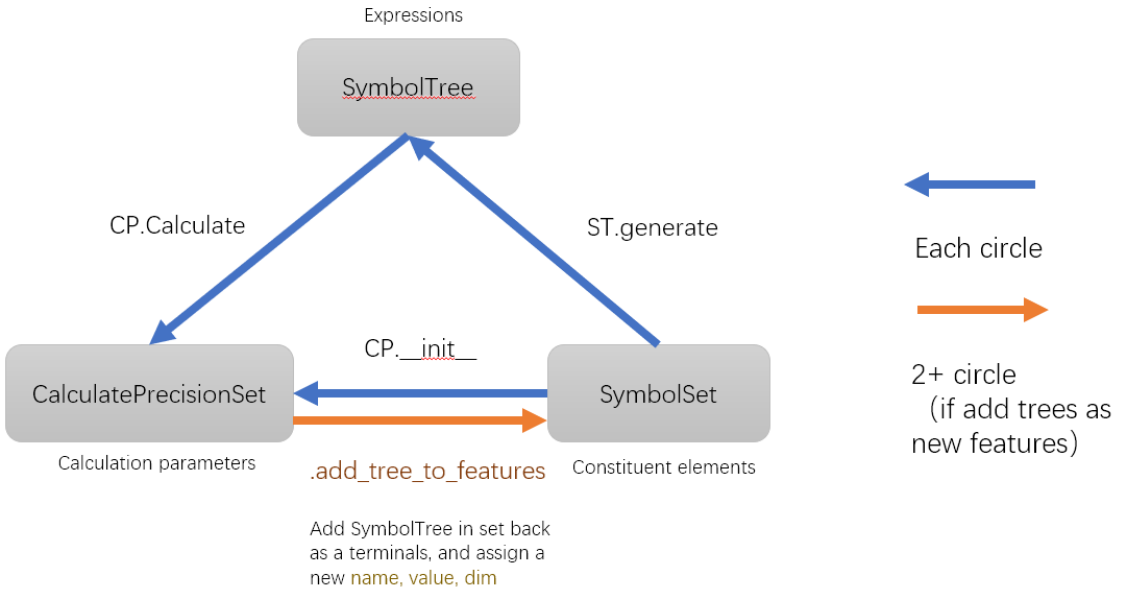
`function`: Operation roles, Vector Operation, Dimension Operation, To define Operation

`calculation`: Expression tree translation, Coefficient addition, Score

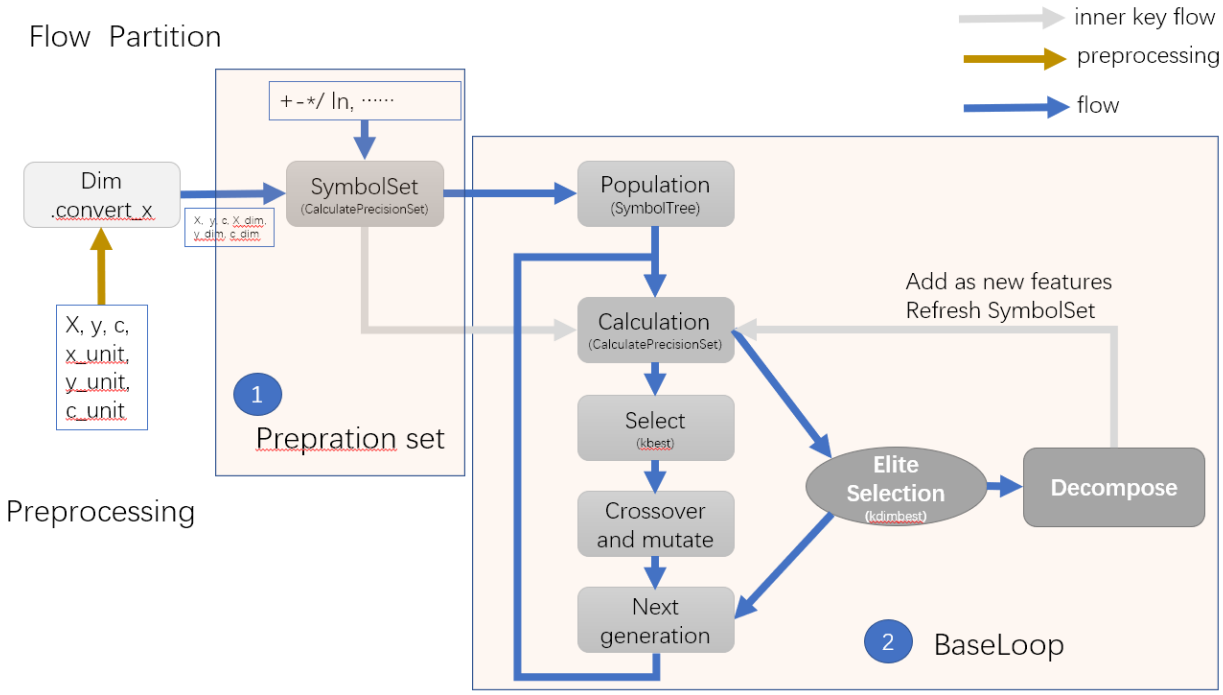
This API is aimed for user but not developer. Therefore some modules are just with brief introduction. If you want to know more math principles and code structure,

Turn to Developer Manual.pdf.

where you can see:



Flow relationship between 3 base objects



Dimension calculation

Example:

a: d_a : dimension 1
b: d_b : dimension 2
c1: d_1 : dimensionless
d: d_{nan} : without dimension

Operation	Dimension calculation	Result
+	$d_a + d_a$	d_a
	$d_a + d_b$	d_{nan}
	$d_a + d_1$	d_a
	$d_1 + d_1$	d_1
-	the same with +	
*	$d_a * d_a$	$2 * d_a$
	$d_a * d_b$	$d_a + d_b$
	$d_a * d_1$	d_a
	$d_1 * d_1$	d_1
/	d_a / d_a	d_1
	d_a / d_b	$d_a - d_b$
	d_a / d_1	d_a
	d_1 / d_1	d_1

Operation	Dimension calculation	Result
$\sum$ madd	the same with +, could accept 1 input and return it	
- re-name sub to msub	the same with -, could accept 1 input and return it invalid if accept one 's shape more than 2.	
$\prod$ mmul	the same with *, could accept 1 input and return it	
/ re-name div to mdiv	the same with /, could accept 1 input and return it invalid if accept one 's shape more than 2.	

Operation	Dimension calculation	Result
-x (negative particular case -)	$-d_a$	d_a
	$-d_1$	d_1
1/x (negative particular case /)	$1/d_a$	$-d_a$
	$1/d_1$	d_1
ln exp sin cos	$f(d_a)$	d_{nan}
	$f(d_1)$	d_1
x^n	d_a^n	$n * d_a$
	d_1^n	d_1
n^x	n^{d_a}	d_{nan}
n^x	n^{d_1}	d_{nan}
abs	abs(d)	d
self	$self(d)$	d

$$d_1 = function(d_1) \quad for\ any\ function$$
$$g(d_{other}) = function(d_1, d_{other}) \quad for\ any\ function$$

$$d_{nan} = function(d_{nan}) \quad for\ any\ function$$
$$d_{nan} = function(d_{nan}, d_{other}) \quad for\ any\ function$$

CHAPTER

FIVE

BGP

5.1 bgp package

5.1.1 Subpackages

bgp.calculation package

Submodules

bgp.calculation.coefficient module

vector coef and vector const, which is a UndefinedFunction to excape the auto calculation of numpy to sympy.

class bgp.calculation.coefficient.**CheckCoef**(*cof_list*, *cof_dict*)
Bases: object

group the coef and pack the calculate part out of loop.

Parameters

- **cof_list** (*Sized*) –
- **cof_dict** (*dict*) –

dec(*ls*)

group(*p*, *decimals=False*)
change the p to grpup

property ind

class bgp.calculation.coefficient.**Coef**(*name*, *arr*)
Bases: sympy.core.function.UndefinedFunction

generate metaclass, the type of identity is .arr,.tp ,rather isinstance.

class bgp.calculation.coefficient.**Const**(*name*, *arr*)
Bases: sympy.core.function.UndefinedFunction

generate metaclass, the type of identity is .arr,.tp ,rather isinstance

bgp.calculation.coefficient.**add_coefficient**(*expr01*, *inter_add=True*, *inner_add=False*, *vector_add=False*, *out_add=False*, *flat_add=False*)
Try add the placeholder coefficient to sympy expression. 1. Add Wi,A,B normal coefficients to expression. 2. Add V, Vi vector coefficients to expression, for this type of coefficient , there should be with expr01.conu for Function("MAdd"), Function("MSub"). more details can be found in ..translate.simple

Parameters

- **expr01** (*Expr*) – sympy expressions
- **inter_add** (*bool*) – bool
- **inner_add** (*bool*) – bool
- **vector_add** (*bool*) – bool
- **flat_add** (*bool*) – bool
- **out_add** (*bool*) – bool

Returns

Return type expr

bgp.calculation.coefficient.**cla**(*pre_y*, *cl=True*)

bgp.calculation.coefficient.**find_args**(*expr_*, *patten*)
find the term of patten, judge by hash rather type

bgp.calculation.coefficient.**flatten_add_f**(*expr01*, *cof_list*, *cof_dict*, *vector_add*)

bgp.calculation.coefficient.**get_args**(*expr*, *sole=True*)

Parameters

- **expr** (*sympy.Expr*) –
- **sole** (*only find unique term*) –

Returns

Return type list

bgp.calculation.coefficient.**inner_add_f**(*expr01*, *cof_list*, *cof_dict*, *vector_add*)

bgp.calculation.coefficient.**out_add_f**(*expr01*, *cof_list*, *cof_dict*, *vector_add*)

bgp.calculation.coefficient.**replace_args**(*expr_*, *old*, *new*)
find the term of patten, judge by hash rather type

bgp.calculation.coefficient.**replace_args_first**(*expr_*, *old*, *new*)
a

bgp.calculation.coefficient.**try_add_coef**(*expr01*, *x*, *y*, *terminals*, *grid_x=None*, *filter_warning=True*, *inter_add=True*, *inner_add=False*, *vector_add=False*, *out_add=False*, *flat_add=False*, *np_maps=None*, *classification=False*)
Try calculate predict y by sympy expression with coefficients. if except error return expr itself.

Parameters

- **flat_add** (*bool*) – add flat coefficient or not
- **out_add** – add outcoefficientt or not
- **vector_add** (*bool*) – add vectorcoefficientt or not
- **expr01** (*sympy.Expr*) – sympy expressions
- **x** (*list of np.ndarray*) – list of xi
- **y** (*np.ndarray*) – y value
- **grid_x** – new x to predict

- **terminals** (*list of sympy.Symbol*) – features and constants
- **filter_warning** (*bool*) – bool
- **inter_add** (*bool*) – bool
- **inner_add** (*bool*) – bool
- **np_maps** (*Callable*) – user np.ndarray function

Returns

- *pre_y* – np.array or None
- **expr01** (*Expr*) – New expr.

bgp.calculation.coefficient.**try_add_coef_times**(*expr01*, *x*, *y*, *terminals*, *grid_x=None*, *filter_warning=True*, *inter_add=True*, *inner_add=False*, *vector_add=False*, *out_add=False*, *flat_add=False*, *np_maps=None*, *classification=False*, *random_state=0*, *return_expr=False*, *resample_number=500*)

bgp.calculation.scores module

Notes

score method.

bgp.calculation.scores.**calcualte_dim**(*expr01*, *terminals*, *dim_list*, *dim_maps=None*)

Parameters

- **expr01** (*Expr*) – sympy expression.
- **terminals** (*list of sympy.Symbol*) – features and constants
- **dim_list** (*list of Dim*) – dims of features and constants
- **dim_maps** (*Callable*) – user dim_maps

Returns

- *Dim* – dimension
- *dim_score* – is target dim or not

bgp.calculation.scores.**calcualte_dim_score**(*expr01*, *terminals*, *dim_list*, *dim_type*, *fuzzy*, *dim_maps=None*)

Parameters

- **expr01** (*Expr*) – sympy expression.
- **terminals** (*list of sympy.Symbol*) – features and constants
- **dim_list** (*list of Dim*) – dims of features and constants
- **dim_maps** (*Callable*) – user dim_maps
- **dim_type** (*list of Dim*) – target dim
- **fuzzy** – fuzzy dim or not

Returns

- *Dim* – dimension
- *dim_score* – is target dim or not

bgp.calculation.scores.**calculate_collect_**(*ind*, *context*, *x*, *y*, *terminals_and_constants_repr*, *gro_ter_con*, *dim_ter_con_list*, *dim_type*, *fuzzy*, *cv=1*, *refit=True*, *scoring=None*, *score_pen=(1,)*, *add_coef=True*, *filter_warning=True*, *inter_add=True*, *inner_add=False*, *vector_add=False*, *out_add=False*, *flat_add=False*, *np_maps=None*, *classification=False*, *dim_maps=None*, *cal_dim=True*, *score_object='y'*, *details=False*)

bgp.calculation.scores.**calculate_cv_score**(*expr01*, *x*, *y*, *terminals*, *scoring=None*, *score_pen=(1,)*, *cv=5*, *refit=True*, *add_coef=True*, *filter_warning=True*, *inter_add=True*, *inner_add=False*, *vector_add=False*, *out_add=False*, *flat_add=False*, *np_maps=None*, *classification=False*, *score_object='y'*, *details=False*)

Use cv spilt for score, return the mean_test_score. Use cv spilt for predict, return the cv_predict_y.(have not be used)

Notes

if cv and refit, all the data is refit to determine the coefficients. Thus the expression is not compact with the this scores, when re-calculated by this expression.

Parameters

- **score_object** – score by y or delta y
- **classification** – classification or not
- **refit** (*True:*) – use forced, refit the coefficient use all data.
- **cv** (*sklearn.model_selection._split._BaseKFold,int*) – the shuffler must be False
- **vector_add** –
- **expr01** (*Expr*) – sympy expression.
- **x** (*list of np.ndarray*) – list of xi
- **y** (*np.ndarray*) – y value
- **terminals** (*list of sympy.Symbol*) – features and constants
- **scoring** (*list of Callbale, default is [sklearn.metrics.r2_score,]*) – See Also sklearn.metrics
- **score_pen** (*tuple of 1 or -1*) – 1 : best is positive, worse -np.inf -1 : best is negative, worse np.inf 0 : best is positive , worse 0
- **add_coef** (*bool*) – bool
- **filter_warning** (*bool*) – bool
- **inter_add** (*bool*) – bool
- **inner_add** (*bool*) – bool
- **flat_add** (*bool*) – bool
- **out_add** (*bool*) – bool

- **np_maps** (*Callable*) – user np.ndarray function

Returns

- **score** (*float*) – score
- **expr01** (*Expr*) – New expr.
- **pre_y** (*np.ndarray or float*) – np.array or None

bgp.calculation.scores.**calculate_derivative_y**(*expr01*, *x*, *terminals*, *np_maps=None*)

Something error for reference:

M. Schmidt, H. Lipson, Distilling free-form natural laws from experimental data, Science, 324 (2009), 81–85.

Parameters

- **expr01** (*Expr*) – sympy expression.
- **x** (*list of np.ndarray*) – list of xi
- **terminals** (*list of sympy.Symbol*) – features and constants
- **np_maps** (*Callable*) – user np.ndarray function

Returns

- **pre_dy_all** (*np.ndarray or float*) – pre-dy
- **dy_all** (*np.ndarray or float*) – dy

bgp.calculation.scores.**calculate_score**(*expr01*, *x*, *y*, *terminals*, *scoring=None*, *score_pen=(1,)*, *add_coef=True*, *filter_warning=True*, *inter_add=True*, *inner_add=False*, *vector_add=False*, *out_add=False*, *flat_add=False*, *np_maps=None*, *classification=False*, *score_object='y'*, *details=False*)

Parameters

- **vector_add** –
- **expr01** (*Expr*) – sympy expression.
- **x** (*list of np.ndarray*) – list of xi
- **y** (*np.ndarray*) – y value
- **terminals** (*list of sympy.Symbol*) – features and constants
- **scoring** (*list of Callbale, default is [sklearn.metrics.r2_score,]*) – See Also sklearn.metrics
- **score_pen** (*tuple of 1 or -1*) – 1 : best is positive, worse -np.inf -1 : best is negative, worse np.inf 0 : best is positive , worse 0
- **add_coef** (*bool*) – bool
- **filter_warning** (*bool*) – bool
- **inter_add** (*bool*) – bool
- **inner_add** (*bool*) – bool
- **np_maps** (*Callable*) – user np.ndarray function

Returns

- **score** (*float*) – score

- **expr01** (*Expr*) – New expr.
- **pre_y** (*np.ndarray or float*) – np.array or None

bgp.calculation.scores.**calculate_y**(*expr01*, *x*, *y*, *terminals*, *add_coef=True*, *x_test=None*, *y_test=None*, *filter_warning=True*, *inter_add=True*, *inner_add=False*, *vector_add=False*, *out_add=False*, *flat_add=False*, *np_maps=None*, *classification=False*)

bgp.calculation.scores.**calculate_y_unpack**(*expr01*, *x*, *terminals*, *classification=False*)

bgp.calculation.scores.**score_dim**(*dim_*, *dim_type*, *fuzzy=False*)

bgp.calculation.scores.**uniform_score**(*score_pen=1*)
return the worse score

bgp.calculation.translate module

bgp.calculation.translate.**compile_**(*expr*, *pset*)
Compile the expression *expr*.

Parameters

- **expr** – Expression to compile. It can either be a PrimitiveTree, a string of Python code or any object that when converted into string produced a valid Python code expression.
- **pset** – Primitive set against which the expression is compile.

Returns a function if the primitive set has 1 or more arguments, or return the results produced by evaluating the tree.

bgp.calculation.translate.**compile_context**(*expr*, *context*, *gro_ter_con*, *simplify=True*)
Compile the expression *expr*.

Parameters

- **expr** – Expression to compile. It can either be a PrimitiveTree, a string of Python code or any object that when converted into string produced a valid Python code expression.
- **context** – dict
- **simplify** – bool
- **gro_ter_con** – list if group_size

Returns a function if the primitive set has 1 or more arguments, or return the results produced by evaluating the tree.

bgp.calculation.translate.**general_expr**(*self*, *pset*, *simplifying=False*)

Parameters

- **simplifying** (*bool*) –
- **self** (*sympy.Expr*) –
- **pset** (SymbolSet) –

bgp.calculation.translate.**general_expr_dict**(*self*, *expr_init_map*, *free_symbol*, *gsym_map*, *simplifying=False*)
gen expr

bgp.calculation.translate.**group_str**(*self*, *pset*, *feature_name=False*)
return expr just build by input feature name.

Parameters

- **self** (*sympy.Expr or* SymbolTree) –
- **pset** (SymbolSet) –
- **feature_name** (*Bool*) –

bgp.calculation.translate.**simple**(*expr01*, *groups*)
str to sympy.Expr function. add conv to MMdd and MMul. the calcualte conv need conform with np_func()!!

is_jump: jump the calculate >= 3 (group_size). keep: the calculate is return then input group_size or 1.

Module contents

bgp.functions package

Submodules

bgp.functions.dimfunc module

Notes

These are some of parts coped from sympy.

class bgp.functions.dimfunc.**Dim**(*data*)
Bases: numpy.ndarray

Redefine the Dimension of sympy, the default dimension SI system with 7 number.

1.can be constructed by list of number.

2.can be translated from a sympy.physics.unit.

Examples:

```
from sympy.physics.units import N
scale,dim = Dim.convert_to_Dim(N)
```

Examples:

```
dim=[1,0,1,0,1,0,0]
dim = Dim(dim)
```

Notes

self.unit = [str(i) for i in SI._base_units]

self.unit_map = {'meter': "m", 'kilogram': "kg", 'second': "s", 'ampere': "A", 'mole': "mol", 'candela': "cd", 'kelvin': "K"}

self.dim = ['length', 'mass', 'time', 'current', 'amount_of_substance', 'luminous_intensity', 'temperature']

allisnan()

anyisnan()

classmethod convert_to(*expr*, *target_units=None*, *unit_system='SI'*)
depend on sympy 1.5-1.6!!!

classmethod convert_to_Dim(*u*, *target_units=None*, *unit_system='SI'*)
depend on sympy 1.5-1.6!!!

Parameters

- **u** (*sympy.physics.unit, Expr of sympy.physics.unit*) – unit.
- **target_units** (*None, list of sympy.physics.unit*) – if None, the target_units is 7 SI units
- **unit_system** (*str*) – default is unit_system="SI"

classmethod convert_x(*x*, *u*, *target_units=None*, *unit_system='SI'*)
depend on sympy 1.5-1.6!!! Quick method. translate x and u to standard system.

Parameters

- **x** (*np.ndarray or list of ndarray,list of float,list of int*) – x
- **u** (*list of sympy.physics.unit or Expr of sympy.physics.unit*) – units
- **target_units** (*None or list of sympy.physics.unit*) – if None, the target_units is 7 SI units
- **unit_system** (*str*) – default is unit_system="SI"

Returns

- **x** (*np.ndarray*)
- **expr** (*Expr*)

classmethod convert_xi(*xi*, *ui*, *target_units=None*, *unit_system='SI'*)
depend on sympy 1.5-1.6!!! Quick method. translate xi and ui to standard system.

Parameters

- **xi** (*np.ndarray*) – xi
- **ui** (*sympy.physics.unit or Expr of sympy.physics.unit*) – unit
- **target_units** (*None or list of sympy.physics.unit*) – if None, the target_units is 7 SI units
- **unit_system** (*str*) – default is unit_system="SI"

Returns

- **xi** (*np.ndarray*)
- **expr** (*Expr*)

get_n(*others*)

classmethod inverse_convert(*dim*, *scale=1*, *target_units=None*, *unit_system='SI'*)
depend on sympy 1.5-1.6!!! Quick method. Translate ui to other unit.

Parameters

- **dim** (Dim) – dim
- **scale** (*float*) – scale generated before.
- **target_units** (*None or list of sympy.physics.unit*) – if None, the target_units is 7 SI units

- **unit_system** (*str*) – default is unit_system="SI"

Returns

- **scale** (*float*)
- **expr** (*Expr*)

classmethod inverse_convert_xi(*xi*, *dim*, *scale=1*, *target_units=None*, *unit_system='SI'*)
depend on sympy 1.5-1.6!!! Quick method. Translate xi, dim to other unit.

Parameters

- **xi** (*np.ndarray*) – xi
- **dim** (Dim) – dim
- **scale** (*float*) – if xi is have been scaled, the scale is 1.
- **target_units** (*None or list of sympy.physics.unit*) – if None, the target_units is 7 SI units
- **unit_system** (*str*) – default is unit_system="SI"

Returns

- **scale** (*float*)
- **expr** (*Expr*)

is_same_base(*others*)

isfloat()

isinteger()

bgp.functions.dimfunc.**check_dimension**(*x*, *y=None*)
check the consistency of dimension.

Parameters

- **x** (*container*) – dim of x
- **y** (Dim) – dim of y

Returns

Return type bool

bgp.functions.dimfunc.**dim_map**()
expr to dim function

bgp.functions.gsymfunc module

class bgp.functions.gsymfunc.**NewArray**(*iterable*, *shape=None*, ***kwargs*)
Bases: sympy.tensor.array.dense_ndim_array.ImmutableDenseNDimArray

as_coefficient(_)

default_assumptions = {}

bgp.functions.gsymfunc.**gsym_map**()
user's sympy.expr to np.ndarray function

bgp.functions.newfunc module

bgp.functions.newfunc.**check_funcD**(*funcs*, *self_grpup=2*)
self_group>=2

bgp.functions.newfunc.**newfuncD**(*operation*, *name='Fc'*, *keep=True*, *is_jump=False*, *check=True*)

Parameters

- **operation** (*callable*) – the detail of opearation only accept +,-,*,/,-(negative),x^n
- **name** (*str*) – name
- **keep** (*bool*) – the group size after this function. true is the input size,and false is 1.
- **is_jump** (*bool*) – the bool means the rem and rem_dim can be treat 2+ domension problems or not.
- **check** (*bool*) – check the function building

bgp.functions.newfunc.**newfuncV**(*operation*, *arity=1*, *name='Fc'*)

Parameters

- **operation** (*callable*) – the detail of operation only accept +,-,*,/,abs,-(negative),x^n.
- **arity** (*int*) – the arity of operation.
- **name** (*str*) – name.

bgp.functions.npfunc module

bgp.functions.npfunc.**np_map**()
user's sympy.expr to np.ndarray function

bgp.functions.symfunc module

bgp.functions.symfunc.**sym_dispose_map**()
user's str to sympy.expr function

bgp.functions.symfunc.**sym_vector_map**()
str to sympy.Expr function

Module contents

Notes: the translation process the three function should be the same key. 1. sym_vector_map: repr of SymbolTree to sympy.Function sym_dispose_map: repr of SymbolTree to "group" sympy.Function. 2. np_map(): repr of "group" sympy.Function to numpy function 3. dim_map(): repr of "group" sympy.Function to Dim function 4. gsym_map():repr of "group" sympy.Function to universal sympy.Function

bgp.iteration package

Submodules

bgp.iteration.newpoints module

find new point from searchspace to add to loop.

bgp.iteration.newpoints.**CalculateEi**(*y*, *meanstd0*)

bgp.iteration.newpoints.**get_max_diff**(*grid_x*, *curves*, *n=1*)

bgp.iteration.newpoints.**get_max_std**(*grid_x*, *curves*, *n=1*)

bgp.iteration.newpoints.**meanandstd**(*predict_dataj*)

bgp.iteration.newpoints.**new_points**(*loop*, *grid_x*, *method='get_max_std'*, *resample_number=500*, *n=1*)

bgp.iteration.newpoints.**search_space**(**arg*)

bgp.iteration.nonewpoints module

to be continued

Module contents

bgp.probability package

Submodules

bgp.probability.preference module

class bgp.probability.preference.**PreMap**(*data*)

Bases: numpy.ndarray

2D probability map

add_new()

add new features to self

down_other_point(**sv*)

Use for binding.rate the others and add the subbed value to the [index1,index2] the rate are [0,1).

Parameters **sv** (*[index1,index2,rate]*) – site to set value

classmethod **from_shape**(*shape*)

Generation.

Parameters **shape** (*int*) – shape of premap.

Returns

Return type *PreMap*

get_ind_value(*ind*, *pset*)

get the value according to ind

Parameters

- **ind** (SymbolTree) –
- **pset** (SymbolSet) –

Returns

Return type probability list

get_indexes_value(*indexes*, *weight=None*)

Parameters

- **indexes** (*tuple, indexes*) – get the value of average of indexes affect
- **weight** (*None, tuple,np.ndarray*) – the same size with self.shape[0]

Returns

Return type probability list

get_nodes_value(*ind=None*, *pset=None*, *node=None*, *site=None*)

get affect value except sites nodes.

Parameters

- **ind** (SymbolTree) –
- **pset** (SymbolSet) –
- **node** (*Terminals*) –
- **site** (*site of Terminals*) –

Returns

Return type probability list

get_one_node_value(*ind=None*, *pset=None*, *node=None*, *site=None*)

get affect value except site node.

Parameters

- **ind** (SymbolTree) –
- **pset** (SymbolSet) –
- **node** (*Terminals*) –
- **site** (*site of Terminals*) –

Returns

Return type probability list

noise()

add noise with 1% scale

set_ratio(**sv*)

Rate the [index1,index2] to sum and add the subbed value to the others. under check.

Parameters **sv** (*[index1,index2,rate]*) – rate in [0,n), if [0,1) down, if [1,n) up.

set_ratio_point(**sv*)

Rate the [index1,index2] to self and add the subbed value to the others. under check.

Parameters **sv** (*[index1,index2,rate]*) – in [0,n)

[0,1) down,

[1,n) up

set_sigle_point(**sv*)

set the value of [index1,index2] the rate are [0,1)

Parameters **sv** (*[index1,index2,rate]*) – site to set value

update(*ind*, *pset*, *ratio=0.5*)

Parameters

- **ind** (SymbolTree) – individual
- **pset** (SymbolSet) – SymbolSet
- **ratio** (*float [0,1]*) – change ratio

Returns

Return type self

Module contents

5.1.2 Submodules

5.1.3 bgp.base module

Base objects for symbolic regression.

Contains:

- Class: SymbolSet
- Class: CalculatePrecisionSet
- Class: SymbolTree
- others

class bgp.base.**CalculatePrecisionSet**(*pset*, *scoring=None*, *score_pen=(1,)*, *filter_warning=True*, *cv=1*, *cal_dim=True*, *dim_type=None*, *fuzzy=False*, *add_coef=True*, *inter_add=True*, *inner_add=False*, *vector_add=False*, *out_add=False*, *flat_add=False*, *n_jobs=1*, *batch_size=20*, *tq=True*, *details=False*, *classification=False*, *score_object='y'*, *batch_para=False*)

Bases: *bgp.base.SymbolSet*

Add score method to SymbolSet. The object can get from a worked SymbolSet object.

Parameters

- **pset** (SymbolSet) – SymbolSet.
- **scoring** (*Callbale, default is sklearn.metrics.r2_score.*) – See Also sklearn.metrics.
- **filter_warning** (*bool*) – bool.
- **score_pen** (*tuple of float*) – 1 : best is positive, worse -np.inf.

  -1 : best is negative, worse np.inf.

  0 : best is positive , worst is 0.

- **cal_dim** (*bool*) – calculate dim or not, if not return dless.
- **add_coef** (*bool*) – bool.
- **inter_add** (*bool*) – bool.
- **inner_add** (*bool*) – bool.
- **fuzzy** (*bool*) – fuzzy or not.
- **dim_type** (*object*) – if None, use the y_dim.
- **n_jobs** (*int*) – running core.
- **batch_size** (*int*) – batch size, advice batch_size*n_jobs = inds.
- **tq** (*bool*) – bool.
- **cv** (*sklearn.model_selection._split._BaseKFold, int*) – the shuffler must be False!

  use cv spilt for score, return the mean_test_score.

  use cv spilt for predict, return the cv_predict_y.(not be used)

  Notes: if cv and refit, all the data is refit to determination the coefficients.

  Thus the expression is not compact with the this scores, when re-calculated by this expression
- **details** (*bool*) – return the expr and predict y cor not.
- **classification** (*bool*) – classification or not.
- **score_object** – score by y or delta y (for implicit function).

calculate_cv_score(*ind*)
just used for calculating single one or check.

calculate_detail(*ind*)
just used for calculated final best one result for showing.

calculate the best expression.

Parameters **ind** (SymbolTree) – best expression.

calculate_expr(*expr*)
just used for calculated final result for showing.

Parameters **ind** (*sympy.Expr*) –

calculate_score(*ind*)
just used for calculating single one or check with cv=1.

Parameters **ind** (SymbolTree) –

calculate_simple(*ind*)
just used for re_Tree, and showing.

calculate the best expression.

Parameters **ind** (SymbolTree) –

compile_context(*ind*)
transform SymbolTree to sympy.Expr.

hasher
alias of `str`

parallelize_calculate_expr(*exprs*)
just used for final results, calculate exprs.

parallelize_score(*inds*)
The main score in each generation of GP!

Parameters **inds** (*list of SymbolTree*) – list of expressions

parallelize_try_add_coef_times(*exprs*, *grid_x=None*, *resample_number=500*)
to be continued

try_add_coef_times(*expr*, *grid_x=None*)
just used for best result, try add coefficient to expr.

update(*pset*)
updata self by input pset.

update_with_X_y(*X*, *y*)
replace x, y data.

class bgp.base.**ShortStr**(*st*)
Bases: object

short version of tree, just left name to simplify the store and transmit.

class bgp.base.**SymbolPrimitive**(*name*, *arity*)
Bases: object

General operator type, do not use directly, but use SymbolPrimitiveDetail.

Parameters

- **name** (*str*) – function name.
- **arity** (*int*) – input parameters numbers of function. such as + with 2, ln with 1.

format_repr(**args*)

format_str(**args*)

class bgp.base.**SymbolPrimitiveDetail**(*name*, *arity*, *func*, *prob*, *np_func=None*, *dim_func=None*, *sym_func=None*)
Bases: *bgp.base.SymbolPrimitive*

General operator type with more details.

Parameters

- **func** (*Callable*) – function. better using sympy.Function Type.

  For Maintainer: If self function and can not be simplified to sympy.Function or elementary function, the function for function.np_map() and dim.dim_map() should be defined.
- **name** (*str*) – function name.
- **arity** (*int*) – function input numbers.
- **prob** (*float*) – default 1.

capsule()
return short one.

class bgp.base.**SymbolSet**(*name='PSet'*)
Bases: object

Definite the preparation set of operations, features, and fixed constants.

Parameters **name** (*str*) – name.

add_accumulative_operation(*categories=None*, *categories_prob='balance'*, *self_categories=None*, *special_prob=None*)

add accumulative operation.

Parameters

- **categories** (*tuple of str*) – categories=(“Self”, “MAdd”, “MSub”, “MMul”, “MDiv”)
- **categories_prob** (*None, "balance" or float.*) – probility of categories in (0, 1], except (“Self”, “MAdd”, “MSub”, “MMul”, “MDiv”),

  ”balance” is 1/n_categories.

  ”MSub”, “MMul”, “MDiv” are only worked on the size of group is 2, else work like “Self”.

  Notes: the (“Self”, “MAdd”, “MSub”, “MMul”, “MDiv”) are set as 1 to be a standard.
- **self_categories** (*list of dict, None*) – the dict can be generate from newfuncD or defination self.

  the function at least containing:

  {“func”: func, “name”: name, “np_func”: npf, “dim_func”: dimf, “sym_func”: gsymf}

  1.func:sympy.Function(name) object, which need add attributes: is_jump, keep.

  2.name:name

  3.np_func:numpy function

  4.dim_func:dimension function

  5.sym_func:NewArray function. (unpack the group, used just for shown)

  See Also bgp.newfunc.newfuncV
- **special_prob** (*None or dict*) –

  Examples: {“MAdd”:0.5, “Self”:0.5}

add_constants(*c*, *c_dim=1*, *c_prob=None*)

Add features with dimension and probability.

Parameters

- **c_dim** (*1, list of Dim*) – the same size wih c.
- **c** (*float, list*) – list of float.
- **c_prob** (*None, float, list of float*) – the same size with c.

add_features(*X*, *y*, *x_dim=1*, *y_dim=1*, *x_prob=None*, *x_group=None*, *feature_name=None*)

Add features with dimension and probability.

Parameters

- **X** (*np.ndarray*) – 2D data.
- **y** (*np.ndarray*) – 1D data.
- **feature_name** (*None, list of str*) – the same size wih x.shape[1].
- **x_dim** (*1 or list of Dim*) – the same size wih x.shape[1], default 1 is dless for all x.
- **y_dim** (*1, Dim*) – dim of y.

- **x_prob** (*None, list of float*) – the same size wih x.shape[1].
- **x_group** (*None or list of list, int*) – features group.

add_features_and_constants(*X*, *y*, *c=None*, *x_dim=1*, *y_dim=1*, *c_dim=1*, *x_prob=None*, *c_prob=None*, *x_group=None*, *feature_name=None*)

combination of add_constant and add_features.

add_operations(*power_categories=None*, *categories=None*, *self_categories=None*, *power_categories_prob='balance'*, *categories_prob='balance'*, *special_prob=None*)

Add operations with probability.

Parameters

- **power_categories** (*Sized, tuple, None*) –

  Examples: (0.5, 2, 3)

- **categories** (*tuple of str*) –

  map table: {'Add': sympy.Add, 'Sub': Sub, 'Mul': sympy.Mul, 'Div': Div} {"sin": sympy.sin, 'cos': sympy.cos, 'exp': sympy.exp, 'ln': sympy.ln, } {'Abs': sympy.Abs, "Neg": functools.partial(sympy.Mul, -1.0), } "Rec": functools.partial(sympy.Pow, e=-1.0)}

  Others:

  "Rem": f(x)=1-x, if x true

  "Self": f(x)=x, if x true

- **categories_prob** (*"balance", float*) – probability of categories, except (+, -, /), in (0, 1]. "balance" is 1/n_categories. The (+, -, /) are set as 1 to be a standard.
- **special_prob** (*None, dict*) – prob for special name.

  Examples:{"Mul":0.6, "Add":0.4, "exp":0.1}

- **power_categories_prob** (*"balance", float*) – float in (0, 1]. probability of power categories, "balance" is 1/power_categories_prob.
- **self_categories** (*list of dict, None*) – the dict can be generate from newfuncV or definition self.

  the function at least containing: {"func": func, "name": name, "arity":2, "np_func": npf, "dim_func": dimf, "sym_func": gsymf}

  1.func:sympy.Function(name) object

  2.name:name

  3.arity:int, the number of parameter

  4.np_func:numpy function

  5.dim_func:dimension function

  6.sym_func:NewArray function. (unpack the group, used just for shown)

  See Also bgp.newfunc.newfuncV

add_tree_to_features(*Tree*, *prob=0.3*)

Add the individual as a new feature to initial features. not sure add success, because the value and name should be check and different to exist.

Parameters

- **Tree** (SymbolTree) – individual or expression
- **prob** (*int*) – probability of this individual

bonding_personal_maps(*pers*)
Personal preference add to permap more control can be found by pset.premap

Bond the points with ratio. the others would be penalty.

For example set the [1, 2, 0.9], the others bond such as (1, 2), (1, 3), (1, 4),. . . ,(2, 3), (2, 4). . . would be with small prob.

Parameters **pers** (*list of list*) –

Examples: [[index1, index2, prob][. . .]] the prob is [0, 1), 1 means the force binding.

property data_x

property dim_ter_con_list

property dispose
accumulate operators

property free_symbol

static get_values(*v*, *mean=False*)
get list of dict values

property init_free_symbol

property primitives
operators

property prob_dispose_list

property prob_pri_list

property prob_ter_con_list

register(*primitives_dict='all'*, *dispose_dict='all'*, *ter_con_dict='all'*)
Register and capsule for simplify.

Parameters

- **primitives_dict** (*None, str, dict*) –
- **dispose_dict** (*None, str, dict*) –
- **ter_con_dict** (*None, str, dict*) –

replace(*X*, *y=None*, *tree_X=None*)

set_personal_maps(*pers*)
personal preference add to permap. more control can be found by pset.premap.***

Just set couples of points and don't chang others.

Parameters **pers** (*list of list*) –

Examples: [[index1, index2, prob]], the prob in [0, 1).

property terminalRatio
Return the ratio of the number of terminals on the number of all kind of primitives.

property terminals_and_constants

property terminals_and_constants_repr

property types

class bgp.base.**SymbolTerminal**(*name*, *init_name=None*)
Bases: object

General feature type, do not use directly.

The name for show (str) and calculation (repr) are set to different string for avoiding repeated calculations.

Parameters

- **name** (*str*) – Represent name. Default "xi".
- **init_name** (*str*) – Just for show, rather than calculate.

  Examples: init_name=[x1, x2] , if is compact features, need[].

  init_name=(x1*x4-x3), if is expr, need ().

format_repr()
representing name

format_str()
represented name

class bgp.base.**SymbolTerminalDetail**(*values*, *name*, *dim=None*, *prob=None*, *init_sym=None*, *init_name=None*)
Bases: *bgp.base.SymbolTerminal*

General feature type.

The name for show (str) and calculation (repr) are set to different string for avoiding repeated calculations.

Parameters

- **values** (*None, number or np.ndarray*) – xi value, the shape can be (n,) or (n_x, n), n is number of samples, n_x is numbers of feature.
- **name** (*str*) – Represent name. Default "xi"
- **dim** (*bgp.dim.Dim or None*) – None.
- **prob** (*float or None*) – None.
- **init_sym** (*list, sympy.Expr*) – list.
- **init_name** (*str or None*) – Just for show, rather than calculate.

  Examples: init_name="[x1, x2]" , if is compact features, need[].

  init_name="(x1*x4-x3)", if is expr, need ().

capsule()

class bgp.base.**SymbolTree**(**arg*, ***kwargs*)
Bases: bgp.base._ExprTree

Individual Tree, each tree is one expression. The SymbolTree is only generated by method: genGrow and genFull.

property capsule
return the short one

compress()
drop unnecessary attributes

depart()
take part the expression

classmethod genFull(*pset*, *min_*, *max_*, *per=False*)
details in genGrow function

classmethod genGrow(*pset*, *min_*, *max_*, *per=False*)
details in genGrow function

ppprint(*pset*, *feature_name=False*)
get a user friendly version

reset()
keep these attribute refreshed

ter_site()
site for feature and constants node

terminals()
Return terminals that occur in the expression tree.

to_expr(*pset*)
transformed to sympy.Expr

5.1.4 bgp.flow module

Some definition loop for genetic algorithm. All the loop is with same run method.

Contains:

-Class: `BaseLoop`

one node mate and one tree mutate.

-Class: `MultiMutateLoop`

one node mate and (one tree mutate, one node Replacement mutate, shrink mutate, difference mutate).

-Class: `OnePointMutateLoop`

one node Replacement mutate: (keep height of tree)

-Class: `DimForceLoop`

Select with dimension : (keep dimension of tree)

class bgp.flow.**BaseLoop**(*pset*, *pop=500*, *gen=20*, *mutate_prob=0.5*, *mate_prob=0.8*, *hall=1*, *re_hall=1*, *re_Tree=None*, *initial_min=None*, *initial_max=3*, *max_value=5*, *scoring=(<function r2_score>,)*, *score_pen=(1,)*, *filter_warning=True*, *cv=1*, *add_coef=True*, *inter_add=True*, *inner_add=False*, *vector_add=False*, *out_add=False*, *flat_add=False*, *cal_dim=False*, *dim_type=None*, *fuzzy=False*, *n_jobs=1*, *batch_size=40*, *random_state=None*, *stats=None*, *verbose=True*, *migrate_prob=0*, *tq=True*, *store=False*, *personal_map=False*, *stop_condition=None*, *details=False*, *classification=False*, *score_object='y'*, *sub_mu_max=1*, *limit_type='h_bgp'*, *batch_para=False*)

Bases: `mgetool.packbox.Toolbox`

Base loop for BGP.

Examples:

```
if __name__ == "__main__":
    pset = SymbolSet()
    stop = lambda ind: ind.fitness.values[0] >= 0.880963
```

(continues on next page)

(continued from previous page)

```
bl = BaseLoop(pset=pset, gen=10, pop=1000, hall=1, batch_size=40, re_hall=3,

n_jobs=12, mate_prob=0.9, max_value=5, initial_min=1, initial_max=2,

mutate_prob=0.8, tq=True, dim_type="coef", stop_condition=stop,

re_Tree=0, store=False, random_state=1, verbose=True,

stats={"fitness_dim_max": ["max"], "dim_is_target": ["sum"]},

add_coef=True, inter_add=True, inner_add=False, cal_dim=True, vector_add=False,

personal_map=False)

bl.run()
```

Parameters

- **pset** (SymbolSet) – the feature x and target y and others should have been added.
- **pop** (*int*) – number of population.
- **gen** (*int*) – number of generation.
- **mutate_prob** (*float*) – probability of mutate.
- **mate_prob** (*float*) – probability of mate(crossover).
- **initial_max** (*int*) – max initial size of expression when first producing.
- **initial_min** (*None,int*) – min initial size of expression when first producing.
- **max_value** (*int*) – max size of expression.
- **limit_type** (*"height" or "length","h_bgp"*) – limitation type for max_value, but don't affect initial_max, initial_min.
- **hall** (*int,>=1*) – number of HallOfFame (elite) to maintain.
- **re_hall** (*None or int>=2*) – Notes: only valid when hall number of HallOfFame to add to next generation.
- **re_Tree** (*int*) – number of new features to add to next generation. 0 is false to add.
- **personal_map** (*bool or "auto"*) – "auto" is using 'premap' and with auto refresh the 'premap' with individual.

  True is just using constant 'premap'.

  False is just use the prob of terminals.
- **scoring** (*list of Callable, default is [sklearn.metrics.r2_score,]*) – See Also sklearn.metrics
- **score_pen** (*tuple of 1, -1 or float but 0.*) – >0 : max problem, best is positive, worse -np.inf. <0 : min problem, best is negative, worse np.inf.

  Notes: if multiply score method, the scores must be turn to same dimension in prepossessing or weight by score_pen. Because the all the selection are stand on the mean(w_i*score_i)

  Examples:

```
scoring = [r2_score,]
score_pen= [1,]
```

- **cv** (*sklearn.model_selection._split._BaseKFold,int*) – the shuffler must be False, default=1 means no cv.
- **filter_warning** (*bool*) – filter warning or not.
- **add_coef** (*bool*) – add coef in expression or not.
- **inter_addbool** – add intercept constant or not.
- **inner_add** (*bool*) – add inner coefficients or not.
- **out_add** (*bool*) – add out coefficients or not.
- **flat_add** (*bool*) – add flat coefficients or not.
- **n_jobs** (*int*) – default 1, advise 6.
- **batch_size** (*int*) – default 40, depend of machine.
- **random_state** (*int*) – None,int.
- **cal_dim** (*bool*) – escape the dim calculation.
- **dim_type** (*Dim or None or list of Dim*) – "coef": af(x)+b. a,b have dimension,f(x)'s dimension is not dnan.

  "integer": af(x)+b. f(x) is with integer dimension.

  [Dim1,Dim2]: f(x)'s dimension in list.

  Dim: f(x) ~= Dim. (see fuzzy)

  Dim: f(x) == Dim.

  None: f(x) == pset.y_dim
- **fuzzy** (*bool*) – choose the dim with same base with dim_type, such as m,m^2,m^3.
- **stats** (*dict*) – details of logbook to show.

  Map:

  values = {"max": np.max, "mean": np.mean, "min": np.mean, "std": np.std, "sum": np.sum}

  keys = {

  "fitness": just see fitness[0],

  "fitness_dim_max": max problem, see fitness with demand dim,

  "fitness_dim_min": min problem, see fitness with demand dim,

  "dim_is_target": demand dim,

  "coef": dim is True, coef have dim,

  "integer": dim is integer,

  ... }

  if stats is None, default is:

  for cal_dim=True: stats = {"fitness_dim_max": ("max",), "dim_is_target": ("sum",)}

  for cal_dim=False stats = {"fitness": ("max",)}

if self-definition, the key is func to get attribute of each ind.

Examples:

```
def func(ind):
    return ind.fitness[0]
stats = {func: ("mean",), "dim_is_target": ("sum",)}
```

- **verbose** (*bool*) – print verbose logbook or not.
- **tq** (*bool*) – print progress bar or not.
- **store** (*bool or path*) – bool or path.
- **stop_condition** (*callable*) – stop condition on the best ind of hall, which return bool,the true means stop loop.

  Examples:

```
def func(ind):
    c = ind.fitness.values[0]>=0.90
    return c
```

- **details** (*bool*) – return expr and predict_y or not.
- **classification** (*bool*) – classification or not.
- **score_object** – score by y or delta y (for implicit function).

check_height_length(*pop*, *site=''*)

maintain_halls(*population*)
maintain the best p expression

re_add()
add the expression as a feature

re_fresh_by_name(**arr*)

run(*warm_start=False*, *new_gen=None*)

Parameters

- **warm_start** (*bool*) – warm_start from last result.
- **new_gen** – new generations for warm_startm, default is the initial generations.

to_csv(*data_all*)
store to csv

top_n(*n=10*, *gen=- 1*, *key='value'*, *filter_dim=True*, *ascending=False*)
Return the best n results.

Note: Only valid in `store=True`.

Parameters

- **n** (*int*) –

  n.

- **gen** – the generation, default is -1.
- **key** (*str*) – sort keys, default is "values".
- **filter_dim** – filter no-dim expressions or not.
- **ascending** – reverse.

Returns

- *top n results.*
- *pd.DataFrame*

varAnd(**arg*, ***kwargs*)

class bgp.flow.**DimForceLoop**(**args*, ***kwargs*)
Bases: *bgp.flow.MultiMutateLoop*

Force select the individual with target dim for next generation

See also BaseLoop

class bgp.flow.**MultiMutateLoop**(**args*, ***kwargs*)
Bases: *bgp.flow.BaseLoop*

multiply mutate method.

See also BaseLoop

varAnd(*population*, *toolbox*, *cxpb*, *mutpb*)

class bgp.flow.**OnePointMutateLoop**(**args*, ***kwargs*)
Bases: *bgp.flow.BaseLoop*

limitation height of population, just use mutNodeReplacementVerbose method.

See also BaseLoop

varAnd(*population*, *toolbox*, *cxpb*, *mutpb*)

5.1.5 bgp.gp module

Notes

This part are one copy from deap, change the random to numpy.random.

bgp.gp.**Statis_func**(*stats=None*)

bgp.gp.**checks_number**(*func*)

bgp.gp.**checkss**(*func*)

bgp.gp.**cxOnePoint**(*ind10*, *ind20*)
Randomly select crossover point in each individual and exchange each subtree with the point as root between each individual.

Parameters

- **ind10** – First tree participating in the crossover.
- **ind20** – Second tree participating in the crossover.

Returns A tuple of two trees.

bgp.gp.**depart**(*individual*)
take part expression.

bgp.gp.**genFull**(*pset*, *min_*, *max_*, *personal_map=False*)
Generate an expression where each leaf has the same depth between *min* and *max*.

Parameters

- **pset** – Primitive set from which primitives are selected.
- **min** – Minimum height of the produced trees.
- **max** – Maximum Height of the produced trees.
- **personal_map** –

Returns A full tree with all leaves at the same depth.

bgp.gp.**genGrow**(*pset*, *min_*, *max_*, *personal_map=False*)
Generate an expression where each leaf might have a different depth between *min* and *max*.

Parameters

- **pset** – Primitive set from which primitives are selected.
- **min** – Minimum height of the produced trees.
- **max** – Maximum Height of the produced trees.
- **personal_map** – bool.

Returns A grown tree with leaves at possibly different depths.

bgp.gp.**genHalf**(*pset*, *min_*, *max_*, *personal_map=False*)

bgp.gp.**generate**(*pset*, *min_*, *max_*, *condition*, *personal_map=False*, **kwargs*)
generate expression.

Parameters

- **pset** (SymbolSet) – pset
- **min** (*int*) – Minimum height of the produced trees.
- **max** (*int*) – Maximum Height of the produced trees.
- **condition** (*collections.Callable*) – The condition is a function that takes two arguments, the height of the tree to build and the current depth in the tree.
- **kwargs** (*None*) – placeholder for future
- **personal_map** (*bool*) – premap

bgp.gp.**mutDifferentReplacementVerbose**(*individual*, *pset*, *personal_map=False*)
choice terminals_and_constants verbose Replaces a randomly chosen primitive from *individual* by a randomly chosen primitive with the same number of arguments from the `pset` attribute of the individual. decrease the probability of same terminals.

Parameters

- **individual** – The normal or typed tree to be mutated.
- **pset** – SymbolSet
- **personal_map** – bool

Returns A tuple of one tree.

bgp.gp.**mutNodeReplacementVerbose**(*individual*, *pset*, *personal_map=False*)
choice terminals_and_constants verbose Replaces a randomly chosen primitive from *individual* by a randomly chosen primitive with the same number of arguments from the `pset` attribute of the individual.

Parameters

- **individual** – The normal or typed tree to be mutated.
- **pset** – SymbolSet
- **personal_map** – bool

Returns A tuple of one tree.

bgp.gp.**mutShrink**(*individual*, *pset=None*)
This operator shrinks the *individual* by choosing randomly a branch and replacing it with one of the branch's arguments (also randomly chosen).

Parameters

- **individual** – The tree to be shrinked.
- **pset** – SymbolSet.

Returns A tuple of one tree.

bgp.gp.**mutUniform**(*individual*, *expr*, *pset*)
Randomly select a point in the tree *individual*, then replace the subtree at that point as a root by the expression generated using method `expr()`.

Parameters

- **individual** – The tree to be mutated.
- **expr** – A function object that can generate an expression when called.
- **pset** – SymbolSet

Returns A tuple of one tree.

bgp.gp.**selBest**(*individuals*, *k*, *fit_attr='fitness'*)
Select the *k* best individuals among the input *individuals*. The list returned contains references to the input *individuals*.

Parameters

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.
- **fit_attr** – The attribute of individuals to use as selection criterion

Returns A list containing the k best individuals.

bgp.gp.**selKbestDim**(*pop*, *K_best=10*, *dim_type=None*, *fuzzy=False*, *fit_attr='fitness'*, *force_number=False*)
Select the individual with dim limitation.

Parameters

- **pop** (SymbolTree) – A list of individuals to select from.
- **K_best** (*int*) – The number of individuals to select.
- **dim_type** (Dim) –
- **fuzzy** (*bool*) – the dim or the dim with same base. such as m,m^2,m^3

- **fit_attr** (*str*) – The attribute of individuals to use as selection criterion, default attr is "fitness".
- **force_number** (*False*) – return the number the same with K.

Returns

Return type A list of selected individuals.

bgp.gp.**selRandom**(*individuals*, *k*)

Select *k* individuals at random from the input *individuals* with replacement. The list returned contains references to the input *individuals*.

Parameters

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.

Returns A list of selected individuals.

This function uses the `numpy.random.choice()` function

bgp.gp.**selTournament**(*individuals*, *k*, *tournsize*, *fit_attr='fitness'*)

Select the best individual among *tournsize* randomly chosen individuals, *k* times. The list returned contains references to the input *individuals*.

Parameters

- **individuals** – A list of individuals to select from.
- **k** – The number of individuals to select.
- **tournsize** – The number of individuals participating in each tournament.
- **fit_attr** – The attribute of individuals to use as selection criterion

Returns A list of selected individuals.

This function uses the `numpy.random.choice()` function

bgp.gp.**staticLimit**(*key*, *max_value*)

bgp.gp.**varAnd**(*population*, *toolbox*, *cxpb*, *mutpb*)

bgp.gp.**varAndfus**(*population*, *toolbox*, *cxpb*, *mutpb*, *fus*, *mutpb_list=1.0*)

Parameters

- **population** –
- **toolbox** –
- **cxpb** –
- **mutpb** –
- **fus** –
- **mutpb_list** (*float,list,None*) –

5.1.6 bgp.postprocess module

bgp.postprocess.**acf**(*expr01*, *x*, *y*, *init_c=None*, *terminals=None*, *c_terminals=None*, *np_maps=None*, *classification=False*, *built_format_input=False*)

Add coef fitting.

Try calculate predict y by sympy expression with coefficients. if except error return expr itself.

Parameters

- **expr01** (*sympy.Expr*) – expr for fitting.
- **x** (*list of np.ndarray or np.ndarray*) – real data with: [x1,x2,x3,...,x_n_feature].
- **y** (*np.ndarray with shape (n_sample,)*) – real data of target.
- **init_c** (*list of float or float,None*) – default 1.
- **terminals** (*List of sympy.Symbol,None*) – placeholder for xi, with the same features in expr01.
- **c_terminals** (*List of sympy.Symbol,None*) – placeholder for ci, with the same coefficients/constants in expr01.
- **np_maps** (*dict,default is None*) – for self-definition. 1. make your function with sympy.Function and arrange in in expr01. >>> x1, x2, x3, c1,c2,c3,c4 = sympy.symbols("x1,x2,x3,c1,c2,c3,c4") >>> Seg = sympy.Function("Seg") >>> expr01 = Seg(x1*x2) 2. write the numpy calculation method for this function. >>> def np_seg(x): >>> res = x >>> res[res>1]=-res[res>1] >>> return res 3. pass the np_maps parameters. >>> np_maps = {"Seg":np_seg}

  In total, when parse the expr01, find the numpy function in sequence by: (np_maps -> numpy's function -> system -> Error)
- **classification** (*bool*) – classfication or not, default False.
- **built_format_input** (*bool*) – use format_input function to check input parameters. Just used for temporary test or single case, due to format_input is repetitive.

Returns

- *pre_y* – np.array or None
- **expr01** (*Expr*) – New expr.

bgp.postprocess.**acfng**(*expr01*, *x*, *y*, *init_c=None*, *terminals=None*, *c_terminals=None*, *np_maps=None*, *classification=False*, *no_gradient_coef=- 1*, *no_gradient_coef_range=array([- 1, 0])*, *n_jobs=1*, *scoring='r2'*)

Add coefficients with no gradient coefficient.

Try calculate predict y by sympy expression with coefficients. if except error return expr itself.

Parameters

- **scoring** (*str*) – score in sklearn.metrics
- **n_jobs** (*int*) – parallize number
- **no_gradient_coef** (*int,sympy.Symbol*) – coefficient in no gradient function, default the last one. Examples: no_gradient_coef=sympy.Symbol("c2") no_gradient_coef=0
- **no_gradient_coef_range** – range of the special coef.
- **expr01** (*sympy.Expr*) – expr for fitting.
- **x** (*list of np.ndarray or np.ndarray*) – real data with: [x1,x2,x3,...,x_n_feature].

- **y** (*np.ndarray with shape (n_sample,)*) – real data of target.
- **init_c** (*list of float or float,None*) – default 1.
- **terminals** (*List of sympy.Symbol,None*) – placeholder for xi, with the same features in expr01.
- **c_terminals** (*List of sympy.Symbol,None*) – placeholder for ci, with the same coefficients/constants in expr01.
- **np_maps** (*dict,default is None*) – for self-definition. 1. make your function with sympy.Function and arrange in in expr01. >>> x1, x2, x3, c1,c2,c3,c4 = sympy.symbols("x1,x2,x3,c1,c2,c3,c4") >>> Seg = sympy.Function("Seg") >>> expr01 = Seg(x1*x2) 2. write the numpy calculation method for this function. >>> def np_seg(x): >>> res = x >>> res[res>1]=-res[res>1] >>> return res 3. pass the np_maps parameters. >>> np_maps = {"Seg":np_seg}

  In total, when parse the expr01, find the numpy function in sequence by: (np_maps -> numpy's function -> system -> Error)
- **classification** (*bool*) – classfication or not, default False.

Returns

- *pre_y* – np.array or None
- **expr01** (*Expr*) – New expr.

bgp.postprocess.**acfs**(*expr01*, *x*, *y*, *init_c=None*, *terminals=None*, *c_terminals=None*, *np_maps=None*, *classification=False*, *built_format_input=False*, *scoring='r2'*)

Add coefficients and score.

See also add_coef_fitting (acf).

bgp.postprocess.**acfsng**(*expr01*, *x*, *y*, *init_c=None*, *terminals=None*, *c_terminals=None*, *np_maps=None*, *classification=False*, *no_gradient_coef=- 1*, *no_gradient_coef_range=array([- 1, 0])*, *n_jobs=1*, *scoring='r2'*)

Add coefficients and score with no gradient coefficient.

See also add_coef_fitting (acf).

bgp.postprocess.**cla**(*pre_y*, *cl=True*)

bgp.postprocess.**format_input**(*expr01*, *x*, *y*, *init_c=None*, *terminals=None*, *c_terminals=None*, *np_maps=None*, *x_mark='x'*, *c_mark='c'*)

Check and format_input for add_coef_fitting.

Parameters

- **expr01** (*sympy.Expr*) – expr for fitting.
- **x** (*list of np.ndarray or np.ndarray*) – real data with: [x1,x2,x3,...,x_n_feature] or x with shape (n_sample,n_feature).
- **y** (*np.ndarray with shape (n_sample,)*) – real data of target.
- **init_c** (*list of float or float.*) – default 1.
- **terminals** (*list of sympy.Symbol*) – placeholder for xi, with the same features in expr01.
- **c_terminals** (*list of sympy.Symbol*) – placeholder for ci, with the same coefficients/constants in expr01.

- **np_maps** (*dict,default is None*) – for self-definition. 1. make your function with sympy.Function and arrange in in expr01. >>> x1, x2, x3, c1,c2,c3,c4 = sympy.symbols("x1,x2,x3,c1,c2,c3,c4") >>> Seg = sympy.Function("Seg") >>> expr01 = Seg(x1*x2,c1) 2. write the numpy calculation method for this function. >>> def np_seg(x,c): >>> res = -x >>> res[res>-c]=0 >>> return res 3. pass the np_maps parameters. >>> np_maps = {"Seg":np_seg}

  In total, when parse the expr01, find the numpy function in sequence by: (np_maps -> numpy's function -> system -> Error)
- **x_mark** (*str*) – mark for x
- **c_mark** (*str*) – mark for c

Returns **format_parameters** – (expr01, x, y, init_c, terminals, c_terminals, np_maps)

Return type tuple

bgp.postprocess.**top_n**(*loop*, *n=10*, *gen=- 1*, *key='value'*, *ascending=False*)

return the top result of loop. PendingDeprecation.

please use loop.top_n() directly.

5.1.7 bgp.preprocess module

class bgp.preprocess.**MagnitudeTransformer**(*standard=1*, *tolerate=0*)

Bases: sklearn.base.TransformerMixin, sklearn.base.BaseEstimator

Transform x, y or c to near to 1, and store the transform Magnitude.

fit(*X*, *y=None*, *group=2*, *apply=None*, *keep=None*)

Parameters

- **X** (*np.ndarray*) –
- **y** (*np.ndarray*) –
- **group** (*group index of x*) –
- **apply** (*specific which index of x to transform*) –
- **keep** (*specific which index of x to not transform*) –

fit_constant(*c*)

fit_transform_all(*X*, *y*, ***fit_params*)

fit_transform_constant(*c*)

inverse_transform(*X*)

inverse_transform_constant(*c*)

inverse_transform_y(*y*)

transform(*X*)

transform_constant(*c*)

transform_y(*y*)

5.1.8 bgp.skflow module

class bgp.skflow.**SymbolLearning**(*loop*, **args*, ***kwargs*)

Bases: sklearn.base.BaseEstimator, sklearn.base.MultiOutputMixin, sklearn.base.TransformerMixin

One simplify Guide for flow.

1. The SymbolLearning is time-costing and not suit for GridSearchCV, the cross_validate are embedded.

2. For the classification problems, please using classification =True, and set the suit classification metrics for scoring and score_pen carefully.

This code does not check and identity the certainty of data.

Parameters **<https** (*`Web of SymbolLearning*) – //bgp.readthedocs.io/en/latest/src/bgp.html#bgp.skflow.SymbolLearni

See also:

bgp.flow.BaseLoop

Parameters

- **loop** (*str,None*) – bgp.flow.BaseLoop

  ['BaseLoop', 'MultiMutateLoop', 'OnePointMutateLoop', 'DimForceLoop' ...].
- **pop** (*int*) – number of population.
- **gen** (*int*) – number of generation.
- **mutate_prob** (*float*) – probability of mutate.
- **mate_prob** (*float*) – probability of mate(crossover).
- **initial_max** (*int*) – max initial size of expression when first producing.
- **initial_min** (*None,int*) – min initial size of expression when first producing.
- **max_value** (*int*) – max size of expression.
- **hall** (*int,>=1*) – number of HallOfFame (elite) to maintain.
- **re_hall** (*None or int>=2*) – Notes: only valid when hall number of HallOfFame to add to next generation.
- **re_Tree** (*int*) – number of new features to add to next generation. 0 is false to add.
- **personal_map** (*bool or "auto"*) – "auto" is using 'premap' and with auto refresh the 'premap' with individual.

  True is just using constant 'premap'.

  False is just use the prob of terminals.
- **scoring** (*list of Callable, default is [sklearn.metrics.r2_score,]*) – See Also sklearn.metrics
- **score_pen** (*tuple of 1, -1 or float but 0.*) – >0 : max problem, best is positive, worse -np.inf. <0 : min problem, best is negative, worse np.inf.

  Notes: if multiply score method, the scores must be turn to same dimension in prepossessing or weight by score_pen. Because the all the selection are stand on the mean(w_i*score_i)

  Examples:

```
scoring = [r2_score,]
score_pen= [1,]
```

- **cv** (*sklearn.model_selection._split._BaseKFold,int*) – the shuffler must be False, default=1 means no cv.
- **filter_warning** (*bool*) – filter warning or not.
- **add_coef** (*bool*) – add coef in expression or not.
- **inter_addbool** – add intercept constant or not.
- **inner_add** (*bool*) – add inner coefficients or not.
- **out_add** (*bool*) – add out coefficients or not.
- **flat_add** (*bool*) – add flat coefficients or not.
- **n_jobs** (*int*) – default 1, advise 6.
- **batch_size** (*int*) – default 40, depend of machine.
- **random_state** (*int*) – None,int.
- **cal_dim** (*bool*) – escape the dim calculation.
- **dim_type** (*Dim or None or list of Dim*) – "coef": af(x)+b. a,b have dimension,f(x)'s dimension is not dnan.

  "integer": af(x)+b. f(x) is with integer dimension.

  [Dim1,Dim2]: f(x)'s dimension in list.

  Dim: f(x) ~= Dim. (see fuzzy)

  Dim: f(x) == Dim.

  None: f(x) == pset.y_dim
- **fuzzy** (*bool*) – choose the dim with same base with dim_type, such as m,m^2,m^3.
- **stats** (*dict*) – details of logbook to show.

  Map:

  values = {"max": np.max, "mean": np.mean, "min": np.mean, "std": np.std, "sum": np.sum}

  keys = {

  "fitness": just see fitness[0],

  "fitness_dim_max": max problem, see fitness with demand dim,

  "fitness_dim_min": min problem, see fitness with demand dim,

  "dim_is_target": demand dim,

  "coef": dim is True, coef have dim,

  "integer": dim is integer,

  ... }

  if stats is None, default is :

  for cal_dim=True: stats = {"fitness_dim_max": ("max",), "dim_is_target": ("sum",)}

  for cal_dim=False: stats = {"fitness": ("max",)}

if self-definition, the key is func to get attribute of each ind.

Examples:

```
def func(ind):
    return ind.fitness[0]
stats = { func: ("mean",), "dim_is_target":("sum",)}
```

- **verbose** (*bool*) – print verbose logbook or not.
- **tq** (*bool*) – print progress bar or not.
- **store** (*bool or path*) – bool or path.
- **stop_condition** (*callable*) – stop condition on the best ind of hall, which return bool,the true means stop loop.

  Examples:

```
def func(ind):
    c = ind.fitness.values[0]>=0.90
    return c
```

- **pset** (SymbolSet) – the feature x and target y and others should have been added.
- **details** (*bool*) – return expr and predict_y or not.
- **classification** (*bool*) – classification or not.

cv_result(*refit=False*)

return the cv_result of best expression. Only valid when cv !=1.

Parameters **refit** (*bool*) – re-fit the data or not. If true, use all the data on the best expression.

fit(*X=None*, *y=None*, *c=None*, *x_group=None*, *x_dim=1*, *y_dim=1*, *c_dim=1*, *x_prob=None*, *c_prob=None*, *pset=None*, *power_categories=(2, 3, 0.5)*, *categories=('Add', 'Mul', 'Sub', 'Div')*, *warm_start=False*, *new_gen=None*)

Method 1. fit with x, y.

Examples:

```
sl = SymbolLearning()
sl..fit(x,y,...)
```

Method 2. fit with customized pset. If need more self-definition, use one defined SymbolSet object to pset.

Examples:

```
pset = SymbolSet()
pset.add_features_and_constants(...)
pset.add_operations(...)
...
sl = SymbolLearning()
sl..fit(pset=pset)
```

Parameters

- **X** (*np.ndarray*) – data.
- **y** (*np.ndarray*) –

y.

- **c** (*list of float, None*) – constants.
- **x_dim** (*1 or list of Dim*) – the same size wih x.shape[1], default 1 is dless for all x.
- **y_dim** (*1,Dim*) – dim of y.
- **c_dim** (*1,list of Dim*) – the same size wih c.shape, default 1 is dless for all c.
- **x_prob** (*None,list of float*) – the same size wih x.shape[1].
- **c_prob** (*None,list of float*) – the same size wih c.
- **x_group** (*list of list*) – Group of x.

  Examples:

  x_group=[[1,2],] or x_group=2

  See Also *bgp.base.SymbolSet.add_features()*
- **power_categories** (*Sized,tuple, None*) – Examples:(0.5,2,3)
- **categories** (*tuple of str*) –

  map table: {"Add": sympy.Add, 'Sub': Sub, 'Mul': sympy.Mul, 'Div': Div} {"sin": sympy.sin, 'cos': sympy.cos, 'exp': sympy.exp, 'ln': sympy.ln, {'Abs': sympy.Abs, "Neg": functools.partial(sympy.Mul, -1.0), "Rec": functools.partial(sympy.Pow, e=-1.0)}

  Others:

  "Rem": f(x)=1-x,if x true

  "Self": f(x)=x,if x true

pset:SymbolSet See Also SymbolSet.

warm_start: bool warm start or not.

Note: If you offer pset in advance by user, please check carefully the feature numbers,especially when use `re_Tree`. because the new features are add.

Reference: CalculatePrecisionSet.update_with_X_y.

new_gen: None,int warm_start generation.

predict(*X*)

predict y from X.

Parameters **X** (*np.ndarray*) – data.

score(*X*, *y*, *scoring*)

Parameters

- **X** (*np.ndarray*) – data.
- **y** (*np.ndarray*) – true y.
- **scoring** (*str*) – scoring method,default is "r2"

5.1.9 Module contents

CHAPTER

SIX

CHINESE DOC

6.1

bgp.skflow.SymbolLearning ”sklearn-type”

loop, pop, gen, random_state, mutate_prob, mate_prob

random_state, batch_size, random_state

stats, verbose, store

max_value, initial_max

add_coef, inner_add, inter_add, out_add, flat_add

re-hall

dim_type, cal_dim, fuzzy, fit(x_dim,y_dim,c_dim)

classification, scoring, score_pen

stats, scoring, score_pen

fit(x_group)

fit(x_prob, c_prob)

fit.(warm_start), fit(new_gen)

— 1. SymbolLearning

```
if __name__ == "__main__":
    from sklearn.datasets import load_boston
    from bgp.skflow import SymbolLearning

    data = load_boston()
    x = data["data"]
    y = data["target"]
    c = [1, 2, 3]

    sl = SymbolLearning(loop="MultiMutateLoop", pop=500, gen=3, cal_dim=True, re_hall=2,␣
↪add_coef=True, cv=1, random_state=1
```

(continues on next page)

(continued from previous page)

```
                )
    sl.fit(x, y, c=c,x_group=[[1, 3], [0, 2], [4, 7]])
    score = sl.score(x, y, "r2")
    print(sl.expr)
```

2. , SymbolSet :py:doc:*bgp.base.SymbolSet* SymbolLearning fitpset

base**flow**

```
if __name__ == "__main__":
    from sklearn.datasets import load_boston
    from bgp.skflow import SymbolLearning
    from bgp.base import SymbolSet

    data = load_boston()
    x = data["data"]
    y = data["target"]
    c = [1, 2, 3]

    pset0 = SymbolSet()
    pset0.add_features(x, y, x_dim=x_dim, y_dim=y_dim, x_group=[[1, 2], [3, 4], [5, 6]])
    pset0.add_constants(c, c_dim=c_dim, c_prob=None)
    pset0.add_operations(power_categories=(2, 3, 0.5),
             categories=("Add", "Mul", "exp"),
             special_prob =  {"Mul":0.5,"Add":0.4,"exp":0.1},
             power_categories_prob = "balance"
             )

    sl = SymbolLearning(loop="MultiMutateLoop", pop=500, gen=3, cal_dim=True, re_hall=2,␣
↪add_coef=True, cv=1, random_state=1
                )
    sl.fit(pset=pset0)
    score = sl.score(x, y, "r2")
    print(sl.expr)
```

Examples

6.1.1 SL

loop: str 'MultiMutateLoop'.

'BaseLoop'

'MultiMutateLoop'

'OnePointMutateLoop'

'DimForceLoop'

pop:int

gen:int

mutate_prob:float

mate_prob:float

initial_max:int

initial_min [None,int]

max_value:int

hall:int,>=1

re_hall:None or int>=2 “hall”

re_Tree: int

personal_map:bool or “auto” “auto”

True

False

scoring: list of Callbale, default is [sklearn.metrics.r2_score,] sklearn.metrics

score_pen: tuple of 1, - 1 or float but 0. >0 : , - np.infr2_scoreaccuracy

<0 : , np.infMAE,MSE

Notes: score_pen

mean(w_i * score_i)

Examples: [r2_score] is [1]

cv:sklearn.model_selection._split._BaseKFold,int (cv)

filter_warning:bool warning

add_coef:bool

inter_addbool

inner_add:bool

out_add:bool

flat_add:bool

vector_add:bool

n_jobs:int

batch_size:int

random_state:int

cal_dim:bool

dim_type:Dim or None or list of Dim

“coef”: af(x)+b. a,b have dimension,f(x) is not dnan. f(x)

“integer”: af(x)+b. f(x) is interger dimension. f(x)

[Dim1,Dim2]: f(x) in list. f(x)

Dim: f(x) ~= Dim. (see fuzzy) f(x)fuzzy

Dim: f(x) == Dim. f(x)

None: f(x) == pset.y_dim f(x)

fuzzy:bool f(x) m,m^2,m^3

stats:dict

values= {“max”: np.max, “mean”: np.mean, “min”: np.mean, “std”: np.std, “sum”: np.sum} keys= { “fitness”: just see fitness[0], “fitness_dim_max”: max problem, see fitness with demand dim, “fitness_dim_min”: min problem, see fitness with demand dim, “dim_is_target”: demand dim, “coef”: dim is True, coef have dim, “integer”: dim is integer,}

cal_dim=True,stats = {“fitness_dim_max”: (“max”,), “dim_is_target”: (“sum”,)}

cal_dim=False,stats = {“fitness”: (“max”,)}

keys :

```
def func(ind):
    return ind.fitness[0]
stats = {func: ("mean",), "dim_is_target": ("sum",)}
```

verbose:bool

tq:bool

store:bool or path

stop_condition:callable :

```
def func(ind):
    c = ind.fitness.values[0]>=0.90
    return c
```

details:bool

classification: bool

pset:SymbolSet None XXyy None, fitpset

6.1.2 SL

fit

X:np.ndarray

y:np.ndarray

c:list of float

x_dim: 1 or list of Dim

y_dim: 1,Dim

c_dim: 1,Dim

x_prob: None,list of float

c_prob: None,list of float

x_group:int, list of list GP

x_group=2

x_group=[[1,2][3,4]],x1x2x3x4

See Also pset.add_features_and_constants

pset:SymbolSet None

pset,

psetNone, fitpset

Note: psetfitpset

warm_start: bool

Note: pset“re_Tree”=True

: CalculatePrecisionSet.update_with_X_y

new_gen: None,int .

6.1.3 SL

loop

best_one: SymbolTree SymbolTree

expr: sympy.Expr sympy.Expr

y_dim: Dim

fitness

6.2

1

2

1.
2. sympy

Dim.convert_x, Dim.convert_xi, Dim.convert_x xyc)

3

10^16

MagnitudeTransformer

4

```
from sympy.physics.units import kg, m
from bgp.functions.dimfunc import Dim, dless

x_u = [kg] * 12 + m
y_u = kg
c_u = [dless, dless, dless]

# Dim    the dim also could get by Dim(numpy.array([****])) directly.
x, x_dim = Dim.convert_x(x, x_u, target_units=None, unit_system="SI")
y, y_dim = Dim.convert_xi(y, y_u)
c, c_dim = Dim.convert_x(c, c_u)
```

CHAPTER

SEVEN

EXAMPLES

7.1 Regression

This is a regression, Max-problem sample.

```
if __name__ == "__main__":
  from sklearn.datasets import load_boston
  from bgp.skflow import SymbolLearning

  data = load_boston()
  x = data["data"]
  y = data["target"]

  sl = SymbolLearning(loop="MultiMutateLoop", pop=500, gen=2, random_state=1)
  sl.fit(x, y)
  score = sl.score(x, y, "r2")
  print(sl.expr)
```

7.2 Classification

This is a classification sample.

```
if __name__ == "__main__":
    from sklearn import metrics
    from sklearn.utils import shuffle
    from sklearn.datasets import load_iris
    from bgp.skflow import SymbolLearning

    data = load_iris()
    x = data["data"][:98, :]
    x[40:60] = shuffle(x[40:60], random_state=2)
    y = data["target"][:98]
    c = None

    sl = SymbolLearning(loop="MultiMutateLoop", pop=500, gen=2, random_state=1,
    classification=True, scoring=[metrics.accuracy_score,], score_pen=[1,])
    sl.fit(x, y)

    print(sl.expr)
```

7.3 Min Problem

This is a Min Problem sample.

```
if __name__ == "__main__":
    from sklearn.datasets import load_boston
    from bgp.skflow import SymbolLearning
    from sklearn import metrics
    data = load_boston()
    x = data["data"]
    y = data["target"]

    sl = SymbolLearning(loop="MultiMutateLoop", pop=500, gen=2, random_state=1,
                        scoring=[metrics.mean_absolute_error,],
                        score_pen=[-1,],
                        stats = {"fitness_dim_min": ("min",), "dim_is_target": ("sum",)},
                        )
    sl.fit(x, y)
    print(sl.expr)
```

7.4 Dimension

This is a Dimension calculation sample.

```
if __name__ == "__main__":
    from bgp.functions.dimfunc import dless
    from sklearn.datasets import load_boston
    from bgp.skflow import SymbolLearning

    data = load_boston()
    x = data["data"]
    y = data["target"]
    x_dim = [dless, dless, dless, dless, dless, dless, dless, dless,dless, dless, dless,␣
↪dless, dless]
    y_dim = dless

    sl = SymbolLearning(loop="MultiMutateLoop", pop=500, gen=2, random_state=1,cal_
↪dim=True, dim_type="coef")
    sl.fit(x, y,x_dim=x_dim, y_dim=y_dim)
    score = sl.score(x, y, "r2")
    print(sl.expr)
```

The details of *Dim* can be found in *Remarks*

7.5 Binding

This is a Binding sample.

```
if __name__ == "__main__":
    from sklearn.datasets import load_boston
    from bgp.skflow import SymbolLearning

    data = load_boston()
    x = data["data"]
    y = data["target"]

    sl = SymbolLearning(loop="MultiMutateLoop", pop=500, gen=2, random_state=1)
    sl.fit(x, y, x_group=[[1, 2], [3, 4], [6, 7]])
    score = sl.score(x, y, "r2")
    print(sl.expr)
```

7.6 Top n best

This is a sample checking the best n results in target generation.

Note: Only valid in `store=True`.

```
if __name__ == "__main__":
    from sklearn.datasets import load_iris
    from bgp.skflow import SymbolLearning
    from sklearn import metrics
    from sklearn.utils import shuffle

    data = load_iris()
    x = data["data"]
    y = data["target"]
    c = [1, 2, 3]

    sl = SymbolLearning(loop="MultiMutateLoop", pop=50,
                        re_hall=3,
                        gen=3, random_state=1,
                        classification=True,
                        scoring=[metrics.accuracy_score, ], score_pen=[1, ],
                        store=True,
                        )

    sl.fit(x, y, c=c)
    score = sl.score(x, y, "r2")
    top_n = sl.loop.top_n(10)
    print(sl.expr)
```

where the top n is a table (Pandas.DataFrame object), as following:

	dim_score	dimension	expr	gen	name	value
100	1.00000	[0. 0. 0. 0. 0. 0. 0.]	['-8.025 + 98.53*x3**2/x1**2']	3.00000	pow0(Div(x3, x1))	0.66667
113	1.00000	[0. 0. 0. 0. 0. 0. 0.]	['1.815*x2**2 + 1.815*x3 - 12.75']	3.00000	Add(x3, pow0(x2))	0.66667
148	1.00000	[0. 0. 0. 0. 0. 0. 0.]	['9.282*(x2**2 + x3)**0.5 - 23.82']	3.00000	pow2(Add(x3, pow0(x2)))	0.66667
147	1.00000	[0. 0. 0. 0. 0. 0. 0.]	['3.366*c0 + 3.366*x2*x3 + 3.366*x2 - 18.31']	3.00000	Add(Add(x2, c0), Mul(x2, x3))	0.66667
146	1.00000	[0. 0. 0. 0. 0. 0. 0.]	['1.286*(x3/x0)**0.5*(x0 - x2**2 - x3)**2 - 9.692']	3.00000	Mul(pow2(Div(x3, x0)), pow0(Sub(x0, Add(x3, pow0(x2)))))	0.66667
144	1.00000	[0. 0. 0. 0. 0. 0. 0.]	['15.75*x2**0.5 + 15.75*x3 - 15.75*(x2/x3)**0.5 - 8.618']	3.00000	Sub(Sub(pow2(Div(x2, x3)), x3), pow2(x2))	0.66667
141	1.00000	[0. 0. 0. 0. 0. 0. 0.]	['2.308*x1 + 2.308*x2**2 + 2.308*x3 - 23.58']	3.00000	Add(Add(x3, x1), pow0(x2))	0.66667
133	1.00000	[0. 0. 0. 0. 0. 0. 0.]	['3.969*c0 + 3.969*x2*x3 + 3.969*x3 - 15.29']	3.00000	Add(Add(x3, c0), Mul(x2, x3))	0.66667
132	1.00000	[0. 0. 0. 0. 0. 0. 0.]	['-30.65 + 63.29*x2/x0']	3.00000	Div(x2, x0)	0.66667
120	1.00000	[0. 0. 0. 0. 0. 0. 0.]	['31.06*x3 - 24.39']	3.00000	x3	0.66667

7.7 Complexity Control

This is a part to equation complexity control from 3 aspect.

1.limitation of length of equation.

initial_max:int max initial size of expression when first producing.

initial_min [None,int] min initial size of expression when first producing.

max_value:int max size of expression.

limit_type: "height" or "length","","h_bgp" limitation type for max_value, but just affect `max_value` rather than `initial_max`, `initial_min`.

2. Sites of fit coefficients.

   add_coef: The main switch of coefficients. default: Add the coefficients of expression. such as y=cf(x).

   inter_add: Add the intercept of expression. such as y=f(x)+b.

   out_add: Add the coefficients of expression. such as y=a(x), but for polynomial join by + and -,the coefficient would add before each term. such as y=af1(x)+bf2(x).

   flat_add: flatten the expression and add the coefficients out of expression. such as y=af`1(x)+bf`2(x)+ef`3(x), (the old expression: y = x*(f1(x)+f2(x)+f3(x))).

   inner_add: Add the coefficients inner of expression. such as y=cf(ax).

   vector_add: only valid when x_group is True, add different coefficients on group x pair.

3. **Dimension limitation.** (To some extent, the Dimension limitation could affects the complexity of the formula indirectly.)

   `cal_dim`: The main switch of calculate dimension or not.

   `dim_type`: What kind of dimension of equation fit the bill.

   "coef": af(x)+b. a,b have dimension, f(x)'s dimension is not dnan.

   "integer": af(x)+b. f(x) is with integer dimension.

   [Dim1,Dim2]: f(x)'s dimension in list.

   Dim: f(x) ~= Dim. (see fuzzy)

   Dim: f(x) == Dim.

   None: f(x) == pset.y_dim

Note: From sample 4, The formula to be more and more complicated.

1. Ordinary SL.:

```
sl = SymbolLearning(loop="MultiMutateLoop", pop=100, gen=10, random_state=1,
                    cal_dim=False, n_jobs = 10,add_coef=False)
sl.fit(x,y)
```

2. SL with add coefficients: af(x).:

```
sl = SymbolLearning(loop="MultiMutateLoop", pop=100, gen=10, random_state=1,
                    cal_dim=False, n_jobs = 10,add_coef=True,inter_add=False)
sl.fit(x,y)
```

3. SL with add coefficients (default,if do not change the default parameters): af(x)+b.:

```
sl = SymbolLearning(loop="MultiMutateLoop", pop=100, gen=10, random_state=1,
                    cal_dim=False, n_jobs = 10,add_coef=True,inter_add=True)
sl.fit(x,y)
```

4. SL with add coefficients, with dimension calculation (default): af(x)+b.:

```
from sympy.physics.units import kg, m, pa, J, mol, K
from bgp.functions.dimfunc import Dim, dless

# Transform to SI unit, and get Dims
gpa_dim= Dim.convert_to_Dim(1e9*pa, unit_system="SI")
j_d_mol_dim = Dim.convert_to_Dim(1000*J/mol, unit_system="SI")
kg_d_m3_dim = Dim.convert_to_Dim(kg/m**3, unit_system="SI")

# or just write Dim by yourself
K_dim= Dim([0,1,0,0,0,0,0])

y_dim = dless
x_dim = [dless,gpa_dim[1],j_d_mol_dim[1],K_dim[1],dless,kg_d_m3_dim]

sl = SymbolLearning(loop="MultiMutateLoop", pop=100, gen=10, random_state=1,
                    cal_dim=True,
                    dim_type=None
                    # dim_type=y_dim
                    n_jobs = 10,add_coef=True,inter_add=True,)
sl.fit(x,y,x_dim=x_dim,y_dim=y_dim)
```

5. SL with add coefficients, with dimension calculation, but relax the requirement: just require that the dimension f(x) is not NaN for af(x)+b.:

```
sl = SymbolLearning(loop="MultiMutateLoop", pop=100, gen=10, random_state=1,
                    cal_dim=True,
                    dim_type="coef"
                    n_jobs = 10,add_coef=True,inter_add=True,)
sl.fit(x,y,x_dim=x_dim,y_dim=y_dim)
```

6. SL with add coefficients, with dimension calculation, but relax the requirement: just require that the dimension f(x) is not NaN for af(x)+b or af(x)+cf(x)+b.:

```
sl = SymbolLearning(loop="MultiMutateLoop", pop=100, gen=10, random_state=1,
                    cal_dim=True,dim_type="coef"
```

(continues on next page)

(continued from previous page)

```
                    n_jobs = 10,add_coef=True,inter_add=True, inner_add=False, out_
↪add=True, flat_add=False)
sl.fit(x,y,x_dim=x_dim,y_dim=y_dim)
```

7. SL with add coefficients, with dimension calculation, but relax the requirement: just require that the dimension f(x) is not NaN for flattened af(x)+cf(x)+b.:

```
sl = SymbolLearning(loop="MultiMutateLoop", pop=100, gen=10, random_state=1,
                    cal_dim=True,dim_type="coef"
                    n_jobs = 10,add_coef=True, inter_add=True, inner_add=False, out_
↪add=False, flat_add=True)
sl.fit(x,y,x_dim=x_dim,y_dim=y_dim)
```

8. SL with add coefficients, with dimension calculation, but relax the requirement: just require that the dimension f(x) is not NaN for af(cx)+b.:

```
sl = SymbolLearning(loop="MultiMutateLoop", pop=100, gen=10, random_state=1,
                    cal_dim=True,dim_type="coef"
                    n_jobs = 10,add_coef=True,inter_add=True, inner_add=True, out_
↪add=False, flat_add=False)
sl.fit(x,y,x_dim=x_dim,y_dim=y_dim)
```

9. SL with add coefficients, with dimension calculation, but relax the requirement: just require that the dimension f(x) is not NaN for af(cx)+b.:

```
sl = SymbolLearning(loop="MultiMutateLoop", pop=100, gen=10, random_state=1,
                    cal_dim=True,dim_type="coef"
                    n_jobs = 10,add_coef=True,inter_add=True, inner_add=True, out_
↪add=False, flat_add=False)
sl.fit(x,y,x_dim=x_dim,y_dim=y_dim)
```

10. SL with add coefficients, with dimension calculation, change max_value.:

    sl = SymbolLearning(loop="MultiMutateLoop", pop=100, gen=10, random_state=1,
    cal_dim=True,dim_type="coef", initial_max=7, initial_min=3,max_value=7,limit_type="h_bgp", n_jobs = 10,add_coef=True,inter_add=True, inner_add=True, out_add=False, flat_add=False)

    sl.fit(x,y,x_dim=x_dim,y_dim=y_dim)

11. Complex equation(most complicated, slowest, unaccountably).:

    sl = SymbolLearning(loop="MultiMutateLoop", pop=100, gen=10, random_state=1, cal_dim=False, max_value=7, n_jobs = 10, add_coef=True, inner_add=True)

    sl.fit(x,y,x_dim=x_dim,y_dim=y_dim)

CHAPTER

EIGHT

CONTACT

Thanks for your reading.

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

Please contact with Changxin.

This tool is a symbol regression tool with dimension calculation, which is aimed to establish expressions with physical limitation.

CHAPTER

NINE

FEATURES:

1. Coefficient fitting and addition
2. Dimension calculation
3. Accumulative operation and free custom
4. Characteristics feedback
5. High efficiency parallelism

CHAPTER

TEN

LINKS

BGP homepage

BGP source code

CHAPTER

ELEVEN

INDEX

- genindex
- modindex

PYTHON MODULE INDEX

b

INDEX

D

F

G

H

I

M

N

O

P

R

S

T

U

V