pspso

Release 0.1.3

Ali Haidar

Sep 18, 2020

CONTENTS

CHAPTER

ONE

OVERVIEW AND INSTALLATION

1.1 Overview

pspso is a python library for selecting machine learning algorithms parameters. The first version supports two single algorithms: Multi-Layer Perceptron (MLP) and Support Vector Machine (SVM). It supports two ensembles: Extreme Gradient Boosting (XGBoost) and Gradient Boosting Decision Trees (GBDT).

Two types of machine learning tasks are supported by pspso:

- Regression.
- Binary classification.

Three scores are supported in the first version of pspso:

- **Regression** :
  - Root Mean Square Error (RMSE)
- **Binary Classication** :
  - Area under the Curve (AUC) of the Receiver Operating Characteristic (ROC)
  - Accuracy

1.2 Installation

Use the package manager pip to install pspso.

```
pip install pspso
```

CHAPTER

TWO

USAGE

2.1 MLP Example (Binary Classification)

pspso is used to select the machine learning algorithms parameters. Below is an example for using the pspso to select the parameters of the MLP. pspso handles the MLP random weights intialization issue that may cause losing the best solution in consecutive iterations.

The following example demonstrates the selection process of the MLP parameters. A variable named *params* was not given by the user. Hence, the default search space of the MLP is loaded. This search space contains five parameters:

```
params = {"optimizer": ["RMSprop", "adam", "sgd",'adamax','nadam','adadelta'] ,
          "learning_rate":  [0.1,0.3,2],
          "neurons": [1,40,0],
          "hiddenactivation": ['relu','sigmoid','tanh'],
          "activation":['relu','sigmoid','tanh']}
```

The task and the score were defined as *binary classification* and *auc* respectively. Then, the PSO was used to select the parameters of the MLP. Results are provided back to the user through the **print_results()** function.

```
from sklearn.preprocessing import MinMaxScaler
from pspso import pspso
from sklearn import datasets
from sklearn.model_selection import train_test_split

breastcancer = datasets.load_breast_cancer()
data=breastcancer.data#get the breast cancer dataset input features
target=breastcancer.target# target
X_train, X_test, Y_train, Y_test = train_test_split(data, target,test_size=0.1,random_
↪state=42,stratify=target)
normalize = MinMaxScaler(feature_range=(0,1))#normalize input features
X_train=normalize.fit_transform(X_train)
X_test=normalize.transform(X_test)
X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train,test_size=0.15,
↪random_state=42,stratify=Y_train)
p=pspso(estimator='mlp',task='binary classification', score='auc')
pos,cost,duration,model,optimizer=p.fitpspso(X_train,Y_train,X_val,Y_val)
p.print_results()#print the results
testscore=pspso.predict(p.model,p.estimator,p.task,p.score, X_test, Y_test)
print(1-testscore)
```

In this example, four parameters were examined: optimizer, learning_rate, hiddenactivation, and activation. The number of neurons in the hidden layer was kept as default.

Output:

```
Estimator: mlp
Task: binary classification
Selection type: PSO
Number of attempts:50
Total number of combinations: 45360
Parameters:
{'optimizer': 'nadam', 'learning_rate': 0.29, 'neurons': 4, 'hiddenactivation':
↪'sigmoid', 'activation': 'sigmoid'}
Global best position: [3.8997699  0.28725911 4.21218138 1.41200923 0.84643591]
Global best cost: 0.0
Time taken to find the set of parameters: 160.3374378681183
Number of particles: 5
Number of iterations: 10
0.9867724867724867
```

2.2 XGBoost Example (Binary Classification)

```
from sklearn.preprocessing import MinMaxScaler
from pspso import pspso
from sklearn import datasets
from sklearn.model_selection import train_test_split

breastcancer = datasets.load_breast_cancer()
data=breastcancer.data#get the breast cancer dataset input features
target=breastcancer.target# target
X_train, X_test, Y_train, Y_test = train_test_split(data, target,test_size=0.1,random_
↪state=42,stratify=target)
normalize = MinMaxScaler(feature_range=(0,1))#normalize input features
X_train=normalize.fit_transform(X_train)
X_test=normalize.transform(X_test)
X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train,test_size=0.15,
↪random_state=42,stratify=Y_train)

params = {
                "learning_rate":  [0.01,0.2,2],
                "max_depth": [1,10,0],
                "n_estimators": [2,200,0],
                "subsample": [0.7,1,1]}
p=pspso(estimator='xgboost',params=params,task='binary classification', score='auc')
pos,cost,duration,model,optimizer=p.fitpspso(X_train,Y_train,X_val,Y_val)
p.print_results()#print the results
testscore=pspso.predict(p.model,p.estimator,p.task,p.score, X_test, Y_test)
print(1-testscore)
```

2.3 XGBoost Example (Regression)

The XGBoost is an implementation of boosting decision trees. Five parameters were utilized for selection: objective, learning rate, maximum depth, number of estimators, and subsample. Three categorical values were selected for the objective parameter. The learning rate parameter values range between *0.01* and *0.2* with *2* decimal point, maximum depth ranges between *1* and *10* with *0* decimal points *(1,2,3,4,5,6,7,8,9,10)*, etc. The task and score are selected as regression and RMSE respectively. The number of particles and number of iterations can be left as default values if needed. Then, a pspso instance is created. By applying the fitpspso function, the selection process is applied. Finally, results are printed back to the user. The best model, best parameters, score, time, and other details will be saved in the created instance for the user to check.

```
from sklearn.preprocessing import MinMaxScaler
from pspso import pspso
from sklearn import datasets
from sklearn.model_selection import train_test_split

boston_data = datasets.load_boston()
data=boston_data.data
target=boston_data.target

X_train, X_test, Y_train, Y_test = train_test_split(data, target,test_size=0.1,random_
↪state=42)
normalize = MinMaxScaler(feature_range=(0,1))#normalize input features
normalizetarget = MinMaxScaler(feature_range=(0,1))#normalize target

X_train=normalize.fit_transform(X_train)
X_test=normalize.transform(X_test)
Y_train=normalizetarget.fit_transform(Y_train.reshape(-1,1))
Y_test=normalizetarget.transform(Y_test.reshape(-1,1))

X_train, X_val, Y_train, Y_val = train_test_split(X_train, Y_train,test_size=0.25,
↪random_state=42)
params = {
                "objective":['reg:tweedie',"reg:linear","reg:gamma"],
                "learning_rate":  [0.01,0.2,2],
                "max_depth": [1,10,0],
                "n_estimators": [2,200,0],
                "subsample": [0.7,1,1]}
p=pspso(estimator='xgboost',params=params,task='regression', score='rmse')
pos,cost,duration,model,optimizer=p.fitpspso(X_train,Y_train,X_val,Y_val)
p.print_results()#print the results
testscore=pspso.predict(p.model,p.estimator,p.task,p.score, X_test, Y_test)
print(testscore)
```

2.4 User Input

The user is required to select the type of the algorithm ('mlp', 'svm', 'xgboost', 'gbdt'); the task type ('binary classification','regression'), score ('rmse', 'acc', or 'auc'). The user can keep the parameters variable empty, where a default set of parameters and ranges is loaded for each algorithm.

```
from pspso import pspso
task='binary classification'
score='auc'
p=pspso.pspso('xgboost',None,task,score)
```

Pspso allows the user to provide a range of parameters for exploration. The parameters vary between each algorithm. Any parameter supported by the Scikit-Learn API for GBDT and XGBoost can be added to the selection process. A set of parameters that contains five XGBoost parameters is shown below. The parameters are encoded in JSON object that consists of *key,value* pairs:

```
params = {"objective":['reg:tweedie',"reg:linear","reg:gamma"],
                "learning_rate":  [0.01,0.2,2],
                "max_depth": [1,10,0],
                "n_estimators": [2,200,0],
                "subsample": [0.7,1,1]}
```

The key can be any parameter belonging to to the algorithm under investigation. The value is a list. Pspso will check the type of the first element in the list, which will determine if the values of the parameter are categorical or numerical.

Categorical Parameters

If the parameter values are *categorical*, string values are expected to be found in the list, as shown in *objective* parameter. The values in the list will be automatically mapped into a list of integers, where each integer represents a value in the original list. The order of the values inside the list affect the position of the value in the search space.

Numerical Parameters

If the parameter is numerical, a list of three elements [lb,ub, rv] is expected to be found:

- **lb**: repesents the lowest value in the search space
- **ub**: represents the maximum value in the search space
- **rv**: represents the number of decimal points the parameter values are rounded to before being added for training the algorithm

For e.g if you want pspso to select n_estimators, add the following list *[2,200,0]*. By that, the lowest n_estimators will be *2*, the highest to be examined is *200*, and each possible value is rounded to an integer value (*0* decimal points).

Other parameters

The user is given the chance to handle some of the default parameters such as the number of epochs in the MLP. Although this parameter can be optimized, but its not encouraged. The user can modify this by changing a pspso class instance. For e.g., to change the number of epochs from default to 10 in MLP training:

```
from pspso import pspso
task='binary classification'
score='auc'
p=pspso.pspso('mlp',None,task,score)# in case of empty set of params (None) default␣
↪search space is loaded
p.defaultparams['epochs']=10
```

The verbosity can be modified for any algorithm, which allows showing details of the training process:

```
from pspso import pspso
task='binary classification'
score='auc'
p=pspso.pspso('mlp',None,task,score)
p.verbosity=1
```

Early stopping rounds can alos be modified, the user can set a value different to the default value:

```
from pspso import pspso
task='binary classification'
score='auc'
```

(continues on next page)

(continued from previous page)

```
p=pspso.pspso('xgboost',None,task,score)
p.early_stopping=10
```

Other parameters such that n_jobs in XGBoost can also be modified before the start of the selection process.

CHAPTER

THREE

FUNCTIONS

3.1 ML Algorithms Functions

forward_prop_gbdt(particle, task, score, . . .)	Train the GBDT after decoding the parameters in variable particle.
forward_prop_xgboost(particle, task, score, . . .)	Train the XGBoost after decoding the parameters in variable particle.
forward_prop_svm(particle, task, score, . . .)	Train the SVM after decoding the parameters in variable particle.
forward_prop_mlp(particle, task, score, . . .)	Train the MLP after the decoding the parameters in variable particle.

3.2 Selection Functions

fitpspso([X_train, Y_train, X_val, Y_val, . . .])	Select the algorithm parameters based on PSO.
fitpsgrid([X_train, Y_train, X_val, Y_val])	Select the algorithm parameters based on Grid search.
fitpsrandom([X_train, Y_train, X_val, . . .])	Select the algorithm parameters based on radnom search.

The fitpsrandom() and fitpsgrid() were implmented as two default selection methods. With fit random search, the number of attempts to be tried is added by the user as a variable. In grid search, all the possible combinations are created and investigated by the package. These functions follow the same encoding schema used in fitpspso(), and were basically added for comparison.

3.3 Parameters Functions

read_parameters([params, estimator, task])	Read the parameters provided by the user.
decode_parameters(particle)	Decodes the parameters of a list into a meaningful set of parameters.
get_default_params(estimator, task)	Set the default parameters of the estimator.
get_default_search_space(estimator, task)	Create a dictionary of default parameters if the user didnt provide parameters.

3.4 Other Functions

f(q, estimator, task, score, X_train, . . .)	Higher-level method to do forward_prop in the whole swarm.
rebuildmodel(estimator, pos, task, score, . . .)	Used to rebuild the model after selecting the parameters.
print_results()	Print the results found in the pspso instance.
calculatecombinations()	A function that will generate all the possible combinations in the search space.
predict(model, estimator, task, score, . . .)	A function used to release the score of a model.

CHAPTER

FOUR

MODULE SUMMARY

class pspso.**pspso**(*estimator='xgboost'*, *params=None*, *task='regression'*, *score='rmse'*)

This class searches for algorithm parameters by using the Particle Swarm Optimization (PSO) algorithm.

calculatecombinations()

A function that will generate all the possible combinations in the search space. Used mainly with grid search

Returns

combinations: list A list that contains all the possible combinations.

static decode_parameters(*particle*)

Decodes the parameters of a list into a meaningful set of parameters. To decode a particle, we need the following global variables:parameters, defaultparameters, paramdetails, and rounding.

static f(*q*, *estimator*, *task*, *score*, *X_train*, *Y_train*, *X_val*, *Y_val*)

Higher-level method to do forward_prop in the whole swarm.

Inputs

x: numpy.ndarray of shape (n_particles, dimensions) The swarm that will perform the search

Returns

numpy.ndarray of shape (n_particles,) The computed loss for each particle

fitpsgrid(*X_train=None*, *Y_train=None*, *X_val=None*, *Y_val=None*)

Select the algorithm parameters based on Grid search.

Grid search was implemented to match the training process with pspso and for comparison purposes. I have to traverse each value between x_min, x_max. Create a list seperating rounding value.

fitpspso(*X_train=None*, *Y_train=None*, *X_val=None*, *Y_val=None*, *psotype='global'*, *number_of_particles=5*, *number_of_iterations=10*, *options={'c1': 1.49618*, *'c2': 1.49618*, *'w': 0.7298}*)

Select the algorithm parameters based on PSO.

Inputs

X_train: numpy.ndarray of shape (a,b) Contains the training input features, a is the number of samples, b is the number of features

Y_train: numpy.ndarray of shape (a,1) Contains the training target, a is the number of samples

X_train: numpy.ndarray of shape (c,b) Contains the validation input features, c is the number of samples, b is the number of features

Y_train: numpy.ndarray of shape (c,1) Contains the training target, c is the number of samples

number_of_particles: integer number of particles in the PSO search space.

number_of_iterations: integer number of iterations.

options: dictionary A key,value dict of PSO parameters c1,c2, and w

Returns

pos: list The encoded parameters of the best solution

cost: float The score of the best solution

duration: float The time taken to conduct random search.

model: The best model generated via random search

combinations: list of lists The combinations examined during random search

results: list The score of each combination in combinations list

fitpsrandom(*X_train=None*, *Y_train=None*, *X_val=None*, *Y_val=None*, *number_of_attempts=20*)

Select the algorithm parameters based on radnom search.

With Random search, the process is done for number of times specified by a parameter in the function.

Inputs

X_train: numpy.ndarray of shape (a,b) Contains the training input features, a is the number of samples, b is the number of features

Y_train: numpy.ndarray of shape (a,1) Contains the training target, a is the number of samples

X_train: numpy.ndarray of shape (c,b) Contains the validation input features, c is the number of samples, b is the number of features

Y_train: numpy.ndarray of shape (c,1) Contains the training target, c is the number of samples

number_of_attempts: integer The number of times random search to be tried.

Returns

pos: list The encoded parameters of the best solution

cost: float The score of the best solution

duration: float The time taken to conduct random search.

model: The best model generated via random search

combinations: list of lists The combinations examined during random search

results: list The score of each combination in combinations list

static forward_prop_gbdt(*particle*, *task*, *score*, *X_train*, *Y_train*, *X_val*, *Y_val*)

Train the GBDT after decoding the parameters in variable particle. The particle is decoded into parameters of the gbdt. Then, The gbdt is trained and the score is sent back to the fitness function.

Inputs

particle: list of values (n dimensions) A particle in the swarm

task: regression, binary classification the task to be conducted

score: rmse (regression), auc (binary classification), acc (binary classification) the type of evaluation

X_train: numpy.ndarray of shape (m, n) Training dataset

Y_train: numpy.ndarray of shape (m,1) Training target

X_val: numpy.ndarray of shape (x, y) Validation dataset

Y_val: numpy.ndarray of shape (x,1) Validation target

Returns

variable, model the score of the trained algorithm over the validation dataset, trained model

static forward_prop_mlp(*particle*, *task*, *score*, *X_train*, *Y_train*, *X_val*, *Y_val*)
Train the MLP after the decoding the parameters in variable particle.

static forward_prop_svm(*particle*, *task*, *score*, *X_train*, *Y_train*, *X_val*, *Y_val*)
Train the SVM after decoding the parameters in variable particle.

static forward_prop_xgboost(*particle*, *task*, *score*, *X_train*, *Y_train*, *X_val*, *Y_val*)
Train the XGBoost after decoding the parameters in variable particle. The particle is decoded into parameters of the XGBoost. This function is similar to forward_prop_gbdt The gbdt is trained and the score is sent back to the fitness function.

Inputs

particle: list of values (n dimensions) A particle in the swarm

task: regression, binary classification the task to be conducted

score: rmse (regression), auc (binary classification), acc (binary classification) the type of evaluation

X_train: numpy.ndarray of shape (m, n) Training dataset

Y_train: numpy.ndarray of shape (m,1) Training target

X_val: numpy.ndarray of shape (x, y) Validation dataset

Y_val: numpy.ndarray of shape (x,1) Validation target

Returns

variable, model the score of the trained algorithm over the validation dataset, trained model

static get_default_params(*estimator*, *task*)
Set the default parameters of the estimator. This function assigns the default parameters for the user. Each algorithm has a set of parameters. To allow the user to search for some parameters instead of the supported parameters, this function is used to assign a default value for each parameter. In addition, it gets other parameters for each algorithm. For e.g, it returns the number of epochs, batch_size, and loss for the mlp.

Inputs

estimator: string value A string value that determines the estimator: 'mlp','xgboost','svm', or 'gbdt'

task: string value A string value that determines the task under consideration: 'regression' or 'binary classification'

Returns

defaultparams: Dictionary A dictionary that contains default parameters to be used.

static get_default_search_space(*estimator*, *task*)
Create a dictionary of default parameters if the user didnt provide parameters.

Inputs

estimator: string value A string value that determines the estimator: 'mlp','xgboost','svm', or 'gbdt'

task: string value A string value that determines the task under consideration: 'regression' or 'binary classification'

Returns

params: Dictionary A dictionary that contains default parameters to be used.

static predict(*model*, *estimator*, *task*, *score*, *X_val*, *Y_val*)

A function used to release the score of a model. If the score is rmse, the value is released. If the score is acc (accuracy), 1-acc is returned back since pso applies a minimization task. If the score is auc, 1-auc is returned back since pso applies a minization task

This class is static and can be used to test the model accuracy over the hold-out sample once the selection process is finalized.

Inputs

model: A trained model

estimator: string value A string value that determines the estimator: 'mlp','xgboost','svm', or 'gbdt'

task: string value A string value that determines the task under consideration: 'regression' or 'binary classification'

score: string value Determines the score ('rmse','auc','acc')

X_val: numpy.ndarray Input features

Y_val: numpy.ndarray Target

Returns

met: float Score value of the model

print_results()

Print the results found in the pspso instance. Expected to print general details like estimator, task, selection type, number of attempts examined, total number of combinations, position of the best solution, score of the best solution, parameters, details about the pso algorithm.

static read_parameters(*params=None*, *estimator=None*, *task=None*)

Read the parameters provided by the user.

Inputs

params: dictionary of key,values added by the user This dictionary determines the parameters and ranges of parameters the user wants to selection values from.

estimator: string value A string value that determines the estimator: 'mlp','xgboost','svm', or 'gbdt'

task: string value A string value that determines the task under consideration: 'regression' or 'binary classification'

Returns

parameters The parameters selected by the user

defaultparams Default parameters

x_min: list The lower bounds of the parameters search space

x_max: list The upper bounds of the parameters search space

rounding: list The rounding value in each dimension of the search space

bounds: dict A dictionary of the lower and upper bounds

dimensions: integer Dimensions of the search space

params: Dict Dict given by the author

static rebuildmodel(*estimator*, *pos*, *task*, *score*, *X_train*, *Y_train*, *X_val*, *Y_val*)

Used to rebuild the model after selecting the parameters.

CHAPTER

FIVE

FUTURE WORK

5.1 New Algorithms

Other machine learning algorithms and packages will be added such as the catboost.

5.2 Cross Validation

We are working towards adding the cross validation support that will take the training data and number of folds.

Then split the records and train each fold. The average performance of cross-validation will be retuned back to the user.

5.3 Multi-Class Classification

We are also working on adding multi-class classification and data oversampling techniques.

CHAPTER

SIX

STEPS FOR ADDING ANOTHER MACHINE LEARNING ALGORITHM

The main reason behind the development of this package is to facilitate the use of the algorithms with a minimum amount of code required. However, the steps to add an algorithm are followed:

- **Step 1:** Add a condition in the **get_default_search_space()** function to include the new algorithm default search space parameters with upper/lower bounds
- **Step 2:** Add a default search space based on the algorithm and the task (binary classification or regression) to the **get_default_params()** function
- **Step 3:** Create a function **forward_prop_algorithmname()** that accepts parameters (similar to forward_prop_gbdt,forward_prop_svm) and returns two variables: the model and fitness value
- **Step 4:** Add a condition in the function **f()** to forward the task to the function created in Step 3
- **Step 5:** Add a condition in the function **predict()** to allow building the model using the function created in Step 3

CHAPTER SEVEN

CONTRIBUTING

Pull requests are welcome. For major changes, please open an issue first to discuss what you would like to change.

Please make sure to update tests as appropriate.

We are working towards adding the cross validation support that will take the training data and number of folds, then split the records and train each fold. Finally, the average performance is retuned to the user.

We are also working on adding multi-class classification and data oversampling techniques.

CHAPTER

EIGHT

LICENSE

INDEX

C

D

F

G

P

R