JiXi 绩溪

Package for Tuning (1st Generation)

11/04/2023

Background

JiXi, the native city of Chinese Paramount Leader Hu Jintao（胡锦涛）, is what this First Generation Tuning Package is named after. The purpose of this package is to provide a sophisticated framework for Brute Force tuning. *The Criteria of First Generation Tuning is to use Brute Force to train every specified discrete combination.*

The package takes in X and y data for train, validate and test as DataFrame, as well as a dictionary of {hyperparameters name -> string: hyperparameter values as a list}, and autogenerates all combinations of these hyperparameters to be tuned.

JiXi allows tuning combinations to proceed 1) in the order of how they would be, if generated by nested loops, and also 2) randomly shuffled. A third and fourth sophisticated method of tuning order whereby 3) first tune the centre combination of the discrete parameter field, then tune the 'layer' immediately touching that, and then the 'layer' immediately touching the previous layer and 4) to first tune all combinations lying on the horizontal and also the diagonal relative to the centre combination, before going layer by layer.

The advantage of 3) and 4) is that, if used alongside YiLong, can guarantee that each hyperparameter's values will quickly see at least one combination, which allows the Data Scientist to discard certain parameter values clearly producing bad results and save time.

Class

Class	Purpose
`JiXi`	Object that performs brute force tuning with four different order choices

Methods:

Methods	Purpose
JiXi()	Initialisation
read_in_data(train_x, train_y, val_x, val_y, test_x, test_y)	Read in Train Test Split data Parameters: train_x – pd.DataFrame train_y - pd.Series val_x - pd.DataFrame val_y - pd.Series test_x - pd.DataFrame test_y – pd.Series
read_in_model(model, type)	Read in the underlying model class that we want to tune to get optimal parameters for Parameters: model – any model **class** that allows .fit() and .predict() type – str – either "Classification" or "Regression"
set_hyperparameters(parameter_choices)	Read in the different values of each hyperparameters we want to try. Function will automatically generate each combination Parameters: parameter_choices – dict of str:list – str is hyperparameter name (strictly as defined in model class), and list is sorted values of hyperparameter which we want to try out.

`set_non_tuneable_hyperparameters(non_tuneable_hyperparameter_choice)`	Reads in values for non-tuneable hyperparameters (i.e. doesn’t need to clog up the tuning output csv) Parameters: non_tuneable_hyperparameter_choices – dict of str:int
`set_features(ningxiang_output)`	Reads in feature combinations for tuning Parameters: ningxiang_output – dict of tuple:float
`set_tuning_result_saving_address(address)`	Set saving address for tuning output csv Parameters: address – str – does not need to include ‘.csv’
`change_tuning_style(type, seed = None, outer_most_layer = 2, randomise = True)`	Set which type of tuning order to use. ‘a’: as if nested (according to order of dictionary input to set_hyperparameters()) ‘b’: (reset to ‘a’) before random shuffle using inputted seed, or default seed 19421221 ‘c’: (reset to a) before setting to layer by layer order ‘d’: (reset to a) (reset to c) before setting to diag-hor -> layer by layer. Automatically randomised by default seed Parameters: type – str – ‘a’ or ‘b’ or ‘c’ or ‘d’

	seed – int – for ‘b’ and ‘c’ outer_most_layer – the outer most layer for ‘c’ and ‘d’ to actually order for, before remaining are all random randomise – bool – whether or not to randomise ‘c’
`tune(key_stats_only = False)`	Begin tuning process If key_stats_only = True then don’t calculate non important stats Parameters: key_stats_only – bool
`tune_parallel(part, splits, key_stats_only = False)`	Begin tuning process, splitting all combinations into *splits* parts and tune the *part*-th part. If key_stats_only = True then don’t calculate non important stats Parameters: key_stats_only – bool
`read_in_tuning_result_df(address)`	Read in existing DataFrame from .csv consisting of tuning result. Automatically populates result array and checked array if csv columns match parameter choices Parameters: address – str – include ‘.csv’
`set_tuning_best_model_saving_address(address)`	Set address for exporting best model as a pickle

	Parameters: address – str - – does not need to include ‘.pickle’
`view_best_combo_and_score()`	View the current best combination and its validation score

Objects:

Objects	Purpose
`train_x`	DataFrame
`train_y`	Series
`val_x`	DataFrame
`val_y`	Series
`test_x`	DataFrame
`test_y`	Series
`tuning_result`	DataFrame
`model`	model **class**
`parameter_choices`	Dictionary -str:list – str is hyperparameter name (strictly as defined in model class), and list is sorted values of hyperparameter which we want to try out.
`hyperparameters`	list
`feature_n_ningxiang_score_dict`	Dictionary -str:float – str is hyperparameter name (strictly as defined in model class), and float is its NingXiang score
`non_tuneable_parameter_choices`	Dictionary -str:str/float/int - str is hyperparameter name (strictly as defined in model class), and values are valid hyperparameter values for model
`checked`	np.array
`result`	np.array
`tuning_result_saving_address`	str
`best_model_saving_address`	str

best_score = -np.inf	int
best_combo	list
best_clf	model **object**
clf_type	str – ‘Regression’ or ‘Classification’
combos	List of lists
n_items	list - denoting how many values in each hyperparameter dimensions
regression_extra_output_columns = ['Train r2', 'Val r2', 'Test r2', 'Train RMSE', 'Val RMSE', 'Test RMSE', 'Train MAPE', 'Val MAPE', 'Test MAPE', 'Time']	list (pre-setted)
classification_extra_output_columns = ['Train accu', 'Val accu', 'Test accu', 'Train balanced_accu', 'Val balanced_accu', 'Test balanced_accu', 'Train f1', 'Val f1', 'Test f1', 'Train precision', 'Val precision', 'Test precision', 'Train recall', 'Val recall', 'Test recall', 'Time']	list (pre-setted)

Dependencies

```
pandas

numpy

sklearn
```