YangZhou 扬州

Package for Tuning (2nd Generation)

11/04/2023

Background

YangZhou, the native city of Chinese Paramount Leader Jiang Zemin（江泽民）, is what this Second Generation Tuning Package is named after. The purpose of this package is to provide a sophisticated framework for Greedy tuning. *The Criteria of Second Generation Tuning is to not have to train every specified discrete combination to get the optimum discrete result – or close to the optimum.*

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.

Algorithm Description

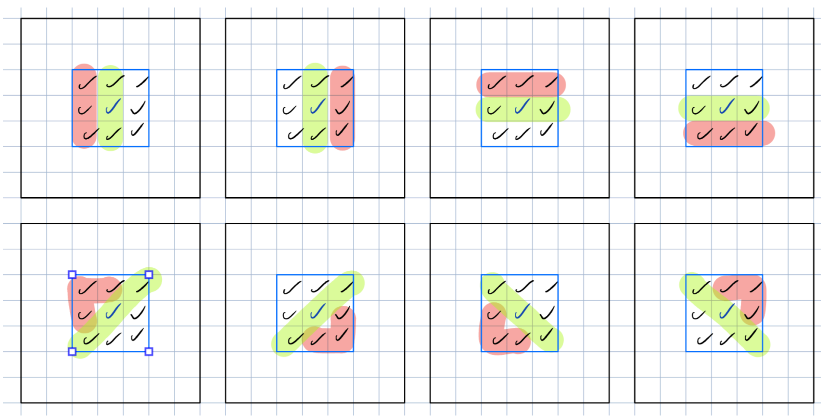
YangZhou begins by train-searching (i.e. searching the field by training the combination) all *Cruise Combinations* (mathematical combinations of all *cruise indices* from each dimension: *cruise indices* being i.e. [0, 4], [0, 5], [0, 3, 6] or [0, 4, 7] for dimensions containing 5, 6, 7 and 8 values respectively. The maximum gap between two indices is 5, minimum is 3).

Then, starting with the median combination (median index of each dimension) as the initial core, the *Guidance Algorithm* is activated, in which the 'square'/'cube'//higher dimension equivalent of (3^d - 1) combinations' surrounding each core (aka *Surrounding Coordinates)* is train-searched.

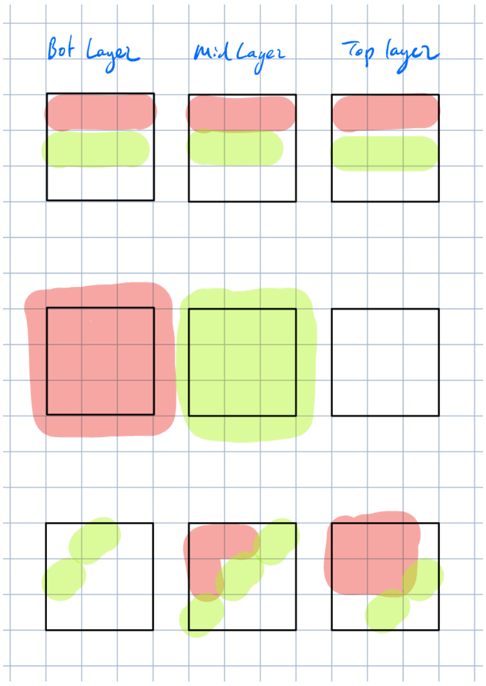
Then, *directions* are identified (see below for further explanation), and an algorithm gathers combinations which are determined to be *treatments* and *nulls (see below for further explanation)* for each *direction*. Then, a <u>two sided welch statistical test</u> (two sampled, assumed different variance) is performed, and for treatments where the directions where the treatment mean is larger than the null mean, and the test p-value <= 0.05:

> The <u>first $n = ceil(d \times \ln(d))$ *combination(s) matching the direction*</u> (see below for further explanation) <u>will be added as new cores</u> to be iteratively train-searched out. (In the case if the 'direction combination' is out of bounds, then the largest scoring combination will replace the 'direction combination' as the new core)

The *Guidance Algorithm* is then repeated for each of the new cores. When no new cores need to be tested, the maximum scoring combination will get a surrounding 3^d train-search (all surrounding (3^d – 1) combinations), and if a new maximum scoring combination is found, then it will also get a surrounding (3^d - 1) combinations search until no new maximum scoring combination can be found. The *Guidance Algorithm* is then terminated.



i.e. 2d



i.e. 3d

The *Cruise algorithm* is then subsequently activated, in which each of the cruise combinations scores will be compared to the current best scoring combination and its surrounding 3^d combination block (including itself). If a cruise combination's score is higher than the

$$warning\ threshold = mean(best\ surrounding\ block) - qt(0.95) * \frac{sd}{\sqrt{3^d}}$$

then the *Guidance Algorithm* will be restarted on that Cruise combination.

The *Cruise Algorithm* terminates once all cruise combinations have been compared to the warning threshold (which could change as the *Cruise Algorithm* goes on)

Once the *Cruise Algorithm* ends, the *Guidance Algorithm* gets activated one more time starting at the current maximum scoring combination, and the whole *YangZhou Algorithm* ends when this call of *Guidance Algorithm* is finished.

Note: although scores of certain combinations will undoubtedly be called upon multiple times, they can be stored and thus the expensive basic operation of train-searching a combination will only ever need to be completed once for each combination.

Further explanations of concepts:

Directions:

Horizontal directions will consist of all combinations that start at [0, 0] and pick one and only one dimension to either be +1 or -1

i.e. for 2d, horizontal directions will consist: [-1, 0], [1, 0], [0, -1], [0, 1]

Diagonal directions will consist of all combinations of all directions being either {-1, 1}

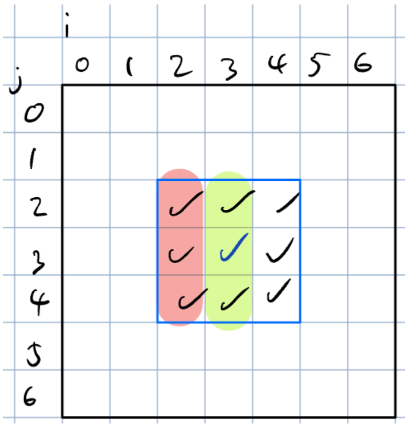
i.e. for 2d, diagonal directions will consist: [-1, -1], [-1, 1], [1, -1], [1, 1]

Horizontal treatment and nulls:

Horizontal treatments will consist of all surrounding combinations with the same indices on the *non 0-valued dimensions* in the *direction* as its *combination matching the direction*.

Horizontal nulls will consist of all surrounding combinations with the same indices on the *non 0-valued dimensions* in the *direction* as its *core*.

i.e. for this example, <u>core</u> is [3, 3]; <u>direction</u> is [-1, 0]; <u>combination matching the direction</u> is [2, 3]; <u>non 0-valued dimension</u> is the 1st dimension (i); <u>horizontal treatments</u> are [2, 2], [2, 3], [2, 4]; <u>horizontal nulls</u> are [3, 2], [3, 3], [3, 4]

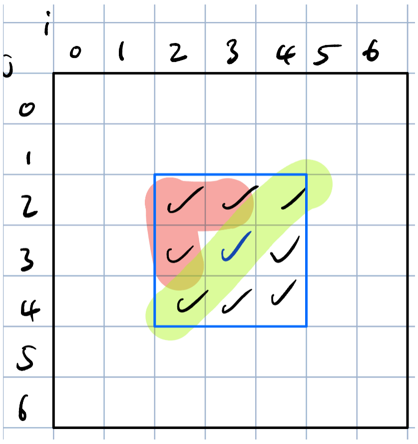


Note: A more intuitive name would be 'Parallel', but 'horizontal' is what has been used in the code.

Diagonal treatments and nulls:

Diagonal treatments will consist of all surrounding combinations that has *relative direction vector* to the core in which every dimension is either exactly the same as that in *direction*, or is set to 0, excluding for the *core* which has *relative direction vector* [0, 0].

Diagonal nulls will consist of the core and all surrounding combinations that has *relative direction vector* to the core which is orthogonal (dot product = 0) to *direction*. This of course includes the *core* which has *relative direction vector* [0, 0] *i.e. for this example, core is [3, 3]; direction is [-1, -1]; combination matching the direction is [2, 2]; diagonal treatments are [2, 2], [2, 3], [3, 2]; diagonal nulls are [2, 4], [3, 3], [4, 2]*



-All these concepts generalise well to higher dimensions

Algorithm Assumptions

1. The scores observed from the same {data, model, hyperparameter combination, split size} belongs to the same underlying population which are normally distributed around a theoretical value**.**

    i.e. Accuracies of SVM on a fixed set of hyperparameters with 80-20 split size on the same set of data (but with random holdouts of 80% training data) is considered to be sampled from the same population (and thereby same distribution)

2. Numerical values/ordinal values of hyperparameters should have trends (may be nonlinear so can't say they are correlated) with model scores. Different hyperparameters may also have interaction.

    => the movement of scores along the space should generally be smooth *(i.e. there won't be one combination with very high score when all close-by combinations have extremely low score)*. Not exactly like a continuous plane/higher-dimensional-structure but the expectation is it behaves like a continuous plane/higher-dimensional-structure

Class

Class	Purpose
`YangZhou`	Object that performs greedy tuning

Methods:

Methods	Purpose
YangZhou()	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'
`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 (for Cruise). 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
`checked_core`	np.array

	value = 1: appended onto list of cores to be checked value = 2: actually checked as a core
`been_cruised`	np.array value = 1: been checked as core, so don't need to be appended as a cruise value = 2: actually checked as a cruise combo
`been_best`	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'
`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',`
	list (pre-setted)

'Val precision', 'Test precision', 'Train recall', 'Val recall', 'Test recall', 'Time']	

Dependencies

`pandas`

`numpy`

`scipy`

`sklearn`

Test Result (Corr)

1. Time

   YangZhou's algorithm will undoubtedly take more time than JiXi on top of the required time for tuning; but from testing, the maximum time required to run YangZhou on a dataset modelled on real data was 13.5435 seconds on Google Colab, which is approximately the time to train one combination for the average model.

   Thus, YangZhou should be a time saver considering the amount of hyperparameter combinations it doesn't need to tune, especially if each hyperparameter combination takes a long time to tune.

2. Accuracy

Batch (Corr)	Percentage of test cases when Algorithm output == Actual Max	Percentage of test cases Algorithm output >= Actual Max – 0.005
1	97.39%	99.96%
2	94.00%	99.63%
3	95.00%	100.00%
7	93.33%	100.00%

Batch	Algorithm output == Actual Max	Algorithm output >= Actual Max – 0.005
Real (4)	95.65%	100.00%

The maximum difference between algorithm output and actual max in batch 4 (real data) was 0.0007.

3. Percentage of Hyperparameter Combinations searched

Batch (Corr)	Mean	Median	Max
1	23.57%	18.34%	88%

2	17.27%	13.17%	71.43%
3	12.12%	6.21%	71.43%
7	14.31%	7.29%	76.67%

Batch	Mean	Median	Max
Real (4)	56.16%	56.81%	74.3%

On average, YangZhou only tunes less than 60% of all designated hyperparameter combinations.

Test Result (Interact)

1. Time

   YangZhou's algorithm will undoubtedly take more time than JiXi on top of the required time for tuning; but from testing, the maximum time required to run YangZhou on a dataset modelled on real data was 13.5435 seconds on Google Colab, which is approximately the time to train one combination for the average model.

   Thus, YangZhou should be a time saver considering the amount of hyperparameter combinations it doesn't need to tune, especially if each hyperparameter combination takes a long time to tune.

2. Accuracy

Batch (Interact)	Percentage of test cases when Algorithm output == Actual Max	Percentage of test cases Algorithm output >= Actual Max – 0.005
1	97.92%	99.65%
2	93.33%	97.5%

Batch	Algorithm output == Actual Max	Algorithm output >= Actual Max – 0.005
Real (3)	95.65%	100.00%

   The maximum difference between algorithm output and actual max in batch 3 (real data) was 0.0007.

3. Percentage of Hyperparameter Combinations searched

Batch (Interact)	Mean	Median	Max
1	31.64%	29.39%	92%
2	18.79%	13.03%	64%

Batch	Mean	Median	Max
Real (3)	56.16%	56.81%	74.3%

On average, YangZhou only tunes less than 60% of all designated hyperparameter combinations.