

pystacked: Stacking generalization and machine learning in Stata

Mark E Schaffer (Heriot-Watt University, IZA)

Achim Ahrens (ETH Zürich)

Christian B Hansen (University of Chicago)

Package website: <https://statalasso.github.io/>

Latest version available [here](#)

2023 Stata Economics Virtual Symposium
November , 2023

Introduction: Stacking

- ▶ The machine learning (ML) toolbox includes a rich set of flexible methods: regularized regression, random forests, SVM, boosting, neural nets.

Introduction: Stacking

- ▶ The machine learning (ML) toolbox includes a rich set of flexible methods: regularized regression, random forests, SVM, boosting, neural nets.
- ▶ When faced with a new prediction or classification task, it is *a priori* rarely obvious which machine learner is best suited for a particular task.
- ▶ *Typical approach:*
 - ▶ Validating learner based on hold-out sample
 - ▶ Cross-validation (K -fold, Leave-one-out, One-step ahead)

The underlying idea: Select *one* learner as the best.

Introduction: Stacking

This approach seems incomplete: combining several different learners could improve performance.

The idea of *stacking generalization*, or simply *stacking*, is to combine learners (Wolpert, 1992; Breiman, 1996). Goes under various names: super learner, model averaging, etc.

Introduction: Stacking

This approach seems incomplete: combining several different learners could improve performance.

The idea of *stacking generalization*, or simply *stacking*, is to combine learners (Wolpert, 1992; Breiman, 1996). Goes under various names: super learner, model averaging, etc.

General idea:

- ▶ Combine a set of “base” (or “level-0”, “candidate”) learners using a “final” (or “level-1”) estimator.
- ▶ It is advisable to include a relatively large and diverse set of base learners to capture different types of pattern in the data.
- ▶ Stacking also provides an effective framework for hyper-parameter tuning.

Introduction: Stata's ML tools

There is a growing number of programs for ML in Stata:

- ▶ `lassopack` for regularized regression (Ahrens, Hansen, and Schaffer, 2020)
- ▶ `rforest` for random forests (Schonlau and Zou, 2020)
- ▶ `svmachines` for support vector machines (Guenther and Schonlau, 2018)
- ▶ Cerulli (2021) and Droste (2020) provide an interface to *scikit-learn* (Pedregosa et al., 2011; Buitinck et al., 2013)
- ▶ `mlrtime` allows Stata users to make use of R's *parsnip* machine learning library (Huntington-Klein, 2021)
- ▶ ...and Stata's own built-in learners

Our contribution: We complement these programs by offering a package that can be used to fit a wide range of machine learners, and for *stacking*.

Introducing pystacked

We introduce `pystacked` for stacking regression and binary classification in Stata.

- ▶ `pystacked` allows to fit multiple machine learning algorithms via Python's *scikit-learn* (Pedregosa et al., 2011; Buitinck et al., 2013)¹ and *combine these into one final prediction* as a weighted average of individual predictions.
- ▶ `pystacked` can also be used to *fit a single machine learner* and thus provides an easy-to-use and versatile API to *scikit-learn's* machine learning algorithms.
- ▶ Our main motivation for developing `pystacked`: Use it in combination with Double-Debiased Machine Learning (Chernozhukov et al., 2018)
- ▶ Forthcoming `pystacked` paper in *The Stata Journal* (working paper version: Ahrens, Hansen, and Schaffer (2022))

¹We stress that `pystacked` relies on *scikit-learn* and the on-going work of the *scikit-learn* contributors. We thus suggest that users cite *scikit-learn* along with this article when using `pystacked`.

Stacking regression

Which machine learner should we use?

We don't know whether we have a sparse or dense problem; linear or non-linear; etc.

Stacking is an ensemble method that combines multiple base learners into one model. As the default, we use *non-negative least squares*:

$$\min_{w_1, \dots, w_J} \sum_{i=1}^n \left(y_i - \sum_{j=1}^J w_j \hat{y}_{(j),i}^{-k(i)} \right)^2 \quad \text{s.t.} \quad w_j \geq 0, \quad \sum_{j=1}^J w_j = 1$$

where $\hat{y}_{(j),i}^{-k(i)}$ are the cross-validated predictions of base learner j .

Voting regression is a special case with unweighted (or user-specified) weights.

Stacking regression

1. *Cross-validation:*

- 1.1 Split the sample $I = \{1, \dots, n\}$ randomly into K partitions ('folds') of approximately equal size. Denote the set of observations in fold $k = 1, \dots, K$ as I_k , and its complement as I_k^c such that $I_k^c = I \setminus I_k$. I_k constitutes the step- k validation set and I_k^c the step- k training sample.

Stacking regression

1. Cross-validation:

- 1.1 Split the sample $I = \{1, \dots, n\}$ randomly into K partitions ('folds') of approximately equal size. Denote the set of observations in fold $k = 1, \dots, K$ as I_k , and its complement as I_k^c such that $I_k^c = I \setminus I_k$. I_k constitutes the step- k validation set and I_k^c the step- k training sample.
- 1.2 For each fold $k = 1, \dots, K$ and each base learner $j = 1, \dots, J$, fit the supervised machine learner j to the training data I_k^c and obtain out-of-sample predicted values $\hat{y}_{(j),i}^{-k}$ for $i \in I_k$.

Stacking regression

1. Cross-validation:

- 1.1 Split the sample $I = \{1, \dots, n\}$ randomly into K partitions ('folds') of approximately equal size. Denote the set of observations in fold $k = 1, \dots, K$ as I_k , and its complement as I_k^c such that $I_k^c = I \setminus I_k$. I_k constitutes the step- k validation set and I_k^c the step- k training sample.
- 1.2 For each fold $k = 1, \dots, K$ and each base learner $j = 1, \dots, J$, fit the supervised machine learner j to the training data I_k^c and obtain out-of-sample predicted values $\hat{y}_{(j),i}^{-k}$ for $i \in I_k$.

2. *Final learner*: Fit the final learner to the full sample. The default choice is NNLS:

$$\min_{w_1, \dots, w_J} \sum_{i=1}^n \left(y_i - \sum_{j=1}^J w_j \hat{y}_{(j),i}^{-k(i)} \right)^2 \quad \text{s.t.} \quad w_j \geq 0, \quad \sum_{j=1}^J w_j = 1$$

The stacking predicted values are defined as $\hat{y}_i^* = \sum_j \hat{w}_j \hat{y}_{(j),i}$ where \hat{w}_j is the estimated stacking weight corresponding to learner j and $\hat{y}_{(j),i}$ are the predicted values from re-fitting learner j on the full sample I .

pystacked overview

pystacked implements stacking regression (Wolpert, 1992) via [scikit learn](#)'s `StackingRegressor` and `StackingClassifier`.

pystacked overview

pystacked implements stacking regression (Wolpert, 1992) via [scikit learn](#)'s `StackingRegressor` and `StackingClassifier`.

Main features:

- ▶ Two alternative syntaxes
- ▶ 10+ different machine learning algorithms supported that can be used stand-alone or as base learners in combination with stacking
- ▶ Regression+classification
- ▶ Graphing and plotting features
- ▶ Supports central *scikit-learn* learn pipelines
- ▶ Supports sparse matrices and parallelization
- ▶ Various options for the final learner (ridge, least squares)

(Base) Machine learners

<code>method()</code>	<code>type()</code>	<i>Machine learner description</i>
<code>ols</code>	<code>regress</code>	Linear regression
<code>logit</code>	<code>class</code>	Logistic regression
<code>lassoic</code>	<code>regress</code>	Lasso with AIC/BIC penalty
<code>lassocv</code>	<code>regress</code>	Lasso with CV penalty
	<code>class</code>	Logistic lasso with CV penalt
<code>ridgecv</code>	<code>regress</code>	Ridge with CV penalty
	<code>class</code>	Logistic ridge with CV penalty
<code>elasticcv</code>	<code>regress</code>	Elastic net with CV penalty
	<code>class</code>	Logistic elastic net with CV
<code>svm</code>	<code>regress</code>	Support vector regression
	<code>class</code>	Support vector classification
<code>gradboost</code>	<code>regress</code>	Gradient boosting regressor
	<code>class</code>	Gradient boosting classifier
<code>rf</code>	<code>regress</code>	Random forest regressor
	<code>class</code>	Random forest classifier
<code>linsvm</code>	<code>class</code>	Linear SVC
<code>nnet</code>	<code>regress</code>	Neural net
	<code>class</code>	Neural net

Note: The first two columns list all allowed combinations of `method(string)` and `type(string)`, which are used to select base learners. Column 3 provides a description of each machine learner. 'CV penalty' indicates that the penalty level is chosen to minimize the cross-validated MSPE. 'AIC/BIC penalty' indicates that the penalty level minimizes either either the Akaike or Bayesian information criterion. SVC refers to support vector classification.

Main syntax

Syntax 1:

```
pystacked depvar predictors [if] [in] [, methods(string)  
cmdopt1(string) cmdopt2(string) ... cmdopt10(string)  
pipe1(string) pipe2(string) ... xvars1(predictors)  
xvars2(predictors) ... general_options ]
```

Notes:

- ▶ `methods(string)` is used to select base learners, where *string* is a list of base learners.
- ▶ Options are passed on to base learners via `cmdopt1(string)`, `cmdopt2(string)`, ...
- ▶ `pipe*(string)` are for pipelines; `xvars*(predictors)` allows to specify a learner-specific variable lists of predictors.

Main syntax

Syntax 2:

```
pystacked depvar [indepvars] || method(string) opt(string)  
pipe(string) xvars(predictors) [ || method(string) opt(string)  
pipe(string) xvars(predictors) ... || ] [if] [in] [ ,  
general_options ]
```

Notes:

Base learners are added before the comma using method(*string*) along with further learner-specific settings and separated by '||'.

Pipelines and learner-specific predictors

Pipelines

scikit-learn uses pipelines to pre-process input data on the fly. In `pystacked`, pipelines can be used to impute missing values, create polynomials and interactions, and to standardize predictors.

Pipelines and learner-specific predictors

Pipelines

scikit-learn uses pipelines to pre-process input data on the fly. In *pystacked*, pipelines can be used to impute missing values, create polynomials and interactions, and to standardize predictors.

Learner-specific predictors

- ▶ By default, *pystacked* uses the same set of predictors for each base learner.
- ▶ This is often not desirable: For example, when using linear machine learners such as the lasso adding polynomials, interactions and other transformations of the base set of predictors might greatly improve out-of-sample prediction performance.
- ▶ *Solution*: Use pipelines or `xvars*(predictors)`

General options I

A full list of general options is provided in the `pystacked` help file. We list only the most important general options here:

`type(string)` allows *reg(ress)* for regression problems or *class(ify)* for classification problems. The default is regression.

`finalest(string)` selects the final estimator used to combine base learners. The default is non-negative least squares without an intercept and the additional constraint that weights sum to 1 (*nnls1*). Alternatives are *nnls0* (non-negative least squares without an intercept and without the sum-to-one constraint), *singlebest* (use the base learner with the minimum MSE), *ls1* (least squares without an intercept and with the sum-to-one constraint), *ols* (ordinary least squares) or *ridge* for (logistic) ridge, which is the *scikit-learn* default.

`foldvars(integer)` specifies the number of folds used for cross-validation. The default is 5.

`foldvar(varname)` is the integer fold variable for cross-validation.

`bfolds(integer)` sets the number of folds used for base learners that use cross-validation (e.g. cross-validated lasso); the default is 5.

General options II

pyseed(*integer*) sets the Python seed. Note that since pystacked uses Python, we also need to set the Python seed to ensure replicability. There are three options:

1. `pyseed(-1)` draws a number between 0 and 10^8 in Stata which is then used as a Python seed; this is pystacked's default behavior. This way, one only needs to deal with the Stata seed. For example, `set seed 42` is sufficient, as the Python seed is generated automatically.
2. Setting `pyseed(x)` with any positive integer x allows to control the Python seed directly.
3. `pyseed(0)` sets the seed to `None` in Python.

njobs(*integer*) sets the number of jobs for parallel computing. The default is 0 (no parallelization), `-1` uses all available CPUs, `-2` uses all CPUs minus 1.

backend(*string*) backend used for parallelization. The default is 'threading'.

voting selects voting regression or classification which uses pre-specified weights. By default, voting regression uses equal weights; voting classification uses a majority rule.

General options III

voteweights(*numlist*) defines positive weights used for voting regression or classification. The length of *numlist* should be the number of base learners – 1. The last weight is calculated to ensure that the sum of weights equals 1.

sparse converts predictor matrix to a sparse matrix. This conversion will only lead to speed improvements if the predictor matrix is sufficiently sparse. Not all learners support sparse matrices and not all learners will benefit from sparse matrices in the same way. One can also use the sparse pipeline to use sparse matrices for some learners but not for others.

printopt prints the default options for specified learners. Only one learner can be specified. This is for information only; no estimation is done.

showopt prints the options passed on to Python.

showpy prints Python messages.

showcoefs (*new*) shows, for each base learner, the coefficient estimates (in the case of ols, logit, regularized regression) or variable importance measures (random forests and gradient boosting).

Demonstration 1: Single base learner

We import the California house price data from Pace and Barry (1997), and split the sample randomly into training and validation partition using a 75/25 split. The aim of the prediction task is to predict median house prices (`medhousevalue`) using a set of house price characteristics

```
. clear all
. use https://statalasso.github.io/dta/cal_housing.dta, clear
. set seed 42
. gen train=runiform()
. replace train=train<.75
(20,640 real changes made)
. replace medh = medh/10e3
variable medhousevalue was long now double
(20,640 real changes made)
```

Demonstration 1: Single base learner

The option `method(gradientboost)` selects gradient boosting. We will later see that we can specify more than one learner in `methods()`, and that we can also fit gradient boosted classification trees.

```
. pystacked medh longi-medi if train, type(reg) methods(gradientboost)
Single base learner: no stacking done.
```

```
Stacking weights:
```

Method	Weight
gradientboost	1.0000000

```
. predict double yhat_gb1 if !train
```

Demonstration 1: Single base learner

The option `method(gradientboost)` selects gradient boosting. We will later see that we can specify more than one learner in `methods()`, and that we can also fit gradient boosted classification trees.

```
. pystacked medh longi-medi if train, type(reg) methods(gradientboost)
Single base learner: no stacking done.
```

```
Stacking weights:
```

Method	Weight
gradientboost	1.0000000

```
. predict double yhat_gb1 if !train
```

The output shows the stacking weights associated with each base learner. Since we only consider one method, the output is not particularly informative and simply shows a weight of one for gradient boosting. Yet, `pystacked` has fitted 100 boosted trees (the default) in the background!

Demonstration 1: Single base learner

We use lasso with cross-validated penalty and display the coefficients using showcoef:

```
. pystacked medh longi-medi if train, type(reg) methods(lassocv) showcoef  
Single base learner: no stacking done.
```

Stacking weights:

Method	Weight
lassocv	1.0000000

Coefficients lasso cv:

Predictor	Value
longitude	-8.4494974
latitude	-8.9253902
houseage	1.4741459
rooms	-1.7358155
bedrooms	4.8881583
population	-3.9955915
households	1.3265723
medinc	7.6752298
_cons	20.7432595

Demonstration 1: Single base learner

`showcoef` displays variable importance for non-parametric learners (if available).

```
. pystacked medh longi-medi if train, type(reg) methods(gradboost) showcoef  
Single base learner: no stacking done.
```

Stacking weights:

Method	Weight
gradboost	1.0000000

Variable importance gradboost:

Predictor	Value
longitude	0.1429768
latitude	0.1454391
houseage	0.0496290
rooms	0.0034422
bedrooms	0.0217799
population	0.0232365
households	0.0092332
medinc	0.6042632

Demonstration 1: Single base learner

Here, we compare lasso with and without the *poly2* pipeline, which creates 2nd-order polynomials and interaction effects:

```
. pystacked medh longi-medi if train, type(reg) methods(lassocv)
Single base learner: no stacking done.
```

Stacking weights:

Method	Weight
lassocv	1.0000000

```
. predict double yhat_lasso1 if !train
```

```
.
```

```
. pystacked medh longi-medi if train, type(reg) methods(lassocv) ///
> pipe1(poly2)
```

```
Single base learner: no stacking done.
```

Stacking weights:

Method	Weight
lassocv	1.0000000

```
. predict double yhat_lasso2 if !train
```

Demonstration 2: Stacking regression

We now consider a stacking regression application with five base learners:

1. linear regression,
2. lasso with penalty chosen by cross-validation,
3. lasso with second order polynomials and interactions,
4. random forest with default settings,
5. gradient boosting with a learning rate of 0.01 and 1000 trees.

Demonstration 2: Stacking regression

Syntax 1:

```
. set seed 42
. pystacked medh longi-medi if train,          ///
>   type(regress)                            ///
>   methods(ols lasso cv lasso cv rf gradboost)  ///
>   pipe3(poly2) cmdopt5(learning_rate(0.01)    ///
>   n_estimators(1000))
```

Stacking weights:

Method	Weight
ols	0.0000000
lasso cv	0.0000000
lasso cv	0.0000000
rf	0.8382714
gradboost	0.1617286

Demonstration 2: Stacking regression

Syntax 2:

```
. set seed 42
. pystacked medh longi-medi          || ///
>   m(ols)                          || ///
>   m(lassocv)                      || ///
>   m(lassocv) pipe(poly2)         || ///
>   m(rf)                           || ///
>   m(gradboost) opt(learning_rate(0.01) n_estimators(1000))  ///
>   if train, type(regress)
```

Stacking weights:

Method	Weight
ols	0.0000000
lassocv	0.0000000
lassocv	0.0000000
rf	0.8382714
gradboost	0.1617286

Demonstration 2: Stacking regression

pystacked supports alternative final estimators. Here, we select the best-performing individual candidate learner.

```
. set seed 42
. pystacked medh longi-medi if train,          ///
>   type(regress)                            ///
>   methods(ols lasso cv lasso cv rf gradboost)  ///
>   pipe3(poly2) cmdopt5(learning_rate(0.01)   ///
>   n_estimators(1000)) finalest(singlebest)    ///
```

Stacking weights:

Method	Weight
ols	0.0000000
lasso cv	0.0000000
lasso cv	0.0000000
rf	1.0000000
gradboost	0.0000000

Demonstration 2: Stacking regression

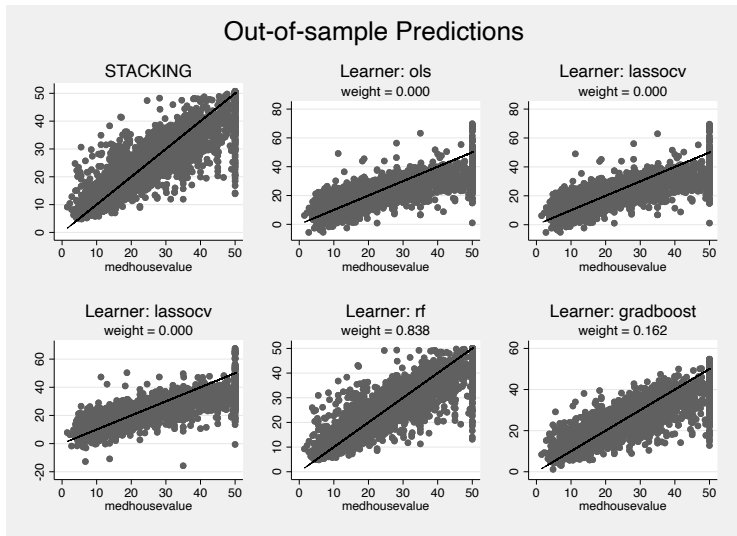
Predicted values. In addition to the stacking predicted values, we can also get the predicted values of each base learner using the `basexb` option:

```
. predict double yhat, xb  
  
. predict double ybase, basexb  
  
. list yhat ybase* if _n <= 5
```

	yhat	ybase1	ybase2	ybase3	ybase4	ybase5
1.	42.875233	41.315834	41.193417	40.02005	43.160831	41.394928
2.	38.736641	41.45306	41.421459	44.443344	38.289108	41.05629
3.	40.683311	38.212036	38.154835	37.796288	41.268902	37.648071
4.	33.559086	32.332497	32.266319	32.546425	33.4869	33.933231
5.	24.190615	25.382839	25.369064	25.269325	23.859699	25.905815

Demonstration 2: Stacking regression

Plotting. The graph option creates a scatter plot of predicted versus observed values for stacking and each base learner:



Demonstration 2: Stacking regression

MSPE table. The table option allows to compare stacking weights with in-sample and out-of-sample MSPE. As with the graph option, we can use table as a post-estimation command:






```
. pystacked, table holdout  
Number of holdout observations: 5192  
RMSPE: In-Sample, CV, Holdout
```

Method	Weight	In-Sample	CV	Holdout
STACKING	.	2.313	4.980	4.939
ols	0.000	6.986	7.008	6.853
lassocv	0.000	6.987	7.008	6.857
lassocv	0.000	6.696	6.699	6.606
rf	0.838	1.847	5.001	4.963
gradboost	0.162	5.312	5.523	5.511





Summary

- ▶ `pystacked` implements *stacked generalization* (Wolpert, 1992) for regression and binary classification via Python's *scikit-learn*.
- ▶ Stacking combines multiple supervised machine learners—the “base” or “level-0” learners—into a single learner.
- ▶ The currently *supported (base) machine learners* include regularized and unregularized regression, random forest, gradient boosting, support vector machines and feed-forward neural nets (multi-layer perceptron).
- ▶ `pystacked` can also be used with as a ‘regular’ machine learning program to fit a single base learner and, thus, provides an easy-to-use API for *scikit-learn*'s machine learning algorithms.





References I

-  Ahrens, Achim, Christian B. Hansen, and Mark E. Schaffer (2020). “lassopack: Model selection and prediction with regularized regression in Stata”. In: *The Stata Journal* 20.1, pp. 176–235. URL: <https://doi.org/10.1177/1536867X20909697>.
-  — (2022). *pystacked: Stacking generalization and machine learning in Stata*. Forthcoming in *The Stata Journal*. URL: <https://arxiv.org/abs/2208.10896>.
-  Breiman, Leo (July 1996). “Stacked regressions”. en. In: *Machine Learning* 24.1, pp. 49–64. URL: <http://link.springer.com/10.1007/BF00117832> (visited on 12/04/2021).
-  Buitinck, Lars et al. (2013). “API design for machine learning software: experiences from the scikit-learn project”. In: *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pp. 108–122.
-  Cerulli, Giovanni (2021). *Machine Learning using Stata/Python*.

References II

-  Chernozhukov, Victor et al. (2018). “Double/debiased machine learning for treatment and structural parameters”. In: *The Econometrics Journal* 21.1. tex.ids= Chernozhukov2018a publisher: John Wiley & Sons, Ltd (10.1111), pp. C1–C68. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1111/ectj.12097>.
-  Droste, Michael (2020). *pylearn*. <https://github.com/mdroste/stata-pylearn/>. [Online; accessed 02-December-2021].
-  Guenther, Nick and Matthias Schonlau (Nov. 2018). *SVMACHINES: Stata module providing Support Vector Machines for both Classification and Regression*. Statistical Software Components, Boston College Department of Economics. URL: <https://ideas.repec.org/c/boc/bocode/s458564.html>.
-  Huntington-Klein, Nick C. (2021). *mlrtime*. <https://github.com/NickCH-K/MLRtime/>. [Online; accessed 02-December-2021].

References III

-  Pace, R. Kelley and Ronald Barry (1997). “Sparse spatial autoregressions”. In: *Statistics & Probability Letters* 33.3, pp. 291–297. URL: <https://www.sciencedirect.com/science/article/pii/S016771529600140X>.
-  Pedregosa, F. et al. (2011). “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12, pp. 2825–2830.
-  Schonlau, Matthias and Rosie Yuyan Zou (2020). “The random forest algorithm for statistical learning”. In: *The Stata Journal* 20.1, pp. 3–29. URL: <https://doi.org/10.1177/1536867X20909688>.
-  Wolpert, David H. (1992). “Stacked generalization”. In: *Neural Networks* 5.2, pp. 241–259. URL: <https://www.sciencedirect.com/science/article/pii/S0893608005800231>.