ddml: Double/debiased machine learning in Stata

Mark E Schaffer (Heriot-Watt University, IZA)
Achim Ahrens (ETH Zürich)
Christian B Hansen (University of Chicago)
Thomas Wiemann (University of Chicago)

Package website: https://statalasso.github.io/
Latest version available here

2023 Stata Economics Virtual Symposium
November 9, 2023
Introduction

A rich and growing literature exploits machine learning to facilitate causal inference.

**A central focus: high-dimensional** controls and/or instruments, which can arise if

- we observe many controls/instruments
- controls/instruments enter through an unknown function

Belloni, Chernozhukov, and Hansen (2014) and Belloni et al. (2012) propose estimators *relying on the Lasso* that allow for high-dimensional controls/instruments.

⇒ Available via *pdslasso* in Stata (Ahrens, Hansen, and Schaffer, 2020)
Introduction

What if we don’t want to use the lasso?

- The Lasso might not be the best-performing machine learner for a particular problem.
- The Lasso relies on the approximate sparsity assumption, which might not be appropriate in some settings.

Chernozhukov et al. (2018) propose Double/Debiased Machine Learning (DDML or sometimes "Double ML") which allow to exploit machine learners other than the Lasso.

Our contribution:

- We introduce ddm1, which implements DDML for Stata.
- We provide simulation evidence on the finite sample performance of DDML.
- Our recommendation is to use DDML in combination with Stacking.
Motivating example. The partially-linear model:

\[ y_i = \theta d_i + g(x_i) + \varepsilon_i. \]

How do we account for confounding factors \( x_i \)? — The standard approach is to assume linearity \( g(x_i) = x_i' \beta \) and consider alternative combinations of controls.

Problems:
- Non-linearity & unknown interaction effects
- High-dimensionality: we might have “many” controls
- We don’t know which controls to include
Background

Motivating example. The partially-linear model:

\[ y_i = \theta d_i + g(x_i) + \varepsilon_i. \]

Post-double selection (Belloni, Chernozhukov, and Hansen, 2014) and post-regularization (Chernozhukov, Hansen, and Spindler, 2015) provide data-driven solutions for this setting.

Both “double” approaches rely on the sparsity assumption and use two auxiliary lasso regressions: \( y_i \rightsquigarrow x_i \) and \( d_i \rightsquigarrow x_i \).

Related approaches exist for optimal IV estimation (Belloni et al., 2012) and/or IV with many controls (Chernozhukov, Hansen, and Spindler, 2015).
Background

These methods have been implemented for Stata in `pdslasso` (Ahrens, Hansen, and Schaffer, 2020), `dsregress` (StataCorp) and R (`hdm`; Chernozhukov, Hansen, and Spindler, 2016).

**Example 1:**

```stata
. clear
. use https://statalasso.github.io/dta/AJR.dta
. pdslasso logpgp95 avexpr ///
   (lat_abst edes1975 avelf temp* humid* steplow-oilres)
```

Variables in parentheses are treated as high-dimensional controls. The lasso selects from them.
Background

These methods have been implemented for Stata in pdslasso (Ahrens, Hansen, and Schaffer, 2020), dsregress (StataCorp) and R (hdm; Chernozhukov, Hansen, and Spindler, 2016).

Example 2:

Select controls, but specify that logem4 is an unpenalized instrument (using partial(logem4)).

```
   . ivlasso logpgp95 (avexpr=logem4) ///
      (lat_abst edes1975 avelf temp* humid* steplow-oilres), ///
      partial(logem4)
```
Background

There are **advantages** of relying on lasso:

▶ intuitive assumption of (approximate) sparsity
▶ computationally relatively cheap (due to plugin lasso penalty; no cross-validation needed)
▶ Linearity has its advantages (e.g. extension to fixed effects; Belloni et al., 2016)

But there are also **drawbacks**:

▶ What if the sparsity assumption is not plausible?
▶ There is a wide set of machine learners at disposable—Lasso might not be the best choice.
▶ Lasso requires careful feature engineering to deal with non-linearity & interaction effects.

⇒ **DDML** (Chernozhukov et al., 2018)
Review of DDML

The partially-linear model:

\[ Y = \theta_0 D + g_0(X) + U \]
\[ D = m_0(X) + V \]

**Naive idea:** We estimate conditional expectation functions (CEFs) \( \ell_0(X) = E[Y|X] \) and \( m_0(X) = E[D|X] \) using ML and partial out the effect of \( X \) (in the style of Robinson, 1988):

\[
\hat{\theta}_{DDML} = \left( \frac{1}{n} \sum_i \hat{V}_i^2 \right)^{-1} \frac{1}{n} \sum_i \hat{V}_i (Y_i - \hat{\ell}),
\]

where \( \hat{V} = D - \hat{m}_i \).
Review of DDML

Yet, there is a problem:

▶ The estimation error of the first step (CEF estimation) may spill-over to the second step (estimation of structural parameters).

▶ For example, the estimation error $\ell(x_i) - \hat{\ell}$ and $v_i$ may be correlated due to over-fitting, leading to poor finite sample performance (*own-observation bias*).

DDML relies on two ingredients:

1. cross-fitting: sample splitting with swapped samples
2. Neyman-orthogonal scores: score functions which are robust to small perturbations
Cross-fitting for the partially-linear model (DML 2)

Split the sample \( \{(Y_i, D_i, X_i)\}_{i=1}^n \) randomly in \( K \) folds of approximately equal size. Denote \( I_k \) the set of observations included in fold \( k \) and \( I_k^c \) its complement.

1. For each \( k \in \{1, \ldots, K\} \):
   1.1 Fit a CEF estimator to the sub-sample \( I_k^c \) using \( Y_i \) as the outcome and \( X_i \) as predictors. Obtain the out-of-sample predicted values \( \hat{l}_{I_k}^c(X_i) \) for \( i \in I_k \).
   1.2 Fit a CEF estimator to the sub-sample \( I_k^c \) using \( D_i \) as the outcome and \( X_i \) as predictors. Obtain the out-of-sample predicted values \( \hat{m}_{I_k}^c(X_i) \) for \( i \in I_k \).

2. Compute

\[
\hat{\theta}_n = \frac{1}{n} \sum_{i=1}^{n} \left( Y_i - \hat{l}_{I_k}^c(X_i) \right) \left( D_i - \hat{m}_{I_k}^c(X_i) \right) \frac{1}{n} \sum_{i=1}^{n} \left( D_i - \hat{m}_{I_k}^c(X_i) \right)^2.
\]  

(1)
The importance of cross-fitting: An MC illustration

DDML+ learner (orange) does almost as well as the oracle (green). Learner with no cross-fitting (blue) is biased. (Learner (a) is gradient-boosted trees; Learner (b) is neural net.)

Notes: Figures (a) and (b) compare the bias of the oracle estimator (which knows the true data-generating process) and gradient-boosted trees with and without sample splitting. We generate 1'000 samples of size $n = 1000$ using the partially-linear model $Y_i = \theta_0 D_i + g(X_i) + \varepsilon_i$, $D_i = g(X_i) + u_i$ where the nuisance function is $g(X_i) = \mathbb{1}\{X_{i1} > 0.3\}\mathbb{1}\{X_{i2} > 0\}\mathbb{1}\{X_{i3} > -1\}$. Gradient boosting uses 1200 trees, a maximum tree depth of 6, a learning rate of 0.1, and early stopping with 20% validation sample.
Remarks

Remark 1: Number of folds.

- The number of cross-fitting folds $K$ is a necessary tuning choice. Theoretically, any finite value is admissible.

- Based on our simulation experience, we find that more folds tends to lead to better performance, especially when the sample size is small.
Remarks

Remark 2: Cross-fitting repetitions.

We recommend running the cross-fitting procedure more than once using different random folds to assess randomness introduced via the sample splitting.

Let $\hat{\theta}^{(r)}_n$ denote the DDML estimate from the $r$th cross-fit repetition and $\hat{s}^{(r)}_n$ its associated standard error estimate with $r = 1, \ldots, R$:

$$\tilde{\hat{\theta}}_n = \text{median}\left(\left(\hat{\theta}^{(r)}_n\right)^R_{r=1}\right)$$

$$\tilde{\hat{s}}_n = \sqrt{\text{median}\left(\left(\left(\hat{s}^{(r)}_n\right)^2 + (\hat{\theta}^{(r)}_n - \tilde{\hat{\theta}}_n)^2\right)^R_{r=1}\right)}.$$

ddml facilitates this using the rep(integer) options.
## DDML models

The DDML framework can be applied to other models (all implemented in \texttt{ddml}): 

### Interactive model

$$Y = g_0(D, X) + U$$  \hspace{1cm} (2)

where $D$ is a scalar binary variable and that $D$ is not required to be additively separable from the controls $X$. In this setting, the parameters of interest are

$$\theta_0^{\text{ATE}} \equiv E[g_0(1, X) - g_0(0, X)]$$

$$\theta_0^{\text{ATET}} \equiv E[g_0(1, X) - g_0(0, X)|D = 1],$$  \hspace{1cm} (3)

which correspond to the \textit{average treatment effect} (ATE) and \textit{average treatment effect on the treated} (ATET), respectively.
DDML models

The DDML framework can be applied to other models (all implemented in ddml):

Partially-linear IV model

\[ Y = \theta_0 D + g_0(X) + U, \]

where we leverage instrumental variables \( Z \) for identification.

Let \( \ell_0(X) \equiv E[Y|X] \), \( m_0(X) \equiv E[D|X] \), and \( r_0(X) \equiv E[Z|X] \).

We assume \( E[Cov(U, Z|X)] = 0 \) and \( E[Cov(D, Z|X)] \neq 0 \), and consider the score function

\[
\psi(W; \theta, \ell, m, r) = \left( Y - \ell(X) - \theta(D - m(X)) \right) \left( Z - r(X) \right),
\]

where \( W \equiv (Y, D, X, Z) \).
The DDML framework can be applied to other models (all implemented in ddml):

### Flexible Partially-Linear IV Model

\[
Y = \theta_0 D + g_0(X) + U,
\]

where we leverage instrumental variables \( Z \) for identification.

Let \( p_0(Z, X) \equiv E[D|Z, X] \).

We assume \( E[U|Z, X] = 0 \) and \( E[Var(E[D|Z, X]|X)] \neq 0 \), and consider the score function

\[
\psi(W; \theta, \ell, m, p) = \left( Y - \ell(X) - \theta(D - m(X)) \right) \left( p(Z, X) - m(X) \right).
\]

The Flexible Partially-Linear IV Model allows for approximation of optimal instruments.
DDML models

The DDML framework can be applied to other models (all implemented in ddml):

**Interactive IV model**

\[ Y = g_0(D, X) + U \]

where \( D \) takes values in \( \{0, 1\} \). The parameter of interest we target is the *local average treatment effect* (LATE)

\[ \theta_0 = E[g_0(1, X) - g_0(0, X) | p_0(1, X) > p_0(0, X)] , \quad (4) \]

where \( p_0(Z, X) \equiv \Pr(D = 1 | Z, X) \).
The choice of machine learner

Which machine learner should we use?

ddml supports a range of ML programs: pylearn, lassopack, randomforest. — Which one should we use?

We don’t know whether we have a sparse or dense problem; linear or non-linear. We don’t know whether, e.g., lasso or random forests will perform better.

Stacking, as implemented in pystacked, provides a solution: We use an ‘optimal’ combination of base learners.
The choice of machine learner

Which machine learner should we use?

The choice of CEF estimator can make a huge difference.

Left: the non-linear learner struggles with the linear DGP.
Right: the linear learner struggles with the non-linear DGP.

Notes: Figures (a) and (b) compare the bias of the oracle estimator (which knows the true data-generating process), cross-validated lasso and gradient-boosted trees under two alternative data-generating processes. We generate 1'000 samples of size $n = 1000$ using the partially-linear model $Y_i = \theta_0 D_i + g(X_i) + \epsilon_i$, $D_i = g(X_i) + u_i$ where the nuisance function is either $g(X_i) = \sum_j 0.9^j X_{ij}$ (linear) or $g(X_i) = \mathbb{1}\{X_{i1} > 0.3\} \mathbb{1}\{X_{i2} > 0\} \mathbb{1}\{X_{i3} > -1\}$ (non-linear DGP). Gradient boosting uses 1000 trees, a learning rate of 0.01 and early stopping with 20% validation sample. See Ahrens et al. (2023, Section 4.2) for details.
The choice of machine learner

Which machine learner should we use?

We have already seen one answer: stacking.

DDML + stacking involves two layers of re-sampling:

1. **Cross-fitting (upper) layer:** Divide the sample into $K$ cross-fitting folds. In each cross-fitting step $k \in \{1, \ldots, K\}$, the stacking learner is trained on the training sample $T_k \equiv I \setminus I_k$.

2. **Cross-validation (lower) layer:** Fitting the stacking learner requires subdividing the training sample $T_k$ further into $V$ cross-validation folds. We denote the cross-validation folds by $T_{k,1}, \ldots, T_{k,V}$.

A DDML-specific variant: ‘pooled stacking’, i.e., stack once at the end to get a single stacked learner (a single set of stacking weights instead of $K$ sets of weights).
The choice of machine learner

1. Split sample into $K$ cross-fitting folds (here $K = 5$).

2. For each $k$, define stacking training sample $T_k \equiv I \setminus I_k$, and split into $V$ folds (here $V = 3$).

3. For each $(k, v, j)$, fit base learner $j$ on $T^c_{k,v} \equiv T_k \setminus T_{k,v}$ and obtain out-of-sample predicted values $\hat{\ell}^{(j)}_{T^c_{k,v}}(X_i)$ for $i \in T_{k,v}$.

4. For each $k$, fit $Y$ against $\hat{\ell}^{(1)}_{T^c_k}(X_i), \ldots, \hat{\ell}^{(J)}_{T^c_k}(X_i)$ with $i \in T_k$ to obtain stacking weights $\hat{w}_{k,j}$. Obtain out-of-sample predicted values as $\sum_j \hat{w}_{k,j} \hat{\ell}^{(j)}_{T_k}$ for $i \in I_k$. 

\[ T_k,1 \quad T_k,2 \quad T_k,3 \]

Learner $j = 1$

\[ j = 2 \]

\[ j = 3 \]
The choice of machine learner

**Short-stacking** takes a short-cut and is computationally much cheaper. The final learner is fit on the cross-fitted predicted values.

1. Split sample into $K$ cross-fitting folds (here $K = 5$).

2. For each $(k, j)$, fit learner $j$ on the training sample $j$ and obtain cross-fitted values as $\hat{\ell}_k^{(j)}(X_i)$ for $i \in I_k$.

3. Use final learner to fit $Y$ against $\hat{\ell}_k^{(1)}(X_i), \ldots, \hat{\ell}_k^{(J)}(X_i)$ on full sample, obtain short-stacking weights $\hat{w}_j$ and cross-fitted short-stacked values as $\sum_j \hat{w}_j \hat{\ell}_k^{(j)}(X_i)$. 

---

$1 \quad I_1 \quad I_2 \quad I_3 \quad I_4 \quad I_5$

$2 \quad$ For each $(k, j)$, fit learner $j$ on the training sample $j$ and obtain cross-fitted values as $\hat{\ell}_k^{(j)}(X_i)$ for $i \in I_k$.

$3 \quad$ Use final learner to fit $Y$ against $\hat{\ell}_k^{(1)}(X_i), \ldots, \hat{\ell}_k^{(J)}(X_i)$ on full sample, obtain short-stacking weights $\hat{w}_j$ and cross-fitted short-stacked values as $\sum_j \hat{w}_j \hat{\ell}_k^{(j)}(X_i)$. 

The **ddml** package

*We introduce* **ddml** for Stata:

- Compatible with various ML programs in Stata (e.g. lassopack, pylearn, randomforest).
  → *Any* program with the classical “`reg y x`” syntax and post-estimation `predict` will work.

- Short (one-line) and flexible multi-line version

- Five models supported: partially-linear model, interactive model, interactive IV model, partially-linear IV model, flexible partially-linear IV.

- **ddml** supports data-driven combinations of multiple machine learners via stacking by leveraging `pystacked` (Ahrens, Hansen, and Schaffer, 2022; Pedregosa et al., 2011; Buitinck et al., 2013).

- Standard stacking, short-stacking, pooled stacking all supported.

- Forthcoming **ddml** paper in *The Stata Journal* (working paper version: Ahrens, Hansen, and Schaffer (2022)).
Extended ddml syntax

**Step 1:** Initialize ddml and select model.

```
ddml init model [, kfolds(integer) fcluster(varname)
    foldvar(varlist) reps(integer) mname(name) prefix ]
```

where `model` is partial, interactive, iv, fiv, or interactiveiv.

The `reps` option repeats the estimation for the specified number of different random cross-fit splits. In this case ddml will report the median or mean estimated coefficient(s) of interest across resamples.

**Step 2:** Add ML programs for estimating conditional expectations.

```
ddml cond_exp : command depvar vars [, cmdopt ]
```

where `cond_exp` selects the conditional expectation to be estimated by the machine learning program `command`. `command` is a ML program that supports the standard `reg y x`-type syntax. `cmdopt` are specific to that program.

Multiple estimation commands per equation are allowed.
## Extended ddml syntax

<table>
<thead>
<tr>
<th>cond_exp</th>
<th>partial</th>
<th>interactive</th>
<th>iv</th>
<th>fiv</th>
<th>late</th>
</tr>
</thead>
<tbody>
<tr>
<td>E[Y</td>
<td>X]</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>E[Y</td>
<td>X,D]</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E[Y</td>
<td>X,Z]</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>E[D</td>
<td>X]</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>E[D</td>
<td>Z,X]</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>E[Z</td>
<td>X]</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Table: The table lists the conditional expectations which need to be specified for each model.
Extended `ddml` syntax

**Step 3:** Cross-fitting.

This step implements the cross-fitting algorithm (the most time-consuming step).

```
ddml crossfit [, mname(name) shortstack poolstack
    nostdstack finalest(name) ]
```

Standard stacking and pooled-stacking rely on `ddml`'s `pystacked` integration; short-stacking is available with all learners.

**Step 4:** Estimation of causal effects

In the last step, we estimate the parameter of interest for all combination of learners added in Step 2.

```
ddml estimate [, mname(name) robust cluster(varname)
    vce(vcetype) att trim spec(string) rep(string) ]
```
Quick syntax: \texttt{qddml}

Syntax for Partially-Linear and Interactive Model

\texttt{qddml \ depvar \ treatment\_vars \ (controls), \}
\texttt{model(partial\|interactive) \ [ \ options \ ]}

Syntax for IV models

\texttt{qddml \ depvar \ (controls) \ (treatment\_vars=excluded\_instruments), \}
\texttt{model(iv\|late\|fiv) \ [ \ options \ ]}

where \texttt{ddml\_options} options are internally passed to the \texttt{ddml} subroutines.

We illustrate with a \texttt{qddml} at the end of this presentation.
Simple `ddml` example

We demonstrate the use of `ddml` using the partially-linear model by extending the analysis of 401(k) eligibility and total financial wealth of Poterba, Venti, and Wise (1995). The data consists of $n = 9915$ households from the 1991 SIPP.

In this simple example, we use two learners, OLS and cross-validated lasso. This gives us 4 possible combinations of learners for $Y$ and $D$; `ddml` will report all 4 and the minimum-MSE specification in detail.

**Step 0:** Load data, define globals

```stata
. use "sipp1991.dta", clear
. global Y net_tfa
. global X age inc educ fsize marr twoearn db pira hown
. global D e401
```

**Step 1:** Initialise `ddml` and select model:

```stata
. set seed 123
. ddml init partial, k folds(4)
```
Simple ddml example (cont’d.)

**Step 2:** Add supervised ML programs for estimating conditional expectations. We used pystacked as the front-end for sklearn.linear_model.LassoCV.

. *** add learners for E[Y|X]
. ddml E[Y|X]: reg $Y $X
Learner Y1_reg added successfully.

. ddml E[Y|X]: pystacked $Y c.($X)#c.($X), type(reg) m(lassocv)
Learner Y2_pystacked added successfully.

. *** add learners for E[D|X]
. ddml E[D|X]: reg $D $X
Learner D1_reg added successfully.

. ddml E[D|X]: pystacked $D c.($X)#c.($X), type(reg) m(lassocv)
Learner D2_pystacked added successfully.

**Step 3:** Cross-fitting with 4 folds

. ddml crossfit
Cross-fitting E[y|X] equation: net_tfa
Cross-fitting fold 1 2 3 4 ...completed cross-fitting
Cross-fitting E[D|X] equation: e401
Cross-fitting fold 1 2 3 4 ...completed cross-fitting
Simple ddml example (cont’d.)

**Step 4: Estimation of causal effects**

```
.ddml estimate, robust allcombos
```

**Model:**
```
partial, crossfit folds k=4, resamples r=1
```

**Mata global (mname):**
```
m0
```

**Dependent variable (Y):**
```
net_tfa
```

**net_tfa learners:**
```
Y1_reg Y2_pystacked
```

**D equations (1):**
```
e401
```

**e401 learners:**
```
D1_reg D2_pystacked
```

**DDML estimation results:**

<table>
<thead>
<tr>
<th>spec</th>
<th>r</th>
<th>Y learner</th>
<th>D learner</th>
<th>b</th>
<th>SE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>Y1_reg</td>
<td>D1_reg</td>
<td>5986.657</td>
<td>(1523.694)</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>Y1_reg</td>
<td>D2_pystacked</td>
<td>9563.875</td>
<td>(1389.172)</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>Y2_pystacked</td>
<td>D1_reg</td>
<td>9175.519</td>
<td>(1371.065)</td>
</tr>
<tr>
<td>*</td>
<td>4</td>
<td>Y2_pystacked</td>
<td>D2_pystacked</td>
<td>9788.291</td>
<td>(1339.797)</td>
</tr>
</tbody>
</table>

* = minimum MSE specification for that resample.

**Min MSE DDML model**

```
y-E[y|X] = y-Y2_pystacked_1
D-E[D|X] = D-D2_pystacked_1
```

**Number of obs** = 9915

| net_tfa | Coefficient | Robust std. err. | z  | P>|z| | [95% conf. interval] |
|---------|-------------|-----------------|----|-----|---------------------|
| e401    | 9788.291    | 1339.797        | 7.31 | 0.000 | 7162.337 - 12414.24 |
| _cons   | 90.93481    | 534.8139        | 0.17 | 0.865 | -957.2813 - 1139.151 |
Extended ddml example

We use the same dataset and model as before, but employ stacking with a wider range of learner. pystacked does the standard stacking; ddml does the short-stacking and pooled stacking.

We could ask for all versions of stacking at the cross-fitting stage. Instead, for illustration purposes, we first estimate using only standard stacking and then re-stack to get the short-stacking and pooled stacking results (re-stacking is very fast).

**Step 0:** Load data, define globals

```
. use "sipp1991.dta", clear
. global Y net_tfa
. global X age inc educ fsize marr twoearn db pira hown
. global D e401
```

**Step 1:** Initialise ddml and select model:

```
. set seed 123
. ddml init partial, kfold(4)
warning - model m0 already exists
all existing model results and variables will
```
Extended `ddml` example (cont’d.)

**Step 2:** Add supervised ML programs for estimating conditional expectations.

. *** add learners for $E[Y|X]$
  . `ddml E[Y|X]: pystacked $Y \text{ }$ $X`  
    > method(ols)  
    > m(lassocv) xvars(c.($X)##c.($X))  
    > m(ridgecv) xvars(c.($X)##c.($X))  
    > m(rf) pipe(sparse) opt(max_features(5))  
    > m(gradboost) pipe(sparse) opt(n_estimators(250) learning_rate(0.01)) , 
    > njobs(5)  
Learner Y1_pystacked added successfully.

. *** add learners for $E[D|X]$
  . `ddml E[D|X]: pystacked $D \text{ }$ $X`  
    > method(ols)  
    > m(lassocv) xvars(c.($X)##c.($X))  
    > m(ridgecv) xvars(c.($X)##c.($X))  
    > m(rf) pipe(sparse) opt(max_features(5))  
    > m(gradboost) pipe(sparse) opt(n_estimators(250) learning_rate(0.01)) , 
    > njobs(5)  
Learner D1_pystacked added successfully.
Extended ddml example (cont’d.)

**Step 3:** Cross-fitting with 4 folds; also report stacking weights

```
. qui ddml crossfit
. ddml extract, show(stweights)
```

Mean stacking weights across folds/resamples for D1_pystacked (e401)

<table>
<thead>
<tr>
<th>learner</th>
<th>mean_weight</th>
<th>rep_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ols</td>
<td>1</td>
<td>.01557419</td>
</tr>
<tr>
<td>lassocv</td>
<td>2</td>
<td>.10077907</td>
</tr>
<tr>
<td>ridgecv</td>
<td>3</td>
<td>.43674242</td>
</tr>
<tr>
<td>rf</td>
<td>4</td>
<td>.02946916</td>
</tr>
<tr>
<td>gradboost</td>
<td>5</td>
<td>.41743516</td>
</tr>
</tbody>
</table>

Mean stacking weights across folds/resamples for Y1_pystacked (net_tfa)

<table>
<thead>
<tr>
<th>learner</th>
<th>mean_weight</th>
<th>rep_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ols</td>
<td>1</td>
<td>.09662631</td>
</tr>
<tr>
<td>lassocv</td>
<td>2</td>
<td>.46475744</td>
</tr>
<tr>
<td>ridgecv</td>
<td>3</td>
<td>.32388159</td>
</tr>
<tr>
<td>rf</td>
<td>4</td>
<td>.09392877</td>
</tr>
<tr>
<td>gradboost</td>
<td>5</td>
<td>.0145518</td>
</tr>
</tbody>
</table>

Note that these are **mean** weights across 4 cross-fits.
Extended ddml example (cont’d.)

Step 4: Estimation of causal effects - standard stacking only

. ddml estimate, robust

Model: partial, crossfit folds k=4, resamples r=1
Mata global (mname): m0
Dependent variable (Y): net_tfa
    net_tfa learners: Y1_pystacked
D equations (1): e401
    e401 learners: D1_pystacked

DDML estimation results:
    spec  r  Y learner  D learner  b  SE  
st  1  Y1_pystacked  D1_pystacked  9406.385 (1300.170)

Stacking DDML model
y-E[y|X] = y-Y1_pystacked_1
D-E[D|X] = D-D1_pystacked_1

|        | Coefficient | Robust std. err. | z   | P>|z|  | [95% conf. interval] |
|--------|-------------|-----------------|-----|-------|----------------------|
| net_tfa|             |                 |     |       |                      |
| e401   | 9406.385    | 1300.17         | 7.23| 0.000 | 6858.099 11954.67    |
| _cons  | 199.9921    | 535.7477        | 0.37| 0.709 | -850.0541 1250.038   |

Stacking final estimator: nnls1
Extended ddml example (cont’d.)

**Step 4:** Estimation of causal effects - all stacking approaches

```
.ddml estimate, robust shortstack poolstack

Model: partial, crossfit folds k=4, resamples r=1
Mata global (mname): m0
Dependent variable (Y): net_tfa
   net_tfa learners: Y1_pystacked
D equations (1): e401
   e401 learners: D1_pystacked

DDML estimation results:
   spec r Y learner D learner b SE
   st 1 Y1_pystacked D1_pystacked 9406.385 (1300.170)
   ss 1 [shortstack] [ss] 9602.257 (1300.825)
   ps 1 [poolstack] [ps] 9500.180 (1298.057)

Shortstack DDML model
y-E[y|X] = y-Y_net_tfa_ss_1
D-E[D|X] = D-D_e401_ss_1

|      | Coefficient | Robust std. err. | z   | P>|z|   | [95% conf. interval] |
|------|-------------|------------------|-----|--------|---------------------|
| e401 | 9602.257    | 1300.825         | 7.38| 0.000  | 7052.686 12151.83   |
| _cons| 83.96648    | 533.9871         | 0.16| 0.875  | -962.6289 1130.562  |

Stacking final estimator: nnls1
```
Extended ddml example (cont’d.)

**Step 3:** Cross-fitting details - pooled stacking weights

.ddml extract, show(psweights)

pool-stacked weights across resamples for e401
final stacking estimator: nnls1

<table>
<thead>
<tr>
<th>learner</th>
<th>mean_weight</th>
<th>rep_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ols</td>
<td>1</td>
<td>0.01402517</td>
</tr>
<tr>
<td>lassocv</td>
<td>2</td>
<td>0.07247975</td>
</tr>
<tr>
<td>ridgecv</td>
<td>3</td>
<td>0.45850746</td>
</tr>
<tr>
<td>rf</td>
<td>4</td>
<td>0.02897607</td>
</tr>
<tr>
<td>gradboost</td>
<td>5</td>
<td>0.42601154</td>
</tr>
</tbody>
</table>

pool-stacked weights across resamples for net_tfa
final stacking estimator: nnls1

<table>
<thead>
<tr>
<th>learner</th>
<th>mean_weight</th>
<th>rep_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ols</td>
<td>1</td>
<td>0.07029722</td>
</tr>
<tr>
<td>lassocv</td>
<td>2</td>
<td>0.54372578</td>
</tr>
<tr>
<td>ridgecv</td>
<td>3</td>
<td>0.28352699</td>
</tr>
<tr>
<td>rf</td>
<td>4</td>
<td>0.10245001</td>
</tr>
<tr>
<td>gradboost</td>
<td>5</td>
<td>0</td>
</tr>
</tbody>
</table>

Pooled stacking uses a **single** set of weights across 4 cross-fits.
Step 3: Cross-fitting details - short-stacking weights

.ddml extract, show(ssweights)

short-stacked weights across resamples for e401
final stacking estimator: nnls1

<table>
<thead>
<tr>
<th>learner</th>
<th>mean_weight</th>
<th>rep_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ols</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lassocv</td>
<td>2</td>
<td>.24106979</td>
</tr>
<tr>
<td>ridgecv</td>
<td>3</td>
<td>.34172854</td>
</tr>
<tr>
<td>rf</td>
<td>4</td>
<td>.05456544</td>
</tr>
<tr>
<td>gradboost</td>
<td>5</td>
<td>.36263623</td>
</tr>
</tbody>
</table>

short-stacked weights across resamples for net_tfa
final stacking estimator: nnls1

<table>
<thead>
<tr>
<th>learner</th>
<th>mean_weight</th>
<th>rep_1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ols</td>
<td>1</td>
<td>.07689168</td>
</tr>
<tr>
<td>lassocv</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>ridgecv</td>
<td>3</td>
<td>.79121732</td>
</tr>
<tr>
<td>rf</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>gradboost</td>
<td>5</td>
<td>.131891</td>
</tr>
</tbody>
</table>

Short-stacking uses a **single** set of weights. Standard stacking is not required so estimation using just short-stacking is fast.
qddml example: partially-linear model

qddml is the one-line (‘quick’) version of ddml and uses a syntax similar to pds/ivlasso.

The qddml default when used with pystacked is to do short-stacking only (much faster than standard stacking).

NB: This can also be done with ddml- use the nostdstack option at the cross-fit stage.

Here is how to do the same DDML estimation in one line using qddml. We choose a different model name for the Mata object and use the prefix option so the estimated model and conditional expectations in Stata’s memory don’t overwrite those from the previous estimation.

NB: All ddml postestimation commands and utilities also work after qddml. Below we illustrate the use of the replay option of ddml estimate.
. global pystacked_opts
  > method(ols)
  > m(lasso_cv) xvars(c.($X)##c.($X))
  > m(ridge_cv) xvars(c.($X)##c.($X))
  > m(rf) pipe(sparse) opt(max_features(5))
  > m(gradboost) pipe(sparse) opt(n_estimators(250) learning_rate(0.01))
  > njobs(5)

. set seed 123
. // suppress output with quietly
. qui qddml $Y $D ($X), model(partial) kfolds(4) robust
>     pystacked($pystacked_opts)

. // illustrate replay option
. ddml estimate, spec(ss) rep(1) notable replay

Shortstack DDML model
y-E[y|X] = y-Y_net_tfa_ss_1                      Number of obs = 9915
D-E[D|X] = D-D_e401_ss_1

| net_tfa | Coefficient | Robust err. | z   | P>|z| | [95% conf. interval] |
|---------|-------------|-------------|-----|------|----------------------|
| e401    | 9602.257    | 1300.825    | 7.38| 0.000| 7052.686 12151.83 |
| _cons   | 83.96648    | 533.9871    | 0.16| 0.875| -962.6289 1130.562 |

Stacking final estimator:
Summary

- **ddml** implements Double/Debiased Machine Learning for Stata:
  - Compatible with various ML programs in Stata
  - Short (one-line) and flexible multi-line version
  - Uses Stacking Regression as the default machine learner; implemented via separate program pystack
  - 5 models supported

- The advantage to *pdslasso* is that we can make use of almost any machine learner.

- **But which machine learner should we use?**
  - We suggest stacking. We don’t know which learner is best suited for a particular problem.
  - Stacking allows to consider multiple learners in a joint framework, and thus reduces the risk of misspecification.
  - ddml supports 3 forms of stacking: standard stacking, short-stacking and pooled stacking. NB: Our MC results (separate paper) suggest short-stacking performs as well or better than the other two versions and is much faster; our recommended default.


