The Stata Journal (2022)
22, Number 4, pp. 772–810 DOI: 10.1177/1536867X221140944

Machine learning using Stata/Python

Giovanni Cerulli
IRCrES-CNR
Rome, Italy
giovanni.cerulli@ircres.cnr.it

Abstract. I present two related commands, `r_ml_stata_cv` and `c_ml_stata_cv`, for fitting popular machine learning methods in both a regression and a classification setting. Using the recent Stata/Python integration platform introduced in Stata 16, these commands provide hyperparameters' optimal tuning via K-fold cross-validation using grid search. More specifically, they use the Python Scikit-learn application programming interface to carry out both cross-validation and outcome/label prediction.

Keywords: pr0076, r_ml_stata_cv, c_ml_stata_cv, get_test_train, machine learning, Python, optimal tuning

1 Introduction

Machine learning (ML) (also known as statistical learning[1]) has emerged as a leading data-science approach in many fields, including business, engineering, medicine, advertising, and scientific research. Placing itself in the intersection of statistics, computer science, and artificial intelligence, ML's main objective is turning information into valuable knowledge by "letting the data speak", limiting the model's prior assumptions, and promoting a model-free philosophy. Relying on algorithms and computational techniques, more than on analytic solutions, ML targets big data and complexity reduction, although sometimes at the expense of results' interpretability (Hastie, Tibshirani, and Friedman 2009; Varian 2014).

Unlike other software, such as R, Python, MATLAB, and SAS, Stata does not have dedicated built-in packages for fitting ML algorithms, if one excludes the Lasso package of Stata 16. Recently, however, the Stata community has developed some popular ML routines that Stata users can suitably exploit. Among them, I mention Schonlau (2005) implementing a boosting Stata plugin; Guenther and Schonlau (2016) providing

1. In the literature, the terms ML and statistical learning are used interchangeably. The term ML, however, was initially coined and used by engineers and computer scientists. Historically, this term was popularized by Samuel (1959) in his famous article on the use of experience-based learning machines for playing the game of checkers as opposed to symbolic/knowledge-based learning machines relying on hard-wired programming rules. Samuel proved that teaching a machine increasingly complex rules to carry out intelligent tasks (as in the case of expert-systems) was a much less efficient strategy than letting the machine learn from experience, that is, from data. Consequently, he pointed out that statistical learning is the basis of a better ability of machines to learn from past (stored) events. We may say, in this sense, that ML is based on statistical learning. Of course, term usage is also reflected in the scientific community of developers, with engineers preferring to use the label "machine" learning and statisticians preferring the label "statistical" learning.

a command fitting support vector machines (SVM); Ahrens, Hansen, and Schaffer (2020) setting out the `lassopack`, a set of commands for model selection and prediction with regularized regression; and Schonlau and Zou (2020) providing a command for the random forests algorithm. All of these are valuable packages for executing popular ML algorithms within Stata.

The absence of an integrated Stata package for carrying out ML algorithms also prevents uniformity and comparability of these methods. To pursue generality, uniformity, and comparability, one should rely on a software platform able to suitably integrate most of the mainstream ML methods. For example, Python has powerful platforms to carry out both ML and deep-learning algorithms (Raschka and Mirjalili 2019). Among them, the most popular are Scikit-learn for fitting many ML methods, and TensorFlow and Keras for more generally fitting neural network and deep-learning techniques. These make Python, which is freeware, probably the most effective and complete software for ML and deep-learning available within the community.

The Stata 16 release introduced a useful Stata/Python application programming interface (API). The Stata Function Interface (`sfi`) module allows users to interact Python's capabilities with core features of Stata. The command can be used interactively or in do-files and ado-files.

Taking advantage of the new Stata/Python integration interface, Droste (2022) has developed `pylearn`, a set of commands to perform supervised learning in Stata. These commands all exhibit a common Stata-like syntax for model estimation and postestimation. The `pylearn` suit currently allows for fitting decision trees, random forests, adaptive boosting, gradient boosting, and multilayer perceptrons (neural networks) directly from Stata. Specifically, `pylearn` is a wrapper of the popular Python library Scikit-learn. The `pylearn` ML commands are nearly exact reproductions of the Python functions implementing the same ML methods. However, at present, they do not provide hyperparameters' tuning via cross-validation (CV).

In a similar manner, still using the Stata/Python integration interface, this article presents two related commands, `r_ml_stata_cv` and `c_ml_stata_cv`, wrapping Python functions for fitting popular ML methods in both a regression and a classification setting. Both commands provide hyperparameters' optimal tuning via K-fold CV by implementing grid search. Like with `pylearn`, they use the Python Scikit-learn API to execute both CV and outcome/label prediction. Compared with `pylearn`, these commands present a smaller set of options but have two important advantages: i) they implement a larger set of learners, and ii) for every learner, they allow users for customized K-fold CV over the most relevant tuning parameters (those entailing a trade-off between prediction variance and prediction bias).

Why would the Stata community need these commands? I see three related answers to this question. First, Stata users who are willing to apply ML methods would benefit from these commands because they can allow them to perform ML without time-consuming investment in learning other software. Second, these commands provide users with standard Stata returns and variables' generation that can be further used in subsequent Stata coding within the same do-file. This makes the user's work-

flow smoother, less error-prone, and faster. Third, these commands represent a sort of case study showing the potential of integrating Stata and Python. In my experience, Stata is unbeatable compared with Python when it comes to performing intensive data management and manipulation, descriptive statistics, and immediate graphing, while Python is more suitable for its large set of implemented ML algorithms, an all-encompassing availability of libraries, and computing speed and sophisticated graphing (not only 3D, but also deep contour plots, and images' reproduction).

The structure of the article is as follows. Section 2 presents a brief introduction to ML, where section 2.1 sets out the basics, section 2.2 the optimal tuning procedure via CV, and section 2.3 the learning methods and architecture. Sections 3.1 and 3.2 present the syntax of `r_ml_stata_cv` and `c_ml_stata_cv`. Section 3.3 presents the syntax of `get_test_train`, a command one can use to generate the training and testing datasets from an initial dataset loaded in the current Stata session. For users to familiarize with these commands, section 4 illustrates a simple step-by-step application for fitting a regression tree, while section 5 shows how to apply the proposed commands to a real dataset (specifically, the popular `iris.dta`) for classification purposes. To deal with a larger dataset, section 6 focuses on the classification of handwritten numerals, a task that can be encompassed within the larger set of image recognition studies. Section 7, finally, develops an ML estimation of conditional average treatment effects (CATEs), thus showing that the methods implemented via `c_ml_stata_cv` and `r_ml_stata_cv` can be used not only for predictive purposes but also for estimating causal effects. Section 8 concludes the article.

2 A brief introduction to ML

2.1 The basics of ML

ML is the branch of artificial intelligence mainly focused on statistical prediction (Boden 2018). The literature distinguishes between supervised and unsupervised learning, referring to a setting where the outcome variable is known (supervised) or unknown (unsupervised). In statistical terms, supervised learning coincides with a regression or classification setup, where regression represents the case in which the outcome variable is numerical, and classification represents the case in which it is categorical. In contrast, unsupervised learning deals with unlabeled data, and its main concern is that of generating a categorical variable via proper clustering algorithms. Unsupervised learning can thus be encompassed within statistical cluster analysis.

In this article, I focus on supervised ML.[2] We are thus interested in predicting either numerical variables (regression) or label classes (classification) as a function of p predictors (or features) that may be quantitative or qualitative, or both. From a statistical point of view, we want to fit the conditional expectation of y, the outcome, on a set of p predictors, $\mathbf{x}$, starting from the following population equation:

$$y = f(x_1, \ldots, x_p) + \epsilon$$

2. This section and the next are based on the book of Hastie, Tibshirani, and Friedman (2009).

where ϵ is an error term with mean equal to zero and finite variance. By taking the expectation over $\mathbf{x}$, and assuming that $E(\epsilon|\mathbf{x}) = 0$, we have

$$y = E(y|\mathbf{x}) + \epsilon = f(x_1, \ldots, x_p) + \epsilon$$

that is,

$$E(y|\mathbf{x}) = f(x_1, \ldots, x_p) \tag{1}$$

The conditional expectation $f(\cdot)$ represents a function mapping predictors and expected outcomes. The main purpose of supervised ML is that of fitting (1) with the aim of reducing as much as possible the prediction error coming up when one wants to predict actual outcome variables. When y is a numerical outcome (regression setting), the prediction error can be defined as $e = y - \widehat{f}(\mathbf{x})$, with $\widehat{f}(\cdot)$ indicating prediction from a specific ML method. In a regression setting, ML scholars focus on minimizing the mean squared error (MSE), defined as $\text{MSE} = E\{y - \widehat{f}(\mathbf{x})\}^2$. However, while the in-sample MSE (the so-called "training-MSE") is generally affected by overfitting (thus going to zero as the model's degrees of freedom—or complexity—increase), the out-of-sample MSE (also known as "test-MSE") has the property to be a convex function of model complexity, and thus characterized by an optimal level of complexity (Hastie, Tibshirani, and Friedman 2009).

It can be proved [see (2.46) in Hastie, Tibshirani, and Friedman (2009)] that the test-MSE, when evaluated at one out-of-sample observation (also called "new instance" in the ML jargon), can be decomposed into three components—variance, squared bias, and irreducible error—as follows:

$$E\left\{y - \widehat{f}\,(\mathbf{x}_0)\right\}^2 = \text{Var}\left\{\widehat{f}(\mathbf{x}_0)\right\} + \left[\text{Bias}\left\{\widehat{f}\,(\mathbf{x}_0)\right\}\right]^2 + \text{Var}(\epsilon)$$

where $\mathbf{x}_0$ is a new instance, that is, an observation that did not participate in producing the ML fit $\widehat{f}(\cdot)$. Figure 1 shows a graphical representation of the pattern of the previous quantities as functions of model complexity. It is immediate to see that, as long as model complexity increases, the bias decreases while the variance increases monotonically. Because of this, the test-MSE sets out a parabola-shaped pattern, which allows us to minimize it at a specific level of model complexity. This is the optimal model tuning, whenever complexity is measured by a specific hyperparameter λ. In the figure, the irreducible error variance represents a constant lower bound of the test-MSE. It is not possible to overcome this minimum test-MSE, because it depends on the nature of the data-generating process (intrinsic unpredictability of the phenomenon under analysis).

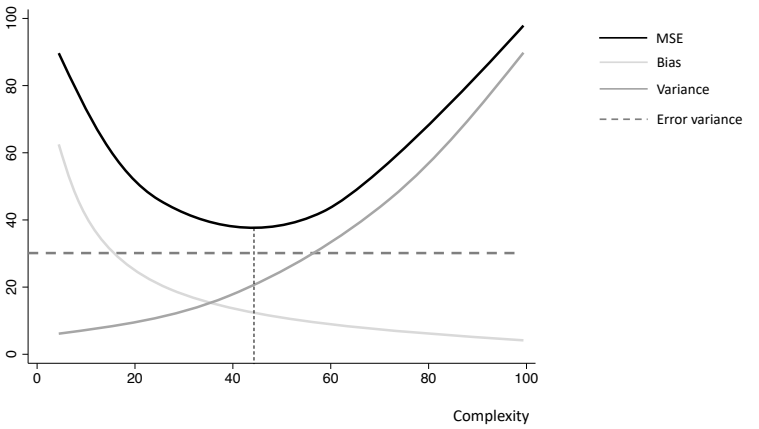


Figure 1. Trade-off between bias and variance as functions of model complexity

In the classification setting, the MSE is meaningless because in this case we have class labels and not numerical values. For classification purposes, the correct objective function to minimize is the (test-)mean classification error (MCE), defined as

$$\text{MCE} = E\left[I\left\{y \neq \widehat{C}(\mathbf{x})\right\}\right]$$

where $I(\cdot)$ is an index function (taking value 1 if its argument is a true statement and 0 otherwise), and $\widehat{C}(\mathbf{x})$ is the fitted classifier. As in the case of the MSE, it can be proved (Hastie, Tibshirani, and Friedman 2009) that the training-MCE overfits the data when model complexity increases, while the test-MCE allows us to find the optimal model's complexity. Therefore, the graph of figure 1 can be likewise extended to the case of the test-MCE.

2.2 Optimal tuning via CV

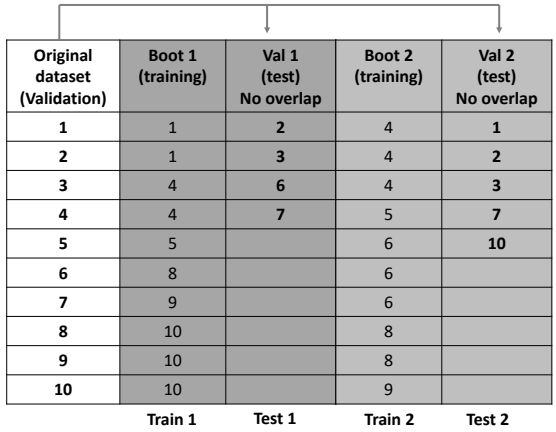
Finding the optimal model complexity, parameterized by a generic hyperparameter λ, is a computational task. There are basically three ways to tune an ML model:

- Information criteria
- Bootstrap
- K-fold CV

Information criteria are based on goodness-of-fit formulas that adjust the training error by penalizing too complex models (that is, models characterized by large degrees of freedom). Traditional information criteria comprise the Akaike information criterion and the Bayesian information criterion, and can be applied to both linear and nonlinear

models (probit, logit, poisson, etc.). Unfortunately, the information criteria are valid only for linear or generalized linear models (GLM), that is, for parametric regression. They cannot be computed for nonparametric methods like—for example—tree-based or nearest neighbor regressions.

For nonparametric models, the test error can be estimated via computational techniques, more specifically, by resampling methods. Bootstrap—resampling with replacement from the original sample—could in theory be a practical solution, provided that the original dataset is used as a validation dataset and the bootstrapped ones as training datasets. Unfortunately, the bootstrap has the limitation of generating observation overlaps between the test and the training datasets, because about two-thirds of the original observations appear in each bootstrap sample. This occurrence undermines its use to validate an out-of-sample ML procedure. Figure 2 illustrates a simple example visualizing bootstrap overlapping in a sample with $N = 10$ observations and $B = 2$ bootstrap replications.



Original dataset (Validation)	Boot 1 (training)	Val 1 (test) No overlap	Boot 2 (training)	Val 2 (test) No overlap
1	1	**2**	4	**1**
2	1	**3**	4	**2**
3	4	**6**	4	**3**
4	4	**7**	5	**7**
5	5		6	**10**
6	8		6	
7	9		6	
8	10		8	
9	10		8	
10	10		9	
	Train 1	**Test 1**	**Train 2**	**Test 2**

Figure 2. Visualizing bootstrap overlapping in a sample with $N = 10$ observations and $B = 2$ bootstrap replications

The observations of the original sample are labeled: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10. The first bootstrap produces the sample Boot 1, which we use as a training dataset. If the original sample has to be the validation sample, we clearly have an overlap; specifically, units 1, 4, 5, 8, 9, 10 appear in both the training and the testing samples. The nonoverlapping sample is in the column with heading Val 1 and contains the following units: 2, 3, 6, 7. We may use these units as validating observations. In conclusion, the model will be trained over Boot 1 and validated over Val 1, thus producing a first estimate of the prediction error (either MSE or MCE). By repeating the same procedure as many times as the number of preset bootstrap samples, we obtain an estimate of the mean prediction error and its standard deviation.

CV is the workhorse of the test-error estimation in ML. The idea is to randomly divide the initial dataset into K equal-sized portions called folds. This procedure suggests to leave out fold k and fit the model to the other $(K-1)$ folds (wholly combined) to then obtain predictions for the left-out kth fold. This is done in turn for each fold $k = 1, 2, \ldots, K$, and then the results are combined by averaging the $(K-1)$ estimates of the error. In a regression setting, where y is numerical, the CV procedure can be carried out as follows:

- Split randomly the initial dataset into K folds denoted as $G_1, G_2, \ldots, G_K$, where G_k refers to part k. Assume that there are n_k observations in fold k. If N is a multiple of K, then $n_k = n/K$.
- For each fold $k = 1, 2, \ldots, K$, compute the mean squared error (MSE):

  $$\text{MSE}_k = \sum_{i \in G_k} \left(y_i - \widehat{y}_i\right)^2 / n_k$$

  where $\widehat{y}_i$ is the fit for observation i, obtained from the dataset with fold k removed.
- Compute

  $$\text{CV}_K = \sum_{i=1}^{K} \frac{n_k}{n} \text{MSE}_k$$

  which is the average out-of-sample MSE obtained fold by fold.

Observe that, by setting $K = n$, we obtain the n-fold or leave-one-out CV. Also, because CV_K is an estimation of the true test error, estimating its standard error can provide a confidence interval and thus a measure of test accuracy's uncertainty.

Finally, for ML classification purposes, the CV procedure follows the same protocol except considering the MCE in place of the MSE. Moreover, in the case of binary classification, other accuracy measures can be used, such as the area under the receiver operating characteristics (ROC) curve and the F1-score (the harmonic mean of recall and precision).[3]

3. More specifically, for binary classification, we can build a matrix called the confusion matrix showing the number of test cases that were correctly and incorrectly classified. Consider a classification setting with two target classes: 1 = positive and 0 = negative. After using a specific ML classifier, we can define four groups: TN, the number of negative cases correctly classified; TP, the number of positive cases correctly classified; FN, the number of positive cases incorrectly classified as negative; and FP, the number of negative cases incorrectly classified as positive. The precision, defined as TP/(TP+FP), is the ratio of correct positive classifications to the total number of predicted positive classifications. The recall, defined as TP/(TP + FN), is the ratio of correct positive classifications to the total number of positive classifications (that is, both correct and incorrect). Finally, the ROC curve plots the true positive rate (the recall) against the false positive rate TN/(TN + FP). We can thus compute the area under the ROC curve. The larger the area, the better the model prediction performance.

2.3 Learning methods and architecture

A learner L_j is a mapping from the set $[\mathbf{x}, \boldsymbol{\theta}_j, \boldsymbol{\lambda}_j, f_j(\cdot)]$ to an outcome y, where $\mathbf{x}$ is the vector of features, $\boldsymbol{\theta}_j$ is a vector of estimation parameters, $\boldsymbol{\lambda}_j$ is a vector of tuning parameters, and $f_j(\cdot)$ is an algorithm taking as inputs $\mathbf{x}$, $\boldsymbol{\theta}_j$, and $\boldsymbol{\lambda}_j$. While the members of the GLM family (linear, probit, and multinomial regressions are classical examples) are highly parametric and not characterized by tuning parameters, ML models—such as local-kernel, nearest-neighbor, or tree-based regressions—may be highly nonparametric and characterized by one or more hyperparameters $\boldsymbol{\lambda}_j$, which may be optimally chosen to minimize the test prediction error, that is, the out-of-sample predicting accuracy of the learner, as stressed in the previous section.

A detailed description of all the available ML methods is beyond the scope of this article. Table 1, however, sets out the most popular ML algorithms proposed in the literature along with the most relevant associated tuning parameters.

Table 1. Main ML methods and associated tuning hyperparameters

ML method	Parameter 1	Parameter 2	Parameter 3
Linear models and GLM	N of covariates		
Lasso	Penalization coefficient		
Elastic net	Penalization coefficient	Elastic parameter	
Nearest neighbor	N of neighbors		
Neural network	N of hidden layers	N of neurons	L_2 penalization
Trees	N of leaves/depth		
Boosting	Learning parameter	N of sequential trees	N of leaves/depth
Random forest	N of features for splitting	N of bootstraps	N of leaves/depth
Bagging	Tree depth	N of bootstraps	
SVM	C	Γ	
Kernel regression	Bandwidth	Kernel function	
Piecewise regression	N of knots		
Series regression	N of series terms		

Combining these methods can produce a computational architecture (that is, a virtual learning machine) enabling increased statistical prediction accuracy and its estimated precision (van der Laan, Polley, and Hubbard 2007). Figure 3 presents the learning architecture proposed by Cerulli (2021). This framework is made of three linked learning processes: i) the learning over the tuning parameter λ, ii) the learning over the algorithm $f(\cdot)$, and iii) the learning over new additional information. The departure is in point 1, from where we set off assuming the availability of a dataset $[\mathbf{x}, y]$.

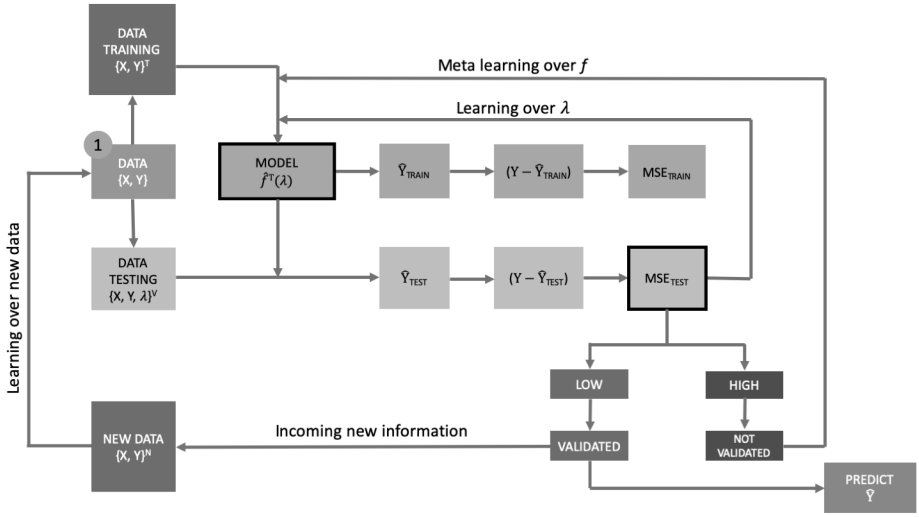


Figure 3. The meta-learning machine architecture; drawn from Cerulli (2021)

The first learning process aims at selecting the optimal tuning parameter(s) for a given algorithm $f_j(\cdot)$. As seen above, ML scholars typically do it using K-fold CV to draw test-accuracy (or, equivalently, test-error) measures and related standard deviations.

At the optimal λ_j, one can recover the largest possible prediction accuracy for the learner $f_j(\cdot)$. Further prediction improvements can be achieved only by learning from other learners, namely, by exploring other $f_j(\cdot)$, with $j = 1, \ldots, M$ (where M is the number of learners at hand).

Figure 3 shows the training estimation procedure corresponding to the medium-gray sequence of boxes leading to $\text{MSE}_{\text{TRAIN}}$, which is, de facto, a dead-end node, being the training error plagued by overfitting.

The light-gray sequence in figure 3 leads to MSE_{TEST}. This sequence helps one make correct decisions about the predicting quality of the current learner. At this node, the analyst can compare the current MSE_{TEST} with a benchmark one (possibly prefixed) and conclude whether to predict using the current learner or to explore alternative learners in the hope of increasing predictive performance. If the level of the current prediction error is too high, the architecture would suggest to explore other learners.

In the ML literature, learning over learners is called meta learning and entails an exploration of the out-of-sample performance of alternative algorithms $f_j(\cdot)$ with the goal of identifying one behaving better than those already explored (van der Laan and Rose 2011). For each new $f_j(\cdot)$, this architecture finds an optimal tuning parameter and a new estimated accuracy (along with its standard deviation). The analyst can either explore the entire bundle of alternatives and select the best one or decide to select the first learner whose accuracy is larger than the benchmark. Either case is automatically run by this virtual machine.

The third and final learning process concerns the availability of new information, via additional data collection. This induces a reiteration of the initial process whose final outcome can lead one to choose a different algorithm and tuning parameter(s), depending on the nature of the incoming information. This process is called online learning (Parisi et al. 2019; Bottou 1998).

As a final step, one may combine predictions of single optimal learners into a single super-prediction (ensemble learning). What is the advantage of this procedure? The prediction error of a learner depends on the sum of (squared) prediction bias and prediction variance. Because the sum of independent and identically distributed random variables has a smaller variance than that of the individual elements of the sum, the benefit of aggregating learners is obtaining a predictor with a smaller variance (Zhou 2012). However, because the bias of the ensemble predictor is not guaranteed to be the smallest among the biases of the individual learners, its estimated error might not be smaller than that of a single learner.[4] As a consequence, using an ensemble method may not lead to accuracy improvements, although, in applications, the opposite case occurs rather often (Kumar 2020).

3 The commands

3.1 Syntax for r_ml_stata_cv

The command `r_ml_stata_cv` fits some popular ML regression algorithms. It considers as the main inputs a continuous response variable y (that is, the *depvar*), a series of predictors (or features) in *varlist* explaining the y, and a series of options.

`r_ml_stata_cv` *depvar varlist* [*if*] [*in*]`, mlmodel(`*modeltype*`)`
`data_test(`*filename*`) seed(`*integer*`)` [*learner_options cv_options*
other_options]

depvar is a numerical variable. *varlist* is a list of numerical variables representing the predictors. When a feature is categorical, it is the user's responsibility to generate the appropriate dummies because the command does not do this by default.

4. To clarify this point, consider the following illustrative example. Assume that the true value to predict in the population is equal to 10. Suppose there are three predictors, L_1, L_2, and L_3, with zero prediction variances but with means respectively equal to $\mu_1 = 15$, $\mu_2 = 20$, and $\mu_3 = 25$. The squared bias of L_1 is thus $b_1^2 = (15-10)^2 = 25$, of L_2 is $b_2^2 = (20-10)^2 = 100$, and of L_3 is $b_3^2 = (25-10)^2 = 225$. Because the ensemble prediction is an average of the predictions from L_1, L_2, and L_3, its squared bias is equal to $\{(15+20+25)/3-10\}^2 = 100$. This value, equal to that of L_2, is not the smallest; L_1 performs better obtaining a MSE equal to 25.

3.1.1 Main options

`mlmodel(`*modeltype*`)` specifies the ML algorithm to be estimated. *modeltype* may be one of the following: `boost` (boosting), `elasticnet` (elastic net), `nearestneighbor` (nearest neighbor), `neuralnet` (neural network), `ols` (ordinary least squares), `randomforest` (bagging and random forests), `svm` (SVM), or `tree` (tree regression). `mlmodel()` is required.

`data_test(`*filename*`)` requests to provide a testing dataset as *filename*. `data_test()` is required.

`seed(`*integer*`)` specifies an integer seed to assure replication of results. `seed()` is required.

3.1.2 learner_options

- Boosting (option `mlmodel(boost)`):
  - `tree_depth(#)` specifies the maximum tree depth. # is an integer or, for cross-validation, a list of integers.
  - `learning_rate(#)` specifies the coefficient that shrinks the contribution of each tree to the boosting's prediction. # is a float or, for cross-validation, a list of floats.
  - `n_estimators(#)` specifies the number of boosting iterations to perform. # is an integer or, for cross-validation, a list of integers.
- Elastic net (option `mlmodel(elasticnet)`):
  - `alpha(#)` specifies the so-called shrinkage parameter, that is, a constant that multiplies the penalty term. `alpha(0)` corresponds to ordinary least squares. # is a float or, for cross-validation, a list of floats.
  - `l1_ratio(#)` specifies the elastic net mixing parameter, varying between 0 (lasso regression) and 1 (ridge regression). Intermediate values of this parameter weigh the lasso and ridge penalization terms differently. # is a float or, for cross-validation, a list of floats.
- Nearest neighbor (option `mlmodel(nearestneighbor)`):
  - `nn(#)` specifies the number of nearest neighbors. # is an integer or, for cross-validation, a list of integers.

- Neural network (option `mlmodel(neuralnet)`):
  - `n_neurons_l1(#)` specifies the number of neurons (or hidden units) in the first (hidden) layer. # is an integer or, for cross-validation, a list of integers.
  - `n_neurons_l2(#)` specifies the number of neurons (or hidden units) in the second (hidden) layer. # is an integer or, for cross-validation, a list of integers.
  - `alpha(#)` specifies the L_2 penalization parameter. # is a float or, for cross-validation, a list of floats.
- Random forest (option `mlmodel(randomforest)`):
  - `tree_depth(#)` specifies the maximum tree depth. # is an integer or, for cross-validation, a list of integers.
  - `max_features(#)` specifies the number of features to consider when looking for the tree best split. # is an integer or, for cross-validation, a list of integers.
  - `n_estimators(#)` specifies the number of bootstrapped trees. # is an integer or, for cross-validation, a list of integers.
- SVM (option `mlmodel(svm)`):
  - `c(#)` specifies the SVM regularization parameter. # is a float or, for cross-validation, a list of floats.
  - `gamma(#)` specifies the kernel coefficient for the radial basis function (RBF). # is a float or, for cross-validation, a list of floats.
- Tree regression (option `mlmodel(tree)`):
  - `tree_depth(#)` specifies the maximum tree depth. # is an integer or, for cross-validation, a list of integers.

3.1.3 cv_options

`cross_validation(`*name*`)` specifies a name for the dataset that will contain CV results. This file is automatically generated and saved in the user's current working directory. The command uses K-fold CV.

`n_folds(`*integer*`)` specifies the number of CV folds.

3.1.4 other_options

`prediction(`*name*`)` generates predictions of the outcome variable *name*. Both training and testing predictions are generated.

`default` allows fitting the specified ML model with parameters equal to Scikit-learn's default values.

`graph_cv` allows display of the CV optimal tuning graph, which draws the pattern of both train and test accuracy.

`save_graph_cv(`*name*`)` saves in the current working directory the CV optimal tuning graph.

3.1.5 Stored results

`r_ml_stata_cv` stores the following in `e()`:

Scalars

`e(N_train_all)`	total number of observations in the initial training dataset
`e(N_train_used)`	number of training observations used
`e(N_test_all)`	total number of observations in the initial testing dataset
`e(N_test_used)`	number of testing observations used
`e(N_features)`	number of features
`e(TRAIN_ACCURACY)`	K-fold CV training average accuracy (= explained variance)
`e(TEST_ACCURACY)`	K-fold CV testing average accuracy (= explained variance)
`e(SE_TEST_ACCURACY)`	K-fold CV standard error of the testing average accuracy (= explained variance)
`e(BEST_INDEX)`	best CV index
`e(N_FOLDS)`	number of folds used for CV
`e(Train_mse)`	MSE on training data
`e(Test_mse)`	MSE on testing data
`e(Train_mape)`	mean absolute prediction error (MAPE) on training data
`e(Test_mape)`	MAPE on testing data

If *learner_options* are used, `r_ml_stata_cv` stores the following in `e()`:

For `mlmodel(boost)`:

Scalars

`e(OPT_MAX_DEPTH)`	maximum depth of the tree
`e(OPT_LEARNING_RATE)`	shrinking contribution of each tree to the boosting's prediction
`e(OPT_N_ESTIMATORS)`	number of boosting iterations

For `mlmodel(elasticnet)`:

Scalars

`e(OPT_ALPHA)`	penalization parameter
`e(OPT_L1_RATIO)`	elastic net mixing parameter: `0` = lasso, `1` = ridge regression

For `mlmodel(nearestneighbor)`:

Scalar

`e(OPT_NN)`	number of nearest neighbors to use

Macro

`e(OPT_WEIGHT)`	local kernel weighting scheme: `uniform` (observations equally weighted) or `distance` (observations weighted by the inverse of their distance from the point of imputation)

For `mlmodel(neuralnet)`:

Scalars

`e(OPT_NEURONS_L_1)`	number of neurons in the first layer
`e(OPT_NEURONS_L_2)`	number of neurons in the second layer
`e(OPT_ALPHA)`	L_2 penalization parameter

For `mlmodel(randomforest)`:

Scalars

`e(OPT_N_ESTIMATORS)`	number of bootstrapped trees for the boosting ensemble prediction
`e(OPT_MAX_DEPTH)`	maximum depth of the tree
`e(OPT_MAX_FEATURES)`	number of splitting features

For `mlmodel(svm)`:

Scalars

`e(OPT_C)`	SVM regularization parameter
`e(OPT_GAMMA)`	kernel coefficient for the RBF

For `mlmodel(tree)`:

Scalar

`e(OPT_DEPTH)`	maximum depth of the tree

3.1.6 Remarks

To run this program, you need to have both Stata 16 (or later) and Python (from version 2.7 onward) installed. You can find information about how to install Python on your machine here: https://www.python.org/downloads. In addition, the Python Scikit-learn (and related dependencies) and `sfi` APIs must be uploaded before running the command. You can find information about how to install and use them at, respectively, https://scikit-learn.org/stable/install.html and https://www.stata.com/python/api17.[5]

The `r_ml_stata_cv` program incorporates the `pylearn, setup` command attributed to Droste (2022) to check whether your computer has the prerequisite Python packages, and if not, will automatically try to install them.

3.2 Syntax for c_ml_stata_cv

The command `c_ml_stata_cv` fits ML classification algorithms. It considers as the main inputs a categorical (integer) response variable y (that is, the *depvar*), a series of predictors (or features) in *varlist* explaining the y, and a series of options.

`c_ml_stata_cv` *depvar varlist* [*if*] [*in*] `, mlmodel(`*modeltype*`)`
`data_test(`*filename*`) seed(`*integer*`)` [*learner_options cv_options*
other_options]

5. The `sfi` API is already installed in Stata 16 and later versions.

depvar is a categorical (integer) variable. *varlist* is a list of numerical variables representing the predictors. When a feature is categorical, it is the user's responsibility to generate the appropriate dummies because the command does not do this by default.

3.2.1 Options

`mlmodel(`*modeltype*`)` specifies the ML algorithm to be estimated. *modeltype* may be one of the following: `boost` (boosting), `multinomial` (multinomial classification), `naivebayes` (naïve Bayes), `nearestneighbor` (nearest neighbor), `neuralnet` (neural network), `randomforest` (bagging and random forests), `regmult` (regularized multinomial), `svm` (SVM), or `tree` (tree classification). `mlmodel()` is required.

`data_test(`*filename*`)` requests to provide a testing dataset as *filename*. `data_test()` is required.

`seed(`*integer*`)` specifies an integer seed to assure replication of results. `seed()` is required.

3.2.2 learner_options

- Boosting (option `mlmodel(boost)`):
  - `tree_depth(#)` specifies the maximum tree depth. # is an integer or, for cross-validation, a list of integers.
  - `learning_rate(#)` specifies the coefficient that shrinks the contribution of each tree to the boosting's prediction. # is a float or, for cross-validation, a list of floats.
  - `n_estimators(#)` specifies the number of boosting iterations to perform. # is an integer or, for cross-validation, a list of integers.
- Nearest neighbor (option `mlmodel(nearestneighbor)`):
  - `nn(#)` specifies the number of nearest neighbors. # is an integer or, for cross-validation, a list of integers.
- Neural network (option `mlmodel(neuralnet)`):
  - `n_neurons_l1(#)` specifies the number of neurons (or hidden units) in the first (hidden) layer. # is an integer or, for cross-validation, a list of integers.
  - `n_neurons_l2(#)` specifies the number of neurons (or hidden units) in the second (hidden) layer. # is an integer or, for cross-validation, a list of integers.
  - `alpha(#)` specifies the L_2 penalization parameter. # is a float or, for cross-validation, a list of floats.

- Random forest (option `mlmodel(randomforest)`):
  - `tree_depth(#)` specifies the maximum tree depth. # is an integer or, for cross-validation, a list of integers.
  - `max_features(#)` specifies the number of features to consider when looking for the tree best split. # is an integer or, for cross-validation, a list of integers.
  - `n_estimators(#)` specifies the number of bootstrapped trees. # is an integer or, for cross-validation, a list of integers.
- Regularized multinomial (option `mlmodel(regmult)`):
  - `alpha(#)` specifies the so-called shrinkage parameter, that is, a constant that multiplies the penalty term. `alpha(0)` corresponds to standard multinomial model. # is a float or, for cross-validation, a list of floats.
  - `l1_ratio(#)` specifies the mixing parameter, varying between 0 (lasso L_1 penalization) and 1 (ridge L_2 penalization). Intermediate values of this parameter weigh the lasso and ridge penalization terms differently. # is a float or, for cross-validation, a list of floats.
- SVM (option `mlmodel(svm)`):
  - `c(#)` specifies the SVM regularization parameter. # is a float or, for cross-validation, a list of floats.
  - `gamma(#)` specifies the kernel coefficient for the RBF. # is a float or, for cross-validation, a list of floats.
- Tree classification (option `mlmodel(tree)`):
  - `tree_depth(#)` specifies the maximum tree depth. # is an integer or, for cross-validation, a list of integers.

3.2.3 cv_options

`cross_validation(`*name*`)` specifies a name for the dataset that will contain CV results. This file is automatically generated and saved in the user's current working directory. The command uses K-fold CV.

`n_folds(`*integer*`)` specifies the number of CV folds.

3.2.4 other_options

`prediction(`*name*`)` generates class and probability predictions of the outcome variable *name*. Both training and testing predictions are generated.

`default` allows fitting the specified ML model with parameters equal to Scikit-learn's default values.

`graph_cv` allows display of the CV optimal tuning graph, which draws the pattern of both train and test accuracy.

`save_graph_cv(`*name*`)` saves in the current working directory the CV optimal tuning graph.

3.2.5 Stored results

`c_ml_stata_cv` stores the following in `e()`:

Scalars

`e(N_train_all)`	total number of observations in the initial training dataset
`e(N_train_used)`	number of training observations used
`e(N_test_all)`	total number of observations in the initial testing dataset
`e(N_test_used)`	number of testing observations used
`e(N_features)`	number of features
`e(TRAIN_ACCURACY)`	K-fold CV training average accuracy (= share of correct classification matches)
`e(TEST_ACCURACY)`	K-fold CV testing average accuracy (= share of correct classification matches)
`e(SE_TEST_ACCURACY)`	K-fold CV standard error of the testing average accuracy (= share of correct classification matches)
`e(BEST_INDEX)`	best CV index
`e(N_FOLDS)`	number of folds used for CV
`e(Train_err)`	MCE on training data
`e(Test_err)`	MCE on testing data

If *learner_options* are used, `c_ml_stata_cv` stores the following in `e()`:

For `mlmodel(boost)`:

Scalars

`e(OPT_MAX_DEPTH)`	maximum depth of the tree
`e(OPT_LEARNING_RATE)`	shrinking contribution of each tree to the boosting's prediction
`e(OPT_N_ESTIMATORS)`	number of boosting iterations

For `mlmodel(naivebayes)`:

Scalar

`e(OPT_VAR_SMOOTHING)`	portion of the largest variance of all predictors that is added to variances for calculation stability

For `mlmodel(nearestneighbor)`:

Scalar

`e(OPT_NN)`	number of nearest neighbors to use

Macro

`e(OPT_WEIGHT)`	local kernel weighting scheme: `uniform` (observations equally weighted) or `distance` (observations weighted by the inverse of their distance from the point of imputation)

For `mlmodel(neuralnet)`:

Scalars

`e(OPT_NEURONS_L_1)`	number of neurons in the first layer
`e(OPT_NEURONS_L_2)`	number of neurons in the second layer
`e(ALPHA)`	L_2 penalization parameter

For `mlmodel(randomforest)`:

Scalars

`e(OPT_N_ESTIMATORS)`	number of bootstrapped trees for the boosting ensemble prediction
`e(OPT_MAX_DEPTH)`	maximum depth of the tree
`e(OPT_MAX_FEATURES)`	number of splitting features

For `mlmodel(regmult)`:

Scalars

`e(OPT_ALPHA)`	penalization parameter
`e(OPT_L1_RATIO)`	mixing parameter: `0` = lasso, `1` = ridge penalization

For `mlmodel(svm)`:

Scalars

`e(OPT_C)`	SVM regularization parameter
`e(OPT_GAMMA)`	kernel coefficient for the RBF

For `mlmodel(tree)`:

Scalar

`e(OPT_DEPTH)`	maximum depth of the tree

3.2.6 Remarks

To run this program, you need to have both Stata 16 (or later) and Python (from version 2.7 onward) installed. You can find information about how to install Python on your machine here: https://www.python.org/downloads. In addition, the Python Scikit-learn (and related dependencies) and `sfi` APIs must be uploaded before running the command. You can find information about how to install and use them at, respectively, https://scikit-learn.org/stable/install.html and https://www.stata.com/python/api17.[6]

The `c_ml_stata_cv` program incorporates the `pylearn, setup` command attributed to Droste (2022) to check whether your computer has the prerequisite Python packages, and if not, will automatically try to install them.

6. The `sfi` API is already installed in Stata 16 and later versions.

3.3 Syntax for get_train_test

The command `get_train_test` can be used to generate the training and testing datasets from an initial dataset loaded in the current Stata session.

`get_train_test, dataname(`*data_name*`) split(`*shares*`) split_var(`*name*`)`
`rseed(`*integer*`)`

3.3.1 Options

`dataname(`*data_name*`)` specifies the name of the dataset open in the current Stata session. `dataname()` is required.

`split(`*shares*`)` requests to provide two numbers (ranging from 0 to 1) representing the shares of observations for the training and testing datasets. `split()` is required.

`split_var(`*name*`)` generates a flag variable distinguishing the training and testing observations. `split_var()` is required.

`rseed(`*integer*`)` specifies an integer seed to assure replication of results. `rseed()` is required.

3.3.2 Returns

The `get_train_test` command generates a training dataset as *data_name*`_train` and a testing dataset as *data_name*`_test`. They are automatically located in the current directory.

4 Application 1: Fitting a tree regression with train and test predictions

In this section, I present an illustrative application of the use of `r_ml_stata_cv` within a cross-section data structure. I use the popular Boston housing dataset (`boston.dta`) from the U.S. Census Service. The dataset contains information in 13 variables about a total of 506 Boston areas. Here I am interested in predicting the median value of owner-occupied homes in $1,000s in each individual area as a function of the remaining 12 predictors measuring house and neighborhood characteristics of the area, such as crime level, percentage of lower status of the population, and average number of rooms per dwelling.

In what follows, I show step by step how to implement a tree regression using `r_ml_stata_cv`.

- **Step 1**. Before starting, install Python (from version 2.7 onward) and the Python packages Scikit-learn, NumPy, Pandas, and SciPy. If you need help installing Python and its packages, refer to the Python webpage.[7] This command uses the `pylearn` package (Droste 2022) to check whether you have Python and all the needed dependencies installed on the machine. To install `pylearn`, type

```
. net install pylearn,
> from(https://raw.githubusercontent.com/mdroste/stata-pylearn/master/src)
```

- **Step 2**. Once you have Python and `pylearn` installed on your machine, you need to install `r_ml_stata_cv` from the *Stata Journal* website and look at the documentation file of the command to explore its syntax:

```
. help  r_ml_stata_cv
```

- **Step 3**. The command requires a training and a testing dataset. I form both datasets using the `get_train_test` command, inducing a split of 80% training and 20% testing observations:

```
. * Load initial dataset
. sysuse boston, clear
(Written by R.              )
. * Form the train and test datasets
. get_train_test, dataname("boston") split(0.80 0.20) split_var(svar)
> rseed(101)
```

7. Specifically, look at the Python installation page: https://realpython.com/installing-python.

- **Step 4**. We can now run `r_ml_stata_cv` with the `mlmodel(tree)` option in default mode (that is, with Scikit-learn's default options):

```
. * Form the target and the features
. global y "medv"
. global X "zn indus chas nox rm age dis rad tax ptratio black lstat"
. * Run tree regression in default mode
. use boston_train, clear
(Written by R.              )
. r_ml_stata_cv $y $X,
> mlmodel(tree) data_test("boston_test")
> default prediction("pred") seed(10)
  (output omitted)
------------------------------------------------------------------------------
Learner: Tree regression

Dataset information

Target variable = "medv"                    Number of features  = 12
N. of training units = 405                  N. of testing units = 101
N. of used training units = 405             N. of used testing units = 101
------------------------------------------------------------------------------

Parameters

Tree depth = largest tree possible
------------------------------------------------------------------------------

Validation results

MSE = mean squared error                    MAPE = mean absolute percentage
> error
Training MSE = 0                            Testing MSE = 49.644951
Training MAPE % = 0                         Testing MAPE % = 21.923632

------------------------------------------------------------------------------
```

Results show three panels.[8] The first panel provides information on the used data. Here we see that the command has used 405 training observations and 101 testing observations. The second panel displays the parameters at which the tree has been fit. For a tree, the main parameter is the depth, which in this case is the largest possible. The third panel presents validation results by contrasting in-sample and out-of-sample MSE and MAPE. Validation signals a strong presence of tree overfitting with training errors equal to 0, but with sizable testing errors.

Because I specified the `prediction()` option, the command has generated two variables: `pred`, containing predictions for both the testing and the training datasets, and `_train_index`, a flag variable indicating whether an observation is a training or a testing one. Note that the testing dataset is appended below the training dataset.

8. Before displaying output, both `r_ml_stata_cv` and `c_ml_stata_cv` display (in red) a panel of "Python warnings" concerning the current fit. For the sake of brevity, I do not report this panel of results here, but they can be useful sometimes, for example, to check whether convergence of certain algorithms is achieved.

The `default` option is appealing, but with it one cannot customize parameters nor run CV to find optimal parameter tuning. Thus, I rerun the command while allowing for a tree depth of size 3 (options `cross_validation()` and `n_folds()` are necessary in this case):

```
. * Run tree regression with specific tree depth
. capture rm cv.dta
. use boston_train, clear
(Written by R.              )
. r_ml_stata_cv $y $X,
> mlmodel(tree) data_test("boston_test")
> prediction("pred") tree_depth(3) cross_validation("cv")
> n_folds(5) seed(10)
  (output omitted)
------------------------------------------------------------------------------
Learner: Tree regression

Dataset information

Target variable = "medv"                    Number of features  = 12
N. of training units = 405                  N. of testing units = 101
N. of used training units = 405             N. of used testing units = 101
------------------------------------------------------------------------------

Cross-validation results

Accuracy measure = explained variance       Number of folds = 5
Best grid index = 0                         Optimal tree depth = 3
Training accuracy = .84580618               Testing accuracy = .2984019
Std. err. test accuracy = .65469445
------------------------------------------------------------------------------

Validation results

MSE = mean squared error                    MAPE = mean absolute percentage
> error
Training MSE = 14.502344                    Testing MSE = 40.720762
Training MAPE % = 16.141842                 Testing MAPE % = 21.355281

------------------------------------------------------------------------------
```

We have now an additional panel of results for the CV run over 5 folds. The optimal tree depth is trivially equal to the one we have fixed in advance (that is, 3). However, to appreciate the real value of using the `cross_validation("CV")` option, one can specify a grid of values for the tree depth and search for the optimal one:

```
. * Run tree regression with cross-validated tree depth
. capture rm cv.dta
. use boston_train, clear
(Written by R.              )
. r_ml_stata_cv $y $X,
> mlmodel(tree) data_test("boston_test")
> prediction("pred") tree_depth(1 2 3 4 5 6 7 8 9) cross_validation("cv")
> n_folds(5) seed(10) graph_cv
```

(*output omitted*)

```
Learner: Tree regression

Dataset information

Target variable = "medv"                    Number of features  = 12
N. of training units = 405                  N. of testing units = 101
N. of used training units = 405             N. of used testing units = 101

Cross-validation results

Accuracy measure = explained variance       Number of folds = 5
Best grid index = 1                         Optimal tree depth = 2
Training accuracy = .71966095               Testing accuracy = .4605888
Std. err. test accuracy = .35495267

Validation results

MSE = mean squared error                    MAPE = mean absolute percentage
> error
Training MSE = 24.584194                    Testing MSE = 31.301179
Training MAPE % = 19.328019                 Testing MAPE % = 20.389068
```

The optimal tree depth is 2, and this corresponds to a testing accuracy of 46% and a MAPE of 20%. As expected, this out-of-sample fit is better than in the previous cases. The use of option `graph_cv` allows one to obtain a graphical representation of the optimal solution. This is visible in figure 4, where the optimal index equal to 1 corresponds to a tree depth of 2 (because Python starts to count from 0, not from 1). The graph clearly shows the in-sample overfitting of our tree regression.

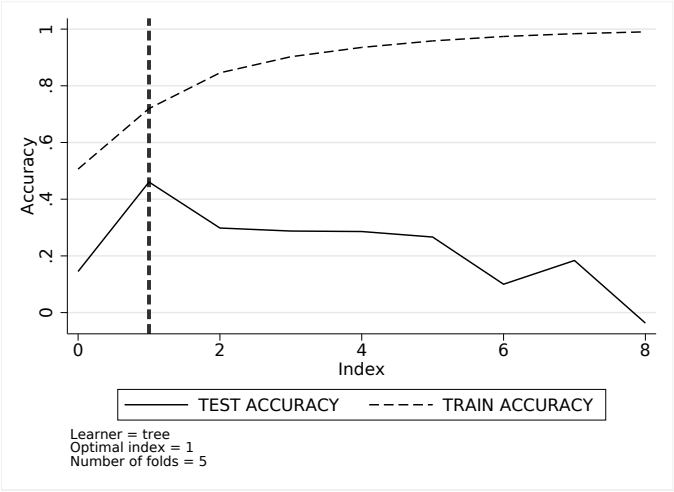


Figure 4. CV graph for a tree regression

Finally, we can easily access the main returns of this command:

```
. ereturn list
scalars:
          e(OPT_DEPTH) =  2
      e(TEST_ACCURACY) =  .4605887984894846
     e(TRAIN_ACCURACY) =  .7196609472314439
         e(BEST_INDEX) =  1
   e(SE_TEST_ACCURACY) =  .3549526699233333
           e(N_FOLDS) =  5
         e(Train_mse) =  24.58419420227465
        e(Train_mape) =  19.32801854899268
          e(Test_mse) =  31.30117890699467
         e(Test_mape) =  20.38906820177442
        e(N_features) =  12
       e(N_train_all) =  405
        e(N_test_all) =  101
      e(N_train_used) =  405
       e(N_test_used) =  101
macros:
           e(dep_var) : "medv"
```

5 Application 2: ML classification

In this application, I use the popular `iris.dta` to perform a classification exercise with `c_ml_stata_cv`. The dataset contains four features (length and width of sepals and petals) of 150 observations of three species of Iris (*Iris setosa*, *Iris virginica*, and *Iris versicolor*). The dataset is often used in data mining, classification and clustering examples, and to test ML algorithms.

I want to classify the three species of Iris based on the four available features. For this purpose, I compare a tree classification and a cross-validated random forest algorithm. I first load the initial dataset and generate the training and testing datasets (using 80% of observations for the former, and 20% of them for the latter):

```
. * Load initial dataset
. webuse iris, clear
(Iris data)
. * Form the train and test datasets
. get_train_test, dataname("iris") split(0.80 0.20) split_var(svar) rseed(101)
. * Form the target and the features
. global y "iris"
. global X "seplen sepwid petlen petwid"
```

Then, I run a cross-validated tree classification as follows:

```
. * Run tree classification
. capture rm cv.dta
. use iris_train, clear
(Iris data)
. c_ml_stata_cv $y $X,
> mlmodel(tree) data_test("iris_test")
> prediction("pred") tree_depth(1 2 3 4 5 6 7 8 9) cross_validation("cv")
> n_folds(5) seed(10)
  (output omitted)
------------------------------------------------------------------------------
Learner: Tree classification

Dataset information

Target variable = "iris"                    Number of features  = 4
N. of training units = 120                  N. of testing units = 30
N. of used training units = 120             N. of used testing units = 30
------------------------------------------------------------------------------

Cross-validation results

Accuracy measure = rate correct matches     Number of folds = 5
Best grid index = 5                         Optimal tree depth = 3
Training accuracy = .99345238               Testing accuracy = .93333333
Std. err. test accuracy = .05651942
------------------------------------------------------------------------------
```

```
Validation results

CER = classification error rate             Training CER = .03333333
Testing CER = .03333333
```

Results suggest that the optimal tree depth is 3, which generates a testing accuracy of 93% with a standard error (over the five iterations) of 0.06. This signals a good performance of this method.

Next, I fit a cross-validated random forest. The three parameters to tune are number of bootstrapped trees (`n_estimators()`), depth of each tree (`tree_depth()`), and maximum number of features to split on (`max_features()`), which are randomly drawn at each split.

```
. * Run random forest classification
. capture rm cv.dta
. use iris_train, clear
(Iris data)
. c_ml_stata_cv $y $X, mlmodel(randomforest) data_test("iris_test")
> tree_depth(2 4 6 8) n_estimators(50 100 150) max_features(3 6)
> prediction("pred") cross_validation("cv") n_folds(5) seed(10)
  (output omitted)
Learner: Random Forest classification

Dataset information

Target variable = "iris"                    Number of features  = 4
N. of training units = 120                  N. of testing units = 30
N. of used training units = 120             N. of used testing units = 30

Cross-validation results

Accuracy measure = rate correct matches     Number of folds = 5
Best grid index = 13                        Optimal tree depth = 4
Optimal n. of splitting features = 3        Optimal n. of trees = 50
Training accuracy = .99930556               Testing accuracy = .95
Std. err. test accuracy = .04859127

Validation results

CER = classification error rate             Training CER = 0
Testing CER = .06666667
```

Over the five folds, the average test accuracy rate is 95%, around two percentage points larger than the tree classification. The standard error is also comparable. The improvement is not an extraordinary one but nonetheless signals a greater ability of the random forest to predict the outcome. The validation results are also comparable, with the tree classifier unexpectedly doing slightly better. This is probably due to the too-small sample size of the validation dataset.

Finally, we can easily access all the stored results:

```
. ereturn list
scalars:
      e(OPT_MAX_DEPTH) =  4
   e(OPT_MAX_FEATURES) =  3
   e(OPT_N_ESTIMATORS) =  50
      e(TEST_ACCURACY) =  .95
     e(TRAIN_ACCURACY) =  .9993055555555556
         e(BEST_INDEX) =  13
   e(SE_TEST_ACCURACY) =  .0485912657903775
            e(N_FOLDS) =  5
          e(Train_err) =  0
           e(Test_err) =  .0666666666666667
         e(N_features) =  4
        e(N_train_all) =  120
         e(N_test_all) =  30
       e(N_train_used) =  120
        e(N_test_used) =  30
macros:
           e(dep_var) : "iris"
```

6 Application 3: ML classification of handwritten numerals

In this application, I focus on the classification of handwritten numerals, a task that can be encompassed within the larger set of image recognition studies. To this purpose, I use the popular Modified National Institute of Standards and Technology dataset. This database has been widely used to test and compare several ML classification algorithms (LeCun et al. 1998).

I use two datasets for training and then testing the ML algorithms implemented by `c_ml_stata_cv`: `mnist_train.dta`, containing 60,000 images of numerals from 0 to 9, and `mnist_test.dta`, containing 10,000 of these images. The dimension of each image is 28×28, implying a total number of pixels per image equal to 784. I stack the image pixels horizontally so that a single image is a row of the dataset represented by 784 values. Consequently, I have 784 variables, from `v1` to `v784`.[9]

9. Interested readers can access https: // nbalov.github.io / index.html, the webpage of Nikolay H. Balov, where he introduces a community-contributed command, `mlp2`, for specifying and learning a type of neural network, the multilayer perceptron, with two hidden layers. Balov performs an application of `mlp2` to the Modified National Institute of Standards and Technology dataset.

To speed up computation, and just for illustrative purposes, I consider only 1,000 images randomly drawn from the training dataset, and I compare all the classifiers available through `c_ml_stata_cv` at their default parameters. The overall estimation procedure takes less than one minute.

The following code implements all the necessary steps to compare the accuracy of these classifiers. Results in terms of accuracy in the test dataset are stored in the matrix `M` generated as empty at the beginning of the code and listed at the end. The code also plots and saves, for the images of the training dataset, the conditional probabilities that image i is the numeral j, that is, $\text{Prob}(y_i = j|\mathbf{x}_i)$, $i = 1, \ldots, N$. The plots of these probabilities provide us with a visual inspection of the quality of each numeral's classification. The entire code is set out below:

```
. * Load the dataset
. use mnist_train, clear
(Training MNIST: 28x28 images with stacked pixel values v* in [0,1] range)
. * Consider only a (random) subsample of the training dataset
. set seed 1010
. generate u=runiform()
. sort u
. keep in 1/1000
(59,000 observations deleted)
. save mnist_train2, replace
(file mnist_train2.dta not found)
file mnist_train2.dta saved
. * Set the "target" (y) and the "features" (X)
. global y "y"
. global X "v1-v784"
. * Set the learners to compare
. global LEARNERS tree randomforest boost regmult
> nearestneighbor neuralnet naivebayes svm multinomial
. * Set a matrix M that will contain the testing accuracy of each ML method
. global M: word count $LEARNERS
. matrix M=J($M,1,.)
. * Run the loop estimating the models and storing the testing accuracy
. local j=1
. foreach L of global LEARNERS {
  2. capture rm cv.dta
  3. preserve
  4. use mnist_train2, clear
  5. c_ml_stata_cv $y $X, mlmodel(`L') data_test("mnist_test")
> prediction("pred_`L'") default seed(10)
  6. matrix M[`j',1]=1-e(Test_err)
  7. keep pred_`L' _train_index y
  8. save pred_`L', replace
  9. local j=`j'+1
 10. restore
 11. }
  (output omitted)
```

```
. * Rename rows of M by learner
. matrix rownames M = $LEARNERS
. * Rename the column of M
. matrix colnames M = Accuracy
. * List matrix M (accuracy by learner)
. matlist M
```

	Accuracy
tree	.6702
randomforest	.8944
boost	.3805
regmult	.8682
nearestnei~r	.8692
neuralnet	.888
naivebayes	.6047
svm	.9146
multinomial	.8554

The best accuracy is reached by the SVM (with an accuracy of 91%), followed by random forest and neural network (both with an accuracy of 89%). Surprisingly, the multinomial logit also performs rather well with an accuracy of 85%.

Figure 5 displays the estimation of $\text{Prob}(y_i = j|\mathbf{x}_i)$, $i = 1, \ldots, N$, in the training sample using the SVM algorithm, where I indicate a training image with i and a numeral (from 0 to 9) with j.

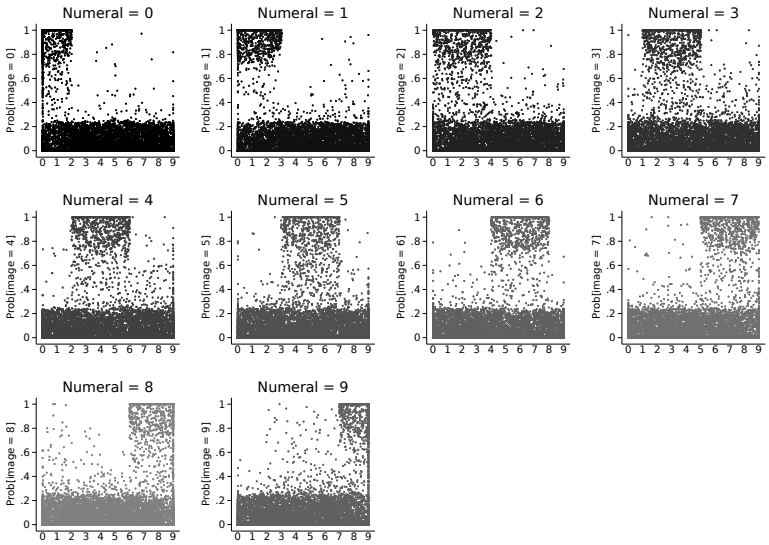


Figure 5. Classification of handwritten numerals; estimation of $\text{Prob}(y_i = j|\mathbf{x}_i)$, $i = 1, \ldots, N$, in the training sample using the SVM algorithm.
NOTE: i = training image; j = numeral.

The more the upper part of each figure is spread over all the numerals, the worse the quality of classification. For example, the numeral "0" seems very well classified compared with the numeral "9".

To better appreciate this, figures 6 and 7 illustrate the jitter plots, respectively, for the numerals "0" and "9". In figure 6, the probability that the numeral is correctly classified as a "0" is rather high compared with the probability that "0" is classified incorrectly.

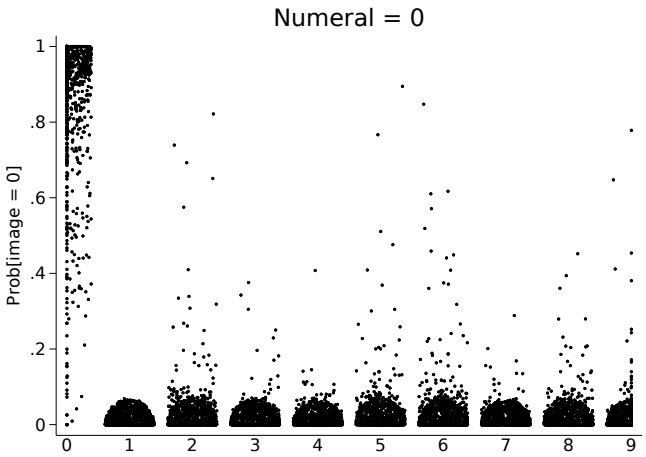


Figure 6. Classification of handwritten numerals. Estimation of $\text{Prob}(y_i = 0|\mathbf{x}_i)$, $i = 1, \ldots, N$, in the training sample using the SVM algorithm.

However, when we look at figure 7, we see a different situation. Although the probability of correctly classifying a "9" is high, the probability is also quite high of classifying it as a "4" or, to a lesser extent, as a "7". These results are reasonable; it seems probable that one can write a "9" and a "4" similarly, thus making correct classification more uncertain. A larger training sample might mitigate, though not eliminate, events like these.

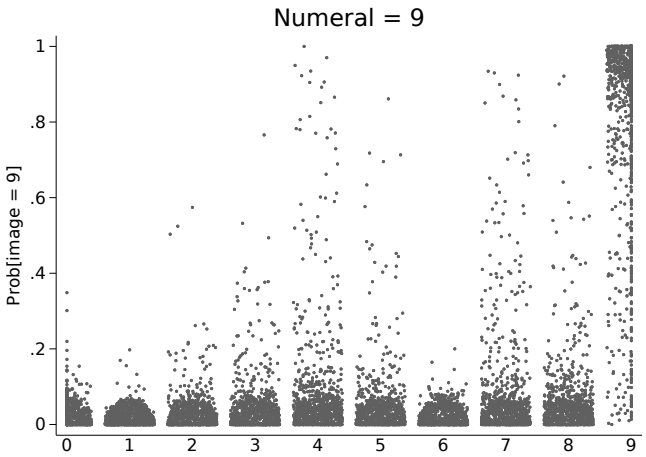


Figure 7. Classification of handwritten numerals. Estimation of $\text{Prob}(y_i = 9|\mathbf{x}_i)$, $i = 1, \ldots, N$, in the training sample using the SVM algorithm.

7 Application 4: Estimating CATEs

In the previous applications, I used ML algorithms for predictive purposes. The same algorithms, however, can be used to estimate causal effects. In this application, I will show how to estimate the CATE with the ML methods implemented via `r_ml_stata_cv`.

I use the popular LaLonde (1986) dataset `jtrain2.dta`, used by Dehejia and Wahba (1999) to evaluate various propensity-score matching methods. I am interested in estimating the effect of attending a job training administrated in 1976 (measured by the binary variable `train`, taking value 1 for treated and 0 for untreated) on real earnings in 1978 (variable `re78`) of a set of individuals in the United States. The dataset is made of 445 total observations, 185 treated and 260 untreated people.

Following the specification of Dehejia and Wahba (1999), our control variables are `age`, age in years; `agesq`, age squared; `educ`, years of schooling; `black`, an indicator variable for Black people; `hisp`, an indicator variable for being Hispanic; `married`, an indicator variable for marital status; `nodegr`, an indicator variable for high school diploma; `re74`, real earnings in 1974; `re74sq`, real earnings in 1974 squared; `re75`, real earnings in 1975; `unemp74`, an indicator variable for being unemployed in 1974; `unemp75`, an indicator variable for being unemployed in 1975; and `u74hisp`, an interaction between `unemp74` and `hisp`.

Under the assumption of unconfoundedness (also known as conditional independence), it is well known (Cerulli 2015) that the CATE takes on this form:

$$\text{CATE}(\mathbf{x}_i) = m_1(\mathbf{x}_i) - m_0(\mathbf{x}_i)$$

where $m_1(\mathbf{x}_i) = E(y_i|\mathbf{x}_i, T_i = 1)$ and $m_0(\mathbf{x}_i) = E(y_i|\mathbf{x}_i, T_i = 0)$ are two conditional expectations that can be estimated by either parametric or nonparametric methods, including any supervised ML method. Notice that, when y is binary, the CATE becomes the difference between two probabilities:

$$\text{CATE}(\mathbf{x}_i) = \text{Prob}(y_i = 1|\mathbf{x}_i, T_i = 1) - \text{Prob}(y_i = 1|\mathbf{x}_i, T_i = 0)$$

As known, the CATE is not identified by observation because the counterfactual treatment status of any observation is itself unidentified. To make this point clearer, table 2 sets out an example of an estimation of the CATE under unconfoundedness. The potential outcomes, y_1 and y_0, are not fully observed. More precisely, y_1 is known only for the treated and y_0 only for the untreated units. Similarly, $m_1(\mathbf{x})$ is estimated in-sample only for the treated units, and $m_0(\mathbf{x})$ is estimated in-sample only for the untreated units. Under unconfoundedness, however, one can impute the missing observations of $m_1(\mathbf{x})$ and $m_0(\mathbf{x})$ by taking the out-of-sample predicted values over the opposite group. This type of imputation of the missing counterfactual is called regression adjustment. Once $m_1(\mathbf{x})$ and $m_0(\mathbf{x})$ are all imputed, the CATE is obtained as their difference. In table 2, as a consequence, the symbol "." indicates a pure missing value, the symbol "x" indicates a value imputed in-sample by regression adjustment under unconfoundedness, and the symbol "*" indicates a value imputed out-of-sample by regression adjustment under unconfoundedness.

Table 2. Example of an estimation of the CATE based on unconfoundedness

ID	T	x	y	y_1	y_0	$m_1(\mathbf{x})$	$m_0(\mathbf{x})$	CATE
1	1	34	22	22	.	x	*	*
2	1	65	45	45	.	x	*	*
3	1	45	89	89	.	x	*	*
4	0	78	50	.	50	*	x	*
5	0	16	55	.	55	*	x	*
6	0	30	45	.	45	*	x	*

NOTE: . = missing value; x = value imputed in-sample by regression adjustment under unconfoundedness; * = value imputed out-of-sample by regression adjustment under unconfoundedness

The imputation of the values * and `x` can be carried out by ML. In theory, every method implemented by `c_ml_stata_cv` or `r_ml_stata_cv` is suitable for this end. The estimation procedure follows these steps:

1. Imputation of $m_1(\mathbf{x})$:
    a. Run a regression (or classification) of y on $\mathbf{x}$ using only the treated sample, and impute in-sample the values with symbol `x`.
    b. Using the previous fit, impute out-of-sample the values of the untreated sample with symbol *, thus obtaining $m_1(\mathbf{x})$ wholly filled in.
2. Imputation of $m_0(\mathbf{x})$:
    a. Run a regression (or classification) of y on $\mathbf{x}$ using only the untreated sample, and impute in-sample the values with symbol `x`.
    b. Using the previous fit, impute out-of-sample the values of the treated sample with symbol *, thus obtaining $m_0(\mathbf{x})$ wholly filled in.
3. Estimate CATE as the difference between the imputed $m_1(\mathbf{x})$ and the imputed $m_0(\mathbf{x})$.

Because our target variable (`re78`) is numerical and continuous, we are in a regression setting, which requires us to use the learners available through `r_ml_stata_cv`. The following code implements the learner-by-learner CATE estimation procedure outlined above and provides graphical representation of the main results:

```
. * Load the dataset
. sysuse jtrain2, clear
. * Set the main inputs
. global y "re78" // outcome
. global w "train" // binary treatment variable
. global X age agesq educ educsq married nodegree black
> hisp re74 re74sq re75 unem74 unem75 u74hisp // controls
. global LEARNERS "ols elasticnet tree randomforest boost nearestneighbor
> neuralnet svm"
. * Look at "teffects ra" result on ATE (it should match the ATE of OLS)
. teffects ra ($y $X) ($w), ate
  (output omitted)
```

```
. foreach L of global LEARNERS{
  2. * Load the dataset
. sysuse jtrain2, clear
  3. * Create the train and test datasets (treated = training dataset)
. preserve
  4. keep if $w==1
  5. save train_w11, replace
  6. restore
  7. preserve
  8. keep if $w==0
  9. save test_w10, replace
 10. restore
 11. * Load the training dataset and generate the predictions to obtain m1(x)
. use train_w11, clear
 12. r_ml_stata_cv $y $X, mlmodel(`L') data_test("test_w10")
> prediction("m1") default seed(10)
 13. * Save m1(x) in a dataset named "m1"
. preserve
 14. keep _train_index m1
 15. save m1, replace
 16. restore
 17. * Create the train and test datasets (untreated = training dataset)
. preserve
 18. keep if $w==0
 19. save train_w00, replace
 20. restore
 21. preserve
 22. keep if $w==1
 23. save test_w01, replace
 24. restore
 25. * Load the training dataset and generate the predictions to obtain m0(x)
. use train_w00, clear
 26. r_ml_stata_cv $y $X, mlmodel(`L') data_test("test_w01")
> prediction("m0") default seed(10)
 27. * Flip test and train datasets
. preserve
 28. keep if $w==0
 29. save top, replace
 30. restore
 31. preserve
 32. keep if $w==1
 33. save down, replace
 34. restore
 35. * Set the correct order of the training and testing observations
. clear
 36. append using down top
 37. * Save m0(x) in a dataset named "m0"
. keep _train_index m0
 38. save m0, replace
 39. * Merge m0 and m1
. merge 1:1 _n using m1
 40. * Generate CATE (by learner)
. generate CATE_`L' = m1 - m0
 41. * Save CATEs in dedicated datasets (by learner)
. keep CATE*
 42. save cate_`L', replace
 43. }
```

(*output omitted*)

```
. * Load "cate_ols" and merge it with all the other CATEs datasets
. use cate_ols, clear
. foreach L of global LEARNERS{
  2. capture drop _merge
  3. merge 1:1 _n using cate_`L'
  4. capture drop _merge
  5. }
```

(*output omitted*)

```
. * Generate the mean of CATEs
. generate CATE_mean=0
. foreach L of global LEARNERS{
  2. replace CATE_mean= CATE_mean + CATE_`L'
  3. }
```

(*output omitted*)

```
. replace CATE_mean=CATE_mean/8
(445 real changes made)
. sum CATE_mean
```

(*output omitted*)

```
. global m_cate=round(r(mean),0.01)
. * Plot jointly the distributions of CATEs
. twoway (kdensity CATE_mean, xline($m_cate, lp(dash))),
> xtitle("Average of CATEs over learners") ytitle("")
. graph export r_ml_stata_cv8.pdf, replace
file r_ml_stata_cv8.pdf saved as PDF format
. * Generate a dot plot of the results
. collapse CATE*
. xpose, clear varname
. rename v1 CATE
. rename _varname learner
. graph dot CATE, over(learner, sort(1)) ytitle("ATE")
> yline($m_cate,lp(dash) lc(black)) note(Average ATE = $m_cate)
```

The main outcomes of the previous code are illustrated in figures 8 and 9. Figure 8 shows the distribution of the average CATE obtained over all the eight learners. Looking at the plot, one can immediately see that the shape of the distributions of CATE is bell-shaped and centered to a value of 1.76.

Figure 9 sets out the average treatment effect (ATE) by sorting it according to the learner type. We can see that all the learners, except tree, SVM, and neural network, provide similar values of ATE. On average over the various learners, the ATE is equal to 1.76, which means that a treated individual in 1976 earns in 1978 $1,760 more than an untreated individual, signaling a positive effect of the policy in question.

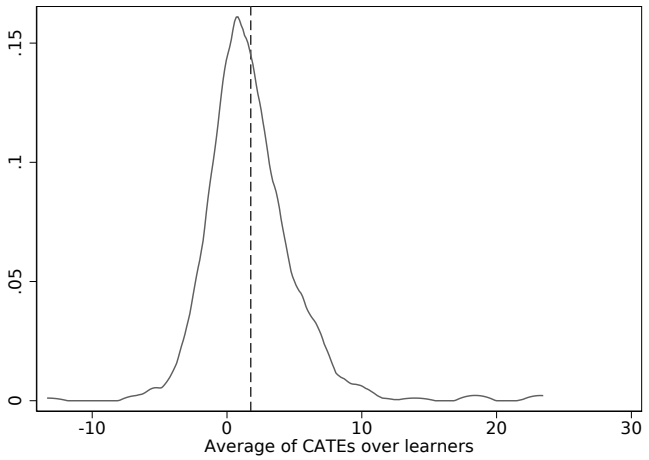


Figure 8. Estimation of the distribution of the CATEs using the ML methods implemented via `c_ml_stata_cv`; dashed vertical line indicates the ATE

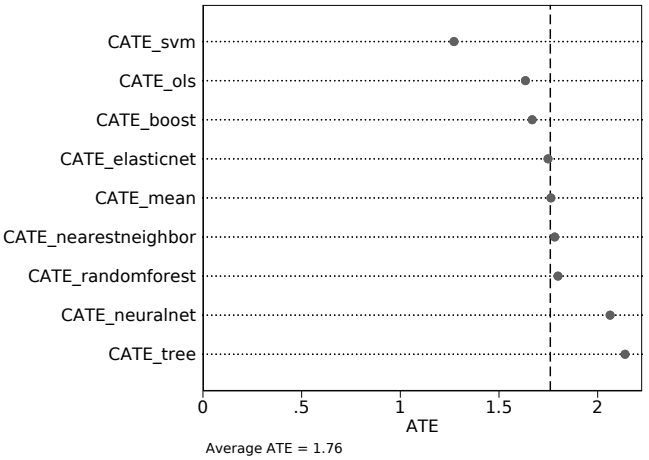


Figure 9. Estimated ATE by learner

8 Conclusion

In the last two decades, advances in statistical learning and computation have radically improved the prediction performance of targeted outcomes in nearly all scientific domains, including engineering, robotics, and artificial intelligence. ML has emerged as a new scientific paradigm to model outcomes and design pragmatic architectures for reasoned decision making in uncertain environments.

Thanks to the recent Stata/Python integration platform introduced within Stata 16, producing Stata routines to fit ML regression and classification has become relatively straightforward.

By exploiting this opportunity, I presented two related commands, `r_ml_stata_cv` and `c_ml_stata_cv`, for fitting popular ML models in both a regression and a classification setting. These commands provide hyperparameters' optimal tuning via K-fold CV using grid search, by wrapping the Python Scikit-learn API to perform CV and outcome/label prediction.

Compared with other popular statistical software, Stata has the advantage of being highly user-friendly and powerful for complex data management. Unfortunately, Stata has not yet embedded a built-in ML package, except for the lasso (which also includes the elastic net).

The commands presented herein thus aim to partly fill this gap by providing Stata users with two simple but powerful commands for fitting various ML methods. Further development of this work may include providing deep-learning Stata routines by wrapping into Stata the Python platforms Keras and TensorFlow.

9 Acknowledgments

Preliminary versions of these commands were presented at the 2021 Stata Conference held virtually on the 5th and 6th of August 2021. I thank the organizers and participants of this conference for their useful comments.

Improved versions were presented at the Italian Stata Conference 2022 held in Fiesole (Florence, Italy) on the 19th and 20th of May 2022. I wish to thank Una Louise Bell, Monica Gianni, Maurizio Pisati, and Rino Bellocco for the perfect organization of the conference and Jeff Pitblado for his inspiring presentation.

A special thanks goes to Maria Boutchkova for pushing me to develop the application of CATE estimated via ML methods.

Finally, I thank Stephen Jenkins and two anonymous referees for their detailed reading of the article, scrupulous check of the commands' reliability, and valuable suggestions for improvements.

10 Programs and supplemental materials

To install a snapshot of the corresponding software files as they existed at the time of publication of this article, type

```
. net sj 22-4
. net install pr0076     (to install program files, if available)
. net get pr0076         (to install ancillary files, if available)
```

The `r_ml_stata_cv` command is also available on the Statistical Software Components archive and can be installed directly in Stata by typing the command

```
. ssc install r_ml_stata_cv
```

11 References

Ahrens, A., C. B. Hansen, and M. E. Schaffer. 2020. lassopack: Model selection and prediction with regularized regression in Stata. *Stata Journal* 20: 176–235. https://doi.org/10.1177/1536867X20909697.

Boden, M. A. 2018. *Artificial Intelligence: A Very Short Introduction*. Oxford: Oxford University Press.

Bottou, L. 1998. Online algorithms and stochastic approximations. In *Online Learning and Neural Networks*, ed. D. Saad, 9–42. Cambridge, U.K.: Cambridge University Press. https://doi.org/10.1017/CBO9780511569920.003.

Cerulli, G. 2015. *Econometric Evaluation of Socio-Economic Programs: Theory and Applications*. Berlin: Springer.

———. 2021. Improving econometric prediction by machine learning. *Applied Economics Letters* 28: 1419–1425. https://doi.org/10.1080/13504851.2020.1820939.

Dehejia, R. H., and S. Wahba. 1999. Causal effects in nonexperimental studies: Reevaluating the evaluation of training programs. *Journal of the American Statistical Association* 94: 1053–1062. https://doi.org/10.1080/01621459.1999.10473858.

Droste, M. 2022. stata-pylearn. GitHub. https://github.com/mdroste/stata-pylearn.

Guenther, N., and M. Schonlau. 2016. Support vector machines. *Stata Journal* 16: 917–937. https://doi.org/10.1177/1536867X1601600407.

Hastie, T., R. Tibshirani, and J. Friedman. 2009. *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. 2nd ed. New York: Springer.

Kumar, S. 2020. Does ensemble models always improve accuracy? https://medium.com/analytics-vidhya/does-ensemble-models-always-improve-accuracy-c114cdbdae77.

LaLonde, R. J. 1986. Evaluating the econometric evaluations of training programs. *American Economic Review* 76: 604–620.

LeCun, Y., L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86: 2278–2324. https://doi.org/10.1109/5.726791.

Parisi, G. I., R. Kemker, J. L. Part, C. Kanan, and S. Wermter. 2019. Continual lifelong learning with neural networks: A review. *Neural Networks* 113: 54–71. https://doi.org/10.1016/j.neunet.2019.01.012.

Raschka, S., and V. Mirjalili. 2019. *Python Machine Learning: Machine Learning and Deep Learning with Python, scikit-learn, and TensorFlow 2*. 3rd ed. Birmingham: Packt Publishing.

Samuel, A. L. 1959. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development* 3: 210–229. https://doi.org/10.1147/rd.33.0210.

Schonlau, M. 2005. Boosted regression (boosting): An introductory tutorial and a Stata plugin. *Stata Journal* 5: 330–354. https://doi.org/10.1177/1536867X0500500304.

Schonlau, M., and R. Y. Zou. 2020. The random forest algorithm for statistical learning. *Stata Journal* 20: 3–29. https://doi.org/10.1177/1536867X20909688.

van der Laan, M. J., E. C. Polley, and A. E. Hubbard. 2007. Super learner. *Statistical Applications in Genetics and Molecular Biology* 6. https://doi.org/10.2202/1544-6115.1309.

van der Laan, M. J., and S. Rose. 2011. *Targeted Learning: Causal Inference for Observational and Experimental Data*. New York: Springer.

Varian, H. R. 2014. Big data: New tricks for econometrics. *Journal of Economic Perspectives* 28: 3–28. https://doi.org/10.1257/jep.28.2.3.

Zhou, Z.-H. 2012. *Ensemble Methods: Foundations and Algorithms*. Boca Raton, FL: Chapman & Hall/CRC.

About the author

Giovanni Cerulli is a senior researcher at the CNR-IRCrES, Research Institute on Sustainable Economic Growth, National Research Council of Italy, Rome. His research interest is in applied econometrics, with a special focus on causal inference and ML. He has developed original causal inference models and provided several implementations. He is currently editor in chief of the *International Journal of Computational Economics and Econometrics*.