Lab 4: Machine Learning, Prediction, Lasso, Python
Mark Schaffer
version of 03.10.2022

Introduction

This lab introduces you to a range of topics all related to "machine learning" (ML): prediction, overfitting, penalised regression, the Lasso and related estimators, etc. We also have a brief look at Python, a powerful general-purpose language. Stata has built-in Python integration, and there are many ML estimation routines available that are written in Python.

Our objective here is **prediction**. (In the time-series context, we would say **forecasting**.) Given some predictors (variables, or in the ML jargon, "features"), we want to predict an outcome.

Machine Learning is about constructing algorithms that can learn from the data. Supervised machine learning seeks to predict an outcome based on predictors. With unsupervised machine learning, there are only input variables, and the objective is dimension reduction (e.g., pattern recognition).

Our concern in this lab and in this course more generally is supervised ML, i.e., prediction.

Basic concepts

Our goal is good **out-of-sample** (OOS) prediction. We estimate the model on a dataset, and we want the model to have good predictive power for an OOS observation.

More formally, we want our estimator to perform well in terms of the **Mean Squared Prediction Error** (MSPE). The MSPE is the expected value of the squared error made by predicting *Y* for an observation **not** in the estimation data set:

$$MSPE = E[y^{OOS} - \hat{y}(\mathbf{X}^{OOS})]^2$$

where y is the variable to be predicted, $\mathbf{X}$ denotes the variables used to make the prediction, and ($\mathbf{X}^{OOS}$, y^{OOS}) are the out-of-sample values of $\mathbf{X}$ and y. The prediction $\hat{y}^{OOS} \equiv \hat{y}(\mathbf{X}^{OOS})$ uses a model estimated using the estimation data set, evaluated at $\mathbf{X}^{OOS}$.

OOS value of *Y*: $y^{OOS} = \beta_1 x_1^{OOS} + ... + \beta_k x_k^{OOS} + \varepsilon^{OOS}$
Prediction (^): $\hat{y}^{OOS} = \hat{\beta}_1 x_1^{OOS} + ... + \hat{\beta}_k x_k^{OOS}$ (note ^s on $\hat{\beta}$s)
Prediction error (^): $y^{OOS} - \hat{y}^{OOS} = (\beta_1 - \hat{\beta}_1) x_1^{OOS} + ... + (\beta_k - \hat{\beta}_k) x_k^{OOS} + \varepsilon^{OOS}$

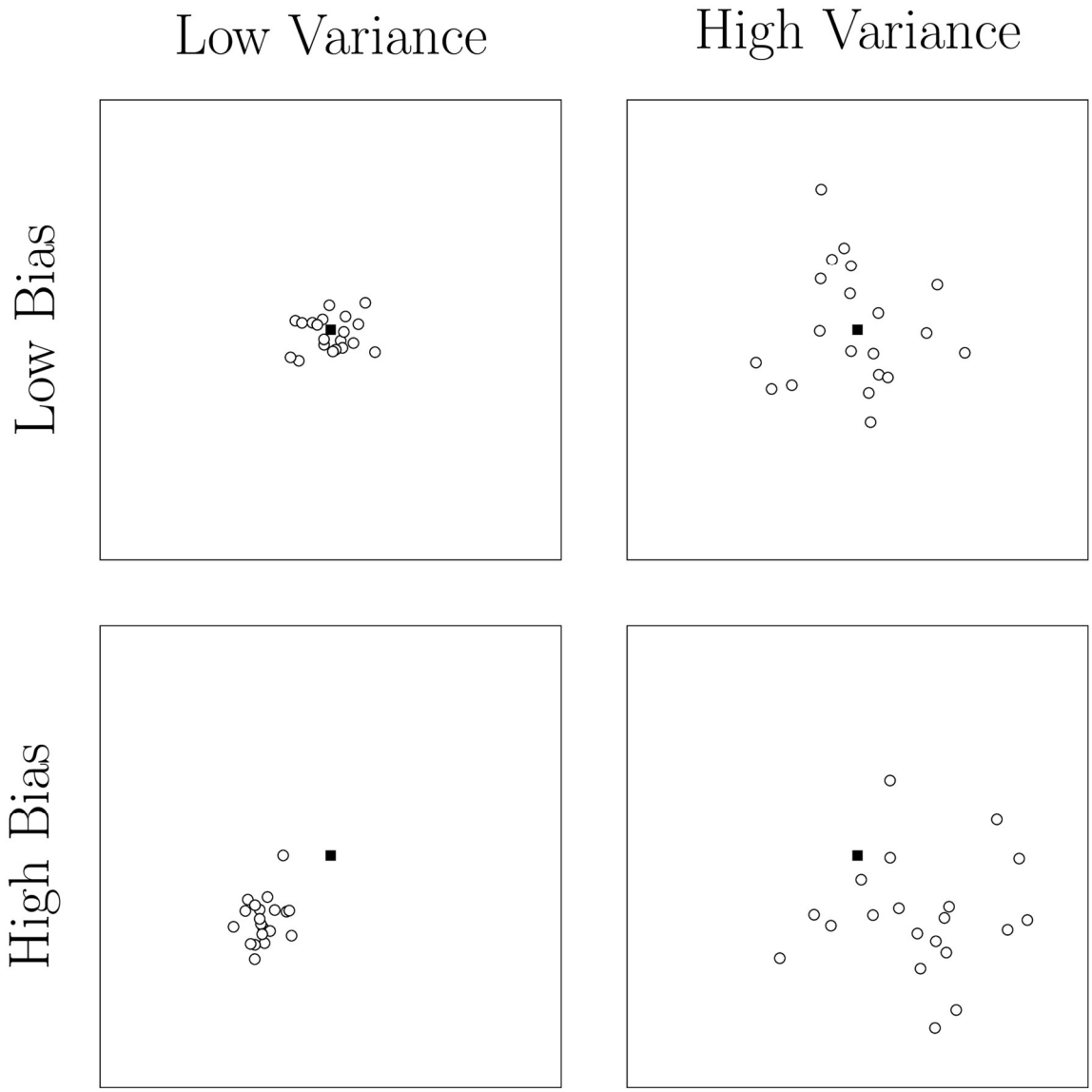
Let $E(\varepsilon^{OOS})^2 = \sigma_\varepsilon^2$ (the error variance). Then the MSPE for the standardised predictive regression model is

$$\begin{aligned} MSPE &= E[y^{OOS} - \hat{y}^{OOS}]^2 \\ &= \sigma_\varepsilon^2 + E\left[(\beta_1 - \hat{\beta}_1) x_1^{OOS} + ... + (\beta_k - \hat{\beta}_k) x_k^{OOS}\right]^2 \end{aligned}$$

The term σ_ε^2 is the MSPE of the "oracle forecast", i.e., the forecast if you knew the actual values of the coefficients $\beta_1, \ldots, \beta_k$. The MSPE for the oracle forecast is the lowest possible – you can't do any better.

The second term arises because the βs aren't known. We must use **estimates** obtained from the estimation sample.

We can decompose the expected value of the square of a random variable into its variance and the square of its mean. Applying this to the above, it implies that **biased** estimates $\hat{\beta}$ will make the MSPE bigger. But **noisy** estimates will **also** make the MSPE bigger. Given the choice between two estimators, *we might prefer one with the bigger bias in $\hat{\beta}$ if it has a small variance*.



The high-bias/low variance $\hat{\beta}$ has a **lower** MSPE than the low-bias/high variance $\hat{\beta}$ … even though it's biased! Tiny variance more than compensates. (Of course, the low-bias/low-variance $\hat{\beta}$ would be our first choice … but it might be impractical or unavailable.)

The **overfitting** problem: the MSPE for OLS (and many other estimators) is **increasing in *k***, the number of predictors. If we use too many predictors, we end up fitting the noise instead of the signal, and the OOS performance of our estimator will be poor.

Preparation

Install the following Stata programs and packages. Alternatively, use Stata's `search` command to search for the package (e.g., `search zval`) and then follow the links.

```
net install zval
ssc install lassopack
ssc install pystacked
```

To enable Python, open Stata and check whether it's already enabled:

```
python query
```

If Python is enabled, you're done. If Python is NOT enabled, first search for the Python executable:

```
python search
```

And then tell Stata where it is:

```
python set exec <pyexecutable>, permanently
```

In the above, replace `<pyexecutable>` with the full executable path. For example,

```
python set exec "C:/Anaconda3/python.exe", permanently
```

Task 1

Task 1 is a repetition and extension of an example we used in an earlier lab.

We use the Mroz dataset to estimate a simple wage equation. To gauge the OOS performance of the model, we loop through observations in the estimation sample. For each observation *i*, we do the following:

1. Estimate **excluding** observation *i*. Observation *i* will simulate an OOS observation.
2. Get the predicted value for observation *i*.
3. Repeat steps 1-2 for all observations *i*=1…,n.

At the end of the loop, we have *n* OOS predictions, one for each observation. This approach is called "Leave-One-Out" or LOO. Another term sometimes used is the "jackknife".

The last step in the process is to calculate the OOS errors, which are just the errors based on the actual values for *y* and the OOS predicted values. The variance of the OOS errors is the estimated MSPE; the standard deviation is the estimated RMSPE (root MSPE). (The do file uses Stata's `summarize` command to obtain these, so they incorporate a small-sample degrees-of-freedom adjustment.)

Looping through all observations this way can be computationally costly. Often we instead divide the dataset into groups of observations called "folds". The algorithm is very similar:

1. Estimate **excluding** all observations in fold *k*. Observations in fold *k* will simulate the OOS observations.
2. Get the predicted values for the observations in fold *k*.
3. Repeat steps 1-2 for all folds *k*=1…,#folds.

The example in the do file uses 4 folds.

Task 2: Overfitting illustrated

To illustrate the overfitting problem, we use a sample dataset from Jeffrey Wooldridge's various econometrics textbooks. The dataset covers house sales and the prediction task is to predict the log sales price based on some house characteristics: number of rooms and bathrooms, the age and the area of the house, and the land area of the property where the house is located. We treat bathrooms as a categorical variable (dummies for whether the house has 1, 2, 3 or 4 bathrooms); the other variables are treated as continuous. There are 142 observations in the sample we will use.

The simplest approach is just to include these variables in levels. If we include all levels, squares and interactions we have a second-order polynomial that we can think of as a 2nd-order Taylor approximation to an arbitrary nonlinear function. If we want, we can take this further and estimate 3rd-order or even 4th-order polynomial specifications.

Higher-order terms can create numerical instability problems because of scaling; standardising the variables to have mean 0 and standard deviation = 1 will help address this. It's also very common in ML practice to standardise variables. We do this for all the continuous variables.

The central problem for us is that the number of regressors grows rapidly. There are 7 level terms (but in practice only 6 because one of the dummies is omitted as the base category). The 2nd-order specification will have up to 21 terms; the 3rd-order specification will have up to 49 terms; the 4th-order specification will have up to 94 terms.

Some of these terms will be dropped because of collinearities, but it's clear that by the time we get to the full 4th-order specification we will have a serious overfitting problem.

Stata's factor variables make it easy to construct sets of interactions, but they don't correspond to how we present them here. For example,

```
regress lprice c.rooms##c.age
```

would use the level of `rooms`, the level of `age`, and the interaction of `rooms*age`, but it wouldn't include the squares of `rooms` and `age`.

For this reason, the code in the do file therefore constructs lists of variables and stores them in global macros. The macro `$p1` has all the level (1st-order) terms, the macro `$p2` has all the 2nd-order terms, etc.

For example, to estimate the 4th-order polynomial model, we say

```
global model_4 $p1 $p2 $p3 $p4
regress lrprice $model_4
```

First, estimate the four models using the **full sample**. See how well they fit the data.

Then, use the code that constructs OOS predictions using the LOO approach. See how well they do in terms of OOS prediction.

Which is your preferred specification and why?

Penalised regression: Lasso, Ridge and cross-validation

Recall the expression for the MSPE:

$$\begin{aligned} MSPE &= E[y^{OOS} - \hat{y}^{OOS}]^2 \\ &= \sigma_\varepsilon^2 + E\left[(\beta_1 - \hat{\beta}_1)x_1^{OOS} + \ldots + (\beta_k - \hat{\beta}_k)x_k^{OOS}\right]^2 \end{aligned}$$

If we "shrink" (bias) the estimated $\hat{\beta}$s towards zero, we introduce bias but reduce the variance. This can give us improved predictive performance because of the bias-variance trade-off we saw above.

The Lasso – the Least Absolute Selection and Shrinkage Operator – is an estimator that penalises the absolute value of the estimated coefficients.

OLS: choose $b_1 \ldots b_k$ to minimize the RSS

$$\min_{b_1 \ldots b_k} S^{OLS}(b) = \sum_{i=1}^{n} [y_i - (b_1 x_{1i} + \ldots + b_k x_{ki})]^2$$

Lasso: choose $b_1 \ldots b_k$ to minimize the **penalised** RSS

$$\min_{b_1 \ldots b_k} S^{Lasso}(b) = \sum_{i=1}^{n} [y_i - (b_1 x_{1i} + \ldots + b_k x_{ki})]^2 + \lambda \sum_{j=1}^{k} |b_j|$$

where $\lambda \sum_{j=1}^{k} |b_j|$ is the total penalty term and λ is the penalty "tuning parameter".

It turns out that the Lasso solution shrinks some coefficients all the way to zero. We say the Lasso solution is **sparse**.

This means the Lasso also does **model selection** for us – the coefficients that are not shrunk to zero correspond to the predictors that remain in the model.

The Lasso uses the L1 norm, i.e., the absolute value of the coefficients. An alternative estimator is the Ridge estimator. The Ridge estimator uses the L2 norm, i.e., the square of the coefficients. The Ridge solution is **dense**: all the predictors appear in the solution, but with varying degrees of shrinkage.

The tuning parameter λ determines the degree of penalisation – a larger λ means a bigger penalty and fewer predictors included in the model. At a high enough level of λ, no predictors will be included; if λ is small enough, all predictors will be included.

The most common method for choosing λ is cross-validation. This is an extension of the algorithm we used above for estimating the MSPE.

***K*-fold cross-validation**, for the case $K = 5$:

A. Assemble a list (grid) of λs.
B. For each λ in the list:
    1. Split the data into 5 subsamples. Use 4 subsamples as the "training" (estimation) data and set one aside for calculating the OOS errors.
    2. Estimate the model on 80% of the data and use it to predict and get the OOS $\hat{\varepsilon}$s for the remaining 20%.
    3. Repeat this 5 times, each time using a different subsample for the OOS errors and the rest as the training data.
    4. Using the full 5 sets of OOS $\hat{\varepsilon}$s to obtain the estimated $\widehat{MSPE}$ for the given λ.
C. Choose the λ that generated the smallest $\widehat{MSPE}$.

Task 3: The Lasso and cross-validation

We illustrate the Lasso using the house price example and a 2nd-order polynomial. The command we use is `lasso2`, from the package `lassopack`.

The following shows the Lasso path, i.e., what happens as the Lasso penalty decreases. Variables enter (and sometimes leave and later re-enter) the Lasso solution. Note that `lasso2` automatically standardises the predictors. The `lglmnet` option means use the parameterisation used by the R package glmnet.

```
lasso2 lrprice                                                    ///
      i.baths##c.rooms##c.lland##c.larea                          ///
            i.baths#(                                             ///
            c.rooms#c.rooms                                       ///
            c.lland#c.lland                                       ///
            c.larea#c.larea                                       ///
            ), lglmnet
```

Next, show the Lasso solution for two different values of λ. The first column of the output is the Lasso solution; the second column is OLS using the Lasso-selected predictors. Which of the two do you expect to select more variables to be in the model?

```
lasso2 lrprice                                                  ///
      i.baths##c.rooms##c.lland##c.larea                        ///
            i.baths#(                                           ///
            c.rooms#c.rooms                                     ///
            c.lland#c.lland                                     ///
            c.larea#c.larea                                     ///
            ), lglmnet lambda(0.2)
lasso2 lrprice                                                  ///
      i.baths##c.rooms##c.lland##c.larea                        ///
            i.baths#(                                           ///
            c.rooms#c.rooms                                     ///
            c.lland#c.lland                                     ///
            c.larea#c.larea                                     ///
            ), lglmnet lambda(0.001)
```

Finally, use the **cvlasso** command from **lassopack** to do 5-fold cross-validation to choose an optimal λ. Note that observations are allocated randomly to folds, so you need to set the random number seed if you want the results to be replicable. The option **lopt** means estimate using the full sample and the λ that minimised the MSPE.

```
cvlasso lrprice                                                 ///
      i.baths##c.rooms##c.lland##c.larea                        ///
            i.baths#(                                           ///
            c.rooms#c.rooms                                     ///
            c.lland#c.lland                                     ///
            c.larea#c.larea                                     ///
            ), lglmnet nfolds(5) lopt seed(42)
```

Stata+Python

Python is a powerful general-purpose programming language in wide use in industry as well as in academia that is often used in ML applications.

A commonly-used Python ML package is scikit-learn (or sklearn): https://scikit-learn.org

Opening Python in Stata and working in it is very like using Mata. Type

```
python
```

at the Stata prompt to get the Python **>>>** prompt, and type **end** at the Python prompt to return to Stata.

Using a do file with Python is also very like using it with Mata. In the do file:

```
<Stata commands>
python:
<Python commands>
end
<Stata commands>
```

And the drawback to this is the same as with Mata: **you can't run a do file from Python**. This means you either have to run the entire do file from Stata, or use Python interactively and copy/paste selected lines directly into Python as needed.

Some miscellaneous Python remarks:

- Comments begin with the symbol **#**.

- Indentation matters for syntax! If you write a loop, for example, the body of the loop has to be indented.

- Python installations typically include widely-used packages such as sklearn, numpy (numeric Python), etc.

- Your Python program will need to import packages as needed. The packages can be given a shorthand name, e.g.,

  ```
  import numpy as np
  ```

  means you can use `np.shape(.)` instead of `numpy.shape(.)`.

- Stata's Python API is called sfi: https://www.stata.com/python/api17/index.html. We use the functions `Data.get(.)` and `Data.store(.)` to move data from Stata into Python and back into Stata.

- The example in the do file uses numpy **arrays**. An array is a generalisation of a matrix. Python comes from the computer science realm, and so indexing of array elements **starts with zero** (and not with 1, like in Matlab or Mata).

- Python/sklearn routines are object-oriented. First you set up a model, then you fit it (estimate), and after fitting, the model object has "attributes" that include the estimated coefficients, etc.

Task 4: Machine learning using Stata+Python

In this task we illustrate how to use the sklearn package via Stata to estimate the house price model.

Basic structure of the do file:

1. Start in Stata.

2. Enter Python.

3. Load/define Python packages and associated routines.

4. Load the predictors in levels into Python using `Data.get(.)`. Call this array `Xlevels`. Load the outcome variable into Python. Call this array `y`.

5. Illustrate some miscellaneous numpy functions.

6. Set up the Lasso (or Ridge) model, e.g.,

```
model = LassoCV(normalize=True,fit_intercept=True,cv=5)
```

   This creates an object called `model` in Python.

7. Create the array of predictors `X` for estimation based on `Xlevels`. We use a utility from sklearn that lets us set up polynomials very easily. (You can change 4 to 2 or whatever you want in the code.)

```
poly = PolynomialFeatures(degree=4)
X = poly.fit_transform(Xlevels)
```

8. Estimate (fit) the model:

```
model.fit(X,y)
```

9. Report some estimation results, e.g.,

```
b = model.coef_
b
```

10. Get the predicted values and residuals:

```
yhatvalues = model.predict(X)
ehatvalues = y - yhatvalues
```

11. Put the predicted values and residuals into Stata variables using `Data.store(.)`.

12. Return to Stata.