Notes for ISL Chapter 6: Linear Model Selection and Regularization

Justin Burruss

2025-12-07

Background

The Python code and notes below are for the ISL study group. This is to try out some of the concepts from Chapter 6 on the `Credit` data set. The document was created in RMarkdown with the Python code running via the `reticulate` library plus a little LaTeX.

Our ground rule: when implementing concepts from the chapter, just use basic Python + Pandas + NumPy. It's OK to use more when visualizing or evaluating results.

Best Subset Selection

As the book points out, given p candidate predictors, we have 2^p possible combinations to go through to do best subset selection. That assumes a purely additive model with no interactions between terms. If we allow for interactions we have more combinations, for example given two predictors X_1 and X_2 we $X_1 + X_2$, $X_1 \cdot X_2$, X_1/X_2, X_2/X_1 when we allow multiplication and ratios. So to test out best subset we'll use the `Credit` data set (having just a few hundred rows of simulated data) and limit our possible predictors to just the following variables.

`Income` Income in thousands of dollars.

`Limit` Credit limit in dollars.

`Rating` Credit rating, higher is better.

`Student` `Yes` for students, `No` otherwise.

`Cards` Number of credit cards.

Rather than typing out all of those combinations we'll create a little builder class `ModelBuilder` to build expressions such as `X[:, 1] + X[:, 2] * X[:, 3]` systematically.

`ModelBuilder` class

Here's our plan for `ModelBuilder` with the minimal features we need for the task at hand.

- Create the builder using just the `Y`
- Build expressions by adding new `X` columns one at a time
  - Only support numeric `X`
  - Automatically distinguish binary `X` columns when added

- Provide a method for getting the model expressions
  - Optional `dims` parameter to limit expressions returned
  - Always named `X`
- Support `+`, `*`, and `/` interactions for numeric variables
  - For dummy variables, only apply `+`

As always, we'll stick with basic Python + Pandas + NumPy.

```
import pandas as pd
import numpy as np
from scipy.optimize import minimize

class ModelBuilder:
    """Class to build models given Y and Xs"""
    def __init__(self, Y):
        self.Y = Y
        self.Columns = ['X[:, [0]]']
        self.VarTypes = ['Ones']
        self.X = np.ones(Y.shape[0]).reshape(-1, 1)
        self.I = 0
        self.Models = ['X[:, [0]]']
        self.Dims = [1]

    def models(self, dims=None):
        """Return models having specified dimension"""
        if dims is not None:
            return [model for model, dim in zip(self.Models,
                                                self.Dims) if dim in dims]
        return self.Models

    def expr(self, dims=None):
        """Return an expression for eval() of the models"""
        models = self.models(dims)
        return [f"np.column_stack(({s.replace('+', ',')}))" if '+' in s
            else s.replace('+', ',') for s in models]

    def add(self, x):
        """Add another independent variable"""

        # cap at 9 (512 to ~0.5M models)
        if self.I >= 9:
            print("Cap reached")
            return

        # add the new column to X
        self.X = np.column_stack((self.X, x))
        self.I += 1

        # record the type of variable
        if np.array_equal(np.unique(x), [0, 1]):
            self.VarTypes.append('Dummy')
        else:
```

```
            self.VarTypes.append('Numeric')

        # create a save the column name
        self.Columns.append('X[:, {0}]'.format(self.I))

        # add new models where new var is additive
        for i in range(len(self.Models)):
            self.Models.append(self.Models[i]+' + '+self.Columns[-1])

        # for numerics, add interaction terms starting with second variable
        if self.I > 1 and self.VarTypes[-1] == 'Numeric':
            for i in range(1, self.I):
                if self.VarTypes[i] == 'Numeric':

                    # build from models that don't already have our A or B
                    models = [model for model in self.Models if
                              self.Columns[i] not in model and
                              self.Columns[-1] not in model]

                    # interaction: A * B
                    newmodels = [model + ' + ' + self.Columns[i] + ' * ' +
                                 self.Columns[-1] for model in models]

                    # interaction: A / B
                    newmodels.extend([model + ' + ' + self.Columns[i] + ' / ' +
                                 self.Columns[-1] for model in models])

                    # interaction: B / A
                    newmodels.extend([model + ' + ' + self.Columns[-1] + ' / ' +
                                 self.Columns[i] for model in models])
                    self.Models.extend(newmodels)

        # refresh dimensions for our models & weights
        self.Dims = [model.count('+') + 1 for model in self.Models]
```

Note in the code above that putting the brackets around the zero in the intercept term `X[:, [0]]` ensures that NumPy does not flatten it to 1D. Likewise, the `reshape(-1, 1)` is ensuring that our starter column is interpreted as a column of 400 1-D vectors, not flattened.

Loading the data set

The Pandas library ("Panda" as in "**Pan**el **da**ta") makes it easy to load the CSV and offers some functionality similar to `data.frame` or `data.table` in R.

```
# read CSV
c = pd.read_csv(r"E:\docs\Classes\ISL\Credit.csv")
```

We'll re-encode `Student` as `Yes=1`, `No=0`.

```
# encode Student as Yes=1, No=0
c['Student'] = c['Student'].map({"Yes":1, "No":0})
```

Intercept-only model

Let's take it one step at a time and start with just an intercept term, i.e., just set $\hat{Y}$ to the arithmetic mean of Y, $\bar{Y}$.

Our response variable is `Balance`, so the idea here with the simulated data is that we should be able to better estimate credit card balances if we're given the predictors (the credit card line limits, the number of cards, etc.).

```
# encode Student as Yes=1, No=0
# response variable
Y = c['Balance'].values
```

We'll use mean squared error (MSE) to measure the quality of our estimates. Our estimate is just fitted weights (we'll use `w` for weights) times the terms of our model (which we'll code as `X`). The use of `@` below is equivalent to `X.dot(w)`.

```
def mse(w, X, Y):
    """Return MSE Y vs Y_hat"""
    Y_hat = X @ w
    MSE = np.mean((Y_hat - Y)**2)
    return(MSE)
```

So if this all works as expected, we ought to fit the intercept-only model such that our single weight coefficient equals the mean (i.e., $\bar{Y}$), and our MSE ought to equal the variance $Var(Y)$. How do we know MSE should equal variance? The MSE of using some estimate $\hat{Y}$ is:

$$MSE(Y, \hat{Y}) = \frac{1}{n}\sum_{i=1}^{n}[(Y_i - \hat{Y}_i)^2]$$

If we simply use $\bar{Y}$ for our estimate for every Y_i that comes out as:

$$MSE(Y, \bar{Y}) = \frac{1}{n}\sum_{i=1}^{n}[(Y_i - \bar{Y})^2] \stackrel{\text{def}}{=} Var(Y)$$

Now let's create our `ModelBuilder` using the response variable, fit the intercept-only model on our whole population, and report the coefficient and MSE to validate that the class works as expected in the smallest case.

```
# build models for our Xs
M = ModelBuilder(Y)

# our built-up X
X = M.X

# test ModelBuilder (should just be same as X)
Xn = eval(M.models()[0])

# fit the model
result = minimize(mse, [0], args=(Xn, Y))

# report the result
print("Coeff =", result.x, ", MSE =", result.fun)
```

```
## Coeff = [520.01527457] , MSE = 210849.77977507538
```

Now let's validate.

```
print("np.mean(Y) =", np.mean(Y), ", np.var(Y) =",np.var(Y, ddof=0))
```

```
## np.mean(Y) = 520.015 , np.var(Y) = 210849.779775
```

They tie.

Models

Let's go ahead and add our predictors then check our 1-variable case. Per the text, the most predictive single variance will be `Rating`.

```
M.add(c['Income'].values)
M.add(c['Limit'].values)
M.add(c['Rating'].values)
M.add(c['Student'].values)
M.add(c['Cards'].values)

# our built-up X
X = M.X

# only run on intercept + 1 variable, should select Rating [Variable 3]
exprs = M.expr([2])
models = M.models([2])
ws = []
MSEs = []
for expr in exprs:

    # Xn
    Xn = eval(expr)

    # weights, one per column of Xn
    wn = np.zeros(Xn.shape[1])

    # minimize
    result = minimize(mse, wn, args=(Xn, Y))
    ws.append(result.x)
    MSEs.append(result.fun)

print(models[MSEs.index(min(MSEs))], "had train MSE", np.round(min(MSEs), 1))
```

```
## X[:, [0]] + X[:, 3] had train MSE 53587.8
```

We did indeed select the third variable.

Next, let's create a function for cross validation and really train and test the candidate models.

```
# cross-fold validation

def cvk(X, Y, k=5, seed=2025):
    """Return mean test MSE from k-fold"""
    indices = np.arange(len(Y))
    rng = np.random.default_rng(seed)
    rng.shuffle(indices)
    folds = np.array_split(indices, k)
    w = np.zeros(X.shape[1])
    MSEs = np.zeros(k)

    for i in range(k):

        # for each test fold, the other folds serve as training data
        test_fold = folds[i]
        train_fold = np.concatenate([folds[j] for j in range(k) if j != i])

        # train the model
        result = minimize(mse, w, args=(X[train_fold], Y[train_fold]))
        w = result.x

        # test the fit model
        MSEs[i] = mse(w, X[test_fold], Y[test_fold])

    return np.mean(MSEs)

exprs = M.expr()
models = M.models()
MSEs = []
for expr in exprs:

    # Xn
    Xn = eval(expr)

    # 5-fold cross validation
    MSEs.append(cvk(Xn, Y, 5))

print(models[MSEs.index(min(MSEs))], "had 5-fold CV MSE",
      np.round(min(MSEs), 1))
```

```
## X[:, [0]] + X[:, 1] + X[:, 2] + X[:, 4] + X[:, 5] / X[:, 3] had 5-fold CV MSE 7945.0
```

Interestingly, a model that had an interaction term won out. Was it much better than just pure additive? Let's look that up.

```
i = models.index('X[:, [0]] + X[:, 1] + X[:, 2] + X[:, 3] + X[:, 4] + X[:, 5]')
print(MSEs[i])
```

```
## 10105.14643996497
```

So it was much more predictive to use `Cards / Rating` than `Cards + Rating`.

What were our coefficients?

```
# what coeffs?
Xn = eval(exprs[MSEs.index(min(MSEs))])
w = np.zeros(Xn.shape[1])
result = minimize(mse, w, args=(Xn, Y))
for coeff in result.x:
    print("{0:10.4f}".format(coeff))
```

```
##  -643.8254
##    -8.3931
##     0.2934
##   433.7179
## 10950.8950
```

So we have a negative intercept, `Income` came out negative, `Limit` positive, `Student` positive, and `CardsToRatingRatio` positive. So higher income borrowers would appear to carry less of a balance, higher limits mean higher balances, students carry higher balances, and borrowers with more cards per rating have higher balances. This may be a simulated data set, but these all seem reasonable.

Forward Stepwise Selection

We can demonstrate forward stepwise, though since we've already got the MSEs we'll just loop through and read the already-calculated MSEs. Let's skip the intercept-only case as we've seen that already.

```
# data structure for forward stepwise results, additive models only
i = [i for i, m in enumerate(M.models()) if '/' not in m and '*' not in m]
AdditiveModels = pd.DataFrame({

    # Model: string showing the right hand side of Y_hat =
    'Model': [M.models()[i] for i in i],

    # TermCnt: the number of terms, e.g., 1 for intercept-only
    'TermCnt': [M.Dims[i] for i in i],

    # Vars: bitmask for variables included, e.g., 0 for intercept-only
    'Vars' : [
        sum(2**(int(x) - 1) for x in M.models()[i] if '1' <= x <= '5')
        for i in i
    ],

    # MSE: mean squared error (which we already calculated)
    'MSE': [MSEs[i] for i in i],

    # Step: the latest step for this model
    'Step': int(0)
    })

# skip intercept-only
AdditiveModels['Step'] = (AdditiveModels['Vars'] > 0).astype(int)
for i in range(1, max(AdditiveModels['TermCnt'])):

    # consider models in the current step
```

```
    Check = AdditiveModels[(AdditiveModels['TermCnt'] == i+1) &
                           (AdditiveModels['Step'] >= i)]

    # display our progress
    print("\nStep {0}: {1} possible model(s), checking {2}" \
          .format(i, sum((
              (AdditiveModels['TermCnt'] >= i+1) &
              (AdditiveModels['Step'] >= i))), len(Check)))

    # select the model with the best (i.e., lowest) MSE
    Best = Check.loc[Check['MSE'].idxmin()]
    print("Step {0} Best:\n{1}".format(i, Best[['Model', 'MSE']].to_string()))

    # models with all the variables we've selected so far make it to next step
    AdditiveModels['Step'] += (
            (AdditiveModels['Vars'] &
             Best['Vars']) == Best['Vars']).astype(int)
```

```
##
## Step 1: 31 possible model(s), checking 5
## Step 1 Best:
## Model    X[:, [0]] + X[:, 3]
## MSE             53926.033315
##
## Step 2: 15 possible model(s), checking 4
## Step 2 Best:
## Model    X[:, [0]] + X[:, 1] + X[:, 3]
## MSE                      26703.467599
##
## Step 3: 7 possible model(s), checking 3
## Step 3 Best:
## Model    X[:, [0]] + X[:, 1] + X[:, 3] + X[:, 4]
## MSE                               10870.202493
##
## Step 4: 3 possible model(s), checking 2
## Step 4 Best:
## Model    X[:, [0]] + X[:, 1] + X[:, 2] + X[:, 3] + X[:, 4]
## MSE                                        10508.2628
##
## Step 5: 1 possible model(s), checking 1
## Step 5 Best:
## Model    X[:, [0]] + X[:, 1] + X[:, 2] + X[:, 3] + X[:,...
## MSE                                      10105.14644
```

Just like we saw in table 6.1 of the text, forward stepwise starts with `Rating`, then `Income + Rating`, then `Income + Rating + Student`, and finally `Income + Limit + Rating + Student`. Notice how, just as in the book, our forward stepwise did *not* lead us to the optimal 4-variable model.

```
# actual best 4-variable (5-term)
print(AdditiveModels[['Model', 'MSE']].loc[
    AdditiveModels['TermCnt'] == 5].loc[
        AdditiveModels['MSE'].idxmin()].to_string())
```

```
## Model    X[:, [0]] + X[:, 1] + X[:, 2] + X[:, 4] + X[:, 5]
## MSE                                                10084.437628
```

Dimension Reduction

The text introduces the idea of using principal component analysis (PCA) to construct the first M principal components $Z_1, Z_2, ..., Z_M$ for use in a principal components regression (PCR). Let's see if we can hand-code both a PCA and a PCR.

How shall we validate? Figure 6.20 in the book shows the outcome for the `Credit` data set: it's actually not particularly useful for these data, with cross-validation MSEs looking to be in the 80,000s for the 1-5 component range (we beat that with just two features). Rather than just eyeballing, we will use `PCA` from `sklearn.decomposition` to validate.

The text recommends we standardize the predictors so that they all have the same adjusted variance and offers up the following formula to scale all variables to have variance of 1.

$$\tilde{X}_{ij} = \frac{X_{ij}}{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(X_{ij} - \bar{X}_j)^2}}$$

In Python we can confirm this scales as required. We use `X[:, 1:]` to skip our initial column of ones.

```
print("Variance Before = ",
      [f"{v:10.2f}" for v in (X[:, 1:]).var(axis=0)],
      "\nVariance After  = ",
      [f"{v:10.2f}" for v in (X[:, 1:]/X[:, 1:].std(axis=0, ddof=0)).var(axis=0)])
```

```
## Variance Before =  ['   1239.05', '5314462.46', '  23879.71', '      0.09', '      1.88']
## Variance After  =  ['      1.00', '      1.00', '      1.00', '      1.00', '      1.00']
```

Let's take that one step further and mean-center our `X` in addition to scaling to unit variance. We will use the mean-centered and scaled data to construct a correlation matrix. If $\bar{X}_j$ is our mean for column j, then our centered and scaled data $\tilde{X}$ can be found via

$$\tilde{X}_{ij} = \frac{X_{ij} - \bar{X}_j}{\sqrt{\frac{1}{n}\sum_{i=1}^{n}(X_{ij} - \bar{X}_j)^2}}$$

The correlation matrix R_x is a matrix of the expected values of the variances and covariances which we can obtain by computing the pairwise column dot products then dividing by the number of observations.

$$R_X = \begin{bmatrix} E[\tilde{X}_1^2] & E[\tilde{X}_1\tilde{X}_2] & E[\tilde{X}_1\tilde{X}_3] & \cdots & E[\tilde{X}_1\tilde{X}_p] \\ E[\tilde{X}_2\tilde{X}_1] & E[\tilde{X}_2^2] & E[\tilde{X}_2\tilde{X}_3] & \cdots & E[\tilde{X}_2\tilde{X}_p] \\ E[\tilde{X}_3\tilde{X}_1] & E[\tilde{X}_3\tilde{X}_2] & E[\tilde{X}_3^2] & \cdots & E[\tilde{X}_2\tilde{X}_p] \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ E[\tilde{X}_p\tilde{X}_1] & E[\tilde{X}_p\tilde{X}_2] & E[\tilde{X}_p\tilde{X}_3] & \cdots & E[\tilde{X}_p^2] \end{bmatrix} = \frac{1}{n}\tilde{\mathbf{X}}^\intercal\tilde{\mathbf{X}}$$

That's easy enough in Python with the built-in `mean()` and `std()` functionality of NumPy.

```
X_std_scl = (X[:, 1:] - X[:, 1:].mean(axis=0)) / X[:, 1:].std(axis=0, ddof=0)
X_corr = X_std_scl.T @ X_std_scl / X_std_scl.shape[0]
```

Above would work just fine, but we want to validate against `PCA` from `sklearn.decomposition`, and that uses sample standard deviation, so let's alter our approach. This means replacing $\frac{1}{n}$ with $\frac{1}{n-1}$ in our $\tilde{X}$.

$$\tilde{X}_{ij} = \frac{X_{ij} - \bar{X}_j}{\sqrt{\frac{1}{n-1}\sum_{i=1}^{n}(X_{ij} - \bar{X}_j)^2}}$$

Similarly, we alter our correlation matrix to replace $\frac{1}{n}$ with $\frac{1}{n-1}$. Let's use a lowercase r now.

$$r_X = \frac{1}{n-1}\tilde{\mathbf{X}}^\intercal\tilde{\mathbf{X}}$$

To code it up we just change the call to `std()` to include `ddof=1` and update the correlation matrix such that we divide by $n-1$ instead of n.

```
X_std_scl = (X[:, 1:] - X[:, 1:].mean(axis=0)) / X[:, 1:].std(axis=0, ddof=1)
X_corr = X_std_scl.T @ X_std_scl / (X_std_scl.shape[0] - 1)
```

Since we're using sample correlation coefficient we could choose to use the NumPy built-in, taking care to use `rowvar=False` since our variables are in columns here.

```
X_corr = np.corrcoef(X_std_scl, rowvar=False)
```

Next we use NumPy to find the eigenvalues and eigenvectors, then sort them descending from highest to lowest. We then set up for a PCR using the top 2 principal factors (the `eigvecs[:, :2]`), adding back our ones for an intercept.

```
# PCA
eigvals, eigvecs = np.linalg.eigh(X_corr)
desc = np.argsort(eigvals)[::-1]
eigvals = eigvals[desc]
eigvecs = eigvecs[:, desc]

# set up for PCR, top 2 principal components
X_pca_2 = np.column_stack((np.ones(Y.shape[0]), X_std_scl @ eigvecs[:, :2]))
```

We import `PCA` from `sklearn.decomposition`. This PCA works a little differently in that it uses singular value decomposition, but that should not make any big difference here.

```
from sklearn.decomposition import PCA
```

Finally we run the Sklearn PCA for 2 principal components, run both our PCRs, and compare. Let's check two data points: the explained variance from each PCA and the MSE from each 2-component PCR.

```
# PCA with 2 components
pca = PCA(n_components=2, random_state=2025)
X_pca_skl = pca.fit_transform(X_std_scl)

# set up for PCR
X_pca_skl_2 = np.column_stack((np.ones(Y.shape[0]), X_pca_skl))

# Compare hand-code with Sklearn
print("Explained variance, hand-code = ",
```

```
      eigvals[:2]/sum(eigvals),
      "\nExplained variance, Sklearn   = ",
      pca.explained_variance_ratio_,
      "\nMSE, hand-code = ",
      cvk(X_pca_2, Y).round(3),
      "\nMSE, Sklearn   = ",
      cvk(X_pca_skl_2, Y).round(3))
```

```
## Explained variance, hand-code =  [0.5449028  0.20574886]
## Explained variance, Sklearn   =  [0.5449028  0.20574886]
## MSE, hand-code =  83060.681
## MSE, Sklearn   =  83060.674
```

The explained variance figures tie, as do the MSEs.