Reducing “Complexity” by Penalized Estimation

Consider a linear regression model with p inputs:

$$Y \;=\; \beta_0 \,+\, \beta_1 X_1 \,+\, \cdots \,+\, \beta_p X_p \,+\, \text{noise}$$

The traditional way of estimating the β_i is to choose the ones that minimize the squared error in predicting the responses in the training set, also called the *Residual Sum of Squares (RSS)*:

$$\text{RSS}(\beta) \;=\; \sum_{i=1}^{n} \Big[y_i - (\beta_0 \,+\, \beta_1 x_{i1} \,+\, \cdots \,+\, \beta_p x_{ip}) \Big]^2$$

We’ve seen that this can produce large variance, when p is big.

One way to reduce variance (at the cost of bias) is to instead minimize a *penalized sum of squares*, equal to $\text{RSS}(\beta) + \text{Penalty}(\beta)$.

Some Possible Penalty Functions for Linear Regression

The penalty should encourage small β_i, so that we more-or-less get rid of some terms in the model. Some possibilities:

$$\text{Penalty}(\beta) = \sum_{i=0}^{p} \lambda \beta_i^2$$

Ridge Regression — pulls β_i towards zero

$$\text{Penalty}(\beta) = \sum_{i=0}^{p} \lambda |\beta_i|$$

The Lasso — pulls β_i to 0, some exactly 0

$$\text{Penalty}(\beta) = \sum_{i=0}^{p} \lambda \log(1 + \beta_i^2/s)$$

A penalty that encourages weights to be close to zero, but allows them to be big if they really want to be.

Notice that these penalty functions all have unknown parameters — λ and s.

It's hard to set these parameters manually, since they have no apparent meaning.

Setting Parameters of Penalty Functions by Cross Validation

We can set parameters of the penalty based on the training data itself by the method of *cross validation* (see Section 7.10 in the text).

For parameter value in some set (eg, for $\lambda = 0.01, 0.03, 0.1, 0.3$), we estimate what our prediction error on test cases would be if we used this value, then pick the one that seems best. To estimate prediction error using some given λ:

1) Divide the training set (of size N) into K (almost) equal portions, each containing about N/K cases.
2) Fit the model K times using the penalty with this λ. For the k'th fit, use all the training data *except* the cases in the k'th portion. This gives us K sets of parameter estimates.
3) For each set of parameter estimates, make predictions for the training cases *not* used in fitting — ie, we use the parameters fit using all except the k'th portion of the training set to make predictions for the left-out k'th portion.
4) Average the prediction errors over all K fits, and all cases in left-out portions.

Having chosen a λ in this way, we use it when fitting to *all* the training data, and use the parameters from this final fit to make predictions on real test cases.

What's Good and Bad About Cross Validation

Good:	**Bad:**
It's simple to do.	But it can take a lot of time, especially if we consider lots of possible parameter values.
It makes very few assumptions and works (sort-of) even if the model is wrong.	But it therefore doesn't use any knowledge we may have about the problem.
	The estimates are noisy, so we can't be sure we're choosing the best parameter values.

Cross Validation for Nearest Neighbor Methods

We can use cross validation for memory-based methods like k-NN as well, in order to choose various parameters of the method, such as:

- The best value of k.
- The best distance function. For instance, we might define a distance by

$$d(x, x')^2 \;=\; \sum_{i=1}^{p} \min[(x_i - x'_i)^2, U]$$

  This distance function limits the effect of any single input, so it may be more robust to errors in the input. But we'd need to choose an appropriate value for U.

- The best way to combine the responses of the neighbors. For instance, we might decide to "shrink" towards the overall mean:

$$\hat{Y}(x) \;=\; \frac{1}{k+\lambda}\Big[\lambda\bar{y} + \sum_{i \in N_k(x)} y_i\Big]$$

  where $\bar{y} = (1/N)\sum_{i=1}^{N} y_i$.

Combining Nearest Neighbor and Linear Regression

We can combine the “memory-based” nearest neighbor and “parametric” linear regression methods — we find the nearest neighbors and then fit a regression model using them.

To make a prediction for a test case with inputs x:

1) Find the k training points with inputs closest to x.

2) Fit a linear regression model to just these k points, obtaining an estimate $\hat{\beta}$.

3) Predict the response in the test case using this $\hat{\beta}$.

Of course, we might use a penalized fit in step (2), and we might choose k and other parameters using cross-validation. There are many variations on this idea.