The Need to Control Overfitting by a Suitable Amount

Following the training data too closely leads to *overfitting* — making predictions based on random aspects of the training data, not the true relationship of y to x.

We've now seen several methods that have adjustable parameters that control how closely predictions follow the training data:

Nearest neighbor methods:

k, the number of neighbors used

Naive Bayes:

α, the constant added to frequencies to avoid zero probabilities

Maximum penalized likelihood (for linear or logistic regression):

λ, the magnitude of the penalty on β, as well as ν for the P_t penalty

Decision trees:

λ, the magnitude of the penalty on tree size

We will seldom know *a priori* what the appropriate values of these parameters are. We need to choose a value based on the training data.

Dividing the Training Data into Estimation and Validation Sets

Our problem: What looks good on the training set may not work on test cases.

One solution: Don't fit on some of the training cases. Instead use them as "pretend" test cases, so we can estimate how well we would do on real test cases.

Method: If we have a training set of n cases, we split it into two new sets:

> An *estimation set* of n_e cases, which we use in the usual way — eg, to estimate model parameters such as β.
>
> A *validation set* of $n_v = n - n_e$ cases, which we try to predict using the estimation set, and then see how well we do.

The cases in the estimation set must be randomly chosen from the full training set — *not* by, for instance, using the first n_e cases in the data file — to ensure that they are likely to be typical of the whole training set.

Selecting Penalty Parameters Using the Validation Set

Here's how we use the estimation and validation sets to decide on the value of a penalty parameter. (Here I'll assume λ is the penalty parameter, but the same procedure is used for k, α, etc.)

- For $\lambda =$ whatever set of values you're considering:
  - Estimate model parameters using the estimation set, with this value of λ (if the method is parametric), then predict the value of y for each case in the validation set.
  - Compute some measure of error/loss for each validation case by comparing the prediction with the true value. Average these to get an estimate of the the average error/loss when using this value of λ.
- Let $\hat{\lambda}$ be the value of λ considered above with smallest average error/loss.
- Estimate model parameters using the whole training set with this $\hat{\lambda}$ (if the method is parametric), and use this $\hat{\lambda}$ and the whole training set to make predictions for real test cases. **OR**: Just continue using the estimates from the estimation set when predicting for real test cases.

Pros and Cons of Cross Validation

It's good because:

The validation set gives us an *unbiased* estimate of how well the method will do on test cases (using a given λ). Performance on the data used for parameter estimation will tend to be an optimistic estimate of performance on test data.

We can use *our real performance criterion* when choosing the best λ — eg, if we're really interested in whether $y > 0$ or $y \leq 0$, we can use accuracy when predicting that to choose λ, even if we're using squared error to fit parameters.

It's not so good because:

The estimate of performance on test cases from the validation set is *noisy* — if the validation set is small, this estimate will be highly variable, and be a poor guide to the best λ.

The method uses data inefficiently — we're using only part of the data to find parameter estimates, and only part to estimate performance on the test data. Re-training on the entire training set at the end helps with this.

N-fold Cross Validation

We can try to make more efficient use of the training data by using many estimation/validation splits of the training set, and then choosing the best λ based on average performance over all these splits.

In *N-fold cross validation*, we first randomly divide the n training cases into N sets, each with approximately n/N cases. Label these "folds" by $f = 1, \ldots, N$.

For each value of λ we are considering, we then make predictions for each fold, f, using the training cases in all the folds other than f to estimate parameters. We average error/loss over all N folds, and over the cases within each fold. We select $\hat{\lambda}$ to minimize this average error/loss.

Finally, we make predictions for real test cases using parameters estimated from all the training data, using $\hat{\lambda}$ as found above.

10-fold cross validation is a commonly used procedure. The extreme form is *leave-one-out* cross validation, in which $N = n$. Estimating parameters N times seems like a lot of work, but sometimes there are short-cuts.