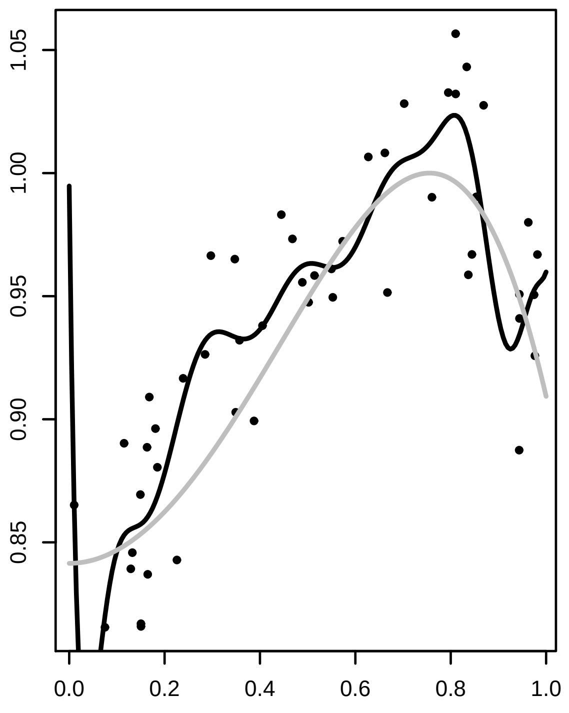
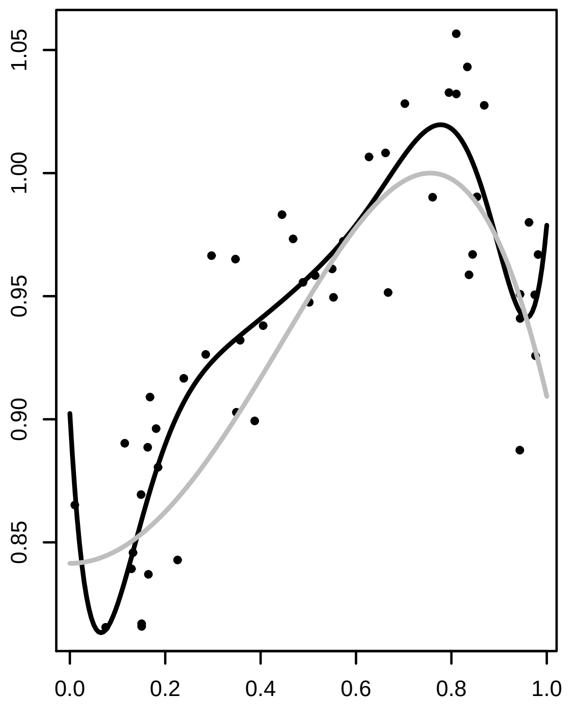
Regularized Linear Basis Function Models

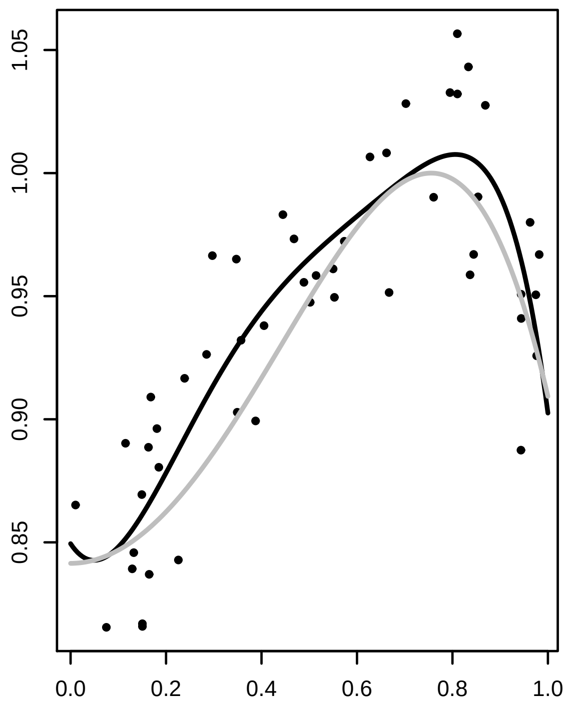
Read Chapter 3 and Section 1.3 in the text by Bishop

Recall the Example From Last Lecture
Using Gaussian Basis Functions

Here are the results of maximum likelihood estimation using Gaussian basis functions (plus $\phi_0(x) = 1$) on the example dataset, with varying width (and spacing) s:


Gaussian basis function fit, s = 0.1


Gaussian basis function fit, s = 0.5


Gaussian basis function fit, s = 2.5

Maximum Penalized Likelihood Estimation

We can try to avoid the poor results of maximum likelihood when there are many parameters by adding a *penalty* to the log likelihood, that favours non-extreme values for the parameters. This procedure is also called *regularization*

For regression with Gaussian noise, we minimize the sum of squared errors on training cases plus this penalty.

Quadratic penalties are easiest to implement, as they combine with the squared error to still allow solution by matrix operations. For basis function models, we might use a penalty that encourages all w_j (except w_0) to be close to zero:

$$\lambda \sum_{j=1}^{M-1} w_j^2$$

Here, λ controls the strength of the penalty, which we must somehow decide on.

Note: In the book, w_0^2 is included in the penalty, but this usually does not make sense. Also, the book multiples both the squared error and the penalty by 1/2, which has no effect in the end.

Solution for Penalized Least Squares

We found the least squares solution before by equating the gradient of the squared error to zero. Now we add the gradient of the penalty function as well, and hence solve

$$2\lambda w^* \;-\; 2\Phi^T(t - \Phi w) = 0$$

where w^* is equal to w except that w_0 is zero.

Solving this, the penalized least squares estimate of w is

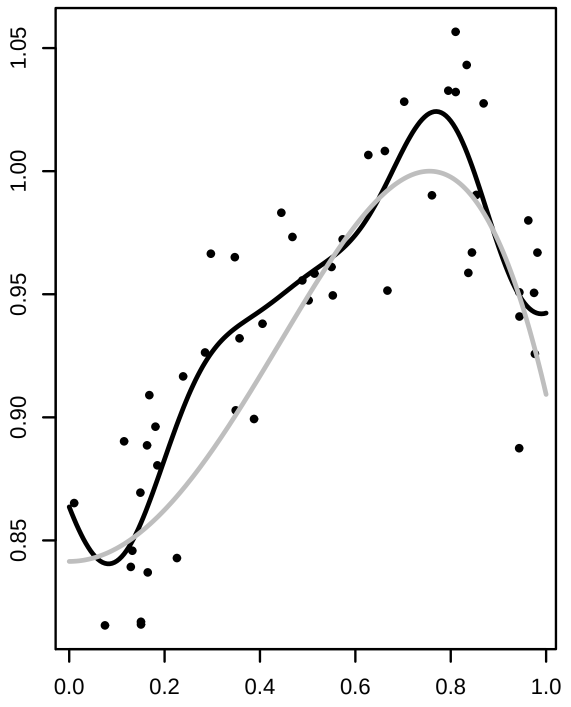
$$\hat{w} \;=\; (\lambda I^* + \Phi^T\Phi)^{-1}\Phi^T t$$

where I^* is like the identity matrix except that $I^*_{1,1} = 0$.

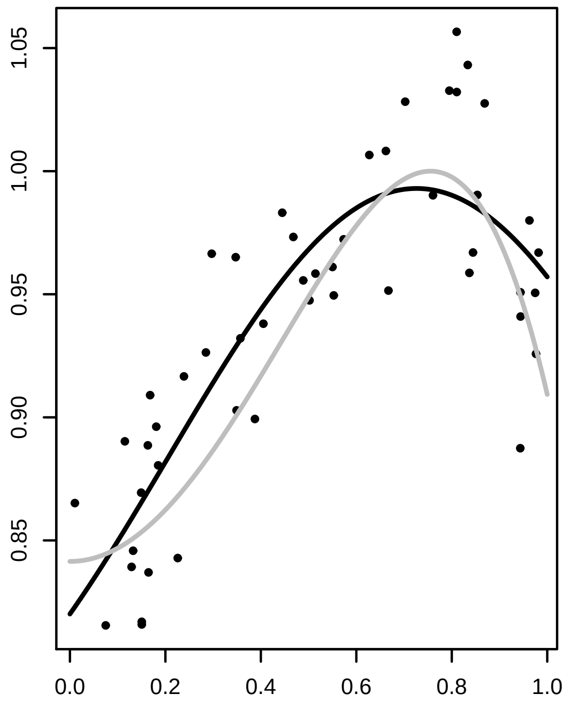
Note that this estimate will be uniquely defined regardless of how big M and N are, as long as λ is greater than zero.

Results with Regularized Gaussian Basis Functions

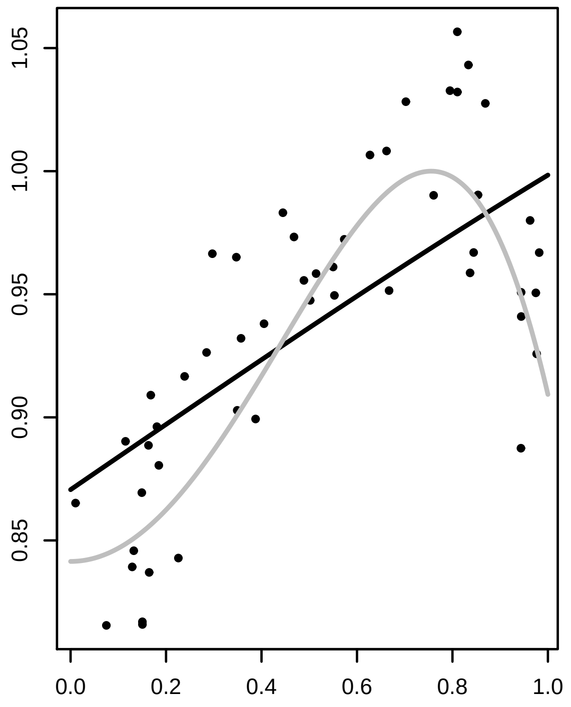
Here are the results with $\lambda = 0.1$:



Gaussian basis function fit, s = 0.1 lambda = 0.1



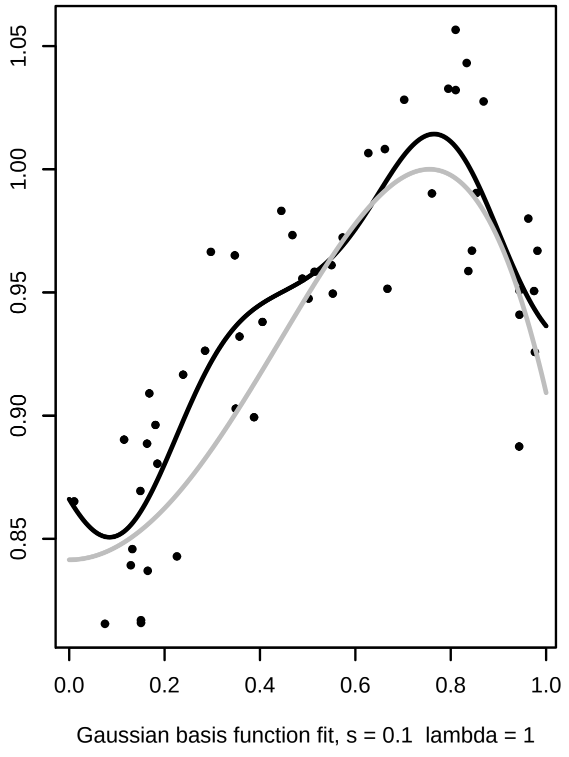
Gaussian basis function fit, s = 0.5 lambda = 0.1

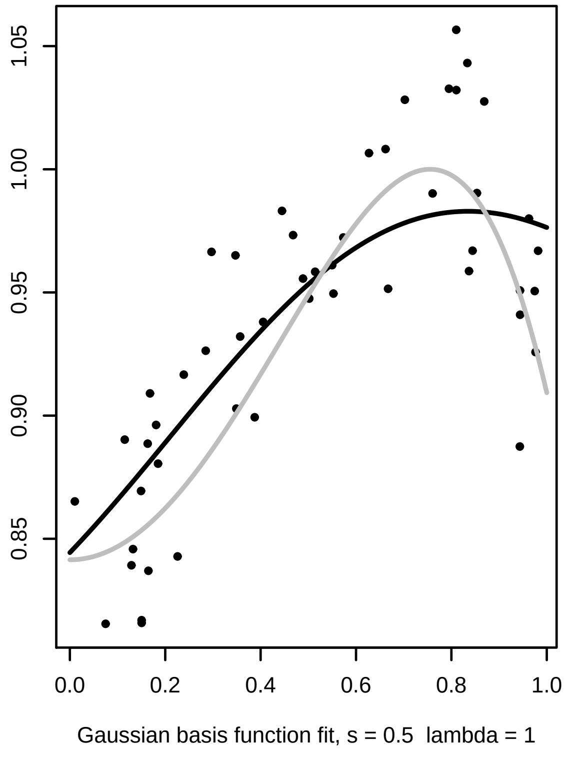


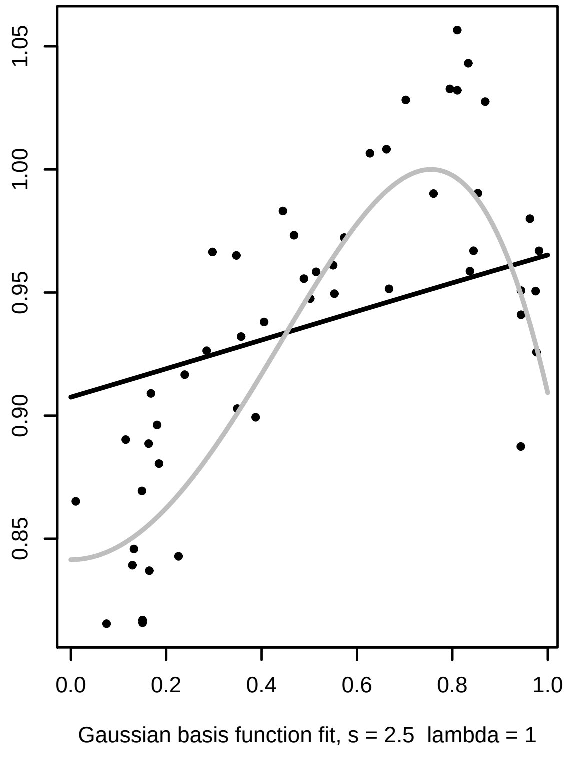
Gaussian basis function fit, s = 2.5 lambda = 0.1

More Results with Regularized Gaussian Basis Functions

Here are the results with $\lambda = 1$:

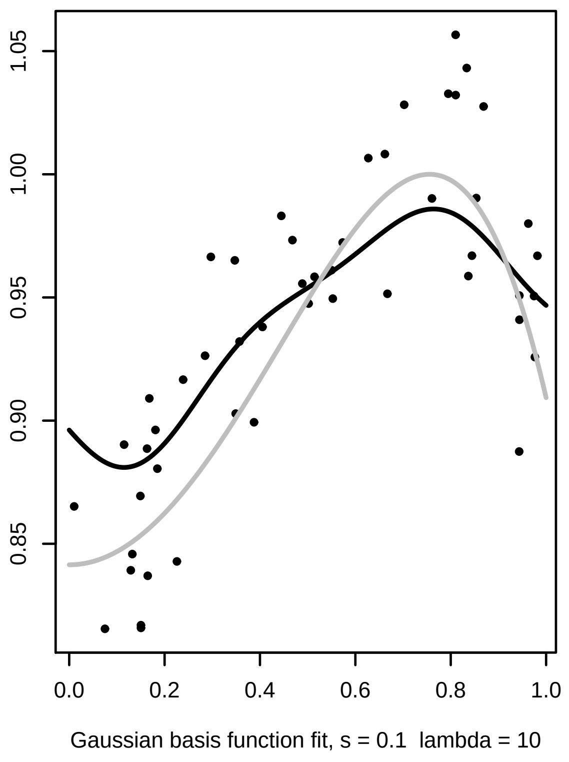


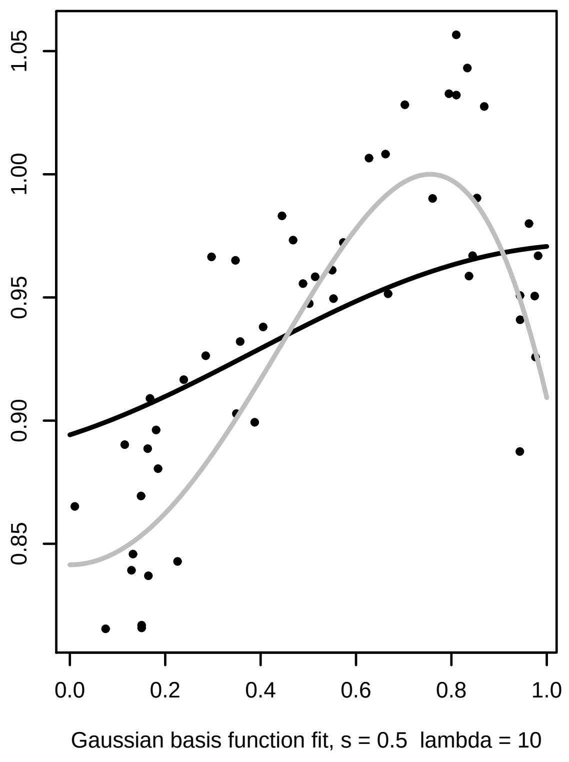


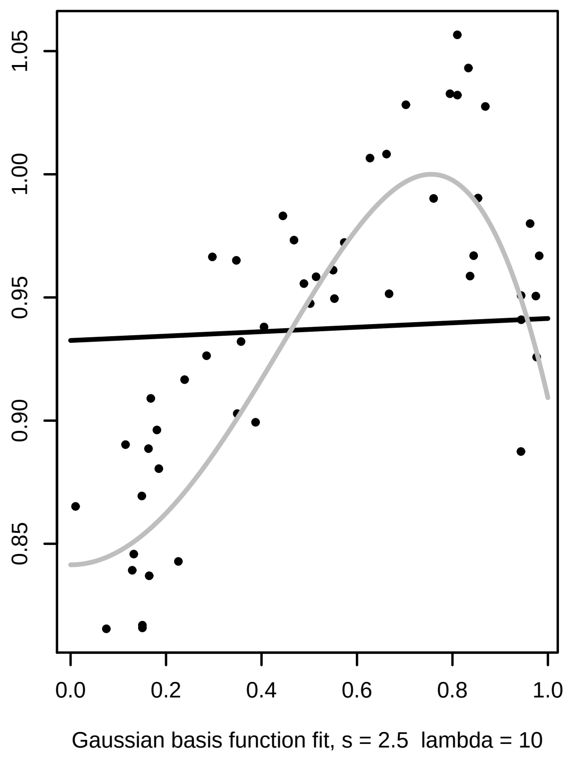


Yet More Results with Regularized Gaussian Basis Functions

Here are the results with $\lambda = 10$:

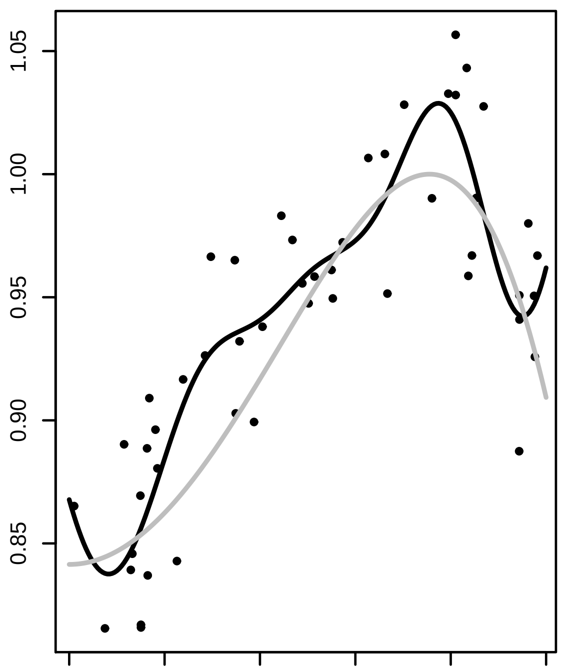




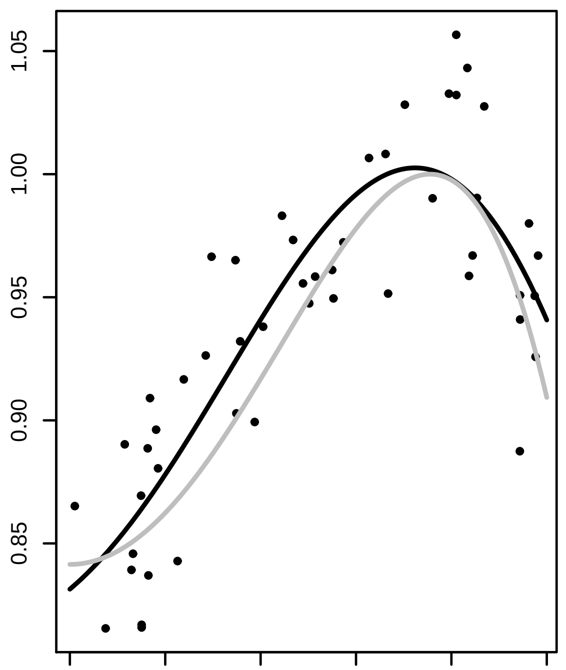


And Yet More Results. . .

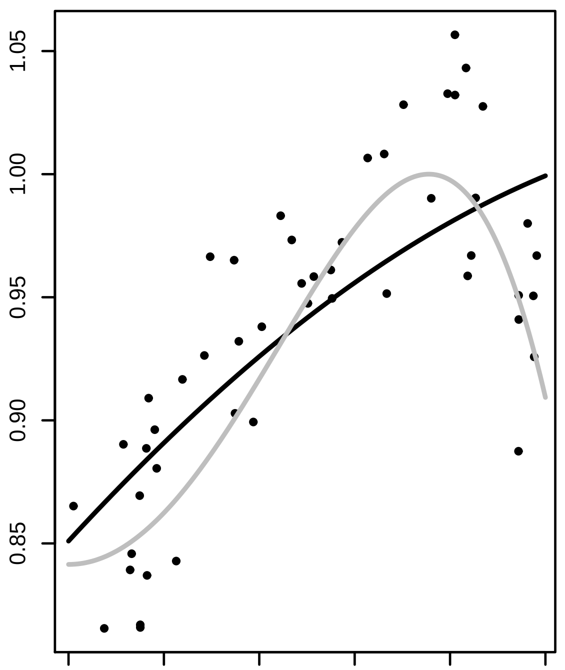
Here are the results with $\lambda = 0.01$:



Gaussian basis function fit, s = 0.1 lambda = 0.01



Gaussian basis function fit, s = 0.5 lambda = 0.01



Gaussian basis function fit, s = 2.5 lambda = 0.01

Conclusions from this Example

We see that we can control overfitting with Gaussian basis functions either by choosing the width of the basis functions, s, to be large, or by using a positive penalty, λ.

It seems that we may need to adjust *both* s and λ to get the best results.

How can we make such adjustments?

In real problems, we can't look at how the results match the true function, as we did here!

Using a Set of Validation Cases

In supervised learning, we try to learn the relationship of targets to inputs from a set of *training cases*, where both are known.

We then use what we have learned to predict the target for some *test case*, where only the inputs are known. We want to adjust our learning method to do as well as possible at these predictions.

One way:

1. Randomly divide the training set into an *estimation set* and a *validation set*.
2. Try out various methods (eg, different basis function widths, different penalty magnitudes) fitting to the data in the estimation set.
3. See how well each method does at predicting cases in the validation set.
4. Use the method with the best average performance on the validation set to make the prediction for the test case.

Choice of Loss Function for Validation

To use this validation procedure, we need some measure of predictive performance to average over validation cases. This can take the form of a *loss function*, ℓ, which takes the actual target, t, and a prediction for the target as argments.

For a point prediction, $\widehat{t}$, two loss functions are commonly used:

$$\begin{array}{rrcl} \textbf{Squared error loss:} & \ell(t,\widehat{t}) & = & (t-\widehat{t})^2. \\ \textbf{Absolute error loss:} & \ell(t,\widehat{t}) & = & |t-\widehat{t}|. \end{array}$$

When the prediction is a distribution, $\widehat{P}$, for t, one can use

$$\textbf{log probability loss:} \quad \ell(t,\widehat{P}) \;=\; -\log \widehat{P}(t).$$

We should choose a loss function that comes close to capturing what we're actually worried about! Squared error is very traditional, however.

Selection of s and λ for the Example

I looked at the square root of the average squared error on validation cases when fitting on the other cases using penalized least squares.

Results using cases 1 to 10 for the validation set:

```
        lambda=0.001 lambda=0.01 lambda=0.1 lambda=1 lambda=10
s=0.02        0.0869      0.0722     0.0476   0.0413    0.0475
s=0.1       ->0.0323    ->0.0323     0.0330   0.0342    0.0400
s=0.5         0.0352      0.0364     0.0389   0.0443    0.0514
s=2.5         0.0435      0.0506     0.0518   0.0553    0.0613
```

Note: The cases here are in random order. If they aren't one shouldn't just take the first 10 as the validation set! A random subset is needed.

It looks like $s = 0.1$ and $\lambda = 0.01$ or $\lambda = 0.001$ is best.

But what if we had used a different validation set?

Selections from Five Validation Sets

Here the selections for s and λ using five validation sets, partitioning the 50 cases:

```
Using cases 1 to 10 for validation set:
       lambda=0.001 lambda=0.01 lambda=0.1 lambda=1 lambda=10
s=0.02       0.0869      0.0722     0.0476   0.0413    0.0475
s=0.1      ->0.0323    ->0.0323     0.0330   0.0342    0.0400
s=0.5        0.0352      0.0364     0.0389   0.0443    0.0514
s=2.5        0.0435      0.0506     0.0518   0.0553    0.0613

Using cases 11 to 20 for validation set:
       lambda=0.001 lambda=0.01 lambda=0.1 lambda=1 lambda=10
s=0.02       0.1285      0.0520     0.0491   0.0522    0.0544
s=0.1        0.0401      0.0359     0.0304   0.0281    0.0340
s=0.5        0.0273    ->0.0269     0.0273   0.0330    0.0442
s=2.5        0.0303      0.0382     0.0409   0.0491    0.0573

Using cases 21 to 30 for validation set:
       lambda=0.001 lambda=0.01 lambda=0.1 lambda=1 lambda=10
s=0.02       0.0409      0.0426     0.0378   0.0344    0.0439
s=0.1        0.0342      0.0331     0.0322   0.0310    0.0358
s=0.5        0.0304      0.0285   ->0.0278   0.0320    0.0457
s=2.5        0.0294      0.0353     0.0394   0.0513    0.0605

Using cases 31 to 40 for validation set:
       lambda=0.001 lambda=0.01 lambda=0.1 lambda=1 lambda=10
s=0.02       0.0558      0.0418     0.0444   0.0452    0.0577
s=0.1        0.0523      0.0404     0.0360 ->0.0347    0.0468
s=0.5        0.0392      0.0383     0.0397   0.0424    0.0587
s=2.5        0.0413      0.0446     0.0499   0.0651    0.0749

Using cases 41 to 50 for validation set:
       lambda=0.001 lambda=0.01 lambda=0.1 lambda=1 lambda=10
s=0.02       0.0688      0.0363     0.0287 ->0.0239    0.0360
s=0.1        0.0305      0.0310     0.0309   0.0299    0.0322
s=0.5        0.0327      0.0328     0.0334   0.0349    0.0384
s=2.5        0.0354      0.0414     0.0405   0.0422    0.0504
```

S-Fold Cross Validation

The variablility in selection of s and λ from the random choice of a subset of validation cases can be reduced by averaging the squared error over S validation sets, that partition the whole training set.

Here are the square roots of the average validation squared error over the $S = 5$ validation sets from the previous slide:

```
        lambda=0.001 lambda=0.01 lambda=0.1 lambda=1 lambda=10
  s=0.02      0.0820      0.0506     0.0422   0.0406    0.0485
  s=0.1       0.0387      0.0347     0.0326 ->0.0317    0.0381
  s=0.5       0.0332      0.0329     0.0338   0.0377    0.0482
  s=2.5       0.0364      0.0423     0.0448   0.0531    0.0614
```

Does the selection of $s = 0.1$ and $\lambda = 1$ seem reasonable given the data points? If you know the true function, does it seem to actually be the best choice?

Other Penalty Functions

Penalty functions other than $\lambda \sum_{j=1}^{M-1} w_j^2$ are sometimes used.

A penalty such as $\lambda \sum_{j=1}^{M-1} \log(1 + w_j^2)$ penalizes w_j when it is near zero (where $\log(1 + w_j^2) \approx w_j^2$), but not much when $|w_j|$ is big. This may be good if you expect a few w_j to be much bigger than the rest, and don't want to shrink them towards zero.

But the maximum penalized likelihood estimate with this penalty may not be unique.

The *lasso* penalty is $\lambda \sum_{j=1}^{M-1} |w_j|$.

Finding the maximum penalized likelihood estimate for this penalty can't be done with simple matrix operations, but it can be done efficiently. There is a unique solution.

The Lasso Produces Sparse Estimates

The lasso penalty has the property that, in the penalized maximum likelihood estimate, often some of the w_j are exactly zero. This doesn't happen when the penalty involves w_j^2.

Why? One way to see the difference is that when w_j is close zero, the derivative of w_j^2 is also close to zero — so the likelihood will dominate (and in general doesn't favour w_j being exactly zero). But even when w_j is close to zero, the derivative of $|w_j|$ is ± 1, so the penalty can drive w_j to be exactly zero.

Is this desirable? It may be if

- You believe that many of the true w_j are exactly zero.
- You prefer many w_j to be exactly zero so the result is easier to interpret.
- You want many w_j to be exactly zero to save computation time later.

But often you don't believe that any w_j are exactly zero, in which case setting them to zero may degrade predictive performance.