Lecture 3: Modern regression and data mining tools (continued)

Alan Lee

Department of Statistics
STATS 760 Lecture 3

April 5, 2017

Outline

Today's agenda

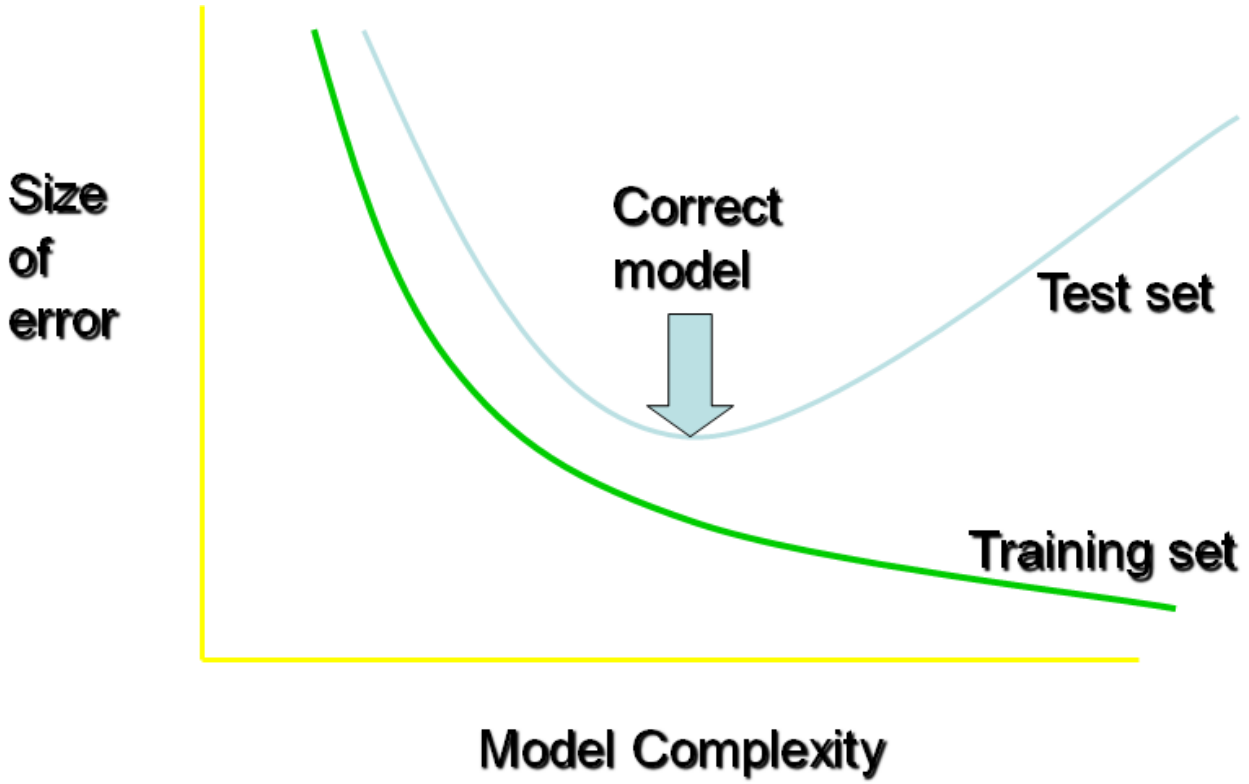
In this lecture we we continue our discussion of modern regression methods that are useful in data mining.

- Model complexity and model choice
- Estimating prediction error
- Regularisation
- Boosting and bagging

Model complexity

- When fitting models to a training set, we do not want to use a model that is too complex. Such a model will capture not only the nature of the true function f but will also model the noise ϵ.
- Future data will have the same f but different noise, which will not be well predicted by a too-complex model.
- Thus, a too-complex model will have a small training set error, but a large test set error, (the error expected when predicting future data.)

Model complexity(cont)



Measuring Complexity

- For linear models, measured by the number of variables
- For trees, also by the number of terminal nodes
- For neural nets, also by the number of units in the hidden layer

Caution: Models that are too complex will not be good for prediction - model noise as well as signal. Models that are too simple will make biaswed predictions. We need to choose a model with moderate complexity, one that minimizes the prediction error.

Comparing models using prediction error

If we are using the model for prediction, we want a model that has small prediction error (PE) when applied to new data. Denoting the training set data by Z, a new data point by (x, y) and the predictor at x by $\hat{f}_Z(x)$, PE is the average prediction error averaged over all new X and Y:

$$PE = E_{X,Y}((Y - \hat{f}_Z(X))^2).$$

This is the expected error for a fixed training set. If the response is continuous, can estimate it by

$$\frac{1}{m}\sum_{i=1}^{m}(y_{i0} - \hat{f}_Z(x_{i0}))^2,$$

where $(x_{i0}, y_{i0}), i = 1, \ldots m$ is some new data (the test set) and $\hat{f}_Z$ is the estimate of f, calculated from the old data Z (the training set).

This form of the PE, for a fixed training set is the *conditional* PE.

Comparing models (cont)

We might also want to average over all possible training sets. This leads to the following definition of PE (*unconditional* PE):

$$PE = E_Z(E_{X,Y}(y - f_Z(X))^2)).$$

In the next few slides, we discuss two methods of estimating PE, cross validation and the bootstrap. It turns out that these tend to estimate the unconditional PE better than the conditional.

Estimating PE

If we have new data, we can estimate PE (for a fixed training set) directly using the estimate on slide 9 (the "test set estimator"). If not, or alternatively, we can use cross validation or the bootstrap. (It may be better to use all the data to get a better predictor.)

Could we use the training set as a test set? Then the estimate of prediction error would be the residual sum of squares divided by the sample size. This is called the *apparent error* or the *training error*, denoted by $\overline{err}$. It usually *underestimates* the true PE.

Comparing regression methods: CPU data

The table shows the mean training- and test set prediction errors for 50 random splits of the CPU data set (FS = forward selection)

Mean:

	Training error	Test error
Linear	0.181	0.219
FS	0.183	0.215
Tree	0.036	0.211
Nnet	0.083	0.201

Comparing methods: California data

The table shows the mean test set prediction errors for 50 random splits of the California data set

Method	Out-of-sample error
Linear	0.359
NNet4	0.249
Tree	0.362

Cross-validation

In cross validation, we split the data randomly into 10 parts, use one part for the test set and the rest for the training set, and estimate the PE with the test set estimator. We repeat for the other ten parts using a different tenth as the test set each time. We average the resulting 10 PE estimates to get a final estimate. We can repeat for different random splits and average. This is (10-fold) cross-validation. (Could also split into 5 parts - 5-fold CV).

Bootstrap approaches

A possible bootstrap approach might be to fit the model to a bootstrap sample $z_1^*, \ldots z_n^*$, and use the original sample as a test set. This doesn't work too well as the test and training sets are too similar.

An alternative is to use the bootstrap sample as both test and training set. This really will be an underestimate. If we take the difference between the two estimates (called the optimism), we get a (possibly too small) estimate of the amount by which $\overline{\text{err}}$ underestimates PE. We could average the optimisms over B bootstrap samples and add the result to $\overline{\text{err}}$ to get a corrected PE estimate. This turns out to work quite well.

The .632 estimator

For about 63.2% of bootstrap samples, the the sample will contain a particular data point, so we are using the similar data for training and testing. It would be preferable to compute the PE over the "out-of-bag" samples, those that do not contain the data point, obtaining a revised estimate ϵ_0. This results in the "0.632 estimate" of PE:

$$PE_{0.632} = (1 - 0.632)\overline{err} + 0.632\epsilon_0.$$

ϵ_0 is a quantity computed from the "out of bag" samples.

Summary

Let PE(test, training) denote the sum

$$\frac{1}{n}\sum_{i=1}^{n}\left(y_i^{(T)} - \hat{f}_Z(x_i^{(T)})\right)^2$$

where $(x_i^{(T)}, y_i^{(T)}), i = 1, 2, \ldots, n$ denotes the test set and Z the training set. The different estimates of PE are

- Training: $PE = PE(data, data) = \overline{err}$
- Test set estimate $PE = PE(test\ set, training\ set)$
- err + opt: $PE = \overline{err} + \overline{PE(data, bs) - PE(bs, bs)}$
  bs=bootstrap sample
- 0.632: $PE = \overline{err} + 0.632(\epsilon_0 - \overline{err})$

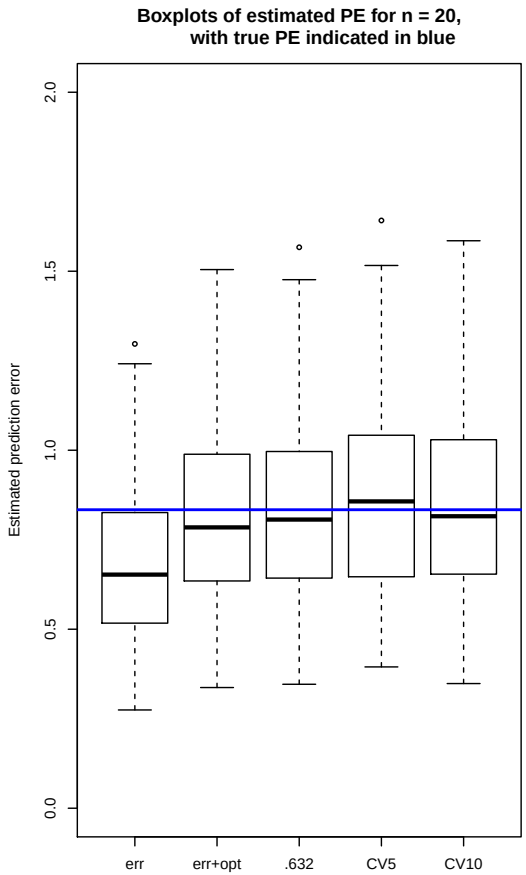
An example

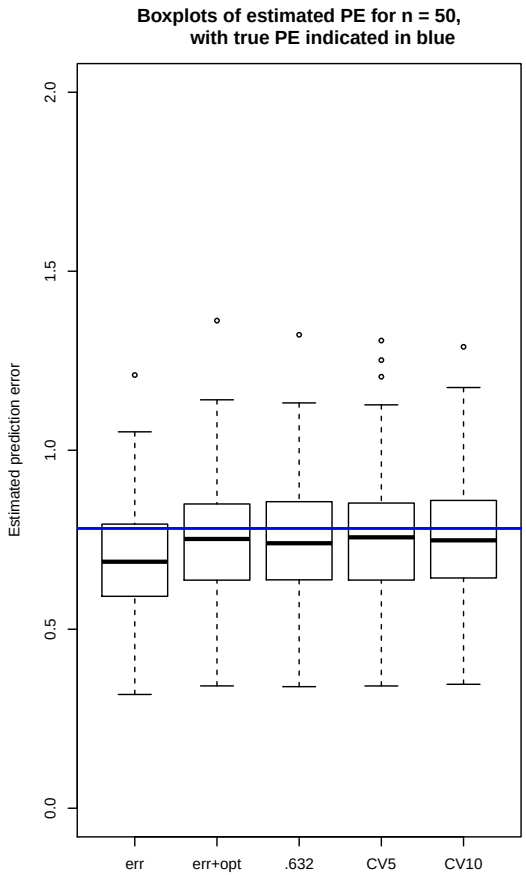
To compare the various estimators, we conducted a small simulation, using data pairs $(x_i, y_i), i = 1, \ldots, n$ sampled from a bivariate normal distribution with means 0, variances 1 and correlation ρ. We want to evaluate the prediction error when using least squares regression as a predictor. Consider the case where ρ is 0.5 and B is 100. We got the following estimates (averaged over 100 replications):

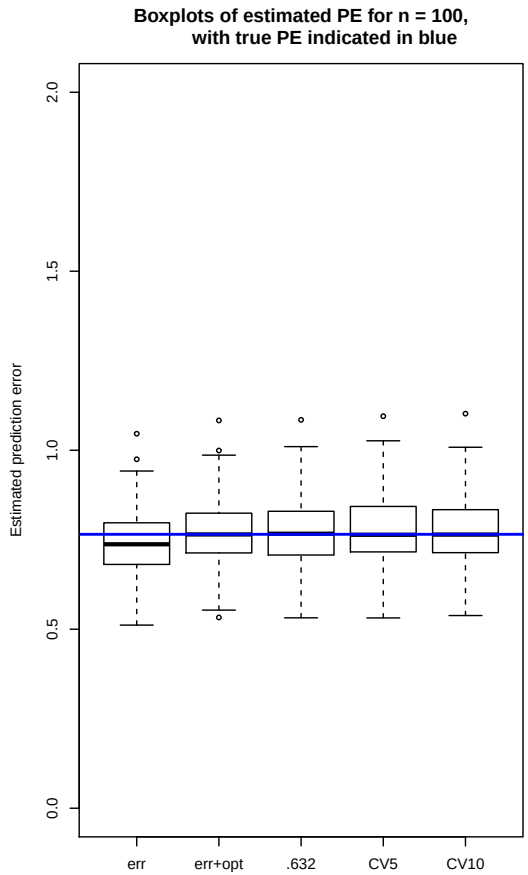
	err	err+opt	.632	CV5	CV10	True PE
$n = 20$	0.675	0.813	0.829	0.860	0.844	0.834
$n = 50$	0.698	0.754	0.755	0.764	0.761	0.781
$n = 100$	0.742	0.771	0.771	0.774	0.773	0.765

The opt+err method looks good. The apparent error $\overline{err}$ underestimates, but less so as the sample size increases.

Boxplots of 100 reps







Regularisation

- We know the perils of over-fitting linear regression models: the model won't predict new data well.
- 330 solution: use model selection techniques to set some coefficients equal to zero (i.e. eliminate some variables)
- Alternative idea: make all the coefficients smaller, rather than setting some to zero. This is "shrinkage".
- Or perhaps a combination: shrink some, set some to zero?

Motivation

For the linear regression model $y = X\beta + \epsilon$:

$$E(\hat{\beta}) = \beta$$

but

$$E(\hat{\beta}^T\hat{\beta}) = \hat{\beta}^T\hat{\beta} + \sigma^2 trace(X^TX)^{-1}.$$

The vector of regression coefficients is too long!

Thus, regularisation: A modification to least squares using a modified criterion that penalizes large coefficients and "shrinks" them towards zero.

Ridge Regression

Instead of minimizing the residual sum of squares, minimize the RSS $||y - Xb||^2$ subject to the constraint

$$\sum_{j=1}^{p} b_j^2 \leq s$$

for a fixed value s. If the least squares estimates $\hat{\beta}_j$ satisfy

$$\sum_{j=1}^{p} \hat{\beta}_j^2 \leq s$$

then this is just least squares, otherwise the fitted coefficients will have shorter length.

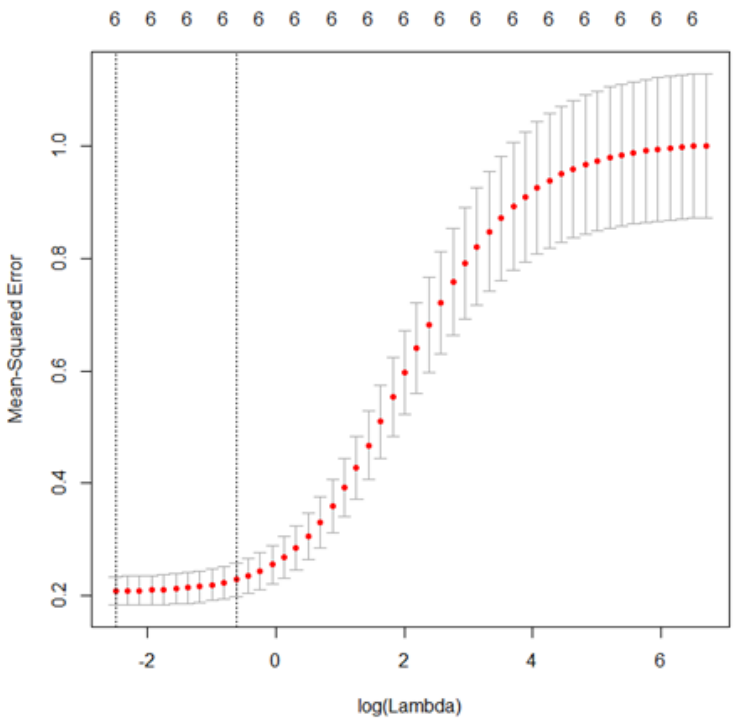
Equivalent formulation

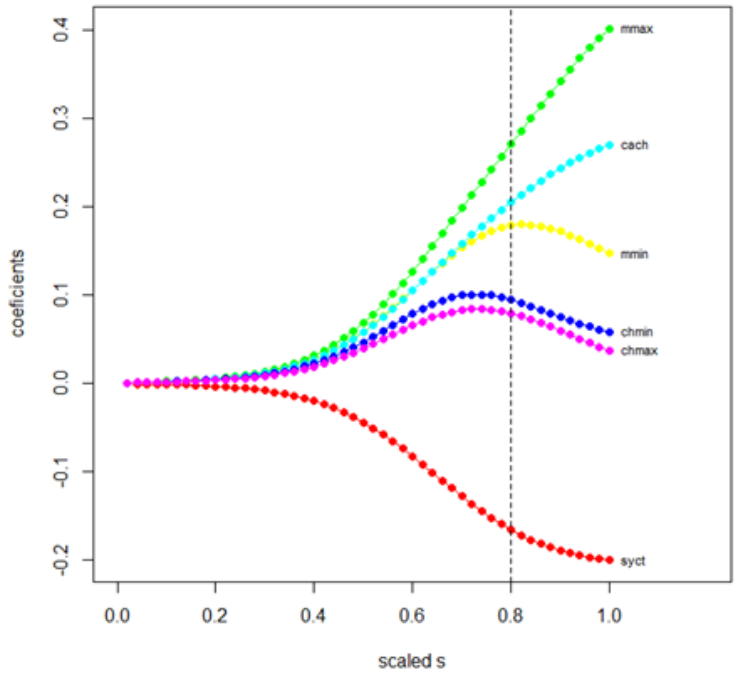
Equivalent to minimizing $||y - Xb||^2 + \lambda||b||^2$ for a fixed value of λ.

The minimizer is

$$\hat{\beta}_{\text{Ridge}} = (X^T X + \lambda I)^{-1} X^T y.$$

Ridge trace: CPU data





The Lasso

Similar to ridge but we are minimizing the residual sum of squares $||y - Xb||^2$ subject to the (different) constraint

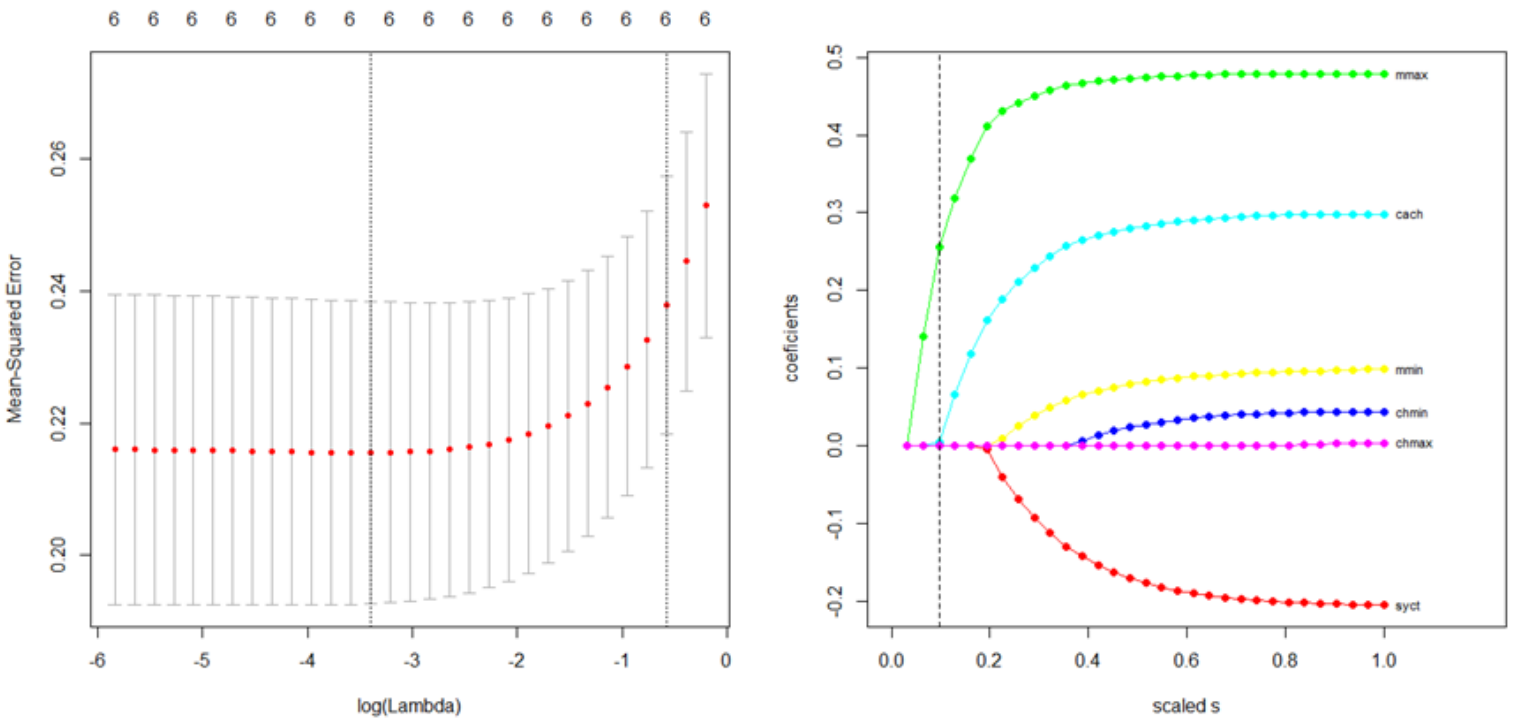
$$\sum_{j=1}^{p} |b_j| \leq s$$

for a fixed value s.

If the LS estimates satisfy this constraint then the lasso is equivalent to LS. Otherwise the lasso coefficient vector has a shorter length.

Some lasso coefficients can be zero. Thus, the lasso is a compromise between ridge (which shrinks but does not set coefficients to 0) and subset selection (which sets some coefficients to 0 but does not shrink).

Lasso trace: CPU data



Elastic net

Combination of ridge and lasso: minimize

$$||y - Xb||^2 + \lambda_1 \sum_j |b_j| + \lambda_2 \sum_j b_j^2$$

Standardization

To ensure all variables are treated equally, it is usual to standardize all variables to have mean 0 and sd 1. Then we don't need a constant term in the regression.

Boosting and Bagging

Next we look at ways of combining models to get a better prediction. We consider two: boosting (see next slide) and bagging. Bagging (=Bootstrap Aggregation) consists of fitting models to different bootstrap samples and averaging the results.

Boosting

The basic idea here is

1. Fit a model to the data. Then repeat
   1.1 Modify the data in some way e.g. by extracting residuals or changing weights.
   1.2 Fit a model to the modified data
2. Combine the results

Ref for Boosting: HTF Chapter 10.

Application to regression

Forward stagewise modelling: (HTF p 341, see also article by Peter Buhlmann and Torsten Hothorn in *Statistical Science* on the course web page)

Many regression models fit an additive combination of "basis functions" i.e. express f as

$$f(x) = \sum_{m=1}^{M} \beta_m b(x, \gamma_m)$$

where the β_m are regression coefficients and $b(x, \gamma)$ is a "basis function" which depends on a parameter γ. Examples:

- Linear: $b(x, \gamma) = x_j$
- Trees: $b(x, \gamma) = I(x \in R_m)$
- Neural nets: $b(x, \gamma) = \sigma(\gamma_0 + \gamma^T x)$

FSM Algorithm

Let $L(y, \eta)$ be a loss function, e.g. $L(y, \eta) = (y - \eta)^2$.

1. Set $f_0(x) = 0$.
2. For $m = 1, 2 \ldots, M$:
   2.1 Compute

$$(\hat{\beta}\hat{\gamma}) = \text{argmin}_{\beta,\gamma} \sum_{i=1}^{n} L(y_i, f_{m-1}(x_i) + \beta b(x_i, \gamma))$$

   2.2 Set $f_m(x) = f_{m-1}(x) + \nu\hat{\beta} b(x, \hat{\gamma})$.

The "regularisation parameter" ν limits the effect of the new basis function, and is typically has value around 0.01.

Squared error loss, linear regression

Here $L(y, \eta) = (y - \eta)^2$, and $b(x, \gamma) = x_j$ (i.e. one of the explanatory variables). At each stage we are fitting a simple regression model involving a single variable to the residuals from the previous fit. In effect, we model the residual from the previous stage by the best-fitting simple linear regression model. We then add ν times the prediction from this model to the previous prediction. We refer to this approach as using a regression "base learner".

Squared error loss, trees

Here $L(y, \eta) = (y - \eta)^2$, and at each stage (instead of fitting a linear regression) we are fitting a tree to the residuals from the previous fit. Usually these trees are very small. This is using a tree as the "base learner".

Points to note

- Trees are usually small (2 -5 terminal nodes)
- M (number of boosting iterations) is large, say 00's. We are averaging many small noisy trees.
- Implemented in R by the `mboost` package - see documentation on the web page
- We need to choose the number of boosting iterations, the tree size (number of terminal nodes) and the regularisation parameter ν.

Example

We use the bodyfat data set in the `TH.data` package. This consists of data on 71 healthy female subjects, containing body fat measurements (variable `DEXfat`) and nine anthropometric measurements. The idea is to develop a predictive model for body fat.

We will standardise the data and fit a boosted regression model and a boosted tree model, using 200 boosting iterations and trees with a maximum of 4 terminal nodes. The code is on the next few slides.

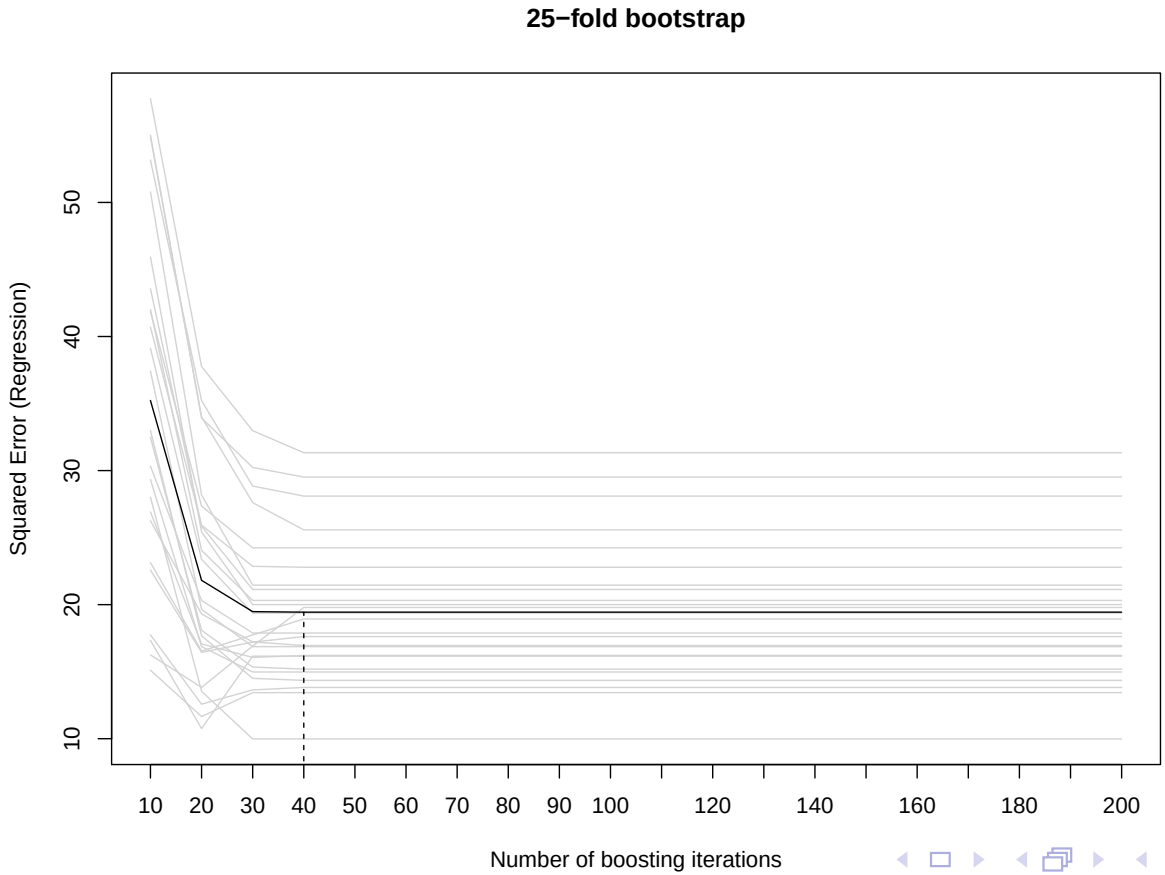
Example: boosted regression

Here we use a linear regression model (fitting one variable at a time) as the base learner.

```
 data("bodyfat", package = "TH.data")
bodyfat=scale(bodyfat)

# Now fit model
myfit = mboost(DEXfat   ., data = bodyfat,
control = boost_control(mstop = 200, nu = 0.1))
```

Choosing number of iterations: Error plots

```
par(mfrow=c(1,1))
my.risk = cvrisk(myfit, grid = 1:20*10)
plot(my.risk)
```



Example: boosted trees

Try boosted tree, 100 iterations, maximum tree depth 3:

```
library(party)
bf_tree <- blackboost(DEXfat ~ ., data = bodyfat,
     control= boost_control (mstop = 100, nu = 0.1,center =
     tree_controls = ctree_control(maxdepth = 3))
plot(cvrisk(bf_tree))
```

Bagging

Bagging stands for *B*ootstrap *Agg*regation. The idea is: to make a prediction at x, we

- Draw B bootstrap samples
- Fit a model f to each sample
- Average the predictions from each model

Often done with trees, with a small tweek in the above. This results in Random Forests. Random Forests were invented by Leo Breiman, a Berkeley professor, and further developed by Adele Cutler (an Auckland graduate) Ref for Bagging: HTF Chapter 8 (section 8.7). Ref for Random Forests HTF Chapter 15, see also `https://www.stat.berkeley.edu/ breiman/RandomForests/cc_home.htm`

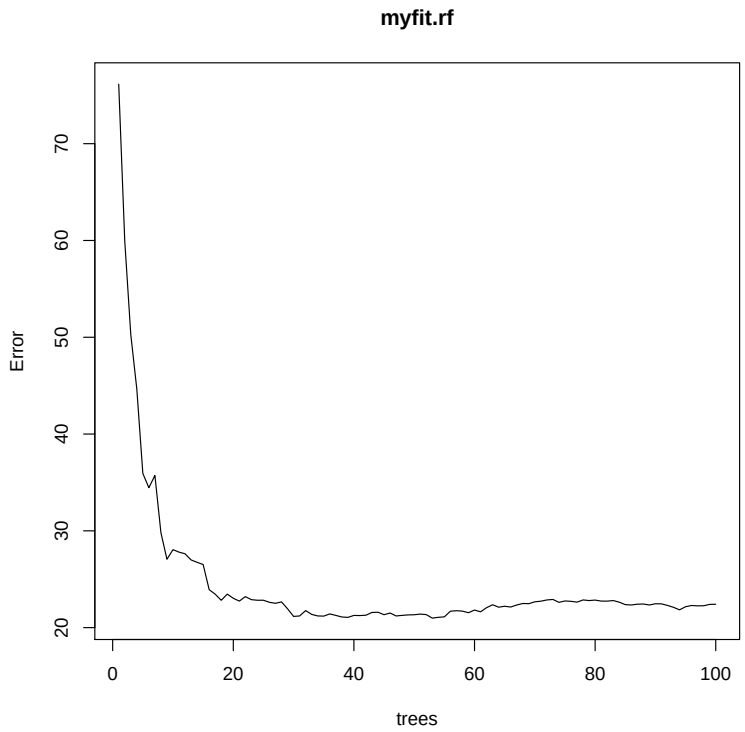
Random Forests

We apply the algorithm above to trees, modifying it as follows. To predict the response at x:

- Draw B bootstrap samples.
- For each sample, fit a tree. At each split, select the splits using only a randomly chosen subset of m variables, rather than choosing from all of them.
- Average the predictions $f(x)$ using only the OOB samples (OOB = "out-of-bag" samples, those not containing x.) Ovefitting does not arise since we are using OOB samples only.

Example: bodyfat

```
library(randomForest)
myfit.rf=randomForest(DEXfat ~ ., data=bodyfat, ntree=100,
mtry=2, maxnodes = 4, importance=TRUE)
plot(myfit.rf)
```



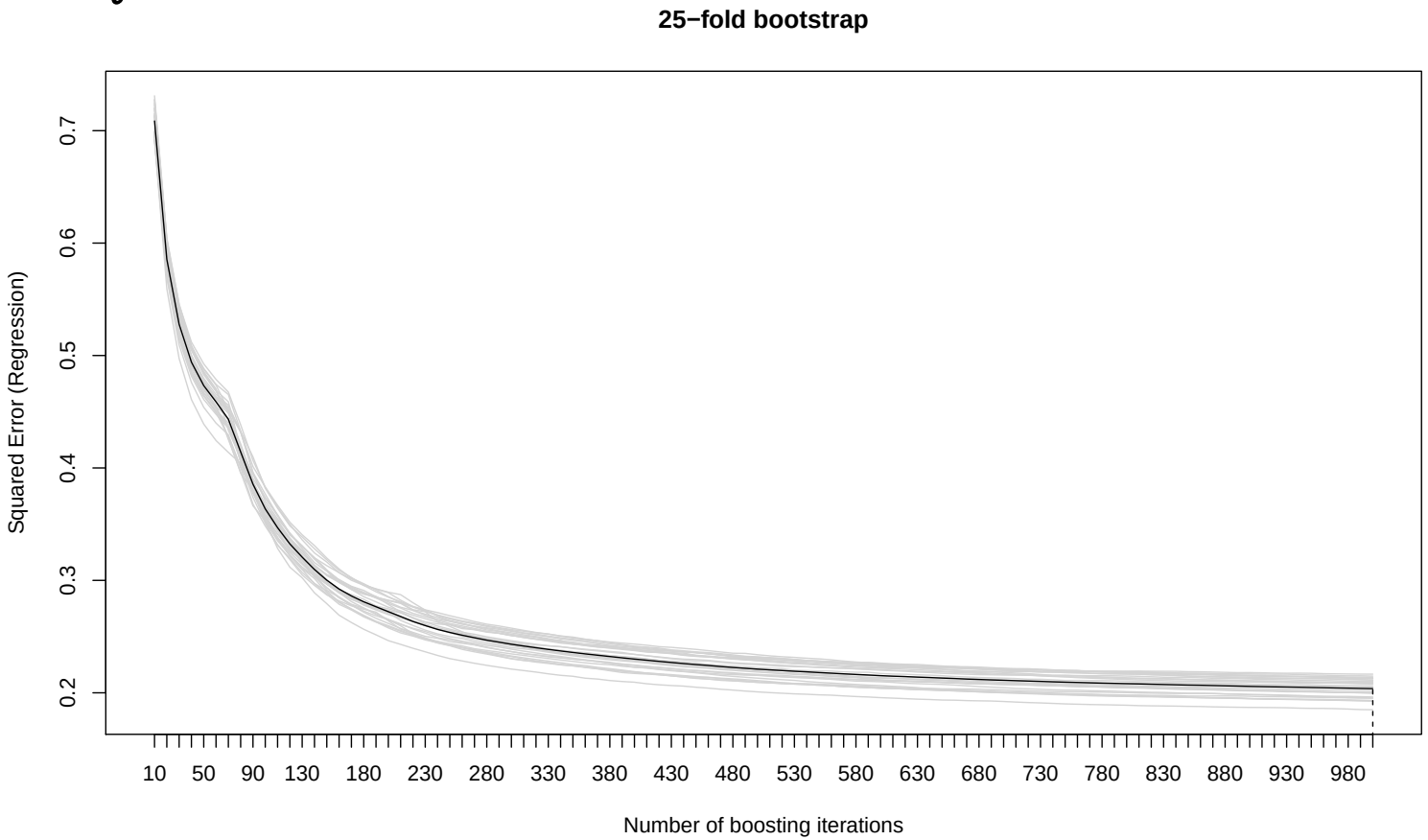
Example: the California data

We try boosting and random forests on the California housing data. First, boosting. We split the housing data into training and test sets as before. Recall that HTF recommend using a small value of ν and a big value of M.

```
> library(mboost)
> myfit = blackboost(mhv~., data = training,
+   control = boost_control(mstop = 300, nu = 0.01),
+  tree_controls = ctree_control(maxdepth = 4))
>  mean((training$mhv-predict(myfit))^2)
[1] 0.163177
> mean((test$mhv-predict(myfit,newdata=test))^2)
[1] 0.2096704
```

Error plots

```
my.risk = cvrisk(myfit, grid = 1:100*10)
plot(my.risk)
```



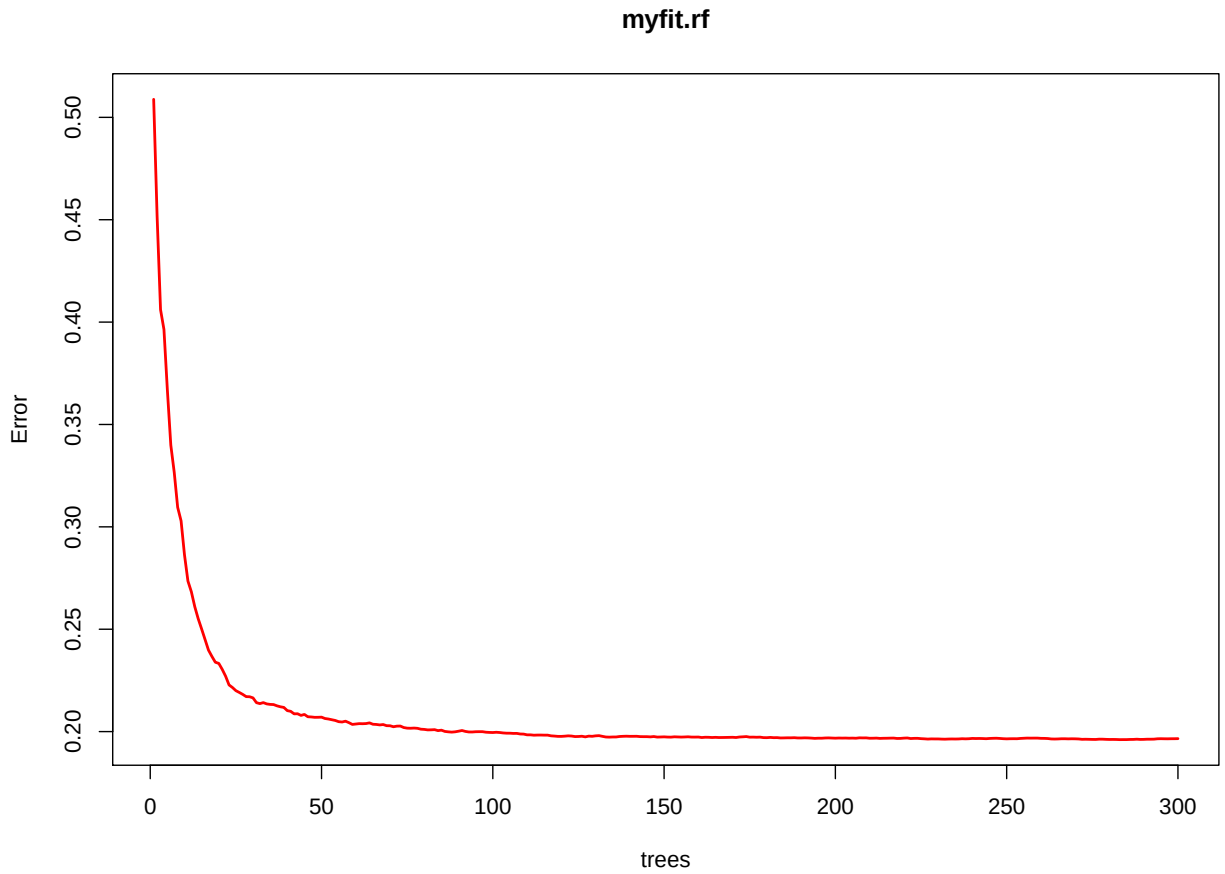
Random Forests: California data

Code for fitting random forest:

```
> myfit.rf=randomForest(mhv~., data=training, ntree=300,
   mtry=2, importance=TRUE)
> mean((training$mhv-predict(myfit.rf))^2)
[1] 0.1965578
> mean((test$mhv-predict(myfit.rf,newdata=test))^2)
[1] 0.1999865
```

Plot of MSE: choosing number of trees

```
> plot(myfit.rf, col="red", lwd=2)
```



Assessing variable importance

We would like to know which variables are important in the predictions. There are two methods for calculating an "importance measure" for each variable j. For a single tree

- method 1: For each non-terminal node, record (a) which variable we are splitting on, and (b) the decrease in RSS associated with the split. Then for each variable, sum up the RSS decreases for the nodes that were split on that variable.

Foe boosted trees, we simply add up the importances for the individual trees.

Assessing variable importance (cont)

- Method 2: In every tree grown in the forest, calculate the predictions for the OOB cases (those not included in the bootstrap sample.) Now randomly permute the values of variable j in the OOB cases and work out predictions for these. Subtract the prediction error of the unpermuted variable from that of the permuted variable. This should be large for important variables.

In both cases, we average over all trees in the forest to get an importance score for variable j.

Example: California data

```
> importance(myfit.rf)
             %IncMSE IncNodePurity
medinc      77.23701     3822.2654
medage      55.31629      520.6379
rooms       26.73666      592.1063
brooms      31.35329      426.8973
pop         43.01975      588.6907
households  32.21874      431.1025
latitude    50.94804     1741.2517
longitude   49.35661     1662.4923
```

Partial dependence plots

How can we explore the relationship between individual variables (or subsets of variables) and the predictor? One way is to consider the average value of f for a fixed value x_j of a particular variable (say X_j), averaged over the other variables. For example, suppose we want to understand the relationship between X_1 and f. For a fixed value x_1 of X_1, we can calculate

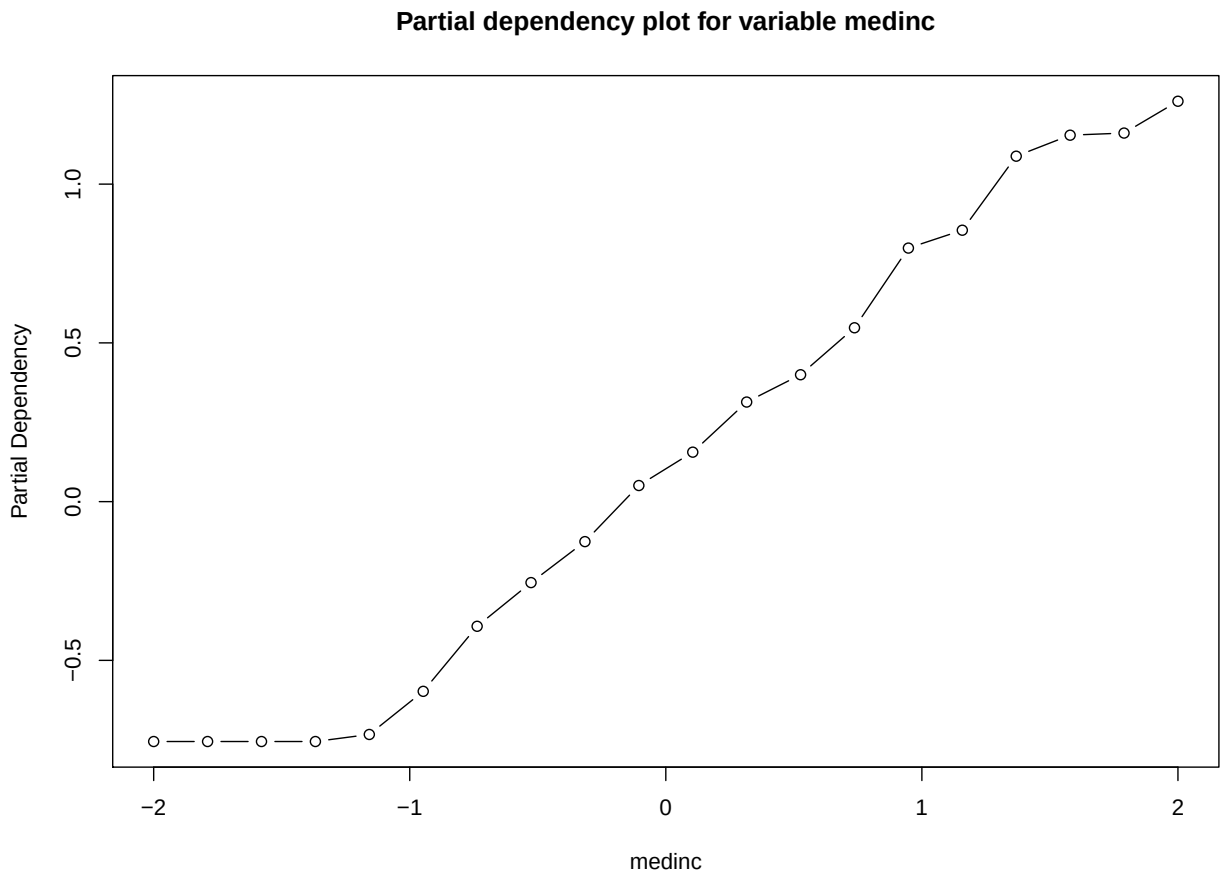
$$\frac{1}{n}\sum_{i=1}^{n}\hat{f}(x_1, x_{i2}, \ldots, x_{ik}).$$

We can repeat this for different values of x and plot the result. This is called a Partial Dependence Plot.

Code

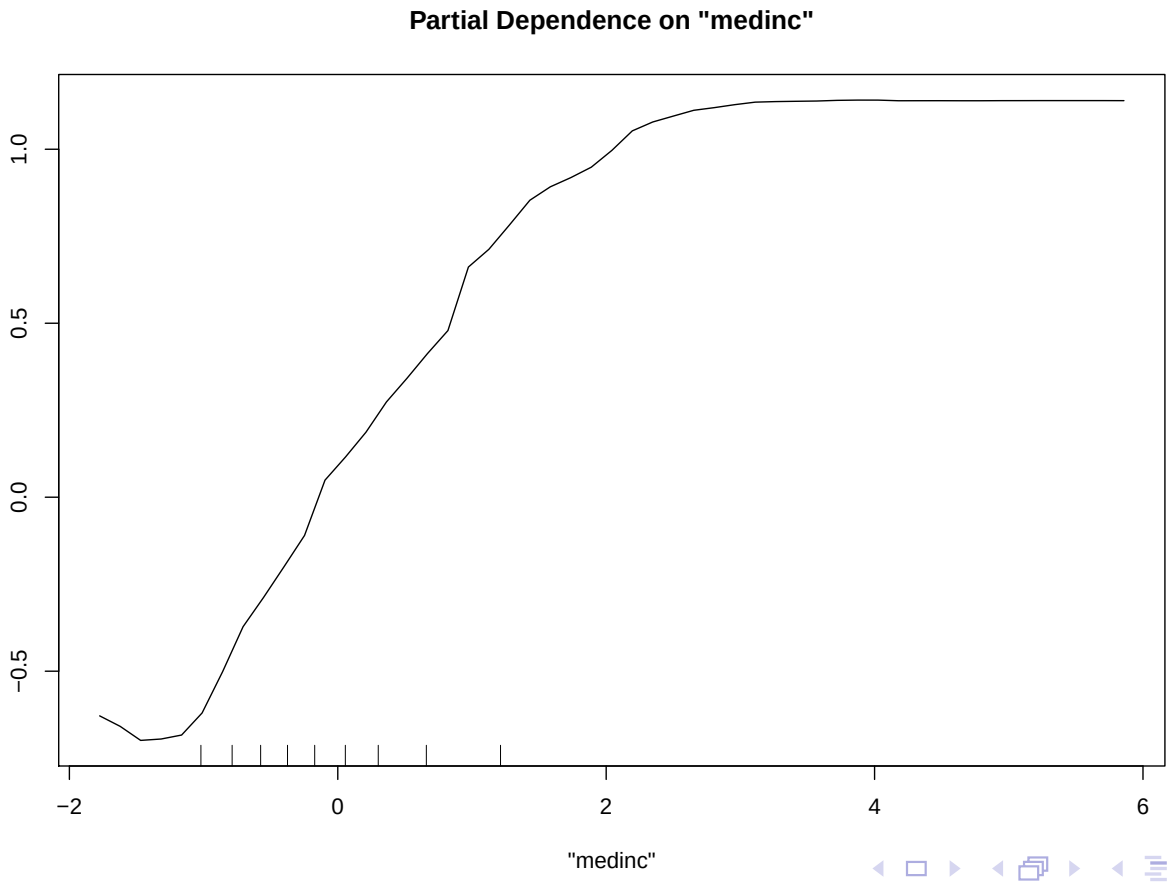
```
x = seq(-2,2, length=20)
mypdp = numeric(20)
for(i in 1:20){
  newdata = training
  newdata[,2] = x[i]
  mypdp[i]=mean(predict(myfit, newdata=newdata))
print(i)
}

plot(x,mypdp, type="b", xlab = "medinc",
ylab = "Partial Dependency",
main= "Partial dependency plot for variable medinc")
```

The plot


Partial dependency plot for variable medinc
Partial Dependency
−0.5
0.0
0.5
1.0
−2
−1
0
1
2
medinc

Dependence Plots for RF

```
partialPlot(myfit.rf, x.var = "waistcirc",
pred.data=training)
```



California Housing data: comparisons

Method	Out-of-sample error
Linear	0.356
NNet4	0.249
Tree	0.262
Boosted Tree	0.209
Random Forest	0.199

Use of the caret package

You will have seen how the various methods must be "tuned", adjusting various parameters that regulate the complexity of the fitted model. For example, in neural networks we must specify the number of units in the hidden layer, in random forests, the number of variables to select for each tree, and the depth of the trees. The parameters are chosen by examining various combinations of tuning parameters and picking the combination that has the smallest PE.

This can be tiresome, but fortunately there is an R package `caret` which automates the process. `caret` = "classification and regression training". We illustrate its use with some sample code on the next slide.
Documentation for `caret` is at
`http://topepo.github.io/caret/index.html`

Example: use of caret

Note: The arguments `decay` and `size` (the number of hidden layer units) are parameters used in the `nnet` function we used to fit a neural network. They need to be "tuned" to find the best predictor. The parameters `maxit`, `trace` and `linout` are additional parameters controlling the fitting process.

```
library(caret)
library(MASS)
data(cpus)
formula =  perf ~ syct + mmin + mmax + cach + chmin + chmax
my.grid <- expand.grid(.decay = c(0.001), .size = 2:10)

nnCV <- train(formula, data = cpuScaled,
 method = "nnet",
    maxit = 1000,
    tuneGrid = my.grid,
    trace = FALSE,
    linout = 1,
    trControl = trainControl(method="cv", number=5,
       repeats=1000))
```

Example: caret output

```
> nnCV
Neural Network

209 samples, 7 predictor
No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 168, 169, 167, 165, 167
Resampling results across tuning parameters:

  size  RMSE       Rsquared   RMSE SD     Rsquared SD
   2    0.4342915  0.8143557  0.06308931  0.06497871
   3    0.4077575  0.8375872  0.04893118  0.01492116
   4    0.4103141  0.8360537  0.05758337  0.03336740
   5    0.4422472  0.8182327  0.06859315  0.03823729
   6    0.4868674  0.7833904  0.08752861  0.06790258
   7    0.4293266  0.8229421  0.04421072  0.02381646
   8    0.5268123  0.7464646  0.10174856  0.07113539
   9    0.6079561  0.6882972  0.15257252  0.10227837
  10    0.5984902  0.7024736  0.14629595  0.11658398

Tuning parameter 'decay' was held constant at a value of 0.001
RMSE was used to select the optimal model using  the smallest value.
The final values used for the model were size = 3 and decay = 0.001.
```