Lecture 2: Modern regression and data mining tools

Alan Lee

Department of Statistics
STATS 760 Lecture 2

March 23, 2017

Outline

Introduction

Smoothing

Curse of dimensionality

Models

Today's agenda

In this lecture we will discuss several modern regression methods that are useful in data mining.

- Overview of supervised learning
- Smoothing techniques
- Regression trees
- Neural networks

We will then apply them to several interesting data sets.

Data sets

- The California housing data: data on 20640 census areas ("block groups") Response Y is average house price, to be predicted from 9 geographic and economic covariates. See https://www.stat.auckland.ac.nz/~lee/760/houses.txt
- The CPU data: data on 209 computer CPU's. Response Y is a performance measure, to be predicted from 6 other variables. See the R `MASS package.`
- The bodyfat data: data on 71 healthy females. Predict percent body fat from 9 anthropomorphic measurements. See the R `mboost` package.
- The spam data: 59 veriables measured on 4096 email messages, classified as spam or genuine. The aim is to develop a classification rule to decide if a new email is spam or genuine. Used in the design of spam filters.

Prediction, or "supervised learning"

Here we use data to make predictions about the future.

> *It's tough to make predictions, especially about the future.*

Yogi Berra, New York Yankees

Suppose our data follow a model

$$y = f(x_1, \ldots, x_K) + \epsilon$$

relating a response y to covariates $x_1, \ldots, x_K$, where f is some unknown smooth function, and K is possibly large.

We want to find an "automatic" estimate of f, to be used to predict future values of the response.

We do this by selecting an estimate of f from some class of functions (for example linear functions) by using some goodness of fit criterion, such as least squares.

Training Sets

We assume we have available a "training set", a set of data (usually rectangular) consisting of observations on both the response (sometimes called the "target") and potential covariates (inputs, features). Our task is to use these data to "train the predictor" i.e. to

- Choose a predictor $\hat{f}$ from some set of functions, hopefully a rich flexible set. (This is called "fitting the model" or "training the predictor". The selected predictor will be an approximation to the true f, hopefully a good one.
- Choose inputs to use from those in the training set. Not all may be used.

There are many possible classes of functions, for example linear functions, neural networks, trees, gams. They should be flexible (approximate the true function well) and easy to fit.

Predicting

Having fitted the model, given new data $x_1, \ldots, x_k$ we predict the corresponding value of y by

$$\hat{f}(x_1, \ldots, x_k).$$

If the response y is a category rather than a numerical quantity, we sometimes speak of "classification" rather than "prediction", and call $\hat{f}$ a "clasification rule".

Most data mining methods can be tweeked to perform both prediction of numerical quantities and classification with only minor changes in the method, using essentially the same software for either type of task.

Smoothing

If the number of inputs is small, say one or two, we can use smoothing techniques such as loess that is used at stage 2. This is based on the idea that the true relationship f between the inputs and the output is a smooth function.

If the number of inputs is large, we must use other methods and other classes of functions - see in a few slides.

Smoothing: loess

You might want to review the explanation of loess given in STATS 2001/8.

- Assume that the number of inputs is small and that the variables are all continuous.
- At each data point, fit a weighted regression, with weights given by a kernel. Points close to the data point under consideration have higher weights. The regression can be quadratic or linear.
- Use regression to predict fitted value at that point.
- Repeat for every point (subset if too many).
- Implemented using the R function `loess`.

Smoothing splines

Data are $(x_i, y_i), i = 1, \ldots n$, where x_i is of low dimension.

Choose the function f to minimise

$$\sum_{i=1}^{n}(y_i - f(x_i))^2 + \lambda \int |f''(x)|^2 dx.$$

Note:

- The first term measures the goodness of fit: how close f is to the response data.
- The integral measures the size of the second derivative: in other words the smoothness of the function f.
- The parameter λ controls the trade-off beween these two requirements of smoothness and goodness of fit.
- The minimising f is a "spline" (a piecewise cubic) with "knots" at every x-value in the data

Choosing λ

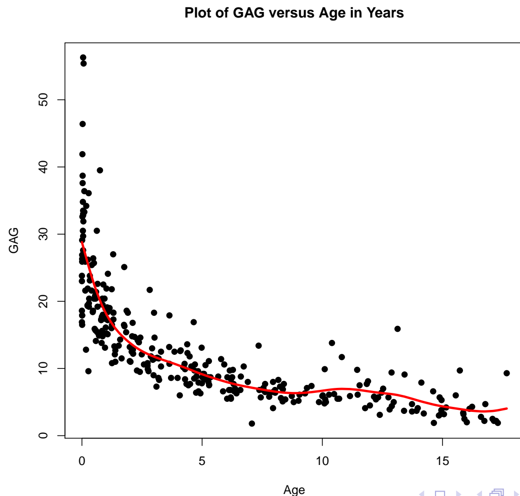
The λ parameter is chosen by cross-validation:

- Divide data into 10 parts at random.
- For fixed λ, find the function f that minimises the criterion, using 9 parts of the data
- Calculate the prediction error $\frac{1}{n}\sum(y - f(x))^2$ summed over the the tenth part (the "test set")
- Repeat using a different tenth as the "test set" and average
- Result is an estimate of the error we get if we use this value of λ to predict new data
- Choose λ to minimize this error

Example: smoothing splines

```
library(MASS)
data(GAGurine)
plot(GAG~Age, pch=19,data=GAGurine, main =
         "Plot of GAG versus Age in Years")
#use GCV to choose smoothing parameter
smooth = smooth.spline(GAGurine$Age, GAGurine$GAG)
lines(smooth$x, smooth$y, lwd=3, col="red")
```

The fit



Regression splines

- Piecewise cubics
- Choose knots i.e. equally spaced
- Generate a “spline basis” in R
- Fit using linear regression

$$y = b_1 f_1(x) + b_2 f_2(x) + \cdots + b_k f_k(x)$$

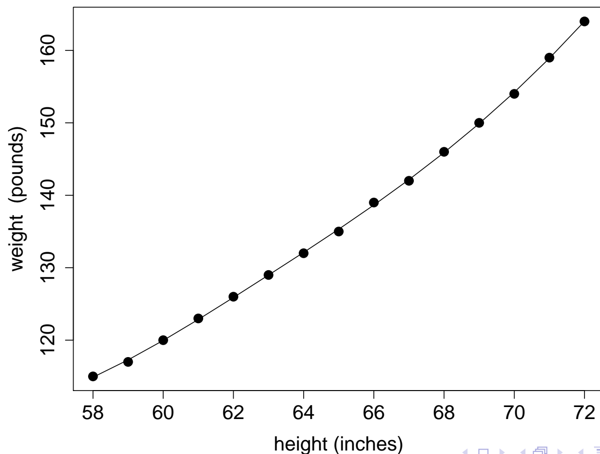
Example: regression splines

```
# Average height and weight of American women 30-39
> women
  height weight
1      58    115
2      59    117
3      60    120
4      61    123
5      62    126
6      63    129
7      64    132
8      65    135
9      66    139
10     67    142
11     68    146
12     69    150
13     70    154
14     71    159
15     72    164
```

R code

```
> round(bs(women$height, df = 5),5)
            1       2       3       4       5
 [1,] 0.00000 0.00000 0.00000 0.00000 0.00000
 [2,] 0.45344 0.05986 0.00164 0.00000 0.00000
 [3,] 0.59694 0.20335 0.01312 0.00000 0.00000
 [4,] 0.53380 0.37637 0.04428 0.00000 0.00000
 [5,] 0.36735 0.52478 0.10496 0.00000 0.00000
 [6,] 0.20016 0.59503 0.20472 0.00009 0.00000
 [7,] 0.09111 0.56633 0.33673 0.00583 0.00000
 [8,] 0.03125 0.46875 0.46875 0.03125 0.00000
 [9,] 0.00583 0.33673 0.56633 0.09111 0.00000
[10,] 0.00009 0.20472 0.59503 0.20016 0.00000
[11,] 0.00000 0.10496 0.52478 0.36735 0.00292
[12,] 0.00000 0.04428 0.37637 0.53380 0.04555
[13,] 0.00000 0.01312 0.20335 0.59694 0.18659
[14,] 0.00000 0.00164 0.05986 0.45344 0.48506
[15,] 0.00000 0.00000 0.00000 0.00000 1.00000
reg.spline<- lm(weight ~ bs(height, df = 5), data = women)
plot(women$height, women$weight)
lines(women$height, predict(reg.spline))
```

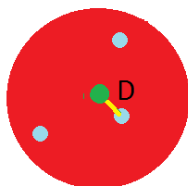
The fit



Curse of dimensionality

Why doesn't smoothing work in higher dimensions?

- Consider n points scattered at random in a K-dimensional unit sphere
- Let D be the distance between the centre of the sphere to the closest point:



K=1

K=2

Median of D

	n=100	n=200	n=500	n=1000
K=1	0.01	0.00	0.00	0.00
K=2	0.08	0.06	0.04	0.03
K=5	0.37	0.32	0.27	0.23
K=10	0.61	0.57	0.52	0.48
K=20	0.78	0.75	0.72	0.70
K=100	0.95	0.94	0.94	0.93

Conclusions

- Smoothing doesn't work in high dimensions: points are too far apart, and smoothing works by averaging responses of cases whose covariates are close.
- Solution: pick an estimate of f from a class of functions that is flexible enough to match the true f reasonably closely.
- We need to be able to compute the estimate easily.

Classes of functions and models

We will consider the following:

- Linear functions
- Additive functions (additive models)
- CART (regression trees)
- Neural networks

There are others such as Projection pursuit and MARS discussed in HTF.

Linear functions

These are functions of the form

$$f(x) = \beta_0 + \beta_1 x_1 + \cdots \beta_k x_k,$$

in other words the standard linear regression model. We fit the model by least squares, solving the normal equations. We may constrain some of the β's to be zero (subset selection). The predictor is

$$\hat{f}_Z(x) = \hat{\beta}_0 + \hat{\beta}_1 x_1 + \cdots \hat{\beta}_k x_k.$$

The $\hat{\beta}$'s are functions of the training set.

Additive models

These are functions of the form

$$f(x) = \phi_1(x_1) + \cdots \phi_k(x_k),$$

i.e. the idea of an additive model. The ϕ's are smooth functions estimated from the training set using the backfitting algorithm (see HTF p 298). The predictor is

$$\hat{f}_Z(x) = \hat{\phi}_1(x_1) + \cdots \hat{\phi}_k(x_k)$$

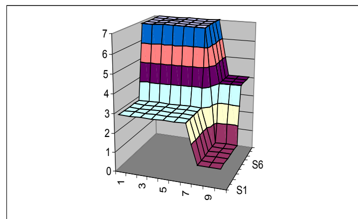
The $\hat{\phi}$'s are functions of the training set.

Ref:V&R p 232, HTF p 295, Ch9

Regression trees (CART)

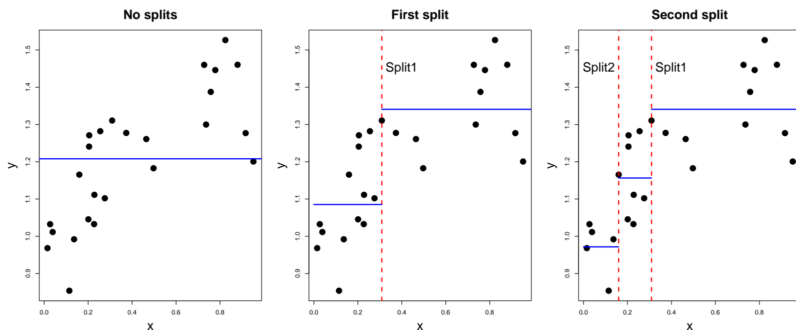
- This class of functions consists of multi-dimensional step functions, constant on regions of X-space.
- They can be defined by means of a tree structure, that enables a nice explanation of how the function behaves.
- The input space is divided into regions in a recursive manner, and the predictor has a constant value on each region.

Ref: V&R p251, ISL p303 (Ch8) HTF p305 (Ch9)



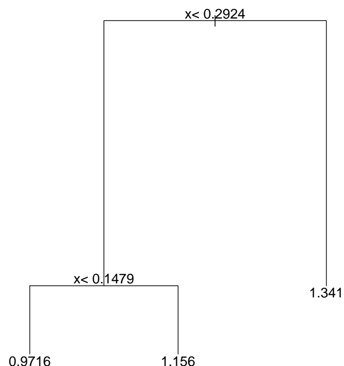
Regression trees (cont)

We illustrate with a simple example with one input. We will approximate successively with step functions:



Regression trees (cont)

We can represent the splitting process with a tree:

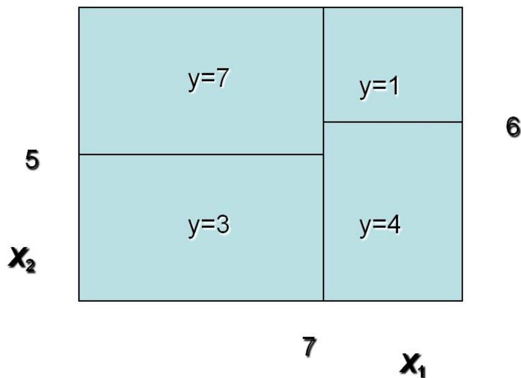


Regression trees (cont)

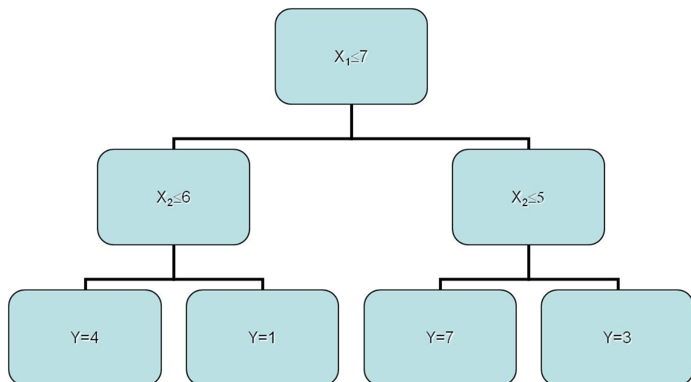
- Each node of the tree corresponds to a region of the input space.
- Each new split divides one region into two new regions.
- We choose the split to get the greatest reduction of the residual sum of squares - see later for math details.

Regression trees (cont)

In two dimensions:



Regression trees (cont)



Regression trees (cont)

To fit a regression tree, we "grow" the tree by adding two "child" nodes to an existing node (splitting the region in two). We start with the "root" node, containg all the observations. To add nodes:

- Split the data contained in a node (region) into two complementary subsets S and S', according as $x_j \leq c$ or $x_j > c$. The split will depend on the variable used (x_j) and the threshold (c).
- Work out the reduction in sum of squares:

$$\text{Residual SS of data at node}(= \sum_i (y - \bar{y})^2)$$

$$-\text{Residual SS of data in S} - \text{Residual SS of data in S}'$$

Regression trees (cont)

- Choose the split that gives the biggest reduction. The two new nodes correspond to the two subsets S and S'.
- Carry on this process until nodes contain fewer that a fixed number of observations (say 5).
- The terminal nodes correspond to regions of X-space that partition the space. The value of the fitted function $\hat{f}$ on a region is the mean of the y values of the observations contained in that node.

Pruning

The process on the previous slide will result in a tree that is too complex. To obtain a tree that will predict better, we "prune" the tree. Suppose we delete all nodes that are below some fixed node. This results in a subtree, T say. Consider the criterion

$$\text{Residual SS of } T + \alpha \times \text{Number of terminal nodes of } T,$$

where α is some fixed number. We keep pruning until the criterion begins to increase. This is equivalent to the R^2 decreasing by more than the quantity $cp = \alpha / RSS_{NULL}$ where RSS_{NULL} is the residual sum of squares of the null model. Note that pruning always decreases the R^2 (why?)

The value of α (equivalently cp) is chosen to give the minimum prediction error (see later for exactly what this means)

R function: `rpart`

Example: the CPU data see also V&R p258

```
> library(stats)
> data(cpus)
> head(cpus)
             name syct mmin  mmax cach chmin chmax perf estperf
1   ADVISOR 32/60  125  256  6000  256    16   128  198     199
2   AMDAHL 470V/7   29 8000 32000   32     8    32  269     253
3   AMDAHL 470/7A   29 8000 32000   32     8    32  220     253
4 AMDAHL 470V/7B   29 8000 32000   32     8    32  172     253
5 AMDAHL 470V/7C   29 8000 16000   32     8    16  132     132
6   AMDAHL 470V/8   26 8000 32000   64     8    32  318     290
```

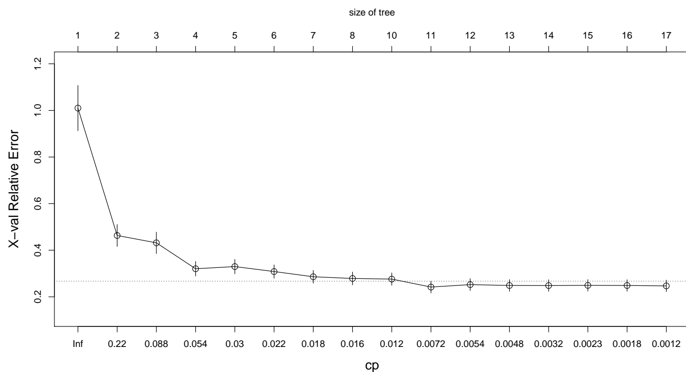
Example (cont)

```
> library(MASS)
> data(cpus)
> library(rpart)
> # tree, with cp=0.001, using reduction in RSS criterion
> formula =  log(perf) ~ syct + mmin + mmax + cach + chmin + chmax
> treeFit = rpart(formula, data=cpus, method="anova", cp = 0.001)
> # calculate R^2
> nullSS = sum((log(cpus$perf)-mean(log(cpus$perf)))^2)
> 100*(1-sum(residuals(treeFit)^2)/nullSS)
[1] 88.34548
> # linear model fit R^2
> summary(lm(formula, data=cpus))$r.squared
[1] 0.8130141
```

R^2 is 88.3% (linear model fit is 81.3%)

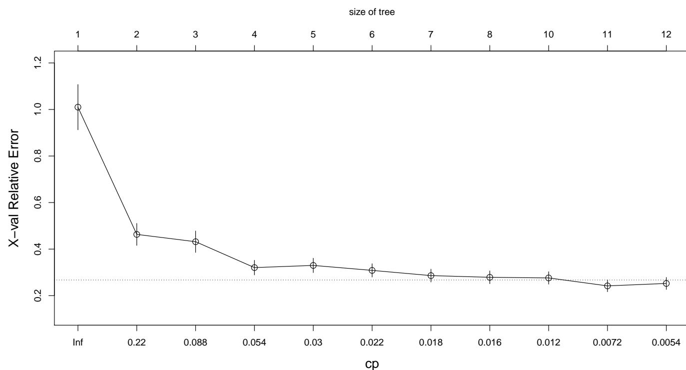
Example: pruning

```
plotcp(treeFit)
```



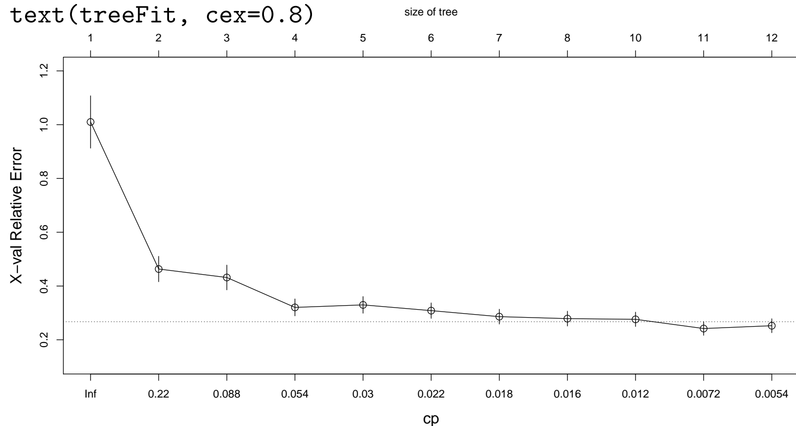
Example: pruning (cont)

```
treeFitPruned = prune(treeFit, cp=0.0054)
plotcp(treeFitPruned)
```



Example (cont)

```
plot(treeFit)
text(treeFit, cex=0.8)
```



Example: predicting

```
> newdata
    syct mmin  mmax cach chmin chmax
200   30 8000 64000  128    12   176
201  180  262  4000    0     1     3

> predict(treeFitPruned, newdata=newdata)
       1        2
6.140506 3.366543
```

True values of `log(perf)` were 7.047517 and 2.484907

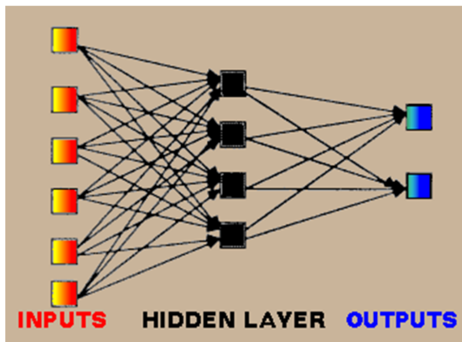
Neural networks

Another class of flexible non-linear functions are the neural networks. These originated in the fields of cognative science and artificial intelligence, and are surrounded by a great deal of hype and mystery. Have a look at the Wikipedia article "Artificial Neural Networks" to get a taste of this. One source of the mystery is the metaphorical language used to describe neural networks, particularly in the artificial intelligence literature.

However, we shall regard them as just another family of functions and ignore the hype. We will consider the simplest type, the single layer feed-forward network. This best described by the diagram on the next slide.

Neural networks: the picture

The diagram explains the process which connects the inputs to the outputs. (This is supposed to mimic the way brain signals are transmitted between neurons). The middle layer is the "hidden layer": we can adjust the fit by changing the number of hidden layer nodes.



Neural networks: mathematical description

Let $\sigma(x) = exp(x)/(1 + exp(x))$. This is called the sigmoid function. The neural network function is made up of non-linear transformations (using the sigmoid function) of linear combinations of the inputs:

$$f(x) = \alpha_0 + \sum_h \beta_h \sigma(\alpha_{0h} + \sum_i \alpha_{ih} x_i)$$

Each hidden layer node h collects a linear combination of inputs, transforms them and transmits them to the output node, where they are combined linearly to form the final output.
Ref: V&R p243, HTF p392 (Ch9)

Fitting neural networks

- Since the NN function depends on a finite set of parameters (weights), the function can be fitted by non-linear least squares, using a specialized iterative algorithm (see HTF p 391 for details).
- The choice of starting values is important as the objective function has multiple local minima - several starts should be tried.
- It is advantageous to add a “regularization penalty” (proportional to the sum of the squared weights) to the least squares criterion to assist the optimization process. If this is done the input data should be standardized.

R fitting function: `nnet` R predicting function: `predict`

Example: CPU data

```
> # neural network
> library(nnet)
> # scale data
> cpu = cpus[,-1] # delete computer names
> cpu$logperf = log(cpu$perf)
> cpu = cpu[,-c(7,8)]  # drop perf, estperf
> cpuScaled = data.frame(scale(cpu))
>
> formula =  logperf ~ syct + mmin + mmax + cach + chmin + chmax
> nnFit = nnet(formula, data=cpuScaled, size=3, linout=TRUE,
+                          decay = 0.01, maxit=500 )
# weights:  25
initial  value 272.024214
iter  10 value 46.813430
............
iter 170 value 25.480517
final  value 25.480516
converged
```

R^2 is 88.2%. Recall the linear model fit has an R^2 of 81.3%.

Example: CPU data

Weights:

```
> summary(nnFit)
 a 6-3-1 network with 25 weights
options were - linear output units  decay=0.01
 b->h1 i1->h1 i2->h1 i3->h1 i4->h1 i5->h1 i6->h1
  0.14   0.68  -0.10  -0.33  -0.36   0.00  -0.09
 b->h2 i1->h2 i2->h2 i3->h2 i4->h2 i5->h2 i6->h2
 -5.52  -0.54  -0.05  -2.76  -1.86  -1.11  -1.08
 b->h3 i1->h3 i2->h3 i3->h3 i4->h3 i5->h3 i6->h3
 -0.25   2.89   0.77  -0.22  -1.10   0.17  -0.57
 b->o h1->o h2->o h3->o
 2.47 -5.45 -3.02  1.60
```

Example: Predicting CPU performance

Predicting using new data:

```
 > newdata
           syct       mmin       mmax       cach     chmin      chmax
200 -0.66787448  1.3231141  4.4517597  2.5300884  1.071177  6.0672434
201 -0.09153423 -0.6718623 -0.6648284 -0.6203922 -0.542608 -0.5872891
> predict(nnFit, newdata=newdata)
           [,1]
200  2.5396976
201 -0.9410328
> cpuScaled[200:201,7]
[1]  2.871518 -1.480719
```