Machine Learning

Week 2

Mladen Kolar (`mkolar@chicagobooth.edu`)

Recall from Week 1

Supervised learning

Training experience: a set of labeled examples (samples, observations) of the form $(x_1, y_1), (x_2, y_2), \ldots, (x_i, y_i), \ldots, (x_n, y_n)$ where $x_i = (1, x_{i1}, \ldots, x_{ip})$ are vectors of input variables (covariates, predictors) and y is the output

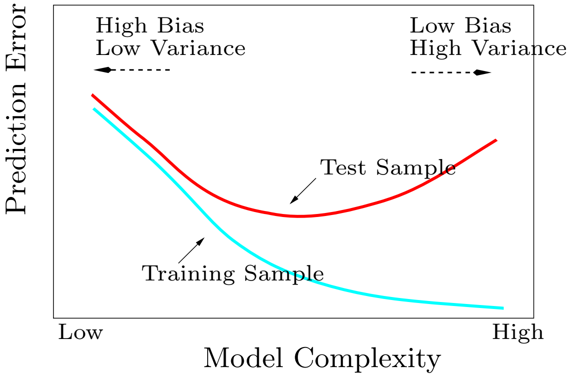
This implies the existence of a "teacher" who knows the right answers

What to learn: A function $f : \mathcal{X}_1 \times \mathcal{X}_2 \times \cdots \times \mathcal{X}_p \mapsto \mathcal{Y}$ which maps the input variables into the output domain

Goal: minimize the error (loss) function

- Ideally, we would like to minimize error on all possible instances
- But we only have access to a limited set of data ...

Bias-Variance Trade-Off



Recap: Out-of-Sample Validation

Validation-set approach

K**-fold cross validation**

Recap: Out-of-Sample Validation

Validation-set approach **K-fold cross validation**

- Give an estimate of the true test error. Tend to overestimate the error.
- Can be used to choose model complexity and to compare different models.

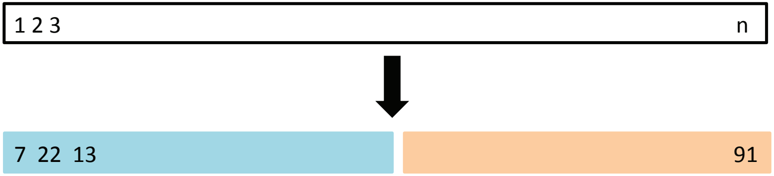
Recap: Out-of-Sample Validation

Validation-set approach	**K-fold cross validation**

- Give an estimate of the true test error. Tend to overestimate the error.
- Can be used to choose model complexity and to compare different models.

Validation-set approach	K-fold cross validation
Highly variable for different splits	Less variable for different splits
Sensitive to split proportion	Less sensitive to choice of K
Computationally less expensive	Computationally more expensive

Validation-set approach

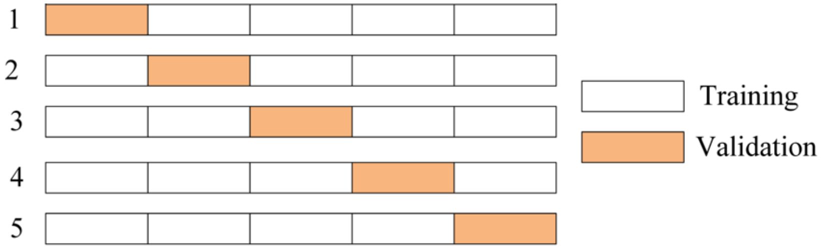


A random splitting into two parts: left part is training set, right part is validation set

The model is fitted on the training set, and the fitted model is used to predict the responses for the observations in the validation set.

K-Fold Cross-Validation

Divide data into K roughly equal-sized parts ($K = 5$ here)



K-Fold Cross-Validation in Detail

Let the K parts be $C_1, C_2, \dots, C_K$, where C_k contains the indices of the observations in part k.
There are n_k observations in part k: roughly, $n_k = n/K$.

Compute the cross-validation score

$$\mathrm{CV} = \sum_{k=1}^{K} \frac{n_k}{n} \mathrm{MSE}_k \left(\approx \frac{1}{K} \sum_{k=1}^{K} \mathrm{MSE}_k \right)$$

where

$$\mathrm{MSE}_k = \frac{1}{n_k} \sum_{i \in C_k} (y_i - \hat{y}_i^{(k)})^2$$

- $\hat{y}_i^{(k)}$ is the fit for observation i, obtained from the data with part k removed.

More on k-NN

More on k-Nearest Neighbors, $p > 1$

We have looked at simple examples of kNN (with one x!!).

In this section we look at kNN more carefully, in particular, how do you use kNN when x has p variables??!!

An important advantage of kNN is that it is feasible for *BIG DATA*, big n **and** big p.

The k-nearest neighbors algorithm will try to **predict** based on similar (close) records on the **training dataset**.

Remember, the problem is to guess a future value Y_f given new values of the covariates $X_f = (x_{1f}, x_{2f}, x_{3f}, \dots, x_{pf})$.

kNN:

What do the Y's look like in the region around X_f?

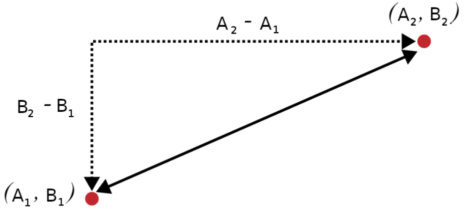
We need to find the k observations in the training dataset that are close to X_f. How? "Nearness" to the i^{th} neighbor can be defined by (Euclidean distance):

$$d_i = \sqrt{\sum_{j=1}^{p} (x_{jf} - x_{ji})^2}, \quad x_i \text{ in training data}$$

Prediction:

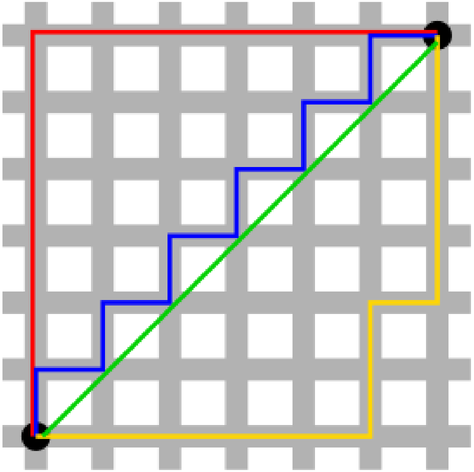
Take the average of the $Y's$ in the k-nearest neighborhood.
Average y_i corresponding to k smallest d_i.

Euclidean distance



$$\sqrt{(A_1 - A_2)^2 + (B_1 - B_2)^2}$$

Another distance – Manhattan



In general:

$$|A_1 - A_2| + |B_1 - B_2| + \ldots + |Z_1 - Z_2|$$

Note:

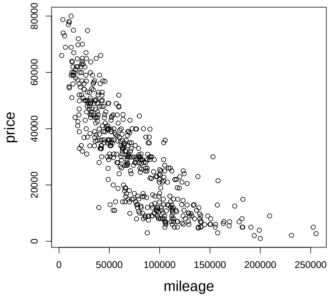
- The distance metric used above is only valid for numerical values of X. When $X's$ are categorical we need to think about a different distance metric or perform some manipulation of the information.
- The scale of X also will have an impact. In general it is a good idea put the $X's$ in the same scale before running kNN.

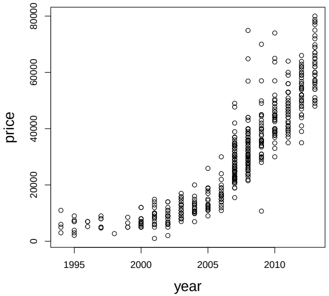
What do we mean by scale?

If weight is in pounds you get one distance, if weight is in kilograms you get a different number!!

To see how this works, let us also look at `year`:

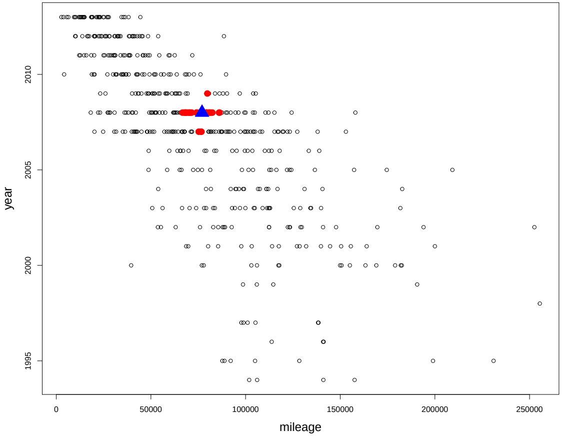
x_1=mileage
x_2=year



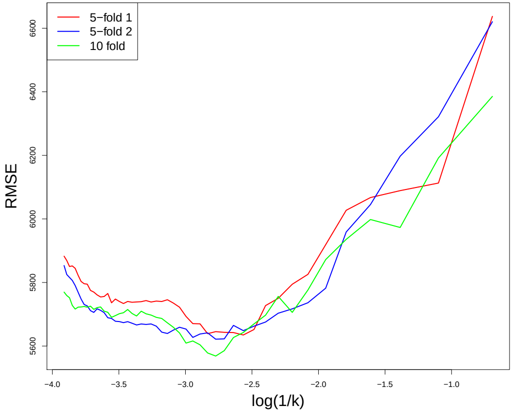


Hmm. how is y=price related to x_2=year ?

To predict y at the blue triangle, we average the y values corresponding to the red points.

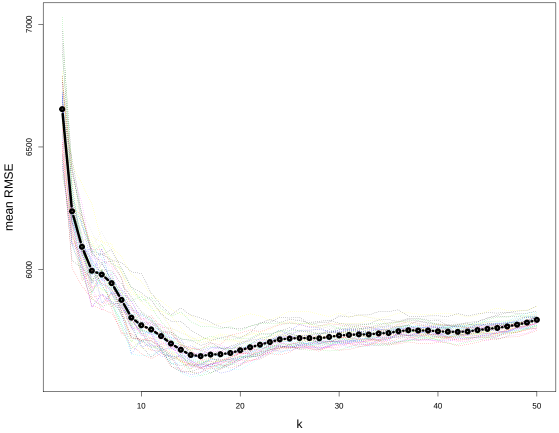


Let's do the CV and see what works out of sample.



Seems to indicate a much smaller k (than when we just used `mileage`), but it is very noisy.

Let's average many CV's.

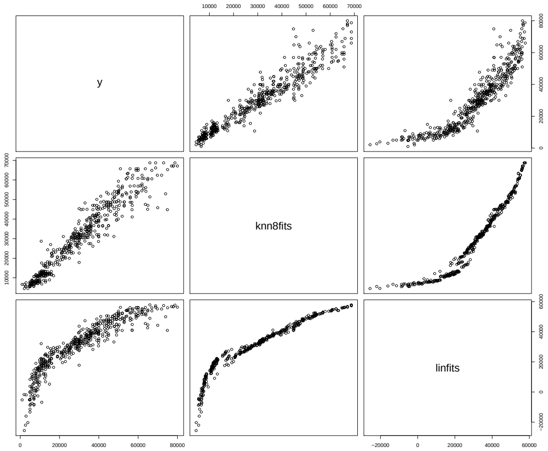


Indicates a *k* of about 15, which is much smaller than when we just had `mileage`. Minimum MSE is smaller than when we just had `mileage`.

What do we do next, after having used cross-validation to choose a model (or tuning parameter)?

It may be an obvious point, but worth being clear: we now fit our estimator to the entire training set (x_i, y_i), $i = 1, \ldots, n$, using the selected model.

Refit using all the data and $k = 15$.



Matching

A lot of data-mining methods work by matching.

- Which ones are most like you ??
- Which training x are most like x_f ??
- Shoppers **like you** have bought . . .

"Like" means a choice of distance, and getting the distance right in high dimensions can be very hard.

Summary of kNN

kNN is a powerfull, widely used, intutitive technique.

In principle, we can use kNN for large n and p.

We have used it to illustrate the Bias-Variance tradeoff which is **a fundamental concept**.

Note:

- Choosing the distance can be tricky.
- Using distance in high dimensions can be tricky.
- Rescaling all the (numeric) x's is common.

Doing CV with a Bigger n

The Used Cars data we have been using as an example only has $n = 1000$ observations.

While the big ideas are the same, some things will work out differently with larger n.

Let's do $n = 20,000$ to illustrate.

The key differences will be:

- We won't have to rerun the CV many times and average.
  With the larger sample size, you will get much less variation in the CV results.

- We will start by leaving out a **test** data set.

We will use CV on the remaining data to make modeling decisions, and then apply our data to the test data to see you well we predict out of sample.

When we refit using all the Used Cars data, we did not have any true out-of-sample data !!

kNN: California Housing

Data: Median home values in census tract plus the following information

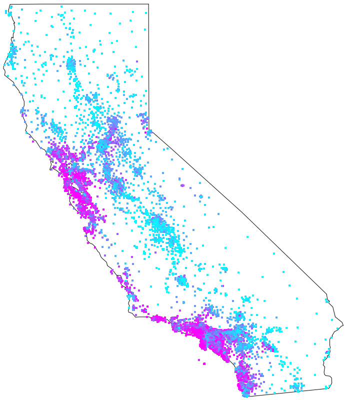
- Location (latitude, longitude)
- Demographic information: population, income, etc...
- Average room/bedroom number, home age
- Let's start using just location as our $X's$... euclidean distance is quite natural here, right?

Goal: Predict $log(MedianValue)$

There are 20,640 observations and 8 x's.

We should spend a long time plotting the data, but let's suppose we are in a hurry.

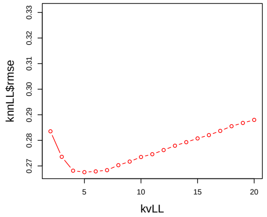
y=logMedVal
vs longitude and latitude.

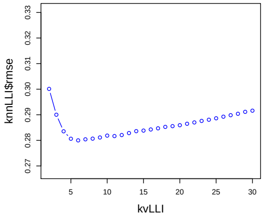


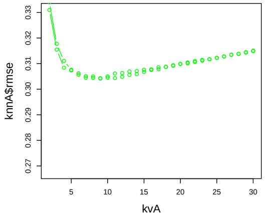
10,000 in train. Rest in test. Standardize each x: (x-m)/s.

Do 5-fold cross-validation on train.

Red: longitude and latitude
Blue: longitude and latitude and Income
Green: all 8 x's (2 runs).







Pretty small k values.
All x is worst.

And, here are the rmse's on the test data.

```
rmse test, long,lat: 0.2533981
rmse test, long,lat,income: 0.2628331
rmse test, all: 0.2897788
```

location, location, location. . . . . .

Classification

Classification

Thus far, regression: predict a continuous value given some input

Classification is a predictive task in which the response takes values across discrete categories (i.e., not continuous), and in the most fundamental case, two categories (response variable is binary: $Y = 0$ or 1).

- Buy or not buy.
- Win or lose.
- Sick or healthy.
- Pay or default.
- Thumbs up or down.

We want to learn a mapping $f : X \mapsto Y$

- X are predictor (input) variables
- Y is the target class or class label

Similar to our regression setup, we observe pairs (x_i, y_i), $i = 1, \dots, n$, where y_i gives the class of the i-th observation, and $x_i \in \mathbb{R}^p$ are the measurements of p predictor variables

Though the class labels may actually be $y_i \in \{\text{healthy, sick}\}$ or $y_i \in \{\text{up, down, } \dots\}$, but we can always encode them as $y_i \in \{0, 1, \dots, K-1\}$ where K is the total number of classes.

How do we estimate f?

Given training data $(x_i, y_i), i = 1, \ldots, n$
we construct the classification rule by $\hat{f}(x)$

Given any $x \in \mathbb{R}^p$, $\hat{f}(x)$ returns

- a class label $\hat{f}(x) \in \{0, \ldots, K-1\}$
- an estimate of the probability $P(Y \mid X = x)$

- It is often more valuable to know what is the probability that a person is going to respond to a marketing campaign, for example.

How do we estimate f?

There is a large number of different machine learning algorithms that estimate $\hat{f}(x)$

- logistic regression
- decision trees
- random forests
- neural networks
- ...

As before, there are two different ways of assessing the quality of $\hat{f}$

- its predictive ability
- interpretative ability

In what situations would we care more about prediction error?

And in what situations more about interpretation?

Example: Credit Card Default

```r
library(ISLR)
df = Default
df$Y = as.numeric(df$default)-1
head(df)
```

```
##   default student balance income Y
## 1      No      No     730  44362 0
## 2      No     Yes     817  12106 0
## 3      No      No    1074  31767 0
## 4      No      No     529  35704 0
## 5      No      No     786  38463 0
## 6      No     Yes     920   7492 0
```

Example: Credit Card Default

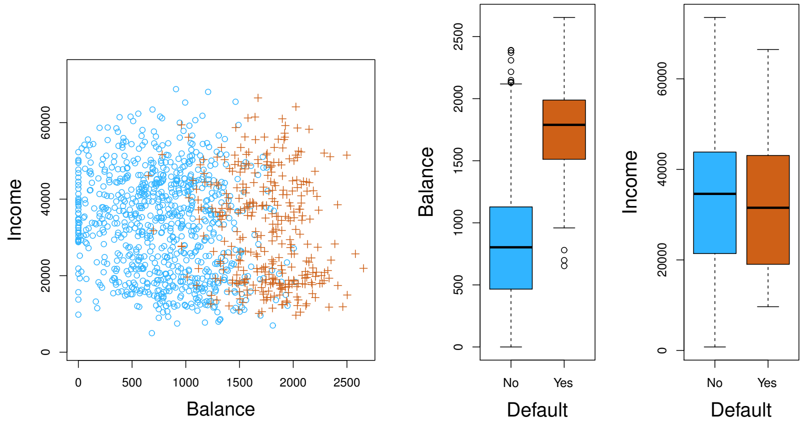
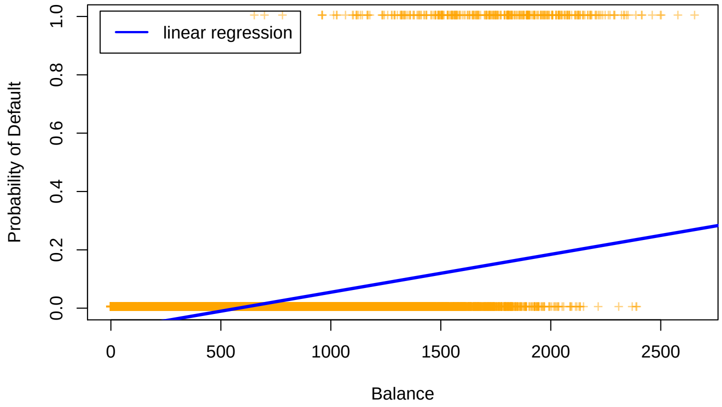


Figure from "An Introduction to Statistical Learning"

Can We Use Linear Regression?



$$p = \beta_0 + \beta_1 \times \text{Balance}$$

Can We Use Linear Regression?

We can, but the fit is very unappealing, as is the interpetation.

```
summary(lm(Y~balance, data=df))
```

```
##
## Call:
## lm(formula = Y ~ balance, data = df)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -0.2353 -0.0694 -0.0263  0.0200  0.9905
##
## Coefficients:
##              Estimate Std. Error t value
## (Intercept) -7.52e-02   3.35e-03   -22.4
## balance      1.30e-04   3.47e-06    37.4
##             Pr(>|t|)
## (Intercept)   <2e-16 ***
## balance       <2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.168 on 9998 degrees of freedom
## Multiple R-squared:  0.123,  Adjusted R-squared:  0.122
## F-statistic: 1.4e+03 on 1 and 9998 DF,  p-value: <2e-16
```

Sometimes we just have a bad fit.

Can We Use Linear Regression?

Least square regression is not ideal

- What does $\hat{y} = 1.05$ or $\hat{y} = -0.4$ mean as a probability?
- Would the assumptions of the multiple regression model be satisfied?

Ideally, we would like a method that ensures the probability is between 0 and 1.

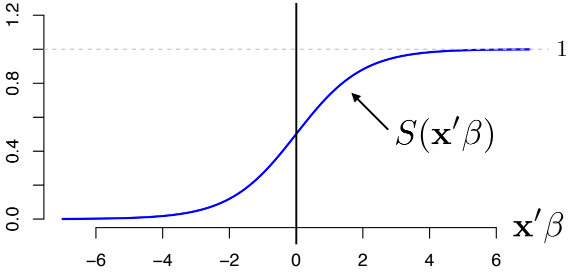
Goal: transform the probability to some quantity taking values on $(-\infty, \infty)$ and model it as a linear combination of the predictors.

$$\text{Transformation}(p) = \beta_0 + \beta_1 x_1 + \cdots + \beta_p x_p.$$

Logistic regression

Modeling directly probability $p(X) = P(Y = 1 \mid X)$

$$P(Y = 1 \mid X = x) = S(x^T\beta), \quad S(a) = \frac{\exp(a)}{1 + \exp(a)}$$

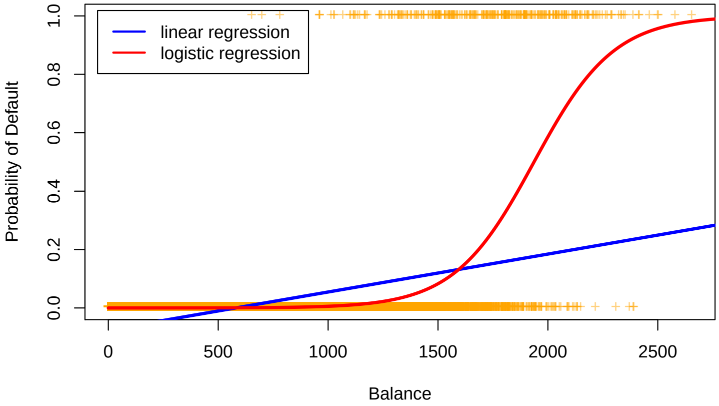


A bit of rearrangement gives

$$\mathrm{logit}(p) = \log\left(\frac{p}{1-p}\right) = x^T\beta = \beta_0 + \beta_1 \cdot x_1 + \beta_2 \cdot x_2 + \cdots$$

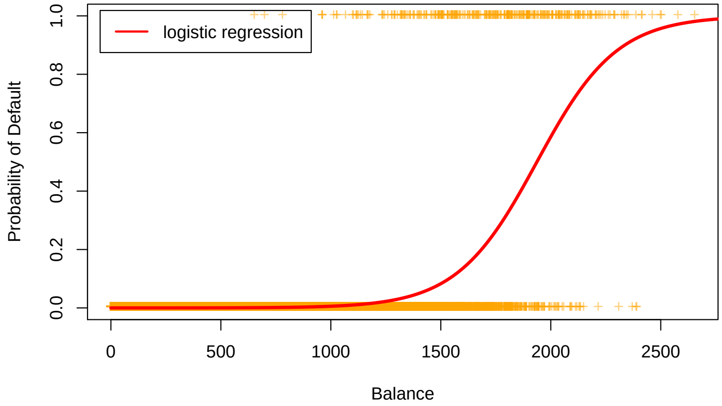
This monotone transformation is called the *log odds* or *logit* transformation of $p(X)$.

From Linear Regression to Logistic Regression



$$p = \beta_0 + \beta_1 \times \text{Balance} \quad \text{vs} \quad \text{logit}(p) = \beta_0 + \beta_1 \times \text{Balance}$$

From Linear Regression to Logistic Regression



$$\text{logit}(p) = \beta_0 + \beta_1 \times \text{Balance} \quad \Rightarrow \quad p = \frac{e^{\beta_0+\beta_1\times\text{Balance}}}{1+e^{\beta_0+\beta_1\times\text{Balance}}}$$

Logistic regression

Directly models $P(Y = 1 \mid X)$

- the logit transformation of $p(x)$ is assumed to be linear in the input variables

$$\mathrm{logit}(p) = \beta_0 + \beta_1 \cdot x_1 + \beta_2 \cdot x_2 + \cdots$$

- coefficients are estimated using maximum likelihood
- we use `glm` function in R
- see Max Farrell BUS 41100 lecture notes
  `https://maxhfarrell.com/bus41100/`

However, in this class we want to learn a more generic specification for $Y \mid X = x$.

Classification and Regression Trees

CART (Classification and Regression Trees)

Tree based methods are a major player in data-mining.

Good:

- flexible fitters, capture non-linearity and interactions.
- do not have to think about scale of x variables.
- handles categorical and numeric y and x very nicely.
- fast.
- interpretable (when small).

Bad:

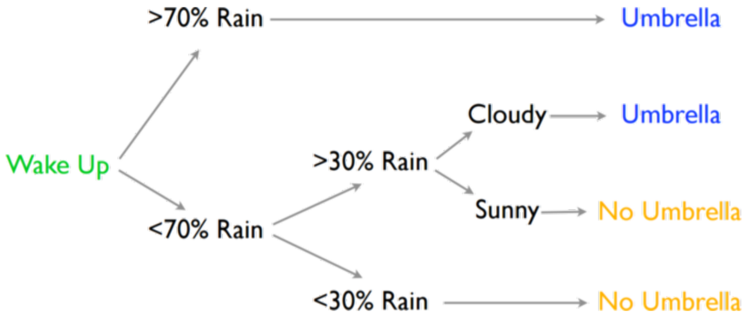
- Not the best in out-of-sample predictive performance (but not bad!).

But, if we bag or boost trees, we can get the best off-the-shelf prediction available. Bagging and Boosting are ensemble methods that combine the fit from many of tree models to get an overall predictor.

Decision tree

Decision tree represents a collection of "if-then" rules, each giving a tailored action for a particular situation.

- They offer a graphical quick-reference outlining procedures in hospitals, emergency response, insurance underwriting, etc.



- Tree-logic uses a series of steps to come to a conclusion.

Tree model

Tree model combines a number of mini-decisions.

Tree-based *supervised learning* methods port that idea to regression and classification tasks.

The idea is partition the input space into a set of rectangles, or set membership rules (in the case of categorical predictors)

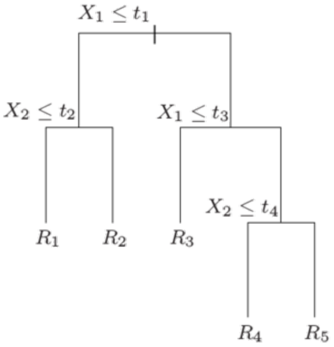
- and then fit a simple model (like a constant) in each element of the partition.

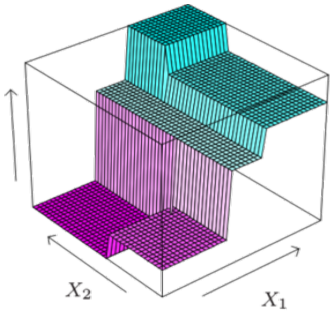
Using training data, the goal is to tune branches to choices that lead to good predictions in new scenarios,

- thereby estimating a **tree model**.

Tree constant model

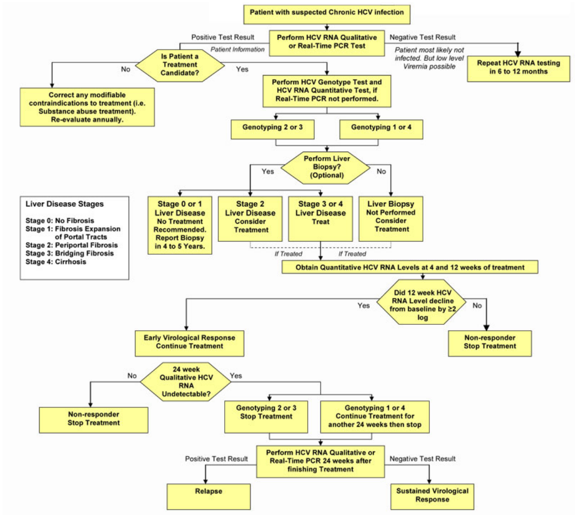
As an example, consider 2d inputs.





- There are binary splitting rules at the **internal nodes**.
- And decision (or prediction) rules at the **leaf nodes**.

Example: HCV treatment flow chart


Patient with suspected Chronic HCV infection
Positive Test Result
Perform HCV RNA Qualitative or Real-Time PCR Test
Negative Test Result
Patient Information
Patient most likely not infected. But low level Viremia possible
Repeat HCV RNA testing in 6 to 12 months
Is Patient a Treatment Candidate?
No
Yes
Correct any modifiable contraindications to treatment (i.e. Substance abuse treatment). Re-evaluate annually.
Perform HCV Genotype Test and HCV RNA Quantitative Test, if Real-Time PCR not performed.
Genotyping 2 or 3
Genotyping 1 or 4
Perform Liver Biopsy? (Optional)
Yes
No
Liver Disease Stages
Stage 0: No Fibrosis
Stage 1: Fibrosis Expansion of Portal Tracts
Stage 2: Periportal Fibrosis
Stage 3: Bridging Fibrosis
Stage 4: Cirrhosis
Stage 0 or 1 Liver Disease No Treatment Recommended. Report Biopsy in 4 to 5 Years.
Stage 2 Liver Disease Consider Treatment
Stage 3 or 4 Liver Disease Treat
Liver Biopsy Not Performed Consider Treatment
If Treated
If Treated
Obtain Quantitative HCV RNA Levels at 4 and 12 weeks of treatment
Did 12 week HCV RNA Level decline from baseline by ≥2 log
Yes
No
Early Virological Response Continue Treatment
Non-responder Stop Treatment
24 week Qualitative HCV RNA Undetectable?
No
Yes
Non-responder Stop Treatment
Genotyping 2 or 3 Stop Treatment
Genotyping 1 or 4 Continue Treatment for another 24 weeks then stop
Positive Test Result
Perform HCV RNA Qualitative or Real-Time PCR 24 weeks after finishing Treatment
Negative Test Result
Relapse
Sustained Virological Response

(From http://hcv.org.nz/wordpress/?tag=treatment-flow-chart)

Binary trees

Tree-based based methods for predicting y from a feature vector x divide up the feature space into rectangles, and then fit a very simple model in each rectangle. This works both when y is discrete and continuous, i.e., both for classification and regression.

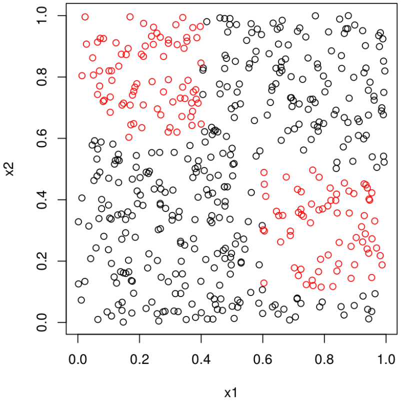
Rectangles can be achieved by making successive binary splits on the predictors variables $X_1, \ldots, X_p$. That is, we choose a variable X_j, $j = 1, \ldots, p$, divide up the feature space according to

$$X_j \leq c \qquad \text{and} \qquad X_j > c$$

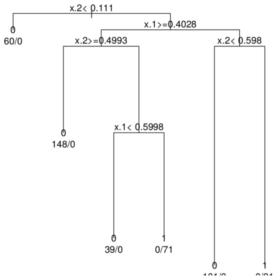
Then we proceed on each half

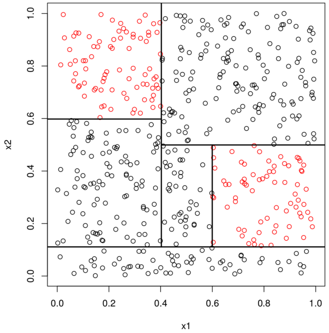
Example: simple classification tree

Example: $n = 500$ points in $p = 2$ dimensions, falling into classes 0 and 1, as marked by colors



Does dividing up the feature space into rectangles look like it would work here?


x.2< 0.111
0
60/0
x.1>=0.4028
x.2>=0.4993
x.2< 0.598
0
148/0
x.1< 0.5998
0
39/0
1
0/71
0
1


x2
x1
0.0
0.2
0.4
0.6
0.8
1.0

Prediction

Once learned, decision trees are like a game of mousetrap.

- You drop your x covariates in at the top,
- and each decision at an **internal node** bounces you either left or right.
- Landing in a **leaf node** with a data subset determined by the decisions above.

$$\begin{tikzcd}
 & & x_i = 0 \arrow[dl] \arrow[dr] & \\
 & x_j = 2 \arrow[dl] \arrow[dr] & & \{\mathbf{x} : x_i > 0\} \\
\{\mathbf{x} : x_i \leq 0, x_j \leq 2\} & & \{\mathbf{x} : x_i \leq 0, x_j > 2\} &
\end{tikzcd}$$

The **prediction rule** at each leaf is

- determined *only* by the training data that ended up in that leaf.

Classification trees

Classification trees are popular because they are interpretable, and maybe also because they mimic the way (some) decisions are made.

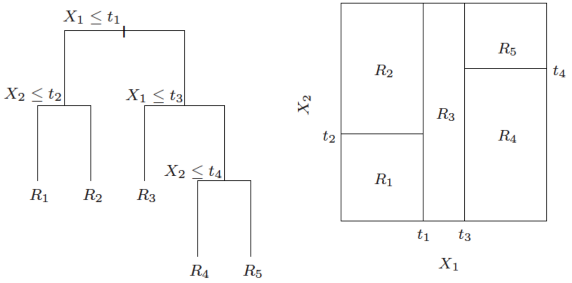
The classification tree can be thought of as defining m regions (rectangles) $R_1, \ldots, R_m$, each corresponding to a leaf of the tree.

We assign each R_j a class label $c_j \in \{1, \ldots, K\}$. We then classify a new point $x \in \mathbb{R}^p$ by

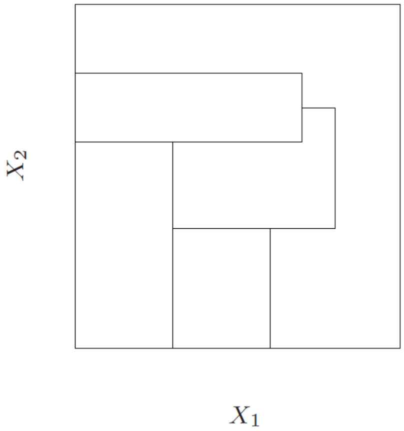
$$\hat{f}^{\text{tree}}(x) = \sum_{j=1}^{m} c_j \cdot 1\{x \in R_j\} = c_j \text{ such that } x \in R_j$$

Finding out which region a given point x belongs to is easy since the regions R_j are defined by a tree—we just scan down the tree. Otherwise, it would be a lot harder (need to look at each region).

Example: regions defined by a tree



Example: regions not defined by a tree



Predicted class probabilities

With classification trees, we can also get not only the predicted classes for new points but also the predicted class probabilities.

Note that each region R_j contains some subset of the training data (x_i, y_i), $i = 1, \dots, n$, say, n_j points. The predicted class c_j is just most common occuring class among these points. Further, for each class $k = 1, \dots, K$, we can estimate the probability that the class label is k given that the feature vector lies in region R_j, $P(Y = k \mid X \in R_j)$, by

$$\hat{p}_k(R_j) = \frac{1}{n_j} \sum_{x_i \in R_j} 1\{y_i = k\}$$

the proportion of points in the region that are of class k. We can now express the predicted class as

$$c_j = \arg \max_{k=1,\dots,K} \hat{p}_k(R_j).$$

How to build trees?

As usual, we'll maximize the data likelihood (minimize the deviance).

- But what are the observation probabilities in a tree model?
- That depends on the data type: is it a classification or regression?

Classification trees have class probabilities at the leaves:

- the chance of heavy rain is 0.9 (so take an umbrella).

Regression trees have a mean response at the leaves:

- the expected amount of rain is 2" (so take an umbrella).

Once the deviance is pinned down, everything else about inference is agnostic to the leaf model,

- leading to the so-called **Classification and regression trees (CART)** framework.

How to build trees?

For example, in regression, we want to find boxes $R_1, \ldots, R_J$ that minimize the RSS

$$\min \sum_{j=1}^{J} \sum_{i \in R_j} \left(y_i - \hat{y}_{R_j}\right)^2$$

where $\hat{y}_{R_j}$ is the mean response for the training observations within the jth box.

How to build trees?

There are two main issues to consider in building a tree:

1. How to choose the splits?
2. How big to grow the tree?

Think first about varying the depth of the tree . . . which is more complex, a big tree or a small tree? What tradeoff is at play here? How might we eventually consider choosing the depth?

Now for a fixed depth, consider choosing the splits. If the tree has depth d (and is balanced), then it has $\approx 2^d$ nodes. At each node we could choose any of p the variables for the split—this means that the number of possibilities is

$$p \cdot 2^d$$

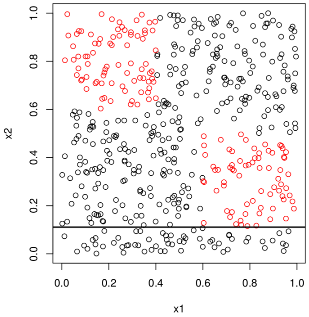
This is huge even for moderate d! And we haven't even counted the actual split points themselves.

The CART algorithm

The CART algorithm[1] chooses the splits in a top down fashion: then chooses the first variable to at the root, then the variables at the second level, etc.

At each stage we choose the split to achieve the biggest drop in misclassification error—this is called a greedy strategy. In terms of tree depth, the strategy is to grow a large tree and then prune at the end

Why grow a large tree and prune, instead of just stopping at some point? Because any stopping rule may be short-sighted, in that a split may look bad but it may lead to a good split below it



[1]Breiman et al. (1984), "Classification and Regression Trees"

Deviance

Regression deviance: $\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$,

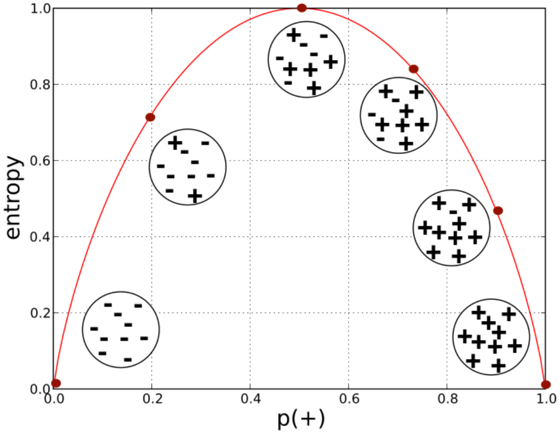
- where $\hat{y}_i = \bar{y}_{\ell(x_i)}$ where $\ell(x_i)$ is the leaf node x_i belongs to.

Classification deviance: $-\sum_i \log(\hat{p}_{y_i})$,

- where $\hat{p}_{y_i} = \frac{1}{n_{\ell(x_i)}} \sum_{y_k \in \ell(x_i)} \mathbb{I}(y_k = y_i)$.
- The **gini deviance** is also common: $-\sum_{i=1}^{n} \hat{p}_{y_i}(1 - \hat{p}_{y_i})$.

That is, instead of being based on $x^\top \hat{\beta}$,

- predicted $\hat{p}$ and $\hat{y}$ are functions of x passed through the decision nodes.
- And the training processes tries to find the best such rules *in-sample*.



entropy
p(+)
0.0
0.2
0.4
0.6
0.8
1.0

Recursive and greedy

Since each internal node split leads to different leaf nodes,

- there are a combinatorially huge number of possible tree configurations whose deviances need to be compared.

Like with `step` we're going to have to take shortcuts, by being greedy in **two ways**.
The first is through **recursion**:

- Given any set (node) of data, find the **optimal split**, and divide into two child sets.
- Repeat on each child, turning it into a parent.

Stop when

- the size of the leaf nodes hits some minimum threshold
  - (e.g., no less than 10 observations),
- or by deploying a minimum deviance threshold.

Optimal split

The second way is through the choice of **optimal split** at each child. Given a set of data $\{x_i, y_i\}_{i=1}^{n_\ell}$ at some leaf node $\ell \in \mathcal{L}$, we turn that leaf into an internal node by choosing a split

- based on *one* of the input coordinates of a particular x_i, i.e., x_{ij}

so that the child sets (new leaves)

$$\text{left: } \{x_k, y_k : x_{kj} \leq x_{ij}\} \quad \text{and} \quad \text{right: } \{x_k, y_k : x_{kj} > x_{ij}\}$$

are as homogeneous in response y as possible.
For example, in *regression trees* we make this choice by minimizing SSE:

$$\sum_{k \in \mathrm{left}} (y_k - \bar{y}_{\mathrm{left}})^2 + \sum_{k \in \mathrm{right}} (y_k - \bar{y}_{\mathrm{right}})^2.$$

- Choose one of the other deviances for *classification trees*.

Optimal split — classification

In a region R_m, the proportion of points in class k is

$$\hat{p}_k(R_m) = \frac{1}{n_m} \sum_{x_i \in R_m} 1\{y_i = k\}.$$

The CART algorithm begins by considering splitting on variable j and split point s, and defines the regions

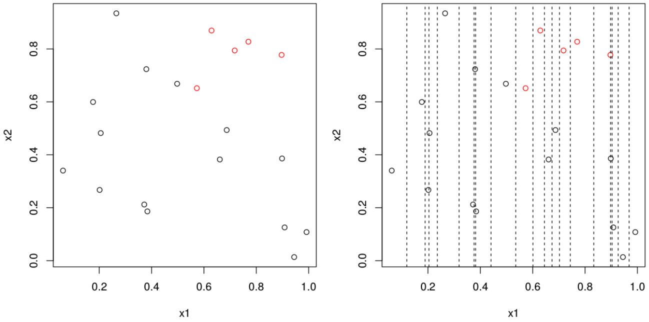
$$R_1 = \{X \in \mathbb{R}^p \mid X_j \leq s\}, \quad R_2 = \{X \in \mathbb{R}^p \mid X_j > s\}.$$

We then greedily chooses j, s by minimizing the misclassification error

$$\arg\min_{j,s} \left([1 - \hat{p}_{c_1}(R_1)] + [1 - \hat{p}_{c_2}(R_2)] \right).$$

Here $c_1 = \arg\max_{k=1,\ldots,K} \hat{p}_k(R_1)$ is the most common class in R_1, and $c_2 = \arg\max_{k=1,\ldots,K} \hat{p}_k(R_2)$ is the most common class in R_2.

How do we find the best split s? Aren't there infinitely many to consider? No, to split a region R_m on a variable j, we really only need to consider n_m splits (or $n_m - 1$ splits)



tree library

There are a couple of CART-like packages in R, including `tree` and `rpart`.

- We'll first showcase `tree`.

```
library(tree)
```

The syntax is essentially the same as for `glm`:
`mytree <- tree(y ~ x1 + x2 + x3 + ..., data=mydata)`
There are a few other possible arguments, all of which dictate criteria for new children.

- `mincut = 5` is the (default) minimum size for a new child.
- `mindev = 0.1` is the (default) minimum (proportion) deviance improvement for proceeding with a new split.

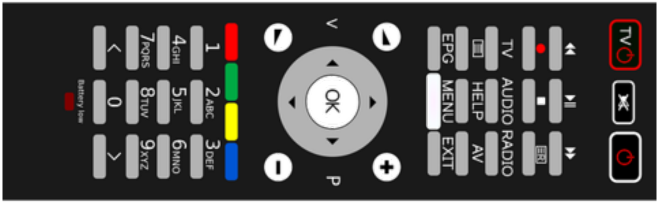
As usual, you can do `summary`, `plot`, `predict`, etc.

NBC data

Consider data from NBC on response to TV pilots.

- 6421 views and 20 questions for 40 shows.

```r
nbc <- read.csv("nbc.csv")
```



- NBC wants to know who watches what, where and when.

Responses

Several potential "response" variables.

- `Genre`: which sorts of people like which types of shows?
- ratings (`GRP`: gross ratings points): an estimate of total viewership
- projected engagement (`PE`): after watching a show, viewers are quizzed on order and detail. This measures their engagement with the show (and ads!).

Lets extract these three potential responses as *y*-values.

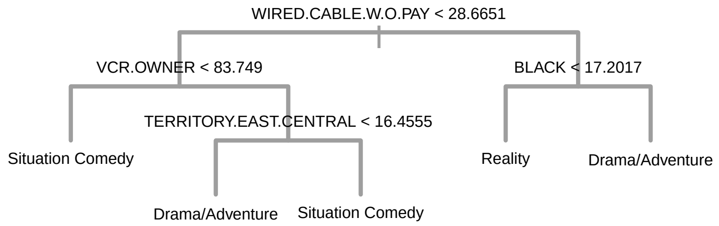
```
genre <- as.factor(nbc$Genre)
GRP <- nbc$GRP
PE <- nbc$PE
```

And then remove them from the `data.frame` along with `Show`, `Network` and `Duration`.

```
Show <- nbc[,1]
nbc <- nbc[,-c(1,58:62)]
```

A classification tree

```
genretree <- tree(genre ~ ., data=cbind(nbc, genre), mincut=1)
plot(genretree, col=8, type="uniform", lwd=2); text(genretree, cex=0.5)
```



Prediction

To get the most probable class label, specify `type="class"`.

```
gp <- predict(genretree, newdata=nbc, type="class")
```

- So these are the "fitted values".

```
disagree <- genre != gp
rbind(show=as.character(Show[disagree]),
    data=as.character(genre[disagree]), pred=as.character(gp[disagree]))
```

```
##      [,1]
## show "America's Next Top Model"
## data "Reality"
## pred "Situation Comedy"
##      [,2]
## show "Cops"
## data "Reality"
## pred "Situation Comedy"
```

- Apparently these reality shows are more like situation comedies.

A regression tree

Consider predicting engagement (`PE`) from ratings (`GRP`) and `genre`, using custom dummies for the latter.

```
x <- model.matrix(PE ~ genre + GRP)[,-1]
x <- as.data.frame(x)
names(x) <- c("reality", "comedy", "GRP")
```

- Making `"drama"` the reference factor level.

Now we're ready to go.

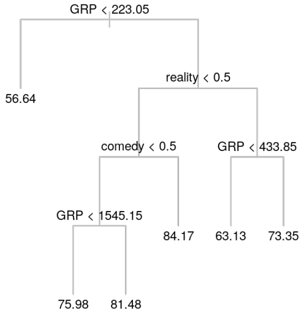
```
nbctree <- tree(PE ~ ., data=x, mincut=1)
```

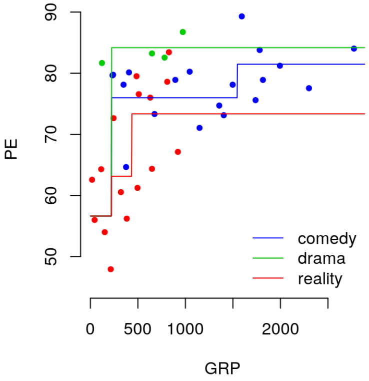
Predict under each `genre` dummy variable for the plot on the next slide.

```
newgrp <- seq(1,3000,length=1000)
pd <- predict(nbctree, newdata=data.frame(GRP=newgrp, drama=1, comedy=0, reality=0))
pc <- predict(nbctree, newdata=data.frame(GRP=newgrp, drama=0, comedy=1, reality=0))
pr <- predict(nbctree, newdata=data.frame(GRP=newgrp, drama=0, comedy=0, reality=1))
```

Nonlinear fit: PE increases with GRP, but in jumps.

```
par(mfrow=c(1,2)); plot(nbctree, type="uniform", col=8); text(nbctree, cex=0.75)
plot(PE ~ GRP, data=nbc, bty="n", col=c(4,2,3)[genre], pch=20, ylim=c(45,90))
lines(newgrp, pd, col=4); lines(newgrp, pc, col=3); lines(newgrp, pr, col=2)
legend("bottomright", c("comedy", "drama", "reality"), col=4:2, bty="n", lty=1)
```





Automatic interaction detection (AID)

We can see from the estimated tree on the previous slide that

- different `genres` are more/less dependent on `GRP` $\Rightarrow$ an interaction!

AID was an original motivation for building decision trees.

- The term goes back to a 1963 paper by Morgan and Sonquist which may be the first paper on trees.
- Older algorithms have it in their name: CHAID, US-AID, ...

This is pretty powerful technology:

- nonlinearity and interaction without having to specify it in advance;
- nonconstant variance (different variance in each leaf)
  - That doesn't mean that $\log y$ would hurt;
  - but what about $\log x$?

Nonparametric

Methods with these characteristics are called **nonparametric**

- because the number of degrees of freedom is roughly equal to the data size.

Now that's a double-edged sword,

- which is why `mincut` and `mindev` are so important.
- Just like with more predictors in least squares, deep trees are content to fit the noise.

The biggest challenge with such flexible models is avoiding overfit.

- It helps to *regularize* trees (and other nonparametric learners) just as with parametric (e.g., least squares) models.
- But how do we measure the complexity of a tree?

Pruning

- How about tree depth?

For CART, the solution to fit/complexity trade-offs is cross validation (CV).

- The basic/default constraints (`mincut`, `mindev`) lead to a "full" free fit.
- Prune this tree by removing split rules from the bottom up.

At each step,

- remove the split that contributes least to deviance reduction, thus reversing CART's growth process.
- Each prune step produces a candidate tree model, and we can compare their out-of-sample prediction performance via CV.

Prostate

Lets look at the prostate data.

```
library(ElemStatLearn)
data(prostate)
```

After tumor detection, there are many treatment options.

- Various chemo + radiation, surgical removal.

Biopsy information is available to help in deciding treatment

- `Gleason Score`: microscopic pattern classes.
- `Prostate Specific Antigen`: protein production.
- `Capsular Penetration`: reach of cancer into gland lining.
- `Benign Prostatic Hyperplasia Amount`: size of prostate.
- `Patient's age`: another influential variable.

The goal is to predict tumor log-volume (size, spread).

Now, fit a deep tree with the same setup as before.

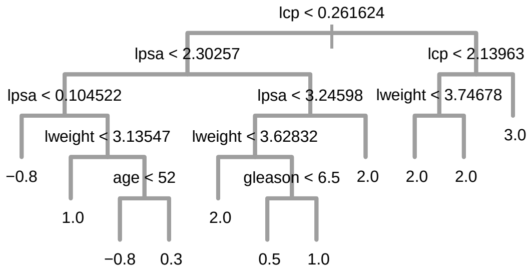
```
pstree <- tree(lcavol ~., data=prostate, mincut=1)
```

In the plot on the next slide

- the leaf node labels for this *regression tree* are expected log tumor volume.

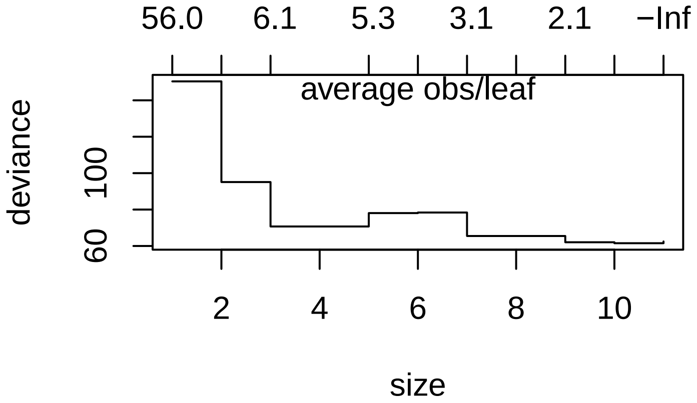
Do we need all these splits? Is the tree fitting signal or noise?

```
plot(pstree, col=8, type="uniform", lwd=2); text(pstree, cex=0.5)
```



CV tree pruning

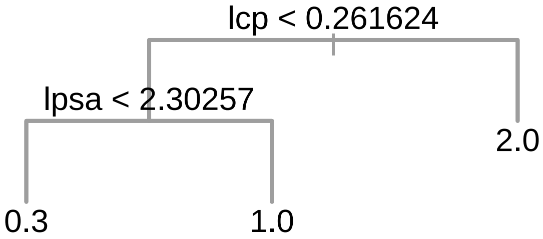
```
cvpst <- cv.tree(pstree, K=90)
plot(cvpst); text(6,145, "average obs/leaf")
```



Pruned fit

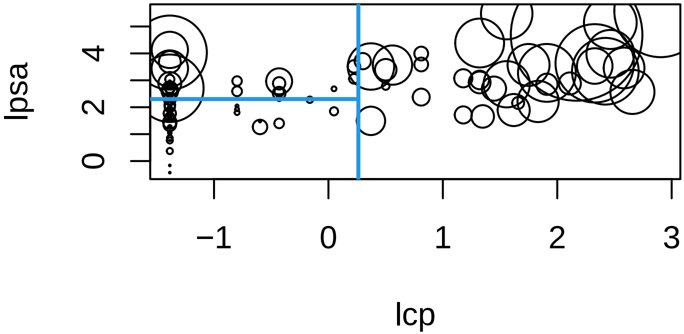
Since `size = 3` has the (first) lowest CV deviance, it is “best”.

```
pstcut <- prune.tree(pstree, best=3)
plot(pstcut, col=8, type="uniform", lwd=2); text(pstcut)
```



With only two relevant inputs, visualization is rather straightforward.

```
plot(prostate[,c("lcp","lpsa")], cex=exp(prostate$lca)*.2)
abline(v=.261624, col=4, lwd=2)
lines(x=c(-2,.261624), y=c(2.30257,2.30267), col=4, lwd=2)
```



More on prunning

For any tree T, let $|T|$ denote its number of leaves. We define

$$C_\alpha(T) = \sum_{j=1}^{|T|} (1 - \hat{p}_j(R_j)) + \alpha \cdot |T|$$

We seek the tree $T \subseteq T_0$ that minimizes $C_\alpha(T)$.

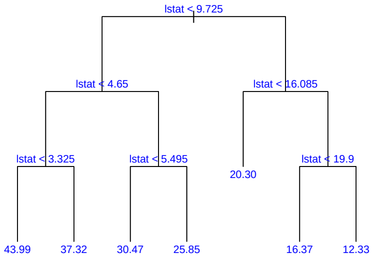
It turns out that this can be done efficiently using dynamic programming.

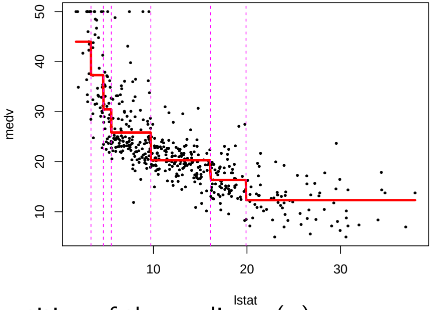
Note that α is a tuning parameter, and a larger α yields a smaller tree. CART picks α by 5- or 10-fold cross-validation.

- example in `rpart`

Example: Boston housing

At left is the tree fit to the data. At each interior node there is a decision rule of the form $\{x < c\}$. If $x < c$ you go left, otherwise you go right. Each observation is sent down the tree until it hits a bottom node or leaf of the tree.

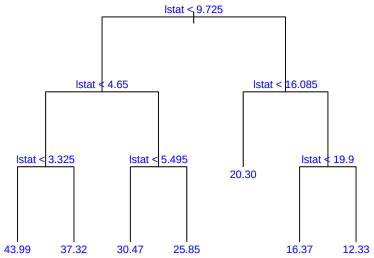


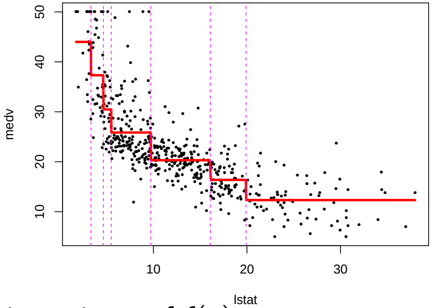


The set of bottom nodes gives us a partition of the predictor (x) space into disjoint regions. At right, the vertical lines display the partition. With just one x, this is just a set of intervals.

Example: Boston housing

Within each region (interval) we compute the average of the y values for the subset of training data in the region. This gives us the step function which is our $\hat{f}$. The $\bar{y}$ values are also printed at the bottom nodes (left plot).



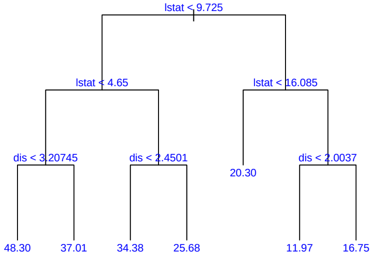


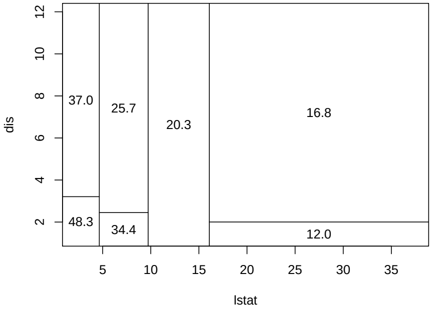
To predict, we just use our step function estimate of $f(x)$.

Equivalently, we drop x down the tree until it lands in a leaf and then predict the average of the y values for the training observations in the same leaf.

Example: Boston housing — two explanatory variables

Here is a tree with $x = (x_1, x_2) = $ (lstat,dis) and y=medv.





At right is the *partition* of the x space corresponding to the set of bottom nodes (leaves).

The average y for training observations assigned to a region is printed in each region and at the bottom nodes.

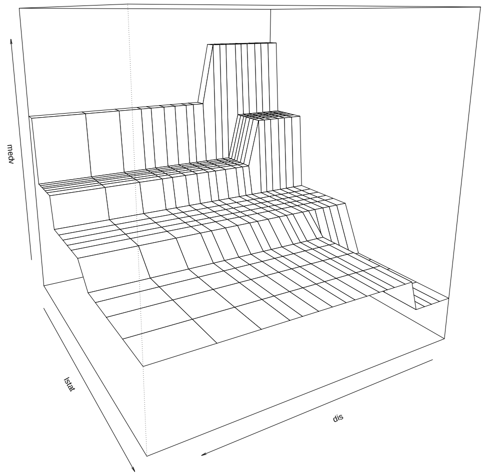
This is the regression function given by the tree.

It is a step function which can seem dumb, but it delivers non-linearity *and* interactions in a simple way and works with a lot of variables.

Notice the interaction.

The effect of `dis`

depends on `lstat`!!



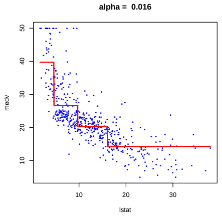
Choosing α

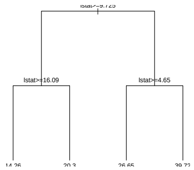
At right are three different tree fits we get from three different α values (using all the data).

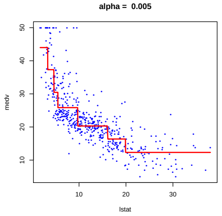
The smaller α is, the lower the penalty for complexity is, the bigger tree you get.

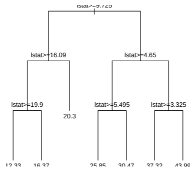
The top tree is a sub-tree of the middle tree, and the middle tree is a sub-tree of the bottom tree.

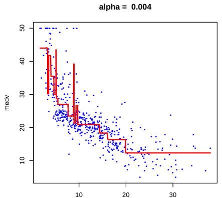
The middle α is the one suggested by CV.

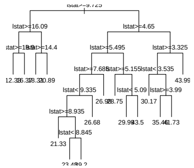






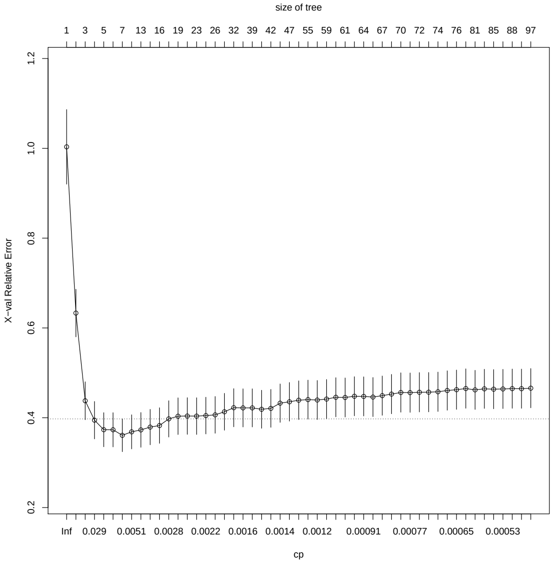






This is the CV plot giving by the R package `rpart` for `y=medv` `x=lstat`.

The error is relative to the error obtained with a single node (fit is $y = \bar{y}$, $\alpha = \infty$).



Tree problems?

Although they have much to recommend them:

- automatic learning of non-linear response functions with interactions and interpretability,

there are certainly some drawbacks.

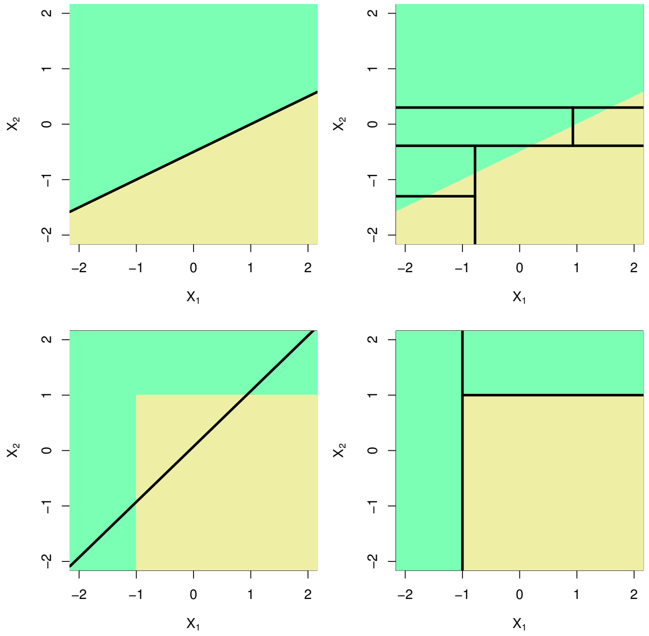
- Easy to over-fit, even with CV: deep tree structure creates stability problems, making optimal depth hard to assess.
- No theory to fall back on.
- When regression response surfaces are smooth, the piece-wise constant nature of fits actually strains interpretability as many splits are required to mimic linear and low degree polynomial relationships.

How well do trees predict?

Unfortunately, the answer is not great. Of course, at a high level, the prediction error is governed by bias and variance, which in turn have some relationship with the size of the tree (number of nodes). A larger size means smaller bias and higher variance, and a smaller tree means larger bias and smaller variance.

But trees generally suffer from high variance because they are quite instable: a smaller change in the observed data can lead to a dramatically different sequence of splits, and hence a different prediction. This instability comes from their hierarchical nature; once a split is made, it is permanent and can never be "unmade" further down in the tree.

We'll learn some variations of trees have much better predictive abilities. However, their predictions rules aren't as transparent.


X₁
X₂
−2
−1
0
1
2

Summary: Trees

Pros:

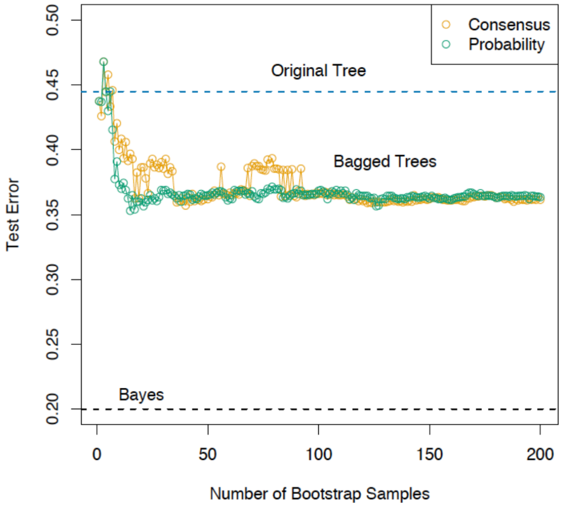
- easy to explain
- closely mirror human decision-making
- can handle both discrete and continuous variables
- fast

Cons:

- not competitive in terms of accuracy
- non-robust — small changes in data can cause a large change in the estimated tree

Combining trees

Fitting small trees on bootstrapped data sets, and averaging predictions at the end, can greatly reduce prediction error.


Consensus
Probability
Original Tree
Bagged Trees
Bayes
Test Error
Number of Bootstrap Samples