ELEMENTS OF DATA SCIENCE AND STATISTICAL LEARNING

SPRING 2020

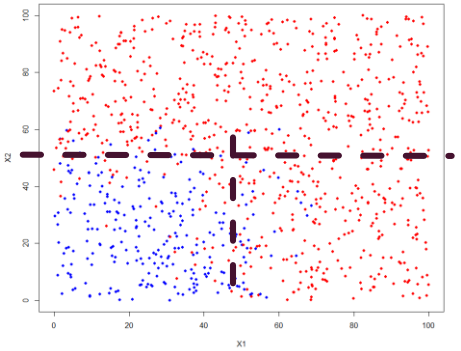
Week 11

OUTLINE

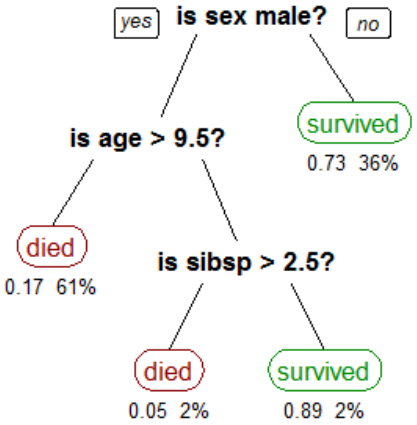
- Trees and forests!
- Introducing regression/classification trees: toy example
- Tree pruning
- Real dataset: comparison between classification tree and lda/LR
- Bagging and random forests

DECISION TREE

- A technique for building both regression AND classification models.
- In some sense similar to KNN: in order to make a prediction for any new value $\boldsymbol{x} = (x_1, x_2, \ldots, x_p)$, we want to use a region *M* from the training set such that $\boldsymbol{x} \in M$
  - KNN: find K "closest" training points (just *assuming* local homogeneity), take mean for regression or majority vote for classification
  - Decision tree: The regions are *precomputed* from the training set. Use a region M the point of interest falls into, prediction is mean{ $\mathbf{x}_k \in$ M } (regression) or majority_vote { $\mathbf{x}_k \in$ M } (classification).
- How to compute the regions? We try to cut out *homogeneous* regions. See the examples on the right: decision tree is a hierarchy of *cuts* in the predictor space. A *non-parametric* approach!
- Things to consider (vs KNN):
  - Cuts can be easily performed on either categorical or continuous variables
  - Order of cuts in each branch is arbitrary: can choose whichever one results in the "best" fit (will discuss)
  - The size/shape of the subspace bounded by conditions (hyperplanes) is highly adaptive
  - The points in the relevant leaf in the decision tree are not necessarily all the "nearest" ones to the observation we predict for, and/or there might be no good notion of "distance" at all
  - If there is a good notion of distance, especially if the latter is a highly specialized one (cf clustering!), KNN might be still a good choice!
  - Trees can be highly interpretable! But there are caveats…



Toy dataset (following slides)

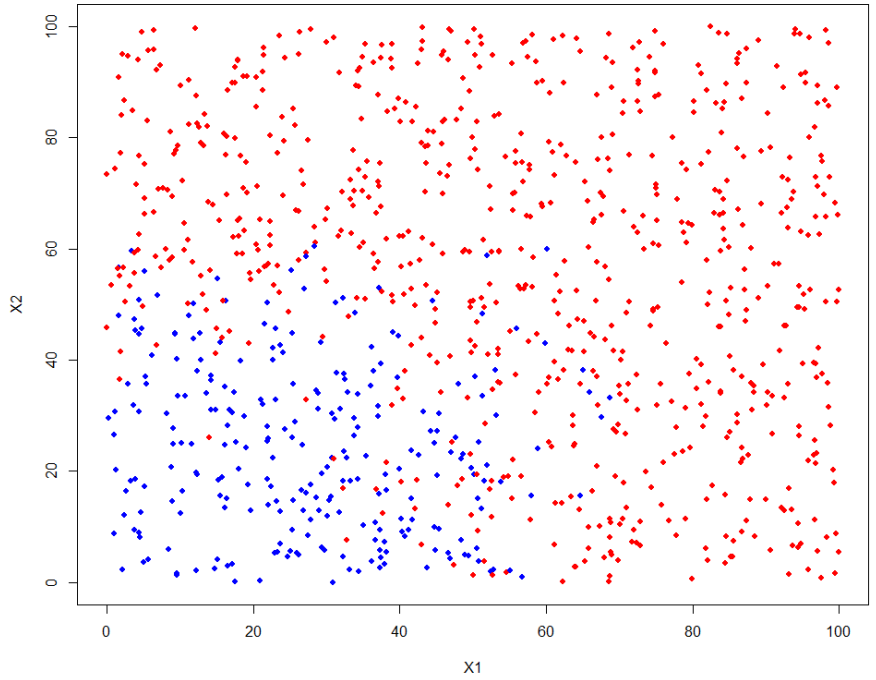


Survival of passengers on Titanic (credit :Wiki)

A TOY DATASET

- Let's first make a simple toy dataset to gain some intuition

```
n=1000
df.1=data.frame(x1=runif(n,0,100), x2=runif(n,0,100))
df.1$y=ifelse(df.1$x2+rnorm(n,sd=10)>50,1,
                    ifelse(df.1$x1+rnorm(n,sd=10)>50,1,0))
plot(df.1$x1,df.1$x2,col=ifelse(df.1$y==1,"red","blue"),
     pch=19,cex=0.8,xlab="X1",ylab="X2")
```

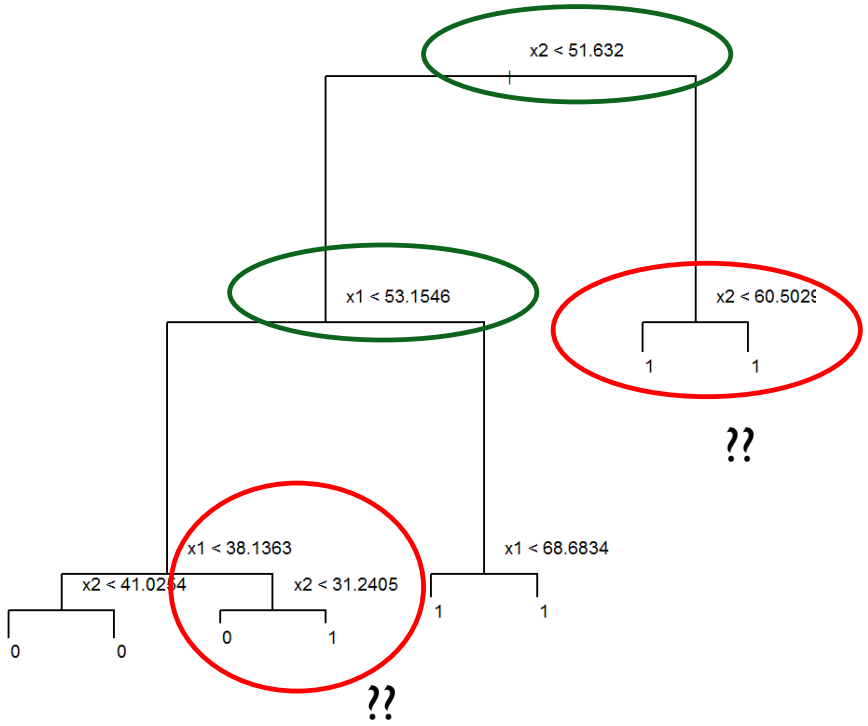


- Do you think the classes are reasonably separable (i.e. there is a hope to train a classifier)?
- Are the data *perfectly* separable (can we train a classifier that always predicts correct class?)
- What is the "decision boundary" here?
- Take a note of how we generated the dataset: if X2> 50: red; otherwise { if X1>50: red ; otherwise blue }

BUILDING A DECISION TREE

- As usual: there is a library!

```
library(tree)
t.1 = tree(as.factor(y) ~ . , data=df.1)
plot(t.1,lwd=2)
text(t.1,pretty=0,cex=0.7 ,adj=c(-0.2,0))
summary(t.1)

Classification tree:
tree(formula = as.factor(y) ~ ., data = df.1)
Number of terminal nodes:  8
Residual mean deviance:  0.3826 = 379.6 / 992
Misclassification error rate: 0.079 = 79 / 1000
```



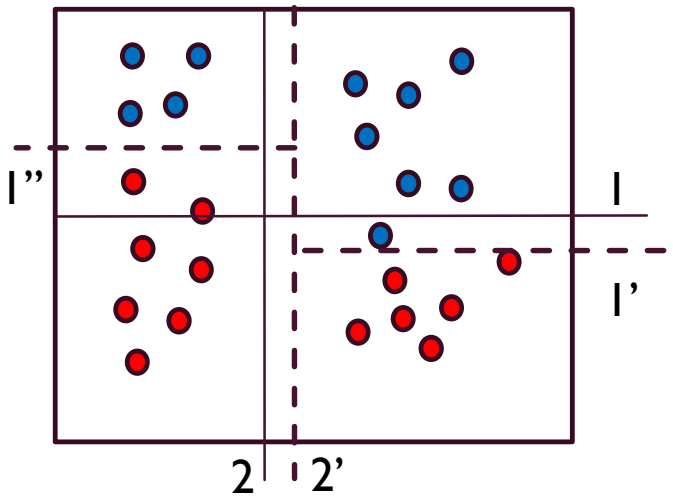
- Note that if you want to fit a classification tree using this library, the outcome *must be* a factor. Otherwise the function will automatically build a regression tree. Here we chose to convert explicitly inside the function call; we could also make the dataframe column a factor from the outset, of course

TREE CONSTRUCTION: THE OPTIMIZATION GOAL

- In each region R_j (a box in the feature space), the predicted value is the mean/majority vote (in regression/classification problems, respectively) – there is nothing else we can do!
- As always, we want to minimize the error :
  - In regression problems: training error is $RSS = \sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2$ where j=1…J enumerates the regions
  - In classification problems we could try using the classification error directly: if we have classes k=1…K, the misclassification rate in each region *j* is $E_j = 1 - \max_k(\hat{p}_{jk})$ where $\hat{p}_{jk}$ is the proportion of the training observations of class *k* in the region *j* (we have agreed to use majority voting rule, so the largest $\hat{p}_{jk}$ is what we predict correctly, and members of all other classes that fell into region *j* *are* misclassified!)
  - Classification error turns out to be a poor metric: not sensitive enough. Alternatives are:
  - Gini index (total variance across the K classes): $G_j = \sum_{k=1}^{K} \hat{p}_{jk} (1 - \hat{p}_{jk})$
  - Entropy: $D_j = -\sum_{k=1}^{K} \hat{p}_{jk} \log \hat{p}_{jk}$
  - It is easy to see that both Gini index and entropy are ***minimal*** when **all p_k=0**, except one (which is equal to 1 since fractions/probabilities should sum to 1 !). In other words, minimum Gini index/entropy corresponds to *pure nodes* (single class)
  - The implementation of `tree()` in package `tree` can use Gini index (set `split="gini"`) or (by default) an error measure closely related to entropy, the *deviance* (essentially weights the entropies D_j by the numbers of observations n_j: $D_j = -2 \sum_{k=1}^{K} n_{jk} \log \hat{p}_{jk}$

TREE CONSTRUCTION: THE ALGORITHM

- Theoretically: we could consider all possible trees (i.e. cuts on each variable in all orders). Obviously impractical.
- Greedy algorithm (not guaranteed to be optimal!!):
  - Select such variable X_m and the cutpoint s in the domain of X_m that splitting the space into the regions $\{X: X_m < s\}$ and $\{X: X_m > s\}$ results in the greatest possible reduction in RSS (regression) or Gini/deviance (classification)
  - Repeat the procedure in each of the two branches (i.e. each of the obtained sub-regions) recursively
  - Stop when no region contains more than *N* observations and/or when certain purity of the node is reached
- Issue 1: the algorithm, while efficient, never looks back. It is possible that we could ultimately get a greater reduction in total RSS if we first made a cut that results in a relatively small reduction, followed by cuts in each branch that result in large reduction. See the diagram: greedy splitting will likely make cut 1 (*almost* perfect!), then (maybe!) 2 (can you think of why?). Instead, it would be better to first make cut 2' followed by cuts 1' and 1" (dashed lines)
- Issue 2: the algorithm is likely to overfit!
- Can you think of why we don't do splits like X_1-X_2>s? That might give much more flexible decision boundary!



PREDICTION ERROR

- We always have to assess prediction error on a *test set* not used for training and/or perform cross-validation!
- In our toy scenario we know exactly the generative model for the data (we came up with it!) so we can simply generate a new test set
  - Otherwise we would need to run cross-validation exactly the same way we did before – LOO, K-fold, bootstrap, etc

```
n=1000
df.2=data.frame(x1=runif(n,0,100), x2=runif(n,0,100))
df.2$y=ifelse(df.2$x2+rnorm(n,sd=10)>50,1,
                  ifelse(df.2$x1+rnorm(n,sd=10)>50,1,0))
pred.df1=as.numeric(as.vector(predict(t.1,type="class")))
pred.df2=as.numeric(as.vector(predict(t.1,newdata=df.2,type="class")))
```

```
assess.prediction(df.1$y,pred.df1)
Total cases that are not NA: 1000
Correct predictions (accuracy): 921(92.1%)
TPR (sensitivity)=TP/P: 93.7%
TNR (specificity)=TN/N: 87.6%
PPV (precision)=TP/(TP+FP): 95.6%
FDR (false discovery)=1-PPV: 4.41%
FPR =FP/N=1-TNR: 12.4%
```

```
assess.prediction(df.2$y,pred.df2)
Total cases that are not NA: 1000
Correct predictions (accuracy): 903(90.3%)
TPR (sensitivity)=TP/P: 93.8%
TNR (specificity)=TN/N: 78.8%
PPV (precision)=TP/(TP+FP): 93.5%
FDR (false discovery)=1-PPV: 6.52%
FPR =FP/N=1-TNR: 21.2%
```

TREE PRUNING

- To avoid overfitting we need to keep the number of cuts (and, correspondingly, of the nodes) in check:
  - A smaller tree might provide worse fit to the training data but also avoid overfitting (larger bias/smaller variance)
- A good way of doing this is to build a large tree first and then *prune* it.
  - Theoretically, we can start from a large tree and try removing splits/nodes while evaluating each such attempt using cross-validation. Too costly in practice.
- A trade-of algorithm: cost complexity pruning
  - Build a large tree T_0
  - Consider a range of values of the tuning parameter α, and for each such value prune the tree to minimize

$RSS = \sum_{j=1}^{J(T)} \sum_{i \in R_j} (y_i - \hat{y}_{R_j})^2 + \alpha |T|$ where $|T| < |T_0|$ is the size of the pruned tree and $J(T) < J(T_0)$ is the number of regions

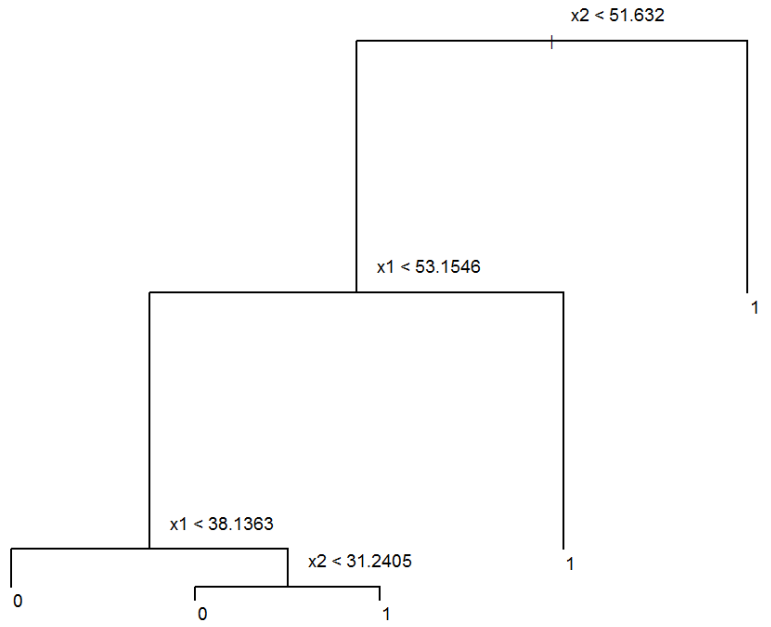
(note that since we are pruning the tree we already have at hand, rather than considering all possible trees for all considered values of α, this procedure is quite efficient)

  - Use cross-validation *just to pick the optimal* α.

TREE PRUNING IN PRACTICE:

- Use the pre-computed tree (we will be using our toy tree `t.1` for now)
- Fortunately, the function `cv.tree()` is made available to us! For cross-validation in classification problems, one can use the deviance (default) or classification error:

```
cv.t.1 = cv.tree(t.1 ,FUN=prune.misclass )
cv.t.1
$size
[1] 8 5 3 1
$dev
[1] 103 103 110 259
$k
[1] -Inf  0.0  8.5 81.5
$method
[1] "misclass"
attr(,"class")
[1] "prune"         "tree.sequence"
prune.t.1=prune.misclass(t.1,best=5)
plot(prune.t.1,lwd=2)
text(prune.t.1,pretty=0,cex=1.2,adj=c(-0.2,0))
```



OUTLINE

- Trees and forests!
- Introducing regression/classification trees: toy example
- Tree pruning
- Real dataset: comparison between classification tree and lda/LR
- Bagging and random forests

HANDWRITTEN DIGITS DATASET

- A classical testing ground for classification methods for image recognition
- Large collection of digitized hand-written digits (0…9)
  - Training set: 60,000 examples + additionally 10,000 strong test set
- The data in binary format (raw pixels) can be downloaded from http://yann.lecun.com/exdb/mnist/
- Data format (quote from the above website):

IMAGE FILE

[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000803(2051)	magic number
0004	32 bit integer	60000	number of images
0008	32 bit integer	28	number of rows
0012	32 bit integer	28	number of columns
0016	unsigned byte	??	pixel
0017	unsigned byte	??	pixel
.....			
xxxx	unsigned byte	??	pixel

Pixels are organized row-wise. Pixel values are 0 to 255. 0 means background (white), 255 means foreground (black).

TRAINING SET LABEL FILE (train-labels-idx1-ubyte):

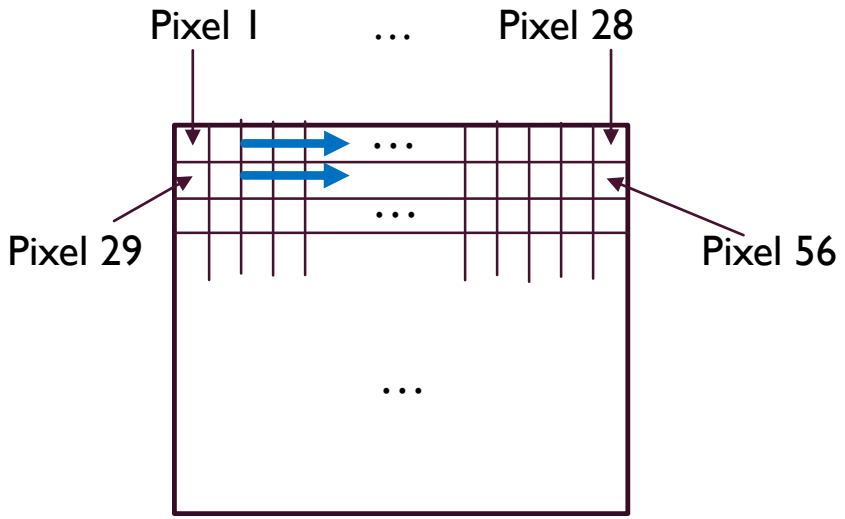
[offset]	[type]	[value]	[description]
0000	32 bit integer	0x00000801(2049)	magic number (MSB first)
0004	32 bit integer	60000	number of items
0008	unsigned byte	??	label
0009	unsigned byte	??	label
.....			
xxxx	unsigned byte	??	label

The labels values are 0 to 9.

READING THE IMAGE DATA

```
read.digit.images=function(file) {
  con=gzfile(file,"rb")
  magic=readBin(con,integer(),endian="big")
  if ( magic != 2051 ) { stop("Wrong magic number") }
  N = readBin(con,integer(),endian="big")
  nrow = readBin(con,integer(),endian="big")
  ncol = readBin(con,integer(),endian="big")
  cat(N," ",nrow,"x",ncol," images in the file\n",sep="")
  m = matrix(NA,nrow=N,ncol=nrow*ncol)
  for ( i in 1:N ) { # read N images
    pixels = readBin(con,integer(),n=nrow*ncol,size=1,signed=F)
    if ( length(pixels) < nrow*ncol) {
      cat("Premature end of file\n")
      break
    }
    m[i,]=pixels
    if ( i %% 10000 == 0 ) { cat(i,"\n") }
  }
  close(con)
  return(m)
}
```

- Use this code to read the images into a numeric matrix. Rows = observations (i.e. individual examples), columns = individual pixels (organized by image row)



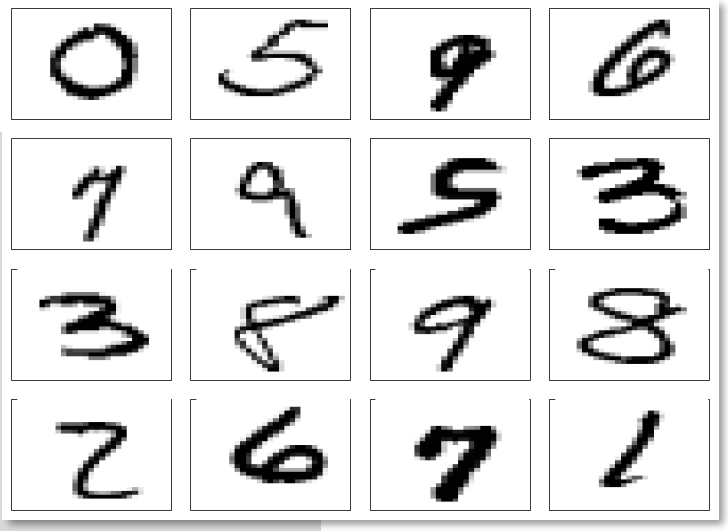
READING THE LABELS

- Use the following code to read the class labels (digits 0 through 9, stored as a single byte)

```
read.digit.labels=function(file) {
  con=gzfile(file,"rb")
  magic=readBin(con,integer(),endian="big")
  if ( magic != 2049 ) {
    stop("Wrong magic number")
  }
  N = readBin(con,integer(),endian="big")
  cat(N," labels in the file\n",sep="")
  labels = readBin(con,integer(),n=N,size=1,signed=F)
  if ( length(labels) < N) {
    cat("Premature end of file\n")
  }
  close(con)
  return(labels)
}
```

VISUALIZING THE IMAGES

- 28x28 pixelized images. Let us manually draw them by representing each pixel as a small rectangle on 28x28 grid
  - rect(x0,y0,x1,y1) – draws a rectangle (see the docs)



```
plot.digit=function(x,rows=28,columns=28) {
  plot(1:columns,1:rows,type='n',xaxt='n',yaxt='n')
  offset=1
  p = colorRampPalette(c("white","black"))(256)
  for ( i in rows:1 ) {
    rect(1:columns,rep(i,columns),2:(columns+1),rep(i+1,columns),
         border=NA,col=p[x[offset:(offset+columns-1)]+1])
    offset=offset+columns
  }
}
# READ DATA IN: ****
m=read.digit.images("train-images-idx3-ubyte.gz") # 60K images
l=read.digit.labels("train-labels-idx1-ubyte.gz")
m.test=read.digit.images("t10k-images-idx3-ubyte.gz") # test set: 10K images
l.test=read.digit.labels("t10k-labels-idx1-ubyte.gz")
# ****
oldpar=par(mfrow=c(4,4),mar=c(1,1,1,1))
for(i in 1:16 ) { plot.digit(m[sample(60000,1),]) }
par(oldpar)
```

FIRST LOOK AT THE DATA: LOGISTIC REGRESSION

- The whole dataset is too large for most home computers (but not a problem for a decent server)
  - Note that R shell (but not the underlying libraries!!) is not most efficient memory-wise
  - We will select a random subset of 5000 images, just for illustration (you can try pushing this limit up!)
- Let's start with logistic regression, both as a recitation and in order to establish some baseline

```
set.seed(1234)
sample.idx=sample(60000,5000)
m1=m[sample.idx,]; l1=l[sample.idx]
k=0  # let's choose a digit to look for
Lk=ifelse(l1==k,1,0) # two-level outcome we are going to fit for now ("digit is k vs it is not k")
digit.0.lr.fit=glm(D~.,data=data.frame(D=Lk,m=m1),family = binomial)
Lk.test=ifelse(l.test==k,1,0)
lr.pred.0=as.numeric(predict(digit.0.lr.fit,newdata = data.frame(m=m.test),type="response") > 0.5)
assess.prediction(Lk.test,lr.pred.0)
Total cases that are not NA: 10000
Correct predictions (accuracy): 9642(96.4%)
TPR (sensitivity)=TP/P: 89.7%
TNR (specificity)=TN/N: 97.2%
PPV (precision)=TP/(TP+FP): 77.4%
FDR (false discovery)=1-PPV: 22.6%
FPR =FP/N=1-TNR: 2.85%
```

glm() is going to complain (warnings).
We will partially fix it later.

MULTICLASS CLASSIFICATION

- But what about multiple classes present in the problem (0…9)? Logistic regression does not seem to be well fit for classifying digits?
  - We will follow a heuristic approach that works pretty well in practice:
  - For each of the class labels $L_1,\ldots,L_K$ build a two-level model that classifies L_i vs *any other*.
  - For prediction on observation ***x***, run all K models and select prediction from the model that gives the highest score (e.g. likelihood, log-odds, class probability etc). NOTE: this is ***not*** a truly probabilistic approach, the models are not orthogonal and the probabilities/scores are not normalized across different models!
- We may run into a memory issue, again (models in R typically hold all their training data, so 10 models = 10 x data). For the purpose of illustration we are going to save only predictions of our multiple models on the test set, not the models themselves. How would you solve this issue in practice if you would want to keep actual models for performing actual classification in the future?

```
# will keep predicted class probabilities for each test observation, for each of 10 models:
glms=matrix(NA,nrow=10,ncol=length(l.test))
for ( k in 0:9 ) {  # fit 10 models "digit=k vs digit is not k":
  Lk = ifelse(l1==k,1,0) # generate binary class label for model k
  digit.k.fit = glm(D~.,data=data.frame(D=Lk,m=m1),family = binomial) # fit LR
  glms[k+1,] = as.numeric(predict(digit.k.fit,newdata = data.frame(m=m.test),type="response"))
}
lr.pred = apply(glms,2,which.max)-1
```

MULTICLASS CLASSIFICATION: CONTINUED

- With multiclass classification based on LR we get 75% accuracy on the *test set*, across 10 class labels!

```
lr.pred[1:10]
[1] 7 0 1 0 4 1 4 5 6 7
l.test[1:10]
[1] 7 2 1 0 4 1 4 9 5 9
table(lr.pred,l.test)
       l.test
lr.pred    0    1    2    3    4    5    6    7    8    9
      0  882    3   57   41   22   37   34   39   92   98
      1    4 1053   46   34   20   18   19   37   33   16
      2    7    7  743   54   17   25   31   53   29   11
      3   16    9   45  749   16   70   29   51   56   34
      4    5    2   22   18  792   31   45   25   40   72
      5   29    6   13   55   19  618   66   30   65   33
      6   13    7   24    4   16   27  709    7   25   16
      7    8    3   31   26    7   23    1  725   15   63
      8   14   43   44   20   15   33   19   12  612   30
      9    2    2    7    9   58   10    5   49    7  636
sum(diag(table(lr.pred,l.test))) # multiclass LR gave us 75% accuracy
[1] 7519
```

WHAT ABOUT LDA?

- Can we get a decent fit with something as simple and straightforward as LDA? Remember: linear boundaries only (hyperplanes in multidimensional space); but our most naïve LR included only linear terms, so it also imposed linear boundary!
- LDA can fit multiple class labels generically, within the single, proper probabilistic model (of course, with usual assumptions)
- LDA really dislikes variables that don't change at all (cannot estimate parameters of their "normal" distribution P(X|Y)), let's ignore pixels that are nearly background in all training images (that would help partially stopping LR from complaining too)

```
v=apply(m1,2,var)
sum(v<100)
no.change=which(v<100)

library(MASS)
digit.lda=lda(D~.,data=data.frame(D=l1,
                        m=m1[,-no.change]))
lda.pred=predict(digit.lda,
  newdata =
    data.frame(m=m.test[,-no.change]))$class
```

```
table(lda.pred,l.test)
        l.test
lda.pred    0    1    2    3    4    5    6    7    8    9
       0  923    0   16    5    0    8   16    5    4   11
       1    0 1092   47    9   13    9   10   36   29    7
       2    2    3  769   26    8    9   11   13   10    4
       3    5    2   42  853    2   60    2   14   37   12
       4    2    1   18    2  868   15   20   22   19   70
       5   19    2    9   37    4  696   39    3   63    4
       6   18    5   41    4    6   14  843    0   14    1
       7    0    1   12   25    0   13    0  835    3   47
       8    9   28   69   37    6   52   17    3  777    8
       9    2    1    9   12   75   16    0   97   18  845
sum(diag(table(lda.pred,l.test)))
[1] 8501   ## !!!!!!!!!!!!!!!! 85%!!
```

USING A TREE FOR IMAGE CLASSIFICATION

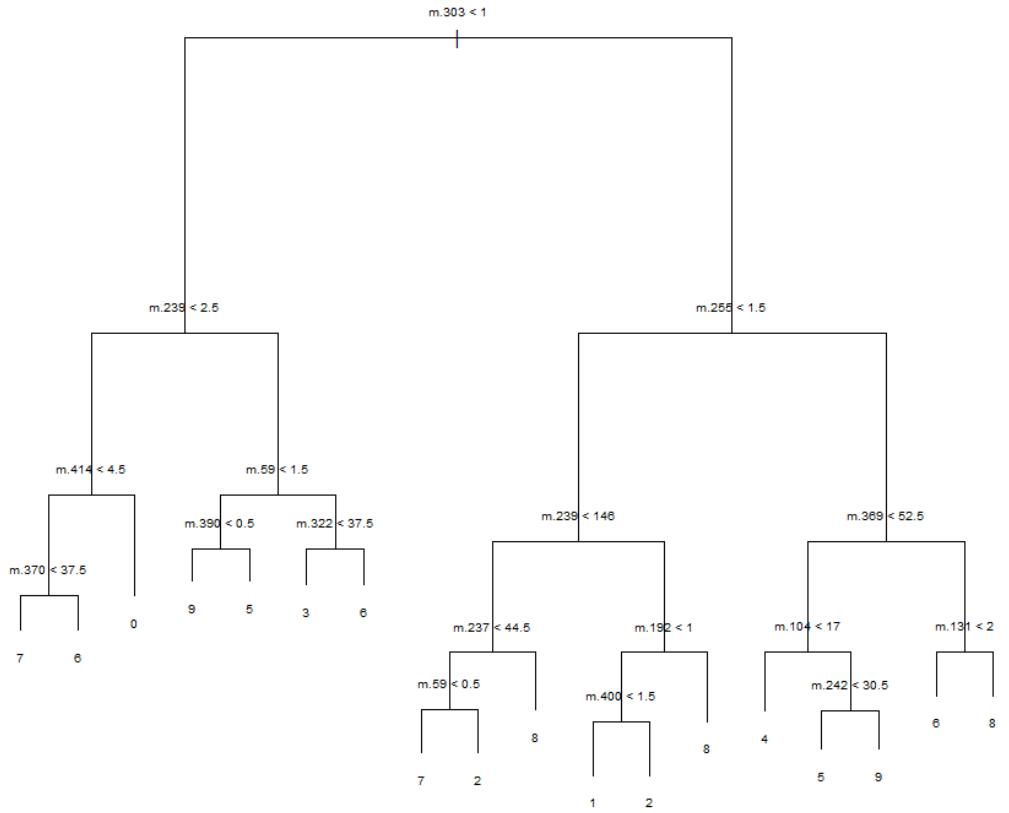
- It is time to grow a tree:

```
library(tree)
digit.tree=tree(D~.,data=data.frame(D=as.factor(l1),m=m1[,-no.change]))
tree.pred=predict(digit.tree,newdata=data.frame(m=m.test[,-no.change]),type="class")
table(tree.pred,l.test)
         l.test
tree.pred    0    1    2    3    4    5    6    7    8    9
        0  805    0   54   50    2   61   58   26    1    4
        1    0 1023   46   19    2    4    5   36   37    3
        2    0   18  563   91    2    4   21   12   25    9
        3    8   24   18  524    3  217    7    2   16   18
        4    0    0   20   12  510   48   41   10   17   30
        5   36    1   17   69   30  272   32    7   18   29
        6   83    0   85   47   52   49  626   15  102   25
        7   29   13   83   25   80   23   26  832    8   86
        8   17   50  133   64   58  125  103   20  683   63
        9    2    6   13  109  243   89   39   68   67  742
sum(diag(table(tree.pred,l.test)))
[1] 6580  ## ONLY 65% ! Worse than LR and MUCH worse than LDA ☹
```

HOW DEEP IS THE TREE?

- Let us examine the tree we just built
- Apparently, it splits only on a few individual pixels!
  - It's actually quite surprising it can fit those relatively complex images at all!
  - The tree would readily grow further if another split existed that would diminish the error. Instead, the data seems to be so intertwined that the algorithm just gives up (finds no further improvement)
  - Is there no hope?

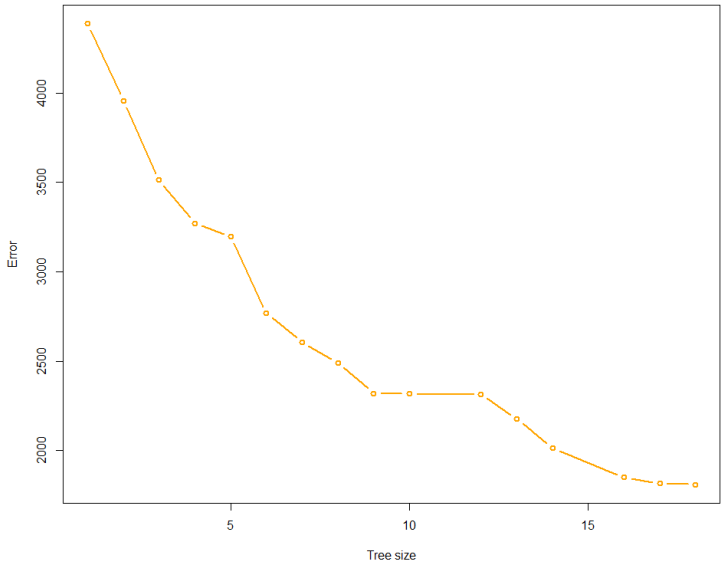
```
plot(digit.tree)
text(digit.tree,pretty = 0,cex=0.6)
```



PRUNING THE TREE

- While seems unlikely from the outset, let's check if our tree, as overly simplistic as it seems, overfits the training set – this could be a reason for relatively poor performance on the test set
  - The answer is no.

```
digit.tree.cv=cv.tree(digit.tree,FUN=prune.misclass)
digit.tree.cv
$size
 [1] 18 17 16 14 13 12 10  9  8  7  6  5  4  3  2  1
$dev
 [1] 1810 1815 1850 2013 2176 2314 2320 2320 2491 2604 2767 3196 ...
$k
 [1] -Inf   10   38   60   70   77   81   82  141  159  188  227  ...
$method
[1] "misclass"
attr(,"class")
[1] "prune"         "tree.sequence"
plot(digit.tree.cv$size,digit.tree.cv$dev,type='b',
    lwd=2,xlab="Tree size",ylab="Error",col="orange")
```



OUTLINE

- Trees and forests!
- Introducing regression/classification trees: toy example
- Tree pruning
- Real dataset: comparison between classification tree and lda/LR
- Bagging and random forests

BAGGING

- The general idea is: if we cannot fit a single model to the data well enough, can we build an *ensemble* of different models, each describing some aspect of the data, and then take the average prediction of all those models?
  - If each of the variables Z_i has a variance σ^2, the average of N such observations has variance σ^2/N
  - Can this work for model predictions? It might – prediction of a model *is* a random variable (since the model parameters are fitted on a randomly drawn realization of the data!)
- We can use bootstrap, but this time not for cross-validation per se, but for generating multiple datasets and thus "turning" different "sides" of data closer to us. We will be averaging models fitted on each of the bootstrapped realizations of the data. The prediction from the ensemble of N such models for observation *x* will be

$$\hat{f}_{avg}(x) = \frac{1}{N}\sum_{n=1}^{N}\hat{f}_n(x)$$

(for regression; in a classification problem – take the majority vote instead)

- Bootstrap AGGregation = BAGGING.

OUT OF BAG ERROR

- Bagging provides an additional bonus: since we are bootstrapping the original dataset, each bootstrapped dataset will likely miss some observations
- Take observation x_i and predict it's class label (or continuous outcome) using all the bootstrapped models that did *not* include that observation:

$$\hat{f}_{avg}(x_i) = \frac{1}{N^*} \sum_{n \in B^*} \hat{f}_n(x_i)$$

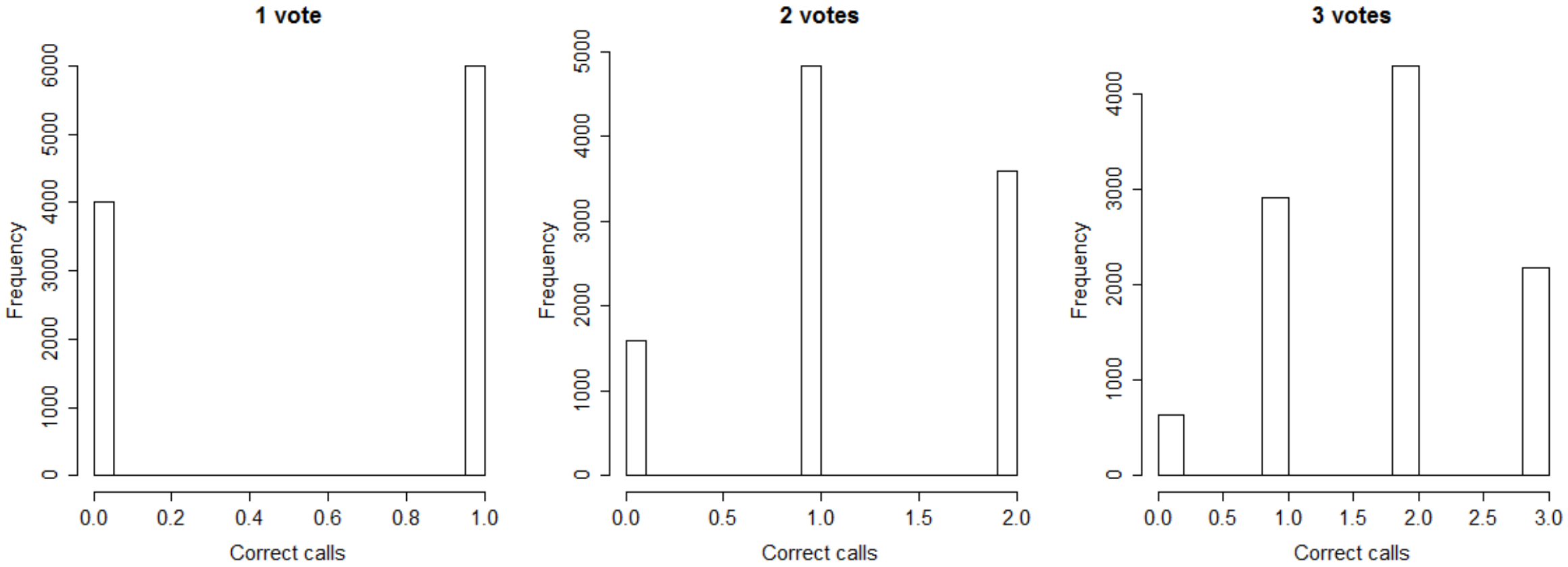
where B^* is the set of models that exclude x_i and N^* is the number of such models. It can be shown that $N^* \sim N / 3$, thus we are getting a reasonable *cross-validation* based estimate for the error of our averaged model!

It can be further shown that with N large enough, out-of-bag (OOB) error is nearly equivalent to "true" leave-one-out cross-validation error.

THE WISDOM OF CROWDS: WHY ENSEMBLE LEARNING WORKS?

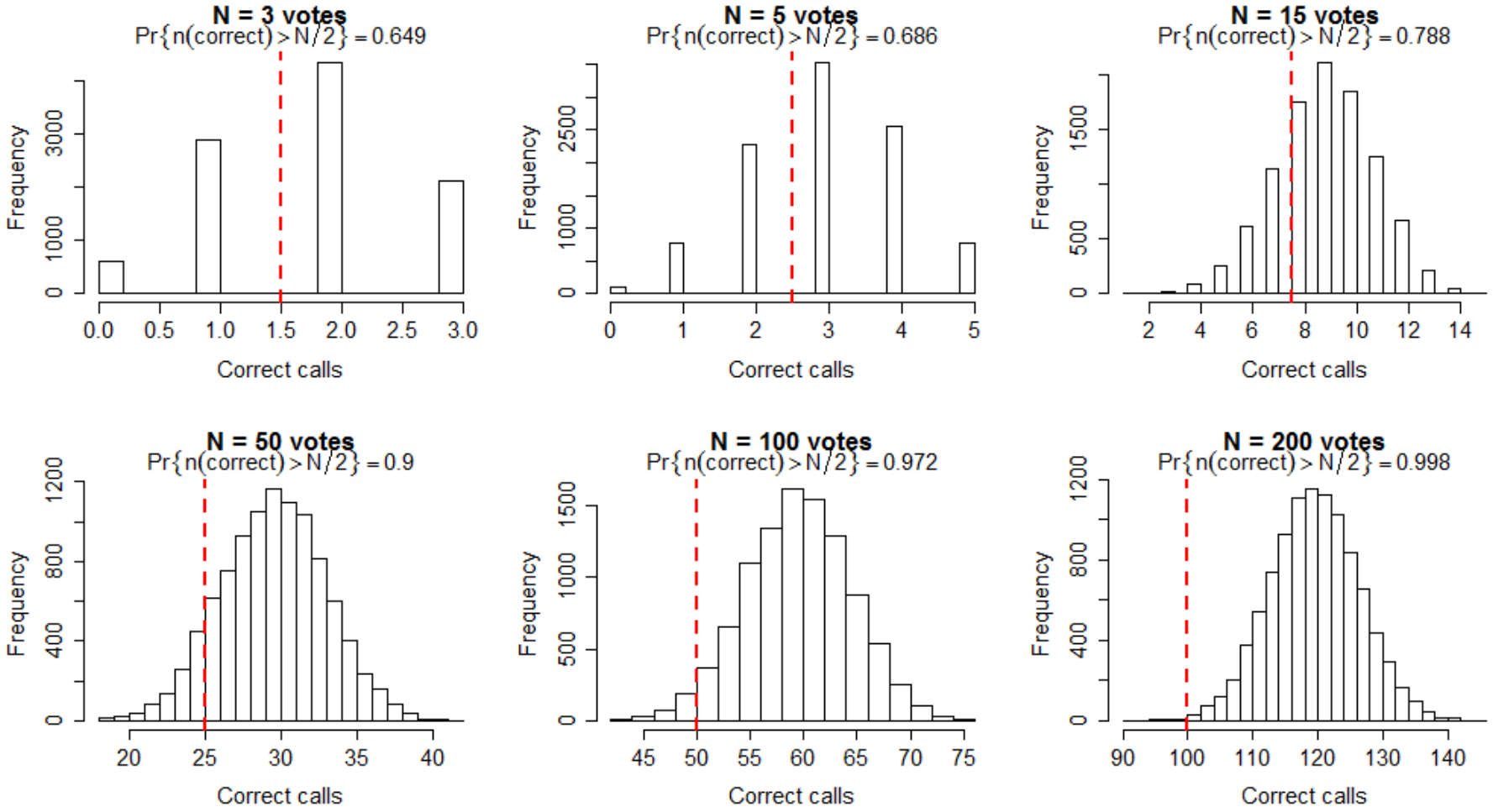
- Short answer: because of error of the mean and CLT (if applicable)
- Consider a two-class problem and a model that makes correct call in 60% of the cases.
- For a set of 100 such models making predictions *independently*, on average the sum of the correct calls each individual model makes (designated with 1's) will be $60 = 0.6 \times 1 \times 100$
- If a given observation is classified on the basis of majority vote of these 100 classifiers, i.e. by comparing their sum with $0.5 \times 1 \times 100 = 50$, the wrong calls will be made for less than 3% of the cases!
  - The sum of $n = 100$ Bernoulli variables with $p = 0.6$ follows binomial (almost Poisson, almost normal) distribution with mean of $np = 60$ and variance of $np(1-p)=24$, so that 50 is about two standard deviations less that 60
- The key assumption here is that of *independence* of those multiple model predictions (if model 2 tends to predict outcome A for observation x_i whenever model 1 predicts outcome A for the same observation, these models are *not* independent!)
  - Not so easy to achieve in practice – how would you go about it when constrained to use only the data that are available?
- Such improvement in predictive accuracy when decision is based on multiple independent models is sometime referred to as "the wisdom of crowds"
  - ESL Ch.8.7 has more examples and technical details (perhaps too technical for ISLR)

THE WISDOM OF CROWDS: A SIMULATION EXAMPLE



- Pr(Correct)=0.6: what is Pr(Correct=2) when n=2? For n=3, Pr(Correct≥2)=0.65 – assuming *independence*!

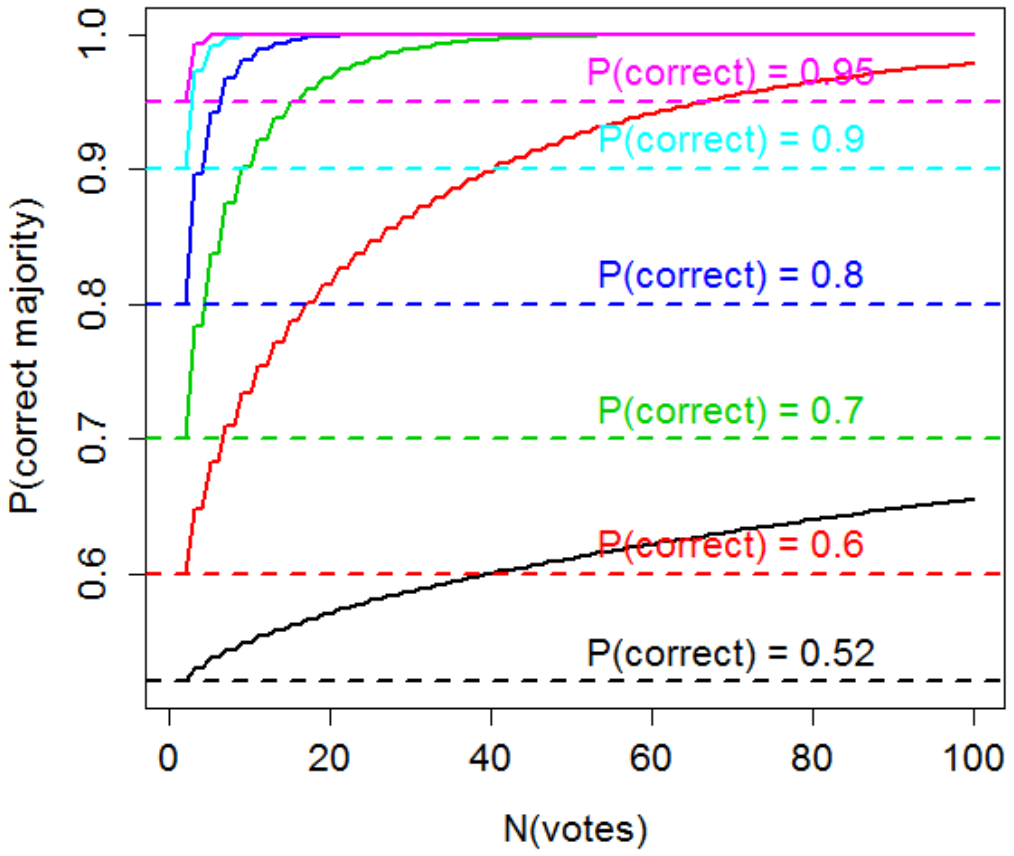
THE WISDOM OF CROWDS: MORE ENSEMBLE SIZES



CODE FOR THE PLOTS

```
truth <- rbinom(10000,size=1,prob=0.5)
flipSmpl <- function(x,ff=0.4){
  tmpIdx<-sample(length(x),ff*length(x))
  x[tmpIdx] <- 1-x[tmpIdx]
  x
}
preds <- apply(matrix(truth,nrow=length(truth),ncol=200),2,flipSmpl)

old.par <- par(mfrow=c(1,3),ps=16)
hist(as.numeric(preds[,1:kTmp]==truth),main="1 vote",xlab="Correct calls")
hist(rowSums(preds[,1:2]==truth),main="2 votes",xlab="Correct calls")
hist(rowSums(preds[,1:3]==truth),main="3 votes",xlab="Correct calls")
par(old.par)

old.par <- par(mfrow=c(2,3),ps=16)
for ( nTmp in c(3,5,15,50,100,200) ) {
  hist(rowSums(preds[,1:nTmp]==truth),20,main=paste("N =",nTmp,"votes"),xlab="Correct calls")
  tmpPcorrect <- mean(rowSums(preds[,1:nTmp]==truth)>nTmp/2)
  mtext(bquote(paste(Pr,group("{",n(correct)>N/2,"}")==.(signif(tmpPcorrect,3)))),cex=0.7)
  abline(v=nTmp/2,lwd=2,lty=2,col=2)
}
par(old.par)
```

WISDOM OF THE CROWDS: FEW MORE COMMENTS



- The plot shows performance of the majority vote as a function of the number of votes for different rates of correct calls
  - Assuming their *independence*, of course!
- This is a gross oversimplification as completely independent models are difficult to come by:
  - The assumption of independence is central to the demonstrated behavior
  - If all votes are perfectly correlated, the performance of their average will be exactly the same as of one of them
  - Models developed on the same set of observations and attributes cannot be fully independent
- However, this provides qualitative demonstration why combining *sufficient* number of *independent enough* models can improve predictive accuracy

CODE FOR THE PLOT

```
old.par <- par(ps=16)
p.correct <- c(0.52, 0.6, 0.7, 0.8, 0.9, 0.95)
p.maj <- NULL
n.votes <- 1:100
for (n in n.votes) {
  p.tmp <- pbinom(n/2, n, p.correct, lower.tail = F)
  if (n%%2 == 0) {
    p.tmp <- p.tmp + dbinom(n/2, n, p.correct)/2
  }
  p.maj <- rbind(p.maj, c(n,p.tmp))
}
matplot(p.maj[,1], p.maj[,2:dim(p.maj)[2]], type="l", lty=1, lwd=2, xlab="N(votes)", ylab="P(correct
majority)")
abline(h = p.correct, col = 1:length(p.correct),lwd=2,lty=2)
text(0.7*max(n.votes), p.correct, paste("P(correct) =",p.correct), pos=3, col=1:length(p.correct))
par(old.par)
```

RANDOM FOREST

- Taking the idea of bagging one step further
  - Remember that we use a greedy algorithm when fitting a tree
  - Even with multiple (actual or bootstrapped) data realizations, the splitting sequence might be still similar, which will result in similar trees and thus highly correlated predictions (i.e. tree number k tends to predict A whenever tree number m predicts A – that's not a good case for "wisdom of crowds" scenario!)
  - As we are now using a large collection of training datasets, we want to make the individual trees explore more "paths" in the parameter space
  - Heuristics: instead of considering all p predictor variables for each split (as a "normal" tree would), randomly select, at each split, a subset of $m<p$ predictors and choose the best split using only the selected variables
  - Clearly, if $m=p$, we have the conventional bagging (so we can and do have a single implementation for both bagging and random forests!)
  - With very large number of very highly correlated predictors we want to keep m small.
  - General rule of thumb: $m \sim \sqrt{p}$

USING BAGGED TREES

- Let us see if bagging can save our tree-based model.
  - We need the library randomForest [NOTE: you may experience difficulties with auto-installation (happened with some older versions at least); in that case download the package and use "Install package(s) from local files…" option!]
  - Remember that bagging is simply a random forest with $m=p$, where p is the total number of predictor variables

```
library(randomForest)
digit.bag=randomForest(D~.,
    data=data.frame(D=as.factor(l1),
        m=m1[,-no.change]),
        mtry=ncol(m1)-length(no.change))
```

RESULTS FOR BAGGING

- As a result of bagging we obtain a noticeable improvement in model accuracy and now we are ~7% better than LDA

```
> digit.bag

Call:
 randomForest(formula = D ~ ., ...
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 519

        OOB estimate of  error rate: 8.3%
Confusion matrix:
    0   1   2   3   4   5   6   7   8   9 class.error
0 499   0   3   1   0   6   2   0   2   0  0.02729045
1   0 600   4   1   1   2   1   2   1   0  0.01960784
2   5   4 434   5   2   6   5   7   8   4  0.09583333
3   3   4  18 436   0  18   3   8   7   6  0.13320080
4   1   1  10   0 423   3   3   2   5  20  0.09615385
5   5   4   2   9   1 419  13   2   4   2  0.09110629
6   1   2   3   1   3   9 509   0   8   0  0.05037313
7   5   4   5   2   3   2   0 441   1  15  0.07740586
8   1   6  11  13   4  13   4   1 409  15  0.14255765
9   4   2   4  16   8   4   2   8   9 415  0.12076271
```

```
> bag.pred=predict(digit.bag,newdata=
    data.frame(m=m.test[,-no.change]),type="class")
> table(bag.pred,l.test)
        l.test
bag.pred    0    1    2    3    4    5    6    7    8    9
       0  950    0   18    8    2   10   16    5    5   10
       1    0 1110   12    4    6    8    6   16    5    6
       2    5    5  918   20    8    1    5   22   15    9
       3    1    3   12  900    1   30    0    3   20   13
       4    2    3    9    3  887    7    8    4    5   22
       5   12    3    8   27    3  789   25    2   19   14
       6    2    5   14    5   17   19  883    0   13    6
       7    3    0   17   13    2    9    1  940    3    9
       8    5    4   21   21   16    8   14   11  865    9
       9    0    2    3    9   40   11    0   25   24  911
sum(diag(table(bag.pred,l.test)))
[1] 9153 ### 92% ACCURACY on the test set!!!!!!
```

Note how OOB estimate obtained during model fitting is very close to the true error rate on the independent test set!

RESULTS FOR RANDOM FOREST

- Random forest with $m = \sqrt{p}$ provides even further improvement:

```
> digit.rf=randomForest(D~.,
   data=data.frame(D=as.factor(l1),m=m1[,-no.change]),
   mtry=as.integer(sqrt(ncol(m1)-length(no.change))))
> digit.rf
Call:
 randomForest(formula = D ~ ., ...
               Type of random forest: classification
                     Number of trees: 500
No. of variables tried at each split: 22

        OOB estimate of  error rate: 5.88%
Confusion matrix:
    0   1   2   3   4   5   6   7   8   9 class.error
0 508   0   0   0   0   1   2   0   2   0 0.009746589
1   0 601   3   0   0   3   1   3   1   0 0.017973856
2   3   0 447   2   1   3   6   5  10   3 0.068750000
3   1   3  10 460   0  10   1   6   6   6 0.085487078
4   1   0   3   0 433   0   5   1   4  21 0.074786325
5   4   3   3  10   3 424   9   1   3   1 0.080260304
6   0   0   1   0   2   5 525   0   3   0 0.020522388
7   2   5   5   1   6   0   0 447   0  12 0.064853556
8   1   5   5  11   2   8   3   0 427  15 0.104821803
9   3   2   0  11   7   1   1   9   4 434 0.080508475
```

```
> rf.pred=predict(digit.rf,newdata=
       data.frame(m=m.test[,-no.change]),type="class")
> table(rf.pred,l.test)
       l.test
rf.pred    0    1    2    3    4    5    6    7    8    9
      0  966    0   11    6    2    8   15    2    4    8
      1    1 1121    1    0    1    6    3   12    3    7
      2    0    3  951   18    3    1    1   24    7    3
      3    0    2   11  940    0   20    0    6   17   11
      4    0    1   12    1  924    6    4    3    5   21
      5    3    0    2   16    0  823   14    0    8   10
      6    6    5    8    3   13   14  919    0   11    3
      7    1    0   21   10    0    2    0  956    4    4
      8    3    2   13   11    3    7    2    6  896    6
      9    0    1    2    5   36    5    0   19   19  936
> sum(diag(table(rf.pred,l.test)))
[1] 9432 ### 94% accuracy on the test set !!!!
```

INTERPRETATION OF RANDOM FORESTS

- Earlier we mentioned interpretability as one of the attractive features of decision trees
- Interpretability clearly suffers when an ensemble of different trees (a random forest) is being averaged.
- It is possible however, to calculate an *average* decrease in RSS/Gini/deviance per split for a specific variable, across multiple trees. This will give some measure of the "importance" of the variable.
- Use `importance(tree)` or `varImpPlot(tree)` to examine the importance, where `tree` is a bagged tree/random forest object (returned by `randomForest()` function).
- Similarly, for a given attribute its marginal effect on class probability (or predicted response for regression) can be also estimated over all models in the ensemble and presented graphically yielding "partial dependence plots":

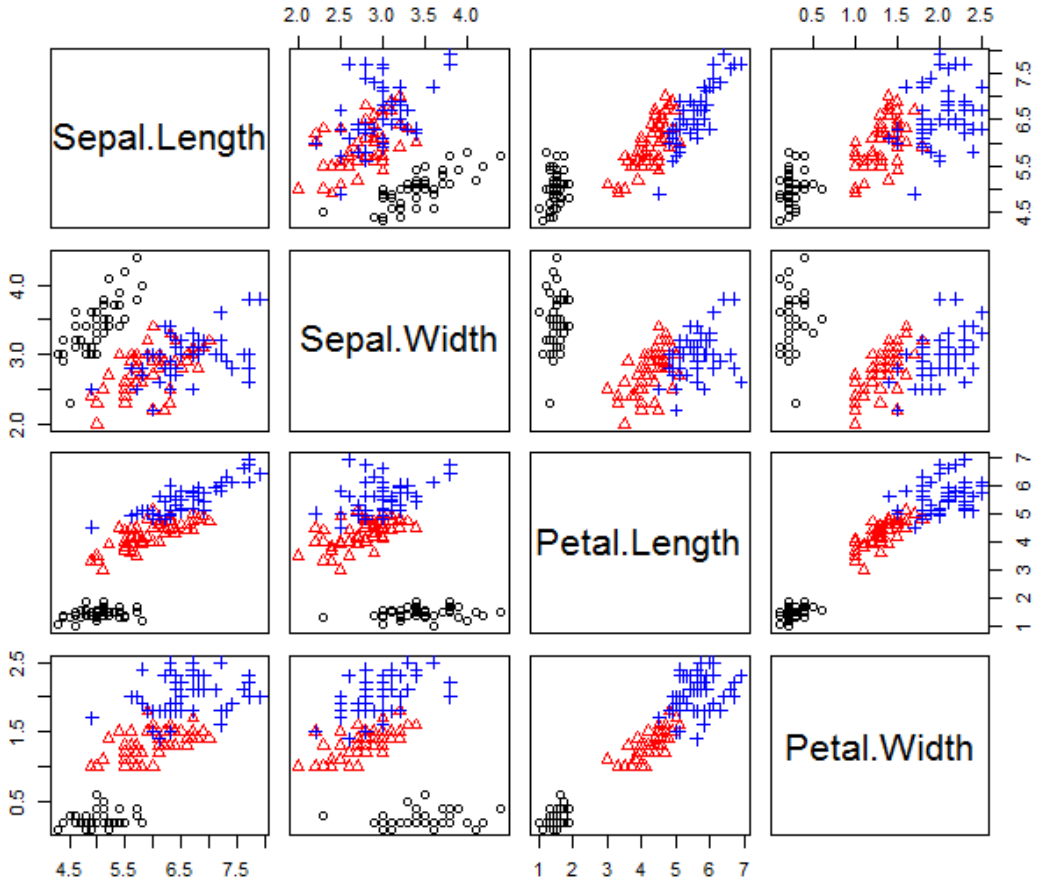
$$\tilde{f}(x) = \frac{1}{n}\sum_{i=1}^{n} f(x, x_{iC}), f(x) = \log p_k(x) - \frac{1}{K}\sum_{j=1}^{K} \log p_j(x)$$

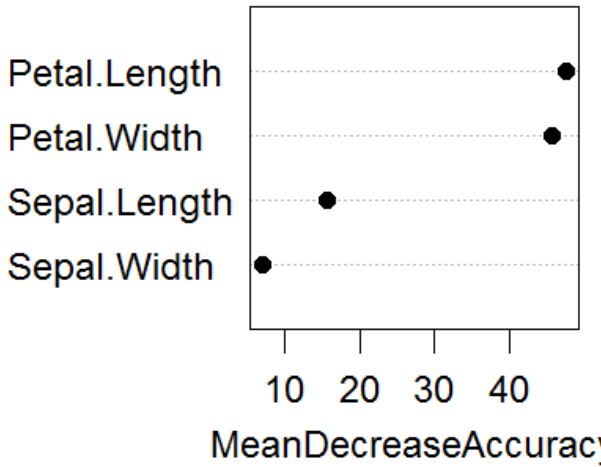
- Implemented in R as `partialPlot(tree, pred.data, x.var, which.class)` – model, data, which variable to profile and its effect on which class to present
- Let's turn to our old friend, iris dataset, for few simple illustrations

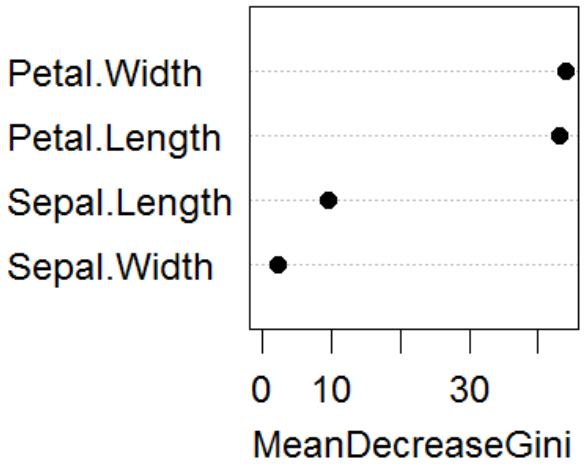
CODE FOR THE PLOTS SHOWN IN THE NEXT SLIDE

```
> table(iris$Species)
    setosa versicolor  virginica
        50         50         50
> tmpIdx <- as.numeric(iris$Species)
> pairs(iris[,-ncol(iris)],col=c("black","red","blue")[tmpIdx],pch=tmpIdx)
> irisRF <- randomForest(Species~.,iris,ntree=1000,importance=TRUE)
> irisRF

Call:
 randomForest(formula = Species ~ ., data = iris, ntree = 1000,      importance = TRUE)
               Type of random forest: classification
                     Number of trees: 1000
No. of variables tried at each split: 2

        OOB estimate of  error rate: 4%
Confusion matrix:
           setosa versicolor virginica class.error
setosa         50          0         0        0.00
versicolor      0         47         3        0.06
virginica       0          3        47        0.06
> varImpPlot(irisRF)
```

VARIABLE IMPORTANCE PLOTS

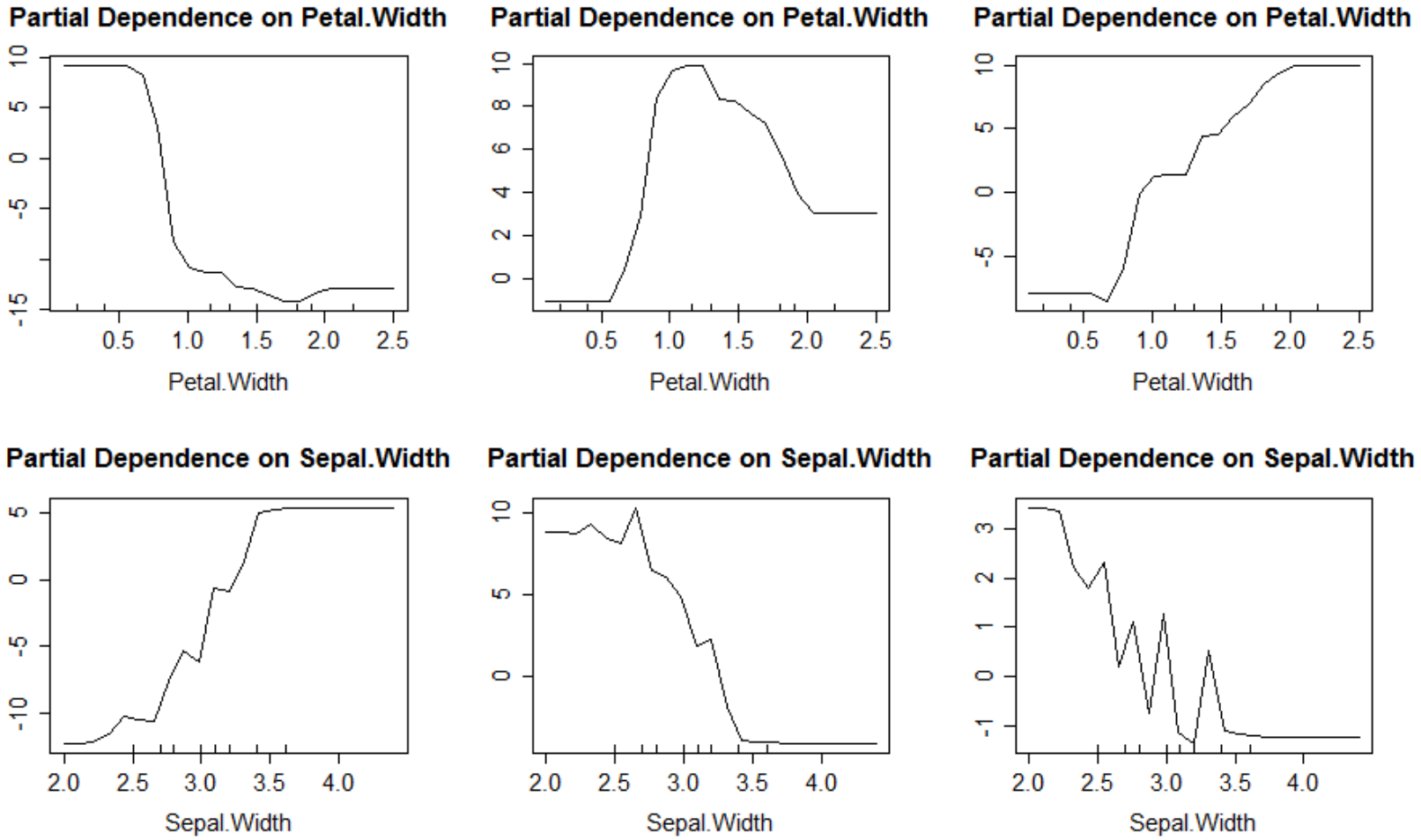






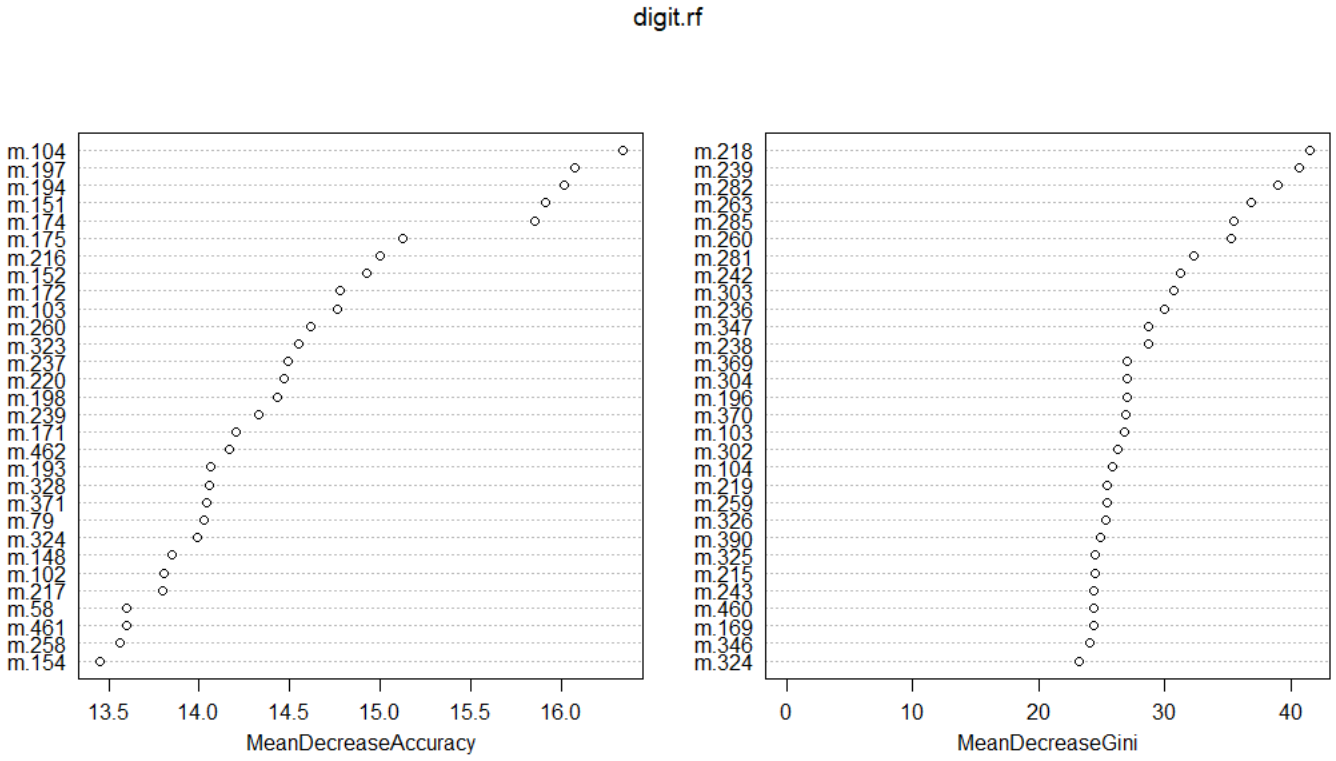
- Represents impact of scrambling each attribute on the classifier performance (as measured by accuracy or Gini)
  - Notice petal length and width switching depending on metric
- For iris data, petal attributes are much better predictors of iris species than those for sepal
- Can be less obvious for more complex effects in higher dimensions

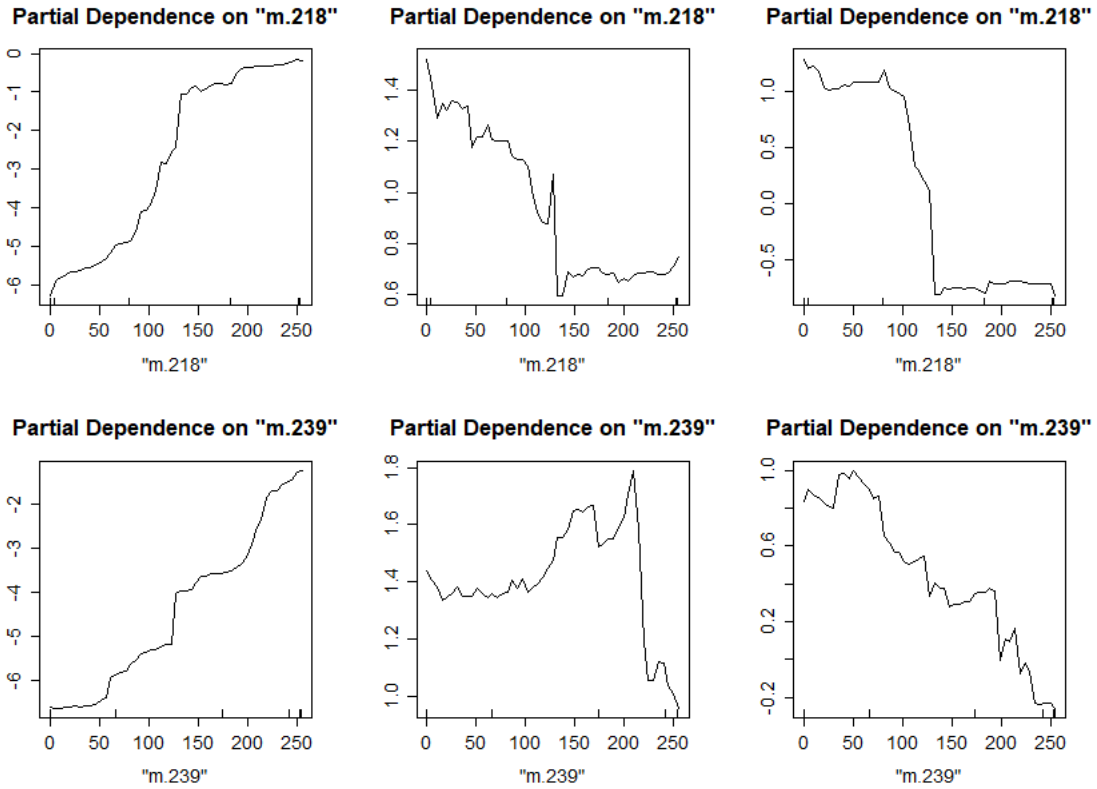
PARTIAL DEPENDENCE PLOTS



- Notice how higher partial dependence for a given class corresponds to the range of attribute values predominantly populated by this class, e.g.:
- Petal.Width<1 for setosa
- 1<Petal.Width<1.5 – versicolor
- Petal.Width>2 – virginica
- SepalWidth>3.5 – setosa
- Sepal.Width<2.5 – versicolor
- Much lower range for partial dependence on Sepal.Width for virginica – why?

DIGITS: VARIABLE IMPORTANCE AND PARTIAL DEPENDENCE



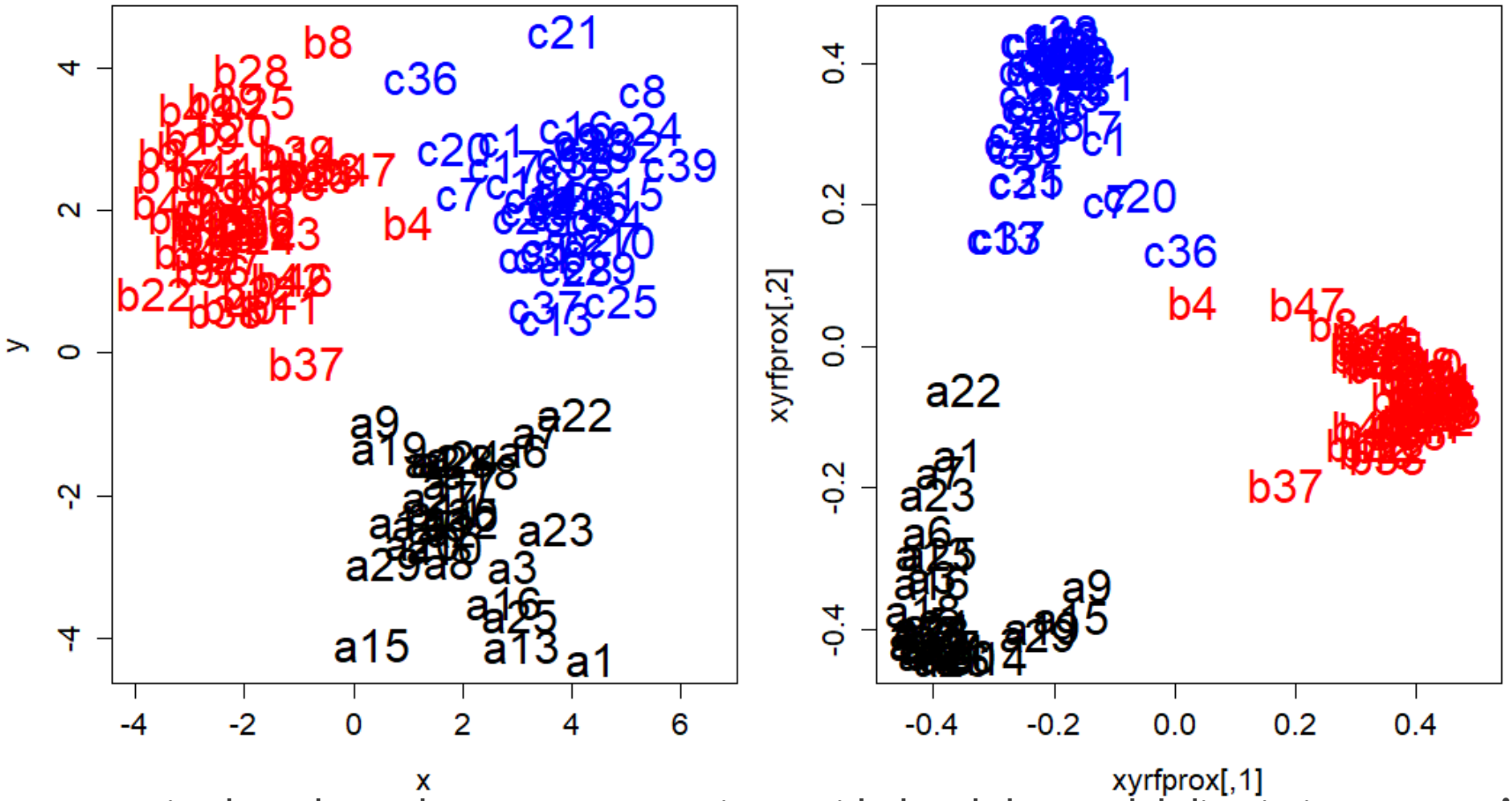


- Although order of importance depends on the metric, they are all (scaled) much higher than with randomized labels
- Range of the values of partial dependencies vary with the class for which they are calculated

CODE FOR THE PREVIOUS PLOTS

```
> digit.rf=randomForest(D~., data=data.frame(D=as.factor(l1), m=m1[,-no.change]),
mtry=as.integer(sqrt(ncol(m1)-length(no.change))), importance=TRUE)
> varImpPlot(digit.rf)
> old.par <- par(mfcol=c(2,3),ps=16)
> partialPlot(digit.rf,data.frame(D=as.factor(l1),m=m1[,-no.change]),"m.218",1)
> partialPlot(digit.rf,data.frame(D=as.factor(l1),m=m1[,-no.change]),"m.239",1)
> partialPlot(digit.rf,data.frame(D=as.factor(l1),m=m1[,-no.change]),"m.218",2)
> partialPlot(digit.rf,data.frame(D=as.factor(l1),m=m1[,-no.change]),"m.239",2)
> partialPlot(digit.rf,data.frame(D=as.factor(l1),m=m1[,-no.change]),"m.218",4)
> partialPlot(digit.rf,data.frame(D=as.factor(l1),m=m1[,-no.change]),"m.239",4)
> par(old.par)
```

RANDOM FOREST FOR UNSUPERVISED LEARNING

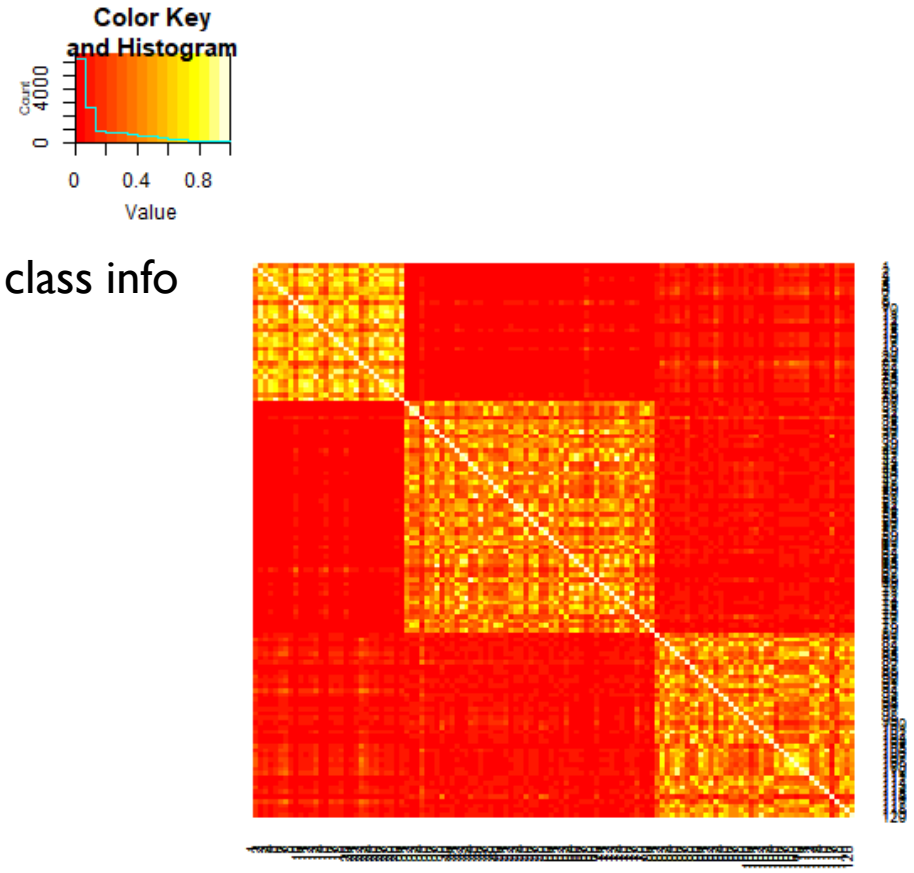


- Can be used in unsupervised mode – when **no** outcome is provided and the model discriminates **provided data** from a **uniform sample**; proximity then is the frequency of two observations to fall in the same terminal node

CODE FOR THE PREVIOUS PLOTS

```
> set.seed(123);invisible(rnorm(10000))
> x=c(rnorm(30,mean=2),rnorm(50,mean=-2),rnorm(40,mean=4))
> y=c(rnorm(30,mean=-2),rnorm(50,mean=2),rnorm(40,mean=2))
> clr=c(rep("black",30),rep("red",50),rep("blue",40))
> lbls=c(paste0("a",1:30),paste0("b",1:50),paste0("c",1:40))
> rf=randomForest(cbind(x,y),proximity=TRUE,ntree=5000)
> heatmap.2(rf$proximity, trace="none", Rowv=FALSE,
+ Colv=FALSE, dendrogram="none")
> dim(rf$proximity)
[1] 120 120
> rf$proximity[1:5,1:5]
          [,1]      [,2]      [,3]      [,4]      [,5]
[1,] 1.0000000 0.3438914 0.5374251 0.2422907 0.3738602
[2,] 0.3438914 1.0000000 0.5865672 0.6832579 0.8251121
[3,] 0.5374251 0.5865672 1.0000000 0.4179331 0.6606876
[4,] 0.2422907 0.6832579 0.4179331 1.0000000 0.6062092
[5,] 0.3738602 0.8251121 0.6606876 0.6062092 1.0000000
> xyrfprox <- cmdscale(1-rf$proximity)
> old.par=par(mfrow=c(1,2),ps=16,mar=c(4,4,0,0)+0.5)
> plot(x,y,type="n",xlim=1.1*range(x))
> text(x,y,lbls,col=clr,cex=1.5)
> plot(xyrfprox,type="n",xlim=1.1*range(xyrfprox[,1]))
> text(xyrfprox,lbls,col=clr,cex=1.5)
> par(old.par)
```

No class info



BOOSTING

- One last technique that works well with decision trees is boosting
- The idea is, again, similar to bagging/random forests: build an ensemble of individual decision trees
- The trees are not random in boosting; instead, we grow each tree to fit the residuals of the model built in the previous step:
  - Start with model f(x)=0 and $r_i=y_i$ (i.e. "residuals" start as full data)
  - In each step n=1…N:
    - Fit decision tree $f^n(x)$ with d splits to the data (X, r) [note that we fit the residuals left after previous step here!]
    - Update f as $f(x) \leftarrow f(x) + \lambda f^n(x)$
    - Update residuals as $r_i \leftarrow r_i - \lambda f^n(x_i)$
- Note that we are not trying to fit as much of the current residuals as possible, but learn only a fraction λ (learning rate). Methods that learn gradually and slowly usually work better!
- Since we are learning the residuals gradually and step by step, and since each tree depends on the trees already grown, we do not have to fit a large tree in each step: d can be small
- Check out the `xgboost` package

BOOSTING ON HANDWRITTEN DIGITS

```
> digits.bst <- xgboost(data=m1[,-no.change], label=l1, nrounds=100,
objective="multi:softmax", num_class=10, eta=0.1, max_depth=4, subsample=0.5)
[1]     train-merror:0.221600
[2]     train-merror:0.164600
[3]     train-merror:0.144600
...
[98]    train-merror:0.003600
[99]    train-merror:0.003400
[100]   train-merror:0.002800
> sum(diag(table(predict(digits.bst,m.test[,-no.change]),l.test)))
[1] 9368
```

- Many parameters to choose from: nrounds, eta, max_depth, subsample, etc.
- Performs very competitively with random forest

SUMMARY

- Today we have examined decision trees
- Decision boundary: piecewise, splits on individual predictors
- Just like other models, a tree can overfit, cross-validation is required
  - One technique for keeping the tree complexity in check and controlling the variance is *pruning*
- We have examined a large realistic dataset and observed that simple decision tree was easily outperformed even by naïve LR and by LDA. This is *not* an exception. A simple, single tree is often a poor (and not very stable) model
- We next examined techniques that allow building an *ensemble* of models (note that these techniques can be in principle generalized to other types of models)
  - Bagging: bootstrap multiple datasets D_i, fit a decision tree T_i to each such dataset, then make a prediction on an observation x by averaging predictions from all T_i . While performing bagging, one can also obtain the OOB estimate of the *test* error
  - Random forest: very similar to bagging, but we try making the trees more diverse. As we build a tree on a bootstrapped dataset, at each split we randomly select a subset of variables from which we are going to chose the best one for splitting
  - Boosting: repeat growing next tree to (partially) fit the residuals of the current state of the model
- We have achieved quite impressive classification performance on a realistic dataset!