Practical ML - Week 3

Aura Frizzati

Week 3: Predicting with trees, Random Forests, & Model Based Predictions

Week 3.1: Predicting with trees

Key ideas

- Iteratively split the outcome by splitting the predictive variables into groups
- Evaluate "homogeneity" of the outcome within each group
- Split again if necessary, until you get outcomes that are separated into groups that are **homogeneous enough** or that are **small enough**.

Pros
- Easy to interpret
- Better performance in non-linear settings (in comparison to linear regression models)

Cons
- Without pruning/cross-validation, can lead to overfitting
- Harder to estimate uncertainty (in comparison to linear regression models)
- Results may be variable and depending on the exact values of parameters or the variables that have been collected

Basic algorithm

1. Start with all variables in one group
2. Find the variable/split that best separates the outcome into two different homogenous groups
3. Divide the data into 2 groups ('leaves') on that split ('node')
4. Within each split, find the best variable/split (including variables we have already used to split the groups) that separates the outcome
5. Continue until the groups are too small or sufficiently 'pure' (i.e. homogenous)

Measures of impurity

They are all based on this probability which can be estimated:

$$\hat{p}_{mk} = \frac{1}{N_m} \sum_{x_i \; in \; Leaf \; m} (y_i = k)$$

Within a particular group (the m leaf) you have N_m total objects you might consider. You can count the number of times that a particular class ($y_i = k$) appears in that leaf.

- **Misclassification error:**

$$1 - \hat{p}_{mk(m)}$$

  $k(m)$ being the most common k outcome class in the dataset

misclassification error = 0 –> perfect purity (no misclassification error)
misclassification error = 0.5 –> no purity

- **Gini index:**

$$\sum_{k \neq k'} \hat{p}_{mk} * \hat{p}_{mk'} = \sum_{k=1}^{K} \hat{p}_{mk}(1 - \hat{p}_{mk}) = 1 - \sum_{k=1}^{K} \hat{p}_{mk}^2$$

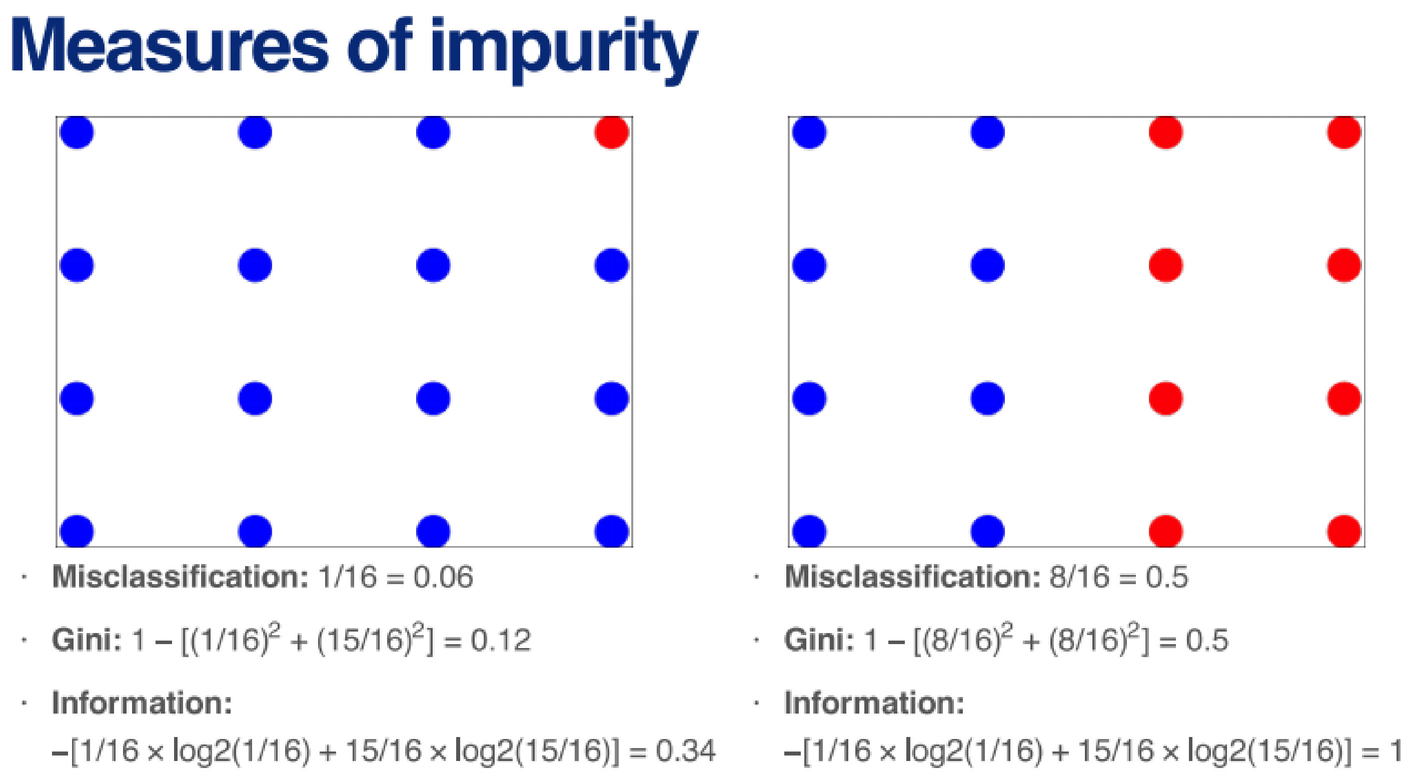
  (to not confuse it with the Gini coefficient used in economics!)

The Gini index is 1 - the sum of the squared probability that a sample belongs to any of the different outcome classes. **Gini index** = 0 –> perfect purity (this implies one outcome class has classification probability = 1, while all the others 0)
Gini index = 0.5 –> no purity (all of the classes are perfectly balanced within each leaf)

- **Deviance/information gain:**

$$-\sum_{k=1}^{K} \hat{p}_{mk} log_2[\hat{p}_{mk}]$$

  This measure is called **deviance** if you use ln or otherwise **information gain** using log_2. **Deviance/information gain** = 0 –> perfect purity **Deviance/information gain** = 1 –> no purity



In the example above, the left panel represents a relatively pure split, while the right one is extremely impure.

Example: Iris data

```
data(iris); library(ggplot2)
names(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

```
table(iris$Species)
```

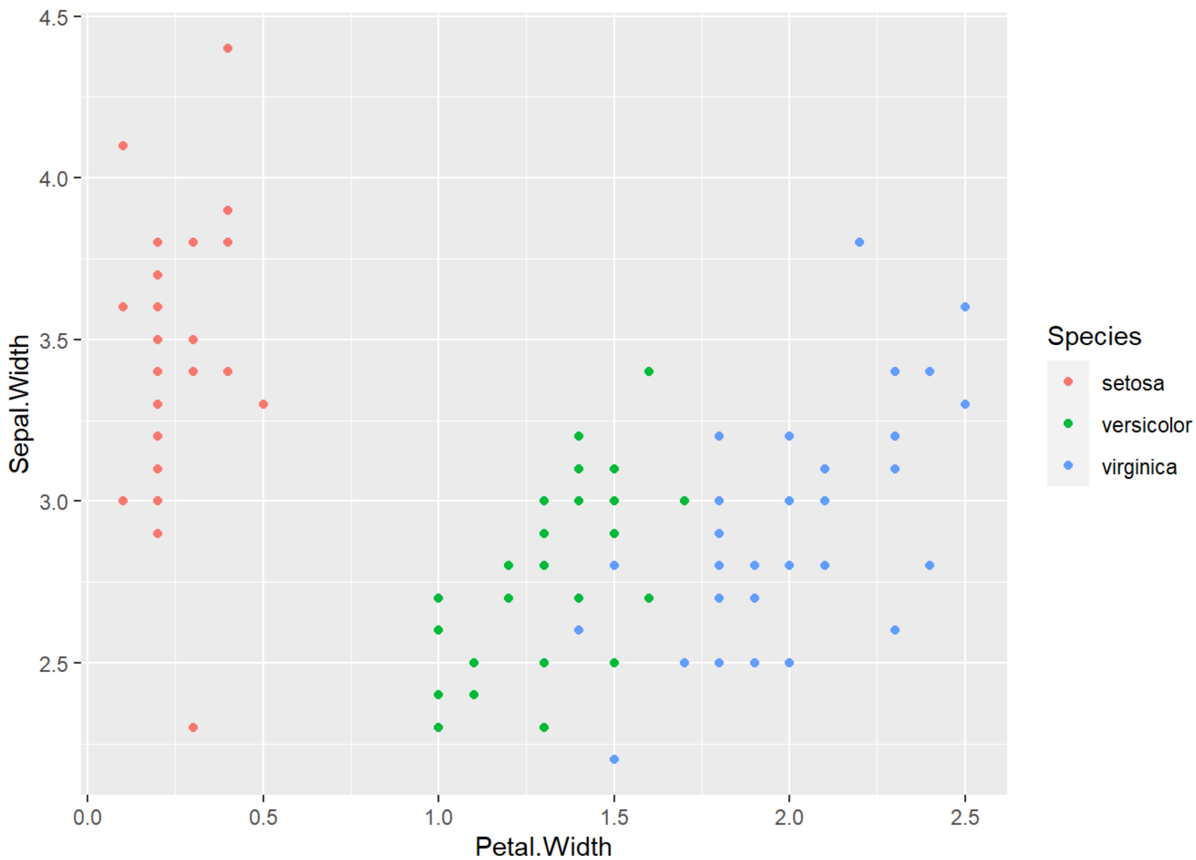
```
##
##     setosa versicolor  virginica
##         50         50         50
```

```
library(caret)
inTrain<-createDataPartition(y = iris$Species, p=0.7,list=F)
training<-iris[inTrain,]
testing<-iris[-inTrain,]
dim(training);dim(testing)
```

```
## [1] 105   5
```

```
## [1] 45  5
```

```
qplot(Petal.Width, Sepal.Width, colour = Species, data= training)
```



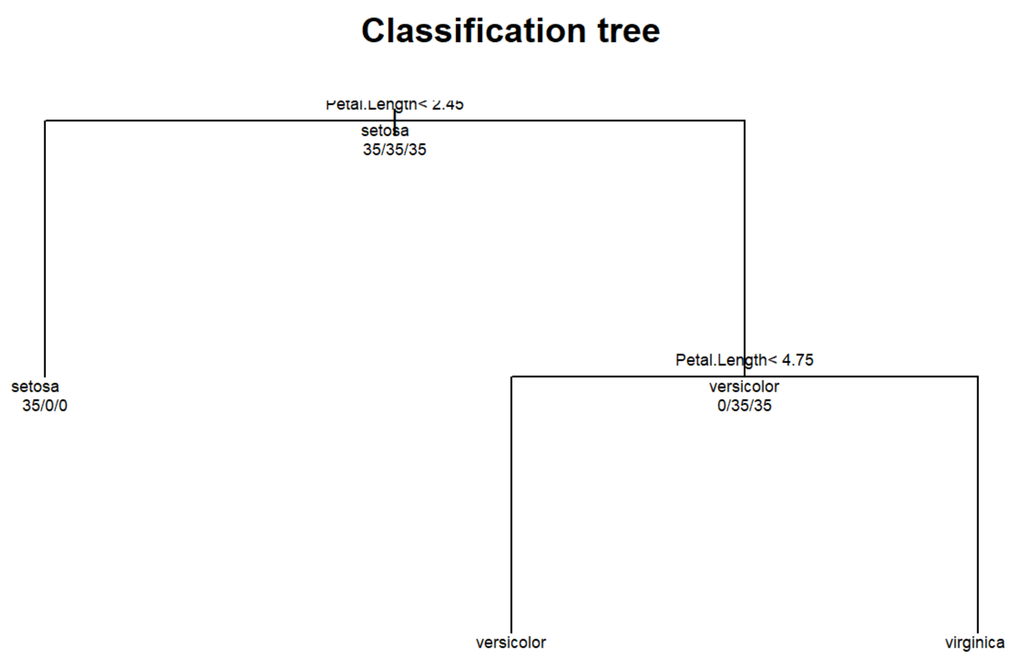
There are 3 very distinct classes, although it could be challenging to predict them for a linear model (but a classification tree would be able to handle non linearity).

```
modFit<-train(Species ~ .,method = 'rpart', data = training)
print(modFit$finalModel)
```

```
## n= 105
##
## node), split, n, loss, yval, (yprob)
##       * denotes terminal node
##
## 1) root 105 70 setosa (0.3333333 0.3333333 0.3333333)
##   2) Petal.Length< 2.45 35  0 setosa (1.0000000 0.0000000 0.0000000) *
##   3) Petal.Length>=2.45 70 35 versicolor (0.0000000 0.5000000 0.5000000)
##     6) Petal.Length< 4.75 32  1 versicolor (0.0000000 0.9687500 0.0312500) *
##     7) Petal.Length>=4.75 38  4 virginica (0.0000000 0.1052632 0.8947368) *
```

`rpart` is an R package for doing regression and classification trees.
`modFit$finalModel` tells you what the final nodes/leaves are, how they split, and the probability for each class to be in each split.

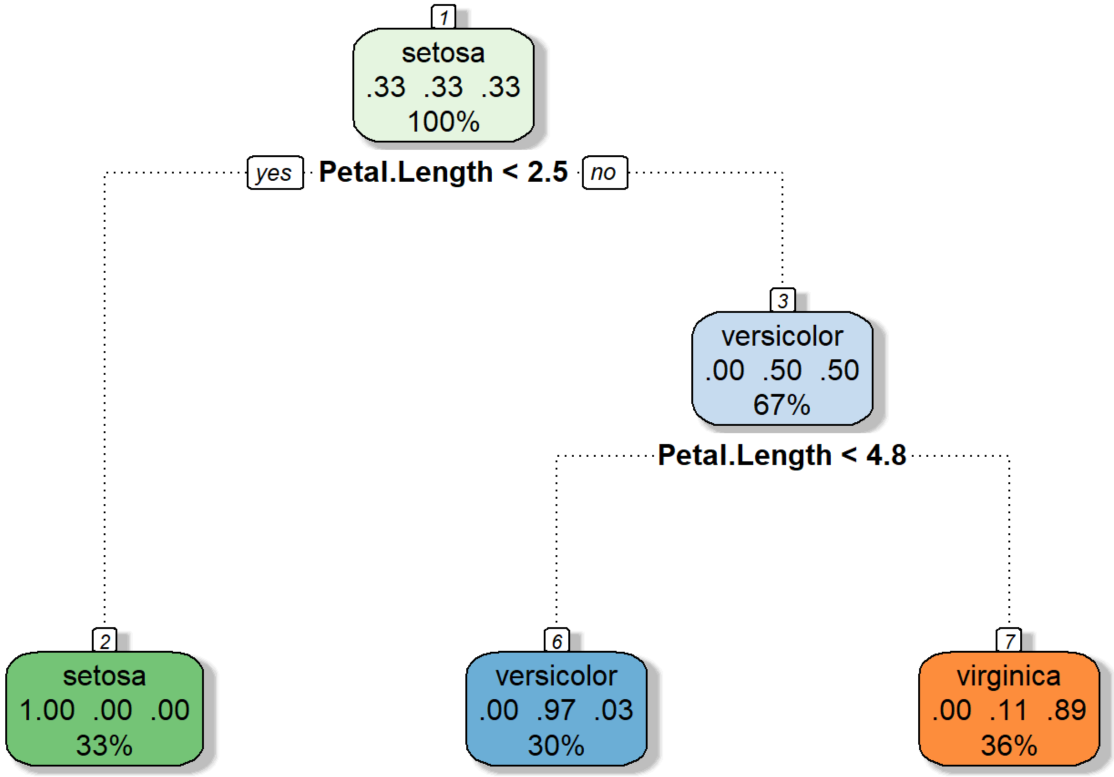
```
plot(modFit$finalModel, uniform = T, main = "Classification tree")
text(modFit$finalModel, use.n = T, all = T, cex =.55)
```



This is a **dendogram**. The branch to the left denotes when the condition is true, while the branch to the right when it is false.

A prettier version of the same plot can be made with the `rattle` package:

```
library(rattle)
fancyRpartPlot(modFit$finalModel)
```



Rattle 2021-Apr-03 13:49:03 au228742

You can predict new values using the `predict` function:

```
predict(modFit,newdata = testing)
```

```
##  [1] setosa     setosa     setosa     setosa     setosa     setosa    
##  [7] setosa     setosa     setosa     setosa     setosa     setosa    
## [13] setosa     setosa     setosa     versicolor virginica  versicolor
## [19] versicolor versicolor versicolor versicolor virginica  versicolor
## [25] versicolor versicolor versicolor versicolor versicolor versicolor
## [31] virginica  virginica  virginica  virginica  virginica  virginica 
## [37] virginica  virginica  virginica  virginica  virginica  virginica 
## [43] virginica  virginica  virginica 
## Levels: setosa versicolor virginica
```

Final notes

- Classification trees are **non-linear models**
  - They use **interactions** between variables
  - Data transformation may be less important (monotone transformations)
  - Trees can also be used for regression problems (continuous outcome)
- Note that there are multiple tree building options in R, both in the caret package (`party` , `rpart`) and out of it (`tree`)

https://www.amazon.com/Classification-Regression-Trees-Leo-Breiman/dp/0412048418
(https://www.amazon.com/Classification-Regression-Trees-Leo-Breiman/dp/0412048418)

Week 3.2: Bagging

When you fit complicated models, sometimes if you average them together you get a smoother model fit that gives a better balance between potential bias and varianc ein your fit.

Boostrap aggregating (bagging)

- Basic idea:
    1. **Resample cases** (similarly to what is done in boostrapping) and **re-calculate predictions**
    2. **Average** or **majority** of votes from the predictors you have built in this way
- Notes:
    - Similar bias (to the bias you would obtain by fitting any of those models individually)
    - **Reduce variance** (because you have averaged a bunch of predictors together)
    - More useful for **non-linear functions** (e.g. trees or smoothing)

Example on the Ozone data

```
ozone<-read.csv("practical ML course/data/ozone_data.csv", header = T)
ozone<-ozone[order(ozone$ozone),]
head(ozone); dim(ozone)
```

```
##     ozone radiation temperature wind
## 17      1         8          59  9.7
## 19      4        25          61  9.7
## 14      6        78          57 18.4
## 45      7        48          80 14.3
## 106     7        49          69 10.3
## 7       8        19          61 20.1
```

```
## [1] 111   4
```

The objective is to predict ‘temperature’ as a function of ‘ozone’.

Bagged LOESS (Locally Estimated Scatterplot Smoothing)

Local regression or **local polynomial regression**,also known as **moving regression**, is a generalization of moving average and polynomial regression. Its most common methods, initially developed for **scatterplot smoothing**, are **LOESS** (locally estimated scatterplot smoothing) and **LOWESS** (locally weighted scatterplot smoothing). They are two strongly related **non-parametric regression methods** that combine **multiple regression models** in a **k-nearest-neighbor-based meta-model**.

LOESS is a kind of smooth curve that fits through the data. It is similar to a spline model fitting.

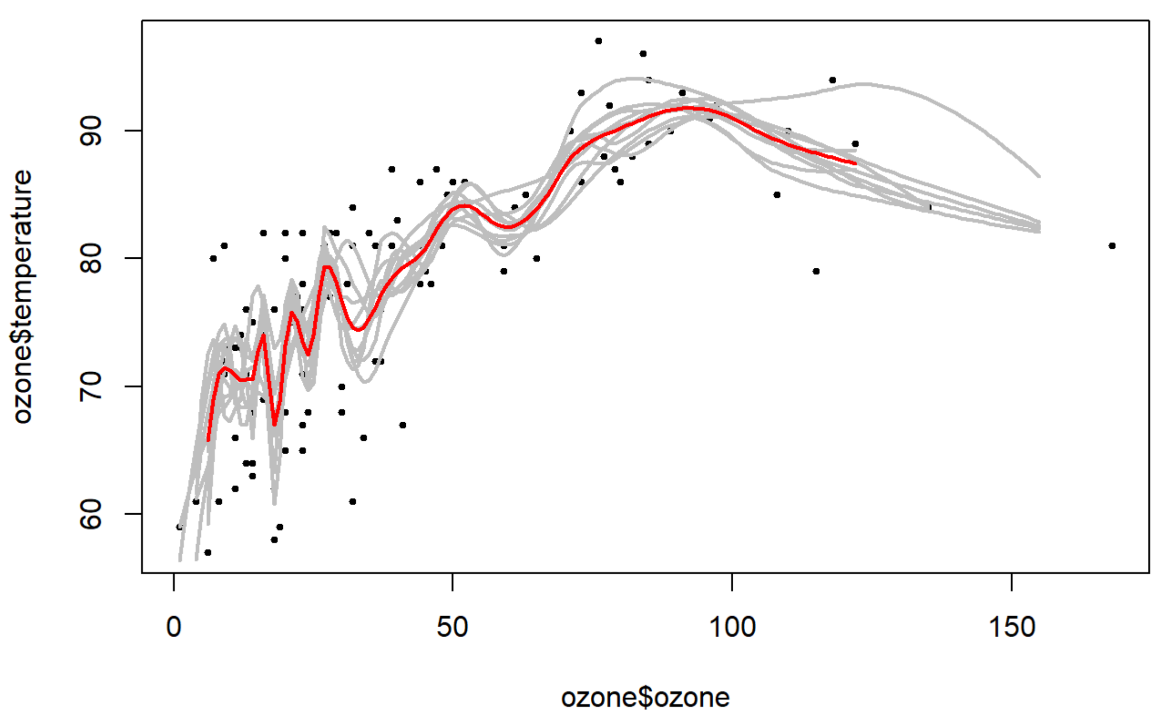
```
ll<-matrix(NA,nrow=10,ncol=155)
for(i in 1:10){
  ss<-sample(1:dim(ozone)[1], replace = T) #dim(ozone)[1] = N rows in ozone df
  ozone0<-ozone[ss,] # resampled dataset
  ozone0<-ozone0[order(ozone0$ozone),]
  # the 'span' for loess measures how smooth the fit will be
  loess0<-loess(temperature ~ ozone, data = ozone0, span = 0.2)
  ll[i,]<-predict(loess0,newdata=data.frame(ozone=1:155))
}
head(ll)
```

```
##          [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]     [,8]
## [1,]       NA       NA       NA       NA       NA 64.89773 70.19492 72.74249
## [2,]       NA       NA       NA 61.21997 62.32309 63.77131 66.03722 70.73670
## [3,]       NA       NA       NA       NA       NA 59.19743 65.10268 69.35389
## [4,] 56.39602 59.63955 62.63709 65.31734 67.87637 70.38038 72.56327 74.15892
## [5,]       NA       NA       NA       NA       NA 60.13440 67.21734 71.16277
## [6,] 59.00478 60.67643 62.22088 63.63669 64.92240 66.07655 67.09770 67.93454
##          [,9]    [,10]    [,11]    [,12]    [,13]    [,14]    [,15]    [,16]
## [1,] 73.25019 73.30859 74.73348 73.35884 70.50015 69.60005 71.99031 72.77043
## [2,] 73.22719 71.54321 68.58573 69.19598 71.17019 71.52438 70.93746 70.01851
## [3,] 71.16744 73.11886 73.83053 73.07516 70.67103 65.90210 71.03505 76.33333
## [4,] 74.90121 73.17130 69.54214 67.03530 66.96636 69.67536 74.03955 77.16881
## [5,] 72.77666 72.07993 70.30595 69.95828 68.41948 68.22835 73.25946 77.00663
## [6,] 68.32931 68.96017 69.55882 70.23541 73.71745 71.99616 72.84947 72.78788
##         [,17]    [,18]    [,19]    [,20]    [,21]    [,22]    [,23]    [,24]
## [1,] 66.58413 60.75302 64.51590 73.20276 78.06522 75.39655 72.53654 71.58738
## [2,] 68.14875 66.84219 69.84663 74.63742 76.30170 75.63740 72.47871 73.67577
## [3,] 71.73777 66.09308 67.52610 70.96390 73.14435 73.14302 72.68402 72.64444
## [4,] 74.98300 72.95066 73.65476 75.88432 77.69295 75.96519 74.20216 74.93017
## [5,] 72.38359 66.29882 66.42095 70.70411 76.50000 76.84167 74.25000 69.66667
## [6,] 67.45751 63.25643 68.88436 73.85041 76.14286 74.48647 72.45455 72.65237
##         [,25]    [,26]    [,27]    [,28]    [,29]    [,30]    [,31]    [,32]
## [1,] 73.03001 75.96026 78.59466 79.14975 77.02969 73.18822 71.93484 71.35385
## [2,] 76.70826 79.82081 81.25800 79.73113 77.92801 76.18355 74.79063 74.53118
## [3,] 74.65186 77.71257 79.62279 80.21347 79.37657 76.68228 73.77261 72.08506
## [4,] 76.31682 77.82801 78.92964 79.08760 77.54382 75.97492 74.00341 72.98444
## [5,] 70.52887 77.90490 82.47890 81.81228 80.53134 79.91751 79.16904 78.05240
## [6,] 73.58947 75.02805 76.73032 78.45848 79.97475 81.04131 81.42038 81.06084
##         [,33]    [,34]    [,35]    [,36]    [,37]    [,38]    [,39]    [,40]
## [1,] 71.63348 72.96197 76.07344 79.13338 81.33347 81.79786 81.99047 81.83223
## [2,] 74.51815 74.73486 75.16461 75.56737 76.79215 78.25512 79.44110 80.39075
## [3,] 70.87547 70.29807 70.48626 71.28081 72.39707 73.55037 74.45606 75.29567
## [4,] 72.82301 73.52076 74.66121 75.82785 77.25688 78.97567 80.25456 80.68209
## [5,] 77.28805 76.38872 75.67667 75.47412 76.00601 77.07004 78.34762 79.52015
## [6,] 80.22689 79.15368 78.07639 77.23018 76.85022 76.93353 77.24666 77.67907
##         [,41]    [,42]    [,43]    [,44]    [,45]    [,46]    [,47]    [,48]
## [1,] 81.37632 80.96667 80.27018 79.61736 79.33872 79.76479 80.68953 81.59856
## [2,] 81.14473 81.74370 82.22834 82.63930 83.01724 83.40284 83.74790 83.98228
## [3,] 76.19797 77.02623 77.88023 78.75087 79.62902 80.63437 81.68147 82.45355
## [4,] 80.61293 79.80382 78.93231 78.67597 79.24299 79.91267 80.82915 81.91759
## [5,] 80.28737 80.70640 80.99475 81.36991 81.68418 81.96346 82.22544 82.54504
## [6,] 78.12019 78.68375 79.44774 80.24805 81.39089 82.81376 83.79949 84.32025
##         [,49]    [,50]    [,51]    [,52]    [,53]    [,54]    [,55]    [,56]
## [1,] 82.35983 82.84128 83.09234 83.26101 83.36316 83.41467 83.43139 83.42920
## [2,] 84.11366 84.14973 84.04338 83.77445 83.39109 82.94143 82.47363 82.03582
## [3,] 82.95696 83.40037 83.77056 84.05429 84.27791 84.47866 84.65972 84.82425
## [4,] 83.10318 84.21556 85.09263 85.67187 85.87461 85.72607 85.31959 84.74849
## [5,] 82.57831 82.63414 82.58329 82.44850 82.25245 82.01788 81.76748 81.52398
## [6,] 84.70330 85.20516 85.15499 84.76440 84.12581 83.33162 82.47423 81.64606
##         [,57]    [,58]    [,59]    [,60]    [,61]    [,62]    [,63]    [,64]
## [1,] 83.42396 83.43154 83.46782 83.54864 83.68989 83.90743 84.21713 84.67236
## [2,] 81.67616 81.44278 81.38383 81.47413 81.63391 81.82941 82.02688 82.19257
## [3,] 84.97542 85.11640 85.25037 85.38048 85.50991 85.64183 85.77941 85.92581
## [4,] 84.10611 83.48580 82.98087 82.68468 82.69055 83.02864 83.56769 84.15893
## [5,] 81.31007 81.14847 81.06190 81.07306 81.20466 81.47942 81.92004 82.58392
## [6,] 80.93951 80.44699 80.26090 80.42279 80.85741 81.48200 82.21379 82.97003
##         [,65]    [,66]    [,67]    [,68]    [,69]    [,70]    [,71]    [,72]
## [1,] 85.27968 85.98476 86.73325 87.47081 88.14310 88.69578 89.07451 89.16987
## [2,] 82.29204 82.49738 82.77561 83.11651 83.50982 83.94532 84.41276 84.90190
```

```
## [3,] 86.08420 86.25775 86.44963 86.66302 86.90107 87.16696 87.46386 87.80936
## [4,] 84.65355 85.16691 85.85839 86.64527 87.44483 88.17435 88.75112 89.19972
## [5,] 83.47560 84.53851 85.71608 86.95174 88.18894 89.37110 90.44165 91.34404
## [6,] 83.66795 84.45818 85.46159 86.55076 87.59831 88.47684 89.05894 89.27983
##         [,73]    [,74]    [,75]    [,76]    [,77]    [,78]    [,79]    [,80]
## [1,] 89.24150 89.39903 89.47479 89.55201 89.41871 89.08306 88.95853 89.17272
## [2,] 85.40250 85.90433 86.39713 86.87068 87.31473 87.71904 88.07337 88.41963
## [3,] 88.20535 88.63141 89.06714 89.49212 89.88594 90.22818 90.49845 90.73277
## [4,] 89.61742 90.00436 90.34588 90.64600 90.90872 91.13808 91.33808 91.47046
## [5,] 92.02168 92.46930 92.79870 93.12896 93.45577 93.70187 93.87687 93.99039
## [6,] 89.44954 89.64171 89.71112 89.76436 89.90803 90.09349 90.23861 90.39974
##         [,81]    [,82]    [,83]    [,84]    [,85]    [,86]    [,87]    [,88]
## [1,] 89.38645 89.67172 90.00443 90.36042 90.71557 91.04577 91.32686 91.53473
## [2,] 88.79012 89.16021 89.50528 89.80068 90.02179 90.21561 90.43125 90.64723
## [3,] 90.96900 91.18943 91.37636 91.51208 91.61332 91.71173 91.81928 91.89858
## [4,] 91.51764 91.51467 91.49659 91.49847 91.55534 91.67014 91.80271 91.92351
## [5,] 94.05202 94.07138 94.05809 94.02175 93.97198 93.90347 93.81026 93.70101
## [6,] 90.57096 90.71850 90.85693 91.00082 91.16474 91.31337 91.41595 91.49391
##         [,89]    [,90]    [,91]    [,92]    [,93]    [,94]    [,95]    [,96]
## [1,] 91.64524 91.67955 91.67823 91.64631 91.58883 91.51081 91.41727 91.31326
## [2,] 90.84204 90.99421 91.08223 91.09335 91.03463 90.91433 90.74073 90.52211
## [3,] 91.91223 91.83870 91.69472 91.50030 91.27548 91.04027 90.81469 90.61878
## [4,] 92.00300 92.07154 92.16776 92.27452 92.37467 92.45109 92.48662 92.46412
## [5,] 93.58438 93.46901 93.36356 93.25743 93.13479 92.99646 92.84324 92.67594
## [6,] 91.56870 91.64583 91.71688 91.78237 91.84285 91.89884 91.95090 91.99954
##         [,97]    [,98]    [,99]   [,100]   [,101]   [,102]   [,103]   [,104]
## [1,] 91.18085 91.00490 90.79451 90.55881 90.30694 90.04801 89.79115 89.54549
## [2,] 90.26671 89.98283 89.67873 89.36268 89.04294 88.72779 88.42551 88.14435
## [3,] 90.40869 90.13573 89.81262 89.45207 89.06679 88.66950 88.27293 87.88977
## [4,] 92.36646 92.20715 92.01379 91.79040 91.54099 91.26956 90.98013 90.67671
## [5,] 92.49537 92.30235 92.09768 91.88218 91.65666 91.42192 91.17878 90.92805
## [6,] 92.04530 92.08873 92.13035 92.17070 92.21031 92.24973 92.28947 92.33009
##        [,105]   [,106]   [,107]   [,108]   [,109]   [,110]   [,111]   [,112]
## [1,] 89.32015 89.12426 88.96694 88.85732 88.77852 88.70758 88.64449 88.58924
## [2,] 87.89259 87.67851 87.51036 87.39642 87.30587 87.20685 87.10607 87.01025
## [3,] 87.53276 87.21461 86.94802 86.74573 86.57861 86.41054 86.24372 86.08033
## [4,] 90.36331 90.04394 89.72261 89.40334 89.09012 88.78698 88.49792 88.22695
## [5,] 90.67054 90.40706 90.13843 89.86544 89.58892 89.30967 89.02850 88.74623
## [6,] 92.37212 92.41608 92.46252 92.51197 92.56496 92.62204 92.68373 92.75057
##        [,113]   [,114]   [,115]   [,116]   [,117]   [,118]   [,119]   [,120]
## [1,] 88.54182 88.50222 88.47044 88.44647 88.43029 88.42190 88.42128 88.42844
## [2,] 86.92610 86.86034 86.81967 86.80298 86.80364 86.82009 86.85081 86.89423
## [3,] 85.92256 85.77262 85.63269 85.50497 85.39165 85.29492 85.20673 85.11764
## [4,] 87.97808 87.75533 87.56271 87.38386 87.20108 87.01635 86.83166 86.64900
## [5,] 88.46366 88.18161 87.90089 87.62231 87.34667 87.07480 86.80749 86.54557
## [6,] 92.82309 92.90184 92.98734 93.08013 93.18075 93.28973 93.39250 93.47443
##        [,121]   [,122]   [,123]   [,124]   [,125]   [,126]   [,127]   [,128]
## [1,] 88.44336 88.46602       NA       NA       NA       NA       NA       NA
## [2,] 86.94884 87.01307       NA       NA       NA       NA       NA       NA
## [3,] 85.02772 84.93705 84.84572 84.75379 84.66136 84.56849 84.47528 84.38180
## [4,] 86.47034 86.29768 86.12928 85.96212 85.79623 85.63167 85.46846 85.30666
## [5,] 86.28983 86.04110 85.80019 85.56789 85.34503 85.13242 84.93086 84.74117
## [6,] 93.53593 93.57741 93.59929 93.60199 93.58592 93.55149 93.49913 93.42925
##        [,129]   [,130]   [,131]   [,132]   [,133]   [,134]   [,135]   [,136]
## [1,]       NA       NA       NA       NA       NA       NA       NA       NA
## [2,]       NA       NA       NA       NA       NA       NA       NA       NA
## [3,] 84.28813 84.19435 84.10054 84.00679 83.91316 83.81974 83.72661 83.63386
## [4,] 85.14631 84.98745 84.83011 84.67435 84.52020 84.36771 84.21691       NA
## [5,] 84.56416 84.40063 84.25140 84.11729 83.99909 83.89763 83.81370       NA
```

```
## [6,] 93.34226 93.23857 93.11862 92.98280 92.83154 92.66525 92.48435 92.28925
##         [,137]   [,138]   [,139]   [,140]   [,141]   [,142]   [,143]  [,144]
## [1,]        NA       NA       NA       NA       NA       NA       NA      NA
## [2,]        NA       NA       NA       NA       NA       NA       NA      NA
## [3,] 83.54155 83.44977 83.35860 83.26812 83.17841 83.08955 83.00162 82.9147
## [4,]        NA       NA       NA       NA       NA       NA       NA      NA
## [5,]        NA       NA       NA       NA       NA       NA       NA      NA
## [6,] 92.08036 91.85812 91.62292 91.37518 91.11533 90.84377 90.56092 90.2672
##         [,145]   [,146]   [,147]   [,148]   [,149]   [,150]   [,151]   [,152]
## [1,]        NA       NA       NA       NA       NA       NA       NA       NA
## [2,]        NA       NA       NA       NA       NA       NA       NA       NA
## [3,] 82.82887 82.74421 82.66079 82.57871 82.49803 82.41884 82.34122 82.26524
## [4,]        NA       NA       NA       NA       NA       NA       NA       NA
## [5,]        NA       NA       NA       NA       NA       NA       NA       NA
## [6,] 89.96303 89.64881 89.32496 88.99191 88.65006 88.29983 87.94163 87.57589
##         [,153]   [,154]   [,155]
## [1,]        NA       NA       NA
## [2,]        NA       NA       NA
## [3,] 82.19100 82.11856 82.04801
## [4,]        NA       NA       NA
## [5,]        NA       NA       NA
## [6,] 87.20302 86.82343 86.43753
```

```
plot(ozone$ozone,ozone$temperature, pch = 19, cex =0.5)
for(i in 1:10){lines(1:155, ll[i,], col='grey',lwd=2)}
lines(1:155, apply(ll,2,mean),col='red', lwd=2)
```



- In the plot, each grey line represents the LOESS fit with one resampled dataset. - The grey lines have a lot of curviness and possibly overfit the dataset variability. - The red line is the average of all the fitted grey curves.

There is a proof that shows the **bagging estimate** will always have **lower variability** but **similar bias** to each of the **individual model fits** from which it has been created.

Bagging in caret

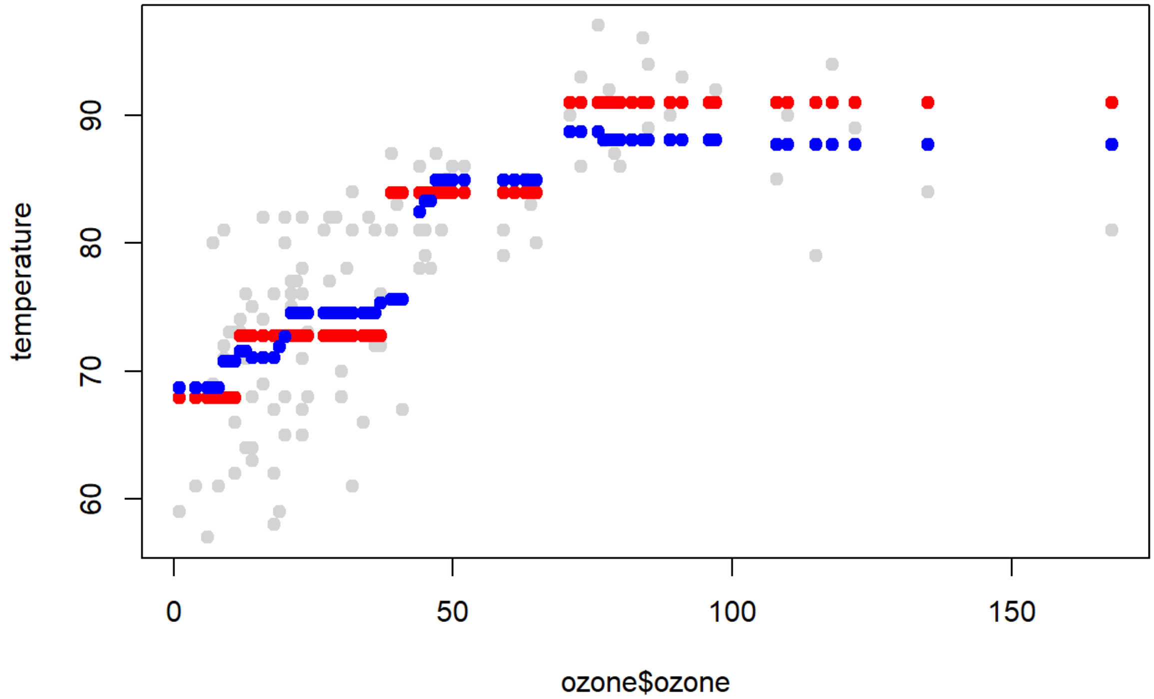
There are some models that automatically perform bagging for you –> in the `train` function consider these `method` options: - `bagEarth` - `treebag` - `bagFDA`

Alternatively, you can bag any model you choose using the `bag` function.

Example of custom bagging

```
predictors = data.frame(ozone=ozone$ozone) ## predictor/s
temperature = ozone$temperature ## outcome variable
treebag<-bag(predictors, temperature, B=10, ## B = n of subsamples to take

             # wit bagControl we specify how to fit the model:
             bagControl = bagControl(fit = ctreeBag$fit, ## function to be applied to train the model usin
g each replicated subsample (the train function of the caret package can be called here as well)
                                     predict = ctreeBag$pred, ## call the predict function from the train
 model
                                     aggregate = ctreeBag$aggregate) ## it specifies how to put all predic
tions together (e.g. take average)
             )
plot(ozone$ozone, temperature, col = 'lightgrey', pch =19)
points(ozone$ozone, predict(treebag$fits[[1]]$fit, predictors), pch =19, col='red')
points(ozone$ozone, predict(treebag, predictors), pch =19, col='blue')
```



- The grey dots represent actual data points - The red dots represent the fit from a single conditional regression tree (it does not capture the data trend very well) - The blue dots represent the fit from the bag regression (i.e. average of the single regression predictions) –> data trends are captured better!

Parts of bagging

```
ctreeBag$fit
```

```
## function (x, y, ...)
## {
##     loadNamespace("party")
##     data <- as.data.frame(x, stringsAsFactors = TRUE)
##     data$y <- y
##     party::ctree(y ~ ., data = data)
## }
## <bytecode: 0x00000000278cf198>
## <environment: namespace:caret>
```

- The `ctreeBag$fit` function takes the predictor df (`x`) and the predictor (`y`) and calls the function `ctree` to train a **conditional regression tree** on the dataset. The model fit is returned.

```
ctreeBag$pred
```

```
## function (object, x)
## {
##     if (!is.data.frame(x))
##         x <- as.data.frame(x, stringsAsFactors = TRUE)
##     obsLevels <- levels(object@data@get("response")[, 1])
##     if (!is.null(obsLevels)) {
##         rawProbs <- party::treeresponse(object, x)
##         probMatrix <- matrix(unlist(rawProbs), ncol = length(obsLevels),
##             byrow = TRUE)
##         out <- data.frame(probMatrix)
##         colnames(out) <- obsLevels
##         rownames(out) <- NULL
##     }
##     else out <- unlist(party::treeresponse(object, x))
##     out
## }
## <bytecode: 0x00000000278cfbe0>
## <environment: namespace:caret>
```

- The `ctreeBag$pred` function takes as input the object created by the `ctreeBag$fit` . The new outcomes from the object and data input are calculated using `treeresponse` .
- These predicted values are then used by the next aggregation function and pooled together (in the example, the `median` prediction is used as pooled outcome prediction):

```
ctreeBag$aggregate
```

```
## function (x, type = "class")
## {
##     if (is.matrix(x[[1]]) | is.data.frame(x[[1]])) {
##         pooled <- x[[1]] & NA
##         classes <- colnames(pooled)
##         for (i in 1:ncol(pooled)) {
##             tmp <- lapply(x, function(y, col) y[, col], col = i)
##             tmp <- do.call("rbind", tmp)
##             pooled[, i] <- apply(tmp, 2, median)
##         }
##         if (type == "class") {
##             out <- factor(classes[apply(pooled, 1, which.max)],
##                 levels = classes)
##         }
##         else out <- as.data.frame(pooled, stringsAsFactors = TRUE)
##     }
##     else {
##         x <- matrix(unlist(x), ncol = length(x))
##         out <- apply(x, 1, median)
##     }
##     out
## }
## <bytecode: 0x00000000278cb4f0>
## <environment: namespace:caret>
```

- Bagging is most useful for nonlinear models.
- It is often used with trees. One of its extension is **random forests**.
- Several models use bagging in caret's `train` function.

https://stat.ethz.ch/education/semesters/FS_2008/CompStat/sk-ch8.pdf (https://stat.ethz.ch/education/semesters/FS_2008/CompStat/sk-ch8.pdf)

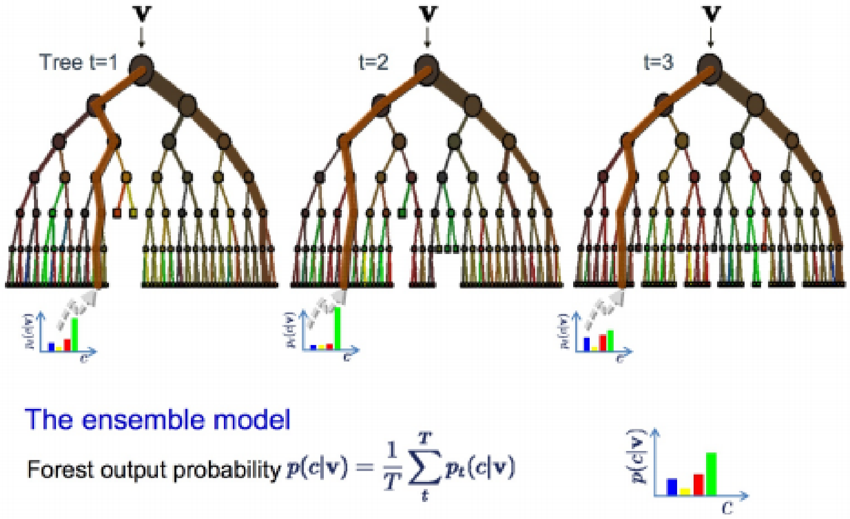
Week 3.3: Random forests

They can be defined as an extension to bagging for classification and regression trees.

1. **Booststrap samples**
2. **At each split, boostrap variables** (thus, only a subset of the variables is considered at each potential split. In this way, a diverse set of potential trees can be built)
3. **Grow multiple** (large number) **trees** and vote (or take the average of predictions)

Pros: - **Accuracy**

Cons: - **Low speed** (slow as it needs to build a large number of trees) - ***Poor interpretability** (as the model relies on **bootstrapped samples** with **bootstrapped nodes**) - **Overfitting** (hard to understand which trees are leading to overfitting –> very important to use **crossvalidation**)



- The basic idea is that you build a large number of trees (T), each created from a bootstrap sample.
- At each node of one tree we allow a subsample of the variables to potentially contribute to the splits.
- The same observation (**v** in the figure above) will end up at possibly a different leaf at the bottom of each tree, corresponding to a particular prediction.
- All the different predictions from all the prediction trees are then averaged (**ensemble model**) to create the final class (c) prediction for the observation (v).

Iris data example

```
data(iris);library(ggplot2);library(caret);library(randomForest)
inTrain<-createDataPartition(y=iris$Species, p=0.7,list=F)
training<-iris[inTrain,]
testing<-iris[-inTrain,]
summary(training)
```

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width   
##  Min.   :4.300   Min.   :2.000   Min.   :1.100   Min.   :0.100  
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.500   1st Qu.:0.300  
##  Median :5.800   Median :3.000   Median :4.400   Median :1.300  
##  Mean   :5.823   Mean   :3.055   Mean   :3.749   Mean   :1.209  
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800  
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500  
##        Species  
##  setosa    :35  
##  versicolor:35  
##  virginica :35  
##                 
##                 
## 
```

```
### RANDOM FORESTS
modFit<-train(data=training,Species ~ .,method='rf',prox=T)# 'rf' = random forest,
# prox=T is to allow to visualise class centers (see below)
modFit
```

```
## Random Forest
##
## 105 samples
##   4 predictor
##   3 classes: 'setosa', 'versicolor', 'virginica'
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 105, 105, 105, 105, 105, 105, ...
## Resampling results across tuning parameters:
##
##   mtry  Accuracy   Kappa
##   2     0.9533341  0.929197
##   3     0.9533341  0.929197
##   4     0.9533341  0.929197
##
## Accuracy was used to select the optimal model using the largest value.
## The final value used for the model was mtry = 2.
```

A bunch of different tuning parameters were tried during training –> `mtry` = Number of variables randomly sampled as candidates at each split

```
getTree(modFit$finalModel,k=2) # k specifies which tree we want to examine (number 2 in this case)
```

```
##    left daughter right daughter split var split point status prediction
## 1              2              3         1        5.45      1          0
## 2              4              5         4        0.75      1          0
## 3              6              7         4        1.75      1          0
## 4              0              0         0        0.00     -1          1
## 5              8              9         3        4.20      1          0
## 6             10             11         4        0.60      1          0
## 7             12             13         2        3.15      1          0
## 8              0              0         0        0.00     -1          2
## 9              0              0         0        0.00     -1          3
## 10             0              0         0        0.00     -1          1
## 11             0              0         0        0.00     -1          2
## 12             0              0         0        0.00     -1          3
## 13            14             15         3        5.05      1          0
## 14             0              0         0        0.00     -1          2
## 15             0              0         0        0.00     -1          3
```

- Each row corresponds to a split
- “var split” is the variable that was used for the split
- “split point” is the value of the variable used for the split

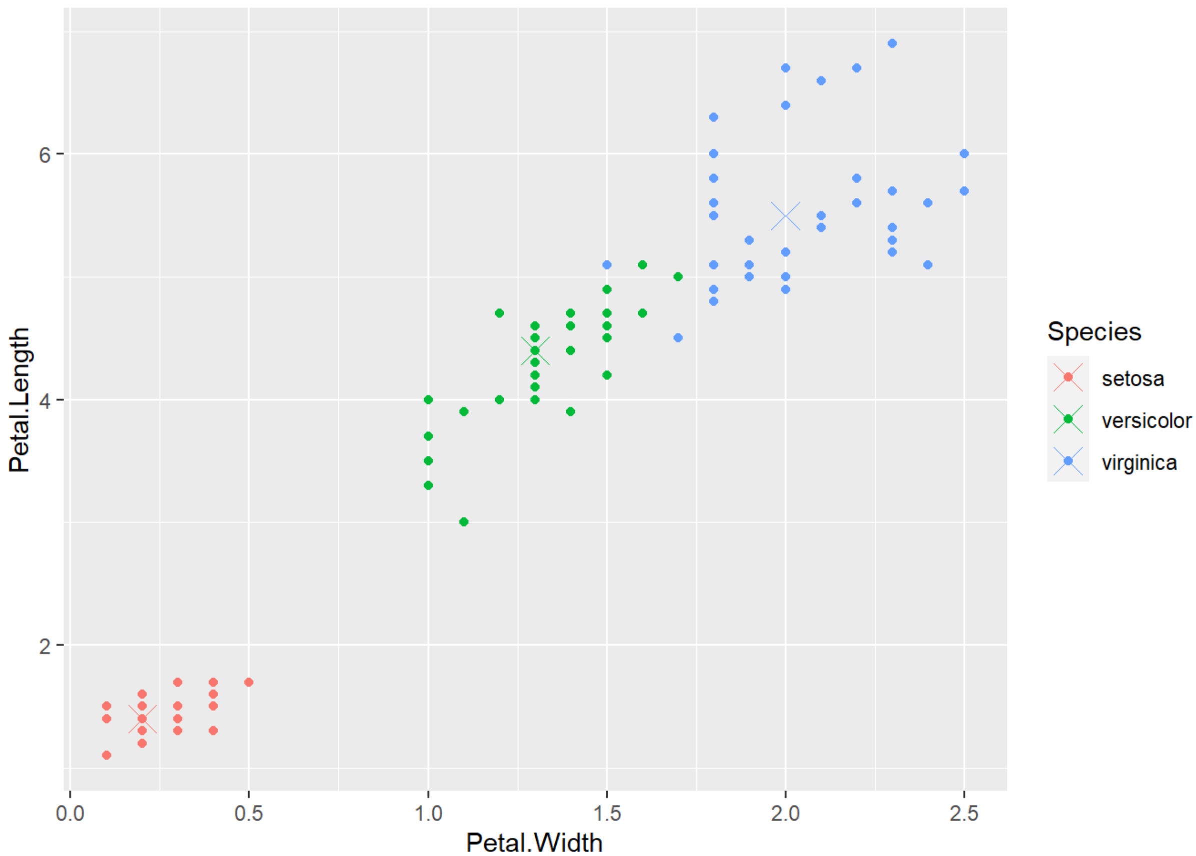
Class ‘centers’

- They specify the centers of the class predictions:

```
library(randomForest);library(ggplot2)
#training[,c(3,4)] denote the two predictors to represent in the plot:
irisP<-classCenter(training[,c(3,4)],training$Species, modFit$finalModel$prox) ##prox is used here
irisP<-as.data.frame(irisP);irisP$Species<-rownames(irisP)
irisP
```

```
##            Petal.Length Petal.Width    Species
## setosa              1.4         0.2     setosa
## versicolor          4.4         1.3 versicolor
## virginica           5.5         2.0  virginica
```

```
p<-qplot(Petal.Width,Petal.Length,col=Species,data=training)
p + geom_point(aes(x=Petal.Width,y=Petal.Length,col=Species), size=5,shape=4,data=irisP)
```

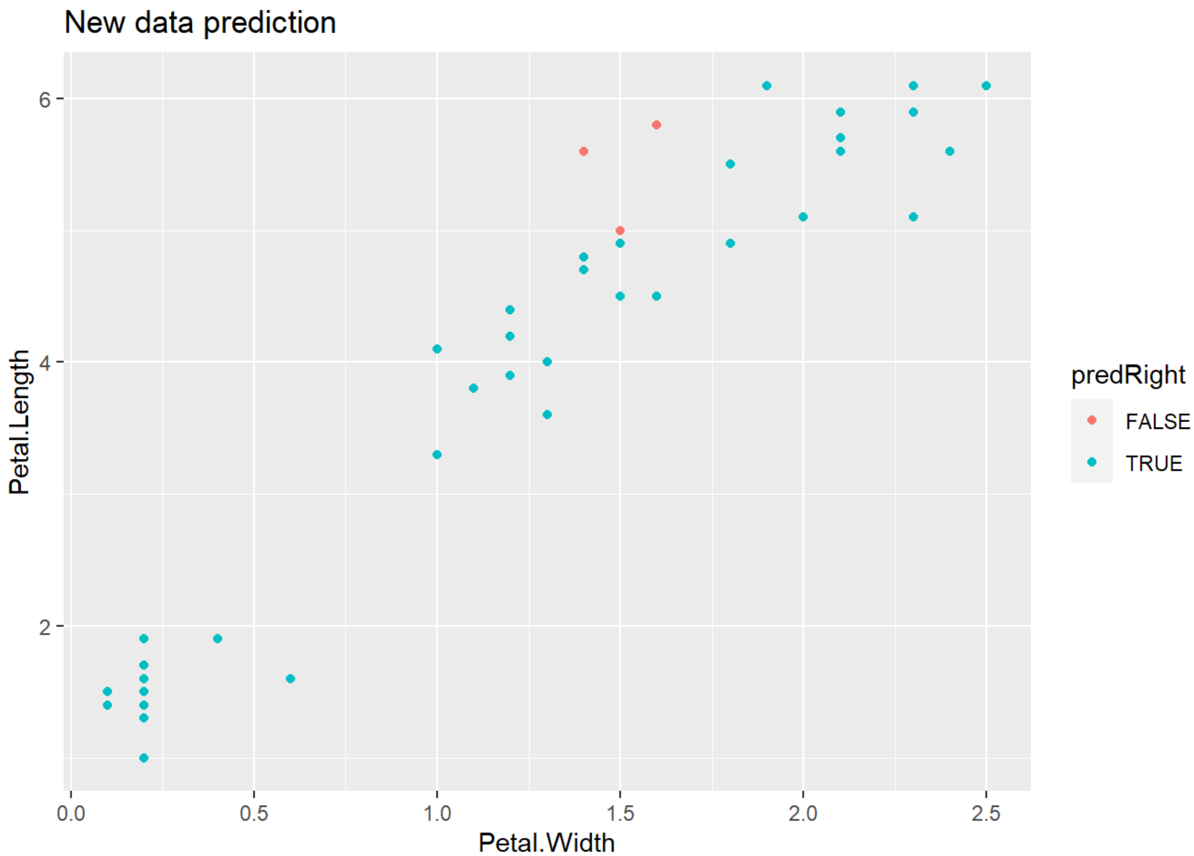


Predicting new values

```
library(caret)
pred<-predict(modFit,testing)
testing$predRight<-pred == testing$Species
table(pred,testing$Species)
```

```
## 
## pred         setosa versicolor virginica
##   setosa         15          0         0
##   versicolor      0         15         3
##   virginica       0          0        12
```

```
qplot(Petal.Width,Petal.Length,colour=predRight, data=testing, main = 'New data prediction')
```




```
confusionMatrix(pred,testing$Species)
```

```
## Confusion Matrix and Statistics
##
##             Reference
## Prediction   setosa versicolor virginica
##   setosa         15          0         0
##   versicolor      0         15         3
##   virginica       0          0        12
##
## Overall Statistics
##
##                Accuracy : 0.9333
##                  95% CI : (0.8173, 0.986)
##     No Information Rate : 0.3333
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.9
##
##  Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##                      Class: setosa Class: versicolor Class: virginica
## Sensitivity                 1.0000            1.0000           0.8000
## Specificity                 1.0000            0.9000           1.0000
## Pos Pred Value              1.0000            0.8333           1.0000
## Neg Pred Value              1.0000            1.0000           0.9091
## Prevalence                  0.3333            0.3333           0.3333
## Detection Rate              0.3333            0.3333           0.2667
## Detection Prevalence        0.3333            0.4000           0.2667
## Balanced Accuracy           1.0000            0.9500           0.9000
```

- The two misclassified samples were at the border between 2 classes.

Notes and further resources

- Random forests are usually one of the top performing algorithms along with boosting in prediction contests.
- Random forests are **difficult to interpret** but **very accurate**.
- Care should be taken to avoid overfitting, using cross-validation with the rf library (see `rfcv` function: https://cran.r-project.org/web/packages/randomForest/randomForest.pdf (https://cran.r-project.org/web/packages/randomForest/randomForest.pdf)) or indirectly via caret.
- Website of creator of random forests: https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm (https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm)

Week 3.4: Boosting

Key ideas

1. Take lots of (possibly) weak predictors
2. Weight them (to take advantage of their strength) and add them up
3. Get a stronger predictor

Basic idea behind boosting

1. Start with a **set of classifiers** $h_1, \ldots, h_k$ They are usually from the same class of classifiers (examples: all possible trees, all possible regression models, all possible cutoffs).

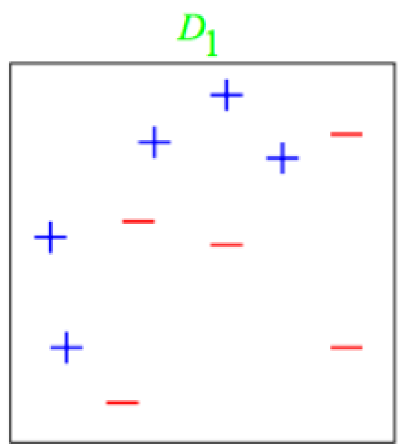
2. Create a classifier that **combines classification functions**: $f(x) = \mathrm{sgn}\left(\sum_{\mathrm{t=1}}^{\mathrm{T}} \mathrm{a_t h_t(x)}\right)$ (a_t is a weight, $h_t(x)$ is a classifier)

- Goal is to **minimise error** (on training set)
- **Iterative**: select one h at each step
- Calculate **weights based on errors**
- Upweight missed classifications and select next h

The most famous boosting algorithm is **Ada boosting**.

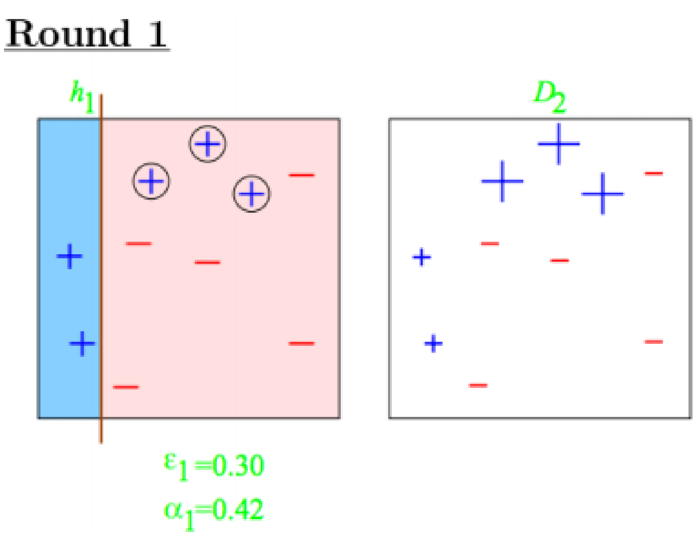
Simple example

https://alliance.seas.upenn.edu/~cis520/dynamic/2020/wiki/index.php?n=Lectures.Boosting (https://alliance.seas.upenn.edu/~cis520/dynamic/2020/wiki/index.php?n=Lectures.Boosting)



- We are trying to separate the +/blue category form the -/red category using 2 dimensions/predictors (X- and Y-axes)

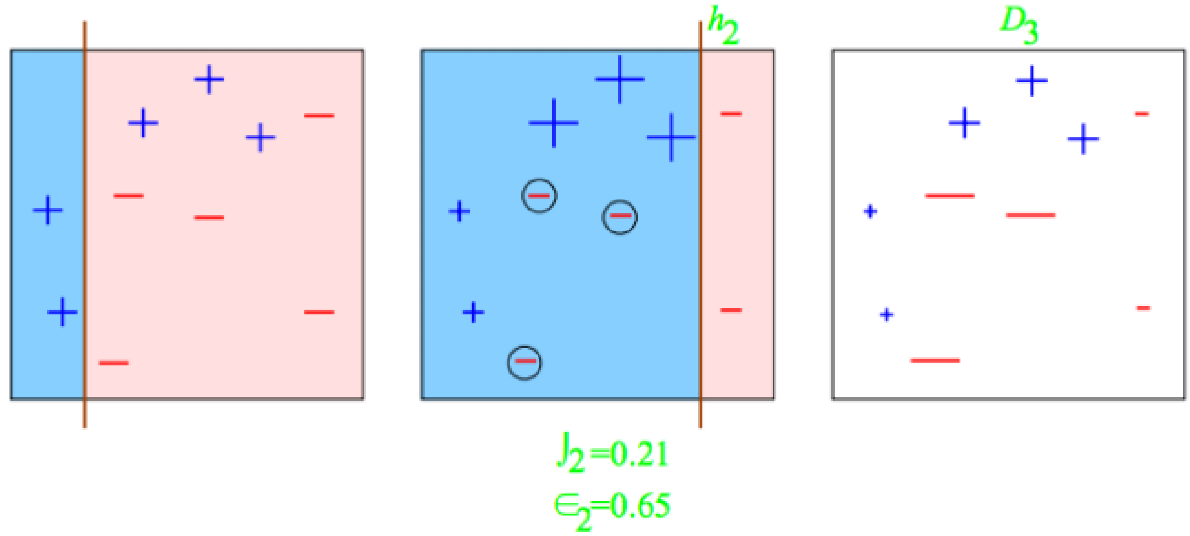
Round 1: adaboost



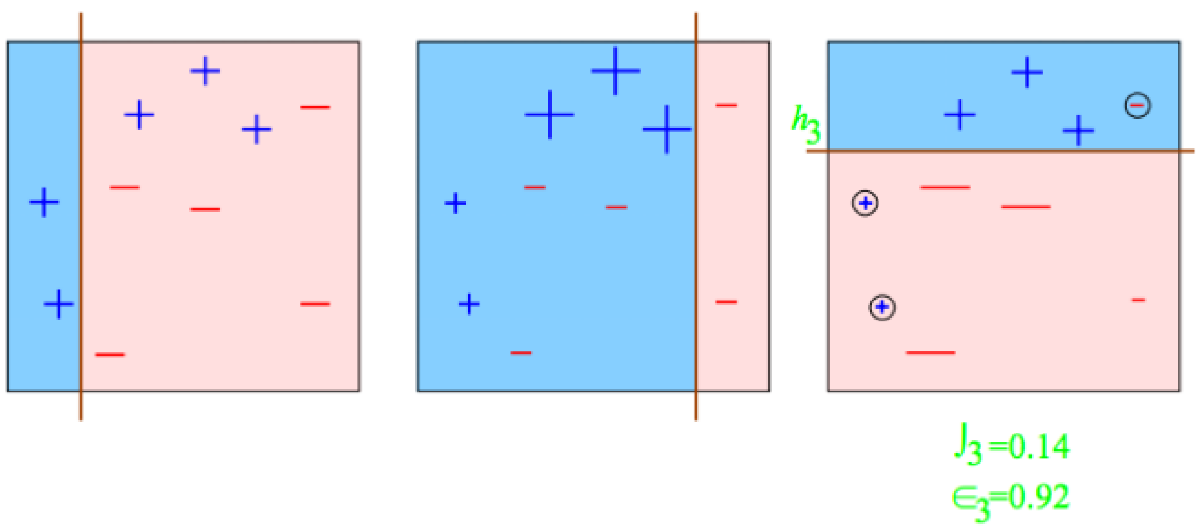
- The first classifier considers anything to the left of the red line as a +/blue and anything to the right as a -/red.
- 3 points were misclassified
- ϵ_1 is the error rate
- The weight of the misclassified samples is increased (**upweighting**), so they get more importance for building the next classifier.

Round 2 & 3: adaboost

Round 2



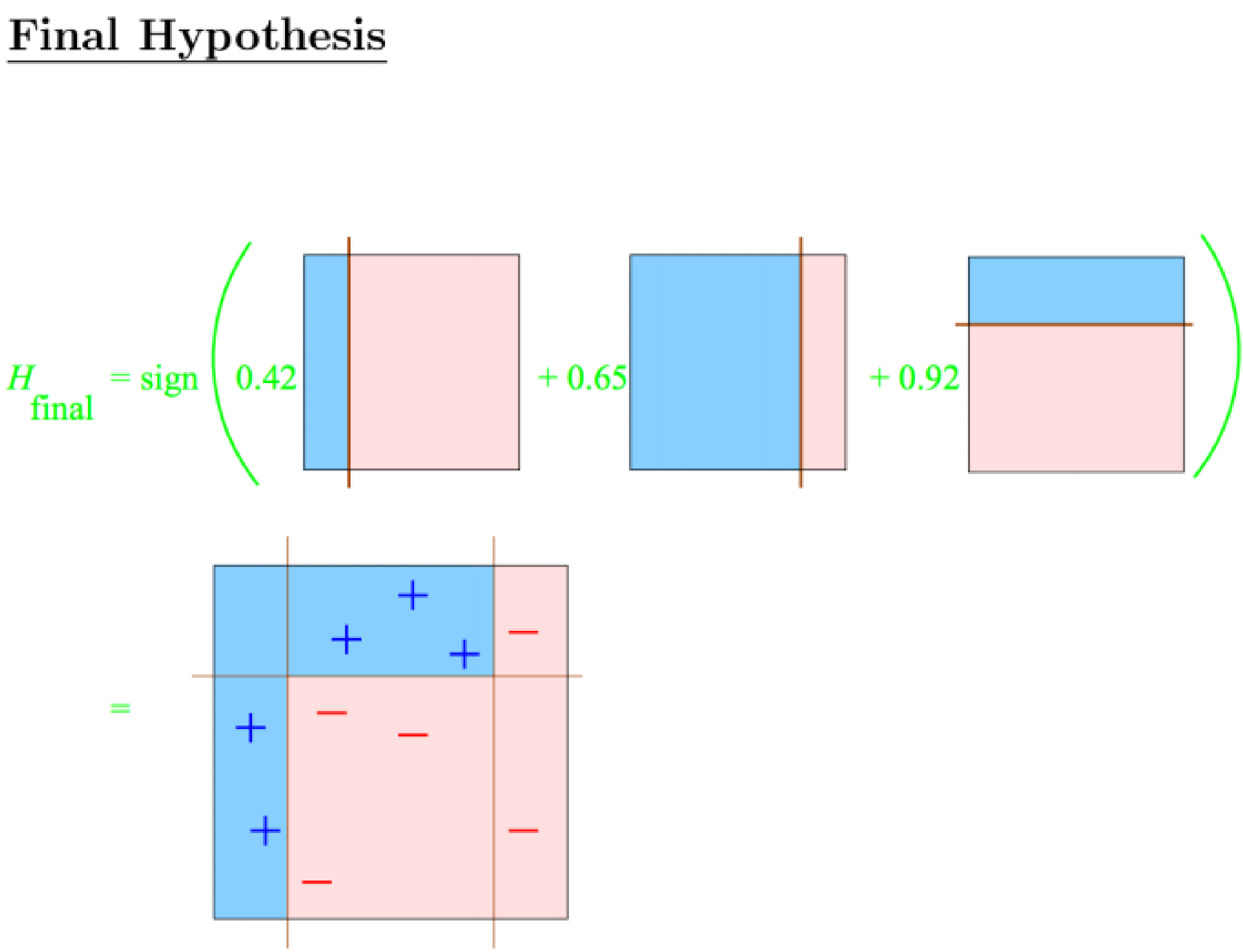
Round 3



- The 2nd classifier predicts as +/blue everything to the left of the red line (3 samples misclassified)
- The 3rd classifier predicts as +/blue everything above the red line (3 samples misclassified); this classifier tries to correctly predict those sampels that were misclassified in round 1 and 2.

Completed classifier: adaboost

- We take the classifiers built in the previous rounds, weight them and add them up



- The resulting classifier is much more accurate! It is the result of the combination of multiple naive simple classifiers (straight lines in this example).

Boosting in R

- Boosting can be used with any subset of (weak) classifiers
- One large subclass is **gradient boosting**
- R has multiple boosting libraries. Differences include the choice of basic classification functions and combination rules:
  - `gbm` : boosting with trees
  - `mboost` : model based boosting
  - `ada` : statistical boosting based on additive logistic regression
  - `gamboost` : for boosting generalised additive models
- Most of these are available in the caret package

Wage example

```
library(ISLR); data(Wage);library(ggplot2); library(caret)
Wage <-subset(Wage,select=-c(logwage)) ##wage is the var want to predict
summary(Wage)
```

```
##       year           age                        maritl           race     
##  Min.   :2003   Min.   :18.00   1. Never Married: 648   1. White:2480  
##  1st Qu.:2004   1st Qu.:33.75   2. Married      :2074   2. Black: 293  
##  Median :2006   Median :42.00   3. Widowed      :  19   3. Asian: 190  
##  Mean   :2006   Mean   :42.41   4. Divorced     : 204   4. Other:  37  
##  3rd Qu.:2008   3rd Qu.:51.00   5. Separated    :  55                  
##  Max.   :2009   Max.   :80.00                                          
##                                                                        
##               education                     region               jobclass   
##  1. < HS Grad      :268   2. Middle Atlantic   :3000   1. Industrial :1544  
##  2. HS Grad        :971   1. New England       :   0   2. Information:1456  
##  3. Some College   :650   3. East North Central:   0                        
##  4. College Grad   :685   4. West North Central:   0                        
##  5. Advanced Degree:426   5. South Atlantic    :   0                        
##                           6. East South Central:   0                        
##                           (Other)              :   0                        
##             health      health_ins        wage       
##  1. <=Good     : 858   1. Yes:2083   Min.   : 20.09  
##  2. >=Very Good:2142   2. No : 917   1st Qu.: 85.38  
##                                      Median :104.92  
##                                      Mean   :111.70  
##                                      3rd Qu.:128.68  
##                                      Max.   :318.34  
## 
```

```
inTrain<- createDataPartition(y=Wage$wage,p=0.7, list=F)
training<-Wage[inTrain,]; testing<-Wage[-inTrain,]
```

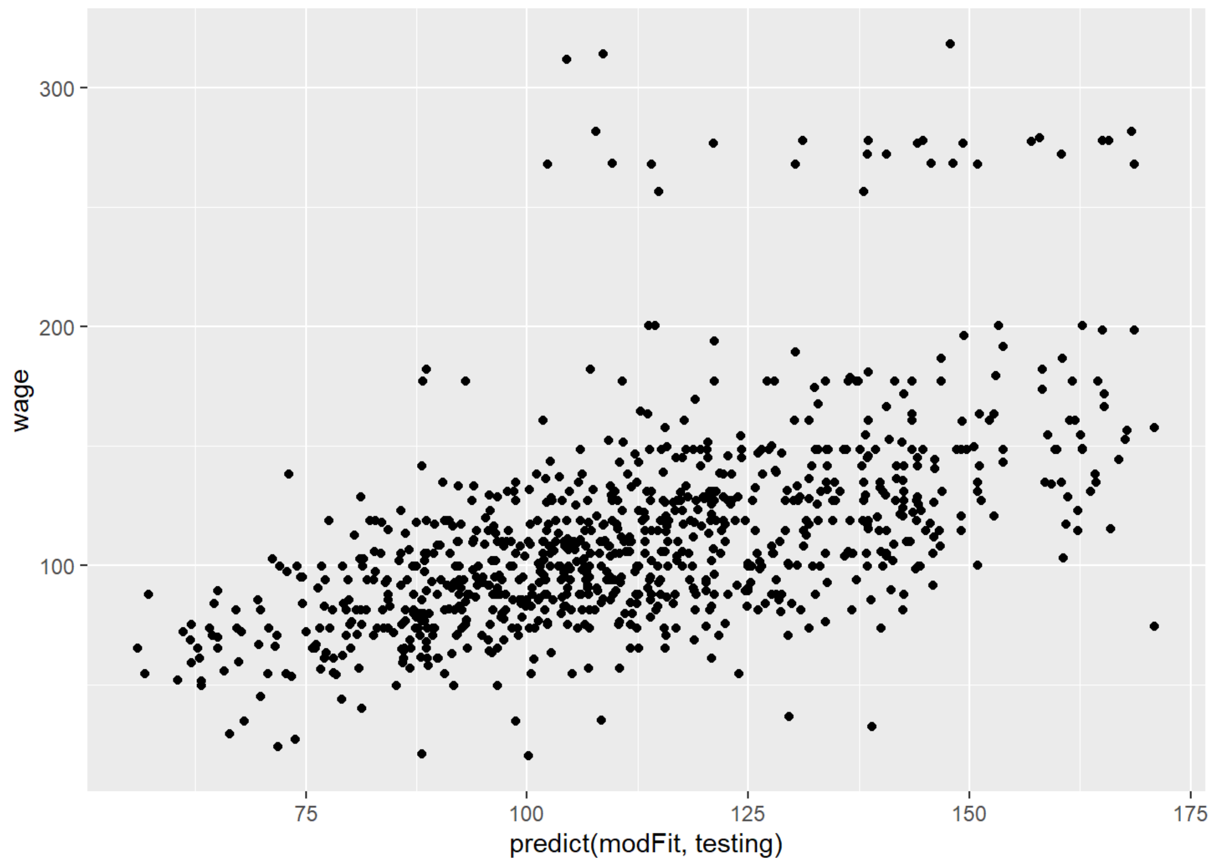
Fit the boosting model

```
modFit<-train(data= training, wage~., method = 'gbm', verbose = F) ##'gbm' to boost with trees// verbose = F, otherwise too much output is produced
modFit
```

```
## Stochastic Gradient Boosting
##
## 2102 samples
##    9 predictor
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 2102, 2102, 2102, 2102, 2102, 2102, ...
## Resampling results across tuning parameters:
##
##   interaction.depth  n.trees  RMSE      Rsquared   MAE
##   1                   50      33.76863  0.3218465  23.16669
##   1                  100      33.25502  0.3325699  22.79275
##   1                  150      33.15231  0.3363811  22.76930
##   2                   50      33.12180  0.3385880  22.65678
##   2                  100      32.99161  0.3428714  22.63563
##   2                  150      33.05016  0.3416926  22.71659
##   3                   50      33.02860  0.3412073  22.62071
##   3                  100      33.10695  0.3400929  22.79829
##   3                  150      33.26547  0.3358659  23.00757
##
## Tuning parameter 'shrinkage' was held constant at a value of 0.1
##
## Tuning parameter 'n.minobsinnode' was held constant at a value of 10
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were n.trees = 100, interaction.depth =
##  2, shrinkage = 0.1 and n.minobsinnode = 10.
```

Plot the results

```
qplot(predict(modFit,testing),wage,data=testing)
```



Notes and further reading

- A nice tutorial for boosting: https://www.cc.gatech.edu/~thad/6601-gradAI-fall2013/boosting.pdf (https://www.cc.gatech.edu/~thad/6601-gradAI-fall2013/boosting.pdf)
- Boosting, random forests and model ensembling are the most common tools that win Kaggle and other prediction contests

Week 3.5: Model-based prediction

Basic ideas

1. **Assume** the data follow a **probabilistic model**
2. Use the **Bayes' theorem** to identify optimal classifiers based on the probabilistic model previously identified.

Pros: - Can take advantage of structure of the data (e.g. if the data follow a specific probabilistic distribution) - May be computationally convenient - Reasonably accurate on real problems

Cons: - Make **additional assumptions** about the data - When the model is incorrect you may get reduced accuracy

Model based approach

1. Our goal is to build a **parametric model** (a model based on probability distributions) for **conditional distribution** $P(Y = k|X = x)$ (k is a specific y-outcome class)
2. A typical approach is to apply **Bayes theorem** (http://en.wikipedia.org/wiki/Bayes'_theorem):

$$Pr(Y = k|X = x) = \frac{Pr(X = x|Y = k)Pr(Y = k)}{\sum_{\ell=1}^{K} Pr(X = x|Y = \ell)Pr(Y = \ell)}$$

$$Pr(Y=k|X=x) = \frac{f_k(x)\pi_k}{\sum_{\ell=1}^{K} f_\ell(x)\pi_\ell}$$

3. Typically **prior probabilities** π_k are set in advance (using the data).
4. A common choice for $f_k(x) = \frac{1}{\sigma_k\sqrt{2\pi}} e^{-\frac{(x-\mu_k)^2}{\sigma_k^2}}$, a **Gaussian distribution** (might be multivariate gaussian distribution if there are multiple x variables). This is the **parametric model** for the distribution of the features (x) given the outcome class (y=k) we assume.
5. **Estimate the parameters** (μ_k,σ_k^2) from the data.
6. Classify to the **class with the highest value** of $P(Y=k|X=x)$

Classifying using the model

A range of models use this approach

- **Linear discriminant analysis** assumes $f_k(x)$ is **multivariate Gaussian** (i.e. the x features have a multivariate Gaussian distribution within each outcome class) with the **same covariance matrix** for each outcome class. This model draws lines through the data (called the **covariate space**)
- **Quadratic discriminant analysis** assumes $f_k(x)$ is **multivariate Gaussian** with **different covariances** (different covariance matrices are allowed for each outcome class). It draws quadratic curves through the data as opposed to lines.
- **Model based prediction** (http://www.stat.washington.edu/mclust/) assumes **more complicated versions for the covariance matrix**
- **Naive Bayes** assumes **independence between features** for model building. This might not be true in reality, although Naive Bayes could still allow to build a useful predictor.

http://statweb.stanford.edu/~tibs/ElemStatLearn/ (http://statweb.stanford.edu/~tibs/ElemStatLearn/)

Why linear discriminant analysis?

$$log\frac{Pr(Y=k|X=x)}{Pr(Y=j|X=x)}$$

- The log is a montonous function, so when the ratio $\frac{Pr(Y=k|X=x)}{Pr(Y=j|X=x)}$ increases, also $log\frac{Pr(Y=k|X=x)}{Pr(Y=j|X=x)}$ increases.
- Using Bayes theorem, you can rewrite the first expression as:

$$= log\frac{f_k(x)}{f_j(x)} + log\frac{\pi_k}{\pi_j}$$

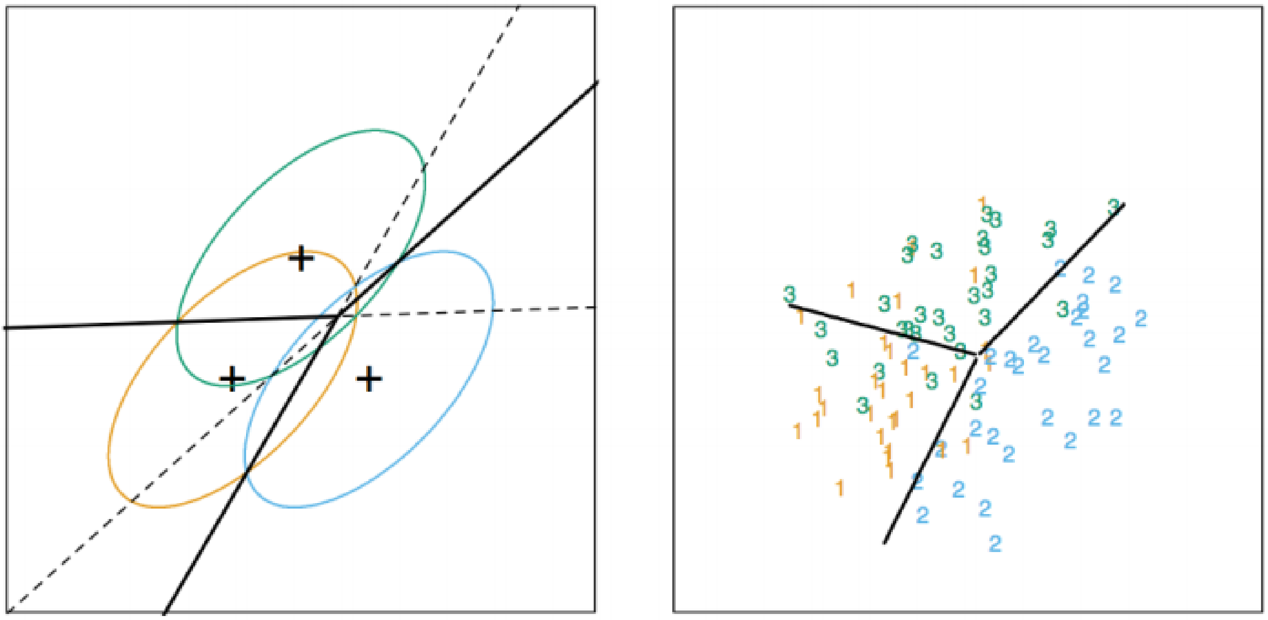
- The term $log\frac{f_k(x)}{f_j(x)}$ can be further transformed into two components, the first one depends on the parameters of the Gaussian/normal distributions (i.e. μ_k and μ_j) …

$$= log\frac{\pi_k}{\pi_j} - \frac{1}{2}(\mu_k+\mu_j)^T\Sigma^{-1}(\mu_k+\mu_j)$$

…the second one is a **linear term** (for this reason the model draws lines through the data. A variable will have a higher probability to belong to a specific class if it is on one side of the line, and to belong to another class if it is located on the other side of the line):

$$+x^T\Sigma^{-1}(\mu_k-\mu_j)$$

Decision boundaries



- We are trying to classify samples into 3 categories using 2 predictors (dimensions of the Cartesian axes)
- The Gaussian distributions are visible as ovals in the left panel
- Lines are drawn in the data space when the probability switches from being higher for a class to another class
- The model basically fits Gaussian distributions to the data and uses them to draw lines, classifying the points according to their highest posterior probability

Discriminant function

$$\delta_k(x) = x^T \Sigma^{-1} \mu_k - \frac{1}{2} \mu_k \Sigma^{-1} \mu_k + log(\mu_k)$$

$\Sigma^{-1} \mu_k$ represent the inverse of the covraince matrix for class k (although it is the same for each class in linear discriminant analysis).

$x^T \Sigma^{-1} \mu_k$ is the linear term

- Decide on class based on $\hat{Y}(x) = argmax_k \delta_k(x)$ (i.e.the value ok k which makes the discriminant function bigger for a specific sample)
- We usually estimate parameters with **maximum likelihood**

Naive Bayes

Naive Bayes tries to further simplify the problem. Suppose we have many predictors, we would want to model: $P(Y = k|X_1, \dots, X_m)$

We could use **Bayes Theorem** to get:

$$P(Y = k|X_1, \dots, X_m) = \frac{\pi_k P(X_1, \dots, X_m|Y = k)}{\sum_{\ell=1}^{K} P(X_1, \dots, X_m|Y = k)\pi_\ell}$$

It can be claimed that $P(Y = k|X_1, \dots, X_m)$ is **proportional** to the numerator of Bayes therom (the term in the denominator is just a constant for all the different probabilities):

$$\propto \pi_k P(X_1, \dots, X_m|Y = k)$$

This can be re-written (breaking down **conditional probabilities** until you get one term for every feature X):

$$P(X_1, \dots, X_m, Y = k) = \pi_k P(X_1|Y = k) P(X_2, \dots, X_m|X_1, Y = k)$$

$$= \pi_k P(X_1|Y = k)P(X_2|X_1, Y = k)P(X_3, \ldots, X_m|X_1, X_2, Y = k)$$

$$= \pi_k P(X_1|Y = k)P(X_2|X_1, Y = k) \ldots P(X_m|X_1 \ldots, X_{m-1}, Y = k)$$

We could make an **assumption** (**if all** Xs **were independent**) to write this (where the conditioning is dropped down):

$$\approx \pi_k P(X_1|Y = k)P(X_2|Y = k) \ldots P(X_m|, Y = k)$$

- Although this assumption is **naive**, it works particularly well in a number of applications (e.g. when you have a very large of binary/categorical features, for example in text/document classification applications)

Example: Iris Data

```
data(iris); library(ggplot2)
names(iris)
```

```
## [1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

```
table(iris$Species)
```

```
##
##     setosa versicolor  virginica
##         50         50         50
```

Create training and test sets

```
library(caret)
inTrain<-createDataPartition(y=iris$Species, p=0.7,list = F)
training<-iris[inTrain,]
testing<-iris[-inTrain,]
dim(training); dim(testing)
```

```
## [1] 105   5
```

```
## [1] 45  5
```

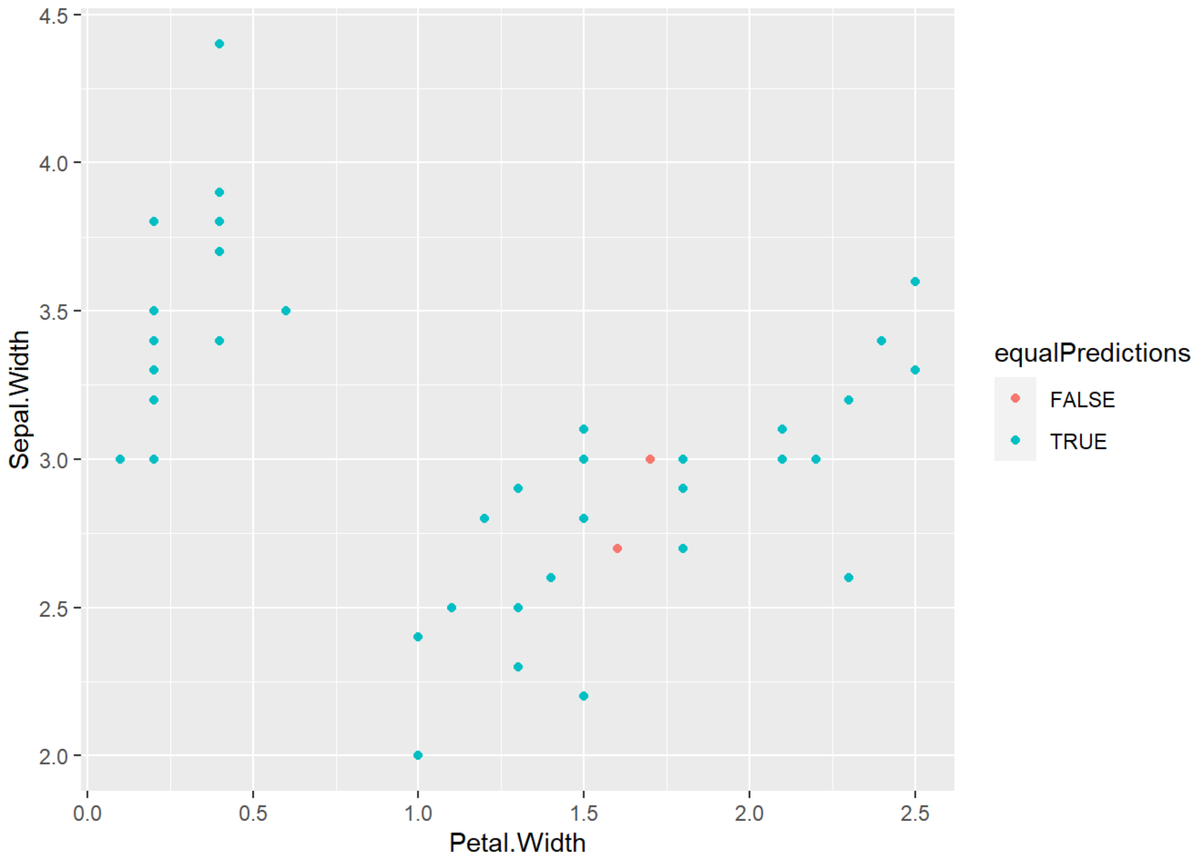
Build predictions

```
modLDA<-train(data=training, Species~., method='lda')
modNB<-train(data=training, Species~., method='nb')
pdLDA<- predict(modLDA, testing)
pdNB<- predict(modNB, testing)
table(pdLDA,pdNB)
```

```
##             pdNB
## pdLDA        setosa versicolor virginica
##   setosa         15          0         0
##   versicolor      0         13         2
##   virginica       0          1        14
```

Comparison of results

```
equalPredictions <- (pdLDA == pdNB)
qplot(Petal.Width,Sepal.Width,colour=equalPredictions,data=testing)
```



Notes and further reading

- Introduction to statistical learning (http://www-bcf.usc.edu/~gareth/ISL/)
- Elements of Statistical Learning (http://www-stat.stanford.edu/~tibs/ElemStatLearn/)
- Model based clustering (http://www.stat.washington.edu/raftery/Research/PDF/fraley2002.pdf)
- Linear Discriminant Analysis (http://en.wikipedia.org/wiki/Linear_discriminant_analysis)
- Quadratic Discriminant Analysis (http://en.wikipedia.org/wiki/Quadratic_classifier)