UWEO StatR201 Lecture 8

Classification 1; Data Manipulation 1

Assaf Oron, February 2013

Tonight's Menu

We will go lighter on the statistics today, learning tools that are more intuitive - and doing more fundamental R work as well.

Introduction to Classification

- Conceptual Overview
- k Nearest Neighbors Classifier
- Trees (the CART Classifier)

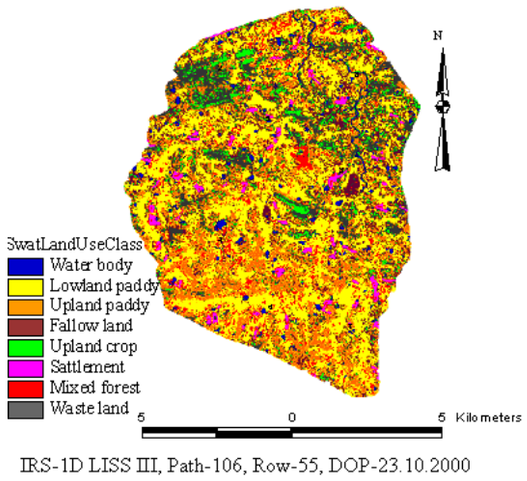
Then, the second half: **Introduction to R Data Manipulation.** Today, some basic ones:

- `match` and `merge`
- `stack/unstack/reshape` - and the `xtabs, as.data.frame.table` workaround

Looking ahead:

- I plan a similar division for the coming 2-3 lectures, running a classification short-course side by side with a data-manipulation short-course.
- We will definitely **not** learn all the classifiers that are out there; not even all the famous ones. Our shortlist will include methods that are intuitively pleasing or instructive, and/or have proven their mettle in a variety of settings - and in a number and complexity mix that we can feasibly digest with our limited time and resources.

Examples for Classification Problems



IRS-1D LISS III, Path-106, Row-55, DOP-23.10.2000

Land Use via Satellite image

Spam Filter

Ordered Classification: Wine Rating.

Plant and Animal Identification

Handwritten Digit Recognition

Biomarkers for medical diagnosis/prognosis

Classification vs. Regression

Whereas regression problems are perceived as computational tasks, solving problems like those shown in the previous slide, is more closely associated in our minds with **intelligence.**

Hence the name, **Machine Learning**, or in 1970s style - *"Artificial Intelligence."*

However, when push comes to shove, classification is quite similar to regression. The generic classification problem can be formulated as

$$\Pr(y = c) = f_c(\mathbf{X}), \quad c = 1, \ldots, C.$$

We want to find the f_c (or just as good and more commonly done nowadays, mimic their effect) **- as a means to the goal of correctly classifying y**

- With $C = 2$ classes, **logistic regression is a special case of the equation above.**
- With more classes, **multinomial regression** plays the same role. There are also analogous regression models for the ordered-classes case. -However, **generally speaking, logistic/multinomial regressions make poor classifiers.** There are exceptions, but in general these tools are far more suited to inference than to prediction.
- First, as usual, some more concepts and terms.

Classification vs. Regression

Here's that generic equation again:

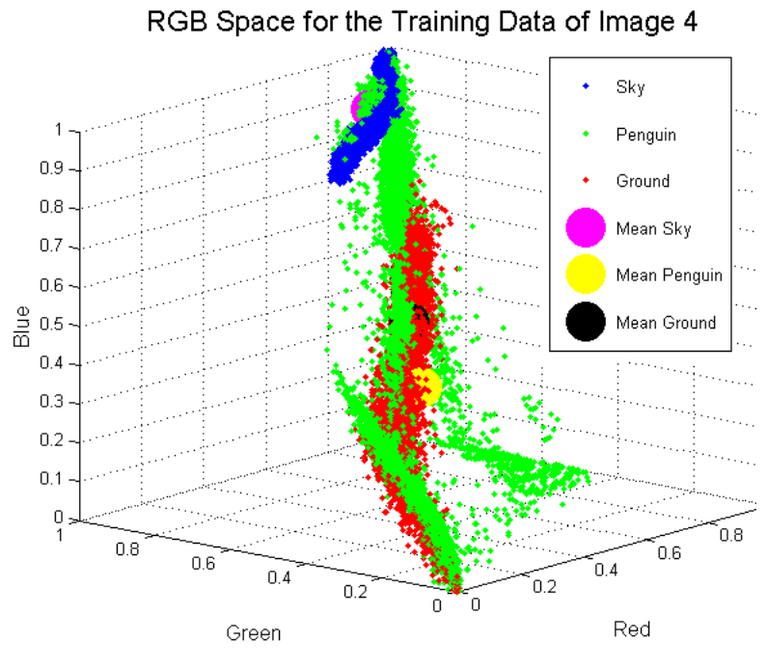
$$\Pr(y = c) = f_c(\mathbf{X}).$$

Terminology note: in classification, covariates are usually called **features.**

The equation is somewhat misleading: **for all but the simplest classification problems, the f_c are not some nice algebraic function, but a complicated partition of the feature space.**

Or, equivalently, a (probabilistic) mapping from this partition onto the classes.

If the classification is *separable* in prinicple, i.e., the classes can be separated completely (and predicted perfectly) once the exact mapping is known - then $\Pr(y = c)$ on the LHS turns into 1's and 0's, and the problem is deterministic.



In the more realistic case of some noise or overlap between classes in feature space, it is impossible to prevent a nonzero amount of classification error. That minimal amount is known as **the Bayes error rate:** the expected error rate, given perfect knowledge of all the f_c.

Classification Error/Loss Metrics

As we've seen in the last 2 lectures, there are unlimited ways to score and compare predictive performance. While with regression, the various common error-loss functions don't make a huge difference - **with classification they might matter a lot. Hence, it is important to tailor the error loss to the application context and needs.**

The simpler kinds of error metrics are $C \times C$ matrices, specifying the penalty (or **loss**) for classifying each class c as c':

$$L = \begin{pmatrix} 0 & L_{12} & \dots & L_{1C} \\ L_{21} & 0 & \dots & L_{2C} \\ & & \dots & \\ L_{C1} & L_{C2} & \dots & 0 \end{pmatrix}$$

Assume the true values are in rows, classifications in columns. Here are some common error-matrix forms:

- The simplest and most widely used: **0-1 Loss**. In matrix form (which is redundant in this case), this would be a matrix of 1's everywhere except the diagonal. In other words, lose a point for each missed point and **the overall loss equals the error rate.**
- Use the real-world price tag of each loss. For example, in public health specify the cost of a **"false negative"** (failing to diagnose a true disease) v. a **"false positive"** (diagonsing healthy as "diseased"). In general, real-cost loss matrices will not be symmetric.

Classification Error/Loss Metrics

- In a similar vein, even in the absence of a known price tag, one can impose a larger penalty on missing the less-common classes. This discourages of fallacy of *"persistence forecasting"*, e.g., always predicting a rainy day in Seattle this time of year.
- A common weight function is the **the Deviance or (Information-)Entropy Loss,**

$$L_{cc'} = -p_c \log(p_c), \quad c \neq c`$$

where p_c is the frequency of class c.

- The entropy-loss matrix will have identical values in each row's non-diagonal cells. For example, in a binary problem when the two classes have a true prevalence ratio of 9:1, the entropy loss for missing the rare class will be about 2.4 times higher.
- For **ordered categories**, it usually makes sense to specify a loss that is a function of the error magnitude (could be least-squares or another, context-dependent function).

Some classification methods are better at internally optimizing their performance to any loss function. Others, not so much. But at least to some degree, any loss can be implemented as a **tuning parameter** criterion, just like we did with predictive regression.

Questions? *Online* questions?

k Nearest Neighbors Classifier

In 1951, Berkeley statistician Evelyn Fix published (together with Joseph Hodges) a technical military-research report about a *"Nonparametric Discriminate…"*

Professor Fix passed away in 1965 at the age of 61, only two years after achieving full-professor rank. Her obituary talks more about her selfless generosity and *"unusual gift for cooking"*, than about her statistical achievements.

Evelyn Fix doesn't even have a Wikipedia entry to her name.

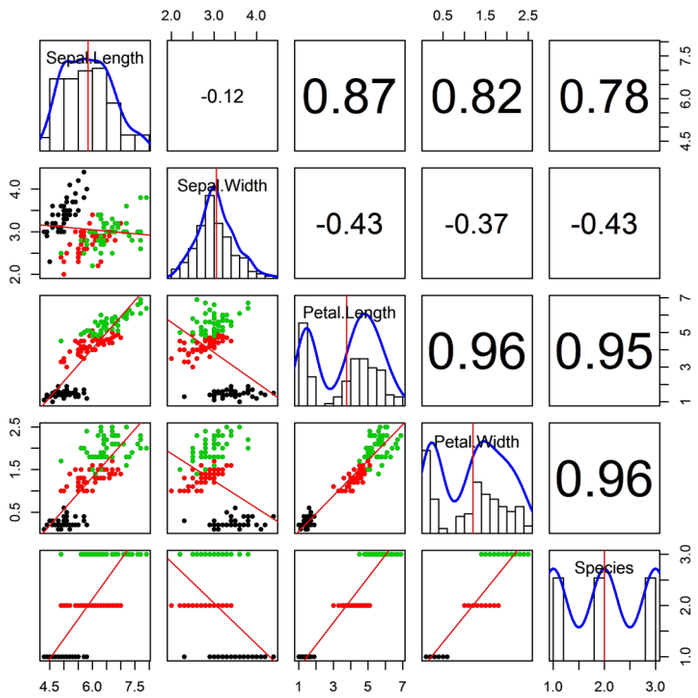
But over 60 years later, **Evelyn Fix's selfless and straightforward classification method still gives the most sophisticated, name-branded and pretentious algorithms a serious run for their money.** *(see the book, Sec. 13.3)*

The principle is as simple as can be:

- Like resembles like.
- Therefore, if you want to know a new data point's class, **ask the points whose features it resembles most!**
- All you need to do is:
- Define a distance metric in p-dimensional feature space;
- Calculate the distance from the new point to all(?) training points;
- **Classify the point to the most common class among its k nearest neighbors.**

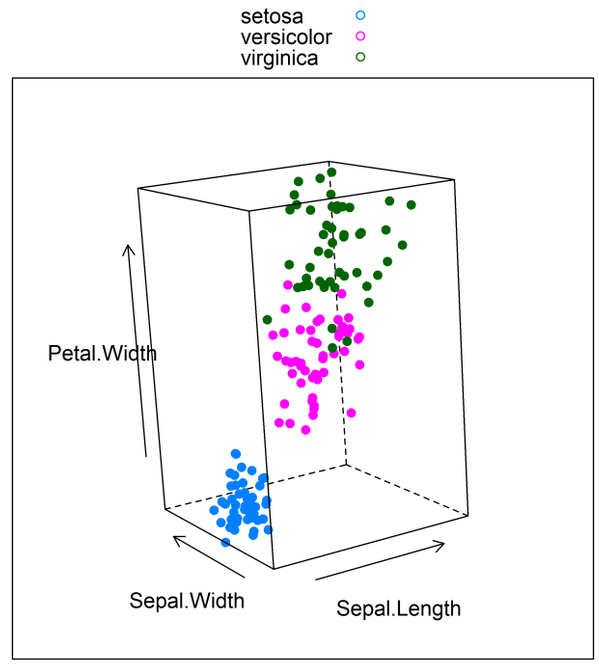
The `iris` Dataset

```
library(class)
pairsPlus(iris, col = iris$Species)
```



`iris`, one of the earliest classification examples, was used by R.A. Fisher to demonstrate his (underwhelming) LDA classifier. It has 3 species with 50 specimens each.

```
cloud(Petal.Width ~ Sepal.Length * Sepal.Width, data = iris, pc
    aspect = c(1, 1.5), screen = list(x = -80, z = 10, y = 30),
```



The data were collected by Edgar Anderson in northern Quebec. Using the available features one species is clearly separable, but the other two might somewhat overlap.

Note: `cloud` is part of `lattice`.

knning iris

```
testid = c(sample(1:50, size = 15), sample(51:100, size = 15), sample(101:150,
    size = 15))
iristrain = iris[-testid, ]
table(iristrain$Species)
```

```
##
##     setosa versicolor  virginica
##         35         35         35
```

```
trainclass = matrix(NA, nrow = 105, ncol = 8)
for (a in seq(1:8)) trainclass[, a] = knn.cv(iristrain[, 1:4], iristrain$Species,
    k = 2 * a - 1)
rbind(seq(1, 15, 2), apply(trainclass, 2, function(x, ref) length(x[x != ref]),
    ref = as.numeric(iristrain$Species)))
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]    1    3    5    7    9   11   13   15
## [2,]    7    6    5    5    4    5    5    5
```

`knn.cv` performs leave-one-out CV. The output above shows the CV error for odd values of k between 1 and 15 (top row of the output).

The problem almost seems too easy. The few errors come from the overlap region. But perhaps we were being too simplistic?

- One of the petal features might be redundant
- We didn't **scale** the features (*why would that matter?*)

knning iris

```
trainclass2 = matrix(NA, nrow = 105, ncol = 8)
for (a in seq(1:8)) trainclass2[, a] = knn.cv(scale(iristrain[, c(1:2, 4)]),
    iristrain$Species, k = 2 * a - 1)
rbind(seq(1, 15, 2), apply(trainclass2, 2, function(x, ref) length(x[x != ref]),
    ref = as.numeric(iristrain$Species)))
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## [1,]    1    3    5    7    9   11   13   15
## [2,]    9    8   10    8    8   11   12   10
```

…seems like the fourth feature was useful after all. Also, scaling doesn't seem to matter much (check it out!)

Of course, we are doing CV in order to determine k for the test set. These runs suggest an optimum somewhere around $k = 7$ or 9.

- **Run it on your machine, see what you get!**
- **Now check out performance on the test set** (can you figure out the syntax? The command is just `knn`)

By the way: in a model-selection analogy, which "model" is bigger: KNN with large k, or KNN with small k?

k Nearest Neighbors Pros vs. Cons

Pros:

- Genuinely **nonparametric:** no model or assumption necessary (beyond *"like resembles like"*)
- Excellent in mapping out irregular class boundaries and *"islands"* in feature space
- In fact, **its principle is a perfect match to the concept of feature-space partitioning!**
- **Robust:** Minor variations in the input data cannot - by definition - have a large impact on performance.
- KNN is the oldest and simplest **voting method.** As we shall see briefly (and you shall see even more if you become active in this field), voting methods are all the rage in classification world. It took them 50+ years to catch onto Evelyn Fix's cooking secrets, but they finally got there ;)
- The voting also provides a natural assessment of the classification's **confidence level.**
- Did I say "simple"?

Cons:

- **Storage and computation:** for each new data point, one might need to calculate all n distances.
- If classes are of unequal sizes, `knn` tends to over-classify to the more prevalent classes (*why?*)
- Like many other 'black box' methods, `knn` doesn't play very nice with categorical data
- Less flexible than most methods Questions? *Online* questions?

Classification Trees, Take 1

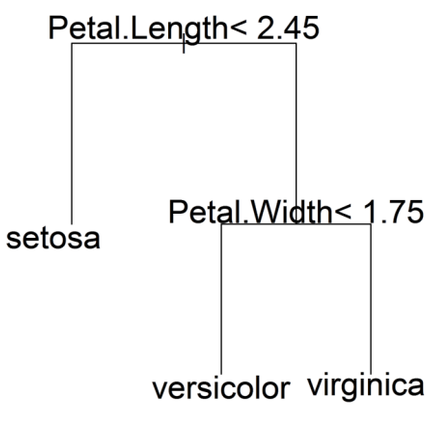
Intuitively, classification trees are even easier than KNN. Trees are close to how the human mind approaches many classification problems - like a game of “20 Questions” or “Guess Who”.

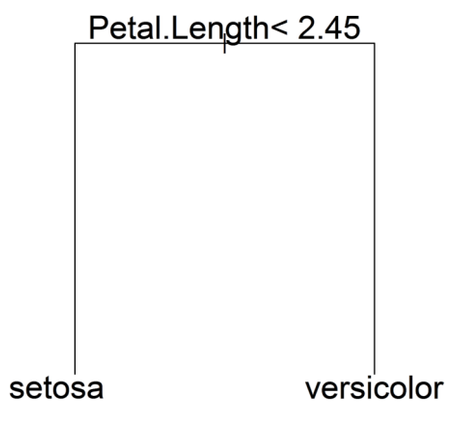
- A tree classifier starts from the entire datset
- At each point, a logical condition on some feature(s) **splits** the data, usually into two.
- In principle, with enough questions you can split the dataset into pure **terminal nodes** - achieving zero (training) error.
- In practice, one uses a **pruned tree.**

Classification Trees, Take 1

```
library(rpart)
iclass1 = rpart(Species ~ ., data = iristrain)
plot(iclass1, margin = 0.25)
text(iclass1, cex = 1.5)
```

```
options(width = 60)
iclass2 = rpart(Species ~ ., data = iristrain, control = rpart.control
plot(iclass2, margin = 0.25)
text(iclass2, cex = 1.5)
```





I had to work hard to make the 2 trees different in this toy example - the problem is just too easy…

Classification Trees: Pruning and Tuning

The parameter `cp` used above is just a linear penalty coefficient on the number of terminal nodes $|T|$. The algorithm minimizes the overall mismatch function

$$Mismatch(T, C_p, \dots) = C_p|T| + \sum_{m=1}^{|T|} n_m Q_m,$$

where n_m is the sample size in terminal node m, and Q_m is the node mismatch score, which is itself controlled via two knobs:

- **The splitting index,** affecting what constitutes (in the algorithm's eyes) a more informative split. Common choices are the node's estimated entropy loss (shown a few slides back), or the Gini coefficient which produces a similar curve and is the `rpart` default. **Both have the effect of encouraging the algorithm to go after minority classes.**
- An optional, separate **loss matrix,** whose elements become coefficients multiplying the splitting index's terms. This might be useful when wishing to impose a real-life cost, or when classifying ordered categories.

CARTing The White-Wine Dataset

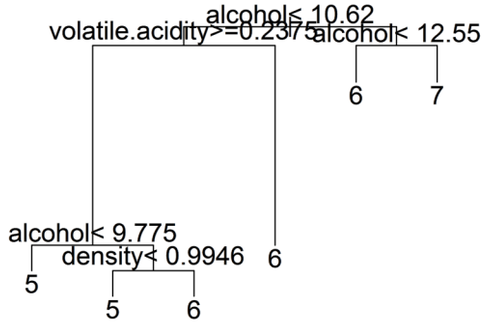
```
wine = read.csv("../Datasets/wineTrain.csv", as.is = T)
dim(wine)
```

```
## [1] 3676   12
```

```
table(wine$quality)
```

```
##
##    3    4    5    6    7    8    9
##   15  123 1093 1649  660  132    4
```

```
vinetree1 = rpart(quality ~ ., data = wine, method = "class")
plot(vinetree1, margin = 0.2)
text(vinetree1, cex = 1.2)
```



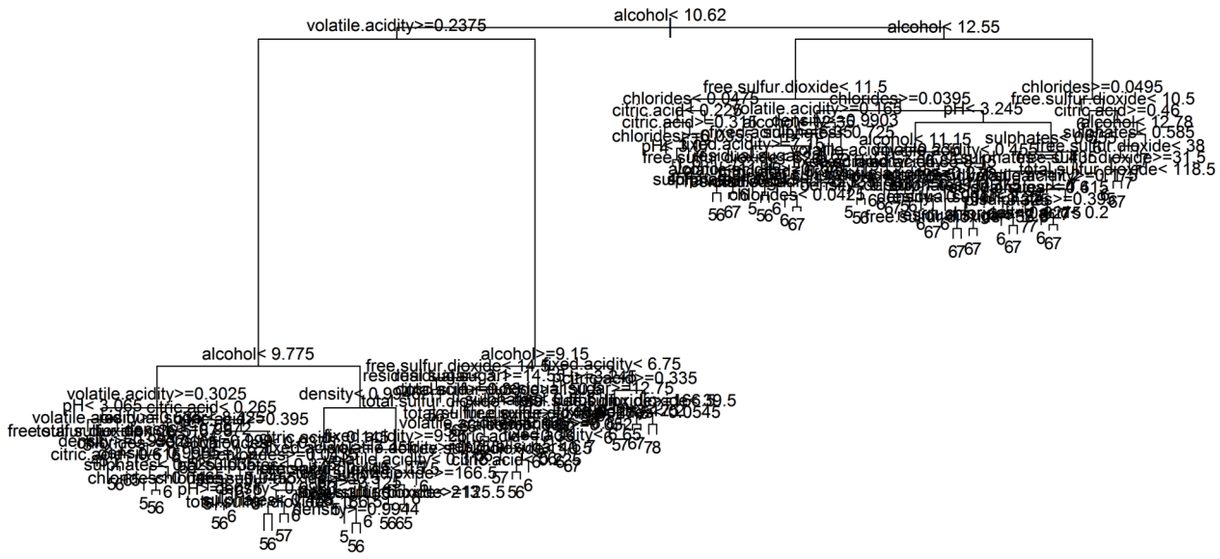
Note the need to specify `method`. Since y is numeric, `cart` would default to a regression tree; hence the tool's full name: Classification and Regression Trees or **CART.**

Now, take a moment to explore this White Wine tasting dataset.

CARTing The White-Wine Dataset

The tree in the previous slide is a bit modest for a $n \approx 3700, p = 11$ dataset. The default penalty is `cp=0.01`. Let's relax it a bit…

```
vinetree2 = rpart(quality ~ ., data = wine, method = "class",
    control = list(cp = 0.001))
plot(vinetree2, margin = 0.05)
text(vinetree2, cex = 0.8)
```



…now we're talking… **clearly, `cp` can function as CART's main tuning parameter.**

Retrieving `rpart` Performance: The Confusion Matrix

```
table(wine$quality, predict(vinetree1, type = "class"))
```

```
##
##        3    4    5    6    7    8    9
##   3    0    0    3   11    1    0    0
##   4    0    0   63   57    3    0    0
##   5    0    0  664  423    6    0    0
##   6    0    0  369 1188   92    0    0
##   7    0    0   40  485  135    0    0
##   8    0    0    4   96   32    0    0
##   9    0    0    0    3    1    0    0
```

```
table(wine$quality, predict(vinetree2, type = "class"))
```

```
##
##        3    4    5    6    7    8    9
##   3    0    0    5   10    0    0    0
##   4    0   32   53   37    1    0    0
##   5    0   17  752  289   35    0    0
##   6    0    0  181 1345  121    2    0
##   7    0    2   34  212  412    0    0
##   8    0    0    8   55   57   12    0
##   9    0    0    1    0    3    0    0
```

Overall, the predictions aren't something to write home about. And this is **in-sample, not CV!** Would they improve if we make the loss matrix more sensible?

```
wineloss = matrix(0, nrow = 7, ncol = 7)
wineloss = (row(wineloss) - col(wineloss))^2
wineloss
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7]
## [1,]    0    1    4    9   16   25   36
## [2,]    1    0    1    4    9   16   25
## [3,]    4    1    0    1    4    9   16
## [4,]    9    4    1    0    1    4    9
## [5,]   16    9    4    1    0    1    4
## [6,]   25   16    9    4    1    0    1
## [7,]   36   25   16    9    4    1    0
```

Now prepare CV indices, in a block-stratified manner.

```
wineStrata = cut(wine$quality, c(2, 4:7, 9) + 0.5)
wineid = rep(NA, length(wineStrata))
for (a in levels(wineStrata)) {
    aid = which(wineStrata == a)
    wineid[aid] = sample(rep(1:5, ceiling(length(aid)/5)), size
}
table(wineid, wine$quality)
```

```
##
## wineid   3   4   5   6   7   8   9
##      1   3  25 219 330 132  26   1
##      2   3  25 219 330 132  27   0
##      3   4  23 218 330 132  24   2
##      4   2  26 219 329 132  27   1
##      5   3  24 218 330 132  28   0
```

`rpart` Cross-Validation Function

```
options(width = 130)
source("../Code/rpartCV.r")
rpartCV
```

```
## function (formula, data, cvid, predtype = "class", cpvec, ...)
## {
##     cat(date(), "\n")
##     cvg = unique(cvid)
##     n = dim(data)[1]
##     cout = matrix(NA, nrow = n, ncol = length(cpvec))
##     for (b in 1:length(cpvec)) {
##         for (a in cvg) {
##             ktree = rpart(formula = formula, data = data[cvid !=
##                 a, ], control = rpart.control(cp = cpvec[b]),
##                 ...)
##             cout[cvid == a, b] = predict(ktree, newdata = data[cvid ==
##                 a, ], type = predtype)
##         }
##         cat(".")
##     }
##     cat(date(), "\n")
##     return(cout)
## }
```

```
vinecv1 = rpartCV(quality ~ ., data = wine, cvid = wineid, method = "class",
    cpvec = 0.1^seq(1, 5, 0.5))
```

```
## Wed Feb 27 21:32:54 2013
## .........Wed Feb 27 21:33:01 2013
```

```
vinecv2 = rpartCV(quality ~ ., data = wine, cvid = wineid, method = "class",
    cpvec = 0.1^seq(1, 5, 0.5), parms = list(split = "information",
        loss = wineloss))
```

```
## Wed Feb 27 21:33:01 2013
## .........Wed Feb 27 21:33:13 2013
```

Ok, Your Turn!

Run the previous chunks on your machine.

Now, we need to score the CV in order to choose the loss and `cp`. **Can you do it?** Hint: for one loss, you need to carefully multiply the confusion matrix by the the least-squares `wineloss`. For the other, just sum up the off-diagonals.

Another hint/warning: My CV function returns with the factor labels stripped. So the classifications are numbers in the range 1-7, vs. 3-9 in the original classes. In short: **add 2 to all the returned values - or convert the quality to factor levels. Also: try to run `knn` on this dataset!**

Advantages of CART:

- Like KNN, it is **nonparametric.**
- Simple and intuitive
- Can seamlessly combine different feature data types

Cons:

- **Performance.** Our white-wine example might be hard, but in general CART is not a major contender.
- **Lack of robustness.** to make matters worse, being a discrete threshold-based classifier, CART is sensitive to minor variations in the input.

Note: the CART version in the `tree` package uses a nearly-identical algorithm, but is said to have less functionality.

Next week we might look at spiced-up extensions of KNN and CART. **Questions?**

Data Manipulation 1.1.1: `match`

```
x = 1:8
y = sample(1:20, size = 8)
y
```

```
## [1] 10  5  7  1  9  4  3 17
```

```
rbind(x, match(x, y))
```

```
##   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
## x    1    2    3    4    5    6    7    8
##      4   NA    7    6    2   NA    3   NA
```

`match` returns the **positions** of the **first** occurrence of its **"Argument 1"** within its **"Argument 2".** If there's no match, `NA` is returned.

```
x = rep(1, 3)
y = c(y, 1)
match(x, y)
```

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

```
x = 1:8
y = rep(1, 3)
match(x, y)
```

```
## [1]  1 NA NA NA NA NA NA NA
```

See? Only the **first** instance in **"Argument 2"** is `match`ed (bottom example). But **all** instances of **"Argument 1"** are `match`ed (top example).

As always with computers: **don't argue with the behavior. Use it.**

Data Manipulation 1.1.1: `match`

`match` is an all-purpose tool, that can be easily (tho carefully) used for dataset merging. Here's an example from our air-pollution modeling endeavor.

```
options(width = 130)
wsy = read.csv("../Datasets/WSnoxScores.csv", as.is = T, row.names = 1)
wsx = read.csv("../Datasets/WinSalemGIScovariates.csv", as.is = T)
head(wsy, 3)
```

```
##               site   n LogAvgNox
## 370670022 370670022 295     2.779
## 371190041 371190041 305     2.982
## W001           W001 100     2.850
```

```
dim(wsx)
```

```
## [1] 125 126
```

```
head(wsx[, 1:10], 3)
```

```
##   native.id latitude longitude lambert.x lambert.y state.plane  county em.nox.s30000 log10.m.to.a1 log10.m.to.airp
## 1      W002    36.03    -80.34   1398612   -207653          NC Forsyth          2395         3.502           4.206
## 2      W001    36.11    -80.22   1407416   -196580          NC Forsyth         47002         3.037           3.386
## 3      W003    36.09    -80.27   1403519   -199338          NC Forsyth         46998         1.633           3.806
```

These MESA Air datasets have the rough-estimated long-term log-mean NOx levels (`wsy`), and > 100 GIS covariates deemed reasonably informative, via automated filtering from an initial set of some 700 (`wsx`).

They are then match-merged for predictive modeling (either Lasso+conSubsets, or PLS).

Data Manipulation 1.1.1: `match`

```
wsxy = cbind(wsx, wsy[match(wsx$native.id, wsy$site), -1])
dim(wsxy)
```

```
## [1] 125 128
```

```
summary(wsxy$LogAvgNox)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.78    2.50    2.79    2.80    3.12    4.52
```

Note the syntax: `match(destinationID,sourceID)`. The row order of the added columns must match that of the data frame they are being `matched` into. Note also the exclusion of the now-redundant ID column from `wsy`.

But… we really used only sites whose rough-averages were based on $n > 1$ observations. Let's repeat after excluding the $n = 1$ rows:

```
wsy = wsy[wsy$n > 1, ]
wsxy = cbind(wsx, wsy[match(wsx$native.id, wsy$site), -1])
summary(wsxy$LogAvgNox)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##    1.78    2.55    2.83    2.84    3.15    4.52      17
```

Neither Lasso nor PLS can do much training with a missing y variable. So these entire rows in `wsxy` are redundant. Of course, it only takes one more command line to clean it up. But there is also a more elegant - and more aptly named - solution.

Data Manipulation 1.1.2: `merge`

```
wsy = read.csv("../Datasets/WSnoxScores.csv", as.is = T, row.names = 1)
wsxy = merge(wsx, wsy[wsy$n > 1, ], by.x = "native.id", by.y = "site")
dim(wsxy)
```

```
## [1] 108 128
```

```
summary(wsxy$LogAvgNox)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    1.78    2.55    2.83    2.84    3.15    4.52
```

`merge` is analogous to the SQL `join`. The default behavior is like `inner join`, but this can be easily changed:

```
wsxy = merge(wsx, wsy[wsy$n > 1, ], by.x = "native.id", by.y = "site",
    all.x = TRUE)
dim(wsxy)
```

```
## [1] 125 128
```

```
summary(wsxy$LogAvgNox)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##    1.78    2.55    2.83    2.84    3.15    4.52      17
```

Of course, there is also an `all.y` argument. You can turn `merge` into an `outer join` by setting `all=TRUE`.

There's some creative work you can do with `merge`, especially within **the same data frame…**

Data Manipulation 1.1.2: `merge`

```
automatch = merge(autos[autos$cyl == 4, -c(9, 11)], autos[autos$cyl ==
    6, -c(9, 11)], by = c("name", "year"))
dim(automatch)
```

```
## [1]  0 16
```

Barring typos, there are no instances in our `autos` training set of the same car and year with both 4 and 6 cylinders. Let's make the merge a bit less tight…

```
options(width = 130)
automatch = merge(autos[autos$cyl == 4, -c(8, 10, 11)], autos[autos$cyl ==
    6, -c(8, 10, 11)], by = "name")
automatch
```

```
##                 name cyl.x volume.x hp.x weight.x accel.x year.x mpg.x cyl.y volume.y hp.y weight.y accel.y year.y mpg.y
## 1        amc concord     4      151   90     3003    20.1     80  24.3     6      232   90     3210    17.2     78  19.4
## 2      buick skylark     4      151   84     2635    16.4     81  26.6     6      231  105     3425    16.9     77  20.5
## 3 chevrolet citation     4      151   90     2678    16.5     80  28.0     6      173  110     2725    12.6     81  23.5
## 4         ford pinto     4      122   80     2451    16.5     74  26.0     6      171   97     2984    14.5     75  18.0
## 5         ford pinto     4       98   75     2046    19.0     71  25.0     6      171   97     2984    14.5     75  18.0
## 6         ford pinto     4      122   85     2310    18.5     73  19.0     6      171   97     2984    14.5     75  18.0
```

Note how the column names have changed.

Questions? *Online* questions? Explore this a bit… (6 cylinders vs. 8, perhaps?)

Which Reminds me: R Trick and Annoyance

```
grep("mazda", autos$name)
```

```
## [1] 181 183 235 248 260 268 281 282
```

```
autos$name = gsub("maxda", "mazda", autos$name)
grep("mazda", autos$name)
```

```
## [1]  88 181 183 235 248 260 268 281 282
```

That's the trick part. `gsub` has a very similar setup to `grep`. Which brings me to the annoyance. Let's try to do the same trick another way:

```
autos = read.csv("../Datasets/autoMPGtrain.csv")  # note: no 'as.is=T'
autos$name[grep("maxda", autos$name)]
```

```
## [1] maxda rx3
## 231 Levels: amc ambassador brougham ... vw rabbit custom
```

```
autos$name[autos$name == "maxda rx3"] = "mazda rx3"
```

```
## Warning: invalid factor level, NAs generated
```

What happened here?

- **The `read.csv` default converts any column containing any strings, to `factor`.** R makes it difficult to mess around with `factor`.
- `gsub` converts `factor` to `character` when needed; so using it, we would have masked this problem, with the side effect of `autos$name` turning into `character`. But the otherwise-identical direct assignment lands us in trouble.
- **Use `as.is` to control dataset read-in behavior.** `as.is` also accepts column indices instead of `TRUE/FALSE`, so you can mix and match (some automatic `factor` setting, some as `character`). There's also a `colClasses`

argument (see `?read.csv`).

Data Manipulation 1.3.1: `stack` and `unstack`

A common data-analysis need is to switch between

- a “long” or **“stacked”** layout (as in regression, with all covariate values spelled out in differnt columns)
- a **“wide”** matrix-like layout, for example to examine correlations or do PCA (e.g, in our air-pollution SVD example last week)

R has built in `stack,unstack` functions for this. But they’re not much. Here’s the example from `stack` help:

```
head(PlantGrowth, 2)
```

```
##   weight group
## 1   4.17  ctrl
## 2   5.58  ctrl
```

```
table(PlantGrowth$group)
```

```
##
## ctrl trt1 trt2
##   10   10   10
```

```
head(unstack(PlantGrowth))
```

```
##   ctrl trt1 trt2
## 1 4.17 4.81 6.31
## 2 5.58 4.17 5.12
## 3 5.18 4.41 5.54
## 4 6.11 3.59 5.50
## 5 4.50 5.87 5.37
## 6 4.61 3.83 5.29
```

(There’s a little cheat here: `PlantGrowth` has a built-in `formula` telling `unstack` what to do)

Data Manipulation 1.3.1: `stack` and `unstack`

But consider this:

```
dim(ChickWeight)
```

```
## [1] 578   4
```

```
head(ChickWeight, 2)
```

```
##   weight Time Chick Diet
## 1     42    0     1    1
## 2     51    2     1    1
```

```
head(unstack(ChickWeight, weight ~ Chick), 2)
```

```
## $`18`
## [1] 39 35
##
## $`16`
## [1] 41 45 49 51 57 51 54
```

If the vector lengths and/or covariate values don't match, `unstack` will return **a list.** Not very useful. Plus, limited capabilities. You might as well code a loop to create your "wide" matrix on your own. `stack` has it easier, so it doesn't return lists. OTOH, it only returns a two-column data frame.

There's a broader-functionality omnibus function called `reshape` which in principle can turn convert between "stacked" and "wide" formats, while carrying over any number of covariates in the "stacked" direction. That function should probably be presented under the *"R Annoyance"* category, as one of the least-friendly functions in R's default installation. I managed to get it to do what I wanted perhaps once or twice - after many tries - and have long forgotten the narrow path that eventually worked.

Instead, I found much nicer workarounds right under my nose.

Data Manipulation 1.3.2: The Stacked/Wide workarounds

```
mywide = xtabs(weight ~ Time + Chick, data = ChickWeight)
dim(mywide)
```

```
## [1] 12 50
```

```
head(mywide[, 1:10])
```

```
##      Chick
## Time 18 16 15 13  9 20 10  8 17 19
##    0  39 41 41 41 42 41 41 42 42 43
##    2  35 45 49 48 51 47 44 50 51 48
##    4   0 49 56 53 59 54 52 61 61 55
##    6   0 51 64 60 68 58 63 71 72 62
##    8   0 57 68 65 85 65 74 84 83 65
##    10  0 51 68 67 96 73 81 93 89 71
```

`xtabs` does cross-tabulation; analogous to Excel **PivotTable.**

Its default is to **sum** up all the y values whose `formula`'s RHS is identical, and **stick zeros in missing entries.** The good news are:

- It returns a nice wide matrix (or array! You can add `Diet` to the formula above);
- The missing-value and summation issues can be easily worked around, as the next slide shows.

Data Manipulation 1.3.2: The Stacked/Wide workarounds

```
options(width = 130)
safeXtab = function(formula, data, type = "sum") {
    # We first count how many obs. in each output array cell We
    # get this behavior by running the same formula with no y
    # variable!
    xvars = all.vars(formula)[-1]
    countform = paste("~", xvars[1])
    for (a in 2:length(xvars)) countform = paste(countform, "+",
        xvars[a])
    countarr = xtabs(as.formula(countform), data = data)
    mainarr = xtabs(formula, data = data)
    mainarr[countarr == 0] = NA

    if (type == "mean") {
        mainarr = mainarr/countarr
    } else if (max(countarr) > 1)
        cat("Warning! Some values are sums of multiple instances.\n")
    return(mainarr)
}  # end function

### ....Now let's do it!!!
mywide = safeXtab(weight ~ Time + Chick, data = ChickWeight)
head(mywide[, 1:10])
```

```
##      Chick
## Time 18 16 15 13  9 20 10  8 17 19
##    0  39 41 41 41 42 41 41 42 42 43
##    2  35 45 49 48 51 47 44 50 51 48
##    4  NA 49 56 53 59 54 52 61 61 55
##    6  NA 51 64 60 68 58 63 71 72 62
##    8  NA 57 68 65 85 65 74 84 83 65
##    10 NA 51 68 67 96 73 81 93 89 71
```

Data Manipulation 1.3.2: The Stacked/Wide workarounds

Want to stack it back?

```
restacked = as.data.frame.table(mywide, responseName = "weight")
dim(restacked)
```

```
## [1] 600   3
```

```
head(restacked)
```

```
##   Time Chick weight
## 1    0    18     39
## 2    2    18     35
## 3    4    18     NA
## 4    6    18     NA
## 5    8    18     NA
## 6   10    18     NA
```

```
restacked = restacked[!is.na(restacked$weight), ]
dim(restacked)
```

```
## [1] 578   3
```

Exercise: check if all the `weight` values match the old ones, using `merge` or `match` !

The behavior of `as.data.frame.table` is explained in the `?table` help page. We will learn more powerful reshaping tools in the coming weeks.

Questions? *Online* questions?