UWEO StatR201 Lecture 9

Classification 2: Advanced Methods based on Trees and Nearest-Neighbor

Assaf Oron, March 2013

Tonight's Menu

Last week we met 2 of the simplest classification methods:

- **k-Nearest-Neighbors (KNN)** the perhaps the best among basic methods, in terms of its performance/effort ratio.
- **Classification and Regression Trees (CART)** are perhaps the most implementation friendly *(see table in H-T-F book)*.

Our first look at more advanced methods, will be extensions based on these two. Next week we will look at one or two completely different, *"Boutique"* classifiers.

Then, the session's final one-third: **Some more R Data Manipulation.**

- `stack/unstack/reshape` - and the `xtabs, as.data.frame.table` workarounds
- The `Xapply` family

Classification, Regression and p-Dimensional Space

Last week we saw similarities between classification and logistic/multinomial regression, in terms of the basic formula $\Pr(y = c) = f_c(\mathbf{X})$.

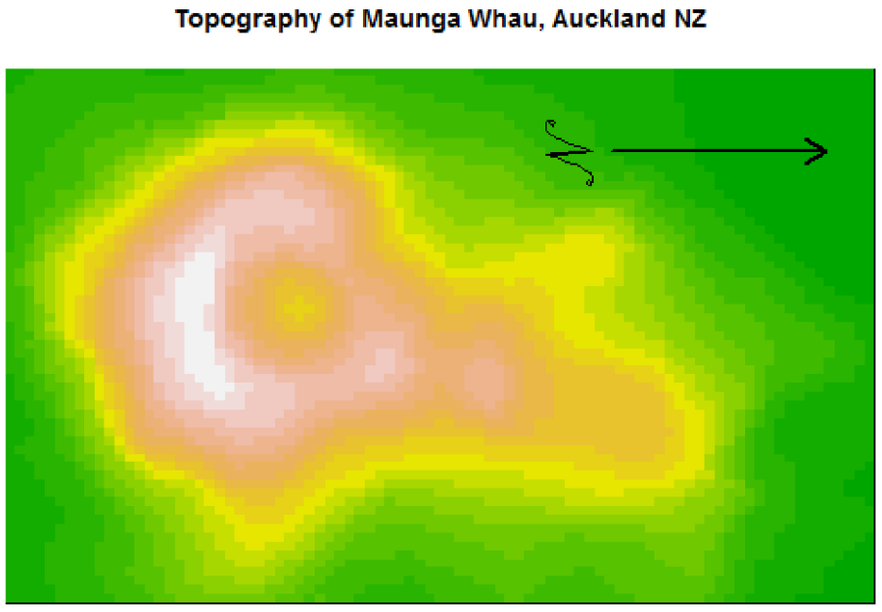
Then we distinguished classification, as **a generalized (probabilistic) partition of p-dimensional feature space, between the classes.**

Logistic/multinomial regression can only offer a limited subset of the possible partitions, using boundaries that are hyper-planes in that space. Via nonlinear terms (polynomials, interactions, splines, etc.), or even **nonlinear regression models**, a broader class of boundaries can be explored; however, generally speaking the methods designed specifically for classification tend to perform better on real-life problems.

Before we continue to these methods, I would like to show another simple and exciting connection between the two worlds - this time between ordered classification (such as the wine-tasting problem) - and regression with a continuous outcome.

Classification, Regression and p-Dimensional Space

Early this quarter we visualized regression in $p + 1$ dimensional space - with y as an additional dimension. But classification is visualized in p-dimensional space, with the class c shown as different-colored points. Can we do this to regression? You bet we can.



in this feature space, $p = 2$ and y is color-coded on a continuous scale. Or, if you want to be particular about it: y is an ordered multi-category variable.

Therefore: evision the y as **the color** of the points (determined by the value of X) in p-dimensional space. This is what *both* regression and classification boil down to.

Questions? *Online* questions?

Random Forests: The Bootstrap

CART's main drawback is performance. This is mostly due to variance, not bias.

Classifiers are estimators. What is the most straightforward (although practically, not always feasible) way to reduce estimation variance?

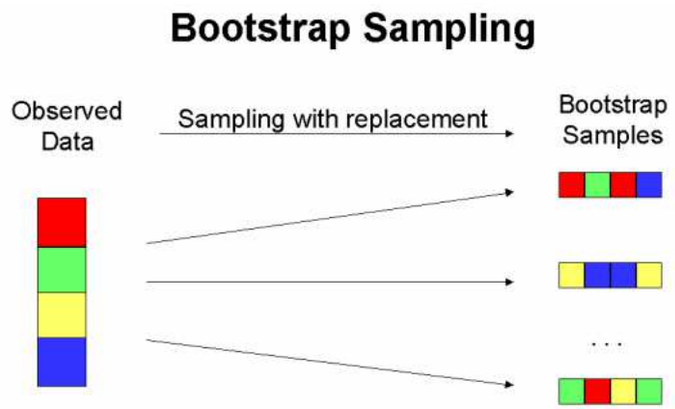
Precisely. Increase the sample size.

Ok, our sample is given. But *if* we could somehow generate many i.i.d. CART estimators, their average (or plurality vote) would certainly perform better than a single tree.

There's a method that can produce for us as many trees as we want, without increasing the sample. It is called Bootstrap Resampling.

What is The Bootstrap?

Bootstrap Resampling is a generic name for any statistical technique that involves **numerically producing an ensemble of virtual samples from the original sample, with replacement.**



The key is **pretending your sample is actually the population.**

1. Take B samples, each of size n, by sampling **with replacement** from your original sample. That is, you generate a "new" dataset, by sampling n *entire individual rows* from your existing dataset, at random with replacement.
2. For each sample indexed b ($b = 1, \ldots, B$) calculate whatever statistic or estimate you are interested in. **In our case: run CART on each sample.**
3. Draw inference or predictions, based on this **bootstrap sample.**

We will learn about the general bootstrap in Spring. Right now, we need to know just enough to understand "Random Forests".

Bagging and Random Forests

The straightforward method of generating B bootstrapped trees, then choosing the class most commonly selected by them, is known as **bagging** (short for *'bootstrap aggregating'*).

It only offers modest performance improvement over CART, because the trees are too highly correlated with each other.

Random Forests add a tweak: you bootstrap the dataset, **but each tree only uses $m < p$ randomly-sampled features.** The feature sampling is, of course *without* replacement.

This reduces the correlation, and usually improves performance over ordinary "bagging."

The number of features available for each bootstrapped tree, m, is the method's main tuning parameter.
The number of bootstrap samples B is of secondary importance: more a cost/benefit decision, since performance will not deteriorate with increasing B - only flatten out.

Random Forests in R

```
options(width = 130)
hitme = function(x, ref) length(x[x == ref])
library(randomForest)
winecartCV = rpartCV(quality ~ ., data = wine, cvid = wineid, method = "class", cpvec = 0.1^seq(2,
    5, 0.5), parms = list(split = "information", loss = wineloss))  ### this is the CART CV we ran last time
```

```
## Wed Mar 06 11:11:34 2013
## .......Wed Mar 06 11:11:44 2013
```

```
apply(winecartCV + 2, 2, hitme, ref = wine$quality)  ### Number of hits, by cp parameter
```

```
## [1] 1515 1521 1507 1489 1472 1472 1472
```

```
wineforest = randomForest(factor(quality) ~ ., data = wine, ntree = 200)
hitme(wineforest$predicted, wine$quality)
```

```
## [1] 1617
```

First, credit where credit is due: **this dataset donated to the UCI repository, by P. Cortez, A. Cerdeira, F. Almeida, T. Matos and J. Reis. Modeling wine preferences by data mining from physicochemical properties. Decision Support Systems, Elsevier, 47(4):547-553, 2009.** Now:

Running `randomForest` using the pre-set defaults (except reducing B from 500 to 200) some improvement over the spruced-up version of CART ($n_{Train} = 2961$).

Wait! Did `randomForest` report CV performance, or in-sample predictions?

Neither. The `predicted` values are **Out-Of-Bag.**

Out-of-Bag Predictions

Recall, each tree in the "Forest" was fit to classify a bootstrapped re-sampling of the original dataset.

Unless n is *very* small, the probability of drawing an exact replica of the true dataset is effectively zero. So some rows will be replicated > 1 times - **and others left out.** In fact, the probability for a specific row being left out is

$$\left(1 - \frac{1}{n}\right)^n, \text{ which as } n \to \infty \text{ tends to } 1/e \approx 0.368.$$

So in each tree, roughly 37% of `wine`'s rows are left out. `randomForest` actually tells us, exactly how many times *each row* is left out:

```
summary(wineforest$oob.times/200)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.255   0.345   0.370   0.368   0.390   0.485
```

The returned `predicted` classes, are a plurality vote among all these left-out cases, for each observation.

We can also find out how close the vote was:

```
summary(apply(wineforest$votes, 1, max))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.239   0.474   0.545   0.558   0.630   0.951
```

In quite a few cases, `randomForest` had a hard time deciding between two or more classes.

Note: bootstrap resampling and OOB predictions, are an increasingly popular alternative to CV for any prediction method - not just for "bagged" methods.

Tuning `randomForest`

```
predmat = matrix(NA, nrow = dim(wine)[1], ncol = 9)
for (a in 1:9) {
    tmp = randomForest(factor(quality) ~ ., data = wine, ntree = 200, mtry = a +
        1)
    predmat[, a] = tmp$predicted
}
apply(predmat + 2, 2, hitme, ref = wine$quality)
```

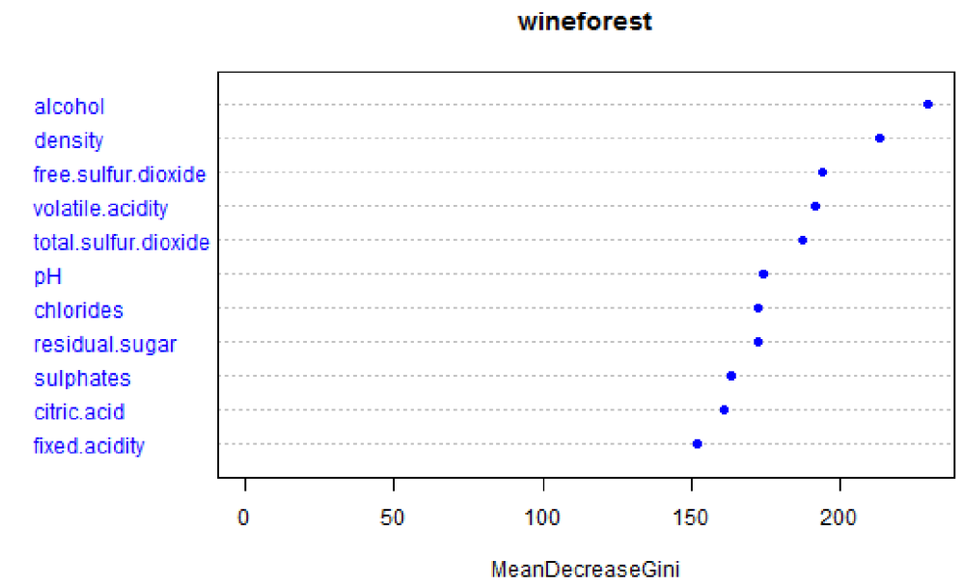
```
## [1] 1619 1614 1615 1610 1575 1599 1588 1607 1585
```

What is this code tuning? (hint: see a few slides back). Note that here, too, class labels have been stripped in the process.

In this dataset, small m values appear to work well (the `randomForest` default is $m = \sqrt{p}$). One reason is that the predictive contribution is fairly evenly distributed across features. How do I know?

Importance here is gauged via **the cumulative improvement in node purity,** from all the splits using each variable, across all trees in the forest.

```
varImpPlot(wineforest, pch = 19, col = 4)
```



Tuning `randomForest`

What about tuning individual-tree behavior? Recall that CART's main tuning parameter was `cp`, controlling tree depth. In `randomForest` there is no `cp` parameter; instead, there's a nearly-equivalent `nodesize`. It tells the trees how fine-grained the terminal nodes should be.

The `randomForest` default for classification is `nodesize=1`: fully-grown trees and zero training error! This also what Hastie-Tibshirani-Friedman recommend doing; their punchline is *"one less tuning parameter."*

Another, perhaps more profitable, tuning idea, which might be relevant to the `wine` dataset: For extreme wine qualities, actually getting more `votes` than their far more common, less-extreme neighbors, is a very high bar to clear. Perhaps lower this a bit using information-entropy bounds?

```
pwine = table(wine$quality)/length(wine$quality)
pinf = -pwine * log(pwine)
round(pinf/sum(pinf), 3)
```

```
##
##     3     4     5     6     7     8     9
## 0.022 0.096 0.276 0.276 0.236 0.089 0.007
```

Using the `cutoff` argument, the selected class might be not the most common vote, but the one beating a class-specific bar by the greatest ratio.

But do play with that: change `nodesize` or `cutoff`. Note that you probably need to re-examine m as you do this, creating a 2D tuning matrix. You can feed `cutoff` any vector of length C that is positive and whose sum is ≤ 1.

Questions? *Online* questions?

From Random Forests to Boosting

Random forests implement a very simple concept motivated by straightforward statistical principles, and manage to substantially improve a base classifier's performance. *btw: this base classifier does not have to be CART.*

However, as Hastie *et al.* demonstrate, it is often defeated by a different tree-ensemble algorithm, which – **the Horrors!** – was neither developed by statisticians, nor motivated by statistical theory. **This method is called Boosting.**

(somewhat saving face, later work revealed it does have a solid, yet implicit, statistical rationale. This eventually led to a generalized "gradient Boosting" algorithm, which is more commonly used nowadays).

Like random forests, boosting eventually takes a vote among its generated ensemble of trees. But unlike random forests, each boosted tree encompasses a **re-training** learned from past trees.

This makes a more complicated algorithm. However, the tradeoff is that while random forest performance flattens out at some B, as the usable information to be gathered from bootstrapping is exhausted – boosting continues to improve for much longer.

Boosting

Boosting in its various flavors, produces a stochastic sequence of **"Stumps", a.k.a. single-split trees,** in the following manner:

1. Fit stump m with weights on the observations.
2. Calculate the error rate err_m.
3. **Amplify** the weights of **misclassified** observations.
4. Rinse. Repeat; until you accumulate M trees or achieve some other threshold.

The final classifier is a weighted plurality vote of all "stumps".

Boosting

```
library(gbm)
boost100 = gbm(factor(quality) ~ ., data = wine, cv.folds = 4, bag.fraction = 1,
    verbose = FALSE)
```

```
## Distribution not specified, assuming multinomial ...
```

```
boost2000 = gbm(factor(quality) ~ ., cv.folds = 4, distribution = "multinomial",
    data = wine, verbose = FALSE, n.trees = 2000, bag.fraction = 1)
boost8000 = gbm(factor(quality) ~ ., cv.folds = 4, distribution = "multinomial",
    data = wine, verbose = FALSE, n.trees = 8000, bag.fraction = 1)
```

The predictions are stored in a log-odds matrix format. Here's how to extract them:

```
hitme(apply(boost100$fit, 1, which.max) + 2, wine$quality)
```

```
## [1] 1443
```

```
hitme(apply(boost2000$fit, 1, which.max) + 2, wine$quality)
```

```
## [1] 1508
```

```
hitme(apply(boost8000$fit, 1, which.max) + 2, wine$quality)
```

```
## [1] 1574
```

On the `wine` dataset, boosting does not appear to outperform random forests; this is probably because of the balance between feature contribution *(why?)*.

Questions? *Online* questions?

Upgrading k Nearest Neighbors

We now return to *"the little method that could."*

One nice thing about KNN is that there is very little "black box" about it. It just compares distances in feature space. Therefore, performance may often be improved using very simple straightforward step – by **modifying** this space before sending it to KNN.

For example, unlike tree methods, KNN *is* sensitive to transformations. If some X is skewed, then a (roughly) symmetrizing transformation such as `log` or `sqrt` might allow a scrunched-up signal to shine through.

In addition, while **scaling** X was bad for `knn` in the `iris` dataset, the reason was atypical and context-specific. Values in each row of `iris` are morphometric measurements **of the same flower.** The relationship between them is part of what distinguishes between species, and their magnitude of different variable was not too different. In that context, scaling each column in X blurs the morphometric uniqueness of each flower, without much benefit in return.

However, had one dimension been, say, 50 times smaller than the others - scaling would have probably helped.

In general datasets like `wine`, where most features are in different units and very different scales, scaling becomes an absolute necessity. Watch:

```
library(class)
kunscaled = knn.cv(wine[, -12], cl = wine$quality)
hitme(kunscaled, wine$quality)
```

```
## [1] 1203
```

```
kscaled = knn.cv(scale(wine[, -12]), cl = wine$quality)
hitme(kscaled, wine$quality)
```

```
## [1] 1446
```

Better, but not very impressive**. See if any symmetrizing transformations can help improve upon this!**

Upgrading k Nearest Neighbors

Another source of potential improvement is tied directly to the **"Curse/Blessing of Dimensionality"** debate. My argument is that as long as we maintain a decent **overall signal-to-noise ratio across dimensions**, KNN's performance will be decent as well.

So we need to carry out a quick-and-dirty, pre-processing subset selection, or as Machine-Learners would call it, *Feature Extraction.* One way is via SVD/PCA; but there are more straightforward methods to gauge signal-to-noise. For example: **just measure the signal-to-noise directly!** We know how to do this - via ANOVA.

```
options(width = 130)
betwith = function(X, y) {
    p = dim(X)[2]
    annie = rep(NA, p)
    names(annie) = substr(names(X), 1, 8)
    for (a in 1:p) annie[a] = anova(lm(X[, a] ~ factor(y)))$F[1]
    return(annie)
}
round(betwith(wine[, -12], wine[, 12]), 1)
```

```
## fixed.ac volatile citric.a residual chloride free.sul total.su  density       pH sulphate  alcohol
##      7.4     39.7      2.4     16.8     27.9     13.7     30.6     80.8      8.8       3.4    166.2
```

Note: we are turning the tables – "predicting" our X's using our y class values. The F statistic is the between-class/within-class square-error ratio.

In `wine`, all features yield a decent signal-to-noise. **As p increases, such "miracles" become less likely.** With large p the between/within screening process can be automated – and to its great advantage, it is completely transparent and self-explanatory. It is also perfectly ok to carry out a small *"2D CV"* matrix over k and those features near the usable signal-to-noise-ratio boundary (typically somewhere between F=1 and 2).

Note: when doing between/within screening with large n and p, you might want to directly calculate the formula, rather than call `lm` which is a bit wasteful.

KNN: Looking Deeper… What Lies Behind It?

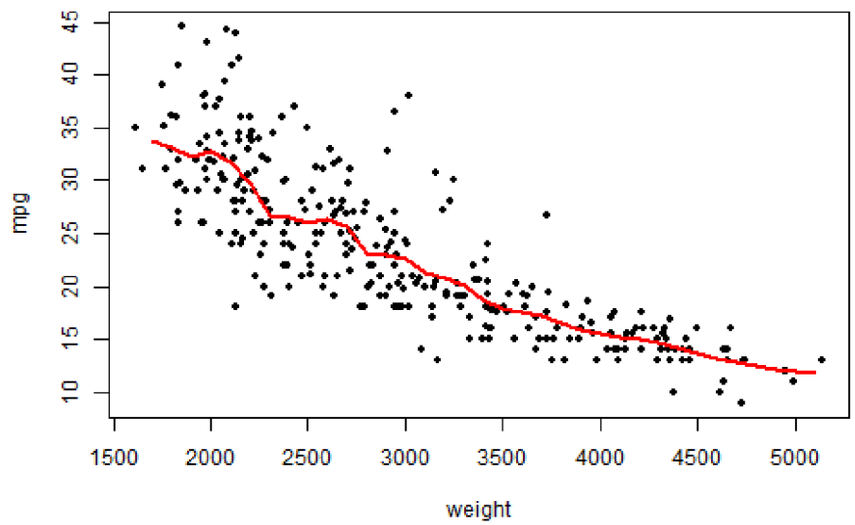
Hastie *et al.* draw an interesting analogy between KNN and randomForests with fully grown (`nodesize=1`) trees. Since the tree squeezes the training data to the max, sending an "out-of-Bag" observation down the tree is equivalent to make it find (one of) its nearest neighbor(s) from among the *leaves.*

Now, the final OOB estimate is a vote among all these trees. **This is essentially a weighted-nearest-neighbor classifier, where the weights are the number of trees in which each near-neighbor ends up being "the one."**

Another, even more accurate analogy, is to **kernel smoothing methods** such as LOWESS ("locally weighted scatterplot smoothing", or *"LOcal regrESSion"*). You might have encountered them; R functions ubiquitously use them to draw smooth fits over scatterplots. They just take a weighted local average over a certain window around each value of x.

Standard KNN is analogous to LOWESS in p dimensions, with a *"rectangular"* (constant) kernel.

```
autos = read.csv("../Datasets/autoMPGtrain.csv", as.is = TRUE)
plot(mpg ~ weight, data = autos, pch = 19, cex = 0.7)
lines((15:55) * 100, predict(loess(mpg ~ weight, data = autos, span =
    100)), col = 2, lwd = 2)
```



Upgrading *and* Generalizing KNN: The `kknn` Package

This package based on K. Hechenbichler's *unpublished* Ph.D. thesis, seems to have it all: `formula`-based KNN and a friendly tuning-CV wrapper? Check.

```
library(kknn)
newine = wine
newine$logsugar = log10(newine$residual.sugar)
newine[, -12] = scale(newine[, -12])
newine = newine[, -4]
k1 = train.kknn(factor(quality) ~ ., data = newine, kernel = "rectangular", kmax = 7)  #Leave-1-out
round(t(k1$MISCLASS), 2)
```

```
##                1    2    3   4    5   6    7
## rectangular 0.52 0.53 0.52 0.5 0.51 0.5 0.51
```

Distance weighted via different kernel forms, for some 2D tuning? Check.

```
k2 = train.kknn(factor(quality) ~ ., data = newine, kmax = 15, kernel = c("rectangular",
    "triangular", "biweight", "optimal"))
round(t(k2$MISCLASS[seq(1, 15, 2), ]), 3)
```

```
##                1     3     5     7     9    11    13    15
## rectangular 0.516 0.521 0.507 0.505 0.493 0.482 0.487 0.488
## triangular  0.516 0.505 0.491 0.486 0.485 0.481 0.481 0.481
## biweight    0.516 0.510 0.500 0.492 0.489 0.488 0.485 0.482
## optimal     0.516 0.516 0.505 0.493 0.482 0.483 0.479 0.477
```

Upgrading *and* Generalizing KNN: The `kknn` Package

Distance calculated as absolute values, or any old power, rather than squares? Check.

```
k4 = train.kknn(factor(quality) ~ ., data = newine, kmax = 15, distance = 3, kernel = c("rectangular",
    "triangular", "optimal"))
round(t(k4$MISCLASS[seq(1, 15, 2), ]), 3)
```

```
##                1     3     5     7     9    11    13    15
## rectangular 0.519 0.522 0.503 0.504 0.493 0.483 0.489 0.481
## triangular  0.519 0.507 0.494 0.482 0.483 0.478 0.479 0.477
## optimal     0.519 0.519 0.504 0.486 0.478 0.480 0.482 0.479
```

Also doing regression? Check.

```
k5 = train.kknn(quality ~ ., data = wine, kernel = c("triangular", "gaussian"), kmax = 15)
round(t(k5$MEAN.SQU[seq(1, 15, 2), ]), 3)
```

```
##               1     3     5     7     9    11    13    15
## triangular 0.874 0.667 0.603 0.569 0.555 0.546 0.541 0.537
## gaussian   0.874 0.629 0.570 0.549 0.545 0.542 0.542 0.537
```

And last but not least: classification on an ordered factor? Check!

```
k6 = train.kknn(factor(quality, ordered = TRUE) ~ ., data = newine, kmax = 22, kernel = c("rectangular",
    "triangular", "optimal"))
round(t(k6$MISCLASS[seq(1, 22, 3), ]), 3)
```

```
##                1     4     7    10    13    16    19    22
## rectangular 0.516 0.492 0.478 0.469 0.470 0.464 0.462 0.458
## triangular  0.516 0.490 0.477 0.468 0.468 0.464 0.460 0.458
## optimal     0.516 0.516 0.482 0.480 0.471 0.470 0.469 0.468
```

With ordinal factor specification, the best-in-class KNN performance is similar to `randomForest`. **Play with it a bit! It's fun!**

Upgrading *and* Generalizing KNN: Other Options

Of course, `kknn` doesn't do **everything**… for example:

- How about "affirmative action" towards minority classes, in the spirit of `randomForest`'s `cutoff` argument?
- How about **weighting features by their signal-to-noise ratio?**
- Also, a non-leave-1-out CV with our own `cvid` provided as input, for comparison across methods?

If we have time next week, I might show you how to carry out KNN "by hand" and explore these options.

As Ben Eacrett shared on the discussion board, there is a fairly new package called `FNN`, implementing some shortcut algorithms for faster KNN. With *"Big Data"*, this can be important. According to Hastie *et al.*, no fancy algorithm could beat KNN in the USPS handwritten digit task. But at the time of implementation, KNN was running to slow, so they trained a neural network to mimic its behavior (which reminds me of the well-known truth: if we had tried to consciously and rationally *teach* our children to speak - they might have never learned :)

The book also presents an algorithm called **DANN** for nearest-neighbors on a locally-calculated transformation of feature space. I don't know whether it is implemented in a package; there's some code to be found online.

A simpler local-reweighting approach would just reweight each local neighborhood by the feature's local signal-to-noise.

Data Manipulation 2.1.1: Summarizing via the `Xapply` Family

First, lest we forget: there's some unfinished business in Lecture 8.

Now… **Very** often in data analysis, we want to obtain group statistics - either as an exploratory product in itself, or to generate a condensed summarized version of a raw dataset, for further analysis.

There is an entire suite of functions right in R's base to carry out these tasks. Naturally, there are also several useful packages with further wizardry.

Today we learn the former. Perhaps we'll start with the most familiar-looking member of `Xapply` - so familiar that it doesn't even have `apply` in its name:)

Data Manipulation 2.1.1: `aggregate`

```
aggregate(Sepal.Length ~ Species, data = iris, FUN = mean)
```

```
##      Species Sepal.Length
## 1     setosa        5.006
## 2 versicolor        5.936
## 3  virginica        6.588
```

```
aggregate(. ~ Species, data = iris, FUN = min)
```

```
##      Species Sepal.Length Sepal.Width Petal.Length Petal.Width
## 1     setosa          4.3         2.3          1.0         0.1
## 2 versicolor          4.9         2.0          3.0         1.0
## 3  virginica          4.9         2.2          4.5         1.4
```

`aggregate` uses a `formula`, making it both convenient and flexible. The contrast with `xtabs` is illuminating:

- `aggregate` can perform any named function, while `xtabs` is limited to count, sum, and (using our `safeXtab` tweak) mean.
- `aggregate` can summarize multiple variables in parallel, whereas `xtabs` (I think) cannot.
- `aggregate` returns a data frame. If there are multiple terms on the `formula` RHS, they will be **stacked in rows by unique level combination.** `xtabs` returns a "wide" array, with a column for each RHS level, and adding a dimension for each RHS term.
- `aggregate` will return summaries only for level-combinations actually present in the data. `xtabs` returns a full array, **with zeros in locations representing missing level-combinations** (`safeXtab` converts these zeros to `NA`).

Check it out, using, e.g., the `ChickWeight` dataset.

`apply` for Regular Arrays and Data Frames

```
apply(ChickWeight[, 1:2], 2, mean)
```

```
## weight   Time
## 121.82  10.72
```

`apply` works on regularly-shaped data (data frames and arrays). It **applies** the function, along the **margin** indicated y the number in its **Argument 2.** As the example shows, the - `1` indicates by rows; `2` by columns. So `apply(x,1,mean)` is equivalent to `rowMeans(x)`, and so forth. - With higher-dimension arrays, it is possible to specify a vector of dimension indices instead of a single margin. The returned object will be of the same dimensionality - with values corresponding to each level combination. **Check it out! - Note regarding all `Xapply` functions:** the `...` argument is used to pass on more information to `FUN`. Like this:

```
apply(ChickWeight[, 1:2], 2, quantile, prob = c(0.1, 0.9))
```

```
##     weight Time
## 10%   47.7    2
## 90%  223.6   20
```

lapply/sapply/vapply/tapply

As far as I can tell, this group is essentially the same function, with various levels of user-interface wrapping.

The most primitive one (literally!) is `lapply`. It accepts **a list** and returns **a list** of the same length, composed of the `FUN` output for each list element.

```
lapply(split(iris$Petal.Length, iris$Species), max)
```

```
## $setosa
## [1] 1.9
##
## $versicolor
## [1] 5.1
##
## $virginica
## [1] 6.9
```

Most commonly, `lapply` will be used on the output of `split` (which splits **Argument 1** into strata by values of **Argument 2**) - but it doesn't have to. It would work on *any* list, as long as `FUN` knows what to do on it.

`sapply` looks the same, but it will **try** to make the ouput more regular whenever feasible (otherwise, a list is returned like in `lapply`)

```
sapply(split(iris$Petal.Length, iris$Species), max)
```

```
##     setosa versicolor  virginica
##        1.9        5.1        6.9
```

Note: both functions would accept whole data frames as input, rather than just vectors. Then `FUN` has a chance to work on a list of data frames, which could be powerful.

lapply/sapply/vapply/tapply

`vapply` ia a more rigid version of `sapply`: when failing to `simplify` the output, it returns an error.

`tapply`, OTOH, is more user-friendly: it does the `split` for you!

```
tapply(iris$Petal.Length, iris$Species, max)
```

```
##     setosa versicolor  virginica
##        1.9        5.1        6.9
```

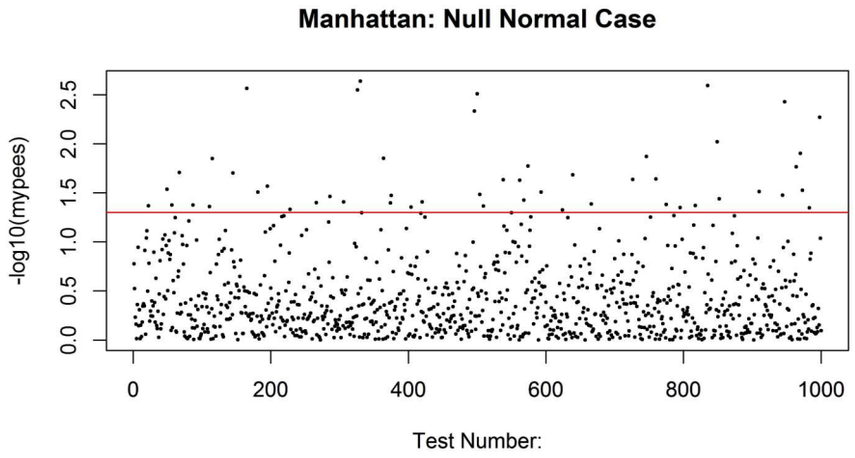
There are wilder members of the `Xapply` family which I've never used, e.g., `rapply` which runs `lapply` recursively, or `dendrapply` which applies `FUN` to the terminal nodes of a tree. I won't bother yourselves and myself with these.

However…

…Enter the Big Guy: `mapply`

You might recall this code bit from Lecture 3 (you were invited to use it in HW2):

```
options(width = 130)
x = matrix(rnorm(10000), nrow = 10)
y = matrix(rnorm(10000), nrow = 10)
mypees = mapply(function(z, w) summary(lm(w ~ z))$coef[2, 4], split(x, col(x)), split(y,
    col(y)))
plot(-log10(mypees), pch = 19, cex = 0.3, main = "Manhattan: Null Normal Case", xlab = "Test Number:")
abline(h = c(-log10(0.05), 3), col = 2)
```



The power of `mapply` is in accepting **multiple** lists as input, and returning an output whose k-th element (or row in case the output is regular) is the result `FUN` run on arguments composed of the k-th elements of all the input lists. This makes it the workhorse of many simulations.

Note the different syntax: `FUN` is moved to the front, and any additional arguments passed to it must be wrapped inside a name-matched list called `MoreArgs` - rather than `...` in its single-input `Xapply` siblings.