UWEO StatR201 Lecture 10

Classification 3: The "Boutique" Methods

Assaf Oron, March 2013

Tonight's Menu

We finally arrive at two of the most famous, *"Boutique"* classifiers.

Neither of the two was developed within the realm of statistics, and each has become its own "industry". However, the book does very nice job of **de-mystifying** them. Eventually, both have a very strong statistical interpretation, and - when the outer shells are removed and the jargon translated to equivalent terms - they turn out to be not so different from methods we've already seen.

- **Support Vector Machines** and the `tune()` utility
- Interlude: ambling towards *"somewhat bigger data"* via the smartphone dataset
- **Neural Networks** and the `train()` utility

Also…

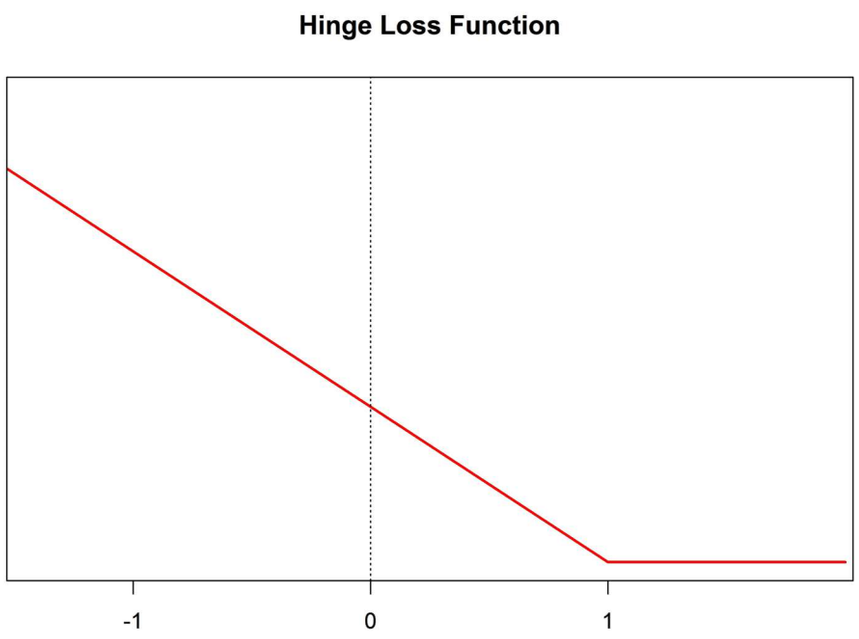
- …Data manipulation via the `plyr` and `reshape2` packages. Unlike previous lectures, some of these (as well as R tricks) will be embedded in the lecture body.
- Fielding some requests for the last lecture.

De-Mystifying Support Vector Machines

By rights, SVM belongs to the regression family. The original method, **optimal separating hyperplanes** (Section 4.5) finds - as its names suggests - a (hyper-)plane that separates two classes.

Does this sound like anything familiar? Oh yes, **logistic regression.** As Figure 4.16 shows, when the classes are separable, the two are essentially identical.

Unlike logistic regression which uses (Iterated Weighted) least-squares, SVM uses a **"hinge loss".**



The loss function has a sharp tip, and some data points eventually don't count at all in the model fit. *(why?)*

Do *these* two attributes remind you of something? (hint: Lecture 4)

- To one direction, the loss penalty is similar to quantile regression.
- To the other direction, it is like robust regression - except that the former ignores gross outliers, while **the hinge loss ignores "inliers"** - those points deemed too deep into the class partition to be helpful for determining its boundaries.

Another peculiar attribute is that the penalty starts increasing **inside the correct class!** (at `x=1` in our figure.) This is because the problem is really formulated (by **Corinna Cortes** and Vladimir Vapnik in 1995), as finding the hyperplane with maximum separation between classes. Separation starts where the penalty starts - so in our case it is 2 units, 1 to each side.

De-Mystifying SVM

Remember my silly *"Assumptions-Properties-Behavior"* terminology? Here's where it becomes useful.

While least-squares (linear/logistic) regression can be traced down to some **assumed** probability model, in reality most people use it for its **properties and behavior.** This is more evident with quantile regression, whose "probability model" is laughable – and becomes unavoidable with robust regression, an algorithm that was developed by statisticians, but not from a model/likelihood perspective.

Similarly, SVM is all about properties and behavior. But as you now know, that doesn't make it so different from most members of the greater regression family.

To nail the point further: AFAIK, the particular loss-function form was chosen because (together with another constraint to be mentioned soon) it enables a numerical solution to the problem (via quadratic programming). There's little use for a great conceptual formula, if it cannot be solved.

Does this (a loss/penalty function that "miraculously" has good properties and a numerical solution) remind you of something?

Components of SVM

Ok, to business. A specific SVM is defined (besides the input data) by

- A **Cost** or total error budget: the maximum allowed sum of all "penetrations" into the separation zone or (Heavens forbid) beyond it and into the wrong class, or - equivalently - **the width of the buffer or "margin",** within which the points play a role in the solution (see below).
- A family of **basis functions.** The actual boundaries will be defined via inner products of basis functions (a.k.a. **kernels**). This makes SVM analogous to nonlinear regression models, and allows the fitting of less-regular boundaries.
- Various parameters used by these basis functions.

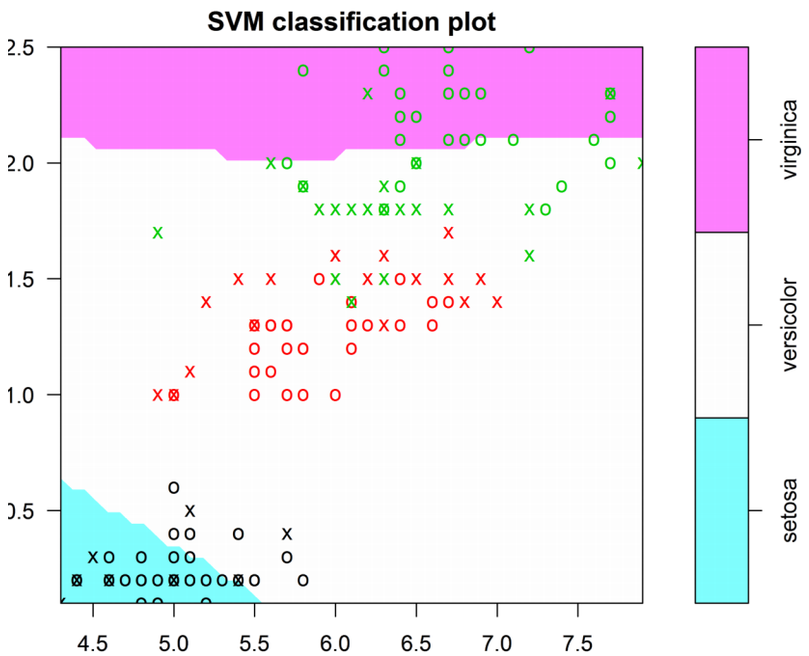
Notes:

- The existence of a numerical solution to SVM, apparently relies on the usage of specific forms of basis-function kernels. This means that the space of possible boundary morphologies is rich - but still constrained.
- The solution itself (i.e., the boundary definition) is **only a function of points with nonzero penalties;** all the "good" points are completely excluded. The points affecting the solution are known as the **support vectors** (the "machine" part refers to the basis-function kernel).
- As said, **the basic SVM application is two-class separation.** With multiclass problems the default solution is to dichotomize every pair of classes, and **choose the class that "wins" the most head-to-head comparisons.** In SVM regression, the formulae remain similar, but instead of a boundary morphology, the solution describes the (hyper-)surface of $\hat{y}$ values.

SVM in R

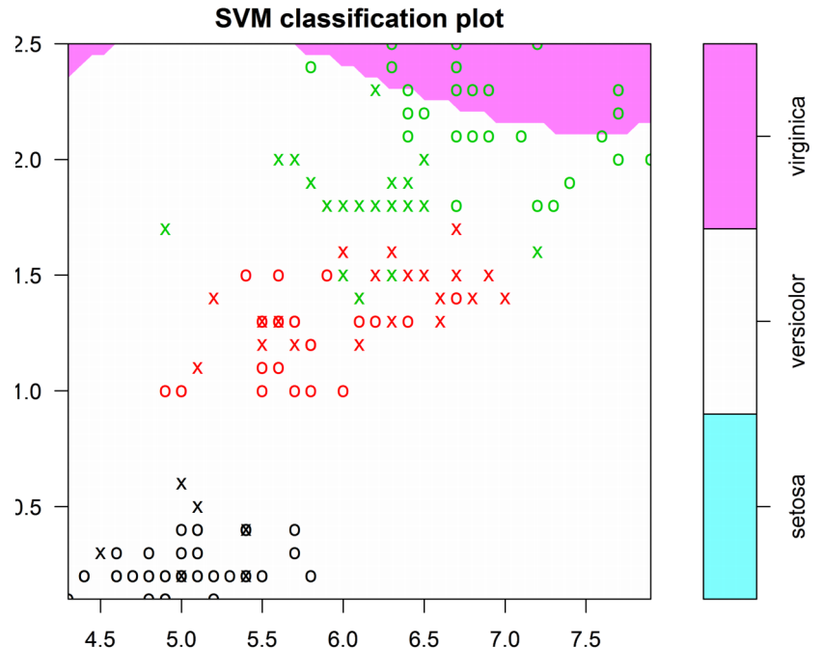
It seems that the blacker the box, the more *"plug and play"* it can become. This little demo is based on an example in `plot.svm` help.
The plots show 2D "slice" cross-sections of feature space. Left: the `radial` (default) kernel, and right: `polynomial`.

```
library(e1071)
par(mar = c(2, 2, 2, 1))
plot(svm(Species ~ ., data = iris), iris, Petal.Width ~ Sepal.Length, slice = list(Sepal.Width = 3,
    Petal.Length = 4))
```



The "x" marks are points participating in the fit (*"support vectors"*).

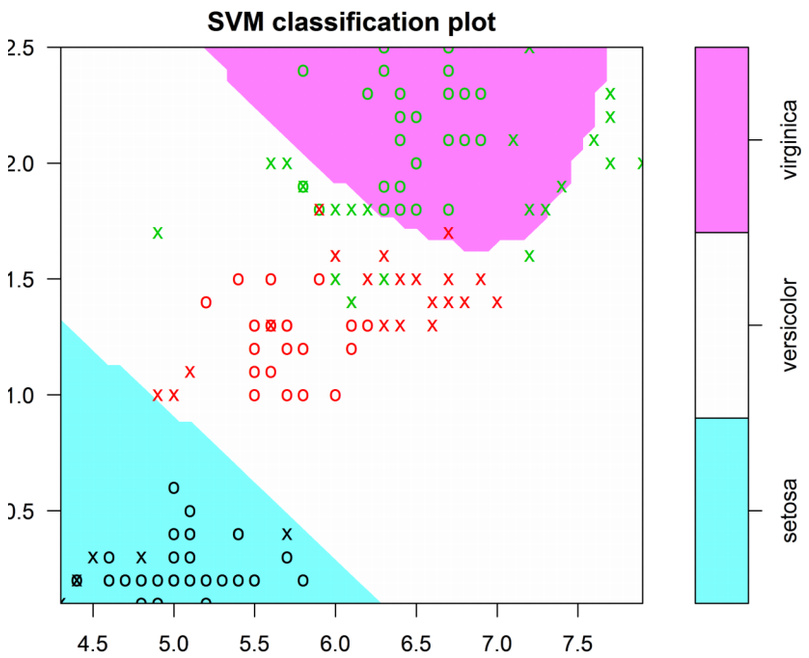
```
par(mar = c(2, 2, 2, 1))
plot(svm(Species ~ ., data = iris, kernel = "polynomial"), iris, Petal.Width
    slice = list(Sepal.Width = 3, Petal.Length = 4))
```



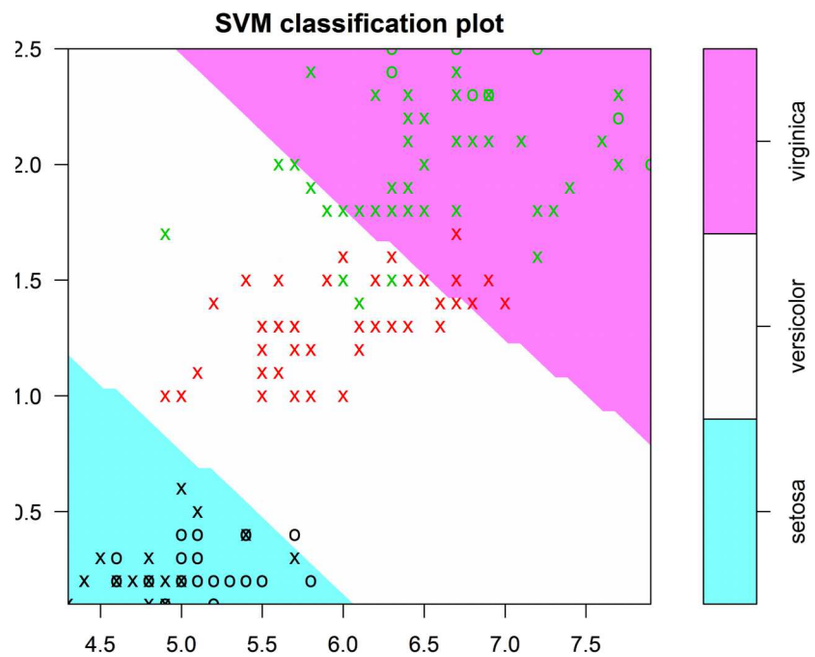
SVM in R

Here, both plots use the `sigmoid` (logistic) kernel. The left with `cost=1` (default); right with `cost=0.1`. `cost` controls the overall "error budget" or - equivalently - the width of the separation buffer within which the data points count (outside of it, they don't). **Smaller `cost` encourages a more aggressive fit.**

```
par(mar = c(2, 2, 2, 1))
plot(svm(Species ~ ., data = iris, kernel = "sigmoid"), iris, Petal.Width ~ Sepal.Length,
    slice = list(Sepal.Width = 3, Petal.Length = 4))
```




```
par(mar = c(2, 2, 2, 1))
plot(svm(Species ~ ., data = iris, kernel = "sigmoid", cost = 0.1), iris, Pe
    Sepal.Length, slice = list(Sepal.Width = 3, Petal.Length = 4))
```



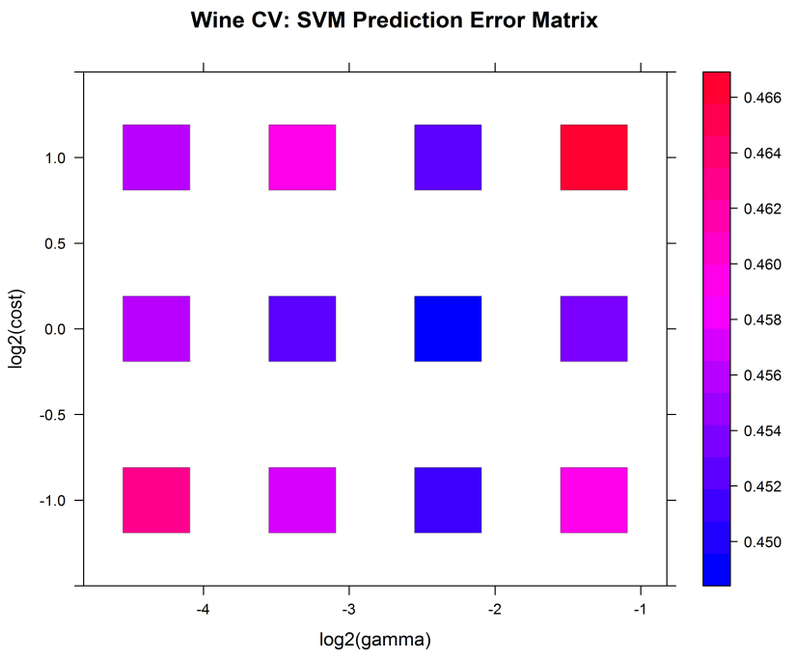
Tuning SVM, or: 2 Cool R Tricks in One

Besides `cost`, there are numerous other tunable parameters. To our aid comes the neat `tune` utility, part of SVM's `e1071` package.

```
options(width = 70)
tuney = tune(svm, factor(quality) ~ ., data = wine, ranges = list(gamma = 0.1 * 2^(-1:2),
    cost = 2^((-1):1)), tunecontrol = tune.control(sampling = "cross"))
library(latticeExtra)
levelplot(error ~ log2(gamma) + log2(cost), data = tuney$performance,

panel = panel.levelplot.points,
    cex = 8, pch = 22, col.regions = rainbow(31, start = 2/3),

        main = "Wine CV: SVM Prediction Error Matrix")
```

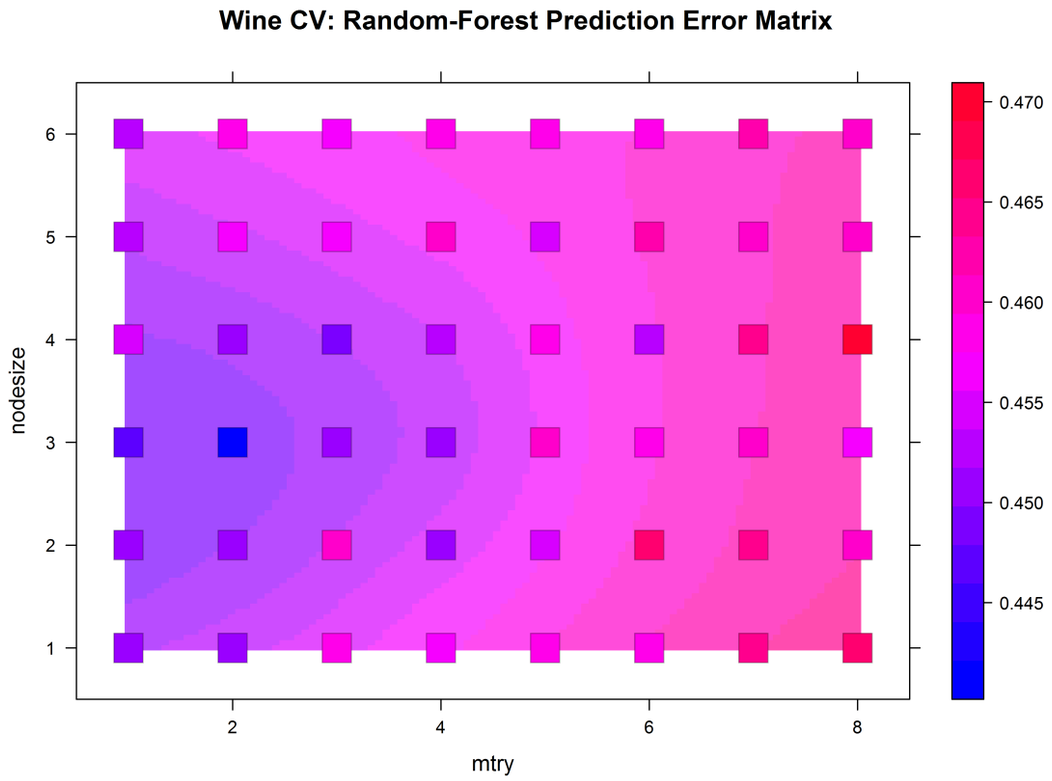


Notes:

- `gamma` is considered a secondary tuning parameter to `cost`, but as we see that's not always the case in practice.
- `gamma` is a decay-rate parameter for the basis functions. It affects all kernel types except `linear`.
- `levelplot` is part of `lattice` itself, but `panel.levelplot.points` which **greatly** enhances this function, is part of `latticeExtra`.
- `panel.levelplot.points` works with irregularly-gridded data as well, which is often a great advantage (this functionality also exists in spatial-stats packages, but is quite rare elsewhere).

More R tricks: `tune`, not just for SVM!

```
library(randomForest)
tuney2 = tune(randomForest, factor(quality) ~ ., data = wine, ranges = list(mtry = 1:8,
    nodesize = 1:6), ntree = 200, tunecontrol = tune.control(sampling = "cross",
    cross = 5))
levelplot(error ~ mtry + nodesize, data = tuney2$performance, panel = panel.levelplot.points,
    cex = 3, pch = 22, col.regions = rainbow(31, start = 2/3), main = "Wine CV: Random-Forest Prediction Error Matrix") +
    layer_(panel.2dsmoother(..., col.regions = rainbow(31, start = 2/3, alpha = 0.7)))
```



`tune` also works with `knn, rpart, nnet.`

SVM Notes, Pros and Cons

Notes:

- In my books, SVM is a variant of 'robustified' nonlinear regression. I may be wrong though…
- `svm` also has a `class.weights` argument, to try and offset the *"tyranny of the majority"* symptom. Tried it on `wine`, seemed to make things worse. Documentation of this feature not great (but package author very responsive).

Pros:

- SVM focuses directly on class-boundary morphology.
- Rich, flexible and growing class of models, and savvy and growing knowledge base.

Cons:

- Tuning required in several dimensions, and across broad parameter ranges (`tune.svm` works with 4-5 parameters regardless of kernel, **Not** including the kernel choice itself).
- Besides the overhead, richness of model and tuning space opens the door to **overfitting.** (cross-validation is not 100% overfit-proof)
- Tool mostly designed for binary classification; multiclass and regression problems, perhaps not as optimal?

Questions? *Online* questions?

And Now… Ambling towards *"Big Data"*…

Once upon a time, the datasets we've dealt with so far might have been considered decent-sized. Nowadays they are definitely *Small Data;* even `wine` doesn't have enough features to be considered anything else.

One feature of today's data deluge is its **streaming** nature, whether from online sources, remote sensing or other electronic devices. I just came across a dataset that might be called **Medium Data,** but nicely incorporates that streaming quantity:

Davide Anguita, Alessandro Ghio, Luca Oneto, Xavier Parra and Jorge L. Reyes-Ortiz. Human Activity Recognition on Smartphones using a Multiclass Hardware-Friendly Support Vector Machine. International Workshop of Ambient Assisted Living (IWAAL 2012). Vitoria-Gasteiz, Spain. Dec 2012

The problem is classifying six activity types (walking straight/up/downstairs, standing, sitting and laying down), according to movement data collected via the person's smartphone.

- There were 30 volunteers aged 19 to 48, of whom 21 are in a training set and 9 in a test set.
- Features were pre-processed and **normalized** to the interval [-1,1].

Disclaimer: unless you are already familiar with the specific research field, datasets of this magnitude might require weeks to find a 'winning' analysis approach towards. I've only had a couple of hours so far. I will show some things that look reasonable, and involve simple straightforward tools, based on that limited exposure and my general experience. More likely than not, there are better ways to be found.

Smartphone Dataset Basic Descriptives

```
options(width = 130)
smartrain = read.csv("../Datasets/smarTrain.csv", as.is = TRUE)
dim(smartrain)
```

```
## [1] 7352  563
```

```
table(is.na(smartrain))
```

```
##
##   FALSE
## 4139176
```

```
names(smartrain)[1:6]
```

```
## [1] "Subject"           "tBodyAcc.mean...X" "tBodyAcc.mean...Y" "tBodyAcc.mean...Z" "tBodyAcc.std...X"  "tBodyAcc.std...Y"
```

```
names(smartrain)[558:563]
```

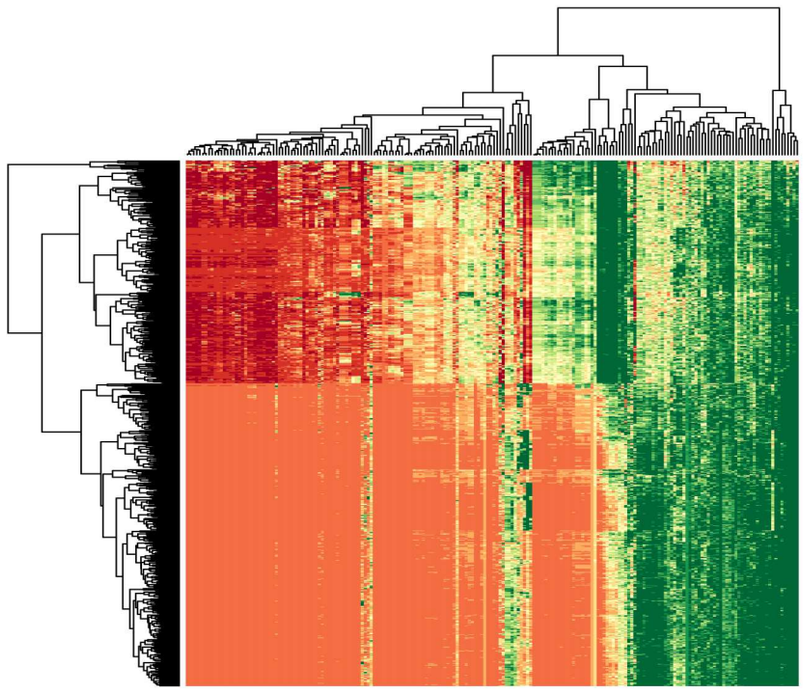
```
## [1] "angle.tBodyGyroMean.gravityMean."     "angle.tBodyGyroJerkMean.gravityMean." "angle.X.gravityMean."
## [4] "angle.Y.gravityMean."                 "angle.Z.gravityMean."                 "Class"
```

```
table(smartrain$Class, smartrain$Subject)
```

```
##
##                    1  3  5  6  7  8 11 14 15 16 17 19 21 22 23 25 26 27 28 29 30
##   1                95 58 56 57 57 48 59 59 54 51 61 52 52 46 59 74 59 57 54 53 65
##   2                53 59 47 51 51 41 54 54 48 51 48 40 47 42 51 65 55 51 51 49 65
##   3                49 49 47 48 47 38 46 45 42 47 46 39 45 36 54 58 50 44 46 48 62
##   4                47 52 44 55 48 46 53 54 59 69 64 73 85 62 68 65 78 70 72 60 62
##   5                53 61 56 57 53 54 47 60 53 78 78 73 89 63 68 74 74 80 79 65 59
##   6                50 62 52 57 52 54 57 51 72 70 71 83 90 72 72 73 76 74 80 69 70
```

Smartphone Dataset Basic Descriptives

```
library(RColorBrewer)
par(mar = c(2, 2, 2, 0))
heatmap(as.matrix(smartrain[
sample(1:7352, size = 800), sample(2:562, size = 200)]), col = brewer.pal(11, "RdYlGn"), breaks = c(-20,
    (-4.5:4.5)/4, 20), labRow = NA, labCol = NA)
```



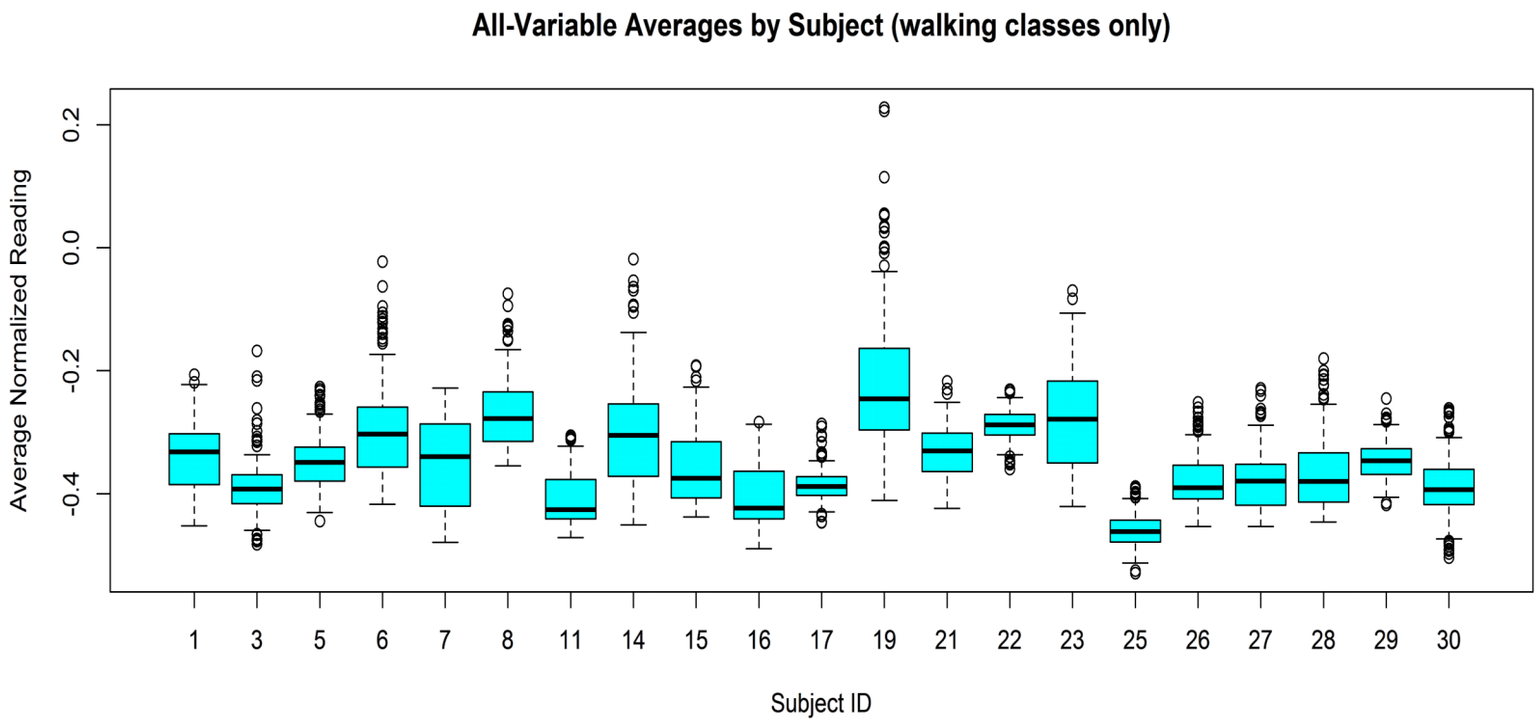
Notes:

- I took random `samples` of rows and columns, just to save time. `heatmap` can be very slow.
- The function simultaneously **clusters** rows and columns. Row-column associations show up as **block patterns in the heatmap.** The identity of the the rows/columns involved can be found via assigning the `heatmap` to a variable.
- `heatmap` scales the rows by default, but this can be changed.
- `RColorBrewer` provides professionally-designed, attractive palettes of 11 color levels each.

Smartphone Data: Dealing with the Grouping

The data are grouped by person; how would that affect things?

```
boxplot(rowMeans(smartrain[smartrain$Class < 4, 2:562]) ~ smartrain$Subject[smartrain$Class < 4], range = 1, col = 5, main = "All-Variable Averages by Subject (walking classes only)",
    xlab = "Subject ID", ylab = "Average Normalized Reading")
```



When different people walk, they generate different typical-intensity readings. In regression there are many ways to take care of that. But in classification based on feature-space values, we probably treat it as **a nuisance** to be removed. Here's how:

Data Manipulation 3.1.1: Summarizing via `ddply`

We now enter our first bit of data manipulation through the back door. What we want is to **scale each subject's data separately.** Something like:

```
aggregate(.~Subject,data=smartphone[,-563],scale)
```

…which returns an error - apparently, because of unequal numbers of rows across subjects. So do `tapply` and `vapply`. `sapply` manages to process this, but returns a list separated by subject, which could potentially be re-bundled.

But fortunately, as usual we are not at the monopolistic mercy of the base packages. To our aid comes `plyr`, courtesy of package juggernaut Hadley Wickham:

```
library(plyr)
smartscale = ddply(smartrain, "Subject", function(x) data.frame(cbind(scale(x[, 2:562]), class = x$Class)))
```

`ddply` splits an input data frame into data frames by its second argument. Its third argument works on these pieces, and (preferably) each returns a compatible data frame - which are then concatenated to one big output data frame. It can do far more than this basic feat, often with (deliberately?) acrobatic syntax.

Btw, we might want to check that this has indeed worked…

Moving Forward:

With “Big Data” (just like with “Small Data”), the first step, then, is **align/scale/clean.** I’m skipping the “clean” part and moving to step 2, whose main goal is to make the data a bit smaller: **filtering, a.k.a. feature extraction.**

We have 561 features. Do we *really* need all of them?

```
source("../Code/filtering.r")
filt0 = apply(smartscale[, 2:562], 2, betwith0, y = smartscale$class)
table(cut(filt0, c(0:4, max(filt0))))
```

```
##
##    (0,1]    (1,2]    (2,3]    (3,4] (4,35.7]
##      201       47       44       37      232
```

201 features have a between/within signal-to-noise ratio of < 1. But this is somewhat of a hatchet job:

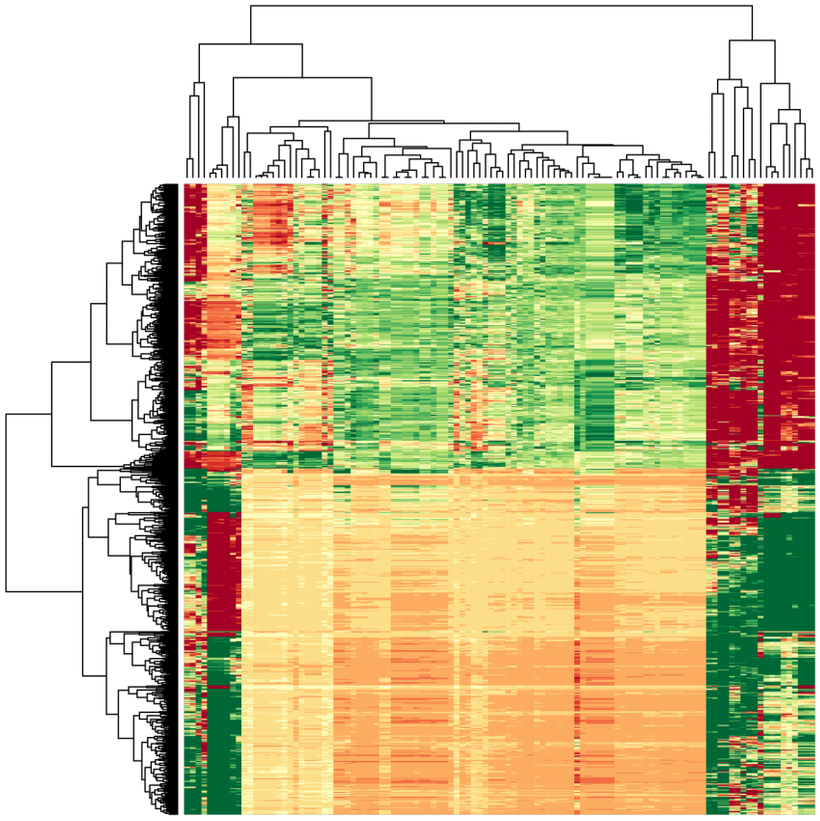
- First, we are cheating a little bit, since in proper Cross-Validation, the filtering should be done separately in each CV cycle.
- Second, we already have indication that signals are grouped into 3 “walking classes” and 3 “stationary classes”. The harder job is to separate **inside** each 3, not the entire 6 (which in most cases, would just be a 3:3 split).

```
options(width = 140)
source("../Code/filtering.r")
filt11 = apply(smartscale[smartscale$class < 4, 2:562], 2, betwith0, y = smartscale$class[smartscale$class < 4])
filt12 = apply(smartscale[smartscale$class > 3, 2:562], 2, betwith0, y = smartscale$class[smartscale$class > 3])
table(cut(filt11, c(0:2, max(filt11))), cut(filt12, c(0:2, max(filt12))))
```

```
##
##           (0,1] (1,2] (2,45.4]
##   (0,1]     451     4       10
##   (1,2]      65     0        1
##   (2,4.5]    30     0        0
```

Smartphone `heatmap` on Filtered Features

```
par(mar = c(2, 2, 2, 0))
heatmap(as.matrix(smartscale[, -1][sample(1:7352, size = 1000), filt11 > 1 | filt12 > 1]), col = brewer.pal(11, "RdYlGn"), breaks = c(-20,
    (-4.5:4.5)/4, 20), labRow = NA, labCol = NA)
```



Cross-Validating a Grouped Dataset

Let's do some classification now…

```
library(class)
testid = sample(1:7352, size = 990)
naive = knn(smartscale[-testid, 2:562], smartscale[testid, 2:562], cl = smartscale$class[-testid])
table(smartscale$class[testid] == naive)
```

```
##
## FALSE  TRUE
##    33   957
```

Wow!!! Big data is Fun! No worries - we got a 3% KNN validation error right off the bat. Right? **Wrong.** We cheated twice:

1. We `scale`d the data within subject, and now borrowed this knowledge.
2. Even worse: each row's nearest neighbor almost surely comes from the same subject! **In short: we must validate while respecting the grouping.**

```
testid = which(smartscale$Subject < 6)
nonaive = knn(smartscale[-testid, 2:562], smartscale[testid, 2:562], cl = smartscale$class[-testid])
table(smartscale$class[testid] == nonaive)
```

```
##
## FALSE  TRUE
##   103   887
```

Wowsa. Let's see if using **only** filtered features can help:

```
filtid = which(filt11 > 1 | filt12 > 1) + 1
kfilt = knn(smartscale[-testid, filtid], smartscale[testid, filtid
table(smartscale$class[testid] == kfilt)
```

```
##
## FALSE  TRUE
##    70   920
```

```
table(smartscale$class[testid], kfilt)
```

```
##    kfilt
##       1   2   3   4   5   6
##   1 208   1   0   0   0   0
##   2   5 153   1   0   0   0
##   3   5   0 140   0   0   0
##   4   0   0   0 121  22   0
##   5   0   0   0  36 134   0
##   6   0   0   0   0   0 164
```

Despite our disregard for *overall* between/within, there is **No** confusion between "walking" and "stationary" classes.

In fact, this prediction error seems rather competitive with what the authors published (paper uploaded to lecture folder).

We still cheated a little bit. How so?

Ok, Time for the Grand Classification Finale.

"There has been a great deal of hype surrounding neural networks, making them seem magical and mysterious." (Hastie et al., p.392)

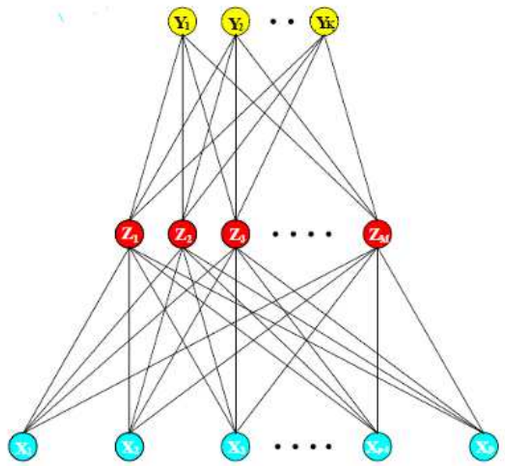


FIGURE 11.2. *Schematic of a single hidden layer, feed-forward neural network.*

Artificial Neural Networks (ANNs) are not some remote member of the regression family (like SVM) – rather, they **are** a regression model – albeit a hiearchical, nonlinear and not-quite-tractable one, typically with 100's to 1000's of parameters to fit… In the single-layer example above,

$$Z_m = \sigma(X\alpha), \quad m = 1, \ldots, M;$$
$$\Pr(Y = k) = \sigma(Z\beta), \quad k = 1, \ldots, K.$$

The model solves for $\hat{\alpha}, \hat{\beta}$ (known in ANN jargon as **weights**) via cycles of local optimization. The fit is penalized with a ridge-regression type (i.e., sum-of-squares) **decay -** or directly prevented from being perfect. The nonlinear functions σ are usually the *sigmoid* (=logistic) curve.

Basic ANNs in R: `nnet`

```
library(nnet)
winetest = sample(1:2961, size = 1000)
net1 = nnet(factor(quality) ~ ., data = wine[-winetest, ], decay = 0.01, size = 20)
```

```
## # weights:  387
## initial  value 4042.162181
## iter  10 value 2596.208760
## iter  20 value 2482.334292
...
## iter 100 value 2021.525152
## final  value 2021.525152
## stopped after 100 iterations
```

```
pret1 = predict(net1, newdata = wine[winetest, ], type = "class")
table(wine$quality[winetest] == pret1)
```

```
##
## FALSE  TRUE
##   483   517
```

```
table(wine$quality[winetest], pret1)
```

```
##    pret1
##       3   4   5   6   7
##   3   0   0   3   2   0
##   4   0   9  26  10   0
##   5   0   8 159 103   7
##   6   1   1 109 300  43
##   7   0   0   9 126  49
##   8   0   1   1  21   9
##   9   0   0   0   1   2
```

`nnet` creates a single layer with `size` parameters.
Performance here is similar to other classifiers.

Unlike regression, with ANNs scaling **does** change the result:

```
scwine = data.frame(cbind(scale(wine[, -12]), quality = wine$quality))
net2 = nnet(factor(quality) ~ ., data = scwine[-winetest, ], decay = 0.01, s
```

```
## # weights:  387
## initial  value 4640.034745
## iter  10 value 2163.226963
## iter  20 value 1997.159855
...
## iter  90 value 1720.162240
## iter 100 value 1709.688104
## final  value 1709.688104
## stopped after 100 iterations
```

```
pret2 = predict(net2, newdata = scwine[winetest, ], type = "class")
table(wine$quality[winetest] == pret2)
```

```
##
## FALSE  TRUE
##   507   493
```

```
table(pret2 == pret1)
```

```
##
## FALSE  TRUE
##   342   658
```

Let's investigate a bit! Should we `scale` inputs or not?

More R tricks: Tuning `nnet` using `caret`

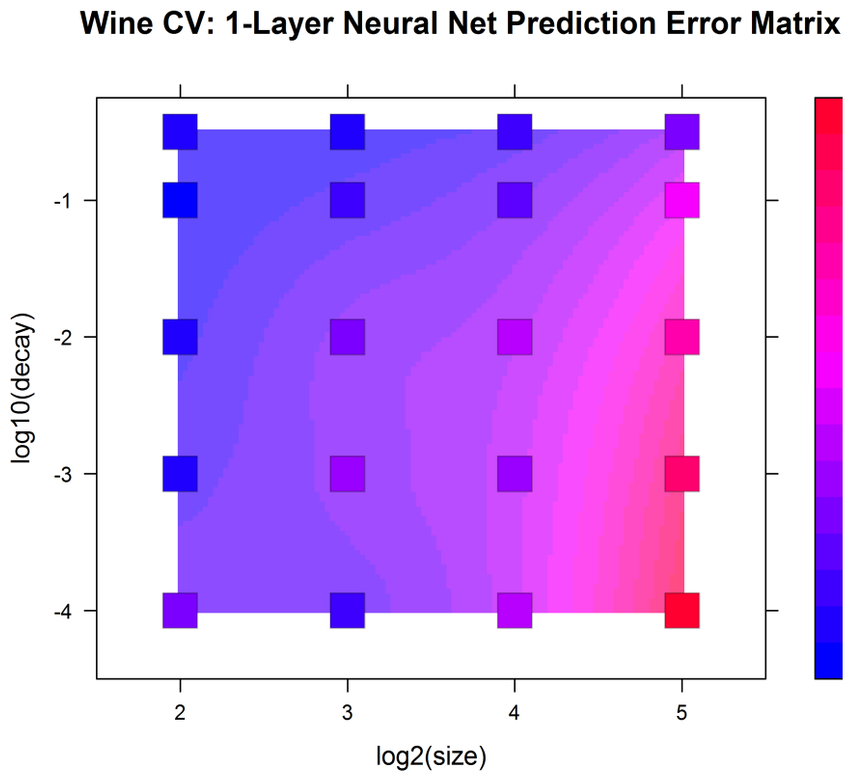
```
library(caret)
tuney3 = train(factor(quality) ~ ., data = scwine, method = "nnet",

maxNWts = 2000, tuneGrid = expand.grid(.size = 2^(2:5), .decay = 0.1^c(0.5,
    1:4)), trControl = trainControl(method = "cv", number = 5))
```

```
## # weights:  83
## initial  value 6479.629310
## iter  10 value 2716.873727
## iter  20 value 2609.547631
...
## iter 100 value 3101.800689
## final  value 3101.800689
## stopped after 100 iterations
```

```
levelplot((1 - Accuracy) ~ log2(size) + log10(decay), data = tuney3$results,
 panel = panel.levelplot.points, cex = 3, pch = 22, col.regions = rainbow(31,
    start = 2/3), main = "Wine CV: 1-Layer Neural Net Prediction Error Matrix")

        + layer_(panel.2dsmoother(..., col.regions = rainbow(31,
    start = 2/3, alpha = 0.7)))
```



The `train` function is part of `caret`, a multi-purpose predictive-modeling aid. It works with a staggering number of model types and packages (see its help page!).

`caret` also includes a suite feature filtering services, similar to the one we did “by hand” for the smartphone dataset - and a host of other useful tools. Check it out!

Multi-Layer ANNs via the `RSNNS` Package

`nnet` is didactically useful, but for most problems I doubt that it would make the best predictive tool (see Section 11.7). The best ANNs are multi-layer, with an **architecture** (i.e., the graph of inter-layer connections) that does justice to the problem's real-life context.

`RSNNS` provides an R wrapper to the Stuttgart Neural Network Simulator. As a more powerful (and somewhat arcane) tool not designed specifically for R, SNNS requires some prework – and `RSNNS` comes up with some solutions that are useful beyond ANNs:

```
library(RSNNS); options(width = 130)
wine = rbind(wine[-winetest, ], wine[winetest, ])
winequal = decodeClassLabels(factor(wine$quality))
head(winequal, 2)
```

```
##      3 4 5 6 7 8 9
## [1,] 0 0 0 1 0 0 0
## [2,] 0 0 0 1 0 0 0
```

```
winetr = normTrainingAndTestSet(splitForTrainingAndTest(wine[, -12], winequal, ratio = 1000/2961))
str(winetr)
```

```
## List of 4
##  $ inputsTrain : num [1:1961, 1:11] 0.207 1.51 0.444 1.51 2.103 ...
##   ..- attr(*, "normParams")=List of 3
##   .. ..$ colMeans: num [1:11] 6.825 0.281 0.333 5.81 0.046 ...
##   .. ..$ colSds  : num [1:11] 0.8441 0.1069 0.1199 4.7344 0.0233 ...
##   .. ..$ type    : chr "norm"
##  $ targetsTrain: num [1:1961, 1:7] 0 0 0 0 0 0 0 0 0 0 ...
##   .. ..$ : chr [1:7] "3" "4" "5" "6" ...
##  $ inputsTest  : num [1:1000, 1:11] -0.741 -0.978 0.207 -0.859 -0.385 ...
##   ..- attr(*, "normParams")=List of 3
##   .. ..$ colMeans: num [1:11] 6.825 0.281 0.333 5.81 0.046 ...
##   .. ..$ colSds  : num [1:11] 0.8441 0.1069 0.1199 4.7344 0.0233 ...
##   .. ..$ type    : chr "norm"
##  $ targetsTest : num [1:1000, 1:7] 0 0 0 0 0 0 0 0 0 0 ...
##   .. ..$ : chr [1:7] "3" "4" "5" "6" ...
```

`splitForTrainingAndTest` turns the **bottom** rows into a test set (that's why I did that strange row shuffle).

`normTrainingAndTestSet` scales the data in the way I explained: both training and test are scaled by the training data's parameters.

Multi-Layer ANNs vis the `RSNNS` Package

```
my1st2layer = mlp(winetr$inputsTrain, winetr$targetsTrain, size = c(6, 3),

 learnFuncParams = c(0.2, 0.1))
twolayerpred = predict(my1st2layer, winetr$inputsTest)
table(wine$quality[1962:2961] == 2 + apply(twolayerpred, 1, which.max))
```

```
##
## FALSE  TRUE
##   474   526
```

```
table(wine$quality[1962:2961], 2 + apply(twolayerpred, 1, which.max))
```

```
##
##       5   6   7
##   3   2   2   1
##   4  35   9   1
##   5 154 113  10
##   6  94 327  33
##   7  11 128  45
##   8   0  18  14
##   9   0   1   2
```

Not too interesting… here's a more challenging example.

```
smartclass = decodeClassLabels(factor(smartscale$class))
cat(date(), " \n")
```

```
## Wed Mar 13 22:10:49 2013
```

```
my1st3layer = mlp(smartscale[-testid, filtid], smartclass[-testid, ],
 size = c(40, 15, 6), learnFuncParams = c(0.2, 0.1))
cat(date(), "\n")
```

```
## Wed Mar 13 22:11:54 2013
```

```
threelayerpred = predict(my1st3layer, smartscale[testid, filtid])
table(smartscale$class[testid] == apply(threelayerpred, 1, which.max))
```

```
##
## FALSE  TRUE
##    58   932
```

```
table(smartscale$class[testid], apply(threelayerpred, 1, which.max))
```

```
##
##       1   2   3   4   5   6
##   1 209   0   0   0   0   0
##   2   0 159   0   0   0   0
##   3   0   0 145   0   0   0
##   4   0   0   0 136   7   0
##   5   0   0   0  51 119   0
##   6   0   0   0   0   0 164
```

Classification: Grand Summary

- I've seen more excitement once we engaged this topic. It seems more related to real-life problems many of you encounter at work.
- Don't forget though, that we got there by building upon basic regression modeling. Both regression perspectives **(prediction and inference)** are still immensely relevant to most professions. Although all non-classification methods we've learned can be retooled for continuous-y prediction, they do not provide the same clarity of regression, and (probably with the exception of neural networks) usually don't predict as well as regression, too.
- I am not a classification expert by any means, so even though I had greatly feared this part of the class, I have ended up enjoying it and learning a huge deal of material and insight. **So Thanks!**
- I skipped some classification methods considered to be standard. In particular, **Linear Discriminant Analysis (LDA)** – the method used by Fisher to first(?) present the problem from a statistical perspective – and its variants (QDA, etc.). As a rule, LDA just has lousy performance. Given our time limitations I didn't feel compelled to present it just because every single classification text (even Hastie *et al.*) starts from it.
- Another skipped method was **Naive Bayes.** One reason is that we've been hiding Bayesian methods from you till Spring (Lecture 2, in the current plan). Another reason is that like LDA, it tends to perform poorly.

Questions? Comments? Anything you would like to see next week? (*small* things :)

Data Manipulation 3.1.2: The Stacked/Wide workarounds

Remember this?

```
source("../Code/safeXtab.r")
library(reshape2)
mywide = safeXtab(weight ~ Time + Chick, data = ChickWeight)
restacked = melt(mywide)
dim(restacked)
```

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

```
head(restacked)
```

```
##   Time Chick value
## 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$value), ]
dim(restacked)
```

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

While `as.data.frame` works fine on this example, it would **not** work on a data frame that is not a table. The `melt` function from `reshape2` (another package by Hadley Wickham) does the trick. There's also a `cast` function for unstacking, and `reshape` for simultaneous stack-unstack operations; the latter, again, with some tricky syntax.