Machine Learning

Dennis Wylie

July 16, 2020

Contents

1 Algorithms That Learn To Predict **3**

2 Data **6**

3 k-Means Clustering **8**

4 Hierarchical Clustering **9**
- 4.1 Dissimilarity metrics . . . 12
- 4.2 Agglomeration Linkage Methods . . . 13
- 4.3 Clustered Heatmaps . . . 14
- 4.4 Predicting Clusters from Hierachical Clustering . . . 15

5 Principal Component Analysis **16**
- 5.1 Modeling Expression Levels with First PC (or Two) . . . 19
- 5.2 Percent Variation Explained by PC k . . . 21
- 5.3 Extracting Sampling Unit Scores . . . 23

6 Hess Data Set **25**

7 k-Nearest Neighbors (knn) **27**

8 Overfitting **29**

9 knn Simulation **32**

10 Cross-Validation **37**

11 Feature Selection **42**

12 Feature Extraction **48**

13 Going deeper **52**

14 Regression Models **53**
- 14.1 Regressing Noise . . . 56

15 Regularization **59**

16 Logistic Regression **61**

17 DLDA and Naive Bayes **64**
17.1 Naive Bayes: does it work? 66

18 Support Vector Machines **67**

19 Bootstrapping **71**
19.1 Bootstrapping for Performance Estimation 72

20 Decision Tree Classifiers **74**

21 Bagging: Bootstrap Aggregating Models **77**

22 Classification Performance Metrics **78**
22.1 Wrap-up: Comparing Models by AUC 82

23 Bibliography **85**

1 Algorithms That Learn To Predict

Machine learning (ML) refers to the use of algorithms which can learn from data. The inputs to a ML algorithm will generally be some sort of data set—which I will refer to as the *training data*—and the output will usually be another algorithm, which I will call a *fit model*, or sometimes just a *fit* if I'm feeling lazy.

- ML algorithm : training data $\rightarrow$ fit model

The fit model itself also takes data as input, and generally requires that the data provided to it must be very similar in nature to that provided to the ML algorithm as training data: For example, assuming the data sets in question are represented in table form, the data provided to the fit model must usually have all or almost all of the same columns as the training data set did. However, the output from the fit model is usually much simpler, often consisting of a predicted *numeric value* or *categorical label* for each individual sampling unit of the data set.

- Fit model : test data $\rightarrow$ predicted values

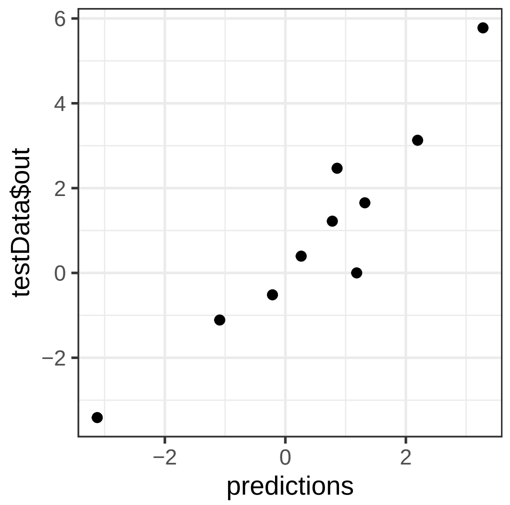
It is often convenient to package an ML algorithm up into a `function`; this is the most common strategy employed by ML algorithms implemented in R. A classic example is the `lm` function in R:

```
> set.seed(123)
> n = 20
>  ## generate some random data for two variables:
> predictor1 = rnorm(n)
> predictor2 = rnorm(n)
>  ## now set a third variable equal to a weighted sum of those
>  ## two variables plus a random error term:
> output = 2*predictor1 + predictor2 + rnorm(n)
>  ## bundle up the three variables composing our data set into a
>  ## data.frame object:
> allData = data.frame(p1=predictor1, p2=predictor2, out=output)
>  ## split allData into training and test sets:
> trainData = allData[1:10, ]
> testData = allData[11:20, ]  ## should not overlap trainData!
>  ## now train model using only trainData:
> fitModel = lm(out ~ p1 + p2, data=trainData)
```

Now we can use the `fitModel` to make predictions on rows 11–20 of `allData`; in R, this is usually, though not universally, done by calling the `predict` function with

1. the object representing the fit model as first argument and
2. the test data set—usually in the form of a `data.frame`—as the second argument:

```
>  ## generate predictions for test data:
> predictions = predict(fitModel, testData)
> library(ggplot2); theme_set(theme_bw())
>  ## plot actual values of out column against predicted values
>  ## for the test data using ggplot2::qplot
> qplot(predictions, testData$out)
```



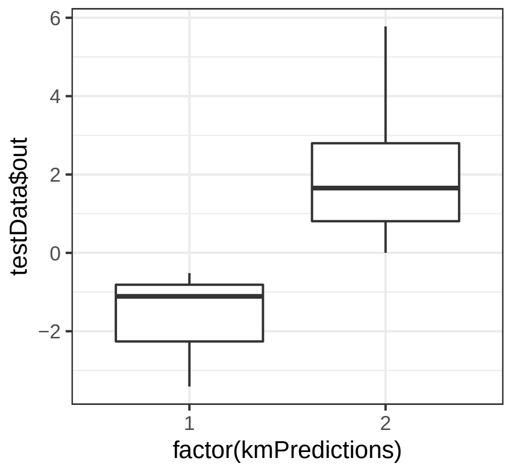
This an example of *supervised learning*, in which one of the variables in the training data set (`out` in this case) is treated as an output to be predicted using the others. The test set does not actually need to have this variable present to make predictions:

```
> predictions2 = predict(fitModel, testData[ , c("p1", "p2")])
>  ## these predictions should be the same as before:
> all(predictions2 == predictions)
[1] TRUE
```

Thus, in supervised learning approaches the fit model requires only a subset of the variables present in the training data to be present in the test data in order to make predictions.

In *unsupervised learning* this is not the case, and we must generally have all variables from the training data also present in any test data that we wish to make predictions on. What is this “unsupervised learning”, you ask, and what might it be used to predict? Let’s consider an example to make things more concrete:

```
>  ## need clue library to make predictions from kmeans fit
>  ## install.packages("clue")  ## uncomment and run if necessary
> library(clue)
>  ## use k-means clustering algorithm to fit 2 clusters to training data
> kmeansFit = kmeans(trainData, 2)
>  ## inspect kmeansFit object in R terminal:
> kmeansFit
K-means clustering with 2 clusters of sizes 4, 6

Cluster means:
          p1         p2        out
1 -0.6856418 -0.5676406 -2.0862528
2  0.5814706 -0.3291710  0.9174634

Clustering vector:
 1  2  3  4  5  6  7  8  9 10
 1  1  2  2  2  2  2  1  1  2

Within cluster sum of squares by cluster:
[1]  4.551067 11.361760
 (between_SS / total_SS =  61.7 %)

Available components:

[1] "cluster"      "centers"      "totss"        "withinss"     "tot.withinss"
[6] "betweenss"    "size"         "iter"         "ifault"
>  ## use clue::cl_predict instead of ordinary predict with kmeans
>  ## to predict which cluster each test datum is in
> kmPredictions = cl_predict(kmeansFit, testData)
> kmPredictions
Class ids:
 [1] 2 2 2 2 1 2 2 1 2 1
>  ## two clusters in this case correspond to low and high values of "out":
> qplot(factor(kmPredictions), testData$out, geom="boxplot")
```



As seen in this example, unsupervised learning algorithms try to find some latent structure in the training data—such as the carving of the variable space (frequently called *feature space* in ML) into two disjoint clusters done by `kmeans`, about which more will be said later.

Many unsupervised learning algorithms, including `kmeans`, produce fit models which can be used to determine how test data would fit into the learned latent structure; for instance, here we were able to assign each test datum to one of the two clusters learned from the training data set. There are some unsupervised learning approaches which generate fit models which are not immediately equipped to make test set predictions, however—hierarchical clustering and tSNE come to mind here—which can limit their utility in some situations.

2 Data

Machine learning—perhaps I should lose the qualifier and just say learning—isn't much without data!

We're going to see how machine learning algorithms work by applying them to both real and simulated data. It's critical to play with real data in learning machine learning, as it is very difficult to replicate many important features of real data via simulation. Simulation does play an important role in ML as well, however: only with simulated data can we check how our algorithms perform when all of the assumptions that underlie their derivation are truly met. It is also frequently much easier to "turn the knobs" on various data set properties of interest—like the number of sampling units n, the number of features m, the degree of correlation between features, etc.—with simulated data than in the lab or the external world!

We will consider two real gene expression data sets:

Neves an RNA-seq data set downloaded from Gene Expression Omnibus (accession GSE120430) analyzing transcriptional targets of core promoter factors in Drosophila neural stem cells (Neves & Eisenman (2019)).

Hess a microarray (!) data set from 2006 collected to predict sensitivity to preoperative chemotherapy using expression levels measured in fine-needle breast cancer biopsy specimens (Hess *et al.* (2006)).

I'll defer further discussion of the Hess data set until we get to supervised analysis methods.

In order to read in the data from file, I'm going to define a convenience function resetting some of the defaults of the `read.table` function:

```
> rt = function(f) {
      read.table(f, sep="\t", row.names=1, header=TRUE,
                 check.names=FALSE, comment.char="", quote="")
  }
```

Now let's use this function to read the Neves data set, along with a file containing Drosophila melanogaster gene annotations, in from the files included here in the github project:

```
> nevesExpr = log2(rt("data/gse120430_deseq_normalized.tsv.gz") + 1)
> nevesExpr[1:5, 1:5]
            SRR7900135 SRR7900136 SRR7900137 SRR7900138 SRR7900139
FBgn0002121  10.665165  11.215162  11.087356   9.465822  12.205824
FBgn0031209   1.488065   4.803333   4.389849   5.156091   4.505843
FBgn0051973   6.875664   7.784755   7.932583   5.667286   5.446360
FBgn0067779   8.348385   8.908152   8.552003   7.828771   8.974554
FBgn0031213   7.918769   7.467841   5.440411   6.242755   7.926314
>  ## (note that gene expression matrix files are usually provided
>  ##  using genes-in-rows format)
>  ## simplify nevesExpr by removing genes with no data:
> nevesExpr = nevesExpr[rowSums(nevesExpr) > 0, ]
>  ## by contrast, sample annotation files generally follow the
>  ## older statistics convention of sampling units-in-rows
> nevesAnnot = rt("data/gse120430_sample_annotation.tsv")
> dmGenes = rt("data/d_melanogaster_gene_annotations.saf.gz")
```

Let's take a quick look at `nevesAnnot`:

```
> nevesAnnot
             group
SRR7900135 mCherry
SRR7900136 mCherry
SRR7900137 mCherry
SRR7900138    TAF9
SRR7900139    TAF9
SRR7900140    TAF9
SRR7900141     TBP
SRR7900142     TBP
SRR7900143     TBP
SRR7900144    TRF2
SRR7900145    TRF2
SRR7900146    TRF2
```

To minimize the chance of any bugs in our analysis code, it is useful to align the rows of the sample annotation data (and gene annotation data, if we have it) to the columns of the expression matrix:

```
>  ## align sample annotations to expression data:
> nevesAnnot = nevesAnnot[colnames(nevesExpr), , drop=FALSE]
>  ## align dmGenes to expression data:
> dmGenes = dmGenes[rownames(nevesExpr), ]
```

The `group` column indicates whether each sample is in group expressing the control (mCherry) or one of the experimental RNAi transgenes (TAF9, TBP, or TRF2).

The sample names in the expression data and sample annotations are Gene Expression Omnibus accession ids; we'll replace these with more descriptive names based on the grouping information in the sample annotations:

```
>  ## use more descriptive names for samples
> betterSampleNames = paste0(nevesAnnot$group, "-", 1:3)
> colnames(nevesExpr) = betterSampleNames
> rownames(nevesAnnot) = betterSampleNames
```

Finally, because the descriptive gene names for the measured Drosophila genes are in one-to-one correspondence with the Flybase gene ids used to label the rows in the file `data/gse120430_deseq_normalized.tsv.gz`, we'll swap them out:

```
>  ## use more descriptive names for genes
> rownames(nevesExpr) = dmGenes$GeneName
```

The code shown above for loading in the Neves data set is also contained in the file `load_neves.R`.

3 k-Means Clustering

As a first example of a simple unsupervised ML algorithm, let's consider k-means clustering (MacQueen (1967)):

1. Initialize k "centroids" $\mathbf{c}_a$
   - bold font indicates a vector;
   - subscript a denotes which cluster and ranges from 1 to k.
2. Assign sampling unit with feature vector $\mathbf{x}_i$ to nearest cluster:
$$\text{clust}(\mathbf{x}_i) = \arg\min_a \|\mathbf{x}_i - \mathbf{c}_a\| \tag{1}$$
   - "arg min" with a below it looks at the expression to the right ($\|\mathbf{x}_i - \mathbf{c}_a\|$ here) and returns the value of a which minimizes it,
   - that is, the cluster a such that the centroid $\mathbf{c}_a$ is closest to $\mathbf{x}_i$.
3. Reset centroids to mean of associated data:
$$\mathbf{c}_a = \frac{1}{|S_a|} \sum_{i \in S_a} \mathbf{x}_i \tag{2}$$
   - where the set
$$S_a = \{i \mid \text{clust}(\mathbf{x}_i) = a\}$$
     contains all sampling units i assigned to cluster a.
   - $|S_a|$ is defined as the number of elements in set S_a.
4. Repeat steps 2-3 until convergence (i.e. the clusters don't change anymore).

(There are many animations of the k-means algorithm in action available online—try googling "k-means clustering animation" if you're interested in seeing a few of them.)

Let's apply the algorithm, as implemented in the R function `kmeans` to the Neves data set:

```
> set.seed(123)
> kmFit = kmeans(t(nevesExpr), centers=4)
```

In order to make predictions—meaning cluster assignments here—on an arbitrary data set, we'll need to use the `cl_predict` function from the `clue` package:

```
> ## install.packages("clue")  ## uncomment and run if necessary
> library(clue)  ## for making predictions from kmeans object
> kmPreds = cl_predict(kmFit, t(nevesExpr))
> head(kmPreds)
[1] 1 1 1 3 3 3
```

The predicted values `kmPreds[i]` are the just the cluster clust$(\mathbf{x}_i)$ for each sample i from 1 to $n = 12$ in the Neves data set. How do they compare with the Neves sample groupings?

```
>  ## table function counts how many combinations there are of the
>  ## discrete values occurring in one or more vectors passed in
>  ## as arguments:
> table(cluster=kmPreds, group=nevesAnnot$group)
```

```
       group
cluster mCherry TAF9 TBP TRF2
      1       3    0   0    1
      2       0    0   0    2
      3       0    3   0    0
      4       0    0   3    0
```

So we see that this simple unsupervised clustering approach finds clusters mostly—though not completely—the same as the sample experimental groupings.

The k-means algorithm has an interesting statistical interpretation: the solution locally minimizes

$$\sum_{a=1}^{k} \sum_{i \in S_a} (\mathbf{x}_i - \mathbf{c}_a)^2 \tag{3}$$

which may be regarded as a sum of squared errors if you regard centroid $\mathbf{c}_a$ as the predicted feature vector for all sampling units i assigned to cluster a.

The link between k-means clustering and statistics suggested by Eq (3) can be understood in more depth by deriving the method as an asymptotic limiting case of probabilistic mixture-of-Gaussians model (Ghahramani (2004)) (where each Gaussian in the mixture has its own centroid vector $\mathbf{c}_a$ but all share a common spherical covariance matrix $\sigma^2\underline{\mathbf{I}}$ and σ is vanishingly small).

This derivation explains why, despite being fast and intuitive, k-means clustering tends to produce (hyper)spherical, equal-sized clusters whether they are appropriate or not. In real data sets this is often at least somewhat problematic!

4 Hierarchical Clustering

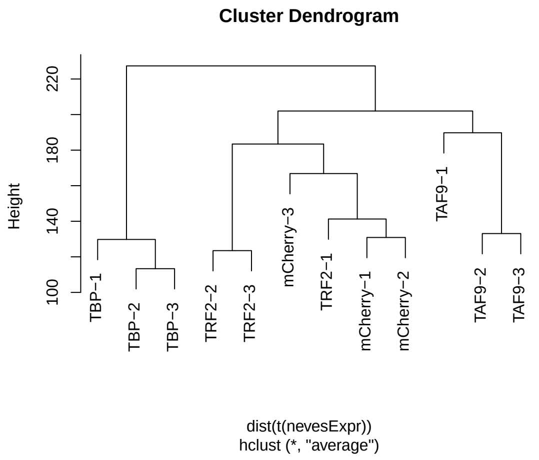
Probably the most popular unsupervised clustering method in bioinformatics is *agglomerative hierarchical clustering* (Mary-Huard *et al.* (2006); Hastie *et al.* (2009)). Hierarchical clustering approaches are so named because they seek to generate a hierarchy of clusterings of the data—generally represented as *dendrogram*, a structure to be discussed shortly.

A hierarchy of clusterings is a set of clusterings with, at the lowest level, n distinct clusters—so that no two objects are assigned to the same cluster—followed by a clustering with $n-1$ clusters, in which exactly two objects are assigned to the same cluster, and then a clustering with $n-2$ clusters, and so on, until finally the top level has only one cluster to which all n objects are assigned.

Each level of the hierarchy also must be consistent with the level below it: this means that the clustering with $m < n$ clusters must be the result of taking the clustering with $m+1$ clusters and merging two of those $m+1$ clusters together into one. This constraint is what makes it possible to represent the hierarchy with a dendrogram; let's consider an example:

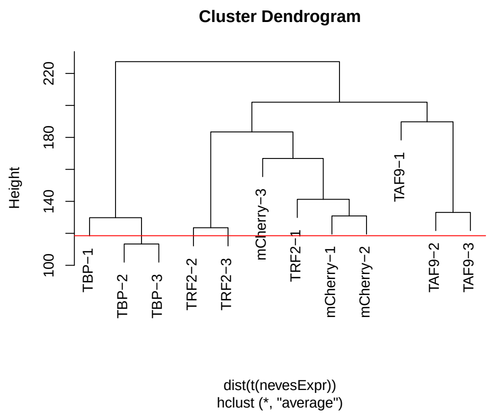
```
>  ## use hclust function to perform hierarchical clustering in R
> sampleClust = hclust(
      dist(t(nevesExpr)),    ## will discuss both of these arguments
      method = "average"     ## below!
```

```
  )
>  ## can use generic plot function to generate dendrogram from hclust
> plot(sampleClust)
```



The different clusterings correspond to different vertical levels in this dendrogram. At the very bottom—below all of the lines—each of the samples are assigned to its own cluster. Then, at the level indicated by the red line here:

```
> plot(sampleClust)
> abline(h=118.5, col="red")  ## draw red horizontal line at y=118.5
```



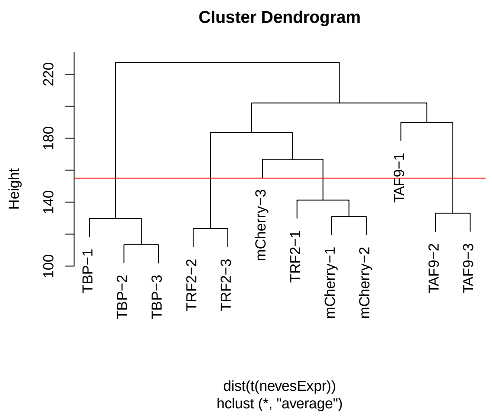
we have joined the two samples TBP-2 and TBP-3 together into a single cluster, since the lines connecting these two samples are below the red line, while each of the other 10 samples is still assigned to its own cluster.

We can also extract the cluster identities directly in R without bothering to look at plots:

```
> cutree(sampleClust, h=118.5)  ## cut tree at height 118.5
mCherry-1 mCherry-2 mCherry-3    TAF9-1    TAF9-2    TAF9-3     TBP-1     TBP-2
        1         2         3         4         5         6         7         8
    TBP-3    TRF2-1    TRF2-2    TRF2-3
        8         9        10        11
>  ## output is vector containing sample cluster labels
>  ## note TBP-2 and TBP-3 are both assigned to cluster 8,
>  ## while all other samples get their own cluster id number
```

Alternatively, if we try a different height cutoff:

```
> cutree(sampleClust, h=155)
mCherry-1 mCherry-2 mCherry-3    TAF9-1    TAF9-2    TAF9-3     TBP-1     TBP-2
        1         1         2         3         4         4         5         5
    TBP-3    TRF2-1    TRF2-2    TRF2-3
        5         1         6         6
> plot(sampleClust)
> abline(h=155, col="red")
```



We find:

- on the far left, cluster **5**, containing TBP-1, TBP-2, and TBP-3, then
- cluster **6** containing TRF2-2 and TRF2-3, followed by
- cluster **2** containing only mCherry-3,
- cluster **1** containing TRF2-1, mCherry-1, mCherry-2,
- cluster **3** containing only TAF9-1, and, finally,
- on the far right, cluster **4** containing TAF9-2 and TAF9-3.

Often when we want a specific clustering, we want to specify the number of clusters instead of trying to figure out what height to cut at; this can be done with `cutree` using the `k` argument instead of the `h` argument:

```
> cutree(sampleClust, k=6)  ## generates same 6 clusters as h=155 did
mCherry-1 mCherry-2 mCherry-3    TAF9-1    TAF9-2    TAF9-3     TBP-1     TBP-2
        1         1         2         3         4         4         5         5
    TBP-3    TRF2-1    TRF2-2    TRF2-3
        5         1         6         6
```

4.1 Dissimilarity metrics

When we first ran `hclust`, we supplied two arguments; the first of these was `dist(t(nevesExpr))`. The `t(nevesExpr)` part of this simply takes the transpose of `nevesExpr`, but we haven't seen the function `dist` before, so let's take a look:

```
> dist(t(nevesExpr))
```

```
          mCherry-1 mCherry-2 mCherry-3   TAF9-1   TAF9-2   TAF9-3    TBP-1
mCherry-2  130.8878                                                         
mCherry-3  173.2262  152.9425                                               
TAF9-1     194.7300  185.8067  210.7132                                     
TAF9-2     195.0815  203.3795  242.9921 181.8847                            
TAF9-3     187.4437  208.0922  245.9061 197.6139 133.0881                   
TBP-1      204.1339  191.9169  209.7321 206.5450 206.9972 206.9326          
TBP-2      234.7608  219.5921  235.1969 236.4809 221.3995 225.0745 130.5170
TBP-3      236.9017  221.8174  236.4044 235.8578 217.7286 223.7864 129.0292
TRF2-1     146.2989  136.2433  174.3065 178.9488 186.7003 200.9020 199.0670
TRF2-2     150.6845  170.6102  219.5452 218.2950 186.1074 174.1057 232.3462
TRF2-3     168.9906  184.9297  230.0823 226.9680 196.7015 193.0967 247.2684
             TBP-2    TBP-3   TRF2-1   TRF2-2
mCherry-2                                    
mCherry-3                                    
TAF9-1                                       
TAF9-2                                       
TAF9-3                                       
TBP-1                                        
TBP-2                                        
TBP-3     113.3402                           
TRF2-1    222.2697 221.4280                  
TRF2-2    254.1267 254.2897 167.9877         
TRF2-3    269.0995 267.7580 174.6102 123.4922
```

What we've done here is to compute the *Euclidean distances* of each of the 12 samples from each of the other 11 samples. The Euclidean distance is defined here as in geometry as the square root of the sum of the squared differences in each coordinate of a vector; since this is more easily comprehended via math or code than English words,

```
> coordinateDifferenceSample1Sample2 = nevesExpr[ , 1] - nevesExpr[ , 2]
> sqrt( sum( coordinateDifferenceSample1Sample2^2 ) )
[1] 130.8878
>  ## results in same value as
> as.matrix(dist(t(nevesExpr)))[1, 2]
[1] 130.8878
```

We want these distances here as a way to measure how dissimilar one sample's expression profile is from another (Euclidean distance is not the only dissimilarity metric which can be used with `hclust`; you can consult the help documentation for `dist` function and its own `method` argument to see what other options are available). The agglomerative hierarchical clustering algorithm implemented by `hclust` uses these quantified dissimarilities between pairs of samples to decide, at each step, which two clusters to join together from the clustering with `m+1` clusters to form the clustering with `m` clusters.

This is easiest to do in the first step, where we start with every sample in its own cluster. In this case, we just pick the two samples with the smallest dissimilarity value (in this case, TBP-2 and TBP-3, with a dissimilarity score of 113.34 between them) and join them together into a cluster.

4.2 Agglomeration Linkage Methods

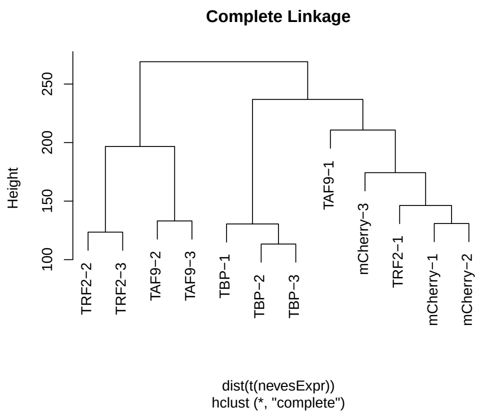
After we've created a cluster containing two separate objects a new problem confronts us: How do we decide in the next round of clustering whether to join together two singleton objects (objects which are in their own cluster not containing any other objects) or instead to join a singleton object into the two-object cluster we created in our first step?

We need a way to assign numeric dissimilarities between *clusters* of objects based on the numeric dissimilarities we've already calculated between individual objects. In the example I've constructed here, I've done this using a very simple approach: The dissimilarity between cluster A and cluster B is defined to be the average of the dissimarilities between all pairs of objects we can form taking one object from cluster A and the other object in the pair from cluster B. This is the meaning of the code `method = "average"` in the `hclust` call above.

(Note: we supplied the second argument to `hclust` using the argument name `method`; keep in mind this is distinct from the `method` argument of the `dist` function referenced above—different functions can use the same argument names to mean different things, since they operate in different scopes.)

This way of defining dissimilarities between clusters based on dissimilarities between objects is known as "average linkage." Many alternatives exist; one particularly common one (the default `method` for `hclust`) is "complete linkage." Complete linkage defines the dissimilarity between cluster A and cluster B as the largest dissimilarity value for all pairs of objects taken one from A and the other from B:

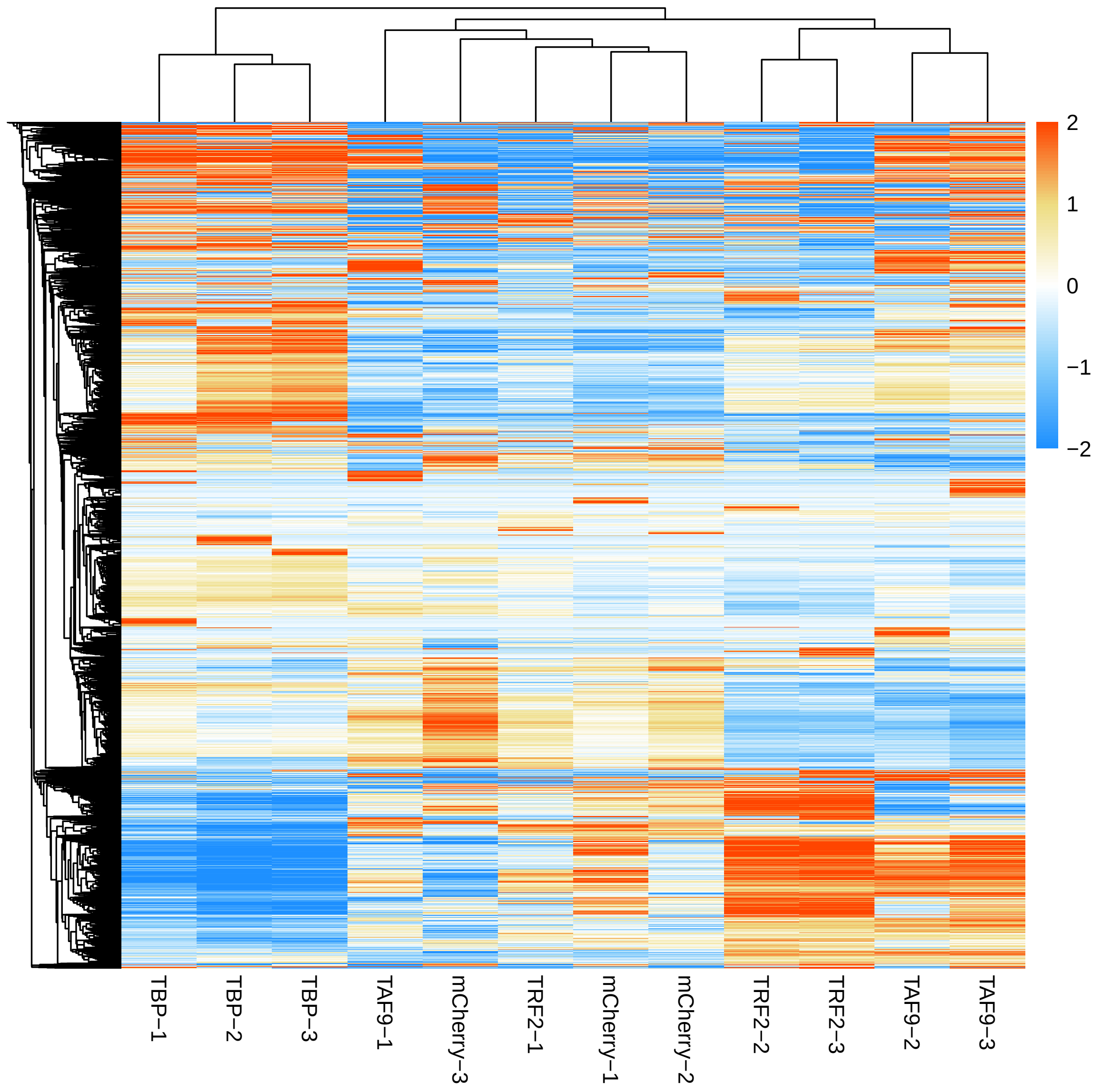
```
> sampleClust2 = hclust(dist(t(nevesExpr)), method="complete")
> plot(sampleClust2, main="Complete Linkage")
```



With complete linkage we can see that at the higher levels of the dendrogram we obtain different clusterings than we did with average linkage. In particular, with average linkage the three samples TBP-1, TBP-2, and TBP-3—are the last to be merged together with the remainder of the sample set, while with complete linkage this is not the case.

4.3 Clustered Heatmaps

```
> ## install.packages("pheatmap")  ## uncomment and run if necessary
> library(pheatmap)
>  ## usually most interested in expression levels relative to mean:
> heatData = nevesExpr - rowMeans(nevesExpr)
>  ## often want to limit dynamic range heatmap considers so that
>  ## color palette range is not dominated by a few extreme values:
> heatData[heatData > 2] = 2; heatData[heatData < -2] = -2
>  ## pheatmp is not a grammar-of-graphics style plotting function:
>  ## specify all options as arguments to single function call
>  ## intead of building plot up in modular fashion:
> pheatmap(
      heatData,
      color = colorRampPalette(c(
          "dodgerblue", "lightskyblue",
          "white",
          "lightgoldenrod", "orangered"
      ))(100),
      clustering_method = "average",
      show_rownames = FALSE
  )
```



4.4 Predicting Clusters from Hierachical Clustering

So...how do we predict the cluster assignment for a new sampling unit with feature vector $\mathbf{x}$ at each level of the clustering hierarchy?

We don't. As I mentioned in the introduction, hierarchical clustering is one of the few machine learning algorithms which doesn't really fit into the "algorithms to learn algorithms to predict" scheme.

5 Principal Component Analysis

(NOTE: in discussion of PCA, I will reserve symbol x for PCA *scores* as opposed to feature values, to accord with R's use of `pca$x`. I will shift *for this section only* to use of z for feature values. I will make one exception at the end in defining the `extractPCs` function, as it will be used again in later sections where I will return to use of x for feature values.)

There are many different ways to describe the underyling idea of PCA (Roweis & Ghahramani (1999); Izenman (2008)); here's one: PCA fits a series of *principal components* to model the expression levels z_{ig} of all genes g across all samples i. We'll start with a single principal component (PC1) model:

$$z_{ig} = \mu_g + x_{i1} r_{g1} + e_{ig}^{(1)} \tag{4}$$

where:

- $\mu_g = \frac{1}{n} \sum_i z_{ig}$ is the mean expression level of gene g,
- x_{i1} is the "score" of sample i on PC1,
- r_{g1} is the "loading" of gene g on PC1, and
- $e_{ig}^{(1)}$ is the error residual for gene g on sample i using PC1 model.

Fitting PC1 thus requires estimating x_{i1} for all samples i and r_{g1} for all genes g. This is generally done so as to minimize the sum of squared residuals $\sum_{i,g} \left(e_{ig}^{(1)} \right)^2$ (PCA is another least-squares algorithm). It so happens that there is a beautifully elegant and efficient algorithm solving just this problem via something known in linear algebra as singular value decomposition (SVD) implemented in the R function `prcomp`:

```
>  ## use prcomp function for PCA in R
>  ## on transposed version of nevesExpr, t(nevesExpr),
>  ## since prcomp assumes variables (genes) are in columns, not rows
>  ## * z_ig = nevesExpr[g, i] = t(nevesExpr)[i, g]
> pca = prcomp(t(nevesExpr))
> class(pca)
[1] "prcomp"
> is.list(pca)
[1] TRUE
> names(pca)
[1] "sdev"     "rotation" "center"   "scale"    "x"
>  ## the PCA scores x_i1 for samples are found in pca$x:
> head(pca$x[ , 1, drop=FALSE])  ## drop=FALSE: keep as matrix w/ one column
                  PC1
mCherry-1 -59.63926
mCherry-2 -33.37750
mCherry-3 -10.70811
TAF9-1    -10.99583
```

```
TAF9-2    -14.34377
TAF9-3    -21.33802
```

```
>  ## the PCA loadings r_g1 for genes are found in pca$rotation:
> head(pca$rotation[ , 1, drop=FALSE])
```

```
                   PC1
l(2)gl    2.445891e-03
Ir21a     2.345373e-03
Cda5     -7.300728e-05
dbr       7.092041e-03
galectin  9.349290e-03
CG11374   2.069868e-02
```

You might notice that I extracted only the first column from both `pca$x` and `pca$rotation`. This is because these matrices contain PCs beyond PC1. For instance, the second column of each of these matrices corresponds to PC2, which is defined by

$$z_{ig} = \mu_g + x_{i1} r_{g1} + x_{i2} r_{g2} + e_{ig}^{(2)} \tag{5}$$

where PC1 is obtained from the single-PC model and PC2 is then again fit to minimize the remaining $\sum_{i,g} \left(e_{ig}^{(2)} \right)^2$.

This process can be repeated to obtain higher order PCs as well; in general, the k^{th} PC has

- scores x_{ik} which can be found in the R object `pca$x[i, k]` and
- loadings r_{gk} stored in `pca$rotation[g, k]`

and minimizes the squared sum of the error residuals

$$e_{ig}^{(k)} = z_{ig} - \mu_g - \sum_{j=1}^{k} x_{ik} r_{gk} \tag{6}$$

It is worth noting that after fitting n PCs (recall n is the number of samples, 12 here), there is no error left—that is, $e_{ig}^{(n)} = 0$—so that we will have a perfect fit for z_{ig}:

$$z_{ig} = \mu_g + \sum_{j=1}^{n} x_{ik} r_{gk} \tag{7}$$

$$= \mu_g + \left[\underline{\mathbf{X}\mathbf{R}}^{\mathrm{T}} \right]_{ig} \tag{8}$$

where in Eq (8) we have switched over to matrix notation. This can be confirmed in R via:

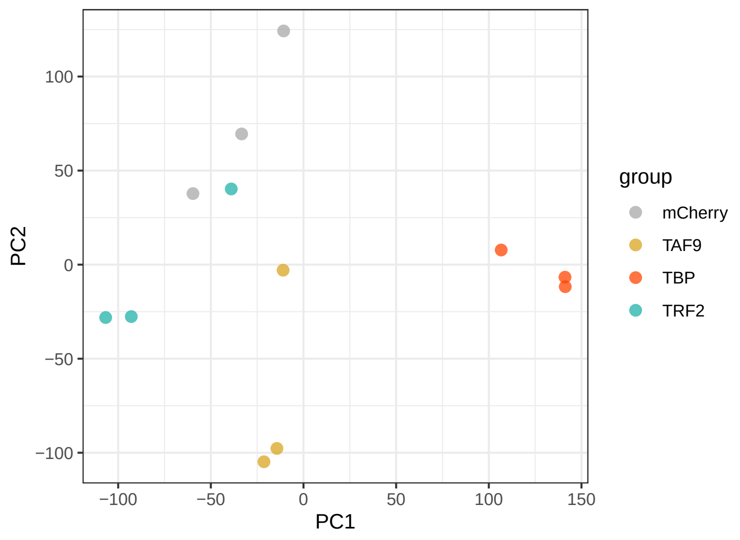
```
> pcaFit = rowMeans(nevesExpr) + t( pca$x %*% t(pca$rotation) )
>  ## have to transpose pca$x %*% t(pca$rotation) above b/c nevesExpr is t(z)
> nevesExpr[1:3, 1:3]
```

```
       mCherry-1 mCherry-2 mCherry-3
l(2)gl 10.665165 11.215162 11.087356
Ir21a   1.488065  4.803333  4.389849
Cda5    6.875664  7.784755  7.932583
```

```
> pcaFit[1:3, 1:3]  ## same thing!
       mCherry-1 mCherry-2 mCherry-3
l(2)gl 10.665165 11.215162 11.087356
Ir21a   1.488065  4.803333  4.389849
Cda5    6.875664  7.784755  7.932583
```

By now you may be wondering what we do with all of these sample scores x_{ik} and gene loadings r_{gk}. Well, the first thing we usually do is make a PCA plot by plotting x_{i2} (which we usually label as simply “PC2”) on the vertical axis against x_{i1} (“PC1”) on the horizontal axis:

```
>  ## set up data.frame pcaData for ggplot...
> pcaData = data.frame(pca$x[ , 1:2])
>  ## add in sample annotation info
> pcaData$group = nevesAnnot[rownames(pcaData), "group"]
>  ## and sample names
> pcaData$sample = rownames(pcaData)
>  ## make sure ggplot2 library is loaded
> library(ggplot2)
>  ## bw theme...
> theme_set(theme_bw())
> gg = ggplot(pcaData, aes(x=PC1, y=PC2, color=group, label=sample))
> gg = gg + geom_point(size=2.5, alpha=0.75)
> colVals = c("darkgray", "goldenrod", "orangered", "lightseagreen")
> gg = gg + scale_color_manual(values=colVals)
> print(gg)
```



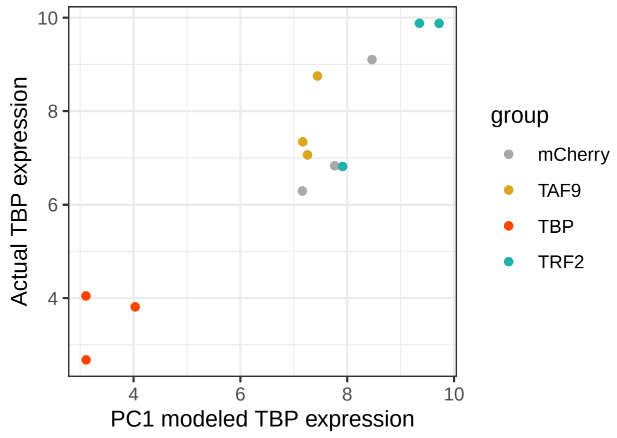
This shows us something interesting: despite PCA knowing nothing about the sample groupings, it has fit PC1 so as to split the TBP experimental group apart from all others (in the sense that the TBP group samples have large positive scores x_{i1} while all other samples have negative PC1 scores). This tells us that, in some sense, TBP is the most different sample group relative to all of the others.

5.1 Modeling Expression Levels with First PC (or Two)

I introduced PCA as a way to model expression levels using multiplicative factors of sample scores and gene loadings. You might well wonder how good it is as such a model, so let's go ahead and look at the expression patterns (sometimes called "profiles") of a gene or two.

Let's start with the gene TBP:

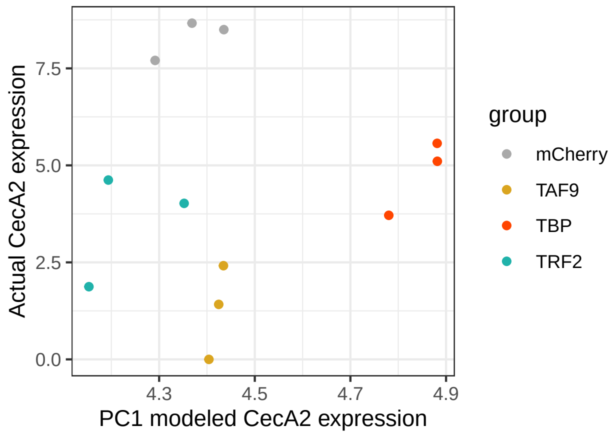
```
> z = t(nevesExpr)
> geneIndex = which(rownames(nevesExpr) == "Tbp")
> tbpData = data.frame(
      pc1model = mean(z[ , "Tbp"]) +
                 pca$x[ , 1] * pca$rotation[geneIndex, 1],
      actual = z[ , "Tbp"],
      group = nevesAnnot$group
  )
> tbpPlot = ggplot(tbpData, aes(pc1model, actual, color=group))
> tbpPlot = tbpPlot + geom_point()
> tbpPlot = tbpPlot + scale_color_manual(values=colVals)
> tbpPlot = tbpPlot + xlab("PC1 modeled TBP expression")
> tbpPlot = tbpPlot + ylab("Actual TBP expression")
> print(tbpPlot)
```



The PC1 (only) model (calculated based on Eq (4) above) for TBP expression appears to do a pretty job! But perhaps we should be suspicious that this performance may not be totally representative, given that we noted that PC1 split the "TBP" sample group out from the other samples. Indeed, recall that this sample group is defined by the presence of an experimental RNAi transgene for TBP, and indeed we see that expression of the TBP gene itself is quite significantly reduced in this sample group relative to all others.

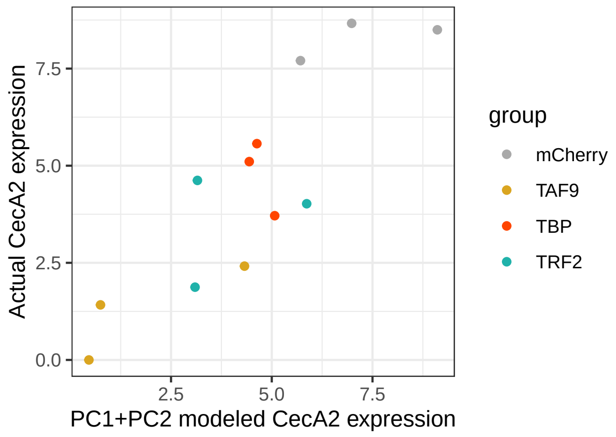
So let's consider a different gene, CecA2:

```
> geneIndex = which(rownames(nevesExpr) == "CecA2")
> ceca2Data = data.frame(
      pc1model = mean(z[ , "CecA2"]) +
                 pca$x[ , 1] * pca$rotation[geneIndex, 1],
      actual = z[ , "CecA2"],
      group = nevesAnnot$group
  )
> ceca2Plot = ggplot(ceca2Data, aes(pc1model, actual, color=group))
> ceca2Plot = ceca2Plot + geom_point()
> ceca2Plot = ceca2Plot + scale_color_manual(values=colVals)
> ceca2Plot = ceca2Plot + xlab("PC1 modeled CecA2 expression")
> ceca2Plot = ceca2Plot + ylab("Actual CecA2 expression")
> print(ceca2Plot)
```



So...PC1 model does not do so well this time! It can't, because `pca$x[ , 1]` assigns the most extreme values to samples from the TBP group, while the actual expression levels of CecA2 in the TBP sample group are quite middle-of-the-road. But don't despair, we can always try PC1+PC2 model as defined by Eq (5):

```
> ceca2Data$pc1and2model = mean(z[ , "CecA2"]) +
                           pca$x[ , 1] * pca$rotation[geneIndex, 1] +
                           pca$x[ , 2] * pca$rotation[geneIndex, 2]
> ceca2Plot = ggplot(ceca2Data, aes(pc1and2model, actual, color=group))
> ceca2Plot = ceca2Plot + geom_point()
> ceca2Plot = ceca2Plot + scale_color_manual(values=colVals)
> ceca2Plot = ceca2Plot + xlab("PC1+PC2 modeled CecA2 expression")
> ceca2Plot = ceca2Plot + ylab("Actual CecA2 expression")
> print(ceca2Plot)
```



Maybe not perfect, but definitely much better! In case you're curious as to why I picked CecA2 in particular, it is the gene with the largest positive loading on PC2 (as you can confirm by running `which.max(pca$rotation[ , 2])` if you're so inclined). Thus we might have expected it to be a gene for which the PC1+PC2 model would be notably better than a PC1-only model.

5.2 Percent Variation Explained by PC k

This leads us to a generally ill-understood point in PCA: quantification of how much each PC contributes to modeling the gene expression values. Perhaps you recall the use of R^2 to characterize the quality of fit for linear models (regression, ANOVA, etc.) in statistics; we can use this idea in PCA as well, but we have to decide what to do about the fact that it is a truly *multivariate* model: We are simultaneously modeling thousands of genes!

The general approach taken is more or less the simplest generalization of the R^2 idea available. Recalling that R^2 gives the "percent of variation explained," quantified as the percentage reduction in sum of squared residuals, we first define

$$\sigma_0^2 = \frac{1}{n-1} \sum_{i,g} (z_{ig} - \mu_g)^2 \tag{9}$$

$$\sigma_k^2 = \frac{1}{n-1} \sum_{i,g} \left(e_{ig}^{(k)}\right)^2 \tag{10}$$

where the k^{th} matrix of error residuals $e_{ig}^{(k}$ is defined by Eq (6). We can regard Eq (9) as just Eq (10) with $k = 0$ if we accept the natural definition

$$e_{ig}^{(0)} = z_{ig} - \mu_g \tag{11}$$

for the error residuals of a "no-PC model." Then we can also rewrite Eq (6) for any $k > 0$ as as:

$$e_{ig}^{(k)} = e_{ig}^{(k-1)} - x_{ik} r_{gk} \tag{12}$$

Let's take a second to do a few of these calculations in R, using the name `resid0[i, g]` to represent $e_{ig}^{(0)}$, `resid1[i, g]` for $e_{ig}^{(1)}$, etc.:

```
> resid0 = t( nevesExpr - rowMeans(nevesExpr) )  ## matrix of residuals e^(0)_ig
>  ## now use outer function to construct matrix with i, g entry
>  ##  pca$x[i, 1] * pca$rotation[g, 1]
>  ## and subtract this from resid0 to obtain resid1:
> resid1 = resid0 - outer(pca$x[ , 1], pca$rotation[ , 1])
>  ## resid1 contains error residuals e^(1)_ig after fitting PC1-only model
> resid2 = resid1 - outer(pca$x[ , 2], pca$rotation[ , 2])
>  ## resid2 contains error residuals e^(2)_ig from PC1+PC2 model
```

and, using `sigmaSq0` to represent σ_0^2, `sigmaSq1` for σ_1^2, etc.:

```
> n = ncol(nevesExpr)  ## number of samples (12)
> sigmaSq0 = sum(resid0^2) / (n-1)
> sigmaSq1 = sum(resid1^2) / (n-1)
> sigmaSq2 = sum(resid2^2) / (n-1)
```

Now the variance explained by PC k is defined to be the overall reduction in variance associated with adding PC k into our model, and is given by

$$\Delta\sigma_k^2 = \sigma_{k-1}^2 - \sigma_k^2 \tag{13}$$

and the “fraction of variation” explained by PC k is finally

$$\frac{\Delta\sigma_k^2}{\sigma_0^2} = \frac{\Delta\sigma_k^2}{\sum_{k=1}^{n}\Delta\sigma_k^2} \tag{14}$$

where the right-hand side of Eq (14) holds because, as we verified above, the variance σ_n^2 remaining after fitting all n PCs is 0; that is

$$\begin{aligned}
0 &= \sigma_n^2 && (15)\\
&= \sigma_0^2 - \left(\sigma_0^2 - \sigma_1^2\right) - \left(\sigma_1^2 - \sigma_2^2\right) - \cdots - \left(\sigma_{n-1}^2 - \sigma_n^2\right) && (16)\\
&= \sigma_0^2 - \sum_{k=1}^{n}\left(\sigma_{k-1}^2 - \sigma_k^2\right) && (17)\\
&= \sigma_0^2 - \sum_{k=1}^{n}\Delta\sigma_k^2 && (18)\\
\sigma_0^2 &= \sum_{k=1}^{n}\Delta\sigma_k^2 && (19)
\end{aligned}$$

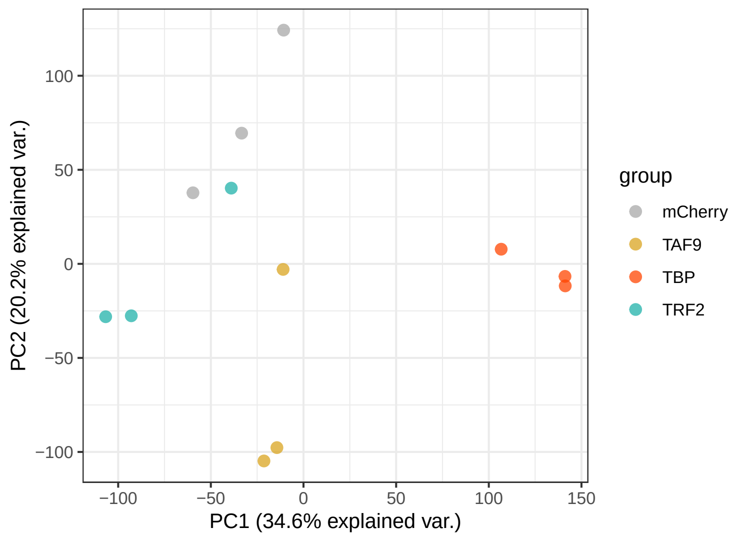
The output `pca` from `prcomp` stores the quantities $\sqrt{\Delta\sigma_k^2}$ in the field `pca$sdev`:

```
>  ## let's compare:
> sqrt(sigmaSq0 - sigmaSq1)
[1] 84.40337
> pca$sdev[[1]]  ## same thing!
[1] 84.40337
>  ## what about the second PC?
> sqrt(sigmaSq1 - sigmaSq2)
[1] 64.41385
> pca$sdev[[2]]  ## same thing!
[1] 64.41385
>  ## fraction of variation explained by PC1, "from scratch":
> (sigmaSq0 - sigmaSq1) / sigmaSq0
[1] 0.3464319
>  ## fraction of variation explained by PC1, using pca$dev:
> pca$sdev[[1]]^2 / sum(pca$sdev^2)  ## same thing!
[1] 0.3464319
```

```
>  ## fraction of variation explained by PC2, from scratch:
> (sigmaSq1 - sigmaSq2) / sigmaSq0
[1] 0.2017701
>  ## fraction of variation explained by PC2, using pca$dev:
> pca$sdev[[2]]^2 / sum(pca$sdev^2)  ## same thing!
[1] 0.2017701
```

So we see that the first PC in the `pca` model explains 34.6% of the overall gene expression variance, while the second PC explains an additional 20.2% of overall gene expression variance. Let's update the axis labels of our PCA plot to include this information:

```
> pctVar = 100 * pca$sdev^2 / sum(pca$sdev^2)
> gg = gg + xlab(
          paste0("PC1 (", round(pctVar[[1]], 1), "% explained var.)"))
> gg = gg + ylab(
          paste0("PC2 (", round(pctVar[[2]], 1), "% explained var.)"))
> print(gg)
```



Keep in mind that these percentages reflect the fraction of variance explained across *all* genes; some genes (e.g., TBP) may be much better explained by PC1 alone than are other genes (e.g., CecA2)!

5.3 Extracting Sampling Unit Scores

Here is a function for learning the PCs from a training set and returning a function `extractor` for assessing the sampling unit scores for a test set `newdata`:

```
>  ## arguments to extractPCs
>  ## - x is matrix or data.frame of feature values
>  ## - m is number of principal component features to extract
> extractPCs = function(x, m, ...) {
       ## assume x is samples-in-rows, genes-in-columns format!
       ## training-set-estimated mean expression of each gene:
      mu = colMeans(x)
      pca = prcomp(x, center=TRUE, scale.=FALSE)
       ## extract matrix needed to project new data onto first m extracted PCs:
      projection = pca$rotation[ , 1:m, drop=FALSE]
       ## define extraction function to extract features from new data:
      extractor = function(newdata) {
           ## sweep out training-set-estimated gene means:
          newdata = sweep(newdata, 2, mu, `-`)
          return(newdata %*% projection)
      }
       ## return the function "extractor" which can be applied to newdata;
       ## this function yields coordinates of samples in newdata in PC-space
       ## learned from the training data passed in as x argument.
      return(extractor)
  }
```

If you look at the math implemented by `extractPCs`, you'll find that it assigns to a sampling unit with feature vector $\mathbf{z}$ a PC-k score of:

$$\sum_g (z_g - \mu_g)\, r_{gk} \tag{20}$$

which is by definition the k^{th} component of the vector $\underline{\mathbf{R}}^{\mathrm{T}}(\mathbf{z} - \boldsymbol{\mu})$.

Note that there are two distinct ways in which PCA is, in a technical sense, an "algorithm which learns to predict":

1. It can be used to predict gene expression values (via the restriction of the sum in Eq (7) to the desired number of PCs), and
2. it can be used to "predict" (we would more commonly say *extract*) the location of a new sampling unit in the PC space defined by the training set via Eq (20).

I'll note that while both of these modes of prediction differ somewhat from the ordinary language meaning of prediction, the second does make PCA a very useful *feature extraction* method, as we will see a bit later!

6 Hess Data Set

The Hess data set I initially mentioned in the introductory notes consists of microarray data taken from fine-needle biopsies taken from breast cancer patients. A number of patient characteristics were collected, but the main focus of the modeling that we will be doing—like the modeling that Hess et al. were doing! (Hess *et al.* (2006))—will be the sensivity to preoperative chemotherapy, with the patients divided into those who exhibited residual disease (RD) or those who did not and were thus classified as have pathologic complete response (pCR) to chemotherapy.

In order to load the Hess data in, let's re-define the function `rt` we used to load the Neves data in before:

```
>  ## define convenience function for loading tabular data
>  ## (just using read.table with different default options)
> rt = function(f) {
      read.table(f, sep="\t", row.names=1, header=TRUE,
                 check.names=FALSE, comment.char="", quote="")
  }
```

Hess et al. obtained two separate data sets, a training set which they used to develop a classifier for RD-vs-pCR, and a test set which they used to assess the performance of the resulting classifier. Let's load in the training data first:

```
>  ## training set:
> hessTrain = rt("data/HessTrainingData.tsv.gz")
> hessTrainAnnot = rt("data/HessTrainingAnnotation.tsv")
>  ## align annotation data.frame with expression data:
> hessTrainAnnot = hessTrainAnnot[colnames(hessTrain), ]
```

And now the test data:

```
>  ## test set:
> hessTest = rt("data/HessTestData.tsv.gz")
> hessTestAnnot = rt("data/HessTestAnnotation.tsv")
>  ## align annotation data.frame with expression data:
> hessTestAnnot = hessTestAnnot[colnames(hessTest), ]
```

Taking a quick look at the training data:

```
> hessTrain[1:5, 1:5]
                   M157     M196     M176     M214     M113
AFFX-BioB-5_at  8.944566 8.377514 8.386897 8.292460 8.531966
AFFX-BioB-M_at  9.852639 9.342453 9.499607 9.295769 9.343919
AFFX-BioB-3_at  9.862932 9.237090 9.476301 9.290295 9.364047
AFFX-BioC-5_at 10.108551 9.489507 9.475409 9.355395 9.378707
AFFX-BioC-3_at 10.317820 9.594679 9.841014 9.678230 9.595369
```

we see that the rows here are not annotated by gene ids but instead by *probe set* ids. We'll load in the microarray probe annotations mapping these probe sets back to genes as well:

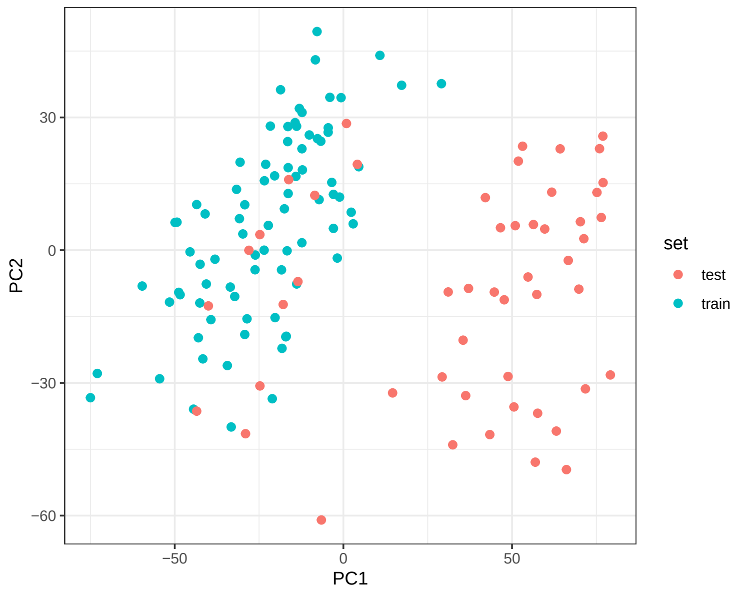
```
> probeAnnot = rt("data/U133A.tsv.gz")
>  ## align hessTrain and hessTest to probeAnnot:
> hessTrain = hessTrain[rownames(probeAnnot), ]
> hessTest = hessTest[rownames(probeAnnot), ]
```

For ease of classification, we'd like to extract the class labels from columns of the sample annotation files into `factor` variables:

```
> hessTrainY = factor(hessTrainAnnot$pCRtxt)
> names(hessTrainY) = rownames(hessTrainAnnot)
> hessTestY = factor(hessTestAnnot$pCRtxt)
> names(hessTestY) = rownames(hessTestAnnot)
```

Finally, let's take a quick look at the test and training set data put together via a PCA plot:

```
>  ## use cbind to column-bind the training and test data sets together:
> combinedData = cbind(hessTrain, hessTest)
> pca = prcomp(t(combinedData))
> library(ggplot2)
> theme_set(theme_bw())  ## get rid of gray backgrounds in ggplot
> ggdata = data.frame(
      PC1 = pca$x[ , 1],
      PC2 = pca$x[ , 2],
      set = c(rep("train", ncol(hessTrain)), rep("test", ncol(hessTest)))
  )
> gg = ggplot(ggdata, aes(x=PC1, y=PC2, color=set))
> gg = gg + geom_point(size=2)
> print(gg)
```



This shows an uncomfortable fact about real-world applications of machine learning: when training and test sets are collected separately, there tend to be systematic differences between them. This can result in degraded test-set performance even when models have been carefully constructed using the most effective algorithms available!

7 k-Nearest Neighbors (knn)

The k-neighest neighbors, or knn, algorithm (Cover (1968)) is a particularly simple and democratic approach to classification:

To classify sampling unit i with feature values x_{ig}:

- find the k sampling units $\{j_1, j_2, \ldots, j_k\}$ from the training set *most similar* to i: these are the "nearest neighbors"
- calculate the fraction $\frac{\sum_b y_{j_b}}{k}$ of the nearest neighbors which have $y_{j_b} = 1$: this is the knn model-predicted probability that $y_i = 1$.

```
>  ## extract vector of gene expression values for test sample 1:
> featuresForTestSample1 = hessTest[ , 1]
>  ## calculate distance of each training sample from test sample 1:
>  ## note: subtraction of vector from matrix is column-wise in R!
> euclideanDistancesFromTrainSamples = sqrt(
      colSums( (hessTrain - featuresForTestSample1)^2 )
  )
>  ## what are the 9 nearest neighbors
>  ## and their distances from test sample 1?
> nn = sort(euclideanDistancesFromTrainSamples)[1:9]
> nn
```

```
    M205     M154     M133     M211     M206     M111     M180     M136
75.76572 83.92735 85.01014 88.52586 88.73677 89.50107 89.54335 89.96744
    M179
93.13950
>  ## what are their classifications?
> hessTrainY[names(nn)]
M205 M154 M133 M211 M206 M111 M180 M136 M179
 pCR   RD   RD  pCR  pCR  pCR  pCR  pCR  pCR
Levels: pCR RD
>  ## 9-nn model predicted probability of RD for test sample 1 is:
> sum( hessTrainY[names(nn)] == "RD" ) / 9
[1] 0.2222222
```

Here I should hasten to point out that all of the ML algorithms we will study here have been been packaged up into more efficient and user-friendly R routines, so there is really no need to go through the pain of re-implementing them from scratch! (I just wanted to give you a sense of how simple knn in particular is "under the hood.")

Here is the way I would actually suggest to apply the knn algorithm in R (using the `knn3` function from the library `caret`):

```
> ## install.packages("caret")  ## uncomment and run if necessary
> library(caret)
>  ## fit model object obtained by running:
> knnFit = knn3(x = t(hessTrain),   ## knn3 wants features-in-columns
                y = hessTrainY,     ## recall hessTrainY is factor!
                k = 9)
>  ## can then generate test set predictions using knnFit:
> knnTestPredictionProbs = predict(knnFit, t(hessTest))
>  ## good to inspect results of predict method in R
>  ## (output from predict not standardized from one alg to next):
> head(knnTestPredictionProbs)
           pCR        RD
[1,] 0.7777778 0.2222222
[2,] 0.2222222 0.7777778
[3,] 0.2222222 0.7777778
[4,] 0.2222222 0.7777778
[5,] 0.3333333 0.6666667
[6,] 0.2222222 0.7777778
>  ## what is predicted probability RD for test sample 1 again?
> knnTestPredictionProbs[1, "RD"]
       RD
0.2222222
```

For some R modeling functions—including `knn3`—can use `type` argument to specify what format you want predictions in:

```
> knnTestPredictionClass = predict(knnFit, t(hessTest), type="class")
> head(knnTestPredictionClass)
```

```
[1] pCR RD  RD  RD  RD  RD
Levels: pCR RD
```

```
>  ## use table function to generate 2x2 contingency table:
> contingency = table(knnTestPredictionClass, hessTestY)
> contingency
```

```
                      hessTestY
knnTestPredictionClass pCR RD
                   pCR   3  2
                   RD   10 36
```

The 2x2 contingency table is a very useful and commonly presented summary of binary classification results. If one class is regarded as "positive" and one as "negative," the various cells of the 2x2 table can be labeled:

	Actual (-)	Actual (+)
Predicted (-)	True Negatives (TN)	False Negatives (FN)
Predicted (+)	False Positives (FP)	True Positives (TP)

Notice that:

- the diagonal elements of the contingency table correspond to accurate classifications, and that
- every (classifiable) sampling unit will fall into one of the four cells.

Thus we can calculate the fraction of sampling units classified correctly—referred to in ML contexts as the *accuracy* of the model fit—by dividing the sum of the diagonals of the contingency table by the sum of all four entries in the contingency table:

```
> estimatedAccuracy = sum(diag(contingency)) / sum(contingency)
> estimatedAccuracy
```

```
[1] 0.7647059
```

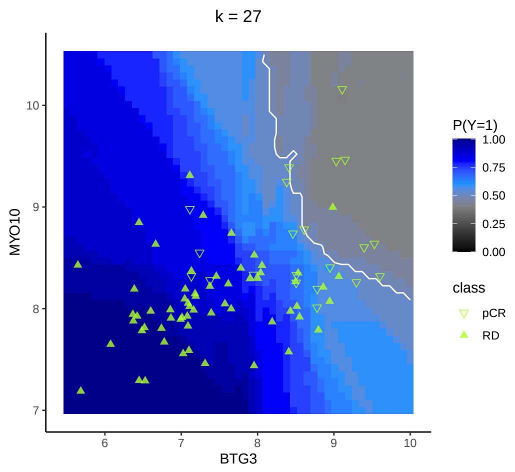
8 Overfitting

Let's use the data for two specific microarray probes, 205548_s_at and 201976_s_at to fit a knn model with $k = 27$:

```
>  ## we'll go ahead and transpose the data.frame to have
>  ## features-in-columns for convenience:
> twoProbeData = t(hessTrain)[ , c("205548_s_at", "201976_s_at")]
>  ## let's use friendlier gene names instead of probe ids here:
> colnames(twoProbeData) =
          probeAnnot[colnames(twoProbeData), "Gene.Symbol"]
> twoProbeFitK27 = knn3(twoProbeData, hessTrainY, k=27)
```

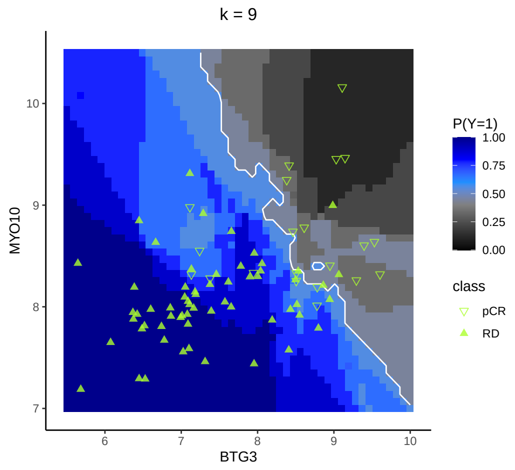
I'm using two probes because I want to be able to make a contour plot of the *decision boundary* of the knn classifier. I'm going to use a function saved in the file `ggfuntile.R` to do this:

```
> source("ggfuntile.R")  ## defines predictionContour function
> predictionContour(twoProbeFitK27, twoProbeData, hessTrainY, "k = 27")
```



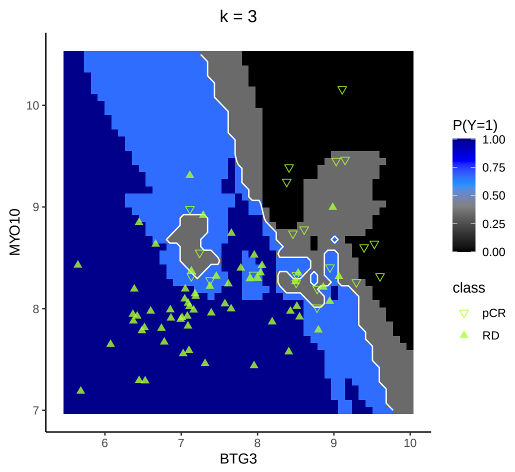
Looking at this decision boundary you might think this classifier looks too conservative about calling pCR: Surely we could push that boundary to the left a bit to catch a few more of those open downward-pointing triangles? Thinking about this a bit more, it seems that perhaps our choice of parameter $k = 27$ is a bit high; after all, 27 is almost a third of all 82 samples in the Hess training set. The appropriate neighborhood for points in the upper right hand corner of the contour plot may be better estimated with a more local knn model defined by, say, $k = 9$:

```
> twoProbeFitK9 = knn3(twoProbeData, hessTrainY, k=9)
> predictionContour(twoProbeFitK9, twoProbeData, hessTrainY, "k = 9")
```



That does look somewhat better! Many more pCR samples correctly called at the cost of only one extra misclassified RD sample. But perhaps we could do better still with an even more local model—let's try $k = 3$:

```
> twoProbeFitK3 = knn3(twoProbeData, hessTrainY, k=3)
> predictionContour(twoProbeFitK3, twoProbeData, hessTrainY, "k = 3")
```



Hmmm...this does appear to catch a few more pCR samples from the training set, but we seem to have generated some swiss cheese-like holes in the RD predicted region, along with a very convoluted bay and peninsula in the center right portion of the main decision boundary. Still, this seems like a very subjective complaint—let's look at some accuracy estimates:

```
>  ## define pair of convenience functions to minimize repeated code:
> contingencize = function(knnFit, data, y) {
      table(predict(knnFit, data, type="class"), y)
  }
> estimateAccuracyFrom2x2 = function(twoByTwo) {
      sum(diag(twoByTwo)) / sum(twoByTwo)
  }
> twoByTwo27 = contingencize(twoProbeFitK27, twoProbeData, hessTrainY)
> estimateAccuracyFrom2x2(twoByTwo27)
```

```
[1] 0.804878
```

```
> twoByTwo9 = contingencize(twoProbeFitK9, twoProbeData, hessTrainY)
> estimateAccuracyFrom2x2(twoByTwo9)
```

```
[1] 0.8780488
```

```
> twoByTwo3 = contingencize(twoProbeFitK3, twoProbeData, hessTrainY)
> estimateAccuracyFrom2x2(twoByTwo3)
```

```
[1] 0.8902439
```

So does this really mean the swiss-cheese decision region is the best...?

Of course not! All of the accuracy estimates we just made suffer from what's called *resubstitution bias* because we tested the model on the same data set that was used to train it. Let's clean that up:

```
>  ## extract test data for our two favorite probes...
> twoProbeTest = t(hessTest)[ , c("205548_s_at", "201976_s_at")]
> colnames(twoProbeTest) =
          probeAnnot[colnames(twoProbeTest), "Gene.Symbol"]
>  ## now let's take another stab at accuracy estimations:
> twoByTwo27 = contingencize(twoProbeFitK27, twoProbeTest, hessTestY)
> estimateAccuracyFrom2x2(twoByTwo27)
```

```
[1] 0.745098
> twoByTwo9 = contingencize(twoProbeFitK9, twoProbeTest, hessTestY)
> estimateAccuracyFrom2x2(twoByTwo9)
[1] 0.8039216
> twoByTwo3 = contingencize(twoProbeFitK3, twoProbeTest, hessTestY)
> estimateAccuracyFrom2x2(twoByTwo3)
[1] 0.7647059
```

While we may be a bit disappointed to see that the best accuracy estimate from the test set is worse than the worst accuracy estimate from resubstitution of the training set, we can find solace in noting that the $k = 3$ model with it's bizarre decision boundary is no longer judged the best.

This is the classic problem of *overfitting*: Models with more freedom to fit very complex patterns in the training data set—such as our very local low-k knn model—have a tendency to find "signals" which are not reproducible in independent data sets, even those of a very similar nature.

Here's an example where you can see the overfitting coming without any computation at all: What do you think the resubstitution accuracy of a 1-nearest neighbor model would be? As a hint, you might think about what the nearest neighbor of training sample i is in the training set...

9 knn Simulation

At this point I will digress away from analysis of the Hess microarray data for a bit to consider simulated data sets. Simulated data can be useful because:

1. we know the true model used to generate the data exactly, and
2. we can systematically vary any parameters that appear in the data generation model so as to study how well our ML algorithms work in a range of situations.

Let's define a function `simulate2group` for simulating

- a simple data set with `n` sampling units (or simulated samples) and `m` features (simulated genes, if you like),
- with the sampling units divided into two groups A and B,
- and `mEffected` $\leq$ `m` of the features being shifted by `effectSize` units on average in group B relative to group A:

```
> simulate2group = function(n = 100,   ## number simulated samples
                            m = 1000,  ## number simulated genes
                            nA = ceiling(0.5*n),  ## first nA samples = group A
                                                  ## last (n-nA) samples = group B
                            mEffected = 10,   ## first mEffected genes will have
                                              ## different expression in group B
                            effectSize = 1 ## how many expression units difference
                                           ## between groups for mEffected genes
                            ) {
      x = matrix(rnorm(n*m), nrow=n, ncol=m)  ## simulate iid expression values
                                              ## (highly unrealistic, but easy)
      y = factor(c(rep("A", nA), rep("B", (n-nA))))
      colnames(x) = paste0("g", 1:m)   ## gene labels like g1, g2, etc.
      rownames(x) = paste0("i", 1:n)   ## sample labels like i1, i2, etc.
      names(y) = rownames(x)   ## assign sample labels as names of grouping vector
      x[y=="B", 1:mEffected] = x[y=="B", 1:mEffected] + effectSize
      return(list(x=data.frame(x), y=y))
  }
```

Because of the second advantage associated with simulated data above—the ability to repeat the analysis while varying simulation parameters—I'm going to package our data generation, model fitting, and model assessment procedure up into a function of those simulation parameters:

```
> simulateAndKnnModel = function(n, m, k, mEffected, effectSize,
                                 rep=1, ...) {
      require(caret)
      trainSet = simulate2group(n = n,
                                m = m,
                                mEffected = mEffected,
                                effectSize = effectSize)
      testSet = simulate2group(n = n,
                               m = m,
                               mEffected = mEffected,
                               effectSize = effectSize)
      knnFit = knn3(trainSet$x, trainSet$y, k)
      resubstitutionPredictions = predict(knnFit, trainSet$x, type="class")
       ## construct contingency table and use to estimate accuracy:
      resub2by2 = table(resubstitutionPredictions, trainSet$y)
      resubAccuracyEst = sum(diag(resub2by2)) / sum(resub2by2)
       ## do same thing for testPredictions:
      testPredictions = predict(knnFit, testSet$x, type="class")
      test2by2 = table(testPredictions, testSet$y)
      testAccuracyEst = sum(diag(test2by2)) / sum(test2by2)
       ## return vector of results along with simulation parameters:
      return(c(m = m,
               k = k,
               rep = rep,  ## rep included to track repetition index
               resubstitution = resubAccuracyEst,
               test = testAccuracyEst))
  }
```

Here's an example using this function to assess the performance of a 5-nearest neighbors model (`k=5`) on a simulated data set of `n=100` sampling units with `m=10` features, of which `mEffected=1` feature has values elevated by `effectSize=2.5` units in group B relative to group A (we'll rely on the `simulate2group` default value of `nA=ceiling(0.5*n)=50` to specify that half of the sampling units are in group A and the other half in group B):

```
> simulateAndKnnModel(n=100, m=10, k=5, mEffected=1, effectSize=2.5)
             m              k            rep resubstitution           test
         10.00           5.00           1.00           0.92           0.87
```

The function reports out some of the simulation parameters along with the estimated accuracy results so that we can keep track of what parameters went into eah simulation when we repeat this procedure many times. We're going to do this by setting up a `data.frame` with one row per simulation and columns specifying the parameters to use:

```
>  ## expand.grid generates data.frame with all combinations of
>  ## supplied arguments
> simulationParameterGrid = expand.grid(
      n = 100,                ## all simulations have n=100
      m = c(2, 5, 10, 25, 50, 100, 250),
      k = c(3, 5, 11, 25),
      rep = 1:10              ## repeat each combination ten times
  )
>  ## we'll say all features are different between group A and B:
> simulationParameterGrid$mEffected = simulationParameterGrid$m
>  ## but with an effect size shrinking with mEffected:
> simulationParameterGrid$effectSize =
          2.5 / sqrt(simulationParameterGrid$mEffected)
> head(simulationParameterGrid)
```

```
    n   m k rep mEffected effectSize
1 100   2 3   1         2  1.7677670
2 100   5 3   1         5  1.1180340
3 100  10 3   1        10  0.7905694
4 100  25 3   1        25  0.5000000
5 100  50 3   1        50  0.3535534
6 100 100 3   1       100  0.2500000
```

```
> nrow(simulationParameterGrid)  ## length(m) * length(k) * 10 repeats
```

```
[1] 280
```

Now that we have our desired simulation parameters nicely organized, we could blast through all of them using a `for`-loop, but one of the advantages of having our simulation and modeling procedure coded up as a function is that it allows us to adopt a slightly more elegant approach using the `apply` function:

```
>  ## simulate and model one data set per row of simulationParameterGrid
>  ## use base-R apply function to do this
> modelingResults = apply(
      X = simulationParameterGrid,
      MARGIN = 1,  ## iterate over rows (first margin) of X argument
      FUN = function(dfrow) {
           ## dfrow has all of the arguments for simulateAndKnnModel,
           ## but they are packed into single vector;
           ## do.call enables function call to unpack a (named) list
           ## into separate (named) arguments:
          do.call(simulateAndKnnModel, args=as.list(dfrow))
      }
  )
> dim(modelingResults)  ## apply here produces matrix with 1 column
```

```
[1]   5 280
```

```
>                       ## per iteration, so let's transpose:
> modelingResults = t(modelingResults)
> head(modelingResults)
```

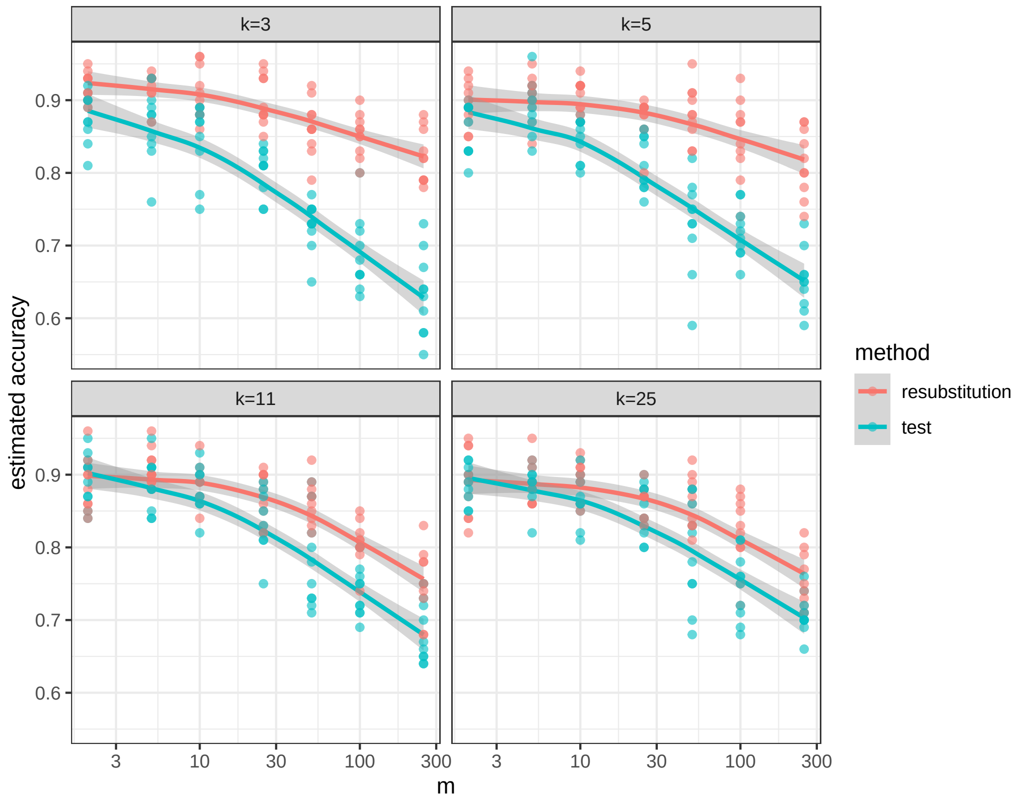
```
       m k rep resubstitution test
[1,]   2 3   1           0.93 0.91
[2,]   5 3   1           0.93 0.88
[3,]  10 3   1           0.89 0.89
[4,]  25 3   1           0.88 0.81
[5,]  50 3   1           0.84 0.72
[6,] 100 3   1           0.85 0.66
```

It's easier to absorb large quantities of quantitative information visually, so let's repackage and plot these results using `tidyr` and `ggplot2`:

```
> ## install.packages("tidyr")  ## uncomment and run if necessary
> library(tidyr)
> ggdata = data.frame(modelingResults) %>%
      pivot_longer(resubstitution:test,
                   names_to = "method",
                   values_to = "estimated accuracy")
> ggdata$k = factor(
      paste0("k=", ggdata$k),
      levels = unique(paste0("k=", ggdata$k))
  )
> head(ggdata)
```

```
# A tibble: 6 x 5
      m k       rep method         `estimated accuracy`
  <dbl> <fct> <dbl> <chr>                         <dbl>
1     2 k=3       1 resubstitution                 0.93
2     2 k=3       1 test                           0.91
3     5 k=3       1 resubstitution                 0.93
4     5 k=3       1 test                           0.88
5    10 k=3       1 resubstitution                 0.89
6    10 k=3       1 test                           0.89
```

```
> gg = ggplot(ggdata, aes(x=m, y=`estimated accuracy`, color=method))
> gg = gg + facet_wrap(~k)
> gg = gg + stat_smooth(method.args=list(degree=1))
> gg = gg + geom_point(alpha=0.6)
> gg = gg + scale_x_log10()
> print(gg)
```



This figure illustrates the degree to which the more flexible (low k) knn models overfit relative to the less flexible (high k) models: the resubstitution accuracy estimate curves lie considerably above the test accuracy estimate curves for `k=3`, with the difference between the two curves shrinking considerably as k rises upwards towards `k=25`.

10 Cross-Validation

When data are scarce, we'd like to be able to both

1. build a classifier using a large fraction—close to 100% if possible—of the available sampling units, while
2. assessing classifier performance without suffering resubstitution bias.

We know how to handle 2: split the data into training and test sets, using only training set to build the classifier and only test set for evaluation of performance. Unfortunately this isn't so great for 1!

One thing we could do with our data split, however, is to swap which set is used to train and which is used to test. This doesn't immediately address point 1 above, but it does at least allow us to use all of our data to test performance.

But we can do better! Why not split our data into three subsets A, B, and C: we can train on (A+B) and test on C, then train on (A+C) and test on B, and finally train on (B+C) and test on A. Now we're making some progress on point 1 above as well as point 2: our training sets are $\frac{2}{3}$ of our full data set and we end up using 100% of the data for testing!

This is the key idea of *cross-validation* (Stone (1974)), which takes it further to allow for 4-fold, 5-fold, 6-fold, ..., n-fold splits of our data set in addition to the 3-fold split just described. The general idea is to fit a model on the data from all but one of the subsets and test on the one held-out subset, repeating this process so that every subset is held-out once.

Performance is generally estimated using this procedure by

- computing accuracy (or whatever other metric one might prefer) separately on each held-out data subset
  - *using the model fit to the the data from all other subsets* so that
  - in no case is a sampling unit i tested using a fit model for which i was part of the training set,
- and then averaging the accuracy estimates from each fold together.

We could code this up from scratch, but it's easier (and less bug-prone) to use the `train` function provided by the `caret` library:

```
> simData = simulate2group(n=100, m=10, mEffected=1, effectSize=2.5)
>  ## recall simData is named list with data.frame simData$x
>  ## and grouping factor simData$y
> cvFolds = 5
> caretOut = train(simData$x, simData$y, method="knn",
                   trControl=trainControl(method="cv", number=cvFolds))
> caretOut
```

```
k-Nearest Neighbors

100 samples
 10 predictor
  2 classes: 'A', 'B'

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 80, 80, 80, 80, 80
Resampling results across tuning parameters:

  k  Accuracy  Kappa
  5  0.84      0.68
  7  0.90      0.80
  9  0.89      0.78

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was k = 7.
```

Notice that all we told `train` about our modeling strategy (ML algorithm) was the string `"knn"`; this works because `train` happens to know about knn. Since we didn't tell it anything about details like what k value to use, `train` went ahead and picked its own values to try, selecting the one that produced the best cross-validation-estimated accuracy value.

While it was convenient to just tell train to use method="knn", we can get a lot more control over exactly what algorithms are trained by putting together a named list of the components train needs in a method argument like so:

```
> knnCaretized = list(
       ## knn3 is in caret library, don't need any others:
      library = NULL,
       ## caret works with both Classification and Regression,
       ## need to tell it we want to do Classification:
      type = "Classification",
       ## tell caret what parameters exist in this model:
      parameters = data.frame(parameter="k", class="integer", label="n_nbrs"),
       ## and also what value(s) of those parameters to try:
      grid = function(x, y, len=NULL, ...) {data.frame(k=9)},
       ## provide caret a function to generate a fit model:
       ## (args x, y, param, and ...; all parameters go in list param):
      fit = function(x, y, param, ...) {knn3(x, y, param$k)},
       ## also provide a function to predict classifications using fit
       ## (important that argument names be modelFit, newdata, and ...):
      predict = function(modelFit, newdata, ...) {
          predict(modelFit, newdata, type="class")
      },
       ## and finally a function to make probabilistic predictions:
      prob = function(modelFit, newdata, ...) {predict(modelFit, newdata)}
  )
> caretOut = train(simData$x, simData$y, method=knnCaretized,
                   trControl=trainControl(method="cv", number=cvFolds))
> caretOut
100 samples
 10 predictor
  2 classes: 'A', 'B'

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 80, 80, 80, 80, 80
Resampling results:

  Accuracy  Kappa
  0.87      0.74

Tuning parameter 'k' was held constant at a value of 9
```

While this is a lot more work on our end, I'm going to use this way of providing method arguments to train from now on, both because for some of the more complicated modeling strategies we'll consider it's the only way to get train to run them and because train tends to run faster when you tell it exactly what to do. While caret offers many useful features, it has a deserved reputation for tying up your computational resources for a while!

Now I'm going to repeat the same many-different-simulations excercise I did above comparing resubstitution and test set accuracy estimates, only replace resubstitution with cross-validation using `train`. First I'll set up a function to facilitate this repetition:

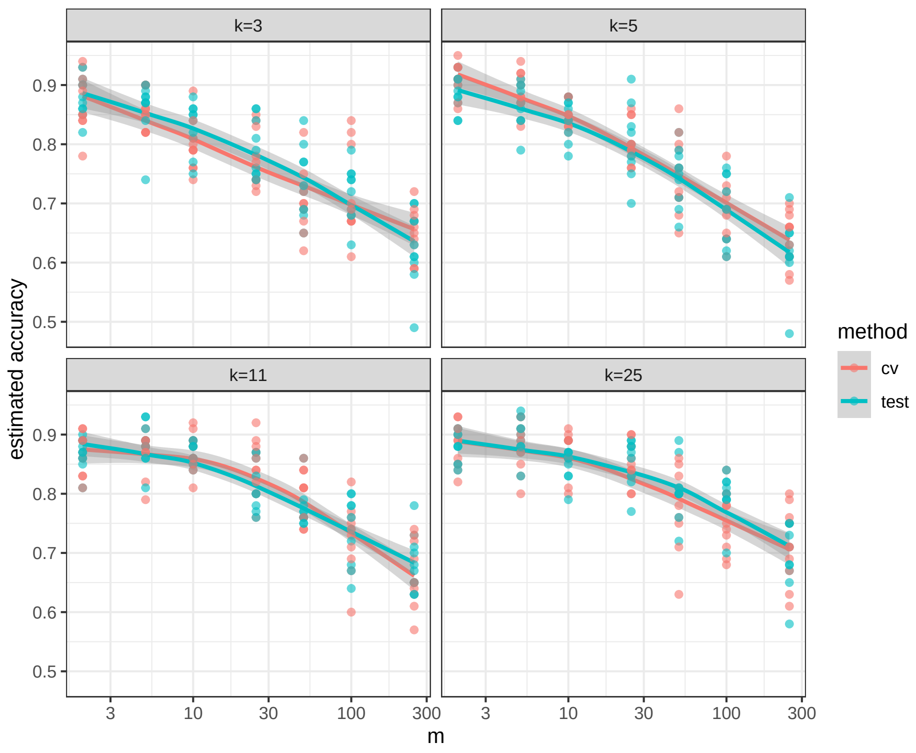
```
> simulateAndCrossValidateKnnModel =
          function(n, m, k, mEffected, effectSize, rep, cvFolds, ...) {
      require(caret)
      trainSet = simulate2group(n=n, m=m,
                                 mEffected=mEffected, effectSize=effectSize)
      testSet = simulate2group(n=n, m=m,
                                mEffected=mEffected, effectSize=effectSize)
      knnCaretized = list(
          library = NULL,
          type = "Classification",
          parameters = data.frame(parameter="k", class="integer", label="n_nbrs"),
          grid = function(x, y, len=NULL, ...) {data.frame(k=k)},
          fit = function(x, y, param, ...) {knn3(x, y, param$k)},
          predict = function(modelFit, newdata, ...) {
              predict(modelFit, newdata, type="class")
          },
          prob = function(modelFit, newdata, ...) {predict(modelFit, newdata)}
      )
      caretOut = train(x = trainSet$x,
                       y = trainSet$y,
                       method = knnCaretized,
                       trControl = trainControl(method="cv", number=cvFolds))
      cvAccuracyEst = caretOut$results$Accuracy
      testPredictions = predict(caretOut$finalModel, testSet$x, type="class")
      test2by2 = table(testPredictions, testSet$y)
      testAccuracyEst = sum(diag(test2by2)) / sum(test2by2)
      return(c(m = m,
               k = k,
               rep = rep,
               cv = cvAccuracyEst,
               test = testAccuracyEst))
  }
```

Now, on to `apply`ing this function to the `simulationParameterGrid` (we'll re-use the same one from before)

```
>  ## add a column to simulationParameterGrid with number of cvFolds:
> simulationParameterGrid$cvFolds = 5
>  ## and now we're ready to go!
> cvModelingResults = t(apply(
      X = simulationParameterGrid,
      MARGIN = 1,
      FUN = function(dfrow) {
          do.call(simulateAndCrossValidateKnnModel, args=as.list(dfrow))
      }
  ))
> head(cvModelingResults)
       m k rep   cv test
[1,]   2 3   1 0.93 0.91
[2,]   5 3   1 0.86 0.74
[3,]  10 3   1 0.89 0.84
[4,]  25 3   1 0.84 0.81
[5,]  50 3   1 0.75 0.65
[6,] 100 3   1 0.67 0.74
```

We can use pretty much the same plotting code from before with only the slightest of modifications:

```
> ggdata = data.frame(cvModelingResults) %>%
      pivot_longer(cv:test,
                   names_to = "method",
                   values_to = "estimated accuracy")
> ggdata$k = factor(
      paste0("k=", ggdata$k),
      levels = unique(paste0("k=", ggdata$k))
  )
> gg = ggplot(ggdata, aes(x=m, y=`estimated accuracy`, color=method))
> gg = gg + facet_wrap(~k)
> gg = gg + stat_smooth(method.args=list(degree=1))
> gg = gg + geom_point(alpha=0.6)
> gg = gg + scale_x_log10()
> print(gg)
```



Cross-validation works! The cross-validated accuracies are pretty much in line with the test accuracies.

There is actually a slight *downward* bias in the accuracy estimates produced by cross-validation resulting from the fact that our training sets using 5-fold cross-validation are only 80% the size of the full data set available for training when use the independent test set. Lest you think that this suggests we should always use the largest possible number of cross-validation (CV) folds—that is, n—you should know that while increasing the number of CV folds decreases the negative bias in accuracy estimation, it also increases the imprecision (variance) in accuracy estimation. As a rule of thumb, you might consider 5- or 10-fold CV as good default `cvFolds` values.

11 Feature Selection

It is often assumed that in any one particular problem the expression patterns of most genes—or, more generally, the values of most features in a high-dimensional data set—are either:

- uninformative or
- redundant with a few maximally useful markers.

Feature selection attempts to identify a restricted set of such useful features for inclusion in a classification model while discarding the rest.

While feature selection may or may not improve the performance of a particular classifier applied to a particular problem, it can

1. reduce computational workload,
2. help to avoid overfitting
    - (though feature selection can itself be susceptible to overfitting!), and
3. facilitate assessment of model using less high-throughput platforms.

A good, if somewhat dated, reference on the use of feature selection methods in bioinformatics is Saeys *et al.* (2007), which breaks down feature selection methods into the following three categories:

Filter Selection done before and independently of classifier construction. Can be univariate or multivariate.

Wrapper Embed classifier construction within feature selection process. Heuristic search methods compare models, favor adding or removing features based on optimization of some specified metric on resulting classifiers.

Embedded Feature selection is inherently built into some classifier construction methods.

For a nice figure providing more detail on the advantages and disadvantages associated with each of these categories, along with some standard examples, you can follow the link to `https://academic.oup.com/view-large/1739292`.

I can't do justice to the wide range of feature selection techniques in the limited time available in this course, so we're going to focus on one particularly simple method: a plain old *t*-test. Here's an R function to select a fixed number of features according to *t*-test *p*-values:

```
> ## install.packages("BiocManager")     ## uncomment and run if necessary
> ## BiocManager::install("genefilter")  ## uncomment and run if necessary
> selectByTTest = function(x, y, m) {
       ## use genefilter package for efficient repeated t-test functions
       ## (here will use genefilter::colttests)
      require(genefilter)
       ## assume x is samples-in-rows, genes-in-columns format!
      p = colttests(x, y)$p.value
       ## sort genes by order of p, return names of first m
      return(colnames(x)[order(p)[1:m]])
  }
```

In order to use this feature selection method as part of a classification "pipeline", we need to connect it ("upstream") to a ("downstream") classification algorithm:

```
>  ## arguments to featSelFit:
>  ## - x is matrix or data.frame of feature values
>  ## - y is factor of classification labels
>  ## - selector is function taking x and y as arguments
>  ##                        and returning selected feature names
>  ## - fitter is a function taking x and y as arguments
>  ##                        and returning fit model object
> featSelFit = function(x, y, selector, fitter) {
      require(caret)
        ## extract features using selector function:
      features = selector(x, y)
       ## retain only selected features in x for fitting knn model:
      x = x[ , features, drop=FALSE]
       ## fit the desired model using the selected feature set:
      fit = fitter(x, y)
       ## package results in list; need to remember features and fit:
      out = list(features=features, fit=fit)
       ## declare this list to be a FeatureSelectedFitModel object
       ## (this will be important for implementing predict method below):
      class(out) = "FeatureSelectedFitModel"
      return(out)
  }
```

Given that we want to make predictions using the `FeatureSelectedFitModel` object output by `featSelFit`, we'll need to implement a `predict` method for this new (S3) class:

```
>  ## arguments for predict method telling R how to make predictions from a
>  ## FeatureSelectedFitModel object:
>  ## - object is a list with class attribute "FeatureSelectedFitModel"
>  ##   (so it should have named elements object$fit, object$features)
>  ## - x is matrix or data.frame of feature values to make predictions for
>  ## - ... any other arguments are passed along to predict.knn3
> predict.FeatureSelectedFitModel = function(object, x, ...) {
       ## first keep only the features in object$features:
      x = x[ , object$features, drop=FALSE]
       ## now predict using object$fit on the selected features:
      return(predict(object$fit, x, ...))
  }
```

OK, now we're ready to try our *t*-test feature selection-then-knn classify modeling strategy out on the Hess data set:

```
>  ## define bindArgs function to allow fixing (i.e. binding)
>  ## the argument for
>  ## - parameter m in selectByTTest and
>  ## - parameter k in knn3
>  ## to extract a function of only the feature data x and
>  ## output class vector y for use in featSelFit:
> bindArgs = function(f, ...) {
      args = list(...)
      return(function(...) {do.call(f, args=c(list(...), args))})
  }
>  ## now let's fit the model:
> fsKnnFit = featSelFit(x = t(hessTrain),
                        y = hessTrainY,
                        selector = bindArgs(selectByTTest, m=25),
                        fitter = bindArgs(knn3, k=9))
>  ## ...and make predictions using with with fit model:
> fsKnnTestPredictionProbs = predict(fsKnnFit, t(hessTest))
> head(fsKnnTestPredictionProbs)
```

```
           pCR        RD
[1,] 0.4444444 0.5555556
[2,] 0.1111111 0.8888889
[3,] 0.1111111 0.8888889
[4,] 0.1111111 0.8888889
[5,] 0.1111111 0.8888889
[6,] 0.0000000 1.0000000
```

```
> fsKnnTestPredictionClass = predict(fsKnnFit, t(hessTest), type="class")
> head(fsKnnTestPredictionClass)
```

```
[1] RD RD RD RD RD RD
Levels: pCR RD
```

```
> table(fsKnnTestPredictionClass, hessTestY)
```

```
                        hessTestY
fsKnnTestPredictionClass pCR RD
                     pCR   4  1
                     RD    9 37
```

```
> fsKnnCaretized = list(
      library = "genefilter",
      type = "Classification",
      parameters = data.frame(
          parameter = c("m", "k"),
          class = c("integer", "integer"),
          label = c("number features", "number neighbors")
      ),
       ## try all combinations of m in {10, 100, 1000, 10000}
       ##                      and k in {5, 9, 19}
      grid = function(x, y, len=NULL, ...) {
          expand.grid(m=c(10, 100, 1000, 10000), k=c(5, 9, 19))
      },
       ## fit should be function of x, y and then any parameters
       ## (here these are m and k) which must be in list named params:
      fit = function(x, y, param, ...) {
          featSelFit(x,
                     y,
                     selector = bindArgs(selectByTTest, m=param$m),
                     fitter = bindArgs(knn3, k=param$k))
      },
       ## caret::train wants predict to make class predictions:
       ## - first argument must be named modelFit
       ## - second should be named newdata
      predict = function(modelFit, newdata, ...) {
          predict(modelFit, newdata, type="class")
      },
       ## let caret::train use plain-old predict
       ## (will really be predict.FeatureSelectedFitModel)
       ## to make probabilistic predictions from fit model object
      prob = predict
  )
```

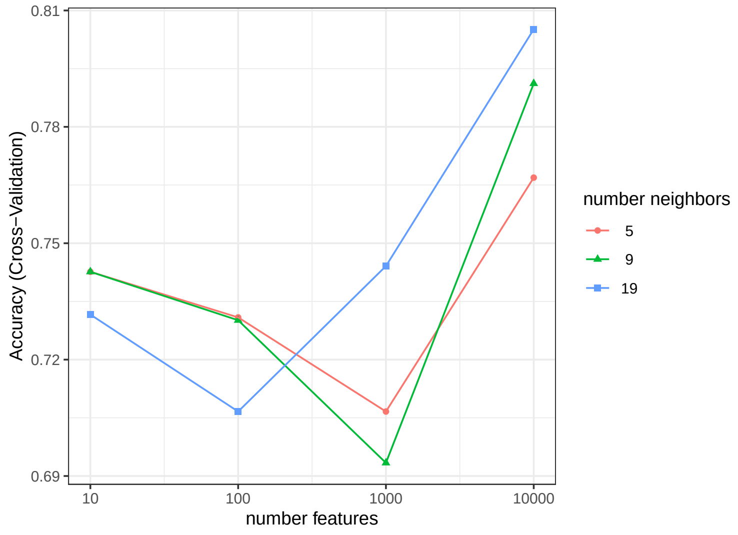
We can now supply `fsKnnCaretized` as the `method` argument for `train`:

```
> caretOut = train(x = t(hessTrain),
                   y = hessTrainY,
                   method = fsKnnCaretized,
                   trControl = trainControl(method="cv", number=5))
> caretOut
```

```
   82 samples
22277 predictors
    2 classes: 'pCR', 'RD'

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 66, 66, 65, 65, 66
Resampling results across tuning parameters:

  m      k   Accuracy   Kappa
     10   5  0.7426471  0.2305198
     10   9  0.7426471  0.2590912
     10  19  0.7316176  0.1406593
    100   5  0.7308824  0.2002049
    100   9  0.7301471  0.2769254
    100  19  0.7066176  0.1751439
   1000   5  0.7066176  0.1521425
   1000   9  0.6933824  0.1243715
   1000  19  0.7441176  0.1958391
  10000   5  0.7669118  0.4250117
  10000   9  0.7911765  0.4360144
  10000  19  0.8051471  0.3393123

Accuracy was used to select the optimal model using the largest value.
The final values used for the model were m = 10000 and k = 19.
```

The objects returned by `caret::train` (of class `train`, naturally) can be `ggplot`ed:

```
> ggplot(caretOut) + scale_x_log10()
```



It is interesting to note that for this data set the size of the selected feature set will vary considerably upon re-running the cross-validation.

12 Feature Extraction

An alternative approach to feature selection to mitigating the problems of overfitting and high computational workload associated with machine learning with high-dimensional data is *feature extraction*.

While

feature selection reduces the size of the feature set presented to a classification or regression algorithm by retaining only a small subset of the feature set,

feature extraction applies a mathematical transformation to the high-dimensional input data to derive a low-dimensional feature set.

For example, if you were trying to classify day vs. night situations with digital image data, you could simply average the intensities of all pixels together to extract a "light level" feature. Note that this single extracted feature still depends on the value of *all* of the input features, so it doesn't reduce the amount of data you need to collect to evaluate the model, but it does massively diminish the complexity of the task confronting whatever downstream classification algorithm you apply!

With gene expression data, the most obvious and widely used method of feature extraction is PCA, so we will use this for our example. Recall that the PC1 scores of a sample are defined as a weighted sum (or linear combination) of feature values with the feature weights learned so as to optimally model feature values based on (feature mean + feature weight * sample score). Higher PCs can then be defined so as to in a similar way so as to successively improve the model.

When building a classification or regression model using PCA for feature extraction, we learn the feature weights for the various principal components (which make up the elements of the "rotation matrix"), as well as the feature mean values, using the training set (only). These weights and means are then fixed parameters of the fit model and should not be updated when presented with test data!

Here is a function for learning the PCs from a training set (provided to the function as a matrix of feature values `x`) which returns a function `extractor` for assessing the sample scores for a test set `newdata`:

```
>  ## arguments to extractPCs
>  ## - x is matrix or data.frame of feature values
>  ## - m is number of principal component features to extract
> extractPCs = function(x, m, ...) {
       ## assume x is samples-in-rows, genes-in-columns format!
       ## training-set-estimated mean expression of each gene:
      mu = colMeans(x)
      pca = prcomp(x, center=TRUE, scale.=FALSE)
       ## extract matrix needed to project new data onto first m extracted PCs:
      projection = pca$rotation[ , 1:m, drop=FALSE]
       ## define extraction function to extract features from new data:
      extractor = function(newdata) {
           ## sweep out training-set-estimated gene means:
          newdata = sweep(newdata, 2, mu, `-`)
          return(newdata %*% projection)
      }
       ## return the function "extractor" which can be applied to newdata;
       ## this function yields coordinates of samples in newdata in PC-space
       ## learned from the training data passed in as x argument.
      return(extractor)
  }
```

We can hook this function for learning the PC features to extract from data up to our knn classification algorithm in a manner similar to what we did for feature selection:

```
>  ## arguments to pcaKnn:
>  ## - x is matrix or data.frame of feature values
>  ## - y is factor of classification labels
>  ## - extractionLearner is function taking x and y as arguments and
>  ##                                 returning extractor function
>  ## - fitter is a function taking x and y as arguments and
>  ##                        returning fit model object
> featExtFit = function(x, y, extractionLearner, fitter) {
       ## use extractionLearner function to learn extractor using data x, y:
      extractor = extractionLearner(x, y)
       ## extract features from x for fitting knn model:
      x = extractor(x)
       ## fit the desired model using the selected feature set:
      fit = fitter(x, y)
       ## package results in list; need to remember extractor and fit:
      out = list(extractor=extractor, fit=fit)
       ## declare this list to be a FeatureExtractedFitModel object:
      class(out) = "FeatureExtractedFitModel"
      return(out)
  }
```

Once again we need to implement a `predict` method for our newly defined `FeatureExtractedFitModel` class:

```
>  ## arguments for predict method telling R how to make predictions from a
>  ## FeatureExtractedFitModel object:
>  ## - object is a list with class attribute "FeatureExtractedFitModel"
>  ##   (so it should have named elements object$fit, object$extractor)
>  ## - x is matrix or data.frame of feature values to make predictions for
>  ## - ... any other arguments are passed along to predict.knn3
> predict.FeatureExtractedFitModel = function(object, x, ...) {
       ## first extract the features using object$extractor:
      x = object$extractor(x)
       ## now predict using object$fit on the extracted features:
      return(predict(object$fit, x, ...))
  }
```

And now we can go ahead and try modeling the Hess data using an ML pipeline with PCA feature extraction feeding into knn classification:

```
> pcaKnnFit = featExtFit(x = t(hessTrain),
                         y = hessTrainY,
                         extractionLearner = bindArgs(extractPCs, m=5),
                         fitter = bindArgs(knn3, k=9))
> pcaKnnTestPredictionClass = predict(pcaKnnFit, t(hessTest), type="class")
> table(pcaKnnTestPredictionClass, hessTestY)
```

```
                         hessTestY
pcaKnnTestPredictionClass pCR RD
                      pCR   5  2
                      RD    8 36
```

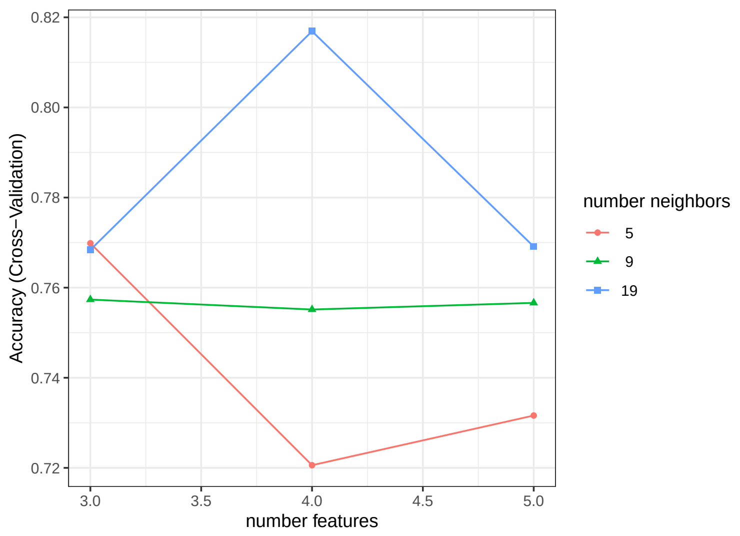
In order to do cross-validation with `caret::train`, we'll need to package everything up in a list with all of the named components `train` will want to see:

```
> pcaKnnCaretized = list(
      library = NULL,
      type = "Classification",
      parameters = data.frame(
          parameter = c("m", "k"),
          class = c("integer", "integer"),
          label = c("number features", "number neighbors")
      ),
       ## try all combinations of m in {3, 4, 5} and k in {5, 9, 19}
      grid = function(x, y, len=NULL, ...) {
          expand.grid(m=3:5, k=c(5, 9, 19))
      },
       ## fit should be function of x, y and named list params:
      fit = function(x, y, param, ...) {
          featExtFit(x,
                     y,
                     extractionLearner = bindArgs(extractPCs, m=param$m),
                     fitter = bindArgs(knn3, k=param$k))
      },
       ## caret::train wants predict to make class predictions:
       ## arguments must be named modelFit and newdata
      predict = function(modelFit, newdata, ...) {
          predict(modelFit, newdata, type="class")
      },
      prob = predict
  )
```

Let's give this `train` and let it do its thing:

```
> caretOut = train(x = t(hessTrain),
                   y = hessTrainY,
                   method = pcaKnnCaretized,
                   trControl = trainControl(method="cv", number=5))
> caretOut
```

```
   82 samples
22277 predictors
    2 classes: 'pCR', 'RD'

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 66, 66, 65, 65, 66
Resampling results across tuning parameters:

  m  k   Accuracy   Kappa
  3   5  0.7698529  0.3399417
  3   9  0.7573529  0.2490326
  3  19  0.7683824  0.2202252
  4   5  0.7205882  0.2310727
  4   9  0.7551471  0.2241718
  4  19  0.8169118  0.3718605
  5   5  0.7316176  0.2533212
  5   9  0.7566176  0.2143860
  5  19  0.7691176  0.2260841

Accuracy was used to select the optimal model using the largest value.
The final values used for the model were m = 4 and k = 19.
> ggplot(caretOut)
```



13 Going deeper

It may have occurred to you that feature selection can be seen as a particularly simple type of feature extraction which “transforms” the input feature matrix by simply projecting it onto a few select dimensions.

Similarly, just as we previously described the transformation of a data set by PCA feature extraction as a type of prediction, we could reverse viewpoints and frame the action of `predict` methods as really just another type of data transformation—albeit with some peculiar restrictions on the transformed output values (e.g., must be probability scores, must be class labels from particular set of possibilities, etc.).

From this point of view, an ML pipeline is an ordered sequence of ML algorithm steps. To train such a pipeline, we go through the sequence:

- training the i^{th} ML algorithm step using the transformed output from step $i-1$ as our input feature matrix for the current step,
- thus learning fit submodel i.
- We then transform the output again using our newly trained fit submodel i and pass it along as input feature matrix to ML algorithm $i+1$.

To make predictions using the fit pipeline model resulting from this training procedure, we iterate through the the ordered sequence of trained submodels, taking the transformed output from step $i-1$ as input feature matrix to be transformed by fit submodel i and then passed along to step $i+1$. The predicted values are then whatever is output from the final step of the fit pipeline.

The field of *deep learning* (Goodfellow *et al.* (2016)) builds such pipelines out of individual steps ("layers") for which the argument feature matrix and output transformed feature matrix are similar enough in nature that the same type of submodel can be linked together repeatedly to generate very long pipelines. Because each individual layer in a deep learning model is itself generally composed of many similar subunits (artificial "neurons"), the structure of a deep learning model is typically referred to as a *network* instead of a pipeline, and we speak of a many-layer network as being *deep* instead of long.

Deep learning is beyond the scope of this course, but if you work on any projects involving machine learning long enough it's bound to come up at some point. Especially in problems with very large numbers of n sampling units, deep learning models can often outperform other methods, though they are prone to overfitting and tend to require a great deal of time and effort to get working correctly.

14 Regression Models

```
> source("maclearn_utils_2020.R")
```

Let's put our study of classification modeling on pause for a moment and briefly consider regression instead. As a reminder, these two terms are generally distinguished in supervised ML contexts by the nature of the output to be predicted:

Classification models predict discrete class labels, while

Regression models predict numeric values.

There are certainly weird edge cases that blur these boundaries, but we won't get into any of those here!

I'm going to jump right into an example using the Hess data set here: Modeling the numeric field `DLDA30.Value` from `hessTrainAnnot` using the gene expression values from `hessTrain`. More specifically, using 10 probe sets selected on the basis of correlation with the desired output to, which will be facilitated by defining:

```
> selectByPearsonCorrelation = function(x, y, m) {
       ## assume x is samples-in-rows, genes-in-columns format!
      r = cor(x, y)[ , 1]
      return(colnames(x)[order(abs(r), decreasing=TRUE)[1:m]])
  }
```

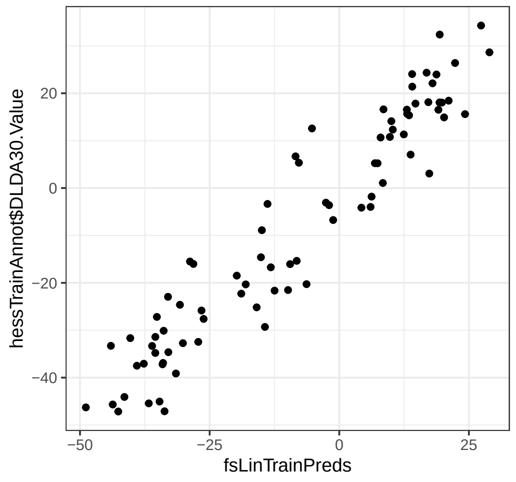
This function is a continuous-`y` analog of the `selectByTTest` function introduced when we initially discussed feature selection.

It will also prove convenient to define an "adapter function" for fitting a linear model in R taking directly as arguments a matrix of gene expression values `x` and a vector of responses `y` instead of the more complex (though also much more flexible!) formula interface of `lm`:

```
> linFitter = function(x, y) {
      lm(y ~ ., data=data.frame(x, check.names=FALSE))
  }
```

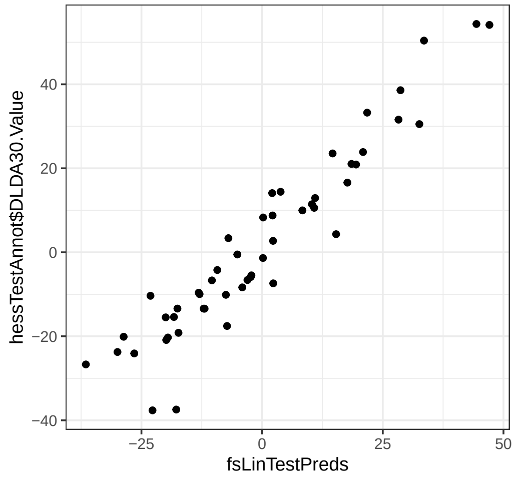
With these preliminaries out of the way, we can now fit our feature-selected linear model:

```
> fsLinFit = featSelFit(
      x = t(hessTrain),
      y = hessTrainAnnot$DLDA30.Value,
      selector = bindArgs(selectByPearsonCorrelation, m=10),
      fitter = linFitter
  )
> fsLinTrainPreds = predict(fsLinFit, t(hessTrain))
>  ## estimate R^2:
> cor(fsLinTrainPreds, hessTrainAnnot$DLDA30.Value)^2
[1] 0.9052456
> qplot(fsLinTrainPreds, hessTrainAnnot$DLDA30.Value)
```



Looks pretty good! Of course, this is comparing predictions to resubsitution-based predictions, so that may or may not be meaningful. Let's try looking at the test set predictions instead:

```
> fsLinTestPreds = predict(fsLinFit, t(hessTest))
> cor(fsLinTestPreds, hessTestAnnot$DLDA30.Value)^2
[1] 0.8999499
> qplot(fsLinTestPreds, hessTestAnnot$DLDA30.Value)
```



Cross-validation can be useful with regression just as it is in classification, and can be performed using `caret::train` in a similar manner:

```
> fsLinModCaretized = list(
      library = NULL,
      type = "Regression",    ## regression, not classification, this time
      parameters = data.frame(parameter="m", class="integer", label="n_features"),
       ## we'll keep using 10 features (probe sets):
      grid = function(x, y, len=NULL, ...) {data.frame(m=10)},
      fit = function(x, y, param, ...) {
          featSelFit(x,
                     y,
                     selector = bindArgs(selectByPearsonCorrelation, m=param$m),
                     fitter = linFitter)
      },
      predict = function(modelFit, newdata, ...) {predict(modelFit, newdata, ...)},
      prob = NULL             ## prob predictions unnecessary for regression
  )
> caretOut = train(x = t(hessTrain),
                   y = hessTrainAnnot$DLDA30.Value,
                   method = fsLinModCaretized,
                   trControl = trainControl(method="cv", number=5))
> caretOut
```

```
   82 samples
22277 predictors

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 66, 66, 65, 66, 65
Resampling results:

  RMSE      Rsquared   MAE
  9.858328  0.8461452  7.961882

Tuning parameter 'm' was held constant at a value of 10
```

Lest you get too excited about these results, I should disclose that `DLDA30.Value` is itself the output from a linear classification algorithm applied by Hess et al.

14.1 Regressing Noise

Having seen what regression results look like from an ML standpoint when everything goes smoothly and there's a nice consistent and easily found signal shared by both training and test data sets, let's consider the opposite extreme of no real signal at all. To that end, we'll define a vector of output `noise` unrelated to any input feature:

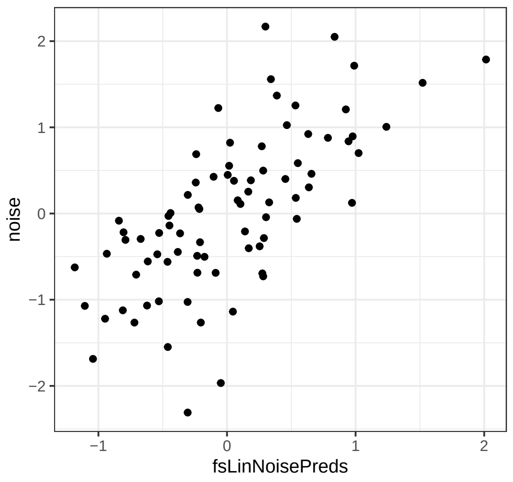
```
> set.seed(123)
> noise = rnorm(ncol(hessTrain))
```

Let's also define one more convenience function for pulling the top few features by Pearson correlation:

```
> topM = function(m) {bindArgs(selectByPearsonCorrelation, m=m)}
```

Onto fitting the noise:

```
> fsNoiseFit = featSelFit(t(hessTrain), noise, topM(10), linFitter)
> fsLinNoisePreds = predict(fsNoiseFit, t(hessTrain))
> cor(fsLinNoisePreds, noise)^2
[1] 0.4748829
> qplot(fsLinNoisePreds, noise)
```



A resubstitution-estimated R^2 value of 47.5%—that's clearly overfit! Cross-validation to the rescue:

```
> fsLinModCaretized = list(
      library = NULL,
      type = "Regression",
      parameters = data.frame(parameter="m", class="integer", label="n_features"),
      grid = function(x, y, len=NULL, ...) {data.frame(m=10)},
      fit = function(x, y, param, ...) {
          featSelFit(x, y, topM(param$m), linFitter)
      },
      predict = function(modelFit, newdata, ...) {predict(modelFit, newdata, ...)},
      prob = NULL
  )
> caretNoise = train(t(hessTrain), noise, fsLinModCaretized,
                     trControl = trainControl(method="cv", number=5))
> caretNoise
   82 samples
22277 predictors

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 66, 65, 65, 66, 66
Resampling results:

  RMSE      Rsquared    MAE
  1.076089  0.03132994  0.8300063

Tuning parameter 'm' was held constant at a value of 10
```

Notice that we've been (correctly) keeping the feature selection step "under cross-validation," meaning that we re-select a (potentially different!) feature set in each fold of cross-validation making sure to exclude the held-out samples from the calculation of feature scores (Pearson correlations here). This is very important—feature selection is a supervised ML step and can be very sensitive to overfitting!

In order to demonstrate this, let's see what happens if we incorrectly apply feature selection prior to cross-validation of only the regression fitting step:

```
>  ## how much overfitting results from feature selection alone?
> topFeatsWholeTrain = data.frame(
       ## re-use features from fsNoiseFit above:
      t(hessTrain)[ , fsNoiseFit$features],
      check.names = FALSE
  )
> noSelectionLinModCaretized = list(
      library = NULL,
      type = "Regression",
       ## define unused dummy parameter to keep caret happy
      parameters = data.frame(parameter="dummy", class="integer", label="dummy"),
      grid = function(x, y, len=NULL, ...) {data.frame(dummy=0)},
      fit = function(x, y, param, ...) {linFitter(x, y)},
      predict = function(modelFit, newdata, ...) {
          predict(modelFit, newdata, ...)
      },
      prob = NULL
  )
> badCaret = train(x = topFeatsWholeTrain,
                   y = noise,
                   method = noSelectionLinModCaretized,
                   trControl = trainControl(method="cv", number=5))
> badCaret
```

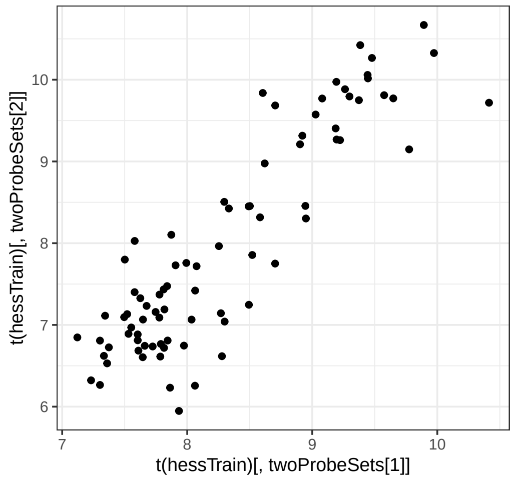
```
82 samples
10 predictors

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 66, 66, 64, 66, 66
Resampling results:

  RMSE      Rsquared   MAE
  0.737936  0.4022355  0.5764094

Tuning parameter 'dummy' was held constant at a value of 0
```

So, to sum up: R^2 of 40.2% for pure, feature-independent noise using "cross-validation" but forgetting to take the overfitting resulting from feature selection into account, compared to an estimate of $R^2 = 3.1\%$ when cross-validation is done correctly! Always, always, *always* keep feature selection under cross-validation.

15 Regularization

Let's go back and look at a couple of the probe sets chosen among the ten features used to predict `DLDA30.Value` above:

```
> twoProbeSets = c("203928_x_at", "203929_s_at")
> twoProbeSets %in% fsLinFit$features
[1] TRUE TRUE
> qplot(t(hessTrain)[ , twoProbeSets[1]], t(hessTrain)[ , twoProbeSets[2]])
```



That's interesting—the measured expression values of these two probe sets are quite similar! There is in fact a good explanation for this:

```
> probeAnnot[twoProbeSets, "Gene.Symbol"]
[1] "MAPT" "MAPT"
```

Microarrays often have multiple probe sets targeting the same gene, and, as a result, in many—though not all!—cases, these probe sets will pick up very similar signals. If one such probe set is correlated with the desired output to be predicted, the other will thus also tend to exhibit such correlation; this is what is happening here.

Given this build up, one might expect that the linear model fit using these features (along with 8 others in this case) would assign similar coefficients to both. Does it?

```
> coef(fsLinFit$fit)[paste0("`", twoProbeSets, "`")]
`203928_x_at` `203929_s_at`
    0.5355169    -2.0742810
```

Not so much—not only are the magnitudes of the coefficients very different, but they even have opposite signs!

Lest you think that perhaps the linear model has found some useful difference between these two probe sets that isn't immediately apparent to us in the Hess data set, consider the following simulated example:

```
> x = matrix(rnorm(20), nrow=10, ncol=2)
> x[ , 2] = x[ , 1] + 0.01 * x[ , 2]       ## force approximate collinearity
> y = -x[ , 1] - x[ , 2] + rnorm(10)       ## true beta_1 = beta_2 = -1
> coef(lm(y ~ ., data=data.frame(x)))
```

```
(Intercept)          X1          X2
 0.20849984  0.01253521 -2.15543699
```

This is in fact a well-known artifact of linear models often referred to as the problem of multicollinearity. One way of dealing with it is to manually remove variables one at a time until you've gotten rid of it, but since this is a course in machine learning we will not do that.

An alternate solution is to modify the algorithm for fitting a linear model by incorporating something called *regularization*.

Unregularized (standard) linear regression uses maximum likelihood to fit the coefficients β_g, where g indexes features g, by ordinary least-squares (OLS) estimator:

$$\hat{\beta}_0, \hat{\boldsymbol{\beta}} = \underset{\beta_0, \boldsymbol{\beta}}{\arg\min} \sum_i \left(y_i - \beta_0 - \boldsymbol{\beta} \cdot \mathbf{x}_i\right)^2 \tag{21}$$

where $\mathbf{x}_i$ is the vector of feature values x_{ig} for sampling unit i, is the vector of coefficients β_g, and $\boldsymbol{\beta} \cdot \mathbf{x}_i = \sum_g \beta_g x_{ig}$ is the "dot product" of the two vectors.

Since $\hat{y}_i = \beta_0 + \boldsymbol{\beta} \cdot \mathbf{x}_i$ is the formula applied by linear regression to predict the value y_i for sampling unit i, Eq (21) says we want to choose the coefficients β_g to minimize the sum of squared *error residuals* $y_i - \hat{y}_i$.

Regularization modifies Eq (21) by adding a penalty term:

$$\hat{\beta}_0, \hat{\boldsymbol{\beta}} = \underset{\beta_0, \boldsymbol{\beta}}{\arg\min} \left\{ \sum_i \left(y_i - \beta_0 - \boldsymbol{\beta} \cdot \mathbf{x}_i\right)^2 + \phi \sum_g |\beta_g|^p \right\} \tag{22}$$

where the exponent $p = 1$ for L1, or "lasso," regression (Tibshirani (1996)), or $p = 2$ for L2, or "ridge," regression (Tikhonov (1943); Hoerl (1962)). This has the effect of biasing the choice of coefficients β_g towards 0 by an amount dependent on the strength of the ϕ of the regulaization applied.

(If you're partial to Bayesian statistics, you might find it interesting to note that L1 regression can be derived from assuming a Laplace-distributed prior for the coefficients β_g, while L2 regression can similarly be derived assuming a more pedestrian Gaussian-distributed prior for the β_g Park & Casella (2008).)

Let's try L2 regularization out using the `glmnet` package (one advantage of which is automated selection of regularization strength parameter):

```
>  ## regularizedGLM, defined in maclearn_utils_2020.R,
>  ## wraps glmnet to facilitate autmated lambda selection
>  ## (lambda controls regularization strength)
>  ## - alpha=0 is L2/ridge regression; alpha=1 is L1/lasso
> l2mod = regularizedGLM(x, y, alpha=0)
> coef(l2mod)
(Intercept)          V1          V2
  0.1942681  -0.9951867  -0.9945844
```

That looks better! What if we try L1 regularization?

```
> l1mod = regularizedGLM(x, y, alpha=1)
> coef(l1mod)
(Intercept)          V1          V2
  0.2068560  -0.2188425  -1.9101953
```

In this case, L1 doesn't look so great, but before you write it off, let me give you a bit of background. L2 regularization is older, much easier (and faster) to fit, and tends to "split the difference" between collinear predictors—as it did here—while L1 regularization is newer, trickier (and slower) to fit, while tending to pick a few variables to assign high magnitude coefficients to while giving all others either exactly 0 or very low magnitudes. That is, L1/lasso regularization is essentially an *embedded feature selection* algorithm!

The multicollinearity problem becomes increasingly severe as the dimensionality of the data set increases until it breaks the classical linear modeling framework entirely when the number of features exceeds the number of sampling units in the training set. Regularization fixes this and allows fitting such "overparametrized" linear models.

16 Logistic Regression

Linear models can be used for classification as well as regression. The most popular linear model for classification goes under the confusing name "logistic regression," despite the fact that it is indeed a classication algorithm.

The idea of logistic regression is to build a linear model to predict the "logit-transformed" probability that a sampling unit should be given a classification label $y = 1$ (as opposed to the other possible label $y = 0$), where the logit function is

$$\text{logit}(p) = \log\left(\frac{p}{1-p}\right) \tag{23}$$

The logit function stretches the range of probabilities (from 0 to 1) out to range all the way from $-\infty$ to $+\infty$: this is good because it turns out to be difficult to fit linear models well in such a way as to constrict the output range to a narrow interval like 0 to 1.

It turns out that if the coefficients β_g are a linear model for logit(p), then the predicted probability of the classification label y taking the value 1 (i.e. whichever class has been declared "positive") for a sampling unit with feature values x_g wrapped up into vector $\mathbf{x}$ is

$$\hat{p} = \text{expit}(\beta_0 + \boldsymbol{\beta} \cdot \mathbf{x}) \tag{24}$$

where

$$\text{expit}(u) = \frac{1}{1 + \exp(-u)} \tag{25}$$

is the *logistic*, or inverse-logit, function. Eq (24) holds because expit is indeed the functional inverse of the logit function: $\text{expit}(\text{logit}(p)) = p$ for all $p \in (0, 1)$.

Logistic regression is a type of *generalized linear model*, or GLM (Nelder & Wedderburn (1972); Agresti (2015)).

```
> logisticFitter = function(x, y) {
      glm(formula = y ~ .,
          data = data.frame(x, y=y, check.names=FALSE),
          family = binomial)  ## family=binomial for logistic regression
  }
```

Logistic regression suffers from the same sort of multicollinearity problems as linear regression and hence requires one (or more) of feature selection, feature extraction, and/or regularization for application in high-dimensional (more features than sampling units) contexts. Here we'll connect our *t*-test feature selector upstream of a `logisticFitter` in a simple ML pipeline:

```
> fsLogisticFit = featSelFit(x = t(hessTrain),
                             y = hessTrainY,
                             selector = bindArgs(selectByTTest, m=10),
                             fitter = logisticFitter)
> fsLogisticTestPreds = predict(
      fsLogisticFit,
       ## predict.glm wants newdata as data.frame, not matrix:
      data.frame(t(hessTest), check.names=FALSE)
  )
>  ## predict.glm returns vector, not matrix!
>  ## - these are logit-transformed probabilities, hence may be <0 or >1
> head(fsLogisticTestPreds)
```

```
     M402       M387       M386       M322       M375       M330
-2.1118273  0.8961195  3.5938794  4.3400570  3.3046953  4.6192480
```

With `glm`, it's easiest to generate class predictions by simply discretizing the logit-transformed prediction vector using `ifelse`:

```
>  ## threshold 0 below b/c logit-transformed p, not p itself!
>  ## - note that logit(0.5) = log(0.5/0.5) = 0
> fsLogisticTestPredClass = factor(ifelse(
      fsLogisticTestPreds < 0, "pCR", "RD"
  ))
> contingency = table(fsLogisticTestPredClass, hessTestY)
> contingency
```

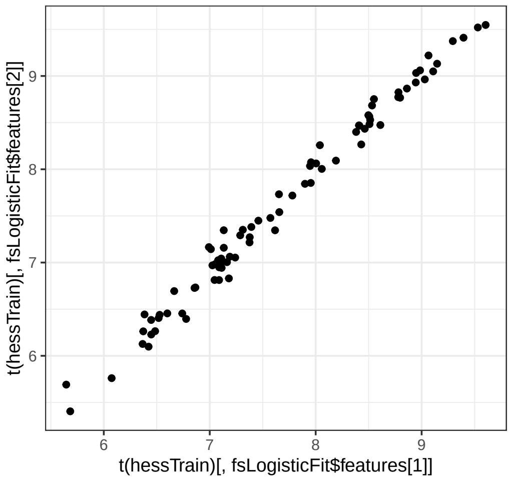
```
                       hessTestY
fsLogisticTestPredClass pCR RD
                    pCR   4  2
                    RD    9 36
```

Let's take a look at the features selected by the pipeline here:

```
> fsLogisticFit$features
 [1] "205548_s_at" "213134_x_at" "201976_s_at" "204825_at"   "219257_s_at"
 [6] "213338_at"   "204750_s_at" "221957_at"   "205339_at"   "214383_x_at"
> probeAnnot[fsLogisticFit$features, "Gene.Symbol"]
 [1] "BTG3"   "BTG3"   "MYO10"  "MELK"   "SPHK1"  "RIS1"   "DSC2"   "PDK3"
 [9] "SIL"    "KLHDC3"
```

Once again we see two probe sets for the same gene (BTG3 in this case) showing up!

```
> qplot(t(hessTrain)[ , fsLogisticFit$features[1]],
        t(hessTrain)[ , fsLogisticFit$features[2]])
```



These two are even more tightly correlated than the two probe sets we ran into in the noise modeling excercise above! Let's check their coefficients in the logistic fit:

```
> coef(fsLogisticFit$fit)[2:3]
`205548_s_at` `213134_x_at`
     1.719878     -2.929572
```

Once again despite the highly similar measured expression values associated with the two probe sets, the coefficients take opposite signs! This logistic model fit will likely be improved by regularization:

```
> fsRegLogisticFit = featSelFit(
      x = t(hessTrain),
      y = hessTrainY,
      selector = bindArgs(selectByTTest, m=10),
      fitter = bindArgs(regularizedGLM, family="binomial", alpha=0)
  )
> coef(fsRegLogisticFit$fit)[2:3]
205548_s_at 213134_x_at
 -0.2878714  -0.2624480
```

So when L2 regularization is used in fitting logistic regression model, the coefficients for the two probe sets for BTG3 are almost identical (and of much more plausible magnitude as well!). Does regularization effect the test set predictions?

```
> fsRegLogisticTestPreds = predict(fsRegLogisticFit, t(hessTest))
> head(fsRegLogisticTestPreds)
      M402       M387       M386       M322       M375       M330 
-0.7605581  0.7770294  1.7509220  2.4714570  1.8245204  2.2623306 
> fsRegLogisticTestPredClass =
          predict(fsRegLogisticFit, t(hessTest), type="class")
> table(fsRegLogisticTestPredClass, hessTestY)
                          hessTestY
fsRegLogisticTestPredClass pCR RD
                         1   5  1
                         2   8 37
```

In this case, regularization produces a model with slightly improved overall accuracy, accurately calling one more pCR and one more RD sample. Beyond this small improvement in estimated model performance, I'd also argue that the regularized model is superior in that the coefficients are more easily interpretable because they do not artificially differentiate between two probe sets for the same underlying gene which show negligible differences in measured expression values.

We could also analyze the performance of either or both of the regularized and unregularized feature-selected logistic classification pipelines in the test set using cross validation just as we did with the knn pipelines, but as it doesn't introduce any new concepts I will in the interests of time instead move on.

17 DLDA and Naive Bayes

"Naive Bayes" describes a family of statistical classification methods sharing the common assumption that the feature values are conditionally independent of each other within each class y (Lewis (1998)):

$$\mathbb{P}(\mathbf{X} = \mathbf{x} \mid Y = y) = \prod_g \mathbb{P}(X_g = x_g \mid Y = y) \tag{26}$$

Eq (26) can be substituted into Bayes' formula to calculate classification probabilities:

$$\mathbb{P}(Y = y \mid \mathbf{X} = \mathbf{x}) = \frac{\pi_y \prod\limits_g \mathbb{P}(X_g = x_g \mid Y = y)}{\sum\limits_{y'} \pi_{y'} \prod\limits_g \mathbb{P}(X_g = x_g \mid Y = y')} \tag{27}$$

where $\pi_y = \mathbb{P}(Y = y)$ is the marginal probability (often called a "prior probability" in this context) of class y given no information about the feature values $\mathbf{x}$.

Diagonal linear discriminant analysis, or DLDA, is a form of naive Bayes classification with the additional assumption that logit($\mathbb{P}(Y = 1 \mid \mathbf{X} = \mathbf{x})$ is linear in $\mathbf{x}$, as will be the case if the conditional probability densities for $\mathbf{X} \mid Y = 0$ and $\mathbf{X} \mid Y = 1$ are both Gaussian with different means but the same (diagonal) covariance (Dudoit *et al.* (2002)). This linearity assumption is shared with logistic regression, though logistic regression generally does *not* make the naive Bayes assumption of Eq (26) and thus usually results in different fit model coefficients.

Before we take a look at DLDA itself, let's simplify our data by first extracting the features "manually" (this is kosher only because we're not going to do based performance estimation here, just examine the resulting model coefficients!):

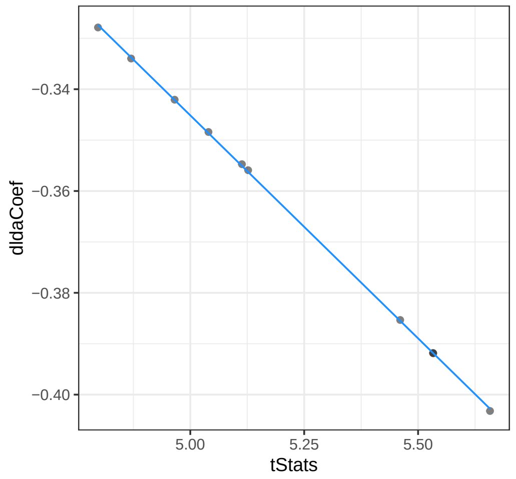
```
>  ## extract feature values for selected features to simplify example
> featData = t(hessTrain)[ , fsLogisticFit$features]
>  ## center features to 0 mean and scale to unit variance:
> featData = scale(featData, center=TRUE, scale=TRUE)
```

While we're at it, let's also take a look at what the t-statistics that led to these features being selected were:

```
> tStats = colttests(featData, hessTrainY)$statistic
```

Now we'll fit a DLDA model using the `HiDimDA` package:

```
> ## install.packages("HiDimDA")  ## uncomment and run if necessary
> library(HiDimDA)
> dldaFit = Dlda(featData, hessTrainY)
> dldaCoef = dldaFit$scaling[ , 1]      ## HiDimDA is weird about coef
> qplot(tStats, dldaCoef, alpha=I(0.5)) +
          stat_smooth(method="lm", se=FALSE, size=0.5, color="dodgerblue")
```

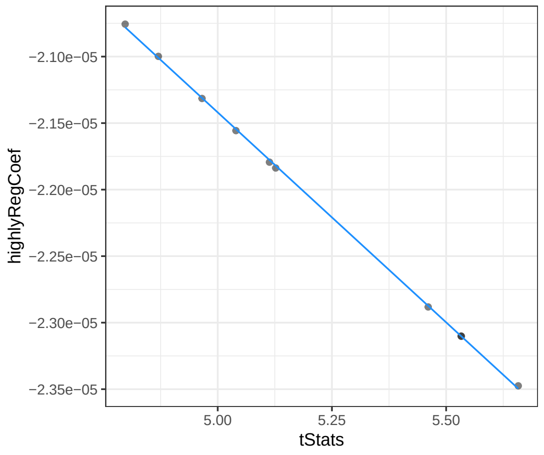


Look at that: the coefficients `dldaCoef` of the fit DLDA model are determined by the t-statistics of the corresponding features! (The functional relationship is actually slightly more complicated than the straight line used for the plot, however.)

This shouldn't be too surprising: naive Bayes assumes (Eq (26)) that the classifier is determined uniquely by the relationship of each feature *individually* with the class labels. This is exactly the sort of bivariate relationship the t-statistic was designed to quantify.

Here's one more comparison that might be a bit more suprising:

```
>  ## force a very high value of the regularization parameter lambda:
> highlyRegLogistic = regularizedGLM(featData, hessTrainY,
                                     family=binomial, lambda=10000)
> highlyRegCoef = coef(highlyRegLogistic)[-1]  ## [-1] to remove intercept
> qplot(tStats, highlyRegCoef, alpha=I(0.5)) +
        stat_smooth(method="lm", se=FALSE, size=0.5, color="dodgerblue")
```



Thus we see that very highly L2-regularized logistic regression produces linear classifiers whose coefficients are a function of individual feature *t*-statistics, just like DLDA.

17.1 Naive Bayes: does it work?

In many cases, yes, naive Bayes (NB) models, including DLDA, work quite well: e.g., the `DLDA30.Value` column is included in `hessTrainAnnot` and `hessTestAnnot` because Hess et al. found that DLDA with 30 features exhibited the best performance under (fancy stratified) cross-validation! More generally, there is a large body of ML literature in which naive Bayes methods have been shown to be surprisingly effective.

I say "surprisingly" because, outside of artificial simulation settings, the underlying conditional independence assumption is basically never true. So why would might it be effective even when false?

1. We may not have enough data to accurately assess true inter-feature covariance—there are order m^2 pairwise relationships between features to estimate, as opposed to only m relationships between feature and modeled outcome—so that attempts to do so just lead to overfitting.
2. While the NB assumption tends to lead to *overconfident* classifiers—probability scores very near 0 or 1 even when wrong—it still often leads to *accurate* classifiers—most calls aren't wrong, even though those that are may be overconfidently wrong.
3. Counterintuitively, you can show mathematically that NB methods will result in very accurate (though overconfident) classifiers assuming that all feature values are in fact *very* strongly correlated with each other within each class (Rish *et al.* (2001))!
   - This may be quite relevant in some gene expression studies!

Motivated by the results of Hess et al., let's try a DLDA model with 30 features out on the test set:

```
> fsDldaFit = featSelFit(t(hessTrain), hessTrainY,
                         bindArgs(selectByTTest, m=30), Dlda)
> dldaTestPredClass = predict(fsDldaFit, t(hessTest))$class
> table(dldaTestPredClass, hessTestY)
```

```
                 hessTestY
dldaTestPredClass pCR RD
                0   9  4
                1   4 34
```

So 43 out of 51 test samples classified correctly. This is slightly better than we did with either logistic regression or knn, but we can't really conclude much from this result since we haven't systematically compared the algorithms using the exact same feature selections or cross-validation folds.

18 Support Vector Machines

```
> set.seed(123)
```

Linear models, including logistic regression and DLDA, are very useful in many contexts but do have some characteristics which can be limiting. *Support vector machines*, or SVMs (Cortes & Vapnik (1995); Hastie *et al.* (2009)), are a type of supervised classifcation algorithm which address two particular limitations:

1. The parameters fit by classical linear classification algorithms are generally sensitive to extremely easy-to-call sampling untis
   - (correctly called sampling units whose feature vectors are very far from the decision boundary)
   - even when a more accurate classifier might result from parameters which move these outlier probabilities "in the wrong direction"
   - but not far enough to change the final classification made.
2. Linearity of response is a very strong and often unrealistic assumption; many real-world response patterns are highly nonlinear.

The term "support vectors" in the name SVMs refers to the feature vectors corresponding to samples which are close to being on the wrong side of the decision boundary; for a nice illustration check out `https://en.wikipedia.org/wiki/Support_vector_machine#/media/File:SVM_margin.png`.

SVM models are fit by positioning the decision boundary so as to keep the support vectors as far on the right sides as possible; the parameters defining the decision boundary thus ultimately depend only on the sampling units corresponding to the support vectors, thus mitigating point 1 above.

Point 2 can also be addressed in the SVM framework using a mathematical technique known as the "kernel trick." The math here is beyond the scope of these notes, but the core idea is that you first apply a nonlinear transformation to the data matrix and then apply SVM in the transformed coordinates, kind of like when we did feature extraction prior to fitting a knn model. The trick is that certain special transformations lead to model fitting problems which may be described in terms of the *un*transformed coordinates in intuitively interesting and useful ways.

(What makes this so tricky is that while it has been proven that there do indeed exist particular transformations that lead to the problems nicely describable as modified versions of the original SVM problem in untransformed coordinates, the actual transformations themselves—which are very complicated—aren't actually needed in doing the computations, just the modified problem description in the original feature space!)

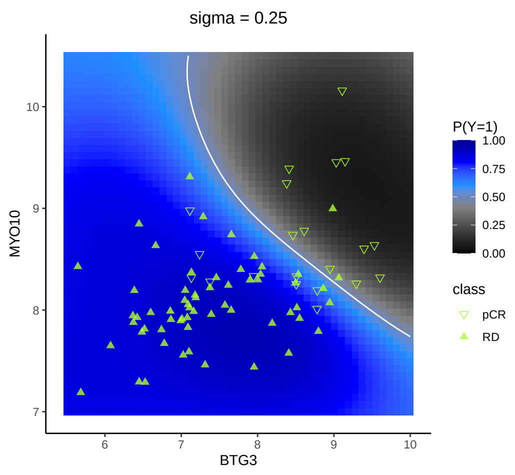
We'll focus on a specific class of transformations: those which replace the standard dot products appearing in the mathematical expressions composing the original SVM problem with so-called "radial basis function" (RBF) kernels.

We'll do this in R using the `kernlab` library; just as we did with `lm` and `glm`, we'll define a convenience function for using the `ksvm` function from this library:

```
> ## install.packages("kernlab")  ## uncomment and run if necessary
>  ## arguments to svmFitter:
>  ## - x is matrix or data.frame of feature values
>  ## - y is factor of classification labels
>  ## - C is cost-of-constraints violation parameter associated with
>  ##   feature vectors getting too close to wrong side of decision
>  ##   boundary
>  ## - sigma is inverse width parameter for radial basis function;
>  ##   higher values of sigma imply more local/less global fits
> svmFitter = function(x, y, C=1, sigma=0.05, ...) {
      require(kernlab)
      ksvm(y ~ .,
           data = data.frame(x, y=y, check.names=FALSE),
           C = C,
           kpar = list(sigma=sigma),
           prob.model = TRUE,
           ...)
   ## prob.model=TRUE argument above supplements traditional SVM algorithm
   ## to allow probabilistic predictions as well as discrete classifications
  }
```

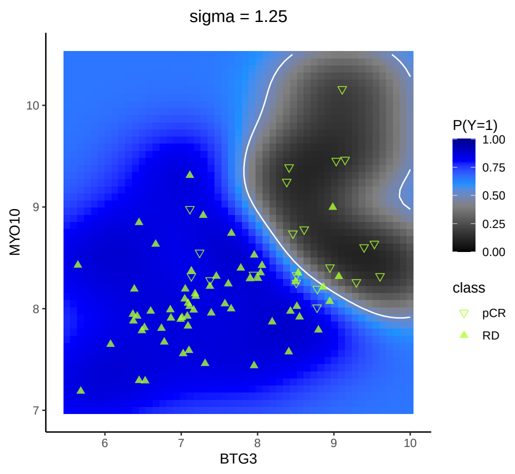
While SVM models are somewhat more complex than the simplicity that is knn, revisiting our two probe set contour plotting strategy using a range of `sigma` parameter values reveals a striking similarity in the types of decision boundaries learned by the methods:

```
> twoProbeSvmFitSig0p25 = svmFitter(twoProbeData, hessTrainY, sigma=0.25)
> predictionContour(twoProbeSvmFitSig0p25,
                    twoProbeData, hessTrainY, "sigma = 0.25")
```



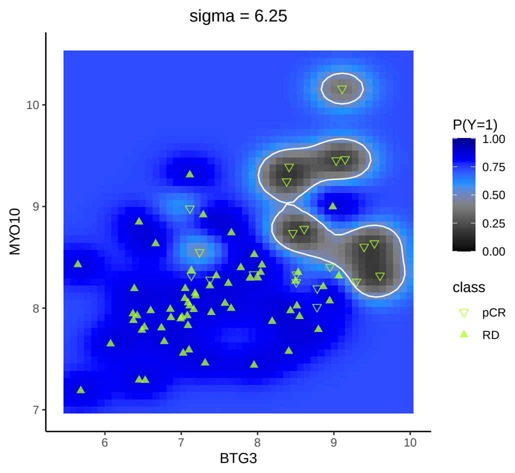
Increasing the `sigma` parameter creates a more local—and in this case, likely more overfit—SVM model (similar to *decreasing* the number k of nearest neighbors in a knn model):

```
> twoProbeSvmFitSig1p25 = svmFitter(twoProbeData, hessTrainY, sigma=1.25)
> predictionContour(twoProbeSvmFitSig1p25,
                    twoProbeData, hessTrainY, "sigma = 1.25")
```



And increasing `sigma` still further...

```
> twoProbeSvmFitSig6p25 = svmFitter(twoProbeData, hessTrainY, sigma=6.25)
> predictionContour(twoProbeSvmFitSig6p25,
                    twoProbeData, hessTrainY, "sigma = 6.25")
```



While SVM models using RBF kernels produce classifiers with somewhat similar properties to knn, you can see that the decision boundaries tend to be smoother. Why might this be?

The knn approach doesn't care whether the $(k+1)^{\text{th}}$ nearest neighbor is just ever so slightly farther away than the k^{th}, or whether the k^{th} nearest neighbor is 10 times farther away than the the $(k-1)^{\text{th}}$, but just gives equal weight to the closest k and zero weight to everything else.

In contrast, the SVM-with-RBF-kernel approach can be seen as making predictions using on a *weighted* sum of the known classifications for nearby training data, with the weightings based on a smooth function of the distance from training feature vector to the feature vector to be classified.

```
> set.seed(123)           ## for replicability
> pcaSvmFit = featExtFit(
      x = t(hessTrain),
      y = hessTrainY,
      extractionLearner = bindArgs(extractPCs, m=5),
      fitter = bindArgs(svmFitter, sigma=0.25)
  )
>  ## b/c SVM is not primarily a probabilistic classifier
>  ## predict method for kvsm objects returns class labels by default
> pcaSvmTestPredictionClass = predict(
      pcaSvmFit,
      data.frame(t(hessTest), check.names=FALSE)
  )
> table(pcaSvmTestPredictionClass, hessTestY)
                         hessTestY
pcaSvmTestPredictionClass pCR RD
                      pCR   2  0
                      RD   11 38
```

We'll assess the performance of this classifier in the training set using `caret::train` as well, but instead of avoiding resubstitution bias using cross-validation we'll try an alternative resampling technique known as *bootstrapping*.

19 Bootstrapping

Machine learning is generally less concerned with questions about whether the internal structure of a model is correct, necessary or interpretable than is classical statistics, but there are still times when we'd like to be able to characterize the uncertainty or repeatability associated with an estimated parameter value.

Put another way: if we had another data set generated in the same way as the one we do have, how similar would the value we estimated for this or that parameter be to what we get using the actually realized training data? Do we expect to get basically the same value or something wildly different?

For linear models, the literature abounds with useful analytical results on confidence intervals, credible intervals, and the like. But for other types of modeling strategies, this is rarely the case!

If gathering data were cheap and easy, we could just go ahead and replicate

- the experiment which generated the data and then
- re-fit the model to the newest round of data

many times to empirically estimate the distribution of fit model parameters.

Bootstrapping is a clever approach to *simulate* such replication using just the one data set we actually have (Tibshirani & Efron (1993)). The bootstrapping process consists of:

1. Generate a case-resampled data set with feature matrix $\underline{\mathbf{X}}^{\text{boot}}$ and outcome vector $\mathbf{y}^{\text{boot}}$ by drawing n random integers $1 \leq r_i \leq n$ *with replacement* and setting

$$x_{ig}^{\text{boot}} = x_{r_i g}$$
$$y_i^{\text{boot}} = y_{r_i}$$

   Note that the r_i will generally not be unique: r_i and r_j may be the same sampling unit even when $i \neq j$, so that the same sampling unit may be included multiple times in the resampled data set!

2. Fit desired model to resampled feature matrix $\underline{\mathbf{X}}^{\text{boot}}$ and outcome vector $\mathbf{y}^{\text{boot}}$ to learn parameters $\hat{\boldsymbol{\theta}}^{\text{boot}}$
   - $\boldsymbol{\theta}$ is just way of writing set of all parameters needed by model pulled together into one big vector, while
   - the "hat" on top of $\hat{\boldsymbol{\theta}}$ indicates that we are talking about an specific data-derived estimate of the parameter values $\boldsymbol{\theta}$, and
   - superscript "boot" on $\hat{\boldsymbol{\theta}}^{\text{boot}}$ just says the parameters were learned from the bootstrap-resampled data as opposed to the original trainind data set.
3. Use fit model with parameters $\hat{\boldsymbol{\theta}}^{\text{boot}}$ to estimate parameter or statistic $\hat{\Omega}^{\text{boot}}$ of interest.
4. Repeat steps 1-3 B times, obtaining values $\hat{\Omega}_b^{\text{boot}}$ for $b \in \{1, \ldots, B\}$ using fit models with parameters $\hat{\boldsymbol{\theta}}_b^{\text{boot}}$.

Note that because bootstrap resampling generates new simulated data sets of the same size n as the original data set but in which some sampling units are repeated, there will necessarily be some sampling units that get left out in any particular resampled data set: on average, a fraction $\frac{1}{e} \approx 0.368$ of all sampling units will be omitted in each bootstrap sample.

19.1 Bootstrapping for Performance Estimation

Bootstrapping can also be used as an alternative to cross-validation for estimation of prediction error Ω.

How should we go about this?

- We might try to estimate distribution of prediction error $\{\hat{\Omega}_b^{\text{full}}\}$
- making predictions with each bootstrap model b with parameters $\hat{\boldsymbol{\theta}}_b^{\text{boot}}$ applied to full (original) training set $\underline{\mathbf{X}}$.

However, since bootstrap training sets were drawn from the same original feature matrix $\underline{\mathbf{X}}$, $\{\hat{\Omega}_b^{\text{full}}\}$ will suffer from resubstitution bias.

Instead we could follow cross-validation methodology:

- use only fit models with parameters $\hat{\boldsymbol{\theta}}_b$ for which
- sampling unit i not used in the b^{th} resampled training set.

Writing R_b to indicate the set of sampling units included in the b^{th} resampled training set:

$$\hat{\Omega}^{\text{loo-boot}} = \frac{1}{n} \sum_i \frac{1}{|\{b \mid i \notin R_b\}|} \sum_{\{b \mid i \notin R_b\}} \hat{\Omega}(\hat{\boldsymbol{\theta}}_b, y_i) \tag{28}$$

(Aside re: set notation: $\{b \mid i \notin R_b\}$ is set of bootstrap iterations b for which sampling unit i does not appear in the set of sampling units R_b, while $|\{b \mid i \notin R_b\}|$ is the number of elements in this set, that is, the number of bootstrap iterations which omitted sampling unit i.)

But while $\{\hat{\Omega}_b^{\text{full}}\}$ are generally overly optimistic, $\hat{\Omega}^{\text{loo-boot}}$ may be too *pessimistic*, since each bootstrap case-resampled training set generally contains only a fraction $1 - \frac{1}{e} \approx 0.632$ of the true training sampling units (albeit with some showing up multiple times!).

Since repeating training sampling units doesn't generally improve models—the repeated units aren't really new data!—we are effectively learning models using only $\approx 63.2\%$ of the available data (albeit randomly upweighting some sampling units relative to others).

Efron & Tibshirani (1997) showed that

$$\hat{\Omega}^{.632} = 0.368\, \hat{\Omega}^{\text{resub}} + 0.632\, \hat{\Omega}^{\text{loo-boot}} \tag{29}$$

strikes a good balance between the optimism of $\hat{\Omega}^{\text{resub}}$ and the pessimism of $\hat{\Omega}^{\text{loo-boot}}$ in some situations.

However, in cases where overfitting is more severe, Efron & Tibshirani (1997) recommend

$$\hat{\Omega}^{.632+} = (1 - \hat{w})\, \hat{\Omega}^{\text{resub}} + \hat{w}\, \hat{\Omega}^{\text{loo-boot}} \tag{30}$$

where $\hat{w} \in [1 - \frac{1}{e}, 1]$ depends on the degree of overfitting.

There is a standard formula for calculating $\hat{w}$ for estimating prediction error using the .632+ bootstrap which you can look up; aside from Efron & Tibshirani (1997), Hastie *et al.* (2009) has a nice treatment.

OK, let's get back to a concrete example: we'll use bootstrapping to assess the performance of a select-10-feature-for-SVM-modeling pipeline using 25 bootstrap resamples (this is a relatively low number for illustration purposes only; most sources suggest ≤ 100 resamples with bootstrapping, but that takes a while!):

```
> fsSvmCaretized = list(
      library = "genefilter",
      type = "Classification",
      parameters = data.frame(
          parameter = c("m", "sigma"),
          class = c("integer", "numeric"),
          label = c("number features", "inverse kernel width")
      ),
      grid = function(x, y, len=NULL, ...) {data.frame(m=10, sigma=0.025)},
      fit = function(x, y, param, ...) {
          featSelFit(x, y,
                     selector = bindArgs(selectByTTest, m=param$m),
                     fitter = bindArgs(svmFitter, sigma=param$sigma))
      },
      predict = function(modelFit, newdata, ...) {predict(modelFit, newdata)},
      prob = NULL
  )
> caretOut = train(x = data.frame(t(hessTrain)),
                   y = hessTrainY,
                   method = fsSvmCaretized,
                    ## here do only 25 bootstrap resamples for speed;
                    ## (usually recommended to do >= 100 in real usage!)
                   trControl = trainControl("boot632", number=25))
> caretOut
```

```
   82 samples
22277 predictors
    2 classes: 'pCR', 'RD'

No pre-processing
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 82, 82, 82, 82, 82, 82, ...
Resampling results:

  Accuracy   Kappa
  0.7758776  0.3301471

Tuning parameter 'm' was held constant at a value of 10
Tuning
 parameter 'sigma' was held constant at a value of 0.025
```

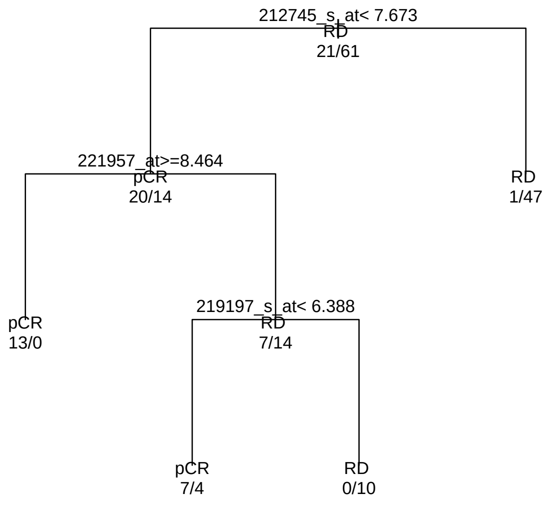
20 Decision Tree Classifiers

Decision trees are probably understood by considering an example. A single decision tree can be constructed in R using the function `rpart` (for "recursive partitioning"). As this function is another formula-interface modeling function, we'll go ahead and define an adapter function giving it into a simpler (though less flexible) x, y argument structure:

```
> library(rpart)
> rpartFitter = function(x, y, control) {
      rpart(y ~ .,
            data = data.frame(x, check.names=FALSE),
            method = "class",   ## use method = "anova" for regression
            control = control)
  }
```

The standard process of fitting a decision trees actually performs a form of embedded feature selection, but the `rpart` function specifically has some unfortunate technical difficulties with very high-dimensional data sets, so we'll connect our by now old-hat t-test feature selector upstream of $rpartFitter$:

```
> fsDecTree = featSelFit(
      x = t(hessTrain),
      y = hessTrainY,
      selector = bindArgs(selectByTTest, m=100),
      fitter = bindArgs(
          rpartFitter,
          control = rpart.control(
              minsplit = 10,    ## don't split if < 10 sampling units in bin
              maxdepth = 3      ## split splits of splits but no more!
          )
      )
  )
> plot(fsDecTree$fit, uniform=TRUE, margin=0.05)
> text(fsDecTree$fit, use.n=TRUE, all=TRUE, cex=0.8)
```



Each node of the tree shown is associated with a subset of the set of all sampling units. The topmost (or root) node contains all samples, which are then split (or partitioned) into those samples for which the expression level for probe set 212745_s_at was measured to be < 7.673, which flow down to the left node, and those samples with higher levels of 212745_s_at expression, which go down the right branch. The fitting algorithm selected the probe set 212745_s_at and the level 7.673 for the top split because this was determined to be the best single split to separate pCR patient samples from RD patient samples.

The "recursive" part of recursive partitioning is then to repeat this splitting process within each of those sample subpopulations, *unless* one of the stopping criteria is met. Stopping criteria are usually based on the size and "impurity" of the sample subpopulation: If the node is associated with too small a sample subpopulation it will not be split, or if the sample subpopulation within the node is sufficiently pure in either one outcome class or the other (either close to all pCR or close to all RD), there is no point in further splitting.

Classification probabilities for any new sample may then be calculated by starting at the root and following the branches of the tree indicated the sample's feature values until a terminal, or leaf, node is reached: the fraction of training set samples in the leaf node with classification RD is then the predicted probability that patient from which the new sample is derived will suffer from residual invasive disease (RD).

```
> fsDecTreeTestPredProbs = predict(fsDecTree, t(hessTest))
> head(fsDecTreeTestPredProbs)
```

```
            pCR        RD
M402 0.63636364 0.3636364
M387 0.02083333 0.9791667
M386 0.02083333 0.9791667
M322 0.02083333 0.9791667
M375 0.02083333 0.9791667
M330 0.02083333 0.9791667
```

```
> fsDecTreeTestPredClass = predict(fsDecTree, t(hessTest), type="class")
> table(fsDecTreeTestPredClass, hessTestY)
```

```
                      hessTestY
fsDecTreeTestPredClass pCR RD
                   pCR   7  5
                   RD    6 33
```

Single decision trees are simple and intuitive but, despite the reasonably good results seen just above, have generally not performed very well in real world classification tasks. The structure of such trees also tends to be very sensitive to small changes in the training data; don't be surprised if you get an entirely different tree if a single sampling unit is added or removed from the training data set!

There is, however, an approach to machine learning based on multiple decision trees which has become very popular in the last few decades...

21 Bagging: Bootstrap Aggregating Models

We could consider using set of B bootstrap case-resample trained models in place of a single model for making predictions.

Repeat for $b \in \{1, \ldots, B\}$:

1. Generate $\underline{\mathbf{X}}_b$ by drawing n random integers $R_b = \{r_{bi}\}$ with replacement and setting $x_{big} = x_{r_{bi}g}$, $y_{bi} = y_{r_{bi}}$.
2. Fit model using $\underline{\mathbf{X}}_b$ and $\mathbf{y}_b$ to obtain fitted parameters $\hat{\boldsymbol{\theta}}_b$.

Bagged predictions for new datum with feature vector $\mathbf{x}$ by simply averaging together the predictions of each bagged submodel b with parameters $\hat{\boldsymbol{\theta}}_b$ for features $\mathbf{x}$.

From Breiman (1996):

> For unstable procedures bagging works well ... The evidence, both experimental and theoretical, is that bagging can push a good but unstable procedure a significant step towards optimality. On the other hand, it can slightly degrade the performance of stable procedures.

In this context, “stability” is of the fit model parameters $\boldsymbol{\theta}$ with respect to the training data $\{\mathbf{x}_i, y_i\}$. Recall that I said in section 20 that decision trees suffered from exactly this sort of instability!

In fact the most well-known application of bagging is indeed the generation of *random forests* of decision trees (Breiman (1999)). A random forest is constructed by repeating, for $b \in \{1, \ldots, B\}$:

1. Generate $\underline{\mathbf{X}}_b$ and $\mathbf{y}_b$ by drawing n random integers $R_b = \{1 \leq r_{bi} \leq n\}$ with replacement and setting $x_{big} = x_{r_{bi}g}$ and $y_{bi} = y_{r_{bi}}$.
2. Randomly select $m' < m$ of the features and fit a decision tree classifier for $\mathbf{y}_b$ using the columns of feature matrix $\underline{\mathbf{X}}_b$ corresponding to those features.
    - m' random features redrawn for each new split.
    - Commonly $m' \approx \sqrt{m}$.

The `randomForest` package in R includes a function of the same name which is quite easy to use (and even handles high-dimensional data sets smoothly, as it already has a non-formula interface built in):

```
> ## install.packages("randomForest")  ## uncomment and run if necessary
> library(randomForest)
> set.seed(321)                        ## replicability
> rf = randomForest(x = data.frame(t(hessTrain), check.names=FALSE),
                    y = hessTrainY,
                    nodesize = 10,     ## randomForest version of minsplit
                    ntree = 100)
>  ## default predict for randomForest is type="class";
>  ## (use type="prob" if you want probabilities in predict call)
> rfPredClass = predict(rf, data.frame(t(hessTest), check.names=FALSE))
> table(rfPredClass, hessTestY)
```

```
           hessTestY
rfPredClass pCR RD
        pCR   9  8
        RD    4 30
```

So...here we found that a single decision tree combined with upstream simple t-test feature selection of 100 probe sets outperformed a random forest of 100 trees. Don't think this is a typical result—random forests have been found to generate very competitive ML classifiers in a wide variety of situations, while single decision trees generally have not. But it does go to show that it can be hard to generalize about ML algorithm performance, especially on relatively small data sets like the Hess example here!

22 Classification Performance Metrics

There are many ways to measure performance for classifiers, of which accuracy is only one. Like accuracy, most are based the discrete classification label calls. For classifiers which output probability scores, this means that some threshold probability ψ (often, but certainly not always, 0.5) must be set.

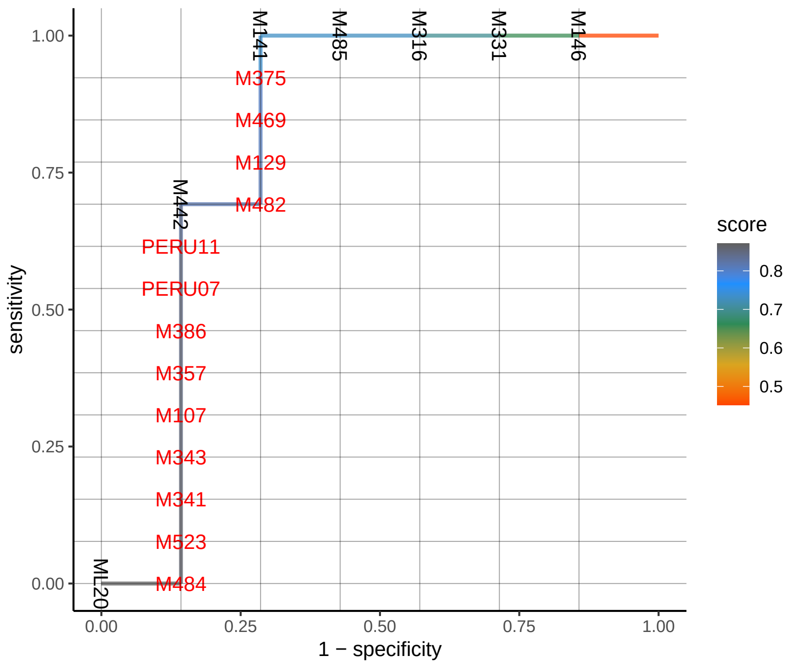
For binary (two-class) classification, when one class can be considered "positive" and the other "negative, the cells of the 2x2 contingency table are often labeled as true positive (TP), true negative (TN), false positive (FP), and false negative (FN), where, e.g., a false positive is sampling unit which the classifier declares positive but for which the true value of the outcome is negative.

We could consider the values of such hard-call metrics over range of threshold values ψ. The so-called receiver operating characteristic (ROC) curve (Fawcett (2006)) does this for sensitivity and specificity:

```
>  ## pick 20 test samples to score with pcaSvmFit classifier:
> set.seed(123)
> xfew = t(hessTest[ , sample(colnames(hessTest), 20)])
> yis1 = hessTestY[rownames(xfew)] == "RD"
> names(yis1) = rownames(xfew)
>  ## do the scoring:
> fewPredProbs = predict(pcaSvmFit, xfew, type="prob")[ , "RD"]
> names(fewPredProbs) = rownames(xfew)
>  ## set up vector all threshold values at which a call would change:
> thresholds = c(none=1, sort(fewPredProbs, decreasing=TRUE), all=0)
>  ## calculate number true positives at each threshold:
> tp = sapply(thresholds, function(thresh) {
      sum(fewPredProbs > thresh & yis1)
  })
>  ## and also number true negatives at each threshold:
> tn = sapply(thresholds, function(thresh) {
      sum(fewPredProbs <= thresh & !yis1)
  })
>  ## scale these by totals to obtain sens, spec at each threshold value:
> sensitivity = tp / sum(yis1)
> specificity = tn / sum(!yis1)
```

Having calculated sensitivity and specificity at every meaningful threshold value, we can now plot the ROC curve using `ggplot` (I should mention that there are several R packages that will do all of this work for you, but I want a very specific format for the plot here):

```
> ggdata = data.frame(
      sample = names(sensitivity),
      actual_class = as.numeric(yis1[names(sensitivity)]),
      score = fewPredProbs[names(sensitivity)],
      sensitivity = sensitivity,
      specificity = specificity
  )
> gg = ggplot(ggdata, aes(x=1-specificity, y=sensitivity))
> gg = gg + geom_line(aes(color=score), size=1, alpha=0.75)
> gg = gg + geom_text(mapping = aes(label=sample),
                      data = ggdata[ggdata$actual_class == 1, ],
                      color = "red")
> gg = gg + geom_text(mapping = aes(label=sample),
                      data = ggdata[ggdata$actual_class == 0, ],
                      angle = -90,
                      color = "black")
> gg = gg + geom_hline(mapping = aes(yintercept=sensitivity),
                       data = ggdata[ggdata$actual_class == 1, ],
                       alpha = 0.35,
                       size = 0.25)
> gg = gg + geom_vline(mapping = aes(xintercept=1-specificity),
                       data = ggdata[ggdata$actual_class == 0, ],
                       alpha = 0.35,
                       size = 0.25)
> gg = gg + scale_color_gradientn(
      colors = c("orangered", "goldenrod", "seagreen", "dodgerblue",
                 rgb(96/255, 96/255, 96/255))
  )
> print(gg + theme_classic())
```



You can see in this plot that there are 13 RD (or "positive") and 7 pCR ("negative") samples in the subampled test data `xfew`: 5 of the 7 negative samples—M146, M141, M485, M331, M316—have lower prediction scores than any of the 13 positive samples. Thus, there are 5 times 13 = 65 light gray vertices below the ROC curve in the 5 columns on the right of the plot.

Adding to this the 9 light gray verices below the ROC curve along the vertical line labeled by sample M442, corresponding to the 9 positive samples with scores higher than that of the negative sample M442, we obtain 74 total ways of pairing one of the positive samples with one of the negative samples for which the positive sample has a higher score than the negative.

This corresponds to a fraction of 74 out of the 91, or 0.8132, vertices in the plot which lie below the curve. This shows that the area under the curve (AUC) for the ROC curve is 0.8132, which must also be the likelihood that if we randomly pick one positive sample and one negative sample from these 20 the positive sample will have a higher score than the negative.

The ROC AUC score is one the most popular metrics for assessing classifier performance. Beyond being threshold-independent—since it aggregates over all possible thresholds by considering the full ROC curve—it has the property that an uninformative classifier will have an AUC of 0.5 even when the two classes are unbalanced (more of one than the other), as they are in the Hess data (almost 3x as many RD as pCR).

This is not the case with accuracy: if you just assign all sampling units the same classification score (ignoring all feature values) and then set the classification threshold so that they are all called the more common class, the accuracy will be the > 0.5 fraction assigned to that class (almost 0.75 in the case of the Hess set!).

We don't usually want to do all of the work we did above to assess the AUC score for a classifier; here are two easier ways to do it:

```
>  ## calculate ROC-AUC using pROC::auc
> ## install.packages("pROC")  ## uncomment and run if necessary
> library(pROC)
> pROC::auc(as.numeric(yis1), fewPredProbs)
Area under the curve: 0.8132
>  ## or can calculate from wilcox.test statistic:
>  ## (this nonparametric test is based on same underlying information):
> wilcoxResults = wilcox.test(fewPredProbs[yis1], fewPredProbs[!yis1])
> wilcoxResults$statistic / (sum(yis1) * sum(!yis1))
        W
0.8131868
```

(In other words, the ROC AUC score is essentially a more interpretable rescaling of the Wilcoxon-Mann-Whitney test (also known as the Mann-Whitney U test) statistic. This makes sense in light of the intepretation of AUC as the chance that a randomly chosen positive case has a higher classification score than does a randomly chosen negative case, since the Wilcoxon-Mann-Whitney test is based on this same idea.)

Of course, we can get better estimate of the AUC using the *whole* test set instead of just `xfew`:

```
> pcaSvmTestPredProbs = predict(pcaSvmFit, t(hessTest), type="prob")
> pROC::auc(as.numeric(hessTestY), pcaSvmTestPredProbs[ , "RD"])
Area under the curve: 0.7409
```

So, a bit worse—but this still shows thus that even though `pcaSvmFit` only managed to correctly call 2 of the 13 test pCR samples as negative:

```
> table(pcaSvmTestPredictionClass, hessTestY)
                         hessTestY
pcaSvmTestPredictionClass pCR RD
                      pCR   2  0
                      RD   11 38
```

the scores of the negative (pCR) samples still tend to be lower than the scores of the positive (RD) samples, even if they are above the default threshold $\psi = 0.5$.

22.1 Wrap-up: Comparing Models by AUC

Let's go back and try a quick head-to-head comparison of five of the different classification models we've covered. First let's make sure we have all of the necessary libraries loaded:

```
> library(caret)         ## knn3
> library(glmnet)
> library(kernlab)       ## ksvm
> library(HiDimDA)       ## Dlda
> library(randomForest)
```

We're going to use one of the apply family of functions—sapply this time—to loop through the five different classification strategies. This function wants to be supplied a list to work with, and if the list has names, sapply will retain those names in the output data structure, so we'll assign those as well:

```
> downstreamFitters = list(
      knn = bindArgs(knn3, k=9),
      l2logistic = bindArgs(regularizedGLM, family=binomial, alpha=0),
      dlda = Dlda,
      svm = svmFitter,
      randomForest = bindArgs(randomForest, ntree=500, nodesize=10)
  )
```

In order to compute ROC AUC scores, we'll need to extract prediction probability scores from each model; because different R packages handle probabilistic prediction differently, it is useful to define a convenience function to handle all of the relevant cases and return the probability scores in a unified format. We'll return these as simple vectors containing the predicted probability of RD:

```
>  ## need single function to make probabilistic predictions
>  ## from all of the classifiers, some require special handling:
> predictProbs = function(modelFit, newdata, ...) {
      testPredProbs = predict(modelFit, newdata, type="prob")
      if (is.list(testPredProbs) && ("Z" %in% names(testPredProbs))) {
           ## HiDimDA::Dlda doesn't directly provide probability scores,
           ## but does provide related scores in list element Z
          testPredProbs = testPredProbs$Z[ , "LD1"]
      }
      if (is.matrix(testPredProbs)) {
          testPredProbs = testPredProbs[ , "RD"]
      }
      return(testPredProbs)
  }
```

Now we're ready to hook up a common upstream feature selection strategy (guess which one we'll use!) to each of the downstreamFitters, fit the resulting pipeline, make predictions on the test set, and calculate the resulting AUC scores:

```
> set.seed(123)
> fsAucs = sapply(downstreamFitters, function(downstreamFitter) {
      fitModel = featSelFit(
          x = data.frame(t(hessTrain), check.names=FALSE),
          y = hessTrainY,
          selector = bindArgs(selectByTTest, m=30),
          fitter = downstreamFitter
      )
      return(as.numeric(pROC::auc(
          as.numeric(hessTestY),
          predictProbs(fitModel, data.frame(t(hessTest), check.names=FALSE))
      )))
  })
> fsAucs
```

```
        knn   l2logistic         dlda          svm randomForest 
  0.9200405    0.8502024    0.8663968    0.7510121    0.8360324 
```

Interesting to note that the simplest strategy, knn, ends up winning according to this comparison! Lots of caveats here: the results might look very different with different methods of feature selection or extraction, different numbers of features retained, different settings of the various modeling parameters (number of nearest neighbors, SVM cost or sigma parameters, number of trees in random forests, etc.), so I wouldn't advise reading too much into this beyond this: sometimes simplicity works.

23 Bibliography

Agresti, Alan. 2015. *Foundations of Linear and Generalized Linear Models.* John Wiley & Sons.

Breiman, Leo. 1996. Bagging Predictors. *Machine Learning*, **24**(2), 123–140.

Breiman, Leo. 1999. Random Forests. *UC Berkeley TR567.*

Cortes, Corinna, & Vapnik, Vladimir. 1995. Support-Vector Networks. *Machine Learning*, **20**(3), 273–297.

Cover, Thomas. 1968. Estimation by the nearest neighbor rule. *IEEE Transactions on Information Theory*, **14**(1), 50–55.

Dudoit, Sandrine, Fridlyand, Jane, & Speed, Terence P. 2002. Comparison of discrimination methods for the classification of tumors using gene expression data. *Journal of the American Statistical Association*, **97**(457), 77–87.

Efron, Bradley, & Tibshirani, Robert. 1997. Improvements on cross-validation: the 632+ bootstrap method. *Journal of the American Statistical Association*, **92**(438), 548–560.

Fawcett, Tom. 2006. An introduction to ROC analysis. *Pattern Recognition Letters*, **27**(8), 861–874.

Ghahramani, Zoubin. 2004. Unsupervised Learning. *Pages 72–112 of: Advanced Lectures on Machine Learning.* Springer.

Goodfellow, Ian, Bengio, Yoshua, & Courville, Aaron. 2016. *Deep Learning.* MIT press.

Hastie, Trevor, Tibshirani, Robert, & Friedman, Jerome. 2009. *The Elements of Statistical Learning.* Springer.

Hess, Kenneth R, Anderson, Keith, Symmans, W Fraser, Valero, Vicente, Ibrahim, Nuhad, Mejia, Jaime A, Booser, Daniel, Theriault, Richard L, Buzdar, Aman U, Dempsey, Peter J, *et al.* . 2006. Pharmacogenomic predictor of sensitivity to preoperative chemotherapy with paclitaxel and fluorouracil, doxorubicin, and cyclophosphamide in breast cancer. *Journal of Clinical Oncology*, **24**(26), 4236–4244.

Hoerl, Arthur E. 1962. Application of ridge analysis to regression problems. *Chemical Engineering Progress*, **58**(3), 54–59.

Izenman, Alan Julian. 2008. *Modern Multivariate Statistical Techniques.* Springer.

Lewis, David D. 1998. Naive (Bayes) at Forty: The Independence Assumption in Information Retrieval. *Pages 4–15 of: European Conference on Machine Learning.* Springer.

MacQueen, James. 1967. Some methods for classification and analysis of multivariate observations. *Pages 281–297 of: Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability*, vol. 1. University of California Press.

Mary-Huard, Tristan, Picard, Franck, & Robin, Stéphane. 2006. Introduction to statistical methods for microarray data analysis. *Mathematical and Computational Methods in Biology. Paris: Hermann.*

Nelder, John Ashworth, & Wedderburn, Robert WM. 1972. Generalized Linear Models. *Journal of the Royal Statistical Society: Series A (General)*, **135**(3), 370–384.

Neves, Alexandre, & Eisenman, Robert N. 2019. Distinct gene-selective roles for a network of core promoter factors in Drosophila neural stem cell identity. *Biology Open*, **8**(4), bio042168.

Park, T., & Casella, G. 2008. The Bayesian lasso. *Journal of the American Statistical Association*, **103**(482), 681–686.

Rish, Irina, Hellerstein, Joseph, & Thathachar, Jayram. 2001. An analysis of data characteristics that affect naive Bayes performance. *IBM TJ Watson Research Center*, **30**.

Roweis, Sam, & Ghahramani, Zoubin. 1999. A unifying review of linear Gaussian models. *Neural Computation*, **11**(2), 305–345.

Saeys, Yvan, Inza, Iñaki, & Larrañaga, Pedro. 2007. A review of feature selection techniques in bioinformatics. *Bioinformatics*, **23**(19), 2507–2517.

Stone, Mervyn. 1974. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society: Series B (Methodological)*, **36**(2), 111–133.

Tibshirani, Robert J. 1996. Regression shrinkage and selection via the lasso. *Journal of the Royal Statistical Society. Series B (Methodological)*, 267–288.

Tibshirani, Robert J, & Efron, Bradley. 1993. *An Introduction to the Bootstrap*. Chapman and Hall New York.

Tikhonov, Andrey Nikolayevich. 1943. On the stability of inverse problems. *Pages 195–198 of: Dokl. Akad. Nauk SSSR*, vol. 39.