Data Sets and Miscellaneous Functions in the `caret` Package

Max Kuhn
max.kuhn@pfizer.com

July 27, 2010

1 Introduction

The `caret` package (short for **c**lassification **a**nd **re**gression **t**raining) contains functions to streamline the model training process for complex regression and classification problems. The package utilizes a number of R packages but tries not to load them all at package start-up[1]. The package "suggests" field includes: `ada`, `affy`, `caTools`, `class`, `e1071,`, `earth (≥ 2.2-3)`, `elasticnet`, `ellipse`, `fastICA`, `foba`, `foreach`, `gam`, `GAMens (≥ 1.1.1),`, `gbm`, `glmnet`, `gpls`, `grid`, `hda`, `HDclassif`, `ipred`, `kernlab`, `klaR`, `lars`, `MASS`, `mboost`, `mda`, `mgcv`, `mlbench`, `neuralnet`, `nnet`, `nodeHarvest`, `pamr,`, `partDSA`, `party`, `penalized`, `pls`, `proxy`, `quantregForest`, `randomForest`, `rda`, `relaxo`, `rocc`, `rpart`, `rrcov`, `RWeka (≥ 0.4-1)`, `sda`, `SDDA`, `sparseLDA (≥ 0.1-1),`, `spls`, `stepPlr`, `superpc`, `vbmp`. `caret` loads packages as needed and assumes that they are installed. Install `caret` using

```
install.packages("caret", dependencies = c("Depends", "Suggests"))
```

to ensure that all the needed packages are installed.

2 Example Data

There are a few data sets included in `caret`. The first three are computational chemistry problems where the object is to relate the molecular structure of compounds (via molecular descriptors) to some property of interest (Blake (2000), Clark and Pickett (2000). The fourth data set listed is a typical spectroscopy data set from chemometrics. The last data set is a pattern recognition problem with a large number of classes, some of which do not have many observations.

[1] By adding formal package dependencies, the package startup time can be greatly decreased

2.1 Blood–Brain Barrier Data

Mente and Lombardo (2005) developed models to predict the log of the ratio of the concentration of a compound in the brain and the concentration in blood. For each compound, they computed three sets of molecular descriptors: MOE 2D, rule–of–five and Charge Polar Surface Area (CPSA). In all, 134 descriptors were calculated. Included in this package are 208 non–proprietary literature compounds. The vector `logBBB` contains the log concentration ratio and the data fame `bbbDescr` contains the descriptor values.

2.2 COX-2 Activity Data

From Sutherland, O'Brien, and Weaver (2003): "A set of 467 cyclooxygenase-2 (COX-2) inhibitors has been assembled from the published work of a single research group, with in vitro activities against human recombinant enzyme expressed as IC50 values ranging from 1 nM to >100 uM (53 compounds have indeterminate IC50 values)." A set of 255 descriptors (MOE2D and QikProp) were generated. To classify the data, we used a cutoff of $2^{2.5}$ to determine activity.

Using `data(cox2)` exposes three R objects: `cox2Descr` is a data frame with the descriptor data, `cox2IC50` is a numeric vector of IC50 assay values and `cox2Class` is a factor vector with the activity results.

2.3 Multidrug Resistance Reversal (MDRR) Agent Data

Svetnik et al (2005) describe these data: "Bakken and Jurs studied a set of compounds originally discussed by Klopman et al., who were interested in multidrug resistance reversal (MDRR) agents. The original response variable is a ratio measuring the ability of a compound to reverse a leukemia cell's resistance to adriamycin. However, the problem was treated as a classification problem, and compounds with the ratio >4.2 were considered active, and those with the ratio <= 2.0 were considered inactive. Compounds with the ratio between these two cutoffs were called moderate and removed from the data for two class classification, leaving a set of 528 compounds (298 actives and 230 inactives). (Various other arrangements of these data were examined by Bakken and Jurs, but we will focus on this particular one.) We did not have access to the original descriptors, but we generated a set of 342 descriptors of three different types that should be similar to the original descriptors, using the DRAGON software." The data and R code are in the Supplemental Data file for the article.

Using `data(mdrr)` exposes two R objects: `mdrrDescr` is a data frame with the descriptor data and `mdrrClass` is a factor vector with the activity results.

2.4 Tecator NIR Data

These data can be found in the datasets section of StatLib. The data consist of 100 near infrared absorbance spectra used to predict the moisture, fat and protein values of chopped meat.

From StatLib:

"These data are recorded on a Tecator Infratec Food and Feed Analyzer working in the wavelength range 850 – 1050 nm by the Near Infrared Transmission (NIT) principle. Each sample contains finely chopped pure meat with different moisture, fat and protein contents.

If results from these data are used in a publication we want you to mention the instrument and company name (Tecator) in the publication. In addition, please send a preprint of your article to: Karin Thente, Tecator AB, Box 70, S-263 21 Hoganas, Sweden."

One reference for these data is Borggaard and Thodberg (1992).

Using `data(tecator)` loads a 215 × 100 matrix of absorbance spectra and a 215 × 3 matrix of outcomes.

2.5 Fatty Acid Composition Data

Brodnjak-Voncina et al. (2005) describe a set of data where seven fatty acid compositions were used to classify commercial oils as either pumpkin (labeled `A`), sunflower (`B`), peanut (`C`), olive (`D`), soybean (`E`), rapeseed (`F`) and corn (`G`). There were 96 data points contained in their Table 1 with known results. The breakdown of the classes is given in below:

```
> data(oil)
> dim(fattyAcids)

[1] 96  7

> table(oilType)

oilType
 A  B  C  D  E  F  G
37 26  3  7 11 10  2
```

As a note, the paper states on page 32 that there are 37 unknown samples while the table on pages 33 and 34 shows that there are 34 unknowns.

3 Data Pre–Processing Functions

caret includes several functions to pre–process the predictor data. Assumes that all of the data are numeric (i.e. factors have been converted to dummy variables via model.matrix or other means).

3.1 Zero– and Near Zero–Variance Predictors

In some situations, the data generating mechanism can create predictors that only have a single unique value (i.e. a "zero–variance predictor"). For many models (excluding tree-based models), this may cause the model to crash or the fit to be unstable.

Similarly, predictors might have only a handful of unique values that occur with very low frequencies. For example, in the drug resistance data, the nR11 descriptor (number of 11-membered rings) data have a few unique numeric values that are highly unbalanced:

```
> data.frame(table(mdrrDescr$nR11))
```

```
  Var1 Freq
1    0  501
2    1    4
3    2   23
```

The concern here that these predictors may become zero–variance predictors when the data are split into cross–validation/bootstrap sub–samples or that a few samples may have an undue influence on the model. These "near–zero–variance" predictors may need to be identified and eliminated prior to modeling.

To identify these types of predictors, the following two metrics can be calculated:

1. the frequency of the most prevalent value over the second most frequent value (called the "frequency ratio"), which would be near one for well–behaved predictors and very large for highly–unbalanced data
2. the "percent of unique values" is the number of unique values divided by the total number of samples (times 100) that approaches zero as the granularity of the data increases

If the frequency ratio is less than a pre–specified threshold *and* the unique value percentage is less than a threshold, we might consider a predictor to be near zero–variance.

We would not want to falsely identify data that have low granularity but are evenly distributed, such as data from a discrete uniform distribution. Using both criteria should not falsely detect such predictors.

Looking at the MDRR data, the `nearZeroVar` function can be used to identify near zero–variance variables (the `saveMetrics` argument can be used to show the details and usually defaults to `FALSE`):

```
> nzv <- nearZeroVar(mdrrDescr, saveMetrics= TRUE)
> nzv[nzv$nzv,][1:10,]
```

```
       freqRatio percentUnique zeroVar  nzv
nTB     23.00000     0.3787879   FALSE TRUE
nBR    131.00000     0.3787879   FALSE TRUE
nI     527.00000     0.3787879   FALSE TRUE
nR03   527.00000     0.3787879   FALSE TRUE
nR08   527.00000     0.3787879   FALSE TRUE
nR11    21.78261     0.5681818   FALSE TRUE
nR12    57.66667     0.3787879   FALSE TRUE
D.Dr03 527.00000     0.3787879   FALSE TRUE
D.Dr07 123.50000     5.8712121   FALSE TRUE
D.Dr08 527.00000     0.3787879   FALSE TRUE
```

By default, `nearZeroVar(data)` will return the positions of the variables that are flagged to be problematic.

3.2 Identifying Correlated Predictors

While there are some models that thrive on correlated predictors (such as `pls`), other models may benefit from reducing the level of correlation between the predictors.

Given a correlation matrix, the `findCorrelation` function uses the following algorithm to flag predictors for removal:

repeat
  Find the pair of predictors with the largest absolute correlation;
  For both predictors, compute the average correlation between each predictor and all of the other variables;
  Flag the variable with the largest mean absolute correlation for removal;
  Remove this column from the correlation matrix;
until *no correlations are above a threshold* ;

For the previous MDRR data, there are 65 descriptors that are almost perfectly correlated ($|\text{correlation}| > 0.999$), such as the total information index of atomic composition (`IAC`) and the total information content index (neighborhood symmetry of 0-order) (`TIC0`) (correlation = 1). The code chunk below shows the effect of removing descriptors with absolute correlations above 0.75.

```
> descrCor <- cor(filteredDescr)
> summary(descrCor[upper.tri(descrCor)])
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-0.99610 -0.05373  0.25010  0.26080  0.65530  1.00000
```

```
> highlyCorDescr <- findCorrelation(descrCor, cutoff = .75)
> filteredDescr <- filteredDescr[,-highlyCorDescr]
> descrCor2 <- cor(filteredDescr)
> summary(descrCor2[upper.tri(descrCor2)])
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
-0.70730 -0.05378  0.04418  0.06692  0.18860  0.74460
```

3.3 Linear Dependencies

The function `findLinearCombos` uses the QR decomposition of a matrix to enumerate sets of linear combinations (if they exist). For example, consider the following matrix that is could have been produced by a less–than–full–rank parameterizations of a two–way experimental layout:

```
> ltfrDesign <- matrix(0, nrow=6, ncol=6)
> ltfrDesign[,1] <- c(1, 1, 1, 1, 1, 1)
> ltfrDesign[,2] <- c(1, 1, 1, 0, 0, 0)
> ltfrDesign[,3] <- c(0, 0, 0, 1, 1, 1)
> ltfrDesign[,4] <- c(1, 0, 0, 1, 0, 0)
> ltfrDesign[,5] <- c(0, 1, 0, 0, 1, 0)
> ltfrDesign[,6] <- c(0, 0, 1, 0, 0, 1)
> ltfrDesign
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    1    0    1    0    0
[2,]    1    1    0    0    1    0
[3,]    1    1    0    0    0    1
[4,]    1    0    1    1    0    0
[5,]    1    0    1    0    1    0
[6,]    1    0    1    0    0    1
```

Note that columns two and three add up to the first column. Similarly, columns four, five and six add up the first column. `findLinearCombos` will return a list that enumerates these dependencies. For each linear combination, it will incrementally remove columns from the matrix and test to see if the dependencies have been resolved. `findLinearCombos` will also return a vector of column positions can be removed to eliminate the linear dependencies:

```
> comboInfo <- findLinearCombos(ltfrDesign)
> comboInfo

$linearCombos
$linearCombos[[1]]
[1] 3 1 2

$linearCombos[[2]]
[1] 6 1 4 5


$remove
[1] 3 6

> ltfrDesign[, -comboInfo$remove]

     [,1] [,2] [,3] [,4]
[1,]    1    1    1    0
[2,]    1    1    0    1
[3,]    1    1    0    0
[4,]    1    0    1    0
[5,]    1    0    0    1
[6,]    1    0    0    0

> findLinearCombos(ltfrDesign[, -comboInfo$remove])

$linearCombos
list()

$remove
NULL
```

These types of dependencies can arise when large numbers of binary chemical fingerprints are used to describe the structure of a molecule.

3.4 Centering and Scaling

The `preProcess` class can be used for many operations on predictors, including centering and scaling[2]. The functiontexttpreProcess estimates the required parameters for each operation and `predict.preProcess` is used to apply them to specific data sets.

[2]Two functions, `processData` and `applyProcessing`, are being replaced by `preProcess`. They are still in the package, but will not be in upcoming versions

In the example below, the half of the MDRR data are used to estimate the location and scale of the predictors. The function `preProcess` doesn't actually pre–process the data. `predict.preProcess` is used to pre–process this and other data sets.

```
> set.seed(96)
> inTrain <- sample(seq(along = mdrrClass), length(mdrrClass)/2)
> training <- filteredDescr[inTrain,]
> test <- filteredDescr[-inTrain,]
> trainMDRR <- mdrrClass[inTrain]
> testMDRR <- mdrrClass[-inTrain]
> preProcValues <- preProcess(training, method = c("center", "scale"))
> trainDescr <- predict(preProcValues, training)
> testDescr <- predict(preProcValues, test)
```

3.5 Transforming Predictors

In some cases, there is a need to use principal component analysis (PCA) to transform the data to a smaller sub–space where the new variable are uncorrelated with one another. The `preProcess` class can apply this transformation by including `"pca"` in the `method` argument. Doing this will also force scaling of the predictors. Note that when PCA is requested, `predict.preProcess` changes the column names to `PC1`, `PC2` and so on.

Similarly, independent component analysis (ICA) can also be used to find new variables that are linear combinations of the original set such that the components are independent (as opposed to uncorrelated in PCA). The new variables will be labeled as `IC1`, `IC2` and so on.

The "spatial sign" transformation (Serneels *et al*, 2006) projects the data for a predictor to the unit circle in p dimensions, where p is the number of predictors. Essentially, a vector of data is divided by its norm. Figures 1 and 2 show two centered and scaled descriptors from the MDRR data before and after the spatial sign transformation. The predictors should be centered and scaled before applying this transformation.

This transformation can also be applied using the `preProcess` class.

3.6 Data Splitting

The function `createDataPartition` can be used to create balanced splits of the data. If the `y` argument to this function is a factor, the random sampling occurs within each class and should preserve the overall class distribution of the data. For example, to create a single 80%\20% split of the MDRR data:

```
> set.seed(3456)
> set1index <- createDataPartition(mdrrClass, p = .8, list = FALSE, times = 1)
```

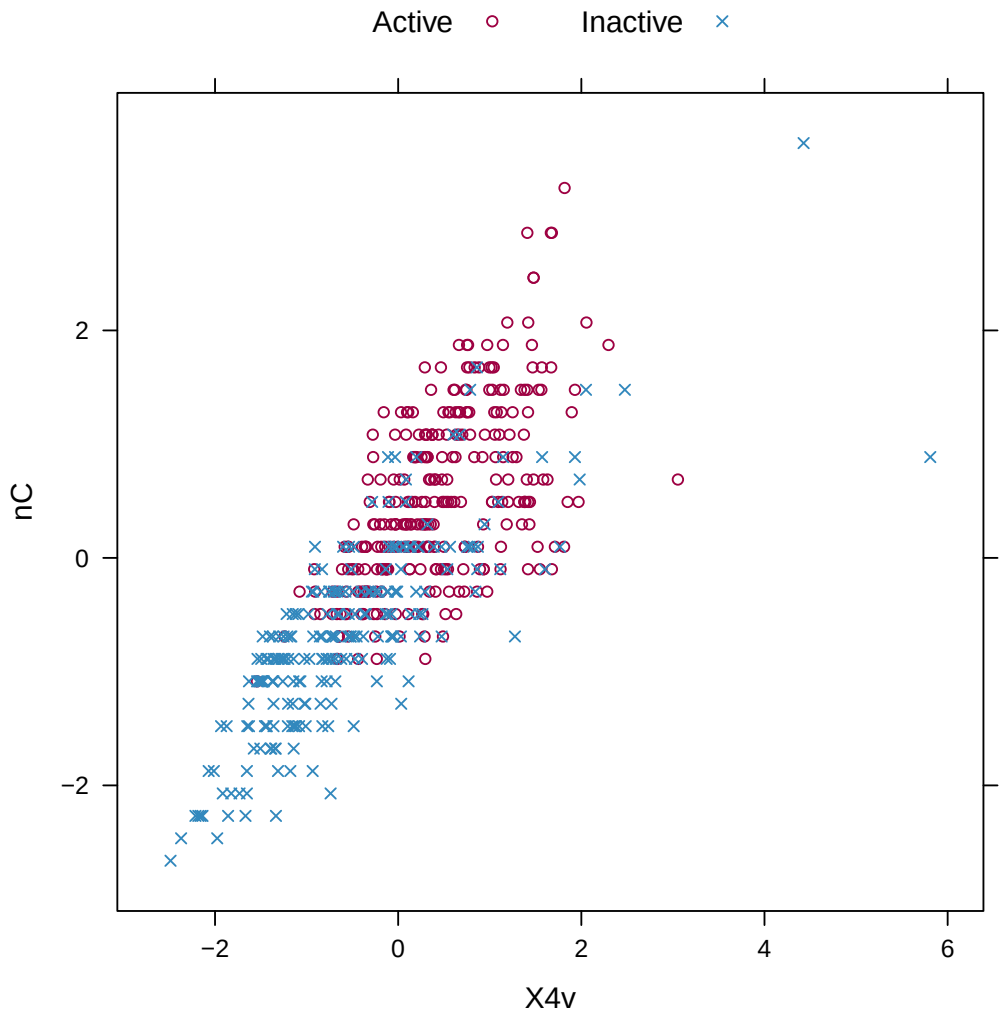

Figure 1: A plot of two descriptors from the MDRR training set (centered and scaled)

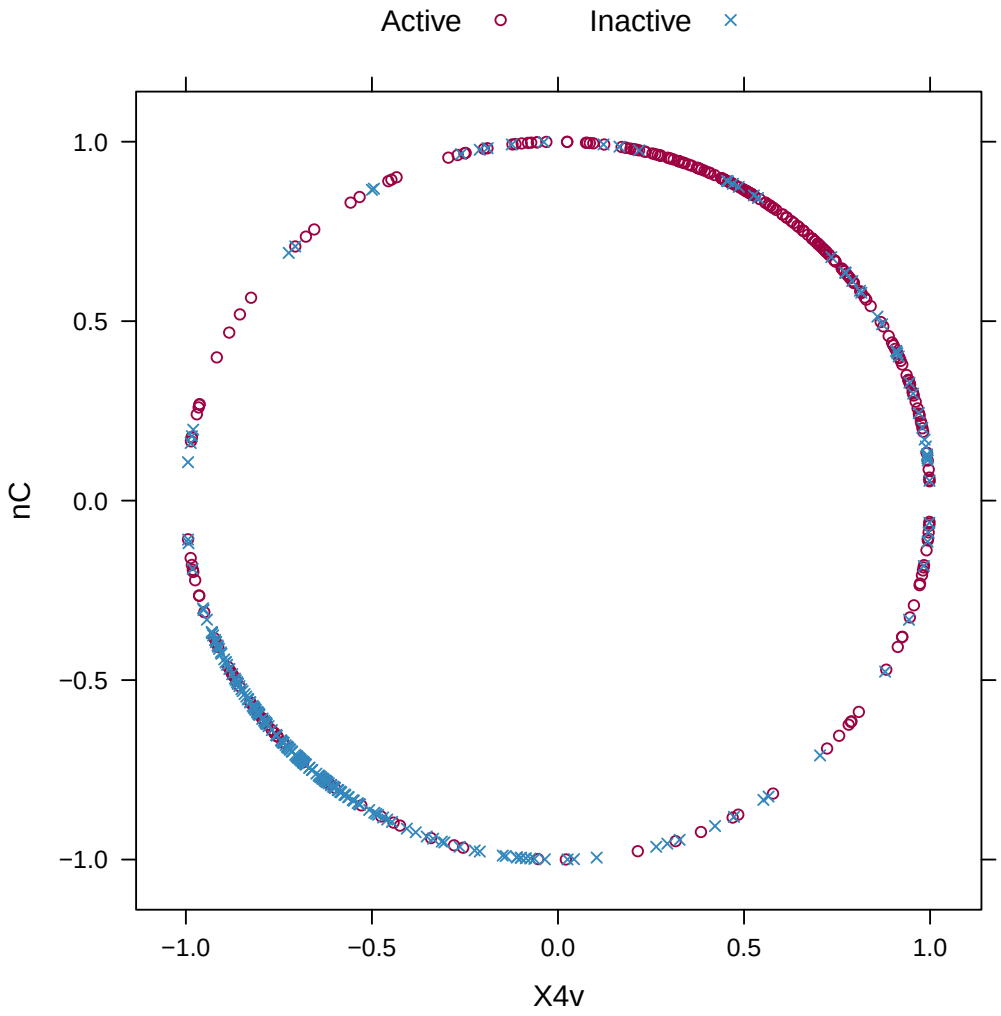

Figure 2: A plot of the two descriptors from the MDRR training set after the spatial sign transformation

```
> set1 <- mdrrClass[set1index]
> round(table(set1)/length(set1), 2)

set1
  Active Inactive
    0.57     0.43

> round(table(mdrrClass)/length(mdrrClass), 2)

mdrrClass
  Active Inactive
    0.56     0.44
```

The `list = FALSE` returns the data in an integer vector. This function also has an argument, `times`, that can create multiple splits at once; the data indices are returned in a list of integer vectors. Similarly, `createResample` can be used to make simple bootstrap samples and `createFolds` can be used to generate balanced cross–validation groupings from a set of data.

Also, the function `maxDissim` can be used to create sub–samples using a maximum dissimilarity approach (Willett, 1999). Suppose there is a data set A with m samples and a larger data set B with n samples. We may want to create a sub–sample from B that is diverse when compared to A. To do this, for each sample in B, the function calculates the m dissimilarities between each point in A. The most dissimilar point in B is added to A and the process continues. There are many methods in R to calculate dissimilarity. `caret` uses the `proxy` package. See the manual for that package for a list of available measures. Also, there are many ways to calculate which sample is "most dissimilar". The argument `obj` can be used to specify any function that returns a scalar measure. `caret` includes two functions, `minDiss` and `sumDiss`, that can be used to maximize the minimum and total dissimilarities, respectfully.

As an example, Figure 3 shows a scatter plot of two chemical descriptors for the Cox2 data. Using an initial random sample of 5 compounds, we can select 20 more compounds from the data so that the new compounds are most dissimilar from the initial 5 that were specified. The panels in Figure 3 show the results using several combinations of distance metrics and scoring functions. For these data, the distance measure has less of an impact than the scoring method for determining which compounds are most dissimilar.

4 Visualizing Data

The `featurePlot` function is a wrapper for different `lattice` plots to visualize the data.

For example, Figures 4, 5, 6 and 7 shows the default plot for continuous outcomes generated using the `featurePlot` function.

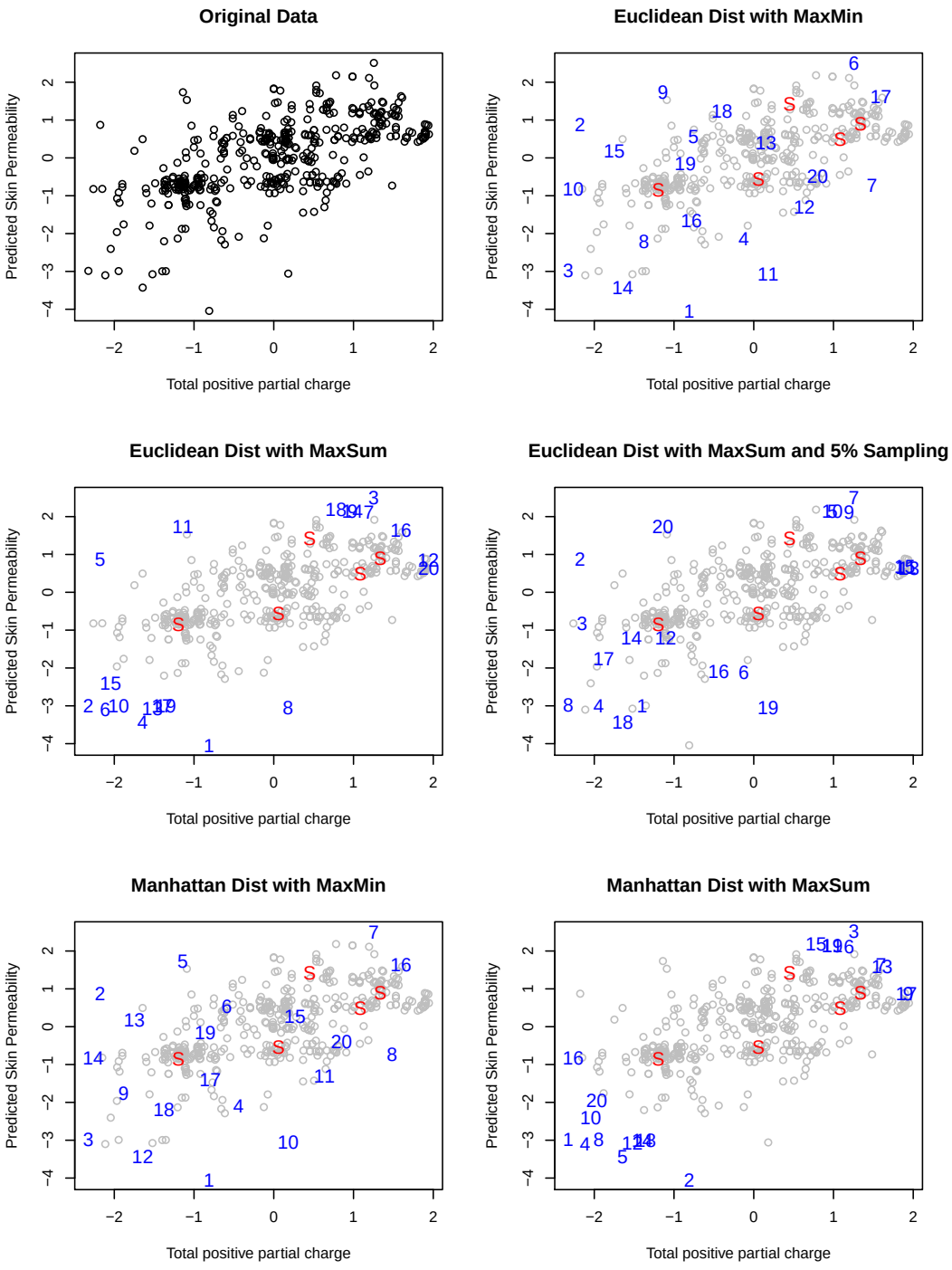

Figure 3: Examples of selecting 20 dissimilar compounds from an initial set of 5 (denoted by "S") for different combinations of arguments to the function `maxDissim`.

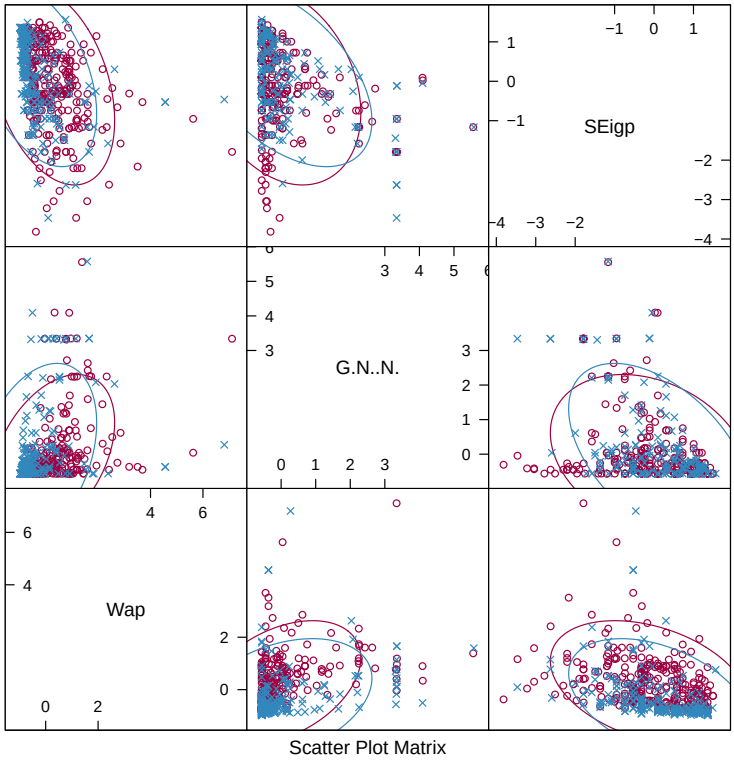

Figure 4: An example plot of three example predictors from the MDRR data using the `featurePlot` function with density ellipses for each class generated with the arguments `plot = "ellipse"` and a factor outcome.

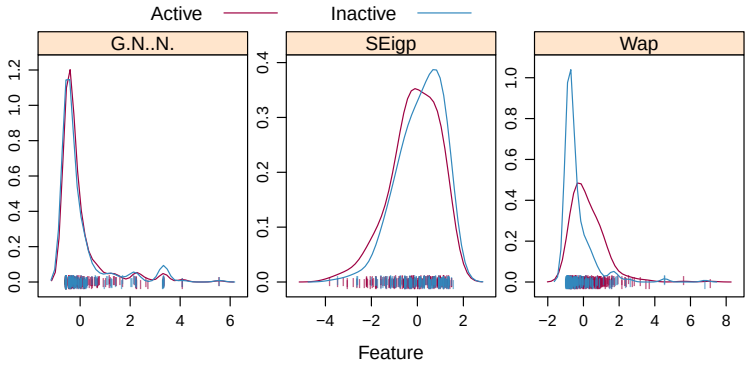

Figure 5: An example of using the `plot = "density"` argument in the `featurePlot` function with three example predictors from the MDRR data and a factor outcome.

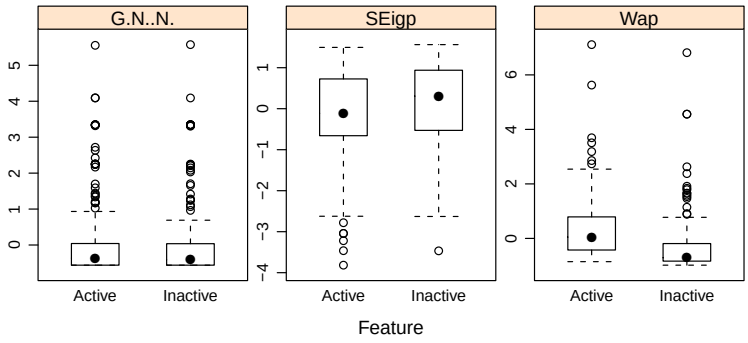

Figure 6: An example of using the `plot = "boxplot"` argument in the `featurePlot` function with three example predictors from the MDRR data and a factor outcome.

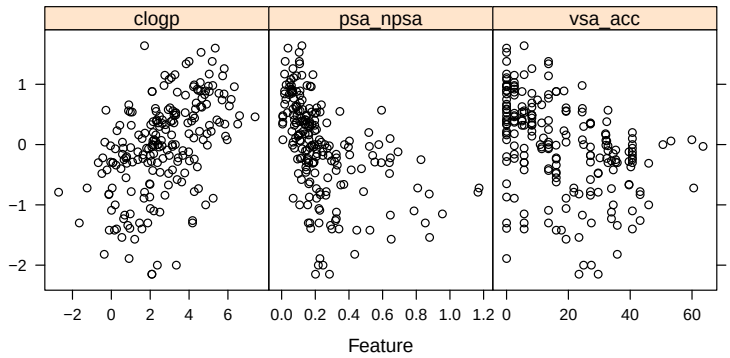

Figure 7: A set of scatter–plots for three example predictors from the blood–brain barrier data and a numeric outcome (on the y–axis).

5 Other Functions

5.1 Processing Affy Arrays

For Affymetrix gene chip data, RMA processing (Irizarry, 2003) is a popular method of processing gene expression data. However, for predictive modeling, it has a drawback in that the processing is batch oriented; if an additional sample is collected, the RMA processing must be repeated using all the samples. This is mainly because of two steps in the processing: the quantile normalization process and the calculation of expression. Quantile normalization normalizes the data such that the between–chip distributions have equivalent quantiles and is very effective in improving the quality of the data. It is possible to let the samples in the training set define the reference distribution for normalization. For the expression calculation, a robust method such as a trimmed mean can be used to summarize the probe level data into a single summary metric per sample. For example,

```
# first, let affy/expresso know that the method exists
normalize.AffyBatch.methods <- c(normalize.AffyBatch.methods, "normalize2Reference")

RawData <- ReadAffy(celfile.path=FilePath)

Batch1Step1 <- bg.correct(RawData, "rma")
Batch1Step2 <- normalize.AffyBatch.quantiles(Batch1Step1)
referencePM <- pm(Batch1Step2)[,1]
Batch1Step3 <- computeExprSet(Batch1Step2, "pmonly", "trimMean")

Batch2Step1 <- bg.correct(RawData2, "rma")
Batch2Step2 <- normalize.AffyBatch.normalize2Reference(Batch2Step1, ref = referencePM)
Batch2Step3 <- computeExprSet(Batch2Step2, "pmonly", "trimMean")
```

5.2 Yet Another k–Nearest Neighbor Function

knn3 is a function for k–nearest neighbor classification. This particular implementation is a modification of the knn C code and returns the vote information for all of the classes (knn only returns the probability for the winning class). There is a formula interface via

```
knn3(formula, data)
```

or by passing the training data directly

```
# x is a matrix, y is a factor vector
knn3(x, y)
```

There are also `print` and `predict` methods.

Similarly, `caret` contains a k–nearest neighbor regression function, `knnreg`. It returns the average outcome for the neighbor.

5.3 Partial Least Squares Discriminant Analysis

The `plsda` function is a wrapper for the `plsr` function in the `pls` package that does not require a formula interface and can take factor outcomes as arguments. The classes are broken down into dummy variables (one for each class). These 0/1 dummy variables are modeled by partial least squares.

From this model, there are two approaches to computing the class predictions and probabilities:

- the softmax technique can be used on a per-sample basis to normalize the scores so that they are more "probability like" (i.e. they sum to one and are between zero and one). For a vector of model predictions for each class x, the softmax class probabilities are computed as

$$prob_k = \frac{exp(x_k)}{\sum_{k=1}^{C} exp(x_k)}$$

  The predicted class is simply the class with the largest model prediction, or equivalently, the largest class probability. This is the default behavior for `plsda`.

- Bayes rule can be applied to the model predictions to form posterior probabilities. Here, the model predictions for the training set are used along with the training set outcomes to create conditional distributions for each class. When new samples are predicted, the raw model predictions are run through these conditional distributions to produce a posterior probability for each class (along with the prior). Bayes rule can be used by specifying `probModel = "Bayes"`. An additional parameter, `prior`, can be used to set prior probabilities for the classes.

The advantage to using Bayes rule is that the full training set is used to directly compute the class probabilities (unlike the softmax function which only uses the current sample's scores). This creates more realistic probability estimates (See Figure 8). The disadvantage is that a separate Bayesian model must be created for each value of `ncomp`, which is more time consuming.

For the MDRR data set, we can fit two PLS models using each technique and predict the class probabilities for the test set. Figure 8 shows the results.

Similar to `plsda`, `caret` also contains a function `splsda` that allows for classification using sparse PLS. A dummy matrix is created for each class and used with the `spls` function in the `spls` package. The same approach to estimating class probabilities is used for `plsda` and `splsda`.

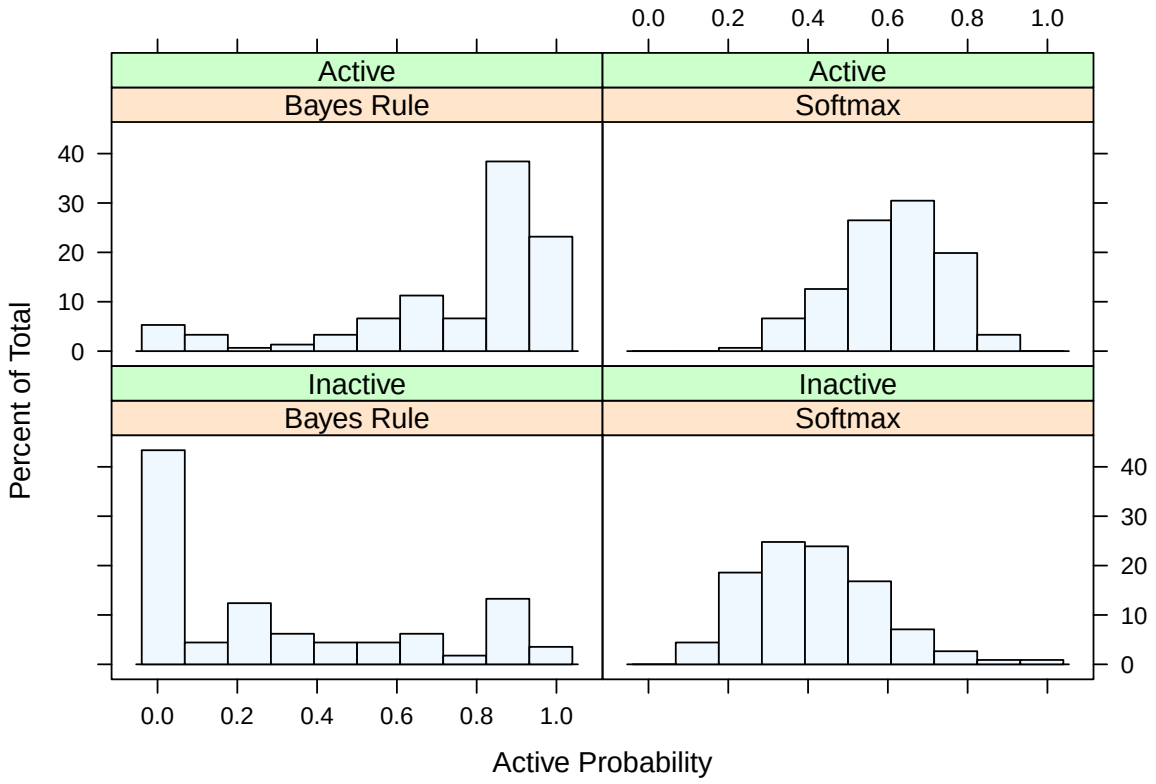

Figure 8: A comparison of two methods for computing class probabilities for PLS classification models on the MDRR data. Using a 5 component PLS model, the test set sample class probabilities were predicted using either the softmax function or Bayes rule. The histograms show the distributions of the active class probabilities for both methods and the true class of the samples. While Bayes rule takes more time to compute (0.64 seconds compared to 0.011 seconds for softmax), the probabilities using Bayes rule appear more appropriate since they encompass the full scale. The softmax probabilities are close to the middle of the distribution and rarely show high confidence in the predicted class. Both techniques show approximately the same accuracy (0.8 for Bayes and 0.77 for softmax) and Kappa values (0.58 for Bayes and 0.52 for softmax).

5.4 Bagged MARS and FDA

Multivariate adaptive regression splines (MARS) models, like classification/regression trees, are unstable predictors (Breiman, 1996). This means that small perturbations in the training data might lead to significantly different models. Bagged trees and random forests are effective ways of improving tree models by exploiting these instabilities. `caret` contains a function, `bagEarth`, that fits MARS models via the `earth` function. There are formula and non–formula interfaces.

Also, flexible discriminant analysis is a generalization of linear discriminant analysis that can use non–linear features as inputs. One way of doing this is the use MARS–type features to classify samples. The function `bagFDA` fits FDA models of a set of bootstrap samples and aggregates the predictions to reduce noise.

5.5 Neural Networks with a Principal Component Step

Neural networks can be affected by severe amounts of multicollinearity in the predictors. the function `pcaNNet` is a wrapper around the `preProcess` and `nnet` functions that will run principal component analysis on the predictors before using them as inputs into a neural network. The function will keep enough components that will capture some pre-defined threshold on the cumulative proportion of variance (see the `thresh` argument). For new samples, the same transformation is applied to the new predictor values (based on the loadings from the training set). The function is available for both regression and classification.

5.6 Class Distance Calculations

`caret` contains functions to generate new predictors variables based on distances to class centroids (similar to how linear discriminant analysis works). For each level of a factor variable, the class centroid and covariance matrix is calculated. For new samples, the Mahalanobis distance to each of the class centroids is computed and can be used as an additional predictor. This can be helpful for non–linear models when the true decision boundary is actually linear.

In cases where there are more predictors within a class than samples, the `classDist` function has arguments called `pca` and `keep` arguments that allow for principal components analysis within each class to be used to avoid issues with singular covariance matrices.

`predict.classDist` is then used to generate the class distances. By default, the distances are logged, but this can be changed via the `trans` argument to `predict.classDist`.

As an example, we can used the Fisher/Anderson iris data.

```
> inTrain <- seq(1, 150, 6)
> centroids <- classDist(iris[inTrain, 1:4], iris$Species[inTrain])
```

```
> distances <- predict(centroids, iris[-inTrain, 1:4])
> head(distances)
```

```
  dist.setosa dist.versicolor dist.virginica
2   2.0414919        5.399257       5.062646
3   0.3929024        5.475529       4.964052
4   0.6155916        5.335095       4.778670
5   1.3192459        5.736292       5.043468
6   2.3251196        5.736759       5.054561
8  -2.6255384        5.613625       4.965432
```

Figure 9 shows a scatterplot matrix of the class distances for the held–out samples.

For data sets with numerical outcomes, the data can be split into groups based on quantiles of the data (more fine granularity can be achieved with the `groups` argument to the `classDist` function).

For example, in the Fisher/Anderson iris data there are four predictors and a class variable describing three species of iris. These data are typically used to demonstrate how the four predictors can be used to predict the three species.

5.7 Independent Component Regression

The `icr` function can be used to fit a model analogous to principal component regression (PCR), but using independent component analysis (ICA). The predictor data are centered and projected to the ICA components. These components are then regressed against the outcome. The user needed to specify the number of components to keep.

The model uses the `preProcess` function to compute the latent variables using the `fastICA` package.

Like PCR, there is no guarantee that there will be a correlation between the new latent variable and the outcomes.

6 Session Information

- R version 2.11.0 Patched (2010-05-11 r51982), `x86_64-apple-darwin9.8.0`
- Locale: `en_US.UTF-8/en_US.UTF-8/C/C/en_US.UTF-8/en_US.UTF-8`
- Base packages: base, datasets, graphics, grDevices, grid, methods, splines, stats, tools, utils

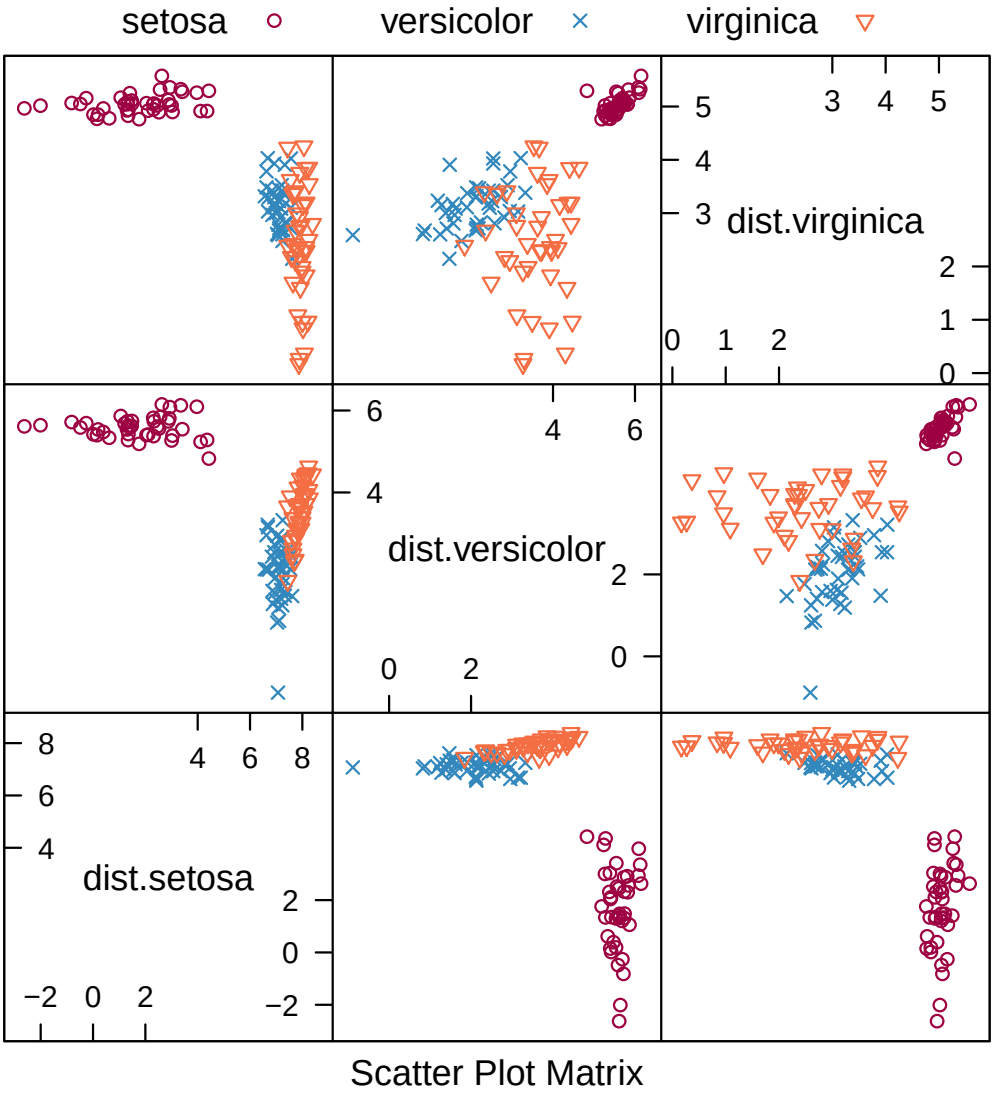

Figure 9: Class distances for the help–out samples for the Fisher/Anderson iris computed using the `classDist` function. The distances have been logged.

- Other packages: caret 4.45, class 7.3-2, e1071 1.5-24, ellipse 0.3-5, gbm 1.6-3.1, ipred 0.8-8, kernlab 0.9-10, klaR 0.6-3, lattice 0.18-5, MASS 7.3-5, mlbench 2.0-0, nnet 7.3-1, pls 2.1-0, plyr 0.1.9, proxy 0.4-6, randomForest 4.5-34, reshape 0.8.3, rpart 3.1-46, survival 2.35-8

7 References

Blake, J. F. (2000), "Chemoinformatics: predicting the physicochemical properties of 'drug–like' molecules," *Current Opinion in Biotechnology*, 11, 104–107.

Borggaard, C., Thodberg, H. H. (1992), "Optimal minimal neural interpretation of spectra," *Analytical Chemistry*, 64, 545–551.

Breiman, L. (1996), "Bagging predictors," *Machine Learning*, 24, 123–140.

Brodnjak-Voncina et al. (2005). "Multivariate data analysis in classification of vegetable oils characterized by the content of fatty acids," *Chemometrics and Intelligent Laboratory Systems*, 75, 31–45.

Clark, D. E., Pickett, S. D. (2000), "Computational methods for the prediction of 'drug–likeness'," *Drug Discov Today*, 5, 49–58.

Irizarry, RA, Bolstad BM, Collin, F, Cope, LM, Hobbs, B, and, Speed, TP (2003), "Summaries of Affymetrix Genechip probe level data," *Nucleic Acids Research*, 31, e15

Mente, S. R., Lombardo, F. (2005), "A recursive-partitioning model for blood–brain barrier permeation," *Journal of Computer–Aided Molecular Design*, 19, 465–481.

Serneels, S., De Nolf, E. and Van Espen, P. J. (2006), "Spatial sign preprocessing: a simple way to impart moderate robustness to multivariate estimators," *Journal of Chemical Information and Modeling*, 46, 1402–1409.

Sutherland, J.J., O'Brien L.A. and Weaver D.F. (2004), "A comparison of methods for modeling quantitative structure-activity relationships," *Journal of Medicinal Chemistry*, 47, 5541–5554.

Svetnik, V., Wang, T., Tong, C., Liaw, A., Sheridan, R. P. and Song, Q. (2005), "Boosting: An ensemble learning tool for compound classification and QSAR modeling," *Journal of Chemical Information and Modeling*, 45, 786 –799.

Willett, P. (1999), "Dissimilarity-Based Algorithms for Selecting Structurally Diverse Sets of Compounds," *Journal of Computational Biology*, 6, 447–457.