15 Miscellaneous Model Functions

Contents

15.1 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 or data frame, y is a factor vector
knn3(x, y)
```

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

For the Sonar data in the `mlbench` package, we can fit an 11-nearest neighbor model:

```r
library(caret)
library(mlbench)
data(Sonar)
set.seed(808)
inTrain <- createDataPartition(Sonar$Class, p = 2/3, list = FALSE)
## Save the predictors and class in different objects
sonarTrain <- Sonar[ inTrain, -ncol(Sonar)]
sonarTest  <- Sonar[-inTrain, -ncol(Sonar)]

trainClass <- Sonar[ inTrain, "Class"]
testClass  <- Sonar[-inTrain, "Class"]

centerScale <- preProcess(sonarTrain)
centerScale


## Created from 139 samples and 60 variables
##
## Pre-processing:
##   - centered (60)
##   - ignored (0)
##   - scaled (60)


training <- predict(centerScale, sonarTrain)
testing <- predict(centerScale, sonarTest)

knnFit <- knn3(training, trainClass, k = 11)
knnFit


## 11-nearest neighbor classification model
## Training set class distribution:
##
##  M  R
## 74 65


predict(knnFit, head(testing), type = "prob")
```

```
##              M         R
## [1,] 0.2727273 0.7272727
## [2,] 0.2727273 0.7272727
## [3,] 0.1818182 0.8181818
## [4,] 0.1818182 0.8181818
## [5,] 0.5454545 0.4545455
## [6,] 0.0000000 1.0000000
```

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

15.2 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. 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 but the disadvantage is that a separate Bayesian model must be created for each value of `ncomp`, which is more time consuming.

For the sonar data set, we can fit two PLS models using each technique and predict the class probabilities for the test set.

```r
plsFit <- plsda(training, trainClass, ncomp = 20)
plsFit
```

```
## Partial least squares classification, fitted with the kernel algorithm.
## The softmax function was used to compute class probabilities.
```

```r
plsBayesFit <- plsda(training, trainClass, ncomp = 20,
                     probMethod = "Bayes")
plsBayesFit
```

```
## Partial least squares classification, fitted with the kernel algorithm.
## Bayes rule was used to compute class probabilities.
```

```r
predict(plsFit, head(testing), type = "prob")
```

```
## , , 20 comps
##
##            M         R
## 4  0.6227621 0.3772379
## 6  0.5240553 0.4759447
## 12 0.3883679 0.6116321
## 16 0.1925378 0.8074622
## 17 0.1800970 0.8199030
## 19 0.1336795 0.8663205
```

```r
predict(plsBayesFit, head(testing), type = "prob")
```

```
## , , ncomp20
##
##               M          R
## 4  0.950775270 0.04922473
## 6  0.585468690 0.41453131
## 12 0.076098620 0.92390138
## 16 0.002768591 0.99723141
## 17 0.003715369 0.99628463
## 19 0.023820018 0.97617998
```

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`.

15.3 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.

This function is deprecated in favor of the `bag` function.

15.4 Bagging

The `bag` function offers a general platform for bagging classification and regression models. Like `rfe` and `sbf`, it is open and models are specified by declaring functions for the model fitting and prediction code (and several built-in sets of functions exist in the package). The function `bagControl` has options to specify the functions (more details below).

The function also has a few non-standard features:

- The argument `var` can enable random sampling of the predictors at each bagging iteration. This is to de-correlate the bagged models in the same spirit of random forests (although here the sampling is done once for the whole model). The default is to use all the predictors for each model.
- The `bagControl` function has a logical argument called `downSample` that is useful for classification models with severe class imbalance. The bootstrapped data set is reduced so that the sample sizes for the classes with larger frequencies are the same as the sample size for the minority class.
- If a parallel backend for the **foreach** package has been loaded and registered, the bagged models can be trained in parallel.

The function's control function requires the following arguments:

15.4.1 The `fit` Function

Inputs:

- `x` : a data frame of the training set predictor data.
- `y` : the training set outcomes.
- `...` arguments passed from `train` to this function

The output is the object corresponding to the trained model and any other objects required for prediction. A simple example for a linear discriminant analysis model from the **MASS** package is:

```
function(x, y, ...) {
   library(MASS)
   lda(x, y, ...)
}
```

15.4.2 The `pred` Function

This should be a function that produces predictors for new samples.

Inputs:

- `object` : the object generated by the `fit` module.
- `x` : a matrix or data frame of predictor data.

The output is either a number vector (for regression), a factor (or character) vector for classification or a matrix/data frame of class probabilities. For classification, it is probably better to average class probabilities instead of using the votes of the class predictions. Using the `lda` example again:

```
## predict.lda returns the class and the class probabilities
## We will average the probabilities, so these are saved
function(object, x) predict(object, x)$posterior


## function(object, x) predict(object, x)$posterior
```

15.4.3 The `aggregate` Function

This should be a function that takes the predictions from the constituent models and converts them to a single prediction per sample.

Inputs:

- `x` : a list of objects returned by the `pred` module.
- `type` : an optional string that describes the type of output (e.g. “class”, “prob” etc.).

The output is either a number vector (for regression), a factor (or character) vector for classification or a matrix/data frame of class probabilities. For the linear discriminant model above, we saved the matrix of class probabilities. To average them and generate a class prediction, we could use:

```
function(x, type = "class") {
  ## The class probabilities come in as a list of matrices
  ## For each class, we can pool them then average over them

  ## Pre-allocate space for the results
  pooled <- x[[1]] * NA
  n <- nrow(pooled)
  classes <- colnames(pooled)
  ## For each class probability, take the median across
  ## all the bagged model predictions
  for(i in 1:ncol(pooled))
  {
    tmp <- lapply(x, function(y, col) y[,col], col = i)
    tmp <- do.call("rbind", tmp)
    pooled[,i] <- apply(tmp, 2, median)
  }
  ## Re-normalize to make sure they add to 1
  pooled <- apply(pooled, 1, function(x) x/sum(x))
  if(n != nrow(pooled)) pooled <- t(pooled)
  if(type == "class")
  {
    out <- factor(classes[apply(pooled, 1, which.max)],
                  levels = classes)
  } else out <- as.data.frame(pooled)
  out
}
```

For example, to bag a conditional inference tree (from the **party** package):

```
library(caret)
set.seed(998)
inTraining <- createDataPartition(Sonar$Class, p = .75, list = FALSE)
training <- Sonar[ inTraining,]
testing  <- Sonar[-inTraining,]
set.seed(825)
baggedCT <- bag(x = training[, names(training) != "Class"],
                y = training$Class,
                B = 50,
                bagControl = bagControl(fit = ctreeBag$fit,
                                        predict = ctreeBag$pred,
                                        aggregate = ctreeBag$aggregate))
summary(baggedCT)
```

```
##
## Call:
## bag.default(x = training[, names(training) != "Class"], y
##  = training$Class, B = 50, bagControl = bagControl(fit =
##  ctreeBag$fit, predict = ctreeBag$pred, aggregate = ctreeBag$aggregate))
##
## Out of bag statistics (B = 50):
##
##        Accuracy     Kappa
##   0.0%   0.5000 -0.004065
##   2.5%   0.5357  0.014293
##  25.0%   0.6667  0.333794
##  50.0%   0.7027  0.397853
##  75.0%   0.7531  0.489808
##  97.5%   0.8009  0.573063
## 100.0%   0.8226  0.641430
```

15.5 Model Averaged Neural Networks

The `avNNet` fits multiple neural network models to the same data set and predicts using the average of the predictions coming from each constituent model. The models can be different either due to different random number seeds to initialize the network or by fitting the models on bootstrap samples of

the original training set (i.e. bagging the neural network). For classification models, the class probabilities are averaged to produce the final class prediction (as opposed to voting from the individual class predictions.

As an example, the model can be fit via `train`:

```
set.seed(825)
avNnetFit <- train(x = training,
                   y = trainClass,
                   method = "avNNet",
                   repeats = 15,
                   trace = FALSE)
```

15.6 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.

This function is deprecated in favor of the `train` function using `method = "nnet"` and `preProc = "pca"`.

15.7 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.