Caret R Package

Classification And REgression Training

Binabaji, Laxar, Pelzmann, Steinkellner

Introduction

The package contains tools for:

- data splitting ✓
- pre-processing (dummies, correlation, PCA or linear dependence)
- model tuning using resampling (use of popular *train* function) ✓
- feature selection (wrapper and filter methods)
- variable importance estimation (trees and forests) ✓

Remember Exercise 27?

"For the German data, try to find better generalized boosted models for *Class* using the available predictors, e.g., by increasing interaction depth."

How did you solve it?

Caret solves it easily.

Split data

```
require(caret)
load("German.Rda")
```

Load libraries

```
#split data:
```

```
set.seed(1)
traind <- createDataPartition(
  y = german$Class,
  ##outcome data
  p = 0.75,
  ##percentage of data in the training set
  list = FALSE
)
```

Stratified random data partition with 75% of the data in the training set

```
training <- german[traind,]
testing <- german[-traind,]
```

Split in training and testing data

Train and *trainControl*

```
gbmFit <- train(
  Class ~ .,
  data = training,
  method = "gbm",
  verbose = FALSE
)
```

Predict Class using all available predictors

Regression Model: generalized boosted models

Takes argument “verbose = FALSE” from gbm function

```
ctrl <- trainControl(method = "repeatedcv",
                     repeats = 3)
```

Modify resampling method:
“repeatedcv” = *K*-fold cross-validation
K is controlled by number argument and defaults to 10.

```
gbmFit1 <- train(
  Class ~ .,
  data = training,
  method = "gbm",
  verbose = FALSE,
  ##added:
  trControl = ctrl
)
```

Add control settings

gbmFit1 output

```
> gbmFit1
Stochastic Gradient Boosting

750 samples
 20 predictor
  2 classes: 'good', 'bad'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 675, 676, 675, 676, 675, 675, ...
Resampling results across tuning parameters:

  interaction.depth  n.trees  Accuracy   Kappa
  1                   50      0.7200713  0.1701654
  1                  100      0.7271944  0.2379431
  1                  150      0.7458805  0.3100201
  2                   50      0.7391955  0.2772213
  2                  100      0.7511906  0.3387763
  2                  150      0.7551617  0.3663378
  3                   50      0.7462070  0.3095884
  3                  100      0.7506934  0.3490235
  3                  150      0.7537688  0.3655527

Tuning parameter 'shrinkage' was held constant at a value of 0.1
Tuning parameter 'n.minobsinnode' was held constant at
 a value of 10
Accuracy was used to select the optimal model using the largest value.
The final values used for the model were n.trees = 150, interaction.depth = 2, shrinkage = 0.1 and n.minob
sinnode = 10.
```

Compared to normal *gbm*() function, no change to 0 and 1 is necessary

Our control settings

By default, *train* uses a search grid of 3 values for *interaction.depth* and *n.trees*, and fixes *shrinkage* and *n.minobsinnode*

Default metrics for classification problems are Accuracy (1 – MCE) and Cohen’s Kappa

Further tuning

```
grid <- expand.grid(interaction.depth = seq(1, 4, by = 1),
                    n.trees = seq(50, 250, by = 50),
                    shrinkage = c(0.01, 0.1),
                    n.minobsinnode = 10)
```

Defines values for tuning parameters

```
gbmFit2 <- train(
  Class ~ .,
  data = training,
  method = "gbm",
  verbose = FALSE,
  trControl = ctrl,
  ##added:
  tuneGrid = grid
)
```

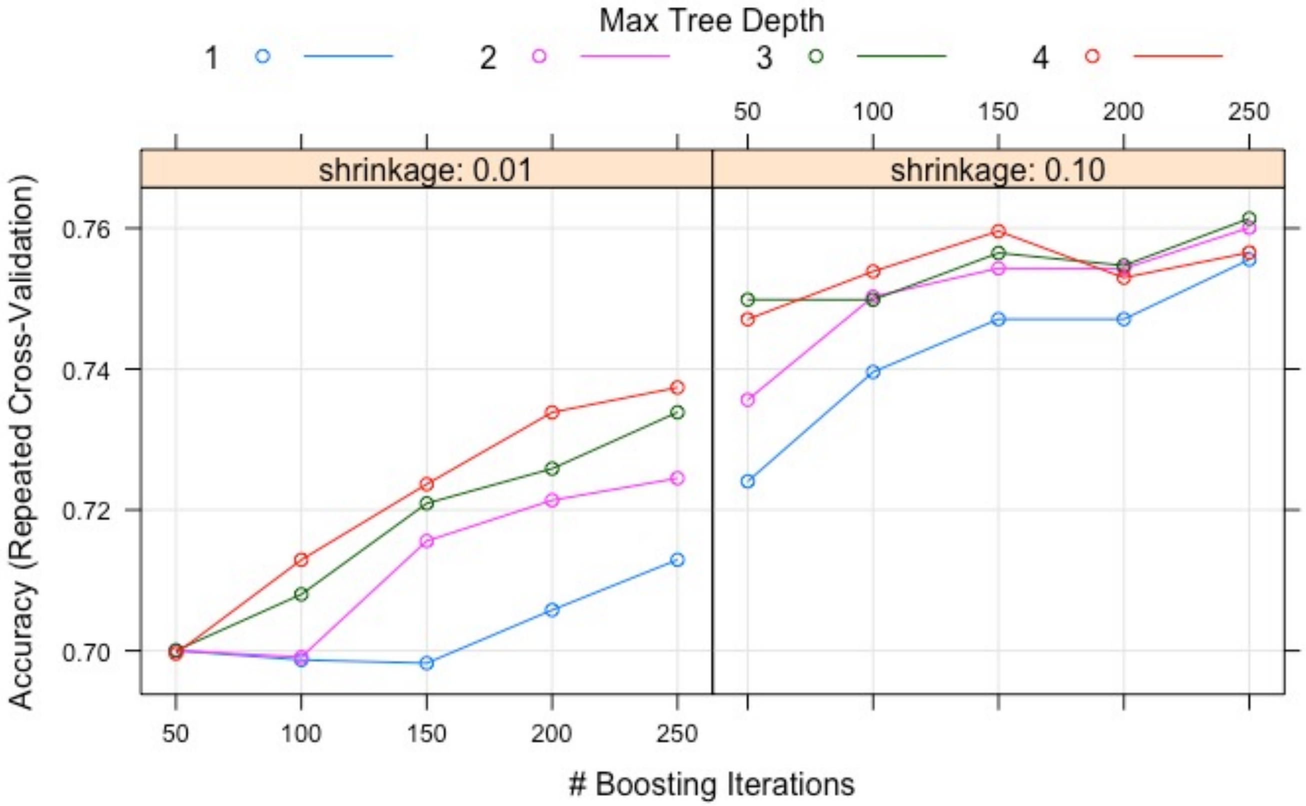
Add grid for tuning parameters

gbmFit2 output

The pre-specified values for the tuning parameters are used to fit the models.

```
> gbmFit2
Stochastic Gradient Boosting

750 samples
 20 predictor
  2 classes: 'good', 'bad'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 674, 676, 675, 676, 675, 675, ...
Resampling results across tuning parameters:

  shrinkage  interaction.depth  n.trees  Accuracy   Kappa
  0.01       1                   50      0.7000213   0.0000000000
  0.01       1                  100      0.6986818  -0.0026315920
  0.01       1                  150      0.6982315   0.0069302123
  0.01       1                  200      0.7057764   0.0498972704
  0.01       1                  250      0.7129118   0.0896148279
  0.01       2                   50      0.7000213   0.0000000000
  0.01       2                  100      0.6990910   0.0175867023
  0.01       2                  150      0.7155788   0.1060957685
  0.01       2                  200      0.7213630   0.1394226161
  0.01       2                  250      0.7244920   0.1639622114
  0.01       3                   50      0.7000213   0.0000000000
                       ⋮
  0.10       3                  150      0.7564843   0.3683977059
  0.10       3                  200      0.7547063   0.3718318687
  0.10       3                  250      0.7613795   0.3908592018
  0.10       4                   50      0.7470670   0.3223475600
  0.10       4                  100      0.7538652   0.3588086474
  0.10       4                  150      0.7595845   0.3830824595
  0.10       4                  200      0.7529761   0.3704772432
  0.10       4                  250      0.7565497   0.3842748486

Tuning parameter 'n.minobsinnode' was held constant at a value of 10
Accuracy was used to select the optimal model using the largest value.
The final values used for the model were n.trees = 250, interaction.depth = 3,
 shrinkage = 0.1 and n.minobsinnode = 10.
```

Graphics and prediction



Relationship between the resampled performance values, the tree depth and the number of trees.

```
> gbmClass <- predict(gbmFit2, testing)
> str(gbmClass)
 Factor w/ 2 levels "good","bad": 1 1 1 1 1 2 1 1 1 1 ...
```

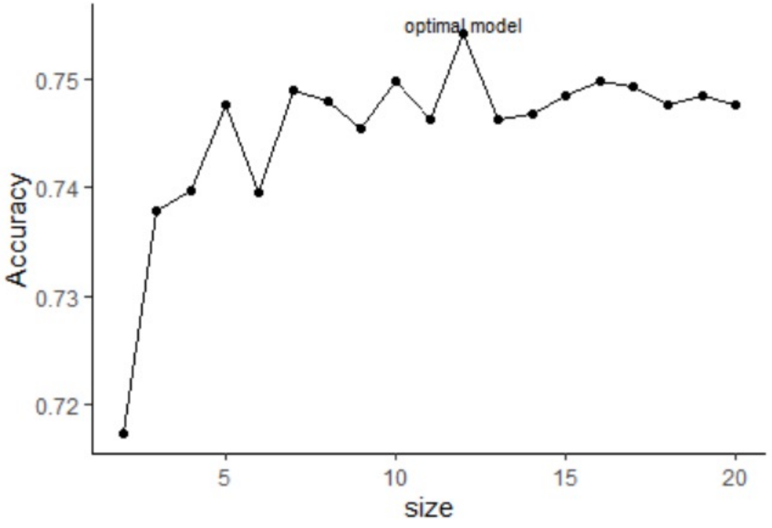
One can use the *predict* function on the testing data set now. This automatically uses the model that led to the best performance metrics.

Random Forest

```
grid <- expand.grid(.mtry = c(2:20))

rfFit1 <- train(Class~.,
                data=training,
                method="rf",
                verbose=F,
                trControl=ctrl,
                tuneGrid=grid,
                importance=T)
rfFit1

pred <- predict(rfFit1,
                newdata=testing,
                type="raw")

table(pred, testing$Class)
```

```
pred   good bad
  good  155  43
  bad    20  32
```

```
Random Forest

750 samples
 20 predictor
  2 classes: 'good', 'bad'

No pre-processing
Resampling: Cross-Validated (10 fold, repeated 3 times)
Summary of sample sizes: 674, 675, 676, 676, 675, 676, ...
Resampling results across tuning parameters:

  mtry  Accuracy   Kappa
   2    0.7173420  0.0884986
   3    0.7377825  0.2206016
   4    0.7396205  0.2503044
   5    0.7475734  0.2850655
   6    0.7395551  0.2683001
   7    0.7489069  0.2984537
   8    0.7480117  0.2972437
   9    0.7453275  0.2959384
  10    0.7497784  0.3068227
  11    0.7462402  0.3049670
  12    0.7542289  0.3264476
  13    0.7462341  0.3107351
  14    0.7467027  0.3131648
  15    0.7484572  0.3246657
  16    0.7498203  0.3254891
  17    0.7493518  0.3267853
  18    0.7475857  0.3200283
  19    0.7484452  0.3265409
  20    0.7475979  0.3238201

Accuracy was used to select the optimal model using the largest value.
The final value used for the model was mtry = 12.
```



Additional final quick notes

trainControl

- the resampling method: "boot", "cv", "LOOCV", "LGOCV", "repeatedcv", "none"

TuneGrid

- The argument tuneGrid can take a data frame with columns for each tuning parameter.
- The column names should be the same as the fitting function's arguments.

Metric

- RMSE, R2, and the mean absolute error (MAE) are computed for regression
- while accuracy and Kappa are computed for classification.

Resamples()

- to characterize the differences between models via their resampling distributions
- A list of models
- Have same metrics argument

```
resamps <- resamples(list(GBM = gbmFit3,
                          SVM = svmFit,
                          RDA = rdaFit))
```

ParallelProcessing

- use DoParallel package and set number of parallel workers
- registerDoParallel () to start and stopCluster() to get back to default

```
ibrary(doParallel)
cl <- makePSOCKcluster(5)
registerDoParallel(cl)

## All subsequent models are then run in parallel
model <- train(y ~ ., data = training, method = "rf")
stopCluster(...)
## When you are done:
stopCluster(cl)
```

The end