Exercise 7

Business Analytics and Data Science WS18/19

Introduction

In the last exercises, you have successfully constructed a range of models that seem to predict reasonably well on the data that they were trained on. And so far we have been measuring the performance on the very same data points that the models were trained on. The reasonable question to ask is "how well does this model predict outcomes in the future?" or, in other words, if the model generalizes to other, yet unseen data. If not, we would have fallen into the trap of *overfitting* the training data. We will avoid this by withholding some of our data from the model during training and using that part of the data to check the performance of the model. In this exercise, we will start with introducing several performance metrics (*confusion matrices*, *receiver-operating-characteristic* and *area-under-the-curve*) to measure and compare model performance, and then look at two approaches (*split sampling* and *cross-validation*) to address the risk of overfitting.

We will use these approaches to answer two questions, where we will need a fresh data set for each question: - How well can our model be expected to perform on new data? We test this on the test data. (Model assesssment) - What is the best parameter choice for our model? We test this on the test data. (Model selection)

Exercise 7.1: Performance measures

When it comes to classification task with binary outcome, confusion matrices and ROC curves standard metrics to evaluate of model's performance. Let's use it to compare the performance of *logit* and *rpart* models that we learnt to train before.

1. Start with loading the *loans* dataset using the usual Helper Function. Train a logit model (lr) and a decision tree (dt), using full dataset (as always, we try to predict BAD)

2. Confusion matrix
   A confusion matrix compares the target values in the data to the predictions made by the classification model by providing a count of correct and incorrent classifications per class. These values can in turn be used to calculate a range of performance measures, e.g. accuracy, specificity and sensitivity. Install package **caret**, which will help you to create readable cross-tables using the function **confusionMatrix()**. For the logit and pruned decision tree model above, plot the true values vs. the model prediction in a confusion matrix. Hint: Confusion tables require discrete class predictions and discrete target values. Choose a threshold value **tau** and transform you probability predictions to class predictions.

3. ROC curves
   Receiver operating characteristic curves, whose strange name is a remnant of their original use with radar systems in WWII, illustrates the performance of a binary classifier for different cut-offs (probability thresholds). For each cut-off **tau** between 0 and 1, it plots the true positive rate (a.k.a. sensitivity) against the false positive rate (a.k.a. 1 - specificity). Take a minute to see how these concepts relate to each other.
   The resulting ROC curves lie between the 45? baseline (no predictive power) and the upper left corner of the plot. The further the ROC curve lies towards the corner, the more accurate the model. Again, take a minute to really understand why this is the case. Keep in mind that each point on the ROC curve visualizes the true and false positive rates given a value for the threshold **tau**.
   Use package **hmeasure** to plot the ROC curves for the logistic regression *lr*, Naive Bayes *nb* and the pruned decision tree *dt*. The package requires the predictions to be in a data frame with one column for every model (call it **predictions**). Use the **HMeasure()** function to create an **HMeasure** object **h** containing all the necessary information for the plot. Then create the plot using function **plotROC()**.

4. The Area Under the Curve (AUC)
   The AUC is a way to quantify the performance of a classifier illustrated by the ROC curve in a single

value. It represents the area under a ROC curve, so it takes on values between 0.5 and 1, where higher is better. Compute the AUC values for all ROC curves. HINT: Extract the AUC values from the **HMeasure** object.

```
# Load necessary packages
library("rpart")
library("caret")

source("BADS-HelperFunctions.R")
loans <- get.loan.dataset()
# If the previous line did not work on your machine, the most likely
# cause is that something with the file directories went wrong.
# Check whether the data is available in your working directory.
loans$EMPS_A <- ifelse(loans$EMPS_A %in% c("U", "N", "Z"), "O", loans$EMPS_A)

lr <- glm(BAD~., data = loans, family = binomial(link = "logit"))
dt <- rpart(BAD ~ ., data = loans, cp = 0.001)

yhat.lr <- predict(lr, newdata = loans, type = "response")
yhat.dt <- predict(dt, newdata = loans, type = "prob")[,2]

# To produce confusion tables, we need discrete class predictions.
# So far, our two classifier gave us probabilistic predictions.
# Hence, we need to define a cut-off. Without additional information
# how this should be done, we pick an arbitrary cut-off of 0.5
tau <- 0.5
# Deal with logistic regression:
# convert probability prediction to discrete class predictions
yhat.dt.class <- factor(yhat.dt> tau, labels = c("good","bad"))   # mind the order!
yhat.lr.class <- factor(yhat.lr> tau, labels = c("good","bad"))


# We can create a simple confusion table with base function table.
# Using, for example, the logit classifier, this equates to:
table(yhat.lr.class, loans$BAD)
```

```
##
## yhat.lr.class good bad
##          good  869 275
##          bad    33  48
```

```
# Function confusionMatrix in the "caret" package gives a more readable confusion matrix
# and automatically calculates several performance metrics
confusionMatrix(data = yhat.dt.class, reference = loans$BAD)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction good bad
##       good  822 144
##       bad    80 179
##
##                Accuracy : 0.8171
##                  95% CI : (0.7943, 0.8384)
##     No Information Rate : 0.7363
##     P-Value [Acc > NIR] : 1.665e-11
```

```
##
##                  Kappa : 0.4971
##  Mcnemar's Test P-Value : 2.561e-05
##
##             Sensitivity : 0.9113
##             Specificity : 0.5542
##          Pos Pred Value : 0.8509
##          Neg Pred Value : 0.6911
##              Prevalence : 0.7363
##          Detection Rate : 0.6710
##    Detection Prevalence : 0.7886
##       Balanced Accuracy : 0.7327
##
##        'Positive' Class : good
##
```

```
confusionMatrix(       yhat.lr.class,             loans$BAD,positive = "good")
```

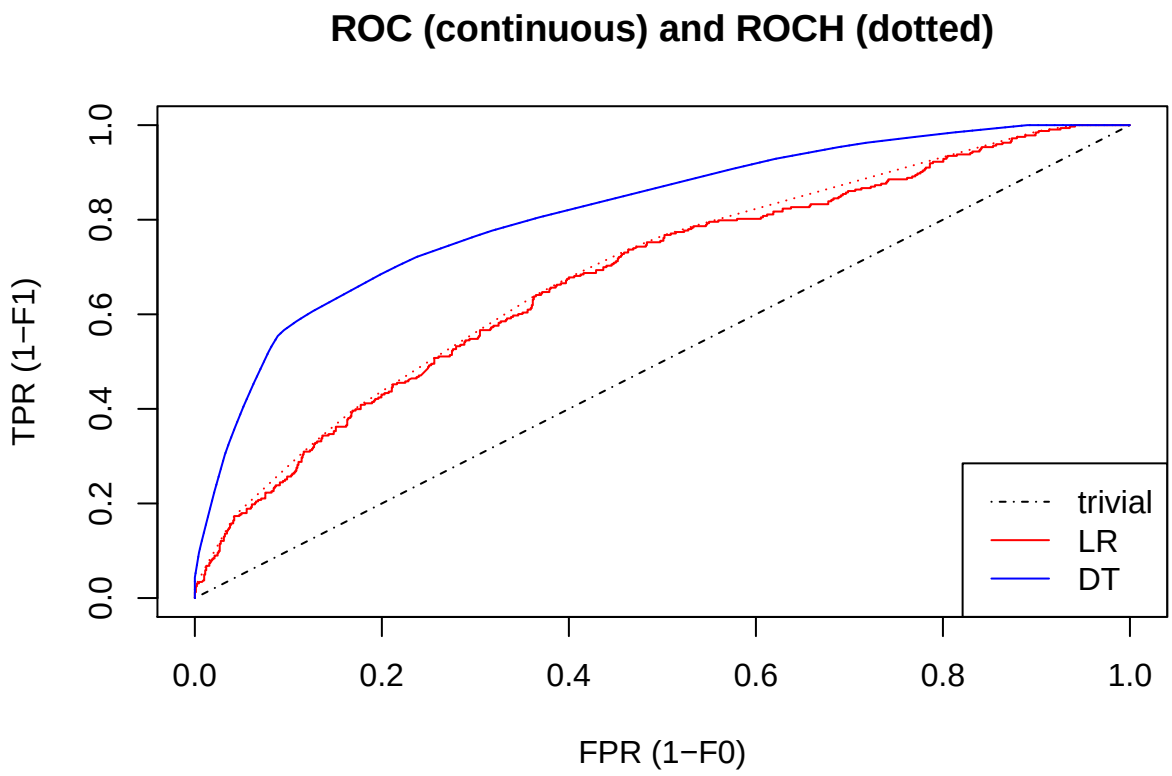
```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction good bad
##       good  869 275
##       bad    33  48
##
##                Accuracy : 0.7486
##                  95% CI : (0.7233, 0.7727)
##     No Information Rate : 0.7363
##     P-Value [Acc > NIR] : 0.1737
##
##                   Kappa : 0.1475
##  Mcnemar's Test P-Value : <2e-16
##
##             Sensitivity : 0.9634
##             Specificity : 0.1486
##          Pos Pred Value : 0.7596
##          Neg Pred Value : 0.5926
##              Prevalence : 0.7363
##          Detection Rate : 0.7094
##    Detection Prevalence : 0.9339
##       Balanced Accuracy : 0.5560
##
##        'Positive' Class : good
##
```

```
## ROC curves ####

# Plot the ROC curves for logistic regression and a
#   decision tree (in the same chart).

# Now we can start with the plot. We use the hmeasure package. Alternativ
# packages are pROC and ROCR
library("hmeasure")
# The hmeasure packages requires all predictions to be available in
# one data.frame
```

```
predictions.roc <- data.frame(LR = yhat.lr, DT = yhat.dt)
# Create on object of type hmeasure. This is basically the main
# call when using the hmeasure package. The performs many different
# tasks and wraps-up all results in one variable of type hmeausre.
# Using this variable, we can draw different plots including an ROC curve
h <- HMeasure(true.class = as.numeric(loans$BAD=="bad"), scores = predictions.roc)
plotROC(h, which = 1) # which = 1 says that we want an ROC curve; see the help system for more informat
```




```
# 3) Compute the AUC of the classifiers
#    Actually, this has already been done. We just need to extract the results
#    from the hmeasure object
h$metrics["AUC"]
```

```
##          AUC
## LR 0.6783309
## DT 0.8167677
```

Exercise 7.2: Split-sample testing

Now let's abandon the dangerous practices of predicting on the same set and move on to splitting. 1. Split your data randomly into a training set of 60% and a test set of 40% by creating two new data frames **train** and **test** to which you randomly assign the correct proportion of observations from data set **loans**. Hint: Check out function **sample()**.

2. Train a logistic regression model **lr2** and the decision trees **dt2** using function rpart *on the training data only*. Use your models to make a prediction for the credit risk of the applicants *in the test data set*.

3. Compare the predicted outcomes to the actual outcomes on the test data set and calculate the brier score. Use the performance measures from task 1 to compare the models.

```
# The exercise asks for a split-sample evaluation of the two classifiers.
# a) Randomly split your data
set.seed(123)
n <- nrow(loans)
sample.size <- ceiling(n*0.6) # split point to partition into training and test set / size of the train

# Option 1:
# Randomly draw a set of row indices
idx.train.opt1 <- sample(n, sample.size) # Draw a random sample of size sample.size from the integers 1
train.opt1 <- loans[idx.train.opt1, ] # training set
test.opt1 <-  loans[-idx.train.opt1, ] # test set (drop all observations with train indeces)

# Option 2:
# Draw a random stratified sample in which both train and test set have roughly the same ratio of the t
# Package caret has several very helpful functions to aid with data preparation
# Its function creatDataPartition returns the indices of a stratified training set with size p * size o

idx.train <- createDataPartition(y = loans$BAD, p = 0.6, list = FALSE) # Draw a random, stratified samp
train <- loans[idx.train, ] # training set
test <-  loans[-idx.train, ] # test set (drop all observations with train indeces)


# Develop models using the training set and compute test set predictions
lr2 <- glm(BAD~.    , data = train, family = binomial(link = "logit"))
dt2 <- rpart(BAD ~ ., data = train, cp = 0.001)


yhat.dt2 <- predict(dt2, newdata = test, type = "prob")[,2]
yhat.lr2 <- predict(lr2, newdata = test, type = "response")
yhat.benchmark <- rep(sum(train$BAD == "bad")/nrow(train), nrow(test))
yhat.test <- list("DT"=yhat.dt2, "lr" = yhat.lr2, "benchmark" = yhat.benchmark)

#  Assess predictive accuracy on the test set
y.test <- as.numeric(test$BAD=="bad") # This is a good example of why you need to be careful when trans
# Define a function that computes the brier score when given a binary vector of outcomes and a vector o
BrierScore <- function(y, yhat){
  sum((y - yhat)^2) / length(y)
}
# Apply the brierScore function to each element of y.test, i.e. all the prediction vectors
brier.test <- sapply(yhat.test, BrierScore, y = y.test, USE.NAMES = TRUE)
print(brier.test)
```

```
##        DT        lr benchmark
## 0.2000941 0.1878988 0.1942113
```

```
# A convenient if-else construct that reports the findings in a human-readable fashion
idx.best <- which.min(brier.test)
if (length(idx.best) == 1)  {
  sprintf("Model %s predicts most accurately.", names(brier.test)[idx.best])
}else {sprintf("Two or more classifiers achieve the same level of accuracy.")}
```

```
## [1] "Model lr predicts most accurately."
```

```
# Go back and compare these scores that you calculated on the complete data set back in previous task
```

Exercise 7.3: Repeated cross-validation

Split sampling is a simple approach and often used in practice. However, a large part of the data is "lost"" for model training and the results from split sampling depend on the random sample that is drawn for testing, i.e. for a different random sample the results are slightly different. More efficient use of the data and a more robust estimate of model performance can be achieved through k-fold cross-validation. For k-fold cross-validation, the data set is split into k subsets. Each fold is used as test set once, while a model is trained on the union of the remaining k-1 folds (as training set). In the end, the average performance is reported.

1. Manually perform k-fold cross-validation of logistic regression without using a package for cross-validation. Specifically, split the data into k subsets of equal size. Write a loop that for each subset:
   - trains the model on the remaining k-1 subsets.
   - classifies the observations in the test subset
   - calculates the brier score and saves it to a vector **results** of length k.
2. Visualize the model performance with a boxplot on the brier scores.

```
#---------------------------------
# DISCLAIMER
#
# The solution tries to illustrate the most important concepts
# to solve the task. To that end, clarity is more important than
# robustness. Hence, you may see an error when executing the
# following codes. If you do, just rerun the code a few times.
# It should work in most cases.
# The reason why you may see an error is that the random sampling
# may lead to a training set that does not include all levels for the
# factor variables. For example, you may get a training set without
# any BAD risks. The same problem may occur - and is more likely -
# with the other factor variables (e.g., RES or EMPS_A). We can mitigate this
# through revising our sampling approach, but this is beyond the scope
# of this demo; there are nice sampling functions readily available to
# take care of factors. We'll cover these in later exercises.

# END OF THE DISCLAIMER
#---------------------------------

#### k-fold cross-validation ####

# We are explicitly asked to not use a black-box cross-validation routine
# in some R package. However, just for your information, packages that support
# cross-validation include DAAG, CARET, E1071

# Note that the exercise did not specify which indicator we should use to
# assess predictive performance. Here, we
# use the MSE (i.e., Brier Score).

# It is often useful to shuffle your data prior to drawing cross-validation samples.
# To that end, we once more use the sample function; this time, however, without
# storing the indices
set.seed(123)
loans.rnd <- loans[sample(nrow(loans)),]
# Create k folds of approximately equal size
```

```r
k <- 5
folds <- cut(1:nrow(loans.rnd), breaks = k, labels = FALSE)
# The variable folds is a vector of integer values from 1 to k
folds
```

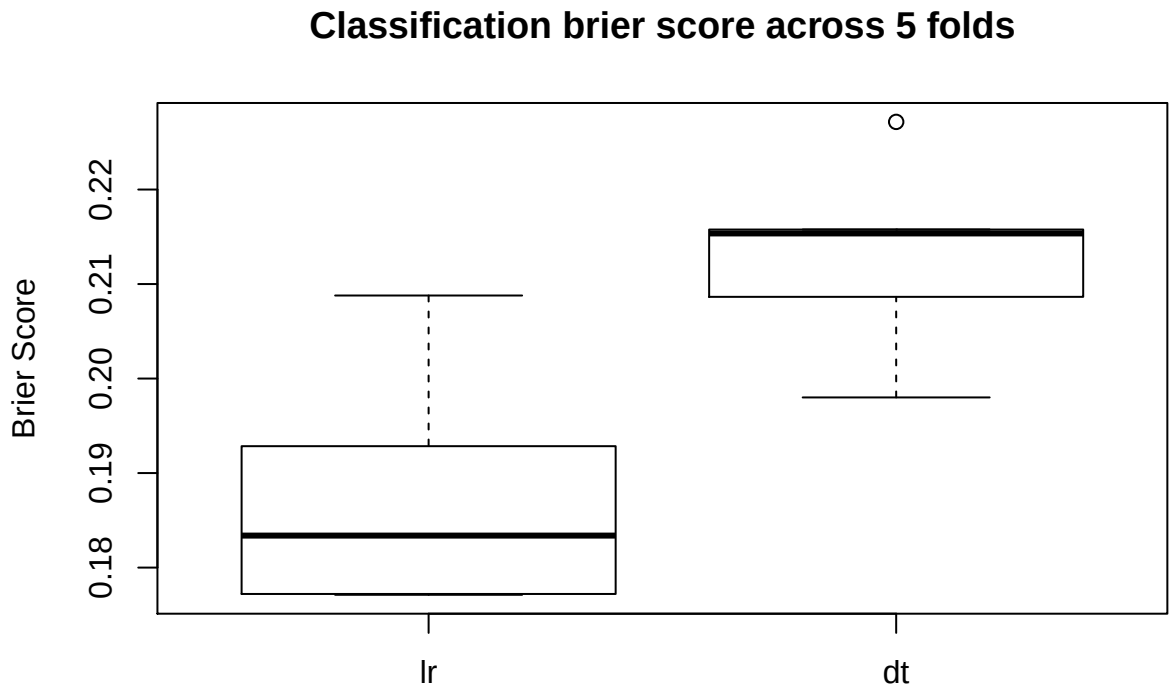
```
##    [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##   [35] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##   [69] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [103] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [137] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [171] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [205] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
##  [239] 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [273] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [307] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [341] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [375] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [409] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [443] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
##  [477] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
##  [511] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
##  [545] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
##  [579] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
##  [613] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
##  [647] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
##  [681] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
##  [715] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 4 4 4 4 4 4 4 4 4 4 4 4
##  [749] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
##  [783] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
##  [817] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
##  [851] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
##  [885] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
##  [919] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4
##  [953] 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 5 5 5 5 5 5 5
##  [987] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
## [1021] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
## [1055] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
## [1089] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
## [1123] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
## [1157] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
## [1191] 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5 5
## [1225] 5
```

```r
# We can use this vector in a logical indexing expression to query the
# training and test data in every cross-validation iteration.
# Make sure to read the online help of the R function cut to fully understand
# this approach
?cut

# Cross-validation loop
# Vector to store results (i.e., performance estimates per CV iteration)
results <- data.frame(lr = numeric(length = k), dt = numeric(length = k))
# recall that we need a cut-off to calculate crisp classifications
for (i in 1:k) {
```

```r
  # Split data into training and test
  idx.test <- which(folds == i, arr.ind = TRUE)
   cv.train <- loans.rnd[-idx.test,]
  cv.test <- loans.rnd[idx.test,]
  # Build and evaluate models using these partitions
  lr <- glm(BAD~., data = cv.train, family = binomial(link = "logit"))
  dt <- rpart(BAD ~ ., data = cv.train, cp = 0.001) # create decision tree classifier
  yhat.lr <- predict(lr, newdata = cv.test, type = "response")
  yhat.dt <- predict(dt, newdata = cv.test, type = "prob")[,2]
  # We use our above function to calculate the classification error
  results[i, "lr"] <- BrierScore(as.numeric(cv.test$BAD)-1, yhat.lr)
  results[i, "dt"] <- BrierScore(as.numeric(cv.test$BAD)-1, yhat.dt)
}

# Average performance ####
# Finally, we are probably interested in the average performance
# of our model across the cross-validation iterations. This is
# readily available as the mean classifcation error:
cv.perf <- apply(results, 2, mean)
cv.perf.sd <- apply(results, 2, sd)
# Now plot the results
txt <- paste("Classification brier score across", as.character(k), "folds", sep=" ")
boxplot(results,  ylab="Brier Score",
        main = txt)
```



```r
#-----------------------------------------------------------------
# Excursus
#*****************************************************************
# One way to improve the previous code would be to write a
# little helper function that performs the model building and
# evaluation. The benefit would be that we only need a single
# line instead of three (as above) to perform the three tasks
# training, computing predictions, and assessing accuracy.
# Here, we implemented the wrong classification rate as the
# loss function
# In particular, to develop such function:
lr.helper <- function(trainingData = NULL, testData = NULL, tau = 0.5) {
  lr <- glm(BAD~., data = trainingData, family = binomial(link = "logit"))
  yhat <- factor( predict(lr, newdata = testData, type = "response") >= tau, labels = c("GOOD", "BAD"))
  tab <- table(as.numeric(testData$BAD=="bad"), yhat)
  return(1 - sum(diag(tab)) / sum(tab)) # Wrong classification rate
}
# we could then call the function as follows
err <- lr.helper(train, test)
# to obtain the performance statistics from an individual
# iteration of the cross-validation.
err
```

```
## [1] 0.2699387
```

```r
#*****************************************************************
```