Exercise: Model assessment and cross-validation

In this exercise, you will know:

- How to calculate performance measures, such as classification accuracy/error rate, sensitivity and specificity
- How to write self-defined functions
- How to get ROC curves
- How to use cross-validation to tune k for kNN

Don't forget to change your working directory!

Calculate classification accuracy

Get the training and test sets and use `knn` to obtain predictions of the test set.

```
library(class)
# get indexes for training data
n=25 # the number of training data in each class
NN=dim(iris3)[1]# the total number of observations in each class
set.seed(983)
index_s=sample(1:NN,n)
index_c=sample(1:NN,n)
index_v=sample(1:NN,n)
# get training and test set
train_rand = rbind(iris3[index_s,,1], iris3[index_c,,2], iris3[index_v,,3])
test_rand = rbind(iris3[-index_s,,1], iris3[-index_c,,2], iris3[-index_v,,3])
# get class factor for training data
train_label= factor(c(rep("s",n), rep("c",n), rep("v",n)))
# get class factor for test data
test_label_true=factor(c(rep("s",NN-n), rep("c",NN-n), rep("v",NN-n)))
# classification using knn, with k=5
kk=5 # number of nearest neighbours
set.seed(275)
knn_pred=knn(train=train_rand,test=test_rand,cl=train_label, k=kk, prob=FALSE)
```

How do we calculate the classification accuracy or error rate of applying the `knn` model on the test set?

Method 1

```
cl <- factor(c(rep("s",25), rep("c",25), rep("v",25)))
test <- rbind(iris3[26:50,,1], iris3[26:50,,2], iris3[26:50,,3])
table(knn_pred,cl)
```

```
##         cl
## knn_pred  c  s  v
##        c 25  0  3
##        s  0 25  0
##        v  0  0 22
```

```
(22+25+25)/75
```

```
## [1] 0.96
```

```
sum(diag(table(knn_pred,cl)))/nrow(test)
```

```
## [1] 0.96
```

We usually prefer the last line than the second last line. This is because if we change the training and test sets, the table may be changed. In this case, we can still use the last line to calculate the accuracy while we have to change the numbers in the second last line to get the correct answer.

Method 2

```
test_label_true=factor(c(rep("s",25), rep("c",25), rep("v",25)))
sum(knn_pred==test_label_true)/length(test_label_true)
```

```
## [1] 0.96
```

```
mean(knn_pred==test_label_true)
```

```
## [1] 0.96
```

Make sure you understand all the codes.

You can choose the one you like or think about other ways to calculate the classification accuracy. Try to calculate the error rate by yourself.

Play with the above codes, to see how change of training/test data, change of variables or change of k will affect the classification accuracy.

Exercise: Calculate the classification accuracies of the predictions in the Standardise data section in the previous exercise.

More on model assessment

The `Caravan` data have 85 predictors that measure demographic characteristics for 5,822 individuals. The response variable is `Purchase`, which indicates whether or not a given individual purchases a caravan insurance policy. {Note that in this dataset, only 6% people purchase a caravan insurance policy. The two classes are very imbalanced!}

This data is part of the `ISLR` library. To use this dataset:

```
library(ISLR)
#summary(Caravan)
```

From `summary`, we can see that some variables have different scales.

To standardise the data:

```
Caravan_scale=scale(Caravan [,-86])
```

Check the standard deviation and mean of the variables are all 1 and 0, respectively.

Create the training and test set and apply a kNN model with $k = 1$ to see the test result.

```
# test sample index
index=1:1000
# get training and test set
train.X=Caravan_scale[-index,]
test.X=Caravan_scale[index,]
train.Y=Caravan$Purchase[-index]
test.Y=Caravan$Purchase[index]
# kNN
```

```
library("class")
set.seed (198)
knn.pred=knn(train.X,test.X,train.Y,k=1)
# mean of error rate
mean(test.Y!=knn.pred)
```

```
## [1] 0.118
```

The error rate is just around 12%, which is good. However, considering the imbalance feature of this data: we can get an error rate of 6% if we predict all test observations as No. Thus the error rate is no longer a good measure to assess the quality of the model. In this dataset, we care more about the accuracy of predicting people would like to buy the insurance. If without any prediction, we have to visit every customer to ask if they would like to buy and the success rate is just 6%. However, if we do our research first, then we could only visit customers who are likely to buy the insurance, which saves time and resources.

```
table(knn.pred,test.Y)
```

```
##         test.Y
## knn.pred  No Yes
##      No  873  50
##      Yes  68   9
```

```
9/(68+9)
```

```
## [1] 0.1168831
```

In our kNN model, we correctly predicted 9 customers who would buy the insurance and the accuracy is 11.7%, which is much larger than 6%. This shows the advantage of using kNN. Try different values of k to see the change of accuracy.

User-defined functions

To calculate other performance measures such as sensitivity and specificity, we can define our own function as follows.

```
###########################################################
#### This function calculates two performance measures:
#### specificity and sensitivity
#### Input: pred: predicted labels (factor)
####        truth: true labels (factor)
####        pos: positive level
####        neg: negative level
#### Output: a list containing sensitivity and specificity
###########################################################
performance.measure<-function(pred,truth,pos,neg){
  #### get confusion table
  confusion=table(pred,truth)
  #### get tn, tp, fn, fp
  tn=confusion[neg,neg]
  tp=confusion[pos,pos]
  fn=confusion[neg,pos]
  fp=confusion[pos,neg]
  #### calculate sensitivity
  sens=tp/(tp+fn)
  #### calculate specificity
```

```
  spec=tn/(tn+fp)
  #### put the two values in a list
  measures=list(sensitivity=sens,specificity=spec)
  #### return the list as function output
  return(measures)
}
```

Save this script as `performance-measure.r` in your working directory.

Here is how to use this user-defined function:

```
# kNN
library(caret)
```

```
## Loading required package: ggplot2
```

```
## Loading required package: lattice
```

```
set.seed(47)
knn3_pred=knn3Train(train.X, test.X, train.Y, k = 9, prob=TRUE)
#### calculate sensitivity and specificity
source("performance-measure.R")
pos=levels(test.Y)[2]; neg=levels(test.Y)[1]
measures=performance.measure(as.factor(knn3_pred),test.Y,pos,neg)
measures
```

```
## $sensitivity
## [1] 0.01694915
##
## $specificity
## [1] 1
##
## $accuracy
## [1] 0.942
```

We can compare the above result with the results from functions in the `caret` function.

```
sensitivity(as.factor(knn3_pred),test.Y,pos)
```

```
## [1] 0.01694915
```

```
specificity(as.factor(knn3_pred),test.Y,neg)
```

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

Exercise

Add the calculation of classification accuracy in `performance-measure.r` and return a list containing sensitivity, specificity and classification accuracy.

ROC curves

We can use the `ROCR` package to get ROC curves.

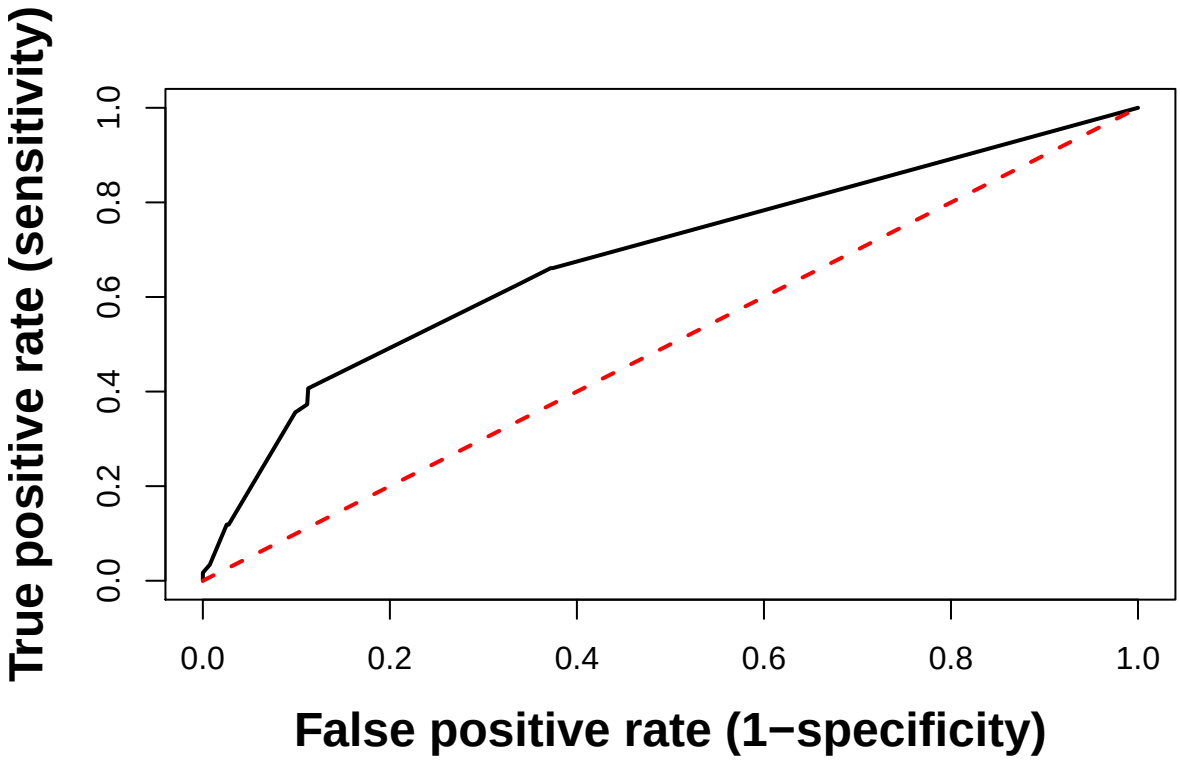
```
library(ROCR)
```

```
## Loading required package: gplots
```

```
##
## Attaching package: 'gplots'
```

```
## The following object is masked from 'package:stats':
##
##     lowess
```

```
#########################################################
#### ROC curve
att=attributes(knn3_pred)$prob
pred.ROCR = prediction(att[,2], (test.Y))
roc.curve = performance(pred.ROCR,"tpr","fpr")
plot(roc.curve,lwd=2,cex.lab=1.5,cex.axis=1.5,  font.lab=2,
     xlab="False positive rate (1-specificity)",
     ylab="True positive rate (sensitivity)")
#### add a line with auc=0.5
x=seq(0,1,0.01); y=x
lines(x,y,lwd =2, col =" red",lty=2)
```



We can also draw the ROC curves with the `pROC` package. Here we show how to do this with the models built in the `caret` package.

```
#### set up train control
fitControl <- trainControl(## 5-fold CV
  method = "repeatedcv",
  number = 5,
  ## repeated five times
  repeats = 5,
  summaryFunction = twoClassSummary,
  classProbs = TRUE)
#### training process
set.seed(5)
```

```
knnFit=train(train.X,train.Y, method = "knn",
             trControl = fitControl,
             metric = "ROC",
             preProcess = c("center","scale"),
             tuneLength=5)
knnFit
```

```
## k-Nearest Neighbors
##
## 4822 samples
##   85 predictor
##    2 classes: 'No', 'Yes'
##
## Pre-processing: centered (85), scaled (85)
## Resampling: Cross-Validated (5 fold, repeated 5 times)
## Summary of sample sizes: 3858, 3858, 3858, 3857, 3857, 3858, ...
## Resampling results across tuning parameters:
##
##   k   ROC        Sens       Spec
##    5  0.5896505  0.9928525  0.020036298
##    7  0.6103408  0.9973973  0.011058681
##    9  0.6189570  0.9990294  0.006218996
##   11  0.6289133  0.9997352  0.001379310
##   13  0.6346906  0.9999558  0.000000000
##
## ROC was used to select the optimal model using the largest value.
## The final value used for the model was k = 13.
```

```
knn.pred <- predict(knnFit,test.X)
confusionMatrix(knn.pred,test.Y)
```

```
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  No Yes
##        No  941  59
##        Yes   0   0
##
##                Accuracy : 0.941
##                  95% CI : (0.9246, 0.9548)
##     No Information Rate : 0.941
##     P-Value [Acc > NIR] : 0.5346
##
##                   Kappa : 0
##
##  Mcnemar's Test P-Value : 4.321e-14
##
##             Sensitivity : 1.000
##             Specificity : 0.000
##          Pos Pred Value : 0.941
##          Neg Pred Value :   NaN
##              Prevalence : 0.941
##          Detection Rate : 0.941
##    Detection Prevalence : 1.000
##       Balanced Accuracy : 0.500
```

```
##
##        'Positive' Class : No
##
```

Now we can draw an ROC curve to check the classification performance on test data.

```
knn.probs <- predict(knnFit,test.X,type="prob")
head(knn.probs)
```

```
##          No        Yes
## 1 0.9230769 0.07692308
## 2 1.0000000 0.00000000
## 3 1.0000000 0.00000000
## 4 1.0000000 0.00000000
## 5 1.0000000 0.00000000
## 6 1.0000000 0.00000000
```

```
library(pROC)
```

```
## Type 'citation("pROC")' for a citation.
```

```
##
## Attaching package: 'pROC'
```

```
## The following objects are masked from 'package:stats':
##
##     cov, smooth, var
```

```
knn.ROC <- roc(predictor=knn.probs$No,
               response=test.Y)
```

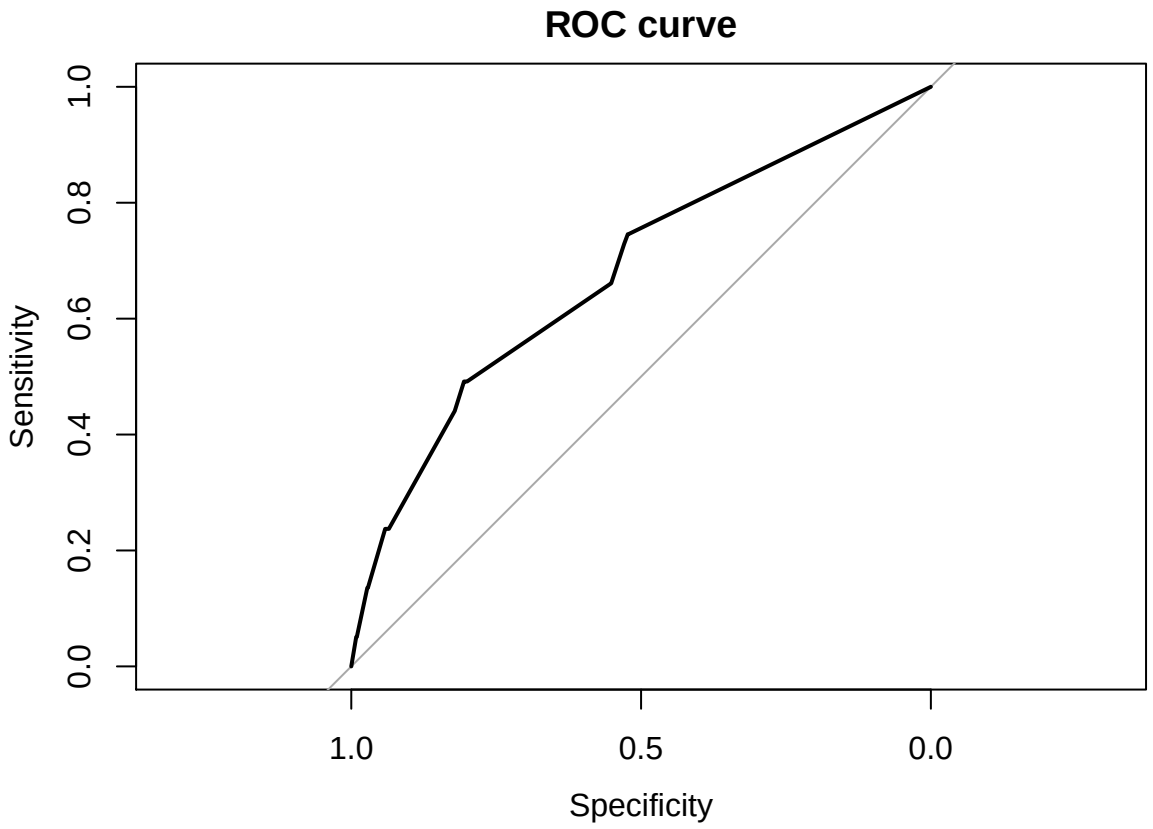
```
## Setting levels: control = No, case = Yes
```

```
## Setting direction: controls > cases
```

```
knn.ROC$auc
```

```
## Area under the curve: 0.6775
```

```
plot(knn.ROC,main="ROC curve")
```



LOOCV to choose the value of k using the training data

Suppose we want to determine which k is the best for classification from five values $1, 3, 5, 7, 9$. Then we need to do LOOCV five times on the training data: calculate the mean accuracy of LOOCV using each k value and then choose the k value with the largest accuracy. We then use this k value in the kNN model to classify the test samples.

To effectively do this, we can use the `for` loop. Here is an example to calculate $1 + 2 + \ldots + 10$:

```
Sum=0
for(ii in 1:10){
  Sum=Sum+ii
}
Sum
```

```
## [1] 55
```

We first initialise `Sum` variable as 0, to store the sum value in each loop. `ii` is the index for the loops and is from 1 to 10. The values of `ii` are indicated in `( )` using `in`. The steps of each loop is written in `{ }`. `ii` starts from 1, so the first loop calculates $0 + 1$ and give this value to `Sum`. In the next loop, `ii` becomes 2, and `Sum` becomes $1 + 2$ (think about why!). `ii` ends in 10, so the last loop is $45 + 10$. After `ii` reaches 10, the loop ends and `Sum` is 55.

`kNN.cv` in the `class` package provides the leave-one-out cross-validation (LOOCV) evaluation. Type the following code to see the input options and the outputs.

```
?knn.cv
```

Below are the codes for our task:

```r
# get indexes for training data
n=25 # the number of training data in each class
NN=dim(iris3)[1]# the total number of observations in each class
set.seed(983)
index_s=sample(1:NN,n)
index_c=sample(1:NN,n)
index_v=sample(1:NN,n)
# get training and test set
train_rand = rbind(iris3[index_s,,1], iris3[index_c,,2], iris3[index_v,,3])
test_rand = rbind(iris3[-index_s,,1], iris3[-index_c,,2], iris3[-index_v,,3])
# get class factor for training data
train_label= factor(c(rep("s",n), rep("c",n), rep("v",n)))
# get class factor for test data
test_label_true=factor(c(rep("s",NN-n), rep("c",NN-n), rep("v",NN-n)))
# evaluate k=1,3,5,7,9 on the training data using LOOCV
kk=c(1,3,5,7,9)
# initialise acc to store the mean accuracy of LOOCV for
# five $k$
acc=vector("numeric",length=length(kk))
set.seed(649)
for(ii in 1:length(kk)){
  knn_pred=knn.cv(train=train_rand, cl=train_label, k = kk[ii])
  acc[ii]=mean(knn_pred==train_label)
}
# get the first k that has the largest accuracy
# here we can also randomly choose any k that has the largest
# accuracy if we have several largest values
acc
```

```
## [1] 0.9466667 0.9733333 0.9733333 0.9733333 0.9466667
```

```r
kk_LOOCV=kk[which.max(acc)]
# use the k tuned by the training set to classify the test set
set.seed(275)
knn_pred_test=knn(train=train_rand,test=test_rand,cl=train_label, k=kk_LOOCV, prob=FALSE)
acc_test=mean(knn_pred_test==test_label_true)
acc_test
```

```
## [1] 0.96
```

Play with the codes to make sure you understand them!

LOOCV to evaluate the performance of kNN with a specific k value

Use LOOCV to evaluate the classification performance of a kNN model with $k = 3$ on the iris data:

```r
# LOOCV on the whole dataset
train=rbind(iris3[,,1], iris3[,,2], iris3[,,3])
cl=factor(c(rep("s",50), rep("c",50), rep("v",50)))
# evaluate kNN model with k=3
kk=3
set.seed(382)
knn.pred=knn.cv(train, cl, k = kk)
acc=mean(knn.pred==cl)
```

```
acc
```

```
## [1] 0.96
```