Chapter 14
Model Performance Assessment

In previous chapters, we used prediction accuracy to evaluate classification models. However, having accurate predictions in one dataset does not necessarily imply that the model is perfect or that it will reproduce when tested on external data. We need additional metrics to evaluate the model performance and to make sure it is robust, reproducible, reliable, and unbiased.

In this chapter, we will discuss (1) various evaluation strategies for prediction, clustering, classification, regression, and decision trees; (2) visualization of ROC curves and performance tradeoffs; and (3) estimation of future performance, internal statistical cross-validation and bootstrap sampling.

14.1 Measuring the Performance of Classification Methods

As mentioned previously, classification model performances could not be evaluated by prediction accuracy alone. We make different classification models for different purposes. For example, in newborns screening for genetic defects we want the model to have as few true negatives as possible. We don't want to classify anyone as "no defect" when they actually have a defect gene, since early treatment might alter the destiny of this newborn.

We can use the following three types of data to evaluate the performance of a classifier model.

- Actual class values (for supervised classification).
- Predicted class values.
- Estimated probability of the prediction.

We are familiar with the first two cases. The last type of validation relies on the predict(model, test_data) function that we have talked about in previous classification and prediction chapters (Chaps. 7, 8, and 9). Let's revisit the model and test data we discussed in Chap. 8; the Inpatient Head and Neck Cancer Medication data.

We will demonstrate prediction probability estimation using this case-study CaseStudy14_HeadNeck_Cancer_Medication.csv

```
pred_raw<-predict(hn_classifier, hn_test, type="raw")
head(pred_raw)

##      early_stage later_stage
## [1,]   0.9381891  0.06181090
## [2,]   0.9381891  0.06181090
## [3,]   0.8715581  0.12844188
## [4,]   0.9382140  0.06178601
## [5,]   0.9675997  0.03240026
## [6,]   0.9675997  0.03240026
```

The above output includes the prediction probabilities for the first 6 rows of the data. This example is based on the Naive Bayes classifier, however the same approach works for any other machine-learning classification or prediction technique.

In addition, we can report the predicted probability with the outputs of the Naive Bayesian decision-support system (`hn_classifier <- naiveBayes (hn_train, hn_med_train$stage)`):

```
(hn_classifier <- naiveBayes(hn_train, hn_med_train$stage)):
```

```
pred_nb<-predict(hn_classifier, hn_test)
head(pred_nb)

## [1] early_stage early_stage early_stage early_stage early_stage
early_stage
## Levels: early_stage later_stage
```

The general `predict()` method automatically subclasses to the specific `predict.naiveBayes(object, newdata, type = c("class", "raw"), threshold = 0.001, ...)` call where `type = "raw"` and `type = "class"` specify the output as the conditional a-posterior probabilities for each class or the class with maximal probability, respectively. Back in Chap. 9, we discussed the `C5.0` and the `randomForest` classifiers used to predict the chronic disease score in a (different) Quality of Life Study.

Below are the (probability) results of the `C5.0` classification prediction:

```
pred_prob<-predict(qol_model, qol_test, type="prob")
head(pred_prob)

##    minor_disease severe_disease
## 10     0.1979698      0.8020302
## 12     0.1979698      0.8020302
## 26     0.3468705      0.6531295
## 37     0.1263975      0.8736025
## 41     0.7290209      0.2709791
## 43     0.3163673      0.6836327
```

These can be contrasted against the C5.0 classification label results:

```
pred_tree<-predict(qol_model, qol_test)
head(pred_tree)
## [1] severe_disease severe_disease severe_disease severe_disease
## [5] minor_disease  severe_disease
## Levels: minor_disease severe_disease
```

The same complementary types of outputs can be reported for most machine-learning classification and prediction approaches

14.2 Evaluation Strategies

In Chap. 7, we saw an attempt to categorize the supervised classification and unsupervised clustering methods. Similarly, Table 14.1 summarizes the basic types of evaluation and validation strategies for different forecasting, prediction, ensembling, and clustering techniques. (Internal) Statistical Cross Validation or external validation should always be applied to ensure reliability and reproducibility of the results. The SciKit clustering performance evaluation and Classification metrics page provide details about many alternative techniques and metrics for performance evaluation of clustering and classification methods.

14.2.1 Binary Outcomes

More details about binary test assessment are available on the Scientific Methods for Health Sciences (SMHS) EBook site. Table 14.2 summarizes the key measures

Table 14.1 Categories of clustering validation and classification evaluation strategies

Inference	Outcome	Evaluation metrics	Example R functions
Classification & Prediction	Binary	Accuracy, Sensitivity, Specificity, PPV/Precision, NPV/Recall, LOR	`caret::confusionMatrix`, `gmodels::CrossTable`, `cluster::silhouette`
Classification & Prediction	Categorical	Accuracy, Sensitivity/Specificity, PPV, NPV, LOR, Silhouette Coefficient	`caret::confusionMatrix`, `gmodels::CrossTable`, `cluster::silhouette`
Regression Modeling	Real Quantitative	correlation coefficient, R^2, RMSE, Mutual Information, Homogeneity and Completeness Scores	`cor`, `metrics::mse`

Table 14.2 Evaluation of binary (dichotomous) statistical tests, classification methods, or forecasting predictions

		Actual condition(or real class label)		Test interpretation
		Absent (H_0 is true)	Present (H_1 is true)	
Test Result (Prediction or Classification Label)	**Negative (fail to reject H_0)**	TN Condition absent + Negative result = True (accurate) Negative	FNCondition present + Negative result = False (invalid) Negative Type II error (proportional to β)	$NPV = \frac{TN}{TN+FN}$
	Positive(reject H_0)	FP Condition absent + Positive result = False Positive Type I error (α)	TP Condition Present + Positive result = True Positive	$PPV = Precision = \frac{TP}{TP+FP}$
Test Interpretation	$\boldsymbol{Power = 1 - \beta = 1 - \frac{FN}{FN+TP}}$	$Specificity = \frac{TN}{TN+FP}$	$Power= Sensitivity= \frac{TP}{TP+FN}$	$LOR = \ln\left(\frac{S1/F1}{S2/F2}\right) = \ln\left(\frac{S1\times F2}{S2\times F1}\right)$, S = success, F = failure for 2 binary variables, 1 and 2

Table 14.3 Cross-table

	Predict_T	predict_F
TRUE	TP	TN
FALSE	FP	FN

commonly used to evaluate the performance of binary tests, classifiers, or predictions.

See also SMHS EBook; Power, Sensitivity and Specificity section.

14.2.2 Confusion Matrices

We talked about this confusion matrices in Chap. 9. For binary classes, these will be 2 × 2 matrices. Each of the cells has specific meaning, see the 2 × 2 Table 14.2 where

- **True Positive**(TP): Number of observations that correctly classified as "yes" or "success"
- **True Negative**(TN): Number of observations that correctly classified as "no" or "failure"
- **False Positive**(FP): Number of observations that incorrectly classified as "yes" or "success"
- **False Negative**(FN): Number of observations that incorrectly classified as "no" or "failure"

Using Confusion Matrices to Measure Performance

The way we calculate accuracy using these four cells is summarized by the following formula:

$$accuracy = \frac{TP + TN}{TP + TN + FP + FN} = \frac{TP + TN}{\text{Total number of observations}} \quad .$$

On the other hand, the error rate, or proportion of incorrectly classified observations, is calculated using:

$$errorrate = \frac{FP + FN}{TP + TN + FP + FN} == \frac{FP + FN}{\text{Total number of observations}}$$
$$= 1 - accuracy.$$

If we look at the numerator and denominator carefully, we can see that the error rate and accuracy add up to 1. Therefore, 95% accuracy implies a 5% error rate.

In R, we have multiple ways to obtain confusion matrices. The simplest way would be to use `table()`. For example, in Chap. 8, to report a plain 2×2 table we used:

```
hn_test_pred<-predict(hn_classifier, hn_test)
table(hn_test_pred, hn_med_test$stage)

##
## hn_test_pred  early_stage later_stage
##   early_stage          69          23
##   later_stage           8           0
```

Then why did we use `CrossTable()` function back in Chapter 8? Because it reports additional useful information about the model performance.

```
library(gmodels)
CrossTable(hn_test_pred, hn_med_test$stage)

##    Cell Contents
## |-------------------------|
## |                       N |
## | Chi-square contribution |
## |           N / Row Total |
## |           N / Col Total |
## |         N / Table Total |
## |-------------------------|
## Total Observations in Table:  100
##              | hn_med_test$stage
## hn_test_pred | early_stage | later_stage |   Row Total |
## -------------|-------------|-------------|-------------|
##  early_stage |          69 |          23 |          92 |
##              |       0.048 |       0.160 |             |
##              |        0.750 |        0.250 |        0.920 |
##              |        0.896 |        1.000 |              |
##              |        0.690 |        0.230 |              |
## -------------|-------------|-------------|-------------|
##  later_stage |            8 |            0 |            8 |
##              |        0.550 |        1.840 |              |
##              |        1.000 |        0.000 |        0.080 |
##              |        0.104 |        0.000 |              |
##              |        0.080 |        0.000 |              |
## -------------|-------------|-------------|-------------|
## Column Total |           77 |           23 |          100 |
##              |        0.770 |        0.230 |              |
## -------------|-------------|-------------|-------------|
```

With both tables, we can calculate accuracy and error rate by hand.

```
accuracy<-(69+0)/100
accuracy

## [1] 0.69

error_rate<-(23+8)/100
error_rate

## [1] 0.31

1-accuracy

## [1] 0.31
```

For matrices larger than 2 × 2, all diagonal elements are observations that have been correctly classified and off-diagonal elements are those that have been incorrectly classified.

14.2.3 Other Measures of Performance Beyond Accuracy

So far, we discussed two performance methods - table and cross-table. A third function is `confusionMatrix()` which provides the easiest way to report model performance. Notice that the first argument is an *actual vector of the labels*, i.e., *Test_Y*, and the second argument, of the same length, represents the *vector of predicted labels*.

This example was presented as the first case-study in Chap. 9.

```r
library(caret)

qol_pred<-predict(qol_model, qol_test)
confusionMatrix(table(qol_pred, qol_test$cd), positive="severe_disease")

## Confusion Matrix and Statistics
##
##
## qol_pred         minor_disease severe_disease
##   minor_disease            149             89
##   severe_disease            74            131
##
##                Accuracy : 0.6321
##                  95% CI : (0.5853, 0.6771)
##     No Information Rate : 0.5034
##     P-Value [Acc > NIR] : 3.317e-08
##
##                   Kappa : 0.2637
##  Mcnemar's Test P-Value : 0.2728
##
##             Sensitivity : 0.5955
##             Specificity : 0.6682
##          Pos Pred Value : 0.6390
##          Neg Pred Value : 0.6261
##              Prevalence : 0.4966
##          Detection Rate : 0.2957
##    Detection Prevalence : 0.4628
##       Balanced Accuracy : 0.6318
##
##        'Positive' Class : severe_disease
```

14.2.4 The Kappa (κ) Statistic

The Kappa statistic was originally developed to measure the reliability between two human raters. It can be harnessed in machine-learning applications to compare the accuracy of a classifier, where `one rater` represents the ground truth (for labeled data, these are the actual values of each instance) and the `second rater` represents the results of the automated machine-learning classifier. The order of listing the **raters** is irrelevant.

Kappa statistic measures the **possibility of a correct prediction by chance alone** and answers the question of `How much better is the agreement (between the ground truth and the machine-learning prediction) than would be expected by chance alone?` Its value is between 0 and 1. When $\kappa = 1$, we have a perfect agreement between a **computed** prediction (typically the result of a model-based or model-free technique forecasting an outcome of interest) and an **expected** prediction (typically random, by chance prediction). A common interpretation of the Kappa statistics includes:

- Poor agreement: less than 0.20
- Fair agreement: 0.20–0.40
- Moderate agreement: 0.40–0.60
- Good agreement: 0.60–0.80
- Very good agreement: 0.80–1

In the above `confusionMatrix` output, we have a fair agreement. For different problems, we may have different interpretations of Kappa statistics. To understand the Kappa statistic better, let's look at its definition:

$$kappa = \frac{P(a) - P(e)}{1 - P(e)} \quad .$$

`P(a)` and `P(e)` simply denote probability of **actual** and **expected** agreement between the classifier and true values.

```
table(qol_pred, qol_test$cd)

## 
## qol_pred          minor_disease severe_disease
##   minor_disease             149             89
##   severe_disease             74            131
```

According to above table, actual agreement is the accuracy:

```
p_a<-(149+131)/(149+89+74+131)
p_a

## [1] 0.6320542
```

The manually and automatically computed accuracies coincide (0.6321). It may be trickier to obtain the expected agreement. Probability rules tell us that the probability of the union of two disjoint events equals to the sum of the individual (marginal) probabilities for these two events. Thus, we have:

$$P(expect\ agreement\ for\ minor\_disease) = P(actual\ type\ is\ minor\_disease) \\ + P(predicted\ type\ is\ minor\_disease)$$

Similarly:

$$P(expect\ agreement\ for\ severe\_disease) = P(actual\ type\ is\ severe\_disease) \\ + P(predicted\ type\ is\ severe\_disease).$$

In our case:

```
p_e_minor <- (149+74)/(149+89+74+131))*((149+89)/(149+89+74+131)
p_e_severe <- ((131+74)/(149+89+74+131)) * ((89+131)/(149+89+74+131))
p_e<-p_e_minor+p_e_severe
p_e

## [1] 0.5002522
```

Plugging in `p_a` and `p_e` into the formula we get:

```
kappa<-(p_a-p_e)/(1-p_e)
kappa

## [1] 0.26
```

We get a similar value as the `confusionTable()` output. A more straightforward way of getting the Kappa statistics is by using `Kappa()` function in the `vcd` package.

```
#install.packages(vcd)
library(vcd)

## Loading required package: grid

Kappa(table(qol_pred, qol_test$cd))

##             value     ASE     z  Pr(>|z|)
## Unweighted 0.2637 0.04573 5.767 8.071e-09
## Weighted   0.2637 0.04573 5.767 8.071e-09
```

The combination of `Kappa()` and `table` function yields a 2×4 matrix. The *Kappa statistic* is under the unweighted value.

Generally speaking, predicting a severe disease outcome is a more critical problem than predicting a mild disease state. Thus, weighted Kappa is also useful. We give the severe disease a higher weight. The Kappa test result is not acceptable since the classifier may make too many mistakes for the severe disease cases. The Kappa value is only -0.0714. Notice that the range of Kappa is not [0,1] for the weighted Kappa.

```
Kappa(table(qol_pred, qol_test$cd),weights = matrix(c(1,10,1,10),nrow=2))

##              value     ASE     z  Pr(>|z|)
## Unweighted 0.26374 0.04573 5.767 8.071e-09
## Weighted   0.06818 0.04009 1.701 8.898e-02
```

When the predicted value is the first argument, the row and column names represent the **true labels** and the **predicted labels**, respectively.

```
table(qol_pred, qol_test$cd)

##
## qol_pred         minor_disease severe_disease
##   minor_disease            149             89
##   severe_disease            74            131
```

Summary of the Kappa Score for Calculating Prediction Accuracy

Kappa compares an **Observed classification accuracy** (output of our ML classifier) with an **Expected classification accuracy** (corresponding to random chance classification). It may be used to evaluate single classifiers and/or to compare among a set of different classifiers. It takes into account random chance (agreement with a random classifier). That makes **Kappa** more meaningful than simply using **accuracy** as a metric. For instance, the interpretation of an `Observed Accuracy of 80%` is **relative** to the `Expected Accuracy`. `Observed Accuracy of 80%` is more impactful for an `Expected Accuracy of 50%` compared to `Expected Accuracy of 75%`.

14.2.5 *Computation of Observed Accuracy and Expected Accuracy*

Consider the following example of a `classifier` generating the following `confusion matrix`. Columns represent the **true labels** and rows represent the **classifier-derived labels** for this binary prediction example (Table 14.4).

In this example, there is a total of 150 observations (50 + 35 + 25 + 40). In reality, 75 are labeled as **True** (50 + 25) and another 75 are labeled as **False** (35 + 40). The classifier labeled 85 as **True** (50 + 35) and the other 65 as **False** (25 + 40).

- Observed Accuracy (OA) is the `proportion of instances` that were classified correctly throughout the entire confusion matrix:

$$OA = \frac{50 + 40}{150} = 0.6.$$

- Expected Accuracy (EA) is the accuracy that any random classifier would be expected to achieve based on the given confusion matrix. EA is the `proportion of instances` of each class (**True** and **False**), along with the number of instances that the automated classifier agreed with the ground truth label. The EA is calculated by multiplying the marginal frequencies of **True** for the true-state and the machine classified instances, and dividing by the total number of instances. The marginal frequency of **True** for the **true-state** is 75 (50 + 25)

Table 14.4 A simulated confusion matrix.

Class	True	False	Total
True	50	35	85
False	25	40	65
Total	75	75	150

and for the corresponding ML classifier is 85 (50 + 35). Then, the expected accuracy for the **True** outcome is:

$$EA(True) = \frac{75 \times 85}{150} = 42.5.$$

We similarly compute the $EA(False)$ for the second, **False**, outcome, by using the marginal frequencies for the true-state ($(False|\, true\ state) = 75 = 50 + 25$) and the ML classifier ($(False|\, classifier) = 65(40 + 25)$). Then, the expected accuracy for the **True** outcome is:

$$EA(False) = \frac{75 \times 65}{150} = 32.5.$$

Finally, the $EA = \frac{EA(True)+EA(False)}{150}$

$$ExpectedAccuracy(EA) = \frac{42.5 + 32.5}{150} = 0.5.$$

Note that $EA = 0.5$ whenever the `true-state` binary classification is balanced (in reality, the frequencies of **True** and **False** are equal, in our case 75).

The calculation of the **kappa statistic** relies on $OA = 0.6$ and $EA = 0.5$:

$$(Kappa)\ \kappa = \frac{OA - EA}{1 - EA} = \frac{0.6 - 0.5}{1 - 0.5} = 0.2.$$

14.2.6 Sensitivity and Specificity

If we take a closer look at the `confusionMatrix()` output, we find there are two important statistics "sensitivity" and "specificity".

Sensitivity, or true positive rate, measures the proportion of "success" observations that are correctly classified.

$$sensitivity = \frac{TP}{TP + FN}.$$

Notice $TP + FN$ are the total number of true "success" observations.

On the other hand, specificity, or true negative rate, measures the proportion of "failure" observations that are correctly classified.

$$specificity = \frac{TN}{TN + FP}.$$

Accordingly, $TN + FP$ are the total number of true "failure" observations.

Using the `table()` output above and using "severe_disease" as “success”, we can compute these two measures directly.

```
sens<-131/(131+89)
sens

## [1] 0.5954545

spec<-149/(149+74)
spec

## [1] 0.6681614
```

Another R package, `caret`, also provides functions to calculate sensitivity and specificity.

```
library(caret)
sensitivity(qol_pred, qol_test$cd, positive="severe_disease")

## [1] 0.5954545
```

Sensitivity and specificity both range from 0 to 1. For either measure, a value of 1 implies that the positive and negative predictions are very accurate. However, simultaneously high sensitivity and specificity may not be attainable in real world situations. There is a tradeoff between sensitivity and specificity. To compromise, some studies loosen the demands on one and focus on achieving high values on the other.

14.2.7 Precision and Recall

Very similar to sensitivity, *precision* measures the proportion of true “success” observations among predicted “success” observations.

$$precision = \frac{TP}{TP + FP}.$$

Recall is the proportion of true “positives” among all “true positive” conditions. A model with high recall captures most “interesting” cases.

$$recall = \frac{TP}{TP + FN}.$$

Again, let’s calculate these by hand for the QoL data:

```
prec<-131/(131+74)
prec

## [1] 0.6390244

recall<-131/(131+89)
recall

## [1] 0.5954545
```

Another way to obtain *precision* would be `posPredValue()` under the `caret` package. Remember to specify which one is the “success” class.

```
posPredValue(qol_pred, qol_test$cd, positive="severe_disease")

## [1] 0.6390244
```

From the definitions of **precision** and **recall**, we can derive the type 1 error and type 2 errors as follow:

$$error_1 = 1 - Precision = \frac{FP}{TP+FP}, \text{ and}$$
$$error_2 = 1 - Recall = \frac{FN}{TP+FN}.$$

Thus, we can compute the type 1 error (0.36) and type 2 error (0.40).

```
error1<-74/(131+74)
error2<-89/(131+89)
error1; error2

## [1] 0.3609756

## [1] 0.4045455
```

14.2.8 The F-Measure

The F-measure or F1-score combines precision and recall using the harmonic mean assuming equal weights. High F-score means high precision and high recall. This is a convenient way of measuring model performances and comparing models.

$$F - measure = \frac{2 \times precision \times recall}{recall + precision} = \frac{2 \times TP}{2 \times TP + FP + FN}.$$

Let’s calculate the F1-score by hand using the confusion matrix derived from the Quality of Life prediction:

```
F1<-(2*prec*recall)/(prec+recall); F1

## [1] 0.6164706
```

The direct calculations of the F1-statistics can be obtained using `caret`:

```
precision <- posPredValue(qol_pred, qol_test$cd, positive="severe_disease")
recall <- sensitivity(qol_pred, qol_test$cd, positive="severe_disease")
F1 <- (2 * precision * recall) / (precision + recall); F1

## [1] 0.6164706
```

14.3 Visualizing Performance Tradeoffs (ROC Curve)

Another choice for evaluating classifiers performance is by using graphs rather than quantitative statistics. Graphs are usually more comprehensive than single statistics.

In R there is a package providing user-friendly functions for visualizing model performance. Details can be found on the ROCR website.

Here, we evaluate the model performance for the Quality of Life case study, see Chap. 9.

```
#install.packages("ROCR")
library(ROCR)

pred<-ROCR::prediction(predictions=pred_prob[, 2], labels=qol_test$cd)
# avoid naming collision (ROCR::prediction), as
# there is another prediction function in neuralnet package.
```

`pred_prob[, 2]` is the probability of classifying each observation as "severe_disease". The above code saved all the model prediction information into object `pred`.

The ROC (Receiver Operating Characteristic) curves are often used to examine the tradeoff between detecting true positives and avoiding the false positives (Fig. 14.1).

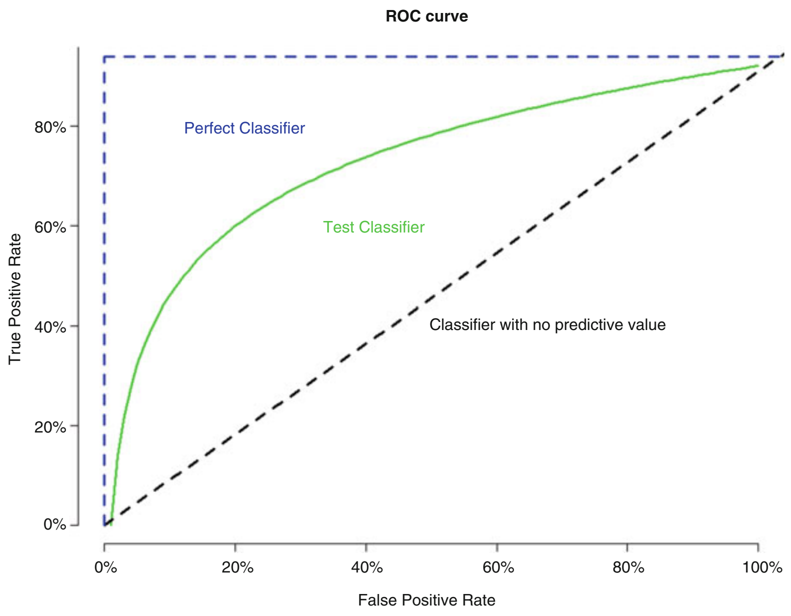

Fig. 14.1 Schematic of quantifying the efficacy of a classification method using the area under the ROC curve

```r
curve(log(x), from=0, to=100, xlab="False Positive Rate", ylab="True Positiv
e Rate", main="ROC curve", col="green", lwd=3, axes=F)
Axis(side=1, at=c(0, 20, 40, 60, 80, 100), labels = c("0%", "20%", "40%", "6
0%", "80%", "100%"))
Axis(side=2, at=0:5, labels = c("0%", "20%", "40%", "60%", "80%", "100%"))
segments(0, 0, 110, 5, lty=2, lwd=3)
segments(0, 0, 0, 4.7, lty=2, lwd=3, col="blue")
segments(0, 4.7, 107, 4.7, lty=2, lwd=3, col="blue")
text(20, 4, col="blue", labels = "Perfect Classifier")
text(40, 3, col="green", labels = "Test Classifier")
text(70, 2, col="black", labels= "Classifier with no predictive value")
```

The **blue line** in the above graph represents the perfect classifier where we have 0% false positive and 100% true positive. The middle **green line** is the test classifier. Most of our classifiers trained by real data will look like this. The black diagonal line illustrates a classifier with no predictive value predicts. We can see that it has the same true positive rate and false positive rate. Thus, it cannot distinguish between the two.

In terms of identifying positive value, we want our ROC curve to be as close to the perfect line as possible. Thus, we measure the area under the ROC curve (abbreviated as AUC) to show how close our curve is to the perfect classifier. To do this, we have to change the scale of the graph above. Mapping 100% to 1, we have a 1×1 square. The area under the perfect classifier would be one, and area under classifier with no predictive value would be 0.5. Then, 1 and 0.5 will be the upper and lower limits for our model ROC curve. We have the following scoring system (numbers indicate area under the curve) for predictive model ROC curves:

- Outstanding: 0.9–1.0
- Excellent/good: 0.8–0.9
- Acceptable/fair: 0.7–0.8
- Poor: 0.6–0.7
- No discrimination: 0.5–0.6.

Note that this rating system is somewhat subjective. Let’s use the `ROCR` package to draw a ROC curve.

```r
roc<-performance(pred, measure="tpr", x.measure="fpr")
```

By specifying `"tpr"`(True positive rate) and `"fpr"`(False positive rate) we made a “performance” object (Fig. 14.2).

```r
plot(roc, main="ROC curve for Quality of Life model", col="blue", lwd=3)
segments(0, 0, 1, 1, lty=2)
```

The segments command draws the dotted line representing the classifier with no predictive value.

To measure this quantitatively, we need to create a new performance object with `measure = "auc"` or area under the curve.

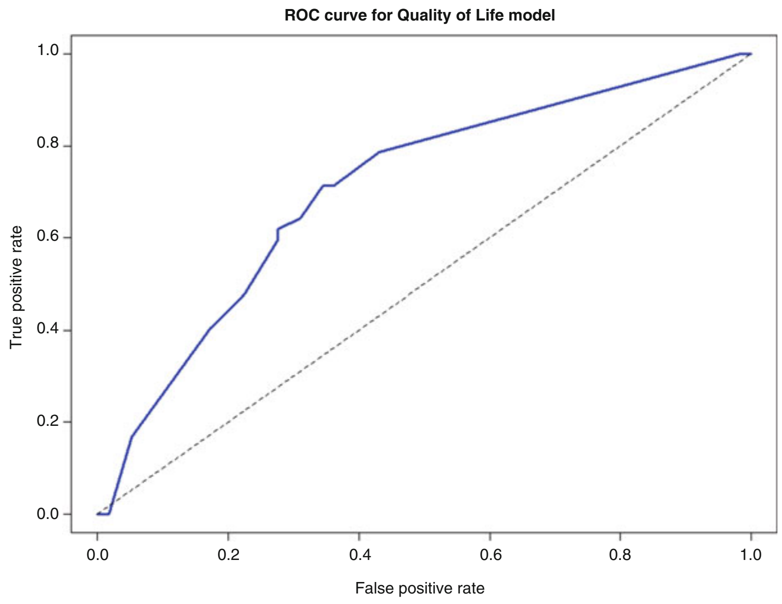

Fig. 14.2 ROC curve of the prediction of disease severity using the quality of life (QoL) data

```r
roc_auc<-performance(pred, measure="auc")
```

Now the `roc_auc` is stored as a S4 object. This is quite different than data frame and matrices. First, we can use `str()` function to see its structure.

```r
str(roc_auc)

## Formal class 'performance' [package "ROCR"] with 6 slots
##   ..@ x.name      : chr "None"
##   ..@ y.name      : chr "Area under the ROC curve"
##   ..@ alpha.name  : chr "none"
##   ..@ x.values    : list()
##   ..@ y.values    :List of 1
##   .. ..$ : num 0.65
##   ..@ alpha.values: list()
```

The ROC object has six members. The AUC value is stored in `y.values`. To extract that we use the @ symbol according to the output of the `str()` function.

```r
roc_auc@y.values

## [[1]]
## [1] 0.6496739
```

Thus, the obtained $AUC = 0.65$, which suggests a fair classifier, according to the above scoring schema.

14.4 Estimating Future Performance (Internal Statistical Validation)

The evaluation methods we have talked about are all measuring re-substitution error. That is, building the model on training data and measuring the model error on separate testing data. This is one way of dealing with unseen data. First, let's introduce the basic ideas, and more details will be presented in Chap. 21.

14.4.1 The Holdout Method

The idea is to partition the entire dataset into two separate datasets, using one of them to create the model and the other to test the model performances. In practice, we usually use a fraction (e.g., 50%, or $\frac{2}{3}$) of our data for training the model, and reserve the rest (e.g., 50%, or $\frac{1}{3}$) for testing. Note that the testing data may also be further split into proportions for internal repeated (e.g., cross-validation) testing and final external (independent) testing.

The partition has to be randomized. In R, the best way of doing this is to create a parameter that randomly draws numbers and use this parameter to extract random rows from the original dataset. In Chap. 11, we used this method to partition the *Google Trends* data.

```
sub<-sample(nrow(google_norm), floor(nrow(google_norm)*0.75))
google_train<-google_norm[sub, ]
google_test<-google_norm[-sub, ]
```

Another way of partitioning is by using `createDataPartition()` under the `caret` package. Instead of using the entire original dataset, we can use the outcome variable, `google_norm$RealEstate`, or any of the independent variables.

```
sub<-createDataPartition(google_norm$RealEstate, p=0.75, list = F)
google_train<-google_norm[sub, ]
google_test<-google_norm[-sub, ]
```

To make sure that the model can be applied to future datasets, we can partition the original dataset into three separate subsets. In this way, we have two subsets for testing. The additional validation dataset can alleviate the probability that we have a good model due to chance (non-representative subsets). A common split among training, test, and validation subsets would be 50%, 25%, and 25% respectively.

```
sub<-sample(nrow(google_norm), floor(nrow(google_norm)*0.50))
google_train<-google_norm[sub, ]
google_test<-google_norm[-sub, ]
sub1<-sample(nrow(google_test), floor(nrow(google_test)*0.5))
google_test1<-google_test[sub1, ]
google_test2<-google_test[-sub1, ]
nrow(google_norm)

## [1] 731

nrow(google_train)

## [1] 365

nrow(google_test1)

## [1] 183

nrow(google_test2)

## [1] 183
```

However, when we only have a very small dataset, it's difficult to split off too much data as this reduces the sample further. There are the following two options for evaluation of model performance using (independent) unseen data: cross-validation and holdout methods. These are implemented in the `caret` package.

14.4.2 Cross-Validation

For complete details see DSPA Cross-Validation (Chap. 21). Below, we describe the fundamentals of cross-validation as an internal statistical validation technique.

This technique is known as *k-fold cross-validation* or *k-fold CV*, which is a standard for estimating model performance. K-fold CV randomly divides the original data into k separate random subsets called folds.

A common practice is to use `k = 10` or 10-fold CV to split the data into 10 different subsets. Each time using one of the subsets to be the test set and the rest to build the model. `createFolds()` under `caret` package will help us to do so. `seet.seed()` insures the folds created are the same if you run the code line twice. `1234` is just a random number. You can use any number for `set.seed()`. We use the normalized Google Trend dataset in this section.

```
library("caret")
set.seed(1234)
folds<-createFolds(google_norm$RealEstate, k=10)
str(folds)
## List of 10
##  $ Fold01: int [1:73] 5 9 11 12 18 19 28 29 54 65 ...
##  $ Fold02: int [1:73] 14 24 35 49 52 61 63 76 99 115 ...
##  $ Fold03: int [1:73] 1 8 41 45 51 74 78 92 100 104 ...
##  $ Fold04: int [1:73] 30 32 37 40 43 57 59 64 70 96 ...
##  $ Fold05: int [1:73] 13 16 25 53 56 68 77 81 93 95 ...
##  $ Fold06: int [1:73] 4 6 15 20 36 69 71 73 79 89 ...
##  $ Fold07: int [1:73] 34 42 44 84 90 98 102 110 112 117 ...
##  $ Fold08: int [1:73] 2 3 48 62 82 85 86 87 88 91 ...
##  $ Fold09: int [1:74] 10 21 23 27 33 39 46 55 58 75 ...
##  $ Fold10: int [1:73] 7 17 22 26 31 38 47 50 60 66 ...
```

Another way to cross-validate is to use `cv_partition()` in package `sparsediscrim`.

```
# install.packages("sparsediscrim")
require(sparsediscrim)
folds2 = cv_partition(1:nrow(google_norm), num_folds=10)
```

And the structure of folds may be reported by:

```
str(folds2)

## List of 10
##  $ Fold1 :List of 2
##   ..$ training: int [1:657] 4 5 6 8 9 10 11 12 16 17 ...
##   ..$ test    : int [1:74] 287 3 596 1 722 351 623 257 568 414 ...
##  $ Fold2 :List of 2
##   ..$ training: int [1:658] 1 2 3 5 6 7 8 9 10 11 ...
##   ..$ test    : int [1:73] 611 416 52 203 359 195 452 258 614 121 ...
##  $ Fold3 :List of 2
##   ..$ training: int [1:658] 1 2 3 4 5 7 8 9 10 11 ...
##   ..$ test    : int [1:73] 182 202 443 152 486 229 88 158 178 293 ...
##  $ Fold4 :List of 2
##   ..$ training: int [1:658] 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ test    : int [1:73] 646 439 362 481 183 387 252 520 438 586 ...
##  $ Fold5 :List of 2
##   ..$ training: int [1:658] 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ test    : int [1:73] 503 665 47 603 348 125 719 11 461 361 ...
##  $ Fold6 :List of 2
##   ..$ training: int [1:658] 1 2 3 4 6 7 9 10 11 12 ...
##   ..$ test    : int [1:73] 666 411 159 21 565 298 537 262 131 600 ...
##  $ Fold7 :List of 2
##   ..$ training: int [1:658] 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ test    : int [1:73] 269 572 410 488 124 447 313 255 360 473 ...
##  $ Fold8 :List of 2
##   ..$ training: int [1:658] 1 2 3 4 5 6 7 8 9 11 ...
##   ..$ test    : int [1:73] 446 215 256 116 592 284 294 300 402 455 ...
##  $ Fold9 :List of 2
##   ..$ training: int [1:658] 1 2 3 4 5 6 7 8 9 10 ...
##   ..$ test    : int [1:73] 25 634 717 545 76 378 53 194 70 346 ...
##  $ Fold10:List of 2
##   ..$ training: int [1:658] 1 2 3 4 5 6 7 8 10 11 ...
##   ..$ test    : int [1:73] 468 609 40 101 595 132 248 524 376 618 ...
```

Now, we have 10 different subsets in the `folds` object. We can use `lapply()` to fit the model. 90% of data will be used for training so we use `[-x,]` to represent all observations not in a specific fold. In Chap. 11 we showed building a neutral network model for the *Google Trends* data. We can do the same for each fold manually; train, test, aggregate the results, and report the agreement (correlations between the predicted and observed RealEstate values).

```
library(neuralnet)

fold_cv<-lapply(folds, function(x){
  google_train<-google_norm[-x, ]
  google_test<-google_norm[x, ]
  google_model<-neuralnet(RealEstate~Unemployment+Rental+Mortgage+Jobs+Inves
ting+DJI_Index+StdDJI, data=google_train)
  google_pred<-compute(google_model, google_test[, c(1:2, 4:8)])
  pred_results<-google_pred$net.result
  pred_cor<-cor(google_test$RealEstate, pred_results)
  return(pred_cor)
  })
str(fold_cv)

## List of 10
##  $ Fold01: num [1, 1] 0.977
##  $ Fold02: num [1, 1] 0.97
##  $ Fold03: num [1, 1] 0.972
##  $ Fold04: num [1, 1] 0.979
##  $ Fold05: num [1, 1] 0.976
##  $ Fold06: num [1, 1] 0.974
##  $ Fold07: num [1, 1] 0.971
##  $ Fold08: num [1, 1] 0.982
##  $ Fold09: num [1, 1] -0.516
##  $ Fold10: num [1, 1] 0.974
```

From the output, we know that in most of the folds the model predicts very well. In a typical run, one fold may yield bad results. We can use the *mean* of these 10 correlations to represent the *overall* model performance. But first, we need to use `unlist()` function to transform `fold_cv` into a vector.

```
mean(unlist(fold_cv))

## [1] 0.8258223801
```

This correlation is high, suggesting strong association between predicted and true values. Thus, the model is very good in terms of its prediction.

14.4.3 Bootstrap Sampling

The second method is called *bootstrap sampling*. In k-fold CV, each observation can only be used once. However, bootstrap sampling is a sampling process *with replacement*. Before selecting a new sample, it recycles every observation so that each observation could appear in multiple folds.

A very special setting of bootstrap uses at each iteration 63.2% of the original data as our training dataset and the remaining 36.8% as the test dataset. Thus, compared to k-fold CV, bootstrap sampling is less representative of the full dataset. A special case of bootstrapping, *0.632 bootstrap*, addresses this issue by changing the final performance metric using the following formula:

$$error = 0.632 \times error_{test} + 0.368 \times error_{train}.$$

This synthesizes the optimistic model performance on training data with the pessimistic model performance on test data by weighting the corresponding errors. This method is extremely good for small samples.

To see the rationale behind *0.632 bootstrap*, consider a standard training set T of cardinality n, where our bootstrap sampling generates m new training sets T_i, each of size n'. Sampling from T is uniform *with replacement*, suggests that some observations may be repeated in each sample T_i. Suppose the size of the sub-samples are of the same order as T, i.e., $n' = n$, then for large n the sample D_i is *expected* to have $\left(1 - \frac{1}{e}\right) \sim 0.632$ unique cases from the complete original collection T, the remaining proportion 0.368 are expected to be repeated duplicates. Hence, the name *0.632 bootstrap* sampling. In general, for large $n \gg n'$, the sample D_i is *expected* to have $n\left(1 - e^{-\frac{n'}{n}}\right)$ unique cases, see On Estimating the Size and Confidence of a Statistical Audit).

Having the bootstrap samples, the m models can be fitted (estimated) and aggregated, e.g., by averaging the outputs (for regression) or by using voting methods (for classification). We will discuss this more in later chapters.

Try to apply the same techniques to some of the other data in the list of Case-Studies.

14.5 Assignment: 14. Evaluation of Model Performance

The ABIDE dataset includes imaging, clinical, genetics and phenotypic data for over 1000 pediatric cases – *Autism Brain Imaging Data Exchange* (ABIDE).

- Apply *C5.0* to predict on part of data (training data).
- Evaluate the model's performance, using confusion matrices, accuracy, κ, precision, and recall, F-measure, etc.
- Explain and compare each evaluation.
- Use the ROC to examine the tradeoff between detecting true positives and avoiding the false positives and report AUC.
- Finally, apply cross validation on *C5.0* and report the CV error.
- You may apply the same analysis workflow to evaluate the performance of alternative methods (e.g., KNN, SVM, LDA, QDA, Neural Networks, etc.)

References

SciKit: http://scikit-learn.org/stable/modules/classes.html

Sammut, C, Webb, GI (eds.) (2011) Encyclopedia of Machine Learning, Springer Science & Business Media, ISBN 0387307680, 9780387307688.

Japkowicz, N, Shah. M. (2011) Evaluating Learning Algorithms: A Classification Perspective, Cambridge University Press, ISBN 1139494147, 9781139494144.