Lecture 12

Evaluating Model Performance

Measuring performance for classification

In the previous lectures, we measured classifier accuracy by dividing the proportion of correct predictions by the total number of predictions. This indicates the percentage of cases in which the learner is right or wrong. For example, we can draw a cross table, showing that a classifier correctly classified 99.99% of all lesions as benign and was wrong in 0.01%, implying an accuracy of 99.99%. On the surface, this looks good. A problem arises, however, if all malignant lesions are only 0.01%, i.e., the classifier systematically fails to recognize malignant lesions. Such classifier is no better than a classifier that classifies each case as benign regardless of its specifics. This is an example of the **class imbalance problem**, which refers to the problem associated with classifying data having a large majority of records belonging to a single class.

Many classifiers provide the estimated probability of prediction in addition to the predicted class values. These prediction probabilities provide useful data to evaluate a model's performance. If two models make the same number of mistakes, but one is more capable of accurately assessing its uncertainty, then it is a better model.

Measuring performance for classification: prediction probabilities with C5.0

Let's go back to identifying risky bank loans with C5.0 in Lecture 7:

```
> credit <- read.csv("credit.csv")
> set.seed(123)
> train_sample <- sample(1000, 900)
> credit_train <- credit[train_sample, ]
> credit_test <- credit[-train_sample, ]
> library(C50)
Warning message:
package 'C50' was built under R version 3.2.3
> credit_model <- C5.0(credit_train[-17],
credit_train$default)
> predicted_prob <- predict(credit_model, credit_test, type =
"prob")
> credit_pred <- predict(credit_model, credit_test)
```

`credit-pred` is a factor with two levels "yes" or "no" depending on whether the loan was classified as risky or not. It does not provide any estimated probability of the prediction.

Measuring performance for classification: prediction probabilities with C5.0

We can use the *predict()* function with type = "prob" as follows:

```
> predicted_prob <- predict(credit_model, credit_test, type = "prob")
```

Now, `predicted_prob` has two columns with the prediction probabilities for ‘yes” and “no”:

```
> head(predicted_prob)
            no        yes
27  0.8782163 0.1217837
28  0.8782163 0.1217837
76  0.8041553 0.1958447
96  0.3117222 0.6882778
101 0.8782163 0.1217837
104 0.8041553 0.1958447
```

Measuring performance for classification: prediction probabilities with Naive Bayes

Let's see how prediction probabilities work with Naïve Bayes. First we, need to repeat the main steps from Lecture 6:

```
> library(tm)
> sms_raw <- read.csv("sms_spam.csv", stringsAsFactors = FALSE)
> sms_raw$type <- factor(sms_raw$type)
> sms_corpus <- VCorpus(VectorSource(sms_raw$text))
sms_dtm2 <- DocumentTermMatrix(sms_corpus, control = list(
+ tolower = TRUE,
+ removeNumbers = TRUE,
+ stopwords =  TRUE,
+ removePunctuation = TRUE,
+ stemming = TRUE
+ ))
> sms_dtm_train <- sms_dtm[1:4169, ]
> sms_dtm_test <- sms_dtm[4170:5559, ]
>  sms_train_labels <- sms_raw[1:4169, ]$type
>  sms_test_labels <- sms_raw[4170:5559, ]$type
> sms_freq_words <- findFreqTerms(sms_dtm_train, 5)
```

Measuring performance for classification: prediction probabilities with Naive Bayes

```
> sms_dtm_freq_train<- sms_dtm_train[ , sms_freq_words]
> sms_dtm_freq_test <- sms_dtm_test[ , sms_freq_words]
> convert_counts <- function(x) { x <- ifelse(x > 0, "Yes", "No") }
> sms_train <- apply(sms_dtm_freq_train, MARGIN = 2, convert_counts)
> sms_test <- apply(sms_dtm_freq_test, MARGIN = 2, convert_counts)
> library(e1071)
> sms_classifier <- naiveBayes(sms_train, sms_train_labels)
```

To output the naive Bayes predicted probabilities, we use *predict()* with type = "raw" as follows:

```
> sms_test_prob <- predict(sms_classifier, sms_test, type = "raw")
```

Now, `sms_test_prob` has two columns for the prediction probabilities for ham and spam.

```
> head(sms_test_prob)
              ham          spam
[1,] 9.999996e-01 3.864077e-07
[2,] 9.999925e-01 7.485993e-06
[3,] 9.998623e-01 1.377424e-04
[4,] 9.999632e-01 3.676120e-05
[5,] 5.090363e-10 1.000000e+00
[6,] 9.998273e-01 1.727230e-04
```

Measuring performance for classification: prediction probabilities with Naive Bayes

The author of the textbook has constructed a data frame, called sms_results, containing the predicted class values, actual class values, as well as the estimated probabilities of spam and ham:

```
> sms_results <- read.csv("sms_results.csv")
> head(sms_results)
  actual_type predict_type prob_spam prob_ham
1         ham          ham   0.00000  1.00000
2         ham          ham   0.00000  1.00000
3         ham          ham   0.00016  0.99984
4         ham          ham   0.00004  0.99996
5        spam         spam   1.00000  0.00000
6         ham          ham   0.00020  0.99980
```

For these six test cases, the predicted and actual SMS message types agree; the model predicted their status correctly. Furthermore, the prediction probabilities suggest that model was extremely confident about these predictions because they all are close to zero or one.

Measuring performance for classification: prediction probabilities with Naive Bayes

Let's see some observations where the prediction probabilities are not that strong. For example, we can use the subset() function to extract the records for which the probability of spam is between 40 and 60 percent:

```
> head(subset(sms_results, prob_spam > 0.40 & prob_spam < 0.60))
     actual_type predict_type prob_spam prob_ham
377         spam          ham   0.47536  0.52464
717          ham         spam   0.56188  0.43812
1311         ham         spam   0.57917  0.42083
```

Since the head() function returns the first 6 records, we can conclude that there are exactly three records for which the probability of spam is between 40 and 60 percent. The prediction probabilities were close to 50% and , therefore, they were no stronger than flipping a coin. It should not come as a surprise that the model was wrong for these three cases.

By using the subset() function we can see how many predictions were wrong:

```
> nrow(subset(sms_results, actual_type != predict_type))
[1] 35
```

Measuring performance for classification: prediction probabilities with Naive Bayes

The first 6 wrong predcitions can be extracred using the head() function:

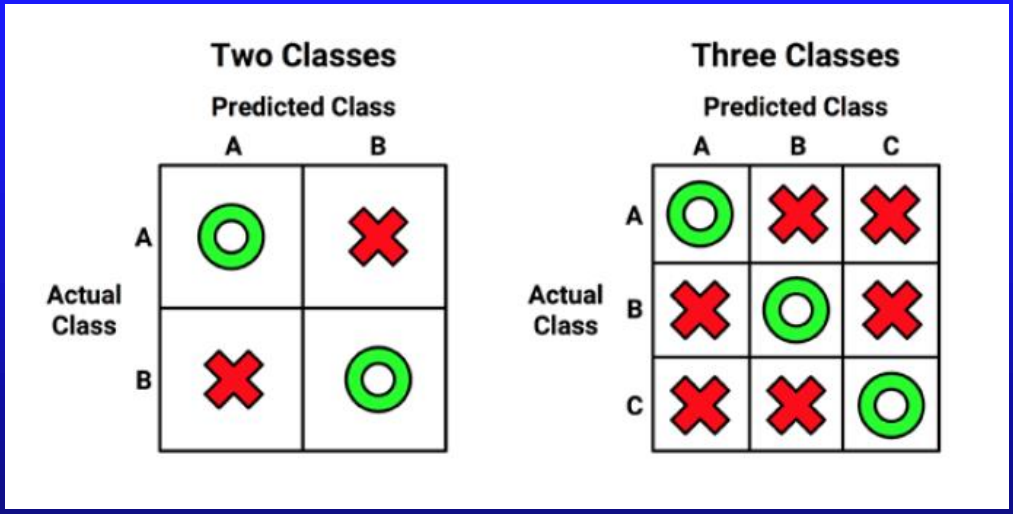
```
> head(subset(sms_results, actual_type != predict_type))
    actual_type predict_type prob_spam prob_ham
53         spam          ham   0.00071  0.99929
59         spam          ham   0.00156  0.99844
73         spam          ham   0.01708  0.98292
76         spam          ham   0.00851  0.99149
184        spam          ham   0.01243  0.98757
332        spam          ham   0.00003  0.99997
```

It is interesting the model was extremely confident (with prediction probabilities of 99% and 98%) and yet it was wrong.

A closer look at confusion matrices

A **confusion matrix,** also known as an error **matrix,** is a table that categorizes predictions according to whether they match the actual values. One of the table's dimensions indicates the possible categories of predicted values, while the other dimension indicates the categories for actual values.

When the predicted value coincides with the actual value, it is a correct classification. Correct predictions fall on the diagonal in the confusion matrix (denoted by **O**). The off-diagonal matrix cells (denoted by **X**) indicate incorrect predictions, i.e., the cases where the predicted value differs from the actual value.



A closer look at confusion matrices

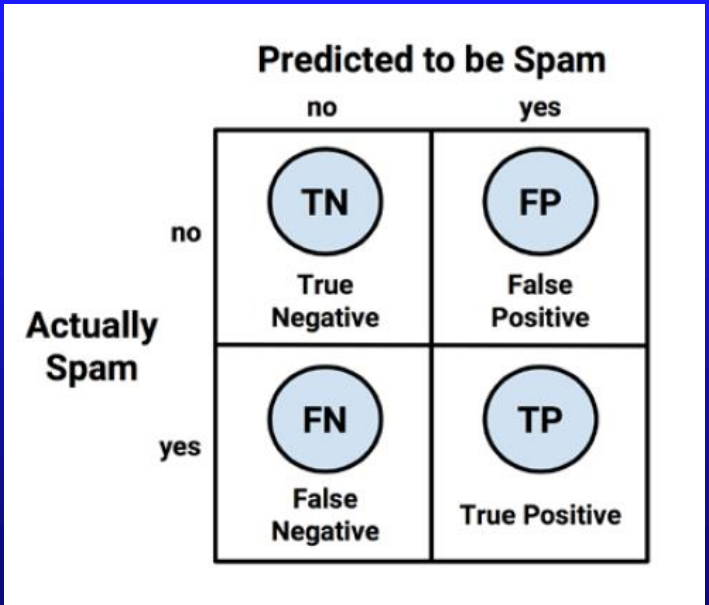
The relationship between the positive class and negative class predictions can be depicted as a 2 x 2 confusion matrix that tabulates whether predictions fall into one of the four categories:

True Positive (TP): An “yes” case correctly classified as “yes” case

True Negative (TN): A “no” case correctly classified as “no” case

False Positive (FP) or Type I error: A “no” case incorrectly classified as “yes” case

False Negative (FN) or Type II error: An “yes” case incorrectly classified as “no” case



Using confusion matrices to measure performance

With the 2 x 2 confusion matrix, we can define **accuracy** (or **success rate**) as:

$$\text{accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}}$$

The numerator equals the number of all correctly classified cases, whereas the denominator equals the total number of cases. In other words, the accuracy measures the frequency of making correct predictions.

The **error rate** or the proportion of the incorrectly classified examples:

$$\text{error rate} = \frac{\text{FP} + \text{FN}}{\text{TP} + \text{TN} + \text{FP} + \text{FN}} = 1 - \text{accuracy}$$

A closer look at confusion matrices

False negatives may different from false negatives in terms of their misclassification cost, importance, etc. For example, in the area of lesion diagnostic, FN (classifying a malignant lesion as benign) is more undesirable and costly than FN (classifying a benign lesion as malignant).

The easiest way to draw a confusion matrix is by using the table() function:

```
> table(sms_test_labels, sms_test_pred)
               sms_test_pred
sms_test_labels  ham spam
           ham  1203    4
           spam   30  153
```

In other words, TP=1203, TN=153, FP=4, FN=30.

The accuracy rate is:

```
> (153 + 1203) / (153 + 1203 + 4 + 30)
[1] 0.9755396
```

A closer look at confusion matrices

The error rate is:

```
> 1 - 0.9748201
[1] 0.0251799
```

A better way to visualize the confusion matrix is by creating a crosstab on the actual and predicted data (we did this in Lecture 6):

```
> library(gmodels)
> CrossTable(sms_test_pred, sms_test_labels,
+ prop.chisq = FALSE, prop.t = FALSE,
+ dnn = c('predicted', 'actual'))
```

A closer look at confusion matrices

Total Observations in Table: 1390

	actual		
predicted	ham	spam	Row Total
ham	1203	30	1233
	0.976	0.024	0.887
	0.997	0.164	
spam	4	153	157
	0.025	0.975	0.113
	0.003	0.836	
Column Total	1207	183	1390
	0.868	0.132	

Other measures of performance

The Classification and Regression Training package *caret* by Max Kuhn includes functions to compute many such performance measures.

```
> install.packages("caret")
> library(caret)
```

The package provides an advanced version of confusion matrix function:

```
> confusionMatrix(sms_test_pred,sms_test_labels, positive = "spam")
```

We need to specify the positive class “spam”. Otherwise, the function would not be able to determine which errors are FP and FN. If we swap the two classes, the FPs become FNs, and vice versa.

The results are shown on the next slide.

Other measures of performance

```
Confusion Matrix and Statistics
          Reference
Prediction  ham spam
      ham  1203   30
      spam    4  153
               Accuracy : 0.9755
                 95% CI : (0.966, 0.983)
    No Information Rate : 0.8683
    P-Value [Acc > NIR] : < 2.2e-16
                  Kappa : 0.8862
 Mcnemar's Test P-Value : 1.807e-05
            Sensitivity : 0.8361
            Specificity : 0.9967
         Pos Pred Value : 0.9745
         Neg Pred Value : 0.9757
             Prevalence : 0.1317
         Detection Rate : 0.1101
   Detection Prevalence : 0.1129
      Balanced Accuracy : 0.9164

       'Positive' Class : spam
```

Confusion Matrix and Statistics

The **No Information Rate** is the proportion of the largest class. In our case, the largest class is ham with 1207 observations. Hence, the No Information Rate is 1207/1390 = 0.8683

A hypothesis test is also computed to evaluate whether the overall accuracy rate is greater than the rate of the largest class:

```
P-Value [Acc > NIR] : < 2.2e-16
```

Suppose that the confusion matrix looks as follows:

		Reference	
	Predicted	Event	No Event
	Event	A	B
	No Event	C	D

Then some of the measures produced by the confusionMatrix() function as defined as:

Confusion Matrix and Statistics

	Reference	
Predicted	Event	No Event
Event	A	B
No Event	C	D

$$Sensitivity = \frac{A}{A+C}$$

$$Specificity = \frac{D}{B+D}$$

$$Prevalence = \frac{A+C}{A+B+C+D}$$

$$PPV = \frac{sensitivity \times prevalence}{((sensitivity \times prevalence) + ((1 - specificity) \times (1 - prevalence))}$$

$$NPV = \frac{specificity \times (1 - prevalence)}{((1 - sensitivity) \times prevalence) + ((specificity) \times (1 - prevalence))}$$

$$Detection\ Rate = \frac{A}{A+B+C+D}$$

$$Detection\ Prevalence = \frac{A+B}{A+B+C+D}$$

The kappa statistic

The kappa statistic**,** also called **Cohen's kappa** statistic, adjusts accuracy by accounting for the possibility of a correct prediction by chance alone. This is especially important for datasets with a severe class imbalance, because a classifier can obtain high accuracy simply by always guessing the most frequent class.

The kappa statistic is defined as:

$$\kappa = \frac{\Pr(a) - \Pr(e)}{1 - \Pr(e)}$$

The kappa statistic

Here, *Pr(a)* is to the proportion of the actual agreement and *Pr(e)* refers to the expected agreement between the classifier and the true values, under the assumption that they were chosen at random. According to Landis and Koch, value between 0–0.20 indicate slight agreement, 0.21–0.40 correspond to fair, 0.41–0.60 are moderate, 0.61–0.80 are substantial, and 0.81–1 are almost perfect agreement.

To compute the kappa statistic let's look again at the confusion matrix on slide #15. The proportion of the actual agreement, *Pr(a),* is the proportion of TPs plus the proportion of TNs:

(TP + TN) / (TP + TN + FP + FN)

Therefore, *Pr(a)* = (1203 + 153)/(1203 + 153 + 30 + 4) = 0.9755. This is the same as the accuracy. How to compute the proportion (or the probability) of the expected agreement? According to the confusion matrix, the classifier has classified 88.7% of all messages as ham and 11.3% as spam. Therefore, it classifies a messages as ham with probability 0.887 and as spam with probability 0.113.

The kappa statistic

On the other hand, 86.8% of all messages are ham and 13.2% are spam. Hence, a randomly chosen message is ham with probability 0.868 and spam with probability 0.132. We can calculate the probability of expected agreement, *Pr(e),* as:

P(message is ham) * P(message is classified as ham) + P(message is spam) * P(message is classified as spam) = 0.887 * 0.868 + 0.113 * 0.132 = 0.7848

Plugging the *Pr(a)* and *Pr(e)* values into the kappa formula yields:

```
> pr_e<-0.7848
> pr_a<- 0.9755
> k <- (pr_a - pr_e) / (1 - pr_e)
> k
[1] 0.8861524
```

Based on the kappa statistic, we can conclude that there is very good agreement between the classifier's predictions and the actual values. The kappa statistic can be directly computed using the postResample() function:

```
> postResample(sms_test_pred,sms_test_labels)
 Accuracy     Kappa
0.9755396 0.8861583
```

Sensitivity and specificity

Finding a useful classifier often involves a balance between predictions that are overly conservative and overly aggressive. For example, an e-mail filter could guarantee to eliminate every spam message by aggressively eliminating nearly every ham message at the same time. On the other hand, guaranteeing that no ham message is inadvertently filtered might require us to allow an unacceptable amount of spam to pass through the filter.

The **sensitivity** of a model (also called the **true positive rate**) measures the proportion of positive examples that were correctly classified:

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

The numerator is the TPs, whereas the denominator is the number of all positives (the ones that are correctly classified plus the ones that are misclassified).

The **specificity** of a model (also called the **true negative rate**) measures the proportion of negative examples that were correctly classified:

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

Sensitivity and specificity

The sensitivity and the specificity are provided by the confusionMatrix() function. We can also compute them by hand:

```
> sens <- 153 / (153 + 30)
> sens
[1] 0.8360656
> spec <- 1203 / (1203 + 4)
> spec
[1] 0.996686
```

Sensitivity and specificity range from 0 to 1, with values close to 1 being more desirable. In many cases, there is a tradeoff between sensitivity and specificity. Usually, changes are made to the model and different models are tested until you find one that meets a desired sensitivity and specificity threshold.

Precision and recall

Precision and recall were initially introduced in the field of information retrieval. The statistics measure how interesting and relevant a model's results are, or whether the predictions are diluted by meaningless noise.

The **precision** (also known as the **positive predictive value**) is defined as the proportion of positive examples that are truly positive; in other words, when a model predicts the positive class, how often is it correct?

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

If the precision is small, the positive prediction of a model cannot be trusted. In the context of information retrieval (Google search, for example), precision is defined as the proportion of the relevant documents in the population of retrieved documents. For example, if only 30% of the retrieved documents are relevant to the search query, then the precision is 30%.

Precision and recall

Formally, recall is the same as sensitivity:

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

The interpretation is slightly different. A model with a high recall captures a large portion of the positive examples, meaning that it has wide breadth. In the context of information retrieval, recall measures the proportion of retrieved documents in the population of all relevant documents. For example, if only 10% of all relevant documents are retrieved, then the recall is 10%.

The precision and the recall can be computed manually using the confusion matrix:

```
> prec <- 153 / (153 + 4)
> prec
[1] 0.9745223
> rec <- 153 / (153 + 30)
> rec
[1] 0.8360656
```

Recall can also be computed using the posPredValue() function:

```
> posPredValue(sms_test_labels, sms_test_pred,
positive = "spam")
[1] 0.8360656
```

Obtaining high precision and recall at the same time is challenging.

The F-measure

The precision and recall can be combined together in one measure, called the F-measure. The F-measure is defined as the harmonic mean of the precision and the recall:

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

In math, the harmonic mean is used instead of the arithmetic mean to measure the average of rates of change. It is defined as the reciprocal of the arithmetic mean of the reciprocals. That is the harmonic mean of 1, 2, and 3 is:

3/(1/1 + 1/2 + 1/3)

For two numbers, x and y, the harmonic mean is:

2/(1/x + 1/y) = 2*x*y/(x +y)

The F measure can be computed manually:

```
> f <- (2 * prec * rec) / (prec + rec)
> f
[1] 0.9
```

The F-measure

One drawback of the F-measure is that it assigns equal weights to precision and recall. For this reason it is also called the F_1-measure. There is a more general version of the F-measure, called F_β-measure, which assigns different weights to precision and recall:

$F_\beta = (1 + \beta^2) * \text{precision} * \text{recall} / (\beta^2 * \text{precision} + \text{recall})$

Visualizing performance trade-offs

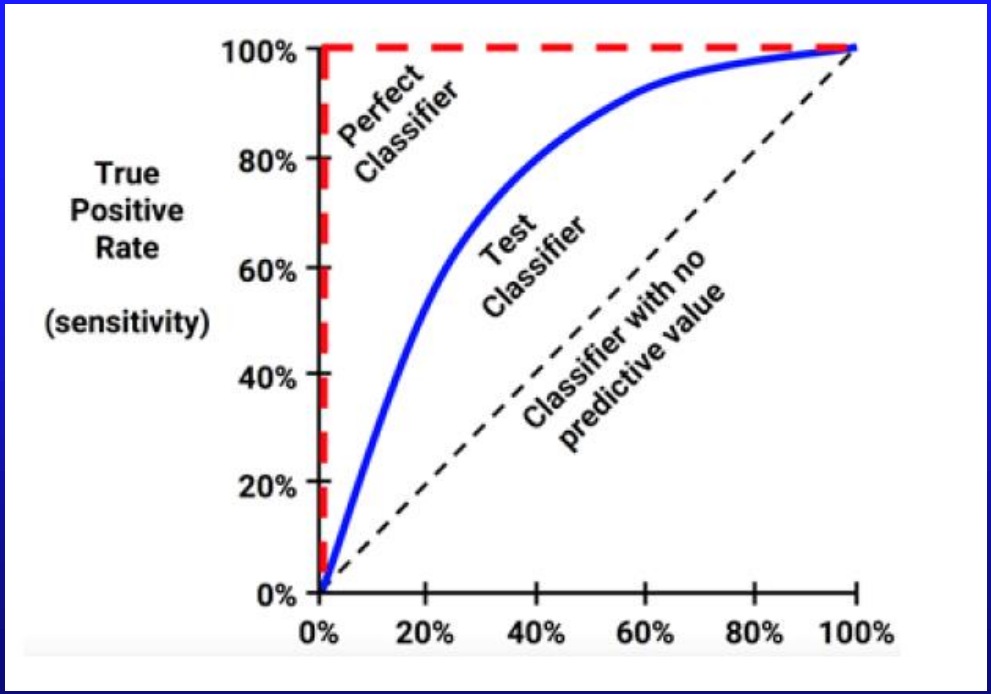
The **Receiver Operating Characteristic** (ROC) curve is commonly used to examine the trade-off between the detection of true positives, while avoiding the false positives. The ROC curve was first developed by electrical engineers and radar engineers during World War II for detecting enemy objects in battlefields. The engineers were developing a series of radar detectors to identify incoming German planes. Since radar detectors would also detect flocks of birds and other "false positive" signals, ROC curves were used to evaluate their performance.

Nowadays, ROC curves are widely used in ML, data mining, psychology, medicine, biometrics, etc. ROC analysis provides tools to select possibly optimal models and to discard suboptimal ones.

The curve is created by plotting the true positive rate against the false positive rate at various threshold settings. The true-positive rate is the same as sensitivity or recall. The false-positive rate is also known as the fall-out and can be calculated as 1 – specificity.

ROC curves

The perfect prediction method would yield a point with coordinates (0,100%) in the upper left corner of the ROC space, representing 100% sensitivity (no false negatives) and 100% specificity (no false positives). A completely random guess would produce a point along a diagonal line (the so-called *line of no-discrimination*) from the left bottom to the top right corner. The diagonal line divides the ROC space into two halves. Points above the diagonal represent classification results that are better than random, whereas points below the diagonal line represent poor results, which are worse than random guess.



ROC curves

The closer the curve is to the perfect classifier, the better it is at identifying positive values. This can be measured using a statistic known as the **area under the ROC curve** (abbreviated **AUC**). The AUC treats the ROC diagram as a two-dimensional square and measures the total area under the ROC curve. AUC ranges from 0.5 (for a classifier with no predictive value) to 1.0 (for a perfect classifier). Based on the AUC, classifiers can be rated as:

Outstanding = 0.9 to 1.0

Excellent/good = 0.8 to 0.9

Acceptable/fair = 0.7 to 0.8

Poor = 0.6 to 0.7

No discrimination = 0.5 to 0.6

Note that very different classifiers may have the same AUC. Therefore, the AUC measure must be considered in combination with the overall shape of the ROC curve.

ROC curves

We will use the ROCR package to draw ROC curves.

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

Every classifier evaluation using ROCR starts with creating a prediction object. This function is used to transform the input data (the probability of spam and the actual labels) into a standardized format.

```
> pred <- prediction(predictions = sms_results$prob_spam,
+ labels = sms_results$actual_type)
```

Then, we need to create a performance object from the prediction object using the the performance() function:

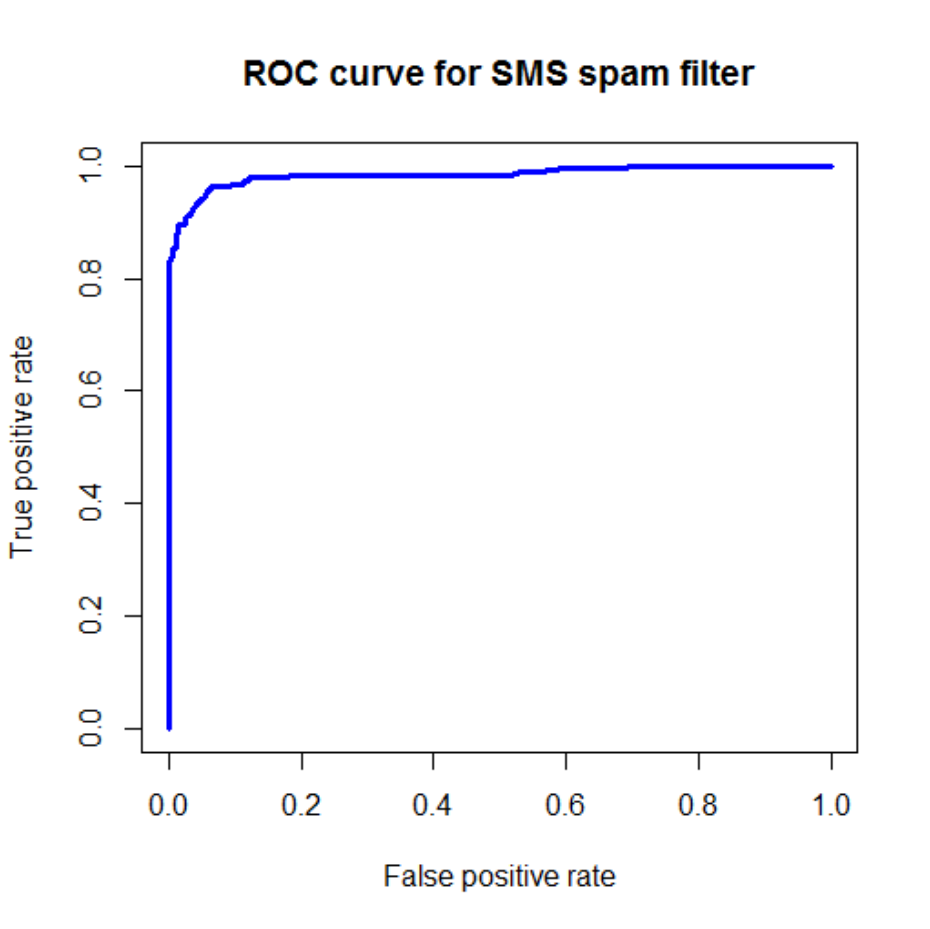
```
> perf <- performance(pred, measure = "tpr", x.measure = "fpr")
```

The performance function is quite versatile and can be used to map different performance measures, such as accuracy, error rate, false positive rate, fallout, true positive rate, recall, sensitivity, precision, AUC, etc.

Finally, we can plot the ROC curve with the plot() function:

```
> plot(perf, main = "ROC curve for SMS spam filter", col = "blue",
+ lwd = 3)
```

ROC curves


ROC curve for SMS spam filter
True positive rate
0.0
0.2
0.4
0.6
0.8
1.0
False positive rate
0.0
0.2
0.4
0.6
0.8
1.0

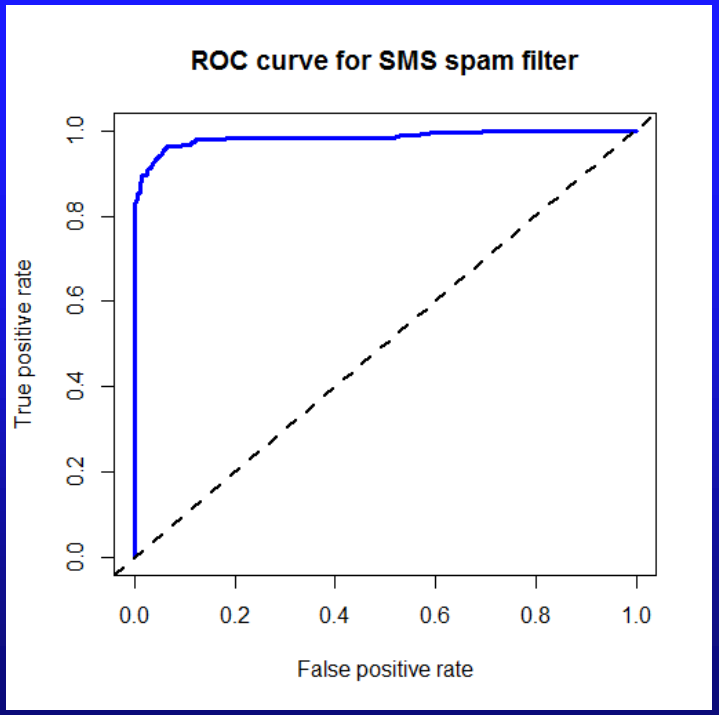
ROC curves

We will add the diagonal line to the ROC curve using the abline() function:

```
> abline(a = 0, b = 1, lwd = 2, lty = 2)
```

- a is the intercept. It is 0 since the line passes through the origin.
- b is the slope. It is 1 since the line is diagonal.
- lwd specifies the line thickness,
- lty parameter specifies the type of the line. It is dashed in our case.

Here is the final plot.



ROC curves

The ROC curve seems to be good because it is closer to a perfect classifier. In order to get a better look at the curve, we will also compute the AUC. We will use the performance() function again:

```
> perf.auc <- performance(pred, measure = "auc")
> perf.auc <- performance(pred, measure = "auc")
> str(perf.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 0.984
  ..@ alpha.values: list()
```

The perf.auc is an R object known as an S4 object:

```
> summary(perf.auc)
     Length       Class        Mode
          1 performance          S4
```

ROC curves

The AUC value is packed in the y.values list. It can be extracted using the unlist() function.

```
> unlist(perf.auc@y.values)
[1] 0.9835862
```

The AUC for the SMS classifier is 0.98, which is quite high. However, how do we know whether the model is likely to generalize and perform well on other data?

Estimating future performance

One way to estimate a model's future performance is by using the model's **resubstitution error**, i.e., the error it makes on the training data. This, however, is not a good measure because a zero resubstitution error does not mean that the model will generalize well on unknown data.

Anther way to estimate a model's future performance is to apply the model on new data. We did this before when we split the available data into a set for training and a testing set. In some cases, however, it is not always ideal to create training and test datasets. For instance, in a situation where you have only a small pool of data, you might not want to reduce the sample any further.

The procedure of partitioning data into training and test datasets is called the **holdout method**. Typically, about one third of the data is held out for testing, and two-thirds is used for training, but this proportion can vary depending on the amount of available data. To ensure that the training and test data do not have systematic differences, their examples are randomly divided into the two groups.

The holdout method

Sometimes, practitioners are temped to build several models on the training data, and selected the one with the highest accuracy on the test data. This should be avoided because the test performance is not an unbiased measure of the performance on unseen data.

It is better to divide the original data so that in addition to the training datasets and the test datasets, a **validation dataset** is available. The validation dataset is used for iterating and refining the model or models chosen, leaving the test dataset to be used only once as a final step to report an estimated error rate for future predictions. A typical split between training, test, and validation would be 50 percent, 25 percent, and 25 percent, respectively.

The simplest way to build the training, test and validation sets is by using a random number generator to assign records to partitions. Suppose we have a data frame named credit with 1000 rows of data. We can divide it into three partitions as follows. First, we create a vector of randomly ordered row IDs from 1 to 1000 using the runif() function:

```
> random_ids <- order(runif(1000))
```

The holdout method

The runif() generates 1000 random numbers between 0 and 1 using a uniform distribution. Hence the name runif. In order to convert these numbers into a random stream of numbers within the range 1…1000, we use the order() function, which takes a vector of numbers and returns a vector of their corresponding ranks. For example,

```
> x<-order(c(0.3,0.1,0.2))
> x
[1] 2 3
```

This way, random_ids is a random reordering of the numbers from 1 to 1000. We can use random_ids to divide the credit data frame into 500, 250, and 250 records comprising the training, validation, and test datasets:

```
> credit_train <- credit[random_ids[1:500], ]
> credit_validate <- credit[random_ids[501:750], ]
> credit_test <- credit[random_ids[751:1000], ]
```

One problem with holdout sampling is that each partition may have a larger or smaller proportion of some classes. For example, a very small class can be omitted from the training set.

The holdout method

In order to reduce the chance of this occurring, we can use a technique called stratified random sampling which guarantees that the random partitions have nearly the same proportion of each class as the full dataset, even when some classes are small.

We will use as an example the credit dataset from Chapter 5:

```
> credit <- read.csv("credit.csv")
```

The createDataPartition() function from the caret package can be used to create partitions based on stratified holdout sampling:

```
> in_train <- createDataPartition(credit$default, p = 0.75, list = FALSE)
```

The partitions are made based on the credit default status, which is a factor with two values: “no” and “yes”. p=0.75 indicates the size of the partition, i.e., 75% of all records will be selected as a training set. list = FALSE parameter prevents the result from being stored in the list format. The in_train vector indicates row numbers included in the training sample. Using the in_train vector, we can select the training and the test sets:

The holdout method

```
> credit_train <- credit[in_train, ]
> credit_test <- credit[-in_train, ]
```

Now, we can see that the two classes “yes” and “no” are proportionally divided between the training and the test sets:

```
> table(credit_train$default=="yes")
FALSE  TRUE
  525   225
> table(credit_test$default=="yes")
FALSE  TRUE
  175    75
```

Although stratified sampling distributes the classes evenly, it does not guarantee other types of representativeness. Some samples may have too many or few difficult cases, easy-to-predict cases, or outliers. This is especially true for smaller datasets, which may not have a large enough portion of such cases to be divided among training and test sets. Another problem with the holdout method is that substantial portions of data must be reserved to test and validate the model Since models trained on larger datasets generally perform better, a common practice is to retrain the model on the full set of data (that is, training plus test and validation) after a final model has been selected and evaluated.

Cross-validation

In k-fold cross-validation (or **k-fold CV**), the dataset is divided into *k* separate random partitions called **folds**. Of the *k* folds, one fold is used as the validation data for testing the model, and the remaining $k - 1$ folds are used as training data. The cross-validation process is repeated *k* times, with each of the *k* folds used exactly once as the validation data. This way, *k* results are obtained, which are averaged to produce a single estimation. The advantage of k-fold cross validation is that all observations are used for both training and validation, and each observation is used for validation exactly once. 10-fold cross-validation is commonly used.

The original dataset can be divided into k folds using the createFolds() function in the caret package:

```
> folds <- createFolds(credit$default, k = 10)
```

Here, we are creating 10 folds. The object folds is a list of 10 vectors containing the row number for each fold:

Cross-validation

```
> str(folds)
List of 10
 $ Fold01: int [1:100] 4 11 13 34 40 46 83 92 99 104 ...
 $ Fold02: int [1:100] 5 6 43 52 62 63 77 78 80 90 ...
 $ Fold03: int [1:100] 8 18 33 41 58 67 88 91 100 132 ...
 $ Fold04: int [1:100] 2 3 22 24 44 66 103 113 115 134 ...
 $ Fold05: int [1:100] 14 16 27 38 56 65 82 84 86 98 ...
 $ Fold06: int [1:100] 21 28 29 45 47 49 57 60 68 72 ...
 $ Fold07: int [1:100] 9 17 31 32 36 42 51 61 64 69 ...
 $ Fold08: int [1:100] 1 19 20 37 54 74 117 125 140 150 ...
 $ Fold09: int [1:100] 10 12 15 25 26 30 35 39 48 50 ...
 $ Fold10: int [1:100] 7 23 53 55 59 71 76 79 89 108 ...
```

To create a test set from the first fold we need to select all rows with numbers belonging fold1:

```
> credit01_test <- credit[folds$Fold01, ]
```

Then, all remaining rows go into the training set:

```
 > credit01_train <- credit[-folds$Fold01, ]
```

Cross-validation

As an example, we will perform 10-fold cross-validation on the training dataset and will estimate the kappa statistic for a C5.0 decision tree model.

```
> library(C50)
> install.packages("irr")
> library(irr)
```

We need the package irr to compute the kappa statistic. Then, we define a function that performs a single round of cross validation and apply the function to the whole list of folds. The results, cv_results, is a list of kappa ststisctics.

```
> cv_results <- lapply(folds, function(x) {
+ credit_train <- credit[-x, ]
+ credit_test <- credit[x, ]
+ credit_model <- C5.0(default ~ ., data = credit_train)
+ credit_pred <- predict(credit_model, credit_test)
+ credit_actual <- credit_test$default
+ kappa <- kappa2(data.frame(credit_actual, credit_pred))$value
+ return(kappa)
+ })
```

Cross-validation

The function takes a fold as a parameter, defines the test and the training set for that fold, builds a C5.0 model for the fold, computes the predicted labels and calculates the kappa statistic using the predicted and the actual labels. The list of the 10 kappa statistics is:

```
>  str(cv_results)
List of 10
 $ Fold01: num 0.343
 $ Fold02: num 0.255
 $ Fold03: num 0.109
 $ Fold04: num 0.107
 $ Fold05: num 0.338
 $ Fold06: num 0.474
 $ Fold07: num 0.245
 $ Fold08: num 0.0365
 $ Fold09: num 0.425
 $ Fold10: num 0.505
```

Cross-validation

The last step of the 10-fold cross-validation is to compute the average kappa statistic. We cannot just type mean(cv_results) because cv_results is a list. To comute the average, we need to unlist it:

```
> mean(unlist(cv_results))
[1] 0.283796
```

The kappa statistic is low, suggesting that the credit scoring model performs only marginally better than random chance.

Quite often, a **repeated k-fold cross-validation is used**, where k-fold CV is repeated many times and the results are averaged.

Bootstrap sampling

Bootstrap (aka bootstrap or bootstrapping) is a procedure that creates several randomly selected training and test datasets, which are then used to estimate performance statistics.

Bootstrapping divides the original training set D into m new training sets, each of the same size by uniform sampling with replacement. Since the sampling is with replacement some observations may be repeated in each newly created dataset. For large datasets, each newly created dataset is expected to have the fraction 1 - 1/e ≈ 63.2% of the unique examples of the original dataset because the rest are duplicates. Then m models are fitted using the above m bootstrap samples and combined by averaging the output. Bootstrapping is often used for artificial neural networks, classification and regression trees, and subset selection in linear regression.

Bootstrap sampling

Because a model trained on only 63.2 percent of the training data is likely to perform worse than a model trained on a larger training set, the bootstrap's performance estimates may be substantially lower than what would be obtained when the model is later trained on the full dataset. A special case of bootstrapping known as the **0.632 bootstrap** accounts for this by calculating the final performance measure as a function of performance on both the training and the test data:

$$\text{error} = 0.632 \times \text{error}_{\text{test}} + 0.368 \times \text{error}_{\text{train}}$$