Naive Bayes Classifier

Short Intro

A Naive Bayes classifier is a probabilistic machine learning model used for classification task. It is based on the *Bayes theorem*:

$$P(c|x) = \frac{P(x|c) * P(c)}{P(x)}$$

Where:

- c represents the class,
- x represents features,
- *P(c|x)* is the *posterior probability* of class c given the predictor (features).
- *P(c)* is the probability of class.
- *P(x|c)* is the *likelihood* which is the probability of predictor given class.
- *P(x)* is the *prior probability* of the predictor.

Using Bayes theorem, we can find the probability of **class x**, given that we know the values of the **feature x**.

The assumption made here is that the predictors/features are independent. That is, the presence of one particular feature does not affect the other. Hence the name of the algorithm *naive*.

Prepare the data set

We will use the same data set (*Carseats*) as in the previous Lab.

```
# load ISLR package
library(ISLR)

# print dataset structure
str(Carseats)
```

```
## 'data.frame':    400 obs. of  11 variables:
##  $ Sales      : num  9.5 11.22 10.06 7.4 4.15 ...
##  $ CompPrice  : num  138 111 113 117 141 124 115 136 132 132 ...
##  $ Income     : num  73 48 35 100 64 113 105 81 110 113 ...
##  $ Advertising: num  11 16 10 4 3 13 0 15 0 0 ...
##  $ Population : num  276 260 269 466 340 501 45 425 108 131 ...
##  $ Price      : num  120 83 80 97 128 72 108 120 124 124 ...
##  $ ShelveLoc  : Factor w/ 3 levels "Bad","Good","Medium": 1 2 3 3 1 1 3 2 3 3 ...
##  $ Age        : num  42 65 59 55 38 78 71 67 76 76 ...
##  $ Education  : num  17 10 12 14 13 16 15 10 10 17 ...
##  $ Urban      : Factor w/ 2 levels "No","Yes": 2 2 2 2 2 1 2 2 1 1 ...
##  $ US         : Factor w/ 2 levels "No","Yes": 2 2 2 2 1 2 1 2 1 2 ...
```

As we did before, we will introduce a categorical (factor) variable *HighSales* to be used as the outcome variable (variable defining the class for each observation). If a sale is greater than the 3rd quartile (9.32), it qualifies as a high sale.

```r
# calculate 3rd quartile
sales.3Q <- quantile(Carseats$Sales, 0.75)

# create a new variable HighSales based on the value of the 3rd quartile
Carseats$HighSales <- ifelse(test = Carseats$Sales > sales.3Q,
                             yes = 'Yes',
                             no = 'No')

# convert HighSales from character to factor
Carseats$HighSales <- as.factor(Carseats$HighSales)
```

We'll remove the *Sales* variable, as we do not need it anymore.

```r
# remove the Sales variable
Carseats <- Carseats[,-1]
str(Carseats)
```

```
## 'data.frame':    400 obs. of  11 variables:
##  $ CompPrice  : num  138 111 113 117 141 124 115 136 132 132 ...
##  $ Income     : num  73 48 35 100 64 113 105 81 110 113 ...
##  $ Advertising: num  11 16 10 4 3 13 0 15 0 0 ...
##  $ Population : num  276 260 269 466 340 501 45 425 108 131 ...
##  $ Price      : num  120 83 80 97 128 72 108 120 124 124 ...
##  $ ShelveLoc  : Factor w/ 3 levels "Bad","Good","Medium": 1 2 3 3 1 1 3 2 3 3 ...
##  $ Age        : num  42 65 59 55 38 78 71 67 76 76 ...
##  $ Education  : num  17 10 12 14 13 16 15 10 10 17 ...
##  $ Urban      : Factor w/ 2 levels "No","Yes": 2 2 2 2 2 1 2 2 1 1 ...
##  $ US         : Factor w/ 2 levels "No","Yes": 2 2 2 2 1 2 1 2 1 2 ...
##  $ HighSales  : Factor w/ 2 levels "No","Yes": 2 2 2 1 1 2 1 2 1 1 ...
```

Numerical variables discretization

NB is often used with nominal (categorical) variables. If numeric variables are to be used, they have to be:

- represented as probabilities based on the appropriate probability distribution (typically Normal distribution), or
- discretized, that is, have their range of values split into several segments, and thus be turned into a discrete set of values.

Let's examine if our numerical variables are normally distributed so that we can decide whether to go with option (1) or to opt for (2).

As we did before, we'll use the Shapiro-Wilk test to check for normality.

```r
# filter all numerical variables
num.vars <- c(1:5,7,8)

# apply the Shapiro-Wilk test to each numerical column (variable)
apply(X = Carseats[,num.vars],
      MARGIN = 2,
      FUN = shapiro.test)
```

```
## $CompPrice
##
##  Shapiro-Wilk normality test
```

```
##
## data:  newX[, i]
## W = 0.99843, p-value = 0.9772
##
##
## $Income
##
##  Shapiro-Wilk normality test
##
## data:  newX[, i]
## W = 0.9611, p-value = 8.396e-09
##
##
## $Advertising
##
##  Shapiro-Wilk normality test
##
## data:  newX[, i]
## W = 0.87354, p-value < 2.2e-16
##
##
## $Population
##
##  Shapiro-Wilk normality test
##
## data:  newX[, i]
## W = 0.95201, p-value = 4.081e-10
##
##
## $Price
##
##  Shapiro-Wilk normality test
##
## data:  newX[, i]
## W = 0.99592, p-value = 0.3902
##
##
## $Age
##
##  Shapiro-Wilk normality test
##
## data:  newX[, i]
## W = 0.95672, p-value = 1.865e-09
##
##
## $Education
##
##  Shapiro-Wilk normality test
##
## data:  newX[, i]
## W = 0.9242, p-value = 2.427e-13
```

Only *CompPrice* and *Price* are normally distributed. The other 5 variables have to be discretized.

We will do **unsupervised discretization**. It consists of dividing a continuous variable into groups (also

referred to as *bins*) without taking into account the class attribute or any other information. It can be done in two ways:

- **equal length intervals** - the range of values of an attribute is split into k number of intervals of (roughly) equal lengths; k needs to be specified;
- **equal frequency intervals** - the range of values of an attribute is split into k number of intervals so that each interval has (roughly) an equal number of values; k needs to be set.

In general, the 2nd option tends to be better of the two.

To do the discretization seamlessly, we'll use the *discretize* f. from the *bnlearn* R package.

```
#install.packages('bnlearn')
# load bnlearn package
library(bnlearn)
```

Let's first examine this function:

```
# print the docs for the discretize f.
?discretize
```

We will discretize our numeric variables using the *equal frequency method* (the better of the two aforementioned options). We'll try to split the variables into 5 equal frequency intervals:

```
# filter all variables to be discretized
to.discretize <- c("Education", "Age", "Population", "Advertising", "Income")

# discretize all variables into 5 bins each
#discretized <- discretize(data = Carseats[,to.discretize],
#                          method = 'quantile',
#                          breaks = c(5,5,5,5,5))
```

R reports an error related to the *Advertising* variable. The reason is a very uneven (skewed) distribution of this variable so that it cannot be split as we intended to. To understand this better, let's check the distribution of this variable:

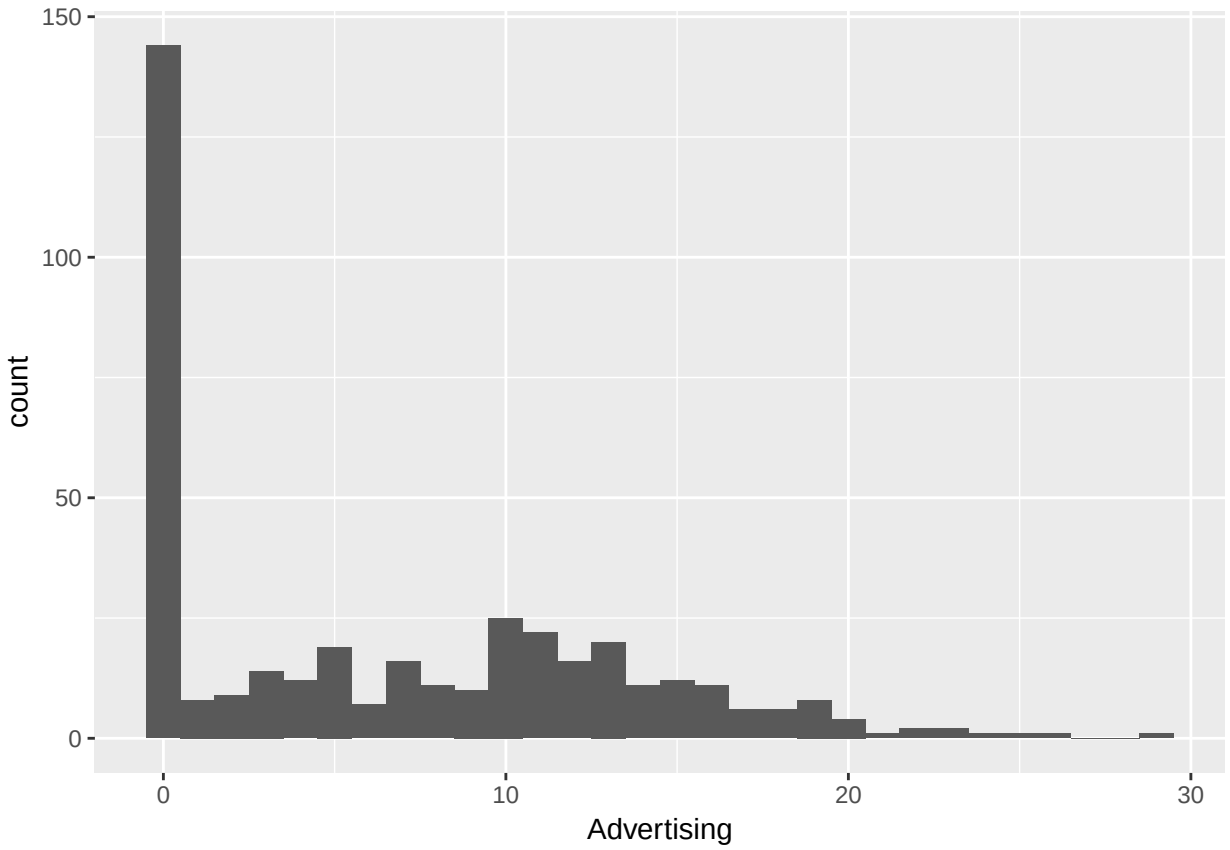
```
# print the summary for the Advertising variable
summary(Carseats$Advertising)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   0.000   0.000   5.000   6.635  12.000  29.000
```

We can also examine the distribution visually.

```
# load ggplot2
library(ggplot2)

# plot the histogram for the Advertising variable
ggplot(data = Carseats, mapping = aes(x = Advertising)) +
  geom_histogram(bins = 30)
```



So, instead of 5, we'll split the *Advertising* variable into 2 equal frequency intervals:

```
# discretize all variables into 5 bins each, but the Advertising variable into 2 bins
discretized <- discretize(data = Carseats[,to.discretize],
                          method = 'quantile',
                          breaks = c(5,5,5,2,5))
```

Examine the newly created variables.

```
# print the summary of the discretized dataset
summary(discretized)
```

```
##    Education         Age         Population Advertising         Income
##  [10,11]:96   [25,36]  :82   [10,110] :80   [0,5] :206   [21,39]   :82
##  (11,13]:92   (36,48.6]:78   (110,219]:80   (5,29]:194   (39,62]   :82
##  (13,15]:76   (48.6,60]:85   (219,318]:80                (62,77]   :77
##  (15,17]:96   (60,70]  :78   (318,412]:80                (77,96.2] :79
##  (17,18]:40   (70,80]  :77   (412,509]:80                (96.2,120]:80
```

Create a new data set by merging the discretized variables with the rest of the columns (variables) from the *Carseats* data set.

```
# calculate the difference between the two vectors (with variable names)
cols.to.add <- setdiff(names(Carseats), names(discretized))

# merge the discretized data frame with other columns from the original data frame
carseats.new <- data.frame(cbind(Carseats[,cols.to.add], discretized))
str(carseats.new)
```

```
## 'data.frame':    400 obs. of  11 variables:
```

```
##  $ CompPrice  : num  138 111 113 117 141 124 115 136 132 132 ...
##  $ Price      : num  120 83 80 97 128 72 108 120 124 124 ...
##  $ ShelveLoc  : Factor w/ 3 levels "Bad","Good","Medium": 1 2 3 3 1 1 3 2 3 3 ...
##  $ Urban      : Factor w/ 2 levels "No","Yes": 2 2 2 2 2 1 2 2 1 1 ...
##  $ US         : Factor w/ 2 levels "No","Yes": 2 2 2 2 1 2 1 2 1 2 ...
##  $ HighSales  : Factor w/ 2 levels "No","Yes": 2 2 2 1 1 2 1 2 1 1 ...
##  $ Education  : Factor w/ 5 levels "[10,11]","(11,13]",..: 4 1 2 3 2 4 3 1 1 4 ...
##  $ Age        : Factor w/ 5 levels "[25,36]","(36,48.6]",..: 2 4 3 3 2 5 5 4 5 5 ...
##  $ Population : Factor w/ 5 levels "[10,110]","(110,219]",..: 3 3 3 5 4 5 1 5 1 2 ...
##  $ Advertising: Factor w/ 2 levels "[0,5]","(5,29]": 2 2 2 1 1 2 1 2 1 1 ...
##  $ Income     : Factor w/ 5 levels "[21,39]","(39,62]",..: 3 2 1 5 3 5 5 4 5 5 ...
```

Change the order of columns so that it is the same as in the initial (*Carseats*) dataset (note: this step is optional).

```
# update the variable order
carseats.new <- carseats.new[,names(Carseats)]

# print the structure of the carseats.new data frame
str(carseats.new)
```

```
## 'data.frame':    400 obs. of  11 variables:
##  $ CompPrice  : num  138 111 113 117 141 124 115 136 132 132 ...
##  $ Income     : Factor w/ 5 levels "[21,39]","(39,62]",..: 3 2 1 5 3 5 5 4 5 5 ...
##  $ Advertising: Factor w/ 2 levels "[0,5]","(5,29]": 2 2 2 1 1 2 1 2 1 1 ...
##  $ Population : Factor w/ 5 levels "[10,110]","(110,219]",..: 3 3 3 5 4 5 1 5 1 2 ...
##  $ Price      : num  120 83 80 97 128 72 108 120 124 124 ...
##  $ ShelveLoc  : Factor w/ 3 levels "Bad","Good","Medium": 1 2 3 3 1 1 3 2 3 3 ...
##  $ Age        : Factor w/ 5 levels "[25,36]","(36,48.6]",..: 2 4 3 3 2 5 5 4 5 5 ...
##  $ Education  : Factor w/ 5 levels "[10,11]","(11,13]",..: 4 1 2 3 2 4 3 1 1 4 ...
##  $ Urban      : Factor w/ 2 levels "No","Yes": 2 2 2 2 2 1 2 2 1 1 ...
##  $ US         : Factor w/ 2 levels "No","Yes": 2 2 2 2 1 2 1 2 1 2 ...
##  $ HighSales  : Factor w/ 2 levels "No","Yes": 2 2 2 1 1 2 1 2 1 1 ...
```

Now that we have prepared the data, we can proceed to create sets for training and testing.

Create train and test data sets

We'll do this as we did it in the previous Labs.

```
# load the caret package
library(caret)
```

We'll take 80% of observations for the training set and the rest for the test set

```
# set seed
set.seed(1010)

# create train and test sets
train.indices <- createDataPartition(carseats.new$HighSales, p = 0.8, list = FALSE)
train.data <- carseats.new[train.indices,]
test.data <- carseats.new[-train.indices,]
```

Model building

To build an NB model, we will use the **naiveBayes** function from the **e1071** package.

```
# load the e1071 package
library(e1071)

# print the docs for the naiveBayes f.
?naiveBayes
```

First, we'll build a model using all the variables.

```
# build a model with all variables
nb1 <- naiveBayes(HighSales ~ ., data = train.data)
```

We can print the model to see the probabilities that the function computed for each variable.

```
# print the model
print(nb1)
```

```
##
## Naive Bayes Classifier for Discrete Predictors
##
## Call:
## naiveBayes.default(x = X, y = Y, laplace = laplace)
##
## A-priori probabilities:
## Y
##        No       Yes
## 0.7507788 0.2492212
##
## Conditional probabilities:
##      CompPrice
## Y         [,1]     [,2]
##   No  124.7344 14.72654
##   Yes 124.4250 18.07773
##
##      Income
## Y       [21,39]   (39,62]   (62,77] (77,96.2] (96.2,120]
##   No  0.2074689 0.2199170 0.1742739 0.2074689  0.1908714
##   Yes 0.1250000 0.1625000 0.2500000 0.2250000  0.2375000
##
##      Advertising
## Y         [0,5]    (5,29]
##   No  0.5726141 0.4273859
##   Yes 0.3125000 0.6875000
##
##      Population
## Y      [10,110] (110,219] (219,318] (318,412] (412,509]
##   No  0.1908714 0.2489627 0.1742739 0.1950207 0.1908714
##   Yes 0.2125000 0.1375000 0.2375000 0.2000000 0.2125000
##
##      Price
## Y         [,1]     [,2]
##   No  120.0332 21.00077
##   Yes  99.9375 25.29539
```

```
##
##      ShelveLoc
## Y           Bad      Good    Medium
##   No  0.2987552 0.1078838 0.5933610
##   Yes 0.0625000 0.5250000 0.4125000
##
##      Age
## Y       [25,36] (36,48.6] (48.6,60]   (60,70]   (70,80]
##   No  0.1950207 0.1950207 0.1867220 0.1867220 0.2365145
##   Yes 0.2250000 0.2625000 0.2875000 0.1375000 0.0875000
##
##      Education
## Y       [10,11]   (11,13]   (13,15]   (15,17]   (17,18]
##   No  0.2240664 0.2323651 0.1908714 0.2323651 0.1203320
##   Yes 0.3125000 0.2000000 0.1750000 0.2250000 0.0875000
##
##      Urban
## Y           No       Yes
##   No  0.3070539 0.6929461
##   Yes 0.2875000 0.7125000
##
##      US
## Y           No       Yes
##   No  0.4024896 0.5975104
##   Yes 0.2000000 0.8000000
```

Note that the values given for the *CompPrice* and *Price* variables are the mean (1st column) and the standard deviation (2nd column) for the two possible values (No, Yes) of the outcome variable (marked as Y). For all other variables, we have probabilities for each possible value of the variable and each outcome value.

Let's evaluate the model on the test set.

```
# make the predictions with nb1 model over the test dataset
nb1.pred <- predict(nb1, newdata = test.data, type = 'class')

# print several predictions
head(nb1.pred)
```

```
## [1] Yes No  No  Yes Yes No
## Levels: No Yes
```

Then, create the confusion matrix:

```
# create the confusion matrix
nb1.cm <- table(true = test.data$HighSales, predicted = nb1.pred)
nb1.cm
```

```
##      predicted
## true  No Yes
##   No  56   4
##   Yes 10   9
```

We'll use the function for computing the evaluation metrics.

```
# function for computing evaluation metrix
compute.eval.metrics <- function(cmatrix) {
  TP <- cmatrix[1,1] # true positive
  TN <- cmatrix[2,2] # true negative
```

```
  FP <- cmatrix[2,1] # false positive
  FN <- cmatrix[1,2] # false negative
  acc = sum(diag(cmatrix)) / sum(cmatrix)
  precision <- TP / (TP + FP)
  recall <- TP / (TP + FN)
  F1 <- 2*precision*recall / (precision + recall)
  c(accuracy = acc, precision = precision, recall = recall, F1 = F1)
}
```

Compute the evaluation metrics based on the confusion matrix.

```
# compute the evaluation metrics
nb1.eval <- compute.eval.metrics(nb1.cm)
nb1.eval
```

```
##  accuracy precision    recall        F1
## 0.8227848 0.8484848 0.9333333 0.8888889
```

We can try to build a better model by reducing the set of variables to those that are expected to be relevant predictors. For example, we can choose only those variables that proved relevant in the decision tree classifier (Lab #3).

```
# build a model with variables ShelveLoc, Price, Advertising, Age, CompPrice
nb2 <- naiveBayes(HighSales ~ ShelveLoc + Price + Advertising + Age + CompPrice,
                  data = train.data)
```

Make predictions using the new model:

```
# make the predictions with nb2 model over the test dataset
nb2.pred <- predict(nb2, newdata = test.data, type = 'class')
```

Evaluate the new model by first creating the confusion matrix:

```
# create the confusion matrix for nb2 predictions
nb2.cm <- table(true = test.data$HighSales, predicted = nb2.pred)
nb2.cm
```

```
##      predicted
## true  No Yes
##   No  59   1
##   Yes 10   9
```

Then, using the confusion matrix, compute evaluation metrics.

```
# compute the evaluation metrics for the nb2 model
nb2.eval <- compute.eval.metrics(nb2.cm)
nb2.eval
```

```
##  accuracy precision    recall        F1
## 0.8607595 0.8550725 0.9833333 0.9147287
```

These results look better than those we got for the first model; let's compare them.

```
# compare the evaluation metrics for nb1 and nb2
data.frame(rbind(nb1.eval, nb2.eval), row.names = c("NB model 1", "NB model 2"))
```

```
##             accuracy precision    recall        F1
## NB model 1 0.8227848 0.8484848 0.9333333 0.8888889
## NB model 2 0.8607595 0.8550725 0.9833333 0.9147287
```

Obviously, the 2nd model is much better.

TASK: Examine if further improvement of the model can be achieved by doing the discretization differently:

- by splitting the variable values into a smaller or larger number of intervals;
- by using equal-interval instead of equal-frequency discretization method.

ROC curves

ROC curve enables us to choose the probability threshold that will result in the desired *specificity* vs *sensitivity* trade-off.

- **Sensitivity** or **True Positive Rate (TPR)** measures the proportion of positives that are correctly identified as such; it is the same as recall; it is calculated as:

$$TPR = \frac{TP}{TP + FN}$$

- **Specificity** or **True Negative Rate (TNR)** measures the proportion of negatives that are correctly identified as such; it is calculated as:

$$TNR = \frac{TN}{TN + FP}$$

To make use of a ROC curve to find the optimal probability threshold, we need to do prediction differently then we did above: instead of predicting a class value for each observation (using the default 0.5 probability threshold), we should, for each observation, compute the probability for each class value (in our example, 'Yes' and 'No').

```
# compute probabilities for each class value for the observations in the test set
nb2.pred.prob <- predict(nb2, newdata = test.data, type = "raw") # note that the type parameter is now
head(nb2.pred.prob)
```

```
##             No       Yes
## [1,] 0.1483420 0.8516580
## [2,] 0.9703448 0.0296552
## [3,] 0.8073813 0.1926187
## [4,] 0.5569032 0.4430968
## [5,] 0.3655349 0.6344651
## [6,] 0.6029657 0.3970343
```

So, now, for each observation from the test set we have probabilities of the class value 'Yes' and the value 'No'. We will now use a ROC curve to determine which probability threshold to use to classify each store (observation) as having high sales ('Yes') or not ('No').

To create ROC curves, we'll use the **pROC** package (more info about pROC can be found here).

```
#install.packages('pROC')
# load pROC package
library(pROC)
```

We'll use the *roc* function to compute the parameters for the ROC curve.

```
# create a ROC curve
nb2.roc <- roc(response = as.numeric(test.data$HighSales),
               predictor = nb2.pred.prob[,1],
               levels = c(2, 1))
```

Note the following:

- the value of the **predictor** parameter is *nb2.pred.prob[,1]* which means that we are taking the probabilities given in the 1st column of the matrix, that is, the probabilities of the 'No' value of the outcome variable since it is the 'positive class' in the context of this problem (the class we are interested in predicting);
- the value of the **level** parameter is set to c(2,1), to indicate that the level 2 ('Yes') represents the so-called 'controls' and the level 1 ('No') stands for 'cases'. This terminology that *roc()* f. uses (controls and cases) originates from research, where 'cases' are those observations we are interested in; in our terminology, 'cases' are observations of the 'positive' class.

Area Under the Curve (AUC) represents degree separability, it tells us how much model is capable of distinguishing between classes. Higher the AUC, better the model. AUC takes values in the [0,1] range.

We can extract the AUC value from the output of the *roc* function:

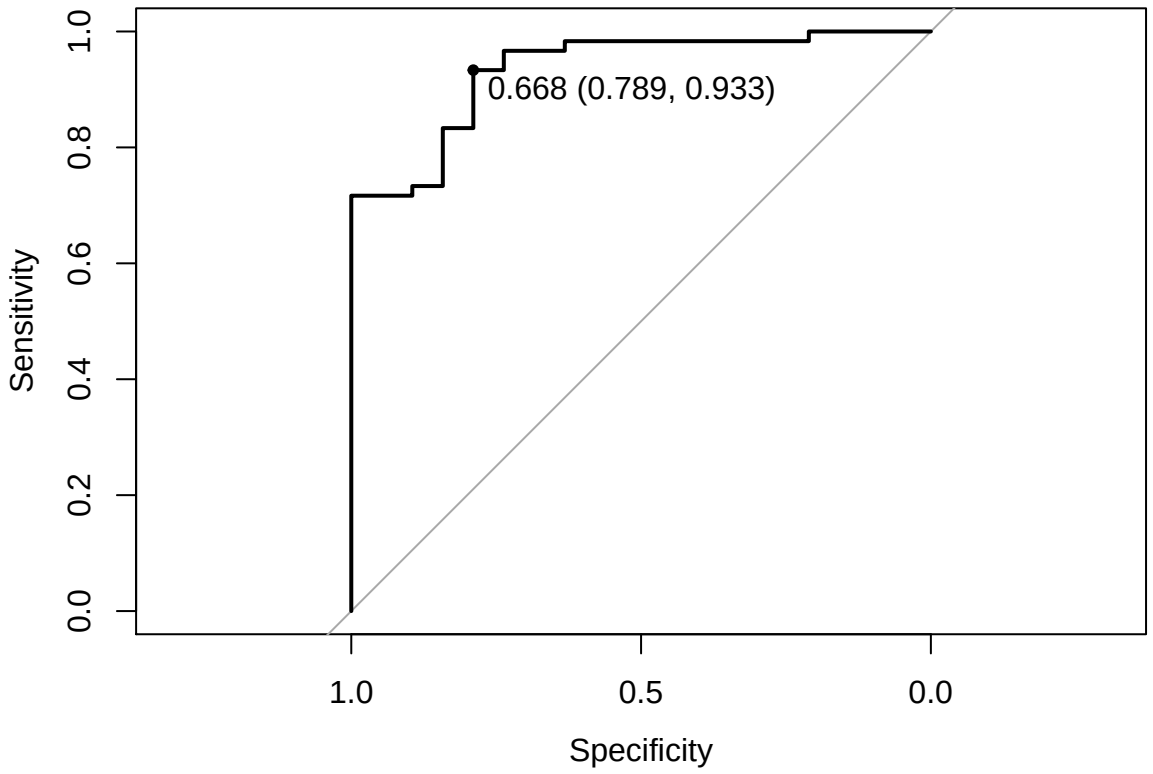
```
# print the Area Under the Curve (AUC) value
nb2.roc$auc
```

```
## Area under the curve: 0.9333
```

A rather high value.

We can draw the ROC curve.

```
# plot the ROC curve
plot.roc(nb2.roc,
         print.thres = TRUE,
         print.thres.best.method = "youden")
```



Note: the plotting f. uses the "*youden*" method for choosing the probability threshold; this method chooses the threshold that maximizes the sum of sensitivity and specificity. Examine the other methods for choosing the threshold in the documentation of the *coords* function.

The plot indicates that if we want to maximize the sum of sensitivity and specificity, we should choose the

probability threshold of 0.668, and that will give us specificity of 0.933, and sensitivity of 0.789.

Instead of using the plot to get the threshold value, we can use the *coords* function (also from the *pROC* package). This function has numerous parameters; for us, the following two are particularly important:

- **x** - the coordinates to look for; we'll use "*local maximas*", meaning that we want to examine the model performance on the points (threshold values) that represent the local maxima of the ROC curve; check the documentation for the other values;
- **ret** - the classification metrics the method should return; we will take accuracy, specificity, sensitivity, and threshold; check the documentation for the other values.

```
# get the coordinates for all local maximas
nb2.coords <- coords(nb2.roc,
                     ret = c("accuracy", "spec", "sens", "thr"),
                     x = "local maximas")
nb2.coords
```

```
##             local maximas local maximas local maximas local maximas
## accuracy        0.8101266     0.8987342     0.9113924     0.8987342
## specificity     0.2105263     0.6315789     0.7368421     0.7894737
## sensitivity     1.0000000     0.9833333     0.9666667     0.9333333
## threshold       0.3520004     0.5493808     0.6035692     0.6679674
##             local maximas local maximas local maximas
## accuracy        0.8354430     0.7721519     0.7848101
## specificity     0.8421053     0.8947368     1.0000000
## sensitivity     0.8333333     0.7333333     0.7166667
## threshold       0.7859801     0.8110555     0.8255424
```

From the above plot, we can see that the ROC curve has 7 local maxima, and we have obtained the metrics for each of them. We can now choose the threshold that will give us the best result in terms of accuracy, sensitivity (recall or true positive rate) or specificity (true negative rate).

For example, in our case, specificity represents the proportion of high sales that we have recognized as such (high sales). So, if we want to maximize specificity, that is, to reduce the number of stores that were falsely classified as not having high sales, we can choose the threshold of 0.78598, as it will give us the specificity of 0.8421, while still providing us with decent sensitivity (0.8333) and accuracy (0.8354). Let's make predictions based on this threshold (0.78598):

```
# choose a threshold of 0.7859801
prob.threshold <- nb2.coords[4,5]
```

Determine the class label (Yes, No) using the predicted probabilities (nb2.pred.prob) and the chosen threshold (prob.threshold):

```
# create predictions based on the new threshold
nb2.pred2 <- ifelse(test = nb2.pred.prob[,1] >= prob.threshold, # if probability of the positive class
                    yes = "No", #... assign the positive class (No)
                    no = "Yes") #... assign the negative class (Yes)
nb2.pred2 <- as.factor(nb2.pred2)
```

Create the confusion matrix and compute the evaluation metrics:

```
# create the confusion matrix for the new predictions
nb2.cm2 <- table(actual = test.data$HighSales, predicted = nb2.pred2)
nb2.cm2
```

```
##       predicted
## actual No Yes
##    No  50  10
##    Yes  3  16
```

Compute the evaluation metrics for the new predictions.

```
# compute the evaluation metrics
nb2.eval2 <- compute.eval.metrics(nb2.cm2)
nb2.eval2
```

```
##  accuracy precision    recall        F1
## 0.8354430 0.9433962 0.8333333 0.8849558
```

Note that the accuracy and sensitivity (recall) are the same as in thecorresponding column (column 5 that corresponds to the chosen threshold) in the nb2.coords matrix.

Compare the results of the 3 models:

```
# compare the evaluation metrics for all three models
data.frame(rbind(nb1.eval, nb2.eval, nb2.eval2),
           row.names = c(paste("NB_", 1:3, sep = "")))
```

```
##       accuracy precision    recall        F1
## NB_1 0.8227848 0.8484848 0.9333333 0.8888889
## NB_2 0.8607595 0.8550725 0.9833333 0.9147287
## NB_3 0.8354430 0.9433962 0.8333333 0.8849558
```