Naive Bayes Classifier

Short Intro to Naive Bayes

Naive Bayes is a probabilistic machine learning algorithm used for the classification task. It is based on the *Bayes theorem* that, in the context of the classification task, can be expressed as follows:

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

Where:

- c represents the class,
- X represents a vector of features (attributes) used for predicting the class,
- *P(c|X)* is the *posterior probability* of class c given the feature vector X.
- *P(c)* is the *prior probability* of class c.
- *P(X|c)* is the conditional probability of feature values X given the class c.
- *P(X)* is the probability of features X.

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

Naive Bayes relies on the assumption that features (attributes) are independent. That is, values of one particular feature are not associated / correlated with values of any other feature. This is in most cases an unrealistic assumption - hence the name of the algorithm *naive*.

Naive Bayes Example

Suppose we have a dataset about purchases made by 300 people. The dependent variable is *Buy* (whether they bought the product or not). Independent variables are *Visited other stores* and *Came alone*.

Sample of the data looks as follows:

	Visited other stores	Alone	Buy
1	TRUE	TRUE	TRUE
2	TRUE	FALSE	FALSE
3	FALSE	FALSE	FALSE
4	TRUE	FALSE	FALSE

Based on the original dataset, we create a frequency table (below) for each combination of values of independent and dependent variables. For example, there were 90 people who visited other stores and did not make a purchase and 40 persons who came alone and made a purchase.

Buy (class)	Visited other stores		Alone		Total
	TRUE	FALSE	TRUE	FALSE	
FALSE	90	10	25	75	**100**
TRUE	75	125	40	160	**200**
Total	**165**	**135**	**65**	**235**	**300**

Recall the Bayes theorem:

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

The variable **c** is the class variable (Buy), while variable **X** represents the features. Since we have two features, **X** is given as:

$$X = (x_{visited}, x_{alone})$$

By substituting for **X** and expanding the formula using the chain rule, we get:

$$P(c|x_{visited}, x_{alone}) = \frac{P(x_{visited}|c) * P(x_{alone}|c) * P(c)}{P(x_{visited}) * P(x_{alone})}$$

We will use our frequency table (given above) to do the computations. To that end, let's convert the frequency table into a table of probabilities.

First, we will compute *prior probability* for each class:

Prior probability of **Won't buy** class: $\frac{100}{100+200} = 0.333$

Prior probability of **Buy** class: $\frac{200}{100+200} = 0.667$

Next, we compute conditional probabilities - first for *Visiting other stores*:

Conditional probability of **Visiting other stores** given the **Won't buy** class: $\frac{90}{100} = 0.9$

Conditional probability of **Visiting other stores** given the **Buy** class: $\frac{75}{200} = 0.375$

Similar computations should be done for *Came Alone*.

As the result, we get the following table of probabilities:

Buy	Visited other stores		Alone		Total
	TRUE	FALSE	TRUE	FALSE	
FALSE	0.900	0.100	0.250	0.750	**0.333**
TRUE	0.375	0.625	0.200	0.800	**0.667**
Total	**0.550**	**0.450**	**0.217**	**0.783**	

Now that we have the table with (prior and conditional) probabilities, we can predict for a new person who, for example, **Visited other stores** (Visited other stores = TRUE) and came **Alone** (Alone = TRUE) whether they will **Buy** or **Won't Buy** a product.

Calculating the probability of Won't Buy

$$P(Won'tbuy|Visited, Alone) = \frac{P(Visited|Won'tbuy) * P(Alone|Won'tbuy) * P(Won'tbuy)}{P(Visited) * P(Alone)}$$

$$P(Won'tbuy|Visited, Alone) = \frac{0.9 * 0.25 * 0.33}{0.55 * 0.217} = \frac{0.07425}{0.11935} = 0.142$$

Calculating the probability of Buy

$$P(Buy|Visited, Alone) = \frac{P(Visited|Buy) * P(Alone|Buy) * P(Buy)}{P(Visited) * P(Alone)}$$

$$P(Buy|Visited, Alone) = \frac{0.9 * 0.25 * 0.333}{0.55 * 0.217} = \frac{0.074925}{0.11935} = 0.627$$

The final posterior probabilities can be standardized between 0 and 1:

$$P(Won't buy|Visited, Alone) = \frac{0.142}{0.627 + 0.142} = 0.18$$

$$P(Buy|Visited, Alone) = \frac{0.627}{0.627 + 0.142} = 0.82$$

Since $P(Buy|Visited, Alone)$ is higher than 0.5 (the default threshold), the classifier will classify the new customer, who **Visited other stores** and came **Alone**, with the class **Buy**.

Prepare the data set

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

```
# load ISLR package
library(ISLR)

# examine the 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). Sales greater than the 3rd quartile qualify as high sales.

```
# calculate the 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 - since it was used for the creation of the outcome variable, it should not be used for prediction.

```
# remove the Sales variable
Carseats$Sales <- NULL
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 ...
```

Discretization on numeric variables

NB is often used with nominal (categorical) variables. If a numeric variable is to be used, it has to be:

- represented as probabilities based on the appropriate probability distribution (typically Normal distribution), or
- discretized, that is, have its 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.

```
# select 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 value of an attribute is split into k intervals of (roughly) equal lengths; k needs to be specified; this method is known as *interval discretization*
- **equal frequency intervals** - the range of value of an attribute is split into k intervals so that each interval has (roughly) equal number of values; k needs to be set; this method is known as *quantile discretization*

In general, the 2nd option tends to give better results.

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:

```
# open 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 intervals of equal frequency:

```
# select 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))
```

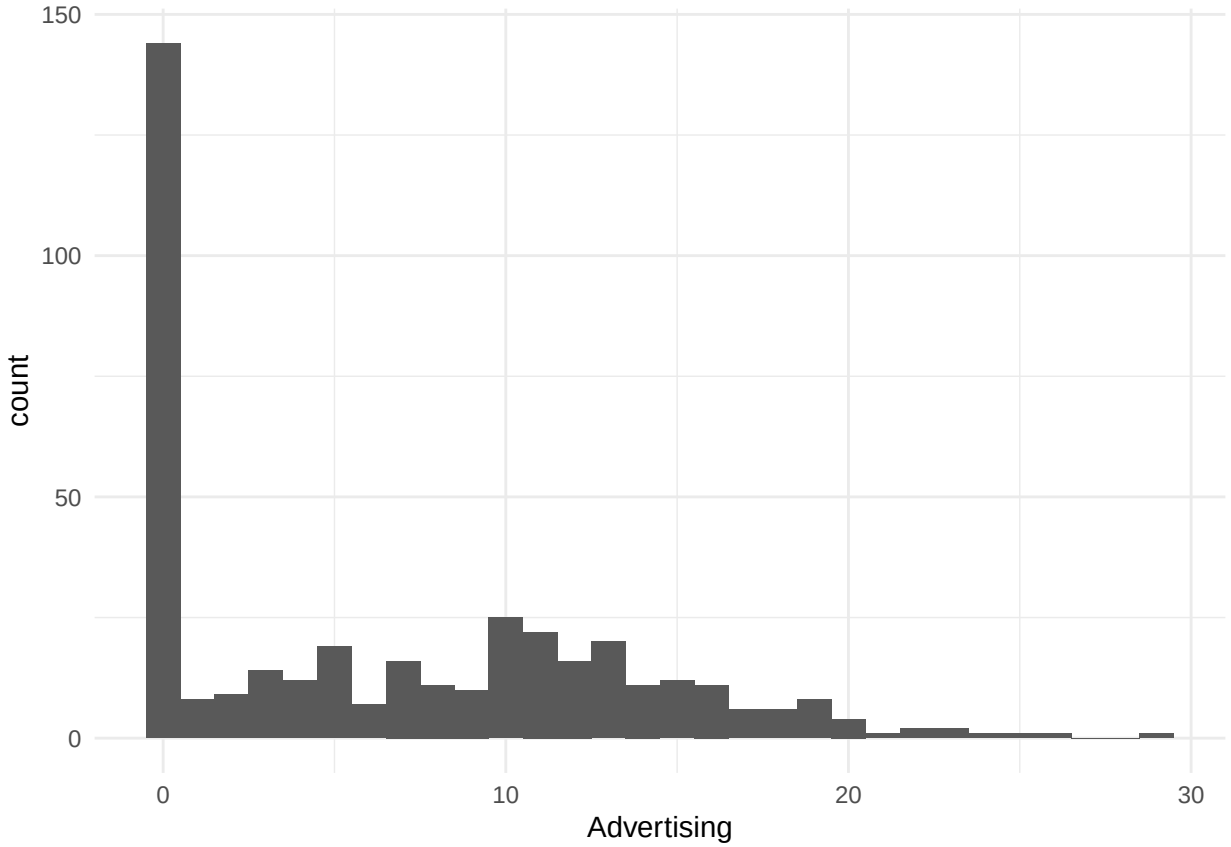
Let's check the summary of the discretized variables

```
summary(discretized)
```

```
##    Education         Age             Population   Advertising         Income
##  [10,11]:96   [25,36]  :82   [10,110.4]   :80   [0,0]   :144   [21,39]   :82
##  (11,13]:92   (36,48.6]:78   (110.4,218.6]:80   (0,2]   : 17   (39,62]   :82
##  (13,15]:76   (48.6,60]:85   (218.6,317.8]:80   (2,8.4] : 79   (62,77]   :77
##  (15,17]:96   (60,70]  :78   (317.8,412.2]:80   (8.4,13]: 93   (77,96.2] :79
##  (17,18]:40   (70,80]  :77   (412.2,509]  :80   (13,29] : 67   (96.2,120]:80
```

Notice that the Advertising variable could not be split into 5 segments of relatively equal frequencies; instead frequencies differ a lot across segments. To understand why this is the case, we need to examine the distribution of the Advertising variable.

```
# plot the histogram for the Advertising variable
library(ggplot2)
ggplot(data = Carseats, mapping = aes(x = Advertising)) +
  geom_histogram(bins = 30) +
  theme_minimal()
```



The plot shows a very uneven (skewed) distribution of this variable, which is the reason why we could not split it as we intended to.

So, instead of 5, we'll split the *Advertising* variable into 3 equal frequency intervals:

```
# discretize all variables into 5 bins each, but the Advertising variable into 3 bins
discretized <- discretize(data = Carseats[,to.discretize],
                          method = 'quantile',
                          breaks = c(5,5,5,3,5))
```

Examine again the summary statistics of the discretized variables

```
summary(discretized)
```

```
##    Education         Age             Population  Advertising         Income
##  [10,11]:96   [25,36]  :82   [10,110.4]   :80   [0,0]  :144   [21,39]   :82
##  (11,13]:92   (36,48.6]:78   (110.4,218.6]:80   (0,10] :131   (39,62]   :82
##  (13,15]:76   (48.6,60]:85   (218.6,317.8]:80   (10,29]:125   (62,77]   :77
##  (15,17]:96   (60,70]  :78   (317.8,412.2]:80                 (77,96.2] :79
##  (17,18]:40   (70,80]  :77   (412.2,509]  :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 <- 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 "[10,110.4]","(110.4,218.6]",..: 3 3 3 5 4 5 1 5 1 2 ...
##  $ Advertising: Factor w/ 3 levels "[0,0]","(0,10]",..: 3 3 2 2 2 3 1 3 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/ 3 levels "[0,0]","(0,10]",..: 3 3 2 2 2 3 1 3 1 1 ...
##  $ Population : Factor w/ 5 levels "[10,110.4]","(110.4,218.6]",..: 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(2421)

# 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)

# open 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  125.2739 14.83969
##   Yes 124.7375 17.27129
##
##      Income
## Y       [21,39]   (39,62]   (62,77] (77,96.2] (96.2,120]
##   No  0.2365145 0.2074689 0.1701245 0.1991701  0.1867220
##   Yes 0.1250000 0.1625000 0.2500000 0.2500000  0.2125000
```

```
##
##      Advertising
## Y         [0,0]    (0,10]   (10,29]
##   No  0.4315353 0.3278008 0.2406639
##   Yes 0.2125000 0.3625000 0.4250000
##
##      Population
## Y     [10,110.4] (110.4,218.6] (218.6,317.8] (317.8,412.2] (412.2,509]
##   No   0.1908714     0.2282158     0.1991701     0.1784232   0.2033195
##   Yes  0.1875000     0.1375000     0.2375000     0.2375000   0.2000000
##
##      Price
## Y         [,1]     [,2]
##   No  121.0290 20.90761
##   Yes 101.8625 24.27362
##
##      ShelveLoc
## Y            Bad       Good     Medium
##   No  0.31950207 0.09958506 0.58091286
##   Yes 0.06250000 0.56250000 0.37500000
##
##      Age
## Y       [25,36] (36,48.6] (48.6,60]   (60,70]   (70,80]
##   No  0.2074689 0.1825726 0.1742739 0.2157676 0.2199170
##   Yes 0.2625000 0.2500000 0.2625000 0.1375000 0.0875000
##
##      Education
## Y       [10,11]   (11,13]   (13,15]   (15,17]   (17,18]
##   No  0.2157676 0.2323651 0.1784232 0.2655602 0.1078838
##   Yes 0.2875000 0.1875000 0.1750000 0.2625000 0.0875000
##
##      Urban
## Y            No       Yes
##   No  0.2863071 0.7136929
##   Yes 0.3250000 0.6750000
##
##      US
## Y            No       Yes
##   No  0.4273859 0.5726141
##   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 combination of values of the predictor and outcome variables.

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] No No No No No 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  8  11
```

We'll use the function for computing the evaluation metrics. Recall (from the previous Labs) that we set No as the positive class.

```
# function for computing evaluation metrics
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
## 0.8481013 0.8750000 0.9333333 0.9032258
```

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 #4).

```
# 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  57   3
##   Yes  8  11
```

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 0.8769231 0.9500000 0.9120000
```

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_1", "NB_2"))
```

```
##       accuracy precision    recall        F1
## NB_1 0.8481013 0.8750000 0.9333333 0.9032258
## NB_2 0.8607595 0.8769231 0.9500000 0.9120000
```

Obviously, the 2nd model is 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 predicted as such; it is the same as recall; it is calculated as:

$$Sensitivity/TPR/Recall = \frac{TP}{TP + FN}$$

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

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

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,] 0.9618947 0.03810527
## [2,] 0.9484467 0.05155333
## [3,] 0.8842262 0.11577377
## [4,] 0.8920884 0.10791164
## [5,] 0.9654819 0.03451810
## [6,] 0.9136761 0.08632389
```

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 a ROC curve, 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))
```

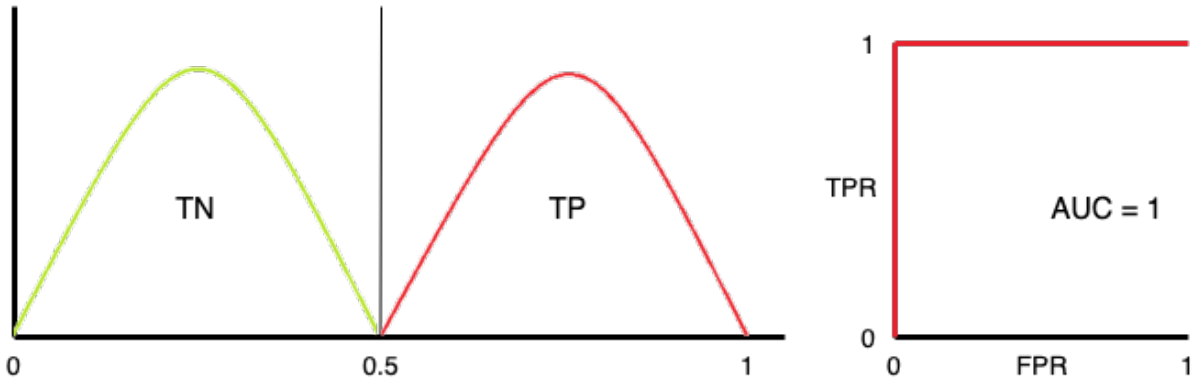
```
## Setting direction: controls < cases
```

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' stand for the 'positive' class, whereas 'controls' stand for the 'negative' class.

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

AUC = 1

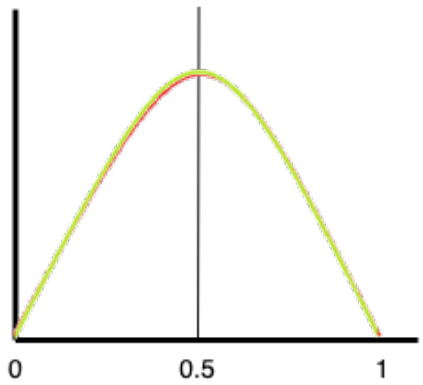


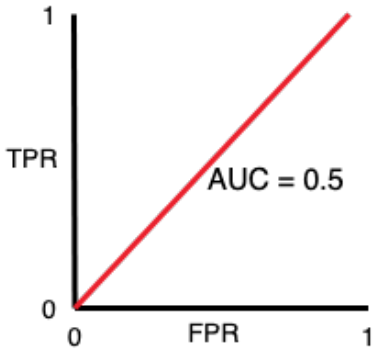
This is an ideal situation - the two curves do not overlap at all, indicating that the model has an ideal measure of separability. It is perfectly able to distinguish between the positive class and the negative class.

AUC = 0.7

When two distributions overlap, we introduce type 1 and type 2 error (FP and FN, respectively). Depending upon the threshold, we can minimize one or the type of error. When AUC is 0.7, it means there is a 70% chance that the model will be able to distinguish between the positive class and the negative class.

AUC = 0.5





This is the worst situation. When AUC is approximately 0.5, the model has no discrimination capacity, that is, it is not able to distinguish between the positive class and the negative class.

In our example, we can extract the AUC value from the output of the *roc* function:

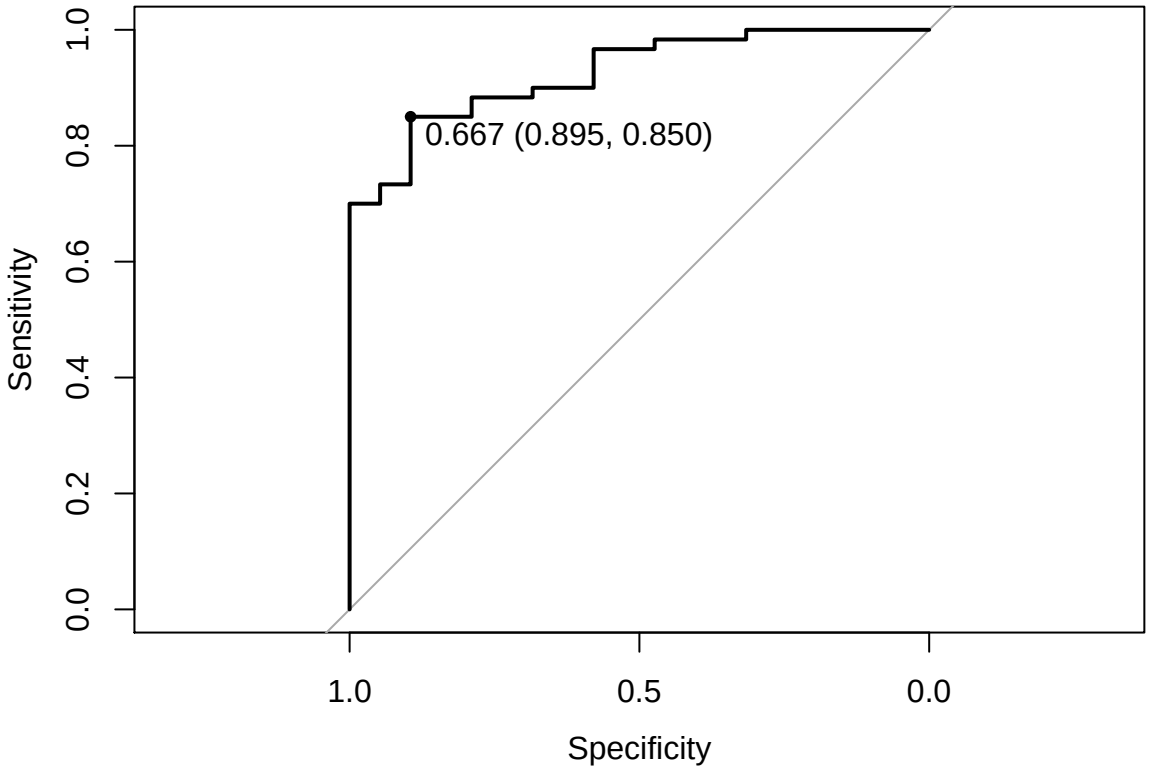
```
# print the Area Under the Curve (AUC) value
nb2.roc$auc
```

```
## Area under the curve: 0.9254
```

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 finds 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 0.779, and that will give us specificity of 0.842, and sensitivity of 0.850.

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
```

```
##    accuracy specificity sensitivity threshold
## 1 0.8354430   0.3157895   1.0000000 0.2676606
## 2 0.8607595   0.4736842   0.9833333 0.4464391
## 3 0.8734177   0.5789474   0.9666667 0.4820252
## 4 0.8481013   0.6842105   0.9000000 0.5739582
```

```
## 5 0.8607595   0.7894737   0.8833333 0.6205684
## 6 0.8607595   0.8947368   0.8500000 0.6671026
## 7 0.7848101   0.9473684   0.7333333 0.7816594
## 8 0.7721519   1.0000000   0.7000000 0.8104750
```

From the above plot, we can see that the ROC curve has 9 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, sensitivity represents the proportion of low sales that we have recognized as such (as being low). So, if we want to maximize sensitivity, that is, to reduce the number of stores that were falsely classified as having high sales, we can choose the threshold of 0.4820, as it will give us the sensitivity of 0.9667, while maximizing accuracy (0.8734) and still providing us with a low, but still decent value for the specificity (0.5789). Let's make predictions based on the chosen threshold:

```
# choose a threshold that maximizes sensitivity while keep decent values of other metrics
prob.threshold <- nb2.coords[3,4]
```

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") #... otherwise, 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  58   2
##    Yes  8  11
```

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
## 0.8734177 0.8787879 0.9666667 0.9206349
```

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

Compare the results of the 3 models:

```r
# 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 0.8481013 0.8750000 0.9333333 0.9032258
## NB_2 0.8607595 0.8769231 0.9500000 0.9120000
## NB_3 0.8734177 0.8787879 0.9666667 0.9206349
```