Chapter 7

Classifiers for Mixture Models

In this chapter we review various kinds of classifiers developed as `discriminators' on datasets that can be described as mixtures of multidimensional Gaussians.

7.1 Gaussian Mixtures

7.1.1 Multi-dimensional Gaussians

Covariance matrices have many nice properties. An important property is that *covariance matrices are (symmetric) nonnegative definite*. Assuming the data values are all real, covariance matrices are symmetric matrices whose eigenvalues are all nonnegative.
Essentially, covariance matrices are ellipsoids in p-dimensional space. Their eigenvectors determine the axes of the ellipsoid, and their eigenvalues determine the spread of the ellipsoid along these axes.
We can define a p-dimensional Gaussian function in the following way. Let $\boldsymbol{x}$ be a p-dimensional value, μ be a p-dimensional vector of means, and Σ be a positive definite $p \times p$ covariance matrix. Then

$$g(\boldsymbol{x}, \boldsymbol{\mu}, \Sigma) \;=\; \frac{1}{(2\,\pi)^{p/2}}\;\frac{1}{\sqrt{\det\,\Sigma}}\;\exp\left(-\frac{1}{2}\;(\boldsymbol{x}-\boldsymbol{\mu})'\;\Sigma^{-1}\;(\boldsymbol{x}-\boldsymbol{\mu})\right).$$

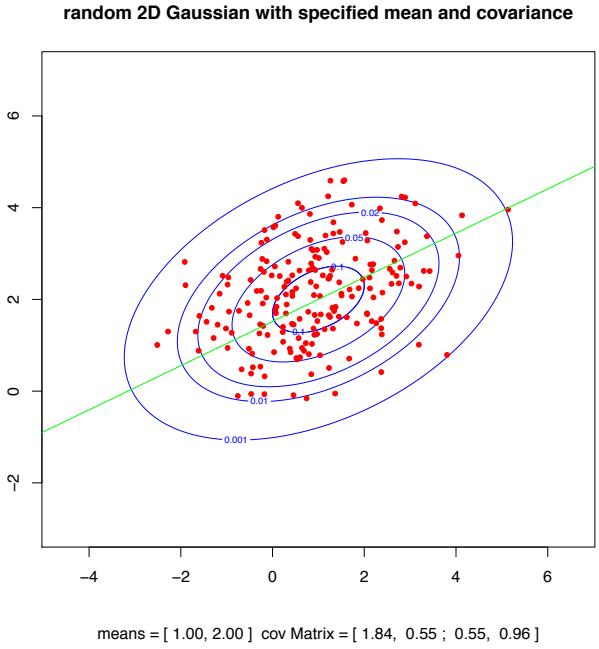
Here we require Σ to be positive definite so that its determinant is positive and the square root is defined. If we let $W = \Sigma^{-1}$ be the inverse of the covariance matrix, we get the more common formula

$$g(\boldsymbol{x}, \boldsymbol{\mu}, W^{-1}) \;=\; \frac{1}{(2\,\pi)^{p/2}}\;\sqrt{\det\,W}\;\exp\left(-\frac{1}{2}\;(\boldsymbol{x}-\boldsymbol{\mu})'\;W\;(\boldsymbol{x}-\boldsymbol{\mu})\right).$$

A contour plot of a 2-dimensional Gaussian (i.e., $p = 2$) using this definition is shown in Figure 7.2.

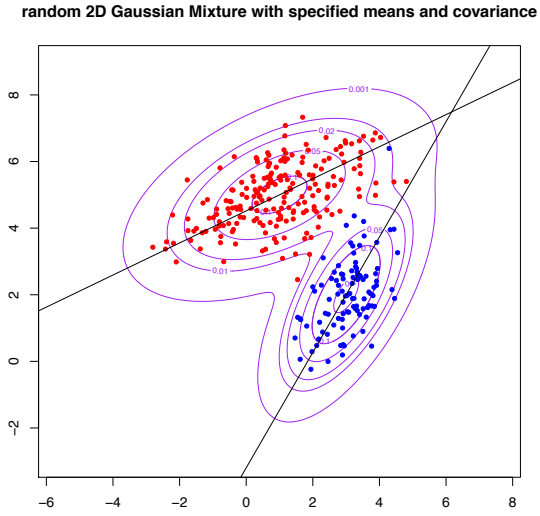
7.1.2 Gaussian Mixtures, and how they arise

A **mixture model** is a two-level random process that first randomly selects among a number m of distributions, each with some fixed probability p_i ($1 \leq i \leq m$) of selection (so that $\sum_i p_i = 1$), and then yields a point at random from this distribution. Each of the distributions can be parametrized differently, so that




```
C = matrix(c( 1.33, 0.24, 0.24, 0.95 ), nr=2, nc=2)

n = 100

Z = matrix( rnorm(2*n), n, 2)   #  Z is a sample of a 2D spherical Gaussian centered at (0,0).
X = Z %*% C                     #  The product  X = Z %*% C  has nontrivial covariance.

covX = cov(X)
inv = function(A) solve(A)
W = inv(covX)

lambda = eigen(W)$values
lambda1 = lambda[1]
lambda2 = lambda[2]
detW = lambda1 * lambda2
info = sprintf( "lambda = [%5.2f,%5.2f ]   cov(X) = [%5.2f, %5.2f ; %5.2f, %5.2f ]   W = [%5.2f, %5.2f ; %5.2f, %5.2f ]",
        lambda1,lambda2, covX[1,1],covX[1,2],covX[2,1],covX[2,2], W[1,1],W[1,2],W[2,1],W[2,2])

xbar = apply( X, 2, mean )      #  xbar = the column means of X

N = 100  #  100 x 100 grid
x1axis = seq( -4, 4, len=N )
x2axis = seq( -4, 4, len=N )

Z = matrix(0, N,N)
for (i in 1:N) {
   for (j in 1:N) {
     x = rbind( x1axis[i], x2axis[j] )
     Z[i,j] = 1/sqrt((2*pi)^2) * det(W) * exp(-1/2 * t(x-xbar) %*% W %*% (x-xbar) )
   }
}
contour( x=x1axis, y=x2axis, Z, col="blue", asp=1.0, sub=info, cex.sub=0.75, main="2D Gaussian" )    # plot the 2D Gaussian
```

Figure 7.1: A random 2-dimensional Gaussian, with the R code that was used to generate it. The points shown have been generated using a prespecified mean at $(1, 2)$ and a prespecified covariance matrix with a major axis at an angle of $\pi/7$ and the axis lengths (dilation factors) of 1.9 and 0.7.

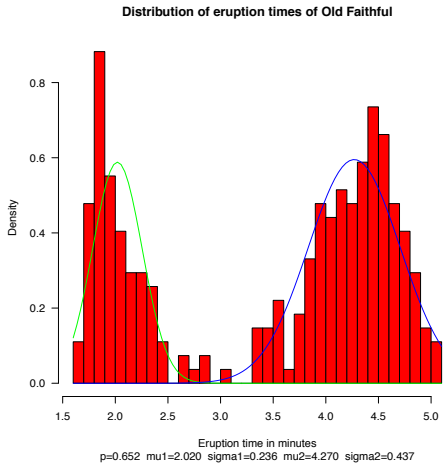



```
covMatrix = function (theta, d1, d2) {
   cos_t = cos(theta); sin_t = sin(theta)
   Rtheta = matrix(c( cos_t, -sin_t, sin_t, cos_t ), nr=2, nc=2)
   Dilation  = matrix(c( d1, 0, 0, d2 ), nr=2, nc=2)
   t(Rtheta) %*% Dilation %*% Rtheta
}
randomGaussianData = function(npoints, means, covMatrix) {
   #  generate a set of Gaussian random points with a specified covariance matrix and mean
   covEigen = eigen(covMatrix)
   transformMatrix = covEigen$vectors %*% diag( sqrt(covEigen$values) ) %*% t(covEigen$vectors)
  matrix( means, npoints, 2, byrow=TRUE ) + matrix( rnorm(2*npoints), npoints, 2 )  %*%  transformMatrix
}
S1 = covMatrix( pi/7, 2.7, 0.7 )
S2 = covMatrix( pi/3, 1.7, 0.3 )
xbar1 = c( 1, 5 )          # mean of Class 1
xbar2 = c( 3, 2 )          # mean of Class 2
inv = function(A) solve(A)
W1 = inv(S1)
W2 = inv(S2)
p = 1/3            # probability of choosing the second Gaussian
n = 300            # total number of points sampled
n1 = n * (1 - p)   # number of points sampled from the first Gaussian
n2 = n * p         # number of points sampled from the second Gaussian

Class1Points = randomGaussianData( n1, xbar1, S1 )
Class2Points = randomGaussianData( n2, xbar2, S2 )
X = rbind( cbind(Class1Points, 1), cbind(Class2Points, 2) )

N = 200; x1axis = seq( -5, 7, len=N );  x2axis = seq( -3, 9, len=N )
Z = matrix(0, N,N)
for (i in 1:N) {
   for (j in 1:N) {
     x = rbind( x1axis[i], x2axis[j] )
     Z[i,j] = 1/sqrt(2*pi)^2 * sqrt(det(W1)) * exp(-1/2 * t(x-xbar1) %*% W1 %*% (x-xbar1) ) +
              1/sqrt(2*pi)^2 * sqrt(det(W2)) * exp(-1/2 * t(x-xbar2) %*% W2 %*% (x-xbar2) )
   }
}
contour( x=x1axis, y=x2axis, Z, col="purple", levels = c(0.3, 0.2, 0.1, 0.5, 0.1, 0.05, 0.02, 0.01, 0.001 ),
         main="random 2D Gaussian Mixture with specified means and covariance", asp=1.0 )
points( X[,1], X[,2], col=c("red","blue")[X[,3]], pch=20 )
slope1 = tan(pi/7); intercept1 = xbar1[2] - xbar1[1] * slope1
slope2 = tan(pi/3); intercept2 = xbar2[2] - xbar2[1] * slope2
abline( intercept1, slope1, col="black" )   # add axis at slope pi/7
abline( intercept2, slope2, col="black" )   # add axis at slope pi/3
```

Figure 7.2: Mixture of Gaussian Mixture, obtained by sampling points from two pre-specified distributions; the upper distribution was drawn with probability 2/3.

7.1.3 Old (Statistically) Faithful

A classic example of a mixture model is a set of geyser eruption times (in minutes), and waiting times between successive eruptions, produced by *Old Faithful*. Plotting the histogram of eruption times shows a bimodal distribution, using the first column of the dataset shown in Figure 7.3.



eruption time (min.)	waiting time (min.)
3.600	79
1.800	54
3.333	74
2.283	62
4.533	85
2.883	55
⋮	⋮

Figure 7.3: Histogram of eruption times for Old Faithful, with superimposed maximum likelihood model.

To the distribution of these points, we can try to fit a one-dimensional mixture model

$$\Pr[\text{ eruption time} = x \mid \boldsymbol{\theta}\,] \;=\; (1-p)\,g(x,\mu_1,\sigma_1) \;+\; p\,g(x,\mu_2,\sigma_2).$$

This is a model with five parameters: p, μ_1, σ_1, μ_2, σ_2. If we define a parameter vector

$$\boldsymbol{\theta} \;=\; (p,\ \mu_1,\ \sigma_1,\ \mu_2,\ \sigma_2)$$

then the goal is to find the value of $\boldsymbol{\theta}$ that best fits the data.

The likelihood$[\,H \mid D\,]$ is proportional to $\Pr[\,D \mid H\,]$, but where $\Pr[\,D \mid H\,]$ takes the hypothesis H as given and D as a variable, likelihood$[\,H \mid D\,]$ takes D as given and H as a variable [1]. We say that that D *supports* one hypothesis over another if the likelihood of one given D exceeds that for the other.

Undoubtedly there are many ways to find such a value, but one way is to maximize the **likelihood** of the parameters being the right ones, given the data. If the set of eruption times is $D = \{\, t_i \mid i = 1,\ldots,n \,\}$, then the

$$\text{likelihood}[\,\boldsymbol{\theta} \mid D\,] \;=\; \Pr[\,D \mid \boldsymbol{\theta}\,] \;=\; \prod_{i=1}^{n} \Pr[\text{ eruption time} = t_i \mid \boldsymbol{\theta}\,]$$

and thus the log-likelihood to be maximized is

$$\log \text{ likelihood}[\,\boldsymbol{\theta} \mid D\,] \;=\; \sum_{i=1}^{n} \log\big(\;(1-p)\,g(t_i,\mu_1,\sigma_1) \;+\; p\,g(t_i,\mu_2,\sigma_2)\;\big)\,.$$

Numerical maximization of this likelihood requires some work. However we can first simplify it by introducing a set of variables r_i taking values in $[0,1]$ and try to maximize the similar expression

$$\sum_{i=1}^{n} \log\big(\,(1-p_i)\,g(x_i \mid \theta_1) \;+\; p_i\,g(x_i \mid \theta_2)\big)$$

Each p_i is the "responsibility" that the second gaussian has for x_i: if $p_i = 0$, then x_i was obtained from the first gaussian; if $p_i = 1$, then x_i was obtained from the second gaussian. More generally, we define an expected responsibility value r_i in $[0, 1]$

$$r_i(\theta) = E[p_i \mid \theta, x] = Pr[p_i = 1 \mid \theta, x]$$

so that r_i is our expected value for p_i. With this, we can solve for r_i and θ by repeating the following two steps iteratively:

1. **Expectation Step:**
   Compute responsibilities of each observation x_i between the 2 gaussians: using our current values for $\theta = (p, \theta_1, \theta_2)$:

$$r_i \;=\; p\, g(x_i \mid \theta_2) \;/\; ((1-p)g(x_i \mid \theta_1) \;+\; pg(x_i \mid \theta_2))$$

2. **Maximization Step:**
   Update the weighted means and variances accordingly:

$$\begin{aligned}
p &= \sum_i r_i/N \\
\mu_1 &= \sum_i (1-r_i)x_i / \sum_i (1-r_i) \\
\mu_2 &= \sum_i r_i x_i / \sum_i r_i \\
\sigma_1 &= \sum_i (1-r_i)(x_i-\mu_1)^2 / \sum_i (1-r_i) \\
\sigma_2 &= \sum_i r_i (x_i-\mu_2)^2 / \sum_i r_i.
\end{aligned}$$

In other words we alternate between two optimization problems:

1. Estimating expectations r
2. Finding optimal θ given r

This does not necessarily obtain a solution that maximizes likelihood; this EM iteration is "greedy" and not particularly clever. However, this approach generalizes naturally for more than 2 gaussians. (See, e.g., [1, Section 8.5]).

```
data(faithful)
x = faithful$eruptions
#  random initial values:
mu1 = sample(x,1)
mu2 = sample(x,1)
sigma1 = 1.0
sigma2 = 1.0
p = 0.5

iterations = 100
N = length(x)
for(i in 1:iterations) {
  cat("iteration ", i, ":\n")
  cat("p = ", p, "\n")
  cat("mu1 = ", mu1, "\n")
  cat("mu2 = ", mu2, "\n")
  cat("sigma1 = ", sigma1, "\n")
  cat("sigma2 = ", sigma2, "\n")
```

```
  f1 = (1 - p) * dnorm( x, mean=mu1, sd=sigma1 )
  f2 =      p  * dnorm( x, mean=mu2, sd=sigma2 )
  r  = (f2 / (f1 + f2))   #  r is a vector of "responsibility values" r_i

  #  weighted means:
  mu1 = sum( (1-r) * x ) / sum(1-r)
  mu2 = sum(    r  * x ) / sum(  r)

  #  weighted variances:
  sigma1 = sqrt(sum( (1-r) * (x - mu1)^2 ) / sum(1-r))
  sigma2 = sqrt(sum(   r   * (x - mu2)^2 ) / sum( r ))

  #  next estimate of p:
  p = sum(r) / N
}
```

This iteration (slowly) to the following solution:

```
  iteration  100 :
  p =  0.6515954
  mu1 =  2.018608
  mu2 =  4.273343
  sigma1 =  0.2356218
  sigma2 =  0.4370631
```

As shown in Figure 7.3, this is a decent fit of a gaussian mixture model to the data. If we are allowed only two gaussians in our mixture, this should be optimal.

7.2 Classification Models

A **classification** of a vector of features $\boldsymbol{x}$ is a value y in a finite set C of **classes**.
Given a training set of feature vectors $\boldsymbol{x}$ and their classes y, the **classification problem** is to identify a classifier function f such that $y = f(\boldsymbol{x})$.
The classification problem is often formalized this way: given a $n \times p$ matrix/dataset X containing n observations of $1 \times p$ feature vectors, and a $n \times 1$ vector $\boldsymbol{y}$ of classifications for these vectors, we seek a classification problem that is as `accurate' as possible. Often this accuracy requirement is formalized as finding a **classification model** $y = \widehat{f}(\boldsymbol{x})$, where accuracy is defined in terms of a **loss function**

$$L(\boldsymbol{y}, \widehat{f}(X)) \;=\; \left|\left|\boldsymbol{y} - \widehat{f}(X)\right|\right|.$$

Minimizing this loss is an optimization problem. The importance of finding a model that gives the `best fit' to input data, in this sense, has made a strong connection between data mining and optimization.

7.2.1 The Two-Class Discriminant Problem

The **two-class** classification problem is important because it can be successfully solvable in ways that **multi-class** problems are not. For the two-class problem, we can assume the two classes are $y = \pm 1$ and that X can be divided into two matrices of red and blue examples, depending on whether $y = +1$ or $y = -1$. (Sometimes it can be more natural to assume $y \in \{0, 1\}$, but assuming ± 1 gives a symmetry that is useful.)
This situation can be set up as follows, if the two classes are defined by normal random variables:

```
# red instances (x-values with classification y=+1)
red.Center = c(2,4)
red.N = 300
red.X = cbind( rnorm(red.N, mean=red.Center[1]), rnorm(red.N, mean=red.Center[2]) )
red.y = rep(+1, red.N)
red.Avg = apply( red.X, 2, mean )

# blue instances (x-values with classification y=-1)
blue.Center = c(6,3)
blue.N = 500
blue.X = cbind( rnorm(blue.N, mean=blue.Center[1]), rnorm(blue.N, mean=blue.Center[2]) )
blue.y = rep(-1, blue.N)
blue.Avg = apply( blue.X, 2, mean )

N = red.N + blue.N
X = rbind( red.X, blue.X )
y = c( red.y, blue.y )
Avg = apply( X, 2, mean )

# plot the data
xmax = 10
plot( red.X, col="red", xlim=c(0,xmax), ylim=c(0,xmax), asp=1 )
points( blue.X, col="blue" )

# also plot the averages for each part of the data (red, blue, and all)
points( matrix(c(red.Avg,blue.Avg,Avg),3,2,byrow=TRUE), pch=19)
```

7.2.2 Linear Discriminants

Using a least squares approach we can solve $X\boldsymbol{a} = \boldsymbol{y}$ for a $p \times 1$ vector of coefficients $\boldsymbol{a}$.
The classifier function can then take the form

$$y \;=\; \widehat{f}(\boldsymbol{x}) \;=\; \sigma(\langle\, \boldsymbol{x},\, \boldsymbol{a} \,\rangle)$$

where σ is a function that `rounds' or `truncates' a real value into an integer class value. If a constant intercept c is desired so $\boldsymbol{x} = (\boldsymbol{z}\ 1)$ and $\boldsymbol{a} = (\boldsymbol{w}\ c)$, say, then

$$\langle\, \boldsymbol{x},\, \boldsymbol{a} \,\rangle \;=\; \langle\, (\boldsymbol{z}\ 1),\, (\boldsymbol{w}\ c) \,\rangle \;=\; \langle\, \boldsymbol{w},\, \boldsymbol{z} \,\rangle \;+\; c$$

--- and the classifier is defined by the **hyperplane** $\langle\, \boldsymbol{w},\, \boldsymbol{z} \,\rangle \;=\; c$.
In the two-class case, for example, we will want something like

$$\sigma(t) \;=\; \mathrm{sgn}(t) \;=\; \begin{cases} +1 & t > 0 \\ -1 & t < 0. \end{cases}$$

Input feature vectors $\boldsymbol{z}$ will be given the classification $\sigma(\,\langle\, \boldsymbol{w},\, \boldsymbol{z} \,\rangle \;+\; c\,)$.
We can find the coefficients $\boldsymbol{w}$ and c via Least Squares as follows:

```
A = cbind( X, 1 )
#  add a column of '1' values for an intercept coefficient

wc = solve(t(A) %*% A) %*% t(A) %*% y
#  alternatively:   wc = lsfit(X, y)$coefficients
#  alternatively:   wc = lm(y ~ X)$coefficients

w = wc[1:2]
c = wc[3]
classifier = function(x, w, c) { 2 * ((x %*% w + c) > 0) - 1 }
                              # i.e., ((x %*% w + c) > 0) ? +1 : -1
                              #        where +1 = red and -1 = blue,
## classifier = function(x, w, c) { sign(x %*% w + c) }
```

```
# plot the discriminant
curve( -(w[1] * x + c)/w[2], col="green", add=TRUE )
```

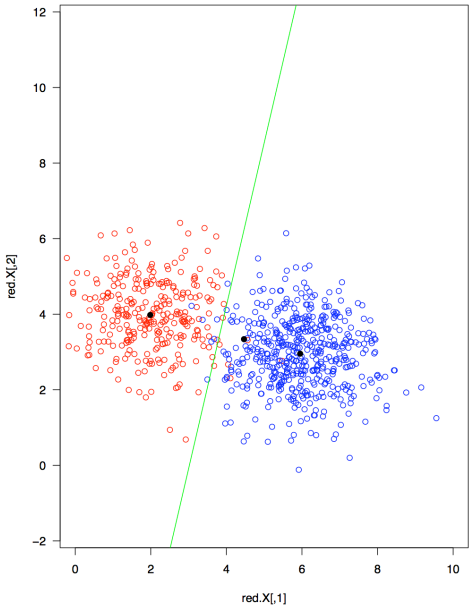


Figure 7.4: Least Squares discriminant for the red/blue Gaussian mixture

For more than two classes, we can construct linear **discriminant functions**

$$d_k(\boldsymbol{x}) \;=\; \langle\, \boldsymbol{x},\, \boldsymbol{a}_k \,\rangle$$

for each class k, and then define the classifier

$$f(\boldsymbol{x}) \;=\; \text{argmax}_k\; d_k(\boldsymbol{x})$$

-- yielding the class k whose discriminant function is highest.

7.2.3 LDA

Linear Discriminant Analysis (LDA) considers problems in which each class k can be modeled with a Gaussian distribution

$$g_k(\boldsymbol{x}) \;=\; 1/(2\pi)^{p/2}\, \sqrt{\det(\Sigma_k^{-1})}\; \exp(\; -1/2\; (\boldsymbol{x}-\boldsymbol{\mu}_k)'\, \Sigma_k^{-1} (\boldsymbol{x}-\boldsymbol{\mu}_k)\;)$$

where Σ_k is the covariance matrix. Thus, for a given input $\boldsymbol{x}$, we can estimate

$$\Pr[\;\text{class} = k \mid \boldsymbol{x}\;] \;=\; \frac{g_k(\boldsymbol{x})\; p_k}{\sum_\ell g_\ell(\boldsymbol{x})\; p_\ell}$$

where p_ℓ is the (prior) probability of $\boldsymbol{x}$ belonging to class i.

If all the covariance matrices are equal, then this simplifies:

$$\log \frac{\Pr[\ \text{class} = k \mid \boldsymbol{x}\ ]}{\Pr[\ \text{class} = \ell \mid \boldsymbol{x}\ ]} \ = \ \log \frac{g_k(\boldsymbol{x})\, p_k}{g_\ell(\boldsymbol{x})\, p_\ell} \ = \ \log \frac{p_k}{p_\ell} - 1/2\, \boldsymbol{\mu}_k{}' \, \Sigma^{-1} \boldsymbol{\mu}_k - 1/2\, \boldsymbol{\mu}_\ell{}' \, \Sigma^{-1} \boldsymbol{\mu}_\ell + (\boldsymbol{\mu}_k - \boldsymbol{\mu}_\ell)' \, \Sigma^{-1} \boldsymbol{x}$$

where Σ is the common covariance matrix. This is a linear function of $\boldsymbol{x}$, defining a hyperplane or discriminant boundary between class k and class i. We can equivalently define the discriminant function

$$d_k \ = \ \log\, p_k \ - \ 1/2\, \boldsymbol{\mu}_k{}' \, \Sigma^{-1} \boldsymbol{\mu}_k \ + \ \boldsymbol{x}' \, \Sigma^{-1} \boldsymbol{\mu}_k$$

since then $d_k > d_\ell$ precisely when $\Pr[\ \text{class} = k \mid \boldsymbol{x}\ ] > \Pr[\ \text{class} = \ell \mid \boldsymbol{x}\ ]$. Thus we can define a classifier function

$$f(\boldsymbol{x}) \ = \ \text{argmax}_k\, d_k(\boldsymbol{x}).$$

Assuming we do not have values for the probabilities p_k, means $\boldsymbol{\mu}_k$, or covariance matrices Σ, we can estimate these naively using the training dataset X: let p_k be the proportion of training examples in class k, $\boldsymbol{\mu}_k$ be the average of the $\boldsymbol{x}$ examples in class k, and define Σ to be the covariance matrix of the dataset X.

```
library(MASS)
LDA.model = lda(y ~ X)

str(LDA.model)
LDA.model$means
LDA.model$scaling
LDAclassification = function(Model,X)  {
   predict(Model,as.data.frame(X))$class
}
curve( (-(w.Simple[1]*x + c.Simple)/w.Simple[2]), col="green", add=TRUE)  #  plot the discriminant

#   we assume here that both gaussians have the identity covariance matrix.
N = 20
red.mean = rbind( mean(red.X[,1]), mean(red.X[,2]) )
red.gaussian = matrix(0, N,N)
blue.mean = rbind( mean(blue.X[,1]), mean(blue.X[,2]) )
blue.gaussian = matrix(0, N,N)

x1 = seq( 0, xmax, len = N );
x2 = seq( 0, xmax, len = N );
for (i in 1:N) {
   for (j in 1:N) {
     x = rbind( x1[i], x2[j] )
     xbar = red.mean
     red.gaussian[i,j]  = 1/sqrt(2*pi) * exp( -1/2 * t(x-xbar) %*% (x-xbar) )
     xbar = blue.mean
     blue.gaussian[i,j] = 1/sqrt(2*pi) * exp( -1/2 * t(x-xbar) %*% (x-xbar) )
   }
}

#   Superimpose the contours of the 2 gaussians;
contour( x=x1, y=x2, red.gaussian,  nlevels=5, add=TRUE )
contour( x=x1, y=x2, blue.gaussian, nlevels=5, add=TRUE )
contour( x=x1, y=x2, red.gaussian - blue.gaussian, nlevels=1, add=TRUE, col="purple" )
```

7.2.4 QDA

The LDA analysis above assumed all the covariance matrices were equal, resulting in cancellation of the quadratic term in the formula above. If we drop this assumption, we get

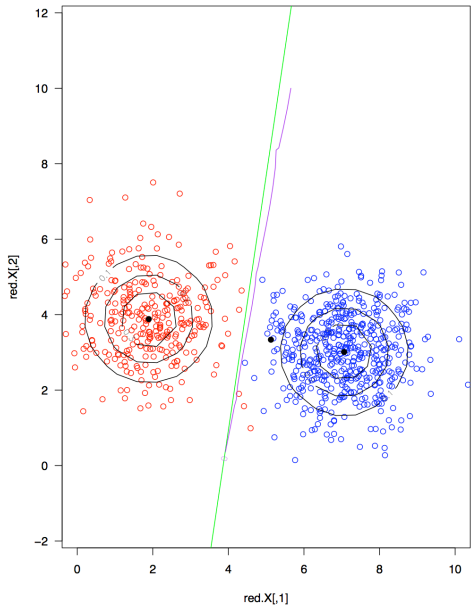


Figure 7.5: LDA for the red/blue Gaussian mixture; the green line is the Least Squares discriminant derived earlier.

Quadratic Discriminant Analysis (QDA). QDA generalizes on LDA by permitting discriminant functions of the form

$$d_k \quad = \quad \log\ p_k \quad - \quad 1/2\ \log\ \det(\Sigma_k) \quad - \quad 1/2\ \boldsymbol{\mu}_k{}'\ \Sigma_k^{-1} \boldsymbol{\mu}_k \quad + \quad (\boldsymbol{x} - \boldsymbol{\mu}_k)'\ \Sigma_k^{-1} (\boldsymbol{x} - \boldsymbol{\mu}_k)$$

where each class k has its own covariance matrix Σ_k. This is a quadratic function of $\boldsymbol{x}$. We can adapt the LDA program above for QDA. For this to be an interesting demonstration, we modified the red/blue mixture to involve some covariance structure, as is visible in Figure 7.6 below.

```
library(MASS)
QDA.model = qda(y ~ X)

str(QDA.model)
QDA.model$means
QDA.model$scaling

QDAclassification = function(Model,X)  {
   predict(Model,as.data.frame(X))$class
}

# find all points whose classifications didn't match

cat("All misclassified instances with the simple Linear discriminant:\n\n")
cat((1:N)[ classifier(X, w.Simple, c.Simple) != y ], "\n\n")

cat("All misclassified instances with the Quadratic discriminant:\n\n")
cat((1:N)[ QDAclassification(QDA.model, X) != y ], "\n\n")


curve( (-(w.Simple[1]*x + c.Simple)/w.Simple[2]), col="green", add=TRUE)  # plot Linear discriminant
```

```
#-------------------------------------------------------------------------------------
#      superimpose the contours of the 2 gaussians
#-------------------------------------------------------------------------------------

n = 30

x1 = seq( 0, xmax, len = n );
x2 = seq( 0, xmax, len = n );

inv = function(A) solve(A)

red.mean  = rbind( mean(red.X[,1]),  mean(red.X[,2]) )   # apply( red.X, 2, mean )
red.W = inv(cov(red.X))
red.det.W = det(red.W)
red.gaussian = matrix(0, n,n)

blue.mean = rbind( mean(blue.X[,1]), mean(blue.X[,2]) )  # apply( blue.X, 2, mean )
blue.W = inv(cov(blue.X))
blue.det.W = det(blue.W)
blue.gaussian = matrix(0, n,n)

for (i in 1:n) {
   for (j in 1:n) {
     x = rbind( x1[i], x2[j] )
     xbar = red.mean
     W = red.W
     red.gaussian[i,j]  = 1/sqrt(2*pi*red.det.W)  * exp( -1/2 * t(x-xbar) %*% W  %*% (x-xbar) )
   }
}
for (i in 1:n) {
   for (j in 1:n) {
     x = rbind( x1[i], x2[j] )
     xbar = blue.mean
     W = blue.W
     blue.gaussian[i,j] = 1/sqrt(2*pi*blue.det.W) * exp( -1/2 * t(x-xbar) %*% W %*% (x-xbar) )
   }
}

contour( x=x1, y=x2, red.gaussian,  nlevels=5, add=TRUE )
contour( x=x1, y=x2, blue.gaussian, nlevels=5, add=TRUE )
contour( x=x1, y=x2, red.gaussian - blue.gaussian, nlevels=1, add=TRUE, col="purple" )
# this purple line is the quadratic discriminant between the two gaussians
```

7.3 Logistic Regression

7.3.1 Sigmoid Functions

We say $\sigma(x)$ is a `*sigmoid*' function if it resembles the signum function $sgn(x)$. Important examples of sigmoid functions include

$$\begin{aligned} \sigma(x) &= \text{logit}^{-1}(c\,x) = \frac{1}{1+e^{-c\,x}} \\ \sigma(x) &= \tanh(c\,x) = \frac{1-e^{-2\,c\,x}}{1+e^{-2\,c\,x}} \end{aligned}$$

where c is a constant, typically near 1, that can be used to `tighten' the sigmoid. These importance of these functions is that they are differentiable approximations of the signum

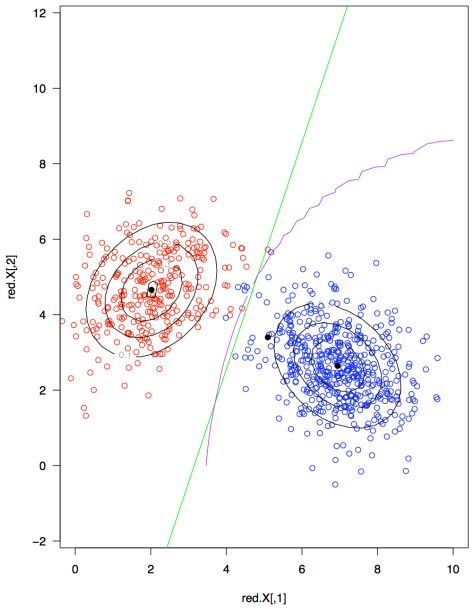


Figure 7.6: QDA for the red/blue Gaussian mixture; the green line is the Least Squares discriminant.

function $\sigma(x) \simeq sign(x)$. The logit function is of particular interest here, since it is the **log odds** function

$$\text{logit}(p) = \log \frac{p}{1-p}.$$

These two sigmoid functions and their inverses are shown in Figure 8.5.

7.3.2 Logistic Regression

Suppose that we are trying to solve a classification problem in which there are two classes: $y = 0$ and $y = 1$. We are given $\boldsymbol{x}$ and seek to find the best value of y. Instead of using a normal regression model like

$$y = \langle \boldsymbol{w}, \boldsymbol{x} \rangle + c$$

we want to use a model like

$$y = \sigma(\langle \boldsymbol{w}, \boldsymbol{x} \rangle + c)$$

where σ is a sigmoid function.
Logistic regression is like regression, but with a sigmoid function σ that is the inverse logistic function (logit^{-1}). Thus the sigmoid form

$$y = \sigma(\langle \boldsymbol{w}, \boldsymbol{x} \rangle + c)$$

is equivalent to

$$\text{logit}(y) = \langle \boldsymbol{w}, \boldsymbol{x} \rangle + c.$$

Again, the target variable y is a variable with discrete values 0 and 1. One approach to classifying in this case is to evaluate both $\Pr[\text{ class} = 0 \mid \boldsymbol{x}\,]$ and $\Pr[\text{ class} = 1 \mid \boldsymbol{x}\,]$ and select the

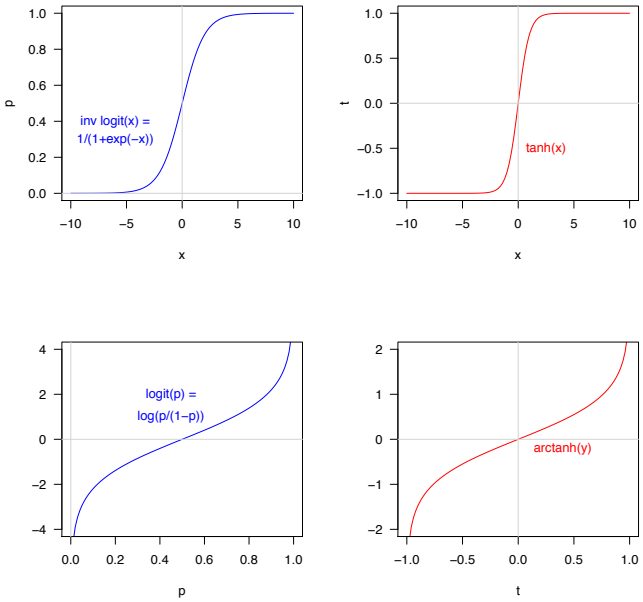


Figure 7.7: Two sigmoid functions, and their inverses.

class for which this probability is highest. Since $\Pr[\text{ class} = 1 \mid \boldsymbol{x}\ ] = 1 - \Pr[\text{ class} = 0 \mid \boldsymbol{x}\ ]$ we can equivalently say compactly that we compute the log-odds

$$\log \frac{\Pr[\text{ class} = 1 \mid \boldsymbol{x}\ ]}{1 - \Pr[\text{ class} = 1 \mid \boldsymbol{x}\ ]}$$

and choose class 1 if it is positive, otherwise choose class 0. Using the logit function

$$\sigma^{-1}(p) = \text{logit}(p) = \log \frac{p}{1-p}$$

we can state more compactly that we want to pick class

$$\text{round}(\text{logit}(\Pr[\text{ class} = 1 \mid \boldsymbol{x}\ ])) \approx \sigma^{-1}(\Pr[\text{ class} = 1 \mid \boldsymbol{x}\ ]).$$

In general, if there are $K > 2$ classes, we can consider all K models of the form

$$\log \frac{\Pr[\text{ class} = k \mid \boldsymbol{x}\ ]}{1 - \Pr[\text{ class} = k \mid \boldsymbol{x}\ ]} = \langle \boldsymbol{w}_k, \boldsymbol{x} \rangle + c_k$$

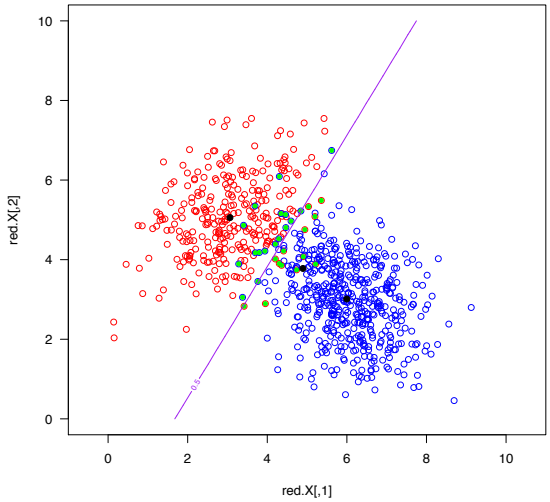
with $\ell \neq k$ and σ is the logistic function. An equivalent formulation of these models is:

$$\Pr[\text{ class} = k \mid \boldsymbol{x}\ ] = \sigma(\langle \boldsymbol{w}_k, \boldsymbol{x} \rangle + c_k).$$

Generally we seek `optimal' values for all K models. If W is the matrix whose K columns are $\boldsymbol{w}_k$ ($k = 1, \ldots, K$), and c is the vector whose K entries are c_k ($k = 1, \ldots, K$), we want to obtain the values of W and $\boldsymbol{c}$ that maximize the log-likelihood function

$$\ell(W, \boldsymbol{c}) = \sum_i \log \Pr[\ y_i \mid \boldsymbol{x}_i, W, \boldsymbol{c}\ ].$$

An iterative procedure for the logistic regression is in [1, Section 4.4]. Figure 7.8 shows an implementation in R using the `glm` (Generalized Linear Model) function.




```
# red instances (x-values with classification y=0)
red.Center = c(3,4)
red.N = 300
red.X = cbind( rnorm(red.N, mean=red.Center[1]), rnorm(red.N, mean=red.Center[2]) )
red.X[,2] = red.X[,1]/3 + red.X[,2]         #  add a little covariance
red.y = rep(0, red.N)
red.Avg = apply( red.X, 2, mean )

# blue instances (x-values with classification y=1)
blue.Center = c(6,5)
blue.N = 500
blue.X = cbind( rnorm(blue.N, mean=blue.Center[1]), rnorm(blue.N, mean=blue.Center[2]) )
blue.X[,2] = -blue.X[,1]/3 + blue.X[,2]     #  add a little covariance
blue.y = rep(1, blue.N)
blue.Avg = apply( blue.X, 2, mean )

N = red.N + blue.N
X = rbind( red.X, blue.X )
y = c( red.y, blue.y )
Avg = apply( X, 2, mean )

logistic.regression.model = glm( y ~ X, family=binomial("logit") )
summary(logistic.regression.model)
    #  Call:
    #  glm(formula = y ~ X, family = binomial("logit"))
    #  Deviance Residuals:
    #       Min        1Q    Median        3Q       Max
    #  -2.67974  -0.09326   0.01364   0.07823   2.72597
    #  Coefficients:
    #              Estimate Std. Error z value Pr(>|z|)
    #  (Intercept)  -4.4202     1.3281  -3.328 0.000874 ***
    #  X1            2.5429     0.2591   9.815  < 2e-16 ***
    #  X2           -1.4189     0.2145  -6.614 3.73e-11 ***
    #  ---
    #  Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
    #  (Dispersion parameter for binomial family taken to be 1)
    #      Null deviance: 1058.50  on 799  degrees of freedom
    #  Residual deviance:  169.18  on 797  degrees of freedom
    #  AIC: 175.18
    #  Number of Fisher Scoring iterations: 8
        logit = qlogis  # function(p) log(p/(1-p))
    inv_logit = plogis  # function(x) 1/(1+exp(-x))
classification = function(Model,X)  { round(logit( predict(Model,as.data.frame(X)) )) }

# plot the data
xmax = 10
plot( red.X, col="red", xlim=c(0,xmax), ylim=c(0,xmax), asp=1 )
points( blue.X, col="blue" )
# also plot the averages for each part of the data (red, blue, and all)
points( matrix(c(red.Avg,blue.Avg,Avg),3,2,byrow=TRUE), pch=19)

misclassified = (1:N)[ classification(logistic.regression.model, X) != y ]
misclassified

    #   [1]  23  61  64  67  81  87  90 121 127 134 169 231 283 320 332 342 385 457 460
    #  [20] 502 514 551 624 631 683 712 771 775 776 799

points( X[misclassified,], col="green", pch=20 )   # plot misclassified points in green
```


Figure 7.8: Example of Logistic Regression.

Bibliography

[1] A.W.F. Edwards, *Likelihood*, Johns Hopkins University Press, 1992.

[2] T. Hastie, R. Tibshirani, J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, Springer-Verlag, 2009.