Classification with Gaussians

Andreas C. Kapourani
(Credit: Hiroshi Shimodaira)

1 Classification

In the previous lab sessions, we applied Bayes' theorem in order to perform statistical pattern recognition, however, we had discrete probabilities and not probability density functions. It turns out that the rules for probability densities are similar to those for discrete probabilities, with the only difference that we replace summation with *integration*.

Again, let C denote the class, taking values $1, ..., K$, and let $P(C_k)$ be the prior probability of class k. Let $P(x|C_k)$ be the likelihood of continuous data x given the class C_k. To perform classification we compute the posterior probability using Bayes' theorem and assign each data point to the class with the highest posterior probability.

The prior probability expresses our beliefs about the class of each data point before any evidence is taken into account. We can compute the prior simply by taking the proportion of each class out of the total observations. Concerning the likelihood, we assume that the *pdf* of a continuous random variable x, given the class C_k, follows a Gaussian distribution with mean μ_k and variance σ_k^2. That is, each class C_k is modeled with a different Gaussian distribution. Then the posterior distribution is given by:

$$\begin{aligned} P(C_k|x) &\propto P(x|C_k)P(C_k) && (1) \\ &\propto \mathcal{N}(x; \mu_k, \sigma_k^2)P(C_k) && (2) \end{aligned}$$

where $\mathcal{N}$ denotes the Gaussian distribution.

In the case of D-dimensional data $\mathbf{x}$ the likelihood is modelled using a multivariate Gaussian distribution with mean vector $\boldsymbol{\mu}_k$ and covariance matrix $\boldsymbol{\Sigma}_k$.

To assign x to the most probable class, we take the posterior probabilities ratio, also known as Bayes decision rule:

$$\frac{P(C_1|x)}{P(C_2|x)} = \frac{P(x|C_1)P(C_1)}{P(x|C_2)P(C_2)} \tag{3}$$

If the ratio in the above equation is greater than 1, then x is classified as belonging to class 1, if x is less than 1, then x is classified in class 2. Due to numerical stability, it often useful to take logs, so for the example of Gaussian likelihoods, the *log odds ratio* becomes:

$$\begin{aligned} \ln\frac{P(C_1|x)}{P(C_2|x)} &= \ln\frac{P(x|C_1)P(C_1)}{P(x|C_2)P(C_2)} && (4) \\ &= \ln P(x|C_1) + \ln P(C_1) - \ln P(x|C_2) - \ln P(C_2) && (5) \\ &= \ln \mathcal{N}(x; \mu_1, \sigma_1^2) + \ln P(C_1) - \ln \mathcal{N}(x; \mu_2, \sigma_2^2) - \ln P(C_2) && (6) \end{aligned}$$

2 Generating synthetic data

To illustrate the use of the Gaussian distribution for the classification task, we will generate two-dimensional synthetic data coming from two distinct classes, C_1 and C_2. From class C_1, 100 training data points are generated from a Gaussian with parameters:

$$\boldsymbol{\mu}_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix} \qquad \boldsymbol{\Sigma}_1 = \begin{pmatrix} 2 & -1 \\ -1 & 2 \end{pmatrix} \tag{7}$$

and from class C_2, 100 training data points using a Gaussian with parameters:

$$\boldsymbol{\mu}_2 = \begin{pmatrix} 2 \\ 2 \end{pmatrix} \qquad \boldsymbol{\Sigma}_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{8}$$

In MATLAB we can generate multivariate normal random numbers using the `mvnrnd` function.

```
rng(4) % for reproducibility

% Mean and covariance for class C1
mu1 = [0 0];
Sigma1 = [2 -1; -1 2];

% Mean and covariance for class C2
mu2 = [2 2];
Sigma2 = [1 0; 0 1];

% Generate 100 training synthetic data of class 1
X1 = cat(2, mvnrnd(mu1,Sigma1,100), ones(100,1));

% Generate 100 training synthetic data of class 2
X2 = cat(2, mvnrnd(mu2,Sigma2,100), 2*ones(100,1));

% Concatenate training data from both classes
X = cat(1, X1, X2);

% Keep the training class labels in a separate vector
train_labels = X(:,3);
```

To assess the performance of the classifier, we will generate another 50 data points from each class. These are the *test data* and will be used only for assessment of the model and not during the training phase.

```
% Test data for class C1 and class C2

% First 50 points in C1 and the rest 50 points in C2
T = cat(1, mvnrnd(mu1,Sigma1,50), mvnrnd(mu2,Sigma2,50));

% Keep the test class labels in a separate vector
test_labels = cat(1, ones(50,1), 2*ones(50,1));
```

We can show the two classes in the same plot for the training data using the following code:

```
% data for class 1 with red colour
scatter(X(train_labels==1, 1), X(train_labels==1, 2), 'r', 'o');
hold on;
% data for class 2 with blue colour
scatter(X(train_labels==2, 1), X(train_labels==2, 2), 'b', 'x');
```

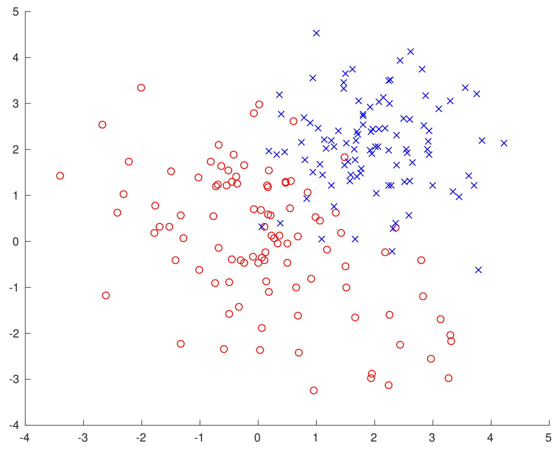

Figure 1: *Training data from two classes: Class 1 (red circles), Class 2 (blue crosses).*

Exercise

Plot the corresponding test data using the `scatter` function as we did for the training data.

3 Estimating model parameters

The prior probabilities $P(C_k)$ for each class can be computed directly by taking the proportion of each class from the total observations, which will result to $P(C_1) = P(C_2) = 0.5$.

```
prior = [0.5 0.5];
```

The likelihood of each class is modeled using a multivariate Gaussian distribution, hence we need to evaluate the pdf of the Gaussian for each data point $\mathbf{x}$. The following function is an extension of the `gaussianMV` function from Lab 6, where we evaluate the Gaussian pdfs for a set of data points together (taken from the *Netlab* toolbox).

```
function y = gaussianMV(mu, covar, X)
% Y = GAUSSIANMV(MU, COVAR, X) evaluates a multi-variate Gaussian
   density in D-dimensions at a set of points given by the rows of
   the matrix X. The Gaussian density has mean vector MU and
   covariance matrix COVAR.
%
% Copyright (c) Ian T Nabney (1996-2001)

  [n, d] = size(X);
  [j, k] = size(covar);

% Check that the covariance matrix is the correct dimension
  if ((j ~= d) | (k ~=d))
    error('Dimension of covariance matrix and data should match');
  end

  invcov = inv(covar);
  mu = reshape(mu, 1, d);     % Ensure that mu is a row vector

  % Replicate mu and subtract from each data point
  X = X - ones(n, 1)*mu;
  fact = sum(((X*invcov).*X), 2);

  y = exp(-0.5*fact);
  y = y./sqrt((2*pi)^d*det(covar));
```

The goal of training is to fit the model to the data, i.e. to estimate the Gaussian parameters μ_k and $\mathbf{\Sigma}_k$ (**Note**: In our case we already knew the actual model parameters since we used them to generate the data; however this is not the case for real life problems, where we are given only a set of (possibly noisy) observations).

The maximum likelihood estimate for the parameters of the Gaussian distribution is the sample mean and sample covariance of the data points that belong to each class:

```
% ML estimate of the parameters for Gaussians distribution
% Estimate means and covariances
mu_hat = zeros(2, 2);
Sigma_hat = zeros(2,2,2);

for k = 1:2
    mu_hat(:,k) = mean(X(train_labels==k, 1:2));
    Sigma_hat(:,:,k) = cov(X(train_labels==k, 1:2), 1);
end

>> mu_hat
mu_hat =
    0.1600    1.9633
   -0.0031    2.0848

>> Sigma_hat
Sigma_hat(:,:,1) =
    1.9749   -1.0334
   -1.0334    2.1867

Sigma_hat(:,:,2) =
    0.8095    0.0047
    0.0047    0.9766
```

Observe the estimated mean vectors and covariance matrices for each class and compare them with the actual parameters that generated the data.

We can plot the contours of the two Gaussian distributions using the `contourGauss2D` function shown below. This function, takes as input a mean vector, a covariance matrix, and a valid colour character, and makes a contour plot between ±2 standard deviations along each dimension.

```
function contourGauss2D(mu, covar, col)
% Modified version of lecture notes
 % make a contour plot of a 2D Gaussian

  if (length(mu) > 2)
    disp('error: Only two dimensional data are allowed');
    return
  end

  step = 0.1;
  C = covar; % the covariance matrix
  A = inv(C); % the inverse covariance matrix

  % Get the variance for each axis
  var = diag(C);
  % Get the maximum variance
  maxsd = max(var(1),var(2));
  % plot between +-2SDs along each dimension
  % location of points at which x is calculated
  x = mu(1)-2*maxsd:step:mu(1)+2*maxsd;
  % location of points at which y is calculated
  y = mu(2)-2*maxsd:step:mu(2)+2*maxsd;

  [X, Y] = meshgrid(x,y); % matrices used for plotting

  % Compute value of Gaussian pdf at each point in the grid
  % writing the quadratic form fully
  z = 1/(2*pi*sqrt(det(C))) * exp(-0.5 * (A(1,1)*(X-mu(1)).^2 ...
      + 2*A(1,2)*(X-mu(1)).*(Y-mu(2)) + A(2,2)*(Y-mu(2)).^2));

  % Create the contour plot of 2D Gaussian
  if nargin < 3
    contour(x, y, z); % If no colour argument is given
  else
    contour(x, y, z, 'Color', col);
  end
end
```

Exercise

Plot in the same figure the training data points together with the contours of the fitted Gaussian distributions as shown in *Figure 2*. **Hint**: Use the `contourGauss2D` function given above.

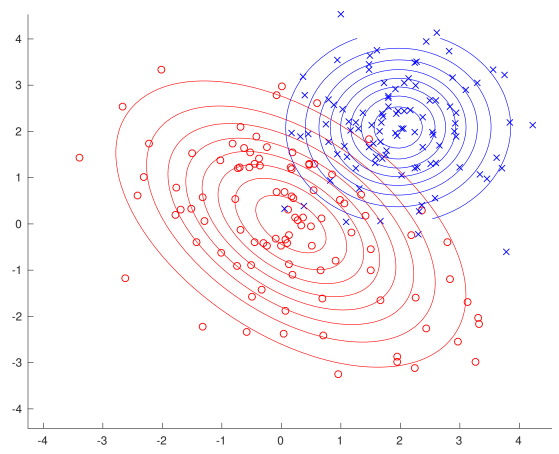

Figure 2: *Gaussian distributions estimated from training data for each class.*

4 Making predictions

After fitting the model to the data, we can go ahead and classify each of the test points. For each class C_k we will evaluate the likelihood $P(\mathbf{x}|C_k)$ using the function `gaussianMV`, that we implemented in the previous section. Then, we will use *Eq. 3* to assign each test data point to the class with the highest posterior probability, or equivalently the highest joint probability, since the denominator of Bayes' rule does not depend on the parameters. In order to assign the data points to each class, we will concatenate the corresponding vectors to a $N \times 2$ matrix, and extract the maximum from each row.

```
% Create an Nx2 matrix
test_prob = zeros(length(T), 2);

% Iterate over each class
for (k = 1:2)
    % Compute likelihood of test data under each class
    lik_k = gaussianMV(mu_hat(:,k), Sigma_hat(:,:,k), T);

    % Multiply with prior to get the joint probability
    test_prob(:,k) = lik_k * prior(k);

end

% Assign each data point to the class with the highest probability,
    stores in variable class_pred
[max_out, test_pred] = max(test_prob, [], 2);
```

Exercise

Check what the above code does and understand the output of the `max` function in each case. What we are interested are only the class label outputs, stored in the `test_pred` variable.

4.1 Create confusion matrix

We can assess the performance of the classifier using a *confusion matrix*. The columns of a confusion matrix correspond to the predicted classes (i.e. classifier outputs). The rows correspond to the actual (true) class labels. The value at position (r, c) is the number of points from true class r that were classified as class c. The number of correctly classified observations is obtained by summing the numbers on the leading diagonal.

```
% Compute confuction matrix using the 'confusionmat' function
conf_mat = confusionmat(test_labels, test_pred)
```

The results stored in the `conf_mat` variable are the following:

		Predicted Class	
Test Data		1	2
Actual	1	47	3
class	2	7	43

From the confusion matrix we can see that the overall proportion of test observations correctly classified is $(47 + 43)/100 = 0.9$.

We can scatter plot the test data with colours based on the classification of the test points. The following code plots only the test data coming from the actual class C_1.

```
T1 = T(test_labels == 1, :);
% Data assigned to class 1 with red colour
scatter(T1(test_pred(1:50)==1,1), T1(test_pred(1:50)==1,2),'r','o')
hold on;
% Data assigned to class 2 with blue colour
scatter(T1(test_pred(1:50)==2,1), T1(test_pred(1:50)==2,2),'b','x')
```

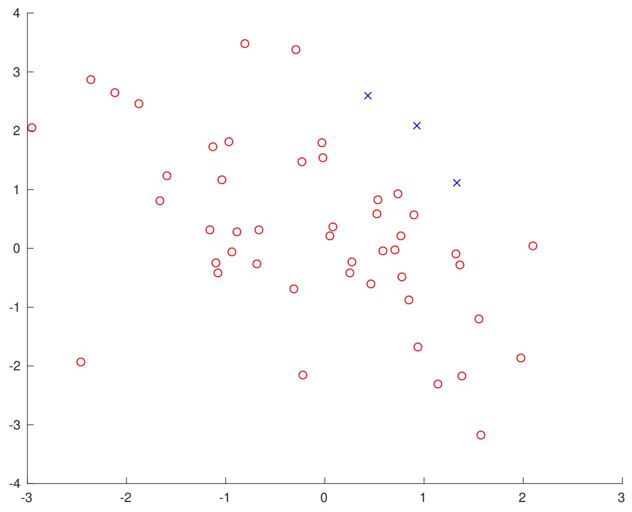

Figure 3: *Classification of test points from class C_1.*

Exercise

Plot the classification of the test points `T` coming from class C_2.

5 Comparison with k-NN classification

In this section, we will compare the Gaussian classification with the k-NN classification that we described in Lab session 5. The training and test data will be the same as described before.

As you remember, the training of the K-NN classifier is simple, we just need to store all the training set. However, the testing procedure involves measuring the distance between each test point and every training point and assigning it to the class where the majority of the K-nearest neighbours belongs.

5.1 Implement k-NN classification algorithm

Now we can implement the k-NN algorithm exactly the same way we did in Lab 5. We will create a function `simplekNN`, which has 4 arguments: k the number of k-Nearest Neighbours, A a matrix containing all the training data, C a column vector containing the corresponding class labels for each entry of matrix A, and finally t is a single test point. The function will return the class label assignment of the test point.

```
function [p] = simplekNN(k, A, C, t)
    % compute square distance between test point and
    % each training observation
    r_zx = sum(bsxfun(@minus, A, t).^2, 2);
    % Sort the distances in ascending order
    [r_zx,idx] = sort(r_zx, 1, 'ascend');

    % Keep only the K nearest neighbours
    r_zx = r_zx(1:k); % keep the first 'Knn' distances
    idx = idx(1:k);   % keep the first 'Knn' indexes

    % majority vote only on those 'Knn' indexes
    p = mode(C(idx));
end
```

The following code runs k-NN classification for all the test data T using 3-nearest neighbours:

```
% Classify based on 3-Nearest Neighbours
k = 3;
N = length(T);
knn_pred = zeros(1, N);
for i = 1:N
  knn_pred(i) = simplekNN(k, X(:, 1:2), train_labels, T(i,:));
end
```

Again, we can asses the performance of the classifier using a *confusion matrix*.

```
% Compute confuction matrix of kNN classifier
knn_conf_mat = confusionmat(test_labels, knn_pred)
```

The results are the following:

		Predicted Class	
Test Data		1	2
Actual	1	46	4
class	2	11	39

From the confusion matrix we can see that the overall proportion of test observations correctly classified is $(46 + 39)/100 = 0.85$.

Hence, the Gaussian classifier has a better performance than the k-NN classifier for this specific dataset with the given parameters (**Note**: We should mention that we cheated somehow, since the data were generated from two Gaussian distributions, hence a Gaussian classifier had an advantage in fitting the data better).

Exercises

- Plot the k-NN classification of the test points, similarly to what we did in *Figure 3*.
- Change the number of k-nearest neighbours and assess the performance of the classifier.