Programming Assignment
Data-Driven Classifiers

Deep Learning Neural Networks
Fall 2017

Introduction

This assignment is thus an introduction to pattern recognition on static data. In this assignment you will explore a variety of data-driven classifiers, that is, classifiers which are trained from a set of training data. You will empirically characterize their performance on a set of data.

The data are produced according to a model described later: the distributions are not Gaussian. Figure 1 shows the training data for $d = 2$ dimensional data. The `x` data is for class 0; the `o` data is for class 1. This training data is in the file `classasgntrain1.dat`. The first two columns of this (ascii) file are the x and y coordinates of $N = 100$ points of the class 0 data; the second two columns of the data are the x and y coordinates of the class 1 data. Instances of data are generated by calling the MATLAB function `gendat2`, as follows:

```
x = gendat2(class,N);
```

where `class` indicates which class of data you want (either 0 or 1), and `N` indicates how many points you want to generate. For example, to generate 20 points of class zero data and store it in the array `samp1` you would call

```
samp1 = gendat2(0,20);
```

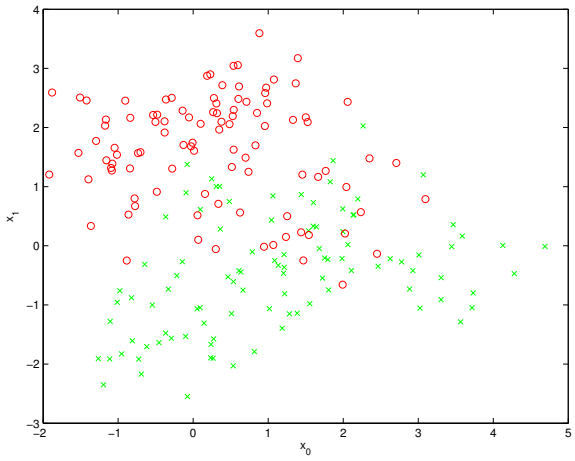

Figure 1: Training data for two-class, two-dimensional problem

Let N_0 be the number of points of training data from class 0, and let N_1 be the number of points of training data from class 1. Let $N = N_0 + N_1$ be the total number of training data. Let $\mathbf{x}_i$ denote the coordinates of a training vector (thinking of this as a **column** vector), and let y_i denote the corresponding class value. That is, either $y_i = 0$ or $y_i = 1$, depending on the class the point comes from. Then the set of data $(\mathbf{x}_i, y_i)$, $i = 1, 2, \ldots, N$ is the total set of training data. For some of the discussions below, we assume that the data are ordered so that the first N_0 training points are from class 0 and the next N_1 training points are from class 1.

In the sections below you are introduced to several different classification algorithms. Your assignment will be to program each of these in PYTHON, then evaluate their performance on the training data provided as well as on new test data that you generate with the `gendat2` function.

The classifiers presented here are described in *The Elements of Statistical Learning Theory* by T. Hastie, R. Tibshirani, and J. Friedman (Springer, 2001).

Linear regression

For some classification problems, it suffices to provide a straight line (or plane, in higher dimensions) dividing the two classes. Linear regression is one means of determining such a dividing line (or plane).

Let $\mathbf{x} = [x_1, x_2, \ldots, x_d]^T$ be a d-dimensional vector. A linear function of $\mathbf{x}$ is

$$f(\mathbf{x}) = \beta_0 + \sum_{j=1}^{d} x_j \beta_j.$$

The term β_0 is to change the "y-intercept" of the line, to account for non-zero mean data. In the usual regression problem, each point $\mathbf{x}_i$ has associated with it some value y_i. We choose the parameters of the line $\beta = [\beta_0, \beta_1, \ldots, \beta_d]^T$ in such a way that we get the best match possible over all training points. That is, we desire to minimize

$$\text{RSS}(\beta) = \sum_{i=1}^{N} (y_i - \beta_0 - \sum_{j=1}^{d} x_{ij}\beta_j)^2$$

where x_{ij} is the jth coordinate of the training data vector $\mathbf{x}_i$, and where RSS stands for "residual sum of squares" (that is, the sum of the squares of the "residuals" or errors). This can be written in more convenient form as follows. First, let

$$\underline{\mathbf{x}} = \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix}.$$

That is, stack a 1 on top of the column vector $\mathbf{x}_i$. Then the linear function can be written

$$f(\mathbf{x}) = \beta^T \underline{\mathbf{x}}.$$

Now let $\mathbf{y}$ be the stack of data output values for all the data:

$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}$$

and $\mathbf{X}$ be the $N \times (d+1)$ matrix formed by stacking the data as *rows* (including the 1):

$$\mathbf{X} = \begin{bmatrix} 1 & \mathbf{x}_1^T \\ 1 & \mathbf{x}_2^T \\ \vdots & \\ 1 & \mathbf{x}_N^T \end{bmatrix}.$$

We can now write

$$\text{RSS}(\beta) = (\mathbf{y} - \mathbf{X}\beta)^T (\mathbf{y} - \mathbf{X}\beta).$$

Problem 1: Show that the β that minimizes $\text{RSS}(\beta)$ is

$$\beta = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}.$$

For a binary classification problem, the $\mathbf{y}$ data are modified somewhat. We form the $N \times 2$ matrix $\mathbf{Y}$ whose 2 columns indicate respective class membership of the given data point. That is, we form the $\mathbf{X}$ matrix ($N \times (d+1)$) and the corresponding $\mathbf{Y}$ matrix ($N \times 2$) as follows:

$$\mathbf{X} = \begin{bmatrix} 1 & \mathbf{x}_1^T \\ 1 & \mathbf{x}_2^T \\ \vdots & \\ 1 & \mathbf{x}_{N_0}^T \\ 1 & \mathbf{x}_{N_0+1}^T \\ 1 & \mathbf{x}_{N_0+2}^T \\ \vdots & \\ 1 & \mathbf{x}_N \end{bmatrix} \qquad \mathbf{Y} = \begin{bmatrix} 1 & 0 \\ 1 & 0 \\ \vdots & \\ 1 & 0 \\ 0 & 1 \\ 0 & 1 \\ \vdots & \\ 0 & 1 \end{bmatrix}$$

The matrix $\mathbf{Y}$ is called the *indicator response matrix*. We then find a $(d+1) \times 2$ matrix $\hat{\mathbf{B}}$ so that

$$\text{RSS}(\hat{\mathbf{B}}) = \|Y - \mathbf{X}\hat{\mathbf{B}}\|^2 \tag{1}$$

is as small as possible. (This approach extends to more general problems with K classes, in which case the the $\mathbf{Y}$ matrix is $N \times K$.)

Problem 2: Show that if the norm $\|\cdot\|^2$ in (1) is the Frobenius norm, then the $\hat{\mathbf{B}}$ minimizing (1) is determined by

$$\hat{\mathbf{B}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}.$$

The estimated indicator response matrix is

$$\hat{\mathbf{Y}} = \mathbf{X}^T\hat{\mathbf{B}}$$

To use the classifier after training, suppose that the vector we want to classify is $\mathbf{x}$. We form the indicator response vector

$$\hat{\mathbf{y}}^T = \begin{bmatrix} 1 & \mathbf{x}^T \end{bmatrix} \hat{\mathbf{B}}.$$

This is a vector with $K = 2$ elements in it, $\hat{\mathbf{y}}^T = [\hat{y}_0, \hat{y}_1]$. The estimated class $\hat{\mathbf{k}}$ corresponds to the column with the largest value:

$$\hat{\mathbf{k}} = \arg \max_{k \in \{0,1,\ldots,K-1\}} \hat{y}_k.$$

Problem 3:

Re-write the function `gendat2.m` into Python.

Using the 100 points of training data in `classasgntrain1.dat`, write PYTHON code to train the coefficient matrix $\hat{B}$. Determine how well this linear regression classifier works on the training data and on 10,000 points of test data (5000 from each class) generated by `gendat2`. Record your results in a table such as the following:

	Error rates	
Method	Training	Test
linear regression		
quadratic regression		
linear discriminant analysis		
quadratic discriminant analysis		
logistic regression		
1-nearest neighbor		
5-nearest neighbor		
15-nearest neighbor		
Bayes		
(etc.)		

You will be filling in other rows of the table as you progress through this assignment.

Also, make a plot indicating the regions of classification. In this linear classifier it is possible to do this analytically, but for other classifiers this is difficult. Instead, simply closely sample the input space and plot the classification results.

To get you started (and to show you this is not an impossibly difficult assignment), I am providing the *solution* to this problem. This code provides a complete solution to this problem.

```
% classasgn1.m
% Sample classifier program


% Load the training data and divide into the different classes
load classasgntrain1.dat
x0 = classasgntrain1(:,1:2)';  % data vectors for class 0 (2 x N0)
N0 = size(x0,2);
x1 = classasgntrain1(:,3:4)';  % data vectors for class 1 (2 x N1)
N1 = size(x1,2);
N = N0 + N1;
```

```
% plot the data
clf;
plot(x0(1,:), x0(2,:),'gx');
hold on;
plot(x1(1,:), x1(2,:),'ro');
xlabel('x_0');
ylabel('x_1');

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% Linear regression classifier
% Build the X matrix
X = [ones(N0,1) x0';
 ones(N1,1) x1'];

% Build the indicator response matrix
Y = [ones(N0,1) zeros(N0,1);
 zeros(N1,1) ones(N1,1)];

% Find the parameter matrix
Bhat = (X'*X) \ X'* Y;

% Find the approximate response
Yhat = X*Bhat;
Yhathard = Yhat > 0.5;  % threshold into different classes

nerr = sum(sum(abs(Yhathard - Y)))/2;  % count the total number of errors
errrate_linregress_train = nerr / N;

% Now test on new (testing data)
Ntest0 = 5000;   % number of class 0 points to generate
Ntest1 = 5000;   % number of class 1 points to generate

xtest0 = gendat2(0,Ntest0);  % generate the test data for class 0
xtest1 = gendat2(1,Ntest1);  % generate the test data for class 1
nerr = 0;
for i=1:Ntest0
  yhat = [1 xtest0(:,i)']*Bhat;
  if(yhat(2) > yhat(1))  % error: chose class 1 over class 0
nerr = nerr+1;
  end
end

for i=1:Ntest1
  yhat = [1 xtest1(:,i)']*Bhat;
  if(yhat(1) > yhat(2))  % error: chose class 0 over class 1
nerr = nerr+1;
  end
end
errrate_linregress_test = nerr / (Ntest0 + Ntest1);

% Plot the performance across the window (that is, plot the classification
% regions)
xmin = min([x0(1,:) x1(1,:)]);  xmax = max([x0(1,:) x1(1,:)]);
ymin = min([x0(2,:) x1(2,:)]);  ymax = max([x0(2,:) x1(2,:)]);
xpl = linspace(xmin,xmax,100);
ypl = linspace(ymin,ymax,100);
redpts = [];  % class 1 estimates
greenpts = [];% class 0 estimatees
```

```
% loop over all points
for x = xpl
  for y = ypl
yhat = [1 x y]*Bhat;
if(yhat(1) > yhat(2)) % choose class 0 over class 1
  greenpts = [greenpts [x;y]];
else
  redpts = [redpts [x;y]];
end
  end
end
plot(greenpts(1,:), greenpts(2,:),'g.','MarkerSize',0.25);
plot(redpts(1,:), redpts(2,:),'r.','MarkerSize',0.25);
axis tight
```

However, this solution is provided in MATLAB, and your code needs to be written in Python, so any use you make of this will need to be translated.

Quadratic regression

Obviously the linear regression technique is going to produce linear (or planar) decision boundaries. A richer structure can be obtained by using other basis functions to build the decision function. Simply including cross terms between independent variables is a straightforward way to do this.

To be a quadratic regressor, we augment the linear regressor as follows. The classifier function is of the form

$$f(\mathbf{x}) = \beta_0 + \sum_{i=1}^{d} \beta_i x_i + \sum_{ij} \beta_{ij} x_i x_j$$

The second summation introduces the quadratic dependency.

We can stack the information as follows. Let $\mathbf{x}_i = [x_1, x_2]^T$ training data point. (This also works in higher dimensions, but we will be specific here.) Form a row of the data matrix $\mathbf{X}$ as

$$\begin{bmatrix} 1 & x_1 & x_2 & x_1^2 & x_1 x_2 & x_2^2 \end{bmatrix}$$

then form the $\mathbf{X}$ matrix as the stack of such rows for all training data. The $\hat{\mathbf{B}}$ matrix has K columns, each of the form

$$\begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \beta_{11} \\ \beta_{12} \\ \beta_{21} \end{bmatrix}$$

so that there are six unknown. Given the $\mathbf{Y}$ matrix as before, the problem has the same structure as before. The least-squares estimate of $\hat{\mathbf{B}}$ is

$$\hat{\mathbf{B}} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{Y}.$$

Problem 4: For the data described in Problem 3, train the regression coefficient matrix $\hat{\mathbf{B}}$. Determine the classification error rate on the training data and 10,000 points of test data (as before) and fill in the corresponding row of the results table. Plot the classification regions as before.

Linear and Quadratic Discriminant analysis

Let the (estimated) mean vector from class i be denoted as $\hat{\boldsymbol{\mu}}_i$, and let the (estimated) covariance matrix from class i be denoted as $Rhat_i$. The mean estimate may be obtained, for example, from

$$\hat{\boldsymbol{\mu}}_0 = \frac{1}{N_0}\sum_{j=1}^{N_0}\mathbf{x}_j$$

(where the data $\mathbf{x}_j$ are all from class 0) with similar modifications for $\boldsymbol{\mu}_1$. The covariance estimate may be obtained, for example, from

$$\hat{R}_0 = \frac{1}{N_0 - 1}\sum_{j=1}^{N_0}(\mathbf{x}_j - \hat{\boldsymbol{\mu}}_0)(\mathbf{x}_j - \hat{\boldsymbol{\mu}}_0)^T.$$

The overall data mean $\hat{\boldsymbol{\mu}}$ and covariance R (averaged over all classes) is

$$\hat{\boldsymbol{\mu}} = \frac{1}{N}\sum_{j=1}^{N}\mathbf{x}_j$$

$$\hat{R} = \frac{1}{N - K}\sum_{k=1}^{K}\sum_{\text{class } k}(\mathbf{x}_i - \hat{\boldsymbol{\mu}}_k)(\mathbf{x}_i - \hat{\boldsymbol{\mu}}_k)^T$$

The estimate of the prior probability of a point being in class k is

$$\pi_k = \frac{N_k}{N}.$$

If the data were Gaussian distributed, then the likelihood function for class k (using the estimated parameters as the true parameters) would be

$$f_k(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}|\hat{R}_k|^{1/2}}\exp\left[-\frac{1}{2}(\mathbf{x} - \hat{\boldsymbol{\mu}}_k)^T\hat{R}_k^{-1}(\mathbf{x} - \hat{\boldsymbol{\mu}}_k)\right].$$

and the log of the posterior probability is equal to (up to a constant which does not depend on the class)

$$\log P(\text{class} = k|X = \mathbf{x}) = \log\hat{\pi}_k + -\frac{1}{2}\log|\hat{R}_k| - \frac{1}{2}(\mathbf{x} - \hat{\boldsymbol{\mu}}_k)^T\hat{R}_k^{-1}(\mathbf{x} - \hat{\boldsymbol{\mu}}_k). \tag{2}$$

Based on this, let

$$\delta_k^q(\mathbf{x}) = \log\hat{\pi}_k + -\frac{1}{2}\log|\hat{R}_k| - \frac{1}{2}(\mathbf{x} - \hat{\boldsymbol{\mu}}_k)^T\hat{R}_k^{-1}(\mathbf{x} - \hat{\boldsymbol{\mu}}_k) \tag{3}$$

(The ^{q} is for quadratic.)

Problem 5: Show that (2) is true. In particular, make sure you understand what is meant by "up to a constant which does not depend on the class."

Note that $\delta_k^q(\mathbf{x})$ is a quadratic function of $\mathbf{x}$.

We can use $\delta_k^q(\mathbf{x})$ as a decision function as follows. For each class $k = 0, 1, \ldots, K-1$, compute $\delta_k(\mathbf{x})$, then select the class which has the largest value of $f_k(\mathbf{x})$. That is,

$$\hat{\mathbf{k}} = \arg\max_k \delta_k^q(\mathbf{x}).$$

This classifier is called the **quadratic discriminant function classifier**. Classification using these functions is called quadratic discriminant analysis (QDA).

Now *impose* the assumption that all of the classes have equal covariance. That is, assume $\hat{R}_0 = \hat{R}_1 = \hat{R}$. Then when comparing values of $\delta_k^q(\mathbf{x})$, the terms $\log|\hat{R}|$ and $\mathbf{x}^T\hat{R}^{-1}\mathbf{x}$ are common to all functions for all values of k, and hence can be eliminated. This gives rise the the function

$$\delta_k^l(\mathbf{x}) = \mathbf{x}^T\hat{R}^{-1}\hat{\boldsymbol{\mu}}_k - \frac{1}{2}\hat{\boldsymbol{\mu}}_k^T\hat{R}^{-1}\hat{\boldsymbol{\mu}}_k + \log\pi_k. \tag{4}$$

(The ^{l} is for "linear.") This function is called the *linear discriminant function*. Classification using these functions is called linear discriminant analysis (LDA).

Problem 6: For the data set described in problem 3, build a LDA classifier. That is, train sample means for each class and a population covariance, and classify based on the linear discriminant functions in (4). Characterize the error rate on the training data and on 10,000 points of test data. Plot the classification regions as before.

Problem 7: For the data set described in problem 3, build a LDA classifier. In this case, you will also need to build the class covariance matrices. Classify based on the quadratic discriminant functions in (3). Characterize the error rate on the training data and on 10,000 points of test data. Plot the classification regions as before. Compare the decision boundaries between QDA and quadratic regression.

If the data truly were Gaussian, then LDA or QDA (depending on the nature of the covariances) would be optimal. However, real world data are rarely Gaussian, so these techniques are simply an approximation to the optimal detector. However, they provide a stable method that has broad applicability.

Linear logistic regression

In linear logistic regression, the ratio of the posterior probability functions is assumed to be a linear function of a measured data vector $\mathbf{x}$:

$$\log \frac{P(\text{class = 0}|X = \mathbf{x})}{P(\text{class = 1}|X = \mathbf{x})} = \beta_0 + \beta^T \mathbf{x}.$$

Based on this, it is straightforward to show that

$$P(\text{class = 0}|X = \mathbf{x}) = \frac{\exp[\beta_0 + \beta^T \mathbf{x}]}{1 + \exp[\beta_0 + \beta^T \mathbf{x}]} \qquad P(\text{class = 1}|X = \mathbf{x}) = \frac{1}{1 + \exp[\beta_0 + \beta^T \mathbf{x}]}$$

If the data in each class are Gaussian with equal covariance, this reduces to the LDA case. But the logistic regression model imposes fewer assumptions on the data.

It will be convenient to let

$$\underline{\beta} = \begin{bmatrix} \beta_0 \\ \beta \end{bmatrix} \qquad \text{and} \qquad \underline{\mathbf{x}} = \begin{bmatrix} 1 \\ \mathbf{x} \end{bmatrix}.$$

Furthermore, let us use the variable Y to represent the class. For class 0, use $Y = 1$, and for class 1 use $Y = 0$. Then the linear logistic model can be expressed as

$$\log \frac{P(Y = 1|X = \mathbf{x})}{P(Y = 0|X = \mathbf{x})} = \underline{\beta}^T \underline{\mathbf{x}}$$

and the corresponding probabilities are

$$P(Y = 1|X = \mathbf{x}) = \frac{\exp[\underline{\beta}^T \underline{\mathbf{x}}]}{1 + \exp[\underline{\beta}^T \underline{\mathbf{x}}]} \qquad P(Y = 0|X = \mathbf{x}) = \frac{1}{1 + \exp[\underline{\beta}^T \underline{\mathbf{x}}]} \tag{5}$$

To use the linear logistic model, the model must be trained for the parameter $\underline{\beta}$. This is accomplished by using the parameterized model in (5), and maximizing the likelihood for this parameterized model. Let $p_k(\mathbf{x}; \underline{\beta})$ denote the probability $P(Y = k|\mathbf{x}; \underline{\beta})$, where the latter argument denotes the parameter value. For N independent observations, the log likelihood function (expressed as a function of the parameter vector) is

$$\ell(\underline{\beta}) = \log \prod_{i=1}^{N} p_{y_i}(\mathbf{x}_i; \underline{\beta}) = \sum_{i=1}^{N} \log p_{y_i}(\mathbf{x}_i; \underline{\beta}),$$

For the two-class case, let us use $p(\mathbf{x}; \underline{\beta}) = p_0(\mathbf{x}; \underline{\beta})$ and $1 - p(\mathbf{x}; \underline{\beta}) = p_1(\mathbf{x}; \underline{\beta})$.

Problem 8: Using the probability model in (5), show that $\ell(\underline{\beta})$ can be written as

$$\begin{aligned} \ell(\underline{\beta}) &= \sum_{i=1}^{N} y_i \log p(\mathbf{x}_i; \underline{\beta}) + (1 - y_i) \log(1 - p(\mathbf{x}_i; \underline{\beta}) \\ &= \sum_{i=1}^{N} y_i \underline{\beta}^T \underline{\mathbf{x}}_i - \log(1 + e^{\underline{\beta}^T \underline{\mathbf{x}}_i}) \end{aligned}$$

To maximize the log-likelihood, we take the gradient with respect to $\underline{\beta}$ and set it equal to 0. (The gradient of the log likelihood function is called the *score function*. We will encounter it again later.)

Problem 9: Show that

$$\frac{\partial \ell(\underline{\beta})}{\partial \underline{\beta}} = \sum_{i=1}^{N} \underline{\mathbf{x}}_i (y_i - p(\mathbf{x}_i; \underline{\beta}).$$

Solving the equation

$$\sum_{i=1}^{N} \underline{\mathbf{x}}_i (y_i - p(\mathbf{x}_i; \underline{\beta}) = 0$$

to maximize is a set of $d+1$ nonlinear equations, which has no closed form expressed. Instead, we use a Newton-Raphson technique. This requires computation of the second derivative of $\ell(\underline{\beta})$, which is called the Hessian matrix.

Problem 10: Show that the Hessian can be computed as follows:

$$\frac{\partial^2 \ell(\underline{\beta})}{\partial \underline{\beta} \partial \underline{\beta}^T} = -\sum_{i=1}^{N} \mathbf{x}_i \mathbf{x}_i^T p(\mathbf{x}_i; \underline{\beta})(1 - p(\mathbf{x}_i, \underline{\beta})).$$

The Newton-Raphson update now works as follows: Given a value of the parameter vector at the mth iteration, $\underline{\beta}^{[m]}$, the updated version is obtained by

$$\underline{\beta}^{[m+1]} = \underline{\beta}^{[m]} - \left(\frac{\partial^2 \ell(\underline{\beta})}{\partial \underline{\beta} \partial \underline{\beta}^T} \right)^{-1} \frac{\partial \ell(\underline{\beta})}{\partial \underline{\beta}}.$$

This can be put into a more streamlined form as follows. Let $\mathbf{y}$ be the $N \times 1$ vector of y_i (class) values. Let $\mathbf{X}$ be the $N \times (d+1)$ matrix of training vectors, as before. Let

$$\mathbf{p}^{[m]} = \begin{bmatrix} p(\mathbf{x}_1; \underline{\beta}^{[m]}) \\ p(\mathbf{x}_2; \underline{\beta}^{[m]}) \\ \vdots \\ p(\mathbf{x}_N; \underline{\beta}^{[m]}) \end{bmatrix}$$

be the vector of likelihoods, evaluated at the parameter $\underline{\beta}^{[m]}$, and let $\mathbf{W}$ be a $N \times N$ diagonal matrix with diagonal elements $p(\mathbf{x}_i; \underline{\beta}^{[m]})(1 - p(\mathbf{x}_i; \underline{\beta}^{[m]}))$.

Problem 11: Show that

$$\frac{\partial \ell(\underline{\beta})}{\partial \underline{\beta}} = \mathbf{X}^T (\mathbf{y} - \mathbf{p}^{[m]})$$

and

$$\frac{\partial^2 \ell(\underline{\beta})}{\partial \underline{\beta} \partial \underline{\beta}^T} = -\mathbf{X}^T \mathbf{W} \mathbf{X}.$$

Show that the Newton-Raphson update step can be written

$$\begin{aligned} \underline{\beta}^{[m+1]} &= \underline{\beta}^{[m]} + (\mathbf{X}^T \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^T (\mathbf{y} - \mathbf{p}^{[m]}) \\ &= (\mathbf{X}^T \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^T \mathbf{W} (\mathbf{X} \underline{\beta}^{[m]} + \mathbf{W}^{-1} (\mathbf{y} - \mathbf{p}^{[m]}). \end{aligned}$$

Letting $\mathbf{z}^{[m]} = \mathbf{X} \underline{\beta}^{[m]} + \mathbf{W}^{-1} (\mathbf{y} - \mathbf{p}^{[m]})$, it can be seen that the effect of this iteration is to solve

$$\underline{\beta}^{[m+1]} = \arg\min_{\underline{\beta}} (\mathbf{z}^{[m]} - X \underline{\beta})^T \mathbf{W} (\mathbf{z}^{[m]} - \mathbf{X} \underline{\beta}),$$

which is a weighted least-squares problem. This algorithm is referred to as *iteratively reweighted least squares* (IRLS). Starting at $\underline{\beta} = \mathbf{0}$ is typically effective, but convergence to a global minimizer is not guaranteed.

Problem 12: For the data set described in problem 3, program the logistic regression classifier. That is, program the IRLS algorithm to determine $\underline{\beta}$ from the training data, then use it to compute the log-likelihood ratio. Use this for classification of the training data and 10,000 points of test data. Plot the classification regions as before. Record the probability of classification error for test and training data on the table.

k-nearest Neighbor Classifier

The k-nearest neighbor classifier operates on the principle that you start to look like the people that you hang around with. That is, the classification of a vector $\mathbf{x}$ should be represented by classifications of neighbors near to it.

Let $N_k(\mathbf{x})$ be the set of the k training vectors $\mathbf{x}_i$ nearest to the point $\mathbf{x}$. Each training input $\mathbf{x}_i$ has a corresponding output (classification) value y_i associated with it. The k-nearest neighbor rule forms a classification function

$$f(\mathbf{x}) = \frac{1}{k} \sum_{x_i \in N_k(\mathbf{x})} y_i.$$

That is, it computes the average of the y values for the k nearest training points to $\mathbf{x}$. If $f(\mathbf{x}) > 0.5$, then the classifier decides that $\mathbf{x}$ is in class 1. If $f(\mathbf{x}) < 0.5$, then the classifier decides that $\mathbf{x}$ is in class 0.

Problem 13: For the data set described in problem 3, program a k-nearest neighbor function. Make it so that you can change the value of k. Use your k-nearest neighbor function for classification of the training data and 10,000 points of test data for $k = 1$, $k = 5$, and $k = 15$. Comment on the probability of error on the training data when $k = 1$. Plot the classification regions. Record the probability of classification error for test and training data on the table.

Naive Bayes Classifier

This technique (sometimes known as the "Idiot's Bayes" classifier) performs well with low complexity. We'll express this in general terms; you can make application to our particular problem.

Suppose that there are J different classes to be classified, each described by a pdf (or pmf) $f_j(\mathbf{X}), j = 1, 2, \ldots, J$. Suppose that the observation vectors $\mathbf{X} = \begin{bmatrix} X_1 & X_2 & \ldots & X_p \end{bmatrix}^T$ are p-dimensional. In general (non-naive) work we would need to perform a multivariate density estimation. This is difficult and requires very large amounts of data for usefully accurate work. In naive Bayes, however, we *assume* that the features X_k are independent, so that

$$f_j(\mathbf{X}) = \prod_{k=1}^{p} f_{jk}(X_k).$$

In practical problems this assumption is rarely if ever true, but it significantly simplifies the estimation.

- "For continuous features, the class-conditioned marginal densities can each be estimated separately using one-dimensional kernel density estimates" (HTF, p. 211).
- "If the components X_j of $\mathbf{X}$ is discrete then a histogram estimate can be used."

Using this assumption we will form the likelihood ratio using class J as the reference. Let G be the true class, and π_ℓ denote the prior probability of the class.

$$\begin{aligned} \log \frac{P(G = \ell|\mathbf{X})}{P(G = J|\mathbf{X})} &= \frac{\pi_\ell f_\ell(\mathbf{X})}{\pi_J f_J(\mathbf{X})} = \log \frac{\pi_\ell \prod_{k=1}^{p} f_{\ell k}(X_k)}{\pi_J \prod_{k=1}^{p} f_{Jk}(X_k)} \\ &= \log \frac{\pi_\ell}{\pi_J} + \sum_{k=1}^{p} \log \frac{f_{\ell k}(X_k)}{f_{Jk}(X_k)} \\ &\triangleq \alpha_\ell + \sum_{k=1}^{p} g_{\ell k}(X_k). \end{aligned}$$

This forms what is called a *generalized additive model*

Key to the naive Bayes approach is finding an estimate of the class-conditional marginal densities f_{jk}. We briefly introduce this concept here.Density estimation is an important topic in its own right. However, a lot of this can be summarized with a couple of concepts. We first form an estimate of the density simply based on a local histogram; this gives a bumpy sort of estimate. This bumpy function is then smoothed to form the *Parzen* estimate.

Let $f_X(x)$ denote a density function to be estimated from a random sample of N points $S = x_1, x_2, \ldots, x_N$. Given this data (only), we desired to estimate $f_X(x)$. We will denote the estimate by $\hat{f}_X(x)$.

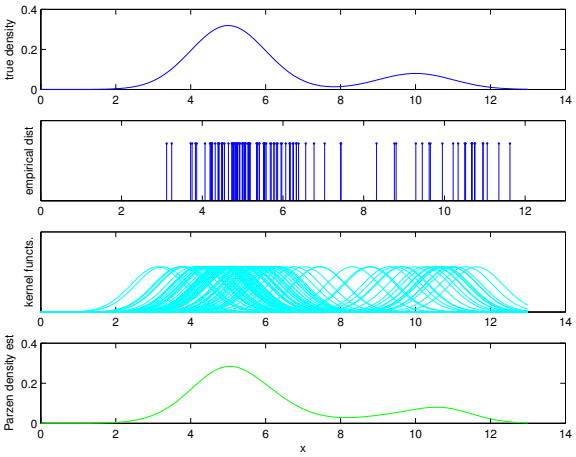

Figure 2: Demonstration of Parzen density estimator

First the bumpy estimate. Let $N_\lambda(x_0)$ denote a neighborhood of sample points around the point x_0, that is, points within a distance λ of x_0:

$$N_\lambda(x_0) = \{x \in S : |x - x_0| < \lambda\}.$$

A (bumpy) estimate of the density can be obtained by

$$\hat{f}_X(x_0) = \frac{\#\{x_i \in N_\lambda(x_0)\}}{N\lambda},$$

where the numerator computes the number of samples in the set $N(x_0)$.

A smoother estimate of the density can be obtained by forming a weighted sum, with a kernel function $K_\lambda(x_0, x_i)$ that decreases smoothly with distance from x_0. The *Parzen* estimate is

$$\hat{f}_X(x_0) = \frac{1}{N\lambda} \sum_{i=1}^{N} K_\lambda(x_0, x_i)$$

A popular choice for the kernel function is a Gaussian function with variance λ

$$K_\lambda(x_0, x_i) = \frac{1}{\sqrt{2\pi\lambda}} e^{-(x_0 - x_i)^2/2\lambda} \triangleq k_\lambda(x_0 - x_i).$$

Then we can express the Parzen density estimate as

$$\hat{f}_X(x_0) = \frac{1}{N} \sum_{i=1}^{N} k_\lambda(x_0 - x_i).$$

There is another way of expressing this. Let $\hat{F}$ denote the sample empirical distribution, which puts probability mass $\frac{1}{N}$ at each of the observed x_i. Then the Parzen density estimate is the convolution

$$\hat{f}_X(x_0) = (\hat{F} * k_\lambda)(x_0).$$

The Parzen density estimate method is demonstrated in figure 2. The top frame shows the true density. The next frame shows the sample empirical distribution — a Kronecker δ of height $\frac{1}{N}$ is placed at the location of each sample drawn from this distribution ($N = 100$ samples). Observe that the sample density is higher where the pdf is larger. The third frame shows the set of kernel functions ($\lambda = 0.5$), one placed at each sample location. The fourth frame shows the sum of these kernel functions.

One of the issues associated with Parzen density estimation is what should the bandwidth λ be. There are theoretical results for this, but they are most valuable when the true density is known (which it is not.) Often it is necessary to play around with various values of λ to get a reasonable result.

Problem 14: The true density shown in figure 2 is a Gaussian mixture,

$$f_X(x) = p_0 N(x, \mu_0, \sigma_0^2) + p_1 N(x, \mu_1, \sigma_1^2),$$

where $N(x, \mu_0, \sigma_0^2)$ means the Gaussian pdf with mean μ_0 and variance σ_0^2. The particular parameters of this mixture are:

$$\begin{array}{lll} p_0 = 0.8 & \mu_0 = 5 & \sigma_0^2 = 1 \\ p_1 = 0.2 & \mu_1 = 10 & \sigma_1^2 = 1 \end{array}$$

Reproduce the results in figure 2. That is, make a plot of the true density, the sample empirical distribution, the kernel functions, and the estimated density. Try different values of λ.

Problem 15: Apply the Naive Bayes estimator to our data set, using the training data in `classassgntrain1.dat` to estimate the densities. Plot the classification regions. Record the probability of classification error for test and training data on the table.

The Optimal Bayes Classifier

For this test data, we can actually write down a distribution function, and hence determine the optimum Bayes classifier.

The data are generated as follows. First, 10 different means were generated for class 0 by drawing from a Gaussian with mean $\left[\begin{smallmatrix}1\\0\end{smallmatrix}\right]$ and covariance I. Then 10 different means were generated by class 1 by drawing from a Gaussian with mean $\left[\begin{smallmatrix}0\\1\end{smallmatrix}\right]$ and covariance I. (These means are shown in the file `gendat2.m`, and are now fixed for consistency's sake.) To draw a data point from a particular class, one of the 10 means is chosen at random with probability 1/10, then perturbed by adding zero-mean Gaussian noise with covariance $\frac{1}{5}I$. This particular model is called a Gaussian mixture model. We will now provide an analytic description of this.

Let the ten means associated with class k be denoted $\mathbf{m}_{k,1}, \ldots, \mathbf{m}_{k,10}$. Let the probability of drawing with mean $\mathbf{m}_{k,i}$ be $\pi_{k,i}$. In our particular problem, $\pi_{k,i} = \frac{1}{10}$. The likelihood (density function) for a point drawn from class k can be written as

$$p_k(\mathbf{x}) = \sum_{i=1}^{10} \pi_{k,i} \mathcal{N}(\mathbf{x}; m_{k,i}, \frac{1}{5} I)$$

where by $\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, R)$ we mean to evaluate the Gaussian density with mean $\mathbf{x}$ and covariance R at the point $\mathbf{x}$:

$$\mathcal{N}(\mathbf{x}; \boldsymbol{\mu}, R) = \frac{1}{(2\pi)^{(d/2)} |R|^{1/2}} \exp[-\frac{1}{2}(\mathbf{x} - \boldsymbol{\mu})^T R^{-1} (\mathbf{x} - \boldsymbol{\mu})].$$

Based on this explicit form for the likelihood, and assuming equal prior probabilities for each class, we can find the optimum (Bayes) decision rule.

Problem 16: For the data set described in problem 3, determine the Bayes error rate on the training data and 10,000 points of test data. Record the probability of classification error for test and training data on the table. Plot the classification regions.

Discussion

You should observe performance differences among the different algorithms. You should be aware that an algorithm which works well on this particular data set may not work well on all data sets. It is well to be familiar with a variety of ways of building classifiers.

Problem 17: Discuss the relative merits of the different classification algorithms on this source of data. Comment on differences in performance between training and test data. Also, comment on operating speed of the classification algorithms (after they have been trained). Summarize what you have learned.

Turn in with this assignment your table of results, plots of data and classification regions, and listings of your PYTHON code.