COMPSCI 527- Homework 2

Due on September 25, 2014

Questions may continue on the back. Type your text. You may write math formulas by hand. What we cannot read, we will not grade.

You may talk about this assignment with others, but do not write down anything while you talk. After that, do the assignment alone. What you hand in must be your work only.

Hand in your solution as a single, stapled paper document at the beginning of class on the due date.

Some questions in this assignment require writing MATLAB code. Please intersperse concise comments with your code whenever useful, but do not go overboard with comments. Try to use matrix notation whenever possible, so as to avoid explicit `for` loops over large arrays. These are very inefficient in MATLAB. Looping over smaller arrays (up to a few hundred entries) is OK.

1. In homework 1 we extensively used the MATLAB function `sum` to compute marginal distributions (and through these also conditional distributions) from joint distributions. For instance, if array `pxy` contains a joint distribution for random variables X and Y (with values of X increasing with the column index of `pxy`), then the marginal distributions of X and Y can be computed as follows:

```
px = sum(pxy, 1);
py = sum(pxy, 2);
```

With continuous probabilities, we cannot do this, because integrals now replace sums. There are two ways to proceed. One is analytic: start with probabilities expressed by formulas (or, in MATLAB, by functions that compute formulas given input values of the random variables) and then integrate by calculus. The other way is numerical: assume that probability distributions are given as arrays of finely spaced samples, and then use numerical integration to approximate the values of the integrals.

This problem explores some of these issues. You may assume that function samples are always given over uniform grids. That is, if values of a function are given for $x = x_0, \ldots, x_m$ and $y = y_0, \ldots, y_n$, then

$$x_i - x_{i-1} = \Delta x \quad \text{for all} \quad i = 1, \ldots, m \quad \text{and} \quad y_j - y_{j-1} = \Delta y \quad \text{for all} \quad j = 1, \ldots, n \; .$$

However, Δx may be different from Δy.

(a) The *trapezoidal rule* computes an approximate definite integral of a function of a single variable as follows

$$\int_a^b f(x)\, dx \approx \frac{\Delta x}{2} \sum_{i=1}^{m} (f(x_{i-1}) + f(x_i))$$

where

$$a = x_0 \, , \quad b = x_m \quad \text{and} \quad x_i - x_{i-1} = \Delta x = \frac{b-a}{m} \; .$$

This computation approximates the function f by a function that is linear between samples, and adds up the area of the trapezoids with vertices $x_{i-1}, x_i, f(x_i), f(x_{i-1})$ for $i = 1, \ldots, m$.

To compute an integral over two variables, one can then use Fubini's theorem

$$\int_a^b dx \int_c^d dy\, f(x,y) = \int_a^b \left[\int_c^d f(x,y)\, dy \right] dx \; .$$

So one first computes the internal integral for every value of x_i by the trapezoidal rule to obtain $m+1$ approximate samples $g(x_0), \ldots, g(x_m)$ of the function

$$g(x) = \int_c^d f(x,y)\, dy \; ,$$

and then one uses the trapezoidal rule again to approximate

$$\int_a^b dx \int_c^d dy\, f(x,y) = \int_a^b g(x)\, dx \; .$$

Write a MATLAB function with header

```
function i = integrate(p, dx, dy)
```

that computes integrals over one or two dimensions using the trapezoidal rule.

Specifically, `p` is either a vector or a two-dimensional array of values, and either or both of the positive scalars `dx` or `dy` are specified. The function `integrate` works as follows:

- If `p` is a vector (you can use `isvector` to check), then `dy` must be either set to the empty matrix `[]` or omitted altogether. Either way, `integrate` approximates the integral of the function whose samples in `p` are assumed to be spaced by an interval Δx equal to the positive scalar `dx`.
- If `p` is a two-dimensional array, `dx` is a positive number, and `dy` is either unspecified or set to the empty matrix, then `integrate` uses the trapezoidal rule to integrate each *row* of `p` (as you would when computing a marginal distribution) to produce a *column* vector of results.
- If `p` is a two-dimensional array, `dx` is is the empty matrix, and `dy` is a positive number, then `integrate` uses the trapezoidal rule to integrate each *column* of `p` to produce a *row* vector of results.
- If `p` is a two-dimensional array and both `dx` and `dy` are set to positive scalar values, then `integrate` uses Fubini's theorem and the trapezoidal rule to integrate the function whose samples in `p` are assumed to be spaced by `dx` and `dy` in the x and y directions. The result in this case is a single number.

Leaving both `dx` and `dy` unspecified, or specifying both of them when `p` is a vector, should cause an error. Use no explicit `for` loops in your code.

Turn in your code for `integrate` and a plot of the result of evaluating the marginal distribution $p(x)$ from the joint distribution $p(x, y)$ computed by running the function `pXYa` (the '`a`' stands for "analytic") provided in the zip file that comes with this assignment on the x and y values generated by the following commands:

```
x = linspace(-0.1, 2.1, 201);
y = linspace(0, 1, 101)';
```

(note that `x` is a row vector and `y` is a column vector). There are various ways to calculate `dx` and `dy` from `x` and `y` for use in `integrate`. Since the spacing in `x` and `y` is regular, it should not matter what calculation you use.

Next to your plot, also show a plot on the same range of `x` values of the function `pXa` provided with this assignment. This function computes the marginal analytically, and you can just call `pXa(x)` to get all the values at once. Label all the axes.

(b) What is the root-mean-square discrepancy per sample ($\|\texttt{pxn} - \texttt{pxa}\|_2/201$) between the two versions (numerical and analytical) of the marginal distribution $p(x)$?

(c) Compute (numerically) the conditional probability $p(y|x)$ from the joint probability density $p(x, y)$ given earlier. Feel free to use either the analytical or the numerical version of the marginal. Wherever the marginal in the definition of $p(y|x)$ is less than

$$\delta = 0.001$$

set the conditional probability equal to zero. Show your MATLAB code snippet and a 20-value contour plot of the result. If the result is in array `pygxn` (`n` for "numerical"), this plot is obtained with the following command:

```
contour(x, y, pygxn, 20, 'Color', 'k')
```

(d) How can you tell immediately from the contour plot in your previous answer that the two random variables X and Y are not independent? State what you would expect the contour plot to look like if X and Y were independent.

(e) Find (numerically) samples of a new joint probability distribution $p'(x, y)$ that has the same marginals in x and y as the distribution $p(x, y)$ from earlier, but where the two random variables are mutually independent. Then compute samples `ppygx` of the conditional probability $p'(y|x)$ from the new joint distribution. Explain your solution, and show your code snippet (starting with `pxya`, `x`, `y`, and `delta` assumed to be already available from before) and the contour plot of the new conditional probability distribution. This plot is obtained with a command similar to the one you used for `pygxn`.

(f) To apply the trapezoidal rule for the computation of a one-dimensional integral takes m sums if the input function is given over $m + 1$ samples. If there are $m + 1$ samples in each of x and y, how many sums does it take to compute a 2-dimensional integral using Fubini's theorem and the trapezoidal rule?

(g) How many steps does it take to compute a d-dimensional integral using Fubini's theorem and the trapezoidal rule?

(h) What does your answer to the previous question suggest about the applicability of Fubini's theorem and the trapezoidal rule to the computation of high-dimensional integrals? The answer to *this* question leads to Monte Carlo integration, a fascinating topic that is beyond the scope of this course.

2. This problem guides you through the implementation of a simple Bayes classifier for handwritten digits. All the required materials are in the zip file that comes with this assignment.

The MNIST handwritten digit database was developed by NYU's Yann LeCun in the late Nineties to provide a benchmark for software that recognizes isolated handwritten digits. The site `http://yann.lecun.com/exdb/mnist/` describes the database, which is contained in the four files in the `data` directory in the homework zip file. An annotated training database is used for learning a classifier, and a similarly annotated testing database is used to evaluate the classifier's performance.

If you make the `code` directory in the zip file your MATLAB working directory and type

```
[train, test] = readMNISTDatabase('../data');
```

then the two resulting structures `train` and `test` will contain the training and testing parts of the database. The `train` structure has a field `train.image` with 60,000 black-and-white images of size 28×28 pixels of type `uint8` arranged in a single array of size $28 \times 28 \times 60{,}000$. Each image shows a single handwritten digit between 0 and 9, properly sized and centered. There is also a structure `train.label` with the corresponding 60,000 labels (each label is a `uint8` number between 0 and 9).

The `test` structure contains an annotated database used for testing. Its format is the same as that of `train`, except that there are 10,000 images and labels.

(a) Assuming you did not delete or move the file `decoder.mat` in the `code` directory, the function `code.m` will compress each of the images in a database into a 64-dimensional vector of numbers. For instance, if you type

```
cimg = code(train.image);
```

the function will produce a $64 \times 60{,}000$ matrix of `double` numbers, each column being the encoding of one of the images in `train.image`.

How many times shorter are the vectors in `cimg`, compared to the original images strung into vectors? The length of a vector is simply the number of its entries.

(b) The function `code` does double-duty. If you type

```
rimg = code(cimg, size(train.image));
```

then the resulting $28 \times 28 \times 60{,}000$ array `rimg` will contain reconstructions of the images in `train.image` from their 64-number compressed representations. So `code` behaves like an encoder if it is called with one argument, and like a decoder if it is called with two (with appropriate contents).

Display pairs of side-to-side images (`train.image(:, :, k(i))`, `rimg(:, :, k(i))`) for all entries `k(i)` of the vector `k` obtained with the following command:

```
[~, k] = unique(train.label);
```

This will give you one pair of images for each distinct digit. No need to show your code, just the 20 pictures. Please try to place all pictures on one page, and let us know where to look for them. These pictures are self-explanatory, so there is no need for captions. To save printer ink, you may want to display the negatives (black on white rather than white on black): If `img` is the image, then display `uint8(255) - img`.

(c) Hopefully, the pictures in your previous answer will have convinced you that the compression achieved by `code` preserves the information in the original images well enough for recognition. If you want to know how this compression works, read section 13.5, *Dimensionality Reduction* in the textbook. However, you do not need to understand this material for this assignment.

You will now use the compressed representations in `cimg` and the labels in `train.label` to estimate the parameters of a generative model for this classification problem. Specifically, the prior $p(w)$ is a categorical distribution (section 3.3 of the textbook) over the 10 labels. The likelihood $p(\mathbf{x}|w)$ of each digit $w \in \{0, \ldots, 9\}$ is assumed to be a normal distribution, defined in equation (5.1) in the textbook. For simplicity, we assume that the covariance matrix Σ_w is diagonal:

$$\Sigma_w = \begin{bmatrix} \sigma_{w1}^2 & 0 & \cdots & 0 \\ 0 & \sigma_{w2}^2 & \ddots & \vdots \\ \vdots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & \sigma_{wd}^2 \end{bmatrix}$$

where d is the dimension of each column $\mathbf{x}$ in `cimg`. This diagonal form is equivalent to assuming that the components of $\mathbf{x}$ are mutually independent given the digit w. Equation (5.1) then simplifies to the following expression:

$$p(\mathbf{x}|w) = \frac{1}{(2\pi)^{d/2}\sqrt{\prod_{i=1}^{d}\sigma_{wi}^2}} \exp\left(-\frac{1}{2}\sum_{i=1}^{d}\frac{(x_i-\mu_{wi})^2}{\sigma_{wi}^2}\right) = \mathrm{Norm}[\boldsymbol{\mu_w}, \Sigma_w] \ .$$

Note the slight notational redundancy in the denominator: Of course,

$$\sqrt{\prod_{i=1}^{d}\sigma_{wi}^2} = \prod_{i=1}^{d}\sigma_{wi} \ ,$$

but perhaps the redundant notation avoids mistakes later.

The vector

$$\boldsymbol{\mu_w} = [\mu_{w1}, \ldots, \mu_{wd}]^T$$

is the mean of the distribution (the superscript T denotes transposition: we view the mean as a column vector, even if we write is as a row vector to save space). It would be wasteful to store Σ_w as a matrix, so we introduce a column vector $\mathbf{s}_w$ that contains the diagonal entries of this matrix:

$$\mathbf{s}_w = [\sigma_{w1}^2, \ldots, \sigma_{wd}^2]^T$$

(variances, not standard deviations!).

Since we have plenty of data, Maximum Likelihood estimation (section 4.4.1) of the $2d$ parameters in $\boldsymbol{\mu_w}$ and $\mathbf{s}_w$ is perfectly adequate. Also, because the components of $\mathbf{x}$ are assumed to be independent of each other given w, we can use the formula (4.12) in the textbook to estimate each entry of $\boldsymbol{\mu_w}$ and formula (4.13) to estimate each entry of $\mathbf{s}_w$. Of course, we need to estimate a pair of vectors $(\boldsymbol{\mu_w}, \mathbf{s}_w)$ for each digit $w \in \{0, \ldots, 9\}$.

Write a MATLAB function with header

```
function [likelihood, prior] = normalModel(X, L)
```

that takes a $d \times n$ matrix `X` of compressed vectors (such as `cimg`) and a row vector `L` of n labels (such as `train.label`) and uses formulas (4.12) and (4.13) in the textbook to produce a structure `likelihood` with two fields `M` and `S`, and a column vector `prior` (replace the symbol I in the formulas with n). Both fields `likelihood.M` and `likelihood.S` are $d \times \ell$ matrices, where $\ell = 10$ is the number of labels (digits). Column `likelihood.M(:, w+1)` is an estimate of the mean $\boldsymbol{\mu}_w$ for digit w. Similarly, column `likelihood.S(:, w+1)` is an estimate of the variances in $\mathbf{s}_w$ for digit w. Turn in your code.

[Hints: Make sure you cast `X` to double, to avoid rounding and overflow issues. The function `normalModel` is rather general, so you don't want to hardwire any number (such as ℓ) in it; one way to obtain a list of all labels (and from it also ℓ) is to write `label = unique(L);`. If you think how to rewrite formulas (4.12) and (4.13) in vector form, you will see that this function can be written with just a loop over the digits by using the built-in functions `mean` and `std` (remember to square standard deviations to obtain variances).]

(d) The function `drawRandom` provided with this assignment draws samples at random from the generative model you just defined. Call the function as follows

```
[sample, slabel] = drawRandom(likelihood, prior, [28 28 64]);
```

to generate 64 sample images from the model, together with labels that tell what likelihood each image was drawn from. Make a single composite of these 64 images (arranged in 8 rows and 8 columns) and show the negative of the result (negative, again, means to display the images in `uint8(255) - sample` instead of those in `sample`). On each image, overlay (with the `text` function) the corresponding `slabel` (make the overlay legible, please). The images will not look like digits, but should share some of the overall morphological features of digits. Perhaps they will each look like various pieces of digits garbled together,with the "right" digit somewhat emphasized. Turn in your composite image.

(e) The square root of the sum in the exponent of the normal distribution,

$$\delta(\mathbf{x}, \boldsymbol{\mu}_w) = \sqrt{\sum_{i=1}^{d}\frac{(x_i-\mu_{wi})^2}{\sigma_{wi}^2}}$$

is called the *Mahalanobis distance* between vectors $\mathbf{x}$ and $\boldsymbol{\mu}_w$ relative to the diagonal covariance Σ_w. This definition generalizes to d dimensions the notion of measuring how many standard deviations a value x is from a reference value μ_w: If $\delta(\mathbf{x}, \boldsymbol{\mu}_w)$ is low, then the likelihood that $\mathbf{x}$ comes from $\mathrm{Norm}[\boldsymbol{\mu}_w, \Sigma_w]$ is high, and *vice versa*. We can use this notion to measure how far apart the means $\boldsymbol{\mu}_w$ are from each other by measuring the ℓ^2 quantities $\delta(\boldsymbol{\mu}_w, \boldsymbol{\mu}'_w)$. If this distance is small, then we expect digit w to be confused often with digit w'. Note that the distance δ is not symmetric in its arguments, that is, $\delta(\boldsymbol{\mu}_w, \boldsymbol{\mu}'_w)$ is in general different from $\delta(\boldsymbol{\mu}'_w, \boldsymbol{\mu}_w)$, because the covariance of the second likelihood is used in the formula. So it is possible that digit w may often be mistaken for digit w' and not *vice versa*.

As a first step in computing a matrix of all Mahalanobis distances between the means of your digit model, and also for later use, write a MATLAB function

```
function [v, delta] = normalValue(X, m, s)
```

that takes a $d \times n$ matrix `X` of data, a d-dimensional mean vector `m`, and a d-dimensional vector `s` of variances, and outputs two row vectors `v` and `delta`, each of length n. The vector `v` contains the values—computed at each of the columns of the data matrix `X`—of the normal distribution $\mathrm{Norm}[\mathtt{m}, \Sigma]$, where Σ is the diagonal matrix that has on its diagonal the values in the vector `s`. The output vector `delta` contains the corresponding Mahalanobis distances (do not forget the square root). Turn in your code. [Hint: This requires no explicit `for` loops.]

(f) Write a MATLAB function with header

```
function D = distances(L)
```

that computes the $\ell \times \ell$ matrix of Mahalanobis distances $\delta(\boldsymbol{\mu}_w, \boldsymbol{\mu}'_w)$ between the digit means. Make sure to use the entries in $\mathbf{s}_{w'} = \mathrm{diag}(\Sigma_{w'})$—and not those in $\mathbf{s}_w$—in the definition of δ. Turn in your code, as well as a printout of the matrix `D` for the likelihood you computed with your function `normalModel`. Please keep only the first two digits after the decimal period, to keep the printout small.

(g) Write a MATLAB function with header

```
function label = classify(img, likelihood, prior)
```

that takes an uncompressed array `img` of images, such as `test.image`, and a generative model in the form of a likelihood and prior as specified earlier, and produces a vector `label` with as many entries as `img` has images. Entry `label(k)` is the digit (a number between 0 and 9) that the Bayes classifier with the model developed earlier (on the compressed images) assigns to the image `img(:, :, k)`. Turn in your code.

(h) Write a MATLAB function with header

```
function [E, errorRate, pCgT] = errorStats(computedLabel, trueLabel)
```

that takes as input two vectors of equal length n. The vector `computedLabel` is a set of labels estimated by a classifier, and the vector `trueLabel` is a set of labels, such as those in `test.label`, that are known to be true. The function computes an $\ell \times \ell$ array `E`, a scalar `errorRate` between 0 and 1, and an array of probabilities `pCgT` the same size as `E`.

Specifically, entry `E(i, j)` of `E` is the number of times that an image that was truly of digit `i-1` was classified as digit `j-1` instead (the entries in `E` should add up to n). The scalar `errorRate` is the fraction (not the percentage) of wrong entries in `computedLabel`. Entry `pCgT(i, j)` of array `pCgT` is the conditional probability, estimated from the data in the input vectors, that a digit was classified as digit `i-1` given that the digit was truly `j-1`.

Turn in your code, and the values your function computes on the result of your classifier on the set `test` of test data. Also express the error rate as a percentage. Use three decimal digits after the period for `pCgT`.

[Hints: Both `errorRate` and `pCgT` can be computed from `E`. Computing `E` may require a nested `for` loop on its entries.]

(i) How can you check that `pCgT` is a valid conditional probability and that it represents $p(\mathrm{computed}|\mathrm{true})$, and not $p(\mathrm{true}|\mathrm{computed})$?

(j) State-of-the-art isolated-digit classifiers achieve error rates around 0.2 percent, so they do much better than the very simple classifier developed in this exercise. The improvements come mainly from the design of image features that are much more sophisticated than those our `code` function computes. Assuming that the classifier error rate is p, and that errors on different digits are mutually independent, what is the probability p_Z that the classifier gets a five-digit ZIP code wrong? Evaluate your answer for $p =$ `errorRate` (your computed rate) and for $p = 0.002$ (the best available rate today).

(k) The US Post Office makes about 400 million deliveries each day. How many zip codes would be classified erroneously each day if the state-of-the-art digit classifier were used?

(l) In light of your previous answer, it is useful for your classifier to output not only a label per digit image, but also the value of the posterior $p(\hat{w}|\mathbf{x})$ for the label $\hat{w}$ chosen by the classifier for input feature $\mathbf{x}$. What is the range of values that this posterior can take? Explain your answer.

(m) How could the US Post Office automatic ZIP code scanner use the value of the posterior $p(\hat{w}|\mathbf{x})$? Give a qualitative answer, no need for formulas.

(n) The considerations above assume that errors on adjacent digits are mutually independent. Is this assumption valid? Why or why not?