CS 195-5: Machine Learning
Problem Set 4

Douglas Lanman
dlanman@brown.edu
10 November 2006

1 Support Vector Machines

Problem 1

In this problem we will implement Support Vector Machine (SVM) classification using Radial Basis Function (RBF) kernels. Make the following modifications to the supplied support code to complete the SVM implementation.

1. Write the function `Krbf.m` which takes as its arguments two sets of examples X_N and Z_M (as column vectors) and the RBF kernel width σ and returns the kernel matrix K.
2. Fill in the two missing pieces in `trainSVM.m`: the calculation of $\mathbf{H}$ and the calculation of w_0.

Try to avoid loops and express as much of the computation as possible in matrix-vector notation.

Let's begin by examining the general structure of `trainSVM.m`: the skeleton code which implements SVM training. On lines 20-23 the training samples in `data` are parsed. On lines 25-29 we evaluate an arbitrary kernel matrix (specified by the user as `kernel` with a single parameter `param`). Next, on lines 31-61 we apply Matlab's built-in quadratic programming `quadprog` routine to determine the support vectors by solving for the Lagrange multipliers α with the margin/slack variable trade-off given by `C`. On lines 63-66 we determine w_0 (i.e., the bias parameter). Finally, the SVM data structure is constructed on lines 68-74.

Recall from the lecture on 10/11/06 that, in order to construct a SVM in the non-separable case, we need to solve the dual problem

$$\underset{\alpha}{\text{argmax}} \left\{ \sum_{i=1}^{N} \alpha_i - \frac{1}{2} \sum_{i=1}^{N} \sum_{j=1}^{N} \alpha_i \alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \right\} \tag{1}$$

$$\text{subject to } \sum_{i=1}^{N} \alpha_i y_i = 0, \quad 0 \leq \alpha_i \leq C \text{ for all } i = 1, \ldots, N, \tag{2}$$

where the Lagrange multipliers are given by $\alpha = (\alpha_1, \ldots, \alpha_N)^T$, the labels of the training samples $\{\mathbf{x}_i, y_i\}$ are given by $\mathbf{y} = (y_1, \ldots, y_N)^T$, and the kernel matrix is $K(\mathbf{x}_i, \mathbf{x}_j)$. Also recall, from the Matlab documentation, that `alpha = quadprog(H,f,A,b,Aeq,beq,l,u,x0)` solves

$$\underset{\alpha}{\text{argmin}} \frac{1}{2} \alpha^T \mathbf{H} \alpha + \mathbf{f}^T \alpha, \tag{3}$$

$$\text{such that } \mathbf{A}\alpha \leq \mathbf{b}, \ \mathbf{Aeq}\alpha = \mathbf{beq}, \text{ and } \mathbf{l} \leq \alpha \leq \mathbf{u}. \tag{4}$$

Comparing Equations 3 and 4 to Equations 1 and 2, it is apparent that the following variable definitions must hold: $\mathbf{f} = (-1, \ldots, -1)^T$, $\mathbf{A} = \mathbf{b} =$ `[]`, $\mathbf{Aeq} = \mathbf{y}^T$, $\mathbf{beq} = 0$, $\mathbf{l} = (0, \ldots, 0)^T$, and

$\mathbf{u} = (C, \ldots, C)^T$. (Note that, since $\mathbf{A} = \mathbf{b} = \texttt{[]}$, `quadprog` will ignore the constraint $\mathbf{A}\alpha \leq \mathbf{b}$). Taking a closer look at Equations 1 and 3, we find

$$\underset{\alpha}{\text{argmin}} \frac{1}{2}\alpha^T\mathbf{H}\alpha + \mathbf{f}^T\alpha = \underset{\alpha}{\text{argmax}} \left\{ \sum_{i=1}^{N} \alpha_i - \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i\alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j) \right\}.$$

Given that $\mathbf{f} = (-1, \ldots, -1)^T$, we can reduce this expression to obtain

$$\underset{\alpha}{\text{argmin}} \frac{1}{2}\alpha^T\mathbf{H}\alpha = \underset{\alpha}{\text{argmin}} \frac{1}{2}\sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i\alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j)$$

$$\Rightarrow \alpha^T\mathbf{H}\alpha = \sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i\alpha_j y_i y_j K(\mathbf{x}_i, \mathbf{x}_j),$$

since minimizing the left-hand expression is equivalent to minimizing the right-hand one. Note that $\alpha^T\mathbf{H}\alpha$ is a quadratic form [7] which can be expressed as

$$\alpha^T\mathbf{H}\alpha = \sum_{i=1}^{N}\sum_{j=1}^{N} \alpha_i\alpha_j H_{ij} \quad \Rightarrow \quad H_{ij} = y_i y_j K(\mathbf{x}_i, \mathbf{x}_j).$$

From this expression it is apparent that, in Matlab's matrix-vector notation, `H = (y*y').*K`, where `K` is the kernel matrix and `y*y'` is the outer product of `y` with itself. Note that this form for $\mathbf{H}$ was used on line 34 in `trainSVM.m`.

At this point, we turn our attention to deriving a robust estimator for the bias parameter ω_0. Recall from Problem Set 3 that, for a linear kernel SVM without slack variables, we found

$$w_0 = \frac{1}{N_\mathcal{S}} \sum_{s\in\mathcal{S}} \left(y_s - \sum_{i\in\mathcal{S}} \alpha_i y_i \mathbf{x}_i^T \mathbf{x}_s \right),$$

where $\mathcal{S}$ denotes the set of $N_\mathcal{S}$ support vectors (i.e., the training samples $\{\mathbf{x}_i, y_i\}$ for which $a_i > 0$). From the lecture on 10/11/06 we recall that, if $0 < \alpha_i < C$, then the corresponding support vector $\mathbf{x}_i$ will be on the margin. As a result, we propose the following estimator for w_0 which once again estimates the bias term using the support vectors on the margin.

$$\boxed{w_0 = \frac{1}{N_\mathcal{M}} \sum_{m\in\mathcal{M}} \left(y_m - \sum_{i\in\mathcal{S}} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}_m) \right)}$$

In this expression, we use $\mathcal{M}$ to denote the set of $N_\mathcal{M}$ support vectors that satisfy the margin constraint with equality (i.e., those that satisfy $0 < \alpha_i < C$). Note that this form for w_0 was used on lines 63-66 in `trainSVM.m`.

To complete our modifications, we briefly review the implementation of the RBF kernel used in `Krbf.m`. As required by the problem statement, `K = Krbf(X,Z,d)`, where `K` is the kernel matrix, {`X`, `Z`} are two data sets, and `d` is the standard derivation σ of the Gaussian kernel such that

$$K_{ij} = \exp\left(-\frac{1}{\sigma^2} \|\mathbf{x}_i - \mathbf{z}_j\|^2 \right). \tag{5}$$

Note that the kernel matrix `K` was computed on lines 19-29 by explicitly evaluating Equation 5 in a loop over the samples contained in the smaller of the two data sets.

Problem 2

Train a SVM on `dataTrain` in `dataSVM.mat` using the RBF kernel for $\sigma = \{0.1, 1, 2, 5, 10, 50\}$ and $C = 10$. Use `testSVM.m` to evaluate the performance of the resulting SVMs on `dataTrain`, `dataTest`, and `dataTest2`. Plot the errors as a function of σ, as well as the decision boundaries obtained with each of the six SVMs. Briefly explain the behavior apparent in the results.

Before presenting the results, we briefly review the function of each m-file required to implement SVM classification. First, note that `trainSVM.m` and `Krbf.m` are used to implement SVM training using a RBF kernel. Next, observe that `testSVM.m` and `svmDiscriminant.m` can be used to test a trained SVM classifier; `svmDiscriminant.m` takes a trained SVM classifier and returns a real number, the sign of which can be used to classify a given test sample. The m-file `testSVM.m` evaluates the performance on a test set and uses `plotData.m` and `plotSVM.m` to plot the results.

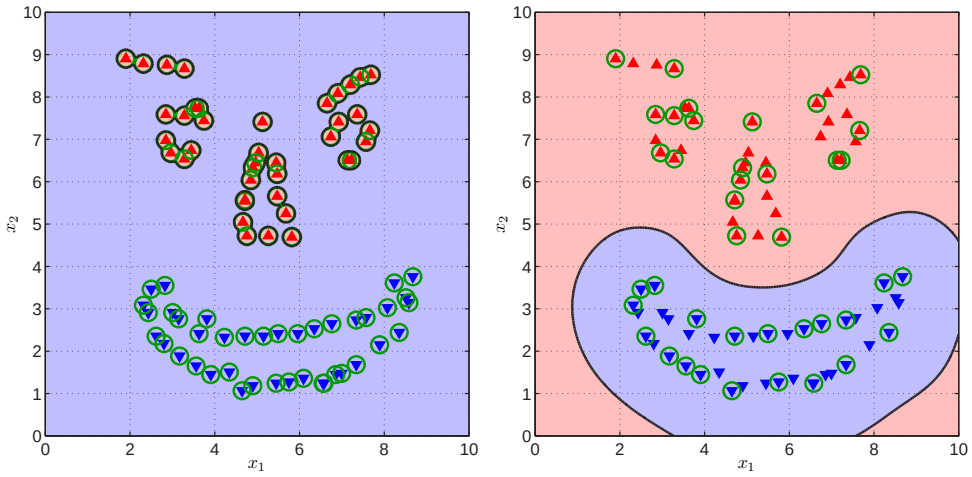
The Matlab script `prob2.m` was used to train SVM classifiers for $\sigma = \{0.1, 1, 2, 5, 10, 50\}$ and $C = 10$ using `dataTrain` from `dataSVM.mat`. After training, the classification errors were estimated on both the original training set, as well as the two test sets: `dataTest` and `dataTest2`. The resulting classification errors are tabulated below. In addition, the decision boundaries, support vectors, and misclassified samples are shown in Figures 1, 2, and 3.

σ	Errors on `dataTrain`	Errors on `dataTest`	Errors on `dataTest2`
0.1	0.0%	33.75%	38.75%
1	0.0%	5.0%	17.5%
2	0.0%	2.5%	13.75%
5	0.0%	3.75%	16.25%
10	0.0%	0.0%	15.0%
50	0.0%	0.0%	3.75%

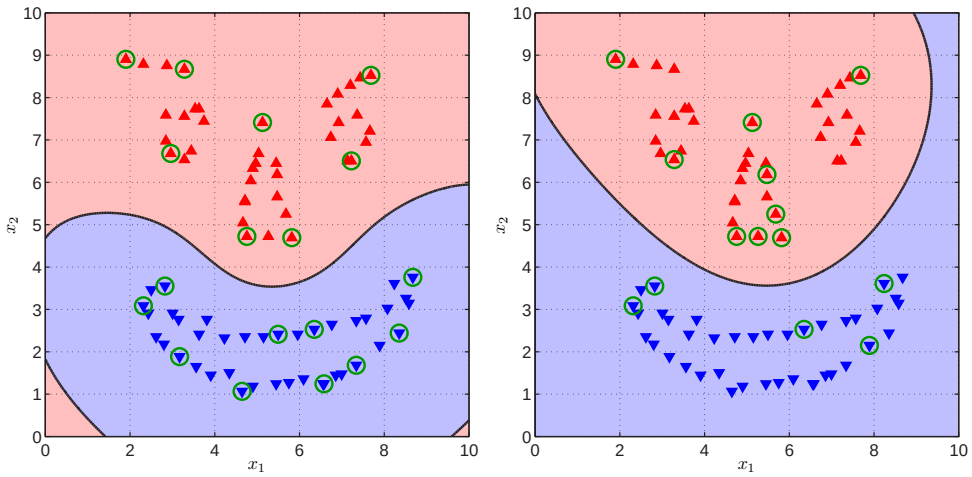
From Figure 1 it is apparent that the standard deviation σ of the RBF kernel has a strong influence on the resulting decision boundary. As discussed in class on 10/11/06, when σ is small we expect every training sample to become a support vector – leading to overfitting. This is confirmed by the results for $\sigma = 0.1$ shown in Figure 1(a). In this example, every point has become a support vector and the SVM decision boundary simply encloses the samples from the $y_i = +1$ class. Since we can interpret the RBF kernel in Equation 5 as a similarity measure, increasing σ tends to allow samples located further apart to be classified as similar. This behavior is apparent in Figure 1. As we increase σ, the decision boundary "relaxes" from a tight bound around the $y_i = +1$ samples to a nearly-linear boundary separating the two classes.

While every value of σ leads to error-free classification of the training samples, the value of σ has a strong impact on the generalization behavior of the SVM. From the tabulated errors, we find that the SVMs with $\sigma = \{10, 50\}$ are error-free when tested using `dataTest`. Similarly, the SVMs with $\sigma = \{10, 50\}$ achieve very low classification errors for `dataTest2`.

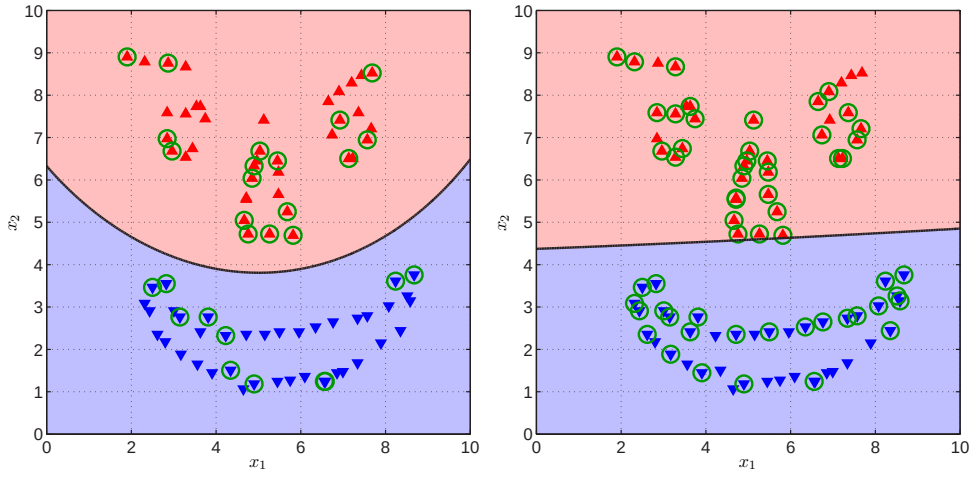
In addition to controlling the "complexity" of the decision boundary, σ also plays an important role in regulating the number of support vectors. As shown in Figure 1, if σ is either too small or too large, then a majority of training samples are selected as support vectors – leading to poor compression of the training data. Alternatively, for $\sigma = \{2, 5\}$ there are fewer support vectors. In conclusion, we find that the selection of σ is a bit of an artform – do you seek few support vectors to compress the data or do you want more to potentially achieve better generalization performance? It remains a task of the SVM architect to balance these demands for a given application.



(a) $\sigma = 0.1$ (b) $\sigma = 1$

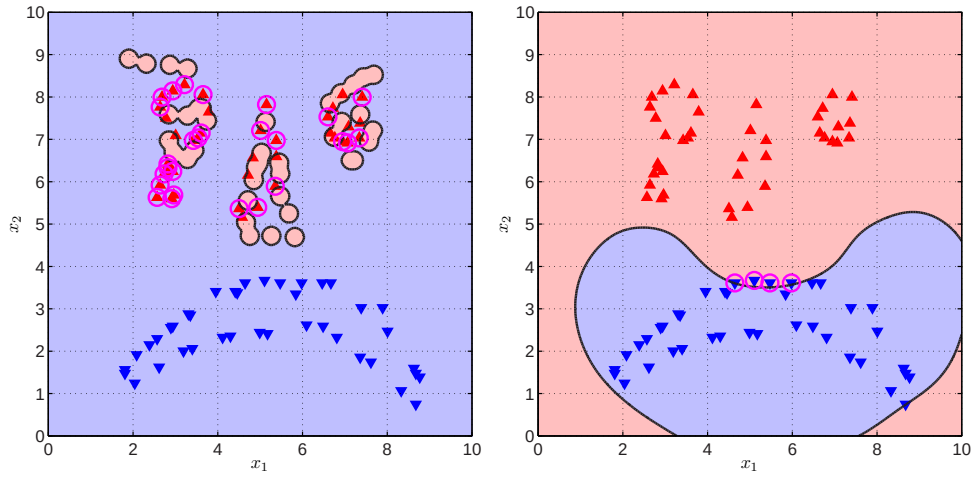


(c) $\sigma = 2$ (d) $\sigma = 5$

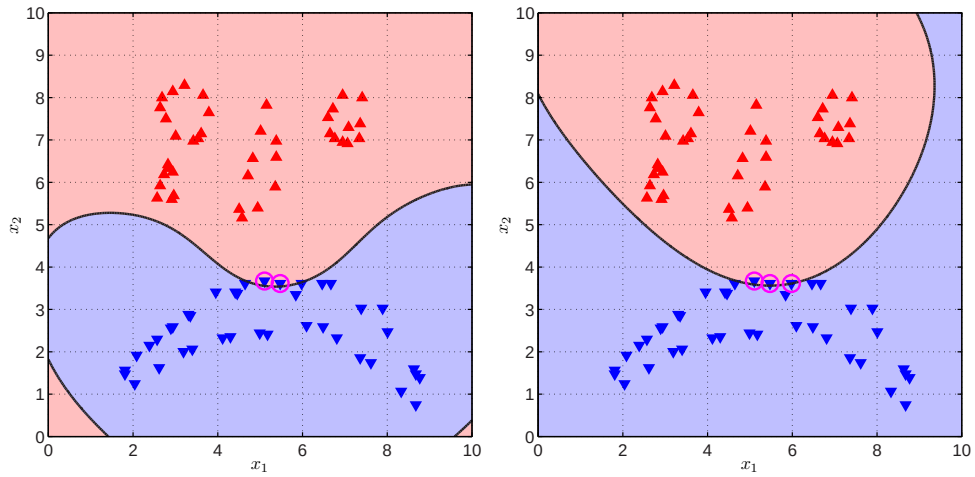


(e) $\sigma = 10$ (f) $\sigma = 50$

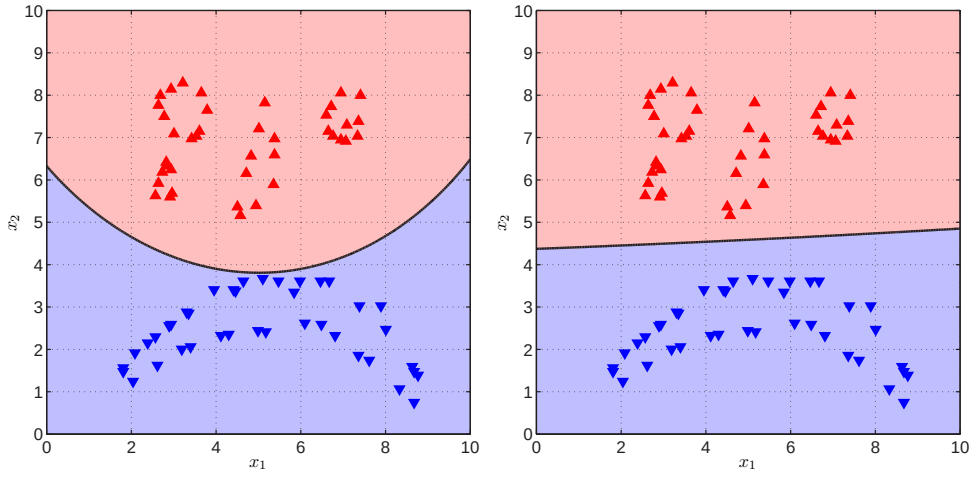
Figure 1: Results of SVM training on `dataTrain` using a RBF kernel. The input samples $\{\mathbf{x}_i, y_i\}$ are drawn from two classes, with $y_i = +1$ denoted by ▲ and $y_i = -1$ denoted by ▼. The support vectors and misclassified samples are indicated by green and magenta circles, respectively. The decision boundary is shown as a black line. Finally, the decision regions are shaded as red and blue.



(a) $\sigma = 0.1$ (b) $\sigma = 1$

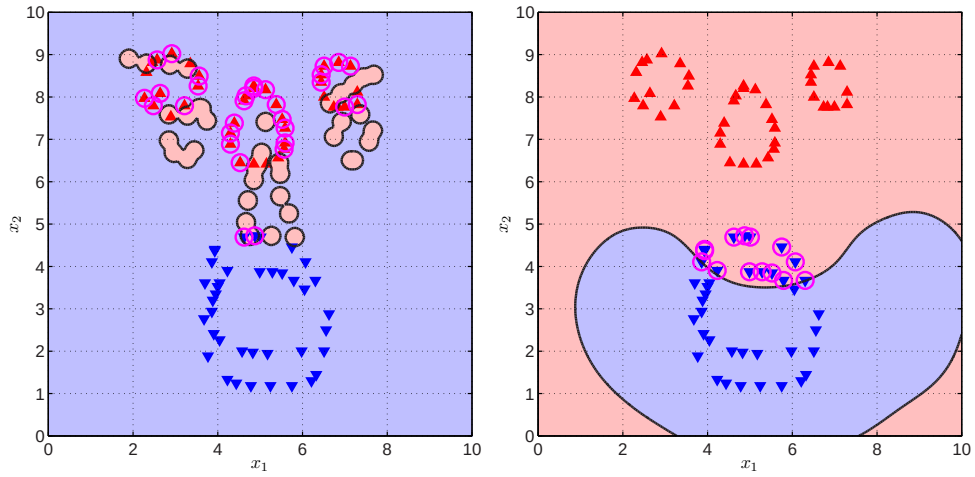


(c) $\sigma = 2$ (d) $\sigma = 5$

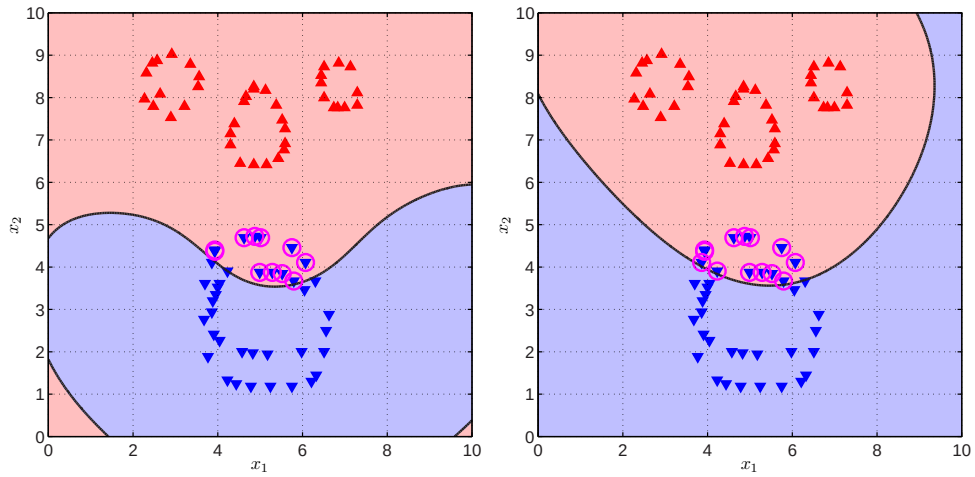


(e) $\sigma = 10$ (f) $\sigma = 50$

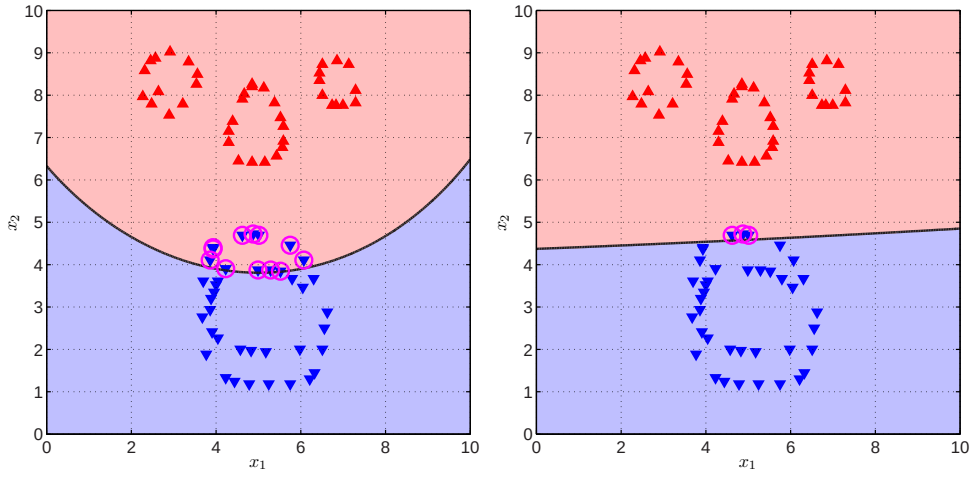
Figure 2: SVM test results for `dataTest` using a RBF kernel. The input samples $\{\mathbf{x}_i, y_i\}$ are drawn from two classes, with $y_i = +1$ denoted by ▲ and $y_i = -1$ denoted by ▼. Misclassified samples are indicated by magenta circles. The decision boundary is shown as a black line. Finally, the decision regions are shaded as red and blue, corresponding to the $+1$ and -1 classes, respectively.



(a) $\sigma = 0.1$ (b) $\sigma = 1$



(c) $\sigma = 2$ (d) $\sigma = 5$



(e) $\sigma = 10$ (f) $\sigma = 50$

Figure 3: SVM test results for `dataTest2` using a RBF kernel. The input samples $\{\mathbf{x}_i, y_i\}$ are drawn from two classes, with $y_i = +1$ denoted by ▲ and $y_i = -1$ denoted by ▼. Misclassified samples are indicated by magenta circles. The decision boundary is shown as a black line. Finally, the decision regions are shaded as red and blue, corresponding to the $+1$ and -1 classes, respectively.

2 Locally Weighted Regression

Problem 3

In this problem we will develop a scheme for locally-weighted regression (LWR) when the number of k nearest neighbors is less than or equal to the dimension of the training data d (i.e., where $k \leq d$). Let $\mathbf{X}'$ be the weighted $k \times d$ design matrix formed from the k nearest neighbors. In this case, the matrix $\mathbf{X}'^T\mathbf{X}'$ is not invertible and, as a result, the least-squares solution using the pseudoinverse is not unique. How do you suggest solving for the regression coefficients?

Let's begin by briefly reviewing locally-weighted regression (LWR). Assume $X_N = \{\mathbf{x}_1, \ldots, \mathbf{x}_N\}$ and $\mathbf{y} = (y_1, \ldots, y_N)^T$ represent the training data points and their labels, respectively. Without loss of generality, we denote the k nearest neighbors (in X_N) of a given test point $\mathbf{x}_0$ as $\mathbf{x}_1, \ldots, \mathbf{x}_k$. In general, we can define the normalized weights $\mathbf{c} = (c_1, \ldots, c_k)^T$ such that

$$c_l = \frac{K(\mathbf{x}_0, \mathbf{x}_l)}{\sum_{m=1}^{l} K(\mathbf{x}_0, \mathbf{x}_m)}, \tag{6}$$

where $K(\mathbf{x}, \mathbf{x}')$ is a user-defined kernel function (e.g., the Gaussian kernel in Equation 5). For a given regression scheme, we can form the weighted design matrix $\mathbf{X}'$ and weighted label vector $\mathbf{y}'$ using the unweighted quantities $\mathbf{X}$ and $\mathbf{y}$ as

$$\mathbf{X}' = \mathbf{C}\mathbf{X} \quad \text{and} \quad \mathbf{y}' = \mathbf{C}\mathbf{y}, \quad \text{where} \quad \mathbf{C} = \begin{pmatrix} 1 & & & \\ & c_1 & & \\ & & \ddots & \\ & & & c_k \end{pmatrix}. \tag{7}$$

For example, in the case of simple linear regression, we have

$$\mathbf{X}' = \begin{pmatrix} c_1 & c_1 x_1 \\ c_2 & c_2 x_2 \\ \vdots & \vdots \\ c_k & c_k x_N \end{pmatrix} \quad \text{and} \quad \mathbf{y}' = \begin{pmatrix} c_1 y_1 \\ c_2 y_2 \\ \vdots \\ c_k y_N \end{pmatrix}.$$

In general, the LWR prediction for the point $\mathbf{x}_0$ is obtained as the least-squares solution given by

$$\hat{y}_0 = \hat{\mathbf{w}}^T \mathbf{x}_0, \quad \text{where} \quad \hat{\mathbf{w}} = (\mathbf{X}'^T\mathbf{X}')^{-1}\mathbf{X}'^T\mathbf{y}'. \tag{8}$$

Note that Equation 8 can only be applied when the inverse of $\mathbf{X}'^T\mathbf{X}'$ exists. Recall from [6] that, in order for the inverse of an $N \times N$ matrix to exist, the matrix must have full rank (i.e., the number of linearly independent rows or columns must be equal to N). In addition, we recall Sylvester's inequality which states that the rank of the product $\mathbf{AB}$ of two matrices $\mathbf{A}$ and $\mathbf{B}$ is at most $\min(\text{rank}(\mathbf{A}), \text{rank}(\mathbf{B}))$. Combining these previous statements, we find that the rank of $\mathbf{X}'^T\mathbf{X}'$ is at most the rank of $\mathbf{X}'$ which, in turn, is bounded by $\min(k, d)$. As a result, if the number of k nearest neighbors is less than or equal to the dimension d of the training data, then the $d \times d$ matrix $\mathbf{X}'^T\mathbf{X}'$ will be rank deficient and therefore noninvertible.

As described, we seek a scheme for solving *rank-deficient* least-squares regression problems (i.e., the case where $k \leq d$). Inspired by previous problem sets, we propose a solution using regularization.

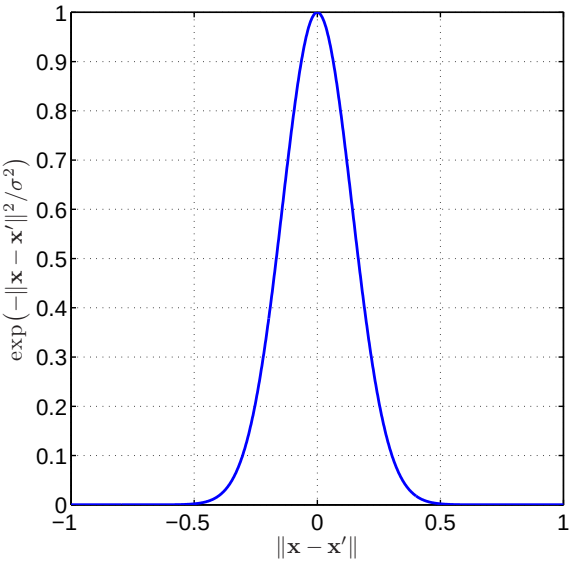


Figure 4: Plot of the Gaussian kernel function $K(\mathbf{x}, \mathbf{x}')$ given by Equation 5.

Recall (e.g., from Problems 1, 2, and 3 in Problem Set 3) that regularization is an efficient scheme for limiting the complexity of a model and preventing overfitting. In this context, we note that there is insufficient data to fit the desired regression model and, as a result, we should limit the model complexity in a similar manner. Recall that the least-squares solution in Equation 8 satisfies

$$\hat{\mathbf{w}}_{LSQ} = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} (y_i' - \mathbf{w}^T \mathbf{x}_i')^2.$$

In order to ensure that $\mathbf{X'}^T\mathbf{X'}$ is well-conditioned (i.e., that it has an inverse), we propose applying ridge regression (RR) such that

$$\hat{\mathbf{w}}_{RR} = \underset{\mathbf{w}}{\operatorname{argmin}} \sum_{i=1}^{N} (y_i' - \mathbf{w}^T \mathbf{x}_i')^2 + \lambda \sum_{j=0}^{d} w_j^2.$$

From Problem 1 in Problem Set 3, we know that the solution of this equation yields an expression for the regularized prediction $\hat{y}_0$ given by

$$\boxed{\hat{y}_0 = \hat{\mathbf{w}}_{RR}^T \mathbf{x}_0, \quad \text{for} \quad \hat{\mathbf{w}}_{RR} = (\mathbf{X'}^T\mathbf{X'} + \lambda\mathbf{I})^{-1}\mathbf{X'}^T\mathbf{y'},} \tag{9}$$

where $\lambda \geq 0$ is a user-defined regularization parameter. In conclusion, we find that ridge regression once again provides an efficient means for controlling model complexity and preventing overfitting (in this case in a situation where insufficient data is available to fit the desired model).

Problem 4

Part 1: Write a function `lwrLinear.m` that implements locally-weighted regression (LWR) in 1D as an extension of the k nearest-neighbor regression (k-NN) routine `knnLinear.m`. This function should fit a linear model to the weighted neighbors of a test point $\mathbf{x}_0$; the weights should be assigned using the Gaussian kernel with fixed width $\sigma = 0.2$. Document the modifications to `knnLinear.m`.

Let's begin by reviewing the structure of `knnLinear.m`. Lines 12-18 handle basic input parsing, whereas lines 20-23 use `findnn.m` to evaluate the k nearest neighbors for each test sample. (Note that `findnn.m` uses the subroutine `lpnorm.m` which, by default, computes the L_1 norm between input samples.) Lines 25-44 implement basic k-NN regression. Note that, on line 38, the solution in Problem 3 is used when $k \leq d$; otherwise, the standard least-squares estimate is used on line 40.

Following the scheme in Problem 3, `knnLinear.m` was modified to produce `lwrLinear.m`. Similar to `knnLinear.m`, lines 13-19 of `lwrLinear.m` parse the input. Lines 21-24 evaluate the k nearest neighbors using `findnn.m`. Note, however, the addition of the LWR weight computation on lines 32-34 (which follow directly from Equation 6 using the Gaussian kernel in Equation 5). To complete the LWR modifications, the weighting scheme from Equation 7 is implemented on lines 36-38.

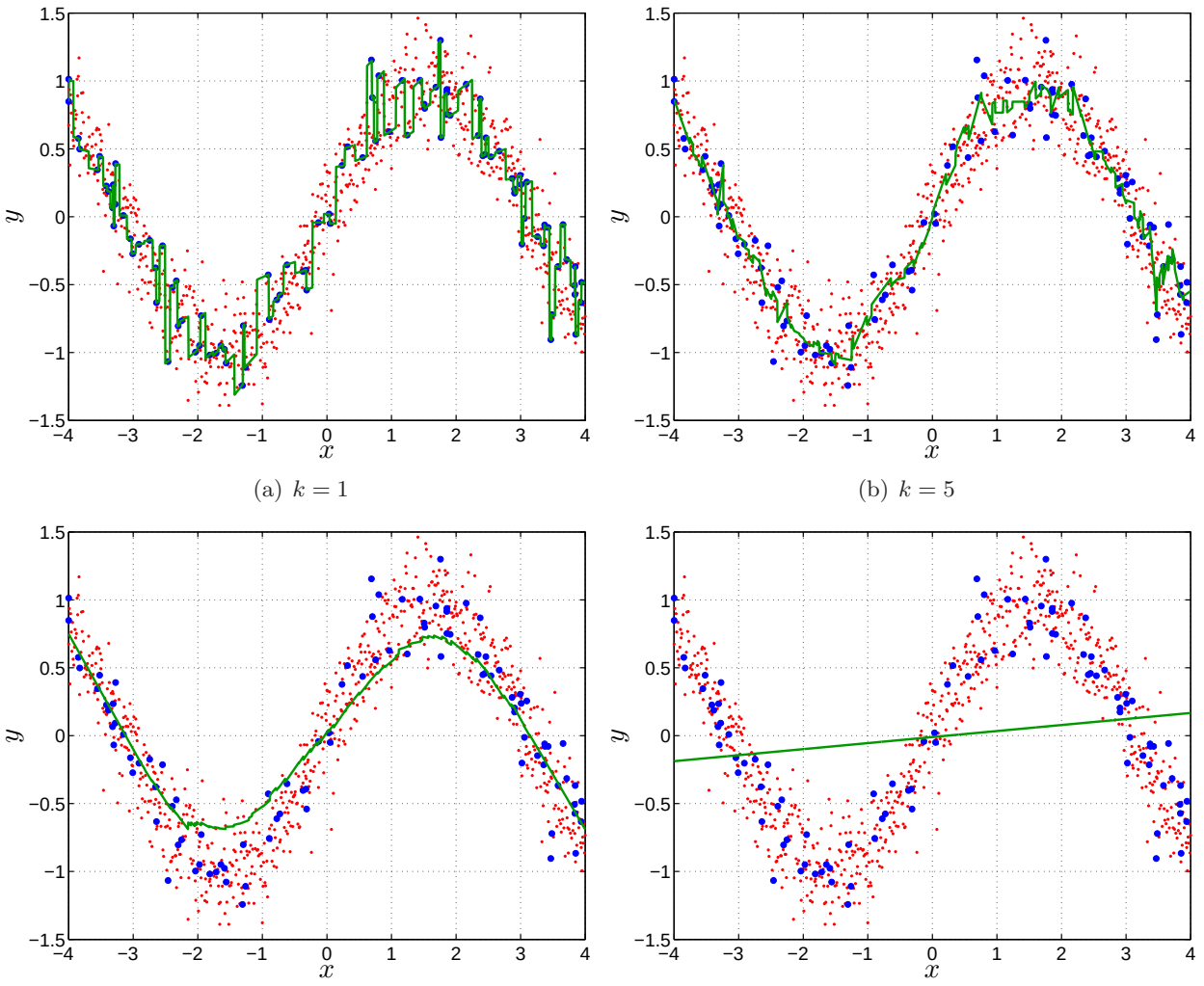
Part 2: Using the training and test data in `lwrData.mat`, evaluate the LWR and k-NN models for $k = \{1, 5, 30, 100\}$. For each k, plot the observed values of y and the fit obtained with both regression models. Report the mean-squared prediction errors and briefly summarize the results.

Using `prob4.m` the k-NN and LWR linear regression models were applied to the data in `lwrData.mat`. The resulting mean-squared prediction errors for each model are tabulated below. In addition, the observed values are compared to the regression models in Figures 5 and 6.

k	MSE for k-NN	MSE for LWR
1	0.0865	0.0865
5	0.0564	0.0586
30	0.0805	0.0551
100	0.4471	0.0551

From the tabulated results, it is apparent that k has a strong impact on k-NN regression. As shown in Figure 5, as we increase k, the model changes from a piecewise-continuous sinusoidal approximation in (a) to a line segment in (d). As confirmed by the tabulated errors, $k = 5$ achieves the lowest error on the test set. This is expected, since the corresponding regression in Figure 5(b) appears to be the best approximation to the underlying distribution. As a result, we find that k-NN is very sensitive to the value of k. As we increase k, we initially smooth the interpolating function; however, if k becomes too large, then the fit smooths over too large a region.

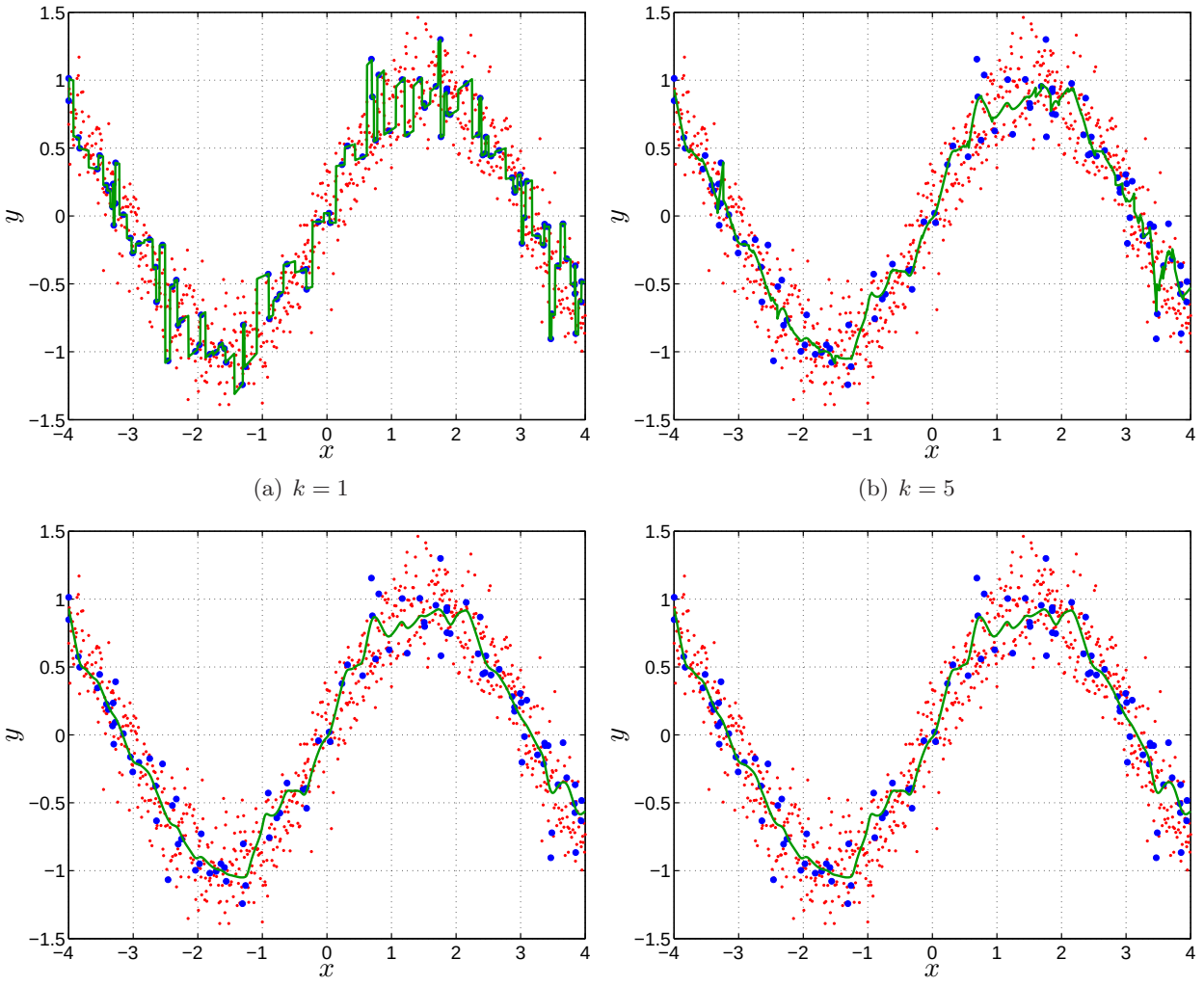
In contrast to k-NN, LWR regression is less sensitive to the number of neighbors k. As shown in Figure 6, as we increase k, the regression model converges to a smooth approximation. This behavior can best be understood by examining the Gaussian kernel in Figure 4; note that the weights fall off rapidly outside of several standard deviations (where $\sigma = 0.2$). Since neighboring training samples are separated by a distance of 0.1, only about 5 neighbors will obtain significant weights. This behavior is confirmed by the tabulated errors; for $k \geq 5$, the resulting error does not change significantly. In conclusion, LWR is less sensitive to k, but very sensitive to the value of σ. As a result, both schemes require adjusting tuning parameters, however LWR tends to produce smoother interpolating functions with minimum additional computation.



(a) $k = 1$ (b) $k = 5$

(c) $k = 30$ (d) $k = 100$

Figure 5: Results of k nearest neighbor linear regression on the training data set in `lwrData` for $k = \{1, 5, 30, 100\}$. The training data points $\{\mathbf{x}_i, y_i\}$ are denoted by •, whereas the test data points are denoted by •. The estimated interpolating function, for each k, is indicated by a green line.



(a) $k = 1$ (b) $k = 5$

(c) $k = 30$ (d) $k = 100$

Figure 6: Results of locally-weighted linear regression (LWR) on the training data in `lwrData` for $k = \{1, 5, 30, 100\}$. The training data points $\{\mathbf{x}_i, y_i\}$ are denoted by •, whereas the test data points are denoted by •. The estimated interpolating function, for each k, is indicated by a green line.

3 The EM Algorithm

Problem 5

Part 1: In this problem we will derive and implement the EM algorithm for a discrete space in which the observations are d-dimensional binary vectors. We will model distributions in this space as a mixture of k multivariate Bernoulli distributions, such that

$$p(\mathbf{x}|\theta, \mathbf{p}) = \sum_{i=1}^{k} p_i p(\mathbf{x}|\theta_i) = \sum_{i=1}^{k} p_i \prod_{j=1}^{d} \theta_{ij}^{x_j} (1 - \theta_{ij})^{1-x_j}, \tag{10}$$

where the i^{th} component of the mixture is parametrized by $\theta_i = (\theta_{i1}, \ldots, \theta_{id})^T$ and the mixing probabilities are subject to $\sum_{i=1}^{k} p_i = 1$. Begin by writing the exact expression for the posterior probability $p(z_{ic}|\mathbf{x}_i; \theta, \mathbf{p})$ in terms of $\mathbf{x}_i$ and the elements of θ and $\mathbf{p}$. Implement the resulting expression in `estep.m` and fill in the missing code for the component parameter updates in `mstep.m`.

Let's begin by introducing the set of binary indicator variables $\mathbf{z}_i = (z_{i1}, \ldots, z_{ik})^T$, where

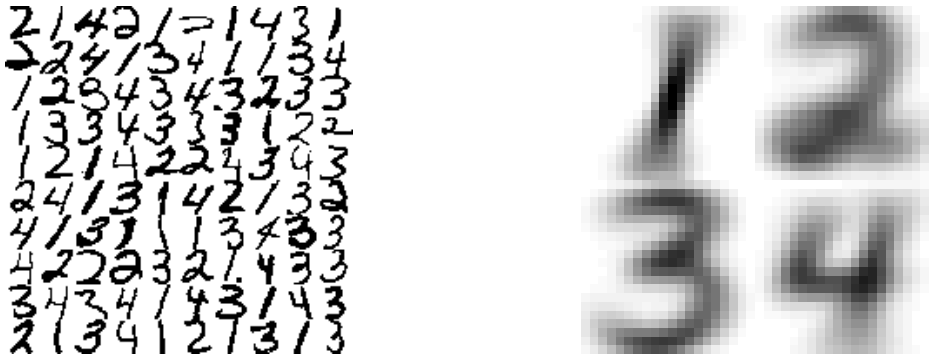
$$z_{ic} = \begin{cases} 1 & \text{if } y_i = c, \\ 0 & \text{otherwise.} \end{cases}$$

At this point, we recall that Bayes' Theorem is given by

$$\Pr(E_i \,|\, B) = \frac{\Pr(E_i \cap B)}{\Pr(B)} = \frac{\Pr(E_i) \Pr(B \,|\, E_i)}{\sum_{j=1}^{k} \Pr(E_j) \Pr(B \,|\, E_j)},$$

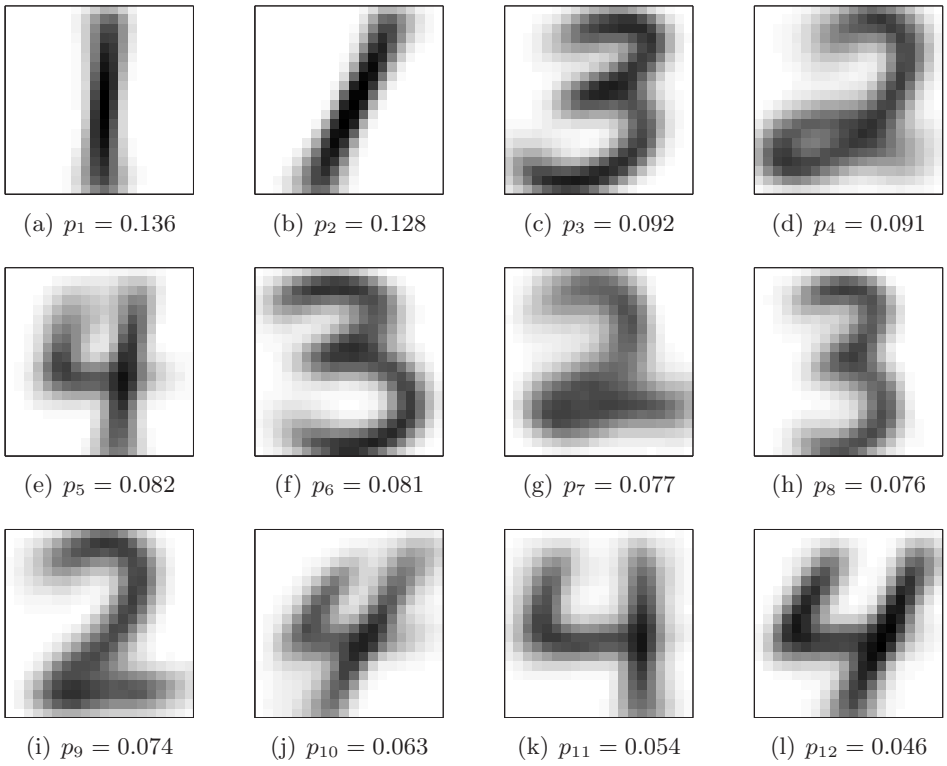
where $E_1, E_2, \ldots, E_k$ are mutually disjoint sets such that $\bigcup_{i=1}^{k} E_i = E$ [5]. We can apply Bayes' Theorem to derive the desired expression for the posterior probability $p(z_{ic}|\mathbf{x}_i; \theta, \mathbf{p})$ as follows.

$$\boxed{p(z_{ic}|\mathbf{x}_i; \theta, \mathbf{p}) = \frac{p(z_{ic})p(\mathbf{x}_i|\theta_c)}{\sum_{j=1}^{k} p(z_{ij})p(\mathbf{x}_i|\theta_j)} = \frac{p_c p(\mathbf{x}_i|\theta_c)}{\sum_{j=1}^{k} p_j p(\mathbf{x}_i|\theta_j)}, \text{with } p(\mathbf{x}|\theta_i) = \prod_{j=1}^{d} \theta_{ij}^{x_j} (1 - \theta_{ij})^{1-x_j}} \tag{11}$$



(a) Sample of the MNIST handwritten data set (b) Class averages for `digitTrain`

Figure 7: Samples and class averages obtained from a subset of the MNIST data set.



(a) $p_1 = 0.136$ (b) $p_2 = 0.128$ (c) $p_3 = 0.092$ (d) $p_4 = 0.091$

(e) $p_5 = 0.082$ (f) $p_6 = 0.081$ (g) $p_7 = 0.077$ (h) $p_8 = 0.076$

(i) $p_9 = 0.074$ (j) $p_{10} = 0.063$ (k) $p_{11} = 0.054$ (l) $p_{12} = 0.046$

Figure 8: The 12 components θ_c and their mixing probabilities p_c for the Bernoulli mixture model, as estimated by applying the EM algorithm to the training data in `digitDataBinary.mat`.

Note that this expression is equivalent to the one derived in class on 11/1/06, as well that on page 446 in [2]. To complete our discussion of the basic EM algorithm for Bernoulli mixture models, we note that the following update equations were provided for the M-step.

$$p_c^{(t+1)} = \frac{N_c^{(t)}}{N} \text{ and } \theta_c^{(t+1)} = \frac{1}{N_c^{(t)}} \sum_{i=1}^{N} \mathbf{x}_i p(z_{ic}|\mathbf{x}_i; \theta^{(t)}, \mathbf{p}^{(t)}), \text{ with } N_c^{(t)} = \sum_{i=1}^{N} p(z_{ic}|\mathbf{x}_i; \theta^{(t)}, \mathbf{p}^{(t)}) \quad (12)$$

The general EM algorithm for Bernoulli mixtures was implemented using `runem.m`. The E-step, given by Equation 11, was implemented on line 18 of `estep.m`. The M-Step, defined by Equation 12, was implemented on lines 13-25 of `mstep.m`. The subroutines `logLikelihood.m` and `conditional.m` were used to compute the log-likelihood given by the logarithm of Equation 10.

Part 2: Run your implementation of the EM algorithm on `digitTrain` using $k = 12$ components. Plot the resulting Bernoulli mixture parameters θ_c for each component. Briefly discuss the results.

The Matlab routine `prob5.m` was used to estimate the Bernoulli mixture model using the training data in `digitDataBinary.mat`. The resulting components are shown in Figure 8. In comparison to the simple class averages shown in Figure 7(b), the Bernoulli mixture model represents a greater degree of variation within the underlying training data. For instance, from the training data excerpt shown in Figure 7(a), it is apparent that the first two components of the mixture model capture the two prominent styles for the number one: vertical and slanted. In conclusion, we find that the EM algorithm has provided an effective unsupervised learning method for discovering both the types of numerals present in the handwriting samples, as well as their major variants.

Problem 6

Part 1: Write a Matlab function that performs Bayesian Information Criterion (BIC) model selection with Bernoulli mixture EM. Using this function, apply the EM algorithm separately to the training data for each digit class from 1 to 4. Let $\theta_c = \{\theta_{c1}, \ldots, \theta_{ck}\}$ and $\mathbf{p}_c = \{p_{c1}, \ldots, p_{ck}\}$ be the estimated mixture parameters for class c, where k is the number of mixture components selected based on BIC. Plot the resulting Bernoulli parameters θ_c for each class-specific mixture.

Rather than explicitly selecting the number of mixture components, as was done in Problem 5, we can choose the mixture model which maximizes the Bayesian Information Criterion (BIC) given by

$$\text{BIC}(\theta, X_N) \triangleq \log p(X_N; \theta) - \frac{\pi(\theta)}{2} \log N, \tag{13}$$

where $\pi(\theta)$ is the number of free parameters in the model. As described in [3], the Bernoulli mixture model (BMM) satisfies

$$\pi_{BMM}(\theta) = k(d+1) - 1, \tag{14}$$

where each dimension has k free parameters (one for each mixture component) and the mixing probabilities yield an additional $k-1$ free parameters (since the normalization requirement can always be used to eliminant one mixing probability).

The Matlab routine `prob6.m` was used to implement BIC model selection for the training data in `digitDataBinary.mat`. The main section implementing model selection is on lines 50-100. Note that Equations 13 and 14 are implemented on line 72 of `prob6.m`. The resulting plots of BIC scores and log-likelihoods are shown in Figure 9. Note that, without the penalty term due to model complexity, the log-likelihood does not appear to saturate. If we consider the BIC scores, however, it is apparent that the selection criterion leads to an optimal model corresponding to the where the BIC score begins decreasing (denoted by black circles in Figure 9). For the provided training data, we report the following number of mixture components: $k_1 = 4, k_2 = 8, k_3 = 7, \text{and} k_4 = 7$. The corresponding plots of the Bernoulli mixture components are shown in Figures 10-13.

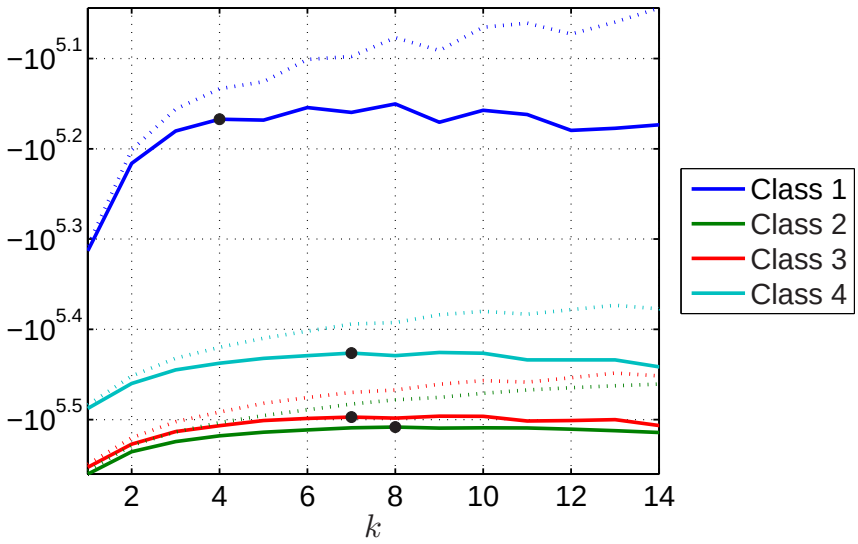
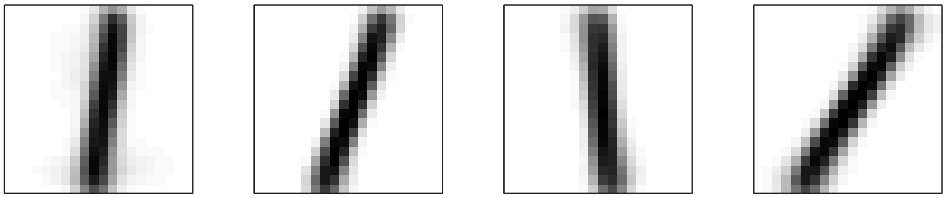
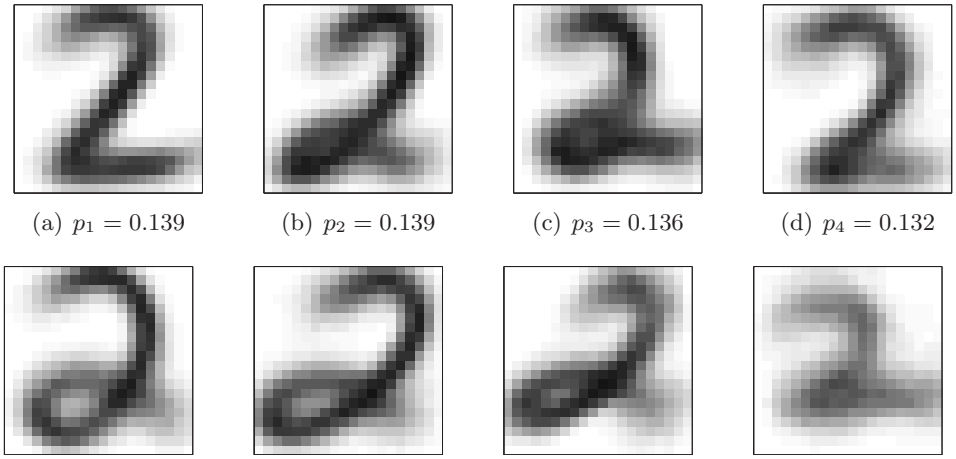


Figure 9: Illustration of BIC model selection. The BIC and log-likelihood are denoted as solid and dashed lines, respectively. The selected number of mixture components are shown as black circles.



(a) $p_1 = 0.277$ (b) $p_2 = 0.273$ (c) $p_3 = 0.232$ (d) $p_4 = 0.218$

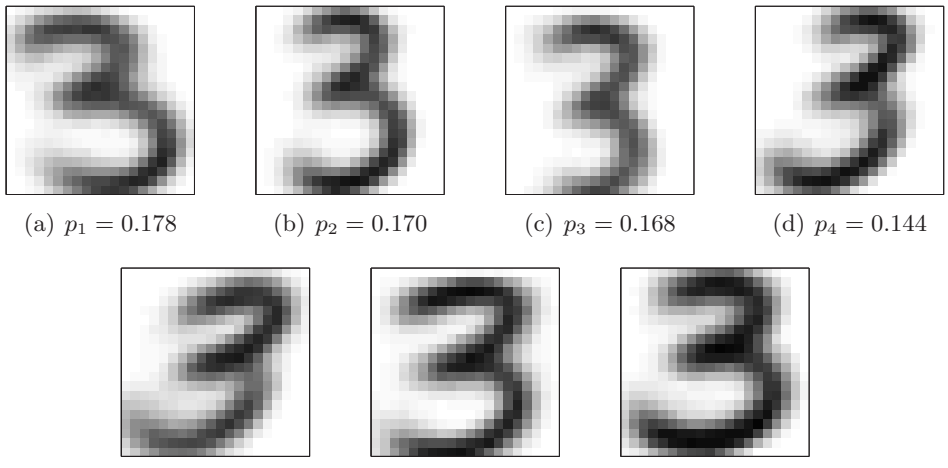
Figure 10: The 4 components θ_c and their mixing probabilities p_c for the Bernoulli mixture model, as found by applying BIC model selection to the first digital class in `digitDataBinary.mat`.



(a) $p_1 = 0.139$ (b) $p_2 = 0.139$ (c) $p_3 = 0.136$ (d) $p_4 = 0.132$

(e) $p_5 = 0.130$ (f) $p_6 = 0.122$ (g) $p_7 = 0.104$ (h) $p_7 = 0.098$

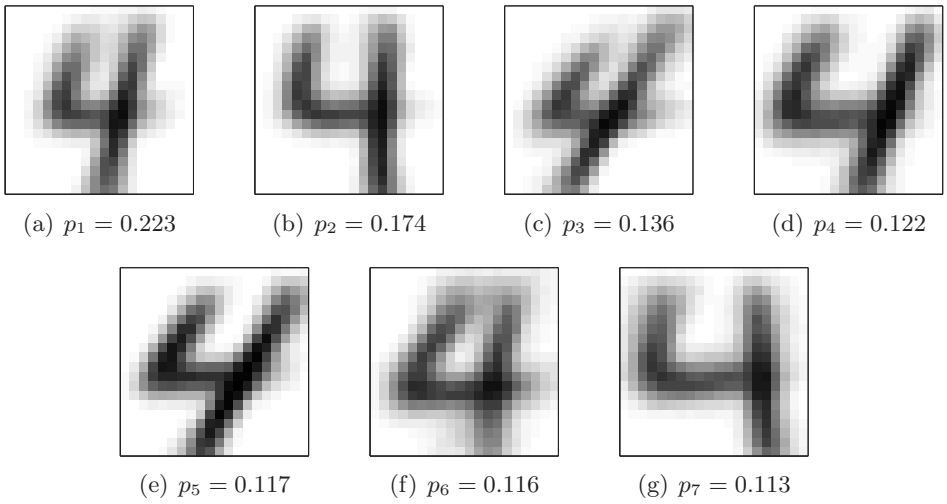
Figure 11: The 8 components θ_c and their mixing probabilities p_c for the Bernoulli mixture model, as found by applying BIC model selection to the second digital class in `digitDataBinary.mat`.



(a) $p_1 = 0.178$ (b) $p_2 = 0.170$ (c) $p_3 = 0.168$ (d) $p_4 = 0.144$

(e) $p_5 = 0.122$ (f) $p_6 = 0.113$ (g) $p_7 = 0.106$

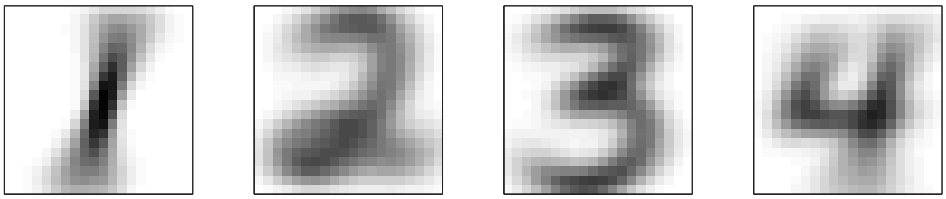
Figure 12: The 7 components θ_c and their mixing probabilities p_c for the Bernoulli mixture model, as found by applying BIC model selection to the third digital class in `digitDataBinary.mat`.



(a) $p_1 = 0.223$ (b) $p_2 = 0.174$ (c) $p_3 = 0.136$ (d) $p_4 = 0.122$

(e) $p_5 = 0.117$ (f) $p_6 = 0.116$ (g) $p_7 = 0.113$

Figure 13: The 7 components θ_c and their mixing probabilities p_c for the Bernoulli mixture model, as found by applying BIC model selection to the fourth digital class in `digitDataBinary.mat`.



(a) (b) (c) (d)

Figure 14: ML estimates of the digit class-conditionals modeled as a single Bernoulli distribution.

Part 2: Now write down the expression of the optimal Bayes classifier, under the assumption that the estimated mixture model is accurate (and assuming equal priors for each digit class). Implement this classifier in Matlab. In addition, construct a "baseline" classifier that models each class-conditional as a single Bernoulli distribution. Report the test error obtained with these two classifiers on `digitTest` and briefly explain the results.

First, recall from Problem 3 in Problem Set 2, that the ML estimator for a Bernoulli random variable X is given by $\hat{\theta} = \frac{1}{N}\sum_{i=1}^{N} x_i$, where $p(X|\theta) = \theta^X(1-\theta)^{1-X}$. Generalizing to d dimensions, we have $p(\mathbf{x}|\theta_i) = \prod_{i=1}^{d} \theta_i^{x_i}(1-\theta_i)^{1-x_i}$ and the corresponding ML estimator is given by

$$\hat{\theta}_i = \frac{1}{N}\sum_{j=1}^{N} x_{ij}, \text{ for } i = \{1, \ldots, d\}. \tag{15}$$

In other words, in order to estimate the "baseline" model with a single Bernoulli component, we simply average the observations in each class. The resulting distributions are tabulated it Figure 14.

At this point we recall, from class on 9/20/06, that the optimal Bayes classifier is given by

$$h^*(\mathbf{x}) = \underset{c}{\operatorname{argmax}} \left\{ \log p_c(\mathbf{x}) + \log P_c \right\}. \tag{16}$$

From Equation 10 we also know that, for the Bernoulli mixture model, we have

$$p(\mathbf{x}|\theta_c, \mathbf{p}_c) = \sum_{i=1}^{k} p_{ci} p(\mathbf{x}|\theta_{ci})$$

Substituting this expression into Equation 16 and assuming the priors are equal (i.e., $P_c = 1/4$), we obtain the following expression for the optimal Bayes classifier for the Bernoulli mixture model.

$$\boxed{h^*_{BMM}(\mathbf{x}|\theta, \mathbf{p}) = \underset{c}{\operatorname{argmax}} \left\{ \log p(\mathbf{x}|\theta_c, \mathbf{p}_c) - \log 4 \right\}} \tag{17}$$

Similarly, if we substitute the estimates obtained in Equation 15, then we obtain the following form for the optimal Bayes classifier in the "baseline" case (i.e., where we have a single mixture component with probability $p_c = 1$).

$$\boxed{h^*_{baseline}(\mathbf{x}|\theta) = \underset{c}{\operatorname{argmax}} \left\{ \log p(\mathbf{x}|\theta) - \log 4 \right\}} \tag{18}$$

The classifiers in Equations 17 and 18 were implemented in `prob6.m` on lines 155-211 and lines 214-240, respectively. Each classifier was evaluated using the test data in `digitDataBinary.mat`; the results are tabulated below.

Test Data	Single Bernoulli Distribution Error	Bernoulli Mixture Model Error
All Classes	6.35%	2.90%
Class 1	6.00%	3.00%
Class 2	11.6%	4.00%
Class 3	5.40%	2.20%
Class 4	2.40%	2.40%

First, note that the baseline results are similar to those we obtained using linear logistic regression in Problem 9 of Problem Set 2. Once again, we find that digit 2 is the most difficult to classify. For the baseline classifier, we report an overall error rate of 6.35%. Not surprisingly, we find that the Bernoulli mixture model significantly outperforms the baseline model – with an overall error rate of 2.9%. This improvement can be directly attributed to the quality of the models represented in Figure 10-13 compared to those in Figure 14. As with logistic regression, we find that a single Bernoulli distribution cannot separate the digit classes robustly. That is, many pixels share high likelihoods across the classes, whereas the mixture model is capable of distinguishing the variations within a class and, as a result, achieves better classification results. In conclusion, we find that unsupervised learning has benefits for both clustering and classification tasks.

References

[1] Christopher M. Bishop. *Neural Networks for Pattern Recognition*. Oxford University Press, 1996.

[2] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer-Verlag, 2006.

[3] Miguel Á. Carreira-Perpiñán and Steve Renals. Practical identifiability of finite mixtures of multivariate bernoulli distributions. Neural Computation, 2000.

[4] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (Second Edition)*. Wiley-Interscience, 2000.

[5] Michael Mitzenmacher and Eli Upfal. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[6] Gilbert Strang. *Introduction to Linear Algebra*. Wellesley-Cambridge Press, 2003.

[7] *PlanetMath.org*. Quadratic form. `http://planetmath.org/encyclopedia/QuadraticForm`.