Homework 3

Daniel Potapov

# Question 1

## Data Distribution

For this question, I opted to generate the data distribution randomly, applying the constraints of the question. I randomly selected 4 vertices of the cube centered around the origin and used the spectral theorem to randomly generate covariance matrices that had an eigen decomposition within the desired range:

NUM\_CLASSES = 4;

%choose 4 of 8 possible cube vertices without replacement

vertexChoices = datasample(0:7, 4, 'Replace', false)';

vertexChoicesBinary = de2bi(vertexChoices);

for i = 1:NUM\_CLASSES

gmmParams.meanVectors(:,i) = (ones(1, 3)-2\*vertexChoicesBinary(i,:))';

%we can take advantage of the spectral theorem to generate random

%covariance matrices with desired eigenvalues (A=Q'DQ)

%generate 3 eigenvalues for the covariance matrix in the range

%[0.5,1.4]

eigs = rand(1,3)\*0.9 + 0.5;

D = diag(eigs);

%now we can create a set of random matrices with a 3 dimensional image

%space and get an orthonormal basis for this space. If the image is not

%3 dimensional, try again until it is

Q = 0;

while rank(Q) ~= 3

A = rand(3);

Q = orth(A);

end

gmmParams.covMatrices(:,:,i) = Q\*D\*Q';

gmmParams.priors(i) = 1/NUM\_CLASSES;

end

Here is a visual representation of the data distribution using the 100-sample training set on the right. The generated covariance matrices create a reasonable amount of overlap between the class-conditional distributions.

## Optimal Classifier

The actual distribution of the dataset was stored in the *gmmParams* variable above, so a theoretically optimal classifier could use those distributions to form a MAP decision rule with a 0-1 loss matrix. With equal class priors, this problem simplified to a maximum likelihood classification.

for i = 1 : length(labels.V100k)

%evaluate likelihood of sample in each class

for j = 1 : 4

likelihood(j) = evalGaussian(data.V100k(:,i), trueParams.meanVectors(:,j), trueParams.covMatrices(:,:,j));

end

[~, classifiedLabels(i)] = max(likelihood);

end

%once classified, find p(error). This will be our target performance of the

%MLP

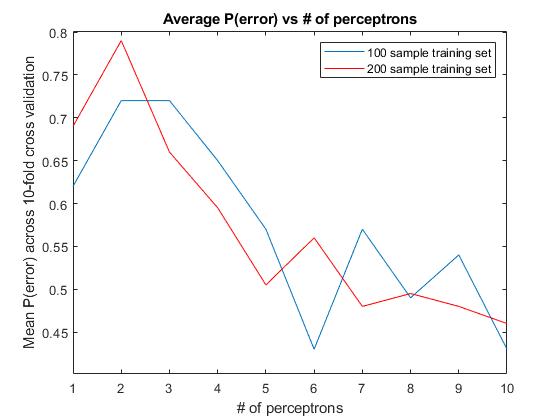
optimalPerror = length(find(labels.V100k ~= classifiedLabels))/length(labels.V100k);

For the distribution generated above, the empirical estimate for the optimal probability of error was 0.20579. Given that a classifier that was given the true model could not do better than this, we do not expect any classifier that needs to estimate or train a model to do better.

## Cross Validation

Each training set generated for this problem may perform better or worse for a neural network with the same number of perceptrons in the hidden layer. Therefore, we performed 10-fold cross validation to find the best perceptron count for each data set.

After trying many ranges of perceptron counts for each data set, it seemed as if for all sets, the average probability of error in classification trended downwards as the number of perceptrons increased. Here we see that downward trend with the 100-sample and 200-sample training sets:



With this behavior, it wasn’t surprising to see that nearly all training sets preferred to be classified with neural networks with 10 perceptrons, the maximum amount tested during cross validation, in the hidden layer.

NumPerceptrons = [6 10 10 10 10 10]

### Neural Network Packages and Verification

During cross validation and training, I opted to use the *patternnet* network object that comes with the MATLAB’s Deep Learning Toolbox. Although there had been some reported issues that *patternnet* would not work properly for this problem, I had no issues using it. Something I did need to do manually change the representation of class labels from a scalar number (1 to 4) to a vector with length 4, where the element of the vector corresponding to the class of the sample was set, and all other elements were cleared. With these inputs, *patternnet* produced outputs that matched the dimensions of my class vectors rather than trying to interpret the meaning of the scalar class labels.

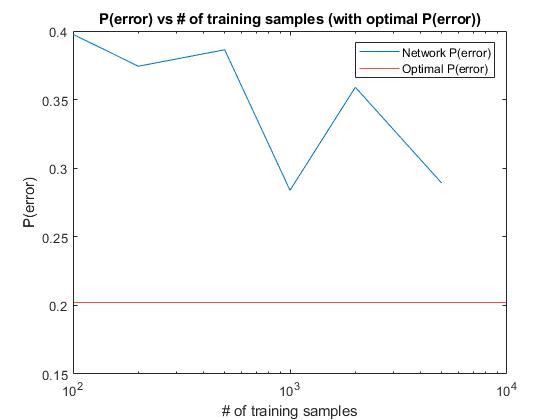
During cross validation, I wanted the only variable between tests to be the number of perceptrons used and the subset of the data used to train the network. To do this, I turned off initial randomization of network weights and biases by setting MATLAB’s random number generation to the default seed of 0. To ensure that my networks behaved the way I wanted them to, I took advantage of the convenient GUIs that the Deep Learning Toolbox provided, which showed me the weights, biases, and overall structure of the network. As cross validation and training ran, I manually inspected the network using this GUI until I was confident that the tool was providing me the functionality I expected.

## Model Training

With the appropriate number of perceptrons selected for each training set, neural networks were trained, one for each set. I used the gradient descent with momentum algorithm provided by the toolbox to optimize the parameters of each network. Since there could be local optima that the network could get stuck in during optimization, I trained the network with different initial weights and biases 10 times and chose the best performing network among them. Indeed, for multiple runs of training a network with the same training set, the probability of error for each network could vary considerably, meaning many of these networks were getting trapped in local optima (by defining a sufficiently large maximum iteration count, I was confident that network training was not stopped prematurely by this count).

## Performance Assessment

For each of the iterations described in the model training above, I assessed the performance of the neural network by running it on the 100 thousand sample validation set and calculating the probability of error for each network. Once the best network was chosen for each training set, I could compare the performance of each network:

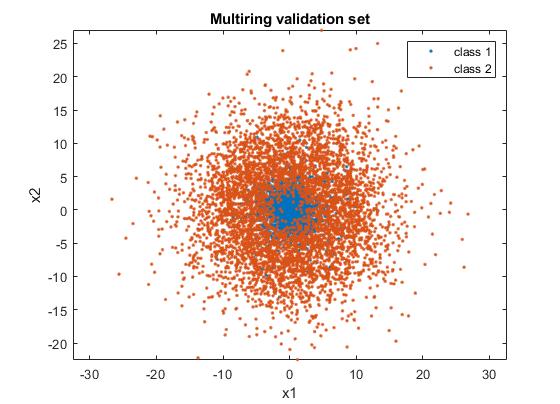


The results seem to match expectation: as the number of training samples went up, the probability of error trended downwards since a larger training set is more likely to accurately represent the distribution. None of the networks exceeded the performance of the optimal classifier, which was also to be expected.

# Question 2

## Data Distribution

In this problem, data was distributed as 2 classes in concentric circles with some overlap in between:



## Cross Validation

Unlike in question 1, now we were cross validating 2 hyperparameters instead of 1, the box constraint and the sigma parameter of the RBF kernel. I based my cross-validation method on the assertion that both of these parameters were independent of one another, and therefore I could select them one at a time instead of in a combination. First, I kept the sigma constant while varying the box constraint value, and then kept the box constraint constant while varying the sigma value, performing 10-fold cross validation on both parameters.

### Support Vector Machine Packages and Verification

For cross validation and training, I used a *ClassificationSVM* object provided by MATLAB, which had the provided functionality to feed in data and get a binary classification. To train the SVM, I used the provided *fitcsvm* function in which I could provide training data and set all the parameters I wanted regarding the SVM kernel. In this case, I could select a Gaussian kernel function, the box constraint, and the RBF gamma value.

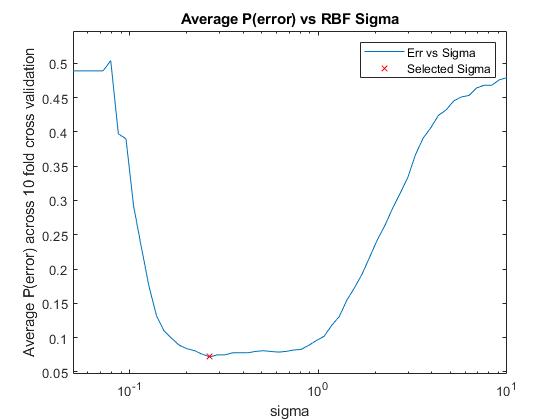
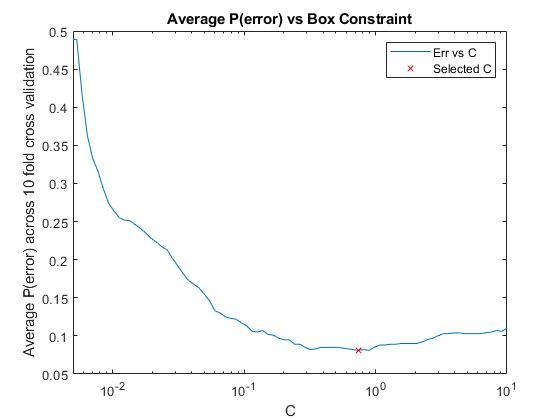
In this package, the gamma value was the only way to manipulate the Gaussian kernel scale:

However, this is not exactly the form of the expression that the class notes use for the RBF kernel, which uses a sigma value:

Therefore, to use sigma with *fitcsvm*, I had to make the following conversion when feeding the sigma parameter into the SVM training function:

Throughout cross validation and training, I could print a textual view of the SVM object and manually confirm that the parameters I was trying to set were being used as expected. I am confident of the correct performance of this package given the manual testing and the good performance quality after training.

Below are the results from cross validation to select C and sigma. It is interesting how in this problem, unlike question 1, both hyper-parameters seem to have a clear optimum where the probability of error is minimized. Seeing these minima increased my confidence in the selection of these parameters (on the other hand, I wasn’t sure whether or not testing higher and higher perceptron counts in question 1 would continue to yield lower average probabilities of error).



Best Sigma = 0.2656 Best C = 0.7391

## Training and Performance

Now having the optimal hyper-parameters from cross validation, I trained a final SVM model using the entire training set:

svm = fitcsvm(data.train', labels.train, 'KernelFunction', 'rbf', 'KernelScale', 1/(2\*bestSig^2), 'BoxConstraint', bestC);

Using the provided *predict* method, I passed in the validation set to the SVM classifier and compared it to the validation labels. The performance of the classifier turned out to be quite accurate, with an empirical P(error) of 0.0707. Below you can see the misclassifications of the final model:

