EECE5644: Assignment #2

Due on February 25, 2020

Professor Deniz Erdogmus

Anja Deric

GitHub Repo: https://github.com/AnjaDeric/MachineLearning

Problem 1

Part One
The parameters of the class-conditional Gaussian pdfs in problem 1 are as follows:

Class 0

$$\mu_0 = \begin{bmatrix} -2 \\ 0 \end{bmatrix} \qquad \Sigma_0 = \begin{bmatrix} 1 & -0.9 \\ -0.9 & 2 \end{bmatrix} \qquad P(L=0) = 0.9$$

Class 1

$$\mu_1 = \begin{bmatrix} 2 \\ 0 \end{bmatrix} \qquad \Sigma_1 = \begin{bmatrix} 2 & 0.9 \\ 0.9 & 1 \end{bmatrix} \qquad P(L=1) = 0.1$$

For part 1 of problem 1, 10000 samples were generated based on these parameters, and used to implement a minimum P(error) classifier, which is specified in the equations below.

$$\frac{g(x|m_1, C_1)}{g(x|m_0, C_0)} \gtrless \frac{P(L=0)}{P(L=1)}(\frac{\lambda_1 0 - \lambda_0 0}{\lambda_0 1 - \lambda_1 1}) = \frac{0.9}{0.1}(\frac{\lambda_1 0 - \lambda_0 0}{\lambda_0 1 - \lambda_1 1})$$

$$(D=1) \qquad \frac{g(x|m_1, C_1)}{g(x|m_0, C_0)} \gtrless 9(\frac{\lambda_1 0 - \lambda_0 0}{\lambda_0 1 - \lambda_1 1}) = \gamma \qquad (D=0)$$

The discriminant scores for the two pdfs could then be calculated as $log(g(x|m_1, C_1)) - log(g(x|m_0, C_0))$, and compared to a varying threshold in order to create the ROC curve. In my implementation, the threshold value (γ) was calculated as the mid-point value of all the discriminant scores. Figure 1 below shows the ROC curve generated after storing the probability of false alarm and correct detection for each value of γ.

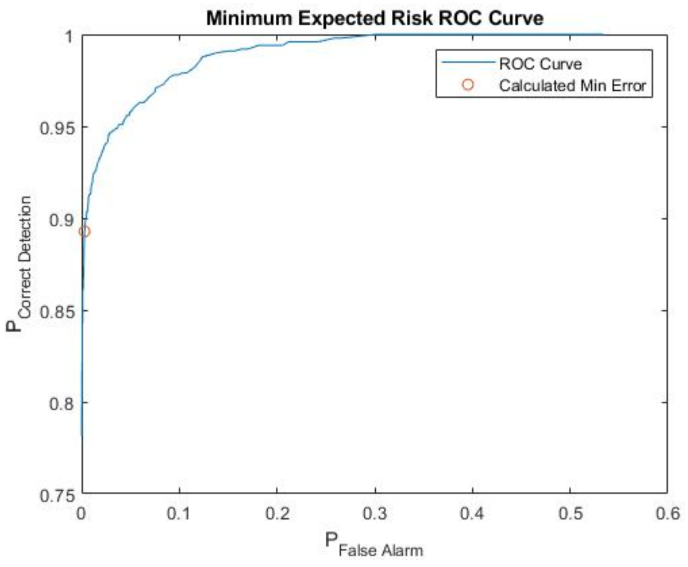


Figure 1: Problem 1 Minimum Expected Risk ROC Curve

The orange circle indicates the point at which the probability of error is at its minimum. In this case, the minimum probability of error was calculated to be 1.41% at the threshold value of 2.08.

Problem 1 continued on next page. . .

Figure 2 below shows the decision boundary generated when this threshold value was implemented and evaluated.

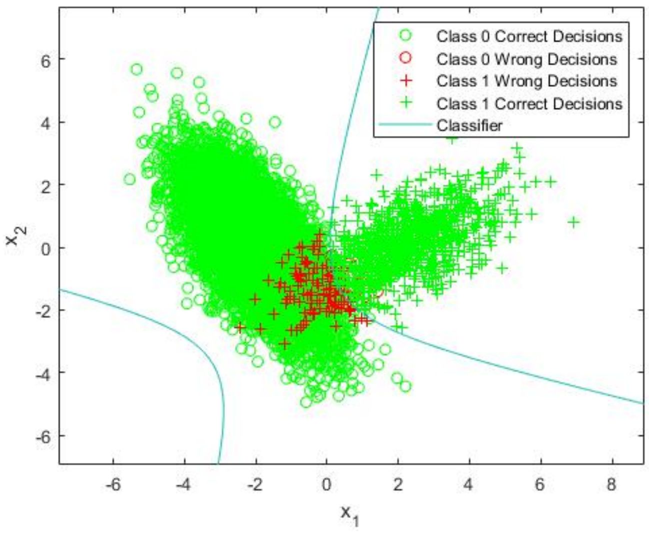


Figure 2: Minimum Error Decision Boundary and Classification

Part Two

For part 2 of the problem, 3 separate logistic-linear-function-based approximations were trained on a set of 10-, 100-, and 1000-sample newly generated training sets. The input parameters x were first mapped to a feature vector $[1, x_1, x_2]^T$ since we were dealing with linear regression. With parameter vector theta first initialized to all zeros, the fminsearch function was used to minimize the negative-log-likelihood of each training set. The cost function minimized in this case is shown below, where l is the label of each data point, x is the actual (2-dimensional) data point, and h is the sigmoid function.

$$cost = -\frac{1}{N}\sum_{i=1}^{N}[l_i ln(h(x_i,\theta)) + (1-l_i)ln(1-h(x_i,\theta))]$$

$$h(x_i,\theta) = \frac{1}{1+e^{-x_i\theta}}$$

After θ parameters were minimized subject to the cost function, the decision boundary could be drawn by picking 2 data points to form a straight line. The x_1 value for each of the points was picked arbitrarily (minimum and maximum value of x_1 in the entire training set +/- 2), while the x_2 value was calculated as follows:

$$0 = \theta_1 + \theta_2 x_1 + \theta_3 x_2$$

$$x_2 = \frac{-1}{\theta_3}(\theta_1 + \theta_2 x_1)$$

Figure 3 below displays each of the training sets and their corresponding classifier.

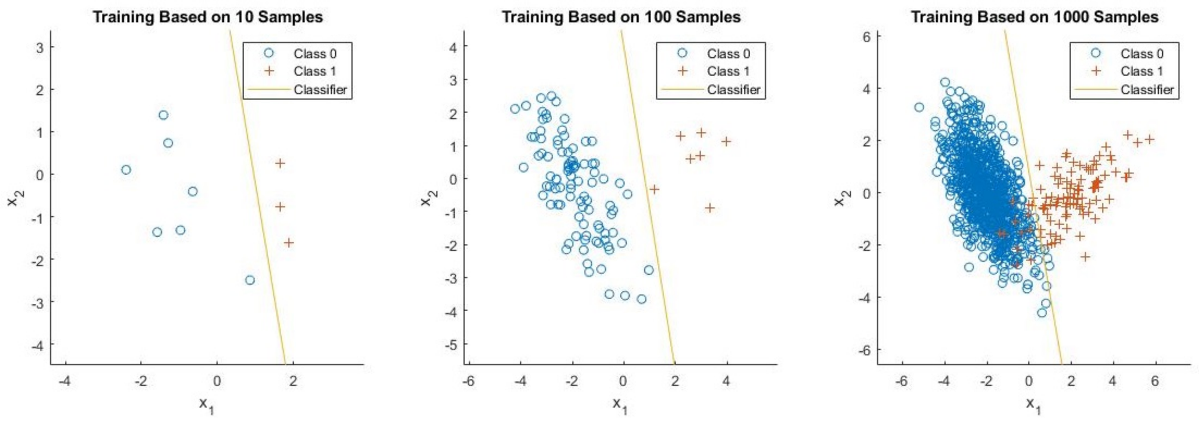


Figure 3: Training of Linear Classifiers for N = 10, 100, 1000

Following the training procedure, each classifier was then tested on the original 10,000 samples generated in part 1 of this question. In a similar fashion to the training sets, the input values of the 10,000-sample data set were mapped to a $x_{val} = [1, x_1, x_2]^T$ vector. To make decisions on each data point, the inequality $x_{val}\theta >= 0$ was used. Since the fminsearch function found θ values that minimize the negative-log-likelihood of each of the training sets, the linear decision boundary for each training set happens when $\theta_1 + \theta_2 x_1 + \theta_3 x_2$ equals to 0. As a result, any value that exceeds 0 is classified as Class 1, and any value under 0 is classified as Class 0. Figure 4 below shows all the incorrect and correct decisions made in the validation data set after each of the 3-classifiers was applied to it.

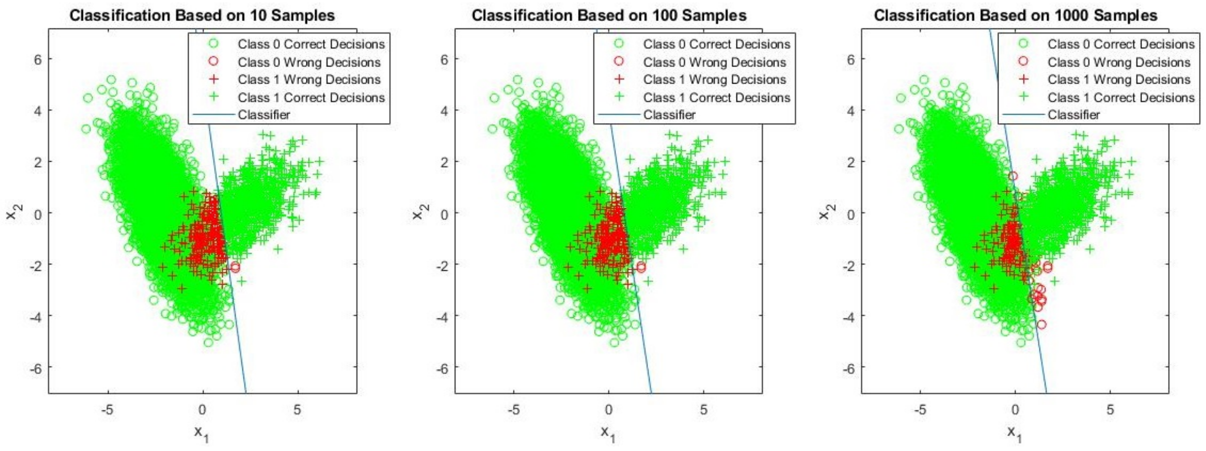


Figure 4: Classification 10,000 Original Validation Samples Based on Linear Classifiers

Problem 1 continued on next page. . .

Part Three

For part 3 of the problem, 3 separate logistic-quadratic-function-based approximations were trained on the same set of 10-, 100-, and 1000-sample training sets. The same cost function (negative-log-likelihood) from part 2 of the problem was minimixed using MATLAB's fminsearch function. This time, however, the input parameters x were mapped to a feature vector $[1, x_1, x_2, x_1^2, x_1x_2, x_2^2]^T$, with the θ vector initialized to all zeros (this time, 6-dimensional).

After θ parameters were minimized subject to the cost function, the decision boundary was drawn by creating a grid of x_1 and x_2 values to cover the axis range of each training set (+/- 2 on each side). For each grid value, the function $\theta_1 + \theta_2 x_1 + \theta_3 x_2 + \theta_4 x_1^2 + \theta_5 x_1 x_2 + \theta_6 x_2^2$ was evaluated in order to get the cost/score for each combination of x_1 and x_2, where a result of 0 indicated that point was on the decision boundary. A countour was then drawn for all points in the grid that resulted in 0 cost, forming the full decision boundary. Figure 5 below shows the decision boundaries drawn for each of the 3 training sets after this same procedure was repeated 3 times.

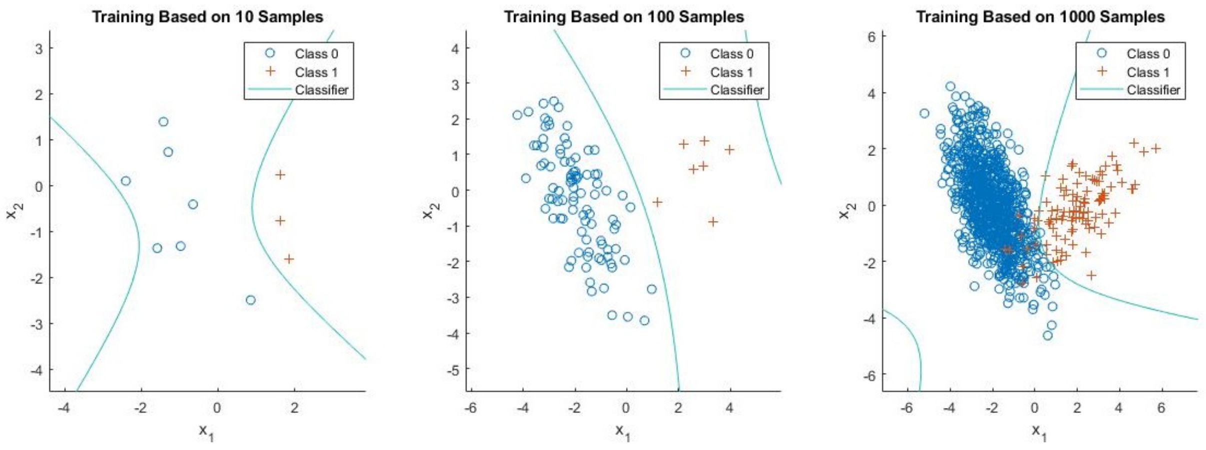


Figure 5: Training of Quadratic Classifiers for N = 10, 100, 1000

Problem 1 continued on next page. . .

Following the training procedure, each classifier was then tested on the original 10,000 samples generated in part 1 of this question. In a similar fashion to the training sets, the input values of the 10,000-sample data set were mapped to a $x_{val} = [1, x_1, x_2, x_1^2, x_1x_2, x_2^2]^T$ vector. To make decisions on each data point, the inequality $x_{val}\theta >= 0$ was used.As described on the previous page, since the fminsearch function found θ values that minimize the negative-log-likelihood of each of the training sets, the linear decision boundary for each training set happens when $\theta_1 + \theta_2x_1 + \theta_3x_2 + \theta_4x_1^2 + \theta_5x_1x_2 + \theta_6x_2^2$ equals to 0. As a result, any value that exceeds 0 is classified as Class 1, and any value under 0 is classified as Class 0. Figure 6 below shows all the incorrect and correct decisions made in the validation data set after each of the 3-classifiers was applied to it.

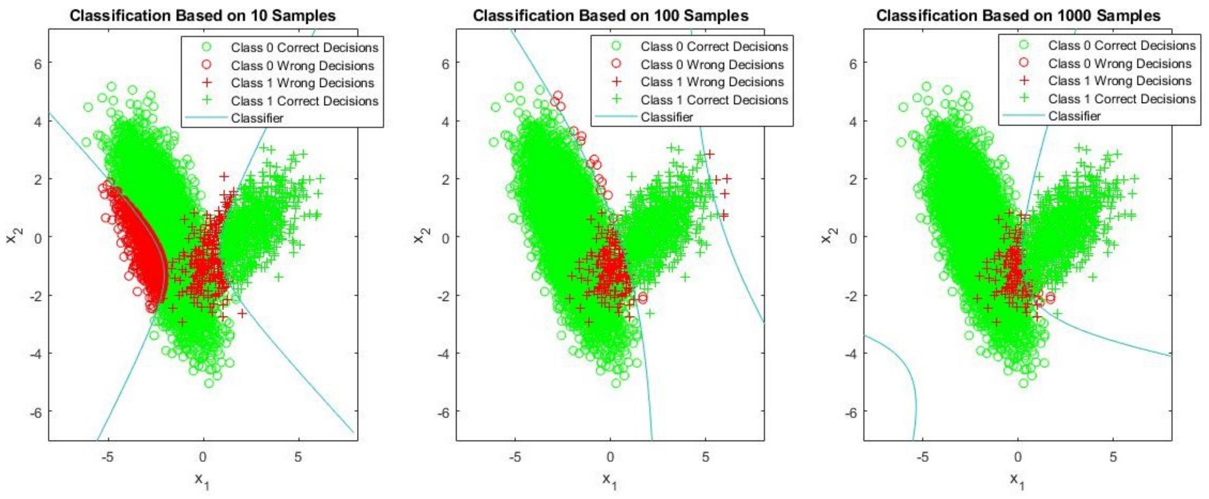
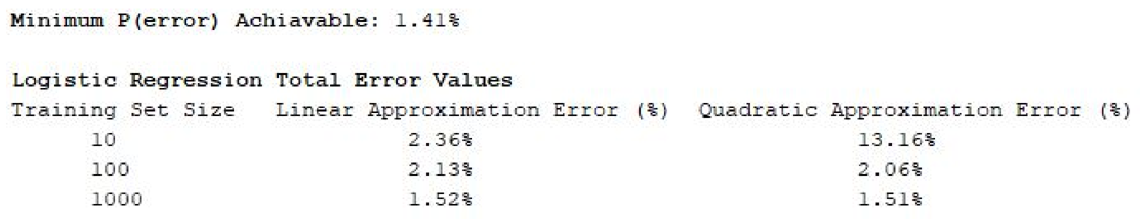


Figure 6: Classification 10,000 Original Validation Samples Based on Quadratic Classifiers

Error Calculations

For each classifier tested on the 10,000-sample validation set, the number of false alarms and missed detections was counted and used to calculate the total error. Figure 7 below summarizes error values attained with each classifier. As can be observed, minimum P(error) classifier from part 1 of the problem achieves lowest possible error value. The linear and the quadratic classifiers have a varying level of error depending on how many samples are used to train the classifier.



```
Minimum P(error) Achiavable: 1.41%

Logistic Regression Total Error Values
Training Set Size   Linear Approximation Error (%)  Quadratic Approximation Error (%)
      10                     2.36%                              13.16%
      100                    2.13%                              2.06%
      1000                   1.52%                              1.51%
```

Figure 7: Training of Quadratic Classifiers for N = 10, 100, 1000

As expected, using only 10 samples to train the model results in error values significantly worse than using 100 or 1000 samples. Minimum error achieved happens when 1000 samples are used to train the model, and these error values get close to the minimum P(error) value of 1.41%. The qudratic function achieves slightly better results, which makes sense since the Gaussian data is oval-shaped and slightly overlaps.

Problem 2

Cubic equation and parameters chosen for this problem:

$$y = A(x-0.7)(x+0.5)(x-0.1) + v = 5x^3 - 1.5x^2 - 1.65x + 0.175 + v$$

$$w_{true} = [5, -1.5, -1.65, 0.175]^T$$

$$v \sim N(0, \sigma_{noise}) \qquad \sigma_{noise} = 0.2$$

$$x \sim Uniform(-1, 1) \qquad \gamma = [10^{-10} : 0.1 : 10^{10}]$$

For this problem, 10 samples of x and v were generated and used to calculate the output y of the cubic function from above. The output was calculated by mapping the x input values to a $z_{i,cubic} = [x_i^3 x_i^2 x_i 1]^T$ vector, multiplying each component by its respective true parameter, and adding noise at the end. Based on these generated output values and the mapping of x input values, the MAP estimator was used to estimate the coefficients for the cubic function (without having prior knowledge of what the true parameters are). The MAP estimator was set up as follows (equation derived in class by Professor Deniz):

$$A = \sum_{i=1}^{10} z_i z_i^T + \frac{\sigma_v^2}{\gamma^2} I$$

$$b = \sum_{i=1}^{N} z_i y_i$$

$$\theta_{MAP} = A^{-1} b$$

For one value of γ and 1 realization of x and y pairs, the following scatter plot was generated:

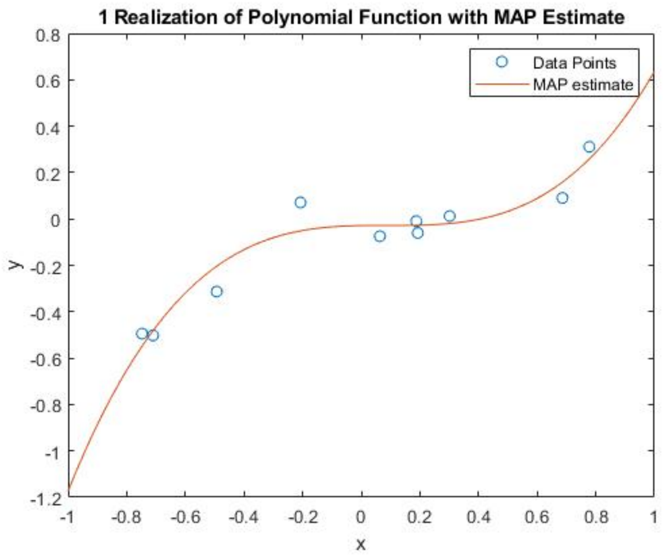


Figure 8: One Realization of X and Y Pairs and MAP Line of Best Fit

Individual points mark the data set D for one particular realization, while the line represents the "line of best fit" generated using the parameters estimated by MAP when $\gamma = 10^{10}$.

To asses the impact of changing gamma values on the parameter estimation, this experiment was repeated on the signle reazlization of x and y pairs with γ values ranging from 10^{-10} to 10^{10} (increasing in 0.1 steps). The MAP parameter estimation for each γ value was stored, and the following plot generated:

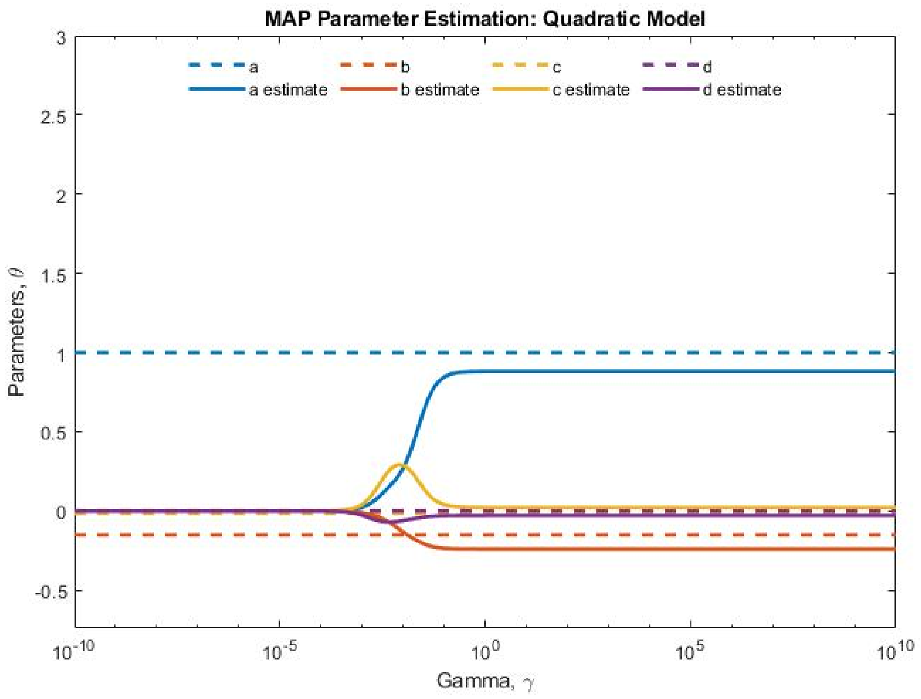


Figure 9: Impact of Changing Gamma on Parameter Estimation

As expected, with low values of γ, the parameters all converge at a single value. However, as γ increases, the paramater estimates start diverging and moving towards their true value. Due to noise impacting the vale of the output y, none of the parameters are estimated with full precision. If noise v were to be reduced to 0, and more samples were collected, the MAP parameter estimates would come close to the true value of the parameters marked by the dashed lines. However, since noise plays a big part in the output function, the MAP parameters are never perfectly estimated.

As the last step for this problem, the procedure described above was repeated on 100 realizations, which were all evaluated for the full range of γ values. In addition, for each realization and γ combination, the squared-error value for the MAP estimator was collected by calculating and squaring the L_2 norm of the difference between the true parameters and the MAP parameters, as noted in the equation below:

$$||w_{true} - wMAP||_2^2$$

Based on all the squared-error values collected, the plot below was generated to show the minimum, 25th percentile, median, 75th percentile, and the maximum of squared-error values for each value of γ.

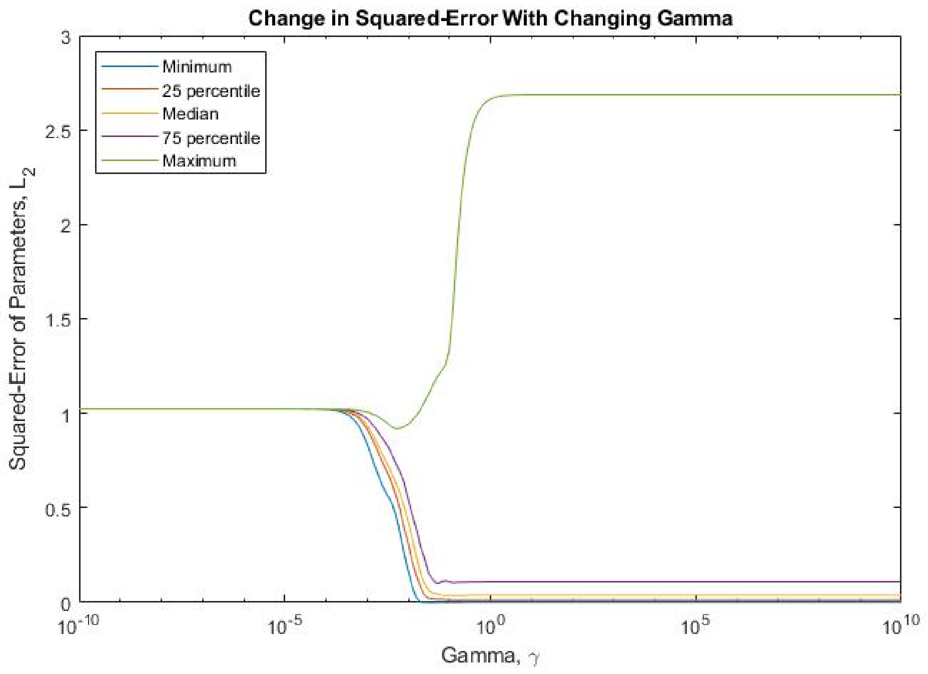


Figure 10: Distribution of Sqared-Error for all Gamma Values

As expected, for small values of gamma, the curves all meet at the same value. However, as the MAP estimate converges to the ML estimate, the curves start deflecting from each other and eventually settle at their level, separate from each other. Majority of the data (up to and including the 75th percentile) rests at a level lower than the squared-error value of low γ values, while the top 25%, indicated with the green line, settles at a value slightly higher than that. Figure 11 on the next page shows a better distinction between the first 75% of data by plotting both axes on a logarithmic scale.

Problem 2 continued on next page. . .

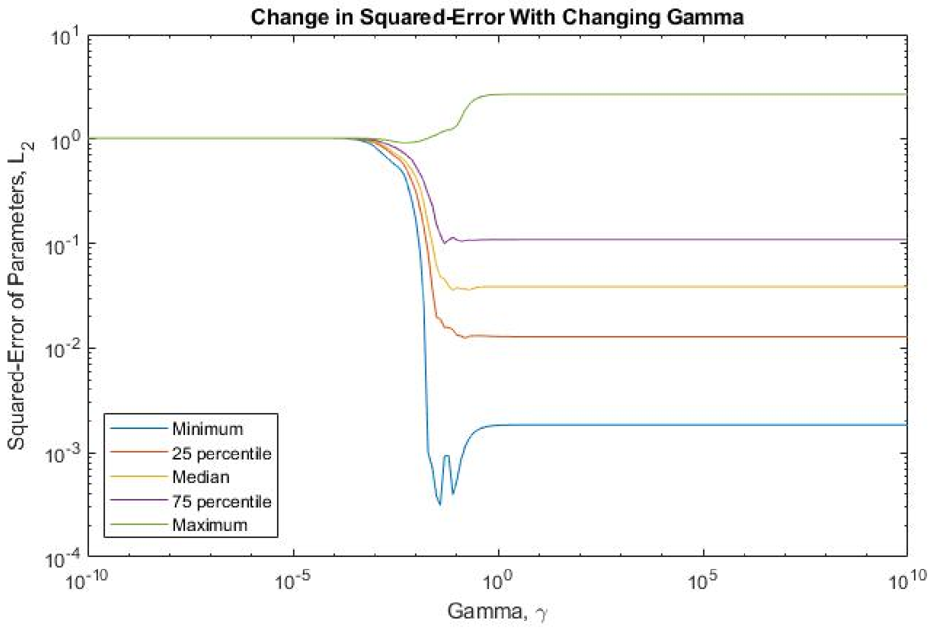


Figure 11: Distribution of Sqared-Error for all Gamma Values, Logarithmic Scale

Problem 3

Step 1
Class conditional PDFs and mixture coefficients:

Class 0

$$\mu_1 = \begin{bmatrix} 5 \\ 1 \end{bmatrix} \qquad \Sigma_1 = \begin{bmatrix} 5 & 1 \\ 1 & 4 \end{bmatrix} \qquad P(Gauss\ 1) = 0.2$$

$$\mu_2 = \begin{bmatrix} 6 \\ 6 \end{bmatrix} \qquad \Sigma_2 = \begin{bmatrix} 3 & 1 \\ 1 & 3 \end{bmatrix} \qquad P(Gauss\ 2) = 0.23$$

$$\mu_3 = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \qquad \Sigma_3 = \begin{bmatrix} 5 & -1 \\ 1 & 3 \end{bmatrix} \qquad P(Gauss\ 3) = 0.27$$

$$\mu_4 = \begin{bmatrix} 1 \\ 5 \end{bmatrix} \qquad \Sigma_4 = \begin{bmatrix} 4 & -2 \\ -2 & 3 \end{bmatrix} \qquad P(Gauss\ 4) = 0.3$$

For step 1 of this problem, data with 10, 100, 1000 samples was generated using the GMM model described above. Figure 12 below shows a particular realization of the model with 1000 samples.

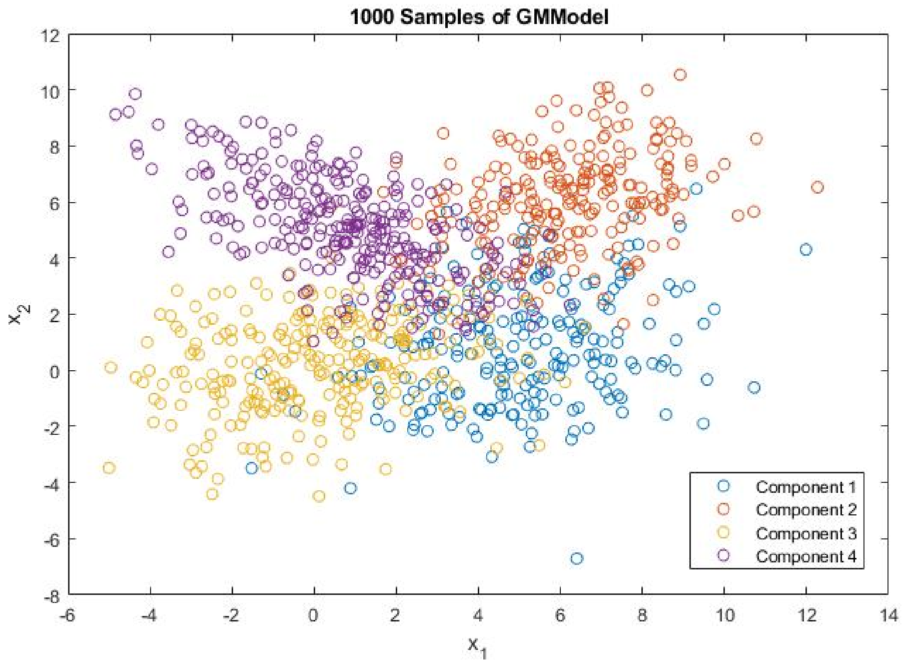


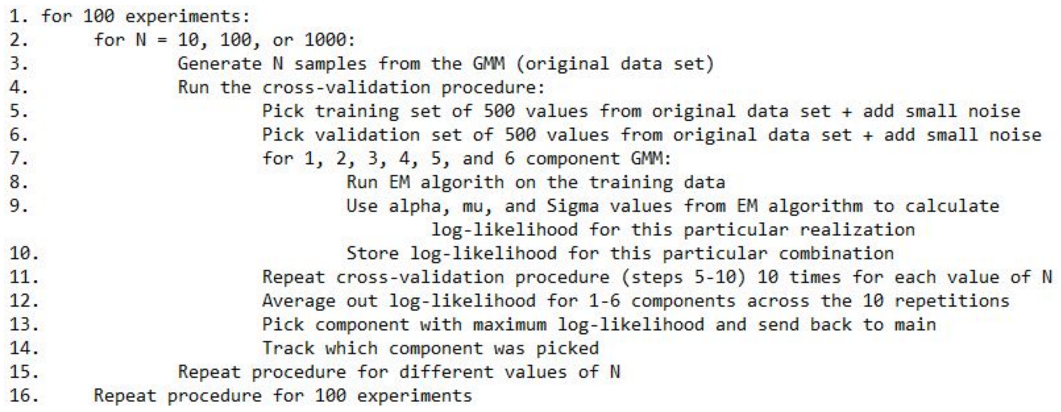
Figure 12: Visualizing 4-Component GMM Distributions

Steps 2 and 3

In step 2 of the problem, bootstrap cross-validation was used in parallel with the Expectation-Maximization (EM) algorithm for all 3 of the generated data sets and a total of 100 experiments. Each experiment started off by randomly selecting points from the original (10, 100, or 1000-sample) data set to be part of the training set and the validation set (points for training and validation picked independently). In order to ensure enough points were selected for GMM orders of 1-6 to be considered, each new set consisted of 500 points.

After the training and validation sets were selected, the EM algorithm was used to determine the maximum likelihood estimates for that particular training set with GMM component counts ranging from 1 to 6. Once the ideal α, μ, and Σ values were computed, the validation set was used to calculate the log-likelihood for each of the 6 component values. Since this cross-validation procedure was completed 10 times for each data set, the log-likelihood was averaged out across all the component values and the maximum selected as the winner. In other words, whichever number of components had the maximum log-likelihood across the 10 iterations for a signle realization of each of the data sets was declared the winner.

The process described above was repeated for a total of 100 experiments, with each experiment testing a particular realization of the 10-, 100-, and 1000- sample data set 10 times for each of the component number values (1 through 6). Figure 13 below shows the pseudocode and summarizes the overall flow of the program.



```
1. for 100 experiments:
2.      for N = 10, 100, or 1000:
3.              Generate N samples from the GMM (original data set)
4.              Run the cross-validation procedure:
5.                      Pick training set of 500 values from original data set + add small noise
6.                      Pick validation set of 500 values from original data set + add small noise
7.                      for 1, 2, 3, 4, 5, and 6 component GMM:
8.                              Run EM algorith on the training data
9.                              Use alpha, mu, and Sigma values from EM algorithm to calculate
                                        log-likelihood for this particular realization
10.                             Store log-likelihood for this particular combination
11.                     Repeat cross-validation procedure (steps 5-10) 10 times for each value of N
12.                     Average out log-likelihood for 1-6 components across the 10 repetitions
13.                     Pick component with maximum log-likelihood and send back to main
14.                     Track which component was picked
15.             Repeat procedure for different values of N
16.     Repeat procedure for 100 experiments
```

Figure 13: Question 3 Pseudocode

Problem 3 continued on next page. . .

To run the EM algorithm on each training set, I used two different approaches- using MATLAB's built-in fitgmdist function, and using a non-built-in EM function (originally written and provided to the class by Professor Deniz).

For the non-built-in function, the EM algorithm was developed step by step, by first initializing the alpha values to equal weights based on the number of components, and picking random mu and Sigma values based on values randomly picked from the training set. Next, the parameters (α, μ, and Σ) were recalculated and updated using equations derived in class and summarized in Figure 14 below. In addition, a regularization term was added to Σ on every iteration to ensure the covariance matrix is aways positive-definite.

k = mixture component #

$P_k(\underline{x}|\alpha_k, \mu_k, \Sigma_k) \rightarrow$ density distribution of each component

$$W_{ik} = \frac{P_k(\underline{x}_i|\alpha_k,\mu_k,\Sigma_k)\cdot\alpha_k}{\sum_{m=1}^{M} P_m(\underline{x}_i|\alpha_m,\mu_m,\Sigma_m)\cdot\alpha_m}, \quad 1 \le k \le M \nearrow \text{\# components}, \quad 1 \le i \le N \rightarrow \text{size of data}$$

new alpha:
$$\alpha_k^{new} = \frac{\sum_{i=1}^{N} W_{ik}}{N}$$

new Sigma:
$$\Sigma_k^{new} = \left(\frac{1}{\sum_{i=1}^{N} W_{ik}}\right)\sum_{i=1}^{N} W_{ik}(\underline{x}_i-\mu_k^{new})(\underline{x}_i-\mu_k^{new})^T$$

new mu:
$$\mu_k^{new} = \left(\frac{1}{\sum_{i=1}^{N} W_{ik}}\right)\sum_{i=1}^{N} W_{ik}\cdot\underline{x}_i$$

Figure 14: EM Algorithm Equations

After recalculating new parameters, convergence was tested by adding up the difference between all the old and new parameter values and comparing it against a threshold δ.

Problem 3 continued on next page. . .

In my implementation, I was not able get the algorithm to converge in all cases, despite adding Gaussian noise to the training set, re-initializing mu values multiple times throughout, or adjusting the original distribution itself. As a result, I had to limit the algorithm at 3000 iterations for each combination of parameters. This does not provide a solution for the convergance issue, but I did manage to gather some data to show the performance of the algorithm. Figure 15 below shows the results of running the algorithm for 100 experiments, and keeping track of when each component number was selected as the winner.

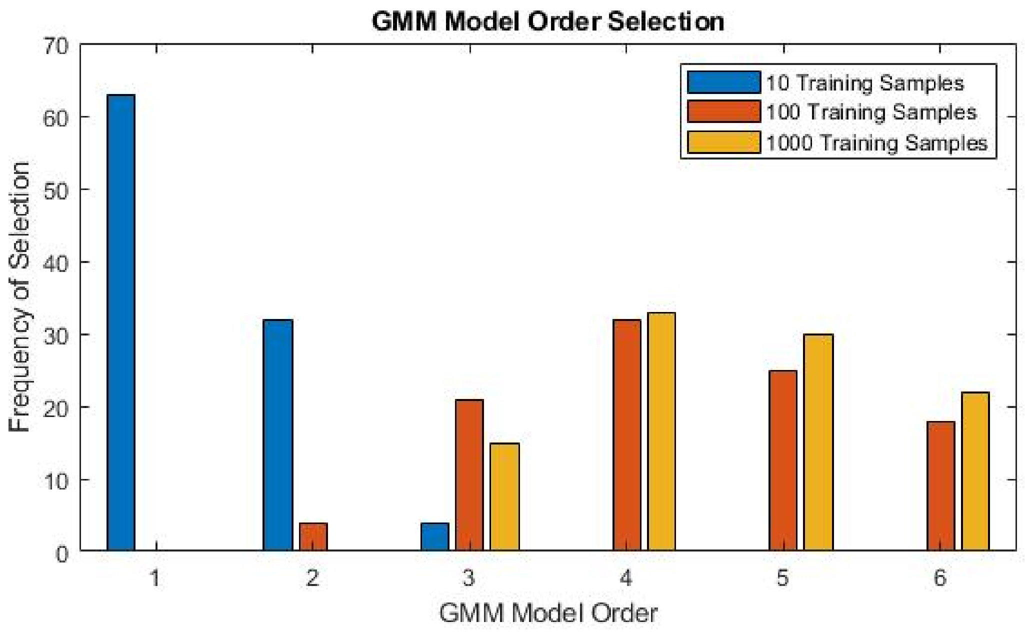


Figure 15: GMM Component Selection Over 100 Experiments

As shown in the figure, when only 10 data points were used to form the training data set, most of the iterations ended with lower-valued component numbers being selected (1, 2, and 3 component GMMs). This makes sense since the 10 points would be very spread out and it would be difficult to form as many as 6 groups based on just those 10 points. With 100 and 1000 sample data sets, however, the distribution was normal-shaped, with the center (winner in most iterations) being a 4-component GMM. This shows that with more points at disposal to form the training set, the algorithm is able to pick the correct component number more often.

Problem 3 continued on next page. . .

For the built-in MATLAB function, for each iteration of the EM algorithm, two things had to be provided- the training set, and the number of components. The function then fits a Gaussian mixture with the selected number of components to the training data using the EM algorithm and reports back on the α, μ, and Σ values for the distributions. When using the built-in function, to ensure the parameters are not constrained in any way, I did not select the covariance to be diagonal or shared between the distributions (covariance remained independent and full for each component within a GMM). I also chose to add a small regularization term (1^{-10}) to ensure that all covariance matrices are positive-definite as the algorithm is being executed.

Figure 16 below summarizes the results of running the EM algorithm for 100 experiments. As can be observed, when only 10 data points were used to form the full 500-point data set, the algorithm seemed to favor the 1-component GMM. On the other hand, when 100 and 1000 samples were used to pick the training and validation sets, the distribution between component selection seems to even out. This is most likely the result of having my original 4 components of the true GMM too close to each other and overlapping. Since MATLAB does not make their function files public, I was unable to look into what the difference is between MATLAB'S implementation of the EM algorithm adn the one we discussed in class. However, the non-built-in function from the previous page seems to have dealt with the same problem better since it was able to pick the tru component number more times than the built-in function.

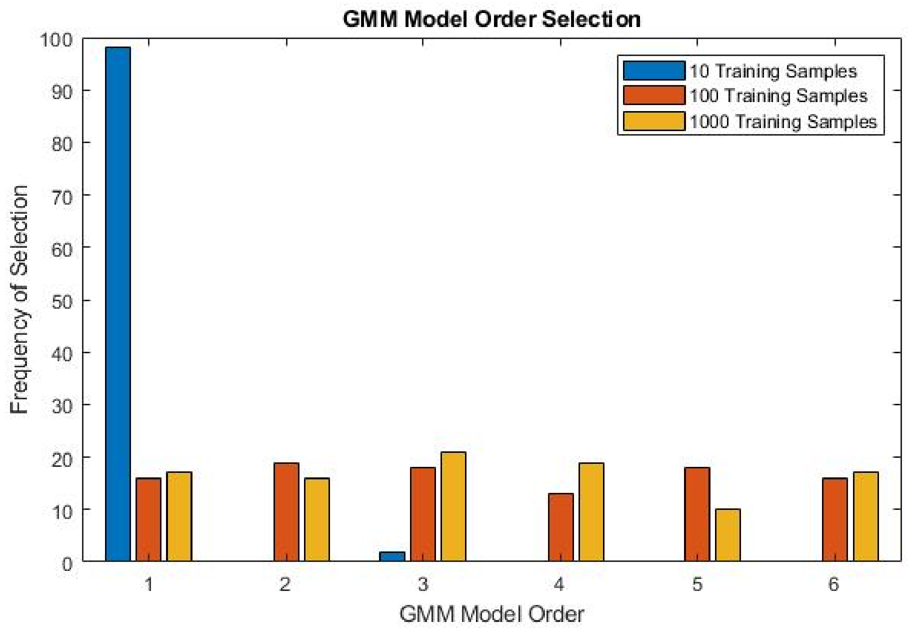


Figure 16: GMM Component Selection Over 100 Experiments Using fitgmdist Function

Appendix A: Question 1 Code

```
%% Exam 2: Question 1
% Anja Deric | February 25, 2020
clear all; close all; clc;

%% Set-Up: given parameters and validation data
% Given parameters
n = 2;                        % number of feature dimensions
N_train = [10;100;1000];      % number of training samples
N_val = 10000;                % number of validation samples
p = [0.9,0.1];                % class priors
mu = [-2 2;0 0];
Sigma(:,:,1) = [1 -0.9;-0.9 2]; Sigma(:,:,2) = [2 0.9;0.9 1];

% Generate true class labels and draw samples from each class pdf
label_val = (rand(1,N_val) >= p(1));
Nc_val = [length(find(label_val==0)),length(find(label_val==1))];
x_val = generate_data(n,N_val,label_val,mu,Sigma,Nc_val);

%% Part 1: Minimum P-Error Classifier
% Calculate dicriminant scores and tau based on classification rule
discriminantScore = log(evalGaussian(x_val,mu(:,2),Sigma(:,:,2)))- ...
    log(evalGaussian(x_val,mu(:,1),Sigma(:,:,1)));
tau = log(sort(discriminantScore(discriminantScore >= 0)));

% Find midpoints of tau to use as threshold values
mid_tau = [tau(1)-1 tau(1:end-1) + diff(tau)./2 tau(length(tau))+1];

% Make decision for every threshold and calculate error values
for i = 1:length(mid_tau)
    decision = (discriminantScore >= mid_tau(i));
    pFA(i) = sum(decision==1 & label_val==0)/Nc_val(1); % False alarm prob.
    pCD(i) = sum(decision==1 & label_val==1)/Nc_val(2); % Correct detection
        prob.
    pE(i) = pFA(i)*p(1)+(1-pCD(i))*p(2);                % Total error prob.
end

% Find minimum error and corresponding threshold + decisions
[min_error,min_index] = min(pE);
min_decision = (discriminantScore >= mid_tau(min_index));
min_FA = pFA(min_index); min_CD = pCD(min_index);

% Plot ROC curve with minimum error point labeled
figure(1); plot(pFA,pCD,'-',min_FA,min_CD,'o');
title('Minimum Expected Risk ROC Curve');
legend('ROC Curve', 'Calculated Min Error');
xlabel('P_{False Alarm}'); ylabel('P_{Correct Detection}');
```

```
% Create grid to cover all data points
h_grid = linspace(floor(min(x_val(1,:)))-2,ceil(max(x_val(1,:)))+2);
v_grid = linspace(floor(min(x_val(2,:)))-2,ceil(max(x_val(2,:)))+2);
[h,v] = meshgrid(h_grid,v_grid);

% Calculate discriminant score for each grid point (0 = decision bound.)
d_score_grid = log(evalGaussian([h(:)';v(:)'],mu(:,2),Sigma(:,:,2)))- ...
    log(evalGaussian([h(:)';v(:)'],mu(:,1),Sigma(:,:,1)))-mid_tau(min_index);
d_scores = reshape(d_score_grid,length(h_grid),length(v_grid));

% Plot classified data points and decision boundary
figure(2);
plot_classified_data(min_decision, label_val, Nc_val, p, [1 1 1], ...
    x_val', [h_grid;v_grid;d_scores], 'Q');

%% Part 2 & 3: Linear and Quadratic Regression
for i = 1:length(N_train)
    % Generate true labels and training data for each sample size
    label = (rand(1,N_train(i)) >= p(1));
    Nc = [length(find(label==0)),length(find(label==1))];
    x = generate_data(n,N_train(i),label,mu,Sigma,Nc);

    % Initialize training parameters (map x to linear and quadratic func.)
    x_L = [ones(N_train(i), 1) x'];          % linear parameters
    initial_theta_L = zeros(n+1, 1);
    x_Q = [ones(N_train(i), 1) x(1,:)' x(2,:)' (x(1,:).^2)' ...
        (x(1,:).*x(2,:))' (x(2,:).^2)'];    % quadratic parameters
    initial_theta_Q = zeros(6, 1);
    label = double(label)';

    % Compute gradient descent to get theta values for linear and quad.
    [theta_L, cost_L] = fminsearch(@(t)(cost_func(t, x_L, label, ...
        N_train(i))), initial_theta_L);
    [theta_Q, cost_Q] = fminsearch(@(t)(cost_func(t, x_Q, label, ...
        N_train(i))), initial_theta_Q);

    % Linear: choose points to draw straight boundary line
    plot_x1 = [min(x_L(:,2))-2,  max(x_L(:,2))+2];
    plot_x2 = (-1./theta_L(3)).*(theta_L(2).*plot_x1 + theta_L(1));

    % Linear: plot training data and trained classifier
    figure(3); plot_training_data(label, i, x_L, [plot_x1;plot_x2], 'L');
    title(['Training Based on ',num2str(N_train(i)),' Samples']);

    % Linear: use validation data (10k points) and make decisions
    test_set_L = [ones(N_val, 1) x_val'];
    decision_L = test_set_L*theta_L >= 0;

    % Linear: plot all decisions and boundary line
    figure(4);
    error_L(i) = plot_classified_data(decision_L, label_val', Nc_val, p, ...
        [1,3,i], test_set_L(:,2:3), [plot_x1;plot_x2],'L');
    title(['Classification Based on ',num2str(N_train(i)),' Samples']);

    % Quadratic: create grid to cover all data points and calculate scores
    figure(5); subplot(2,3,i);
    h_grid = linspace(min(x_Q(:,2))-6, max(x_Q(:,2))+6);
    v_grid = linspace(min(x_Q(:,3))-6, max(x_Q(:,3))+6);
    score = get_boundary(h_grid,v_grid,theta_Q); % score of 0 = decision bound
        .

    % Quadratic: plot training data and trained classifier
    plot_training_data(label, i, x_Q, [h_grid;v_grid;score], 'Q')
    title(['Training Based on ',num2str(N_train(i)),' Samples']);

    % Quadratic: use validation data (10k points) and make decisions
    test_set_Q = [ones(N_val, 1) x_val(1,:)' x_val(2,:)' (x_val(1,:).^2)' ...
        (x_val(1,:).*x_val(2,:))' (x_val(2,:).^2)'];
    decision_Q = test_set_Q*theta_Q >= 0;

    % Quadratic: plot all decisions and boundary countour
    figure(6);
    error_quad(i) = plot_classified_data(decision_Q, label_val', Nc_val, ...
        p, [1,3,i], test_set_Q(:,2:3),[h_grid;v_grid;score],'Q');
    title(['Classification Based on ',num2str(N_train(i)),' Samples']);
end

%% Print all calculated error values
fprintf('<strong>Minimum P(error) Achiavable:</strong> %.2f%%\n\n',min_error
   *100);
fprintf('<strong>Logistic Regression Total Error Values</strong>\n');
fprintf('Training Set Size\tLinear Approximation Error (%%)\tQuadratic
   Approximation Error (%%)\n');
fprintf('\t  %i\t\t\t\t\t  %.2f%%\t\t\t\t\t\t\t%.2f%%\n',[N_train';error_L;
   error_quad]);

%% Functions
function x = generate_data(n, N, label, mu, Sigma, Nc)
    % Generate N Gaussian samples based on distribution of class priors
    x = zeros(n,N);
    for L = 0:1
        x(:,label==L) = mvnrnd(mu(:,L+1),Sigma(:,:,L+1),Nc(L+1))';
    end
end

function plot_training_data(label, fig, x, bound, type)
    % Plots original class labels and decision boundary
    subplot(1,3,fig); hold on;
    plot(x(label==0,2),x(label==0,3),'o',x(label==1,2),x(label==1,3),'+');

    if type == 'L'
        % Plot straight line if boundary is linear
        plot(bound(1,:), bound(2,:));
    elseif type == 'Q'
        % Plot decision countour if non-linear (discriminant scores are 0)
        contour(bound(1,:), bound(2,:), bound(3:end,:), [0, 0]);
    end

    % Restrict axis and add all labels
    axis([min(x(:,2))-2, max(x(:,2))+2, min(x(:,3))-2, max(x(:,3))+2]);
    legend('Class 0','Class 1','Classifier');
    xlabel('x_1'); ylabel('x_2'); hold on;
end

function error = plot_classified_data(decision, label, Nc, p, fig, x, bound,
    type)
    % Plots incorrect and correct decisions (and boundary) based on original
        class labels

    % Find all correct and incorrect decisions
    TN = find(decision==0 & label==0);  % true negative
    FP = find(decision==1 & label==0); pFA = length(FP)/Nc(1); % false
        positive
    FN = find(decision==0 & label==1); pMD = length(FN)/Nc(2); % false
        negative
    TP = find(decision==1 & label==1);  % true positive
    error = (pFA*p(1) + pMD*p(2))*100;  % calculate total error

    % Plot all decisions (green = correct, red = incorrect)
    subplot(fig(1),fig(2),fig(3));
    plot(x(TN,1),x(TN,2),'og'); hold on;
    plot(x(FP,1),x(FP,2),'or'); hold on;
    plot(x(FN,1),x(FN,2),'+r'); hold on;
    plot(x(TP,1),x(TP,2),'+g'); hold on;

    % Plot boundary based on whether its linear(L) or non-linear(Q)
    if type == 'L'
        % Plot straight line from 2 points
        plot(bound(1,:), bound(2,:));
    elseif type == 'Q'
        % Plot decision contour (when discriminant scores are 0)
        contour(bound(1,:), bound(2,:), bound(3:end,:), [0, 0]); % p
    end

    % Restrict axis and add all labels
    axis([min(x(:,1))-2, max(x(:,1))+2, min(x(:,2))-2, max(x(:,2))+2])
    legend('Class 0 Correct Decisions','Class 0 Wrong Decisions', ...
        'Class 1 Wrong Decisions','Class 1 Correct Decisions','Classifier');
    xlabel('x_1'); ylabel('x_2');
end

function cost = cost_func(theta, x, label,N)
    % Cost function to be minimized to get best fitting parameters
    h = 1 ./ (1 + exp(-x*theta));        % Sigmoid function
    cost = (-1/N)*((sum(label' * log(h)))+(sum((1-label)' * log(1-h))));
end

function score = get_boundary(hGrid, vGrid, theta)
    % Generates grid of scores that spans the full range of data (where
    % a score of 0 indicates decision boundary level)
    z = zeros(length(hGrid), length(vGrid));
    for i = 1:length(hGrid)
        for j = 1:length(vGrid)
            % Map to a quadratic function
            x_bound = [1 hGrid(i) vGrid(j) hGrid(i)^2 hGrid(i)*vGrid(j) vGrid(
                j)^2];
            % Calculate score
            z(i,j) = x_bound*theta;
        end
    end
    score = z';
end

function g = evalGaussian(x,mu,Sigma)
    % Evaluates the Gaussian pdf N(mu,Sigma) at each coumn of X
    [n,N] = size(x);
    C = ((2*pi)^n * det(Sigma))^(-1/2);      % coefficient
    E = -0.5*sum((x-repmat(mu,1,N)).*(inv(Sigma)*(x-repmat(mu,1,N))),1); %
        exponent
    g = C*exp(E);    % final gaussian evaluation
end
```

Appendix B: Question 2 Code

```
%% Ecam 2: Question 2
% Anja Deric | February 24, 2020
clear; close all; clc;

%% Initialize all constants and parameters

N = 10;                             % Number of samples
SigmaV = 0.2;                       % Variance of 0-mean Gaussian noise
gamma_array = 10.^[-10:0.1:10]; % Array of gamma values
realizations = 100;                 % Total experiments for each gamma

% True parameter array (values picked so y has 3 real roots)
SigmaV=0.005;
a=1; b=-0.15; c=-0.015; d=0.001;
w_true = [a; b; c; d];

% Calculate noise and input values
v = SigmaV^0.5*randn(1,N);
x = unifrnd(-1,1,1,N);

% Map to a cubic function and calculate output y
zC = [x.^3; x.^2; x; ones(1,N)];
y = zC'*w_true + v';

%% Estimate MAP Parameters and Plot 1 Realization Only

% MAP estimation
for i = 1:length(gamma_array)
    gamma = gamma_array(i);
    w_MAP(:,i) = inv((zC*zC')+SigmaV^2/gamma^2*eye(size(zC,1)))* ...
        sum(repmat(y',size(zC,1),1).*zC,2);
end

% Calculate x and y coordinates to plot MAP line of best fit
x_fit = linspace(-1,1);
best_theta = w_MAP(:,end);
y_fit = best_theta(1).*x_fit.^3+best_theta(2).*x_fit.^2+best_theta(3).*x_fit+
   best_theta(4);

% Plot original points with the MAP line of best fit
figure; scatter(x,y'); hold on; box on;
plot(x_fit,y_fit); legend('Data Points','MAP estimate');
title('1 Realization of Polynomial Function with MAP Estimate');
xlabel('x');ylabel('y');

%% Plot MAP Parameter Variation With Gamma
```

```
% Plot tue parameters
figure; hold on; box on; ax=gca; ax.XScale = 'log';
axis([gamma_array(1) gamma_array(end) min([w_true;w_MAP(:)])-0.5 ...
    max([w_true;w_MAP(:)])+2]);

% Plot paramater estimates for all gamma values
plot(gamma_array,repmat(w_true,1,length(gamma_array)),'--','LineWidth',2);
set(gca,'ColorOrderIndex',1);
plot(gamma_array,w_MAP,'-','LineWidth',2);

% Add labels and legend
xlabel('Gamma, \gamma'); ylabel('Parameters, \theta'); title('MAP Parameter
   Estimation: Quadratic Model')
lgnd=legend('a','b','c','d','a estimate','b estimate','c estimate','d estimate
   ');
lgnd.Location = 'north'; lgnd.Orientation = 'horizontal'; lgnd.NumColumns = 4;
    box(lgnd,'off');

%% Estimate MAP across all gammas for 100 realizations

clearvars -except w_true mu Sigma SigmaV gamma_array realizations N;
for n = 1:realizations
    % Generate noise and input values
    v = SigmaV^0.5*randn(1,N);
    x = unifrnd(-1,1,1,N);

    % Map to a cubic function and calculate true and noisy output
    zC = [x.^3; x.^2; x; ones(1,N)];
    y_truth{1,n} = zC'*w_true;
    y = y_truth{1,n} + v';

    % Estimate parameters and error for all gamma values
    for i = 1:length(gamma_array)
        gamma = gamma_array(i);
        % Calculate MAP parameter estimate
        %w_MAP{1,n}(:,i) = inv((zC*zC')+SigmaV^2/gamma^2*eye(size(zC,1)))*...
        %    sum(repmat(y',size(zC,1),1).*zC,2);
        w_MAP{1,n}(:,i) = inv((zC*zC')+SigmaV^2/gamma^2*eye(size(zC,1)))*...
            (zC*y);
        % Calculate squared-error-value (2nd-norm)
        L2_norm(n,i) = norm(w_true - w_MAP{1,n}(:,i),2).^2;
    end

    avMsqError(n,1:length(gamma_array)) = length(w_true)\sum((w_MAP{1,n} - ...
        repmat(w_true,1,length(gamma_array))).^2);
end

%%
percentileArray = [0,25,50,75,100];
```

```
figure;
ax = gca; hold on; box on;
prctlMsqError = prctile(avMsqError,percentileArray,1);
p=plot(ax,gamma_array,prctlMsqError,'LineWidth',2);
xlabel('gamma'); ylabel('average mean squared error of parameters'); ax.XScale
    = 'log';
lgnd = legend(ax,p,[num2str(percentileArray'),...
    repmat(' percentile',length(percentileArray),1)]); lgnd.Location = '
        southwest';


%% Plot MAP Ensemble Squared-Error Values

% Calculate min, 25%, median, 75%, and max sq-error for each gamma
prctl_array = [0,25,50,75,100];
L2_norm_prctl = prctile(L2_norm,prctl_array,1);

% Plot change over all gamma values
figure; semilogx(gamma_array,L2_norm_prctl);
title('Change in Squared-Error With Changing Gamma');
xlabel('Gamma, \gamma'); ylabel('Squared-Error of Parameters, L_2');
lgnd = legend('Minimum', '25 percentile','Median','75 percentile','Maximum');
lgnd.Location = 'northwest';

% Plot change over all gamma values (both axes log scale)
figure; loglog(gamma_array,L2_norm_prctl);
title('Change in Squared-Error With Changing Gamma');
xlabel('Gamma, \gamma'); ylabel('Squared-Error of Parameters, L_2');
lgnd = legend('Minimum', '25 percentile','Median','75 percentile','Maximum');
lgnd.Location = 'northwest';
```

Appendix C: Question 3 Code

```
% Question 3 | Take Home Exam #3
% Anja Deric | February 24, 2020
clear all; close all; clc;

%% Part 1
n=2; experiments = 100;
N = [10 100 1000];    % number of iid samples
num_GMM_picks = zeros(length(N),6);

for i = 1:experiments
    % True mu and Sigma values for 4-component GMM
    mu_true(:,1) = [7;0]; mu_true(:,2) = [6;6];
    mu_true(:,3) = [0;0]; mu_true(:,4) = [-1;7];
    Sigma_true(:,:,1) = [5 1;1 4]; Sigma_true(:,:,2) = [3 1;1 3];
    Sigma_true(:,:,3) = [5 1;1 3]; Sigma_true(:,:,4) = [4 -2;-2 3];
    alpha_true = [0.2 0.23 0.27 0.3];

    % Generate Gaussians with N samples and run cross-validation
    for i = 1:length(N)
        x = generate_samples(n, N(i), mu_true, Sigma_true, cumsum(alpha_true))
            ;
        % Store GMM with highest performance for each iteration
        GMM_pick = cross_val(x);
        num_GMM_picks(i,GMM_pick) = num_GMM_picks(i,GMM_pick)+1;
    end

    % Plot frequency of model selection
    bar(num_GMM_picks');
    legend('10 Training Samples','100 Training Samples','1000 Training Samples
        ');
    title('GMM Model Order Selection');
    xlabel('GMM Model Order'); ylabel('Frequency of Selection');
end

%% Question 3 Functions
function x = generate_samples(n, N, mu, Sigma, p_cumulative)
    % Draws N samples from each class pdf to create GMM
    x = zeros(n,N);
    for i = 1:N
        % Generate random probability
        num = rand(1,1);
        % Assign point to 1 of 4 Gaussians based on probability
        if (num > p_cumulative(1)) == 0
            x(:,i) = mvnrnd(mu(:,1),Sigma(:,:,1),1)';
        elseif (num > p_cumulative(2)) == 0
            x(:,i) = mvnrnd(mu(:,2),Sigma(:,:,2),1)';
        elseif (num > p_cumulative(3)) == 0
```

```
            x(:,i) = mvnrnd(mu(:,3),Sigma(:,:,3),1)';
        else
            x(:,i) = mvnrnd(mu(:,4),Sigma(:,:,4),1)';
        end
    end
end

function best_GMM = cross_val(x)
    % Performs EM algorithm to estimate parameters and evaluete performance
    % on each data set B times, with 1 through M GMM models considered

    B = 10; M = 6;           % repetitions per data set; max GMM considered
    perf_array= zeros(B,M); % save space for performance evaluation

    % Test each data set 10 times
    for b = 1:B
        % Pick random data points to fill training and validation set and
        % add noise
        set_size = 500;
        train_index = randi([1,length(x)],[1,set_size]);
        train_set = x(:,train_index) + (1e-3)*randn(2,set_size);
        val_index = randi([1,length(x)],[1,set_size]);
        val_set = x(:,val_index) + (1e-3)*randn(2,set_size);

        for m = 1:M
           % Non-Built-In: run EM algorith to estimate parameters
           %[alpha,mu,sigma] = EMforGMM(m,train_set,set_size,val_set);

           % Built-In function: run EM algorithm to estimate parameters
           GMModel = fitgmdist(train_set',M,'RegularizationValue',1e-10);
           alpha = GMModel.ComponentProportion;
           mu = (GMModel.mu)';
           sigma = GMModel.Sigma;

           % Calculate log-likelihood performance with new parameters
           perf_array(b,m) = sum(log(evalGMM(val_set,alpha,mu,sigma)));
        end
    end

    % Calculate average performance for each M and find best fit
    avg_perf = sum(perf_array)/B;
    best_GMM = find(avg_perf == max(avg_perf),1);
end

function [alpha_est,mu,Sigma]=EMforGMM(M, x, N,val_set)
% Uses EM algorithm to estimate the parameters of a GMM that has M
% number of components based on pre-existing training data of size N

    delta = 0.04;        % tolerance for EM stopping criterion
```

```
reg_weight = 1e-2;   % regularization parameter for covariance estimates
d = size(x,1);       % dimensionality of data

% Start with equal alpha estimates
alpha_est = ones(1,M)/M;

% Set initial mu as random M value pairs from data array
shuffledIndices = randperm(N);
mu = x(:,shuffledIndices(1:M));

% Assign each sample to the nearest mean
[~,assignedCentroidLabels] = min(pdist2(mu',x'),[],1);
% Use sample covariances of initial assignments as initial covariance
    estimates
for m = 1:M
    Sigma(:,:,m) = cov(x(:,find(assignedCentroidLabels==m))') + reg_weight
        *eye(d,d);
end

% Run EM algorith until it converges
t = 0;
Converged = 0;
while ~Converged
    % Calculate GMM distribution according to parameters
    for l = 1:M
        temp(l,:) = repmat(alpha_est(l),1,N).*evalGaussian(x,mu(:,l),Sigma
            (:,:,l));
    end
    pl_given_x = temp./sum(temp,1);

    % Calculate new alpha values
    alpha_new = mean(pl_given_x,2);

    % Clculate new mu values
    w = pl_given_x./repmat(sum(pl_given_x,2),1,N);
    mu_new = x*w';

    % Calculate new Sigma values
    for l = 1:M
        v = x-repmat(mu_new(:,l),1,N);
        u = repmat(w(l,:),d,1).*v;
        Sigma_new(:,:,l) = u*v' + reg_weight*eye(d,d); % adding a small
            regularization term
    end

    % Change in each parameter
    Dalpha = sum(abs(alpha_new-alpha_est'));
    Dmu = sum(sum(abs(mu_new-mu)));
    DSigma = sum(sum(abs(abs(Sigma_new-Sigma))));
```

```
        % Check if converged
        Converged = ((Dalpha+Dmu+DSigma)<delta);
        % Update old parameters
        alpha_est = alpha_new; mu = mu_new; Sigma = Sigma_new;
        %log_lik = sum(log(evalGMM(val_set,alpha_est,mu,Sigma)))
        %Converged = (log_lik <-2.3);
        t = t+1;
    end
end

function g = evalGaussian(x,mu,Sigma)
    % Evaluates the Gaussian pdf N(mu,Sigma) at each coumn of X
    [n,N] = size(x);
    invSigma = inv(Sigma);
    C = (2*pi)^(-n/2) * det(invSigma)^(1/2);
    E = -0.5*sum((x-repmat(mu,1,N)).*(invSigma*(x-repmat(mu,1,N))),1);
    g = C*exp(E);
end

function gmm = evalGMM(x,alpha,mu,Sigma)
    % Evaluates GMM on the grid based on parameter values given
    gmm = zeros(1,size(x,2));
    for m = 1:length(alpha)
        gmm = gmm + alpha(m)*evalGaussian(x,mu(:,m),Sigma(:,:,m));
    end
end
```