Luke Jaffe
CS6140 Machine Learning
10/20/15

HW3 – Generative Models

Problem 1. Gaussian Discriminant Analysis

A) Shared covariance matrix
Usage: python gda.py -s 1

Training accuracy:
Fold 1: 0.897826086957
Fold 2: 0.902415458937
Fold 3: 0.898550724638
Fold 4: 0.896376811594
Fold 5: 0.899033816425
Fold 6: 0.901449275362
Fold 7: 0.901449275362
Fold 8: 0.9
Fold 9: 0.902898550725
Fold 10: 0.903140096618
Average across k folds: 0.900314009662
Testing accuracy:
Fold 1: 0.87852494577
Fold 2: 0.917570498915
Fold 3: 0.887201735358
Fold 4: 0.898047722343
Fold 5: 0.891540130152
Fold 6: 0.887201735358
Fold 7: 0.885032537961
Fold 8: 0.891540130152
Fold 9: 0.90021691974
Fold 10: 0.887201735358
Average across k folds: 0.892407809111

B) Separate covariance matrices
Usage: python gda.py -s 0

Training accuracy:
Fold 1: 0.844927536232
Fold 2: 0.8538647343
Fold 3: 0.851449275362
Fold 4: 0.851207729469
Fold 5: 0.857004830918
Fold 6: 0.855555555556
Fold 7: 0.851449275362
Fold 8: 0.846376811594
Fold 9: 0.848550724638
Fold 10: 0.851690821256
Average across k folds: 0.851207729469
Testing accuracy:
Fold 1: 0.872017353579
Fold 2: 0.845986984816
Fold 3: 0.832971800434
Fold 4: 0.845986984816

Fold 5: 0.848156182213
Fold 6: 0.85032537961
Fold 7: 0.830802603037
Fold 8: 0.85032537961
Fold 9: 0.848156182213
Fold 10: 0.85032537961
Average across k folds: 0.847505422993

Problem 2.

1. Error Tables

A) Bernoulli
Usage: python nb.py -m “bernoulli”

Training error:
Fold 1: TPR=0.835684, FPR=0.054205, Error=0.097585
Fold 2: TPR=0.837523, FPR=0.052590, Error=0.095871
Fold 3: TPR=0.835071, FPR=0.054582, Error=0.098044
Fold 4: TPR=0.842525, FPR=0.054205, Error=0.094905
Fold 5: TPR=0.841912, FPR=0.052611, Error=0.094180
Fold 6: TPR=0.844975, FPR=0.051016, Error=0.092007
Fold 7: TPR=0.840686, FPR=0.050219, Error=0.093214
Fold 8: TPR=0.841912, FPR=0.050219, Error=0.092731
Fold 9: TPR=0.835784, FPR=0.049821, Error=0.094905
Fold 10: TPR=0.836397, FPR=0.052212, Error=0.096112
Average across k folds: TPR=0.839247, FPR=0.052168, Error=0.094955
Testing error:
Fold 1: TPR=0.857143, FPR=0.028674, Error=0.073753
Fold 2: TPR=0.813187, FPR=0.032374, Error=0.093478
Fold 3: TPR=0.890110, FPR=0.046763, Error=0.071739
Fold 4: TPR=0.839779, FPR=0.035842, Error=0.084783
Fold 5: TPR=0.812155, FPR=0.046595, Error=0.102174
Fold 6: TPR=0.773481, FPR=0.082437, Error=0.139130
Fold 7: TPR=0.812155, FPR=0.068100, Error=0.115217
Fold 8: TPR=0.839779, FPR=0.050179, Error=0.093478
Fold 9: TPR=0.861878, FPR=0.071685, Error=0.097826
Fold 10: TPR=0.867403, FPR=0.053763, Error=0.084783
Average across k folds: TPR=0.836707, FPR=0.051641, Error=0.095636

B) Gaussian
Usage: python nb.py -m “gaussian”

Training error:
Fold 1: TPR=0.894543, FPR=0.141491, Error=0.127295
Fold 2: TPR=0.872471, FPR=0.119522, Error=0.122676
Fold 3: TPR=0.841815, FPR=0.130677, Error=0.141512
Fold 4: TPR=0.828431, FPR=0.110801, Error=0.134750
Fold 5: TPR=0.814951, FPR=0.131527, Error=0.152620
Fold 6: TPR=0.843750, FPR=0.120367, Error=0.134509
Fold 7: TPR=0.854167, FPR=0.114388, Error=0.126781
Fold 8: TPR=0.847426, FPR=0.135911, Error=0.142478
Fold 9: TPR=0.854167, FPR=0.104823, Error=0.120985
Fold 10: TPR=0.848039, FPR=0.095656, Error=0.117846
Average across k folds: TPR=0.849976, FPR=0.120516, Error=0.132145
Testing error:

Fold 1: TPR=0.895604, FPR=0.136201, Error=0.123644
Fold 2: TPR=0.857143, FPR=0.122302, Error=0.130435
Fold 3: TPR=0.912088, FPR=0.111511, Error=0.102174
Fold 4: TPR=0.834254, FPR=0.107527, Error=0.130435
Fold 5: TPR=0.790055, FPR=0.086022, Error=0.134783
Fold 6: TPR=0.795580, FPR=0.129032, Error=0.158696
Fold 7: TPR=0.801105, FPR=0.111111, Error=0.145652
Fold 8: TPR=0.850829, FPR=0.150538, Error=0.150000
Fold 9: TPR=0.872928, FPR=0.118280, Error=0.121739
Fold 10: TPR=0.900552, FPR=0.129032, Error=0.117391
Average across k folds: TPR=0.851014, FPR=0.120155, Error=0.131495

C) Histogram (4 bins)
Usage: python nb.py -m “histogram” -b 4

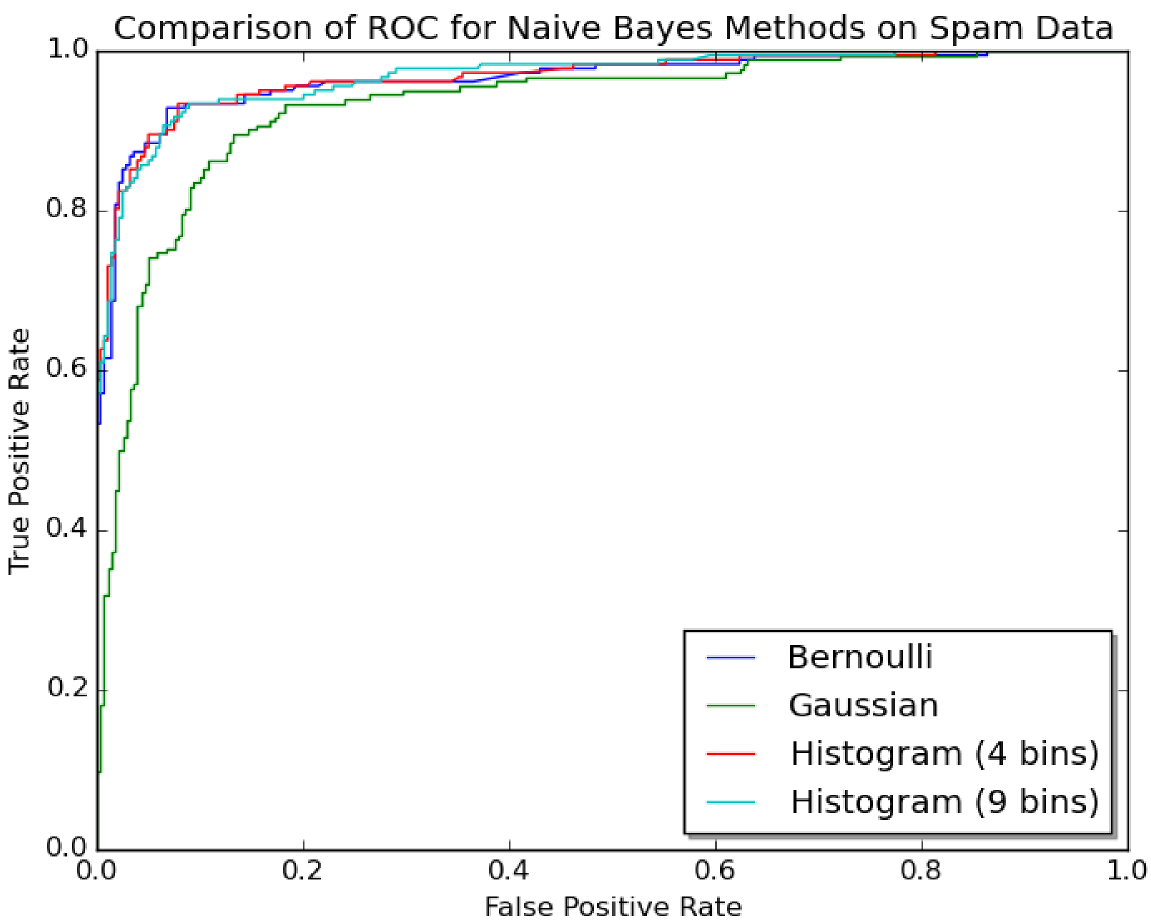
Training error:
Fold 1: TPR=0.830166, FPR=0.051016, Error=0.097826
Fold 2: TPR=0.835071, FPR=0.050996, Error=0.095871
Fold 3: TPR=0.828326, FPR=0.050199, Error=0.098044
Fold 4: TPR=0.835784, FPR=0.051813, Error=0.096112
Fold 5: TPR=0.836397, FPR=0.051415, Error=0.095629
Fold 6: TPR=0.837010, FPR=0.047828, Error=0.093214
Fold 7: TPR=0.837623, FPR=0.049821, Error=0.094180
Fold 8: TPR=0.831495, FPR=0.048226, Error=0.095629
Fold 9: TPR=0.829657, FPR=0.047828, Error=0.096112
Fold 10: TPR=0.832108, FPR=0.053009, Error=0.098285
Average accross k folds: TPR=0.833364, FPR=0.050215, Error=0.096090
Testing error:
Fold 1: TPR=0.862637, FPR=0.043011, Error=0.080260
Fold 2: TPR=0.796703, FPR=0.035971, Error=0.102174
Fold 3: TPR=0.901099, FPR=0.053957, Error=0.071739
Fold 4: TPR=0.817680, FPR=0.046595, Error=0.100000
Fold 5: TPR=0.817680, FPR=0.043011, Error=0.097826
Fold 6: TPR=0.767956, FPR=0.060932, Error=0.128261
Fold 7: TPR=0.790055, FPR=0.053763, Error=0.115217
Fold 8: TPR=0.828729, FPR=0.053763, Error=0.100000
Fold 9: TPR=0.872928, FPR=0.075269, Error=0.095652
Fold 10: TPR=0.845304, FPR=0.046595, Error=0.089130
Average across k folds: TPR=0.830077, FPR=0.051287, Error=0.098026

D) Histogram (9 bins)
Usage: python nb.py -m “histogram” -b 9

Training error:
Fold 1: TPR=0.879828, FPR=0.091271, Error=0.102657
Fold 2: TPR=0.887186, FPR=0.092430, Error=0.100459
Fold 3: TPR=0.881668, FPR=0.090837, Error=0.101666
Fold 4: TPR=0.881127, FPR=0.087286, Error=0.099734
Fold 5: TPR=0.882353, FPR=0.089279, Error=0.100459
Fold 6: TPR=0.884804, FPR=0.088083, Error=0.098768
Fold 7: TPR=0.885417, FPR=0.089279, Error=0.099251
Fold 8: TPR=0.883578, FPR=0.088481, Error=0.099493
Fold 9: TPR=0.881127, FPR=0.081706, Error=0.096354
Fold 10: TPR=0.878676, FPR=0.087684, Error=0.100942
Average across k folds: TPR=0.882577, FPR=0.088634, Error=0.099978
Testing error:
Fold 1: TPR=0.895604, FPR=0.060932, Error=0.078091

Fold 2: TPR=0.868132, FPR=0.075540, Error=0.097826
Fold 3: TPR=0.923077, FPR=0.097122, Error=0.089130
Fold 4: TPR=0.878453, FPR=0.082437, Error=0.097826
Fold 5: TPR=0.878453, FPR=0.078853, Error=0.095652
Fold 6: TPR=0.834254, FPR=0.096774, Error=0.123913
Fold 7: TPR=0.867403, FPR=0.107527, Error=0.117391
Fold 8: TPR=0.861878, FPR=0.093190, Error=0.110870
Fold 9: TPR=0.900552, FPR=0.125448, Error=0.115217
Fold 10: TPR=0.928177, FPR=0.103943, Error=0.091304
Average across k folds: TPR=0.883598, FPR=0.092177, Error=0.101722

2. ROC Curves



3. AUC

A) Bernoulli: AUC = 0.957845917523

B) Gaussian: AUC = 0.924809957068

C) Histogram (4 bins): AUC = 0.95988420182

D) Histogram (9 bins): AUC = 0.96034700067

Problem 3. EM Algorithm

```
function [label, model, llh] = emgm(X, init)                 % emgm function definition
% Perform EM algorithm for fitting the Gaussian mixture model.
%   X: d x n data matrix
%   init: k (1 x 1) or label (1 x n, 1<=label(i)<=k) or center (d x k)
% Written by Michael Chen (sth4nth@gmail.com).
%% initialization
fprintf('EM for Gaussian mixture: running ... \n');
R = initialization(X,init);                                  % Initialize the assignment of elements to clusters
[~,label(1,:)] = max(R,[],2);
R = R(:,unique(label));

tol = 1e-10;                                                 % Initialize variables, such as max iterations allowed
maxiter = 500;
llh = -inf(1,maxiter);
converged = false;
t = 1;
while ~converged && t < maxiter i                            % Iterate until convergence or max iterations reached
   t = t+1;
   model = maximization(X,R);                                % Maximization step
   [R, llh(t)] = expectation(X,model);                       % Expectation step

   [~,label(:)] = max(R,[],2);
   u = unique(label);   % non-empty components
   if size(R,2) ~= size(u,2)                                 % Test for convergence
      R = R(:,u);   % remove empty components
   else
      converged = llh(t)-llh(t-1) < tol*abs(llh(t));
   end

end
llh = llh(2:t);
if converged
   fprintf('Converged in %d steps.\n',t-1);                  % Convergence condition reached
else
   fprintf('Not converged in %d steps.\n',maxiter);          % Finished max iterations before convergence
end

function R = initialization(X, init)
[d,n] = size(X);
if isstruct(init)  % initialize with a model                 % Use a predetermined model
   R  = expectation(X,init);
elseif length(init) == 1  % random initialization            % Use k Gaussians as the model
   k = init;
   idx = randsample(n,k);                                    % Randomly generate k numbers in [1,n]
   m = X(:,idx);                                             % Get the values at those indices in k dimensions
   [~,label] = max(bsxfun(@minus,m'*X,dot(m,m,1)'/2),[],1);
   [u,~,label] = unique(label);
   while k ~= length(u)
      idx = randsample(n,k);
      m = X(:,idx);
      [~,label] = max(bsxfun(@minus,m'*X,dot(m,m,1)'/2),[],1);
      [u,~,label] = unique(label);
   end
   R = full(sparse(1:n,label,1,n,k,n));                      % Create the soft membership vector (same as Zim)
elseif size(init,1) == 1 && size(init,2) == n  % initialize with labels
   label = init;
```

```
    k = max(label);
    R = full(sparse(1:n,label,1,n,k,n));
elseif size(init,1) == d  %initialize with only centers
    k = size(init,2);
    m = init;
    [~,label] = max(bsxfun(@minus,m'*X,dot(m,m,1)'/2),[],1);
    R = full(sparse(1:n,label,1,n,k,n));
else
    error('ERROR: init is not valid.');
end

function [R, llh] = expectation(X, model)                 % Estimate the soft membership values for all data
mu = model.mu;
Sigma = model.Sigma;
w = model.weight;

n = size(X,2);
k = size(mu,2);
logRho = zeros(n,k);

for i = 1:k
    logRho(:,i) = loggausspdf(X,mu(:,i),Sigma(:,:,i));   % Evaluate the multivariate Gaussian for each cluster
end
logRho = bsxfun(@plus,logRho,log(w));                    % Incorporate the model weight
T = logsumexp(logRho,2);
llh = sum(T)/n;                                          % Calculate the log likelihood
logR = bsxfun(@minus,logRho,T);
R = exp(logR);                                           % Reassign the soft membership vector Zim


function model = maximization(X, R)                      % Estimate the model parameters
[d,n] = size(X);
k = size(R,2);

nk = sum(R,1);
w = nk/n;                                                % Weight is sum of soft membership / total points
mu = bsxfun(@times, X*R, 1./nk);                         % Calculate the mean using soft membership vector

Sigma = zeros(d,d,k);                                    % Initialize the covariance matrix Sigma
sqrtR = sqrt(R);
for i = 1:k                                              % Calculate the covariance matrix for each Gaussian
    Xo = bsxfun(@minus,X,mu(:,i));
    Xo = bsxfun(@times,Xo,sqrtR(:,i)');
    Sigma(:,:,i) = Xo*Xo'/nk(i);
    Sigma(:,:,i) = Sigma(:,:,i)+eye(d)*(1e-6);           % Add a prior for numerical stability
end

model.mu = mu;
model.Sigma = Sigma;
model.weight = w;

function y = loggausspdf(X, mu, Sigma)                   % Formula to evaluate the (log)multivariate Gaussian
d = size(X,1);                                           % function with data, mean, and covariance
X = bsxfun(@minus,X,mu);
[U,p]= chol(Sigma);
if p ~= 0
    error('ERROR: Sigma is not PD.');
end
Q = U'\X;
```

```
q = dot(Q,Q,1);  % quadratic term (M distance)
c = d*log(2*pi)+2*sum(log(diag(U)));   % normalization constant
y = -(c+q)/2;
```

Problem 4. EM on simple data

A) 2 Gaussian Dataset
Usage: python em.py -g 2

Converged after 36 iterations:
Gaussian 1:
    Elements: 2008.96425321 (soft membership)
    Mean: [2.99431914 3.05207381]
    Covariance: [[1.01055502 0.02715647] [0.02715647 2.93768741]]
Gaussian 2:
    Elements: 3991.03574679 (soft membership)
    Mean: [7.01324501 3.9831899]
    Covariance : [[0.97460111 0.4973741] [0.4973741   1.00107128]]

B) 3 Gaussian Dataset
Usage: python em.py -g 3

Converged after 77 iterations:
Gaussian 1 :
    Elements: 2983.84641442 (soft membership)
    Mean: [7.02196217 4.0156787]
    Covariance: [[0.98974532 0.50069764] [0.50069764 0.99563133]]
Gaussian 2 :
    Elements: 4955.8106475 (soft membership)
    Mean: [5.01264988 7.00229035]
    Covariance: [[0.97860395 0.18445926] [0.18445926 0.97345893]]
Gaussian 3 :
    Elements: 2060.34293808 (soft membership)
    Mean: [3.04158658 3.05385045]
    Covariance: [[1.02997817 0.03036423] [0.03036423 3.39553018]]

Problem 5.

A) Prove that:

P(A|B,C) = P(B|A,C)*P(A|C) / P(B|C)

P(A|B,C) = P(A,B,C) / P(B,C)

    P(A,B,C) = P(B|A,C)*P(A,C)

P(A|B,C) = P(B|A,C)*P(A,C) / P(B,C)

$P(A,C) = P(A|C)*P(C)$

$P(B,C) = P(B|C)*P(C)$

$P(A|B,C) = P(B|A,C)*P(A|C)*P(C) / [P(B|C)*P(C)]$

Thus, $P(A|B,C) = P(B|A,C)*P(A|C) / P(B|C)$

B) Coin Problem

-coin is either fair or double-headed
P{prior} = F/(F+1)

P{not fair | data} = 0.5
P{not fair | data} = P{data | not fair} * P{prior} / P{data}

Need to solve for P{data | not fair} and P{data}

Assuming each flip is independent, we can model P{data | not fair} as:
P{d1 | not fair}* P{d2 | not fair}* ...*P{dn | not fair}

n is the number of flips, and also the number of heads in a row required to set the likelihood = 0.5, and solve for F.

The problem can clearly be modeled with Bernoulli trials:
P{k out of n} = kCn * p^k * (1-p)^(n-k)
set n = n, since every flip will be a heads:
P{n out of n} = nCn * p^n * (1-p)*(n-n) = 1 * p^n * 1 = p^n
Substitute in our prior for p, and we get:
P{data | not fair} = (F/(F+1))^n

In this case, P{data} = 1

So we have:
P{not fair | data} = (F/(F+1))^n * (F/(F+1)) = (F/(F+1))^(n+1) = 0.5

Take the log of both sides:
log((F/(F+1))^(n+1)) = log(0.5) →
(n+1)*log(F/(F+1)) = log(0.5) →
n+1 = log(0.5) / log(F/(F+1))

Therefore,
n = [log(0.5) / log(F/(F+1))] - 1