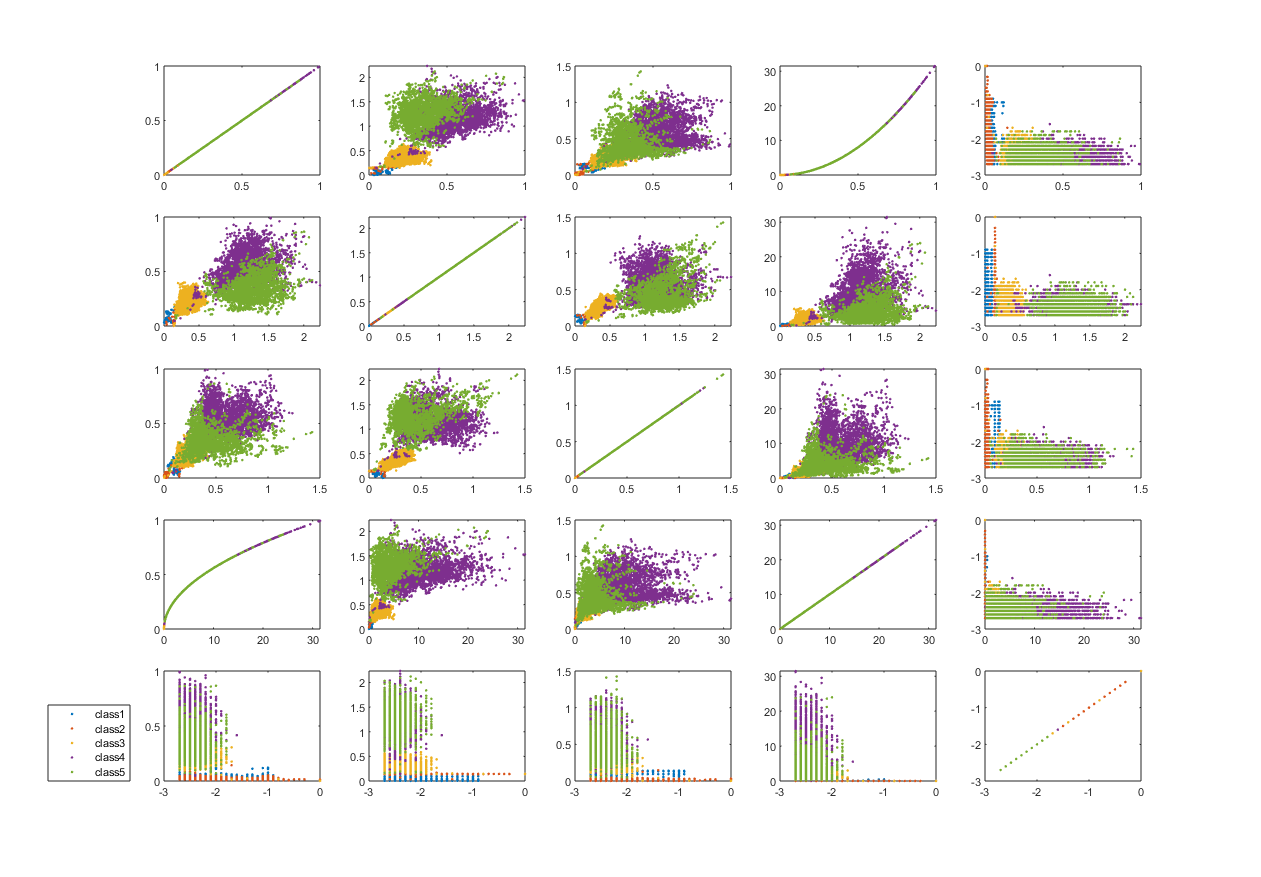
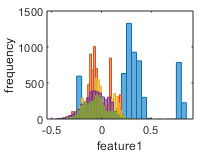
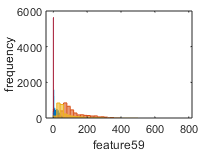
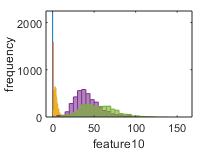
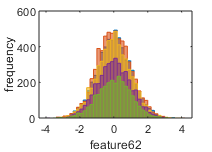
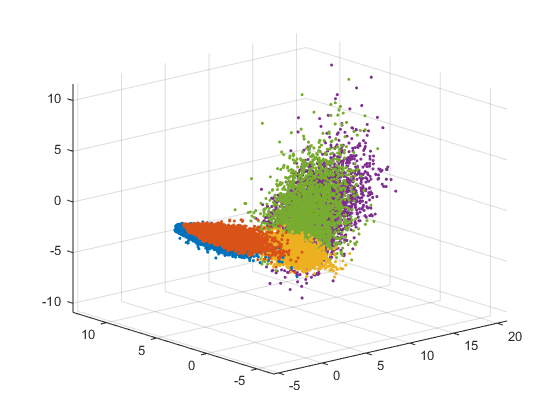
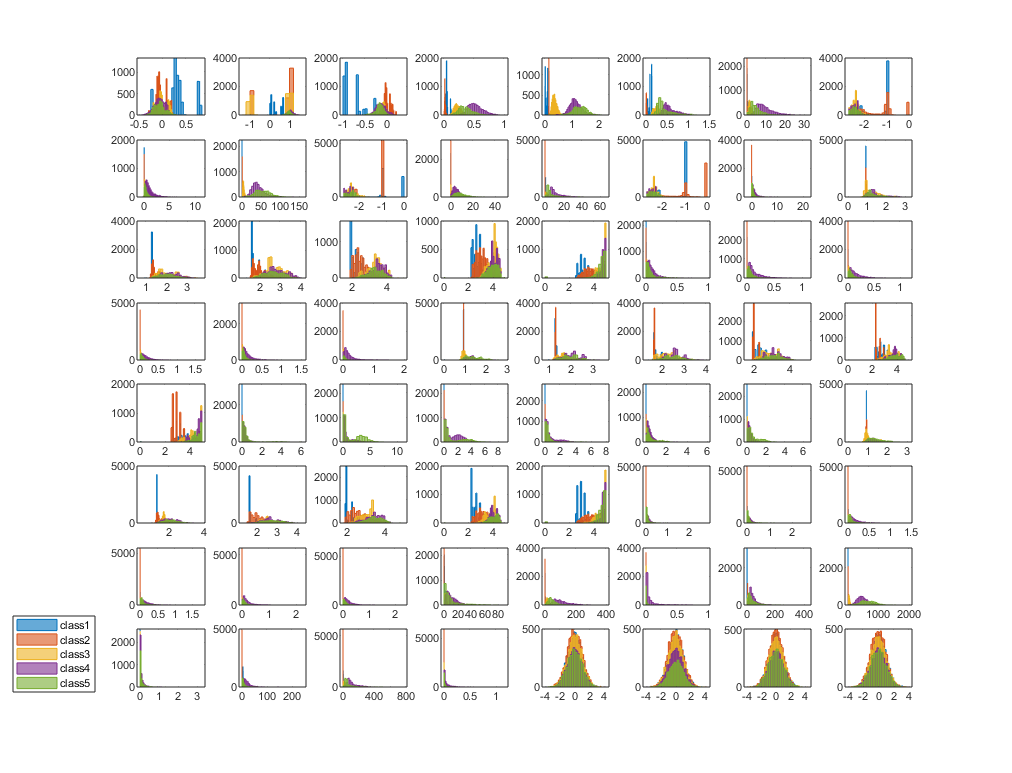
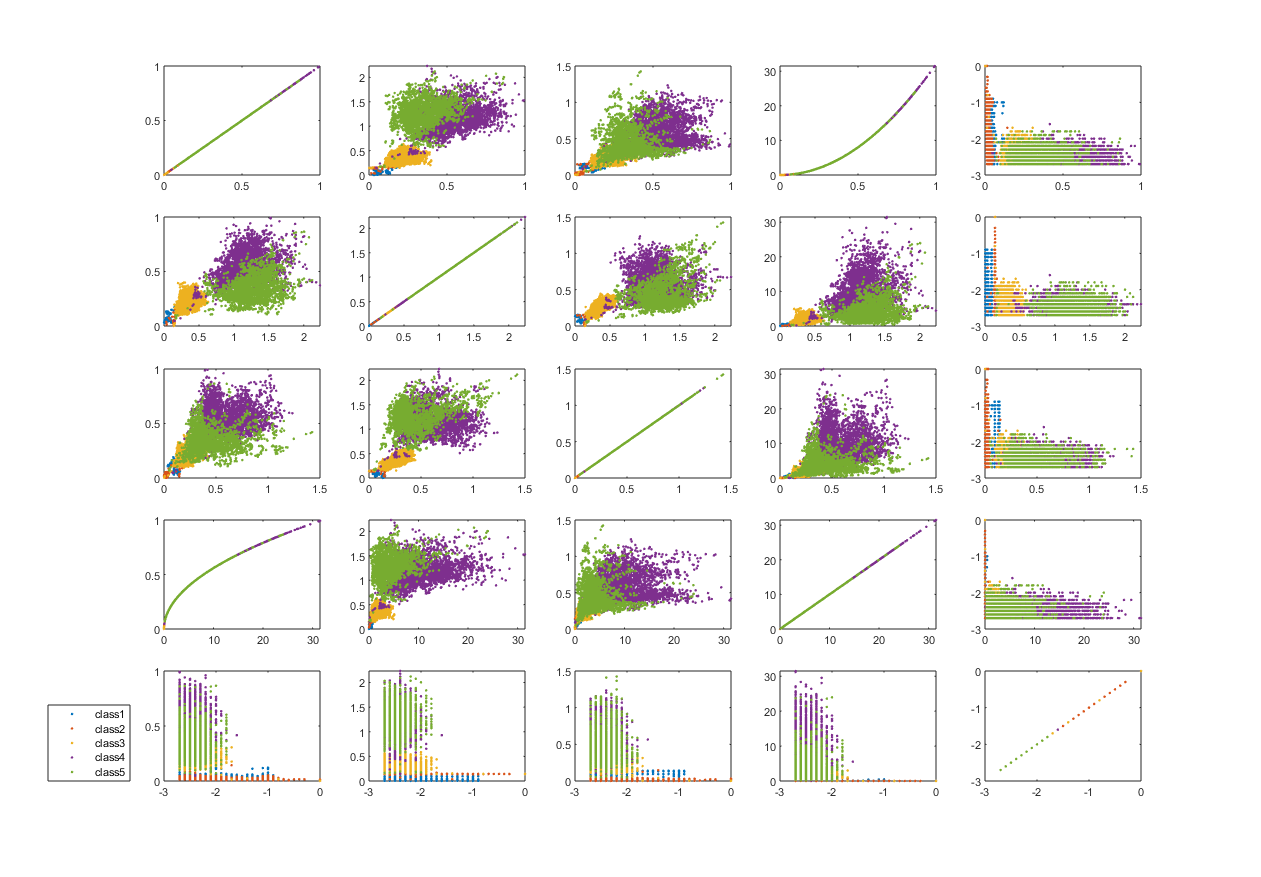
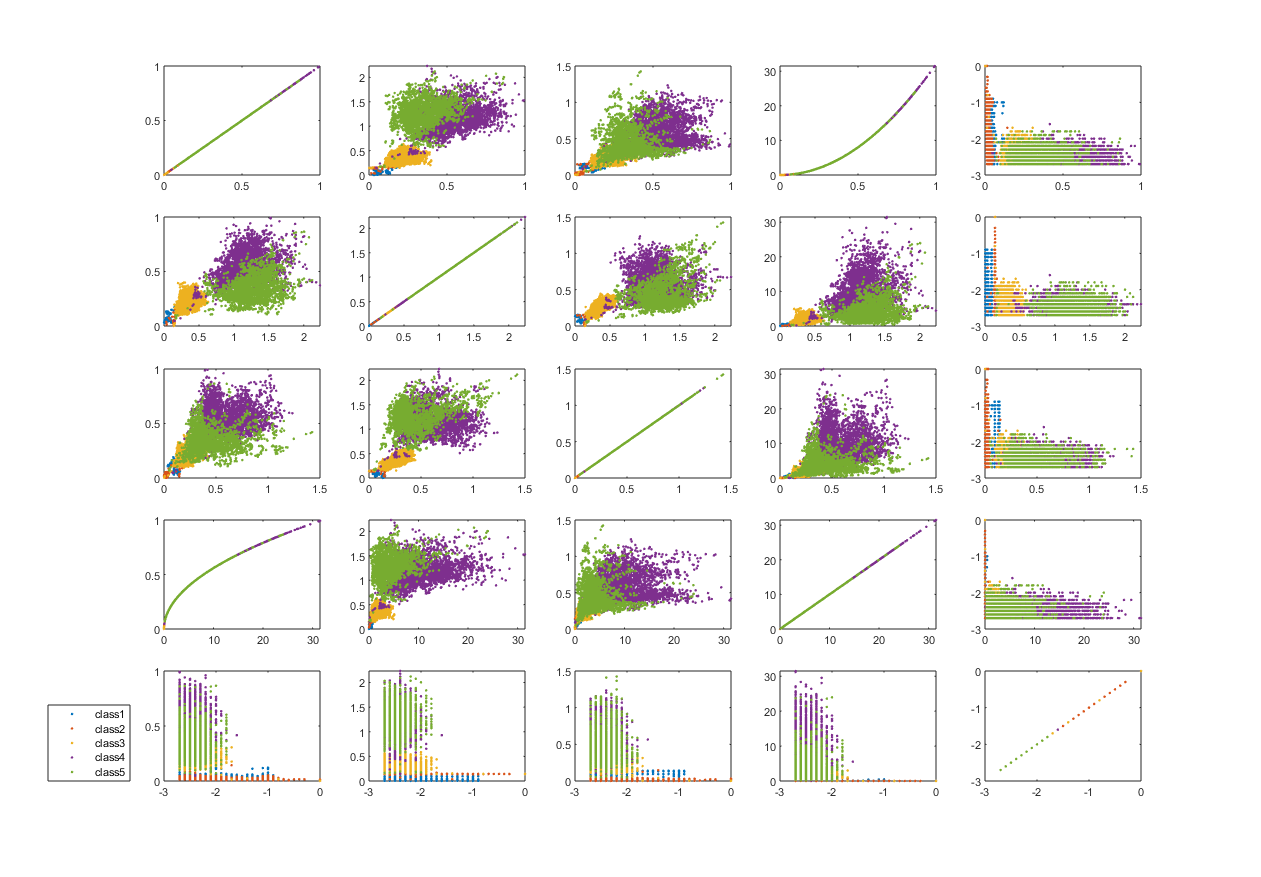
**Coursework2 Report-Luxi ZHANG**

**Q1.** I visualize the dataset to observe the dataset with several methods. (code showing in appendix)

(a) 3D distribution after PCA (b) Part of distribution against each feature

Feature6

Feature5

Feature4

Feature3

Feature6

Feature5

Feature4

Feature3

(c) Part of multiclass scatterplot matrix

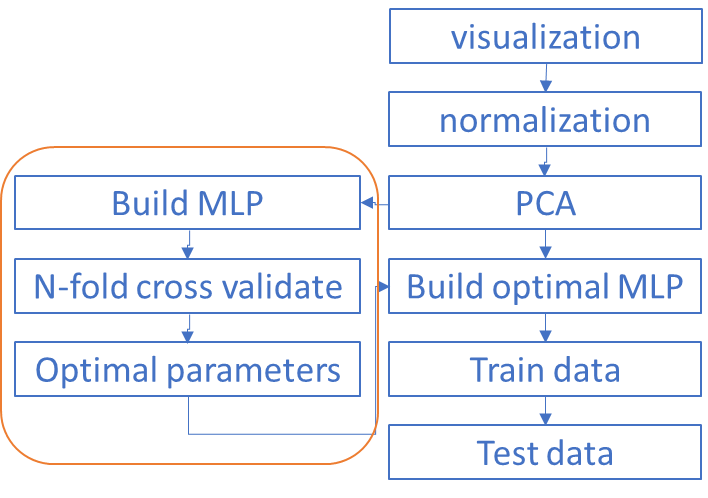
Firgure1 Visualization of the data

Figure 1(a) is a 3-dimensional scatter plot by PCA, it remains about 58% variance, figure 1(b) is a part of distribution against each feature, and figure 1(c) is a part of multiclass scatterplot matrix. (the whole figures are too large to put here, so I only put a part here to explain what is happening)

From figure 1(a), we can find these five classes **cannot** be classified well by a linear method, so algorithm suitable for non-linear classification will be chosen. In addition, the boundaries between some classes are not so clear, so it is a challenge in classification.

From figure 1(b), we can find scale of each feature varies (the absolute value from no more than 1 to hundreds), so **normalization** will be needed in pre-processing. In addition, the data set has approximate normal distribution in some features such as feature62, while it may not so close to distribution in some other features, such as feature1, 10 and 59. As a result, if we use Gaussian distribution to fit every feature, error may occur.

From figure 1(c), we can find some features are associated, such feature3 against feature6, so it will be efficient to implement classification if we use **principal component analysis** to reduce the dimension/ number of features. What’s more, these dependent features will also lead to error in fitting with Gaussian distribution.

Based on analysis above, the classification pipeline shows on the left (figure 2).

In pre-processing steps, I normalize the data and do PCA to transform all features in the same scale and reduce the dimension. It makes sense to the efficiency and accuracy.

KNN and MLP maybe better choices for this classification based on the analysis. Gaussian Naïve Bayes will work fast, but because the date doesn’t fulfil the assumptions very well, its output may not as accurate as the previous ones. Deep learning can solve the task accurately, but the time cost is inacceptable. In addition, I also implement them to compared as following. (average accuracy and time cost from N-fold cross validate, n=5.)

Firgure2 Classification pipeline

Table1 Comparison of three classifiers

|  |  |  |  |
| --- | --- | --- | --- |
|  | Gaussian Naïve Bayes | KNN (optimal k=3) | BP (30 hidden units, λ=1, iterations=50) |
| Accuracy | 97.08% | 98.30% | 98.77% |
| time cost | 0.04 sec | 3.52 sec | 11.6 sec |

I select the MLP (3 layers, 1 hidden layer) with Backpropagation method because its more accurate than KNN and the addition time cost (in second scale) is acceptable. In addition, the time cost for KNN increases fast with the increase of the size of test set while time cost for BP methods has no significant change. To implement the BP method, we should find the optimal structure for the task, we should decide the number of units in hidden layer, lambda for overfitting penalty and suitable numbers of iteration. Then for the new dataset from the same task we can use the optimal parameters to build MLP to classify them.

**Q2.** In Function TrainClassifierX, I did the following to implement training. (code showing in appendix)

Pre-Process

Normalization, return parameter σ

Implement PCA (remaining 99% variance), return parameter

Train the with BP method

Set the parameters for network structure

Initialization weight matrix

Repeat (number of iterations)

implement forward propagation to get for any ,(sigmoid function)

Compute cost function with penalty on overfitting

Implement Backpropagation to compute partial derivatives

Implement advanced optimization method with backpropagation to try to minimize

End

Return parameter

In ClassifyX, I did the following to implement classification/predict. (code showing in appendix)

Pre-Process

Normalization (with parameter σ)

Implement PCA (with parameter )

Predict test dataset

Compute the output layer (with parameter )

Predict as the maximum class

**Q3.** There are three main parameters need to be chosen in my implement, number of iterations, hidden layer units and λ to penalty overfitting. To get better parameters, I use for loops to modify and set these parameters as inputs of TrainClassifierX function, and then used n-fold (n=5) cross validate method to choose parameters that generate better performance based on the validate set.

To evaluate the performance, both accuracy and efficiency should be taken into consideration, so I calculated accuracy, confusion matrice and time cost. The accuracy is computed based on the validate set and the ratio of size of training dataset to size of validate dataset is 4:1. Time cost is calculated by ‘tic’ and ‘toc’ in MATLAB. The time cost is the average time cost of TrainClassifierX Function and ClassifyX Function in n-fold (n=5) cross validate test. (Figures in the following tables are approximate because the datasets are assigned randomly each time running the code. code showing in appendix)

**Number of iterations**

Table2 Comparison in different numbers of iterations

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 10 | 20 | 30 | 40 | 50 | 60 | 70 | 80 | 90 | 100 |
| accuracy | 93.88% | 97.50% | 98.11% | 98.38% | 98.72% | 98.84% | 98.98% | 99.09% | 99.08% | 99.09% |
| time cost/sec | 2.5133 | 4.6468 | 6.5879 | 9.1332 | 11.0948 | 13.8371 | 16.3175 | 18.9022 | 21.9399 | 23.7548 |

We can find that with implement the algorithm with more interations can get more accurate output but the time cost increase as well, so it should stop at a suitable iteration. I sected 70 as the number of iterations.

**Number of hidden units and overfitting penalty λ**

Table3 Comparison in different combination of number of hidden units and λ

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | 20 hidden units | | | 30 hidden units | | | 40 hidden units | | |
| λ=0 | λ=1 | λ=3 | λ=0 | λ=1 | λ=3 | λ=0 | λ=1 | λ=3 |
| accuracy | 98.89% | 98.86% | 98.78% | 98.92% | 98.99% | 98.81% | 99.02% | 99.03% | 98.91% |
| time cost/sec | 15.11 | 15.12 | 15.41 | 16.72 | 17.27 | 17.63 | 18.30 | 18.53 s | 18.37 s |

Some of the trials of combination show in Table 3 (70 iterations). We can find that large λ makes slow convergence but do well in avoiding overfitting and increasing hidden units contributes to accurate output, but it leads to the slight increase of time cost as well. To make a balance, I choose λ=1 and units hidden units=30.

**Classification accuracy and confusion matrix**

With the given dataset (size=24000), I took 80% as training data(size=19200), and 20% as test data(size=4800). For 30 hidden units, λ=1 and 70 iterations, the confusion matrix is following.

Table4 Confusion matrix

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | predict=1 | predict=2 | predict=3 | predict=4 | predict=5 | total |
| label=1 | 1136 | 0 | 0 | 0 | 0 | 1136 |
| label=2 | 1 | 1234 | 2 | 0 | 0 | 1237 |
| label=3 | 0 | 6 | 1100 | 3 | 0 | 1109 |
| label=4 | 0 | 0 | 5 | 764 | 16 | 785 |
| label=5 | 0 | 2 | 0 | 12 | 519 | 533 |
| total | 1137 | 1242 | 1107 | 779 | 535 | 4800 |

**Advantages:**

This algorithm performs well in the non-linear multiple classification, especially in accuracy, and the time cost is acceptable.

It takes several parameters to do predict tasks. Once the training completed, we can use the trained parameters to do classification in limited time, not like non-parameter method KNN. In KNN, the time cost increases with the increase of the size of test set, because it resorts distances.

This algorithm takes into the computation of cost, to reducing overfitting.

**Disadvantages:**

It needs time to get convergence. With small number of iterations, it gives worse predict. As a result, it doesn’t run as fast as Gaussian Naïve Bayes Classifier (accuracy=97%, time cost=0.04sec). In order to get a better predict, it sacrifices some time. If we don’t have such needs on accuracy, Gaussian Naïve Bayes Classifier can be an alternative method.

**Appendix**

**Q1 visualization.m**

clear;

load data

%n=size(data,1);

%elems = randperm(n)';

inputs=data(:,2:65);

labels=data(:,1);

%% each feature

[m, m1] = size(inputs);

idx1=labels==1;

idx2=labels==2;

idx3=labels==3;

idx4=labels==4;

idx5=labels==5;

for i=1:64

%5 classes againest each feature

subplot(8,8,i),h=histogram(inputs(idx1,i));h.EdgeColor=h.FaceColor;hold on;

subplot(8,8,i),h=histogram(inputs(idx2,i));h.EdgeColor=h.FaceColor;hold on;

subplot(8,8,i),h=histogram(inputs(idx3,i));h.EdgeColor=h.FaceColor;hold on;

subplot(8,8,i),h=histogram(inputs(idx4,i));h.EdgeColor=h.FaceColor;hold on;

subplot(8,8,i),h=histogram(inputs(idx5,i));h.EdgeColor=h.FaceColor;

%xlabel(['feature',num2str(i)]);

%ylabel('frequency');

end

%% scatter matrix

figure;

[m, m1] = size(inputs);

idx1=find(labels==1);

idx2=find(labels==2);

idx3=find(labels==3);

idx4=find(labels==4);

idx5=find(labels==5);

f=4;%number of features

for i=1:f

for j=1:f

%5 classes in scatter matrix

subplot(f,f,(i-1)\*f+j),plot(inputs(idx1,i),inputs(idx1,j),'.');hold on;

subplot(f,f,(i-1)\*f+j),plot(inputs(idx2,i),inputs(idx2,j),'.');hold on;

subplot(f,f,(i-1)\*f+j),plot(inputs(idx3,i),inputs(idx3,j),'.');hold on;

subplot(f,f,(i-1)\*f+j),plot(inputs(idx4,i),inputs(idx4,j),'.');hold on;

subplot(f,f,(i-1)\*f+j),plot(inputs(idx5,i),inputs(idx5,j),'.');

%{

% to plot part of the scatter matrix

ii=i+57;

jj=j+57;

subplot(f,f,(i-1)\*f+j),plot(inputs(idx1,ii),inputs(idx1,jj),'.');hold on;

subplot(f,f,(i-1)\*f+j),plot(inputs(idx2,ii),inputs(idx2,jj),'.');hold on;

subplot(f,f,(i-1)\*f+j),plot(inputs(idx3,ii),inputs(idx3,jj),'.');hold on;

subplot(f,f,(i-1)\*f+j),plot(inputs(idx4,ii),inputs(idx4,jj),'.');hold on;

subplot(f,f,(i-1)\*f+j),plot(inputs(idx5,ii),inputs(idx5,jj),'.');

%}

end

end

%% 3d

figure

%implement normalization

mu = mean(inputs,1);%normalization parameters

inputs = bsxfun(@minus, inputs, mu);

sigma = std(inputs,1);%normalization parameters

inputs = bsxfun(@rdivide, inputs, sigma);

%implement PCA

[m, m1] = size(inputs);

Sigma=1/m\*(inputs')\*inputs;

[U,S,~]=svd(Sigma);%compute U S (V)

%reduce to 3D

inputs=inputs\*U(:,1:3);

%compute variance

variance=sum(sum(S(1:3,1:30)))/sum(sum(S));

for i=1:5

idx=find(labels==i);

plot3(inputs(idx,1),inputs(idx,2),inputs(idx,3),'.');

hold on;

end

legend('class1','class2','class3','class4','class5')

grid on

**Q2 TrainClassifierX.m**

function parameters = TrainClassifierX(inputs, output)%,lambda,units,iter

%implement normalization

parameters.mu = mean(inputs,1);%normalization parameters

inputs = bsxfun(@minus, inputs, parameters.mu);

parameters.sigma = std(inputs,1);%normalization parameters

inputs = bsxfun(@rdivide, inputs, parameters.sigma);

%implement PCA

[m, m1] = size(inputs);

Sigma=1/m\*(inputs')\*inputs;

[U,S,~]=svd(Sigma);%compute U S (V)

%find the value k to remain 99% variance

for K=1:m1

if (sum(sum(S(1:K,1:K)))/sum(sum(S))>=0.99)

break;

end

end

inputs=inputs\*U(:,1:K);%Pca on inputs

parameters.U=U(:,1:K);%Pca paprameters

%initialize papameters for bp network

parameters.input\_layer\_size=K; %number of features after pca

parameters.hidden\_layer\_size=30;%units; %units in hidden layer

parameters.num\_labels=5; % 5 labels

%randam initialze weighs

initial\_Theta1 = randInitializeWeights(parameters.input\_layer\_size,parameters.hidden\_layer\_size);

initial\_Theta2 = randInitializeWeights(parameters.hidden\_layer\_size,parameters.num\_labels);

% Unroll parameters

initial\_nn\_params = [initial\_Theta1(:) ; initial\_Theta2(:)];

% set lambda and iter

lambda = 1;

iter=70;

% Create "short hand" for the cost function to be minimized

costFunction = @(p) nnCostFunction(p, ...

parameters.input\_layer\_size, ...

parameters.hidden\_layer\_size, ...

parameters.num\_labels,...

inputs, output, lambda);

% Now, costFunction takes in only one argument (the neural network parameters)

[nn\_params, ~] = fmincg(costFunction, initial\_nn\_params, iter);

% Obtain Theta1 and Theta2 back from nn\_params

parameters.Theta1 = reshape(nn\_params(1:parameters.hidden\_layer\_size \* ...

(parameters.input\_layer\_size + 1)), parameters.hidden\_layer\_size, ...

(parameters.input\_layer\_size + 1));

parameters.Theta2 = reshape(nn\_params((1 + (parameters.hidden\_layer\_size \* ...

(parameters.input\_layer\_size + 1))):end),...

parameters.num\_labels, (parameters.hidden\_layer\_size + 1));

end

function g = sigmoid(z)

%Compute sigmoid functoon

g = 1.0 ./ (1.0 + exp(-z));

end

function g = sigmoidGradient(z)

%returns the gradient of the sigmoid function evaluated at z

g=sigmoid(z).\*(1-sigmoid(z));

end

function W = randInitializeWeights(L\_in, L\_out)

%Randomly initialize the weights of a layer with L\_in

%incoming connections and L\_out outgoing connections

%The first column of W corresponds to the parameters for the bias unit

epsilon\_init=0.12;

W=rand(L\_out, 1+L\_in)\*2\*epsilon\_init-epsilon\_init;

end

function [J,grad] = nnCostFunction(nn\_params,input\_layer\_size, ...

hidden\_layer\_size, ...

num\_labels,X, y, lambda)

%Implements the neural network cost function for a two layer

%neural network which performs classification

% Reshape nn\_params back into the parameters Theta1 and Theta2,

% the weight matrices for our 2 layer neural network

Theta1 = reshape(nn\_params(1:hidden\_layer\_size \* (input\_layer\_size + 1)), ...

hidden\_layer\_size, (input\_layer\_size + 1));

Theta2 = reshape(nn\_params((1 + (hidden\_layer\_size \* (input\_layer\_size + 1))):end), ...

num\_labels, (hidden\_layer\_size + 1));

% Setup some variables

m = size(X, 1);

J = 0;

%Theta1\_grad = zeros(size(Theta1));

%Theta2\_grad = zeros(size(Theta2));

% Feedforward the neural network and return the cost in the variable J.

a1=[ones(m,1) X];

z2=a1\*Theta1';

a2=[ones(m,1) sigmoid(z2)];

z3=a2\*Theta2';

a3=sigmoid(z3);

%cost without regularization

for k=1:size(Theta2,1)

%index=find(y==k);

J=J+1/m\*sum(-(y==k)'\*log(a3(:,k))-(1-(y==k))'\*(log(1-a3(:,k))));

end

%cost with regularization

J=J+lambda/2/m\*(sum(sum(Theta1(:,2:end).^2))+sum(sum(Theta2(:,2:end).^2)));

%bp

%d3=zeros(size(a3));

%d2=zeros(size(a2));

% Implement the backpropagation algorithm to compute the gradients

% Theta1\_grad and Theta2\_grad.

Delta1 = zeros(size(Theta1));

Delta2 = zeros(size(Theta2));

for i=1:m

yi=zeros(1,size(Theta2,1));

yi(y(i))=1;

delta3=a3(i,:)-yi;

t=Theta2'\*delta3';

delta2 = t(2:end,:) .\* sigmoidGradient(z2(i, :)');

% delta2 = t(2:end,:) .\* sigmoidGradient(z2(i, :)');

Delta1 = Delta1 + delta2\* a1(i, :);

% Delta1 = Delta1 + delta2(2:end) \* X(i, :);

Delta2 = Delta2 + delta3' \* a2(i,:);

end

%Implement regularization with the gradients.

Theta1\_grad = Delta1 / m;

Theta1\_grad(:, 2:end) = Theta1\_grad(:, 2:end) + lambda/m\*Theta1(:, 2:end);

Theta2\_grad = Delta2 / m;

Theta2\_grad(:, 2:end) = Theta2\_grad(:, 2:end) + lambda/m\*Theta2(:, 2:end);

% Unroll gradients

grad = [Theta1\_grad(:) ; Theta2\_grad(:)];

end

function [X, fX, i] = fmincg(f, X, length)

% Minimize a continuous differentialble multivariate function.

RHO = 0.01; % a bunch of constants for line searches

SIG = 0.5; % RHO and SIG are the constants in the Wolfe-Powell conditions

INT = 0.1; % don't reevaluate within 0.1 of the limit of the current bracket

EXT = 3.0; % extrapolate maximum 3 times the current bracket

MAX = 20; % max 20 function evaluations per line search

RATIO = 100; % maximum allowed slope ratio

argstr = ['feval(f, X)']; % compose string used to call function

i = 0; % zero the run length counter

ls\_failed = 0; % no previous line search has failed

fX = [];

[f1,df1] = eval(argstr); % get function value and gradient

i = i + (length<0);

s = -df1;% search direction is steepest

d1 = -s'\*s; % this is the slope

z1 = 1/(1-d1);% initial step

while i < abs(length) % while not finished

i = i + (length>0);

X0 = X; f0 = f1; df0 = df1; % make a copy of current values

X = X + z1\*s; % begin line search

[f2,df2] = eval(argstr);

i = i + (length<0);

d2 = df2'\*s;

f3 = f1; d3 = d1; z3 = -z1; % initialize point 3 equal to point 1

%if length>0, M = MAX; else M = min(MAX, -length-i); end

M = MAX;

success = 0; limit = -1;% initialize quanteties

while 1

while ((f2 > f1+z1\*RHO\*d1) || (d2 > -SIG\*d1)) && (M > 0)

limit = z1; % tighten the bracket

if f2 > f1

z2 = z3 - (0.5\*d3\*z3\*z3)/(d3\*z3+f2-f3); % quadratic fit

else

A = 6\*(f2-f3)/z3+3\*(d2+d3); % cubic fit

B = 3\*(f3-f2)-z3\*(d3+2\*d2);

z2 = (sqrt(B\*B-A\*d2\*z3\*z3)-B)/A;

end

if isnan(z2) || isinf(z2)

z2 = z3/2; % if we had a numerical problem then bisect

end

z2 = max(min(z2, INT\*z3),(1-INT)\*z3);% don't accept too close to limits

z1 = z1 + z2; % update the step

X = X + z2\*s;

[f2,df2] = eval(argstr);

M = M - 1; i = i + (length<0);

d2 = df2'\*s;

z3 = z3-z2; % z3 is now relative to the location of z2

end

if f2 > f1+z1\*RHO\*d1 || d2 > -SIG\*d1

break; % failure

elseif d2 > SIG\*d1

success = 1; break; % success

elseif M == 0

break; % failure

end

A = 6\*(f2-f3)/z3+3\*(d2+d3); % make cubic extrapolation

B = 3\*(f3-f2)-z3\*(d3+2\*d2);

z2 = -d2\*z3\*z3/(B+sqrt(B\*B-A\*d2\*z3\*z3)); % num. error possible - ok!

if ~isreal(z2) || isnan(z2) || isinf(z2) || z2 < 0

if limit < -0.5 % if we have no upper limit

z2 = z1 \* (EXT-1); % the extrapolate the maximum amount

else

z2 = (limit-z1)/2; % otherwise bisect

end

elseif (limit > -0.5) && (z2+z1 > limit)

z2 = (limit-z1)/2; % bisect

elseif (limit < -0.5) && (z2+z1 > z1\*EXT) % extrapolation beyond limit

z2 = z1\*(EXT-1.0); % set to extrapolation limit

elseif z2 < -z3\*INT

z2 = -z3\*INT;

elseif (limit > -0.5) && (z2 < (limit-z1)\*(1.0-INT))

z2 = (limit-z1)\*(1.0-INT);

end

f3 = f2; d3 = d2; z3 = -z2; % set point 3 equal to point 2

z1 = z1 + z2; X = X + z2\*s; % update current estimates

[f2,df2] = eval(argstr);

M = M - 1; i = i + (length<0);

d2 = df2'\*s;

end % end of line search

if success % if line search succeeded

f1 = f2; fX = [fX' f1]';

% fprintf('%4i,%4.6e;', i, f1);

% scatter(i,f1);

s = (df2'\*df2-df1'\*df2)/(df1'\*df1)\*s - df2; % Polack-Ribiere direction

tmp = df1; df1 = df2; df2 = tmp; % swap derivatives

d2 = df1'\*s;

if d2 > 0 % new slope must be negative

s = -df1; % otherwise use steepest direction

d2 = -s'\*s;

end

z1 = z1 \* min(RATIO, d1/(d2-realmin)); % slope ratio but max RATIO

d1 = d2;

ls\_failed = 0; % this line search did not fail

else

fprintf('fail');

X = X0; f1 = f0; df1 = df0; % restore point from before failed line search

if ls\_failed || i > abs(length) % line search failed twice in a row

break; % or we ran out of time, so we give up

end

tmp = df1; df1 = df2; df2 = tmp; % swap derivatives

s = -df1; % try steepest

d1 = -s'\*s;

z1 = 1/(1-d1);

ls\_failed = 1; % this line search failed

end

end

end

**ClassifierX.m**

function class = ClassifyX(input, parameters)

%%implement normalization for test data

input = bsxfun(@minus, input, parameters.mu);

input = bsxfun(@rdivide, input, parameters.sigma);

input=input\*parameters.U;

m = size(input, 1);

%num\_labels = size(parameters.Theta2, 1);

% predict with Theta

h1 = sigmoid([ones(m, 1) input] \* parameters.Theta1');

h2 = sigmoid([ones(m, 1) h1] \* parameters.Theta2');

%[dummy, p] = max(h2, [], 2);

[~, class] = max(h2, [], 2);

end

function g = sigmoid(z)

%Compute sigmoid functoon

g = 1.0 ./ (1.0 + exp(-z));

end

**Q3 CrossvalidateCheck**

% crossvalidate to choose parameters and compute accuracy and confusion matrix

clear;

assert(fopen('TrainClassifierX.m') > 0,'Could not find TrainClassifierX.m function file')

assert(fopen('ClassifyX.m') > 0,'Could not find ClassifyX.m function file')

load data

% Each datapoint is described by 3 distinct features and labelled with a

% single integer value.

n=size(data,1);

elems = randperm(n)';

%p=zeros(1,5);

%t=zeros(1,5);

for lambda=[0 0.5 0.7 1 3 5] %set different lambda

for iter=10:10:100 %set different iterations

for units=20:5:40 %set different units

for i=1:5

% set n fold cross validate set

test\_idx=elems(1:floor(n/5));

train\_idx=elems(floor(n/5)+1:n);

elems=[elems(floor(n/5)+1:n);elems(1:floor(n/5))];

train\_data=data(train\_idx,2:65);

train\_labels=data(train\_idx,1);

test\_data=data(test\_idx,2:65);

test\_labels=data(test\_idx,1);

tic;

%start compute time cost

%train data

parameters = TrainClassifierX(train\_data, train\_labels,lambda,units,iter);

%test data

predicted\_labels = ClassifyX(test\_data, parameters);

%check output

assert(max(predicted\_labels) < 6 && min(predicted\_labels) > 0, 'Classifier output label in invalid range.')

%compute accuracy

p(i)=length(find(predicted\_labels==test\_labels))/length(test\_labels);

%disp('ClassifyX has been implemented correctly.')

%disp('Sanity check passed!')

t(i)=toc;

%end compute time cost

%disp(toc);

%disp('TrainClassifierX has been implemented correctly.')

%compute confusion matrix

confu=zeros(5,5);

%initialization

for classi=1:5

for classj=1:5

%compute

confu(classi,classj)=length(find(((test\_labels==classi).\*predicted\_labels)==classj));

end

end

end

%show performance

%disp(lambda);

%disp(layers);

fprintf('%2.0f %2.0f %f %f %f\n',iter,units,lambda,mean(p),mean(t));

%disp(p);

%disp(mean(p));

%disp(mean(t));

end

end

end

return