**APPLIED DATA SCIENCE FINAL PROJECT**

**FACIAL KEYPOINTS DETECTION**

**-Log Book**

Yuezhi Wang (yw2586)   Wenkai Pan (wp2191)

Min Shi (ms4786)    Mi Xiong (mx2152)

1. Clean the data:

We use the last 1000 observations as the testing data. For each observation, there are 6 indices to compute: left\_eye\_center\_x, left\_eye\_center\_y, right\_eye\_center\_x, right\_eye\_center\_y, nose\_tip\_x, nose\_tip\_y.

1. Exploratory Analysis

1). Visualize the image and mark the feature on the face:

>im <- matrix(data=rev(im.train[1,]), nrow=96, ncol=96)

>image(1:96, 1:96, im, col=gray((0:255)/255),axes=F,xlab="",ylab="")

>points(96-d.train$nose\_tip\_x[1], 96-d.train$nose\_tip\_y[1], col="red",pch=19)

>points(96-d.train$left\_eye\_center\_x[1], 96-d.train$left\_eye\_center\_y[1], col="blue",pch=19)

>points(96-d.train$right\_eye\_center\_x[1], 96-d.train$right\_eye\_center\_y[1], col="green",pch=19)



2). Find all the noses on one face to check variation:

> im <- matrix(data=rev(im.train[100,]), nrow=96, ncol=96)

> image(1:96, 1:96, im, col=gray((0:255)/255),axes=F,xlab="",ylab="")

> for(i in 1:nrow(d.train)) {

+ points(96-d.train$nose\_tip\_x[i], 96-d.train$nose\_tip\_y[i], col="red")

+ }

> idx <- which.max(d.train$nose\_tip\_x)

> im <- matrix(data=rev(im.train[idx,]), nrow=96, ncol=96)

> image(1:96, 1:96, im, col=gray((0:255)/255),axes=F,xlab="",ylab="")

> points(96-d.train$nose\_tip\_x[idx], 96-d.train$nose\_tip\_y[idx], col="red",pch=19)

3). Compute the covariance matrix of the training data and out put the diagonal values:

> diag(cov(na.omit(y.train)))

left\_eye\_center\_x left\_eye\_center\_y right\_eye\_center\_x right\_eye\_center\_y

11.408429 9.566210 8.755643 9.009784

nose\_tip\_x nose\_tip\_y

17.508400 33.413633

1. Noise Elimination:
2. Baseline Method: Naïve Mean Method:

>pred<- colMeans(y.train, na.rm=T)

>sqrt(mean((y.test - pred)^2,na.rm=T))

[1] 19.5395

So the baseline here is 19.535. We can use this value to assess the model we built.

1. Regression Method: Elastic Regression, using the tuning parameter alpha = 0.5.

require(glmnet)

nose\_x<- y.train$nose\_tip\_x

lasso.fit.nosex<-glmnet(train, nose\_x, alpha=0.5)

pre\_nosex<-predict(lasso.fit.nosex,newx = test)

nose\_y<- y.train$nose\_tip\_y

lasso.fit.nosey<-glmnet(train, nose\_y, alpha=0.5)

pre\_nosey<-predict(lasso.fit.nosey,newx = test)

left\_eye\_x<- y.train[,1]

lasso.fit.left\_eye\_x<-glmnet(train[-which(is.na(left\_eye\_x)),],left\_eye\_x[-which(is.na(left\_eye\_x))], alpha=0.5)

pre\_leftx<-predict(lasso.fit.left\_eye\_x,newx = test)

left\_eye\_y<- y.train[,2]

lasso.fit.left\_eye\_y<-glmnet(train[-which(is.na(left\_eye\_y)),], left\_eye\_y[-which(is.na(left\_eye\_y))], alpha=0.5)

pre\_lefty<-predict(lasso.fit.left\_eye\_y,newx = test)

right\_eye\_x<- y.train[,3]

lasso.fit.right\_eye\_x<-glmnet(train[-which(is.na(right\_eye\_x)),],right\_eye\_x[-which(is.na(right\_eye\_x))], alpha=0.5)

pre\_rightx<-predict(lasso.fit.right\_eye\_x,newx = test)

right\_eye\_y<- y.train[,4]

lasso.fit.right\_eye\_y<-glmnet(train[-which(is.na(right\_eye\_y)),], right\_eye\_y[-which(is.na(right\_eye\_y))], alpha=0.5)

pre\_righty<-predict(lasso.fit.right\_eye\_y,newx = test)

> pred.reg<- cbind(pre\_leftx, pre\_lefty,pre\_rightx, pre\_righty, pre\_nosex,pre\_nosey)

> sqrt(mean((y.test - pred.reg)^2, na.rm=T))

[1] 23.3546

**Comment:** The test error of Elastic Regression is very poor even compared to the simple Naïve Mean Method. This is Plausible since the location of certain feature has little thing to do with the pixel value. And the background information can be quite disturbing in using regression method.

1. Decision Trees

lm.data<-data.frame(y = y.train[,1],train[,])

tree1 <- tree(y~.,lm.data)

yhat1 <- predict (tree1 ,newdata = data.frame(test))

lm.data<-data.frame(y = y.train[,2],train[,])

tree2 <- tree(y~.,lm.data)

yhat2 <- predict (tree2 ,newdata = data.frame(test))

lm.data<-data.frame(y = y.train[,3],train[,])

tree3 <- tree(y~.,lm.data)

yhat3 <- predict (tree3 ,newdata = data.frame(test))

lm.data<-data.frame(y = y.train[,4],train[,])

tree4 <- tree(y~.,lm.data)

yhat4 <- predict (tree4 ,newdata = data.frame(test))

lm.data<-data.frame(y = y.train[,5],train[,])

tree5 <- tree(y~.,lm.data)

yhat5 <- predict (tree5 ,newdata = data.frame(test))

lm.data<-data.frame(y = y.train[,6],train[,])

tree6 <- tree(y~.,lm.data)

yhat6 <- predict (tree6 ,newdata = data.frame(test))

pred.tree<- cbind(yhat1,yhat2,yhat3,yhat4,yhat5,yhat6)

>sqrt(mean((y.test - pred.tree)^2, na.rm=T))

[1] 3.98568

It turns out that the result of decision tree is quite well compared to the baseline method, let alone the regressions. This makes sense since tree method is kind of pattern searching approach, if the image satisfies certain character, it would be chosen. The computer vision method we learned later is also based on trees.

1. Principle Component Analysis based on original data:

First we compressed the data from 96\*96 to 24\*24. This is to reduce the dimension and we will see the performance of this approach to decide whether it is applicable.

#This file is to reduce 96\*96 to 24\*24

#"data.Rd"is the clean version of

load("data.Rd")

my.melt<-function(x){

y.mat<-matrix(0,24,24)

x.mat<-matrix(x,96,96)

for (i in 1:24){

for (j in 1:24){

y.mat[i,j]=mean(x.mat[(4\*i-3):(4\*i),(4\*j-3):(4\*j)])}}

y.vec1<-as.vector(y.mat[,24:1])

return(y.vec1)

}

red.im.train<-t(apply(im.train,1,FUN=my.melt))

red.im.test<-t(apply(im.test,1,FUN=my.melt))

save(red.im.train, red.im.test, file='reducedata.Rd')

The code for implement Principle Component Analysis:

load('reducedata.Rd')

dim(red.im.train)

red.pic<-rbind(red.im.train, red.im.test)

#PCA and KNN(1-NN)

red.pic<-red.pic-colMeans(red.pic)

pca.res<-princomp(red.pic)

pca.loading<-loadings(pca.res)

#first 30 eigenfaces

score30.pic<-pca.res$scores[,1:30]

score30.train<-score30.pic[1:7049,]

score30.test<-score30.pic[7050:8832,]

#Distant matrix

dis.all<-as.matrix(dist(score30.pic))

dim(dis.all)

traintestdis<-dis.all[1:7049,7050:8832]

myind.vec<-apply(traintestdis,2,which.min)

#Here myind.vec represent which index in training is the nearest for test.

#try to see myind.vec[1:5]

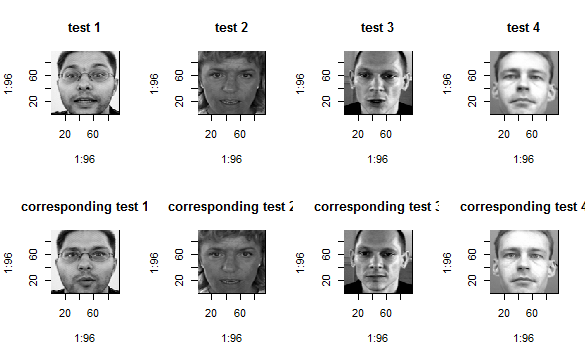
im<-matrix(rev(im.train[155,]),96,96)

image(1:96, 1:96, im, col=gray((0:255)/255))

im<-matrix(rev(im.test[1,]),96,96)

image(1:96, 1:96, im, col=gray((0:255)/255))

We can see that PCA did a great job in finding out the similar images. The following codes are to compute the test error rate, which is 5.654 in total. This is a very satisfying result. For further comment please see the final project report.

****

#Prediction Error

dis.all<-as.matrix(dist(score30.pic))

save(dis.all,file="disall.Rd")

dis.all1<-dis.all[1:7049,1:7049]

samp<-sample(1:7049)[1:6049]

traintestdis<-dis.all1[samp,-samp]

myind.vec<-apply(traintestdis,2,which.min)

mean.vec=rep(0,8)

j=1

for (i in c(1,2,3,4,21,22,29,30)){

mean.vec[j]=sqrt(mean(na.omit(d.train[myind.vec,i]-d.train[-samp,i])^2))

j=j+1

}

mean(mean.vec)

mean.vec=rep(0,8)

j=1

for (i in c(1,2,3,4,21,22,29,30)){

mean.vec[j]=sqrt(mean(na.omit(d.train[19,i]-d.train[8,i])^2))

j=j+1

}

mean(mean.vec)

1. Pattern Matching

In this part, instead of use the whole image to do principle component analysis, we use features. For “eigen-features”, we use correlation to measure the distance with test image to find the best match. Then the pattern with highest probability is assigned to be the feature. This part is implemented in MATLAB.

The result of this methods tend to be very well. Please see details in the project report.

Since the code is too long to include, here we just present some key part. This part of code is to use eigen features to search the matching part in test image:

function boxes = detect(input, model, thresh)

% Keep track of detected boxes and features

BOXCACHESIZE = 100000;

cnt = 0;

boxes.s = 0;

boxes.c = 0;

boxes.xy = 0;

boxes.level = 0;

boxes(BOXCACHESIZE) = boxes;

% Compute the feature pyramid and prepare filters

pyra = featpyramid(input,model);

[components,filters,resp] = modelcomponents(model,pyra);

for c = randperm(length(components)),

minlevel = model.interval+1;

levels = minlevel:length(pyra.feat);

for rlevel = levels(randperm(length(levels))),

parts = components{c};

numparts = length(parts);

% Local part scores

for k = 1:numparts,

f = parts(k).filterid;

level = rlevel-parts(k).scale\*model.interval;

if isempty(resp{level}),

resp{level} = fconv(pyra.feat{level},filters,1,length(filters));

end

parts(k).score = resp{level}{f};

parts(k).level = level;

end

% Walk from leaves to root of tree, passing message to parent

% Given a 2D array of filter scores 'child', shiftdt() does the following:

% (1) Apply distance transform

% (2) Shift by anchor position (child.startxy) of part wrt parent

% (3) Downsample by child.step

for k = numparts:-1:2,

child = parts(k);

par = child.parent;

[Ny,Nx,foo] = size(parts(par).score);

[msg,parts(k).Ix,parts(k).Iy] = shiftdt(child.score, child.w(1),child.w(2),child.w(3),child.w(4), ...

child.startx, child.starty, Nx, Ny, child.step);

parts(par).score = parts(par).score + msg;

end

% Add bias to root score

rscore = parts(1).score + parts(1).w;

[Y,X] = find(rscore >= thresh);

if ~isempty(X)

XY = backtrack( X, Y, parts, pyra);

end

% Walk back down tree following pointers

for i = 1:length(X)

x = X(i);

y = Y(i);

if cnt == BOXCACHESIZE

b0 = nms\_face(boxes,0.3);

clear boxes;

boxes.s = 0;

boxes.c = 0;

boxes.xy = 0;

boxes.level = 0;

boxes(BOXCACHESIZE) = boxes;

cnt = length(b0);

boxes(1:cnt) = b0;

end

cnt = cnt + 1;

boxes(cnt).c = c;

boxes(cnt).s = rscore(y,x);

boxes(cnt).level = rlevel;

boxes(cnt).xy = XY(:,:,i);

end

end

end

boxes = boxes(1:cnt);

% Backtrack through dynamic programming messages to estimate part locations

% and the associated feature vector

function box = backtrack(x,y,parts,pyra)

numparts = length(parts);

ptr = zeros(numparts,2,length(x));

box = zeros(numparts,4,length(x));

k = 1;

p = parts(k);

ptr(k,1,:) = x;

ptr(k,2,:) = y;

% image coordinates of root

scale = pyra.scale(p.level);

padx = pyra.padx;

pady = pyra.pady;

box(k,1,:) = (x-1-padx)\*scale + 1;

box(k,2,:) = (y-1-pady)\*scale + 1;

box(k,3,:) = box(k,1,:) + p.sizx\*scale - 1;

box(k,4,:) = box(k,2,:) + p.sizy\*scale - 1;

for k = 2:numparts,

p = parts(k);

par = p.parent;

x = ptr(par,1,:);

y = ptr(par,2,:);

inds = sub2ind(size(p.Ix), y, x);

ptr(k,1,:) = p.Ix(inds);

ptr(k,2,:) = p.Iy(inds);

% image coordinates of part k

scale = pyra.scale(p.level);

box(k,1,:) = (ptr(k,1,:)-1-padx)\*scale + 1;

box(k,2,:) = (ptr(k,2,:)-1-pady)\*scale + 1;

box(k,3,:) = box(k,1,:) + p.sizx\*scale - 1;

box(k,4,:) = box(k,2,:) + p.sizy\*scale - 1;

end

% Cache various statistics from the model data structure for later use

function [components,filters,resp] = modelcomponents(model,pyra)

components = cell(length(model.components),1);

for c = 1:length(model.components),

for k = 1:length(model.components{c}),

p = model.components{c}(k);

x = model.filters(p.filterid);

[p.sizy p.sizx foo] = size(x.w);

p.filterI = x.i;

x = model.defs(p.defid);

p.defI = x.i;

p.w = x.w;

% store the scale of each part relative to the component root

par = p.parent;

assert(par < k);

ax = x.anchor(1);

ay = x.anchor(2);

ds = x.anchor(3);

if par > 0,

p.scale = ds + components{c}(par).scale;

else

assert(k == 1);

p.scale = 0;

end

% amount of (virtual) padding to hallucinate

step = 2^ds;

virtpady = (step-1)\*pyra.pady;

virtpadx = (step-1)\*pyra.padx;

% starting points (simulates additional padding at finer scales)

p.starty = ay-virtpady;

p.startx = ax-virtpadx;

p.step = step;

p.level = 0;

p.score = 0;

p.Ix = 0;

p.Iy = 0;

components{c}(k) = p;

end

end

resp = cell(length(pyra.feat),1);

filters = cell(length(model.filters),1);

for i = 1:length(filters),

filters{i} = model.filters(i).w;

end

1. Computer Science Vision