# APPLIED DATA SCIENCE FINAL PROJECT FACIAL KEYPOINTS DETECTION

## -Log Book

Yuezhi Wang (yw2586)    Wenkai Pan (wp2191)
Min Shi (ms4786)    Mi Xiong (mx2152)

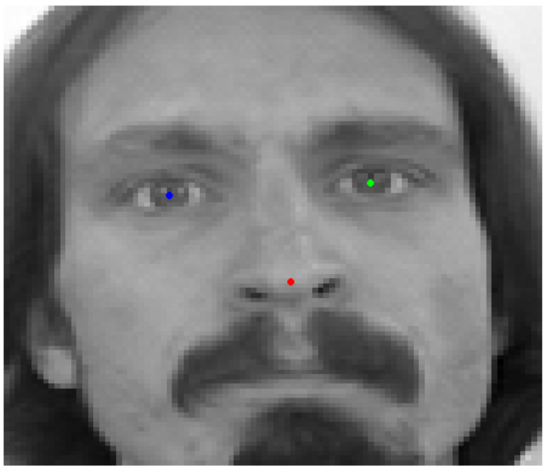
## 1. Clean the data:

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.

## 2. Exploratory Analysis

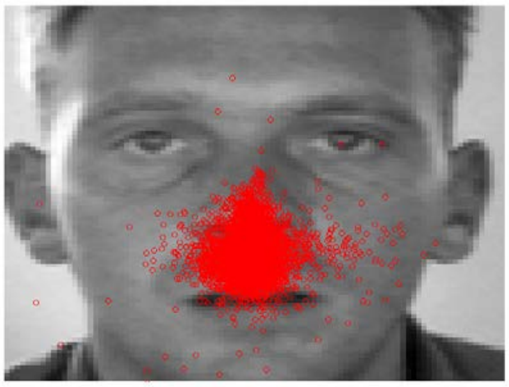
1). Visualize the image and mark the feature on the face:

```
>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
```

**3. 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.

**4. 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.

5. **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
```

**Comment:** 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.

6. **Feature Box Method:**

This method uses feature box to do pattern searching. Instead of utilizing all information of a picture, we use small box to extract out matching area in picture for one specific key point. The result is good and inspires us for the next steps. See details in the report.

The code of Feature Box Method for nose:

```
library(reshape2)
library(doParallel)
registerDoParallel()
install.packages('foreach')
require(foreach)
load('data.Rd')
coord <- "nose_tip"
patch_size <- 10
coord_x <- paste(coord, "x", sep="_")
coord_y <- paste(coord, "y", sep="_")
patches <- foreach (i = 1:nrow(d.train), .combine=rbind) %do% {
  im    <- matrix(data = im.train[i,], nrow=96, ncol=96)
  x     <- d.train[i, coord_x]
  y     <- d.train[i, coord_y]
  x1    <- (x-patch_size)
  x2    <- (x+patch_size)
  y1    <- (y-patch_size)
  y2    <- (y+patch_size)
  if ( (!is.na(x)) && (!is.na(y)) && (x1>=1) && (x2<=96) && (y1>=1) && (y2<=96) )
  {
    as.vector(im[x1:x2, y1:y2])
  }
  else
  {
    NULL
  }
}
mean.patch <- matrix(data = colMeans(patches), nrow=2*patch_size+1, ncol=2*patch_size+1)
par(mfrow=c(1,4))
image(1:21, 1:21, mean.patch[21:1,21:1], col=gray((0:255)/255),main="Mean Nose_tip")
head(d.train)
i=12
cord.xy<-as.numeric(as.vector(round(d.train[i,21:22],0)))
im<-matrix(rev(as.numeric(im.train[i,])),96,96)
image(1:21, 1:21, im[(cord.xy[1]-10):(cord.xy[1]+10),(cord.xy[2]-10):(cord.xy[2]+10)],
col=gray((0:255)/255),main="Sample Nose Box")
```

## 7. 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.

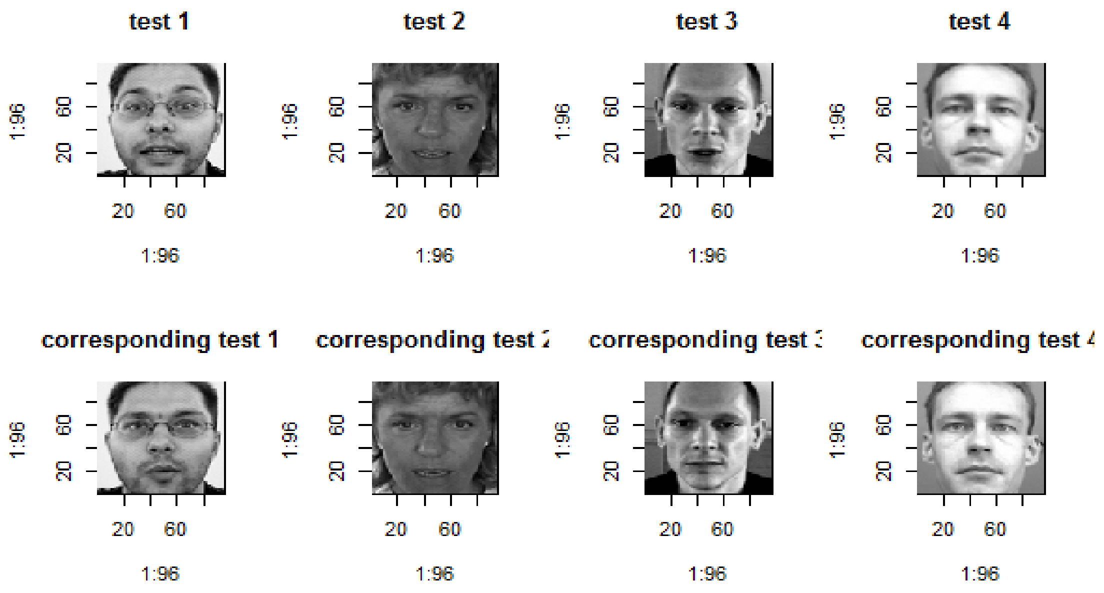
```
#This file is to reduce 96*96 to 24*24
#"data.Rd"is the clean version of
load("data.Rd")
```

```
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)
```

## 8. Combine PAC and Feature Box:

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
```

9. Computer Science Vision - **Tree structured part model:**

Write each tree $T_m = (V_m, E_m)$as linearly parameterized, tree-structured pictorial structure, where m indicates a mixture and $V_m \in V$($V$: shared pool of parts). Write I for an image, and $l_i = (x_i, y_i)$ for the pixel location of part I. Score a configuration of parts $\mathrm{L} = \{l_i : i \in V\}$ as:

$$S(I, L, m) = App_m(I, L) + Shape_m(L) + \alpha^m \qquad (1)$$

$$App_m(I, L) = \sum_{i \in V_m} \omega_i^m \cdot \phi(I, l_i) \qquad (2)$$

$$Shape_m(L) = \sum_{ij \in E_m} a_{ij}^m dx^2 + b_{ij}^m dx + c_{ij}^m dy^2 + d_{ij}^m dy \quad (3)$$

Through the visualization results we can see the model is surprisingly effective in capturing facial keypoints. Concerning of performance, the test errors of the model are left-eye-center-error: 3.5253; right-eye-center-error: 3.3748; nose-tip-error 4.4337, we conclude the new added trees are performing well.

The code of this part is too long to include.