SVD and EigenFaces

Matt Richey

04/25/2019

Setup and Load Libraries

Load libraries

```
suppressMessages(library(tidyverse))
suppressMessages(library(factoextra))
suppressMessages(library(ggrepel))
```

Introduction

Let's implement the process of creating Eigenspaces associated with classes of images. We will then use these Eigenspaces as a means of performing some basic image classification.

Data

Our data consists of images of (very cute) cats and dogs.

```
dataDir <- "/Users/richeym/Dropbox/COURSES/ADM/ADM_S20/CODE/Chap10"
catsAll <- read.csv(file.path(dataDir,"cats.csv"),header=F)
dogsAll <- read.csv(file.path(dataDir,"dogs.csv"),header=F)
```

Each dataset of cats and dogs contain 99 images of, no surprise, cats and dogs. Each image is 64x64 image stacked into a row 4096=64*64 greyscale values. Note that there is one row for each of the 4096 pixels and one column for each image.

```
dim(catsAll)
```

```
## [1] 4096   99
```

```
dim(dogsAll)
```

```
## [1] 4096   99
```

Subset this into training and testing sets. We will build our eigenspaces on the training data and then see if we can use these to identify a testing image as either a cat or a dog.

```
cats <- catsAll[,1:90]
catsOther <- catsAll[,91:99]
dogs <- dogsAll[,1:90]
dogsOther <- dogsAll[,91:99]
```

Extract the dimensions

```
(n <- nrow(cats))
```

```
## [1] 4096
```

```
(p <- ncol(cats))
```

```
## [1] 90
```

Note that n= number of observations, an observation is a pixel value. Also, p = number of features, in this case, its a the pixal value across all the cats!

Helper functions

In order to work with these images, we need to be able to view them. To do so, let's create a couple helper functions that will facilitate going back and forth between flat and square images.

The first function converts an image into 64x64 matrix.

```
imageConv <- function(flatImage,size=64){
    matrix(flatImage,nrow=size,byrow=T)
}
```

The second function reverses the process.

```
flatConv <- function(rectImage,size=64){
    matrix(t(rectImage),nrow=size^2,ncol=1)[,1]
}
```

This makes it easy to plot our images with a minimum of fuss and muss.

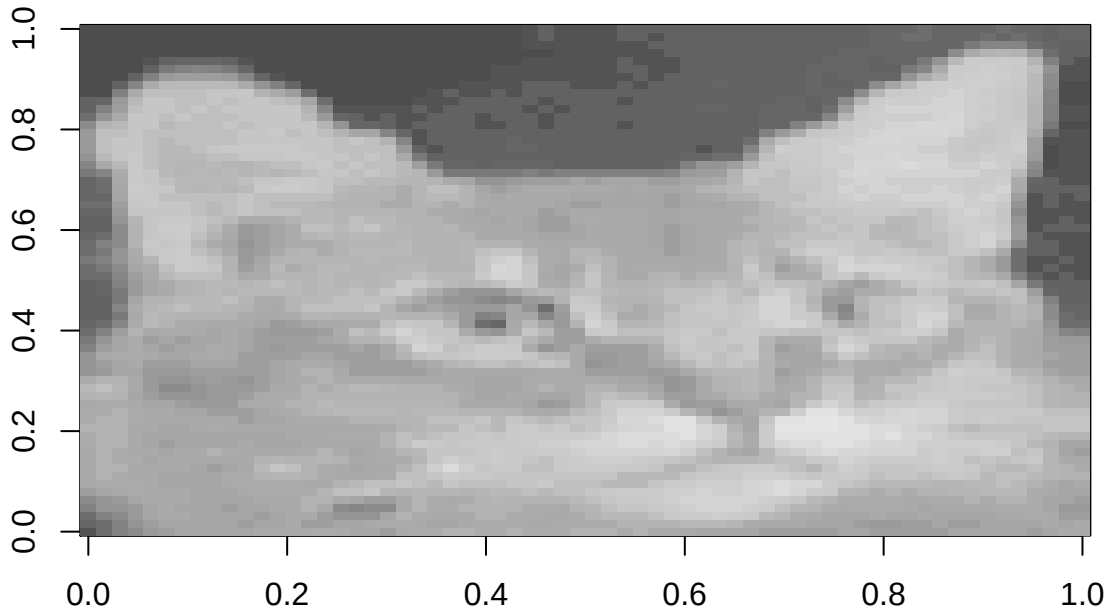
```
flatImage <- function(dat) {
  img <- imageConv(dat)
  image(img,col=grey.colors(256))
}
```

Here's looking at some images

Here's how to plot an image. It's pretty easy.

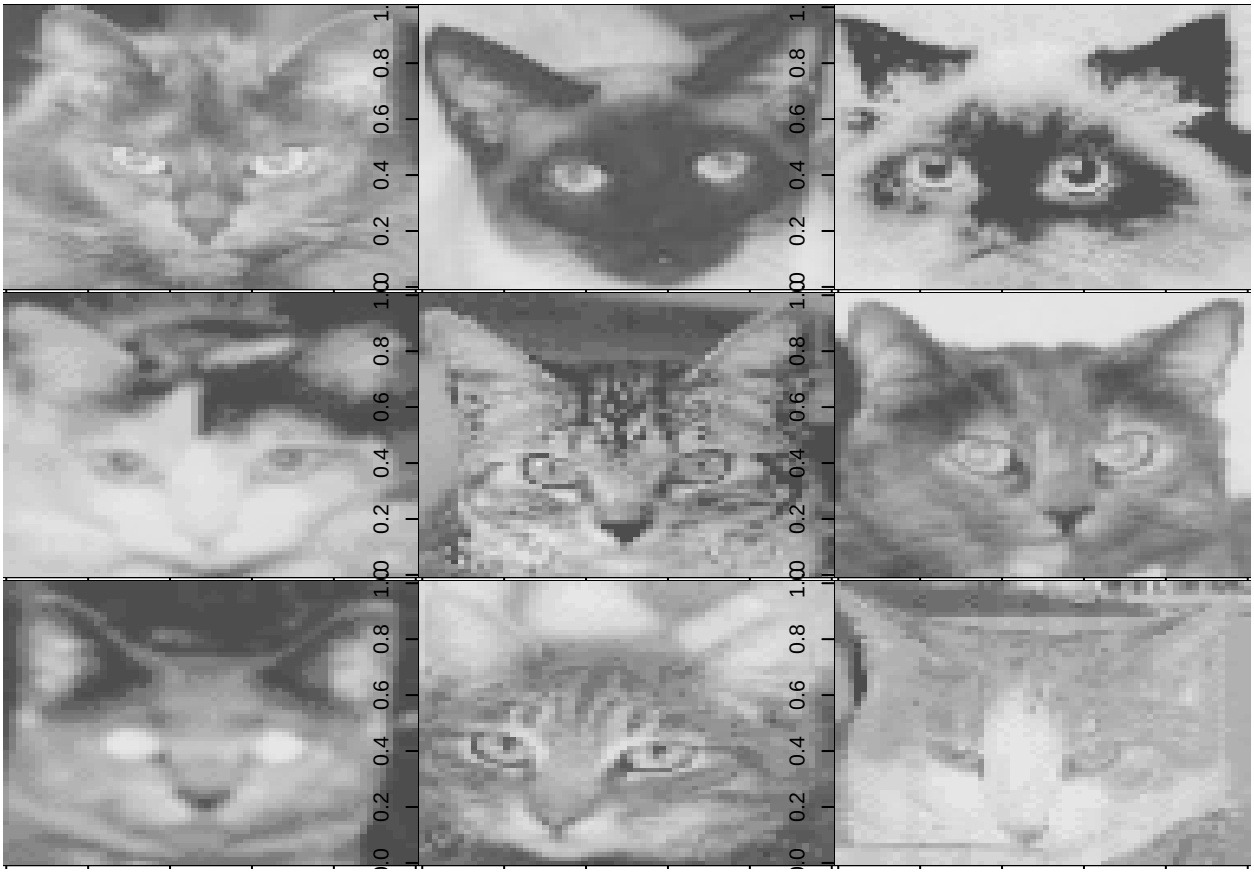
Grab a cat and plot it.

```
flatCat <- cats[,1]
flatImage(flatCat)
```



Look at 9 randomly selected cats.

```
samp <- sample(1:p,9,rep=F)
catSamp <- cats[,samp]
op <- par(mfrow=c(3,3),mai=c(0.02,0.,0,0))
for(i in 1:9){
   flatImage(catSamp[,i])
}
```

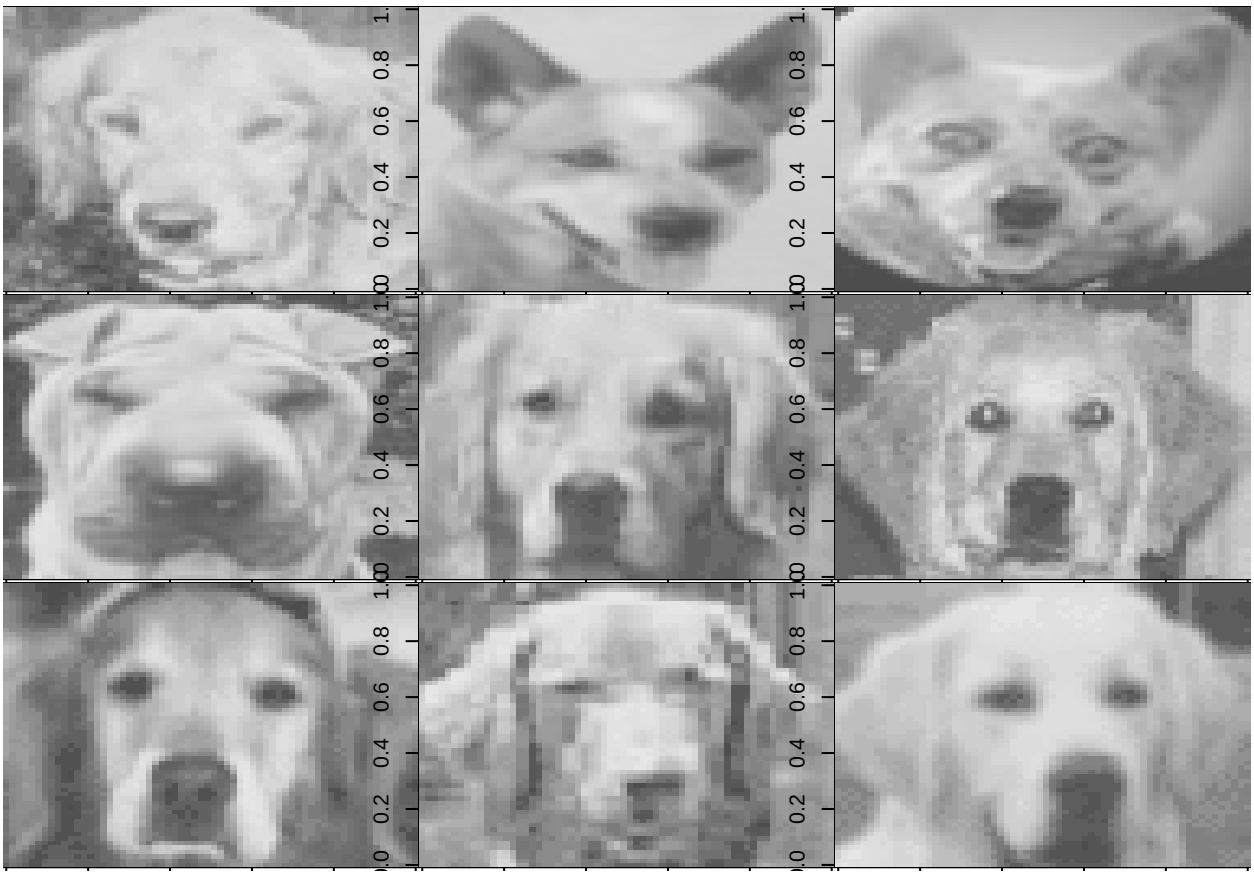


```
par(op)
```

Those are some cute cats.

Do the same for dogs. They couldn't possibly be as cute. Look at 9 randomly selected cats

```
samp <- sample(1:p,9,rep=F)
dogSamp <- dogs[,samp]
op <- par(mfrow=c(3,3),mai=c(0.02,0.,0,0))
for(i in 1:9){
    flatImage(dogSamp[,i])
}
```



```
par(op)
```

Maybe I was wrong. Pretty cute too.

Singular Value Decomposition

Now we can begin the computation in the image space. Remember the plan.

- Perform the SVD decomposion of the full data set.
- Identify the eigenvectors that span the column space (subset of image space).

- Use the eigenvectors to define the projection mapping from the full image space onto the column space.
- The closer an image is to a particular eigenspace, the more it is like the common images.

To make the SVD work, we need to center the data.

```
cats0 <- scale(cats,scale=F)
```

Check.. the new cats should have column means which are zero.

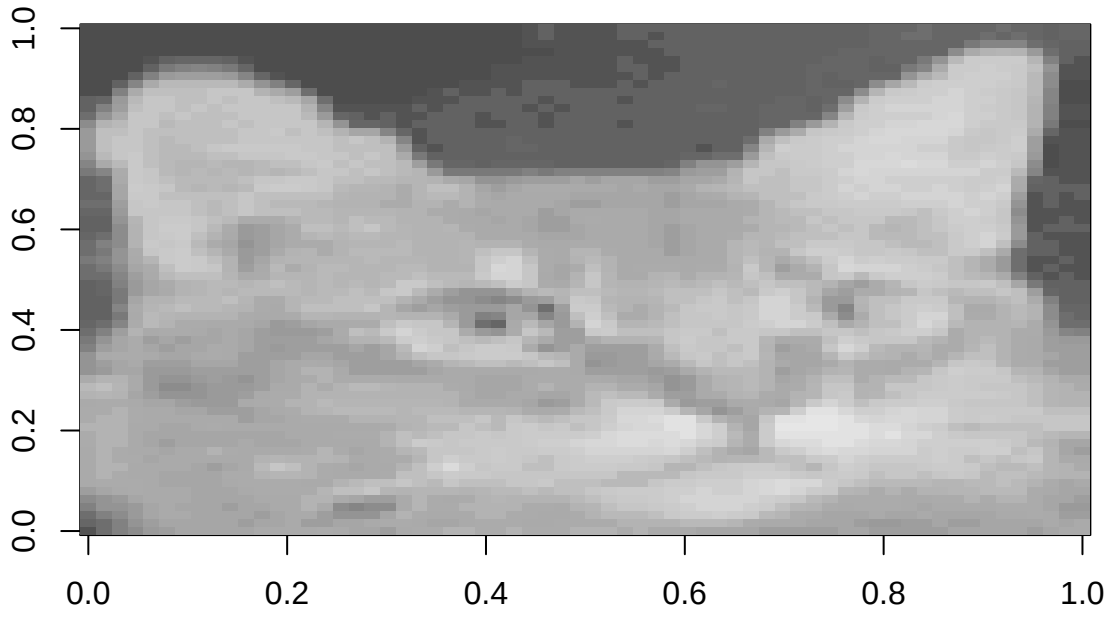
```
colMeans(cats0)
```

```
##  V1  V2  V3  V4  V5  V6  V7  V8  V9 V10 V11 V12 V13 V14 V15 V16 V17 V18 V19 V20
##   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
## V21 V22 V23 V24 V25 V26 V27 V28 V29 V30 V31 V32 V33 V34 V35 V36 V37 V38 V39 V40
##   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
## V41 V42 V43 V44 V45 V46 V47 V48 V49 V50 V51 V52 V53 V54 V55 V56 V57 V58 V59 V60
##   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
## V61 V62 V63 V64 V65 V66 V67 V68 V69 V70 V71 V72 V73 V74 V75 V76 V77 V78 V79 V80
##   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
## V81 V82 V83 V84 V85 V86 V87 V88 V89 V90
##   0   0   0   0   0   0   0   0   0   0
```

Check...

And they should plot ok...

```
flatImage(cats0[,1])
```



Check...

Now the Singular Value Decomposition of the cat images

```
cat.svd <- svd(cats0)
```

Here are the u, d, and v matrices

```
U <- cat.svd$u
D <- cat.svd$d
V <- cat.svd$v
```

Check the dimensions

```
dim(U)
```

```
## [1] 4096   90
```

```
length(D)
```

```
## [1] 90
```

```
dim(V)
```

```
## [1] 90 90
```

Looking good.

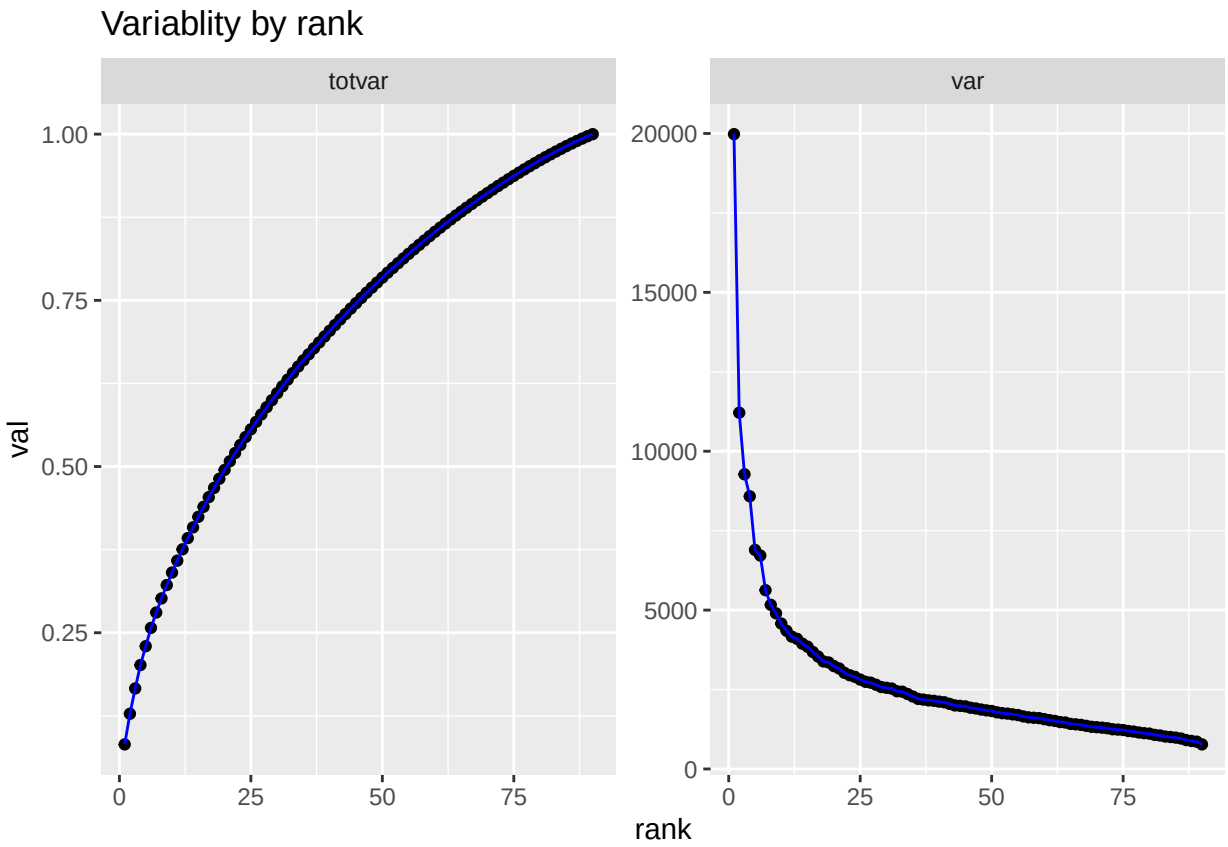
As well, you could check that U and V are orthonormal.

The diagonal values of D measurethe variabililty contained in each component Pack into data frame. Include both the variability in each component and the cumulative variability.

```
proj.df <- data.frame(rank=1:p,var=D,totvar=cumsum(D)/sum(D))
```

What do we have

```
proj.df %>%
    gather(type,val,var:totvar) %>%
    ggplot()+
    geom_point(aes(rank,val))+
        geom_line(aes(rank,val),color="blue")+
    facet_wrap(~type,scales="free")+
    labs(title="Variablity by rank")
```



This is a fairly typical situation. The first 10-20 components account for about half of the total variability.

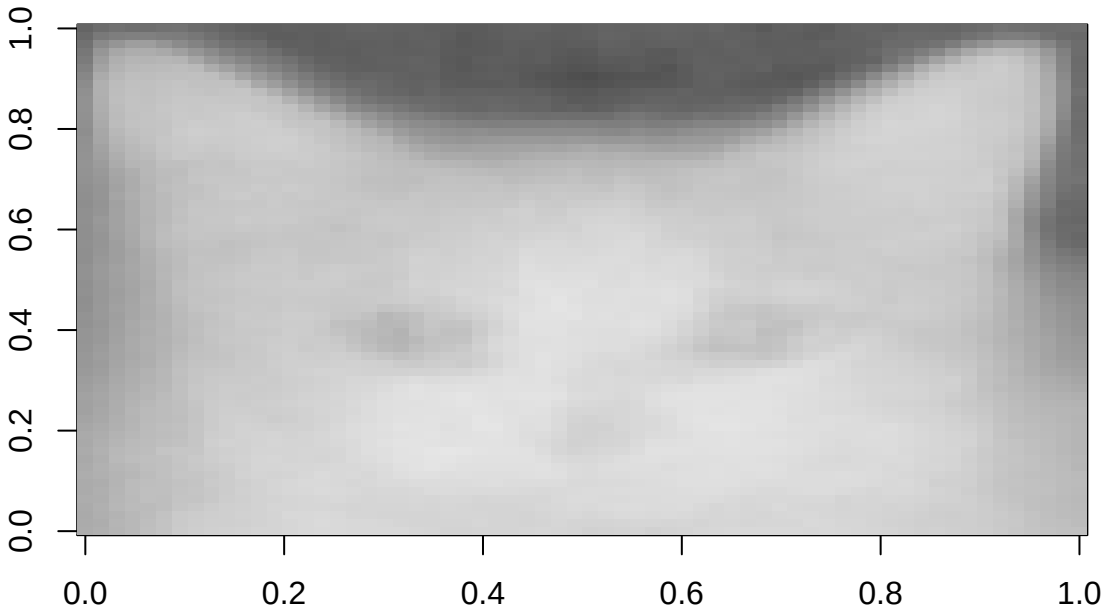
Eigenface reconstructions

The columns of the U matrix are the eigenfaces. The first eigenface is the one with the largest eigenvalue. In some ways, it is capturing the most "cat-ness" of all the eigenfaces

```
eigenFace1=U[,1]
```

What does it look like?

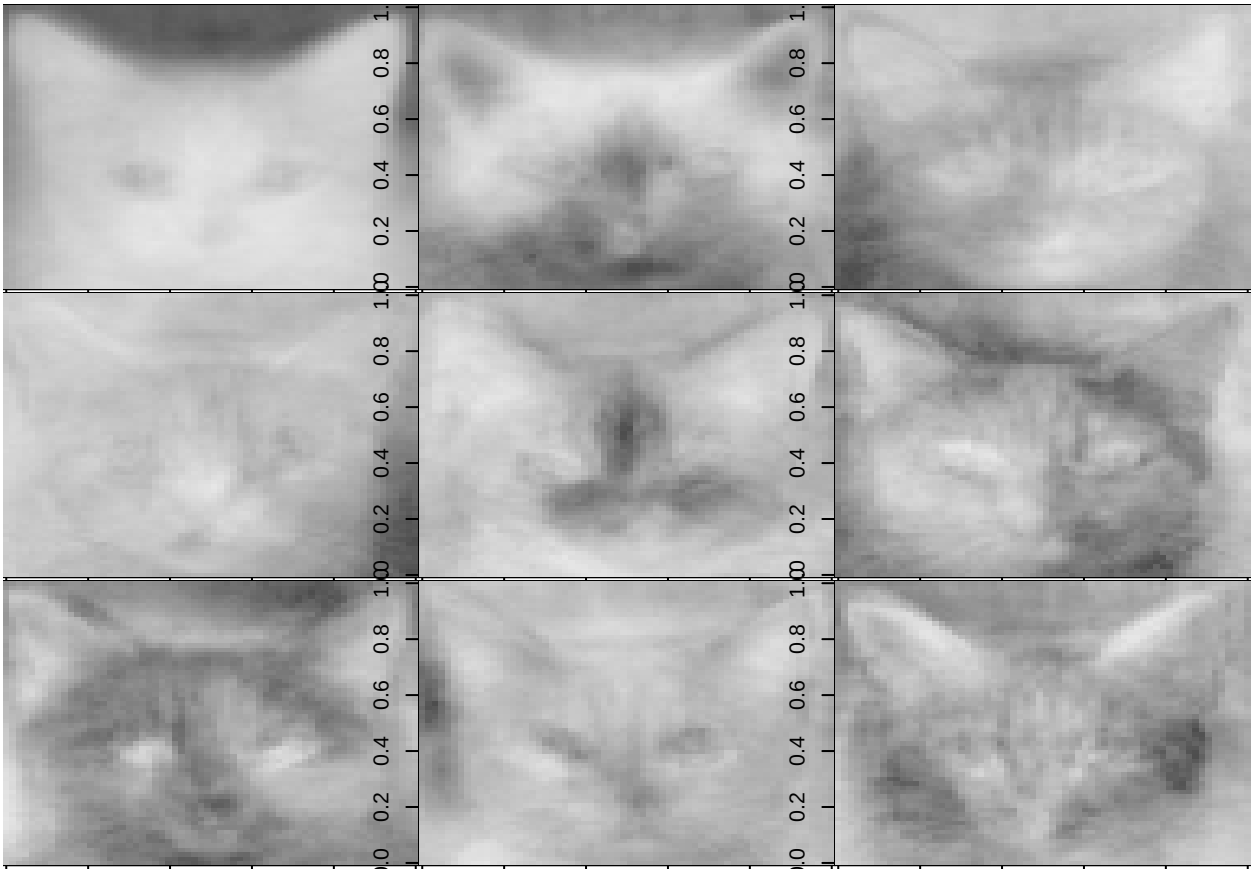
```
flatImage(eigenFace1)
```



Very cat-line. You can think of this as a first order approximation of the average of all the cats in the data set.

Now let's look at first 9 eigenfaces.

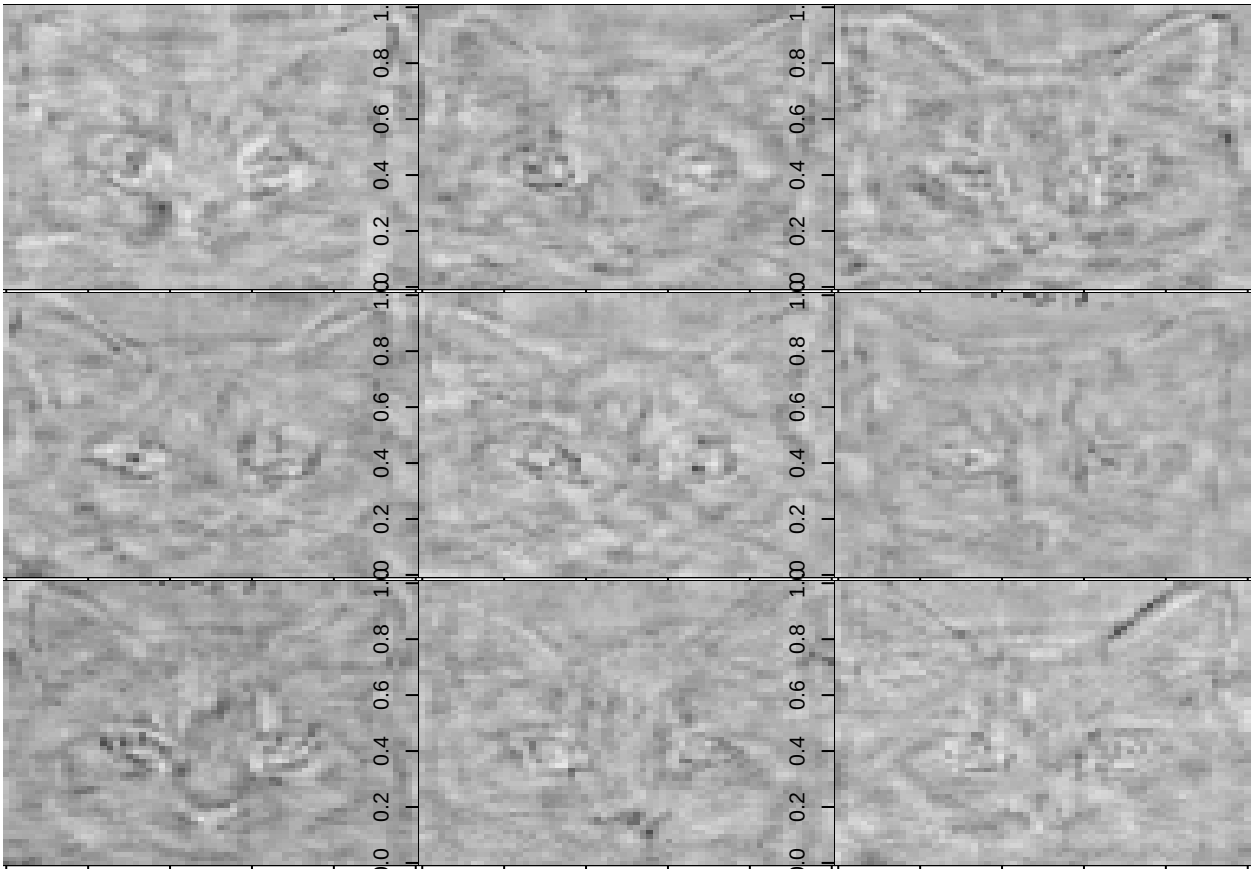
```
samp <- 1:9
catSamp <- U[,samp]
op <- par(mfrow=c(3,3),mai=c(0.02,0.,0,0))
for(i in 1:9){
   flatImage(catSamp[,i])
}
```



```
par(op)
```

How about the last 9 eigenfaces?

```
samp <- 81:90
catSamp <- U[,samp]
op <- par(mfrow=c(3,3),mai=c(0.02,0.,0,0))
for(i in 1:9){
     flatImage(catSamp[,i])
}
```



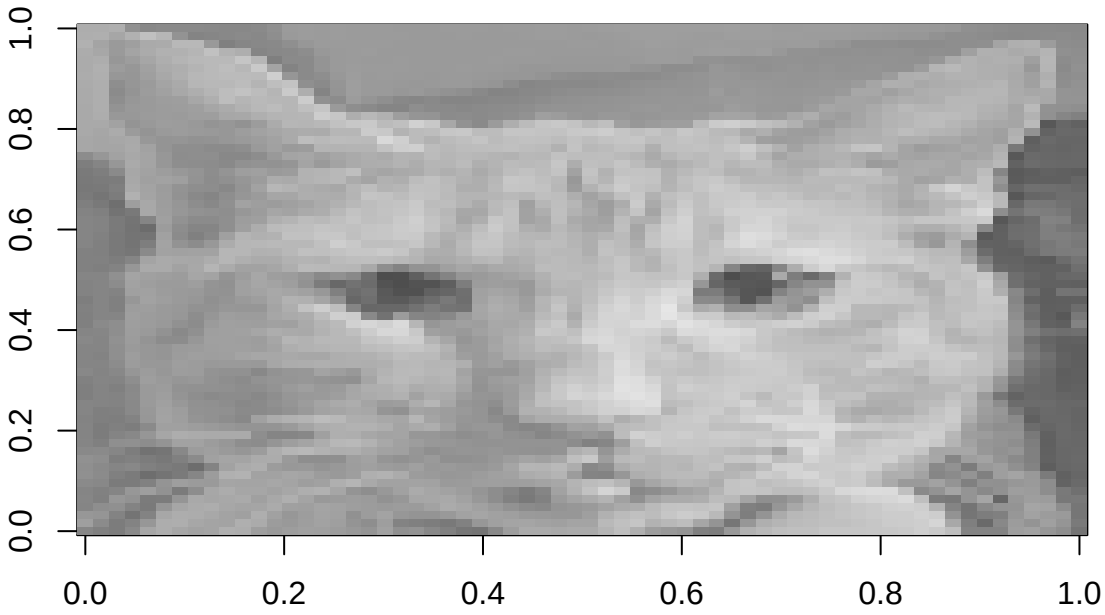
```
par(op)
```

At this point, we still have cats, but we've lost a great deal of the distinguishing features of "cat-ness"

Projections onto cat space: Eigenface reconstuctions

Grab a cat from the test space.

```
aCat <- catsOther[,1]
flatImage(aCat)
```

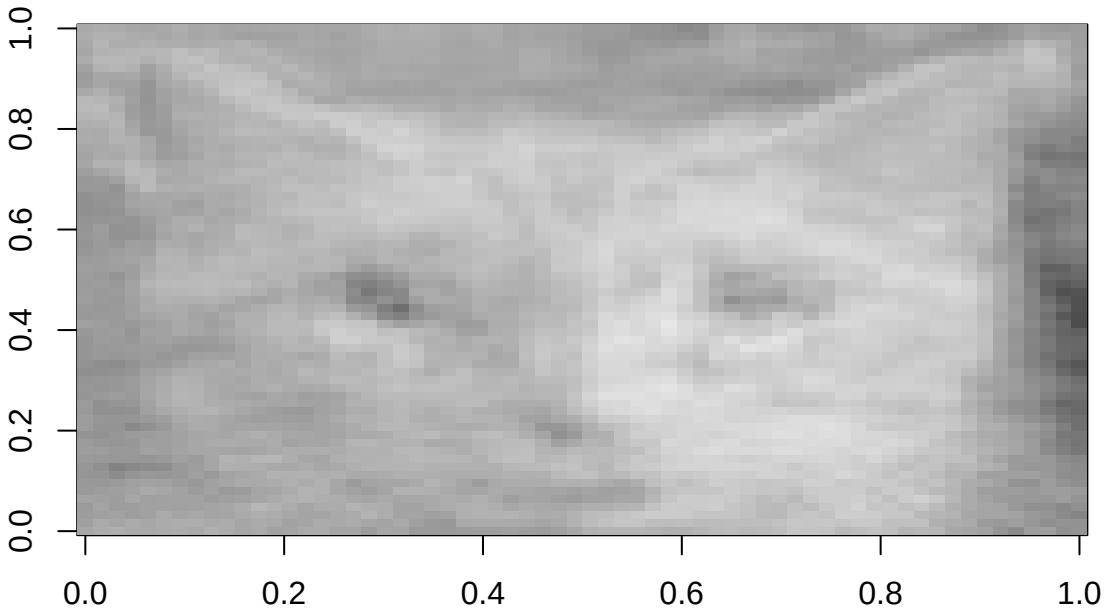


What does it look like after projecting onto the cat space? To get it there we compute it's projection using the "hat matrix" $U\,U^t$.

```
catHat <- U %*% t(U)
aCatHat <- catHat %*%aCat
```

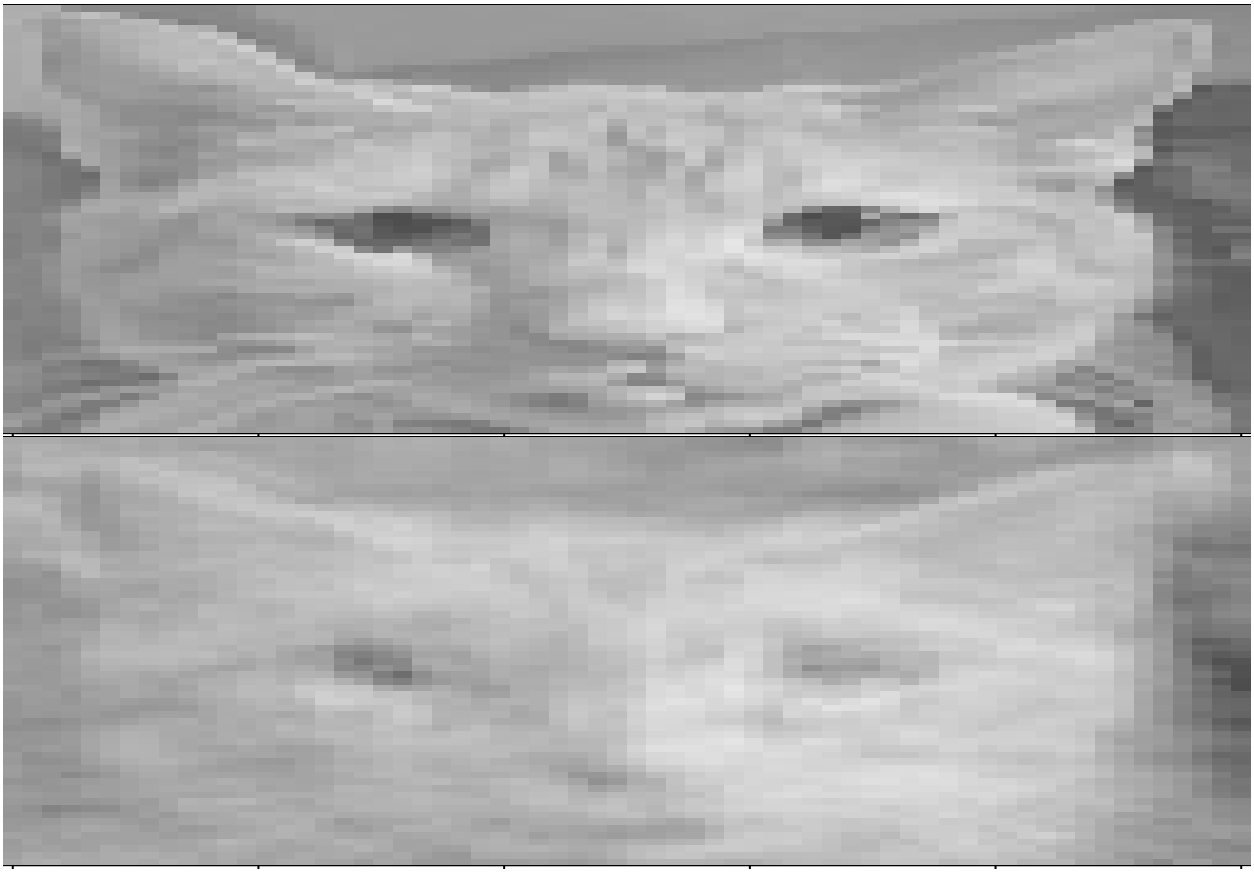
What do we have

```
flatImage(aCatHat)
```



Compare together

```
op <- par(mfrow=c(2,1),mai=c(0.02,0.,0,0))
flatImage(aCat)
flatImage(aCatHat)
```



```
par(op)
```

The comparison reveals both similarities and differences. Clearly, a fair amount of the unique large-scale features of aCat are preserved after projecting. At the same time, a lot of finer features are missing. This is because we trying to use a relatively small number of cats to capture the essence of "cat-ness"!

Facial Recognition Challange

Here's a game you can play. Take some cats from the test set, project them onto the eigenspace, and see if we can pick out the correct pairings

The number of images

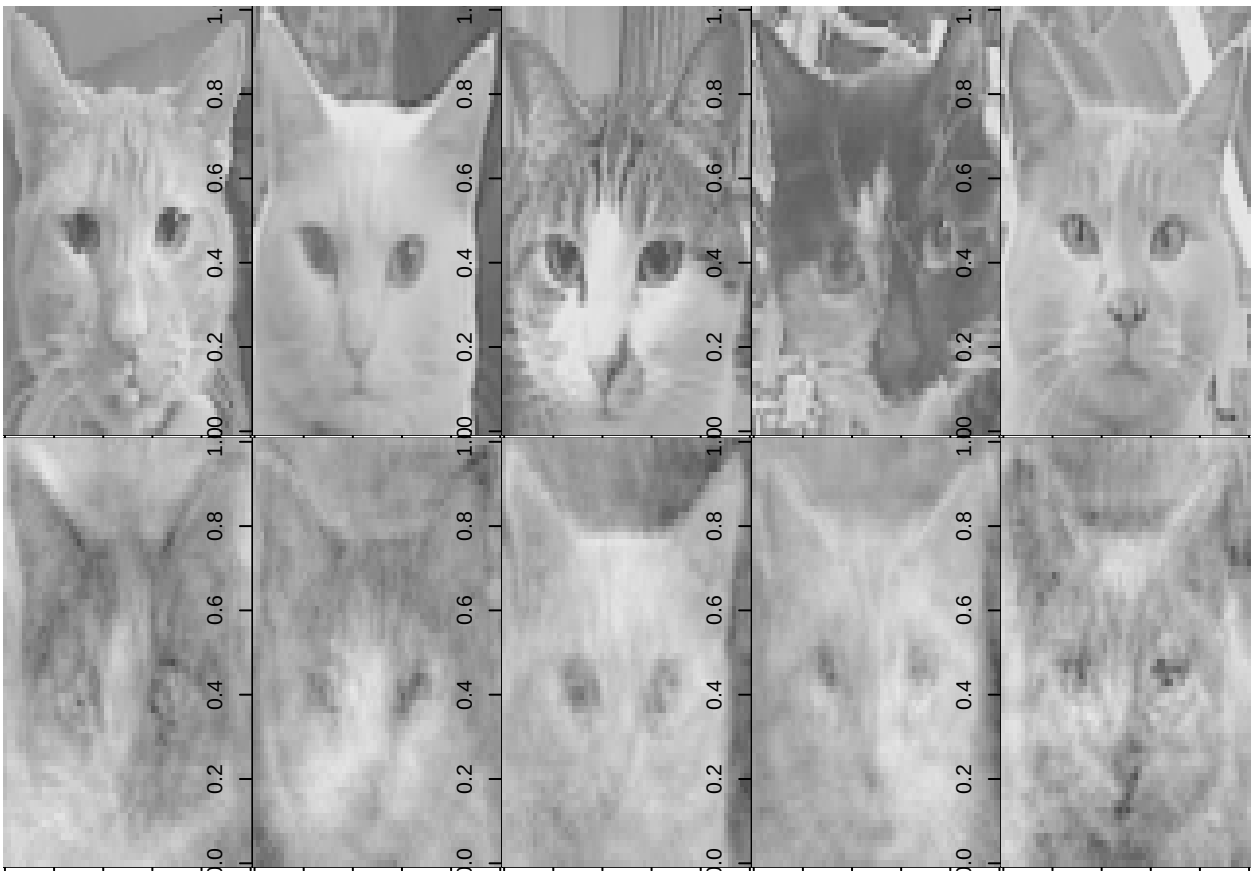
```
num <- 5
```

Pick num cats at random and build eigenFace reconstruction

```
samp <- sample(1:9,num,rep=F)
catHats <- matrix(nrow=4096,ncol=num)
for(i in 1:num){
    catHats[,i] <- catHat %*% catsOther[,i]
}
```

Set up the matches. . . ..

```
## Scramble the order
newOrd <- sample(1:num,num,rep=F)
op <- par(mfrow=c(2,num),mai=c(0.02,0.,0,0))
for(i in 1:num){
   flatImage(catsOther[,i])
}
## Scrambled Hatted Images
for(i in 1:num){
    flatImage(catHats[,newOrd[i]])
}
```

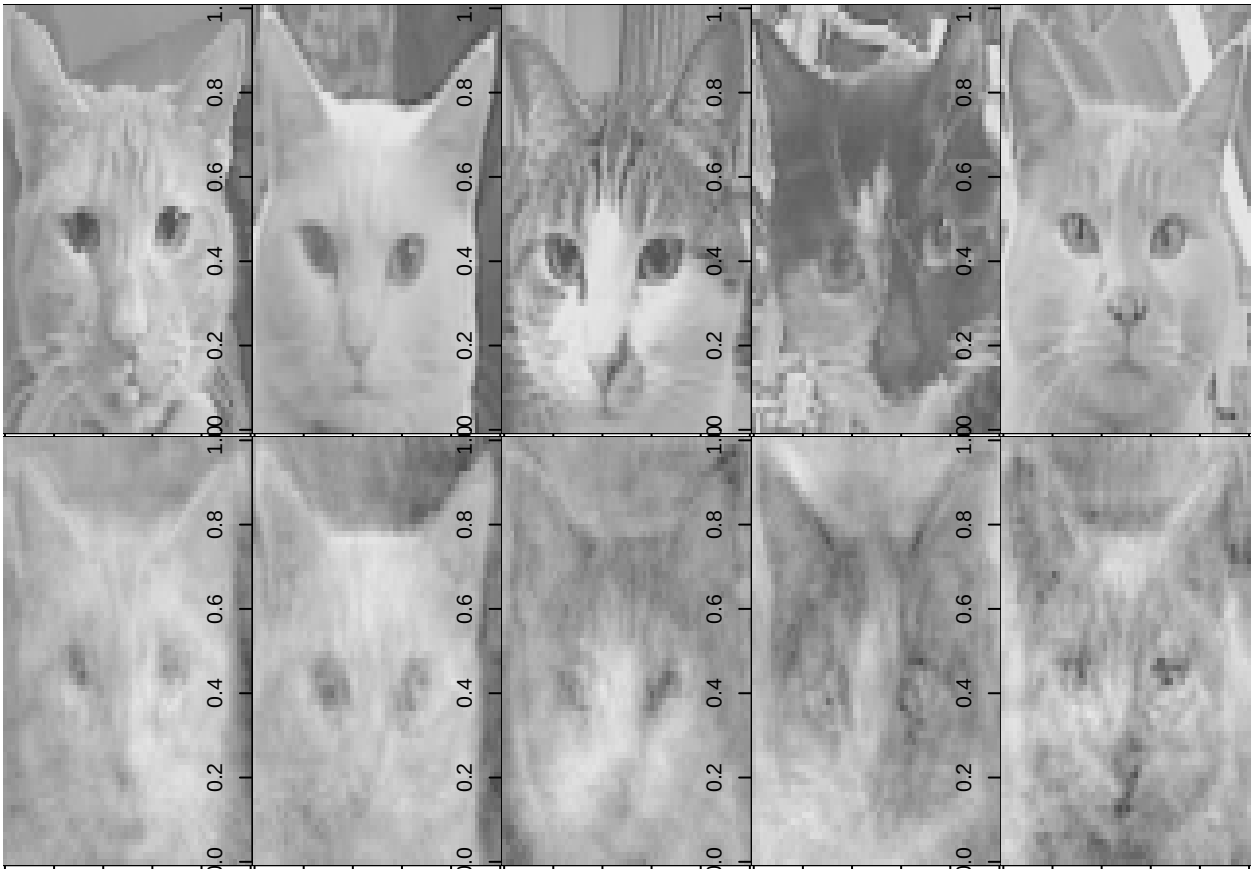


```
par(op)
```

Challenge: Match the original cats on the top row with their projections on the bottom row!

The answer....

```
op <- par(mfrow=c(2,num),mai=c(0.02,0.,0,0))
for(i in 1:num){
    image(imageConv(catsOther[,i]),col=grey.colors(256))
}
for(i in 1:num){
    image(imageConv(catHats[,i]),col=grey.colors(256))
}
```



```
par(op)
```

Here's the actual order. This says that the images are in the order newOrd[1],newOrd[2],newOrd[3],newOrd[4],newOrd[5]

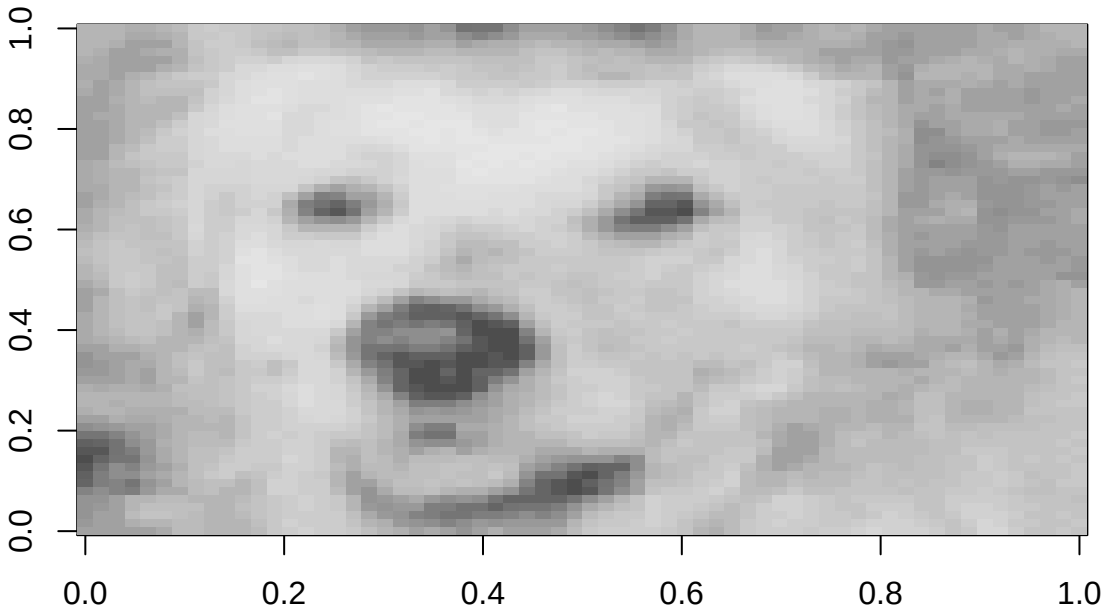
```
newOrd
```

```
## [1] 4 3 2 1 5
```

Dog Space

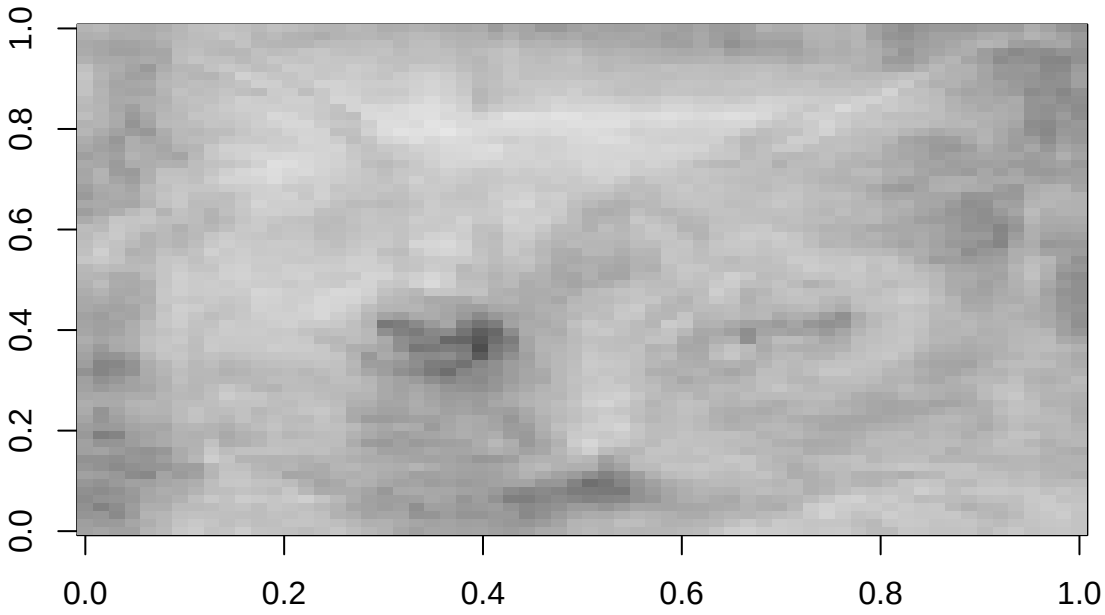
Pull off a dog!

```
aDog <- dogsAll[,5]
flatImage(aDog)
```



Let's project the dog onto the cat eigenspace.

```
aDogHat <- catHat %*% aDog
flatImage(aDogHat)
```



Compare together

```
op <- par(mfrow=c(2,1),mai=c(0.02,0.,0,0))
flatImage(aDog)
flatImage(aDogHat)
```

```
par(op)
```

What do you think?

Dueling Eigenspaces

Now let's create eigenspaces for both Cats and Dogs.

Define all the cat stuff

```
cats0 <- scale(cats,scale=F)
cat.svd <- svd(cats0)
catU <- cat.svd$u
catHat <- catU %*% t(catU)
```

Same for dogs...

```
dogs0 <-  scale(dogs,scale=F)
dog.svd <- svd(dogs0)
dogU <- dog.svd$u
dogHat <- dogU %*% t(dogU)
```

If we have an any image, we can project it into both the cat and the dog eigenspaces.

```
anImage <- catsOther[,6]
```

```
catProj <- catHat %*% anImage
dogProj <- dogHat %*% anImage
```

Which space is the original image closest to?

```
catDist <- mean((anImage-catProj)^2)
dogDist <- mean((anImage-dogProj)^2)
```

Compare the distances.

```
c(catDist,dogDist)
```

```
## [1] 17412.39 17519.82
```

Try something from the dog images.

```
anImage <- dogsOther[,6]
catProj <- catHat %*% anImage
dogProj <- dogHat %*% anImage
catDist <- mean((anImage-catProj)^2)
dogDist <- mean((anImage-dogProj)^2)
c(catDist,dogDist)
```

```
## [1] 25718.62 25516.28
```

Closer to the dog space (as well it should be).

More faces....

Let's take a crack at the the other faces set.

```
facesAll <- read.csv(file.path(dataDir,"face.csv"),header=F)
dim(facesAll)
```

```
## [1] 4096    2
```

Only two faces.

Try to classify the first of these.

```
aFace1 <- facesAll[,1]
aFaceCat1 <- catHat %*% aFace1
aFaceDog1 <- dogHat %*% aFace1
catDist1 <- mean((aFace1-aFaceCat1)^2)
dogDist1 <- mean((aFace1-aFaceDog1)^2)
```

Compare the distances.

```
c(catDist1,dogDist1)
```

```
## [1] 22702.42 22516.73
```

Much more dog-like!

Now try the other one.

```
aFace2 <- facesAll[,2]
aFaceCat2 <- catHat %*% aFace2
aFaceDog2 <- dogHat %*% aFace2
catDist2 <- mean((aFace2-aFaceCat2)^2)
dogDist2 <- mean((aFace2-aFaceDog2)^2)
c(catDist2,dogDist2)
```

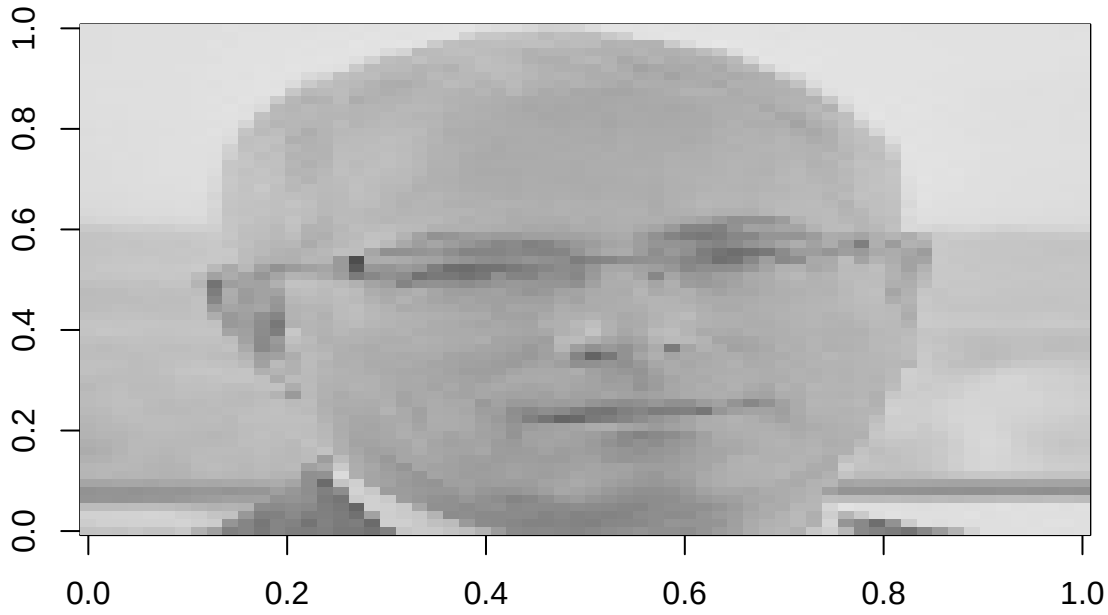
```
## [1] 19327.05 19444.41
```

This one is more cat-like

What do these faces look like? Is there some reason to understand the difference.

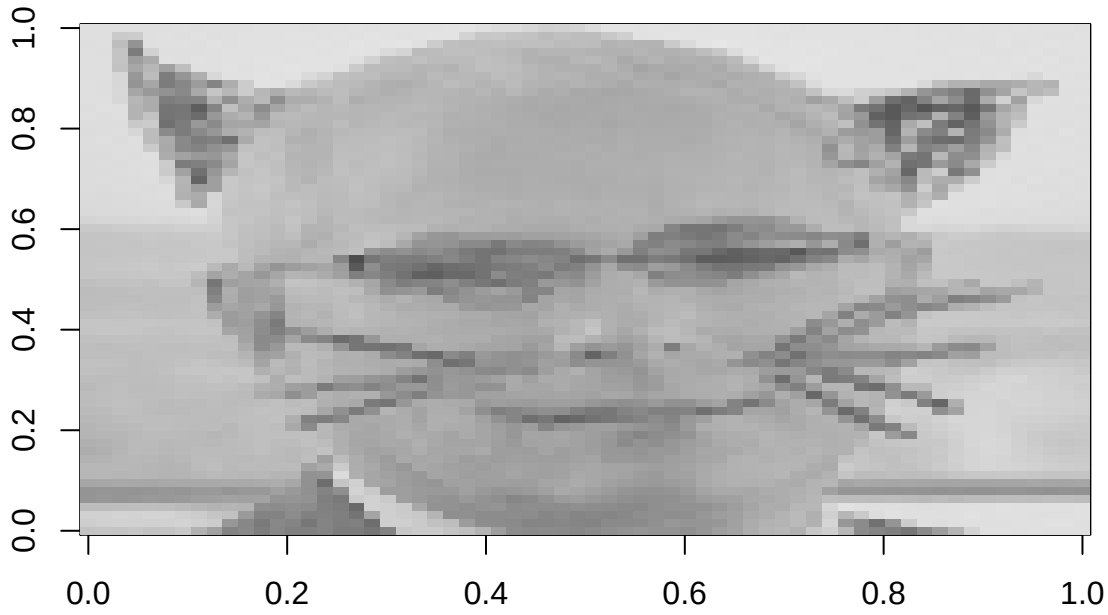
The first face.

```
flatImage(aFace1)
```



Interesting...

```
flatImage(aFace2)
```



It's all so clear now.

Assigment: Prediction with Eigenfaces

Does prediction into the dog or cat eigenspace reliability predict if an images is a dog or a cat? That is, for faces not in the training sets, does the quality of the reconstruction indicate "catness" versus "dogness"? Classify all the test cases and track your error rate.

Next, how does this method compare to using penalized regression? To do so, you now need to think of a model in which each observation is a face and the predictors are the pixel values. You need to mildly modify the data so that you have one large data set with rows associated with all the cat and dog images and a response variable indicating which type of image it is (cat or dog).

Note: In the big data sets, there are 99 cat images and 99 dog images. Each has 4096 pixels. Combine these, along with binary response variable to create a data set of 198 observations and 4096 predictors and 1 response.

```
classifyImageCat <- function(image){
  imageCat <- catHat %*% image
  imageDog <- dogHat %*% image
  distCat <- mean((image-imageCat)^2)
  distDog <- mean((image-imageDog)^2)
```

```
  distCat < distDog
}
```

Test it out.

```
catImage <- catsOther[,1]
dogImage<- dogsOther[,1]
classifyImageCat(catImage)
```

```
## [1] TRUE
```

```
classifyImageCat(dogImage)
```

```
## [1] FALSE
```

Run against all the other dogs and cats.

Calculate error rates for cats, dogs, and combined.

```
catsClass <- apply(catsOther,2,classifyImageCat)
mean(!catsClass)
```

```
## [1] 0.1111111
```

```
dogsClass <- apply(dogsOther,2,classifyImageCat)
mean(dogsClass)
```

```
## [1] 0.2222222
```

Overall

```
(err.hat <- mean(c(!catsClass,dogsClass)))
```

```
## [1] 0.1666667
```

Not bad.

Confusion Matrix, of sorts.

```
trueVals <- rep(c("Cat","Dog"),each=9)
table(trueVals,(c(catsClass,!dogsClass)))
```

```
##
## trueVals FALSE TRUE
##      Cat     1    8
##      Dog     2    7
```

Penalized regression

We need to build the data matrices

First the training matrices. We need to transpose the dogs and cats matrices.

```
trainPets <- rbind(t(cats),t(dogs))
dim(trainPets)
```

```
## [1]  180 4096
```

This is the training matrix. We need a response.

```
trainResp <- rep(c("Cat","Dog"),each=90)
length(trainResp)
```

```
## [1] 180
```

The same for the testing data

```
testPets <- rbind(t(catsOther),t(dogsOther))
dim(testPets)
```

```
## [1]   18 4096
```

The test response.

```
testResp <- rep(c("Cat","Dog"),each=9)
length(testResp)
```

```
## [1] 18
```

```
library(glmnet)
```

```
## Loading required package: Matrix
```

```
##
## Attaching package: 'Matrix'
```

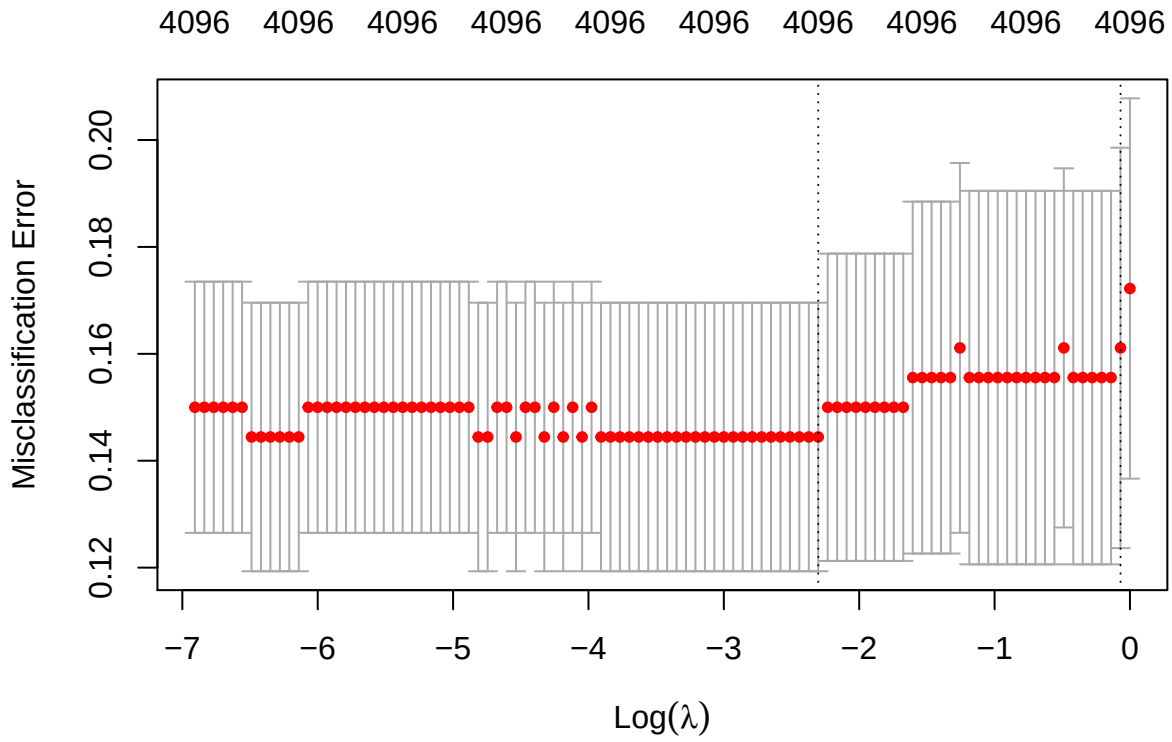
```
## The following objects are masked from 'package:tidyr':
##
##     expand, pack, unpack
```

```
## Loaded glmnet 3.0-2
```

First, ridge regression

```
lambda.grid <- 10^seq(-3,0,length=100)
mod.ridge <- cv.glmnet(trainPets,trainResp,
                  alpha=0,
                  family="binomial",
                  type.measure="class",
                  lambda=lambda.grid)

plot(mod.ridge)
```



Cross-validated error

```
(err.ridge <- with(mod.ridge,cvm[lambda==lambda.1se]))
```

```
## [1] 0.1611111
```

Get a look at the confusion matrix and the error on the test data.

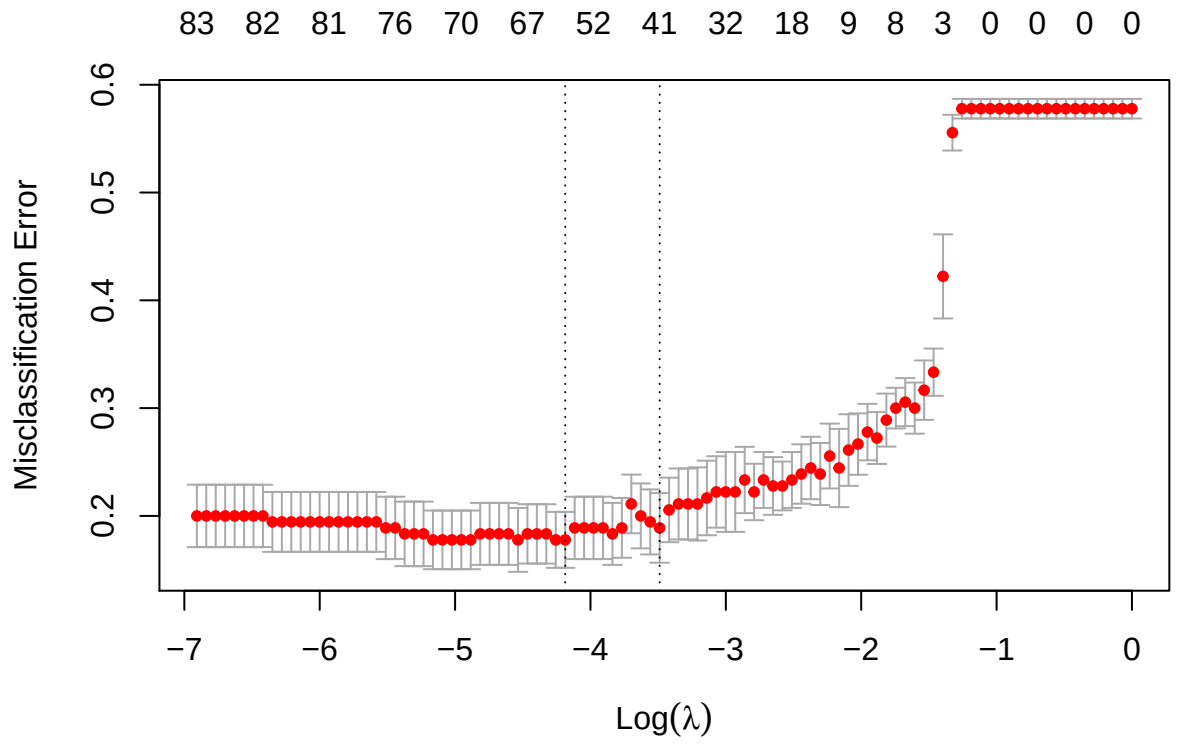
```
mod.ridge.opt <- glmnet(trainPets,trainResp,
                  alpha=0,
                  family="binomial",
                  type.measure="class",
                  lambda=mod.ridge$lambda.1se)
pred.ridge <- predict(mod.ridge.opt,newx=testPets,type="response") > 0.5
table(testResp,pred.ridge)
```

```
##         pred.ridge
## testResp FALSE TRUE
##      Cat     7    2
##      Dog     1    8
```

Same for lasso.

```
mod.lasso <- cv.glmnet(trainPets,trainResp,
                       alpha=1,
                       family="binomial",
                       type.measure="class",
                       lambda=lambda.grid)

plot(mod.lasso)
```




```
##Cross-validated error
(err.lasso <- with(mod.lasso,cvm[lambda==lambda.1se]))
```

```
## [1] 0.1888889
```

As before, get a look at the confusion matrix and the error on the test data.

```
mod.lasso.opt <- glmnet(trainPets,trainResp,
                       alpha=1,
                       family="binomial",
                       type.measure="class",
                       lambda=mod.lasso$lambda.1se)
pred.lasso <- predict(mod.lasso.opt,newx=testPets,type="response") > 0.5
table(testResp,pred.lasso)
```

```
##         pred.lasso
```

```
## testResp FALSE TRUE
##      Cat     7    2
##      Dog     2    7
```

Overall....

```
c(err.hat,err.ridge,err.lasso)
```

```
## [1] 0.1666667 0.1611111 0.1888889
```

In summary, they are all within shouting distance of each other.