Algorithmic Notes on Face Recognition

Oliver Dürr

```
EVAL = TRUE
ECHO = TRUE
```

Introduction

These algorithmic notes eluciade the classifiers used in the SoE project *piVision*. For illustrating purpose the algrithms orignially implemented in python using the openCV and scikit library are partly reimplemented in R.

To evaluate the algorithm we used the `Rasberry Pi` with an attached picam to take several photographs for each person of the team in two batches, indoors and outdoors. We use the indoor pictures as a training set to learn the classifier and the outdoor batch for evaluation. (**TODO** or other way around?)

What happend before (preprocessing)?

Before the feature extraction and classification we aligned and cropped the faces from the photos using the following method:

- **TODO** add the tricks with downscaling for performance increase.
- In a first step faces are detected from the picture using the Viola-Jones Algorithm provided in `openCV` library. This boosting algorithm worked quite well and is descriped in XXX.
- **TODO** include a typical image.
- An ellipse around the face is set to zero (weak effect)
- The image is split into 3 vertcial stripes and each stripe is normalized (**TODO** provide details)

In the following the base-line classification algorithms are descriped and documented using the R. Slighty better results can be obtained when local binary pattern instead of the original grey values per pixels are used. This can be further enhanced by applying the LBP at different resolution on the image or parts of the image. See end to this report for some notes on these more adavanced techniques.

Training data.

The following loads the 226 aligned faces from the training set and plots several of them.

```
trainingFile = "../../data/training_48x48_aligned_large.p_R.csv.gz"
testFile = "../../data/testing_48x48_aligned_large.p_R.csv.gz"
#trainingFile = "../../data/training_48x48_unaligned_large.p_R.csv.gz"
#testFile = "../../data/testing_48x48_unaligned_large.p_R.csv.gz" #TODO check this testfile can be wr
source("Utils.R")
dumm <- read.table(trainingFile, sep=",", stringsAsFactors = FALSE)
ddd <- as.matrix(dumm);
X_training <- ddd[,-1]
y_training <- ddd[,1]
#perm <- sample(1:ncol(X_training), replace = FALSE)
```

```
  #X_training <- X_training[,perm]
  N <- sqrt(ncol(X_training))
  cat("Loaded Training set ", dim(X_training), " Dimension of pixels: ", N, "x", N)
```

```
## Loaded Training set  226 2304  Dimension of pixels:  48 x 48
```

```
  plotExamples(y_training,X_training, title = "Training ", mfrow=c(3,6))
```

Test data

Same loading and plotting but now for the test-data.

```
dumm <- read.table(testFile, sep=",", stringsAsFactors = FALSE)
ddd <- as.matrix(dumm);X_testing <- ddd[,-1];y_testing <- ddd[,1]
#X_testing <- X_testing[,perm]
N <- sqrt(ncol(X_testing))
cat("Loaded Test set ", dim(X_testing), " Dimension of pixels: ", N, "x", N, " number of y ", length(y_
```

```
## Loaded Test set  215 2304  Dimension of pixels:  48 x 48  number of y  215
```

```
plotExamples(y_testing,X_testing, title = "Testing ", mfrow=c(3,6))
```

Eigenfaces, the spucky images

We unroll the images into a $48 \cdot 48 = 2304$ dimensional vector[1]. Each image is thus a point in a 2304 dimensional vector space. We now want to find a new basis in the 2304 dimensional space so that the variance (of the pixel intensities) is maximal in the first basis component, second maximal in the second component (which also needs to be othogonal to the first one). This can be achieved by first writing each image row by row into a matrix of the dimension 226x2304. The correlation matrix can be obtained by scaling the matrix so that the mean columns is 0 and the sd is 1. Due to constant values in some columns (e.g. in the masked regions) we cannot force the standard deviation to be 1 and therefore just scale the mean. This corresponds to using the covariance instead of the correlation.

```
X = scale(X_training, center = TRUE, scale = FALSE) / N
dim(X)
```

```
## [1]  226 2304
```

```
max(colSums(X))
```

```
## [1] 6.921e-14
```

The covariance matrix is simply $X^T X$.

```
cov = t(X)%*%X
dim(cov) #2304 x 2304
```

```
## [1] 2304 2304
```

To get the direction of with maximal covariance, we simply have to calculate the Eigenvectors of the matrix

[1]The data was in that format anyway

```r
cov = t(X)%*%X
e <- eigen(cov)
```

We simply roll the 2304 dimensional Eigenvectors back into an 48x48 dimensional images and have the infamous Eigenfaces.

However, it take quite some time to calculate the Eigenvectors from a 2304 dimensional matrix. Here we can apply the following trick[2]. Note that the singular value decomposition (SVD) decomposes a Matrix M under no assumptions! (**TODO: check**) into:

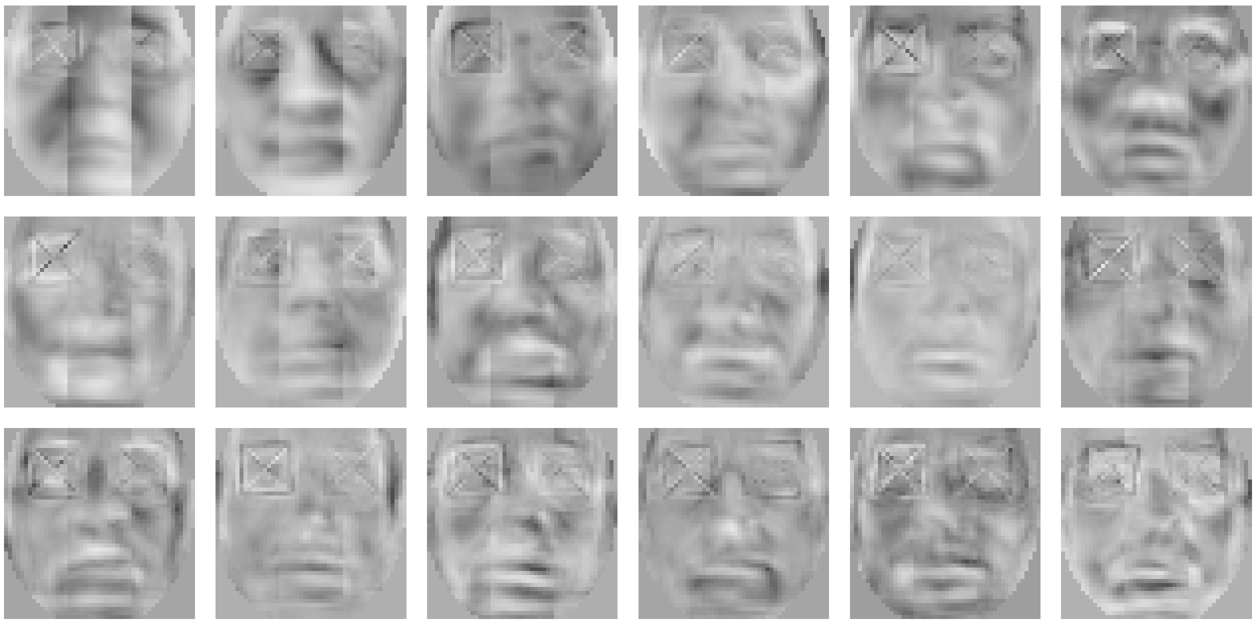
$$M = U \Sigma V^*$$

with U consisting of the Eigenvetors of MM^* with M^* being the conjugate transpose of M, Σ is a diagonal matrix of the Eigenvalues and V^* has the Eigenvectors of M^*M in its columns. So to calculate the Eigenvalue of X^TX we simply have to do an SVD of X^T and take $U \in \mathcal{R}^{2304x226}$ which contains then the Eigenvectors of X^TX as we wish.

```r
    res = svd(t(X))
    dim(res$u)
```

```
## [1] 2304  226
```

```r
    par(mfrow=c(3,6));par(mai=c(0.1,0.1,0.1,0.1))
    for (i in 1:18) {
      sm <- matrix(rev(res$u[,i]), ncol=N, byrow=TRUE)
      image(t(sm), useRaster=TRUE, main=NULL, col=gray.colors(255), axes = FALSE)
    }
```



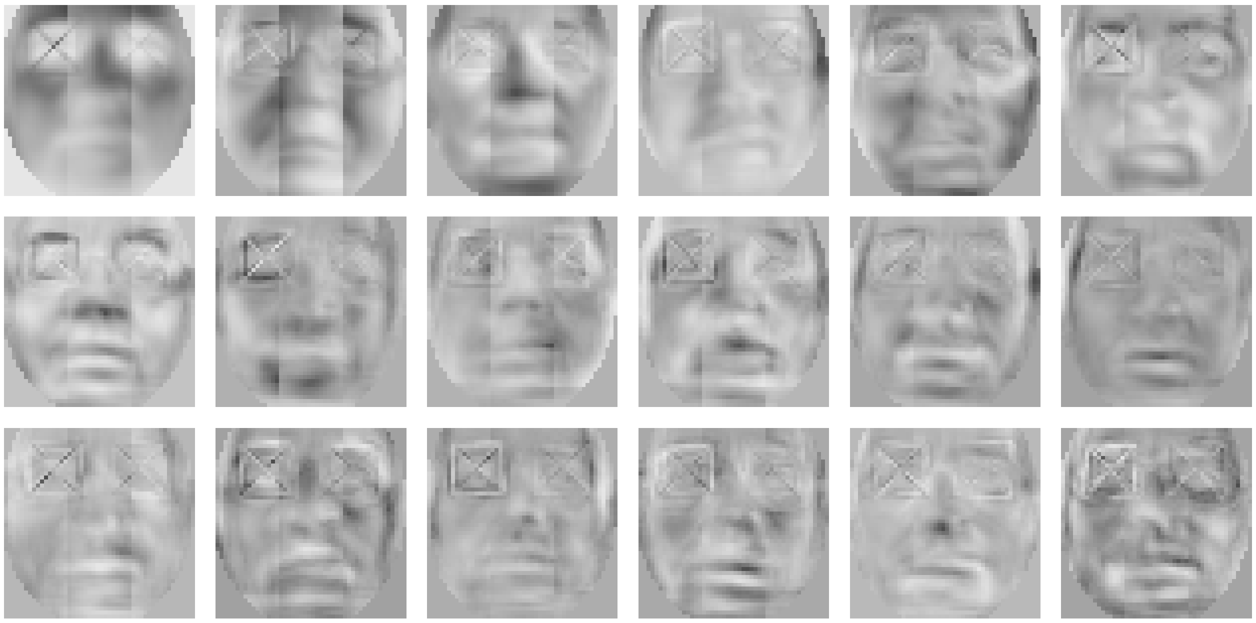
Alternatively we can also do this in R using the `princomp` function. **TODO** Explain the difference to R

```r
fit <- princomp(t(X_training), cor=FALSE)
res.sc <- fit$scores # the principal components
par(mfrow=c(3,6));par(mai=c(0.1,0.1,0.1,0.1))
dim(res.sc)
```

[2]There are also other numerical ways to calculate the principal components

```
## [1] 2304  226
```

```
for (i in 1:18) {
 # m <- scale(res.sc[,i])
  m <- res.sc[,i]
  sm <- matrix(rev(m), ncol=N, byrow=TRUE)
  image(t(sm), useRaster=TRUE, main=NULL, col=gray.colors(255), axes = FALSE)
}
```



```
par(mfrow=c(1,1))
```

The labeled training in the Eigenspaces

In the following we plot the training and testdata in the space spanned by the first two Eigenvectors. We color the datapoint according to the person they belong to.

```
par(mai=c(1.02,0.82,0.82,0.42))
pc.cr <- prcomp(X_training, center = FALSE)
X.train.pca <- pc.cr$x
dim(X.train.pca)
```

```
## [1] 226 226
```

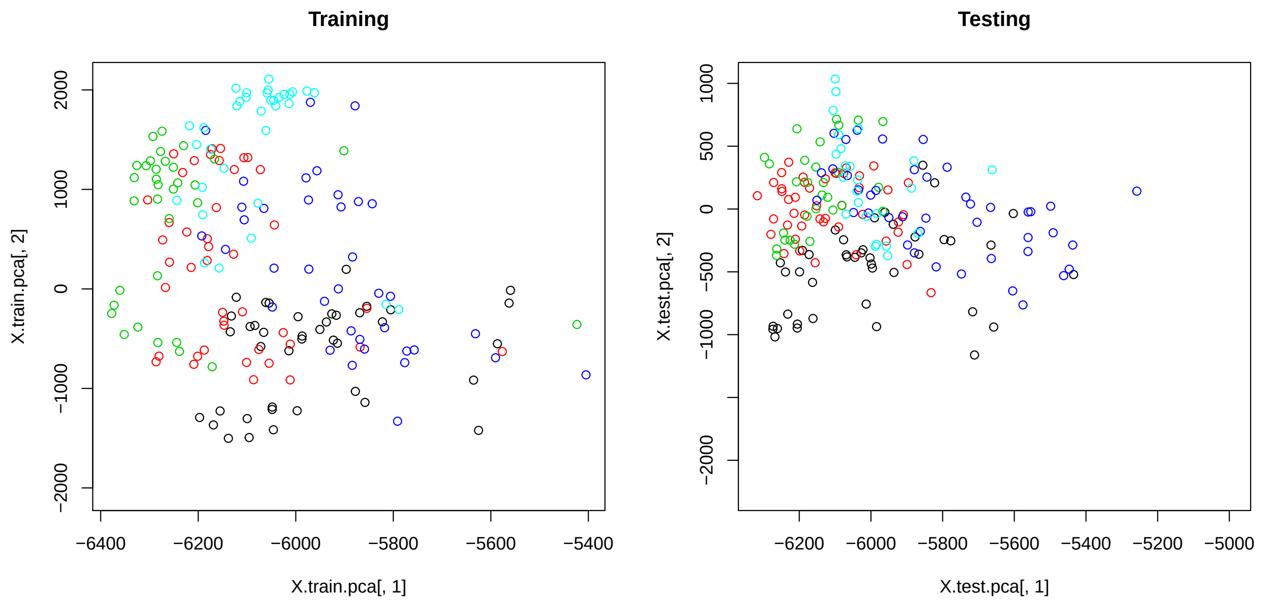
```
par(mfrow=c(1,2))
plot(X.train.pca[,1], X.train.pca[,2], col=y_training, main="Training")
dim(X.train.pca)
```

```
## [1] 226 226
```

```
### ... and the test data with labels
par(mai=c(1.02,0.82,0.82,0.42))
X.test.pca <- predict(pc.cr, X_testing) #
dim(X.test.pca)
```

```
## [1] 215 226
```

```
plot(X.test.pca[,1], X.test.pca[,2], col=y_testing, main="Testing")
```



```
par(mfrow=c(1,1))
```

We see some seperation of the individual persons already in the two dimensional slice. The classification algorithms below try to find a seperation of the space of `{r dim(pc.cr$x)` dimensions spanned by the Eigenvectors.[3] Note that the transformation is learned on the training set and then applied on the test set.

Classification

We now use the PCA for dimension reduction and apply serveral algorithms for classification.

- Simple KNN in the space of the Eigenfaces which is what people call: "Face Regonition with Eigenfaces"
- Fisher LDA in the space of the Eigenfaces aka Fisherfaces
- SVM in the space of the Eigenfaces

The classifier is learned on the training set and the performance is evaluted to the test set.

Eigenfaces

[3]Sometime only the first few EV are used, but we did not find a benifit in this evaluation.

```r
library(class)
sum(knn(train = X.train.pca, test = X.test.pca, cl = y_training) == y_testing) / length(y_testing)  # 0
```

```
## [1] 0.7349
```

Result from our openCV-pipeline (Build 246) Eigenfaces 0.720930233

Fisher Faces

Here we use the Fisher LDA in which the the space has to smaller than the number of examples, hence we need to have a dimension reduction such as the PCA.

```r
library(MASS)
z <- lda(X.train.pca[,1:200], y_training)
res <- predict(z, X.test.pca[,1:200])
sum(res$class == y_testing) / length(y_testing) #0.9162791
```

```
## [1] 0.9163
```

Result from our openCV-pipeline (Build 246) Fisherfaces 0.893023256

Running a SVM after PCA

```r
require(e1071)
```

```
## Loading required package: e1071
```

```r
#table(as.factor(y_training))
model <- svm(X.train.pca, as.factor(y_training), kernel='linear', cost=1, scale = TRUE)
test.svm <- predict(model, X.test.pca)
#table(test.svm)
sum(test.svm == y_testing)/ length(y_testing) #0.9023 for all
```

```
## [1] 0.9023
```

Result from our openCV-pipeline (Build 246) 3|4 (Simple unrolling FE) 0.874418605. Properbly because scaling is note set. (**TODO**) Set scaling to true.

Random forest on PCA

```r
library("randomForest")
```

```
## randomForest 4.6-7
## Type rfNews() to see new features/changes/bug fixes.
```

```
d.train <- data.frame(y = as.factor(y_training), X=X.train.pca)
model <- randomForest(y ~ ., data=d.train)
d.test <- data.frame(X=X.test.pca)
sum(predict(model, d.test) ==  y_testing)/ length(y_testing)
```

```
## [1] 0.7395
```

```
sum(predict(model, d.test) ==  y_testing)
```

```
## [1] 159
```

Not in our test cases.

Mulinomial regression

```
require(nnet)
```

```
## Loading required package: nnet
```

```
d.train <- data.frame(y = as.factor(y_training), X=X.train.pca[,1:40])
model <- multinom(y ~ ., data=d.train)
```

```
## # weights:  252 (205 variable)
## initial  value 404.937640
## iter  10 value 5.350377
## iter  20 value 2.930266
## iter  30 value 2.712086
## iter  40 value 2.603695
## iter  50 value 2.577603
## iter  60 value 2.570608
## iter  70 value 2.557677
## iter  80 value 2.525893
## iter  90 value 2.283754
## iter 100 value 1.812038
## final  value 1.812038
## stopped after 100 iterations
```

```
d.test <- data.frame(X=X.test.pca[,1:40])
sum(predict(model, d.test) ==  y_testing)/ length(y_testing)
```

```
## [1] 0.8093
```

Notes on advanced algorithms and the performance of variantions

Note: the following methods are not implemented in R (yet).

The face detection can be enhanced by taking not the original pixels as a starting point but use methods to extract features from them. We tested the following feature extractors

- 0 : Nothing needed for the original openCV faceregonizers
- 1 : SimpleLBHPFeatureExtractor (Whole Image)
- 2 : dito +1 Zooming
- 3 : dito +2 Zooming
- 4 : Unrolling FE, just use the pixels as in the above methods
- 5 : LBPHFeatureExtractor, extract patches near the nose and eyes and calculate LBP at these regions in various zoomings.

The extracted features are then used in the following classifiers

- LBHP: from openCV
- Fisher: FisherFaceRecognizer from openCV
- Eigen: EigenfaceRecognizer from openCV
- SVN: SVM after PCA
- SVN1: Same as SVN but different implementation (small deviations are expected)

For the aligned 48x48 pictures from above we get the following results (rev = revision number of SVN repository)

rev	LBPH	Fisher	Eigen	SVN/1	SVN/2	SVN/3	SVN/4	SVN/5	SVN1/4	SVN1/4
246	0.7813	0.8930	0.7209	0.9720	0.9674	0.9488	0.8744	0.93488	0.9813	0.9116

We see that taking the LBHP features (feature extractor 1) instead of the original pixels (feature extractor 5) results in a performance increase of about 4% from `SVN/5` to `SVN/1`.

The more complex feature extractors 2,3 and 5 play their strength only for larger images. The next table shows in the first line (rev 164) the result when using 120x120 images instead of the 48x48 ones. (Small remark images from Dejan just have been included starting from approx rev. <= 244).

rev	LBPH	Fisher	Eigen	SVN/1	SVN/2	SVN/3	SVN/4	SVN/5	SVN1/1	SVN1/4
164	0.8461	0.9230	0.7252	0.9560	0.9505	0.95604	0.89010	0.9725	0.9505	0.901
163	0.2362	0.2472	0.26373	0.2472	0.2142	0.14285	0.24725	0.1703	0.3076	0.247

We see that for larger images the complex feature extractors become relevant, SVN/5 is now better than SVN/1. However, the increase is not dramatic. The need for more complex FE should be evaluated probably with more challenging data-sets. We note also that for smaller images the simple FE 1 is as good as the complex FE 5 on larger images. This strongly suggest to use 48x48 images with the simple and LBHP feature extractor (**TODO** use in final revisions). Especially since the running time for the classification of one image of approx 250ms using SVN/1 increases by a factor of approximately 4 **TODO loma check**.

In revision 163 we switched off the alignment and observe a dramic drop of the performance pointing out the primordial importance of good alignment.

The effect of the different preprocessing method is evaluated in the following table:

rev	LBPH	Fisher	Eigen	SVN/1	SVN/2	SVN/3	SVN/4	SVN/5	SVN1/1	SVN1/4
161	0.868	0.874	0.522	0.868	0.852	0.841	0.874	0.857	0.868	0.868
173	0.863	0.890	0.692	0.956	0.945	0.945	0.907	0.967	0.951	0.863

rev	LBPH	Fisher	Eigen	SVN/1	SVN/2	SVN/3	SVN/4	SVN/5	SVN1/1	SVN1/4
183	0.830	0.918	0.725	0.956	0.951	0.956	0.912	0.978	0.956	0.918
164	0.846	0.9230	0.7252	0.956	0.951	0.956	0.890	0.972	0.9505	0.901

Full preprocessing

The best preprocessing so far splits the image into 3 vertical parts and equalizers the histogram for each part of the seperately

```
X1 = cv2.equalizeHist(X[0:, 0:SIZE_FOR_PCA[1] / 3])
X2 = cv2.equalizeHist(X[0:, SIZE_FOR_PCA[1] / 3:(SIZE_FOR_PCA[1] * 2 / 3)])
X3 = cv2.equalizeHist(X[0:, (SIZE_FOR_PCA[1] * 2 / 3):SIZE_FOR_PCA[1]])
X = np.append(np.append(X1, X2, 1), X3, 1)
```

Simple preprocessing

Use histogram equalization

```
X = cv2.equalizeHist(X)
```

Simple preprocessing

Is actually not to simple

```
# From https://github.com/bytefish/facerec/blob/master/py/facerec/preprocessing.py
alpha = 0.1
tau = 10.0
gamma = 0.2
sigma0 = 1.0
sigma1 = 2.0

sigma0 = 2.0
sigma1 = 5.0

X = np.array(X, dtype=np.float32)
X = np.power(X, gamma)
X = np.asarray(ndimage.gaussian_filter(X, sigma1) - ndimage.gaussian_filter(X, sigma0))
X = X / np.power(np.mean(np.power(np.abs(X), alpha)), 1.0 / alpha)
X = X / np.power(np.mean(np.power(np.minimum(np.abs(X), tau), alpha)), 1.0 / alpha)
X = tau * np.tanh(X / tau)
```

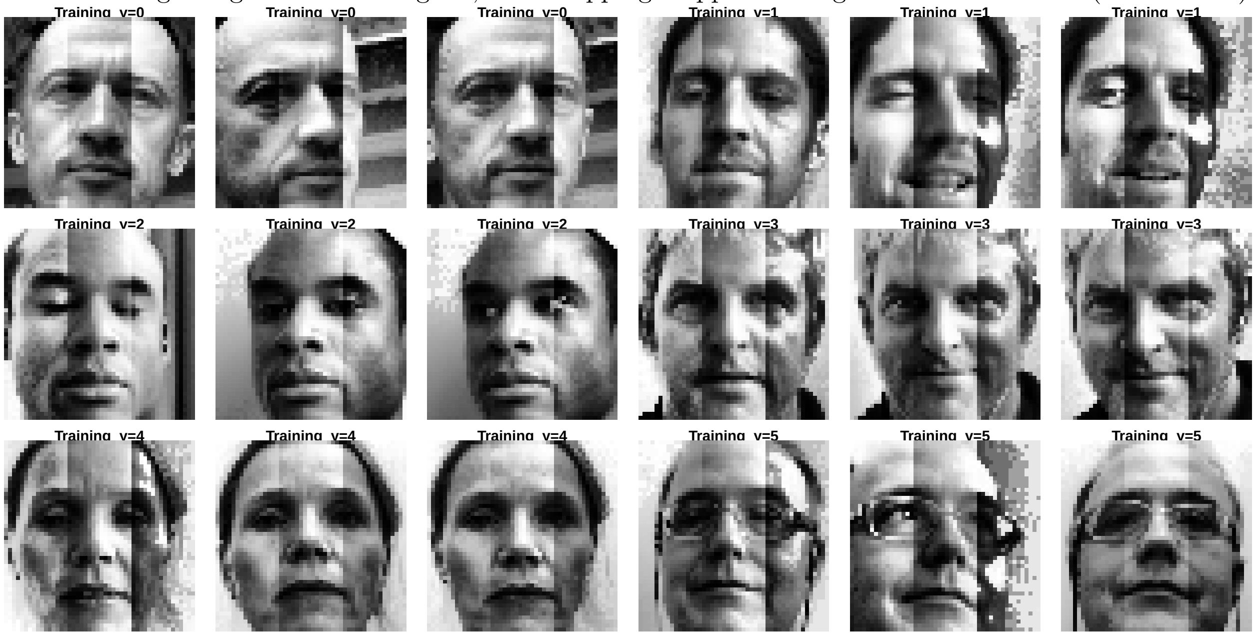
Effect of the alignment.

Since the effect of the alignment is so important, we like to elucidate a bit on it in the following.

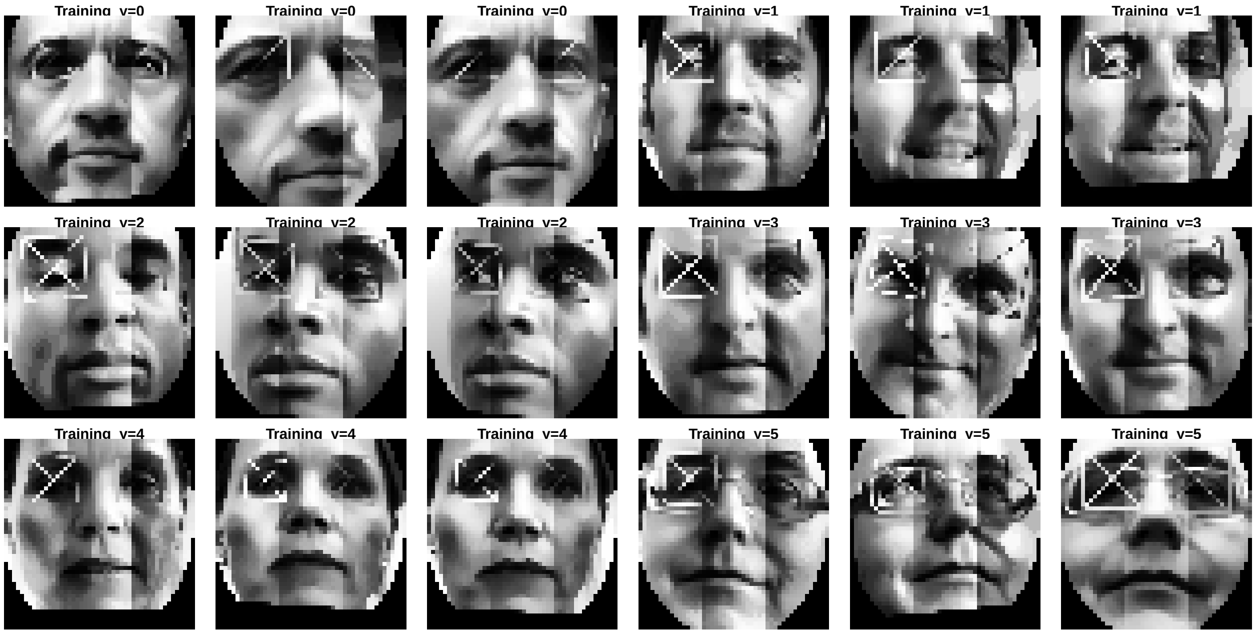
Below is a comparison of the pictures without and with the simple “two eyes” same position alignment.

Unaligned images

The following images are not aligned, the cropping happend using the detected face (Viola-Jones).

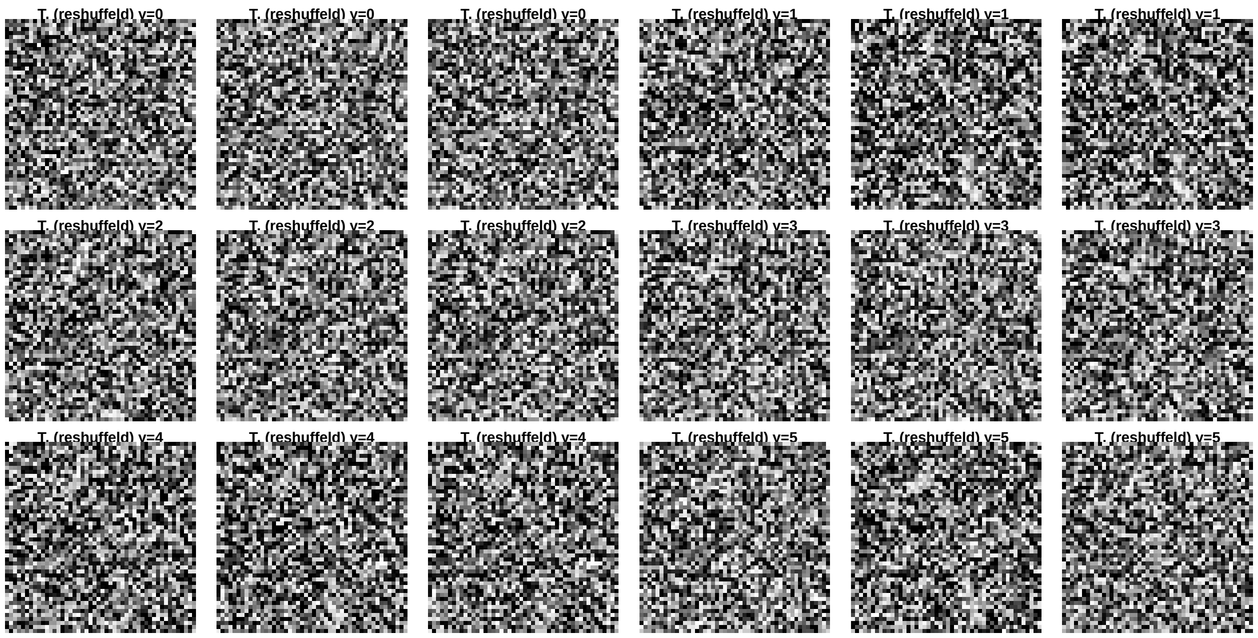


Aligned images



It's really hard to believe that the small distortions have such a strong effect. However the algorithms are not very robust against little transformations. Accually the Algorithms don't care if you reshuffel the image, the performance is as good as you let it run on those images.

Aligned images (reshuffeled)



Do you see that the first three pictures belong to the same person, well the algorithm does. I really did the experiment the performance stays the same, as long as you perform the same reshuffeling on each image!

.	Eigenfaces	Fisher	SVM
Reshuffeled	0.7349	0.9163	0.9023
Original	0.7349	0.9163	0.9023

For the more advaced methods (not implemented in R see below) this is not true anymore but still the performance decreases significantly if you don't align.

Evaluation with existing

http://vis-www.cs.umass.edu/lfw/