Face Recognition with Python

Philipp Wagner
http://www.bytefish.de

May 10, 2015

Contents

1 Introduction **1**

2 Face Recognition **2**
2.1 Face Database 2
2.1.1 Reading the images with Python 3
2.2 Eigenfaces 4
2.2.1 Algorithmic Description 4
2.2.2 Eigenfaces in Python 5
2.3 Fisherfaces 10
2.3.1 Algorithmic Description 10
2.3.2 Fisherfaces in Python 12

3 Conclusion **15**

1 Introduction

In this document I'll show you how to implement the Eigenfaces [13] and Fisherfaces [3] method with Python, so you'll understand the basics of Face Recognition. All concepts are explained in detail, but a basic knowledge of Python is assumed. Originally this document was a Guide to Face Recognition with OpenCV. Since OpenCV now comes with the cv::FaceRecognizer, this document has been reworked into the official OpenCV documentation at:

- http://docs.opencv.org/trunk/modules/contrib/doc/facerec/index.html

I am doing all this in my spare time and I simply can't maintain two separate documents on the same topic any more. So I have decided to turn this document into a guide on Face Recognition with Python only. You'll find the very detailed documentation on the OpenCV cv::FaceRecognizer at:

- FaceRecognizer - Face Recognition with OpenCV
  - FaceRecognizer API
  - Guide to Face Recognition with OpenCV
  - Tutorial on Gender Classification
  - Tutorial on Face Recognition in Videos
  - Tutorial On Saving & Loading a FaceRecognizer

By the way you don't need to copy and paste the code snippets, all code has been pushed into my github repository:

- github.com/bytefish
- github.com/bytefish/facerecognition_guide

Everything in here is released under a BSD license, so feel free to use it for your projects. You are currently reading the Python version of the Face Recognition Guide, you can compile the GNU Octave/MATLAB version with `make octave`.

2 Face Recognition

Face recognition is an easy task for humans. Experiments in [6] have shown, that even one to three day old babies are able to distinguish between known faces. So how hard could it be for a computer? It turns out we know little about human recognition to date. Are inner features (eyes, nose, mouth) or outer features (head shape, hairline) used for a successful face recognition? How do we analyze an image and how does the brain encode it? It was shown by David Hubel and Torsten Wiesel, that our brain has specialized nerve cells responding to specific local features of a scene, such as lines, edges, angles or movement. Since we don't see the world as scattered pieces, our visual cortex must somehow combine the different sources of information into useful patterns. Automatic face recognition is all about extracting those meaningful features from an image, putting them into a useful representation and performing some kind of classification on them.

Face recognition based on the geometric features of a face is probably the most intuitive approach to face recognition. One of the first automated face recognition systems was described in [9]: marker points (position of eyes, ears, nose, ...) were used to build a feature vector (distance between the points, angle between them, ...). The recognition was performed by calculating the euclidean distance between feature vectors of a probe and reference image. Such a method is robust against changes in illumination by its nature, but has a huge drawback: the accurate registration of the marker points is complicated, even with state of the art algorithms. Some of the latest work on geometric face recognition was carried out in [4]. A 22-dimensional feature vector was used and experiments on large datasets have shown, that geometrical features alone don't carry enough information for face recognition.

The Eigenfaces method described in [13] took a holistic approach to face recognition: A facial image is a point from a high-dimensional image space and a lower-dimensional representation is found, where classification becomes easy. The lower-dimensional subspace is found with Principal Component Analysis, which identifies the axes with maximum variance. While this kind of transformation is optimal from a reconstruction standpoint, it doesn't take any class labels into account. Imagine a situation where the variance is generated from external sources, let it be light. The axes with maximum variance do not necessarily contain any discriminative information at all, hence a classification becomes impossible. So a class-specific projection with a Linear Discriminant Analysis was applied to face recognition in [3]. The basic idea is to minimize the variance within a class, while maximizing the variance between the classes at the same time (Figure 1).

Recently various methods for a local feature extraction emerged. To avoid the high-dimensionality of the input data only local regions of an image are described, the extracted features are (hopefully) more robust against partial occlusion, illumation and small sample size. Algorithms used for a local feature extraction are Gabor Wavelets ([14]), Discrete Cosinus Transform ([5]) and Local Binary Patterns ([1, 11, 12]). It's still an open research question how to preserve spatial information when applying a local feature extraction, because spatial information is potentially useful information.

2.1 Face Database

I don't want to do a toy example here. We are doing face recognition, so you'll need some face images! You can either create your own database or start with one of the available databases, face-rec.org/databases gives an up-to-date overview. Three interesting databases are[1]:

AT&T Facedatabase The AT&T Facedatabase, sometimes also known as *ORL Database of Faces*, contains ten different images of each of 40 distinct subjects. For some subjects, the images were taken at different times, varying the lighting, facial expressions (open / closed eyes, smiling / not smiling) and facial details (glasses / no glasses). All the images were taken against a dark homogeneous background with the subjects in an upright, frontal position (with tolerance for some side movement).

Yale Facedatabase A The AT&T Facedatabase is good for initial tests, but it's a fairly easy database. The Eigenfaces method already has a 97% recognition rate, so you won't see any improvements with other algorithms. The Yale Facedatabase A is a more appropriate dataset for initial experiments, because the recognition problem is harder. The database consists of 15 people (14 male, 1 female) each with 11 grayscale images sized 320×243 pixel. There are

[1] Parts of the description are quoted from face-rec.org.

changes in the light conditions (center light, left light, right light), facial expressions (happy, normal, sad, sleepy, surprised, wink) and glasses (glasses, no-glasses).

The original images are not cropped or aligned. I've prepared a Python script available in `src/py/crop_face.py`, that does the job for you.

Extended Yale Facedatabase B The Extended Yale Facedatabase B contains 2414 images of 38 different people in its cropped version. The focus is on extracting features that are robust to illumination, the images have almost no variation in emotion/occlusion/.... I personally think, that this dataset is too large for the experiments I perform in this document, you better use the AT&T Facedatabase. A first version of the Yale Facedatabase B was used in [3] to see how the Eigenfaces and Fisherfaces method (section 2.3) perform under heavy illumination changes. [10] used the same setup to take 16128 images of 28 people. The Extended Yale Facedatabase B is the merge of the two databases, which is now known as Extended Yalefacedatabase B.

The face images need to be stored in a folder hierachy similar to `<datbase name>/<subject name>/<filename>.<ext>`. The AT&T Facedatabase for example comes in such a hierarchy, see Listing 1.

Listing 1:

```
philipp@mango:~/facerec/data/at$ tree
.
|-- README
|-- s1
|   |-- 1.pgm
|   |-- ...
|   |-- 10.pgm
|-- s2
|   |-- 1.pgm
|   |-- ...
|   |-- 10.pgm
...
|-- s40
|   |-- 1.pgm
|   |-- ...
|   |-- 10.pgm
```

2.1.1 Reading the images with Python

The function in Listing 2 can be used to read in the images for each subfolder of a given directory. Each directory is given a unique (integer) label, you probably want to store the folder name as well. The function returns the images and the corresponding classes. This function is really basic and there's much to enhance, but it does its job.

Listing 2: src/py/tinyfacerec/util.py

```
def read_images(path, sz=None):
  c = 0
  X,y = [], []
  for dirname, dirnames, filenames in os.walk(path):
    for subdirname in dirnames:
      subject_path = os.path.join(dirname, subdirname)
      for filename in os.listdir(subject_path):
        try:
          im = Image.open(os.path.join(subject_path, filename))
          im = im.convert("L")
          # resize to given size (if given)
          if (sz is not None):
            im = im.resize(sz, Image.ANTIALIAS)
          X.append(np.asarray(im, dtype=np.uint8))
          y.append(c)
        except IOError:
          print "I/O error({0}): {1}".format(errno, strerror)
        except:
          print "Unexpected error:", sys.exc_info()[0]
          raise
      c = c+1
  return [X,y]
```

2.2 Eigenfaces

The problem with the image representation we are given is its high dimensionality. Two-dimensional $p \times q$ grayscale images span a $m = pq$-dimensional vector space, so an image with 100×100 pixels lies in a $10,000$-dimensional image space already. That's way too much for any computations, but are all dimensions really useful for us? We can only make a decision if there's any variance in data, so what we are looking for are the components that account for most of the information. The Principal Component Analysis (PCA) was independently proposed by Karl Pearson (1901) and Harold Hotelling (1933) to turn a set of possibly correlated variables into a smaller set of uncorrelated variables. The idea is that a high-dimensional dataset is often described by correlated variables and therefore only a few meaningful dimensions account for most of the information. The PCA method finds the directions with the greatest variance in the data, called principal components.

2.2.1 Algorithmic Description

Let $X = \{x_1, x_2, \ldots, x_n\}$ be a random vector with observations $x_i \in \mathbb{R}^d$.

1. Compute the mean μ
$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{1}$$
2. Compute the the Covariance Matrix S
$$S = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)(x_i - \mu)^T \tag{2}$$
3. Compute the eigenvalues λ_i and eigenvectors v_i of S
$$Sv_i = \lambda_i v_i, i = 1, 2, \ldots, n \tag{3}$$
4. Order the eigenvectors descending by their eigenvalue. The k principal components are the eigenvectors corresponding to the k largest eigenvalues.

The k principal components of the observed vector x are then given by:

$$y = W^T(x - \mu) \tag{4}$$

where $W = (v_1, v_2, \ldots, v_k)$. The reconstruction from the PCA basis is given by:

$$x = Wy + \mu \tag{5}$$

The Eigenfaces method then performs face recognition by:

1. Projecting all training samples into the PCA subspace (using Equation 4).
2. Projecting the query image into the PCA subspace (using Listing 5).
3. Finding the nearest neighbor between the projected training images and the projected query image.

Still there's one problem left to solve. Imagine we are given 400 images sized 100×100 pixel. The Principal Component Analysis solves the covariance matrix $S = XX^T$, where $size(X) = 10000 \times 400$ in our example. You would end up with a 10000×10000 matrix, roughly $0.8GB$. Solving this problem isn't feasible, so we'll need to apply a trick. From your linear algebra lessons you know that a $M \times N$ matrix with $M > N$ can only have $N - 1$ non-zero eigenvalues. So it's possible to take the eigenvalue decomposition $S = X^T X$ of size NxN instead:

$$X^T X v_i = \lambda_i vi \tag{6}$$

and get the original eigenvectors of $S = XX^T$ with a left multiplication of the data matrix:

$$XX^T(Xv_i) = \lambda_i(Xv_i) \tag{7}$$

The resulting eigenvectors are orthogonal, to get orthonormal eigenvectors they need to be normalized to unit length. I don't want to turn this into a publication, so please look into [7] for the derivation and proof of the equations.

2.2.2 Eigenfaces in Python

We've already seen, that the Eigenfaces and Fisherfaces method expect a data matrix with observations by row (or column if you prefer it). Listing 3 defines two functions to reshape a list of multi-dimensional data into a data matrix. Note, that all samples are assumed to be of equal size.

Listing 3: src/py/tinyfacerec/util.py

```
def asRowMatrix(X):
  if len(X) == 0:
    return np.array([])
  mat = np.empty((0, X[0].size), dtype=X[0].dtype)
  for row in X:
    mat = np.vstack((mat, np.asarray(row).reshape(1,-1)))
  return mat

def asColumnMatrix(X):
  if len(X) == 0:
    return np.array([])
  mat = np.empty((X[0].size, 0), dtype=X[0].dtype)
  for col in X:
    mat = np.hstack((mat, np.asarray(col).reshape(-1,1)))
  return mat
```

Translating the PCA from the algorithmic description of section 2.2.1 to Python is almost trivial. Don't copy and paste from this document, the source code is available in folder `src/py/tinyfacerec` . Listing 4 implements the Principal Component Analysis given by Equation 1, 2 and 3. It also implements the inner-product PCA formulation, which occurs if there are more dimensions than samples. You can shorten this code, I just wanted to point out how it works.

Listing 4: src/py/tinyfacerec/subspace.py

```
def pca(X, y, num_components=0):
  [n,d] = X.shape
  if (num_components <= 0) or (num_components>n):
    num_components = n
  mu = X.mean(axis=0)
  X = X - mu
  if n>d:
    C = np.dot(X.T,X)
    [eigenvalues,eigenvectors] = np.linalg.eigh(C)
  else:
    C = np.dot(X,X.T)
    [eigenvalues,eigenvectors] = np.linalg.eigh(C)
    eigenvectors = np.dot(X.T,eigenvectors)
    for i in xrange(n):
      eigenvectors[:,i] = eigenvectors[:,i]/np.linalg.norm(eigenvectors[:,i])
  # or simply perform an economy size decomposition
  # eigenvectors, eigenvalues, variance = np.linalg.svd(X.T, full_matrices=False)
  # sort eigenvectors descending by their eigenvalue
  idx = np.argsort(-eigenvalues)
  eigenvalues = eigenvalues[idx]
  eigenvectors = eigenvectors[:,idx]
  # select only num_components
  eigenvalues = eigenvalues[0:num_components].copy()
  eigenvectors = eigenvectors[:,0:num_components].copy()
  return [eigenvalues, eigenvectors, mu]
```

The observations are given by row, so the projection in Equation 4 needs to be rearranged a little:

Listing 5: src/py/tinyfacerec/subspace.py

```
def project(W, X, mu=None):
  if mu is None:
    return np.dot(X,W)
  return np.dot(X - mu, W)
```

The same applies to the reconstruction in Equation 5:

Listing 6: src/py/tinyfacerec/subspace.py

```
def reconstruct(W, Y, mu=None):
  if mu is None:
    return np.dot(Y,W.T)
  return np.dot(Y, W.T) + mu
```

Now that everything is defined it's time for the fun stuff. The face images are read with Listing 2 and then a full PCA (see Listing 4) is performed. I'll use the great matplotlib library for plotting in Python, please install it if you haven't done already.

Listing 7: src/py/scripts/example_pca.py

```
import sys
# append tinyfacerec to module search path
sys.path.append("..")
# import numpy and matplotlib colormaps
import numpy as np
# import tinyfacerec modules
from tinyfacerec.subspace import pca
from tinyfacerec.util import normalize, asRowMatrix, read_images
from tinyfacerec.visual import subplot

# read images
[X,y] = read_images("/home/philipp/facerec/data/at")
# perform a full pca
[D, W, mu] = pca(asRowMatrix(X), y)
```

That's it already. Pretty easy, no? Each principal component has the same length as the original image, thus it can be displayed as an image. [13] referred to these ghostly looking faces as *Eigenfaces*, that's where the Eigenfaces method got its name from. We'll now want to look at the Eigenfaces, but first of all we need a method to turn the data into a representation matplotlib understands. The eigenvectors we have calculated can contain negative values, but the image data is excepted as unsigned integer values in the range of 0 to 255. So we need a function to normalize the data first (Listing 8):

Listing 8: src/py/tinyfacerec/util.py

```
def normalize(X, low, high, dtype=None):
  X = np.asarray(X)
  minX, maxX = np.min(X), np.max(X)
  # normalize to [0...1].
  X = X - float(minX)
  X = X / float((maxX - minX))
  # scale to [low...high].
  X = X * (high-low)
  X = X + low
  if dtype is None:
    return np.asarray(X)
  return np.asarray(X, dtype=dtype)
```

In Python we'll then define a `subplot` method (see `src/py/tinyfacerec/visual.py`) to simplify the plotting. The method takes a list of images, a title, color scale and finally generates a subplot:

Listing 9: src/py/tinyfacerec/visual.py

```
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.cm as cm

def create_font(fontname='Tahoma', fontsize=10):
  return { 'fontname': fontname, 'fontsize':fontsize }

def subplot(title, images, rows, cols, sptitle="subplot", sptitles=[], colormap=cm.
    gray, ticks_visible=True, filename=None):
  fig = plt.figure()
  # main title
  fig.text(.5, .95, title, horizontalalignment='center')
  for i in xrange(len(images)):
    ax0 = fig.add_subplot(rows,cols,(i+1))
    plt.setp(ax0.get_xticklabels(), visible=False)
    plt.setp(ax0.get_yticklabels(), visible=False)
```

```
    if len(sptitles) == len(images):
      plt.title("%s #%s" % (sptitle, str(sptitles[i])), create_font('Tahoma',10))
    else:
      plt.title("%s #%d" % (sptitle, (i+1)), create_font('Tahoma',10))
    plt.imshow(np.asarray(images[i]), cmap=colormap)
  if filename is None:
    plt.show()
  else:
    fig.savefig(filename)
```

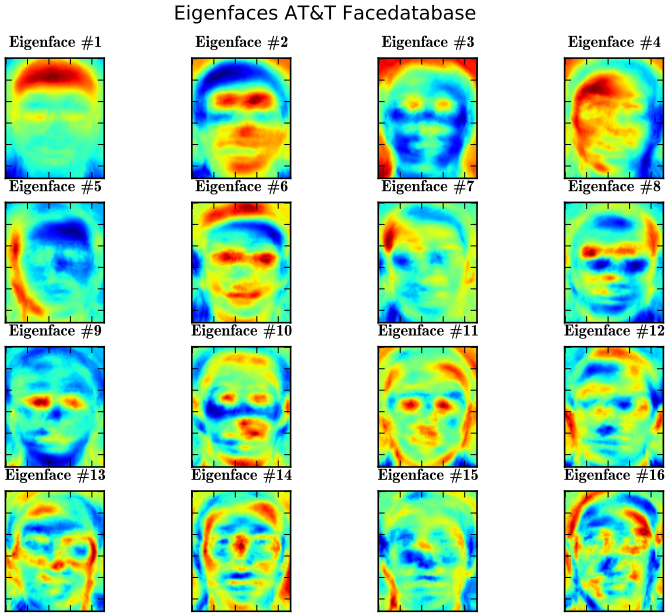
This simplified the Python script in Listing 10 to:

Listing 10: src/py/scripts/example_pca.py

```
import matplotlib.cm as cm

# turn the first (at most) 16 eigenvectors into grayscale
# images (note: eigenvectors are stored by column!)
E = []
for i in xrange(min(len(X), 16)):
  e = W[:,i].reshape(X[0].shape)
  E.append(normalize(e,0,255))
# plot them and store the plot to "python_eigenfaces.pdf"
subplot(title="Eigenfaces AT&T Facedatabase", images=E, rows=4, cols=4, sptitle="
    Eigenface", colormap=cm.jet, filename="python_pca_eigenfaces.pdf")
```

I've used the jet colormap, so you can see how the grayscale values are distributed within the specific Eigenfaces. You can see, that the Eigenfaces do not only encode facial features, but also the illumination in the images (see the left light in Eigenface #4, right light in Eigenfaces #5):



We've already seen in Equation 5, that we can reconstruct a face from its lower dimensional approximation. So let's see how many Eigenfaces are needed for a good reconstruction. I'll do a subplot with $10, 30, \ldots, 310$ Eigenfaces:

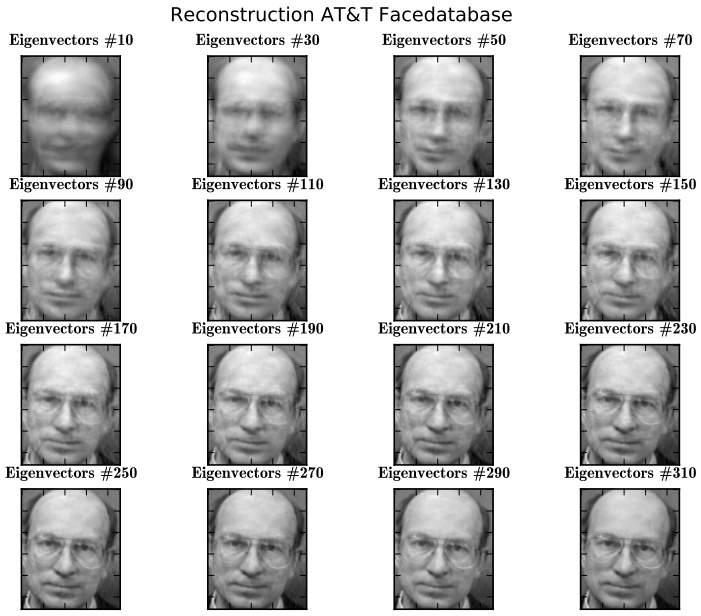
Listing 11: src/py/scripts/example_pca.py

```
from tinyfacerec.subspace import project, reconstruct

# reconstruction steps
steps=[i for i in xrange(10, min(len(X), 320), 20)]
E = []
for i in xrange(min(len(steps), 16)):
  numEvs = steps[i]
```

```
  P = project(W[:,0:numEvs], X[0].reshape(1,-1), mu)
  R = reconstruct(W[:,0:numEvs], P, mu)
  # reshape and append to plots
  R = R.reshape(X[0].shape)
  E.append(normalize(R,0,255))
# plot them and store the plot to "python_reconstruction.pdf"
subplot(title="Reconstruction AT&T Facedatabase", images=E, rows=4, cols=4, sptitle="
    Eigenvectors", sptitles=steps, colormap=cm.gray, filename="
    python_pca_reconstruction.pdf")
```

10 Eigenvectors are obviously not sufficient for a good image reconstruction, 50 Eigenvectors may already be sufficient to encode important facial features. You'll get a good reconstruction with approximately 300 Eigenvectors for the AT&T Facedatabase. There are rule of thumbs how many Eigenfaces you should choose for a successful face recognition, but it heavily depends on the input data. [15] is the perfect point to start researching for this.



Now we have got everything to implement the Eigenfaces method. Python is object oriented and so is our Eigenfaces model. Let's recap: The Eigenfaces method is basically a Pricipal Component Analysis with a Nearest Neighbor model. Some publications report about the influence of the distance metric (I can't support these claims with my research), so various distance metrics for the Nearest Neighbor should be supported. Listing 12 defines an `AbstractDistance` as the abstract base class for each distance metric. Every subclass overrides the call operator `__call__` as shown for the Euclidean Distance and the Negated Cosine Distance. If you need more distance metrics, please have a look at the distance metrics implemented in https://www.github.com/bytefish/facerec.

Listing 12: src/py/tinyfacerec/distance.py

```
import numpy as np

class AbstractDistance(object):
  def __init__(self, name):
    self._name = name

  def __call__(self,p,q):
    raise NotImplementedError("Every AbstractDistance must implement the __call__
        method.")

  @property
  def name(self):
    return self._name
```

```
  def __repr__(self):
    return self._name

class EuclideanDistance(AbstractDistance):

  def __init__(self):
    AbstractDistance.__init__(self,"EuclideanDistance")

  def __call__(self, p, q):
    p = np.asarray(p).flatten()
    q = np.asarray(q).flatten()
    return np.sqrt(np.sum(np.power((p-q),2)))

class CosineDistance(AbstractDistance):

  def __init__(self):
    AbstractDistance.__init__(self,"CosineDistance")

  def __call__(self, p, q):
    p = np.asarray(p).flatten()
    q = np.asarray(q).flatten()
    return -np.dot(p.T,q) / (np.sqrt(np.dot(p,p.T)*np.dot(q,q.T)))
```

The Eigenfaces and Fisherfaces method both share common methods, so we'll define a base prediction model in Listing 13. I don't want to do a full k-Nearest Neighbor implementation here, because (1) the number of neighbors doesn't really matter for both methods and (2) it would confuse people. If you are implementing it in a language of your choice, you should really separate the feature extraction and classification from the model itself. A real generic approach is given in my facerec framework. However, feel free to extend these basic classes for your needs.

Listing 13: src/py/tinyfacerec/model.py

```
import numpy as np
from util import asRowMatrix
from subspace import pca, lda, fisherfaces, project
from distance import EuclideanDistance

class BaseModel(object):
  def __init__(self, X=None, y=None, dist_metric=EuclideanDistance(), num_components
      =0):
    self.dist_metric = dist_metric
    self.num_components = 0
    self.projections = []
    self.W = []
    self.mu = []
    if (X is not None) and (y is not None):
      self.compute(X,y)

  def compute(self, X, y):
    raise NotImplementedError("Every BaseModel must implement the compute method.")

  def predict(self, X):
    minDist = np.finfo('float').max
    minClass = -1
    Q = project(self.W, X.reshape(1,-1), self.mu)
    for i in xrange(len(self.projections)):
      dist = self.dist_metric(self.projections[i], Q)
      if dist < minDist:
        minDist = dist
        minClass = self.y[i]
    return minClass
```

Listing 20 then subclasses the `EigenfacesModel` from the `BaseModel`, so only the `compute` method needs to be overriden with our specific feature extraction. The prediction is a 1-Nearest Neighbor search with a distance metric.

Listing 14: src/py/tinyfacerec/model.py

```
class EigenfacesModel(BaseModel):
```

```
  def __init__(self, X=None, y=None, dist_metric=EuclideanDistance(), num_components
      =0):
    super(EigenfacesModel, self).__init__(X=X,y=y,dist_metric=dist_metric,
        num_components=num_components)

  def compute(self, X, y):
    [D, self.W, self.mu] = pca(asRowMatrix(X),y, self.num_components)
    # store labels
    self.y = y
    # store projections
    for xi in X:
      self.projections.append(project(self.W, xi.reshape(1,-1), self.mu))
```

Now that the `EigenfacesModel` is defined, it can be used to learn the Eigenfaces and generate predictions. In the following Listing 15 we'll load the Yale Facedatabase A and perform a prediction on the first image.

Listing 15: src/py/scripts/example_model_eigenfaces.py

```
import sys
# append tinyfacerec to module search path
sys.path.append("..")
# import numpy and matplotlib colormaps
import numpy as np
# import tinyfacerec modules
from tinyfacerec.util import read_images
from tinyfacerec.model import EigenfacesModel

if __name__ == '__main__':

    if len(sys.argv) != 2:
        print "USAGE: example_model_eigenfaces.py </path/to/images>"
        sys.exit()

    # read images
    [X,y] = read_images(sys.argv[1])
    # compute the eigenfaces model
    model = EigenfacesModel(X[1:], y[1:])
    # get a prediction for the first observation
    print "expected =", y[0], "/", "predicted =", model.predict(X[0])
```

2.3 Fisherfaces

The Linear Discriminant Analysis was invented by the great statistician Sir R. A. Fisher, who successfully used it for classifying flowers in his 1936 paper *The use of multiple measurements in taxonomic problems* [8]. But why do we need another dimensionality reduction method, if the Principal Component Analysis (PCA) did such a good job?

The PCA finds a linear combination of features that maximizes the total variance in data. While this is clearly a powerful way to represuccsent data, it doesn't consider any classes and so a lot of discriminative information may be lost when throwing components away. Imagine a situation where the variance is generated by an external source, let it be the light. The components identified by a PCA do not necessarily contain any discriminative information at all, so the projected samples are smeared together and a classification becomes impossible.

In order to find the combination of features that separates best between classes the Linear Discriminant Analysis maximizes the ratio of between-classes to within-classes scatter. The idea is simple: same classes should cluster tightly together, while different classes are as far away as possible from each other. This was also recognized by Belhumeur, Hespanha and Kriegman and so they applied a Discriminant Analysis to face recognition in [3].

2.3.1 Algorithmic Description

Let X be a random vector with samples drawn from c classes:

$$X = \{X_1, X_2, \ldots, X_c\} \tag{8}$$

$$X_i = \{x_1, x_2, \ldots, x_n\} \tag{9}$$

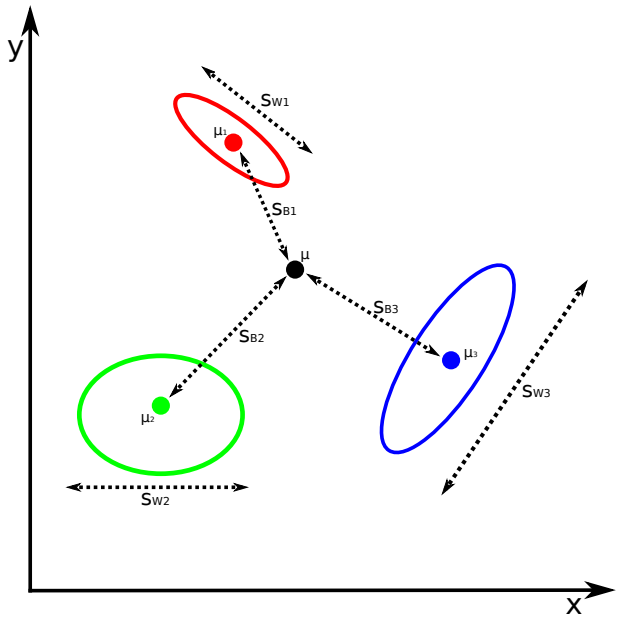


Figure 1: This figure shows the scatter matrices S_B and S_W for a 3 class problem. μ represents the total mean and $[\mu_1, \mu_2, \mu_3]$ are the class means.

The scatter matrices S_B and S_W are calculated as:

$$S_B = \sum_{i=1}^{c} N_i(\mu_i - \mu)(\mu_i - \mu)^T \tag{10}$$

$$S_W = \sum_{i=1}^{c} \sum_{x_j \in X_i} (x_j - \mu_i)(x_j - \mu_i)^T \tag{11}$$

, where μ is the total mean:

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{12}$$

And μ_i is the mean of class $i \in \{1, \ldots, c\}$:

$$\mu_i = \frac{1}{|X_i|} \sum_{x_j \in X_i} x_j \tag{13}$$

Fisher's classic algorithm now looks for a projection W, that maximizes the class separability criterion:

$$W_{opt} = \arg\max_W \frac{|W^T S_B W|}{|W^T S_W W|} \tag{14}$$

Following [3], a solution for this optimization problem is given by solving the General Eigenvalue Problem:

$$\begin{aligned} S_B v_i &= \lambda_i S_w v_i \\ S_W^{-1} S_B v_i &= \lambda_i v_i \end{aligned} \tag{15}$$

There's one problem left to solve: The rank of S_W is at most $(N-c)$, with N samples and c classes. In pattern recognition problems the number of samples N is almost always samller than the dimension of the input data (the number of pixels), so the scatter matrix S_W becomes singular (see [2]). In [3] this was solved by performing a Principal Component Analysis on the data and projecting the

samples into the $(N-c)$-dimensional space. A Linear Discriminant Analysis was then performed on the reduced data, because S_W isn't singular anymore.
The optimization problem can be rewritten as:

$$W_{pca} = \arg\max_W |W^T S_T W| \tag{16}$$

$$W_{fld} = \arg\max_W \frac{|W^T W_{pca}^T S_B W_{pca} W|}{|W^T W_{pca}^T S_W W_{pca} W|} \tag{17}$$

The transformation matrix W, that projects a sample into the $(c-1)$-dimensional space is then given by:

$$W = W_{fld}^T W_{pca}^T \tag{18}$$

One final note: Although S_W and S_B are symmetric matrices, the product of two symmetric matrices is not necessarily symmetric. so you have to use an eigenvalue solver for general matrices. OpenCV's cv::eigen only works for symmetric matrices in its current version; since eigenvalues and singular values aren't equivalent for non-symmetric matrices you can't use a Singular Value Decomposition (SVD) either.

2.3.2 Fisherfaces in Python

Translating the Linear Discriminant Analysis to Python is almost trivial again, see Listing 16. For projecting and reconstructing from the basis you can use the functions from Listing 5 and 6.

Listing 16: src/py/tinyfacerec/subspace.py

```
def lda(X, y, num_components=0):
  y = np.asarray(y)
  [n,d] = X.shape
  c = np.unique(y)
  if (num_components <= 0) or (num_component>(len(c)-1)):
    num_components = (len(c)-1)
  meanTotal = X.mean(axis=0)
  Sw = np.zeros((d, d), dtype=np.float32)
  Sb = np.zeros((d, d), dtype=np.float32)
  for i in c:
    Xi = X[np.where(y==i)[0],:]
    meanClass = Xi.mean(axis=0)
    Sw = Sw + np.dot((Xi-meanClass).T, (Xi-meanClass))
    Sb = Sb + n * np.dot((meanClass - meanTotal).T, (meanClass - meanTotal))
  eigenvalues, eigenvectors = np.linalg.eig(np.linalg.inv(Sw)*Sb)
  idx = np.argsort(-eigenvalues.real)
  eigenvalues, eigenvectors = eigenvalues[idx], eigenvectors[:,idx]
  eigenvalues = np.array(eigenvalues[0:num_components].real, dtype=np.float32, copy=
      True)
  eigenvectors = np.array(eigenvectors[0:,0:num_components].real, dtype=np.float32,
      copy=True)
  return [eigenvalues, eigenvectors]
```

The functions to perform a PCA (Listing 4) and LDA (Listing 16) are now defined, so we can go ahead and implement the Fisherfaces from Equation 18.

Listing 17: src/py/tinyfacerec/subspace.py

```
def fisherfaces(X,y,num_components=0):
  y = np.asarray(y)
  [n,d] = X.shape
  c = len(np.unique(y))
  [eigenvalues_pca, eigenvectors_pca, mu_pca] = pca(X, y, (n-c))
  [eigenvalues_lda, eigenvectors_lda] = lda(project(eigenvectors_pca, X, mu_pca), y,
      num_components)
  eigenvectors = np.dot(eigenvectors_pca,eigenvectors_lda)
  return [eigenvalues_lda, eigenvectors, mu_pca]
```

For this example I am going to use the Yale Facedatabase A, just because the plots are nicer. Each Fisherface has the same length as an original image, thus it can be displayed as an image. We'll again load the data, learn the Fisherfaces and make a subplot of the first 16 Fisherfaces.

Listing 18: src/py/scripts/example_fisherfaces.py

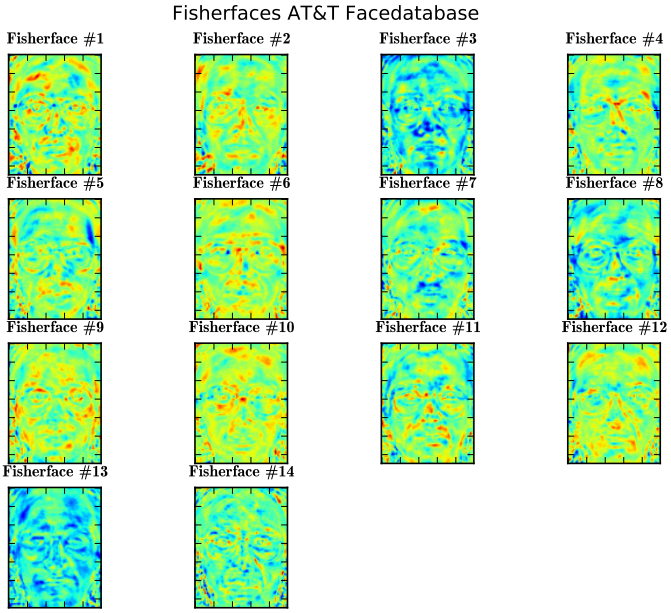
```
import sys
# append tinyfacerec to module search path
sys.path.append("..")
# import numpy and matplotlib colormaps
import numpy as np
# import tinyfacerec modules
from tinyfacerec.subspace import fisherfaces
from tinyfacerec.util import normalize, asRowMatrix, read_images
from tinyfacerec.visual import subplot

if __name__ == '__main__':

    if len(sys.argv) != 2:
        print "USAGE: example_fisherfaces.py </path/to/images>"
        sys.exit()

    # read images
    [X,y] = read_images(sys.argv[1])
    # perform a full pca
    [D, W, mu] = fisherfaces(asRowMatrix(X), y)
    #import colormaps
    import matplotlib.cm as cm
    # turn the first (at most) 16 eigenvectors into grayscale
```

The Fisherfaces method learns a class-specific transformation matrix, so the they do not capture illumination as obviously as the Eigenfaces method. The Discriminant Analysis instead finds the facial features to discriminate between the persons. It's important to mention, that the performance of the Fisherfaces heavily depends on the input data as well. Practically said: if you learn the Fisherfaces for well-illuminated pictures only and you try to recognize faces in bad-illuminated scenes, then method is likely to find the wrong components (just because those features may not be predominant on bad illuminated images). This is somewhat logical, since the method had no chance to learn the illumination.



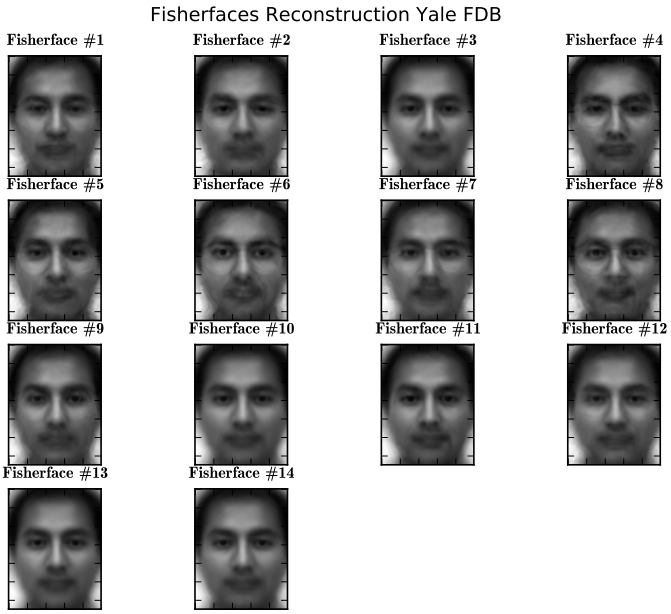
The Fisherfaces allow a reconstruction of the projected image, just like the Eigenfaces did. But since we only identified the features to distinguish between subjects, you can't expect a nice approximation of the original image. We can rewrite Listing 11 for the Fisherfaces method into Listing 19, but this time we'll project the sample image onto each of the Fisherfaces instead. So you'll have a visualization, which features each Fisherface describes.

Listing 19: src/py/scripts/example_fisherfaces.py

```
    E = []
    for i in xrange(min(W.shape[1], 16)):
      e = W[:,i].reshape(X[0].shape)
      E.append(normalize(e,0,255))
    # plot them and store the plot to "python_fisherfaces_fisherfaces.pdf"
    subplot(title="Fisherfaces AT&T Facedatabase", images=E, rows=4, cols=4, sptitle="
        Fisherface", colormap=cm.jet, filename="python_fisherfaces_fisherfaces.png")

    from tinyfacerec.subspace import project, reconstruct

    E = []
    for i in xrange(min(W.shape[1], 16)):
      e = W[:,i].reshape(-1,1)
```



The implementation details are not repeated in this section. For the Fisherfaces method a similar model to the `EigenfacesModel` in Listing 20 must be defined.

Listing 20: src/py/tinyfacerec/model.py

```
class FisherfacesModel(BaseModel):

  def __init__(self, X=None, y=None, dist_metric=EuclideanDistance(), num_components
      =0):
    super(FisherfacesModel, self).__init__(X=X,y=y,dist_metric=dist_metric,
        num_components=num_components)

  def compute(self, X, y):
    [D, self.W, self.mu] = fisherfaces(asRowMatrix(X),y, self.num_components)
    # store labels
    self.y = y
    # store projections
    for xi in X:
      self.projections.append(project(self.W, xi.reshape(1,-1), self.mu))
```

Once the `FisherfacesModel` is defined, it can be used to learn the Fisherfaces and generate predictions. In the following Listing 21 we'll load the Yale Facedatabase A and perform a prediction on the first image.

Listing 21: src/py/scripts/example_model_fisherfaces.py

```
import sys
# append tinyfacerec to module search path
sys.path.append("..")
# import numpy and matplotlib colormaps
import numpy as np
# import tinyfacerec modules
from tinyfacerec.util import read_images
from tinyfacerec.model import FisherfacesModel

if __name__ == '__main__':

    if len(sys.argv) != 2:
        print "USAGE: example_model_fisherfaces.py </path/to/images>"
        sys.exit()

    # read images
    [X,y] = read_images(sys.argv[1])
    # compute the eigenfaces model
    model = FisherfacesModel(X[1:], y[1:])
    # get a prediction for the first observation
    print "expected =", y[0], "/", "predicted =", model.predict(X[0])
```

3 Conclusion

This document explained and implemented the Eigenfaces [13] and the Fisherfaces [3] method with GNU Octave/MATLAB, Python. It gave you some ideas to get started and research this highly active topic. I hope you had fun reading and I hope you think cv::FaceRecognizer is a useful addition to OpenCV.
More maybe here:

- http://www.opencv.org
- http://www.bytefish.de/blog
- http://www.github.com/bytefish

References

[1] Ahonen, T., Hadid, A., and Pietikainen, M. Face Recognition with Local Binary Patterns. *Computer Vision - ECCV 2004* (2004), 469–481.

[2] A.K. Jain, S. J. R. Small sample size effects in statistical pattern recognition: Recommendations for practitioners. *IEEE Transactions on Pattern Analysis and Machine Intelligence 13*, 3 (1991), 252–264.

[3] Belhumeur, P. N., Hespanha, J., and Kriegman, D. Eigenfaces vs. fisherfaces: Recognition using class specific linear projection. *IEEE Transactions on Pattern Analysis and Machine Intelligence 19*, 7 (1997), 711–720.

[4] Brunelli, R., and Poggio, T. Face recognition through geometrical features. In *European Conference on Computer Vision (ECCV)* (1992), pp. 792–800.

[5] Cardinaux, F., Sanderson, C., and Bengio, S. User authentication via adapted statistical models of face images. *IEEE Transactions on Signal Processing 54* (January 2006), 361–373.

[6] Chiara Turati, Viola Macchi Cassia, F. S., and Leo, I. Newborns face recognition: Role of inner and outer facial features. *Child Development 77*, 2 (2006), 297–311.

[7] Duda, R. O., Hart, P. E., and Stork, D. G. *Pattern Classification (2nd Edition)*, 2 ed. November 2001.

[8] Fisher, R. A. The use of multiple measurements in taxonomic problems. *Annals Eugen. 7* (1936), 179–188.

[9] Kanade, T. *Picture processing system by computer complex and recognition of human faces.* PhD thesis, Kyoto University, November 1973.

[10] Lee, K.-C., Ho, J., and Kriegman, D. Acquiring linear subspaces for face recognition under variable lighting. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI) 27*, 5 (2005).

[11] Maturana, D., Mery, D., and Soto, A. Face recognition with local binary patterns, spatial pyramid histograms and naive bayes nearest neighbor classification. *2009 International Conference of the Chilean Computer Science Society (SCCC)* (2009), 125–132.

[12] Rodriguez, Y. *Face Detection and Verification using Local Binary Patterns.* PhD thesis, École Polytechnique Fédérale De Lausanne, October 2006.

[13] Turk, M., and Pentland, A. Eigenfaces for recognition. *Journal of Cognitive Neuroscience 3* (1991), 71–86.

[14] Wiskott, L., Fellous, J.-M., Krüger, N., and Malsburg, C. V. D. Face recognition by elastic bunch graph matching. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE 19* (1997), 775–779.

[15] Zhao, W., Chellappa, R., Phillips, P., and Rosenfeld, A. Face recognition: A literature survey. *Acm Computing Surveys (CSUR) 35*, 4 (2003), 399–458.