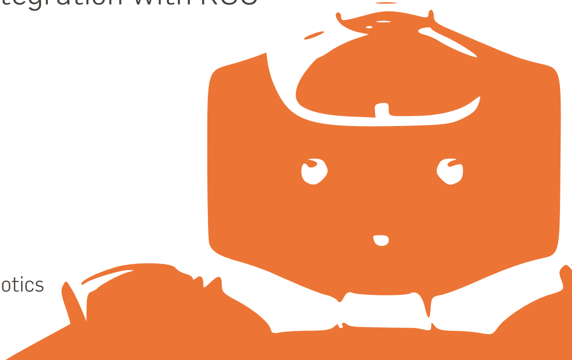
ROBOTICS
WITH
PLYMOUTH
UNIVERSITY

ROCO222
Intro to Sensors and Actuators

Face recognition and Integration with ROS



Séverin Lemaignan

Centre for Neural Systems and Robotics
Plymouth University

FACE RECOGNITION: PRINCIPAL COMPONENT ANALYSIS

PRINCIPAL COMPONENT ANALYSIS

Principal Component Analysis (PCA) is a technique to find the sources of variance in a dataset.

[*Source: Mark Richardson, Principal Component Analysis*]

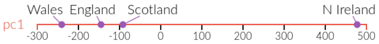
PRINCIPAL COMPONENT ANALYSIS

	England	N Ireland	Scotland	Wales
Alcoholic drinks	375	135	458	475
Beverages	57	47	53	73
Carcase meat	245	267	242	227
Cereals	1472	1494	1462	1582
Cheese	105	66	103	103
Confectionery	54	41	62	64
Fats and oils	193	209	184	235
Fish	147	93	122	160
Fresh fruit	1102	674	957	1137
Fresh potatoes	720	1033	566	874
Fresh Veg	253	143	171	265
Other meat	685	586	750	803
Other Veg	488	355	418	570
Processed potatoes	198	187	220	203
Processed Veg	360	334	337	365
Soft drinks	1374	1506	1572	1256
Sugars	156	139	147	175

[*Source: Mark Richardson, Principal Component Analysis*]

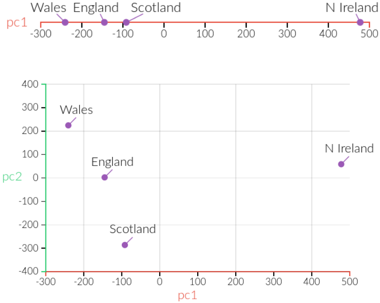
PRINCIPAL COMPONENT ANALYSIS

	England	N Ireland	Scotland	Wales
Alcoholic drinks	375	135	458	475
Beverages	57	47	53	73
Carcase meat	245	267	242	227
Cereals	1472	1494	1462	1582
Cheese	105	66	103	103
Confectionery	54	41	62	64
Fats and oils	193	209	184	235
Fish	147	93	122	160
Fresh fruit	1102	674	957	1137
Fresh potatoes	720	1033	566	874
Fresh Veg	253	143	171	265
Other meat	685	586	750	803
Other Veg	488	355	418	570
Processed potatoes	198	187	220	203
Processed Veg	360	334	337	365
Soft drinks	1374	1506	1572	1256
Sugars	156	139	147	175



PRINCIPAL COMPONENT ANALYSIS

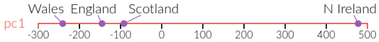
	England	N Ireland	Scotland	Wales
Alcoholic drinks	375	135	458	475
Beverages	57	47	53	73
Carcase meat	245	267	242	227
Cereals	1472	1494	1462	1582
Cheese	105	66	103	103
Confectionery	54	41	62	64
Fats and oils	193	209	184	235
Fish	147	93	122	160
Fresh fruit	1102	674	957	1137
Fresh potatoes	720	1033	566	874
Fresh Veg	253	143	171	265
Other meat	685	586	750	803
Other Veg	488	355	418	570
Processed potatoes	198	187	220	203
Processed Veg	360	334	337	365
Soft drinks	1374	1506	1572	1256
Sugars	156	139	147	175

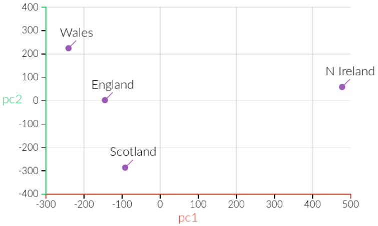


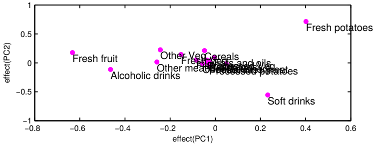
[*Source: Mark Richardson, Principal Component Analysis*]

PRINCIPAL COMPONENT ANALYSIS

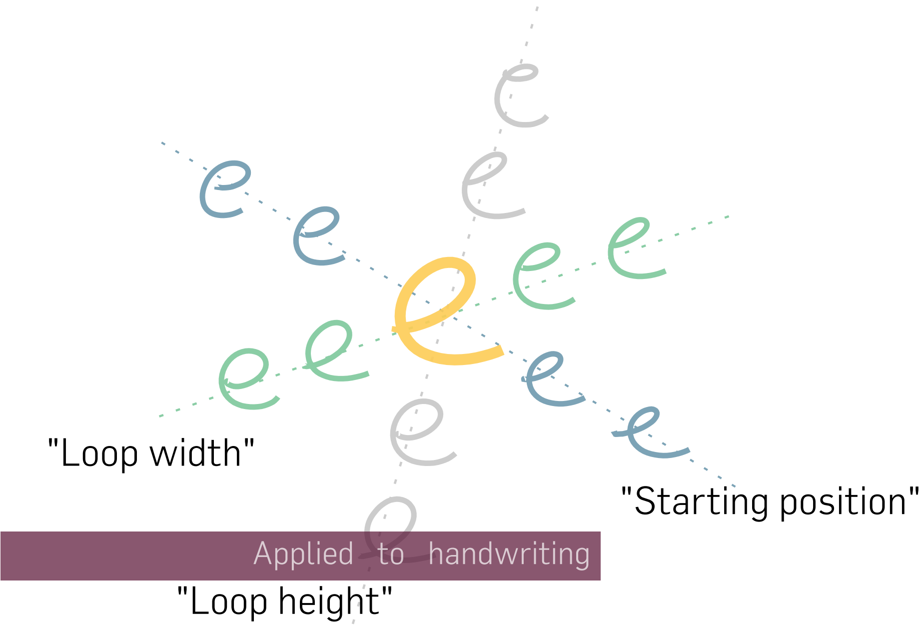
	England	N Ireland	Scotland	Wales
Alcoholic drinks	375	135	458	475
Beverages	57	47	53	73
Carcase meat	245	267	242	227
Cereals	1472	1494	1462	1582
Cheese	105	66	103	103
Confectionery	54	41	62	64
Fats and oils	193	209	184	235
Fish	147	93	122	160
Fresh fruit	1102	674	957	1137
Fresh potatoes	720	1033	566	874
Fresh Veg	253	143	171	265
Other meat	685	586	750	803
Other Veg	488	355	418	570
Processed potatoes	198	187	220	203
Processed Veg	360	334	337	365
Soft drinks	1374	1506	1572	1256
Sugars	156	139	147	175







[*Source: Mark Richardson, Principal Component Analysis*]


"Loop width"
"Starting position"
Applied to handwriting
"Loop height"

Getting the NAO to write

Writing letters badly

Learning to write well

AT&T Face dataset

Applied to faces

PCA ALGORITHM

Let $\mathbf{X} = \{\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n\}$ be a vector with observations $\mathbf{x}_i \in \mathbb{R}^d$.

1. Compute the mean μ

$$\mu = \frac{1}{n} \sum_{i=1}^{n} \mathbf{x}_i$$

2. Compute the the Covariance Matrix S

$$S = \frac{1}{n} \sum_{i=1}^{n} (\mathbf{x}_i - \mu)(\mathbf{x}_i - \mu)^T$$

3. Compute the eigenvalues λ_i and eigenvectors $\mathbf{v}_i$ of S

$$S \cdot \mathbf{v}_i = \lambda_i \mathbf{v}_i \quad \text{with } i = 1, 2, \ldots, n$$

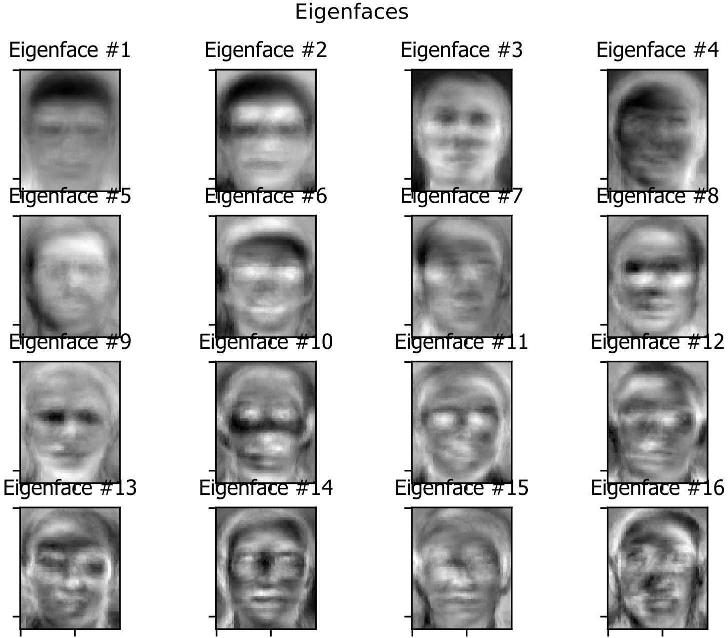
4. Order the eigenvectors descending by their eigenvalue. The k principal components are the eigenvectors corresponding to the k largest eigenvalues.

[*Source: Philipp Wagner, Face Recognition with Python* (code source available as well)]

PYTHON CODE

```
def pca(X):

    mu = X.mean(axis=0)
    X = X - mu
    C = np.dot(X.T,X)
    eigenvalues, eigenvectors = np.linalg.eigh(C)

    # sort eigenvectors descending by their eigenvalue
    idx = np.argsort(-eigenvalues)
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:,idx]
    return eigenvalues, eigenvectors, mu

# D: eigenvalues, W: eigenvectors, mu: mean, X: 40 X 10304 image array
D, W, mu = pca(X)

# plot the first 16 'eigenfaces'
images = []
for i in range(16):
    image = W[:,i].reshape(X[0].shape)
    images.append(normalize(image,0,255))

subplot(title="Eigenfaces", images=images, rows=4, cols=4)
```

AT&T Face dataset


Eigenfaces
Eigenface #1
Eigenface #2
Eigenface #3
Eigenface #4
Eigenface #5
Eigenface #6
Eigenface #7
Eigenface #8
Eigenface #9
Eigenface #10
Eigenface #11
Eigenface #12
Eigenface #13
Eigenface #14
Eigenface #15
Eigenface #16

PCA PROJECTION AND RECONSTRUCTION

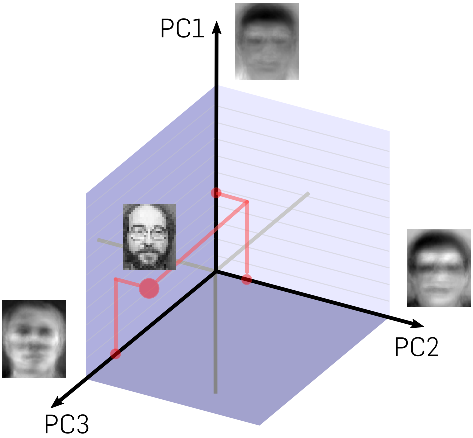
The k principal components of an observed vector $\mathbf{x}$ are then given by:

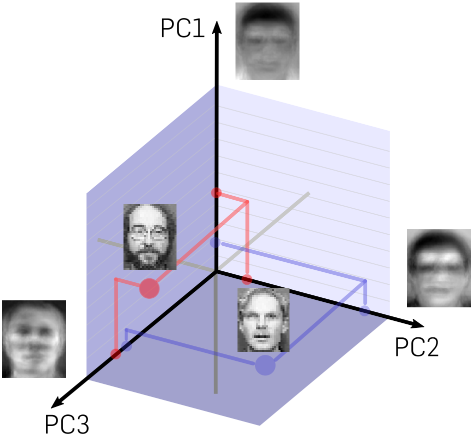
The image of a face!

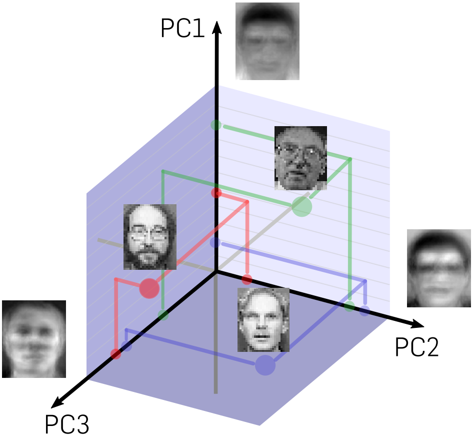
$$\mathbf{y} = W^T(\mathbf{x} - \mu)$$

where $W = (\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k)$.

The PCA basis


PC1
PC2
PC3


PC1
PC2
PC3


PC1
PC2
PC3

PCA PROJECTION AND RECONSTRUCTION

The k principal components of an observed vector $\mathbf{x}$ are then given by:

The image of a face!

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

where $W = (\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k)$.

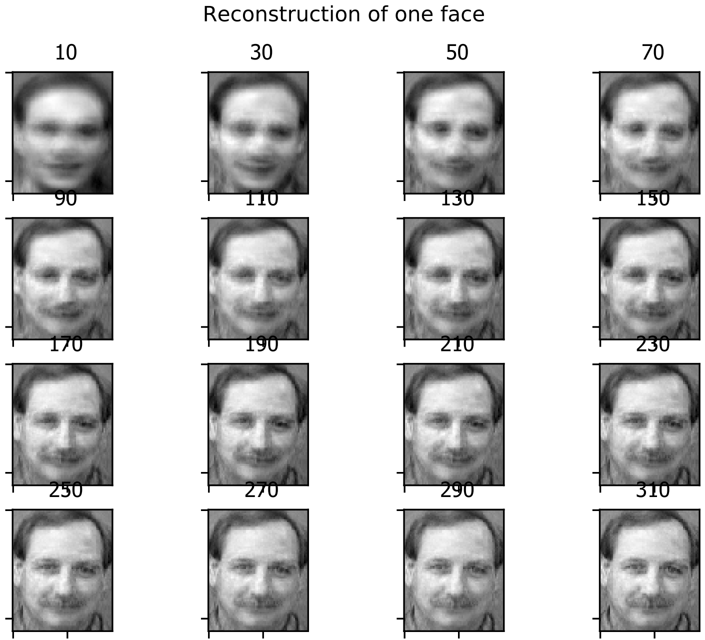
The PCA basis

The reconstruction from the PCA basis is given by:

$$\mathbf{x} = W \cdot \mathbf{y} + \mu$$

PYTHON CODE

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

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

images = []
for nb_evs in range(10, 310, 20):
    P = project(W[:,0:nb_evs], X[0].reshape(1,-1), mu)
    R = reconstruct(W[:,0:nb_evs], P, mu)

    R = R.reshape(X[0].shape)
    images.append(normalize(R,0,255))

subplot(title="Reconstruction of one face", images=images, rows=4, cols=4)
```


Reconstruction of one face
10
30
50
70
90
110
130
150
170
190
210
230
250
270
290
310

WHY IS IT USEFUL?

Original images: $\textit{dim}(\mathbf{x}) = 92 \times 112 = 10304$ pixels: large number of dimensions!

$\Rightarrow$ difficult to tell whether 2 images represent the same person (i.e. ***classify*** them).

WHY IS IT USEFUL?

Original images: $dim(\mathbf{x}) = 92 \times 112 = 10304$ pixels: large number of dimensions!

$\Rightarrow$ difficult to tell whether 2 images represent the same person (i.e. ***classify*** them).

With the PCA, we project our test image onto a PCA basis of k principal components: $\mathbf{y} = W^T(\mathbf{x} - \mu)$ with $W = (\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k)$.

$dim(\mathbf{y}) = k$ is much smaller than $dim(\mathbf{x})$

WHY IS IT USEFUL?

Original images: $dim(\mathbf{x}) = 92 \times 112 = 10304$ pixels: large number of dimensions!

$\Rightarrow$ difficult to tell whether 2 images represent the same person (i.e. ***classify*** them).

With the PCA, we project our test image onto a PCA basis of k principal components: $\mathbf{y} = W^T(\mathbf{x} - \mu)$ with $W = (\mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k)$.

$dim(\mathbf{y}) = k$ is much smaller than $dim(\mathbf{x})$

We effectively "summarize" our image into a few key values, along the principal axes of variation of our dataset.

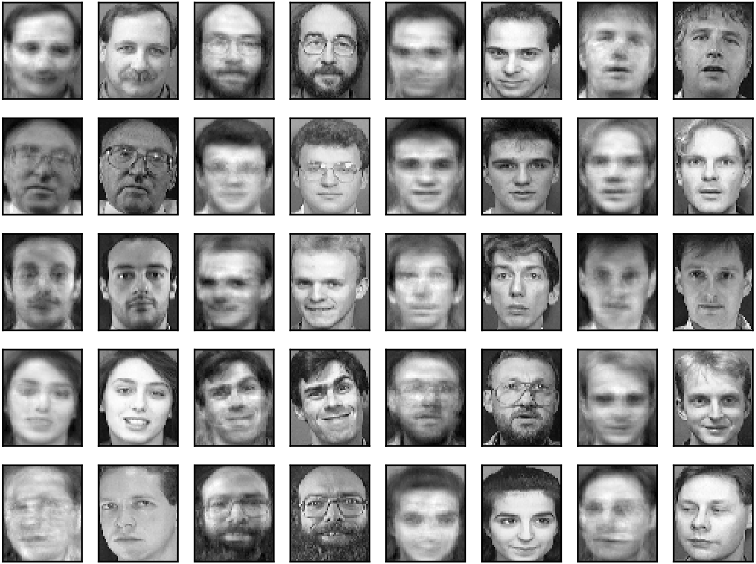
$\Rightarrow$ these values discriminate effectively amongst our images

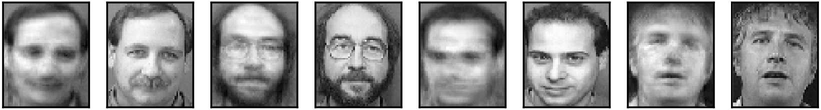
$\Rightarrow$ **Well suited for classification!**

Reconstruction with 1 Eigenvectors

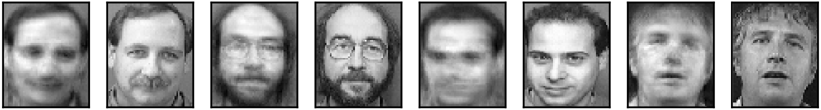
Reconstruction with 10 Eigenvectors

Reconstruction with 50 Eigenvectors





Remember: these faces are reconstructed from 50 values (to be compared to the 10304 values required for the original photos).

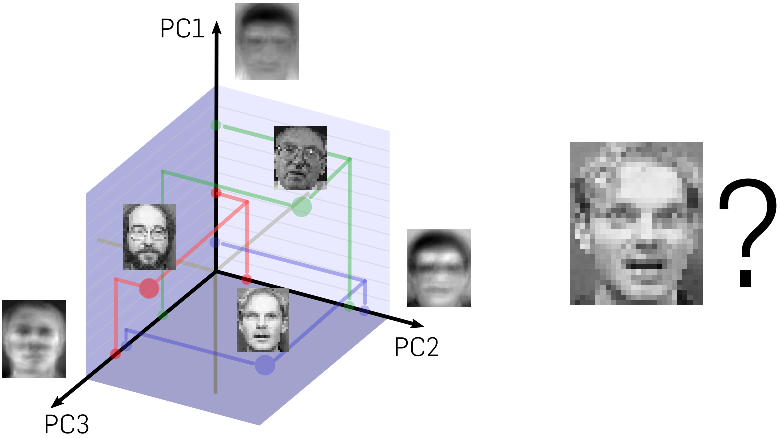


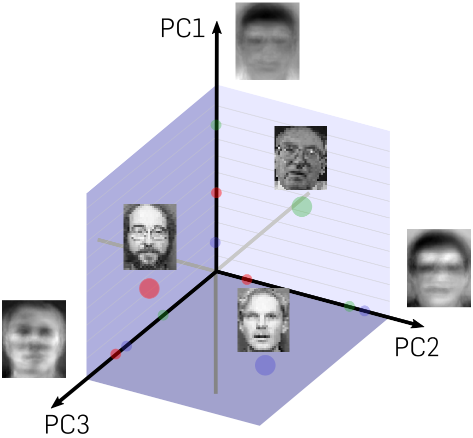
Remember: these faces are reconstructed from 50 values (to be compared to the 10304 values required for the original photos).

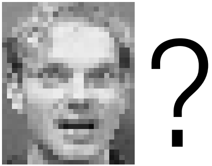
PCA is often used as a **dimensionality reduction** technique (i.e. a kind of data lossy data compression).

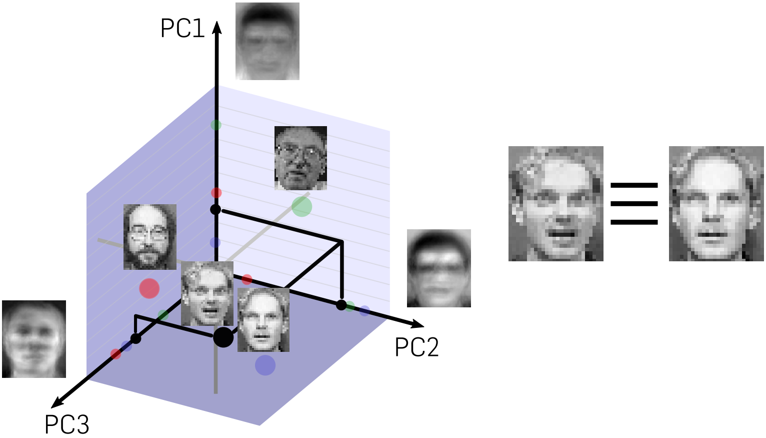
FACE RECOGNITION

 ?


PC1
PC2
PC3
?


PC1
PC2
PC3


?


PC1
PC2
PC3

RECOGNITION

1. **learn a model** by projecting the training set onto the PCA basis
2. **project the test image** as well
3. **find the 1-nearest neighbour**

PYTHON CODE

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


def learn_model(X):
    D, W, mu = pca(X, nb_evs=10)
    # compute projections
    projections = []
    for xi in X:
        yi = project(W,
                     xi.reshape(1,-1),
                     mu)
        projections.append(yi)

    return W, projections
```

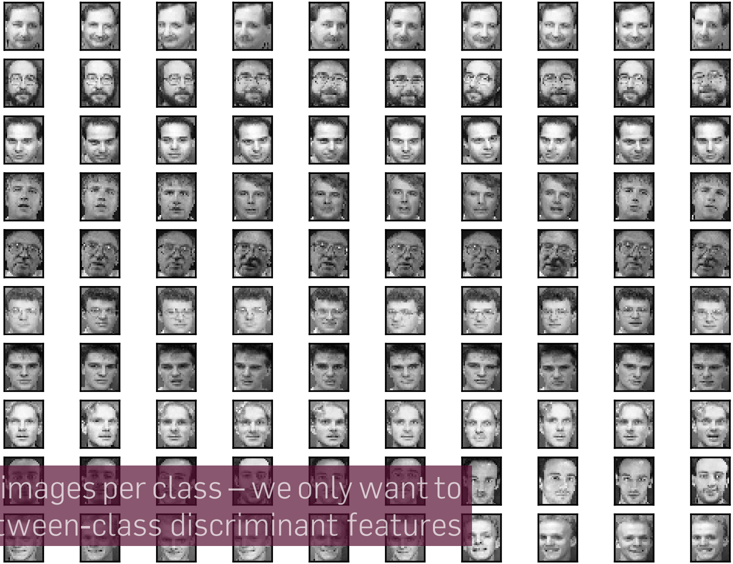
```
def predict(X, W, projections):
    minDist = np.finfo('float').max
    minClass = -1
    Q = project(W, X.reshape(1,-1), mu)

    for i in range(len(projections)):
        dist = dist(projections[i], Q)
        if dist < minDist:
            minDist = dist
            faceClass = faceClasses[i]
    return faceClass


X, faceClasses = read_images()
W, projections = learn_model(X)
predict(test_image, W, projections)
```

LIMITS OF THE PCA APPROACH (EIGENFACES)

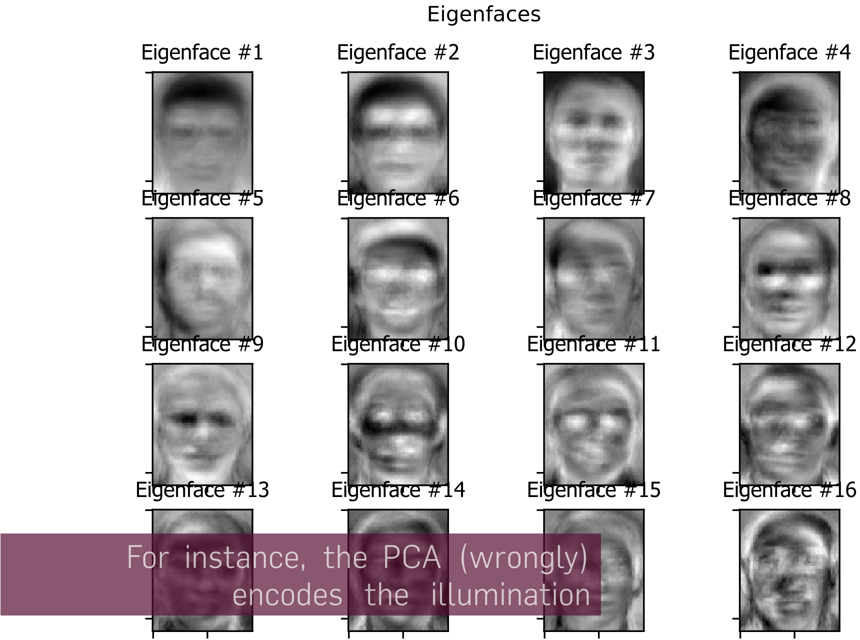
PCA tries to find a combination of linear features that maximizes the total variance (i.e. the “axes of maximum variation”).

No concept of class!

AT&T Face dataset



Here, 10 images per class – we only want to learn between-class discriminant features


Eigenfaces
Eigenface #1
Eigenface #2
Eigenface #3
Eigenface #4
Eigenface #5
Eigenface #6
Eigenface #7
Eigenface #8
Eigenface #9
Eigenface #10
Eigenface #11
Eigenface #12
Eigenface #13
Eigenface #14
Eigenface #15
Eigenface #16
For instance, the PCA (wrongly) encodes the illumination

LIMITS OF THE PCA APPROACH (EIGENFACES)

$\Rightarrow$ Linear Discriminant Analysis (LDA) (and the corresponding *Fischerfaces*)

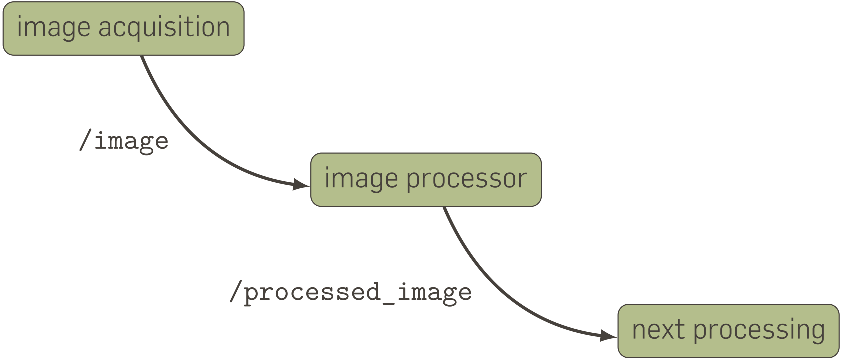
LDA tries to find a combination of linear features that maximizes the ratio of between-classes to within-classes scatter.

[*Source: Belhumeur, Hespanha and Kriegman, Eigenfaces vs. fisherfaces: Recognition using class specific linear projection*]

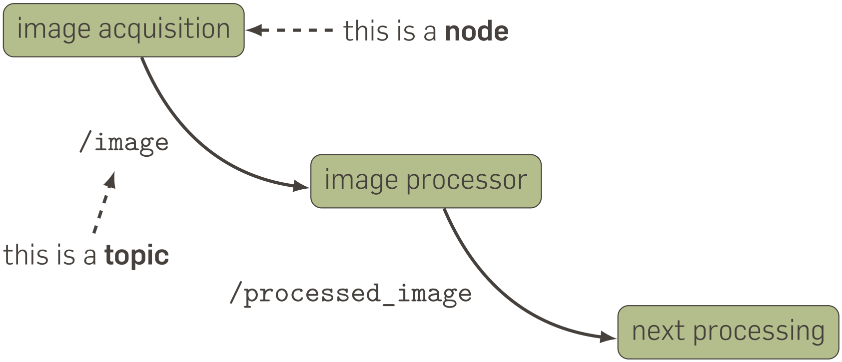
10 min break

HOW TO INTEGRATE THAT WITH ROS?

REMINDER: A SIMPLE IMAGE PROCESSING PIPELINE



REMINDER: A SIMPLE IMAGE PROCESSING PIPELINE



```python
import sys, cv2, rospy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge

def on_image(image):
    cv_image = bridge.imgmsg_to_cv2(image, "bgr8")
    rows, cols, channels = cv_image.shape
    cv2.circle(cv_image, (cols/2, rows/2), 50, (0,0,255), -1)
    image_pub.publish(bridge.cv2_to_imgmsg(cv_image, "bgr8"))

rospy.init_node('image_processor')
bridge = CvBridge()
image_sub = rospy.Subscriber("image",Image, on_image)
image_pub = rospy.Publisher("processed_image",Image)

while not rospy.is_shutdown():
    rospy.spin()
```

HOW TO USE THIS CODE?

First, we need to write data onto the `/image` topic, for instance from a webcam:

```
> rosrun usb_cam usb_cam_node
```

HOW TO USE THIS CODE?

First, we need to write data onto the `/image` topic, for instance from a webcam:

```
> rosrun usb_cam usb_cam_node
```

Then, we run our code:

```
> python image_processor.py image:=/usb_cam/image_raw
```

HOW TO USE THIS CODE?

First, we need to write data onto the `/image` topic, for instance from a webcam:

```
> rosrun usb_cam usb_cam_node
```

Then, we run our code:

```
> python image_processor.py image:=/usb_cam/image_raw
```

Finally, we run a 3rd node to display the image:

```
> rqt_image_view image:=/processed_image
```

CREATING THE FACEREC ROS PACKAGE

LET'S CREATE A PROPER ROS PACKAGE

```
> cd $HOME
> mkdir src && cd src
> catkin_create_pkg facerec rospy
```

LET'S CREATE A PROPER ROS PACKAGE

```
> cd $HOME
> mkdir src && cd src
> catkin_create_pkg facerec rospy
```

```
> ls facerec
CMakeLists.txt  package.xml  src
```

FIRST, ADD SOME CODE

```
> cd facerec
> mkdir -p src/facerec && cd src/facerec
> touch __init__.py # required to create a Python module
> gedit recognition.py
```

FIRST, ADD SOME CODE

```
> cd facerec
> mkdir -p src/facerec && cd src/facerec
> touch __init__.py # required to create a Python module
> gedit recognition.py
```

Just a simple stub for a Python module:

```python
def run(dataset):
    print('Dataset: ' + dataset)
```

FIRST, ADD SOME INITIAL CODE

Create as well an executable (our future ROS node) in `scripts/`:

```
> cd ../..
> mkdir -p scripts && cd scripts
> gedit reco
```

FIRST, ADD SOME INITIAL CODE

Create as well an executable (our future ROS node) in `scripts/`:

```
> cd ../..
> mkdir -p scripts && cd scripts
> gedit reco
```

```python
#! /usr/bin/env python

import facerec.reco

if __name__ == '__main__':
    facerec.reco.run("my_faces")
```

FIRST, ADD SOME INITIAL CODE

Create as well an executable (our future ROS node) in `scripts/`:

```
> cd ../..
> mkdir -p scripts && cd scripts
> gedit reco
```

```python
#! /usr/bin/env python

import facerec.reco

if __name__ == '__main__':
    facerec.reco.run("my_faces")
```

```
> chmod +x reco
```

CONFIGURE THE PYTHON 'BUILD'

Because our node is written in Python, our `CMakeLists.txt` is simple:

```
cmake_minimum_required(VERSION 2.8.3)
project(facerec)

find_package(catkin REQUIRED COMPONENTS
  rospy
)

catkin_python_setup()
catkin_package()

install(PROGRAMS
   scripts/reco
   DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION}
)
```

CONFIGURE THE PYTHON 'BUILD'

However, we need a `setup.py` (standard Python `distutils`-based packaging):

```
from distutils.core import setup
from catkin_pkg.python_setup import generate_distutils_setup

# fetch values from package.xml
setup_args = generate_distutils_setup(
                    packages=['facerec'],
                    package_dir={'': 'src'},
                )

setup(**setup_args)
```

INSTALL THE NODE

We can now install our node:

```
> cd ..
> mkdir -p build && cd build
> cmake -DCMAKE_INSTALL_PREFIX=<install prefix> ..
> make install
```

INSTALL THE NODE

We can now install our node:

```
> cd ..
> mkdir -p build && cd build
> cmake -DCMAKE_INSTALL_PREFIX=<install prefix> ..
> make install
```

Assuming ROS is correctly installed, we can run our node:

```
> export ROS_PACKAGE_PATH=<prefix>/share:$ROS_PACKAGE_PATH
> rosrun facerec reco
Dataset: my_faces
```

IMAGE PROCESSING

Let's update the node `reco` and the library (Python *module*) `recognition.py` to perform simple image processing:

`recognition.py`:

```
import cv2

def run(image):
    rows, cols, channels = image.shape
    cv2.circle(image, (cols/2, rows/2), 50, (0,0,255), -1)
```

IMAGE PROCESSING

`reco`:

```
#! /usr/bin/env python

import sys, rospy
from sensor_msgs.msg import Image
from cv_bridge import CvBridge

import facerec.reco

def on_image(image):
    cv_image = bridge.imgmsg_to_cv2(image, "bgr8")
    facerec.reco.run(cv_image)
    image_pub.publish(bridge.cv2_to_imgmsg(cv_image, "bgr8"))

if __name__ == '__main__':
    rospy.init_node('image_processor')
    bridge = CvBridge()
    image_sub = rospy.Subscriber("image",Image, on_image)
    image_pub = rospy.Publisher("processed_image",Image, queue_size=1)

    while not rospy.is_shutdown():
        rospy.spin()
```

TO USE THE NODE

```
> rosrun usb_cam usb_cam_node
```

```
> rosrun facerec reco image:=/usb_cam/image_raw
```

```
> rqt_image_view image:=/processed_image
```

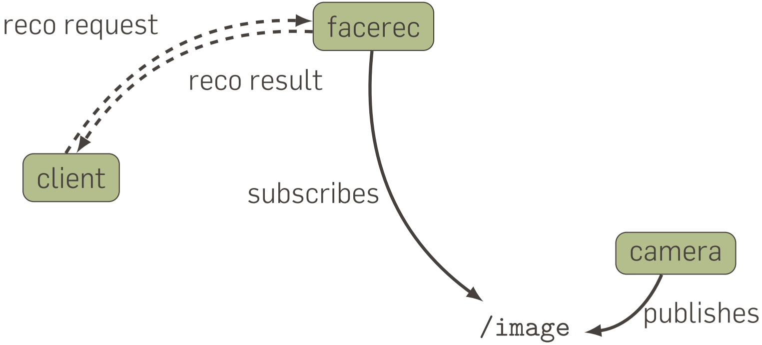
Let's try it!

WHAT ARE THE NEXT STEPS FOR FACE RECOGNITION?

We need to:

- acquire reference images for each of the face we want to recognise
- re-train the model every time we add new faces to the dataset
- when requested, attempt to recognise the person $\rightarrow$ ROS action

POSSIBLE NETWORK



RECOGNITION LOGIC

Inside `facerec`:

- when incoming request, attempt recognition
- if recognition fails: acquire a couple of images of that person; create a new class; re-train
- if recognition succeeds: add image to corresponding class; re-train
- if unsure: ask for confirmation

RECOGNITION LOGIC

Inside `facerec`:

- when incoming request, attempt recognition
- if recognition fails: acquire a couple of images of that person; create a new class; re-train
- if recognition succeeds: add image to corresponding class; re-train
- if unsure: ask for confirmation

Implementation left as an exercice!

That's all, folks!

Questions:
Portland Square B316 or **severin.lemaignan@plymouth.ac.uk**

Slides:
github.com/severin-lemaignan/module-introduction-sensors-actuators