Decomposition

March 7, 2025

[1]:
```
# # This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')
#
# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cse455/assignments/assignment4/'
# FOLDERNAME = None
# assert FOLDERNAME is not None, "[!] Enter the foldername."
#
# # Now that we've mounted your Drive, this ensures that
# # the Python interpreter of the Colab VM can load
# # python files from within it.
# import sys
# sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))
#
# # This downloads the COCO dataset to your Drive
# # if it doesn't already exist.
# %cd /content/drive/My\ Drive/$FOLDERNAME/cse455/datasets/
# !bash get_datasets.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME
```

1 Assignment 4 Part 1

This notebook includes both coding and written questions. Please hand in this notebook file with all the outputs as a pdf on gradescope for "HW6 pdf". Upload the three files of code (`compression.py`, `k_nearest_neighbor.py` and `features.py`) on gradescope for "HW6 code".

This assignment covers: - image compression using SVD - kNN methods for image recognition. - PCA and LDA to improve kNN

[2]:
```
# Setup
from time import time
from collections import defaultdict

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import rc
from skimage import io
```

```
%matplotlib inline
plt.rcParams['figure.figsize'] = (15.0, 12.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# for auto-reloading external modules
%load_ext autoreload
%autoreload 2
```

1.1 Part 1 - Image Compression (15 points)

Image compression is used to reduce the cost of storage and transmission of images (or videos). One lossy compression method is to apply Singular Value Decomposition (SVD) to an image, and only keep the top n singular values.

[3]:
```
image = io.imread('pitbull.jpg', as_gray=True)
plt.figure(figsize=(10,10))
plt.imshow(image)
plt.axis('off')
plt.show()
```

Let's implement image compression using SVD.
We first compute the SVD of the image, and as seen in class we keep the `n` largest singular values and singular vectors to reconstruct the image.

Implement function `compress_image` in `compression.py`.

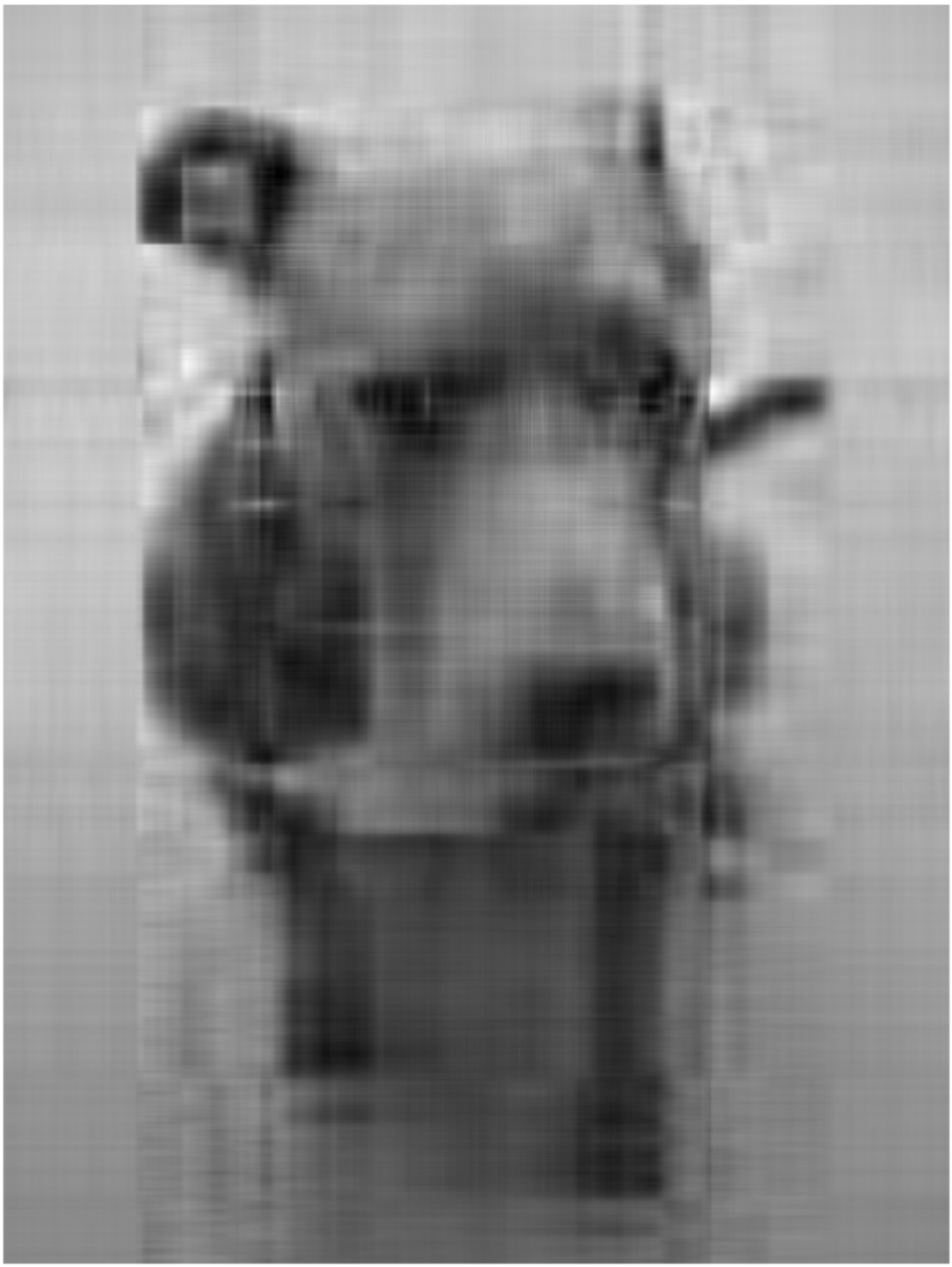
[4]:
```
from compression import compress_image

compressed_image, compressed_size = compress_image(image, 100)
compression_ratio = compressed_size / image.size
print('Original image shape:', image.shape)
print('Compressed size: %d' % compressed_size)
print('Compression ratio: %.3f' % compression_ratio)

assert compressed_size == 298500
```

```
Original image shape: (1704, 1280)
Compressed size: 298500
Compression ratio: 0.137
```

[5]:
```
# Number of singular values to keep
n_values = [10, 50, 100]

for n in n_values:
    # Compress the image using `n` singular values
    compressed_image, compressed_size = compress_image(image, n)

    compression_ratio = compressed_size / image.size

    print("Data size (original): %d" % (image.size))
    print("Data size (compressed): %d" % compressed_size)
    print("Compression ratio: %f" % (compression_ratio))


    plt.figure(figsize=(10,10))
    plt.imshow(compressed_image, cmap='gray')
    title = "n = %s" % n
    plt.title(title)
    plt.axis('off')
    plt.show()
```

```
Data size (original): 2181120
Data size (compressed): 29850
Compression ratio: 0.013686
```

n = 10



```
Data size (original): 2181120
Data size (compressed): 149250
```

```
Compression ratio: 0.068428
```


```
Data size (original): 2181120
```

```
Data size (compressed): 298500
Compression ratio: 0.136856
```

1.2 Face Dataset

We will use a dataset of faces of celebrities. Download the dataset using the following command in a code block:

```
!bash get_dataset.sh
```

The face dataset for CS131 assignment. The directory containing the dataset has the following structure:

```
faces/
    train/
        angelina jolie/
        anne hathaway/
        ...
    test/
        angelina jolie/
        anne hathaway/
        ...
```

Each class has 50 training images and 10 testing images.

[6]:
```
from utils import load_dataset

X_train, y_train, classes_train = load_dataset('faces', train=True,␣
 ↪as_gray=True)
X_test, y_test, classes_test = load_dataset('faces', train=False, as_gray=True)

assert classes_train == classes_test
classes = classes_train

print('Class names:', classes)
print('Training data shape:', X_train.shape)
print('Training labels shape: ', y_train.shape)
print('Test data shape:', X_test.shape)
print('Test labels shape: ', y_test.shape)
```

```
Class names: ['angelina jolie', 'anne hathaway', 'barack obama', 'brad pitt',
'cristiano ronaldo', 'emma watson', 'george clooney', 'hillary clinton',
'jennifer aniston', 'johnny depp', 'justin timberlake', 'leonardo dicaprio',
'natalie portman', 'nicole kidman', 'scarlett johansson', 'tom cruise']
Training data shape: (800, 64, 64)
Training labels shape:  (800,)
Test data shape: (160, 64, 64)
Test labels shape:  (160,)
```

[7]:
```
# Visualize some examples from the dataset.
# We show a few examples of training images from each class.
num_classes = len(classes)
samples_per_class = 10
```

```
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow(X_train[idx])
        plt.axis('off')
        if i == 0:
            plt.title(y)
plt.show()
```

[8]:
```
# Flatten the image data into rows
# we now have one 4096 dimensional featue vector for each example
X_train = np.reshape(X_train, (X_train.shape[0], -1))
X_test = np.reshape(X_test, (X_test.shape[0], -1))
print("Training data shape:", X_train.shape)
print("Test data shape:", X_test.shape)
```

```
Training data shape: (800, 4096)
Test data shape: (160, 4096)
```

1.3 Part 2 - k-Nearest Neighbor (30 points)

We're now going to try to classify the test images using the k-nearest neighbors algorithm on the **raw features of the images** (i.e. the pixel values themselves). We will see later how we can use kNN on better features.

The gist of the k-nearest neighbors algorithm is to predict a test image's class based on which classes the k nearest train images belong to. For example, using k = 3, if we found that for test image X, the three nearest train images were 2 pictures of Angelina Jolie, and one picture of Audrey Hepburn, we would predict that the test image X is a picture of Angelina Jolie.

Here are the steps that we will follow:

1. We compute the L2 distances between every element of X_test and every element of X_train in `compute_distances`.
2. We split the dataset into 5 folds for cross-validation in `split_folds`.
3. For each fold, and for different values of `k`, we predict the labels and measure accuracy.
4. Using the best `k` found through cross-validation, we measure accuracy on the test set.

Resources for understanding cross-validation: https://towardsdatascience.com/why-and-how-to-cross-validate-a-model-d6424b45261f Implement function `compute_distances` and `predict_labels` in `k_nearest_neighbor.py`.

[9]:
```
from k_nearest_neighbor import compute_distances

# Step 1: compute the distances between all features from X_train and from␣
↪X_test
dists = compute_distances(X_test, X_train)
assert dists.shape == (160, 800)
print("dists shape:", dists.shape)
```

```
dists shape: (160, 800)
```

[10]:
```
from k_nearest_neighbor import predict_labels

# We use k = 1 (which corresponds to only taking the nearest neighbor to decide)
y_test_pred = predict_labels(dists, y_train, k=1)

# Compute and print the fraction of correctly predicted examples
num_test = y_test.shape[0]
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 38 / 160 correct => accuracy: 0.237500
```

1.3.1 Cross-Validation

We don't know the best value for our parameter `k`.
There is no theory on how to choose an optimal `k`, and the way to choose it is through cross-validation.

We **cannot** compute any metric on the test set to choose the best `k`, because we want our final test accuracy to reflect a real use case. This real use case would be a setting where we have new examples come and we classify them on the go. There is no way to check the accuracy beforehand on that set of test examples to determine `k`.

Cross-validation will make use split the data into different fold (5 here).
For each fold, if we have a total of 5 folds we will have: - 80% of the data as training data - 20% of the data as validation data

We will compute the accuracy on the validation accuracy for each fold, and use the mean of these 5 accuracies to determine the best parameter `k`. Implement function `split_folds` in `k_nearest_neighbor.py`.

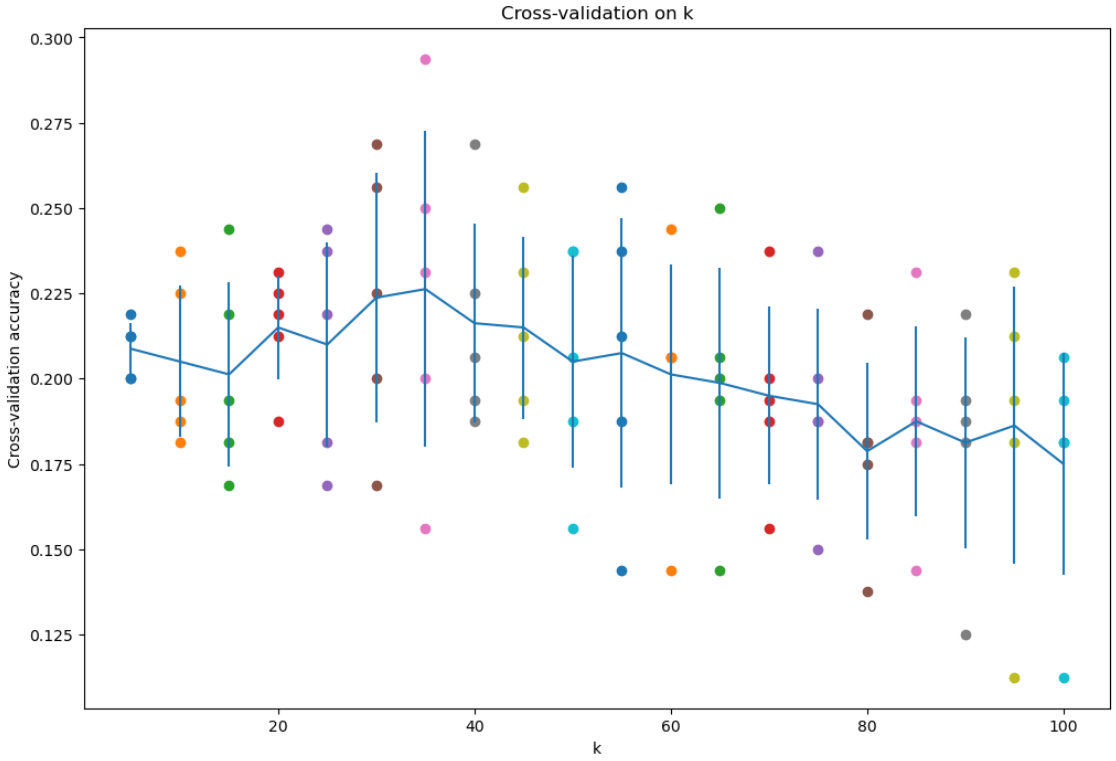
[11]:
```
from k_nearest_neighbor import split_folds

# Step 2: split the data into 5 folds to perform cross-validation.
num_folds = 5

X_trains, y_trains, X_vals, y_vals = split_folds(X_train, y_train, num_folds)

assert X_trains.shape == (5, 640, 4096)
assert y_trains.shape == (5, 640)
assert X_vals.shape == (5, 160, 4096)
assert y_vals.shape == (5, 160)
```

[12]:
```
# Step 3: Measure the mean accuracy for each value of `k`

# List of k to choose from
k_choices = list(range(5, 101, 5))

# Dictionnary mapping k values to accuracies
# For each k value, we will have `num_folds` accuracies to compute
# k_to_accuracies[1] will be for instance [0.22, 0.23, 0.19, 0.25, 0.20] for 5␣
 ↪folds
k_to_accuracies = {}

for k in k_choices:
    print("Running for k=%d" % k)
    accuracies = []
    for i in range(num_folds):
        # Make predictions
        fold_dists = compute_distances(X_vals[i], X_trains[i])
        y_pred = predict_labels(fold_dists, y_trains[i], k)
```

```
        # Compute and print the fraction of correctly predicted examples
        num_correct = np.sum(y_pred == y_vals[i])
        accuracy = float(num_correct) / len(y_vals[i])
        accuracies.append(accuracy)

    k_to_accuracies[k] = accuracies
```

```
Running for k=5
Running for k=10
Running for k=15
Running for k=20
Running for k=25
Running for k=30
Running for k=35
Running for k=40
Running for k=45
Running for k=50
Running for k=55
Running for k=60
Running for k=65
Running for k=70
Running for k=75
Running for k=80
Running for k=85
Running for k=90
Running for k=95
Running for k=100
```

[13]:
```
# plot the raw observations
plt.figure(figsize=(12,8))
for k in k_choices:
    accuracies = k_to_accuracies[k]
    plt.scatter([k] * len(accuracies), accuracies)

# plot the trend line with error bars that correspond to standard deviation
accuracies_mean = np.array([np.mean(v) for k,v in sorted(k_to_accuracies.
 ↪items())])
accuracies_std = np.array([np.std(v) for k,v in sorted(k_to_accuracies.
 ↪items())])
plt.errorbar(k_choices, accuracies_mean, yerr=accuracies_std)
plt.title('Cross-validation on k')
plt.xlabel('k')
plt.ylabel('Cross-validation accuracy')
plt.show()
```



[14]:
```
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 26% accuracy on the test data.

best_k = None

# YOUR CODE HERE
# Choose the best k based on the cross validation above
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

best_k = k_choices[np.argmax(accuracies_mean)]

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
# END YOUR CODE

y_test_pred = predict_labels(dists, y_train, k=best_k)

# Compute and display the accuracy
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('For k = %d, got %d / %d correct => accuracy: %f' % (best_k, num_correct,␣
 ↪num_test, accuracy))
```

```
For k = 35, got 46 / 160 correct => accuracy: 0.287500
```

1.4 Part 3: PCA (30 points)

Principal Component Analysis (PCA) is a simple yet popular and useful linear transformation technique that is used in numerous applications, such as stock market predictions, the analysis of gene expression data, and many more. In this tutorial, we will see that PCA is not just a "black box", and we are going to unravel its internals in 3 basic steps.

1.4.1 Introduction

The sheer size of data in the modern age is not only a challenge for computer hardware but also a main bottleneck for the performance of many machine learning algorithms. The main goal of a PCA analysis is to identify patterns in data; PCA aims to detect the correlation between variables. If a strong correlation between variables exists, the attempt to reduce the dimensionality only makes sense. In a nutshell, this is what PCA is all about: Finding the directions of maximum variance in high-dimensional data and project it onto a smaller dimensional subspace while retaining most of the information.

1.4.2 A Summary of the PCA Approach

- Standardize the data.
- Obtain the Eigenvectors and Eigenvalues from the covariance matrix or correlation matrix, or perform Singular Vector Decomposition.
- Sort eigenvalues in descending order and choose the k eigenvectors that correspond to the k largest eigenvalues where k is the number of dimensions of the new feature subspace ($k \leq d$).
- Construct the projection matrix $\mathbf{W}$ from the selected k eigenvectors.
- Transform the original dataset $\mathbf{X}$ via $\mathbf{W}$ to obtain a k-dimensional feature subspace Y.

[15]:
```
from features import PCA

pca = PCA()
```

1.4.3 3.1 - Eigendecomposition

The eigenvectors and eigenvalues of a covariance (or correlation) matrix represent the "core" of a PCA: The eigenvectors (principal components) determine the directions of the new feature space, and the eigenvalues determine their magnitude. In other words, the eigenvalues explain the variance of the data along the new feature axes.

Implement **`_eigen_decomp`** for class PCA in `features.py`.

[16]:
```
# Perform eigenvalue decomposition on the covariance matrix of training data.
e_vecs, e_vals = pca._eigen_decomp(X_train - X_train.mean(axis=0))

print(e_vals.shape)
print(e_vecs.shape)
```

```
(4096,)
(4096, 4096)
```

1.4.4 3.2 - Singular Value Decomposition

Doing an eigendecomposition of the covariance matrix is very expensive, especially when the number of features (`D = 4096` here) gets very high.

To obtain the same eigenvalues and eigenvectors in a more efficient way, we can use Singular Value Decomposition (SVD). If we perform SVD on matrix X, we obtain U, S and V such that:

$$X = USV^T$$

- the columns of U are the eigenvectors of XX^T
- the columns of V^T are the eigenvectors of X^TX
- the values of S are the square roots of the eigenvalues of X^TX (or XX^T)

Therefore, we can find out the top `k` eigenvectors of the covariance matrix X^TX using SVD.

Implement **`_svd`** for class PCA in `features.py`.

[17]:
```
# Perform SVD on directly on the training data.
u, s = pca._svd(X_train - X_train.mean(axis=0)) # hugely confusing to call u...

print(s.shape)
print(u.shape)
```

```
(800,)
(4096, 4096)
```

[18]:
```
# Test whether the square of singular values and eigenvalues are the same.
# We also observe that `e_vecs` and `u` are the same (only the sign of each␣
 ↪column can differ).
N = X_train.shape[0]
assert np.allclose((s ** 2) / (N - 1), e_vals[:len(s)])

for i in range(len(s) - 1):
    assert np.allclose(e_vecs[:, i], u[:, i]) or np.allclose(e_vecs[:, i], -u[:
 ↪, i])
    # (the last eigenvector for i = len(s) - 1 is very noisy because the␣
 ↪eigenvalue is almost 0,
    #  so imprecisions in the computation build up)
```

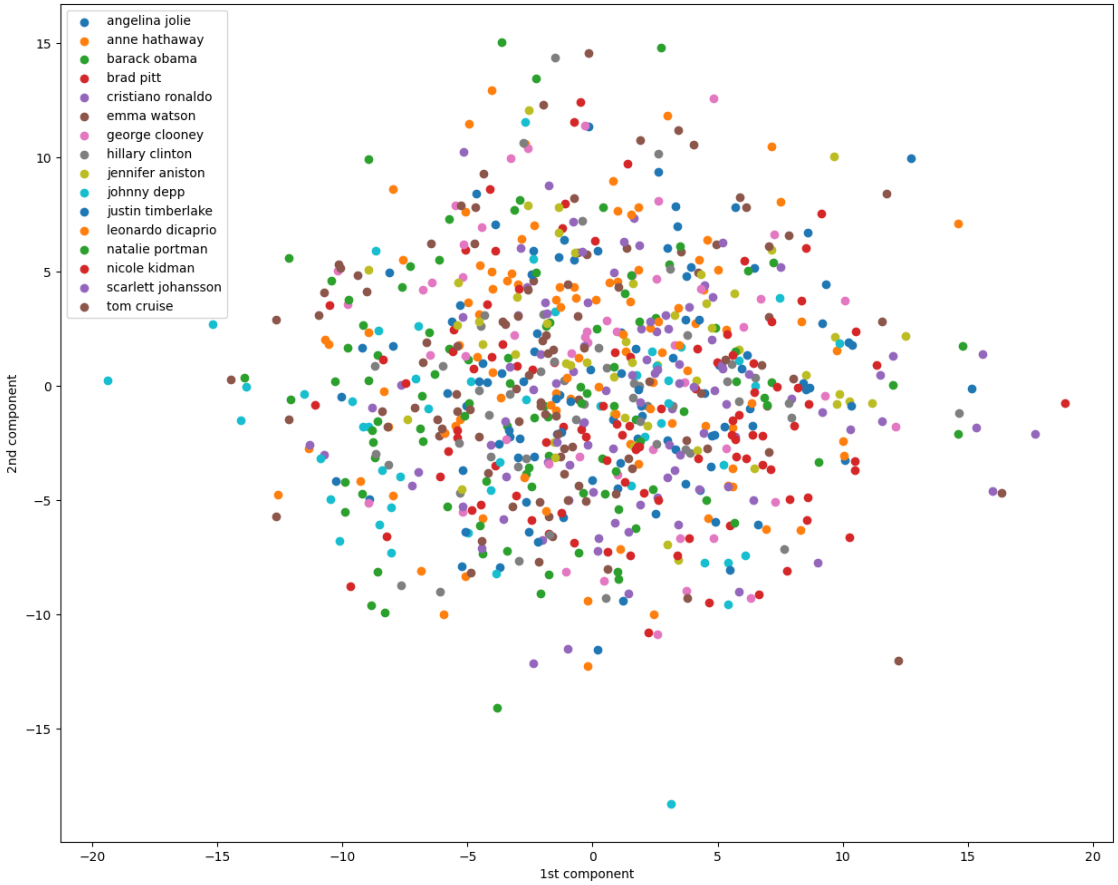
1.4.5 3.3 - Dimensionality Reduction

The top k principal components explain most of the variance of the underlying data.

By projecting our initial data (the images) onto the subspace spanned by the top k principal components, we can reduce the dimension of our inputs while keeping most of the information.

In the example below, we can see that **using the first two components in PCA is not enough** to allow us to see pattern in the data. All the classes seem placed at random in the 2D plane. Implement **`fit`** for class PCA in `features.py`.

[19]:
```
# Dimensionality reduction by projecting the data onto
# lower dimensional subspace spanned by k principal components

# To visualize, we will project in 2 dimensions
n_components = 2
pca.fit(X_train)
X_proj = pca.transform(X_train, n_components)

# Plot the top two principal components
for y in np.unique(y_train):
    plt.scatter(X_proj[y_train==y,0], X_proj[y_train==y,1], label=classes[y])

plt.xlabel('1st component')
plt.ylabel('2nd component')
plt.legend()
plt.show()
```



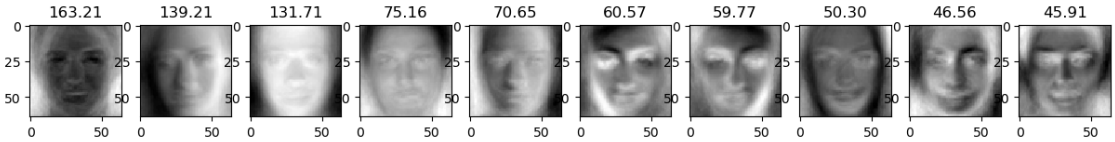
1.4.6 3.4 - Visualizing Eigenfaces

The columns of the PCA projection matrix `pca.W_pca` represent the eigenvectors of $X^T X$.

We can visualize the biggest singular values as well as the corresponding vectors to get a sense of what the PCA algorithm is extracting.

If we visualize the top 10 eigenfaces, we can see that the algorithm focuses on the different shades of the faces. For instance, in face n°2 the light seems to come from the left.

[20]:
```
for i in range(10):
    plt.subplot(1, 10, i+1)
    plt.imshow(pca.W_pca[:, i].reshape(64, 64))
    plt.title("%.2f" % s[i])
plt.show()
```



[21]:
```
# Reconstruct data with principal components
n_components = 64  # Experiment with different number of components.
X_proj = pca.transform(X_train, n_components)
X_rec = pca.reconstruct(X_proj)

print(X_rec.shape)
print(classes)

# Visualize reconstructed faces
samples_per_class = 10
for y, cls in enumerate(classes):
    idxs = np.flatnonzero(y_train == y)
    idxs = np.random.choice(idxs, samples_per_class, replace=False)
    for i, idx in enumerate(idxs):
        plt_idx = i * num_classes + y + 1
        plt.subplot(samples_per_class, num_classes, plt_idx)
        plt.imshow((X_rec[idx]).reshape((64, 64)))
        plt.axis('off')
        if i == 0:
            plt.title(y)
plt.show()
```

```
(800, 4096)
['angelina jolie', 'anne hathaway', 'barack obama', 'brad pitt', 'cristiano
ronaldo', 'emma watson', 'george clooney', 'hillary clinton', 'jennifer
aniston', 'johnny depp', 'justin timberlake', 'leonardo dicaprio', 'natalie
portman', 'nicole kidman', 'scarlett johansson', 'tom cruise']
```

1.4.7 Written Question 1 (5 points)

Question: Consider a dataset of N face images, each with shape (h, w). Then, we need $O(Nhw)$ of memory to store the data. Suppose we perform dimensionality reduction on the dataset with p principal components, and use the components as bases to represent images. Calculate how much memory we need to store the images and the matrix used to get back to the original space.

Said in another way, how much memory does storing the compressed images **and** the uncompresser cost.

Answer: For a fixed image, we flatten it to be have shape $(hw,)$ and then we project onto some smaller p dimensional subspace. This means that each image takes only p values to be stored. Since we have N images, this gives Np spcae for the compressed images. On the other hand, the projection matrix, which stores the first p eigenvalues of the covariance matrix, has shape (hw, p), so it takes hwp values to be stored. So in total we get $Np + hwp$. For $p \ll N$, this is much better!

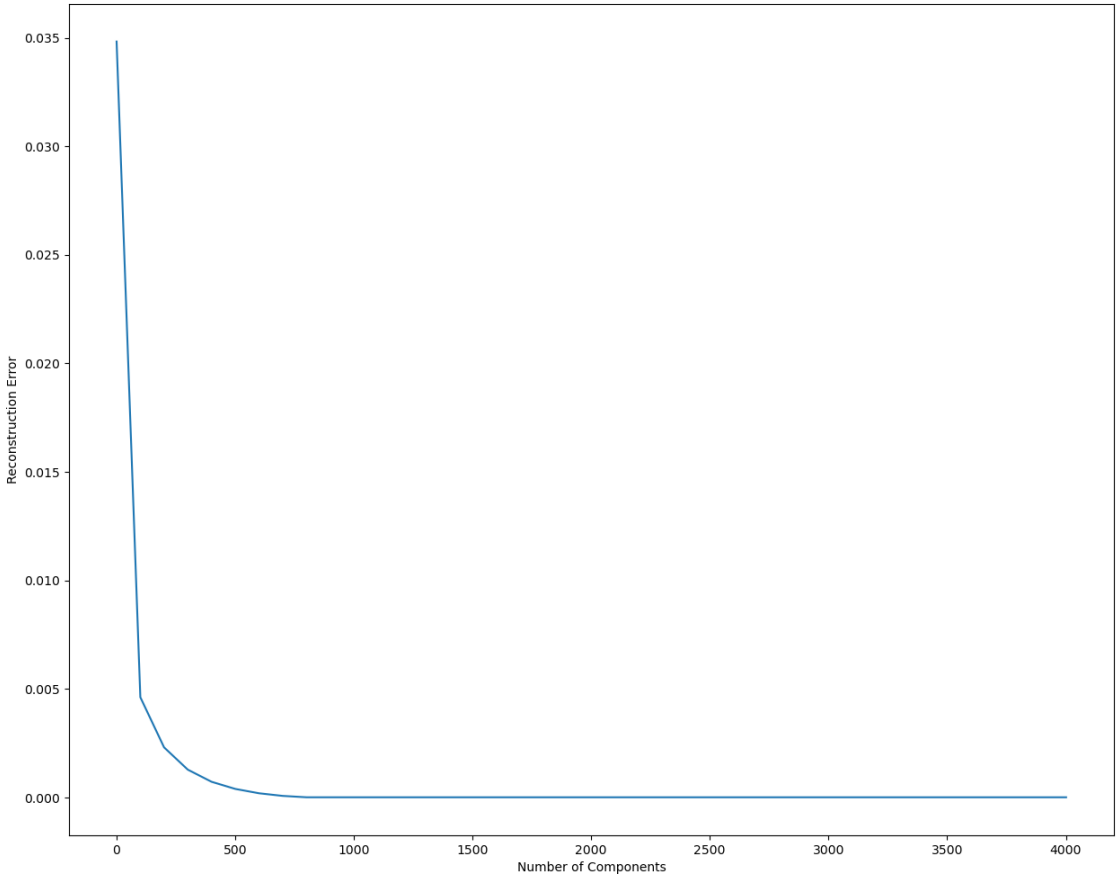
1.4.8 3.5 - Reconstruction error and captured variance

We can plot the reconstruction error with respect to the dimension of the projected space.

The reconstruction gets better with more components.

We can see in the plot that the inflexion point is around dimension 200 or 300. This means that using this number of components is a good compromise between good reconstruction and low dimension.

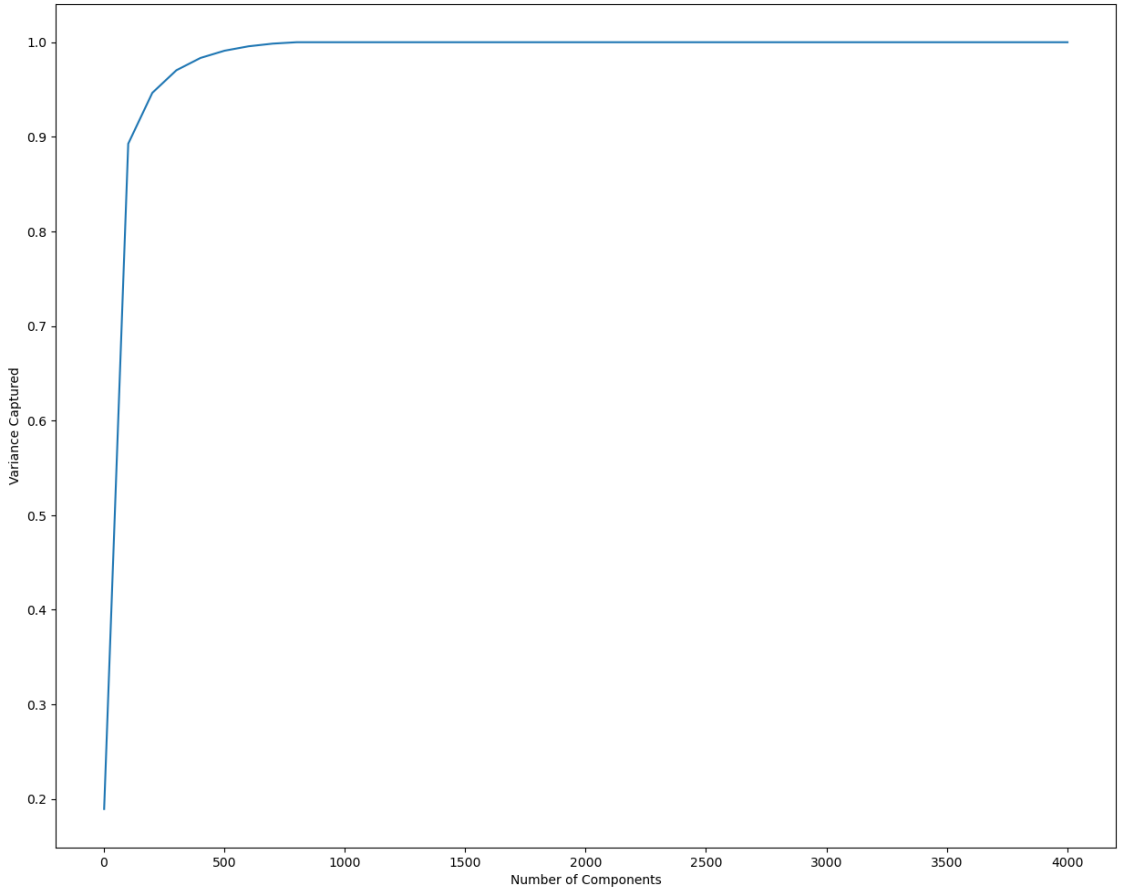
[22]:
```
# Plot reconstruction errors for different k
N = X_train.shape[0]
d = X_train.shape[1]

ns = range(1, d, 100)
errors = []

for n in ns:
    X_proj = pca.transform(X_train, n)
    X_rec = pca.reconstruct(X_proj)

    # Compute reconstruction error
    error = np.mean((X_rec - X_train) ** 2)
    errors.append(error)

plt.plot(ns, errors)
plt.xlabel('Number of Components')
plt.ylabel('Reconstruction Error')
plt.show()
```



We can do the same process to see how much variance is captured by the projection.

Again, we see that the inflexion point is around 200 or 300 dimensions.

[23]:
```
# Plot captured variance
ns = range(1, d, 100)
var_cap = []

for n in ns:
    var_cap.append(np.sum(s[:n] ** 2)/np.sum(s ** 2))

plt.plot(ns, var_cap)
plt.xlabel('Number of Components')
plt.ylabel('Variance Captured')
plt.show()
```



1.4.9 3.6 - kNN with PCA

Performing kNN on raw features (the pixels of the image) does not yield very good results.
Computing the distance between images in the image space is not a very good metric for actual proximity of images.
For instance, an image of person A with a dark background will be close to an image of B with a dark background, although these people are not the same.

Using a technique like PCA can help discover the real interesting features and perform kNN on them could give better accuracy.

However, we observe here that PCA doesn't really help to disentangle the features and obtain useful distance metrics between the different classes. We basically obtain the same performance as with raw features.

[24]:
```
num_test = X_test.shape[0]

# We computed the best k and n for you
best_k = 20
best_n = 500
```

```
# PCA
pca = PCA()
pca.fit(X_train)
X_proj = pca.transform(X_train, best_n)
X_test_proj = pca.transform(X_test, best_n)

# kNN
dists = compute_distances(X_test_proj, X_proj)
y_test_pred = predict_labels(dists, y_train, k=best_k)

# Compute and display the accuracy
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
Got 42 / 160 correct => accuracy: 0.262500
```

1.5 Part 4 - Fisherface: Linear Discriminant Analysis (25 points)

LDA is a linear transformation method like PCA, but with a different goal.
The main difference is that LDA takes information from the labels of the examples to maximize the separation of the different classes in the transformed space.

Therefore, LDA is not totally **unsupervised** since it requires labels. PCA is **fully unsupervised**.

In summary: - PCA perserves maximum variance in the projected space. - LDA preserves discrimination between classes in the project space. We want to maximize scatter between classes and minimize scatter intra class.

[33]:
```
from features import LDA

lda = LDA()
```

1.5.1 4.1 - Dimensionality Reduction via PCA

To apply LDA, we need $D < N$. Since in our dataset, $N = 800$ and $D = 4096$, we first need to reduce the number of dimensions of the images using PCA.
More information at: http://www.scholarpedia.org/article/Fisherfaces

[34]:
```
N = X_train.shape[0]
c = num_classes

pca = PCA()
pca.fit(X_train)
X_train_pca = pca.transform(X_train, N-c)
X_test_pca = pca.transform(X_test, N-c)
```

```
print(X_train_pca.shape)
print(X_test_pca.shape)
```

```
(800, 784)
(160, 784)
```

1.5.2 4.2 - Scatter matrices

We first need to compute the within-class scatter matrix:

$$S_W = \sum_{i=1}^{c} S_i$$

where $S_i = \sum_{x_k \in Y_i} (x_k - \mu_i)(x_k - \mu_i)^T$ is the scatter of class i.

We then need to compute the between-class scatter matrix:

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

where N_i is the number of examples in class i. Implement **`_within_class_scatter`** and **`_between_class_scatter`** for class LDA in `features.py`.

[55]:
```
# Compute within-class scatter matrix
S_W = lda._within_class_scatter(X_train_pca, y_train)
print(S_W.shape)
```

```
(784, 784)
```

[56]:
```
# Compute between-class scatter matrix
S_B = lda._between_class_scatter(X_train_pca, y_train)
print(S_B.shape) # has jank, outer products don't seem to be PSD as 783␣
↪eigenvalues are 0, so creep into the negative
```

```
(784, 784)
```

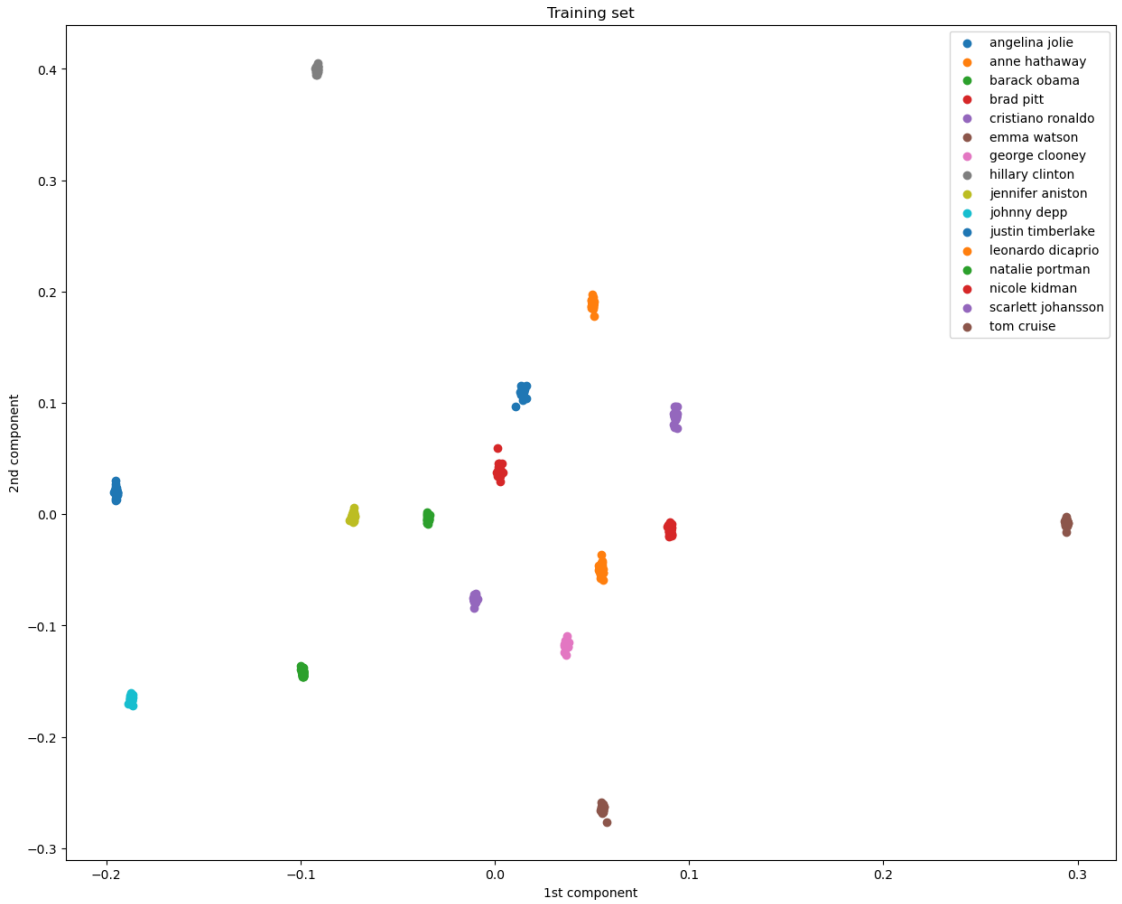
1.5.3 4.3 - Solving generalized Eigenvalue problem

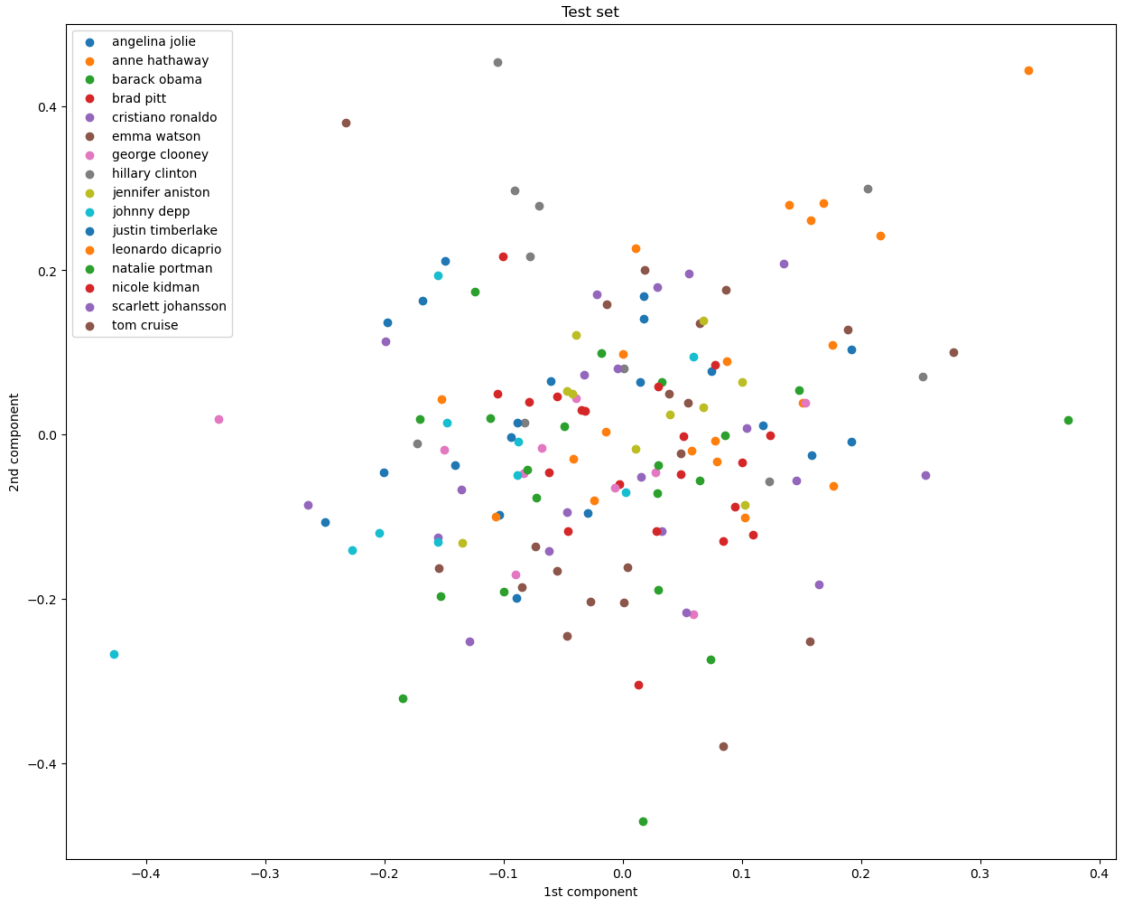
Implement methods `fit` and `transform` of the `LDA` class.

[57]:
```
lda.fit(X_train_pca, y_train)
```

[58]:
```
# Dimensionality reduction by projecting the data onto
# lower dimensional subspace spanned by k principal components
n_components = 2
X_proj = lda.transform(X_train_pca, n_components)
X_test_proj = lda.transform(X_test_pca, n_components)

# Plot the top two principal components on the training set
for y in np.unique(y_train):
    plt.scatter(X_proj[y_train==y, 0], X_proj[y_train==y, 1], label=classes[y])
```

```
plt.xlabel('1st component')
plt.ylabel('2nd component')
plt.legend()
plt.title("Training set")
plt.show()

# Plot the top two principal components on the test set
for y in np.unique(y_test):
    plt.scatter(X_test_proj[y_test==y, 0], X_test_proj[y_test==y,1],␣
 ↪label=classes[y])

plt.xlabel('1st component')
plt.ylabel('2nd component')
plt.legend()
plt.title("Test set")
plt.show()
```





1.5.4 4.4 - kNN with LDA

Thanks to having the information from the labels, LDA gives a discriminant space where the classes are far apart from each other.
This should help kNN a lot, as the job should just be to find the obvious 10 clusters.

However, as we've seen in the previous plot (section 4.3), the training data gets clustered pretty well, but the test data isn't as nicely clustered as the training data (overfitting?).

Perform cross validation following the code below (you can change the values of `k_choices` and `n_choices` to search). Using the best result from cross validation, obtain the test accuracy.

[31]:
```
num_folds = 5

X_trains, y_trains, X_vals, y_vals = split_folds(X_train, y_train, num_folds)

k_choices = [1, 5, 10, 20]
n_choices = [5, 10, 20, 50, 100, 200, 500]
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
k_choices = [10, 15, 20]
n_choices = [10, 20, 30, 40, 50, 60, 70]

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****


# n_k_to_accuracies[(n, k)] should be a list of length num_folds giving the␣
 ↪different
# accuracy values that we found when using that value of n and k.
n_k_to_accuracies = defaultdict(list)

for i in range(num_folds):
    # Fit PCA
    pca = PCA()
    pca.fit(X_trains[i])

    N = len(X_trains[i])
    X_train_pca = pca.transform(X_trains[i], N-c)
    X_val_pca = pca.transform(X_vals[i], N-c)

    # Fit LDA
    lda = LDA()
    lda.fit(X_train_pca, y_trains[i])

    for n in n_choices:
        X_train_proj = lda.transform(X_train_pca, n)
        X_val_proj = lda.transform(X_val_pca, n)

        dists = compute_distances(X_val_proj, X_train_proj)

        for k in k_choices:
            y_pred = predict_labels(dists, y_trains[i], k=k)

            # Compute and print the fraction of correctly predicted examples
            num_correct = np.sum(y_pred == y_vals[i])
            accuracy = float(num_correct) / len(y_vals[i])
            n_k_to_accuracies[(n, k)].append(accuracy)


for n in n_choices:
    print()
    for k in k_choices:
        accuracies = n_k_to_accuracies[(n, k)]
        print("For n=%d, k=%d: average accuracy is %f" % (n, k, np.
 ↪mean(accuracies)))
```

```
For n=10, k=10: average accuracy is 0.297500
```

```
For n=10, k=15: average accuracy is 0.298750
For n=10, k=20: average accuracy is 0.298750

For n=20, k=10: average accuracy is 0.377500
For n=20, k=15: average accuracy is 0.371250
For n=20, k=20: average accuracy is 0.381250

For n=30, k=10: average accuracy is 0.377500
For n=30, k=15: average accuracy is 0.376250
For n=30, k=20: average accuracy is 0.373750

For n=40, k=10: average accuracy is 0.386250
For n=40, k=15: average accuracy is 0.376250
For n=40, k=20: average accuracy is 0.381250

For n=50, k=10: average accuracy is 0.393750
For n=50, k=15: average accuracy is 0.391250
For n=50, k=20: average accuracy is 0.386250

For n=60, k=10: average accuracy is 0.391250
For n=60, k=15: average accuracy is 0.385000
For n=60, k=20: average accuracy is 0.392500

For n=70, k=10: average accuracy is 0.381250
For n=70, k=15: average accuracy is 0.375000
For n=70, k=20: average accuracy is 0.375000
```

[32]:
```python
# Based on the cross-validation results above, choose the best value for k,
# retrain the classifier using all the training data, and test it on the test
# data. You should be able to get above 40% accuracy on the test data.

best_k = None
best_n = None
# YOUR CODE HERE
# Choose the best k based on the cross validation above
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

best_k = 10
best_n = 50

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
# END YOUR CODE

N = len(X_train)

# Fit PCA
pca = PCA()
```

```
pca.fit(X_train)
X_train_pca = pca.transform(X_train, N-c)
X_test_pca = pca.transform(X_test, N-c)

# Fit LDA
lda = LDA()
lda.fit(X_train_pca, y_train)

# Project using LDA
X_train_proj = lda.transform(X_train_pca, best_n)
X_test_proj = lda.transform(X_test_pca, best_n)

dists = compute_distances(X_test_proj, X_train_proj)
y_test_pred = predict_labels(dists, y_train, k=best_k)

# Compute and display the accuracy
num_correct = np.sum(y_test_pred == y_test)
accuracy = float(num_correct) / num_test
print("For k=%d and n=%d" % (best_k, best_n))
print('Got %d / %d correct => accuracy: %f' % (num_correct, num_test, accuracy))
```

```
For k=10 and n=50
Got 67 / 160 correct => accuracy: 0.418750
```

Detection

March 7, 2025

[1]:
```
# # This mounts your Google Drive to the Colab VM.
# from google.colab import drive
# drive.mount('/content/drive')
#
# # TODO: Enter the foldername in your Drive where you have saved the unzipped
# # assignment folder, e.g. 'cse455/assignments/assignment4/'
# FOLDERNAME = None
# assert FOLDERNAME is not None, "[!] Enter the foldername."
#
# # Now that we've mounted your Drive, this ensures that
# # the Python interpreter of the Colab VM can load
# # python files from within it.
# import sys
# sys.path.append('/content/drive/My Drive/{}'.format(FOLDERNAME))
#
# # This downloads the COCO dataset to your Drive
# # if it doesn't already exist.
# %cd /content/drive/My\ Drive/$FOLDERNAME/cse455/datasets/
# !bash get_datasets.sh
# %cd /content/drive/My\ Drive/$FOLDERNAME
```

1 Assignment 4 Part 2

In this homework, we will implement a simplified version of object detection process. Note that the tests on the notebook are not comprehensive, autograder will contain more tests.

[2]:
```
from __future__ import print_function
import random
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.patches as patches
from skimage import io, color
from skimage.feature import hog
from skimage import data, color, exposure
from skimage.transform import rescale, resize, downscale_local_mean
import glob, os
import fnmatch
```

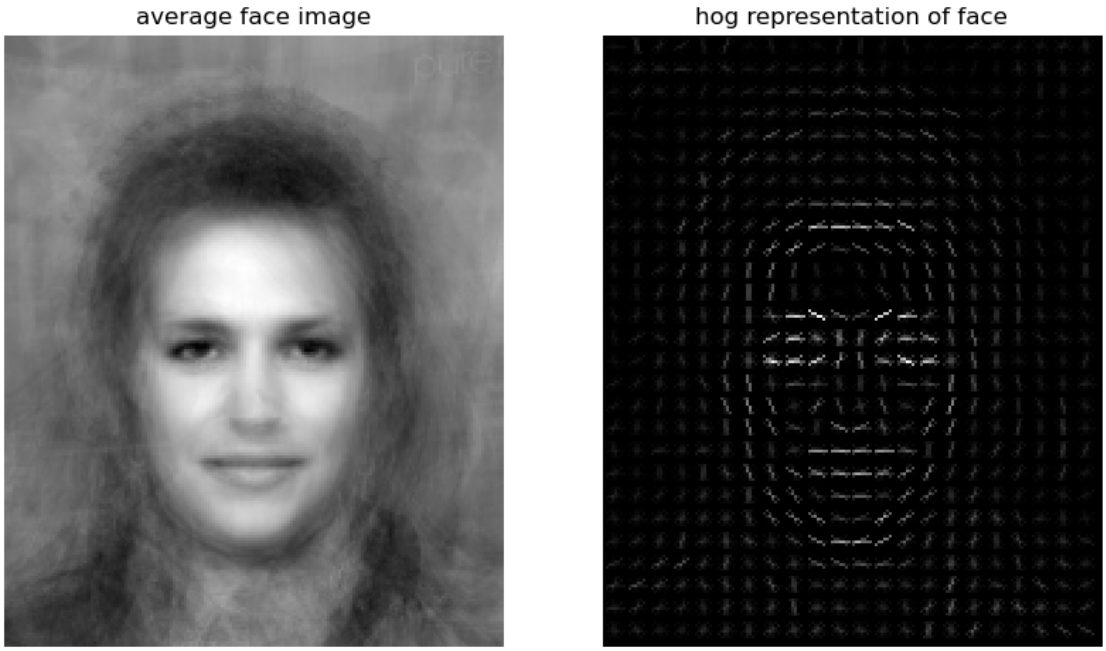
```
import time

import warnings
warnings.filterwarnings('ignore')

from detection import *
from visualization import *
from util import *

# This code is to make matplotlib figures appear inline in the
# notebook rather than in a new window.
%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

# Some more magic so that the notebook will reload external python modules;
# see http://stackoverflow.com/questions/1907993/
  ↪autoreload-of-modules-in-ipython
%load_ext autoreload
%autoreload 2
%reload_ext autoreload
```

2 Part 1: Hog Representation (10 points)

In this section, we will compute the average hog representation of human faces. There are 31 aligned face images provided in the `\face` folder. They are all aligned and have the same size. We will get an average face from these images and compute a hog feature representation for the averaged face. Use the hog function provided by skimage library, and implement a hog representation of objects. Implement **`hog_feature`** function in `detection.py`

[3]:
```
image_paths = fnmatch.filter(os.listdir('./face'), '*.jpg')
list.sort(image_paths)
n = len(image_paths)
face_shape, avg_face = load_faces(image_paths, n)

(face_feature, hog_image) = hog_feature(avg_face)

print(np.sum(face_feature))
assert np.abs(np.sum(face_feature) - 499.970465079) < 1e-2

plot_part1(avg_face, hog_image)
```

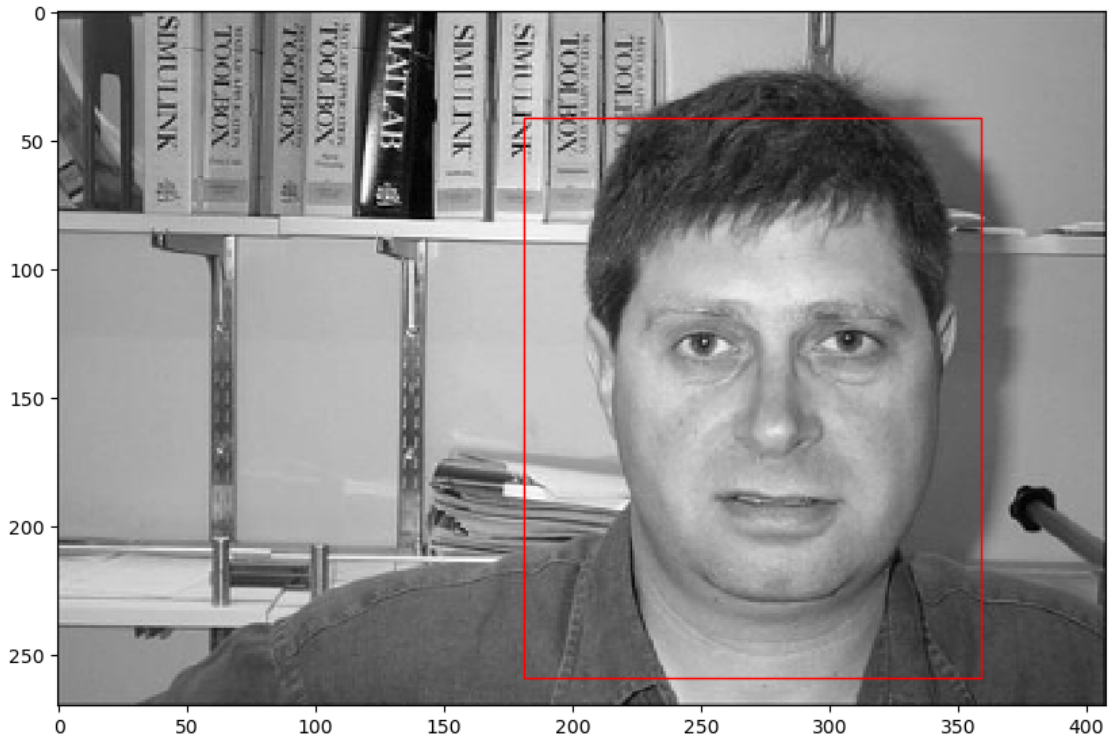
```
499.9704650788743
```

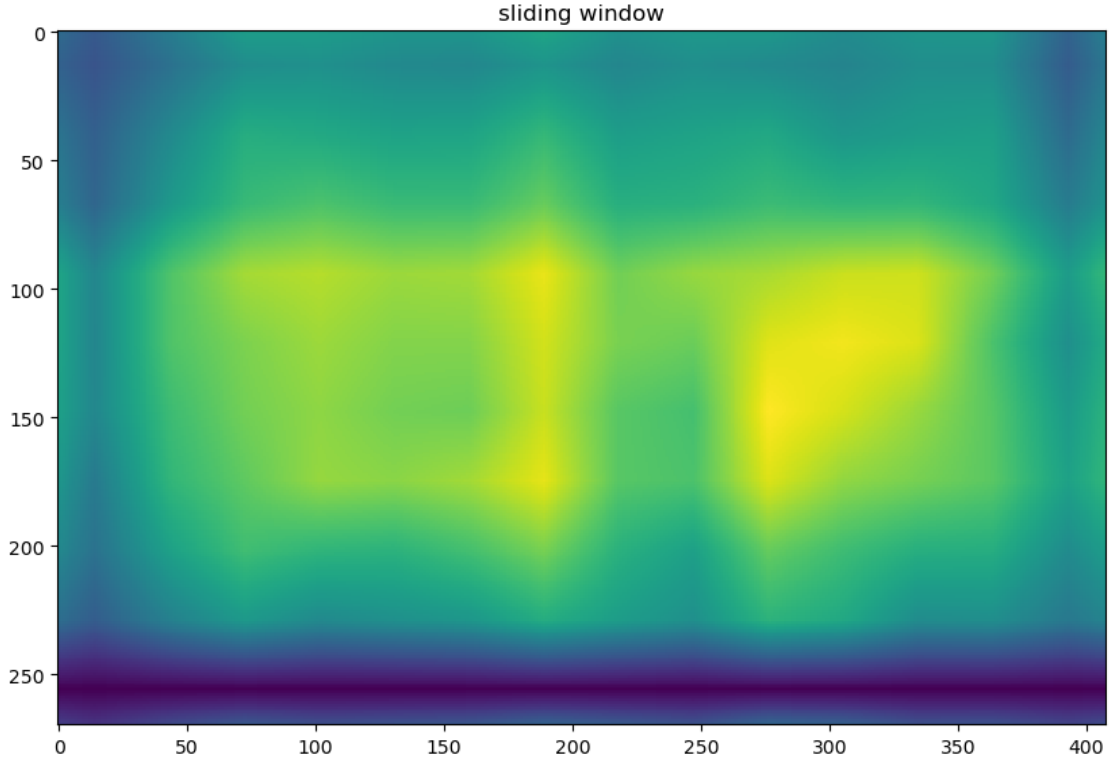


3 Part 2: Sliding Window (30 points)

Implement **`sliding_window`** function to have windows slide across an image with a specific window size. The window slides through the image and check if an object is detected with a high score at every location. These scores will generate a response map and you will be able to find the location of the window with the highest hog score.

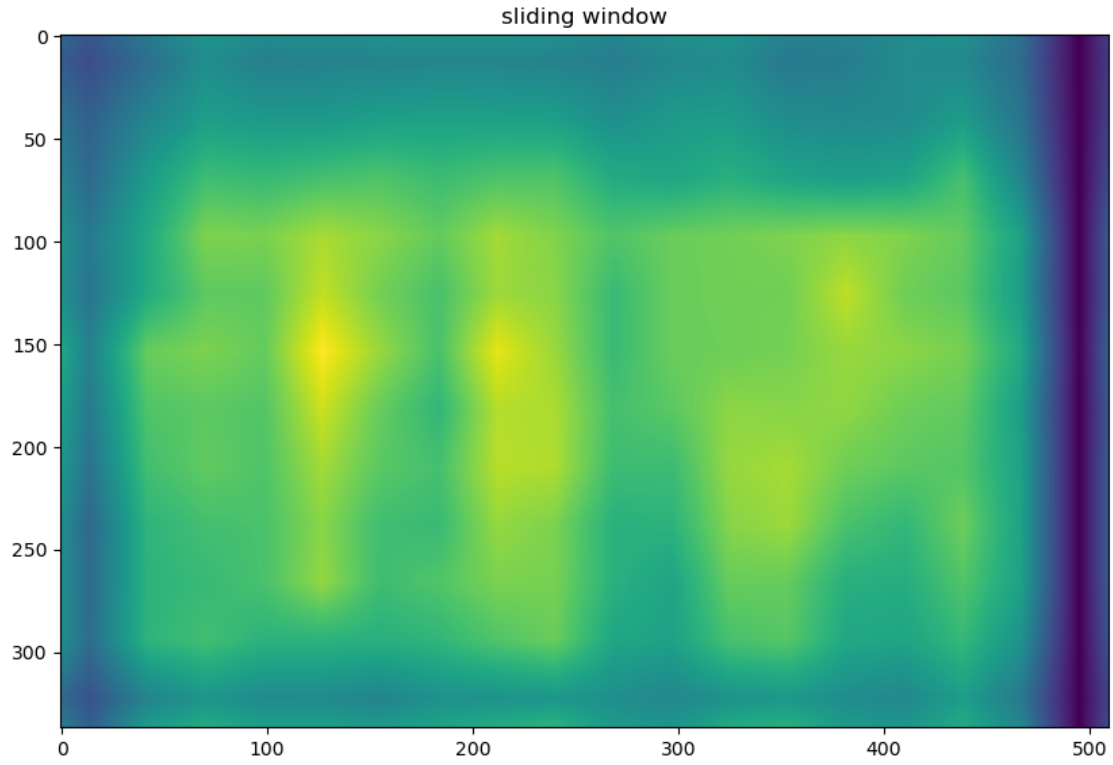
[4]:
```
image_path = 'image_0001.jpg'

image = color.rgb2gray(io.imread(image_path))
image = rescale(image, 0.8)

(hogFeature, hogImage) = hog_feature(image)

(winH, winW) = face_shape
(score, r, c, response_map) = sliding_window(image, face_feature, stepSize=30,␣
 ↪windowSize=face_shape)
crop = image[r:r+winH, c:c+winW]

plot_part2(image, r, c, response_map, winW, winH)
```


0
50
100
150
200
250
0
50
100
150
200
250
300
350
400
SIMULINK
TOOLBOX
TOOLBOX
MATLAB
SIMULINK
SIMULINK
TOOLBOX


sliding window
0
50
100
150
200
250
0
50
100
150
200
250
300
350
400

Sliding window successfully found the human face in the above example. However, in the cell below, we are only changing the scale of the image, and you can see that sliding window does not work once the scale of the image is changed.

[5]:
```
image_path = 'image_0001.jpg'
image = color.rgb2gray(io.imread(image_path))
image = rescale(image, 1.0)

(winH, winW) = face_shape
(score, r, c, max_response_map) = sliding_window(image, face_feature,␣
 ↪stepSize=30, windowSize=face_shape)

crop = image[r:r+winH, c:c+winW]

plot_part2(image, r, c, max_response_map, winW, winH)
```



4 Part 3: Image Pyramids (25 points)

In order to make sliding window work for different scales of images, you need to implement image pyramids where you resize the image to different scales and run the sliding window method on each resized image. This way you scale the objects and can detect both small and large objects.

4.0.1 3.1 Image Pyramid (10 points)

Implement **`pyramid`** function in `detection.py`, this will create pyramid of images at different scales. Run the following code, and you will see the shape of the original image gets smaller until it reaches a minimum size.

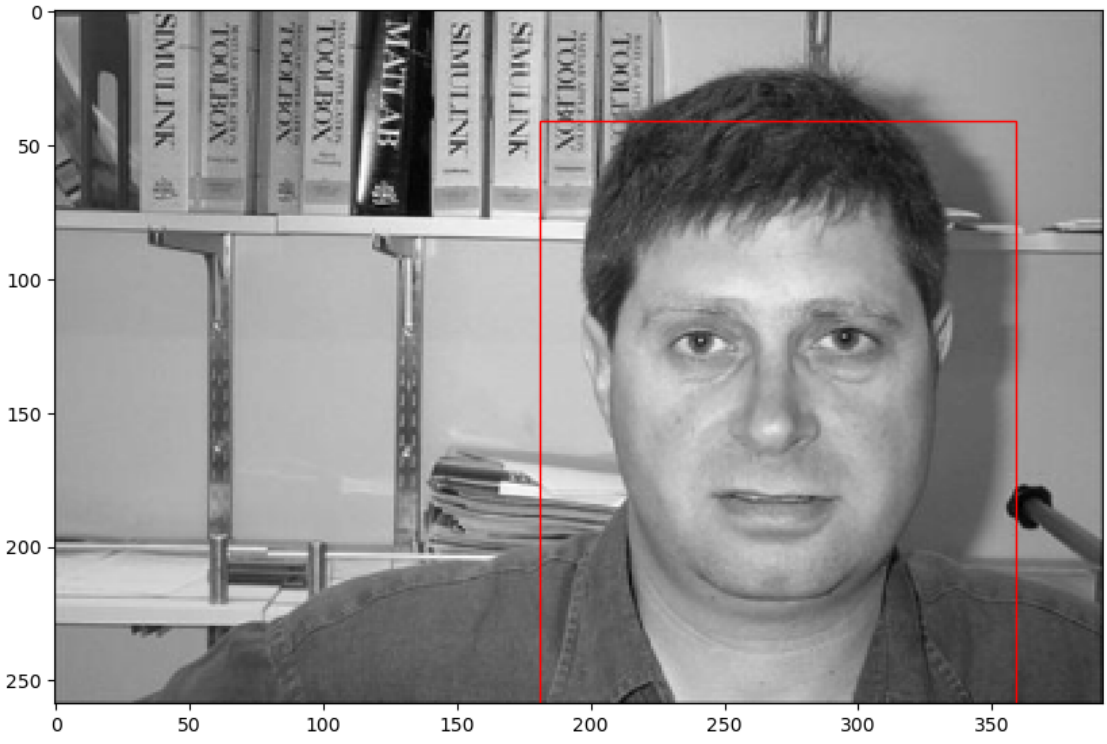
[6]:
```
image_path = 'image_0001.jpg'

image = color.rgb2gray(io.imread(image_path))
image = rescale(image, 1.2)

images = pyramid(image, scale = 0.9)

plot_part3_1(images)
```

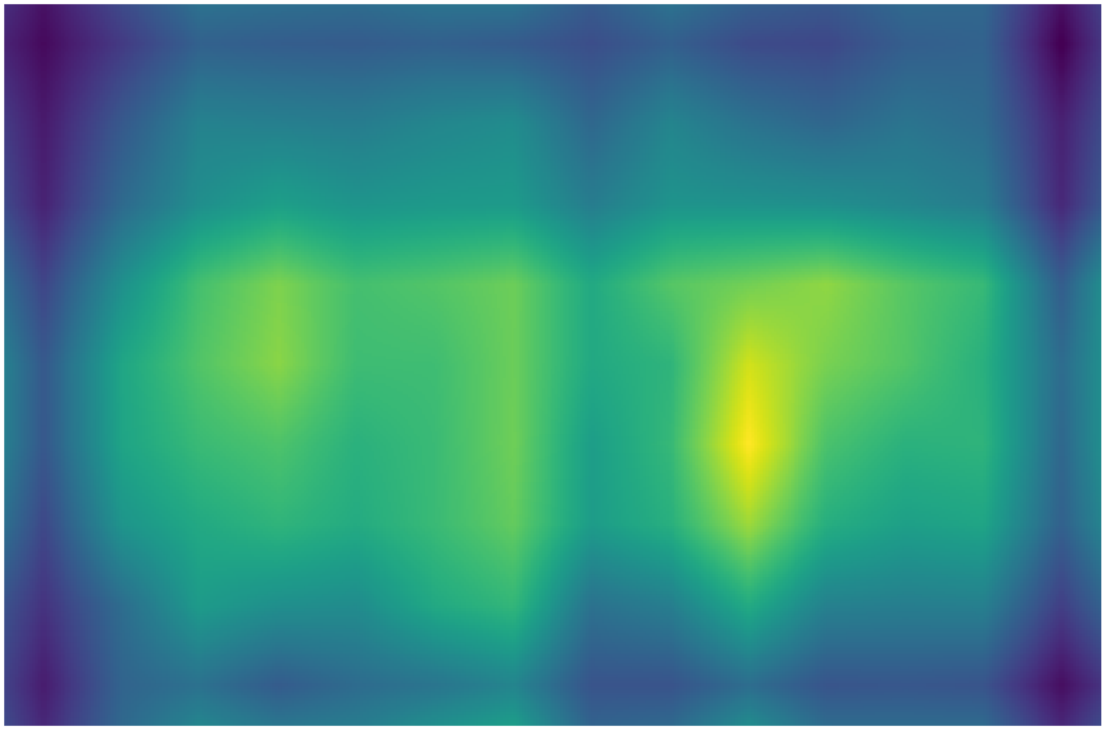
4.0.2 3.2 Pyramid Score (15 points)

After getting the image pyramid, we will run sliding window on all the images to find a place that gets the highest score. Implement **`pyramid_score`** function in `detection.py`. It will return the highest score and its related information in the image pyramids.

[7]:
```
image_path = 'image_0001.jpg'

image = color.rgb2gray(io.imread(image_path))
```

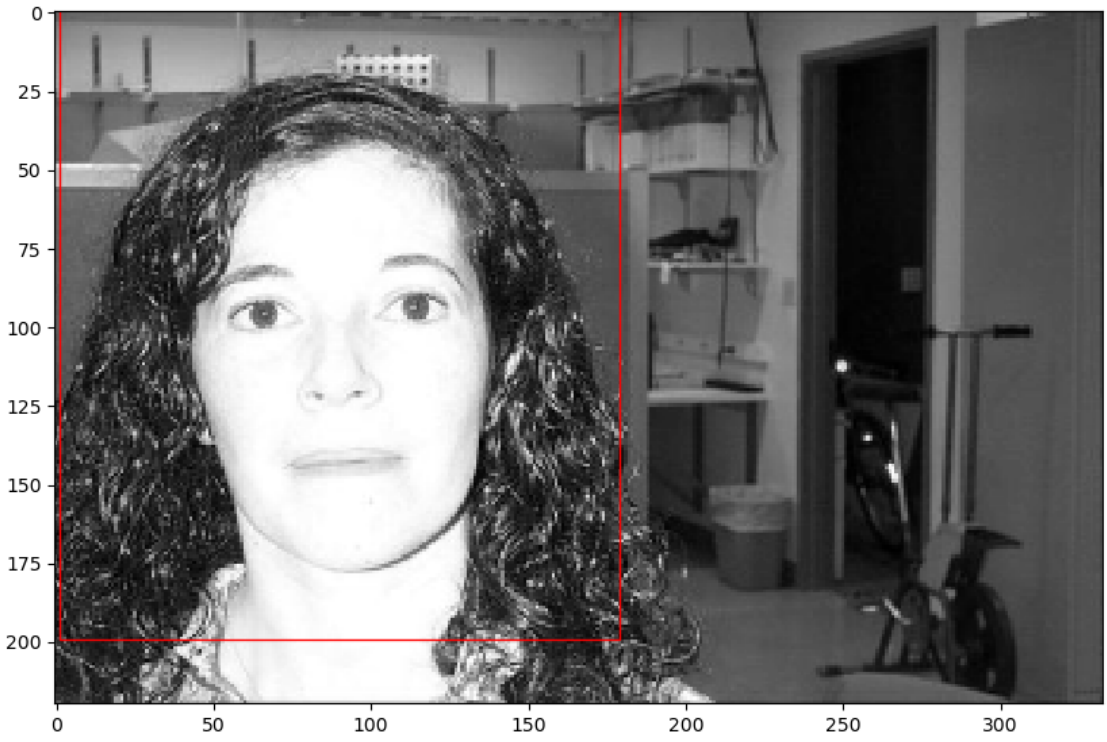
```
image = rescale(image, 1.2)

(winH, winW) = face_shape
max_score, maxr, maxc, max_scale, max_response_map = pyramid_score \
        (image, face_feature, face_shape, stepSize = 30, scale=0.8)

plot_part3_2(image, max_scale, winW, winH, maxc, maxr, max_response_map)
```

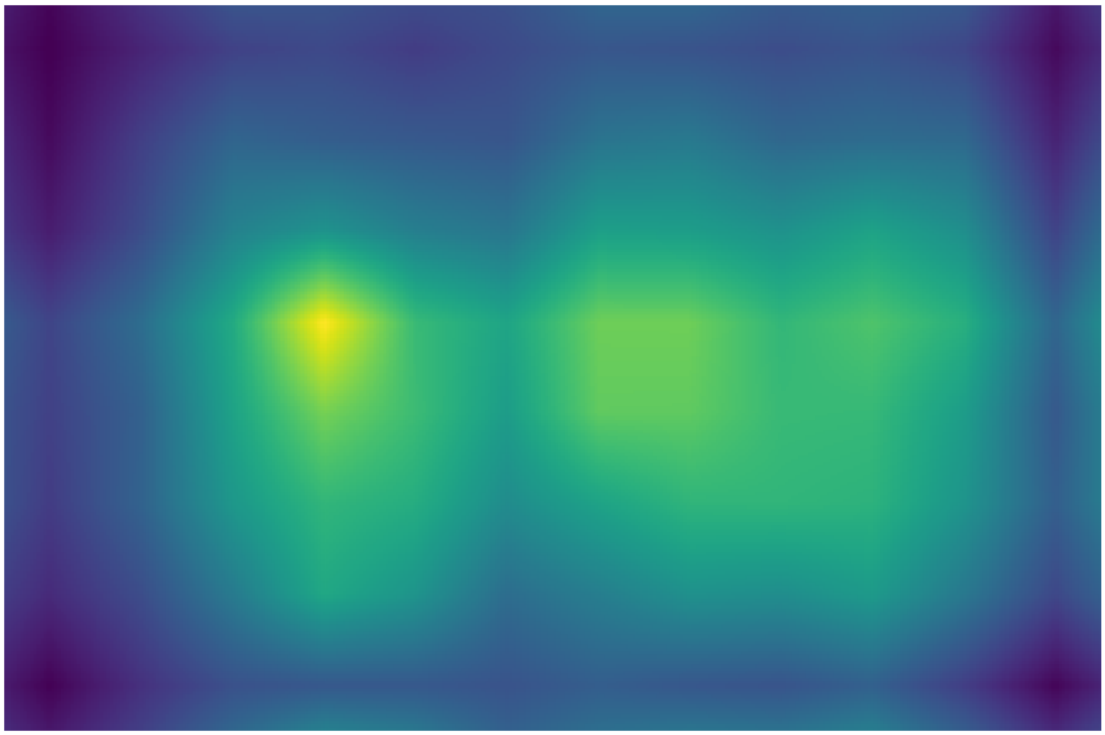




From the above example, we can see that image pyramid has fixed the problem of scaling. Then in the example below, we will try another image and implement a deformable parts model.

[8]:
```
image_path = 'image_0338.jpg'
image = color.rgb2gray(io.imread(image_path))
image = rescale(image, 1.0)

(winH, winW) = face_shape

max_score, maxr, maxc, max_scale, max_response_map = pyramid_score \
    (image, face_feature, face_shape, stepSize = 30, scale=0.8)

plot_part3_2(image, max_scale, winW, winH, maxc, maxr, max_response_map)
```


0
25
50
75
100
125
150
175
200
0
50
100
150
200
250
300

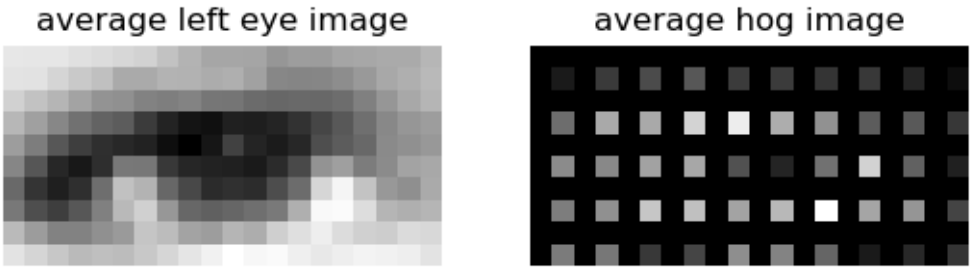


5 Part 4: Deformable Parts Detection

In order to solve the problem above, you will implement deformable parts model in this section, and apply it on human faces. The first step is to get a detector for each part of the face, including left eye, right eye, nose and mouth. For example for the left eye, we have provided the groundtruth location of left eyes for each image in the `\face` directory. This is stored in the `lefteyes` array with shape `(n,2)`, each row is the `(r,c)` location of the center of left eye. You will then find the average hog representation of the left eyes in the images.

Run through the following code to get a detector for left eyes.

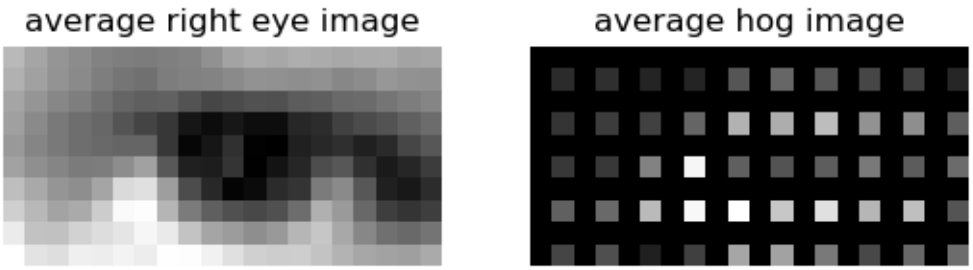
[9]:
```
image_paths = fnmatch.filter(os.listdir('./face'), '*.jpg')

parts = read_facial_labels(image_paths)
lefteyes, righteyes, noses, mouths = parts

# Typical shape for left eye
lefteye_h = 10
lefteye_w = 20

lefteye_shape = (lefteye_h, lefteye_w)

avg_lefteye = get_detector(lefteye_h, lefteye_w, lefteyes, image_paths)
(lefteye_feature, lefteye_hog) = hog_feature(avg_lefteye, pixel_per_cell=2)

plot_part4(avg_lefteye, lefteye_hog, 'left eye')
```



Run through the following code to get a detector for right eye.

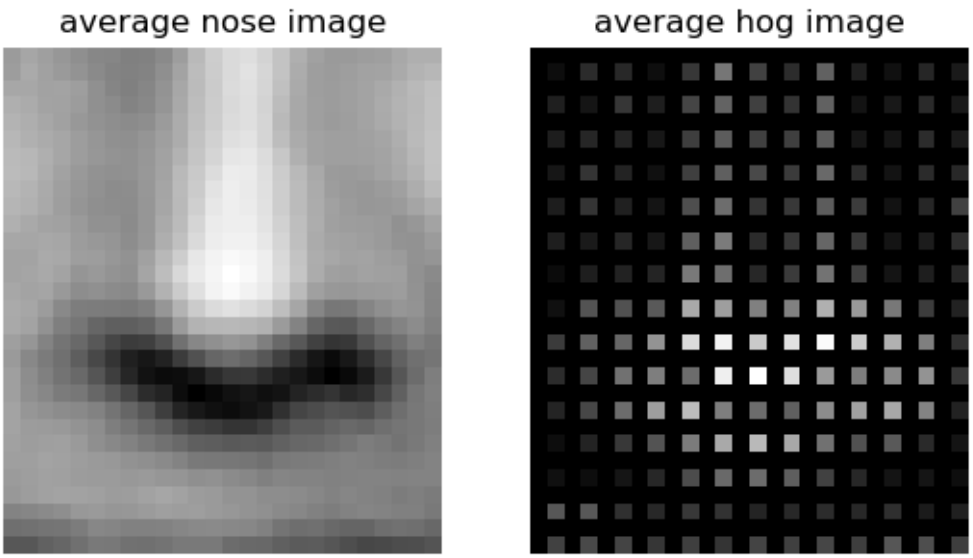
[10]:
```
righteye_h = 10
righteye_w = 20

righteye_shape = (righteye_h, righteye_w)

avg_righteye = get_detector(righteye_h, righteye_w, righteyes, image_paths)

(righteye_feature, righteye_hog) = hog_feature(avg_righteye, pixel_per_cell=2)
```

```
plot_part4(avg_righteye, righteye_hog, 'right eye')
```



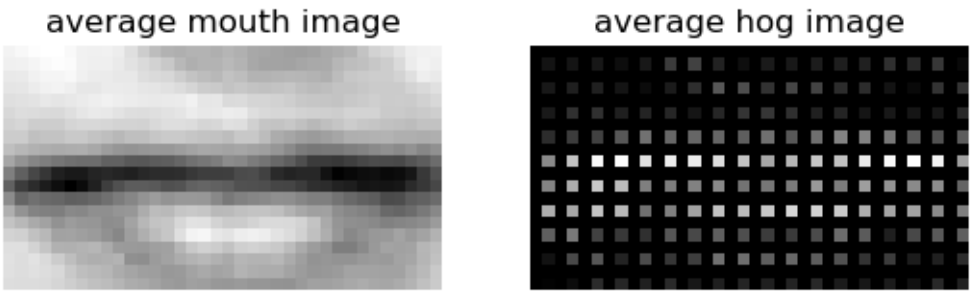
Run through the following code to get a detector for nose.

[11]:
```
nose_h = 30
nose_w = 26

nose_shape = (nose_h, nose_w)

avg_nose = get_detector(nose_h, nose_w, noses, image_paths)

(nose_feature, nose_hog) = hog_feature(avg_nose, pixel_per_cell=2)

plot_part4(avg_nose, nose_hog, 'nose')
```



Run through the following code to get a detector for mouth

[12]:
```
mouth_h = 20
mouth_w = 36

mouth_shape = (mouth_h, mouth_w)

avg_mouth = get_detector(mouth_h, mouth_w, mouths, image_paths)

(mouth_feature, mouth_hog) = hog_feature(avg_mouth, pixel_per_cell=2)

detectors_list = [lefteye_feature, righteye_feature, nose_feature,␣
 ↪mouth_feature]

plot_part4(avg_mouth, mouth_hog, 'mouth')
```



6 Part 5: Human Parts Location (15points)

6.0.1 5.1 Compute displacement (10 points)

Implement **compute_displacement** to get an average shift vector mu and standard deviation sigma for each part of the face. The vector mu is the distance from the main center, i.e the center of the face, to the center of the part.

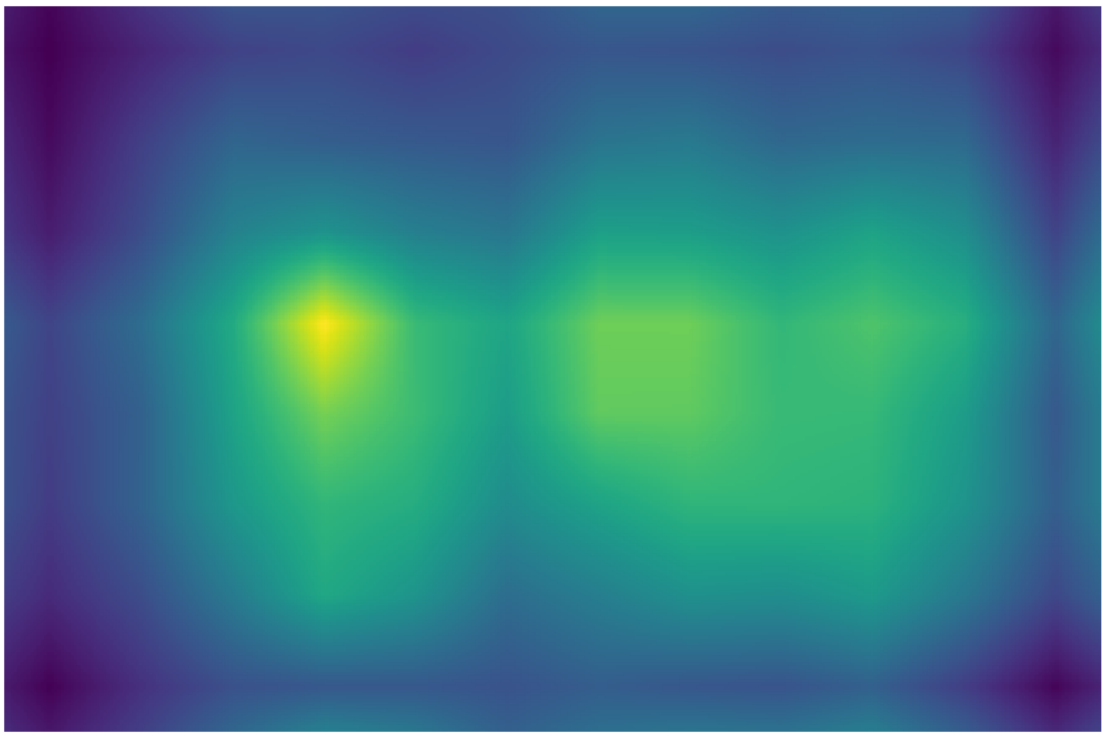
[13]:
```
# test for compute_displacement
test_array = np.array([[0,1],[1,2],[2,3],[3,4]])
test_shape = (6,6)
mu, std = compute_displacement(test_array, test_shape)
assert(np.all(mu == [1,0]))
assert(np.sum(std-[ 1.11803399,  1.11803399])<1e-5)
print("Your implementation is correct!")
```

```
Your implementation is correct!
```

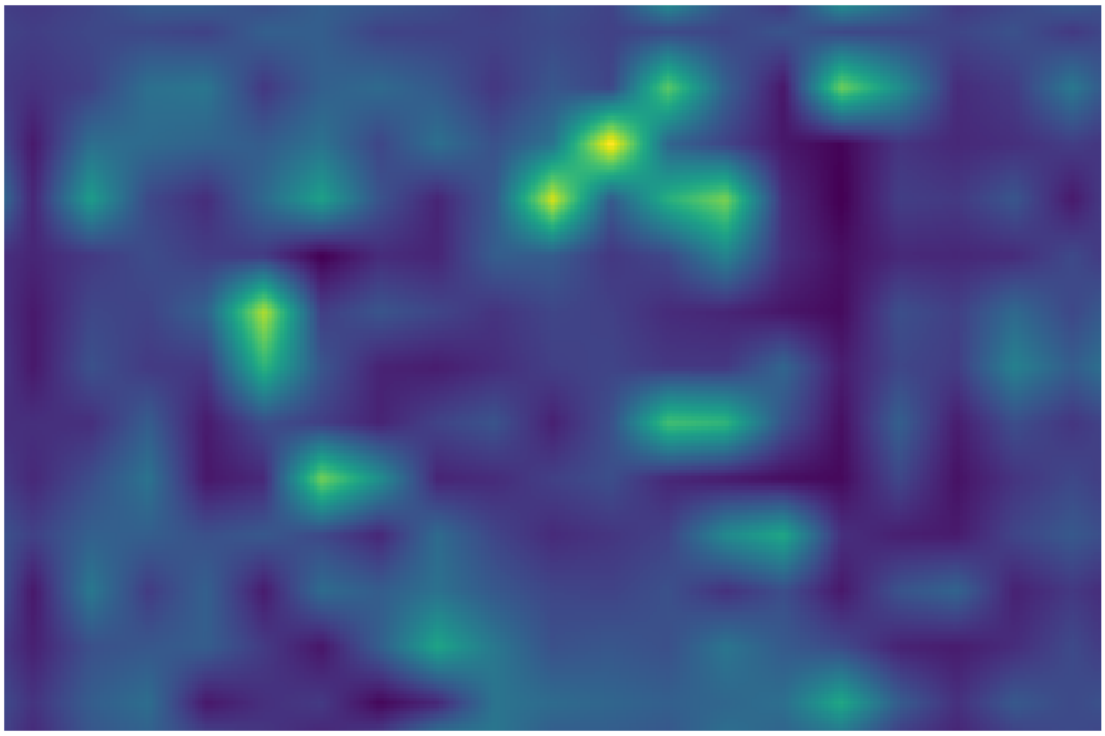
[14]:
```
lefteye_mu, lefteye_std = compute_displacement(lefteyes, face_shape)
righteye_mu, righteye_std = compute_displacement(righteyes, face_shape)
nose_mu, nose_std = compute_displacement(noses, face_shape)
mouth_mu, mouth_std = compute_displacement(mouths, face_shape)
```

After getting the shift vectors, we can run our detector on a test image. We will first run the following code to detect each part of left eye, right eye, nose and mouth in the image. You will see a response map for each of them.

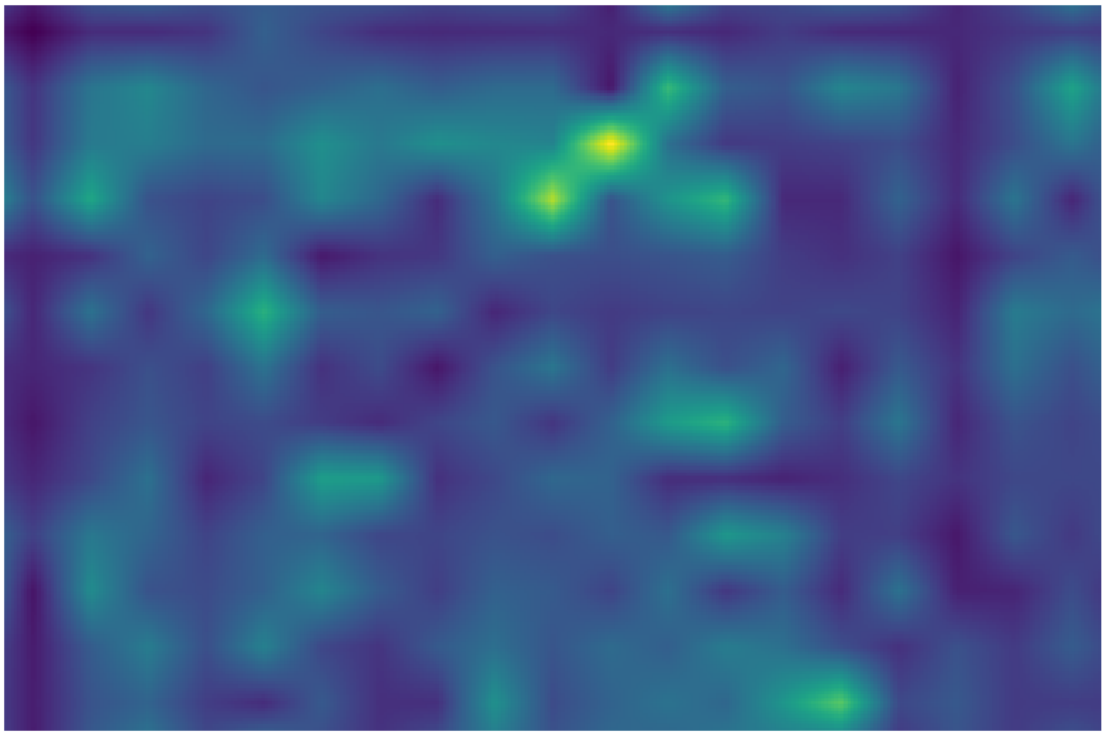
[15]:
```
image_path = 'image_0338.jpg'
image = color.rgb2gray(io.imread(image_path))
image = rescale(image, 1.0)

(face_H, face_W) = face_shape
max_score, face_r, face_c, face_scale, face_response_map = pyramid_score\
    (image, face_feature, face_shape,stepSize = 30, scale=0.8)

plot_part5_1(face_response_map)
```



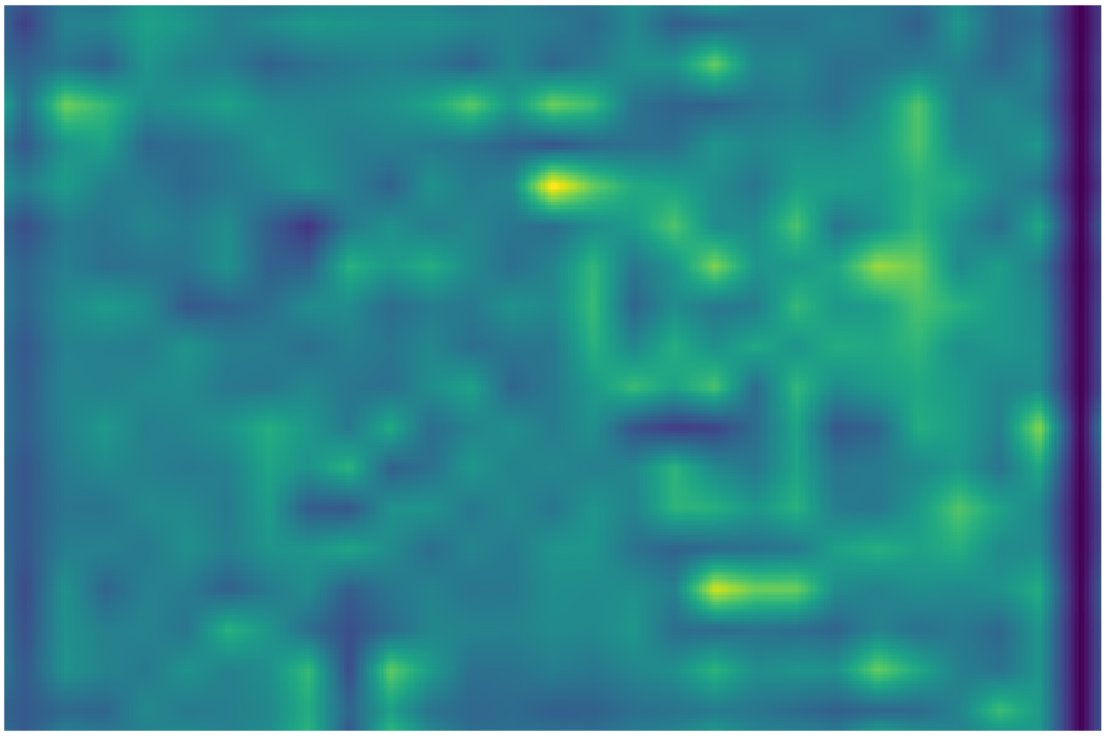
[16]:
```
max_score, lefteye_r, lefteye_c, lefteye_scale, lefteye_response_map = \
    pyramid_score(image, lefteye_feature,lefteye_shape, stepSize = 20,scale=0.
 ↪9, pixel_per_cell = 2)

lefteye_response_map = resize(lefteye_response_map, face_response_map.shape)

plot_part5_1(lefteye_response_map)
```



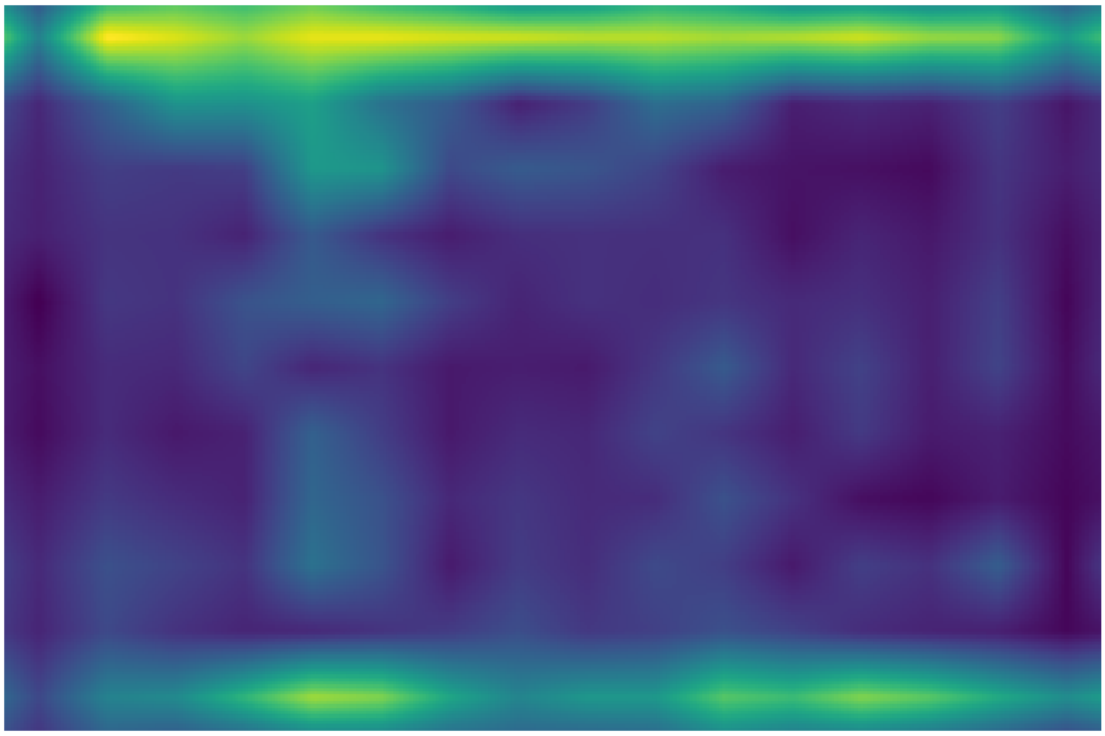
[17]:
```
max_score, righteye_r, righteye_c, righteye_scale, righteye_response_map = \
    pyramid_score (image, righteye_feature, righteye_shape, stepSize =␣
 ↪20,scale=0.9, pixel_per_cell=2)

righteye_response_map = resize(righteye_response_map, face_response_map.shape)

plot_part5_1(righteye_response_map)
```



[18]:
```
max_score, nose_r, nose_c, nose_scale, nose_response_map = \
    pyramid_score (image, nose_feature, nose_shape, stepSize = 20,scale=0.9,␣
 ↪pixel_per_cell = 2)

nose_response_map = resize(nose_response_map, face_response_map.shape)

plot_part5_1(nose_response_map)
```



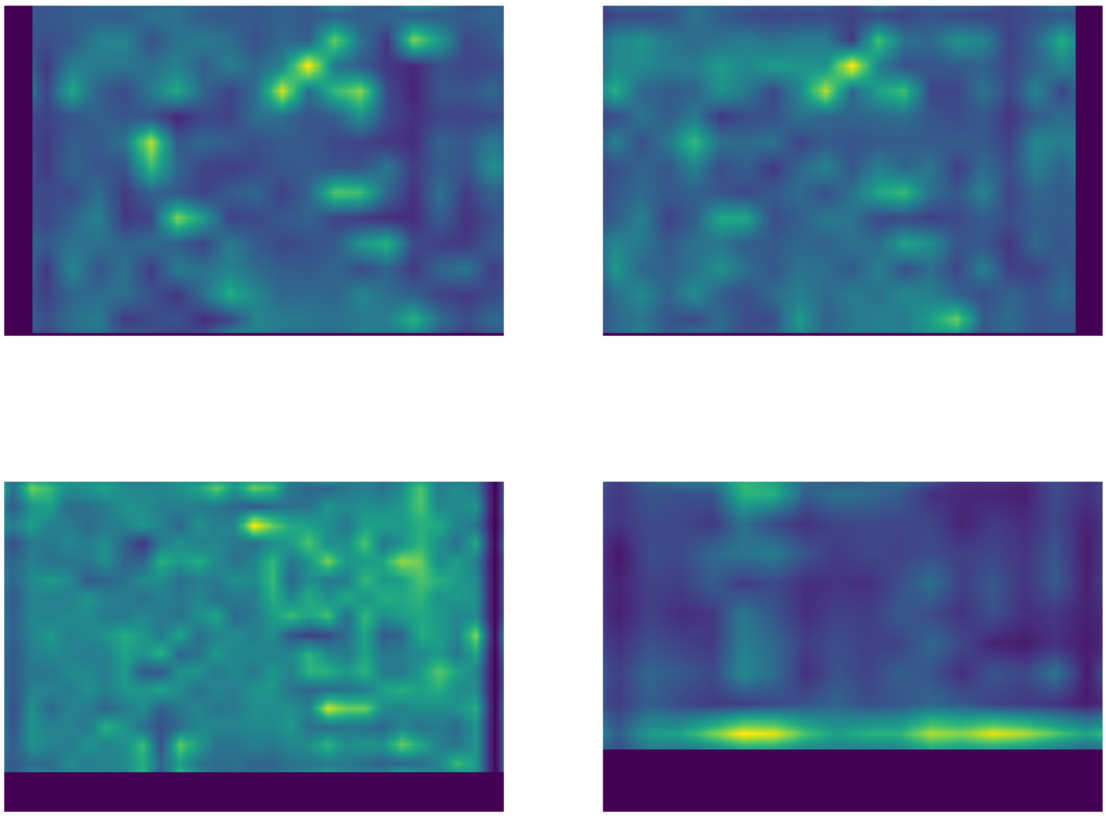
[19]:
```
max_score, mouth_r, mouth_c, mouth_scale, mouth_response_map =\
    pyramid_score (image, mouth_feature, mouth_shape, stepSize = 20,scale=0.9,␣
 ↪pixel_per_cell = 2)

mouth_response_map = resize(mouth_response_map, face_response_map.shape)

plot_part5_1(mouth_response_map)
```



6.0.2 5.2 Shift heatmap (5 points)

After getting the response maps for each part of the face, we will shift these maps so that they all have the same center as the face. We have calculated the shift vector mu in `compute_displacement`, so we are shifting based on vector mu. Implement `shift_heatmap` function in `detection.py`.

[21]:
```
face_heatmap_shifted = shift_heatmap(face_response_map, [0,0])

lefteye_heatmap_shifted = shift_heatmap(lefteye_response_map, lefteye_mu)
righteye_heatmap_shifted = shift_heatmap(righteye_response_map, righteye_mu)
nose_heatmap_shifted = shift_heatmap(nose_response_map, nose_mu)
mouth_heatmap_shifted = shift_heatmap(mouth_response_map, mouth_mu)

plot_part5_2(lefteye_heatmap_shifted, righteye_heatmap_shifted,
                 nose_heatmap_shifted, mouth_heatmap_shifted)
```



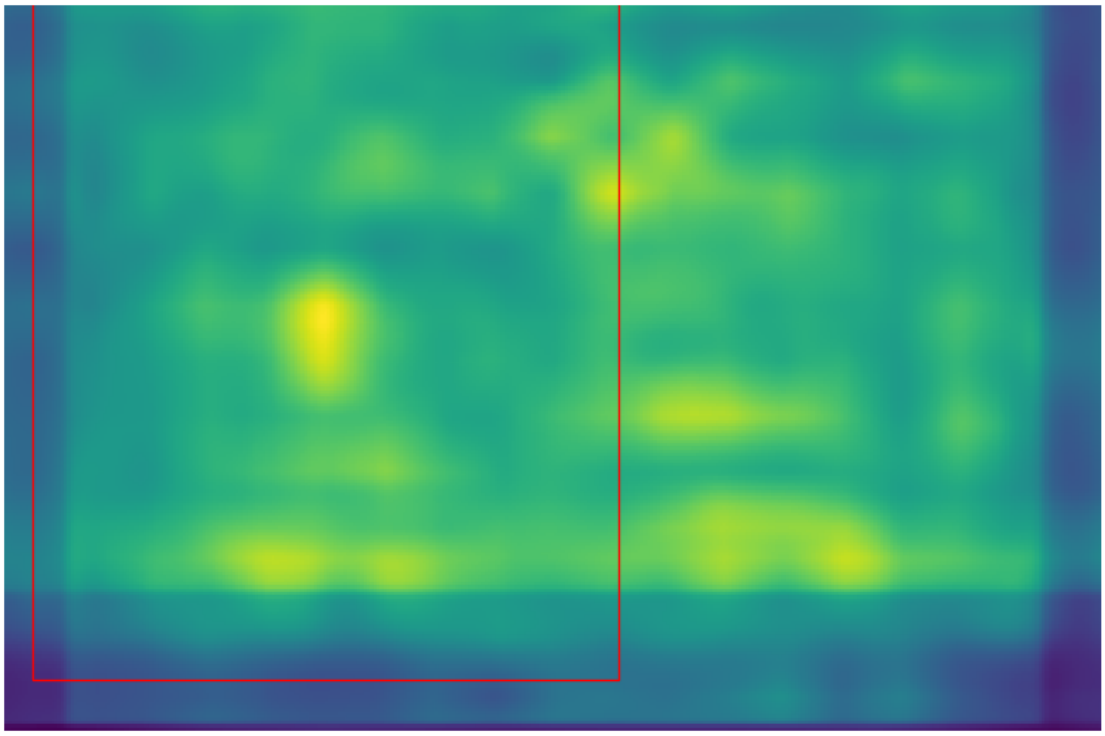
7 Part 6: Gaussian Filter (20 points)

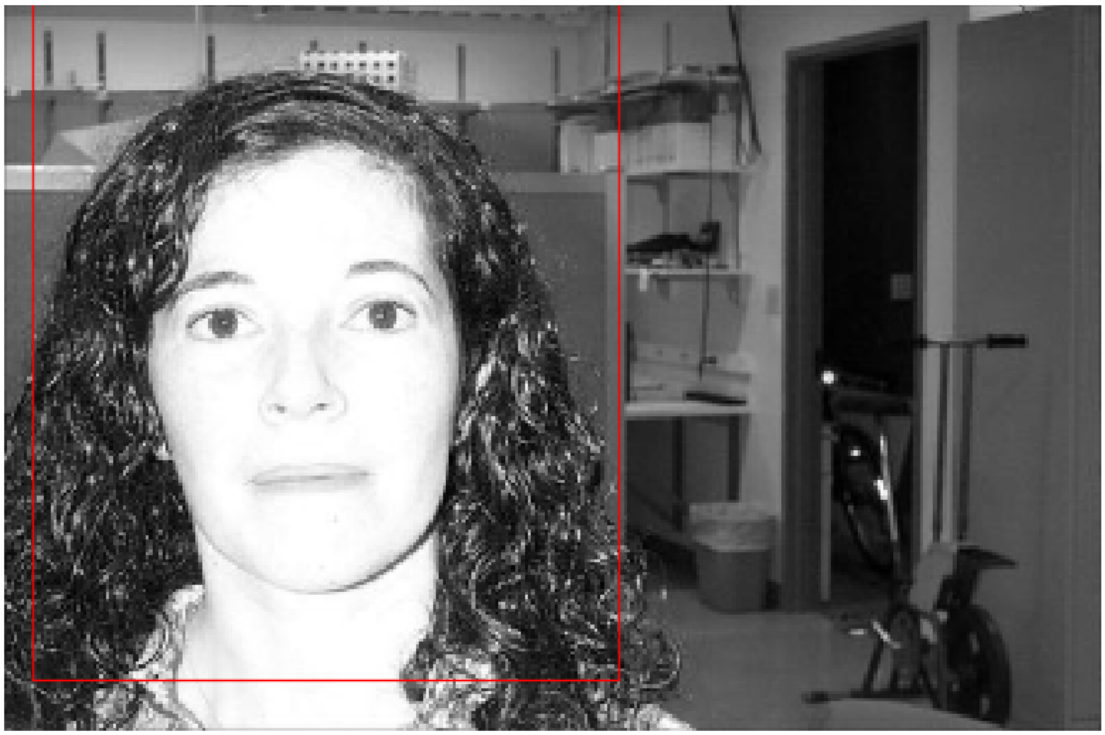
7.1 Part 6.1 Gaussian Filter

In this part, apply gaussian filter convolution to each heatmap. Blur by kernel of standard deviation sigma, and then add the heatmaps of the parts with the heatmap of the face. On the combined heatmap, find the maximum value and its location. You can use function provided by skimage to implement **gaussian_heatmap**.

[24]:
```
heatmap_face= face_heatmap_shifted

heatmaps = [lefteye_heatmap_shifted,
            righteye_heatmap_shifted,
            nose_heatmap_shifted,
            mouth_heatmap_shifted]
sigmas = [lefteye_std, righteye_std, nose_std, mouth_std]

heatmap, i , j = gaussian_heatmap(heatmap_face, heatmaps, sigmas)
print(heatmap.shape, image.shape)
plot_part6_1(winH, winW, heatmap, image, i, j)
```

```
(220, 333) (344, 520)
```





7.2 6.2 Result Analysis (10 points)

Does your DPM work on detecting human faces? Can you think of a case where DPM may work better than the detector we had in part 3 (sliding window + image pyramid)? You can also have examples that are not faces.

Your Answer: Write your answer in this markdown cell. The DPM does seem to work quite well with detecting faces. It's a really intuitive process where we search for a face and then each body part separately to verify that we really have found the right face. However, human faces are (mostly) homogeneous. We (mostly) have two eyes, one mouth, in around the same place relative to each other (videogames like to exaggerate this), two eyebrows, one nose. It is exactly the kind of thing you would expect sliding window + image pyramid to work really well on, and DPM is a generalization of that so of course it works well too. This analysis also helps us discover where sliding window + image pyramid would work worse than DPM. When things are not approximately all at the same place as each other, that's when things get tough. For example, if we wanted to detect a house, this would be very hard for sliding window + image pyramid. Houses come in so many different shapes and sizes. Some houses are very small while others are extremely large. This would make sliding window + image pyramid not work very well because, however could it know that roofs could be so long? On the other hand, every house has a front door and a couple of windows, and most have grass in the front. DPM could have the general idea of a house, and then look closely for each of those things separately to tell if we really had a house or not. This would work much better than image pyramid + sliding window.

7.3 Extra Credit (1 point)

You have tried detecting one face from the image, and the next step is to extend it to detecting multiple occurences of the object. For example in the following image, how do you detect more than one face from your response map? Implement the function **`detect_multiple`**, and write code to visualize your detected faces in the cell below.

[ ]:
```
image_path = 'image_0002.jpg'
image = color.rgb2gray(io.imread(image_path))
plt.imshow(image)
plt.show()
```

[ ]:
```
image_path = 'image_0002.jpg'
image = color.rgb2gray(io.imread(image_path))
heatmap = get_heatmap (image,face_feature, face_shape,detectors_list, parts )

plt.imshow(heatmap, cmap='viridis', interpolation='nearest')
plt.show()
```

[ ]:
```
detected_faces = detect_multiple(image, heatmap)

# Visualize your detected faces

### YOUR CODE HERE
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
```

```
pass

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
### END YOUR CODE
```