CS/NEUR125 Brains, Minds, and Machines

Lab 5: Recognizing Faces by Matching Templates

Due: Wednesday, March 1

This lab introduces new MATLAB programming concepts and commands, and provides practice with this new material through the implementation of a face recognition strategy based on ***template matching***. You will test this implementation with images of famous Wellesley alumnae, Hillary Clinton '69, Madeleine Albright '59, Jane Bolin '28, Soong Mei-Ling '17, Pamela Melroy '83, and Diane Sawyer '67. The first part of the lab includes some MATLAB review and introduces the new concepts and commands. In the second part of the lab, you will complete a MATLAB script to implement and test the template matching strategy, which you will submit. The second part also has questions related to the results of your testing and comparison of this face recognition strategy to other methods that we explored.

To begin, create a copy of this Google document, modify the title of the copy to include your partner names, and share the copy between partners, as you did in previous labs. A description of the code submission and questions for you to answer are shown in blue.

Start MATLAB on the lab Mac that you are sharing, and use **Fetch** to download the folder named **faceLab** from the CS file server to the Desktop on your Mac. You will find this folder inside the cs125/download folder in your individual account on the CS server. For this lab, **set the Current Folder in MATLAB to the faceLab folder**. To do this, you can first set your Current Folder to the Desktop of your Mac as you did in previous labs, and then double-click on the faceLab folder that is listed among the contents of the Current Folder on the left side of the MATLAB window.

I. Exploring New MATLAB Programming Concepts and Commands

(a) Review of variables, vectors, and indexing

In previous labs, you learned how to create a variable in MATLAB that is assigned to a vector of values, and how to access the contents of individual locations of a vector using an index. You also learned how to find the minimum and maximum values stored in a vector, and the location of these extreme values. The following examples illustrate these concepts:

```
>> nums = [5  0  8  -3  6]
nums =
    5    0    8   -3    6
>> nums(2)
ans =
    0
>> [minVal minInd] = min(nums)
```

```
minVal =
   -3
minInd =
    4
>> nums(minInd)
ans =
   -3
>> [maxVal maxInd] = max(nums)
maxVal =
    8
maxInd =
    3
>> nums(maxInd)
ans =
    8
>> nums(4) = 1
nums =
    5    0    8    1    6
```

In the last example above, suppose you instead entered nums(1) = 4. How would the results change?

What do you expect to happen when the following statements are evaluated? Create a 5-element vector named **nums** in MATLAB and try out both statements.

```
>> nums(9)
```

```
>> nums(9) = 5
```

Hmmm… what does MATLAB do when you try to store a new value at a vector index that is beyond the length of the vector?

(b) Creating vectors and matrices with zeros and ones

There are times when it is helpful to create an initial vector or matrix of a particular size, to store new contents at a later time. The built-in **zeros** and **ones** functions can be used to create vectors or matrices that initially store all 0s or 1s, respectively. They each have two inputs corresponding to the desired number of rows and columns. Enter the following examples and view the results:

```
>> vec1 = zeros(1,4)
```

```
>> vec2 = zeros(4,1)
```

```
>> vec3 = ones(1,6)
```

```
>> mx1 = zeros(2,3)
```

vec1 and vec3 are **row vectors** and vec2 is a **column vector** - we can also think of them as one-dimensional matrices, but we will often use the separate words **vector** and **matrix**. Try to predict the new contents of vec1 and mx1 before executing the following two statements:

```
>> vec1(3) = 7

>> mx1(2,1) = 4
```

(c) Performing arithmetic computations on entire vectors and matrices

We also learned that arithmetic operations (+, -, *, /) can be applied to an entire vector or matrix all at once, as in the following example:

```
>> nums
nums =
    5     0     8     1     6
>> nums*2
ans =
    10     0    16     2    12
```

After executing the second statement above, suppose you now enter the following expression - what do you expect to see, 8 or 16? Why?

```
>> nums(3)
```

Construct two statements that create a matrix with 3 rows and 4 columns, whose locations all contain the number 8. Use the **zeros** function in one statement and **ones** in the other.

Multiple vectors and matrices can be combined with arithmetic operations that are applied in an element-by-element fashion. The following examples illustrate adding and subtracting the contents of two vectors or two matrices **(of the same size!):**

```
>> nums1 = [3  1   2  4];
>> nums2 = [4  5  -1  0];
>> nums1 + nums2
ans =
    7     6     1     4
>> nums1 - nums2
ans =
   -1    -4     3     4
>> mx1 = [1 3 2; 0 4 2]
mx1 =
    1     3     2
    0     4     2
>> mx2 = [0 3 5; 2 1 3]
mx2 =
    0     3     5
    2     1     3
>> mx1 + mx2
```

```
ans =
    1    6    7
    2    5    5
>> diff = mx1 - mx2
diff =
    1    0   -3
   -2    3   -1
```

There are many additional MATLAB functions that can be performed on an entire vector or matrix all at once. Two examples that we will use in this lab are **abs** and **mean**. Try the following examples:

```
>> nums = [-5  4  2  -3  -1]

>> absNums = abs(nums)

>> avg = mean(absNums)
```

Not let's try the same functions with a matrix (note the **semi-colon** in the first statement!):

```
>> mx1 = [-1 0 -3; 5 -2 4]

>> absMx1 = abs(mx1)

>> mean(absMx1)
```

Hmmm… what did MATLAB return, when you applied the **mean** function to a matrix? What did MATLAB do in this case? In order to compute a single average value for a 2D matrix, you can use one of the following two strategies (try to dissect what MATLAB is doing in each case):

```
>> mean(mean(absMx1))
ans =
   2.5000
>> mean(absMx1(:))
ans =
   2.5000
```

(d) Printing information with disp

It is often helpful for your program to print information to the user. You can print out the values of variables in a script by omitting the semi-colon at the end of an assignment statement, or writing the name of the variable without a semi-colon, as we did in the examples above. You can also use the **disp** function. The input to disp is a **string** to be printed. In MATLAB, a string is a sequence of characters typed within single quotes, which can include letters, digits, spaces, punctuation, and other symbols. We previously used strings to label plots. In code, MATLAB displays strings in purple. To print a single string, type the desired text inside single quotes:

```
>> disp('computing final results…')
computing final results…
```

The input string can also be the result of concatenating multiple parts, which allows us to include the values of variables in the string, for example. Square brackets **[…]** are used to concatenate the parts, which should be separated by spaces or commas. If a numerical value is being concatenated into the string, we need to use the **num2str** function to convert the number to a string. Try the examples below, carefully noting the spacing and punctuation.

```
>> name = 'Mike';
>> favNum = 7;
>> disp(['Hi '  name  ', I see your favorite number is '  num2str(favNum)])
Hi Mike, I see your favorite number is 7
```

Suppose you create a vector named nums that is assigned to [5 2 9 3 7]. Construct a disp statement that produces the following output:

The average of nums is 5.2 and the maximum is 9

(e) Storing different kinds of things in a cell array

We have so far used vectors to store only numerical values that we can access using an index. A **cell array** can store a greater variety of things that we can also access with an index. Curly braces **{ … }** are used to construct a cell array and access the contents of a particular location. The following statement creates a cell array storing the names of students in the class:

```
>> classNames = {'Adrienne',  'Grace',  'Hillary',  'Imogen',  'Kiana',  'Zuzanna'}
classNames =
    'Adrienne'    'Grace'    'Hillary'    'Imogen'    'Kiana'    'Zuzanna'
>> classNames{2}
ans =
Grace
>> disp([classNames{1}  ' and '  classNames{6}  ' are working together today'])
Adrienne and Zuzanna are working together today
```

Suppose you accidentally use square brackets instead of curly braces to combine the class names, as in the example below. What value will get assigned to classNames?

```
>> classNames = ['Adrienne',  'Grace',  'Hillary',  'Imogen',  'Kiana',  'Zuzanna']
```

We will also use cell arrays to store a collection of images, for example:

```
>> faceIms = {sawyer, soong, melroy, bolin, clinton, albright};
>> imshow(faceIms{6})
```

(f) Repeating actions with a for loop

Finally, situations often arise where we want to repeat an action multiple times, perhaps varying something with each new repetition. The simplest way to repeat actions is with a **for** statement, which has the following general form:

```
for variable-name = vector-of-values
        code statements to repeat
end
```

The for and end are *keywords* that MATLAB displays in blue. Suppose we put the following **for loop** in a MATLAB script file named testing.m. When run, this code repeats a print statement four times:

```
for i = 1:4
   disp('here we go again')
end
```

```
>> testing
here we go again
here we go again
here we go again
here we go again
```

When executed, the variable **i** (referred to as the *control variable* for the loop) will be assigned, one by one, to each of the values in the vector 1:4, and the code inside the **for** statement (the *body of the loop*) is executed for each value of i. Any valid variable name can be used for the control variable, and any vector of values can be used to control the number of times the loop is executed. The value of i can also be used inside the loop, as in the following example:

```
classNames = {'Adrienne',  'Grace',  'Hillary',  'Imogen',  'Kiana',  'Zuzanna'};
for i = 1:6
   disp(['hello '  classNames{i}  ', how are you today?'])
   disp(['number of students left: '  num2str(6 - i)])
end
```

```
>> testing
hello Adrienne, how are you today?
number of students left: 5
hello Grace, how are you today?
number of students left: 4
hello Hillary, how are you today?
number of students left: 3
hello Imogen, how are you today?
number of students left: 2
hello Kiana, how are you today?
number of students left: 1
hello Zuzanna, how are you today?
number of students left: 0
```

II. Recognizing the Faces of Famous Wellesley Alumnae

Wellesley College is proud to have some very distinguished alumnae! For this part of the lab, you will complete a MATLAB script to recognize the faces of a few special graduates, shown in the figure below: Diane Sawyer ‘67, Soong Mei-Ling ‘17, Pamela Melroy ‘83, Jane Bolin ‘28, Hillary Clinton ‘69, and Madeleine Albright ‘59. You will see three files in the **faceLab** folder: faceImages.mat, setupFaces.m, and recognize.m. To begin, execute the setupFaces script in the Command Window:

```
>> setupFaces
```

This script loads the faceImages.mat file into the MATLAB workspace, which contains both full-face and cropped images of the alums, and displays all the images in two figure windows. Figure 1 (below) shows a set of 6 full images that comprise the database of “known” alums that will be used for recognition. The name of the matrix that stores each image is shown above each picture. There are also 6 examples of cropped face images, taken from roughly the same region of each face, stored in matrices with the names shown above each cropped picture below. The array of images is displayed in one figure window using the **subplot** command, which you can read about if you’re interested - enter **doc subplot** in the Command Window.

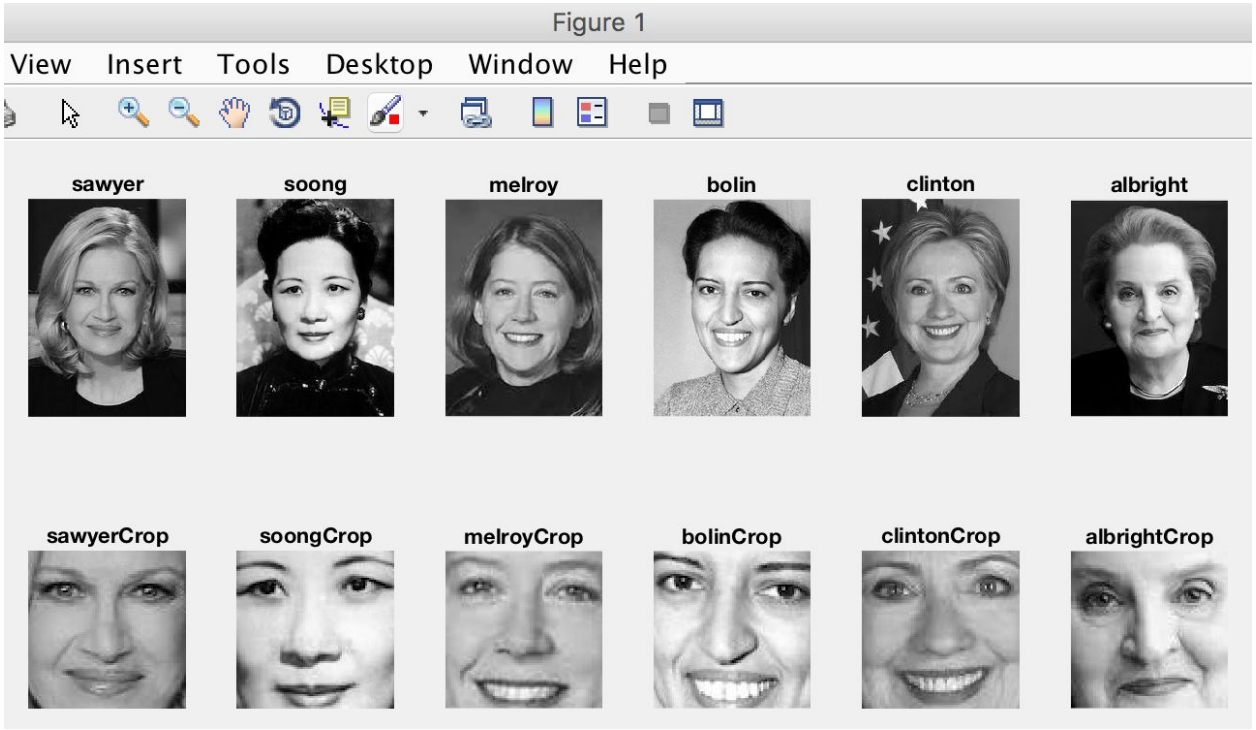


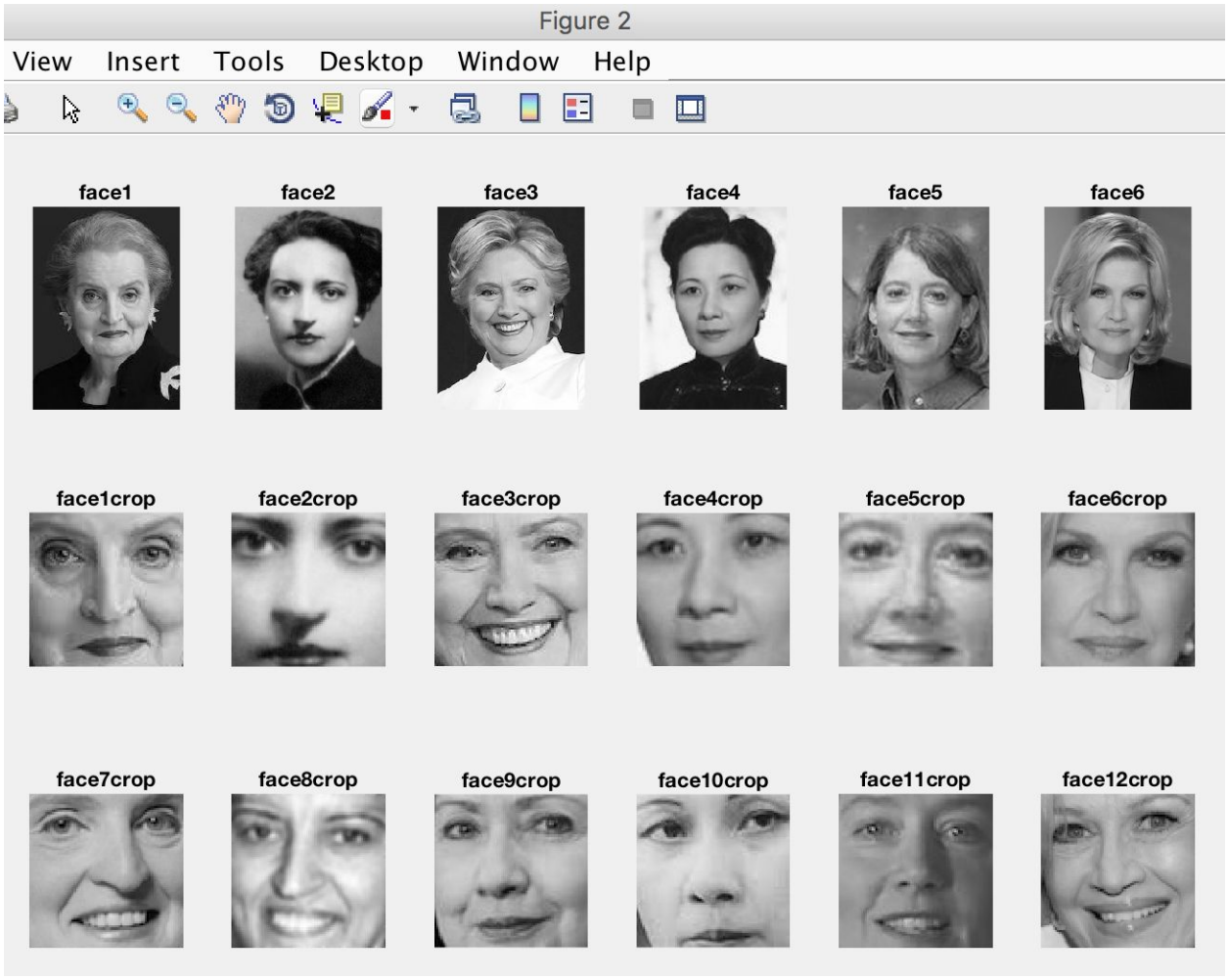
Figure 2, shown on the next page, shows 6 full-face images and 12 samples of cropped faces (two for each alum) to be recognized. ***Note that this is a challenging recognition problem and the strategy you’ll be using is far from perfect!***

Your coding will all be done in the **recognition.m** script file, which you should open in the Editor. Initially this file contains mostly comments. If you view the **setupFaces.m** script, you will

see three assignment statements in the code near the beginning of the file, creating variables that you will refer to in the code that you write in the recognition.m script ***(you do not need to make any changes to the setupFaces.m code file!):***

```
names = {'Diane Sawyer',  'Soong Mei-Ling',  'Pamela Melroy',  'Jane Bolin',  ...
         'Hillary Clinton',  'Madeleine Albright'};
newIms = {face1,  face2,  face3,  face4,  face5,  face6};
newCrops = {face1crop,  face2crop,  face3crop,  face4crop,  face5crop,  face6crop,  ...
            face7crop,  face8crop,  face9crop,  face10crop,  face11crop,  face12crop};
```

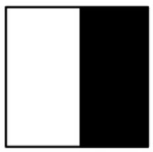
Note the "**...**" at the end of two of the lines, which indicates that the statement continues onto the next line in the file.



(a) Recognizing the full-face images

You will first write code to recognize the 6 full-face images shown above. You can then combine some cutting-and-pasting with minor editing to recognize the cropped images. One strategy we can use to recognize an unknown image is to ***measure the difference between the pattern of brightness values in the unknown image and the patterns of brightness in each of a set***

of known images. We can then select the known image that represents the ***closest match***. Consider a very simple example where we have two known image patterns corresponding to a vertical or horizontal edge, as shown below:



100	0
100	0

vertical

100	100
0	0

horizontal

Suppose we are given a “mystery” image and want to determine whether it has a vertical or horizontal edge pattern:

80	30
90	20

mystery

We can first calculate the element-by-element *difference* between each known image and our new mystery image:

20	-30
10	-20

vertical - mystery

20	70
-90	-20

horizontal - mystery

We are really only interested in the *amount* of difference between the two patterns, so we can take the absolute value of the differences. ***(What MATLAB function can be used here?)***

20	30
10	20

|vertical – mystery|

20	70
90	20

|horizontal – mystery|

On average, the brightness values in the mystery image differ from the brightness values in the vertical image by only 20, while the brightness values differ from those in the horizontal image by 50 (on average), so we recognize it as a vertical edge. We will refer to the measure of the difference between two images illustrated here as the **mean absolute difference**.

In this part, you will write code to determine which of the six known face images (named sawyer, soong, melroy, bolin, clinton, and albright) ***best matches*** each of the six unknown face images stored in the **newlms** cell array, using the **mean absolute difference** between pairs of images to assess how well they match. We refer to this strategy as ***template matching***, in that the known face images serve as *templates* for the appearance of the known people in our database.

Before moving ahead with the programming, let's take a moment to reflect on how this face recognition strategy differs from the PCA based (Eigenfaces) approach explored previously.

> **Q1.** For the PCA based method, what information is stored for each face image in the database of known faces, to enable recognition? Which method (PCA or template matching) uses a more compact representation of each image? For the PCA method, given a new face image to be recognized, how do we measure the difference between this new face image and each face image stored in the database, in order to determine which stored face is the best match? How does this measure differ from the difference measure that we are using here in the template matching method?

To implement the template matching strategy, you will add code to the **recognize.m** script file, which contains initial comments to help guide you through the steps of the coding process. The following are **some additional tips to help organize your program:**

- Create a vector to store the mean absolute difference between an unknown face image and each of the six known face images - this will help you to determine the best match. (Note that this vector can be reused for each unknown image.)
- Construct a loop to step through each unknown face image in the **newlms** cell array and recognize the face. Use **i** for the control variable in the loop. **Inside the loop:**
    - Store the computed mean absolute difference between the new unknown face image and each known face image (sawyer, soong melroy, bolin, clinton, albright) in the vector that you created. ***Try to construct a single code statement to measure the difference between the unknown face image and a single known face image, and store this difference in the vector.***
    - Print the contents of the vector of differences between the unknown face image and six known face images, to make sure the values look reasonable.
    - Find the best match and print a message to the user with the name of the alumna who is the best match - the **names** cell array will be helpful here.
    - In the comments that appear in the initial recognition.m script file, there are two code statements that print out the correct answer for each unknown image. When you have completed your code for the loop, uncomment these statements (be sure they are also contained within the loop).

> **Q2.** Observe the results of recognizing the unknown full-face images. Who was recognized correctly? **For those who were not recognized correctly,** observe the

vector of measured differences between these unknown face images and the six known faces. How different are the measures of mean absolute difference, between the "correct" face and the one that was chosen as the correct match? Looking at the actual images being compared, why do you think the strategy got the wrong answer in these cases? Do you think it would help to use a "cropped" region of the face for recognition, which does not include the background scene around the head, or the person's clothing?

(b) Recognizing the cropped face images

To analyze the cropped image samples, start by copying-and-pasting the loop code from part (a), and then make following changes: (1) modify the **for** statement to loop through 12 images instead of 6, (2) get the unknown images from the **newCrops** cell array, and (3) compare the unknown images to the cropped images of each known face (sawyerCrop, soongCrop, melroyCrop, bolinCrop, clintonCrop, albrightCrop). Note that you can reuse the vector you created to store the measured differences between images.

Q3. Observe the results of recognizing the 12 cropped images. Who was recognized correctly? The first six cropped images were taken from the full-face images you analyzed in part (a) (see the images shown in Figure 2 above). Were any of the examples that were previously recognized incorrectly **(in Q1)** now recognized correctly when only the face region was used for recognition? **For those who were not recognized correctly,** why do you think the strategy got the wrong answer in each case?

Q4. Given your observations in **Q1** and **Q2** about why the strategy sometimes fails, how might you modify the strategy to perform better? (There's no need to do any coding here to test these ideas.)

Q5. Comparing the template-matching strategy to the PCA based (Eigenfaces) approach, how well do the two methods appear to handle changes in lighting, pose (orientation) of the face, or facial expression?

Q6. How might this template-matching strategy be used in conjunction with a feature based face recognition method, i.e., could you use this method to detect the important parts of a face?

Q7. (Code submission) While you are still working on the code for this lab, use **Fetch** to upload the **faceLab** folder (with your current version of the recognition.m script) to the cs125 folder in your account on the CS file server. (You may want to store this folder in the accounts of both partners, but be careful to coordinate between partners on any future revisions to the code). **When your code is complete, drag the recognize.m Code file into the drop folder of one of the partners, and indicate the location of the final code file below.** Mike and Ellen have access to files contained in your drop folder.

Optional coding challenge: In your two loops, replace the six assignment statements that store the measures of mean absolute difference with an **inner for loop** that implements this process with more compact code.

Performance of Humans vs. Machines at Recognizing Faces Under Changing Illumination

In Lab 3 that explored the PCA based (Eigenfaces) approach to face recognition, it was observed that this method performed poorly with new lighting conditions that were not present in the set of face images used to compute the principal components. One of the student teams raised the question of how the model's performance compares to human performance on the challenging task of recognizing faces under varying illumination conditions. If you're interested in exploring this further, the following two papers address human performance on this task, and comparison of this performance to a number of computer vision models of face recognition.

Braje, W., Kersten, D., Tarr, M., and Troje, N. (1998) Illumination effects in face recognition, *Psychobiology 26(4)*, 371-380.

O'Toole, A., Phillips, P. Jiang, F., Ayyad, J., Penard, N., and Abdi, H. (2007) Face recognition algorithms surpass humans matching faces over changes of illumination, *IEEE Trans. Pattern Analysis and Machine Intelligence 29(9)*, 1642-1646.