CPSC 425: Assignment 2
Name: Terence Chen
Student number: 42602136

Q1)

I unzipped the files as specified in the assignment description

Q2)

Code:

```
#Question2
def MakePyramid(image, minsize):
    # Scale factor of 1 in the first image before any scaling occurs
    scale_factor = 1
    image_list = []
    minSizeImg = min(image.size)
    # Stops the scaling once the scaled image will have a dimension less than minsize
    while(minSizeImg * scale_factor  > minsize):
        # Scales the image according to the sacling factor of 0.75 appends the image
        image_list.append(image.resize((int(image.size[0] * scale_factor),int(image.size[1] * scale_factor)), Image.BICUBIC))
        # Scale factor of 0.75 that is multiplied each subsequent loop
        scale_factor = scale_factor * 0.75
    return image_list
```

Q3)

Code:

```
#Question 3
def ShowPyramid(pyramid):
    # Note: Rather than a horizontal stack we were told to construct the image shown as the zebra image in class. Also specified by the prof on piazza post @96
    # Initializes height and width as 0
    height = width = 0
    # Set the offsets as 1
    offset_x = offset_y = 1
    for img in pyramid:
        # Width will be the sum of the largest and second largest image
        width = pyramid[0].size[0] + pyramid[1].size[0]
        # Compute the sum of all the heights of the images
        height += img.size[1]

    # Height will be the sum of all height minus the largest image
    height -= pyramid[0].size[1]

    # creates the background image with a white color
    blankImage = Image.new("L", (width, height), "white")
    for img in pyramid:
        # If this is the first image then we simply paste the image into the canvas and saves offset value
        if (img == pyramid[0]):
            blankImage.paste(img, (offset_x, offset_y))
            offset_x = img.size[0]
        else:
            # Otherwise for each subsequent image we apply a y axis offset that is equivalent to the height of each level of image.
            blankImage.paste(img, (offset_x, offset_y))
            offset_y += img.size[1]

    blankImage.show()
```

Using a min size of 50. ShowPyramid(MakePyramid(sports, 50)) I get the image:

This stacking layout was specified by the prof in class and discussed in piazza, on post 96.

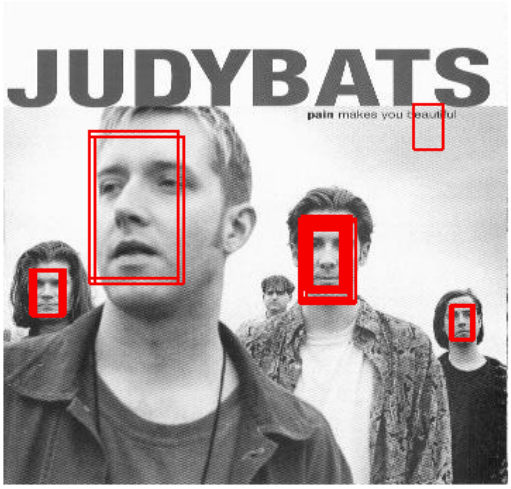
Q4)
Code:

```
# Question 4
def FindTemplate(pyramid, template, threshold):
        #desired template width
        tempWidth = 15
        #empty list to store matches above threshold
        detectedMatches = []
        #x and y size for each template
        x = template.size[0]
        y = template.size[1]
        #resizing template
        newTemp = template.resize((int(tempWidth), int(y/(x/tempWidth))), Image.BICUBIC)
        # Allow the first picture (only need to show matches for first image) to display colored box
        coloredImg = pyramid[0].convert('RGB')

        for image in pyramid:
                # Calculate the NCC and append the components to detectedMatches
                matchedTemp = np.where(ncc.normxcorr2D(image,newTemp) > threshold)
                detectedMatches.append(zip(matchedTemp[1],matchedTemp[0]))

        for imgLvl in range(len(detectedMatches)):
            for coord in detectedMatches[imgLvl]:
                #drawing out the boxes
                draw = ImageDraw.Draw(coloredImg)
                # x and y boundary coordinates of matching template, increases height width and height depending on t
                x1 = int(coord[0]/0.75 ** imgLvl) - int(newTemp.size[0]/(2 * 0.75 ** imgLvl))
                x2 = int(coord[0]/0.75 ** imgLvl) + int(newTemp.size[0]/(2 * 0.75 ** imgLvl))
                y1 = int(coord[1]/0.75 ** imgLvl) - int(newTemp.size[1]/(2 * 0.75 ** imgLvl))
                y2 = int(coord[1]/0.75 ** imgLvl) + int(newTemp.size[1]/(2 * 0.75 ** imgLvl))

                # draw out 4 lines to form a rectangle using coordinates
                draw.line([(x1,y2),(x1,y1)], fill = "red", width = 2)
                draw.line([(x1,y2),(x2,y2)], fill = "red", width = 2)
                draw.line([(x1,y1),(x2,y1)], fill = "red", width = 2)
                draw.line([(x2,y1),(x2,y2)], fill = "red", width = 2)

                del draw
        return coloredImg
```

Q5) So for question 5 I simple tried out different threshold values then calculated the error rate by counting the false negatives and false positives by manually checking each image. I notice that as we get to the range of 0.55 – 0.65 we get pretty good results. A relatively good threshold value is 0.61. I will show the images below using 0.61 as the threshold value.

judybats:

```
In [61]: FindTemplate(MakePyramid(jb, 50), temp, 0.61)
Out[61]:
```



students:

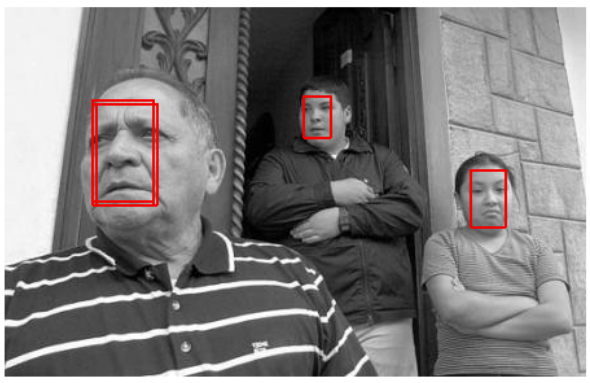
```
In [64]: FindTemplate(MakePyramid(stu, 50), temp, 0.61)
Out[64]:
```

tree:

```
In [65]: FindTemplate(MakePyramid(tree, 50), temp, 0.61)
Out[65]:
```

family:

```
In [62]: FindTemplate(MakePyramid(family, 50), temp, 0.61)
Out[62]:
```

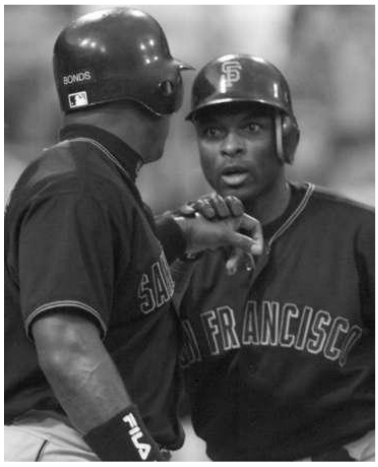


fans:

```
In [68]: FindTemplate(MakePyramid(fans, 50), temp, 0.61)
Out[68]:
```

sports:

```
In [67]: FindTemplate(MakePyramid(sports, 50), temp, 0.61)
Out[67]:
```



Q6) A recall rate is defined as the set of retrieved and relevant documents over the set of all relevant documents. I am going to assume that every face on the image is a relevant document. Using the above images I get the following recall rates.
For “judybats” we have a total of 4 faces and 3 retrieved thus we get a recall rate of 3/4.
For “students” we have a total of 27 faces and 22 retrieved thus we get a recall rate of 22/27.
For “tree” we have a total of 0 faces and 0 retrieved thus we get a recall rate of 0/0 (undefined).
For “family” we have a total of 3 faces and 3 retrieved thus we get a recall rate of 1.
For “fans” we have a total of 3 faces and 0 retrieved thus we get a recall rate of 0/3.
For “sports” we have a total of 1 faces and 0 retrieved thus we get a recall rate of 0/1.
We see that we get very low recall rates for certain images, I think this is because of how we defined what belongs into the relevant document set. The NCC finds matching based on the similarity between our template image and the corresponding images. Since I defined that all faces belong in the relevant documents, for images like the sports.jpg, tree.jpg and fans.jpg there are not a lot of similarities between them and the template image we used. While for images like judybats.jpg, students.jpg and family.jpg the faces in these images are a lot more similar to our template image thus we get more correct matchings which increases our recall rate.