Laboratory deliverable

Bertiche, Hugo
Serra, Xavier
Master in Artificial Intelligence - Computer Vision

November 19, 2015

1 Image smoothing and simple geometric operations in Matlab

1.1 Creating images of 3 channels (color images)

In this exercise all the necessary information is contained in the provided script *exercise1_1.m*.

1.2 Displaying color images

The code necessary to test this exercise is provided in the script *exercise1_2.m*.

Questions:

- When displayed separately, each color channel behaves as a greyscale image. Therefore, each pixel will be lighter if that color is very present in the respective pixel in the RGB image. In our image, as the white color is a combination of the three different channels, most of the image remains white. In relation to the chairs, as they have a red tonality in the original image, they are white in the red channel, and dark in the remaining two.

- If we interchange the channels, the chairs turn the color of the new position of the red channel. This is so because they had a high red channel, and when moved to another position, it makes that color very strong.

- If we replace a whole channel by 0, we will end up with totally different colors, as the original white will lose one of its components and become a different one. For example, if we delete the green color, the white turns into purple, the fusion of red and blue channels.

1.3 Managing different sizes and filters

The code necessary to test this exercise is provided in the script *exercise1_3.m*.

Questions:

b The histogram does change: the vertical axis range decreases (obviously, as the number of pixels decreases) and it becomes crispier, with a less smooth contour, than in the original image. When returning the reduced image to its original size, it recovers the initial vertical range, but the crispy contour is maintained.

c The bigger the mask, the blurrier the image. If the mask has a vertical form, the resulting image becomes blurry with a vertical deformation (all forms in the image look like they have been vertically stretched), and the same effect appears if the mask has a horizontal form, but this time, the deformation appears, logically, in the horizontal axis. In order to be convoluted, an image must be in a 2D form, each cell, or pixel, containing a single value (like in the grayscale). Therefore, a RGB image cannot, technically speaking, receive a filter, as it is a concatenation of 3 matrices. However, Matlab simulates the effect by applying the filter to each layer, or matrix, individually. The final effect is the same that appears in a gray filtered image. If we do not normalize the mask, the values of the pixels in the filtered image are the result of the direct sum of other pixels, which may result in many (even all) values being over 255 and, therefore, burning the image. Thats why it is vital to normalize the mask, in order to keep the resulting values in the [0...255] range.

d The **median filter** is a non-linear filter, and consists in replacing every pixel by the median of its neighbors. It is very well considered because of the fact that it can preserve the edges. Considering the results obtained with sides of 3, 6 and 9, we conclude that a 3 by 3 filter works reasonably high, with a 6 by 6 it may be useful in some cases, but more than 6 it takes away too much of the detail.

1.4 Simple geometric operations

In this exercise all the necessary information is contained in the provided script *exercise1_4.m*.

1.5 Image binarization

The code necessary to test this exercise is provided in the script *exercise1_5.m*.

Questions:

a The higher the threshold, the more pixels will become a 0, as their gray value will be below the given threshold. Therefore, the image will be generally darker and it will gradually lose detail.

c Multiplying an image by its binarized one will paint black all pixels being black in the binarized image (as they will be multiplied by 0), and will keep the rest of the pixels as they are in the original (as they will be multiplied by 1).

d Multiplying an image by it inverted binarized will paint black all pixels being white in the simply binarized image (as they will be multiplied by 0), and will keep the rest of the pixels as they are in the original (as they will be multiplied by 1).

1.6 Treating color images

In this exercise all the necessary information is contained in the provided script *exercise1_6.m*.

2 Edges and contours

2.1 Hybrid images construction

In order to make an hybrid image we have to add the low frequencies of an image to the high frequencies of another one. To get the low frequencies of an image, we must apply a "low-pass filter" on the image. This filter has a parameter σ which indicates the variance in the used kernel. To get the high frequencies of an image, we must subtract the low frequencies to the image itself.

The resulting images are shown in the appendix. The best σ values depend on many properties of the image: frequencies, brightness, contrast, etc. In figure the 1 we can see the result with $\sigma = 3$ in both images. As we can see, these values for σ are not good. In the figure 2 we have both images with $\sigma = 7$. Here, the result is good, since we can see Einstein in the biggest image and Marylin Monroe in the smallest one. In the figures 3 and 4 we have used different sigmas (3 and 7). We can appreciate that the results are not good.

In the figures 5 and 6 we interchange the images. In this case, the best result is given using $\sigma = 3$ and $\sigma = 7$.

2.2 Determine the optimal edges

We have used 6 different edge detectors: Canny, log, Prewitt, Roberts, Sobel and zerocross. Canny and log use 2 paremeters: σ and threshold. For these two detectors, we have used σ from 1 to 3 and $threshold = 0.3$, $threshold = 0.6$ and not using it at all.

These detectors have been applied on three images, shown in the figures 7, 8, 9 and 10. The best results are obtained from the Canny detector, which is the state of the art, with $\sigma = 1$.

In the starbucks image, the results obtained with Canny detector using $sigma = 1$ are the same for each value of threshold (without using it, 0.3 and 0.6). But in the second image, the rooms with chairs and the dolphin, there are clear differences: when we use a threshold, too much edges are skipped.

In images like the one in figure 10, we have too much high frequencies in the whole images. We can see that the result is not enough good because of the see. We can correct it by using higher values in the threshold parameter. In contrast, images like the starbucks logo get good results because the edges are few and very clear.

3 Enhancing images with edges

The best way to obtain the edges is using the Canny detector, which is the state of the art. But the best parameters depend on the frame: in frames with high frequencies, like the ones of the sea, it is better to set an high threshold. In frames with lower frequencies a low threshold is better. We can see an example of sea frame without using threshold in figure 12 and using $threshold = 0.3$ in figure 11.

This effect has the advantage that, in some images, the objects shown are highlighted, giving a beautiful effect. But in other images, specially the ones with a lot of edges in the background, this highlighting is not desired at all.

3.1 Appendix

3.1.1 Hybrid images

Figure 1: Marilyn + Einstein using $\sigma = 3$ in both cases.

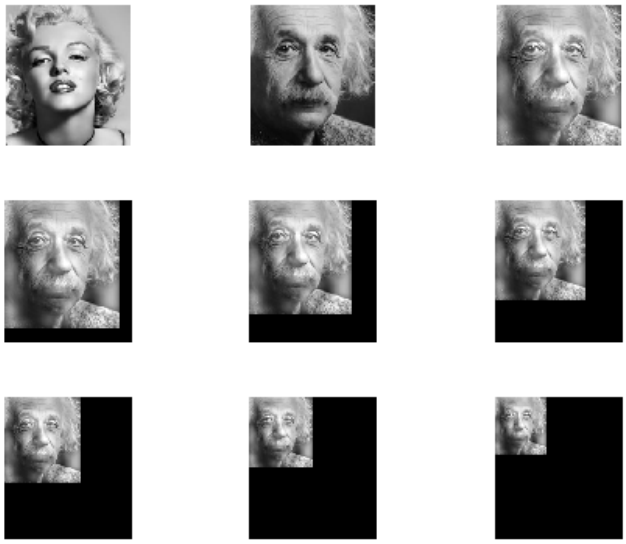

Figure 2: Marilyn + Einstein using $\sigma = 7$ in both cases.

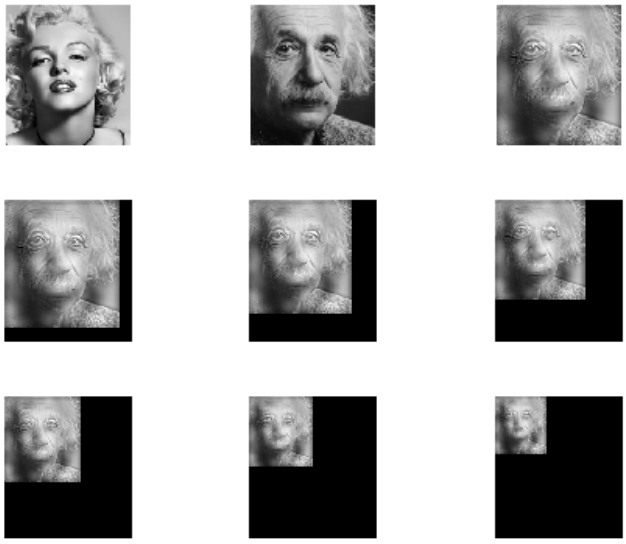

Figure 3: Marilyn with $\sigma = 7$ + Einstein with $\sigma = 3$.

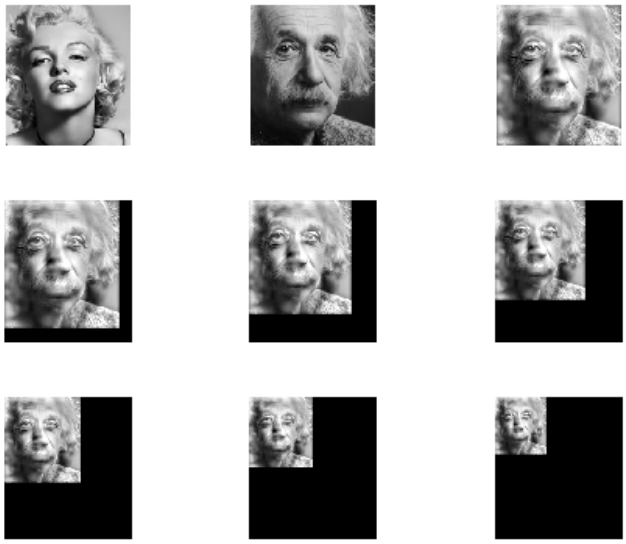

Figure 4: Marilyn with $\sigma = 3$ + Einstein with $\sigma = 7$.

Figure 5: Einstein + Marilyn using $\sigma = 3$ in both cases.

Figure 6: Einstein with $\sigma = 3$ + Marilyn with $\sigma = 7$.

3.1.2 Edges

Figure 7: Canny detector with $\sigma = 1$.

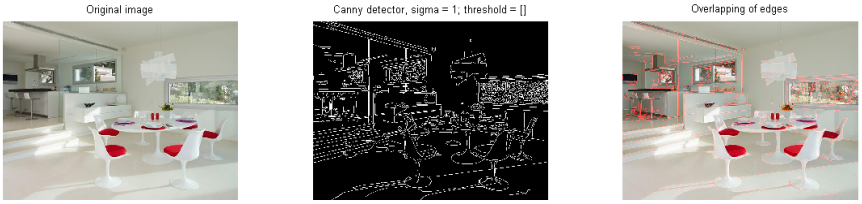

Figure 8: Canny detector with $\sigma = 1$.

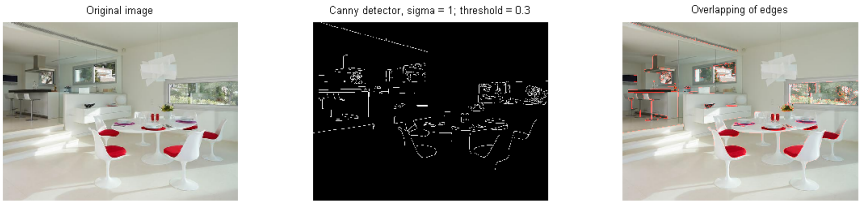

Figure 9: Canny detector with $\sigma = 1$ and $threshold = 0.3$.

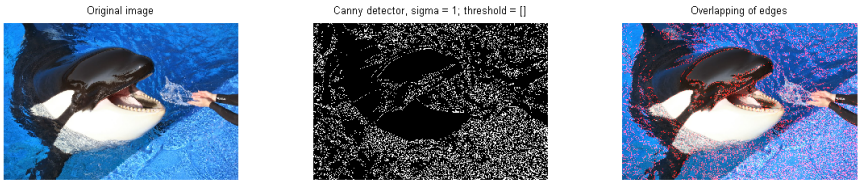

Figure 10: Canny detector with $\sigma = 1$.

3.1.3 Enhancing images with edges

Figure 11: Canny detector with $\sigma = 1$.

Figure 12: Canny detector with $\sigma = 1$ and threshold=0.3.

4 Retrieval of images based on texture

In this exercise, we'll use textures in order to, given an image or scene, retrieve similar images of the same kind. Three different classes shall be used: buildings, forests and sunsets.

4.1 Function *getFeatures*

First of all, we need to be able to extract texture features from our images. To do so, we use Leung-Malik filters. These filters are matrix capable of identifying different patterns when we perform a convolution of the image with them. In the Figure 13 visual representation of these filters appear. This has been made with MatLab command *imagesc*(), which scales a matrix and shows it as a colormap, which associates each color to a different value, in order to know this color scale we use MatLab command *colorbar*.

Then, for a given image, as stated before, we perform a convolution with every filter. This shall give a set of 48 filtered images, as shown on Figure 15, finally, we average every matrix in order to obtain an array of 48 real numbers, which are the texture descriptors.

Depending on the image, some filters shall produce a better response than others, which shall result in higher values for its corresponding descriptors. Figure 14 shows an example image upon which we had applied all these filters, resulting on Figure 15.

In this case, those filters obtaining better response with the image are those regarding "dots" and vertical lines, such as those on the last line, or the forth one. This is a logical result, taking into account the fact that in the image the most frequent shapes are small dots and vertical lines.

Also, the sky is also strongly recognized, which may introduce some distortion in the recognition of future images.

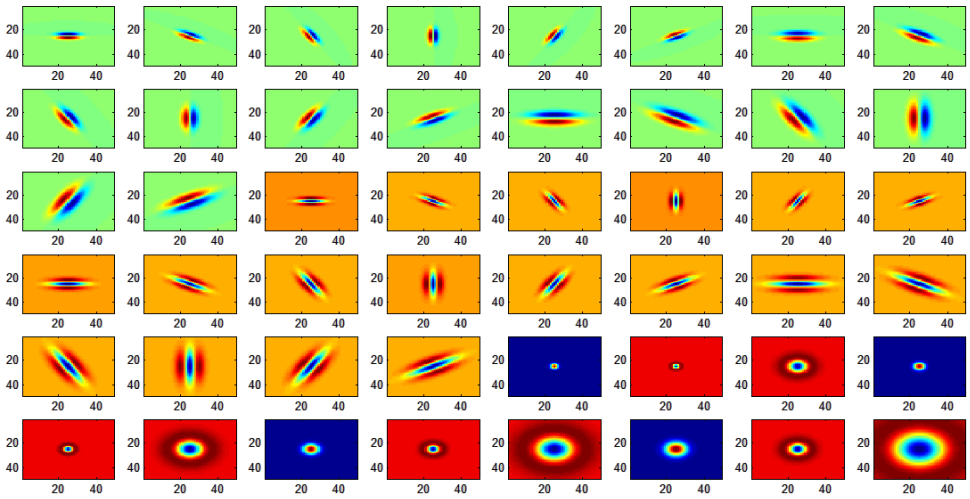

Figure 13: Different filters

Figure 14: Image to filter

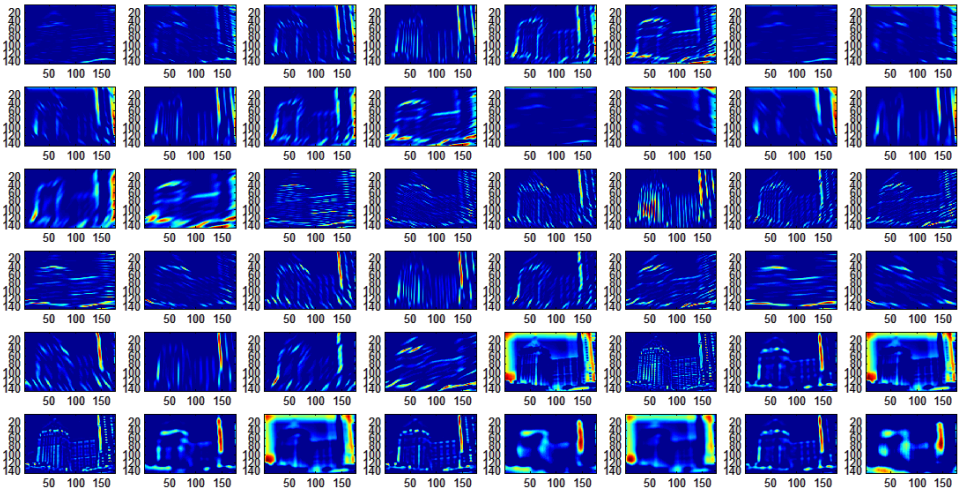

Figure 15: Different filters applied to an image

4.2 Function *getClassFeatures*

The next thing to do is to apply this feature extractor to our bank of images, which includes three classes: *buildings*, *forest* and *sunset*.

This process will output a database of descriptors which shall be used later in order to efficiently retrieve images of the corresponding class.

4.3 Function *retrieveKImages*

Finally, this last function will take an image as an input and also a natural number which indicates how many of the closest images from the database shall be retrieved. To perform this, the difference between the input image descriptors and the descriptors of each database image is computed and the closest neighbors are returned and displayed.

To test this, we've chosen three random images (each one of a different class). This is the result:

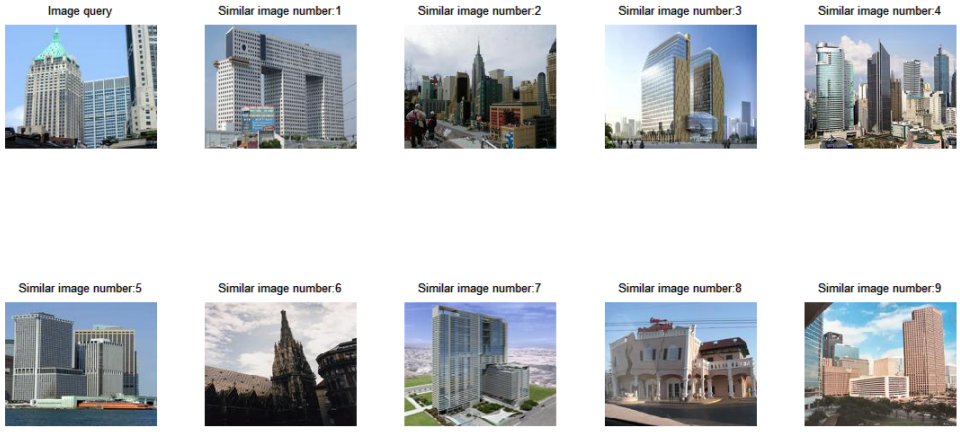

Figure 16: Output of *retrieveKImages* in a building image

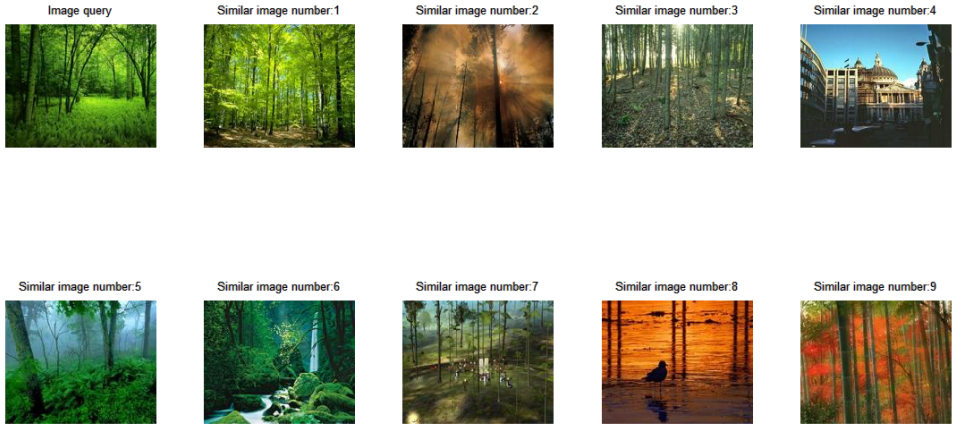

Figure 17: Output of *retrieveKImages* in a forest image

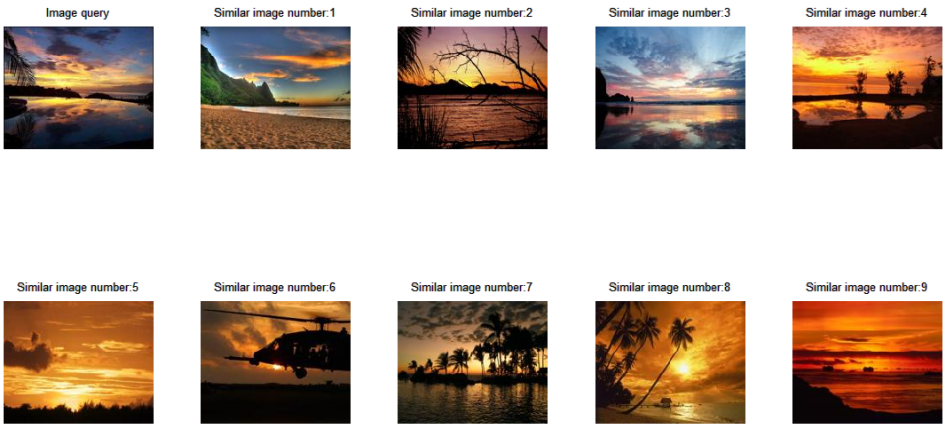

Figure 18: Output of *retrieveKImages* in a sunset image

Analyzing these figures, we might conclude that the retrieval is generally good, but it is not possible to ignore the amount of wrong images. This might be due to some common element in some images such as the sky, or vegetation, and also because it is possible to find similar textures in different scenarios. For example, straight line both in buildings and in forests. In order to improve this, we might consider using another additional criteria besides texture.

4.4 Implementing color descriptors

As the result of the previous section was not good enough for us, we decided to keep ongoing and implement an additional criteria for retrieval, which is color.

There are many different approaches for this task, such as the average value of each color channel, or its standard deviation. Nevertheless, as this would output very few descriptors, it wouldn't have much influence upon texture descriptors, which is 48-dimensional. In order to give a greater influence to color descriptors, we chose to use histograms for each color channel, this methodology produces a 30-dimensional color descriptor, which is comparable to the amount of texture features. Therefore, we end up with a total amount of 78 dimensions for the features space.

After applying the same procedure as before, but including color features, this is the result obtained for the same images analyzed before:

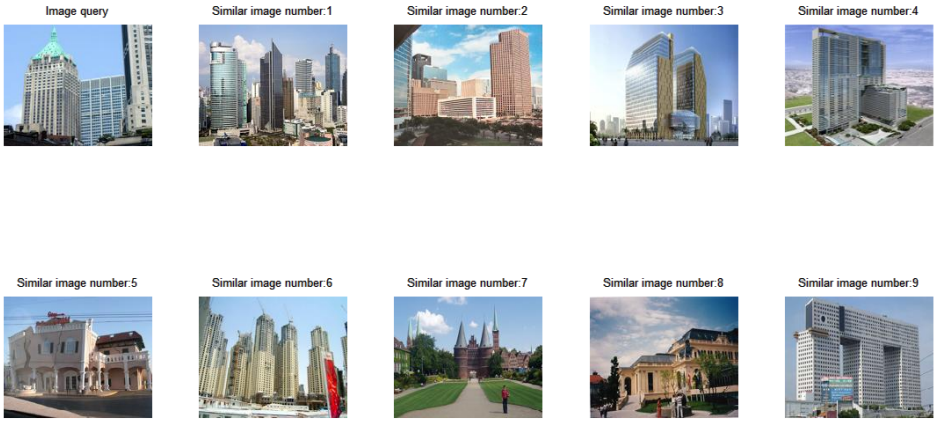

Figure 19: Output of *retrieveKImages* with color features in previous building image

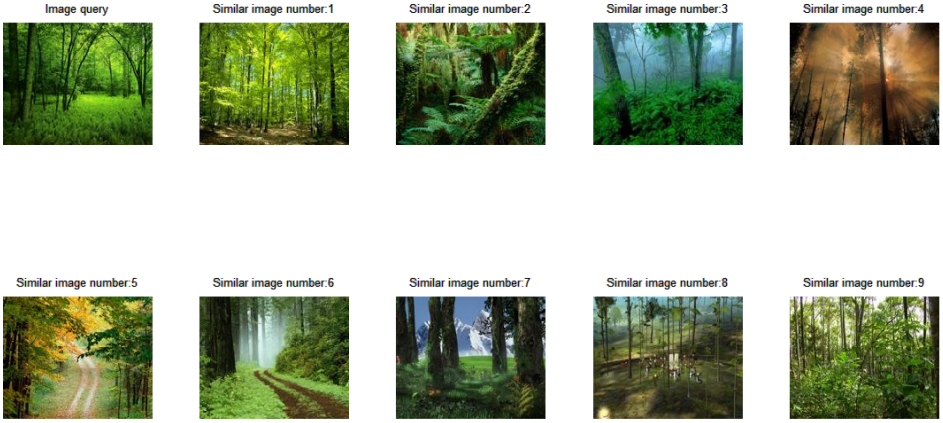

Figure 20: Output of *retrieveKImages* with color features in previous forest image

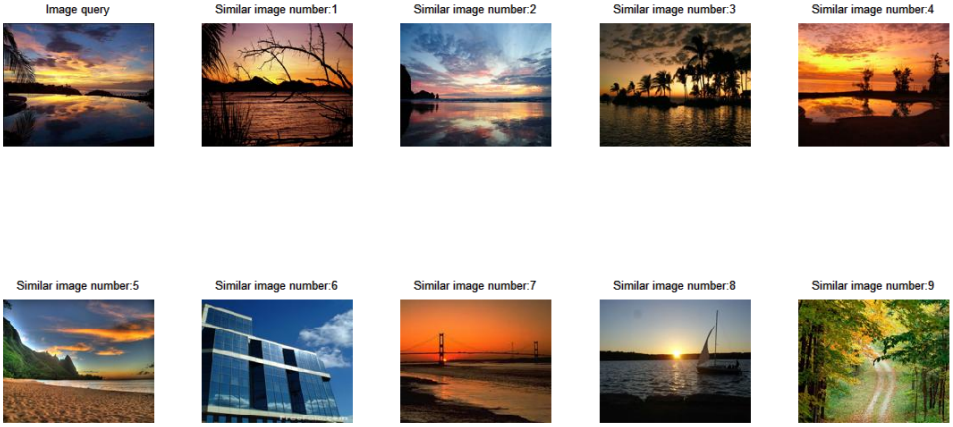

Figure 21: Output of *retrieveKImages* with color features in previous sunset image

We can see, comparing with previous results, that performance is better in some cases when taking color descriptors into account. It might be possible to improve the system by defining new color descriptors, decreasing the bin size of the histograms, working with HSV instead RGB or maybe working with color gradient instead of color itself. It could be also an improvement to take only into account descriptor above a certain threshold or giving different weights to color or texture features. There are a lot of strategies and approaches that might result on better performance. Nevertheless, in order to ensure this further experimentation should be carried out.