









































Step 1: Color Distance

Design Choices

I originally began to test my code with the bin value set to 2 for each channel. Below, you will find a screenshot of select output. Because the output format is quite lengthy, I've decided that selecting certain rows to show you here in the write-up would be the most efficient. My full output will be attached at the end of this document.





































 i20.jpg	 i05.jpg 6	 i23.jpg 142	 i07.jpg 1	149
 i21.jpg	 i25.jpg 0	 i03.jpg 0	 i24.jpg 11	11
 i22.jpg	 i29.jpg 113	 i30.jpg 21	 i13.jpg 0	134
 i23.jpg	 i05.jpg 5	 i06.jpg 9	 i20.jpg 76	90
 i24.jpg	 i21.jpg 6	 i03.jpg 3	 i36.jpg 31	40
 i25.jpg	 i21.jpg 0	 i19.jpg 1	 i17.jpg 16	17
 i26.jpg	 i18.jpg 19	 i02.jpg 14	 i17.jpg 15	48
 i27.jpg	 i32.jpg 207	 i24.jpg 32	 i28.jpg 30	269
 i28.jpg	 i11.jpg 1	 i33.jpg 149	 i24.jpg 2	152
 i29.jpg	 i31.jpg 1	 i34.jpg 1	 i35.jpg 1	

The final bin arrangement I settled on was 2, 6, and 8 for red, green, and blue respectively. This combination of bin sizes was picked as final because all the photos with a strong red presence typically only possess red, thus not warranting a high resolution of bins. The same goes for green and photos being predominantly green. The reason why the blue bin value was bumped up so high was because the photos with the most blue present had speckles of green or red in them, too. A higher bin value allows for the data to be read with a higher sensitivity to subtle patterns in the data, and blue would only show up in the capacity we see below:



These are the only photos with significant blue presence.

Thus warranting the highest bin count of the three channels. I decided to keep red low to avoid overfitting the data and skewing already reasonable results towards inaccuracy. Below are the results from using these bin values:

 i20.jpg	 i35.jpg 108	 i23.jpg 142	 i18.jpg 10	260
 i21.jpg	 i34.jpg 3	 i28.jpg 3	 i17.jpg 99	105
 i22.jpg	 i34.jpg 20	 i38.jpg 4	 i36.jpg 84	108
 i23.jpg	 i35.jpg 53	 i40.jpg 16	 i28.jpg 8	77
 i24.jpg	 i36.jpg 31	 i38.jpg 46	 i19.jpg 222	299
 i25.jpg	 i17.jpg 16	 i18.jpg 7	 i28.jpg 178	201
 i26.jpg	 i07.jpg 0	 i11.jpg 0	 i06.jpg 0	0
 i27.jpg	 i32.jpg 207	 i33.jpg 12	 i30.jpg 28	247
 i28.jpg	 i33.jpg 149	 i21.jpg 3	 i25.jpg 228	380

As you can see, their row score (the column on the far right that holds the sum of that row's photos' crowd scores) is far higher than it was when I was using 2 bins per channel.

Algorithm

The code calculates the color histograms for each image using the `cv2.calcHist()` function. The histograms are calculated separately for each channel (red, green, and blue) of the image, and the resulting histograms are flattened into 1D arrays. The code then passes this data into the `compute_color_difference()` function, which takes two histograms as input and calculates the absolute difference between corresponding histogram bins. The sum of these differences is divided by the total number of bins in the histograms to obtain a single value describing the color distance between two images (the I-value).









































Output Observations

The photos that had the most accurate rankings were photos with a high texture that had very little blackspace behind them. For example:



i06 as "Q" is on the far left, with t1-t3 listed to its right.

Below is a snippet of the output chart to view more examples of this algorithm's output for low-blackspace images. The complete chart can be found at the end of this document.

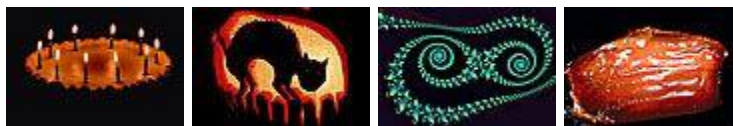
 i03.jpg	 i04.jpg 272	 i36.jpg 1	 i01.jpg 115	388
 i04.jpg	 i03.jpg 295	 i36.jpg 9	 i01.jpg 88	392
 i05.jpg	 i06.jpg 309	 i23.jpg 0	 i07.jpg 136	445
 i06.jpg	 i11.jpg 72	 i07.jpg 142	 i05.jpg 259	473
 i07.jpg	 i09.jpg 178	 i40.jpg 48	 i11.jpg 58	284
 i08.jpg	 i03.jpg 233	 i01.jpg 260	 i19.jpg 0	493
 i09.jpg	 i07.jpg 239	 i11.jpg 61	 i02.jpg 0	300
 i10.jpg	 i01.jpg 136	 i03.jpg 33	 i16.jpg 272	441
 i11.jpg	 i09.jpg 98	 i07.jpg 122	 i06.jpg 231	451
 i12.jpg	 i09.jpg 98	 i07.jpg 122	 i06.jpg 231	

Photos with more blackspace sometimes had a more difficult time with this algorithm. The most accurate ranking concerning photos like this can be seen below.



i35 as "Q" is on the far left, with t1-t3 listed to its right

Other rows were not as fortunate:



i33 as "Q" with its three winners.

How Well I Believe My Design Worked

I think that this design worked worse than my shape algorithm, but better than my texture algorithm. Regarding images with low blackspace, I think it worked incredibly well. Regarding photos with blue presence, I don't think it did that great of a job despite the value I set for blue bins. If I were to do this project again, I would try to create a more robust system that would be able to perceive the presentations of blue that this photoset presented.

Step 2: Texture Distance

Design Choices

I originally used the following parameters for the Laplacian function. Take a look at this specific row, given those parameters.

```
laplacian = cv2.Laplacian(gray, cv2.CV_64F)
```



i12 as “Q” on the left, with t1-t3 listed to the right.

I decided to increase the strength of the Laplacian operator by upping the default kernel value from 1 to 3. These parameters ended up working better for more textural images, such as i12.

```
laplacian = cv2.Laplacian(gray, cv2.CV_64F, ksize=3)
```



i12 as “Q” on the left, with t1-t3 listed to the right.

This adjustment also allowed for greater accuracy regarding images with a higher amount of blackspace. On the top row is before the kernel adjustment, and the bottom is after, both regarding i38 as Q:







































TOP ROW: kernel = 1. BOTTOM ROW: kernel = 3.

Algorithm

The code converts the image to grayscale by averaging the intensity values across all color channels. It then applies openCV’s Laplacian operator to the grayscale image using the `cv2.Laplacian()` function, where the edges in the original image are emphasized and all other details are dulled out. From this Laplacian image, another histogram is produced, but this one only has one channel opposed to three that we saw in the previous step. The same bin comparison computation is used, however (takes two histograms as input and calculates the absolute difference between the corresponding histogram bins), to get a single value to describe the distance between these two mono channel histograms.

Output Observations

Below is a sample of the output. In the sample, you can see that this algorithm does a decent job at picking up on the textural details of objects that aren’t a solid texture, like in lines i12 and i23 where a solid shape has some detail within its borders. It matches it to other images that have some variation within a distinct shape. Below, I go into further detail on how it handles images of solid textures.

 i18.jpg	 i20.jpg 7.0	 i36.jpg 0.0	 i17.jpg 236.0	243.0
 i19.jpg	 i40.jpg 0.0	 i24.jpg 235.0	 i38.jpg 76.0	311.0
 i20.jpg	 i18.jpg 10.0	 i36.jpg 8.0	 i32.jpg 0.0	18.0
 i21.jpg	 i34.jpg 3.0	 i22.jpg 157.0	 i29.jpg 27.0	187.0
 i22.jpg	 i34.jpg 20.0	 i21.jpg 203.0	 i29.jpg 113.0	336.0
 i23.jpg	 i38.jpg 0.0	 i18.jpg 150.0	 i17.jpg 106.0	256.0
 i24.jpg	 i38.jpg 46.0	 i19.jpg 222.0	 i28.jpg 4.0	272.0
 i25.jpg	 i28.jpg 178.0	 i17.jpg 16.0	 i38.jpg 0.0	194.0
 i26.jpg	 i35.jpg 32.0	 i30.jpg 0.0	 i22.jpg 0.0	32.0

How Well I Believe My Design Worked

This algorithm works strikingly well, compared to my other metrics. I even managed to tune it to the granularity of the objects in the image, to my surprise, on images that present a solid texture pattern with minimal blackspace:



i09 as “Q.” These photos have a fine texture because they’re capturing small objects from a distance.



i12 as “Q,” These photos are composed of larger objects, both in reality and in their representation in the image.

This is the reason why I experimented with giving texture a higher influence on the Gestalt vector, but as I will discuss later, that experimentation did not lead to the greatest grand total score for the output table.























Step 3: Shape Distance (Background vs. Foreground)

Design Choices

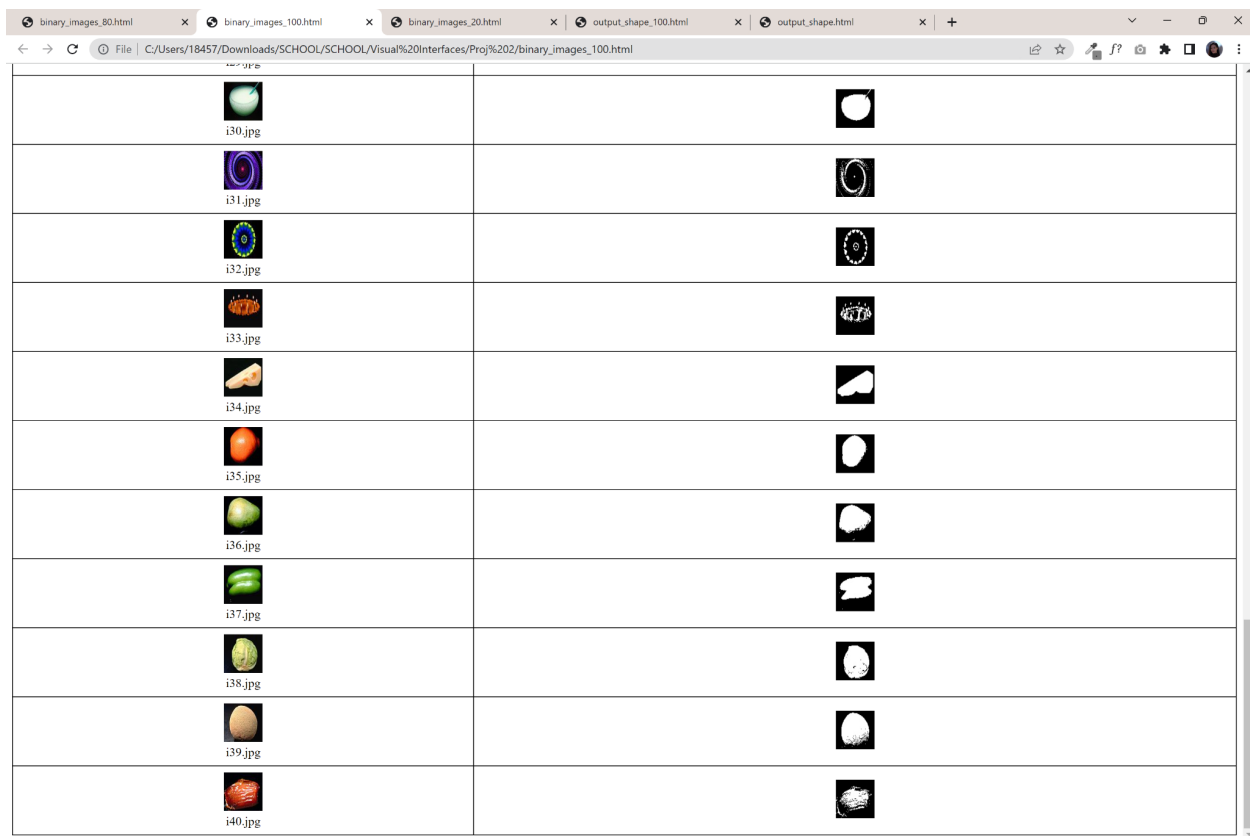
I mostly toyed with the threshold used to binarize the images. I experimented with the values 20, 80, and 100.

I started with 80 as `threshold` in the below function, which got me the output you see directly below it. Again, I've selected a sample for ease of viewing. You can view the full binarization test output using the final threshold value at the end of this document.

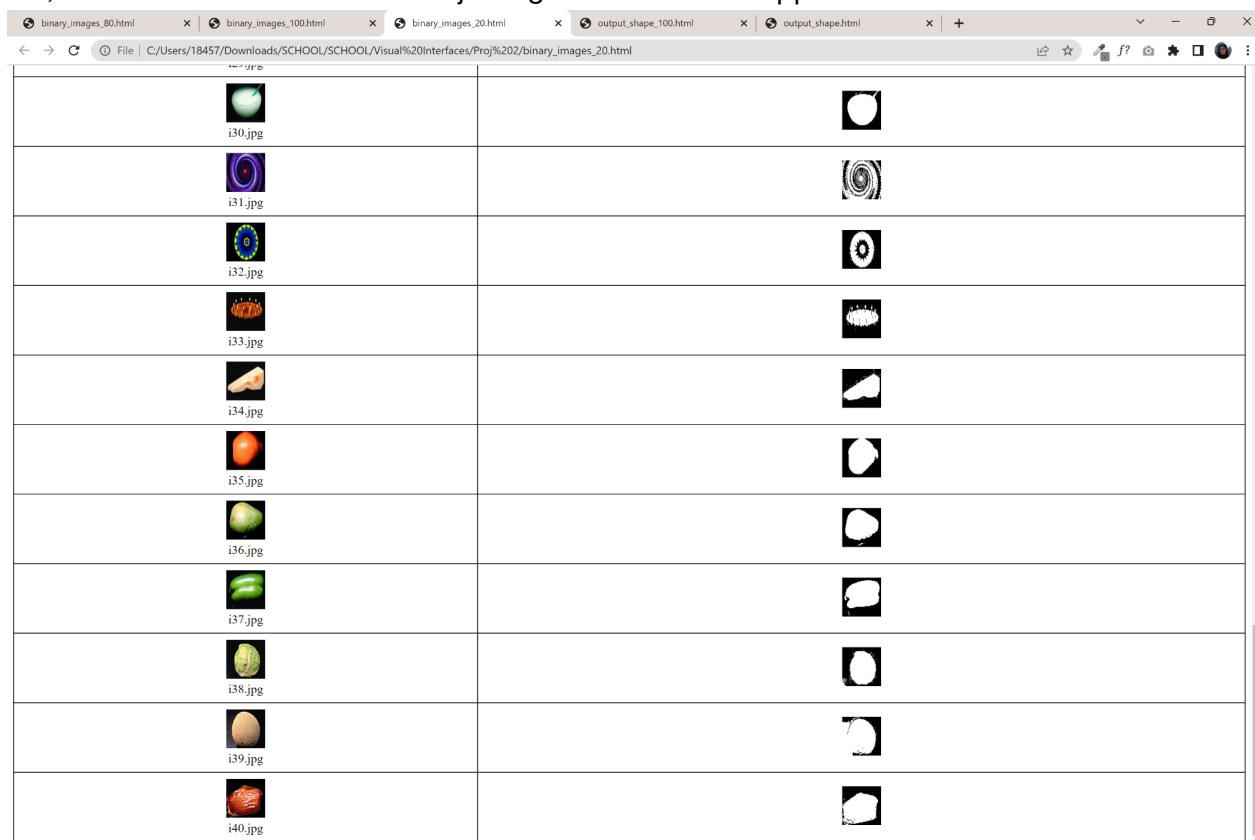
```
_, bin_img = cv2.threshold(gray, threshold, 255, cv2.THRESH_BINARY)
```

Focus on the last three images. There's some patchiness around the bottom corners of the photo that extend past the shape. Below is my trial with 100 as threshold, thinking that this value would define black as a value that would encapsulate more of those shadows.



I was correct, but I tried 20 as well to test adjusting the value in the opposite direction:







































100 as the threshold value *did* show a clearer shape for the images, specifically the last three, and had a higher grand total score than the outputs using 20 and 80, so I decided to proceed with a threshold value of 100. With this threshold value did I find a good definition of “black” to work with.

Algorithm







































First, I use the `binarize_image()` function, which converts a color image to grayscale using the `cv2.cvtColor()` function and applies a threshold of 20 to create a binary image using the `cv2.threshold()` function. The `shape_difference()` function then takes in two binary images, `image1` and `image2`, and calculates the difference between them by counting the number of pixels that “disagree” between the two images. This is done by first creating a Boolean array that indicates where the two images differ (`image1 != image2`), and then summing up the number of True values in the array using `np.sum()`. The resulting count of differing pixels is divided by the total number of pixels in the images ($2 * 89 * 60$) to obtain a single value, *S*, to describe their shape distance.

Output Observations

Below is a sample of some of the best lines of output from this algorithm. As you can see, it handles distinct shapes quite well, like with rows `i17-i20`.

i14.jpg  i15.jpg	i12.jpg  i12.jpg 7	i13.jpg  i13.jpg 1	i16.jpg  i26.jpg 0	8
i16.jpg  i16.jpg	i15.jpg  i15.jpg 129	i12.jpg  i12.jpg 0	i13.jpg  i13.jpg 4	133
i17.jpg  i17.jpg	i23.jpg  i23.jpg 70	i18.jpg  i18.jpg 268	i38.jpg  i38.jpg 3	341
i18.jpg  i18.jpg	i23.jpg  i23.jpg 85	i17.jpg  i17.jpg 236	i22.jpg  i22.jpg 1	322
i19.jpg  i19.jpg	i23.jpg  i23.jpg 24	i18.jpg  i18.jpg 65	i38.jpg  i38.jpg 76	165
i20.jpg  i20.jpg	i18.jpg  i18.jpg 10	i38.jpg  i38.jpg 0	i23.jpg  i23.jpg 142	152
i21.jpg  i21.jpg	i17.jpg  i17.jpg 99	i38.jpg  i38.jpg 1	i40.jpg  i40.jpg 0	100
i22.jpg  i22.jpg	i30.jpg  i30.jpg 21	i17.jpg  i17.jpg 9	i18.jpg  i18.jpg 18	48
i23.jpg  i23.jpg	i18.jpg  i18.jpg 150	i17.jpg  i17.jpg 106	i19.jpg  i19.jpg 27	283

It performs decently for solid textured images, as oftentimes there will be a blatantly round object with strong blackspace behind it as a contender for similarity to a pile of string beans, as seen in line `i04`.

 i03.jpg	 i39.jpg 0	 i18.jpg 0	 i04.jpg 272	
 i04.jpg	 i39.jpg 0	 i26.jpg 0	 i40.jpg 0	0
 i05.jpg	 i12.jpg 2	 i15.jpg 72	 i26.jpg 0	74
 i06.jpg	 i26.jpg 0	 i12.jpg 1	 i15.jpg 94	95
 i07.jpg	 i15.jpg 46	 i12.jpg 1	 i14.jpg 6	53
 i08.jpg	 i01.jpg 260	 i14.jpg 0	 i39.jpg 0	260
 i09.jpg	 i15.jpg 49	 i12.jpg 18	 i13.jpg 5	72
 i10.jpg	 i15.jpg 5	 i12.jpg 8	 i13.jpg 20	33
 i11.jpg	 i15.jpg 36	 i12.jpg 0	 i13.jpg 0	36
 i12.jpg	 i15.jpg 12	 i13.jpg 175	 i26.jpg 0	187

How Well I Believe My Design Worked

I think I could have improved this function slightly more. It has the lowest of all the grand total scores, but second to texture by only about 100 or so.

Step 4: Shape Distance (Symmetry)

Unfortunately I was not able to complete this step.

Step 5:

Design Choices

This is how I distributed my vectors. I chose these distributions so each vector would have a descending impact on the output. Note the two arrangements I experimented with and their corresponding grand total scores. I chose the arrangement on the left as final because its score was higher.

```
#gestalt = grand_total_score = 11852
i_weight = 0.33      #color
s_weight = 0.22      #shape
```

```
t_weight = 0.11      #texture
#gestalt1 = grand_total_score = 11205
i_weight = 0.33      #color
```

t_weight = 0.22 #texture

s_weight = 0.11 #shape

Algorithm

What I did was for every image, I created a “profile” which set up its color histogram, single-channel histogram, and created its binarized image. For every other photo for it to be paired with, I created *that photo’s* profile, then compared all components of the two using the algorithms I had developed to derive a color distance value (I), texture distance value (T), and shape distance value (S). These values were used to compute the Gestalt value, G. The algorithm in this code sorts these g_values in ascending order, meaning that the output will be the top 3 images with the lowest G values, indicating the images that are most similar to the input image.

Later alteration to the algorithm: I utilized python’s dictionary functionality to optimize the script’s runtime. When calculating each Q photo’s *and* each contender’s photo’s metrics on the spot for every iteration, runtimes exceeded 2 minutes. This is why I thought to calculate all these metrics for each photo first, store them in a dictionary, and reference that dictionary in the iterations, thus reducing the runtime by a factor of 40.

Output Observations

For simple images, like those showing a single object with a distinct edge, usually in a round shape, and with a single dominant color, the algorithm would adequately rank photos. For photos with a high texture that looked like a pattern filling up the entire image, it would also adequately rank photos. One area where it fails, though, is its output for photos with blue or purple (a combination of red and blue).



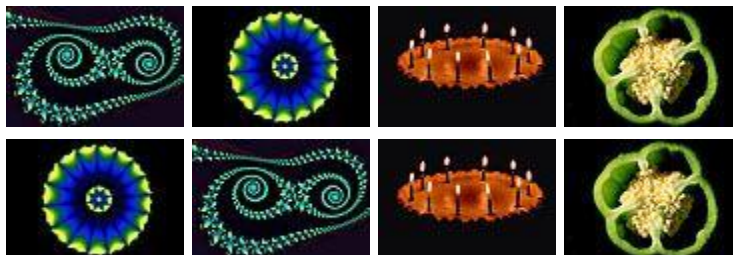
Highly textural image, i03 as Q. This row had the highest row score of the table, which was 576. Predominantly green.



Another example of a highly textural image that’s predominantly red. i06 as Q. Row score was 473.



Row i31 as Q, a photo with a purple swirl (indistinct edges and a non-predominantly RGB color scheme). The score here was 117.



The row scores for these rows (i27 and i32, respectively) were higher than that of i31, but a red photo appears in the rankings both times, which does not closely match the target image.

How Well I Believe My Design Worked

I think my design, overall, worked well for certain types of photos and not others, as I explained in the “Output Observations” above. In terms of grand total scores, when compared to textural rankings, shape rankings, and color rankings, it beat out color (the algorithm with the highest grand total score out of those three) by 1516. Therefore, I believe that with this combination of vectors, I was able to mimic the crowd’s opinion the most accurately.

Step 6: Alterations to the Gestalt Vector Composition

At first, I tried using the settings I had used when using the Crowd.txt file as the scoring mechanism. The output I got for my Personal Happiness Score, which was calculated in the same way as the Grand Total Score, was 34. These vector weights are as such:

```
#gestalt = grand_total_score = 11852
i_weight = 0.33      #color
s_weight = 0.22      #shape
t_weight = 0.11      #texture
```

I then experimented with the other distribution I had used in a previous step...

```
#gestalt1 = grand_total_score = 11205
i_weight = 0.33      #color
t_weight = 0.22      #texture
s_weight = 0.11      #shape
```

...and the resulting output increased to 35. Since the grand total score for color (10336) was far greater than both texture (6509) and shape (6217), and those last two numbers are similar, I thought to try this distribution:

```
#custom_gestalt3 = grand_total_score = 11205
i_weight = 0.50      #color
t_weight = 0.25      #texture
s_weight = 0.25      #shape
```

The personal happiness score decreased to 33, however, so I abandoned this approach in favor of the attempt right before it.

Improvement given from personalization

I saw the biggest improvement in terms of accuracy from the original Gestalt weights when using my customized Gestalt parameters. The difference would only regard one or two images, but it was an improvement nonetheless. An example of such is below:



i15: Personal Gestalt Weights. The last photo more closely resembles Q on the basis of color and texture.