GIAVANNA ZAREMSKI, GZ2337
COMS W4735 ASSIGNMENT 2
CONTENT-BASED IMAGE RETRIEVAL SYSTEM

Abstract

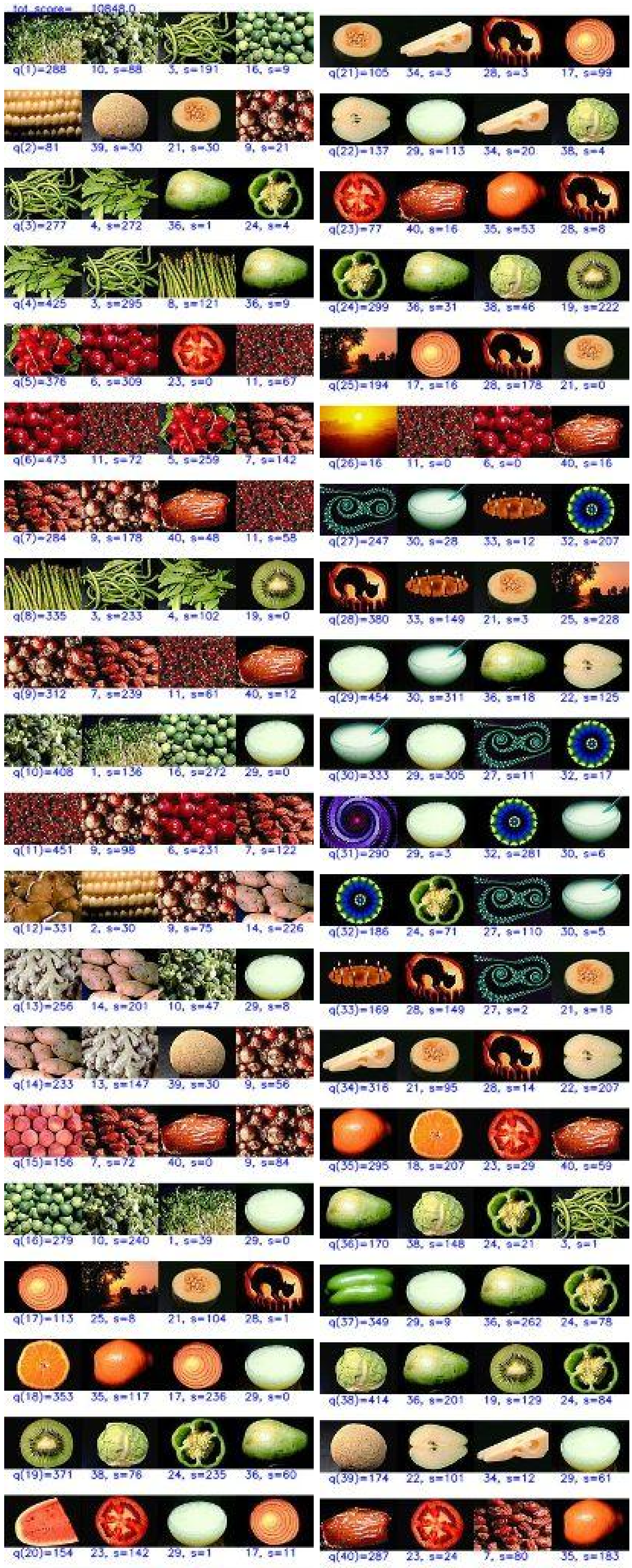
The goal of this project is to create a content-based image retrieval system that intends to analyze a set of images, compare them to one another and determine which three of the other images match most closely. The system accounts for the image's color, texture, shape, and symmetry. This project details the iterative process of developing the overall system using a crowd-sourced file of human results to determine the accuracy of the system. The code for this project can be found in the appendix and assumes there are 40 input images of size 60x89. The project makes use of Python and the OpenCV library to carry out the intended function.

Part 1: Color Identification

One of the main ways humans categorize visual input is through the use of color. This project utilizes color histograms to compare images' color to one another. First, the input images are read as JPG files through the use of OpenCV. In order to reduce the possible color combinations from (255,255,255) the number of identifiable colors by the system are reduced by factors. Originally, the function was implemented with equal factors for blue, green and red at (3,3,3). This resulted in a score of around 10000, or around a 55% oracle success rate as compared to the crowd-sourced data. Due to the large number of possible combinations, a script was used to maximize the score by iterating through a list of reasonable possible values for each color. The end result was a color binning of (5,6,6) resulting in a total possible recognizable colors in an image to be 180. The numbers for green and red are slightly higher than blue. The cones of human eyes are more sensitive to red and green color changes so this deviation is logical (Higham). For this binning, the score is around 11000, or an oracle success rate of 61%.

The set intersection of the program's versus the author's preferences was calculated by summing the overlapping target images for each query image. The result for this intersection was found to be 52/120, around a 43% match. Overall, the performance regarding color is promising. The results have better performance towards the crowd results, but still perform rather well for the personal author preferences. Since this program only considered color and not other measures that a human may consider such as shape or texture, the results are to be expected.

The total and individual results from the program are displayed in the image below.

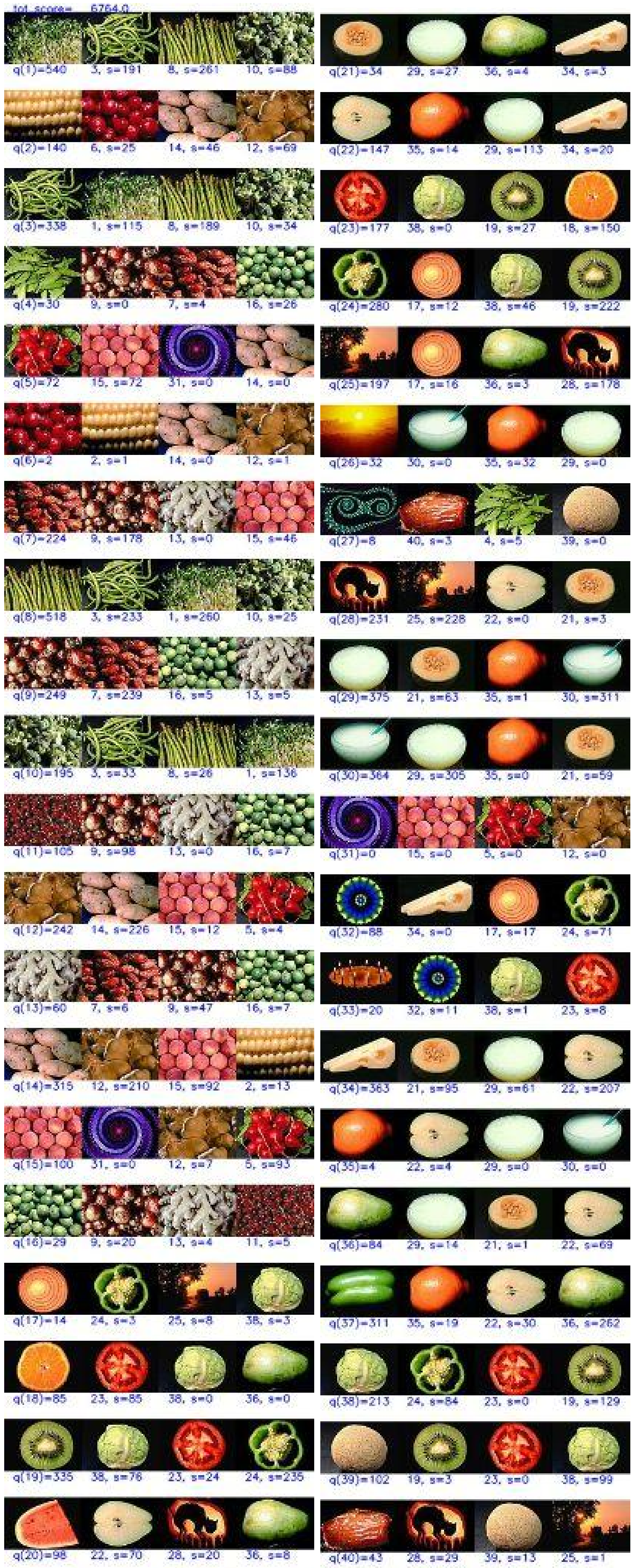

tot_score= 10848.0
q(1)=288 10, s=88 3, s=191 16, s=9
q(2)=81 39, s=30 21, s=30 9, s=21
q(3)=277 4, s=272 36, s=1 24, s=4
q(4)=425 3, s=295 8, s=121 36, s=9
q(5)=376 6, s=309 23, s=0 11, s=67
q(6)=473 11, s=72 5, s=259 7, s=142
q(7)=284 9, s=178 40, s=48 11, s=58
q(8)=335 3, s=233 4, s=102 19, s=0
q(9)=312 7, s=239 11, s=61 40, s=12
q(10)=408 1, s=136 16, s=272 29, s=0
q(11)=451 9, s=98 6, s=231 7, s=122
q(12)=331 2, s=30 9, s=75 14, s=226
q(13)=256 14, s=201 10, s=47 29, s=8
q(14)=233 13, s=147 39, s=30 9, s=56
q(15)=156 7, s=72 40, s=0 9, s=84
q(16)=279 10, s=240 1, s=39 29, s=0
q(17)=113 25, s=8 21, s=104 28, s=1
q(18)=353 35, s=117 17, s=236 29, s=0
q(19)=371 38, s=76 24, s=235 36, s=60
q(20)=154 23, s=142 29, s=1 17, s=11
q(21)=105 34, s=3 28, s=3 17, s=99
q(22)=137 29, s=113 34, s=20 38, s=4
q(23)=77 40, s=16 35, s=53 28, s=8
q(24)=299 36, s=31 38, s=46 19, s=222
q(25)=194 17, s=16 28, s=178 21, s=0
q(26)=16 11, s=0 6, s=0 40, s=16
q(27)=247 30, s=28 33, s=12 32, s=207
q(28)=380 33, s=149 21, s=3 25, s=228
q(29)=454 30, s=311 36, s=18 22, s=125
q(30)=333 29, s=305 27, s=11 32, s=17
q(31)=290 29, s=3 32, s=281 30, s=6
q(32)=186 24, s=71 27, s=110 30, s=5
q(33)=169 28, s=149 27, s=2 21, s=18
q(34)=316 21, s=95 28, s=14 22, s=207
q(35)=295 18, s=207 23, s=29 40, s=59
q(36)=170 38, s=148 24, s=21 3, s=1
q(37)=349 29, s=9 36, s=262 24, s=78
q(38)=414 36, s=201 19, s=129 24, s=84
q(39)=174 22, s=101 34, s=12 29, s=61
q(40)=287 23, s=24 7, s=80 35, s=183

Part 2: Texture Identification

The next aspect of an image the program considers is the texture of the image. In order to find the edginess of each image, the images are first converted to grayscale and then each pixel is converted to their laplacian values. Then, as in the color step, the values are binned and for each image the pixels are summed if their value falls into a certain bin. The images are then compared to one another to find similarly edgy images. The program first attempted to bin into 180 possible 'textures' to match the color binning. This resulted in a score of about 5700, or an oracle success rate of around 32%. It was to be expected the texture algorithm would not perform as well as the color algorithm as it would not be the primary way humans categorize images. However, the binning allowed room for improvement. A higher number of bins was required to more accurately assess the images texture, yet after about 1100 bins the performance began to degrade. At this point the images became too specifically textured and true matches would likely be ignored. Peak performance was found at 1000 bins with a score of 6800 and an oracle success rate of around 38%.

The set intersection for the author's results versus the program's were calculated as described above. In this case the result was 29/120 or 16% match. Performance versus the author's results degrades considerably versus the program. The discrepancy between the author and the program could be for a number of reasons, but mainly the author may not use texture as a major differentiating factor for images. The range of answers given by the crowd helps the texture algorithm perform better where the personal results fail.

From the image below the results can be analyzed to see most of the images with a large central object are matched with one another as those tend to have lower amounts of texture than the images with a large amount of smaller numbers. From the crowd-sourcing results it is clear most people were more likely to match images of lots of smaller objects with each other than with an image of one large central object. Compared to the color algorithm which would sometimes match textured images with non-textured images, this algorithm avoids that mistake. Therefore, we can conclude that the algorithm's performance is desirable and will be an important factor for the overall gestalt computation.



tot score= 6764.0
q(1)=540 3, s=191 8, s=261 10, s=88
q(2)=140 6, s=25 14, s=46 12, s=69
q(3)=338 1, s=115 8, s=189 10, s=34
q(4)=30 9, s=0 7, s=4 16, s=26
q(5)=72 15, s=72 31, s=0 14, s=0
q(6)=2 2, s=1 14, s=0 12, s=1
q(7)=224 9, s=178 13, s=0 15, s=46
q(8)=518 3, s=233 1, s=260 10, s=25
q(9)=249 7, s=239 16, s=5 13, s=5
q(10)=195 3, s=33 8, s=26 1, s=136
q(11)=105 9, s=98 13, s=0 16, s=7
q(12)=242 14, s=226 15, s=12 5, s=4
q(13)=60 7, s=6 9, s=47 16, s=7
q(14)=315 12, s=210 15, s=92 2, s=13
q(15)=100 31, s=0 12, s=7 5, s=93
q(16)=29 9, s=20 13, s=4 11, s=5
q(17)=14 24, s=3 25, s=8 38, s=3
q(18)=85 23, s=85 38, s=0 36, s=0
q(19)=335 38, s=76 23, s=24 24, s=235
q(20)=98 22, s=70 28, s=20 36, s=8
q(21)=34 29, s=27 36, s=4 34, s=3
q(22)=147 35, s=14 29, s=113 34, s=20
q(23)=177 38, s=0 19, s=27 18, s=150
q(24)=280 17, s=12 38, s=46 19, s=222
q(25)=197 17, s=16 36, s=3 28, s=178
q(26)=32 30, s=0 35, s=32 29, s=0
q(27)=8 40, s=3 4, s=5 39, s=0
q(28)=231 25, s=228 22, s=0 21, s=3
q(29)=375 21, s=63 35, s=1 30, s=311
q(30)=364 29, s=305 35, s=0 21, s=59
q(31)=0 15, s=0 5, s=0 12, s=0
q(32)=88 34, s=0 17, s=17 24, s=71
q(33)=20 32, s=11 38, s=1 23, s=8
q(34)=363 21, s=95 29, s=61 22, s=207
q(35)=4 22, s=4 29, s=0 30, s=0
q(36)=84 29, s=14 21, s=1 22, s=69
q(37)=311 35, s=19 22, s=30 36, s=262
q(38)=213 24, s=84 23, s=0 19, s=129
q(39)=102 19, s=3 23, s=0 38, s=99
q(40)=43 28, s=29 39, s=13 25, s=1

Part 3: Shape Identification

The next iteration of the program aimed to analyze the shape regarding the foreground and background discrepancies between each pair of images. First, each image is converted from its grayscale form to a binary image, with the intention for background pixels to be given a value of pure black, 255, and foreground pixels to be given a value of 0, pure white. The binary conversion was done using the openCV thresholding algorithm, originally with a value of 127. If a pixel had a value greater than the threshold value it would become black, white otherwise. This resulted in a score of 5100, or 28%. After the author analyzed the binary intermediates between images the author would have assumed to match well shape-wise, it was noticed the edges of images were very distinct which could cause the algorithm to incorrectly claim images that should have been a match as not so. To counteract this, before each binary conversion the images were blurred using the openCV blur function which resulted in smoother edges for each image. After blurring was implemented and with a threshold value of 127 the new results were 6000, or 33%. After experimenting with both factors the best results were obtained with a blur of kernel size (30,30) and a threshold value of 127 which results in the 33% oracle success rate.

The personal happiness score found by set intersection for this algorithm is 33/120 or 28%. This algorithm performed slightly worse in regards to the crowd data versus the texture algorithm. However, its results aligned better with the author's choices. This could perhaps occur because the author places more emphasis on foreground-background shape than texture when comparing images.

It is important to note when viewing the results below, that a great deal of pictures are matched to images 5,6,7. This is due to the blur being performed before the images are converted to binary. A number of the more highly textured images are blurred so much it results in a pure black image which when compared with other pure black images will result in a perfect match. Through testing, it was determined that the algorithm performed better when compared to the crowd data if these images were ignored by the blurring to pure black. If the images are not blurred, the algorithm performs better subjectively when viewing the results through the lens of matching shapes to other shapes. However, since the ultimate goal of this program is to maximize results for the crowd data, the algorithm will be considered with this blur enacted. If a user wanted to simply view shape comparisons, the blur can be reduced easily and the results can be viewed. Holistically, we can view this blur choice as the idea that shape is very important for human comparisons when it is readily apparent such as in the large central figure images, but for highly textured images it is less important.

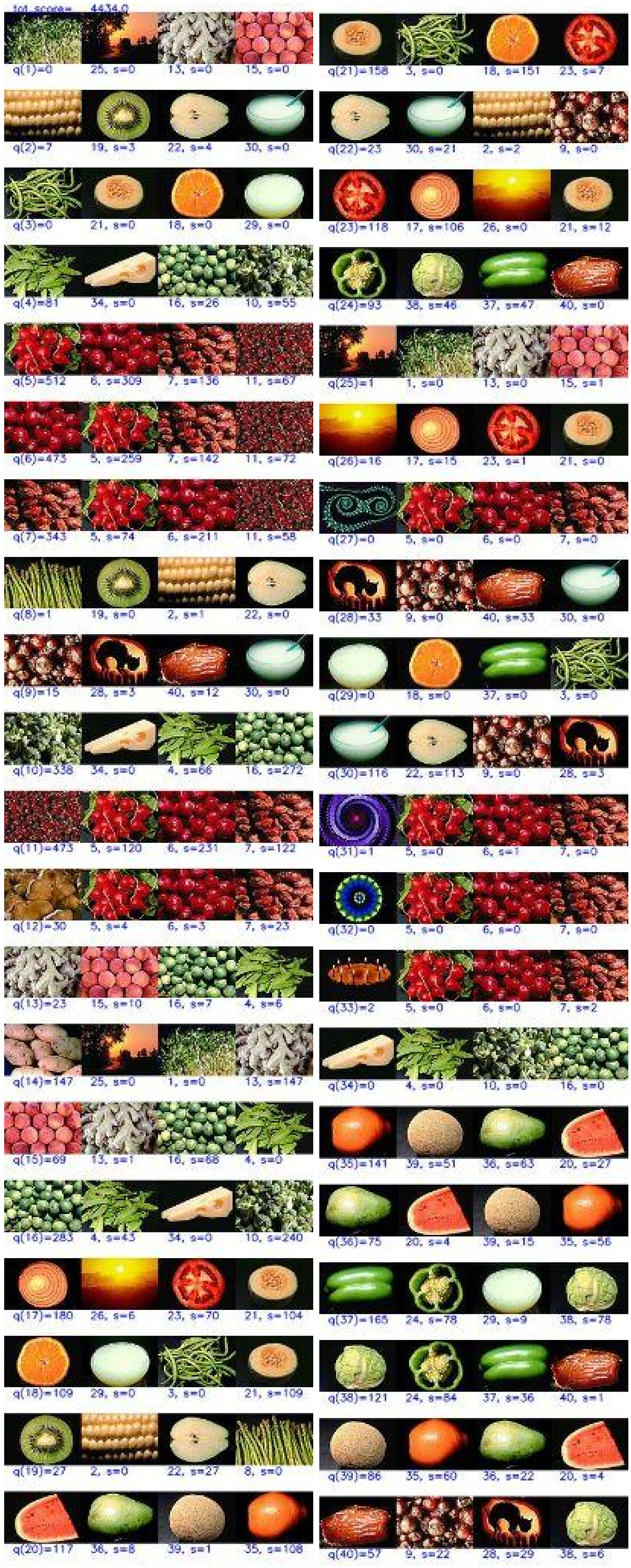

tot score= 6069.0
q(1)=261 8, s=261 5, s=0 6, s=0
q(2)=123 26, s=36 25, s=3 16, s=84
q(3)=0 5, s=0 6, s=0 7, s=0
q(4)=11 5, s=7 6, s=0 7, s=4
q(5)=512 6, s=309 7, s=136 11, s=67
q(6)=473 5, s=259 7, s=142 11, s=72
q(7)=343 5, s=74 6, s=211 11, s=58
q(8)=0 5, s=0 6, s=0 7, s=0
q(9)=455 5, s=40 6, s=176 7, s=239
q(10)=13 5, s=2 6, s=1 7, s=10
q(11)=473 5, s=120 6, s=231 7, s=122
q(12)=30 5, s=4 6, s=3 7, s=23
q(13)=219 14, s=201 29, s=8 15, s=10
q(14)=147 13, s=147 29, s=0 20, s=0
q(15)=0 30, s=0 18, s=0 38, s=0
q(16)=66 5, s=7 6, s=52 7, s=7
q(17)=375 38, s=3 21, s=104 18, s=268
q(18)=237 38, s=0 17, s=236 22, s=1
q(19)=265 24, s=235 21, s=28 3, s=2
q(20)=12 17, s=11 38, s=0 29, s=1
q(21)=123 19, s=13 24, s=11 17, s=99
q(22)=198 18, s=18 29, s=113 39, s=67
q(23)=14 5, s=5 6, s=9 7, s=0
q(24)=228 19, s=222 40, s=0 21, s=6
q(25)=0 5, s=0 6, s=0 7, s=0
q(26)=304 2, s=14 36, s=0 25, s=290
q(27)=0 5, s=0 6, s=0 7, s=0
q(28)=0 5, s=0 6, s=0 7, s=0
q(29)=438 22, s=125 30, s=311 20, s=2
q(30)=69 38, s=10 21, s=59 37, s=0
q(31)=1 5, s=0 6, s=1 7, s=0
q(32)=0 5, s=0 6, s=0 7, s=0
q(33)=2 5, s=0 6, s=0 7, s=2
q(34)=134 21, s=95 30, s=21 40, s=18
q(35)=92 21, s=24 36, s=63 37, s=5
q(36)=262 39, s=15 35, s=56 37, s=191
q(37)=57 21, s=0 35, s=19 19, s=38
q(38)=7 17, s=0 18, s=2 30, s=5
q(39)=101 36, s=22 18, s=19 35, s=60
q(40)=24 24, s=0 23, s=24 5, s=0

Part 4: Symmetry Identification

The final aspect of each image to consider is the image symmetry. This is carried out on the binary version of each image. Each column is compared pixel-wise to its twin on the other side of the image. The normalized sum of matching pixels then gives the final symmetric result. Each image's symmetric result is then compared to the other images' results to find the closest matching images by symmetry. Using the same binary thresholds and blurring resulted in a score of 3432, or 19%. The first thought was to reduce the blur on the images as it is very high and causing many images to be pure black and perfectly symmetric which would not be the actual case. Reducing the blur to a kernel size of (15,15) increased the results to an oracle success rate of 22%, which is an improvement but still not performing as well as hoped. The next consideration to include is the binary threshold. By increasing the binary threshold to 140, the new results are a score of 4434 or 24%. Further testing was unable to increase the success rate past this point. Due to the symmetry being calculated purely vertically and also by pixel could be why it does not match from a human point of view. The normalized results give images that may have the same average symmetry, but this symmetry could be completely different image to image. For example, one could lean more towards the right of the image and the other towards the left, but if they lean the same amount they will be considered a match.

The personal happiness score assessed for the symmetry algorithm is 26/120 or 22%. This is the first case where the results for crowd and author are very close percentage wise. Likely, the author and crowd as a whole view symmetry similarly and place a similar weight when comparing images.

From the results displayed below, some of the methods from the algorithm can be seen in practice. Image number 26 which is a landscape sunset that fills the entire image is matched to only images that are large focal points with the black background. The sunset is very symmetric vertically as are these focally large images so the comparison makes logical sense from the algorithm's point of view. However, from a human point of view, these images would not likely be marked as visually similar due to subject background, texture and color. Image 32 is also a symmetric image, however, due to the blur it is changed to pure black through binarization. This is a failure of the algorithm but reducing the blur results in poorer performance over a greater number of images.

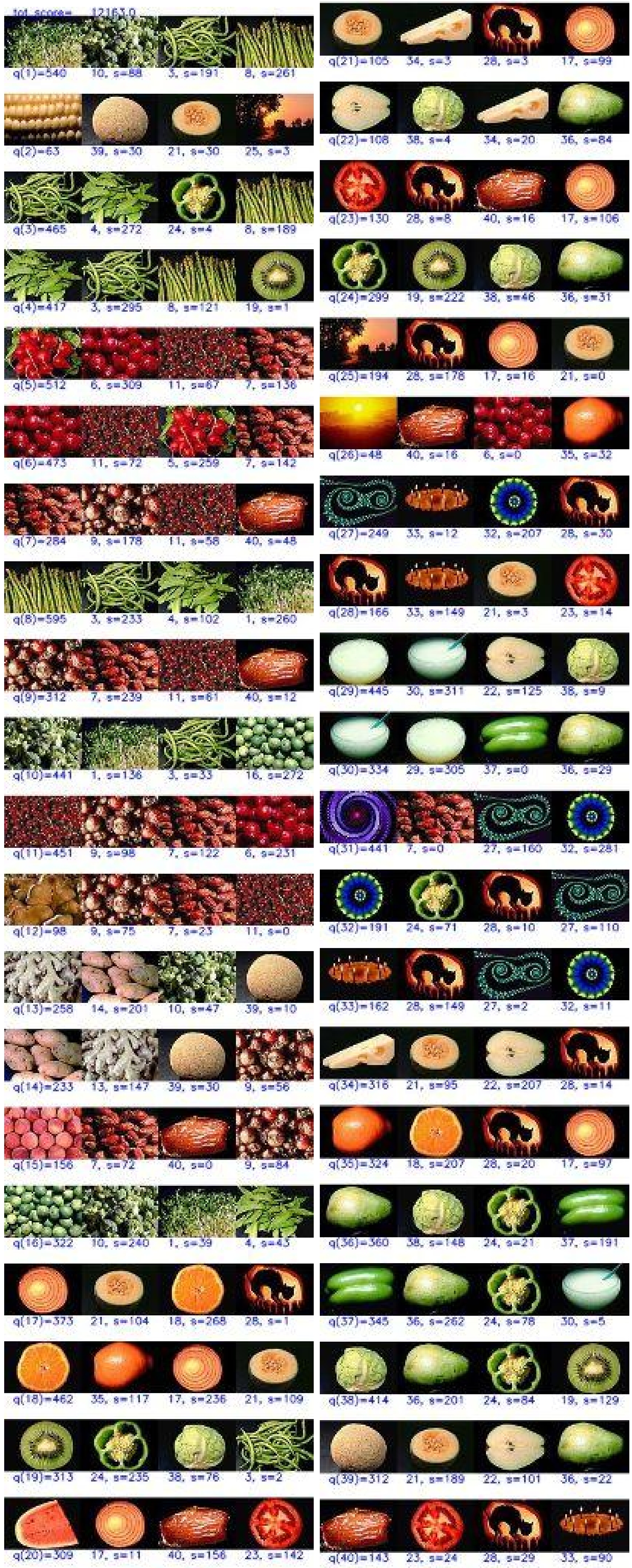

tot score= 4434.0
q(1)=0 25, s=0 13, s=0 15, s=0
q(2)=7 19, s=3 22, s=4 30, s=0
q(3)=0 21, s=0 18, s=0 29, s=0
q(4)=81 34, s=0 16, s=26 10, s=55
q(5)=512 6, s=309 7, s=136 11, s=67
q(6)=473 5, s=259 7, s=142 11, s=72
q(7)=343 5, s=74 6, s=211 11, s=58
q(8)=1 19, s=0 2, s=1 22, s=0
q(9)=15 28, s=3 40, s=12 30, s=0
q(10)=338 34, s=0 4, s=66 16, s=272
q(11)=473 5, s=120 6, s=231 7, s=122
q(12)=30 5, s=4 6, s=3 7, s=23
q(13)=23 15, s=10 16, s=7 4, s=6
q(14)=147 25, s=0 1, s=0 13, s=147
q(15)=69 13, s=1 16, s=68 4, s=0
q(16)=283 4, s=43 34, s=0 10, s=240
q(17)=180 26, s=6 23, s=70 21, s=104
q(18)=109 29, s=0 3, s=0 21, s=109
q(19)=27 2, s=0 22, s=27 8, s=0
q(20)=117 36, s=8 39, s=1 35, s=108
q(21)=158 3, s=0 18, s=151 23, s=7
q(22)=23 30, s=21 2, s=2 9, s=0
q(23)=118 17, s=106 26, s=0 21, s=12
q(24)=93 38, s=46 37, s=47 40, s=0
q(25)=1 1, s=0 13, s=0 15, s=1
q(26)=16 17, s=15 23, s=1 21, s=0
q(27)=0 5, s=0 6, s=0 7, s=0
q(28)=33 9, s=0 40, s=33 30, s=0
q(29)=0 18, s=0 37, s=0 3, s=0
q(30)=116 22, s=113 9, s=0 28, s=3
q(31)=1 5, s=0 6, s=1 7, s=0
q(32)=0 5, s=0 6, s=0 7, s=0
q(33)=2 5, s=0 6, s=0 7, s=2
q(34)=0 4, s=0 10, s=0 16, s=0
q(35)=141 39, s=51 36, s=63 20, s=27
q(36)=75 20, s=4 39, s=15 35, s=56
q(37)=165 24, s=78 29, s=9 38, s=78
q(38)=121 24, s=84 37, s=36 40, s=1
q(39)=86 35, s=60 36, s=22 20, s=4
q(40)=57 9, s=22 28, s=29 38, s=6

Part 5: Overall Gestalt

The overall gestalt computation is done using the algorithms generated above and each given a unique factor by which to use them in the final distance between two images. The first testing was done using a constant factor of .25 for each aspect. This resulted in a score of 10500 or 58%. This is a promising starting point but it is performing worse than by color alone which is not desirable. By increasing the factor of color to .50 and reducing texture, shape, and symmetry to .20, .20, and .10 respectively results in a score of 12200 or 68%. This result is much more accurate. Further attempts to better the results by increasing the color factor failed as did changing other factors. The final best results were (.50, .20, .20, .10).

The set intersection for these results were 58/120, or 48%. This is by far the best match between the program and the author's personal results. The average match between any two people in the crowd data was 56/120, so the algorithm matched a little higher than average in this case. The program is about as similar to a human-human comparison which is desirable performance for this algorithm. One aspect the program fails to consider is the naturalness of each image. There are some unnatural images, although most are fruit or some nature. Most humans grouped the unnatural images together and would not list them as the same towards the natural images. The algorithm does not have this ability, and sometimes unnatural images are paired with natural images reducing overall performance.

The visualization is displayed below and subjectively very few images seem inaccurate to a human viewer. The combination of each aspect greatly improved the overall results. Color reduces many of the incongruencies that shape, symmetry, and texture judged incorrectly, whereas they all helped to improve where color fails.



tot_score= 12163.0
q(1)=540 10, s=88 3, s=191 8, s=261
q(2)=63 39, s=30 21, s=30 25, s=3
q(3)=465 4, s=272 24, s=4 8, s=189
q(4)=417 3, s=295 8, s=121 19, s=1
q(5)=512 6, s=309 11, s=67 7, s=136
q(6)=473 11, s=72 5, s=259 7, s=142
q(7)=284 9, s=178 11, s=58 40, s=48
q(8)=595 3, s=233 4, s=102 1, s=260
q(9)=312 7, s=239 11, s=61 40, s=12
q(10)=441 1, s=136 3, s=33 16, s=272
q(11)=451 9, s=98 7, s=122 6, s=231
q(12)=98 9, s=75 7, s=23 11, s=0
q(13)=258 14, s=201 10, s=47 39, s=10
q(14)=233 13, s=147 39, s=30 9, s=56
q(15)=156 7, s=72 40, s=0 9, s=84
q(16)=322 10, s=240 1, s=39 4, s=43
q(17)=373 21, s=104 18, s=268 28, s=1
q(18)=462 35, s=117 17, s=236 21, s=109
q(19)=313 24, s=235 38, s=76 3, s=2
q(20)=309 17, s=11 40, s=156 23, s=142
q(21)=105 34, s=3 28, s=3 17, s=99
q(22)=108 38, s=4 34, s=20 36, s=84
q(23)=130 28, s=8 40, s=16 17, s=106
q(24)=299 19, s=222 38, s=46 36, s=31
q(25)=194 28, s=178 17, s=16 21, s=0
q(26)=48 40, s=16 6, s=0 35, s=32
q(27)=249 33, s=12 32, s=207 28, s=30
q(28)=166 33, s=149 21, s=3 23, s=14
q(29)=445 30, s=311 22, s=125 38, s=9
q(30)=334 29, s=305 37, s=0 36, s=29
q(31)=441 7, s=0 27, s=160 32, s=281
q(32)=191 24, s=71 28, s=10 27, s=110
q(33)=162 28, s=149 27, s=2 32, s=11
q(34)=316 21, s=95 22, s=207 28, s=14
q(35)=324 18, s=207 28, s=20 17, s=97
q(36)=360 38, s=148 24, s=21 37, s=191
q(37)=345 36, s=262 24, s=78 30, s=5
q(38)=414 36, s=201 24, s=84 19, s=129
q(39)=312 21, s=189 22, s=101 36, s=22
q(40)=143 23, s=24 28, s=29 33, s=90

Part 6: Crowd vs. Personal

After generating a sparse matrix based on the author's personal opinions, the gestalt algorithm was run again with the same aspect factor values of (.50, .20, .20, .10). This resulted in a score of 126, or around 53%.Optimization could not find a weighted vector that performed better than the original vector. The first choice was to increase the color and shape factors and reduce the texture as the personal happiness scores calculated from the individual algorithms indicate a better match for those over texture. However, the performance degraded. Perhaps the author was already very close to the average in their scoring to the crowd so attempts to personalize failed. The difference between one person versus the crowd largely stems from the way the individual defines sameness between images. For the author they just happened to not have a different view than most regarding most of the images.

Conclusion

Overall, the performance of the system was satisfactory. The individual algorithms for each aspect had a lower performance, but when combined were able to reach a level that would please the majority of a crowd. Future improvements could be made to the way each aspect is calculated in order to further increase the accuracy of the system. For example, it could consider horizontal symmetry along with vertical symmetry. A future iteration of the program could also attempt to detect local colors if it were given images without a stark contrast between the main image and the background. The functions described in this program can be found in the Appendix.

Citations

Divyanshu. (2023, January 3). *Concatenate images using opencv in python*. GeeksforGeeks. Retrieved March 9, 2023, from https://www.geeksforgeeks.org/concatenate-images-using-opencv-in-python/

Higham, J. P. (2021, April 8). *The red and green specialists: Why human colour vision is so odd: Aeon ideas*. Aeon. Retrieved March 9, 2023, from https://aeon.co/ideas/the-red-and-green-specialists-why-human-colour-vision-is-so-odd

Appendix

1. Crowd.txt

0	0	191	80	0	0	0	261	0	88	1	0	0	0	0	9	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	5	1	7	25	15	11	21	3	14	69	8	46	106	84	11	7	3	0	30	4	0	6	3	36	0	0	0	0	0	0	5	64	5	3	5	0	30	2
115	0	0	272	0	0	0	189	0	34	0	0	0	0	0	9	0	0	0	0	0	0	0	4	0	0	1	0	0	0	0	0	0	0	0	1	5	0	0	0
88	0	295	0	7	0	4	121	0	55	0	0	6	1	0	26	0	0	1	0	0	0	0	1	0	0	1	0	0	0	0	0	0	0	0	9	13	2	0	0
0	0	0	1	0	309	136	0	35	0	67	2	0	0	72	5	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0
0	1	0	0	259	0	142	1	52	0	72	1	0	0	94	2	0	0	1	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	1
0	0	0	2	74	211	0	0	178	6	58	1	0	6	46	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	48
260	1	233	102	0	0	0	0	0	25	0	0	0	0	0	2	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	5	1	0	0
0	0	1	0	40	176	239	2	0	7	61	18	5	9	49	5	3	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	12
136	0	33	66	2	1	10	26	24	0	6	8	20	0	5	272	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	17	0	0
3	1	0	0	120	231	122	1	98	8	0	0	0	0	36	7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3
0	30	0	3	4	3	23	0	75	14	0	0	175	226	12	1	9	2	1	1	3	2	0	0	3	0	0	4	0	0	0	0	5	1	4	3	0	0	20	6
3	19	0	6	4	1	6	0	47	47	0	230	0	201	10	7	3	0	0	0	2	10	0	0	0	0	0	1	8	3	0	1	0	10	1	0	0	0	10	0
0	13	0	1	3	8	43	0	56	1	1	210	147	0	92	2	2	0	0	0	0	5	0	0	0	1	0	0	0	0	0	0	0	2	5	7	0	0	30	1
0	9	0	0	93	242	72	0	84	0	22	7	1	29	0	68	1	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
39	5	24	43	7	52	7	6	20	240	5	0	4	3	129	0	0	2	6	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	25	2	10	0	0
0	0	0	0	0	0	0	0	2	0	0	3	2	2	0	0	0	268	42	7	104	2	70	3	8	6	1	1	3	0	20	7	0	0	66	0	0	3	10	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	236	0	51	7	109	1	85	3	0	3	0	0	0	0	0	14	1	2	117	0	0	0	1	0
0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	34	65	0	3	28	27	24	235	0	0	1	4	6	4	2	51	0	0	0	60	7	76	1	0
0	0	0	0	6	3	1	0	0	0	1	0	0	0	1	0	11	10	4	0	71	70	142	1	0	0	0	20	1	0	0	0	11	3	108	8	1	0	1	156
0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	99	151	13	29	0	157	7	11	0	0	0	3	27	5	0	0	8	3	16	4	0	1	93	0
0	2	0	0	0	0	0	0	0	0	0	0	0	2	2	0	9	18	24	31	203	0	2	8	0	0	1	0	113	21	0	1	1	20	14	84	2	4	67	1
0	0	0	0	5	9	0	0	0	0	0	0	0	0	1	0	106	150	27	76	12	0	0	154	0	0	0	8	0	0	0	7	1	0	53	1	2	0	2	16
0	0	3	0	1	0	0	0	0	0	0	0	0	0	0	6	12	18	222	0	6	7	165	0	0	1	5	4	3	0	1	51	0	0	0	31	47	46	1	0
0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	16	7	1	1	0	0	0	0	0	292	7	178	0	0	0	1	96	5	11	3	0	0	0	10
0	14	0	0	0	0	0	0	0	0	0	0	0	0	3	0	15	19	1	1	0	0	1	0	290	0	6	148	0	0	1	0	82	1	32	0	0	0	0	16
4	3	12	5	0	0	0	1	0	10	0	0	1	0	0	1	8	0	21	0	2	1	0	32	5	4	0	30	2	28	197	207	12	0	0	18	22	1	0	3
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	8	4	0	10	3	0	14	2	228	115	12	0	0	0	5	3	149	0	40	2	1	0	0	33
0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	2	0	2	2	63	125	0	2	0	1	5	3	0	311	0	8	6	10	1	18	0	9	58	1
0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	9	0	59	113	0	2	0	0	11	3	305	0	7	17	8	10	0	29	0	10	42	2
0	0	0	0	0	1	0	0	1	0	1	0	0	0	0	0	63	3	20	0	0	0	24	14	1	6	160	16	3	6	0	281	25	0	1	0	1	1	0	2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	17	24	107	0	10	0	17	71	0	1	110	10	3	5	229	0	17	0	2	3	0	2	1	1
0	0	0	0	0	0	2	0	1	0	0	0	0	0	0	0	28	16	0	7	18	3	8	0	118	60	2	149	4	5	2	11	0	10	99	0	0	1	0	86
0	91	0	0	0	0	0	0	0	0	0	2	4	10	0	0	1	0	0	2	95	207	0	0	1	0	5	14	61	21	0	0	17	0	11	9	13	1	47	18
0	1	0	0	0	1	0	0	0	0	0	1	0	2	0	0	97	207	0	27	24	4	29	0	1	9	2	20	0	0	0	0	20	3	0	63	5	4	51	59
0	2	1	7	0	0	0	0	0	0	0	0	0	0	0	6	2	3	71	4	1	69	0	21	1	0	7	1	14	7	0	0	0	1	56	0	191	148	15	2
3	4	16	23	0	0	0	6	0	3	0	0	0	0	0	10	0	0	38	2	0	30	2	78	0	6	9	2	9	5	0	1	0	5	19	262	0	78	2	17
3	0	4	11	0	0	0	3	0	16	0	0	0	0	0	11	0	2	129	0	2	4	0	84	0	3	1	0	22	5	0	0	0	0	7	201	36	0	85	1
0	6	0	0	0	0	0	0	0	0	0	1	0	7	0	0	20	19	3	4	189	101	0	4	0	0	0	1	61	10	0	0	0	12	60	22	2	99	0	9
0	0	0	0	0	4	80	0	22	1	1	0	0	0	1	0	9	7	0	94	11	5	24	0	1	7	0	29	0	0	0	0	90	5	183	16	21	6	13	0

2. gz2337.txt

01	08	03	10
02	26	34	22
03	04	10	16
04	03	36	16
05	06	07	09
06	05	07	15
07	06	05	11
08	01	04	03
09	07	06	05
10	16	01	03
11	06	07	15
12	40	28	13
13	14	12	28
14	12	13	22
15	06	09	05
16	10	06	04
17	18	21	23
18	17	35	21
19	24	38	36
20	40	35	23
21	18	22	17
22	21	18	38
23	35	18	17
24	23	19	29
25	26	28	33
26	25	35	33
27	32	31	17
28	33	25	40
29	30	22	34
30	29	27	32
31	32	27	17
32	27	24	31
33	25	28	40
34	22	02	29
35	18	36	21
36	37	35	38
37	36	35	40
38	39	19	24
39	38	22	21
40	20	09	07

3. gz2337_borda.txt

```
0	0	2	0	0	0	0	3	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	3
0	0	0	0	0	0	0	2	0	0	0	0	0
0
0	0	0	3	0	0	0	0	0	2	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0
0	0	3	0	0	0	0	0	0	0	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	2	0	0	0
0
0	0	0	0	0	3	2	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0
0	0	0	0	3	0	2	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0
0	0	0	0	2	3	0	0	0	0	1	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0
3	0	1	2	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0
```

0	0	0	0	1	2	3	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0												
2	0	1	0	0	0	0	0	0	0	0	0	0
0	0	3	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0												
0	0	0	0	0	3	2	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0												
0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0
0	2	0	0	0	0	0	0	0	0	0	0	0
3												
0	0	0	0	0	0	0	0	0	0	0	2	0
3	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0
0												
0	0	0	0	0	0	0	0	0	0	0	3	2
0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0												
0	0	0	0	1	3	0	0	2	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0												
0	0	0	1	0	2	0	0	0	3	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
0												
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	3	0	0	2	0	1	0	0	0

0 0 0 0 0 0 0 0 0 0 0 0 0

0

0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 3 0 0 0 1 0 0 0 0 0

0 0 0 0 0 0 0 0 2 0 0 0 0

0

0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 3 0 0

0 0 0 0 0 0 0 0 0 1 0 2 0

0

0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 1 0 0 0

0 0 0 0 0 0 0 0 2 0 0 0 0

3

0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 1 3 0 0 0 2 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 0

0

0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 2 0 0 3 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 1 0

0

0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 1 2 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 3 0 0 0 0

0

0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 2 0 0 0 3 0 0 0

0 0 1 0 0 0 0 0 0 0 0 0 0

0

0 0 0 0 0 0 0 0 0 0 0 0 0

0 0 0 0 0 0 0 0 0 0 0 0 3

0 2 0 0 0 0 1 0 0 0 0 0 0

0

0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	3	0
0	0	0	0	0	0	1	0	2	0	0	0	0
0												
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	2	3	0	0	0	0	0	0	0
0												
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	2	0
0	0	0	0	0	0	3	0	0	0	0	0	0
1												
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	2	0	0	0	0
0	0	0	3	0	0	0	1	0	0	0	0	0
0												
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0
2	0	3	0	0	1	0	0	0	0	0	0	0
0												
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0	0	0	0	0	0
2	0	0	0	0	3	0	0	0	0	0	0	0
0												
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0
3	0	0	0	0	0	0	2	0	0	0	0	0
0												
0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	3	0
0	2	0	0	0	0	0	0	0	0	0	0	0
1												
0	2	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	3	0	0	0	0

0 0 1 0 0 0 0 0 0 0 0 0 0
0
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 3 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 2 0 0 0
0
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 2 0 3 1 0
0
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 2 3 0 0 0
1
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 2 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 3
0
0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 2 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 3 0
0
0 0 0 0 0 0 1 0 2 0 0 0 0
0 0 0 0 0 0 3 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0
0

4. Code

```
import cv2
import numpy as np
import math

#color bins
blue = 5
green = 6
red = 6

#texture bins
texture = 1000

#number of input images
num_imgs = 40

#loads data from crowd and author
crowd_data = np.loadtxt("gz2337_borda.txt")
my_data = np.loadtxt('gz2337.txt')

# variables for generating text for results display
font = cv2.FONT_HERSHEY_SIMPLEX
org = (10, 80)
fontScale = 0.4
color = (255, 0, 0)
thickness = 1


def color_histogram(img_str, red = red, blue = blue, green = green):
    '''
    purpose: generates color histogram of image given rgb values

    inputs:
    img_str = string of image file to be analyzed
    red = number of red bins
    blue = number of blue bins
    green = number of red bins

    output:
    color_hist = histogram of image with colors binned according to givens
    '''
    #read image through opencv
```

```
    img = cv2.imread(img_str)

    #storage for histogram
    color_hist = {}

    #generate factors to reduce pixel values
    r_factor = math.floor(256/red)
    b_factor = math.floor(256/blue)
    g_factor = math.floor(256/green)

    #find size of image
    dim = img.shape

    #iterate through image rows
    for i in range(dim[0]):
        #iterate through image columns
        for j in range(dim[1]):

            #retrive pixel and BGR values
            pixel = img[i,j]
            b_val = pixel[0]
            g_val = pixel[1]
            r_val = pixel[2]

            #reduce values to requested number of bins
            b_val_fac = math.floor(b_val/b_factor)
            g_val_fac = math.floor(g_val/g_factor)
            r_val_fac = math.floor(r_val/r_factor)

            #want to zero index bins, in case of match to bin # reduce
            if b_val_fac == blue:
                b_val_fac = b_val_fac - 1

            if g_val_fac == green:
                g_val_fac = g_val_fac - 1

            if r_val_fac == red:
                r_val_fac = r_val_fac - 1

            #generate color string
            color_str = str(b_val_fac) + "," + str(g_val_fac) + "," +
str(r_val_fac)

            #if color already exists in image add to count, else add to
histogram
```

```
            if color_str in color_hist:
                color_hist[color_str] += 1
            else:
                color_hist[color_str] = 1

    #return final histogram for image
    return color_hist

def texture_histogram(img_str, texture = texture):
    '''
    purpose: generates texture histogram for input image

    inputs
    img_str = image to be analyzed
    texture = number of bins for texture values

    outputs
    texture_hist = histogram representing overall texture of image
    '''
    #read image to analyze and convert to grayscale
    img = cv2.imread(img_str)
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

    #create storage for laplacian image conversion
    laplace = np.empty((60,89))

    #storage for histogram
    texture_hist = {}

    #calculates factor to reduce each pixel value by
    texture_fac = math.floor(2040/texture)

    #find size of image
    dim = gray.shape

    #iterate through rows of image
    for i in range(dim[0]):
        #iterate through columns of image
        for j in range(dim[1]):
            #find original value of each pixel
            pixel = gray[i][j]

            #initialize values for neighbors of pixel
            upper_left = 1
            upper_center = 1
```

```
            upper_right = 1
            bottom_left = 1
            bottom_center = 1
            bottom_right = 1
            center_left = 1
            center_right = 1

            #at top of image, make neighbors 0
            if i - 1 < 0:
                upper_left = 0
                upper_center = 0
                upper_right = 0
            else:
                #finds value of upper center neighbor
                upper_center = int(gray[i - 1][j])

            #at bottom of image, make neighbors 0
            if i + 1 > dim[0] - 1:
                bottom_left = 0
                bottom_center = 0
                bottom_right = 0
            else:
                #finds value of bottom center neighbor
                bottom_center = int(gray[i + 1][j])

            #at left of image, make neighbors 0
            if j - 1 < 0:
                upper_left = 0
                center_left = 0
                bottom_left = 0
            else:
                #finds value of center left neighbor
                center_left = int(gray[i][j - 1])

            #at right of image, make neighbors 0
            if j + 1 > dim[1] - 1:
                upper_right = 0
                center_right = 0
                bottom_right = 0
            else:
                #finds value of center right neighbor
                center_right = int(gray[i][j + 1])

            #fill in values for corner neighbors if not determined to not
exist
```

```
            if upper_left != 0:
                upper_left = int(gray[i - 1][j - 1])

            if upper_right != 0:
                upper_right = int(gray[i - 1][j + 1])

            if bottom_left != 0:
                bottom_left = int(gray[i + 1][j - 1])

            if bottom_right != 0:
                bottom_right = int(gray[i + 1][j + 1])

            #sum of neighbors to current pixel
            neighbors_sum = upper_center + upper_left + upper_right +
bottom_center + bottom_left + bottom_right + center_left + center_right

            #calculates laplacian value of pixel and adds to new laplace
img
            l_pix = (8*pixel) - (neighbors_sum)
            laplace[i][j] = abs(l_pix)

            #reduces value of pixel by number of bins
            texture_val_fac = math.floor(laplace[i][j]/texture_fac)

            #reduce if case where factor results in max value
            if texture_val_fac == texture:
                texture_val_fac = texture_val_fac - 1

            #if texture value already in image, increment, else add to
histogram
            if texture_val_fac in texture_hist:
                texture_hist[texture_val_fac] += 1
            else:
                texture_hist[texture_val_fac] = 1

    #return final image histogram
    return texture_hist

def shape_overlap(img_str1, img_str2):
    '''
    purpose: sums the overlapping pixels of two images to find
approximation of shape similarity

    inputs:
    img_str1 - first image to compare
```

```
    img_str2 - second image to compare

    outputs
    overlap - normalized value of summed matching pixels
    '''
    #convert images to grayscale, blur, then threshold to convert to
binary
    #blur is neccessary to smooth shape edges, and the chosen threshold
maximizes
    #results for finding which pixels are background vs. foreground
    img1 = cv2.imread(img_str1)
    gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    blur1 = cv2.blur(gray1, (30,30))
    ret, bw1 = cv2.threshold(blur1, 127, 255, cv2.THRESH_BINARY)
    #cv2.imshow('blur', bw1)

    img2 = cv2.imread(img_str2)
    gray2 = cv2.cvtColor(img2, cv2.COLOR_BGR2GRAY)
    blur2 = cv2.blur(gray2, (30,30))
    ret, bw2 = cv2.threshold(blur2, 127, 255, cv2.THRESH_BINARY)
    #cv2.imshow('blur2', bw2)
    #cv2.waitKey(0)
    #cv2.destroyAllWindows()

    #find size of images, assumes images are both same size
    dim = bw1.shape

    #initialize summation value
    summation = 0

    #iterate through image pixels and compare
    #add to sum if they are equal
    for i in range(dim[0]):
        for j in range(dim[1]):
            if bw1[i][j] != bw2[i][j]:
                summation += 1

    #normalize summation by image size
    overlap = summation / (60*89)

    #return overlap distance calculated between the two images
    return overlap

def symmetry(img_str1):
    '''
```

```
    purpose: finds symmetry of image by folding image vertically and
comparing pixel values

    inputs:
    img_str1: image to analyze

    outputs:
    symm: symmetry value normalized by image size
    '''
    #read image and convert to grayscale, blur and threshold to convert to
binary
    #blur is used to soften edges of image and make less sensitive to
slight changes
    #threshold value is chosen to maximize results
    img1 = cv2.imread(img_str1)
    gray1 = cv2.cvtColor(img1, cv2.COLOR_BGR2GRAY)
    blur1 = cv2.blur(gray1, (15,15))
    ret, bw1 = cv2.threshold(blur1, 140, 255, cv2.THRESH_BINARY)

    #initialize summation value
    summation = 0

    #iterate through columns comparing left and right sides of image
    for i in range(0,44):
        for k in range(0,60):
            #chose pixel in same row
            pixel1 = bw1[k][i]
            pixel2 = bw1[k][88 - i]

            #add to total sum if they match
            if pixel1 == pixel2:
                summation += 1

    #normalize summation by image size
    symm = summation / (60*44)

    #returns normalized symmetry value
    return symm



def color_distance(hist1, hist2, blue = blue, green = green, red = red):
    '''
    purpose: computes the distance between two image histograms by color
```

```
    inputs:
    hist1: first image histogram to compare
    hist2: second image histogram to compare
    blue: number of blue bins
    green: number of green bins
    red: nuber of red bins

    (assumes number of bgr bins are the same as those used to create the
histogram)

    outputs:
    distance: normalized distance between the two image histograms
    '''
    #initialize summation value
    summation = 0

    #iterate through all possible color bins
    for i in range(blue):
        for j in range(green):
            for k in range(red):
                #colors stored as strings for key in dict
                color_str = str(i) + "," + str(j) + "," + str(k)

                #check if color in first image, if not set value to zero
                if color_str in hist1:
                    count1 = hist1[color_str]
                else:
                    count1 = 0

                #check if color in second image, if not set value to zero
                if color_str in hist2:
                    count2 = hist2[color_str]
                else:
                    count2 = 0

                #find absolute difference between two color counts
                diff = abs(count1 - count2)

                #add current color value difference to total summation
                summation = summation + diff

    #normalize value by image size
    distance = summation / (2 * 60 * 89)

    #return final distance between two image histograms
```

```
    return distance

def texture_distance(hist1, hist2, texture = texture):
    '''
    purpose: find distance between two texture histograms

    inputs:
    hist1: first texture histogram to be analyzed
    hist2: second texture histogram to be analyzed
    texture: number of bins used to create histograms

    outputs:
    distance: normalized distance between the two image histograms
    '''
    #initialize summation value
    summation = 0

    #iterate through each texture bin
    for i in range(texture):

        #check if texture value is present in first image
        if i in hist1:
            count1 = hist1[i]
        else:
            count1 = 0

        #check if texture value is present in second image
        if i in hist2:
            count2 = hist2[i]
        else:
            count2 = 0

        #find absolute value of difference between images for this texture
bin
        diff = abs(count1 - count2)

        #add to total distance summation
        summation = summation + diff

    #normalize distance by image size
    distance = summation / (2 * 60 * 89)

    #return normalized distance between the two histograms
    return distance
```

```
def get_match_from_hist(color_or_text, texture = texture, blue = blue,
green = green, red = red, num_imgs = num_imgs):
    '''
    purpose: find top three matches for all 40 images provided using
either the generated color or texture histograms

    inputs
    color_or_text: used to determine if algorithm should compare by
texture or color
    texture: texture bins
    blue: blue bins
    green: green bins
    red = red bins
    num_imgs: number of images to compare against one another

    outputs
    top_three: dict with each query image as a key and a nested dict
containing the top three matches found
    '''
    #initialize storage
    all_hist = {}
    top_three = {}

    #iterate through each image
    for i in range(1,num_imgs + 1):
        #generate img string
        if i < 10:
            i_str = "0" + str(i)
        else:
            i_str = str(i)

        img_str = "i" + i_str + ".jpg"

        #generate list of histograms for each input image to compare
        if color_or_text == 'color':
            all_hist[i] = color_histogram(img_str, blue, green, red)
        elif color_or_text == 'text':
            all_hist[i] = texture_histogram(img_str, texture)

    #counter variable used to track query image
    i = 1
    #iterate through each image histogram
    for key1 in all_hist:
        hist1 = all_hist[key1]
```

```
        #initialize match list
        dist_list = {1:0,2:0,3:0}

        #set distance to greatest possible
        match1 = 1.1
        match2 = 1.1
        match3 = 1.1

        #iterate through all other images
        for key2 in all_hist:
            if key1 != key2:
                hist2 = all_hist[key2]
                #compute distance between histograms by color or texture
                if color_or_text == 'color':
                    dist = color_distance(hist1, hist2, blue, green, red)
                elif color_or_text == 'text':
                    dist = texture_distance(hist1, hist2, texture)

                #find top three matches iteratively
                if dist < match1:
                    match3 = match2
                    dist_list[3] = dist_list[2]
                    match2 = match1
                    dist_list[2] = dist_list[1]
                    match1 = dist
                    dist_list[1] = key2
                elif dist < match2:
                    match3 = match2
                    dist_list[3] = dist_list[2]
                    match2 = dist
                    dist_list[2] = key2
                elif dist < match3:
                    match3 = dist
                    dist_list[3] = key2
        #add matches found for image to running dict
        top_three[i] = dist_list
        i += 1
    #return dict of matches for each input image
    return top_three

def get_match_from_shape( ):
    '''
    purpose: find top three matches by shape overlap

    input:
```

```
    none

    output:
    top_three: dict with each input image as key with a nested dict
containing the top three matches for the query image
    '''
    #initialize
    top_three = {}

    #iterate through each input image, assumes 40
    for i in range(1,41):
        #initialize match to greater than highest possible
        match1 = 1.1
        match2 = 1.1
        match3 = 1.1
        dist_list = {1:0,2:0,3:0}
        #iterate through all other input images
        for j in range(1,41):
            if i != j:
                #generate image strings to read images from file
                if i < 10:
                    i_str = "0" + str(i)
                else:
                    i_str = str(i)

                str1 = "i" + i_str + ".jpg"

                if j < 10:
                    j_str = "0" + str(j)
                else:
                    j_str = str(j)

                str2 = "i" + j_str + ".jpg"

                #calculate distance between both images
                result = shape_overlap(str1, str2)

                #iteratively find top three matches for query image
                if result < match1:
                    match3 = match2
                    dist_list[3] = dist_list[2]
                    match2 = match1
                    dist_list[2] = dist_list[1]
                    match1 = result
                    #print("new match1:  " + str(j) + "at " + str(result))
```

```
                    dist_list[1] = j
                elif result < match2:
                    match3 = match2
                    dist_list[3] = dist_list[2]
                    match2 = result
                    #print("new match2:  " + str(j) + "at " + str(result))
                    dist_list[2] = j
                elif result < match3:
                    match3 = result
                    #print("new match3:   " + str(j) + "at " +
str(result))
                    dist_list[3] = j

        top_three[i] = dist_list

    #return dict of top three matches for each input image
    return top_three

def get_match_from_symm():
    '''
    purpose: find top three matching images by symmetry

    inputs:
    none

    outputs
    top_three: dict with each input image as key with a nested dict
containing the top three matches for the query image

    '''
    #find the symmetry value for each input image
    symm_list = {}
    for i in range(1,41):
        if i < 10:
            i_str = "0" + str(i)
        else:
            i_str = str(i)

        str1 = "i" + i_str + ".jpg"

        result = symmetry(str1)

        symm_list[i] = result
```

```
    #iterate through each to find top three matching images, see earlier
functions for more detail
    top_three = {}
    for i in range(1,41):
        match1 = 1.1
        match2 = 1.1
        match3 = 1.1
        dist_list = {1:0, 2:0, 3:0}
        for j in range(1,41):
            if i != j:
                dist = abs(symm_list[i] - symm_list[j])

                if dist < match1:
                    match3 = match2
                    dist_list[3] = dist_list[2]
                    match2 = match1
                    dist_list[2] = dist_list[1]
                    match1 = dist
                    dist_list[1] = j
                elif dist < match2:
                    match3 = match2
                    dist_list[3] = dist_list[2]
                    match2 = dist
                    dist_list[2] = j
                elif dist < match3:
                    match3 = dist
                    dist_list[3] = j
        top_three[i] = dist_list

    #return dict of top three matches for each input image
    return top_three

def gestalt(c_fac, t_fac, sh_fac, symm_fac):
    '''
    purpose: uses color, texture, shape, and symmetry to find the top
three matches between input images

    inputs
    c_fac: factor by which to multiply color distance
    t_fac: factor by which to multiply texture distance
    sh_fac: factor by which to multiply shape distance
    symm_fac: factor by which to multiply symmetry distance

    outputs:
```

```
    top_three: dict with each input image as key with a nested dict
containing the top three matches for the query image
    '''

    top_three = {}

    #iterate through each input image, assumes 40
    for i in range(1,41):
        match1 = 1.1
        match2 = 1.1
        match3 = 1.1

        #generate first image string to read image
        if i < 10:
            i_str = "0" + str(i)
        else:
            i_str = str(i)

        str1 = "i" + i_str + ".jpg"

        dist_list = {1:0, 2:0, 3:0}

        #iterate through all other input images to find matches
        for j in range(1,41):
            #generate second image string
            if j < 10:
                j_str = "0" + str(j)
            else:
                j_str = str(j)

            str2 = "i" + j_str + ".jpg"

            #only compare different images
            if i != j:

                #find color distance between both images
                c_hist_1 = color_histogram(str1)
                c_hist_2 = color_histogram(str2)
                c_dist = color_distance(c_hist_1, c_hist_2)

                #find texture distance between both images
                t_hist_1 = texture_histogram(str1)
                t_hist_2 = texture_histogram(str2)
                t_dist = texture_distance(t_hist_1, t_hist_2)
```

```
                #find shape distance between both images
                sh_dist = shape_overlap(str1, str2)

                #find symmetry distance between both images
                symm_dist = abs(symmetry(str1) - symmetry(str2))

                #use given factors to calculate final distance between
images
                dist = c_fac * c_dist + t_fac * t_dist + sh_fac * sh_dist
+ symm_fac * symm_dist

                #iteratively find top three matches
                if dist < match1:
                    match3 = match2
                    dist_list[3] = dist_list[2]
                    match2 = match1
                    dist_list[2] = dist_list[1]
                    match1 = dist
                    dist_list[1] = j
                elif dist < match2:
                    match3 = match2
                    dist_list[3] = dist_list[2]
                    match2 = dist
                    dist_list[2] = j
                elif dist < match3:
                    match3 = dist
                    dist_list[3] = j
        top_three[i] = dist_list

    #return dict of top three matches for each input image
    return top_three

def get_score(top_three):
    '''
    purpose: uses crowd sourced borda counts to find total score over all
input images and their calculated matches

    inputs
    top_three: dict containing top three matches for each query image

    outputs
    total: sum of borda counts for each target image vs. query image
    '''
    total = 0
    #iterate through each input image
```

```
    for key in top_three:
        #calculate total score across all three matches
        score = crowd_data[key - 1, top_three[key][1] - 1] +
crowd_data[key - 1, top_three[key][2] - 1] + crowd_data[key - 1,
top_three[key][3] - 1]
        #print(str(score))
        total = total + score
    #returns total score across all input images
    return total

def concat_vh(list_2d):
    '''
    purpose: vertically and horizontally concatenates images of the same
size

    source:
https://www.geeksforgeeks.org/concatenate-images-using-opencv-in-python/

    This is listed as opensource and uses openCV to perform its intended
function.
    '''

    #concatenates images horizontally first then vertically
    return cv2.vconcat([cv2.hconcat(list_h)
                        for list_h in list_2d])


def generate_visual(results, name):
    '''
    purpose: generates 40x4 images with each row representing a query
image and its top three matches with columns 1-3 being the number 1,2, and
3
    matches respectively. Also calculates scores for each target image,
row, and overall score for all images.

    inputs
    results: dict containing top three matches for each query image
    name: file name to save generated image to

    outputs
    none
    '''
    h_img_list_half1 = []

    bg = np.zeros([30,89,3],dtype=np.uint8)
```

```
    bg.fill(255)
    total_score = get_score(results)

    for query in results:
        row_image_list = []
        if query < 10:
            query_image_str = 'i0' + str(query) + '.jpg'
        else:
            query_image_str = 'i' + str(query) + '.jpg'

        row_score = int(crowd_data[query - 1, results[query][1] - 1] +
crowd_data[query - 1, results[query][2] - 1] + crowd_data[query - 1,
results[query][3] - 1])
        query_img = cv2.imread(query_image_str)
        query_img = cv2.copyMakeBorder(query_img, 10, 20, 0, 0,
cv2.BORDER_CONSTANT, value=[255, 255, 255])
        if query == 1:
            query_img = cv2.putText(query_img, 'tot_score=', (10,10),
font,
                    fontScale, color, thickness, cv2.LINE_AA)
        query_img = cv2.putText(query_img, 'q(' + str(query) + ')=' +
str(row_score), org, font,
                    fontScale, color, thickness, cv2.LINE_AA)
        row_image_list.append(query_img)

        for key in results[query]:
            if results[query][key] < 10:
                img_str = 'i0' + str(results[query][key]) + '.jpg'
            else:
                img_str = 'i' + str(results[query][key]) + '.jpg'

            score = int(crowd_data[query - 1, results[query][key] - 1])
            img = cv2.imread(img_str)
            img = cv2.copyMakeBorder(img, 10, 20, 0, 0,
cv2.BORDER_CONSTANT, value=[255, 255, 255])
            if query == 1 and key == 1:
                img = cv2.putText(img, str(total_score), (10,10), font,
                    fontScale, color, thickness, cv2.LINE_AA)
            img = cv2.putText(img, str(results[query][key]) + ', s=' +
str(score), org, font,
                    fontScale, color, thickness, cv2.LINE_AA)
            row_image_list.append(img)

        h_img_list_half1.append(row_image_list)
```

```
    img_tile1 = concat_vh(h_img_list_half1)

    cv2.imwrite(name, img_tile1)

def happiness(results):
    '''
    purpose: calculates the set intersection of a generated top three
matches versus the author's personal results

    inputs
    results: dict containing top three matches for each query image

    outputs
    set_int: value of number of images matching over all input images
    '''
    set_int = 0
    for i in range(1,41):
        img_tt = results[i]
        for j in range(1,4):
            if my_data[i - 1][j] in img_tt.values():
                set_int += 1

    print(str(set_int))
    return set_int
```