REPORT – PROJECT -2

Piyush Goel

Description:

I created a content based image retrieval program, which when given a target image, returns a set of top matches for that image from a given image database. These top matches can be calculated using a number of matching methods and distance metrics (numerous of which have been implemented). All this is accessible through a command line program and the way to use it explained in the **readme**.

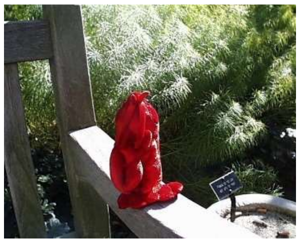
Basic overview of the class structure used to make adding new algorithms easier and the code cleaner and to avoid code duplication:

1. ImageFeaturizer: This is the class which handles all the code to calculate the features of a given image (which are later used to compare the images for the content). All the algorithms for featurizing the image are implemented as the subclasses of this class.
2. DistanceMetric: This is the class which deals with calculation of the distance between two images, given the features.
3. Matcher: This is the class which takes in the objects of all the other classes and makes everything work together to calculate the feature for all the images and then return the top matches for the target image.

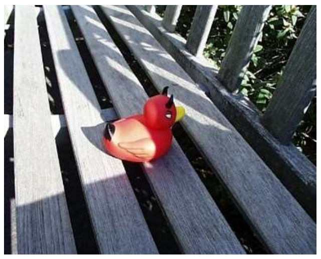
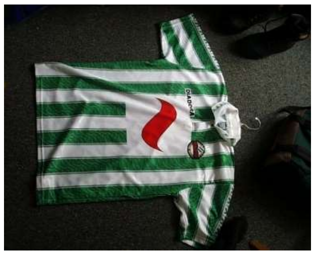
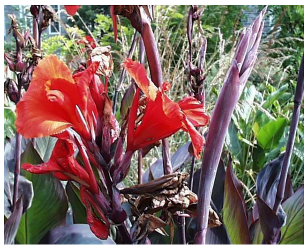
The Main.cpp file contains the code for the command line program. I’ve provided a lot of flexibility to the user. My class structure permits me to allow the user to use any combination of featurizing algorithms and distance metrics (normalized or not), the user can also customize some of the algorithms to some extent. Although not all of the combinations would give good results, but the user still has the flexibility.

Task 1 (See BaselineFeaturizer in the code)

Use the following arguments to execute obtain the result (b -> baseline matching, l2 -> L-2 norm or sum of squared distance, 3 -> top 3 matches): **pic.1016.jpg <abs path to the image database> b l2 3**

Input Image: pic.1016 

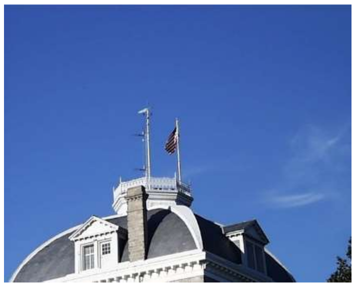
Top 3 matches (in order of best to worst): pic.0986, pic.0641, pic.0233

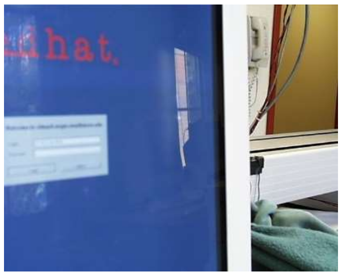
Observation: As expected every image which was the best match has red color in the middle but the matching algorithm didn't care about the actual objects.

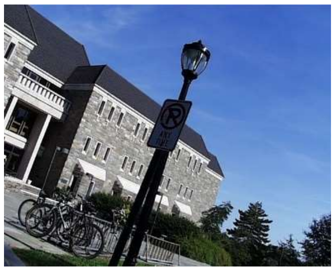
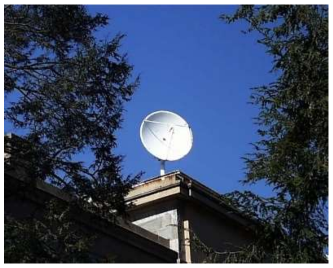
Task 2 (See RGHistogramFeaturizer in the code)

Use the following arguments to execute obtain the result (h-32 -> histogram matching with 32 buckets, i -> histogram intersection, 3 -> top 3 matches): **pic.0164.jpg <abs path to the image database> h-32 i 3**

Input Image: pic.0164 

Top 3 matches (in order of best to worst): pic.0080, pic.0426, pic.0110



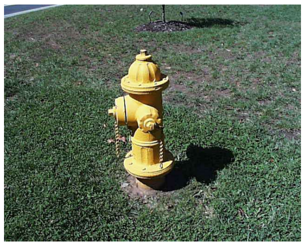
Observation: As expected of a color histogram, the best matches contain the most similar colors. And since it's a RG-chromaticity histogram, it didn't care for the exact shades of the colors, but just the color themselves.

Note: Instead of using histogram intersection directly, I used negative of histogram intersection as it is a similarity metric (the higher the similarity, the better the match) and I wrote my code to work with distance metrics (the higher the distance, the worse the match). So, just using the negative instead of using the intersection directly, did the trick, without having to make any unreasonable changes to the code or making it more complicated.

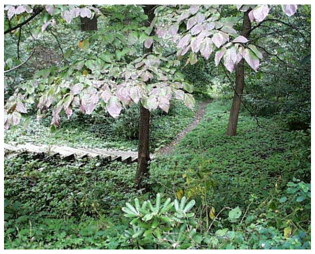
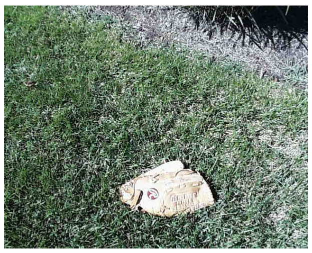
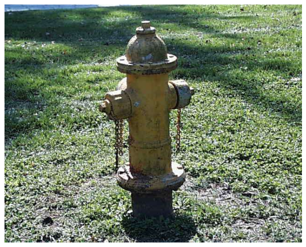
Task 3 (See CenterFullMultiRGHistogramFeaturizer in the code)

For this task I used a combination of two rg-chromaticity histograms one over the complete image and one over the center 100x100 patch of the image, and both with 32 buckets.

Use the following arguments to execute obtain the result (mh-cf-32 -> multi-histogram matching with one with the center part of the image and one with the full image both with 32 buckets, i-n -> histogram intersection (the -n stands for normalized), 3 -> top 3 matches): **pic.0135.jpg <abs path to the image database> mh-cf-32 i-n 3**

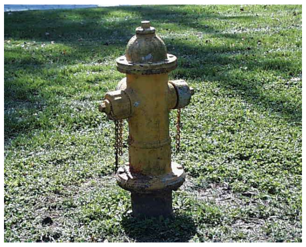
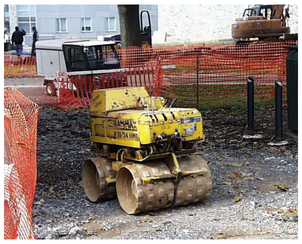
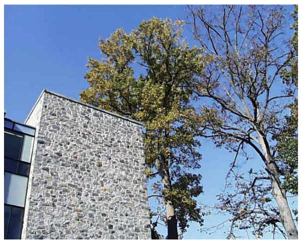
Input image: pic.0135 

Top 3 matches (in order of best to worst) – By giving equal weightage to both the histograms (i.e. 1.0 and 1.0): pic.0719, pic.0123, pic.0897

(Below is a little extension/experimentation I did by messing around with the weights)

Top 3 matches (in order of best to worst) – By giving more weightage to the central histogram and less weightage to the full image histogram (5.0 and 1.0 respectively): pic.0897, pic.0731, pic.1021

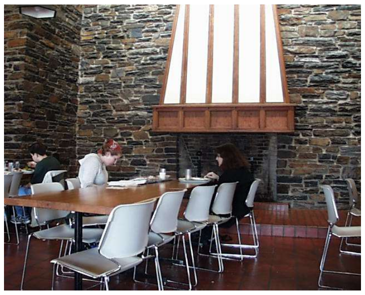
Note: The weightages can be changed at line 184 of the file ImageFeaturizer.cpp – I thought about taking this as input from the user as well, but I realized that that might make the program a little too complicated to use and hence decided against it.

Observation: I expected the results to better by increasing the weightage of the histogram for only the central part of the image and the best match was significantly better like expected, but then the other results weren’t so great. This might’ve been because of some over emphasis on the central part. So I tried using the weights 2.0 and 1.0 and I found the results to be better (but I couldn’t experiment any further due to the lack of time).

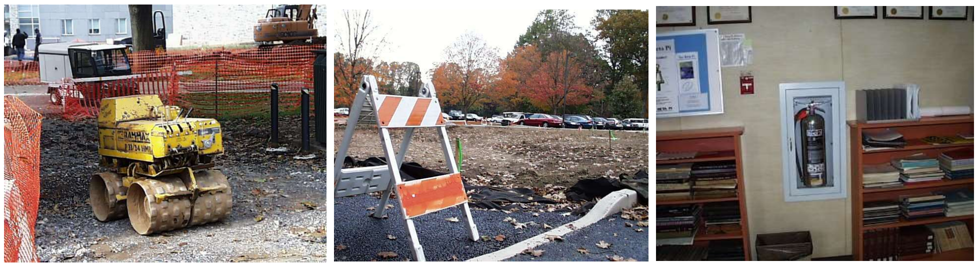
Task 4 (See RGHistogramAndSobelOrientationTextureFeaturizer in the code)

For this task I first created the sobel orientation filter for which I used the code of my previous project. Then after I took the output of the sobel orientation filter (whose output I scaled to be between 0 and 1 for ease) and fed it to a Averaging histogram (which first averages all the color values at each pixel and then generates a 1-D histogram by bucketing the values) with 32 buckets. And then I combined this with my histogram matcher from task 2. I only present the results with equally weighted algorithms as that was asked in the task description but the weights can be varied by modifying line 215 of ImageFeaturizer.cpp

Use the following arguments to execute obtain the result (h-s-32 -> histogram matching combined with sobel texture matching both with 32 buckets, i-n -> histogram intersection (the -n stands for normalized), 3 -> top 3 matches): **pic.0535.jpg <abs path of image db> h-s-32 i-n 3**

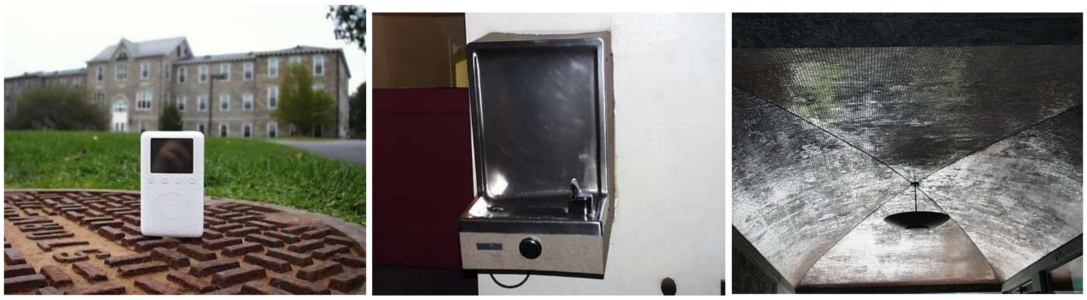
Input Image: pic.0535 

Top 3 matches (in order of best to worst): pic.0731, pic.1106, pic.0795



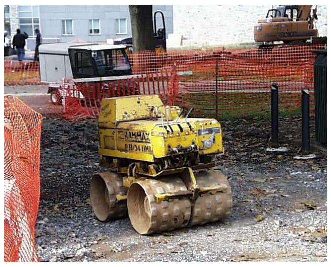
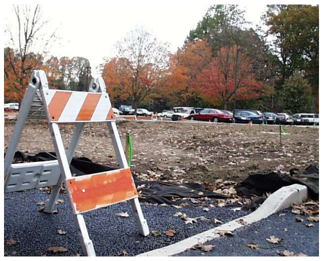
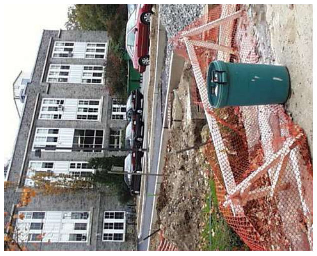
Observation: Although these best matches look nothing like the input image at the first look but if we look carefully there are a lot of common colors in the images, and the texture is also very similar. The first pic is a good match because of the matching color as well as the texture (lots of texture with horizontal lines in the wall in the target image and the net in the best match), the second one mainly because of matching color (lots of brown, white and some shades of red) and the third one has a little of both (similar looking colors – brown and white shades, with some matching texture – horizontal lines because of the books). The matching color observation is also reinforced by the results of the task 2 algorithm on this target image.

Best matches according to task 1 matching algorithm – pic.1075, pic.0585, pic.1026



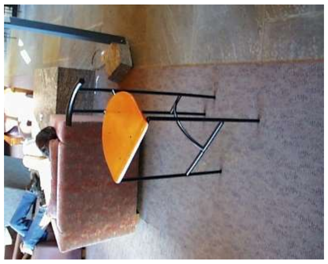
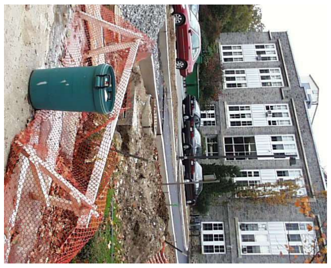
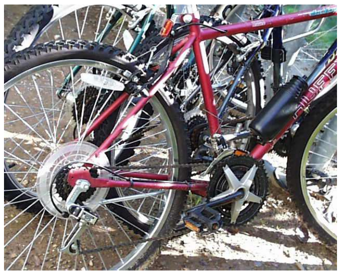
Observation: These matches are completely unrelated to the target image, but as expected the center point of these images contain almost the same color as the target image.

Best matches according to task 2 matching algorithm – pic.0731, pic.1106, pic.1104

Observation: The first two matches here are the same as in the results of task 4 algorithm, and these reinforce our belief that the first two images have strongly matching color (chromaticity) to the target image.

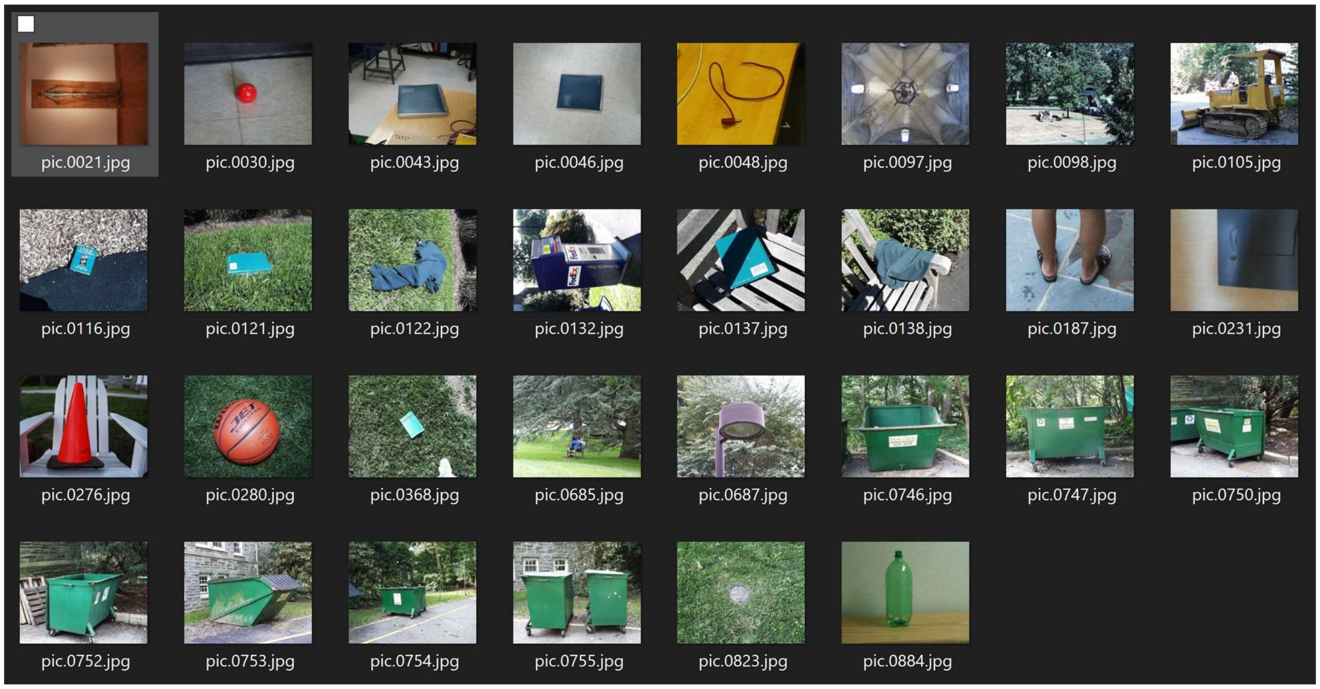
Best matches according to task 3 matching algorithm (with weights {1.0, 1.0}) – pic.0341, pic.1104, pic.0150

Observation: Again, not at all similar results to the target image, but pretty much what was expected as with similar color in the whole image but more strongly similar in the center part.

Task 5 (See RGFullAndCenterSobelTopAndBottomFullFeaturizer in the code)

For this task I chose the following set of 30 images



And my focus was to create an algorithm which is able to successful match images of the green trashcans when one of them is given as input. So the images have 4 main parts -> 1) the center part of the image has a big area with just green color. 2) the complete image has also matching colors (lots of green and some grey). 3) The top half of the image has a certain texture because of the leaves of the tree. 4) The bottom half has a certain different texture due to the road.

Therefore I created an algorithm as a combination of 4 different ones -> 1) A 2-D rg-chromaticity histogram of the central 100x100 pixels with 32x32 buckets 2) another A 2-D rg-chromaticity histogram of the complete image with 32x32 buckets as well. 3) A 1-D histogram of sobel orientation texture for the top half of the image with 32 buckets. 4) Another sobel texture histogram on the bottom half.

Use the following arguments to execute obtain the result (mh-cf-ms-tb-32 -> the matching algorithm developed for task 5, i-n -> histogram intersection (the -n stands for normalized), 30 -> top 30 matches): **i <abs path to the image db for task-5> mh-cf-ms-tb-32 i-n 30**

Now the program would ask you to input the names of the images you want to give as target (after it has finished processing for all the images) and then return a list of the best matches. And after you're done just type 'q' and hit enter to exit the program. (This bit is the part of the interactive part of the program, explained more in the extension)

My expectation was that when given an image from the range 0746 to 0755 (i.e. images of green trash cans) as the target, the others within the same range should show up very high in the best match results. But I also expect the images with green objects in the middle and the grass/trees around it to also show up relatively high, as they too are actually very similar images.

Below I've just written the images in the order of best to worst (and not attached all the images as there would be way too much) for the images of the green trash cans and two images of other green objects in similar environments. I've bolded the images which also contain green trash cans to make it easy to see where they lie in the matches.

For pic.0746:

Top 30 matches (in order of best to worst): pic.**0754**, pic.**0750**, pic.**0755**, pic.**0753**, pic.**0752**, pic.**0747**, pic.0368, pic.0685, pic.0105, pic.0098, pic.0122, pic.0823, pic.0687, pic.0132, pic.0280, pic.0121, pic.0097, pic.0116, pic.0043, pic.0138, pic.0030, pic.0231, pic.0276, pic.0137, pic.0187, pic.0046, pic.0884, pic.0021, pic.0048

For pic.0747:

Top 30 matches (in order of best to worst): pic.0685, pic.0132, pic.0687, pic.0823, pic.0097, pic.**0755**, pic.**0754**, pic.**0746**, pic.**0753**, pic.**0750**, pic.0098, pic.0105, pic.**0752**, pic.0122, pic.0368, pic.0280, pic.0138, pic.0121, pic.0116, pic.0030, pic.0043, pic.0231, pic.0276, pic.0137, pic.0187, pic.0046, pic.0884, pic.0048, pic.0021

For pic.0750:

Top 30 matches (in order of best to worst): pic.**0753**, pic.**0746**, pic.**0754**, pic.**0752**, pic.**0755**, pic.**0747**, pic.0098, pic.0105, pic.0687, pic.0132, pic.0116, pic.0685, pic.0043, pic.0138, pic.0122, pic.0097, pic.0137, pic.0030, pic.0368, pic.0231, pic.0046, pic.0280, pic.0121, pic.0823, pic.0276, pic.0187, pic.0884, pic.0048, pic.0021

For pic.0752:

Top 30 matches (in order of best to worst): pic.**0753**, pic.**0746**, pic.**0750**, pic.0098, pic.**0747**, pic.**0754**, pic.**0755**, pic.0368, pic.0105, pic.0685, pic.0122, pic.0687, pic.0132, pic.0116, pic.0138, pic.0043, pic.0097, pic.0137, pic.0121, pic.0030, pic.0823, pic.0231, pic.0046, pic.0280, pic.0276, pic.0187, pic.0884, pic.0048, pic.0021

For pic.0753:

Top 30 matches (in order of best to worst): pic.**0750**, pic.**0754**, pic.**0755**, pic.**0752**, pic.**0746**, pic.**0747**, pic.0098, pic.0685, pic.0122, pic.0368, pic.0687, pic.0105, pic.0132, pic.0121, pic.0097, pic.0138, pic.0823, pic.0030, pic.0116, pic.0137, pic.0276, pic.0043, pic.0884, pic.0231, pic.0046, pic.0280, pic.0187, pic.0048, pic.0021

For pic.0754:

Top 30 matches (in order of best to worst): pic.**0746**, pic.**0753**, pic.**0755**, pic.0122, pic.**0750**, pic.0685, pic.**0747**, pic.0368, pic.0823, pic.0098, pic.0121, pic.**0752**, pic.0687, pic.0105, pic.0132, pic.0116, pic.0138, pic.0097, pic.0884, pic.0043, pic.0030, pic.0280, pic.0276, pic.0231, pic.0137, pic.0046, pic.0187, pic.0048, pic.0021

For pic.0755:

Top 30 matches (in order of best to worst): pic.0685, pic.0687, pic.**0754**, pic.**0753**, pic.**0747**, pic.**0746**, pic.0122, pic.0098, pic.0105, pic.0823, pic.0097, pic.**0750**, pic.0132, pic.0368, pic.**0752**, pic.0116, pic.0043, pic.0121, pic.0884, pic.0030, pic.0138, pic.0231, pic.0280, pic.0137, pic.0046, pic.0276, pic.0187, pic.0048, pic.0021

Observation: As expected this algorithm is performing well as for almost all the images (except pic.0747, which seems reasonable as it is the pretty much the last of the highlighted images int the best matches for the other images). After a lot of experimentation and iterations I chose the relative weights of the algorithms as 2.0, 5.0, 1.0, 1.0 respectively (in the order full rg-hist, center rg-hist, top half sobel texture, bottom half sobel texture), as these were giving me the best possible results.

As I also expected, images like 0021, 0048, 0187, 0046, etc (which are very different from the images of the green trash can) appear at the end of the list of the best matches.

A little extra experimentation with some other images:

For pic.0121 (pic of a green notebook on grass):

Top 30 matches (in order of best to worst): pic.0116, pic.0368, pic.0122, pic.0754, pic.0823, pic.0137, pic.0685, pic.0746, pic.0755, pic.0753, pic.0747, pic.0138, pic.0098, pic.0752, pic.0687, pic.0750, pic.0132, pic.0105, pic.0280, pic.0276, pic.0884, pic.0097, pic.0043, pic.0046, pic.0187, pic.0231, pic.0030, pic.0048, pic.0021

For pic.0122 (pic of a green cloth on the grass):

Top 30 matches (in order of best to worst): pic.0685, pic.0754, pic.0368, pic.0755, pic.0046, pic.0121, pic.0823, pic.0687, pic.0116, pic.0746, pic.0138, pic.0098, pic.0753, pic.0747, pic.0884, pic.0043, pic.0752, pic.0105, pic.0750, pic.0132, pic.0231, pic.0280, pic.0137, pic.0276, pic.0097, pic.0030, pic.0187, pic.0048, pic.0021

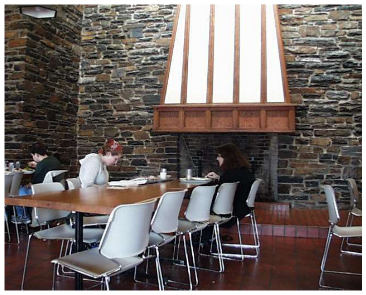
Extensions

1. Co-occurrence matrix and all its features (See CoOccurrenceMatrix in the code)

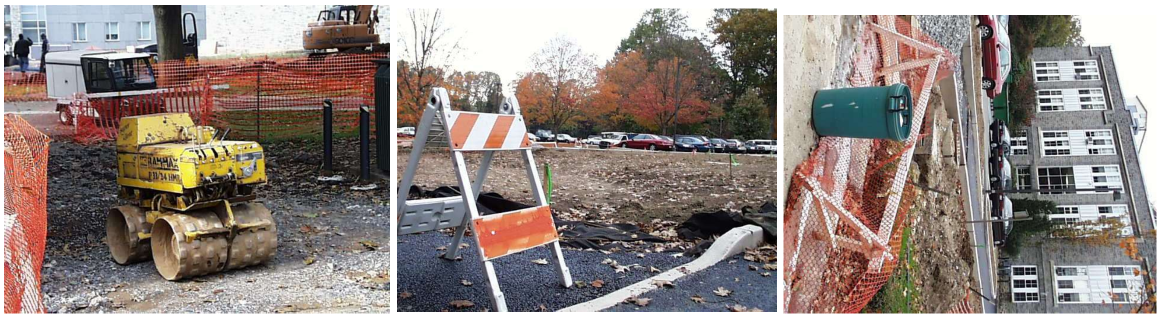
- I created the code for co-occurrence matrix to calculate different features to be able to have another measure of the texture of the image.
- The features used for this are – Energy, Entropy, Contrast, Homogeneity, Maximum probability. The formulaes for these are the same as mentioned in the lecture notes.
- I also had to normalize these metrics separately to bring their vales within the range of 0 to 1, to be able to use them as features and so that they are on the same scale.
- The user also has the flexibility to specify the axis on which they want the co-occurrence to be seen and also the distance they want between the pixels.

2. A combination of RG-histogram over the whole image and the co-occurrence matrix features over the whole image. (See RGCoOccFullFeaturizer in the code)

- Using the co-occurrence matrix features and the rg-chromaticity histogram, I implemented this matching algorithm by combining these two over the whole images.
- The relative weights of the 2-d histogram and the co-occurrence matrix features can be varied easily by modifying line 371 in ImageFeaturizer.cpp
- I wanted to compare the results of this with task-4, so here are the results.
- Use the following arguments to execute obtain the result (h-s-32 -> histogram matching combined with sobel texture matching both with 32 buckets, i-n -> histogram intersection (the -n stands for normalized), 3 -> top 3 matches): **pic.0535.jpg <abs path of image db> h-c i-n 3**
  And when the program asks for it, put in the desired (but valid) values of the axis, distance and the number of buckets.



- **Input Image:** pic.0535
- **Top 3 matches (in order of best to worst, with axis = 0, distance = 2, buckets = 32, and with equal weightage to everything):** pic.0731, pic.1106, pic.1104

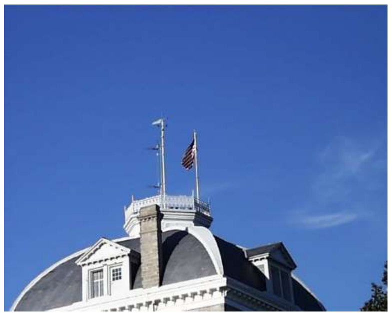


Observation: As we can see, the results here are pretty similar to the results of the other algorithms, so we can't say much about this algorithm in particular from just these results, if it's better than the others or not.

3. Pure histogram (See HistogramFeaturizer in the code)

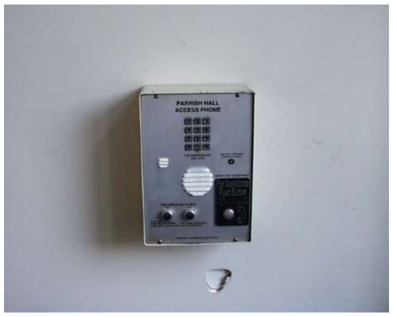
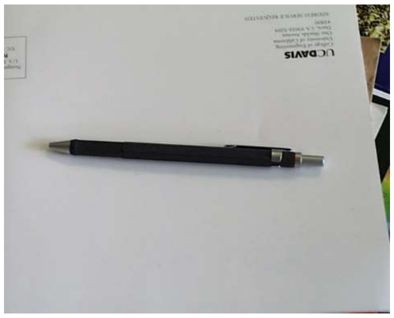
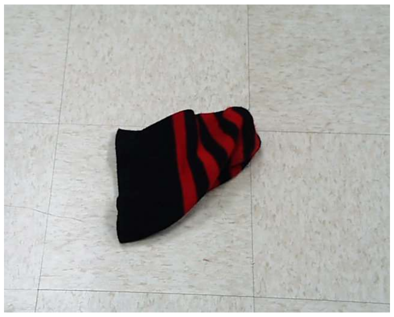
- I created a matching algorithm for histogram of the actual rgb pixel values, as I was curious to see how it would perform on task 2. Due to the insane amount of time it was taking to process I couldn't test the 3-D histogram, but I did test 1-D and 2-D ones and will present the result for some of them below.

- It takes as input a mask of 3 integers and they can either be 0 or 1. Each value in the mask corresponds to one color out of B, G, and R (in that respective order). The meaning of this mask is that the color corresponding to the integers which are set to 1 will be considered while creating a histogram. For example the mask {1,0,1} would result in a 2-D histogram with the B values on one axis and R values on the other.
- The results for the 1-D histogram with blur color is shown below.



Target Image: pic.0164.jpg

Top 3 matches (in order of best to worst): pic.0910, pic.0033, pic.0396

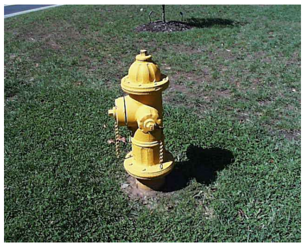
Observations: As expected these images are nothing like the target image, despite them having a lot of same "blue" color values in the pixels.

4. Average histogram (but this was made in the first place as it was required for some other task) (See AverageHistogram in the code)

- This one just averages the color values for each pixel individually and then creates a 1-D histogram out of it.
- Task-4 required some sort of functionality like this so instead of writing a little code to do that, I just created a whole new matching algorithm out of it. Although I expected this not to perform well on any of the tasks so I did not test it much and am not presenting the results.

5. Multi-histogram with a histogram for the top half and one for the bottom half. (See TopBottomMultiRGHistogramFeaturizer in the code)

- I created this multi-histogram to compare with the multi-histogram I created for task 3.
- The command line arguments for this experiment as follows: **pic.0135.jpg <abs path to image db> mh-tb-32 i-n 3**



Input Image: pic.0135

Top 3 matches (in order of best to worst) – By giving equal weightage to both the histograms (i.e. 1.0 and 1.0): pic.0719, pic.0680, pic.0918

Observations: As expected this algorithm performs a little worse than the original task-3 algorithm as it doesn't give any more importance to the central pixels than it gives to the rest of the pixels, and ends up finding the images of grass and leaves as the best matches.

6. Many different distance metrics – L-1 norm, L-N norm, hamming distance. (See the Metrics.cpp file for the implementation)

- Implemented several other distance metrics to test and see if they perform better in certain conditions, but on a lot of experimentation (for each of the tasks) I found that the best performing one in most of the cases was histogram intersection. That was also expected, but this experimentation helped verify my though process.
- The L-N norm metric is generic and the user can give in any value of n greater than 0.

7. Interactive command line mode where the user can give as input multiple images. And more little features in the command line program. (See Main.cpp file for the code of this program)

- If the user instead of the name of the pic, enters the character 'i' then the interactive mode of the program would be launched where the user can type in the image names (along with the extension) one by one and the program would display the best matches corresponding to those images. This will also be super fast as all the database processing would be done before taking the first input and the features for all the images would be stored in a temporary directory which would be later deleted before exiting the program. And the program would use this temporary database to calculate the best matches for all the input images. And to quit the program just type 'q' and hit enter.
- If the user wishes to store the feature data base and not have it deleted as soon as the program quits then the user can give the path to a directory where the features would be saved. Then this directory could be given as the last argument again and the program would automatically recognize that the directory contains the features and not try to calculate them again. **Note:** The program would only check if the directory is empty or not, it won't check if the directory contains valid features corresponding to the chosen matching algorithm.
- The user can choose any combination of distance metrics and matching algorithms. The user can also choose to normalize the histograms by appending "-n" to the end of the distance metric's argument.
- Most of the matching algorithms give a default implementation as well as a flexible implementation (i.e. where the user can choose some of the specifics of the algorithm by entering the detail when the command line program asks for it). For example if the chosen algorithm is 'h' (stands for 2-d rg-chromaticity histogram matching algorithm) then the program would ask the user for the number of buckets to use and the user can give in any number greater than 0.
- The same flexibility is also available for the L-N norm distance metric, where the user can choose the 'n' value.

Reflections

1. Implementing the histograms and matching algorithms from scratch, I learned a great deal about them and the small things at which one can get stuck while writing the code.
2. I had learned a lot about basic C++ syntax and memory management in the previous project, but in this one I could further that learning.

3. Though the main this I learned in this project (apart from the Comp Vision stuff) was about object oriented programming in C++ because of the class structure I decided to use, I could bring over some of my knowledge of OOP from Java to C++.
4. Using classes the way I did it took me a really long time to come up with the design and implement the first algorithm, but once that was ready, implementing the rest was relatively easy. This also made debugging easier as I only had to fix bugs once.
5. I also learned that we may need to make some compromises in our design. For example, while implementing the co-occurrence matrix algorithm, I needed a way to do some post processing to all the features. So the way I decided to do so was by introducing a post-processing method to its superclass which is always called but for the other classes it's an empty function. I've also discussed this approach in my code in the comments.

References

1. Lecture notes provided by the professor, and the class lectures.
2. StackOverflow
3. C++ documentation (cppreference)
4. Cleared a few doubts from the professor over email.
5. OpenCV documentation
6. Google searches
7. I discussed with some of my classmates about how they presented the results of task 5 in the report.