Jennifer Friedmann
December 20, 2012
Computational Vision & Biological Perception
Final Project

Introduction

Imagine you're an intern at the Metropolitan Museum of Art. You show up to work one day and receive a daunting task: cataloguing 10,000 lesser-known artifacts from the archives into the museum's database. You're responsible for tagging the image of each item based on type, relevant era, content, artist, and any other general cataloging data that might be useful in a search function later. You imagine the hours of research and processing this will require from you, both to sort through the images and apply general category tags, then to return to each subsection and develop more detailed ones.

Tagging every individual image sounds like a huge pain, so you immediately start to think of ways in which you could make this task easier. You consider how Google manages its huge base of images – but no luck. Google has relied upon filenames or context to pull relevant tags for its images, although it did have the effective idea of crowdsourcing human tagging by making it into a game, the Google Image Labeler, from 2007-2011. But you want something that doesn't require human input, something that will cover at least basic or common tags so that further classification is optional or at least easier. You look to computer image processing in hopes of finding an automatic image classifier.

Problem

As a first step towards developing an automated image classifier, I chose to develop two full exemplary feature processors. These would demonstrate a set of

tools or patterns that could be easily expanded into other feature detectors. I chose to create a landscape detector as an example of a general feature categorizer and a Jackson Pollock detector as an example of a specific feature categorizer.

I made some initial design choices that limited the scope of my project. My main goal was to develop a classifier that did *not* rely on machine learning techniques. Most algorithms of this sort use neural nets, clustering, or reinforcement learning to take in a huge set of image data, then train based on which ones do or do not fit a requirement, while incorporating feedback (on correct or incorrect classifications) along the way. I chose not to approach with this method because machine learning is a useful tool, but not very interesting from a computational vision perspective. I wanted to develop a classifier that sought out actual, not just statistical, features in an image based on logical definitions of those features.

I also chose not to use any built-in vision processing functions in MATLAB. For example, I considered making a portrait-detector, but this would have relied on the use of face recognition software (found in the Computer Vision toolbox), which in itself is an impressive thing to code, but not necessarily to use.

Landscape Detector

Before coding a function that would detect whether or not an image was an outdoor landscape, I had to decide on the definition of a landscape. Based on my experience in art history and photography, I chose the following constraints:

- A landscape, at its most basic, is composed of a sky, a horizon (not necessarily straight, and possibly occluded), and a foreground.

- The sky will always be brighter and bluer than the foreground because of the scattering of light.
- The foreground will always be more textured than the sky, which demonstrates a smoother texture even with the presence of clouds because the edges are not as sharp. Objects in the foreground (whether trees, rocks, houses, people, etc.) have much more potential for diverse textures.

I chose to make my algorithm's priority the presence of a definite horizon line. Other images might by chance display greater texture or less luminance on the bottom half, but the presence of one horizon line is a clear giveaway. To make a rigorous algorithm, I chose to define the horizon line as the y-value at which the blue sky covered less than 70% of the frame.

After first isolating the blue color channel in the RGB image and binarizing it, I labeled connected blue components of the image using the region-growing function bwlabel, which lumps all connected pixels (1s) at all 8 points of connection (including diagonals) into salient objects. After eliminating noise, I used this blue region to define the horizon, by finding all y-values for which blue covered 70% or more of the image, then taking the last y-value of that group. Here I set an arbitrary threshold test– which are necessary to the completion of this categorizing task[1]. I ruled out any images that indicated a blue-designated horizon in either the top sixth or the bottom fifth of the image. While high horizons are common in East Asian

[1] The incorporation of thresholds reflects a design decision on my part to create *rules* for various features in the image. While all thresholds are in some sense arbitrary and can be adjusted at great length to rule out false positives or include true negatives, the

paintings, and low horizons are often found in seascapes, all horizons tend towards the middle, not the extremes, of the image.

Results of region-growing in the blue color channel with low threshold and noise elimination. The blue incorporates even clouds.

Results of the horizon-line test (faint orange line). The horizon falls at the point where only 70% of the horizontal line is blue. (Constable)

I ran two secondary tests on the image once it had passed the horizon test in order to satisfy my other landscape constraints. I first dividing the image into two parts at the horizon, then ran an entropy test to measure texture variance and a luminance test to measure relative brightness. The luminance test required transforming the image into an NTSC (YIB) color scheme and isolating the Y ('luminance') color-channel (although a grayscale could have worked equally well). The image had to pass both tests in order to be declared a landscape.

I ran 25 images against my detector. The function worked well for all images with a blue sky, and even managed to rule out tricky instances of lakes and blue walls (see images below). However, it failed to classify images for which the sky exhibited other colors, such as with pictures of the sunset. Future tests might demonstrate failures on images with abstract/textured skies or a uniform distribution of blue.

Passes tests with reflected sky, as long as there is a clear horizon. (author photograph)

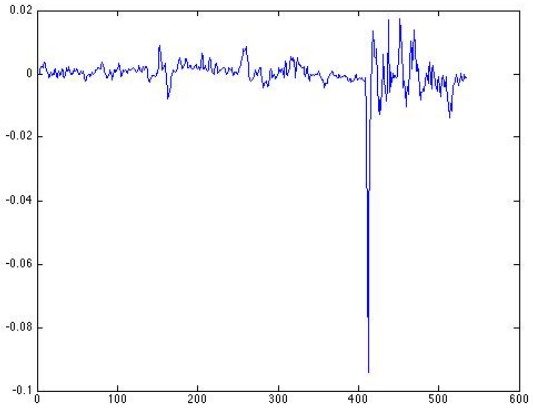
Passes tests with unusual color or occluded horizon. (J.M.W. Turner)

Passes tests with uneven horizons. (Constable-esque photograph, Wikimedia)

Passes tests with distractor blue backgrounds. (author photograph, background: Andy Warhol)

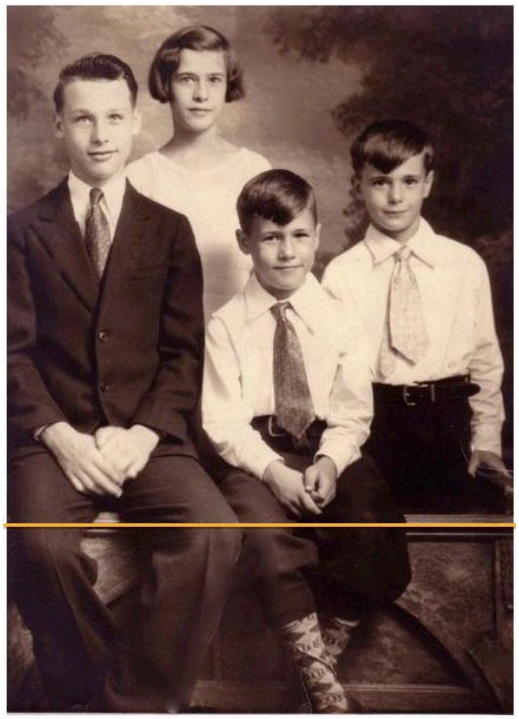
In order to create a more rigorous test for non-blue backgrounds, I created a luminance detector. As stated above, the sky will always be more illuminant than the foreground during the day – a common problem in taking pictures of landscapes. I followed a similar logic to my first function in creating this detector. To find the horizon, I isolated the Y channel of the converted NTSC (YIB) image, then took the mean along every row. When passing the resulting luminance column through a gradient, it was clear where the image shifted from an area of high luminance to low luminance:

This image of a sunset, which did not pass the first test, had a clear horizon with the luminance test.



The horizon is located at the point where the gradient shows a rapid decrease (shift from high to low luminance) – e.g. the spike above

The luminance test was actually much more accurate in determining the horizon of the image than was the blue horizon test, but I ran into the issue that *any* image with a change in luminance could demonstrate a horizon. I decided on a few features: my luminance test would only run on images that did NOT pass the first landscape test, and thus I could make the luminance test extremely strict. I implemented a secondary test with two determinants of texture – entropy (as in the first test) and edge complexity. The edge complexity test ran a Canny edge detector with a high threshold of 0.3, then analyzed the total pixels in the segmented edge image, which effectively cut out the false positives of family portraits and other uniformly-textured images.



This image is brighter on the top and demonstrates a horizon, even if it is not a landscape.

Pollock Detector

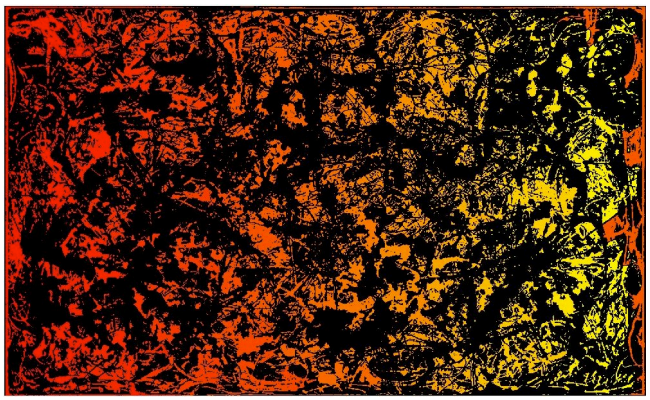
Having implemented a general feature detector that identified all landscapes for which I ran tests (40 in total), I chose to move to a more specific feature detector in order to demonstrate the other side of the potential for an automatic classifier. I chose to create a Jackson Pollock detector because the artist has a particular style (paint splattering) within a genre (abstract expressionism) that is easily recognizable. However, any artist with a distinctive style – whether that lay in the width of their lines, their color palette, the size of their brushstrokes, the medium they use, or some combination of features – could be written into a rule-based detector.

In determining which features to seek, I considered a number of Pollock paintings. Each showed particular characteristics, but most noticeably across all were a high level of complexity and a huge number of isolated color regions with no gradient between them. I chose to make this second feature, which is akin to a splatter paint indicator, my main test.

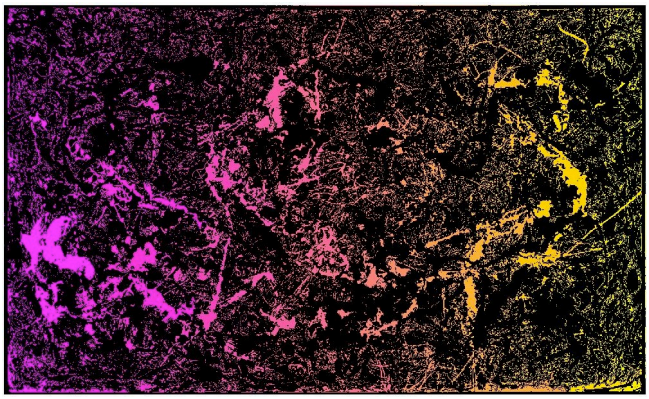
To complete this task, I wanted to isolate a small number of the main colors in the image. Researching this portion of the test took a while and a lot of trial and error – particularly since MATLAB makes it difficult to manipulate histograms or to isolate the dominant colors in an image. I developed a way to complete this task in Mathematica, by collecting all the colors in an image (formatted as (R, G, B) values) into an array and then using a Euclidean distance comparison to 'gather' them into similar groups. However, I struggled to implement this in MATLAB, so I instead chose to automatically reduce the number of colors in the image by indexing the image with its own colormap and setting the top boundary to 4 colors with

'nodither.' Each pixel in the image was then labeled 0 through 3. I created binary images for each color, then used region-growing to isolate the objects in the image. I found that I only needed the first three colors (the fourth tended to be too noisy). The separation of the Pollock into color regions yielded these images:

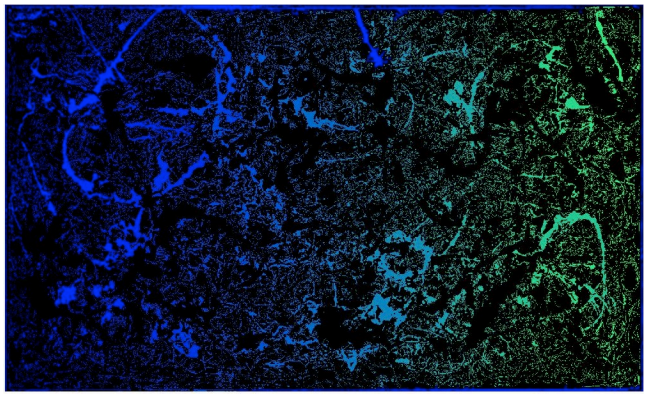
Original image indexed to four colors



Labeled map of objects with tag 0



Labeled map of objects with tag 1



Labeled map of objects with tag 2

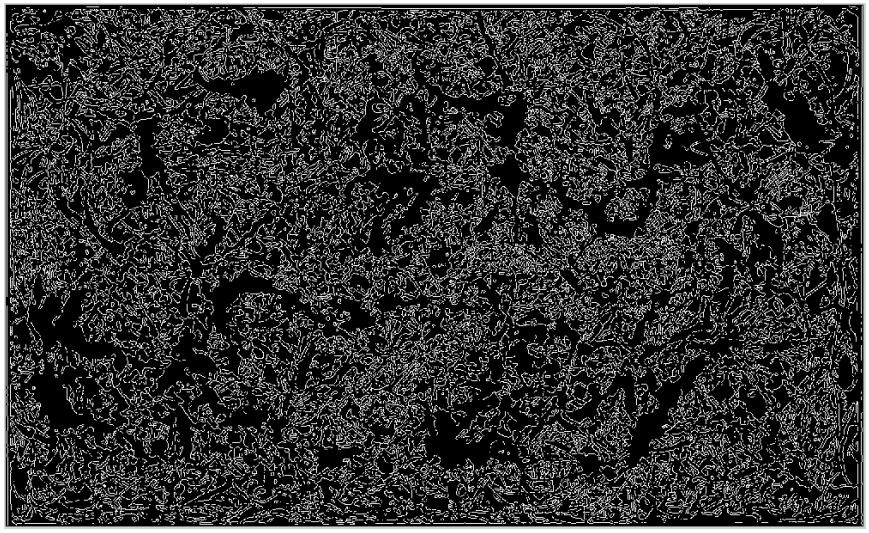
Original Pollock:

Convergence, 1952

My first test after segmenting these images was to detect the number of objects in each region. My logic was that, with such a reduction of color space, any paintings that do not exhibit widely scattered objects will generalize over much vaster regions, yielding a smaller object count overall. After running some tests on my landscape images to confirm my suspicions (their exhibited object count was very low – max 200), I set my thresholds at 1000 for each grouping of objects.

If the painting passed this first test of object prevalence, I submitted it to a related but distinct test of edge complexity. Similar to my edge detector in the landscape image, I set the Canny edge function to have a high threshold of 0.3, which weeded out most small edges in non-complex images:

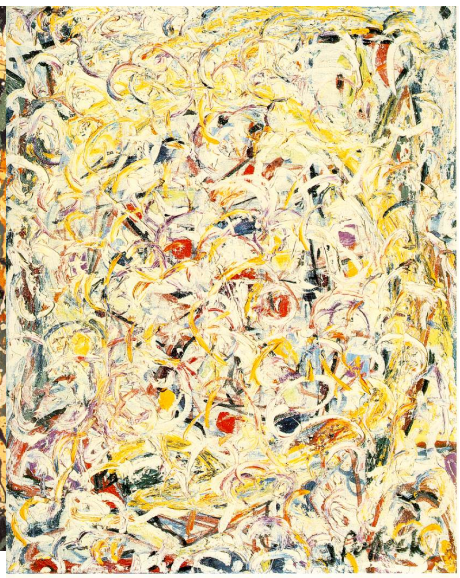
Warhol image – few edges at Canny threshold = 0.3



Pollock 'Convergence' at Canny threshold = 0.3 – still a high level of edge complexity.

I tested my function on seven Pollocks (including some that didn't classify as splatter paint but still showed an extreme prevalence of color regions) and ten non-Pollocks (including Van Gogh's Starry Night) for excellent results.

Typical Pollock: Pass



Non-Typical Pollock: Pass

Non-Typical Pollock: Pass

Van Gogh Starry Night: Fail

Conclusions

While I coded two successful feature detectors, there are infinitely more steps to be taken. The intern at the Met would still have much work to be done if only armed with these two detectors. However, some of the tools I used – particularly region growing, luminance analysis, and color separation, would continue to be useful for other detectors.

My next step would be to code a foreground-background object detector that isolated the main object in the image so that further tests could be performed on it. For instance, this detector would be useful for distinguishing between people and statues, or for analyzing the material out of which a statue is composed (a bronze statue will have a higher albedo and exhibit more specular reflection than a marble statue). Although I began to code this detector,[2] I ran into some issues with unanimity. Mine currently works for paintings of high color variance (eg. Andy Warhol), but not for realistic images, based on the color channel I chose to isolate. Picking a successful color channel might necessitate various foreground-background detectors for different types of images.

I also want to switch out of MATLAB and try using the Python Image Toolbox for creating classifiers, as I suspect that I would have more handle over the limitations I ran up against in MATLAB.

The power of an automatic image tagger depends on the rules that define it. While I expect that this project would be extremely useful for museums or even internet image search databases, its accuracy depends on its initial specifications, even if machine learning algorithms are used. This necessitates a bit of background in art history, or at least a discerning eye, to encompass all possibilities while minimizing error. Overall, though, I believe this is a worthwhile project with much potential use for the world, and I plan to continue developing it!

Note: All images in this paper are either the author's own or are copyright free.

---

[2] I followed the logic in http://www.mathworks.com/help/images/examples/detecting-a-cell-using-image-segmentation.html to code this detector.