Introduction [added for Github]

Today the field of computer vision has come to be associated with deep learning, but it was not always so. There are many classical algorithms and approaches to solve problems that do not require time and computation-intensive neural networks. The following is one such example. The task was to count the number of red blood cells (RBCs) in a series of images. It was successfully accomplished by utilizing a series of classical computer vision techniques. These include manual and automatic thresholding (Otsu’s algorithm), investigating different colour spaces (e.g. HSV), and their individual channels, to find the highest contrast, low-pass filtering to clean noise, and finally flood-fill with labeling (colouring) to count objects (connected-component counting). This approach takes a fraction of the time and computation that would have been needed with even a simple deep-learning based approach.

This task was done as part of a university assignment. The main output was a Jupyter notebook with all the code, and a summary report. The report follows in this document. The notebook is also there in the same folder as this report. This report is much shorter than the notebook, and is the recommended starting point if you wish to explore the problem.

Please note that direct use of OpenCV functions was not allowed, except to open and display the file. All the functions have been coded by hand (e.g. for thresholding, flood-fill, connected component counting etc). This made it an excellent learning exercise.

Assignment Report, Computer Vision

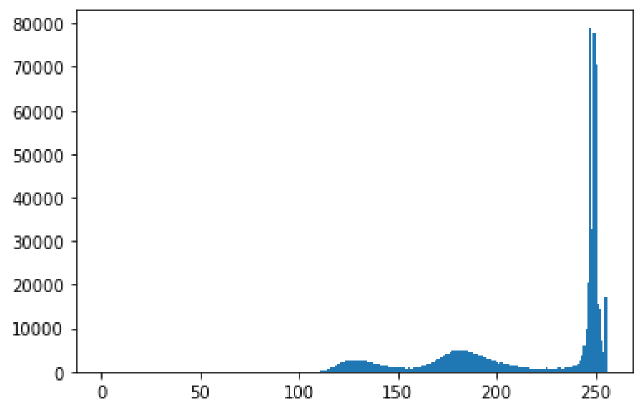
Note: The Jupyter notebook has detailed markup notes and commented code, and all images. This report is a summarized version of the notebook. Please refer to the notebook if more detail is required on any topic.

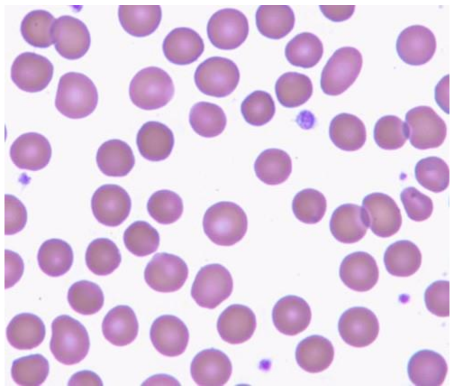
Task 1: Thresholding

Otsu Thresholding

Otsu's method works best on **bimodal distributions**, i.e. where the intensity values in the image are clustered around two major values (the modes). In such images, the histogram has two clear humps with flat areas in between. Otsu finds the best value at which to place the threshold boundary between the humps, such that if we look at the intensity of pixels in the two classes, the within-class variability is minimized, and between-class variability is maximized.

Below is the histogram for c1.jpg and the image itself:





We can see three main humps. The highest spike at right, centered around 245 intensity, is the near-white background. The leftmost hump (around 125) is the main body of the cells, and the middle hump (centered 175) is the lighter-shaded area in the middle of cells. Even though there are three modes (trimodal distribution), we are going to treat the lower two humps as one category and separate it from the brightest set of background pixels. We are treating our distribution as bimodal.

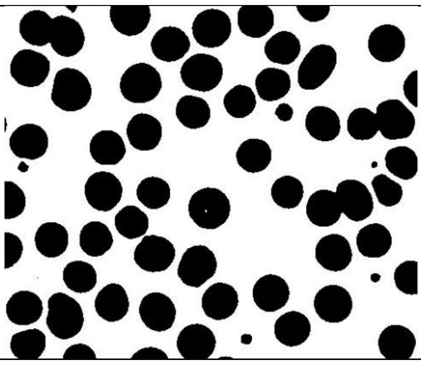
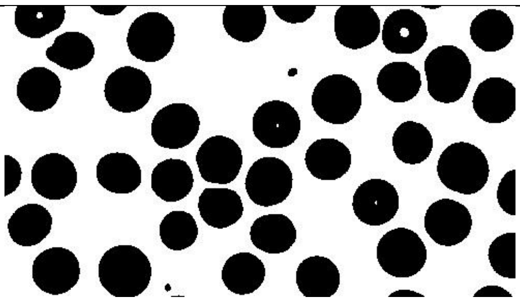
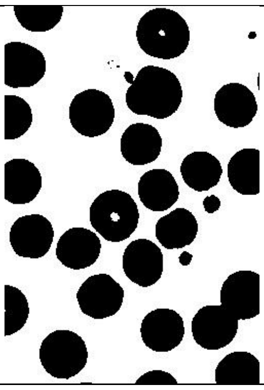
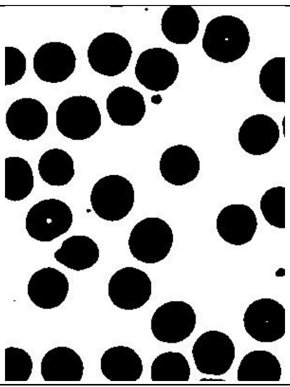
We have used the approach of maximizing the between-class variance, as this is less expensive to compute than minimizing the within-class variance.

Briefly, the **pseudocode** for Otsu can be given as:

```
For each threshold from 0 to 255:
    1. Separate the pixels into two classes according to the threshold.
    2. Find the weights (total probabilities) of the pixels in each class, by multiplying each intensity
       level by the number of pixels (their histogram count).
    3. Find the mean of each class.
    4. Square the difference between the means and multiply by the weights.
    Return the threshold value that maximizes this difference.
```

We are using Numpy vectorized operations for our Otsu implementation, as they are much faster than nested for loops. Well-commented code is given in the notebook and is not being repeated here due to space limitations.

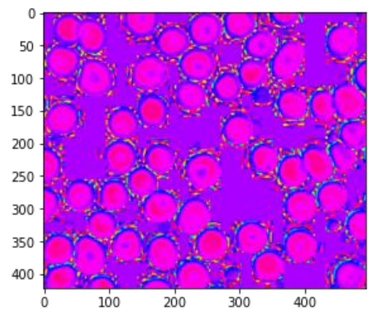
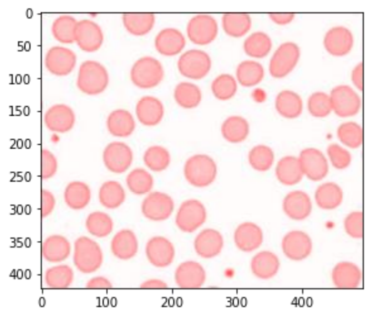
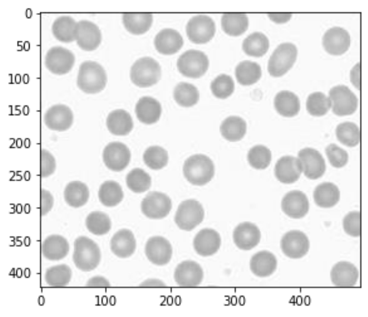
The Otsu function gives the value of **202** as the threshold for all four images. The threshold is the same as the images come from the same source and are very similar in characteristics. The four output images are as follows:

	
c1	**c2**
	
c3	**c4**

HSV Thresholding

HSV stand for Hue (shade of color), Saturation (how deep that color is), and Value (darkness or brightness). Some images that are harder to segment in RGB space can be easier to segment in H, S, or V, or a combination of them.

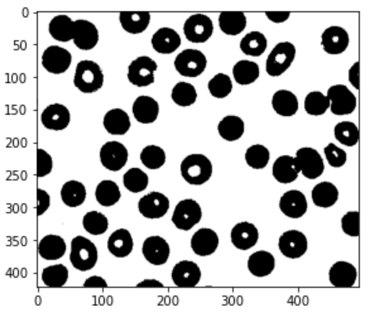
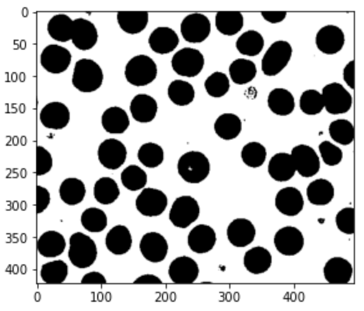
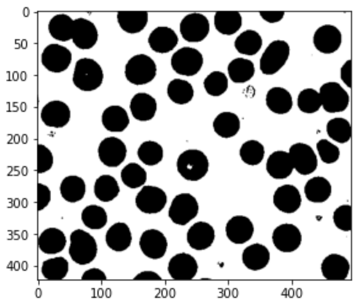
In file c1.jpg, getting pure Hue, Value or Saturation, while setting the others to 0, got these results:

		
Pure Hue	**Pure Saturation**	**Pure Value**

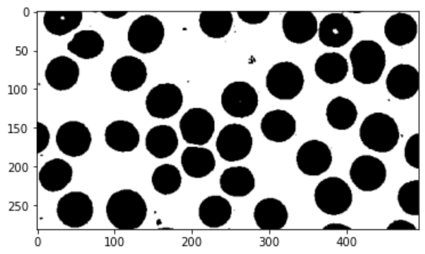
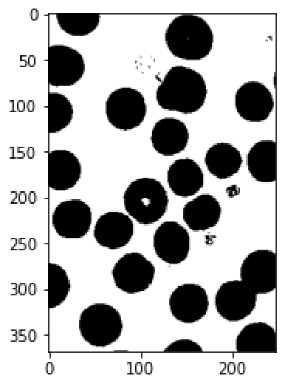
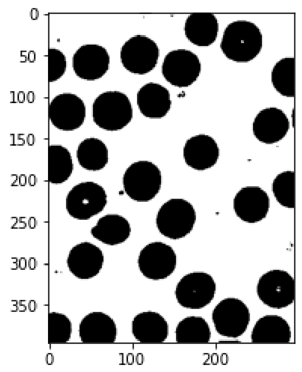
We can see that purely using the Hue is unsuitable, since the background and foreground seem to have a similar purplish shade (other hue images were also tried but were not found suitable, refer to the Notebook for an example). Pure Saturation looks more suitable, but when converted to grayscale, would be quite similar to the RGB images we thresholded in the previous section, and would not gain us additional benefit. However, the Value image looks promising. The small cells, which are probably not RBCs and are some other kind of cell or noise, become very faint in the Value dimension. Therefore, further analysis was done on pure value images.

Applying Otsu's method on the pure value image returned a threshold of 220. It was good at reducing/removing small cells, but was giving holes in many RBCs, because their central region is fainter than the outer body. So different threshold values were tried manually. Higher thresholds were filling in the holes but were also making the noise cells more prominent. An optimal value of 232 for the first image was established (by visual inspection). It gives very minor holes in one or two RBCs, but keeps the noise cells faint, so that they can be easily removed by filtering in the next task.

Below are some example images for c1.jpg, more are in the notebook.

		
Threshold 220	**Threshold 235**	**Threshold 232**

Similar experimentation was done on the other images to find the optimal thresholds. The finally chosen thresholds and outputs are below.

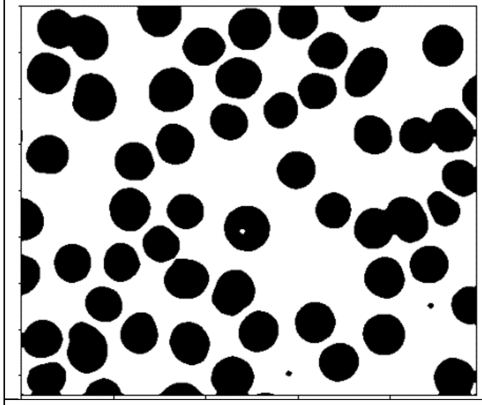
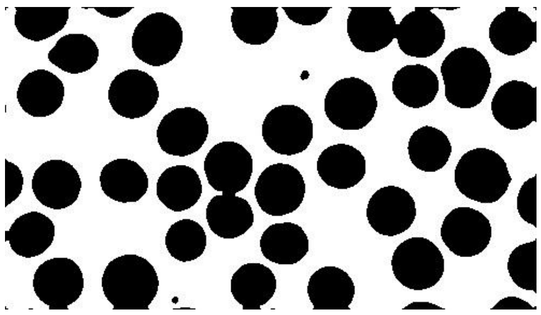
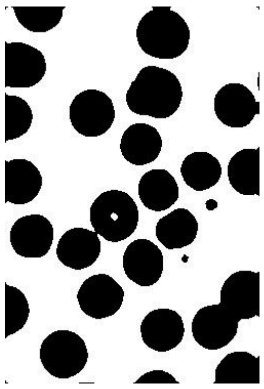
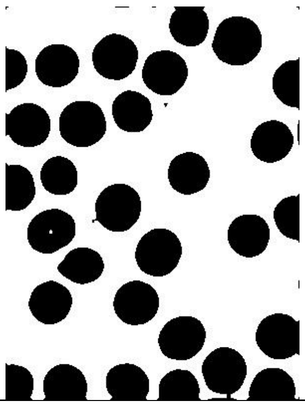
		
c2, threshold 241	**c3, threshold 235**	**c4, threshold 235**

Task 2: Cell counting

Median filter

A simple median filter was applied to the output from the HSV threshold in the previous step (HSV was chosen as it was giving better segmentation than RGB). Median filter with kernel size 3 was giving some cleaning of noise, but it improved with larger kernels. But at larger kernels the edge cells were bleeding into the edge, and cells which were close by were growing towards each other into an overlap. After experimentation, kernel size 7 was found to be the best balance.

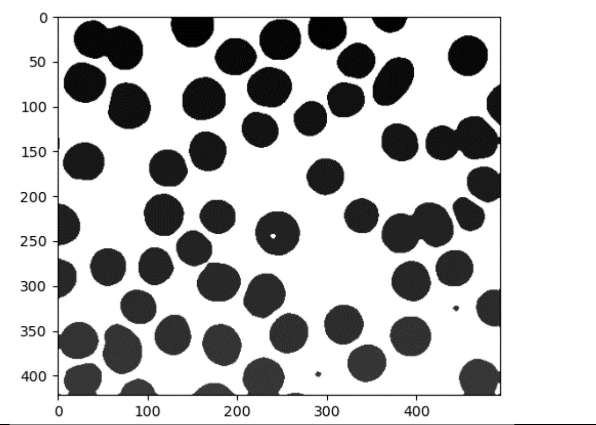
Here is the output for all images with 7-size kernels. We can see that we have significantly removed noise compared to the Value threshold output images shown above, but in c2 & c3, close cells have overlapped. But we will account for overlapping cells in the cell-counting algorithm.

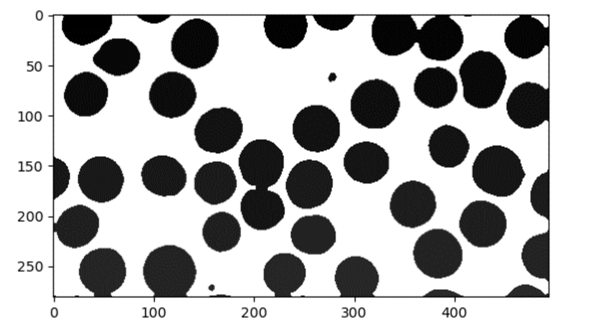
	
c1 median-filtered image, with kernel size 7	**c2**
	
c3	**c4**

Connected Component Counting

We first colored the cells. The algorithm traverses the image horizontally and vertically, looking for pixels with value 0 (black). When found, a flood-fill algorithm is called for that pixel. It colors the cell with a gray value, and looks at the 4-neighbourhood of the pixel, using depth-first search implemented recursively, and continues the same till the whole area is filled. Then it returns to the parent function, which increments the color counter and continues traversing the image to find the next black pixel.

Since each connected region (cell) is given the next higher colour, it causes the cells lower in the image to get slightly brighter shades. Here are two example colored images. The cells lower in the image are slightly brighter (gray) than the higher ones (black). All four images are not given below due to space constraints, they are in the notebook if needed for reference.





To get areas of cells, a simple np.unique() with return_counts set to True was used, to neatly return lists of colors and pixel counts of each color. Here is the output for c1, formatted as a dictionary.

```
{1: 5,           12: 1299,       23: 1369,       34: 920,        45: 958,        56: 1578,
 2: 16,          13: 1793,       24: 1452,       35: 3140,       46: 1211,       57: 1505,
 3: 1421,        14: 1643,       25: 32,         36: 1902,       47: 1516,       58: 1235,
 4: 1386,        15: 1764,       26: 1538,       37: 1222,       48: 29,         59: 29,
 5: 580,         16: 1837,       27: 1403,       38: 1283,       49: 1504,       60: 531,
 6: 6,           17: 1931,       28: 1332,       39: 1328,       50: 1429,       61: 424,
 7: 5,           18: 1303,       29: 1231,       40: 1355,       51: 1612,       62: 5,
 8: 1686,        19: 531,        30: 1637,       41: 725,        52: 1451,       63: 5,
 9: 2909,        20: 1130,       31: 975,        42: 1524,       53: 1891,       64: 28,
 10: 1566,       21: 1272,       32: 1176,       43: 1665,       54: 1576,       255: 129913}
 11: 1470,       22: 2932,       33: 1192,       44: 1740,       55: 1412,
```

255 is the background color, so it has a much higher number of pixels. There are some noise or edge cells with very small areas. The rest of the values are proper red blood cells. After removing the background and sorting the above data, and trying to see the patterns, **400 was set as the threshold for min_area, and 2500 for overlap_area** (the reasoning is further explained in the notebook.)

Based on this, the following cell counts were observed. The algorithm counts cells that are mostly present in this image, ignores noise and cells that are only little visible, and correctly counts overlapping cells as two cells.

Image	**c1**	**c2**	**c3**	**c4**
Cell count	57	38	24	30

References

Otsu's method, Wikipedia, https://en.wikipedia.org/wiki/Otsu%27s_method, accessed 14th March 2021

Connected Component Labeling, Wikipedia, https://en.wikipedia.org/wiki/Connected-component_labeling, accessed 14th March 2021

Flood Fill, Wikipedia, https://en.wikipedia.org/wiki/Flood_fill, accessed 14th March 2021

Handbook of Image and Video Processing, Alan C. Bovik, Academic Press, 2010