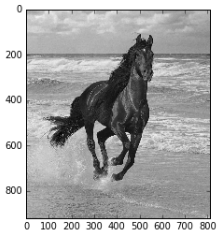
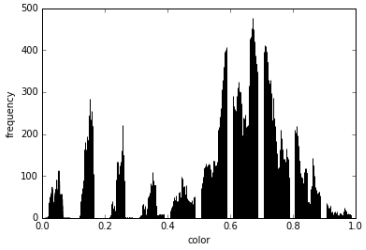
Clustering using Otsu's Method

Suppose that we have a gray-scale image, that we want to cluster into a light and dark set. For example, a dark foreground and a light background. Consider the following image as an example. (I found this on pinterest and have some trouble attributing it to a source.)



We covert to a 2D $n \times m$ image, where each pixel is a single real number (representing the gray-scale). Let's consider the pixels simply as an ordered list of colors, and look at the histogram. we can use the `img.reshape(..)` command to reshape a 2D array to be a 1D array (a list), together with the `np.sort(..)` command to sort it. We call the result the "bag" of pixels, since we have forgotten about their structure in the image.

```
img2=img[:,:,0].copy()
n,m=img2.shape
bag=sort(img2.reshape([1,n*m]))
plt.hist(img2)
```



We see that there is indeed such a separation in out image, and we notice two clusters in the color histogram. These clusters may not always be the same size, and there is no reason for the separation to be in the middle of the color spectrum. Therefore, picking a fixed color, or a mean or median of the distribution does not necessarily separate the clusters. When we say that a distribution is "clustered", we mean that there is low variation within clusters, and high variation between clusters. This concept is very important in all kinds of clustering problems, for example in machine learning problems. (When Spotify recommends music to you, it is because they have clustered you with other music fans.) We will talk more about these kinds of problems later in the course.

For this problem, suppose that we want to use a threshold `t` that separates the foreground from the background. Let μ_{FG} be the mean color of the foreground cluster, and μ_{BG} be the mean color of the background cluster. We can then write the within-cluster variance as

$$R(t) = \frac{1}{n \times m} \sum (c_i - \mu_{G(i)})^2 ,$$

where c_i is the color of pixel i and $G(i)$ is the group (or cluster) of pixel i, which is either foreground of background. So,

$$G(i) = \begin{cases} FG, & \text{if } c_i \leq t, \\ BG, & \text{if } c_i > t. \end{cases}$$

You can think of $R(t)$ as a measure of how concentrated/clustered the groups are around their repsective means. You do not need to understand much probability theory for this course, but the law of total variance ensures that, instead of looking for a minimal within-group variance, we can look for maximal between-group variance. The more we

separate $\mu_F G$ and $\mu_B G$ from μ (the overall mean of the set of pixels), the lower $R(t)$ will be. The between-group variance can be written as

$$R'(t) = \frac{|FG|}{n \times m} (\mu_{FG} - \mu)^2 + \frac{|BG|}{n \times m} (\mu_{BG} - \mu)^2 .$$

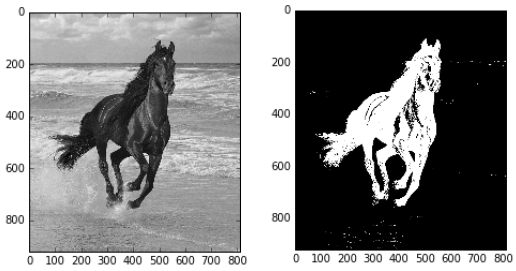
This second function, $R'(t)$, is a little easier to write in code. The numbers $|FG|$ and $|BG|$ are simply the sizes of the foreground cluster and background cluster, respectively. So, we want to search for a t_m, which is the threshold t for which the function $R'(t)$ is maximal. To do this, we can simply loop over possible values of $0 \leq t \leq 1$ (with steps of $1/1000$, for example), and find the one with the highest $R'(t)$. In many clustering problems, the number of possible ways to cluster is exponentially large, so it is not possible to simply loop over all possibilities. In this case, there are relatively few possibilities, because of the constraints on our clusters, so this is an easy problem to optimize.

Numpy gives us a few useful commands to compute $R'(t)$ given t. The function *np.argmax*(..) finds the first index for which a condition holds. For example:

```
In:   L=np.array([1,2,3,4])
In:   np.argmax(L>2)
Out:  2
```

So, if we set `ind = np.argmax(bag > t)`, then this gives us the first index of the background. Then, $|FG|$ =`ind`, $|BG| = n \cdot m -$`ind`. Furthermore, numpy has a built-in mean function. So, μ_{FG} =`np.mean(bag[:ind])` and μ_{BG} =`np.mean(bag[ind:])`.

Finally, once we have found the optimal threshold t_m, we can create a mask which is `True` when a pixel is on the foreground, and `False` when a pixel is in the background, using the command `mask=img2<=tm`. In order to visualize a mask with `True`/ `False` values, it helps to set the colormap to gray-scale. Then, `plt` maps `True` to white and `False` to black. (Otherwise, it defaults to red/blue which is hard to read.) The result will look like this:



This particular picture was chosen to work quite well with this method. You should try different images (and create your own simple ones!!) to see when and how this method works. Of course it also works well if an image has a light foreground and dark background. In color images, this method may sometimes work well on a specific color. For example, a picture of a red kite in a blue sky[1] will cluster well of we cluster by blue or by red. If this image is first converted to gray-scale, it may not cluster well at all.

[1] thephoto-news.com/apps/pbcs.dll/article?AID=/20120803/ENTERTAINMENT/120809992/Kites-over-the-Hudson-set-for-Aug-25-